CAPÍTOL 1. ESPECIFICACIÓ I IMPLEMENTACIÓ D'ESTRUCTURES DE DADES

1.1. Estructures de dades i disseny d'algorismes

La tasca de programar ordinadors es va convertint en una activitat gairebé quotidiana; l'ordinador personal forma part del conjunt de «mobles» de moltes llars i és ja un element indispensable a molts llocs de treball i ensenyament.

La programació és la qualitat per excel·lència dels ordinadors, car permet aprofitar a fons la potència lògica que ens ofereixen. Programar un ordinador és una tasca metòdica, caracteritzada per l'exquisidesa dels resultats o programes. Tot i que és una activitat relativament senzilla, suposa una adequada formació d'acord amb els diferents nivells que la componen.

L'algorísmica és la branca de la informàtica que s'ocupa de l'estudi de tècniques i metodologies de construcció d'algorismes, l'evolució de les quals es caracteritza per la cerca contínua de patrons que donin resposta a les necessitats, cada vegada més manifestes, d'abordar projectes de software de gran envergadura amb totals garanties de fiabilitat.

L'especificació i la verificació formal de programes són les grans esperances per aconseguir els objectius proposats de correctesa, malgrat que encara no es trobin a un nivell de desenvolupament que permeti fer-ne un ús pràctic eficient, circumstància que en dificulta l'aplicació per part de la gran majoria de programadors.

Les estructures de dades constitueixen l'eix fonamental dels programes, car a més de participar de totes les etapes del seu cicle de construcció, com són l'especificació, la implementació i la verificació, condicionen decisivament les característiques dels programes que en fan ús, fins al punt que formen el centre de les modernes metodologies de programació, com són els *tipus abstractes de dades* i la *programació orientada a objectes*, metodologia aquesta derivada de la primera.

Totes les revolucions metodològiques de la programació han tingut com a premissa la superació dels problemes de complexitat que suposava, en el seu moment, el disseny de programes; així varen néixer entre mitjan anys seixanta i principi dels setanta la metodologia de la programació estructurada i el llenguatge de programació PASCAL – successor de l'ALGOL W, acabat de dissenyar l'any 1965– com a suport d'aquesta i patró de tota una família de llenguatges posteriors (Alphard, CLU, Euclid, Mesa, MODULA, Gypsy, etc.). Però aviat els investigadors i programadors s'adonaren que els problemes de complexitat de molts grans projectes de software no desapareixien mitjançant l'ús aïllat de la metodologia de la programació estructurada. Calia una nova revolució metodològica.

L'any 1977 ja es comptava amb el bessó de la dita revolució sota el nom d'*abstracció de dades*. Mentre que la programació estructurada basava la construcció de programes en el disseny estructurat de la part de control, això és, en el conjunt d'accions que conformen un programa, l'abstracció de dades centra l'atenció en les estructures de dades que participen en el programa i manifesta els principis següents:

- Les estructures de dades constitueixen les interfícies lògiques entre les diferents parts o els mòduls d'un programa.
- Les interfícies de dades han de ser independents de la seva implementació, que es manté oculta i flexible. Així també, el seu funcionament ha de ser independent del dels mòduls que en fan ús.
- Les interfícies de dades han d'estar protegides de qualsevol element que no hi tingui accés.

El primer principi mostra una nova tendència en la construcció de programes, orientada al disseny modular basat en interfícies de dades. Un programa es concep ara com un conjunt de mòduls lligats per les dades que aquests comparteixen, de les quals cada mòdul fa ús. Aquestes dades s'anomenen *dades externes als mòduls del programa* o també *dades internes al programa*. La figura 1.1 il·lustra la idea del disseny modular i mostra un programa com un conjunt de mòduls connectats per les dades compartides. El mateix principi s'aplica al disseny dels mòduls del programa; així, un mòdul es pot veure format per un conjunt d'altres mòduls connectats per una interfície de dades, les *dades internes al mòdul.* En tot cas, si un mòdul és prou senzill, el disseny es pot abordar directament a partir de la metodologia de la programació estructurada.

La descomposició modular d'un programa afavoreix la independència entre els mòduls, en el sentit que l'agrupació de les dades en una interfície facilita el deslligament funcional dels mòduls que hi tenen accés; així, els mòduls seran dissenyats de forma que cadascun tingui funcions diferents sobre les dades de la interfície. Malgrat això, d'altra banda ens trobam que, encara que els mòduls siguin funcionalment independents, comparteixen la mateixa interfície de dades, fet que pot provocar alguns problemes, com ara que la fallada d'un dels mòduls repercuteixi en la interfície de dades i hi provoqui errors o malfuncionaments interns, o més encara, que accions com aquestes repercuteixin en altres mòduls en contacte amb la interfície. La solució d'aquests problemes es troba en els principis segon i tercer, en què es manifesten diverses característiques de les interfícies de dades que permetran un disseny pràctic i fiable del programa.

Aconseguir un funcionament totalment independent entre els diferents mòduls i la interfície de dades garantirà la no propagació de problemes i malfuncionaments –tal com s'ha mencionat a l'anterior paràgraf– dels mòduls a la interfície i viceversa. Si a més es té en compte que les interfícies poden ser tan complexes com requereixi el problema, aleshores calen tècniques de disseny pràctiques i alhora que dotin de seguretat les estructures de

dades. La metodologia dels tipus abstractes de dades és darrere aquests objectius, per a la qual finalitat es basa, d'una banda, en el concepte d'independència entre l'especificació i la implementació de les estructures, que possibilita un disseny pràctic i força segur; i, d'altra banda, en la idea d'encapsulació de les dades de l'estructura, que no permet que cap procediment alliè a l'estructura de dades hi tingui accés i dota de la protecció i la seguretat necessàries les dades, tal com s'exposa en el tercer principi.

La metodologia dels tipus abstractes de dades es troba a un nivell de desenvolupament molt acceptable, I té l'avantatge que incorpora uns sòlids principis formals sobre els quals basar-se.

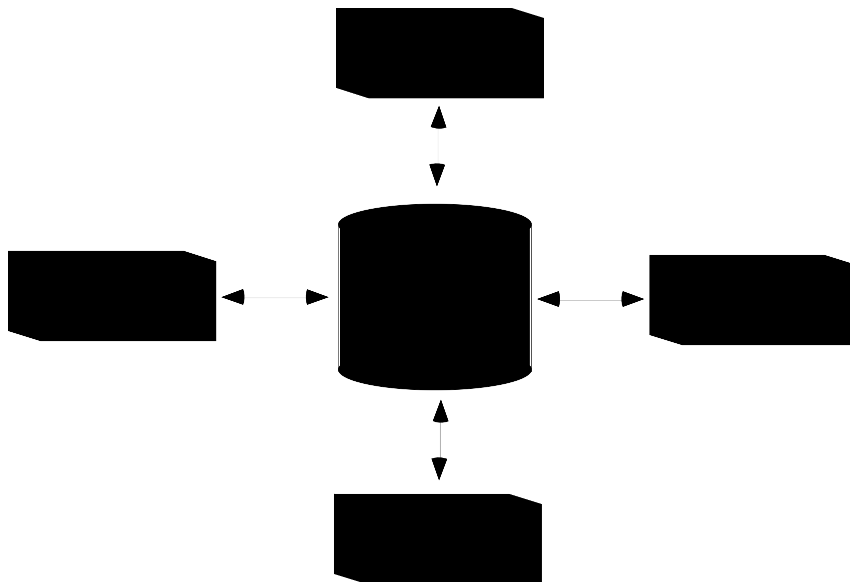


Figura 1.1. Descomposició modular d'un programa

1.2. Tipus de dades, tipus abstractes de dades i estructures de dades

El significat dels conceptes de «tipus de dades» i «tipus abstractes» de dades sol variar d'acord amb el marc d'estudi. La discussió se centrarà en l'àrea de la programació d'ordinadors, àmbit original de la teoria dels tipus abstractes de dades i, per tant, on es manté l'autenticitat dels conceptes.

En programació, el «tipus de dades» d'una variable és el conjunt de valors que la variable pot assumir. Per exemple, una variable de tipus booleà pot adoptar només dos valors possibles: «vertader» o «fals». A més, hi ha un conjunt limitat però ben definit

d'operacions que tenen sentit sobre els valors d'un tipus de dades; així, operacions típiques sobre el tipus booleà són «AND», «OR»…

Els llenguatges de programació assumeixen un nombre determinat de tipus de dades, que pot variar d'un llenguatge a un altre; així, en PASCAL tenim els «enters», els «reals», els «booleans», els «caràcters»… Aquests tipus de dades són anomenats en el context dels llenguatges de programació «tipus de dades bàsics».

Tal com s'acaba de descriure, els tipus de dades són definits pel rang de valors del tipus i per les operacions permeses sobre aquests valors. Fins fa pocs anys, tota la programació es basava en aquest concepte de tipus i no eren pocs els problemes que apareixien, lligats molt especialment a la complexitat de les dades que s'havien de definir. Per facilitar la comprensió, s'introduiran a manera d'exemple alguns dels problemes que portava l'ús dels tipus de dades, i posteriorment es discutiran diverses solucions.

• Imaginem la diferència entre nombres i dates. Les dates són exemples discrets en el temps que poden representar-se a l'ordinador mitjançant nombres, però moltes de les operacions amb nombres no són vàlides per a les dates. Alternativament, les dates es poden representar també mitjançant cadenes de caràcters, a les quals també es poden aplicar les mateixes restriccions.

• Considerem una estructura de dades senzilla, com pot ser una llista lineal, la qual es troba implementada, en un programa molt gran, mitjançant arrays, i a la qual s'accedeix des de molts de punts del programa. En un moment donat, per les causes que sigui, la implementació resulta ineficient i els programadors decideixen de canviar-la. És clar que si l'estructura ha estat definida d'acord amb la nomenclatura dels tipus de dades, el canvi d'implementació pot resultar inviable o, en el millor dels casos, molt costós.

• Una empresa de construcció de software acaba de rebre un mateix encàrrec per part de nombroses empreses de serveis, a causa d'una nova normativa legal sobre indústria de serveis que acaba de ser publicada en el BOE. L'encàrrec és molt senzill i pretén fer justícia pel que fa a l'atenció al client. Es tracta simplement de fer una gestió FIFO (*First Input, First Output*) controlada per ordinador de les «cues» de clients de totes les empreses. En aquest cas el disseny de les estructures FIFO, també dites cues, es pot simplificar moltíssim si es fa un ús correcte de la parametrització, en el sentit que totes adopten la mateixa especificació, l'únic que canvia són possiblement les característiques dels elements de les estructures, que es poden concebre com a paràmetres.

Respecte al primer exemple, cal dir que a vegades no és suficient un tipus de dades (com ara els reals o els caràcters) per definir les estructures necessàries (les dates). Les estructures que es volen definir abracen tots els tipus de dades útils per implementar-les, sempre sota les restriccions de les operacions necessàries adequades per manipular-les (això són les operacions sobre les llistes), és a dir, qualsevol tipus de dades és útil per representar les estructures sempre que disposi de les operacions necessàries i adequades.

Per tant, el tipus «data» es pot veure com una entitat abstracta que engloba tots els tipus útils per representar les dates. D'altra banda, es pot observar que la utilització d'operacions no adequades sobre un tipus provoca una desprotecció d'aquest enfront de resultats indefinits o erronis.

Del segon exemple es dedueix que l'especificació del tipus «llista lineal», això és, la seva manera de comportar-se o funcionar, ha de ser independent dels tipus de dades emprats per implementar-la. Si és així aconseguim tres avantatges clars i molt importants: d'una banda, tal com s'expressa a l'exemple, si en un moment donat se n'ha de canviar la implementació, aleshores serà suficient fer els canvis oportuns en les definicions de tipus corresponents, sense cap necessitat de modificar-ne l'especificació; d'altra banda, si durant el disseny d'una estructura de dades se separen les etapes d'especificació i implementació, aleshores la complexitat de l'esmentat disseny es redueix enormement, es facilita la tasca i es redueix notablement la possibilitat d'errades en el programa; finalment, en cas que el programa contingui alguna errada a causa de les seves estructures de dades, aquesta serà molt més fàcil de localitzar i, conseqüentment, de corregir.

El tercer exemple posa en relleu la importància de l'ús de la parametrització en el disseny d'estructures de dades, que permet diferenciar els elements comuns de l'especificació d'aquells que poden variar, dits paràmetres. Els paràmetres, però, només han de ser particularitzats a l'etapa d'implementació de les estructures, de manera que les seves característiques només variaran d'una implementació a l'altra.

D'acord amb els exemples anteriors, el concepte de tipus abstracte de dades és, doncs, més extens que el de tipus de dades:

un tipus abstracte de dades (TAD) es pot veure com una col·lecció de tipus de dades que satisfan un conjunt ben determinat d'operacions, les quals doten alhora el tipus abstracte de la necessària protecció no permetent-hi cap accés si no es fa a través seu. A més, el disseny d'un TAD ha de permetre diferenciar les etapes d'especificació i implementació del tipus; d'aquesta manera en facilita el disseny i la correcció.

Els algorismes es dissenyen en termes de TAD, però per implementar-los en un determinat llenguatge de programació s'ha de trobar alguna forma de representar els corresponents TAD en termes dels tipus de dades i operadors suportats pel llenguatge. Per representar o implementar els TAD s'empren les «estructures de dades», que són col·leccions de variables connectades de diverses formes.

La «cel·la» és el bloc bàsic de construcció d'estructures de dades. Una cel·la es pot veure com una caixa capaç de contenir un valor d'algun tipus de dades del llenguatge de programació en qüestió. Un «tipus del llenguatge» pot ser simple o compost; un tipus

simple, dit també «bàsic», és el tipus d'una variable, mentre que un tipus compost és un tipus format per agregació de tipus simples mitjançant els corresponents mecanismes del llenguatge. Una estructura pot ser doncs un tipus bàsic, un tipus compost o agregacions de diferents tipus, bàsics i/o composts. Per exemple, un registre del PASCAL amb dos camps, un camp d'informació (INFO) i un camp punter (PUNTER), és, d'acord amb les definicions exposades, un tipus compost del PASCAL, però també és una estructura de dades, tal com s'il·lustra a la figura 1.2. En canvi, una col·lecció de registres encadenats per punters, tal com s'il·lustra a la figura 1.3, és una estructura de dades però no un tipus compost, ja que no es pot declarar com a tipus mitjançant els mecanismes d'agregació del PASCAL. Per tant, la construcció d'una estructura de dades pot haver de menester, a més de la declaració dels tipus del llenguatge escaients, els procediments corresponents.

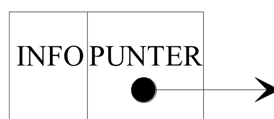
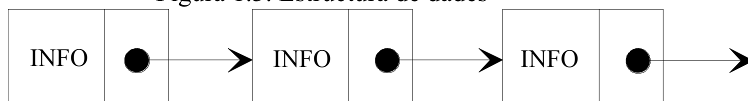


Figura 1.2. Tipus de dades compost del PASCAL

Figura 1.3. Estructura de dades



La resta del capítol se centra en la descripció dels principals mecanismes d'agregació de dades comuns a la majoria dels llenguatges de programació didàctics, com ara el PASCAL i l'ADA.

1.3. Mecanismes d'agregació de dades

Per facilitar la comprensió dels algorismes i el desenvolupament dels exercicis, s'utilitza al llarg del llibre un llenguatge pseudocodi anomenat «pseudoPASCAL», versió simplificada del PASCAL. No obstant això, el codi del pseudoPASCAL que es fa servir és prou genèric perquè pugui ser usat o traduït directament a altres llenguatges familiars al PASCAL, com ara els llenguatges C i ADA. Pseudollenguatges d'aquest tipus són emprats amb molt d'èxit en nombroses obres escrites, i també a diferents universitats, sempre amb finalitats didàctiques; així, a la Universitat Politècnica de Catalunya es fa ús del llenguatge MERLÍ, adoptat també a la Universitat de les Illes Balears i a d'altres; diversos llibres d'autors reconeguts fan servir versions del que anomenen un «SuperPASCAL», etc.

1.3.1. PseudoPASCAL

En aquesta secció es defineixen les característiques del pseudollenguatge de programació utilitzat per implementar els algorismes dels capítols posteriors.

L'estructura general d'un algorisme és la següent:

```
BEGIN
    a1;
    a2;
    ...
    an
END
```

on a_1, a_2,…, a_n són les accions de l'algorisme. Les accions poden ser simples o compostes. D'una acció composta en formen part diverses accions simples, disposades seqüencialment.

El llenguatge consta de tres tipus d'accions:

• Accions imperatives.

• Accions alternatives.

• Accions repetitives.

Les accions imperatives descriuen una ordre o un manament que s'ha d'executar irremeiablement i són identificades pel nom de l'ordre que s'ha d'executar. Així, per exemple, *imprimir(fitxer)* és una acció parametritzada que dóna l'ordre d'imprimir el contingut del paràmetre *fitxer*.

Les accions alternatives indiquen el camí que s'ha de seguir i, conseqüentment, les accions que s'han d'executar en un conjunt de dos camins possibles. El format és el següent:

```
IF «condició» THEN
    a
[ELSE
    b]
```

on «condició» és una expressió lògica que pot ser vertadera o falsa, de forma que si és vertadera, s'executa l'acció *a* i, en cas contrari, s'executa l'acció alternativa *b*. Les accions *a* i *b* poden ser simples o compostes; en cas que siguin compostes el conjunt de sentències

simples que componen l'acció *b* van tancades entre un BEGIN al principi i un END després de l'última sentència, la qual cosa no és necessària en el cas de l'acció *a*. Les claus [...] indiquen que l'escriptura de l'alternativa de la condició és opcional.

Les accions iteratives o repetitives permeten executar diverses vegades un mateix conjunt d'accions, d'acord amb la satisfacció d'una determinada condició. Es distingeixen tres tipus d'accions iteratives:

• L'acció iterativa WHILE, amb format:

WHILE** «condició» **DO

a

on «condició» és una condició del mateix tipus que a les alternatives i *a* és una acció simple o composta; en cas que sigui composta el conjunt de sentències simples que la componen va tancat entre un BEGIN al principi i un END després de l'última sentència.

La sentència WHILE provoca que l'acció *a* s'executi repetidament mentre «condició» sigui vertadera. Abans de la primera execució de *a* i després de cadascuna de les execucions s'avalua «condició», de manera que, si aquesta és vertadera, la sentència *a* s'executa de bell nou.

• L'acció iterativa FOR, amb dos formats possibles:

1) ***FOR** i :=x **to** y **DO***

a

2) ***FOR** i :=x **downto** y **DO***

a

L'acció *a* pot ser simple o composta; en cas que sigui composta el conjunt de sentències simples que la componen va tancat entre un BEGIN al principi i un END després de l'última sentència, *x* i *y* són els identificadors de dues variables o constants que indiquen respectivament els límits inferior (versió 1 del FOR)/superior (versió 2 del FOR) i superior (versió 1 del FOR)/inferior (versió 2 del FOR) dels termes d'una progressió aritmètica de raó unitat, i *i* s'anomena variable de control de l'acció FOR i s'utilitza per emmagatzemar els termes de la progressió.

La sentència FOR indica que l'acció *a* s'executarà repetidament, una vegada per a cadascun dels valors de la progressió, els quals s'aniran assignant a la variable de control *i,* de forma creixent, des del límit inferior al límit superior.

• L'acció iterativa REPEAT, amb format:

REPEAT

a

***UNTIL** «condició»*

L'acció REPEAT indica que l'acció *a*, que pot ser simple o composta, s'executarà repetidament mentre la condició «condició» sigui vertadera. A més el REPEAT s'executa almenys una vegada, a diferència del WHILE.

Els procediments i les funcions són els elements bàsics per estructurar els algorismes. La declaració dels procediments adopta el format següent:

procediment *nom(paràmetres del procediment);*
part de declaracions;
BEGIN
    Cos_del_procediment
END

Els paràmetres del procediment se separen per un punt i coma «;» i es declaren a través de l'identificador corresponent seguit dels dos punts i els tipus de paràmetre. L'exemple:

procediment *prova(a: enter; var b: real)*

indica que *a* és un paràmetre de tipus enter i *b*, de tipus real; a més, *a* és un paràmetre-valor i *b* un paràmetre-variable, igual que en PASCAL. Observau com els paràmetres-variable són precedits de la paraula reservada *var*.

La crida a un procediment es fa mitjançant l'escriptura del seu nom seguit de la corresponent llista de paràmetres tancats entre claus i separats per comes. No s'indica aquí el tipus del paràmetre ni les propietats (paràmetre-valor o paràmetre-variable). L'exemple:

prova(x, y)

és una crida al procediment prova anterior.

La *part de declaracions* és formada pel conjunt d'elements del procediment (tipus, variables, constants, procediments…), mentre que el *Cos_del_procediment* està format pel conjunt de sentències que formen el procediment.

Una funció és similar a un procediment; de fet, es pot escriure sempre una funció que faci el mateix que un procediment. En tot cas, és important distingir quan cal emprar una funció i quan un procediment; el consell és molt senzill: una funció s'ha d'emprar quan es vol calcular un resultat únic; d'altra banda, quan es vol calcular més d'un resultat és millor emprar un procediment.

La declaració d'una funció és molt similar a la d'un procediment, amb les diferències que ara no hi pot haver paràmetres-variable i s'ha de declarar el tipus de la *funció(tipus_de_la_funció)*:

funció *nom(paràmetres de la funció): tipus_de_la_funció;*
part de declaracions;
BEGIN
Cos_de_la_funció
END

la part de declaracions i el *Cos_de_la_funció* tenen el mateix significat que els d'un procediment.

Per cridar una funció se n'ha d'escriure el nom dins una expressió; així, *a :=nom* assigna el valor de la funció *nom* a la variable *a*; és clar que la variable i la funció han de ser del mateix tipus.

1.3.2. Els arrays

Els arrays o matrius constitueixen el mecanisme d'agregació potser més senzill de la majoria de llenguatges de programació. Un array és una estructura homogènia, constituïda per cel·les del mateix tipus, anomenat «tipus_cel·la». També s'anomena array d'una estructura d'accés aleatori, ja que es pot accedir arbitràriament a totes les cel·les de l'array i totes són igualment accessibles. Per accedir a una cel·la de l'array s'ha de disposar d'un «nom» o identificador de l'array en qüestió i d'un «índex» que ens permeti seleccionar la cel·la concreta de l'array identificat, el tipus del qual s'anonomena «tipus_índex» de l'array. Així, la declaració pseudoPASCAL del tipus array «MATRIU» és tal com s'indica a continuació:

Tipus *MATRIU =* ***array****[tipus_índex]* ***of*** *tipus_cel·la*

L'esmentada declaració indica que el tipus MATRIU representa arrays amb cel·les del tipus *tipus_cel·la* i índexs del tipus *tipus_índex*. Cal notar que el *tipus_índex* determina les mesures de l'array (la quantitat de cel·les). Una variable del tipus MATRIU és una successió de cel·les consecutives, una per a cada valor de *tipus_índex*, el contingut de les quals és del tipus *tipus_cel·la*.

Exemples:

Tipus *document:* ***array****[1..6]* ***of*** *enter*

Tipus *identificació:* ***array****[1..90]* ***of*** *caràcter*

Una variable definida com a:

Var *A: document*

conté sis cel·les. L'accés a una cel·la particular es fa a través de la variable A, a continuació de la qual s'escriu entre claus i fent ús dels valors de *tipus_índex*, la cel·la que es vol referenciar. Per exemple, A[1] fa referència a la primera cel·la de l'array, A[2] a la segona…

Si el contingut dels components de A és tal com s'il·lustra a la taula 1.1, aleshores es verifiquen les propietats següents:

1	10
2	12
3	14
4	16
5	18
6	20

Taula 1.1. Array de tipus document

A[1]=10; A[2]=12; A[3]=14; A[4]=16; A[5]=18; A[6]=20.

L'assignació d'un valor a una cel·la d'un array es fa de manera similar a l'assignació de valors a una variable, amb la diferència que ara s'ha d'indicar a través del tipus *tipus_índex* de l'array la cel·la específica a la qual es vol assignar el valor, tal com a continuació s'indica:

var_array[valor_tipus_índex] :=valor_tipus_cel·la

L'anterior expressió indica que s'assigna el valor *valor_tipus_cel·la* de tipus *tipus_cel·la* a la cel·la d'índex *valor_tipus_índex* de la variable array *var_array*. Notau que el valor *valor_tipus_índex* és del tipus *tipus_índex*. Així, l'expressió A[3] :=7 assigna el valor 7 a la tercera cel·la de l'array A, que queda tal com s'il·lustra a la taula 1.2.

1	10
2	12
3	7
4	16
5	18
6	20

Taula 1.2. Array de tipus document

Els índexs dels arrays han de ser de tipus escalar, la qual cosa permetrà que aquests puguin ser calculats; és a dir, s'hi podrà fer referència a través d'expressions i no tan sols mitjançant un valor constant.

1.3.3. Els registres

La forma més general d'agregar dades per obtenir tipus estructurats consisteix a agrupar informació possiblement de diferents tipus. Els registres constitueixen un mecanisme comú en alguns llenguatges de programació que permet agrupar cel·les amb diferents característiques de tipus en una única entitat, el registre mateix. Les cel·les d'un registre s'anomenen camps, els quals poden adoptar alhora una estructura registre, d'aquí ve el concepte de tipus estructurat i jerarquitzat.

El tipus registre «REGISTRE» es defineix de la forma següent:

Tipus *REGISTRE =* ***record***

c_1: T_1;

c_2: T_2;

...

c_n: T_n

end

on c_1, c_2,…, c_n són els camps del registre, i T_1, T_2,…, T_n, els tipus dels camps esmentats respectivament.

Exemples:

```
Tipus data = record
        dia: 1..31;
        mes: 1..12;
        any: 1999..1999
end

Tipus estudiant = record
        nom: array[1..30] of caracter;
        dni: array[1..8] of enter;
        naixement: data;
        sexe: (home, dona)
                end
```

Mentre que el tipus registre del primer exemple conté tres camps o cel·les, totes de tipus subrang dels enters, el tipus del segon exemple en conté quatre, tots de tipus diferent. Observau com el camp naixement del segon exemple és del tipus «data», corresponent al primer exemple.

La variable B declarada com a:

Var *B: estudiant*

contindrà doncs quatre cel·les. La figura 1.4 il·lustra un cas particular de la variable esmentada.

nom	dni	naixement	sexe
Pere	42.992.230	05-02-1961	home

Figura 1.4. Variable registre

Per fer referència a una cel·la d'un registre, s'escriu l'identificador de la variable registre seguit d'un punt i del nom del camp o cel·la; per exemple, *B.naixement* es refereix al camp *naixement* de la variable registre B, mentre que *B.nom* es refereix al camp *nom*. L'assignació d'un *valor* a una cel·la del tipus *tipus_cel·la* d'una variable registre *var_registre* es fa d'acord amb el format següent:

var_registre.cel·la := valor

Així, la seqüència d'assignacions

B.nom[1] :='T';

B.nom[2] :='o';

B.nom[3] :='n';

B.nom[4] :='i';

provoca el canvi del camp nom tal com s'il·lustra a la figura 1.5.

nom	dni	naixement	sexe
Toni	42.992.230	05-02-1961	home

Figura 1.5. Variable registre

1.3.4. Els punters i els cursors

A més de les formes d'agrupar cel·les que s'han exposat, es poden representar relacions entre cel·les mitjançant l'ús de «punters» –o «apuntadors»– i «cursors».

Els punters

Les estructures dinàmiques de memòria es caracteritzen perquè és possible variar-ne la mida, circumstància que permet crear i eliminar elements de l'estructura a mesura que les necessitats del programa així ho requereixen. Exemples típics d'aplicacions amb una demanda dinàmica d'espai de memòria són els programes recursius i els compiladors. La tècnica que s'utilitza més sovint en aquests casos és realitzar una assignació de memòria en el temps o assignació dinàmica, és a dir, assignar memòria per als components del programa alhora que aquests són creats durant l'execució del programa mateix, i no en temps de compilació, ja que el compilador assigna una quantitat fixa de memòria per al component declarat. En una tal assignació de memòria, els elements de l'estructura creats dinàmicament no necessàriament ocupen posicions de memòria consecutives.

Considerem l'exemple d'una llista d'elements subjecta a nombrosos canvis, com per exemple la capacitat de créixer i reduir-se indefinidament en el temps. En aquest cas, és clar que l'ús d'un array per emmagatzemar-hi la llista no sembla adequat, ja que l'espai de l'array pot resultar insuficient en un moment donat, o pot ser que estigui infrautilitzat durant un període de temps llarg. Sembla que la tècnica adequada per gestionar la llista és una assignació dinàmica de memòria, que permeti crear i eliminar elements quan calgui. Així, el nombre d'elements de la llista s'ajustarà exactament a les necessitats de cada instant, cosa que no ocorria, com ja s'ha comentat, en el cas d'emprar un array per implementar la llista.

Els punters constitueixen un tipus de dades comú a diversos llenguatges dins l'àmbit dels objectius d'aquest llibre, com són el PASCAL, l'ADA, el C, etc. El contingut d'una variable de tipus «punter» és una adreça de memòria que indica la posició d'un determinat element anomenat «element apuntat». Per tant, es diu que una variable punter «apunta» a un element de memòria (o possiblement a cap, per a la qual cosa adopta un valor especial que s'anomena *nil*).

La declaració d'APUNTADOR com un tipus «punter» es fa de la forma següent:

Tipus *APUNTADOR =* ↑*tipus_apuntat*

Aquesta declaració indica que el tipus APUNTADOR apunta a elements del tipus *tipus_apuntat*.

Exemple:

Tipus *estudiant =* ***record***
nom: array[1..30] of caracter;
dni: array[1..8] of enter;
naixement: data;

sexe: (home, dona)

end

***Tipus** apunta =* $\uparrow$*estudiant*

El tipus *apunta* és un tipus punter que apunta a dades del tipus *estudiant*. Una variable A de tipus punter es declara com a:

***Var** A: tipus_punter*

on «tipus_punter» és un tipus *punter*.

Si es vol representar una llista encadenada mitjançant punters, això és, una llista en la qual cada un dels elements apunta a l'element de la posició següent dins la llista, aleshores convé que tots els elements de la llista tinguin, a més de la capacitat de fer referència a l'element següent, la possibilitat de contenir informació pròpia. La figura 1.6 il·lustra aquesta idea.

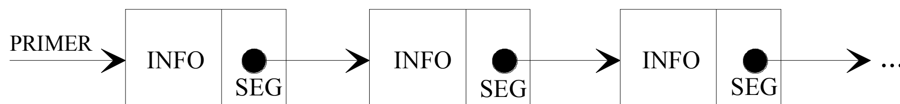


Figura 1.6. Llista encadenada per punters

A la figura 1.6 s'observa que els elements o cel·les de la llista contenen dos components o cel·les més bàsics, el component INFO, que conté informació referent a les característiques pròpies de la cel·la, i el component SEG, que és del tipus punter i es fa servir per poder accedir a l'element següent de la llista. De fet, sense el component SEG no seria possible accedir als elements de la llista. També cal notar la presència d'un element PRIMER, que no pertany a la llista però que es fa servir per accedir al primer element d'aquesta. Es pot inserir fàcilment un nou element a la llista, fent els corresponents reajustaments de punters. Per exemple, si es vol inserir un element NOU entre el primer i el segon element de la llista, els reajustaments de punters s'haurien de fer tal com s'indica a la figura 1.7, en què el camp o cel·la SEG del primer element de la llista apunta ara a l'element NOU inserit, mentre que el camp SEG de l'element NOU apunta a l'element de la llista que abans de la inserció ocupava la segona posició.

Els elements de la llista de la figura 1.6 es poden representar mitjançant registres, declarats tal com segueix:

Tipus** element_llista =* ***record

INFO: tipus_informació;

SEG: $\uparrow$*element_llista*

end

Es pot observar a partir de la declaració anterior que l'element SEG és un punter a elements de la llista.

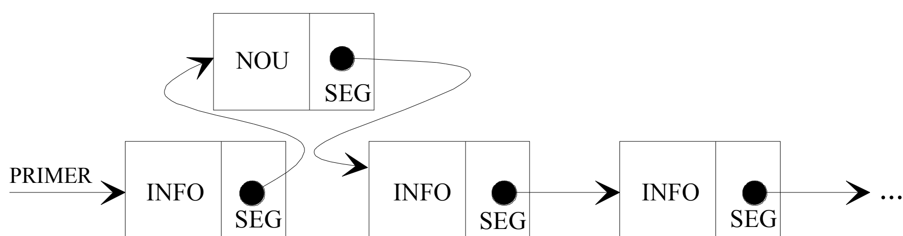


Figura 1.7. Inserció d'un element en una llista encadenada

Si A i B són dues variables punter, la sentència *A :=B* indica que la variable A adopta el mateix valor que la variable B (tal com en una sentència d'assignació típica), i per tant, l'esmentada variable apuntarà al mateix lloc de la memòria que B. L'expressió B↑ fa referència al contingut de la variable apuntada per B (s'entén per contingut totes les cel·les simples que formen part de la cel·la composta apuntada per B). La sentència *A↑ :=B↑* el que fa és copiar el contingut de la variable apuntada per B dins la variable apuntada per A. Les figures 1.8, 1.9 i 1.10 il·lustren el significat de les sentències esmentades abans i després ser executades respectivament.

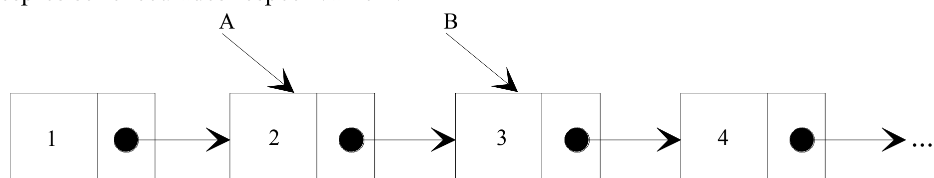


Figura 1.8. Llista lineal L

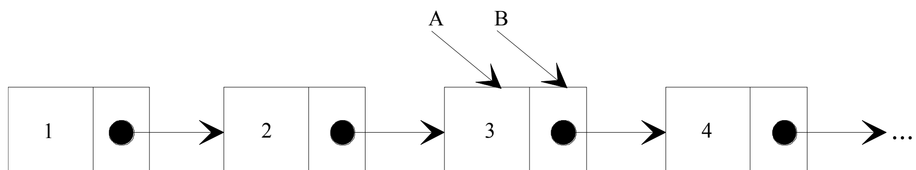


Figura 1.9. Llista lineal L després de l'operació *A :=B*

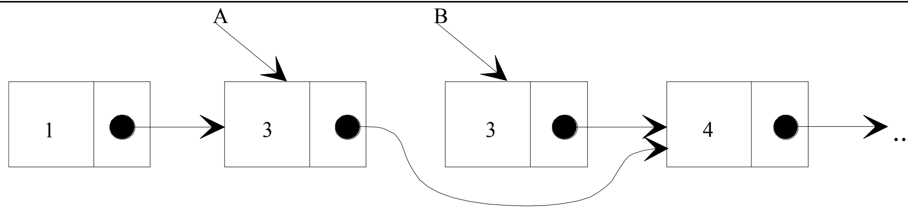


Figura 1.10. Llista lineal L després de l'operació $A\uparrow:=B\uparrow$

Si *nou_element* és una variable punter a cel·les del tipus *tipus_cel·la*, aleshores la sentència:

new*(nou_element)*

crea una cel·la del tipus *tipus_cel·la* apuntada per la variable *nou_element*, tal com s'il·lustra a la figura 1.11.

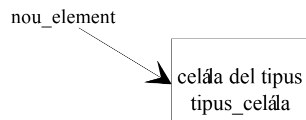


Figura 1.11. Creació d'una cel·la referenciada per un punter

Pel que fa a les cel·les apuntades per variables punter, l'estudi es limitarà al tipus registre, tal com s'ha definit a la secció 1.3.3. Per tant, partint de les declaracions següents:

Tipus *REGISTRE =* ***record***

c_1: T_1;

c_2: T_2;

...

c_n: T_n

end;

Var *punter:* $\uparrow$*REGISTRE;*

les expressions:

$punter\uparrow.\ c_1$

$punter\uparrow.\ c_2$

...

$punter\uparrow.\ c_n$

fan referència als camps c_1, c_2,..., c_n respectivament de la cel·la de tipus *REGISTRE* apuntada.

Exemple. Siguin les declaracions de tipus següents:

Tipus *PERSONA* = ***record***

sexe: (home, dona);

seg: ↑PERSONA

end*;*

Var *A, B: ↑PERSONA*

Si la situació de les variables *A* i *B* en un cert moment és tal com s'il·lustra a la figura 1.12, després d'executar les sentències:

A↑.sexe := B↑.sexe;

A↑.seg := B↑.seg;

l'estat de les variables serà el que s'il·lustra a la figura 1.13.

Cal destacar que l'execució de les dues sentències anteriors equival a l'execució de l'expressió següent:

A↑ *:=B↑*

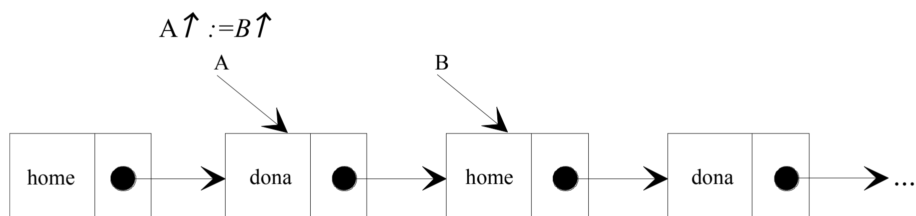


Figura 1.12. Llista lineal

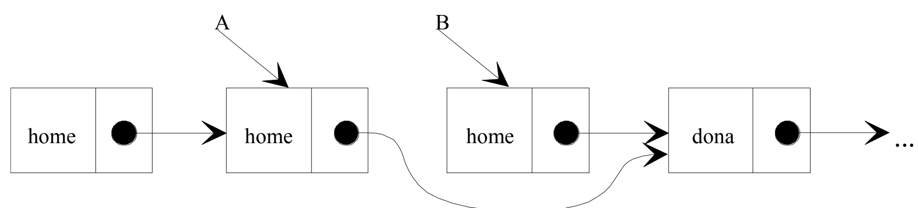


Figura 1.13. Llista lineal

Els cursors

Un cursor és una cel·la valuada entera emprada per fer referència a les cel·les d'un array. La figura 1.14 il·lustra un exemple d'una estructura amb punters i cursors.

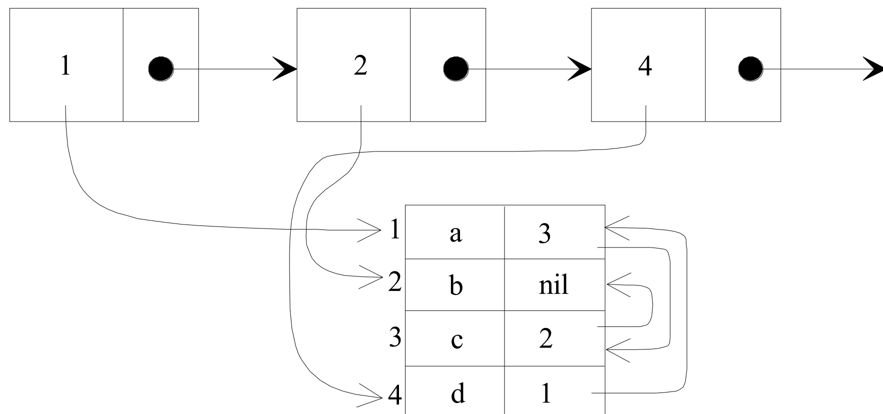


Figura 1.14. Estructura amb punters i cursors

D'acord amb la figura 1.14, hi ha una llista lineal encadenada amb punters, en la qual el primer camp de les cel·les de la llista és un camp cursor, que fa referència a posicions de l'array que forma part de l'estructura. A més, el segon camp de les cel·les de l'array és també un camp cursor, el qual fa referència a posicions dins l'array.

L'ús de cursors té les arrels en la manca de tècniques que permetin definir punters a les cel·les d'un array, estructura aquesta molt emprada per emmagatzemar col·leccions de dades diverses i amb organitzacions dispars, com ara llistes lineals i estructures jeràrquiques, per exemple els arbres.

Llenguatges de programació com el PASCAL no compten amb el tipus cursor entre els possibles tipus permesos, per la qual cosa en aquesta seccions i posteriors se n'ometrà la declaració com un tipus formal del llenguatge.