a.a. 1998-99

Dispense di Programmazione 1

E. Giovannetti

versione provvisoria del 05/11/98

Capitolo 1 1
Introduzione 1
1.1 Programmi, linguaggi, calcolatori. 1
1.2 Lingue e dialetti. 4
1.3 Sintassi e semantica. 4
1.4 Primi programmi. 5
1.5 Variabili, valori, espressioni. 7
1.5.1 Variabili 7
1.5.2 Espressioni. 8
1.5.3 Valori, tipi di valori, rappresentazioni. 8
1.6 L'istruzione di assegnazione. 10
1.7 Istruzioni strutturate: le istruzioni condizionali. 11
1.7.1 Introduzione. 11
1.7.2 Le istruzioni *if-then-else* e *if-then*. 11
1.7.3 L'istruzione *case*. 12
1.8 Istruzioni strutturate: le istruzioni di iterazione. 12
1.8.1 L'istruzione *for*. 13
1.8.2 L'istruzione *while*. 16
1.8.3 Ciclo *while* e non-terminazione. 17
1.8.4 L'istruzione *repeat*. 18
1.8.5 *For* e *repeat* espressi per mezzo del *while*. 18
1.9 I valori booleani 19
1.10 Istruzioni condizionali e iterative: 21
una semantica piú precisa. 21
1.11 Valutazione *pigra* delle espressioni booleane. 22
1.12 La primitiva *break*. 23
1.13 Riepilogo: la nozione di stato. 25
1.14 Stile di programmazione e uso delle variabili. 25
1.15 Esercizi. 27
Capitolo 2 28
Procedure e funzioni. 28
2.1 Introduzione. 28
2.2 Procedura senza parametri e senza variabili. 28
2.3 Procedura con variabili locali. 30
2.4 Allocazione dinamica e allocazione statica 31
delle aree-dati locali. 31
2.5 Parametri per valore. 32
2.6 Parametri per riferimento. 35
2.7 Funzioni. 42
2.8 Particolarità delle funzioni. 43
2.9 Record di attivazione di funzioni. 46
2.10 Chiamate annidate di sottoprogrammi. 48
2.11 Nomi e visibilità. 50
2.12 Lo *stack* (la pila) dei record di attivazione. 51
2.13 Parametri formali di un sottoprogramma passati come argomenti ad un altro sottoprogramma. 55
2.13.1 Introduzione. 55
2.13.2 Parametro per valore passato come argomento per valore. 56
2.13.3 Parametro per valore passato come argomento per riferimento. 57
2.13.4 Parametro formale per riferimento 58

passato come argomento per valore. 58
2.13.5 Parametro formale per riferimento 59
passato come argomento per riferimento. 59
2.14 La primitiva *exit*. 60
Capitolo 3. 61
Tipi. 61
3.1 Numeri reali. 61
3.2 Tipi numerici diversi e conversioni. 61
3.3 Tipi enumerati. 62
3.4 Tipi strutturati. 62
3.4.1 Tipi di valore composti. 62
3.4.2 Tipi di variabile composti 63
3.4.3 Tipi strutturati in Pascal: i tipi *array*. 63
3.4.4 Tipi strutturati in Pascal: i tipi *record*. 65
3.4.5 Equivalenza fra tipi strutturati. 66
3.5 Passaggio di parametri di tipi strutturati. 67
Capitolo 4 70
Correttezza e complessità. 70
4.1 Problemi di programmazione. 70
4.2 Il principio di induzione matematica (semplice). 73
4.3 Induzione e progettazione di cicli: introduzione. 75
4.3.1 Un esempio: la somma di una sequenza di numeri. 75
4.3.2 Un altro esempio: il massimo di una sequenza. 78
4.4 Correttezza e induzione: invariante di ciclo. 80
4.4.1 L'invariante negli esempi del massimo e della somma. 80
4.4.2 L'esempio dell' esponenziazione ingenua. 81
4.4.3 Test del *while* e condizione di uscita. 82
4.4.4 Osservazioni. 84
4.4.5 Dimostrazione di correttezza dell'esponenziale ingenuo. 84
4.4.6 Prudenza e sicurezza. 86
4.4.7 Un altro esempio: esponenziale ingenuo con ciclo discendente. 86
4.5 Tempo di calcolo: introduzione. 88
4.6 Un'applicazione: l'esponenziale veloce. 89
4.6.1 Costruzione del programma. 89
4.6.2 Terminazione e correttezza totale. 92
4.6.3 Complessità. 93
4.7 Uno degli algoritmi piú vecchi del mondo: l'algoritmo di Euclide per il MCD. 94
4.7.1 L'invenzione dell'algoritmo. 94
4.7.2 Analisi della complessità. 97
4.8 Correttezza dei programmi con risultato booleano. 98
4.9 Dimostrazioni e regole di inferenza. 100
4.10 Regole per altri costrutti del Pascal. 102
4.11 Dimostrazioni di correttezza e annotazioni. 104
4.12 Un esempio di programma annotato. 105
4.13 Complessità asintotica: definizioni. 107
4.13.1 La notazione asintotica nell'analisi matematica. 107
4.13.2 Alcune proprietà. 107
4.13.3 Complessità di algoritmi e problemi: definizioni. 108
4.14 Un esempio di risoluzione ottima di un problema. 110
Capitolo 5 113
Programmazione iterativa con i vettori. 113

5.1 I tipi vettore. 113
5.1.1 Parametri array aperti. 114
5.1.2 Vettori di lunghezza effettiva variabile o vettori "parzialmente riempiti". 115
5.2 Vettori di record. 116
5.3 Iterazione sui vettori. 117
5.4 Terminologia e notazione. 117
5.5 Ricerca sequenziale in un vettore non ordinato. 118
5.5.1 Considerazioni generali e soluzione TurboPascal. 118
5.5.2 Soluzione TurboPascal senza uscite forzate. 119
5.5.3 Dimostrazione di correttezza della soluzione precedente (traccia). 120
5.5.4 Versione che restituisce l'indice dell'elemento trovato. 122
5.5.5 Soluzioni Pascal Standard. 123
5.5.6 Un esempio di programma principale. 124
5.6 Funzione che stabilisce se un vettore è ordinato. 125
5.6.1 Costruzione della funzione. 125
5.6.2 Dimostrazione di correttezza (traccia) 126
5.7 Ricerca del primo elemento uguale alla somma dei *k* precedenti. 127
5.8 Cancellazione da un vettore (con compattazione) di tutti gli elementi uguali ad un dato valore. 129
5.9 Il piú lungo segmento ordinato. 132
5.10 Ricerca di una stringa in un testo. 134
5.10.1 Costruzione della soluzione con un solo *while*. 134
5.10.2 Dimostrazione di correttezza. 137
5.11 Realizzazione del crivello di Eratostene 140
con uso di un vettore di booleani. 140
Capitolo 9. 143
Primi algoritmi di ordinamento. 143
9.1 Uguali ma diversi: stabilità. 143

Capitolo 1
Introduzione

1.1 Programmi, linguaggi, calcolatori.

Un programma è un insieme strutturato di istruzioni che può dirigere un calcolatore a risolvere un dato problema o eseguire un dato compito; simmetricamente, un calcolatore è una macchina in grado di (e)seguire un insieme strutturato di istruzioni per risolvere un problema o eseguire un compito specificato.

Un tale insieme strutturato di istruzioni deve essere composto a partire da certi ben determinati costituenti di base secondo regole ben precise - meccaniche -, cioè i costituenti e le regole per la cui esecuzione la macchina è stata progettata.

I primi calcolatori potevano essere considerati composti di tre parti, secondo il noto modello di von Neumann, valido - nelle sue linee generali - ancora oggi:

- una memoria, da concepirsi metaforicamente come un insieme di contenitori o "scatole" ognuno dei quali, identificato da un nome (o "indirizzo") univoco e immutabile, può contenere un dato numerico o piú generalmente simbolico (ad es. un numero intero, un numero con la virgola, un carattere alfabetico, ecc.) che può variare nel tempo. Naturalmente i dati non numerici (caratteri, colori, ecc.) sono codificati come numeri.

- una (o piú) unità di comunicazione con l'esterno, anticamente lettori e perforatori di schede e telescriventi, oggi tastiere, schermi, microfoni, altoparlanti, ecc.;

- una unità centrale di elaborazione, o *cpu*, che è in grado di eseguire dei programmi costituiti da sequenze di ben determinati tipi di *istruzioni-di-macchina*, che in generale modificano lo stato della memoria, cioè i contenuti delle celle di memoria;

Una istruzione-di-macchina è costituita a sua volta, di solito, dalla specifica del tipo di operazione aritmetica o logica da eseguire (ad esempio somma, o sottrazione, o confronto, ecc.) e dalle identità degli operandi, cioè dai nomi (o meglio, indirizzi) dei contenitori in cui andare a prendere gli operandi (ad es. gli addendi) e in cui andare a mettere il risultato (ad es. il risultato dell'addizione); oltre a queste vi sono delle istruzioni "di salto" che permettono di "saltare" ad eseguire un'istruzione diversa da quella successiva, in particolare permettono di "saltare" o no ad un'altra parte del programma a seconda del risultato di un certo confronto od operazione (istruzioni di salto condizionato); possiamo infine assumere, per semplicità, che vi siano delle istruzioni di input/output per comunicare con l'esterno (anche se la realtà è piú complessa).

Esempio: un programma che dati in ingresso due numeri interi ne calcola la somma, e poi se il risultato è negativo o nullo vi aggiunge 15, altrimenti vi sottrae un terzo numero che richiede in ingresso, e infine comunica all'esterno il risultato:

(programma 1)

istruz.1: INPUT 2025
(preleva un numero dal dispositivo di input e memorizzalo nella cella 2025)

istruz.2: INPUT 2026
(preleva un numero dal dispositivo di input e memorizzalo nella cella 2026)

istruz.3: ADD 2025 2026 2027
(leggi i contenuti delle celle 2025 e 2026, calcolane la somma,
e metti il risultato nella cella 2027)

istruz.4: BPOS 7 (Branch on Positive to instruction 7)
(se il risultato dell'ultima operazione è > 0, salta all'istruz.7)

istruz.5: ADDI 15 2027 (ADD Immediate 15 to...)
(leggi il contenuto della cella 2027, calcolane la somma con 15,
e metti il risultato di nuovo nella cella 2027)

istruz.6: BR 9
(salta all'istruzione 9)

istruz.7: INPUT 2028
(preleva un numero dal dispositivo di input e memorizzalo nella cella 2028)

istruz.8: SUB 2027 2028 2027
(leggi i contenuti della cella 2027 e della cella 2028, fanne la differenza,
e metti il risultato di nuovo nella cella 2027)

istruz.9: OUTPUT 2027
(leggi il contenuto della cella 2027 e invialo al dispositivo di output)

istruz.10: HALT
(ferma l'esecuzione)

La sequenza di istruzioni-di-macchina costituente un programma, per poter essere eseguita dal calcolatore, deve anch'essa essere codificata in forma numerica; in particolare devono esserlo i nomi simbolici delle istruzioni, come ADD, SUB, ADDI, ecc.: ad esempio ADD sarà codificato con il numero 214, ADDI con il numero 215, SUB con il numero 216, ecc.; l'istruz.3 sarà allora codificata come 214 2025 2026 2027, ecc. Un programma completamente codificato in forma numerica e direttamente eseguibile dallo hardware di una macchina viene detto programma in *linguaggio-macchina*; vi sono naturalmente tanti linguaggi-macchina diversi quanti sono i tipi di cpu esistenti.

Come si vede, la strutturazione di questo tipo di programma - cioè il modo in cui le istruzioni debbono o possono venire "disposte" - è molto povera, poichè coincide con la semplice sequenza (tanto che in tal caso si parla di programmi non strutturati).

Anche oggi la forma finale di programma che viene eseguita dallo hardware di una macchina è simile a quella sopra ricordata. Tuttavia raramente i programmi vengono scritti in tale forma; vengono invece scritti, per lo piú, in un cosiddetto *linguaggio di alto livello*, di piú facile concezione e comprensione per l'uomo, con una strutturazione piú ricca ed articolata che la semplice sequenza, e indipendente (almeno in linea di principio) dalla particolare macchina. In un linguaggio di alto livello il programma precedente verrebbe scritto in un modo simile al seguente:

(programma 1')

```
read(a);
read(b);
c:= (a+b);
if c<=0 then c:= c+15
else
  begin
    read(d);
    c:= c-d
  end;
write(c).
```

Uno speciale programma detto *compilatore* traduce poi tali programmi "di alto livello" in programmi in linguaggio-macchina, che possono essere - come prima - eseguiti direttamente dal calcolatore; in questo caso tradurrebbe il programma 1' nel programma 1.

Il Pascal è uno di tali linguaggi di alto livello. Un programma Pascal è un testo composto secondo certe regole ben precise, affinchè il programma compilatore, progettato in base a tali regole, sia in grado di tradurlo univocamente. Un programma Pascal è quindi analogo ad un discorso (un testo) in una lingua naturale (italiano, francese, ecc.), che deve obbedire ad una certa grammatica o sintassi. La grammatica dei linguaggi di programmazione è però molto piú semplice e rigida di quelle delle lingue naturali.

Per quanto riguarda invece la *semantica*, ossia la descrizione di ciò che un dato programma Pascal "fa" quando viene (compilato e poi) eseguito, è conveniente - all'inizio - considerare il traduttore (o compilatore) Pascal e la macchina che esegue il programma tradotto (insieme ad altri programmi come il collegatore, il caricatore, ecc.) come costituenti complessivamente un virtuale *esecutore* Pascal il quale legge un programma Pascal e ne elabora le istruzioni (e, come vedremo, le "dichiarazioni") generando il corrispondente processo di calcolo.

Naturalmente il processo di calcolo cosí generato sarà, nei suoi dettagli fisici, diverso per ogni diverso tipo di macchina fisica; adottando però un punto di vista sufficientemente astratto si può parlare in maniera univoca di processo di esecuzione di un programma, e si può anzi descrivere il funzionamento dell'esecutore Pascal o *macchina virtuale Pascal* senza preoccuparsi di come essa sia effettivamente realizzata. Nel seguito faremo spesso riferimento implicito, per descrivere il comportamento dei programmi, a una tale macchina virtuale.

Si osservi che, proprio perchè il funzionamento della macchina (sia quella "reale" sia quella virtuale) è determinato da "istruzioni" e non semplicemente dai "dati" contenuti in memoria, lo *stato della macchina* ad ogni dato istante è definito non solo dai contenuti di tutte le celle di memoria su cui stanno operando le istruzioni, ma anche da quale sia l'istruzione che si sta eseguendo, o - meglio ancora - da quale sia la successiva istruzione da eseguire (che può essere quella immediatamente seguente nel testo del programma oppure no), cioè dall'indirizzo dell'istruzione successiva.

D'altra parte il programma è anch'esso, durante l'esecuzione, caricato nella memoria centrale del calcolatore, anche se di solito, a differenza dei dati, non può essere modificato (in realtà nei primi calcolatori i programmi che si automodificavano costituivano una tecnica di programmazione abbastanza comune).

1.2 Lingue e dialetti.

In queste dispense faremo spesso riferimento ad una particolare versione del Pascal, il TurboPascal versione 7, di cui in realtà esistono realizzazioni soltanto per certi tipi di macchine, cioè i PC con sistema operativo DOS o Windows, e non per altri, ad es. i Macintosh o le macchine con sistema operativo Unix. Tale versione del Pascal è infatti quella che gli studenti del primo anno si trovano a dover usare, ed è anche una delle migliori e piú diffuse in commercio per calcolatori personali.

Il TurboPascal è (eccetto che per alcune caratteristiche marginali) un soprainsieme del Pascal standard, nel senso che ogni programma Pascal standard è - in linea di massima - anche un programma TurboPascal, ma non viceversa. Naturalmente, questo vuol dire che un programma che usi le caratteristiche "dialettali" del Turbo non è - come si dice - portabile, cioè non può essere compilato ed eseguito con successo su altri tipi di macchine con compilatori Pascal prodotti da altre case.

Crediamo tuttavia che ciò non costituisca un problema, poichè il Pascal è un linguaggio utilizzato prevalentemente in ambito didattico, mentre nella produzione effettiva del software si usano piuttosto linguaggi come il C, il C++, Java, ecc.

D'altra parte vi sono ancora, riteniamo, delle buone ragioni didattiche per scegliere il Pascal come linguaggio del primo corso di programmazione; inoltre proprio il TurboPascal, nelle sue ultime versioni, ha incorporato alcune utili caratteristiche del C e dei linguaggi più moderni, che permettono al programmatore di esprimere molti algoritmi in modo piú semplice o piú naturale che nel Pascal standard, senza tuttavia dover affrontare subito la meno pulita, piú criptica, e parzialmente diseducativa sintassi del C.

1.3 Sintassi e semantica.

Riassumiamo: i linguaggi di programmazione, esattamente come le lingue naturali, hanno una sintassi (o grammatica) e una semantica.

La sintassi di una lingua è costituita dalle regole di formazione cui deve obbedire una frase o un discorso per poter essere definito una frase o un discorso in quella lingua, cioè per poter avere un significato, indipendentemente da quale sia tale significato (eventualmente assurdo).

La semantica di una lingua o di un linguaggio definisce invece che cosa è tale significato, cioè che cosa un discorso corretto in quella lingua vuol dire.

Ad esempio la frase "Pechino è la capitale degli Stati Uniti" è una frase italiana dotata di significato (di cui si può dire che è falsa), cosí come la frase "dammi la luna", che esprime un ordine (che può essere assurdo in certi contesti e ragionevole in altri), ecc.

A differenza delle lingue naturali, dove al confine fra correttezza e scorrettezza vi è spesso una zona d'incertezza, la sintassi dei linguaggi di programmazione è definita in base a regole precise e non ambigue: per qualunque sequenza di caratteri si può sempre stabilire se essa è un programma sintatticamente corretto oppure no.

Una frase o un discorso in una lingua naturale ha di solito un significato multiforme, ricco di inesauribili sfumature ed ambiguità. Il significato di un programma o di una porzione di programma Pascal è invece univoco e non ambiguo, ed è - come abbiamo detto - la descrizione di ciò che l'esecutore "fa" quando esegue tale programma.

A differenza delle lingue naturali, dove frasi sintatticamente scorrette possono avere comunque un significato, come "io domani partire", nei linguaggi di programmazione un programma sintatticamente scorretto non ha significato, nel senso che non può venir eseguito.

Il controllo della correttezza sintattica viene effettuato dal compilatore: se il programma non è corretto vengono generati i cosiddetti "errori di compilazione", e il programma non viene eseguito; altrimenti viene tradotto in linguaggio macchina ed eseguito dal calcolatore.

Come si è detto, per semplificare la descrizione della semantica è conveniente immaginare l'esistenza di un esecutore (o interprete) Pascal, che esegue - o meglio elabora, o interpreta - direttamente il testo del programma (o *programma sorgente*), benchè alcune delle funzioni di tale esecutore siano in realtà svolte dal compilatore stesso.

Naturalmente un programma sintatticamente corretto può essere semanticamente scorretto, nel senso che può generare degli errori durante l'esecuzione, oppure può non realizzare lo scopo per cui è stato scritto.

Del Pascal studieremo contemporaneamente la sintassi e la semantica; per poter scrivere dei programmi bisogna arrivare a padroneggiare la sintassi con disinvoltura, ma la cosa davvero importante e difficile è la semantica: imparare a scrivere programmi che "facciano" ciò per cui sono stati progettati.

1.4 Primi programmi.

Un programma TurboPascal 7.0 è costituito da, nell'ordine:

- una *intestazione* opzionale - che di solito ometteremo - la cui forma esamineremo piú avanti;
- una *clausola uses* opzionale, che per ora non consideriamo;
- un *blocco*;
- un punto (segno ortografico) finale.

Un programma può quindi essere costituito semplicemente da un *blocco*. A sua volta un blocco è costituito da, nell'ordine:

- una *sezione delle dichiarazioni;*
- una *sezione delle istruzioni;*

Una *sezione delle dichiarazioni* è, come vedremo, una sequenza di *dichiarazioni*, eventualmente vuota. Una *sezione delle istruzioni* è semplicemente un' *istruzione composta*.

Un'*istruzione composta* è una sequenza di una o piú *istruzioni* separate fra loro da punto e virgola, racchiusa fra un `begin` iniziale e un `end` finale.
Le *istruzioni* possono essere di vario genere, come vedremo; la piú semplice è l'*istruzione vuota*, tanto semplice che ... non si scrive!

Seguendo tale grammatica siamo in grado di scrivere il programma TurboPascal corretto piú corto del mondo:

```
begin  end.
```

Tale programma può venire compilato ed eseguito senza errori: naturalmente, è un programma che non fa assolutamente niente!

Le istruzioni composte sono istruzioni, quindi anche il seguente è un programma Pascal sintatticamente corretto:

```
begin
  begin
    begin
    end
  end
end.
```

(costituito da un'istruzione composta a sua volta costituita da un'istruzione composta costituita da ...)

Il seguente è un programma Pascal costituito da un'istruzione composta costituita da tre istruzioni:

```
begin
  begin
  end;
  begin
  end;
  begin
  end
end.
```

Il seguente è un programma Pascal corretto, costituito da dieci istruzioni (vuote, separate da nove ";"):

```
begin ;;;;;;;;; end.
```

Possiamo scrivere infiniti programmi diversi che, esattamente come il primo, non fanno assolutamente niente (in realtà essi sono diversi solo dal punto di vista del *sorgente*, poichè vengono tradotti tutti nello stesso modo in linguaggio macchina, cioè in un'istruzione di terminazione immediata dell'esecuzione).

Per ottenere un programma con un effetto visibile introduciamo fra il `begin` e l'`end` un'istruzione di output, e precisamente un'istruzione che visualizza una scritta sullo schermo:

```
begin
  writeln('ciao bellocci')
end.
```

L'istruzione `writeln('ciao bellocci')` ha l'effetto di far uscire sulla periferica designata come standard output - cioè uscita ordinaria - del programma (di solito lo schermo collegato al calcolatore) la sequenza di caratteri costituenti la scritta *ciao bellocci* (senza apici), andando a capo alla fine.

Scriviamo ora un programma che prenda dall'esterno, attraverso una periferica di input, due numeri interi, ne calcoli la somma e la scriva sullo schermo.

Per far ciò occorre disporre di due "contenitori" o *variabili* in cui memorizzare i valori immessi da tastiera; sostituiamo allora nel programma precedente la sezione delle dichiarazioni vuota (e quindi invisibile!) con una *dichiarazione di variabili* che definisca due variabili di tipo intero, ad esempio di nomi `m` ed `n`, che usiamo poi con l'istruzione `readln`:

```
var m,n: integer;
begin
  readln(m,n);
  writeln(m+n);
end.
```

L'istruzione `readln(m,n)` ha l'effetto di prelevare i primi due numeri interi immessi da tastiera (immessi digitandone le cifre nella usuale notazione decimale, separando il primo numero dal secondo per mezzo di almeno uno spazio o altro carattere "bianco", e pigiando alla fine il tasto ENTER) e depositarli rispettivamente nei "contenitori" `m` ed `n`. Se i numeri non sono ancora stati ancora immessi, l'effetto è quello di sospendere l'esecuzione del programma in attesa che venga completata l'operazione di input.

Si noti che l'immissione avviene soltanto dopo la pressione del tasto ENTER (o RETURN, o INVIO), in modo da permettere di correggere sulla riga di input eventuali errori di battitura.

Si noti inoltre che la tastiera e lo schermo sono due periferiche (anche fisicamente) distinte, rispettivamente di input e di output: non ci si faccia ingannare dal fatto che di solito quando si pigia un tasto della tastiera del calcolatore compare sullo schermo il carattere corrispondente al tasto (pigiato digitando un comando DOS o scrivendo un testo per mezzo di un programma di videoscrittura oppure digitando un numero o una frase in risposta ad una `read`, ecc.) è il sistema operativo - o, per mezzo di esso, il programma di videoscrittura, o la primitiva `read`, ecc. - che invia il carattere allo schermo.

1.5 Variabili, valori, espressioni.

1.5.1 Variabili

La nozione di **variabile** (o meglio, come vedremo, di "variabile assegnabile") incontrata nella sezione precedente è fondamentale nella programmazione in Pascal o in qualunque altro dei linguaggi cosiddetti "imperativi".

Una variabile è, come si è detto, un contenitore che può contenere valori di un determinato tipo; in Pascal, infatti, vi sono tipi diversi di valori, come interi, reali, caratteri, ecc.; ogni variabile è etichettata con un certo tipo, e può contenere soltanto valori di quel tipo.

Per poter usare una variabile, occorre prima crearla o - come si dice meglio - definirla o dichiararla. In particolare, i contenitori che vengono utilizzati nelle istruzioni del programma (non considerando per ora i sottoprogrammi) devono essere dichiarati nella *parte dichiarativa*, cioè prima del `begin`. La dichiarazione delle variabili è costituita dalla parola chiave `var` seguita dai nomi delle variabili con i loro tipi, ad esempio:

```
var m, n: integer;
    r: real;
    s: string;
```

Le variabili definite nella parte dichiarativa di un programma vengono dette **variabili globali** del programma (in opposizione alle variabili locali e variabili dinamiche, che vedremo in seguito), perchè esse vengono "create" all'inizio dell'elaborazione del programma, cioè prima dell'esecuzione della prima istruzione, e vengono "distrutte" solo alla fine dell'esecuzione del programma. Si potrebbe anzi dire che le variabili globali sono create al tempo di compilazione, nel senso che quando il programma compilato viene caricato in memoria centrale per l'esecuzione, insieme ad esso viene "caricato" (o meglio, *allocato*) anche lo spazio per le variabili globali, sicchè quando inizia l'esecuzione del programma esse "ci sono già", nel senso che nessuna istruzione di macchina deve essere eseguita per "crearle".

In altre parole: la traduzione (compilazione) della parte dichiarativa di un programma non genera nessuna istruzione-di-macchina (ma viene utilizzata nella traduzione della parte-istruzioni). Per questo le variabili globali vengono talvolta dette anche variabili statiche, e l'area di memoria in cui sono contenute viene chiamata *area-dati statica*.

1.5.2 Espressioni.

La semplice espressione aritmetica `m+n` usata come argomento della `writeln` è un esempio di *espressione* anche nel senso dei linguaggi di programmazione, in questo caso il Pascal.

Quando durante l'esecuzione di un programma l'esecutore Pascal (o di altro linguaggio dello stesso tipo) incontra un'espressione, la *valuta*, ossia tenta di ridurla - secondo ben determinate regole - ad un valore, proprio come uno studente di scuola media di fronte ad un'espressione aritmetica; le regole sono naturalmente le stesse, e altrettanto naturalmente i nomi di variabili denotano (cioè indicano) i valori in esse contenuti**.**

1.5.3 Valori, tipi di valori, rappresentazioni.

I valori e le espressioni di tipo intero non sono gli unici possibili in Pascal. Vi sono i valori e le espressioni degli altri tipi numerici, come i numeri con la virgola (in diversi formati); vi sono i caratteri, e - in TurboPascal - le *stringhe*, cioè le sequenze di caratteri.

Non si confondano i valori, che sono degli enti astratti, con le loro rappresentazioni sia esterne (sullo schermo o sulla tastiera) che interne (nella memoria del calcolatore), sia concrete che astratte: il numero 15 non è nè la coppia di cifre 1 e 5, nè la sua rappresentazione binaria per mezzo di dispositivi elettronici all'interno del calcolatore. Analogamente, il carattere "a minuscolo", che in Pascal viene indicato come `'a'`, non coincide nè con il segno d'inchiostro impresso dalla stampante su questo foglio, nè con il numero 97 con cui è rappresentato all'interno del calcolatore in base al codice ASCII.

La rappresentazione di un ente astratto può essere realizzata per mezzo di enti astratti di un altro tipo: ad esempio il numero 3, che ovviamente non è un oggetto materiale, può essere rappresentato dal carattere "3", che è un ente astratto il quale può essere a sua volta rappresentato da un numero, ad esempio nella codifica ASCII dal numero 51, che a sua volta è rappresentato dalla sequenza di cifre binarie 00110011, la quale a sua volta è rappresentata dallo stato di certi microscopici circuiti elettronici all'interno di un frammento di silicio... Ma il numero 3 non deve essere confuso con il carattere "3" (in Pascal `'3'`)!

A questo punto ci si potrà domandare: se nel calcolatore tutto è rappresentato per mezzo di numeri, come fa la macchina a distinguere ad esempio fra il numero 51 inteso per se stesso e il numero 51 inteso come rappresentazione del carattere "3"? entrambi sono rappresentati dalla sequenza binaria 00110011!

La risposta è che sono diverse le operazioni che vengono compiute su di essi!

Quando il numero 97 è la rappresentazione del carattere "a minuscolo" esso non verrà mai, in quanto tale, sottratto o sommato con altri numeri, ma soltanto tradotto, ad esempio, in una matrice di pixel da inviare allo schermo. La stessa cosa quando il numero 51 è la rappresentazione ASCII del carattere "3".

Abbiamo allora la seguente importante equazione fra concetti:

TIPO DI DATO = STRUTTURA ALGEBRICA

Infatti un tipo di dato è costituito (o caratterizzato) da un insieme di elementi e dalle operazioni definite su di esso: che è proprio la definizione generale di *struttura algebrica*.

Ritornando all'esempio dei numeri e dei caratteri, facciamo notare che si può passare da un numero mono-cifra al carattere che lo rappresenta (e viceversa) semplicemente aggiungendo (o sottraendo) 48 (poichè nel codice ASCII i caratteri numerici da 0 a 9 sono rappresentati dai numeri da 48 a 57 nello stesso ordine).

Cosí, quando ad esempio in risposta ad una `read(n)`, con `n` variabile di tipo `integer`, si pigia il tasto col carattere "3" e poi il tasto ENTER, il calcolatore - dopo alcuni passaggi intermedi che non descriviamo - riceve il numero 51; la primitiva `read`, dovendo leggere un intero, gli sottrarrà 48 ottenendo cosí il numero 3, che sarà memorizzato nella variabile `n`. Se invece dopo il tasto "3" pigiassi, prima di ENTER, il tasto "5", il calcolatore riceverebbe la coppia (ordinata) di numeri 51 e 53; la primitiva `read`, dopo averli trasformati rispettivamente nei numeri 3 e 5, effettuerebbe il calcolo 3*10 + 5, per ottenere finalmente il numero 35.

Il numero cosí ottenuto potrà essere allora sommato, moltiplicato, ecc. con altri numeri, per produrre dei risultati che eventualmente saranno, nel corso di una `write`, ritradotti in sequenze di codici ASCII che infine si trasformeranno in configurazioni di pixel sullo schermo...

Per scrivere programmi in Pascal la rappresentazione interna dei diversi tipi di dato di solito non ci interessa; ad esempio, la conversione fra caratteri e numeri è fatta automaticamente dalle primitive di lettura e scrittura, ecc. La sua conoscenza può tuttavia essere utile in casi particolari.

1.5.4 Variabili, indirizzi, valori di sinistra (l-valori).

Nella sottosezioni precedenti abbiamo introdotto una netta separazione fra variabili da un lato e valori dall'altra, associata alla metafora del contenitore e del contenuto, che è molto importante per aiutare il principiante a padroneggiare il nuovo concetto di variabile, evitando di confonderla appunto con il suo contenuto.

Vedremo nella seconda parte del corso che in realtà anche le variabili possono essere considerate dei valori, semplicemente di un tipo diverso. Se infatti, come abbiamo detto, valore può essere un qualunque ente astratto, e un tipo è un insieme di valori su cui sono state definite certe operazioni, allora anche l'insieme - poniamo - di tutte le possibili variabili di tipo *integer* può essere visto come un insieme di valori di un nuovo tipo, distinto dal tipo *integer*, che potremmo chiamare ad esempio *var integer.* Sui valori di questo nuovo tipo non sono definite le usuali operazioni aritmetiche (non sono

degl'*integer!*), ma soltanto le operazioni di estrazione del contenuto, e di modifica del contenuto (assegnazione).

Da un punto di vista astratto tali *valori-variabile* - talvolta detti *l-valori*, ossia valori di sinistra (*left-values*) - potrebbero essere identificati con i loro indirizzi, come avviene in alcuni linguaggi dotati di una fondazione matematica piú chiara. Nei linguaggi come il C e il TurboPascal, invece, è piú conveniente considerare una *variabile*, il suo *contenuto*, e il suo *indirizzo* come tre entità distinte, anche se, come vedremo nella prossima sezione, il nome di una variabile può denotare, a seconda del contesto, sia la variabile (o l-valore) che il suo contenuto.

1.6 L'istruzione di assegnazione.

L'istruzione di assegnazione è l'istruzione per mezzo della quale si possono cambiare i contenuti delle variabili, cioè i loro valori. Ad esempio, se `n` è il nome di una variabile di tipo *integer*, l'esecuzione dell'istruzione `n:= 5` ha l'effetto di depositare il valore 5 nel "contenitore" `n`. Se `n` ed `m` sono i nomi di due variabili di tipo numerico, l'esecuzione dell'istruzione `n:= n+m` calcola la somma dei valori contenuti in `n` e in `m`, e deposita il risultato in `n`, cancellandone cosí il precedente valore.

Un nome di variabile ha dunque un significato diverso a seconda che compaia a destra o a sinistra del simbolo di assegnazione; a destra esso denota semplicemente il valore contenuto nella variabile; a sinistra, invece, denota il contenitore stesso.

Se indicassimo esplicitamente con un simbolo, ad esempio "!", l'operazione "contenuto di" citata nella sezione precedente (cioè se scrivessimo *!a* per indicare il *contenuto di a*), l'istruzione *a:= a+b* dovrebbe a rigore essere scritta: *a := !a + !b*, evidenziando cosí, oltre all'asimmetria dell'operatore di assegnazione, il fatto che la somma è fra i contenuti, non fra le "variabili". In qualche linguaggio di programmazione fondato su modelli di calcolo diversi (da quello cosiddetto *imperativo*) dove l'uso di variabili assegnabili non è cosí frequente, viene effettivamente adottata tale notazione. Nella sintassi dei linguaggi del genere del Pascal, invece, poichè l'operazione di estrazione del contenuto di una variabile è di uso continuo nei programmi, essa è lasciata implicita nel contesto.

Dunque, a sinistra del simbolo di assegnazione ci deve essere il nome di una variabile (o, piú in generale, un *l-valore*: vedremo infatti, con i vettori ed i puntatori, che vi possono essere espressioni equivalenti a variabili, cioè espressioni che hanno per valore una variabile), non ci può essere un valore numerico, carattere, ecc.; un'istruzione `3:=` ... è priva di senso, il valore 3 non è un contenitore in cui si possa memorizzare qualcosa.

Analogamente, gli argomenti della `readln`, al contrario di quelli della `writeln`, non possono essere dei valori, nè piú in generale delle espressioni, ma soltanto delle variabili; infatti un'istruzione ad es. della forma `readln(5)` dovrebbe leggere qualcosa (un intero?) da tastiera e "depositarlo" nel numero 5!?

L'istruzione di assegnazione non ha corrispettivo nel linguaggio matematico tradizionale, in cui si possono incontrare forme apparentemente simili, come ad esempio le frasi *sia m = 1024* oppure *sia a = b · h/2*, seguite da un'espressione aritmetica o da un discorso in cui compare il nome *m* o il nome *a*; con esse però si attribuisce semplicemente un nome simbolico ad una costante o al risultato di un'espressione, cioè si usa il nome simbolico come *sinonimo* di un valore. Il valore cosí attribuito non può essere cambiato, a meno che si inizi un altro discorso in cui si riusi lo stesso nome come sinonimo di un'entità diversa. Ad esempio, potremmo scrivere le tabelline della scuola elementare nel modo seguente:

...
Sia $n = 5$*: allora* $1 \cdot n = 5$*;* $2 \cdot n = 10$*;* $3 \cdot n = 15$*; ...*
Sia $n = 6$*: allora* $1 \cdot n = 6$*;* $2 \cdot n = 12$*;* $3 \cdot n = 18$*; ...*
...
Non possiamo tuttavia scrivere frasi della forma: *sia* $n = n+1$. Infatti essa afferma l'uguaglianza fra n ed $n+1$, ed è quindi un'equazione che non ha soluzioni finite.

L'istruzione di assegnazione `n:= n+1`, invece, semplicemente incrementa di 1 il contenuto della variabile `n`. La nozione di variabile come "contenitore" è estranea alla matematica tradizionale.

1.7 Istruzioni strutturate: le istruzioni condizionali.

1.7.1 Introduzione.

Il modello di macchina virtuale Pascal che emerge dalle sezioni precedenti è, come si vede, abbastanza simile al modello classico di calcolatore descritto nella *sezione 1.1*: un programma è costituito dalla definizione di un insieme di celle di memoria da usare come spazio di lavoro, e da una <u>sequenza</u> di istruzioni di calcolo che permettono di modificare i contenuti di tali celle.

In generale, nel corso di un calcolo, si deve poter scegliere se eseguire una oppure un'altra sequenza di istruzioni a seconda del verificarsi o no di una certa condizione riguardante il risultato di un calcolo precedente oppure un evento esterno.

Ad esempio, nella risoluzione di un'equazione di secondo grado si deve calcolare il discriminante, e poi intraprendere azioni diverse a seconda che esso risulti positivo, negativo, o nullo.

Oppure potremmo aver bisogno di ripetere una stessa sequenza di istruzioni un certo numero di volte, ecc.

Per questo accanto alle *istruzioni semplici* (come l'assegnazione) vi sono in Pascal le *istruzioni strutturate*; fra queste le *istruzioni condizionali*.

1.7.2 Le istruzioni *if-then-else* e *if-then*.

L'*istruzione if-then-else* ha la forma:

```
if Condizione then Istruzione1 else Istruzione2
```

Il suo significato è intuitivamente semplice: se la *Condizione* è vera viene eseguita l'*Istruzione1*, se è falsa viene eseguita l'*Istruzione2*. Poi si passa all'istruzione successiva. Le due istruzioni *Istruzione1* e *Istruzione2* possono essere istruzioni di qualunque genere, in particolare possono essere istruzioni composte, cioè sequenze di istruzioni racchiuse fra un *begin* e un *end*.

Esempio: un programma che chieda il nome dell'utente, e poi lo saluti chiamandolo per nome con *Buongiorno* o *Buonasera* a seconda che sia mattino o pomeriggio.

Nel modulo (o *unit*) TurboPascal di nome `dos`, che è un insieme di primitive non facenti strettamente parte del linguaggio, vi è una primitiva `gettime` che legge l'orologio interno del calcolatore e restituisce ora, minuti, secondi, e centesimi in quattro "contenitori" (cioè variabili!) che gli devono essere forniti dal programmatore, e devono

inoltre essere di tipo *word* (cioè intero senza segno); dichiariamo allora quattro variabili con dei nomi opportuni, anche se della seconda, terza e quarta non ce ne faremo niente. Dopo l'invocazione della la primitiva `gettime`, ordiniamo di eseguire un'istruzione o un'altra a seconda che il contenuto della variabile `ora` sia o no minore di 13.

```
uses dos; {necessario per poter usare la primitiva gettime}
var s: string;
    ora, min, sec, cent: word;
begin
  write('come ti chiami?'); readln(s);
  gettime(ora,min,sec,cent);
  if ora < 13 then writeln('Buongiorno, ',s)
  else writeln('Buonasera, ',s)
end.
```

Note:

Una variabile di tipo *string* è un contenitore che può contenere sequenze di caratteri di lunghezza arbitraria (fino ad un massimo stabilito dall'implementazione, ad es. 256).

I nomi `ora`, `min`, ecc., essendo nomi di variabili definite dal programmatore, sono arbitrari, si sarebbe potuto chiamarli `hour`, `minute`, ecc; ciò che non è arbitrario è il fatto che la primitiva *gettime* metta nel primo argomento l'ora, nel secondo argomento i minuti, ecc. Quindi, se avessimo scritto `gettime(min,ora, ...` avremmo poi ottenuto le ore in `min` e i minuti in `ora`.

L'*istruzione if-then* ha la forma:

```
if Condizione then Istruzione1
```

Essa equivale ad un istruzione *if-then-else* con istruzione vuota nel ramo *else*:

```
if Condizione then Istruzione1 else; ...
```

<vedi esempi sui lucidi>
<inserire altri esempi>

1.7.3 L'istruzione *case*.

<vedi qualunque manuale Pascal, e un esempio sui lucidi>

Esercizio 1. Scrivere un programma che chieda il nome e il sesso dell'utente (m o f), e in risposta visualizzi sullo schermo la scritta "Ciao, caro ..." o "Ciao, cara ..." appropriata.
Esempio:
Come ti chiami?
Ada
Ciao, cara Ada!

1.8 Istruzioni strutturate: le istruzioni di iterazione.

1.8.1 L'istruzione *for*.

Supponiamo di voler visualizzare cinque volte di seguito sullo schermo la scritta: "Buongiorno, ragazzi!". Naturalmente possiamo scrivere cinque volte l'istruzione di scrittura:

```
begin
  writeln('Buongiorno, ragazzi!');
  writeln('Buongiorno, ragazzi!');
  writeln('Buongiorno, ragazzi!');
  writeln('Buongiorno, ragazzi!');
  writeln('Buongiorno, ragazzi!');
end.
```

Potremmo però risparmiare spazio-istruzioni usando un'istruzione iterativa, precisamente un'istruzione di ripetizione della stessa istruzione 5 volte:

```
var i: integer;
begin
  for i:= 1 to 5 do writeln('Buongiorno, ragazzi!');
end.
```

Supponiamo ora di voler modificare il nostro programma in modo che la scritta venga visualizzata un numero di volte fornito dall'utente (tramite tastiera) durante l'esecuzione.

In questo caso, a differenza del precedente, il numero N di ripetizioni non è noto al momento della scrittura del programma; non è quindi possibile, come nel primo - inelegante - dei due programmi precedenti, scrivere N volte l'istruzione *writeln*, è invece indispensabile adottare il secondo metodo. Cioè: *invece di ripetere N volte la scrittura di una stessa istruzione, scrivere un'istruzione che faccia eseguire N volte quell'istruzione!*

Naturalmente bisognerà disporre di una variabile `n` in cui depositare il numero N immesso da tastiera.

```
var i,n: integer;
begin
  write('quante ripetizioni? ');
  readln(n);
  for i:= 1 to n do writeln('Buongiorno, ragazzi!')
end.
```

Un altro esempio: la somma di N interi immessi da tastiera, con N immesso anch'esso da tastiera (naturalmente prima!):

```
var i,n,x,somma: integer;
begin
  write('quanti numeri vuoi sommare? ');
  readln(n);
  somma:= 0;
  write('immetti ',n,' numeri: ');
  for i:= 1 to n do begin
    read(x);
    somma:= somma + x;
  end;
  writeln('la somma è ',somma)
end.
```

L'istruzione (o sequenza di istruzioni) che viene ripetuta, cioè l'istruzione che segue il *do* (ossia, di solito, la sequenza di istruzioni compresa fra il *do begin* e l'*end*), viene generalmente detta *corpo del ciclo*.

Il principiante potrebbe chiedersi perchè nell'istruzione *for* ci debba essere una variabile esplicita (come la `i` degli esempi) che funziona da contatore; negli esempi precedenti basterebbe avere un'istruzione della forma "fai n volte", ad es.:

```
(* ATTENZIONE: NON È PASCAL *)
do n times begin
  read(x); somma:= somma + x
end;
```

In Cobol un'istruzione simile esiste proprio: *perform n times*. La ragione della sua assenza in tutti i linguaggi moderni è che molto spesso il valore del contatore deve essere utilizzato all'interno del ciclo; poichè d'altra parte nella traduzione in linguaggio-macchina un contatore deve naturalmente esserci, tanto vale richiederlo comunque anche nel linguaggio ad alto livello.

Il primo esempio di utilizzo del contatore è un semplice programma che scrive sullo schermo il testo della nota canzoncina-filastrocca infantile "2 elefanti si dondolavano ...":

```
const verso1 = ' elefanti si dondolavano sopra un filo di ragnatela';
      verso2 = 'e giudicando la cosa interessante';
      verso2 = 'andarono a chiamare un altro elefante';

var i: integer;

begin
  for i:= 2 to 10 do begin
    writeln(i, verso1); writeln(verso2); writeln(verso3);
    readln; (* attende che l'utente pigi il tasto ENTER *)
  end;
end.
```

Un altro esempio è il calcolo del fattoriale di un numero naturale, secondo la nota definizione $n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$:

```
var i, n: integer;
    fact: longint;
begin
  readln(n);
  fact:= 1;
  for i:= 1 to n do fact:= i*fact;
  writeln(fact)
end.
```

Si noti che, come nel programma per la somma di una sequenza di numeri, anche qui si ha una variabile in cui si "accumula" via via il risultato; come là era necessario inizializzare tale variabile a 0, che è l'elemento neutro della somma, cosí ora è necessario inizializzarla a 1, elemento neutro del prodotto (poichè l'esecutore TurboPascal inizializza automatica-mente le variabili globali a 0, nel caso della somma il programma in realtà funziona anche senza inizializzazione esplicita, che è invece nel prodotto è indispensabile).

La forma generale dell'*istruzione for* è:

```
for Variabile:= EsprIniz to EsprFin do Istruzione
```

<*inserire diagramma di flusso del for: vedi lucidi*>

Il contatore deve essere una ordinaria variabile di tipo intero (di nome qualunque!), opportunamente dichiarata; gli estremi inferiore e superiore *EsprIniz* e *EsprFin* non sono necessariamente due costanti, ma possono essere due arbitrarie espressioni di tipo intero; come si vede nel diagramma di flusso, tali espressioni vengono dall'esecutore valutate una sola volta all'inizio del ciclo e memorizzate (in due locazioni temporanee, ad es. registri, invisibili al programmatore); i successivi confronti del contatore vengono fatti con il valore dell'estremo superiore calcolato all'inizio una volta per tutte, anche quando il valore dell'espressione cambia durante il ciclo. Si consideri la seguente bizzarra porzione di programma:

```
readln(n);
for i:= 1 to n do n:= n+1;
```

Contrariamente a ciò che qualcuno potrebbe aspettarsi, il precedente non solo è un programma Pascal perfettamente corretto (anche se di stile assolutamente sconsigliabile!), ma è un programma che termina dopo N iterazioni, dove N è il valore immesso in `n` da tastiera; il valore di n all'uscita dal *for* è 2N.

Il numero di iterazioni del corpo è quindi sempre dato da *(EsprFin - EsprIniz + 1)*, se è *EsprIniz* *EsprFin*; se invece il valore iniziale è maggiore di quello finale il corpo del ciclo non viene eseguito neppure una volta.

La variabile-contatore può essere usata all'interno del ciclo come negli esempi precedenti, ma non può essere modificata: un ciclo *for* al cui interno si modifichi il contatore è sintatticamente corretto, cioè non genera errori di compilazione, ma il comportamento all'esecuzione è "indefinito" e quindi imprevedibile. Non si può, ad esempio, usare un ciclo *for* con un'istruzione di incremento al suo interno per realizzare un passo 2 invece che 1:

```
(* ATTENZIONE: PROGRAMMA SEMANTICAMENTE SCORRETTO! *)
...
ris:= 1;
for i:= 1 to n do begin
  ris:= ris*i;
  i:= i+1 (* l'effetto di questa istruzione è "indefinito" *)
end;
```

Il contatore, essendo una variabile ordinaria, può essere usata anche fuori dal ciclo *for*, per altri scopi (anche se di solito non è una buona pratica); ci si può chiedere allora quale sia il suo valore all'uscita dal ciclo. Sarebbe ragionevole aspettarsi che fosse N+1, dove N è l'estremo superiore; invece secondo la definizione del linguaggio il valore della variabile-contatore del *for* è *indefinito* all'uscita dal ciclo.

Naturalmente, una variabile contiene sempre un valore, ma la definizione del linguaggio non specifica quale, lasciando su ciò libero l'implementatore. Cosí, in talune implementazioni o in taluni casi si potrebbe trovare nel contatore a fine ciclo proprio il previsto valore N+1; il programmatore, però, non deve farvi affidamento. Infatti in altre realizzazioni o in altri casi l'esecutore Pascal potrebbe aver tenuto in un registro durante il ciclo una copia della variabile-contatore, senza aggiornare alla fine la variabile originale, che quindi si troverebbe a contenere ancora il valore iniziale.

In conclusione, un programma che usi il valore del contatore dopo l'uscita dal *for* è sintatticamente corretto ma logicamente scorretto:

```
(* ATTENZIONE: PROGRAMMA SEMANTICAMENTE SCORRETTO! *)
```

```
...
for i:= 1 to n do ...;
j:= i+2;
...
```

È invece perfettamente legittimo, come si è detto, dopo l'uscita dal ciclo assegnare alla variabile-contatore un nuovo valore e utilizzarlo per altri scopi, magari per un altro ciclo *for*! Ad esempio:

```
...
readln(n);
for i:= 1 to n do writeln('ciao');
readln(n);
for i:= 1 to n do writeln('salve');
...
```

Vi è anche una versione "discendente" dell'*istruzione for*, in cui il contatore viene decrementato (invece che incrementato) di 1 ad ogni iterazione, per il resto del tutto analoga alla versione "ascendente":

```
for Variabile:= EsprIniz downto EsprFin do Istruzione;
```

Naturalmente in questo caso se *EsprIniz*   *EsprFin* si hanno *(EsprIniz - Esprfin + 1)* iterazioni, se *Espriniz* < *EsprFin* il corpo non viene eseguito nemmeno una volta.

1.8.2 L'istruzione *while*.

Nell' ultimo esempio, cosí come in quello della somma di una sequenza di *n* numeri e nel precedente, il numero di ripetizioni del corpo del ciclo non è noto al momento della scrittura del programma (cioè, come anche si dice, al *tempo di compilazione*), ma è comunque fissato (durante l'esecuzione) prima di entrare per la prima volta nel ciclo.

L'istruzione *while* permette di scrivere cicli in cui la decisione se ripetere il corpo del ciclo oppure uscire è presa ogni volta in base ad una condizione che può essere di forma arbitraria e che, se vale, può diventare non valida per effetto dell'esecuzione del corpo stesso; il numero delle ripetizioni, pertanto, non è noto nemmeno al momento in cui si entra nel ciclo.

La sua forma generale è la seguente:

```
while Condizione do Istruzione;
```

Esempio: la somma di una sequenza di numeri interi immessa da tastiera "su una sola riga" (cioè prima di pigiare il tasto ENTER).

```
var i,x,somma: integer;
begin
  writeln('immetti la sequenza, terminando con ENTER:');
  somma:= 0;
  while not eoln do begin
    read(x);
    somma:= somma + x
  end;
  writeln('somma = ', somma)
end.
```

Note.

- La primitiva *eoln* (cioè *end of line*) controlla - senza effettuare alcun input - se si è arrivati alla fine della riga.

- Come suggerisce la sintassi, la condizione viene ogni volta controllata prima di eseguire il corpo del ciclo; quindi, se quando viene controllata per la prima volta essa è già subito falsa, il corpo del ciclo non viene eseguito nemmeno una volta; in questo caso, se viene subito pigiato il tasto ENTER senza aver digitato alcun numero, viene quindi visualizzato come risultato il numero 0 (il che può essere accettabile oppure no, dipende dalla specifica esatta del problema).

<inserire altri esempi: vedi lucidi>

1.8.3 Ciclo *while* e non-terminazione.

Un programma Pascal costituito da soli cicli *for*, assegnazioni, e istruzioni condizionali, è un programma la cui esecuzione sicuramente termina.

Con l'introduzione dell'istruzione *while* si introduce la possibilità della non-terminazione; se infatti l'esecuzione del corpo del ciclo, per quanto ripetuta, non rende mai falsa la condizione di controllo, la ripetizione continua per sempre.

Scrivere un ciclo che non termina è molto facile:

```
var i: integer;
begin
  i:= 1;
  while i<>0 do i:= i+2
end;
```

In realtà, se il controllo di *overflow* (traboccamento) è abilitato, il programma si interrompe con un errore quando, a forza di incrementare `i`, il suo valore giunge ad eccedere il massimo *integer* possibile; alcuni linguaggi di programmazione dispongono tuttavia di un tipo di dato "intero" senza limitazioni di grandezza (ovviamente realizzato con spazio di memoria non fissato); se, come in Pascal, tale tipo di dato non è fra quelli primitivi, lo si può realizzare con la programmazione, come vedremo. Con un tale tipo di dato anche il programma seguente non termina, naturalmente.

```
var i: integer;
begin
  i:= 1;
  while i<>0 do i:= i+1
end;
```

In Pascal se il controllo di overflow non è abilitato il programma termina normalmente perchè continuando a incrementare `i` di 1 alla fine si ritorna a 0 (come nel contachilometri); se invece il controllo è abilitato, si ha come prima un arresto con errore. Il programma seguente, in cui il corpo del *while* è costituito dall'istruzione vuota, non termina nè in teoria nè in pratica:

```
var i: integer;
begin
  i:= 1;
  while i<>0 do
end;
```

In generale, un'istruzione *while* può terminare se eseguita a partire da certi stati iniziali e non terminare se eseguita a partire da certi altri. Ad esempio:

```
...
while n<>5 do
  if n<maxint then n:= n+1;
```

l'istruzione termina se la variabile `n` contiene inizialmente un valore 5, altrimenti non termina.

1.8.4 L'istruzione *repeat*.

L'istruzione *repeat* è una variante del *while*, in cui il corpo del ciclo viene sempre eseguito almeno una volta e il test della condizione viene fatto, invece che all'inizio, alla fine dell'esecuzione del corpo, come la sintassi di questa istruzione aiuta a ricordare:

```
repeat SequenzaDiIstruzioni until Condizione
```

dove *SequenzaDiIstruzioni* è una sequenza di *istruzioni* separate da ";" (si noti la diversità rispetto al *while*: il corpo del *repeat* non è necessariamente una sola istruzione, quindi anche quando il corpo è una sequenza di piú istruzioni non è necessario racchiuderle fra un *begin* e un *end*).

All'opposto del *while*, l'uscita dal ciclo si ha quando la condizione viene raggiunta, cioè quando diventa vera (all'opposto del C, dove l'analoga istruzione con test alla fine, la *do-while*, esce quando la condizione è falsa, come il *while*).

Esempio: il programma seguente continua a richiedere caratteri finchè l'utente non immette la stringa "si" oppure "no":

```
var s: string;
begin
  writeln('rispondi si o no');
  repeat
    readln(s)
  until (s = 'si') or (s = 'no');
  ...
end.
```

L'istruzione seguente, in cui si fa uso della primitiva *keypressed* (cioè "tasto premuto") del modulo TurboPascal *crt* (che deve essere segnalato in una *clausola uses* all'inizio del programma) rimane in attesa - continuando ad eseguire il test e virtualmente l'istruzione vuota - finchè l'utente non pigia un qualunque tasto:

```
repeat until keypressed;
```

Essa può essere utilmente inserita nei punti in cui si vuole che il programma si sospenda (ad esempio per permettere di esaminare con calma l'output sullo schermo), e riprenda l'esecuzione solo per iniziativa dell'utente.

<inserire altri esempi>

1.8.5 *For* e *repeat* espressi per mezzo del *while*.

Dei tre costrutti iterativi il *while* è il piú generale; infatti il *for* e il *repeat* possono essere considerati dei casi particolari di *while*, e possono essere espressi per mezzo di esso. L'istruzione composta:

```
for i:= Espr1 to Espr2 do Corpo;
```

è equivalente ad esempio alla sequenza di istruzioni:

```
temp1:= Espr1;
temp2:= Espr2;
i:= temp1;
while i <= temp2 do begin
  Corpo;
  i:= i+1
end;
i:= indefinito;
```

L'istruzione composta:

```
repeat Corpo until Condizione;
```

è equivalente alla sequenza di istruzioni:

```
Corpo;
while not Condizione do begin Corpo end;
```

1.9 I valori booleani

Un tipo di valori particolarmente importante, distinto da quelli numerici e da stringhe e caratteri, è il tipo *boolean*, cioè dei *booleani* o *valori di verità*: i valori di questo tipo sono soltanto due, *false* e *true*, e sono rappresentati all'interno del calcolatore rispettivamente come 0 e 1 (o meglio, di solito 0 rappresenta *false* mentre qualunque numero diverso da 0 rappresenta *true*); per programmare in Pascal, però, la loro rappresentazione interna non ci interessa (a differenza che in C, dove non esistono come tipo primitivo, e quindi anche nel linguaggio si opera direttamente con le loro rappresentazioni).

Si faccia ben attenzione al fatto che *true* e *false* non sono stringhe, ma (nomi di) enti astratti, cosí come sono enti astratti i numeri; i valori *true* e *'true'* sono (i nomi di) due entità ben distinte: il primo è un valore di tipo *boolean*, il secondo è un valore di tipo *string*.

Le operazioni definite sui booleani sono le ordinarie operazioni logiche *and*, *or*, *not*, ecc., definite dalle note tabelle:

false and false = false	**false or false = false**	**not false = true**
false and true = false	**false or true = true**	**not true = false**
true and false = false	**true or false = true**	
true and true = true	**true or true = true**	

È importante convincersi che - in programmazione - un'espressione "relazionale" come 3>5 non è un'asserzione errata, bensí un'espressione analoga a 3+5, che invece di avere come risultato un terzo numero 8, ha come risultato un valore di un altro tipo, precisamente il valore *false* di tipo booleano. Il simbolo relazionale ">" denota

un'operazione, proprio come il simbolo "+"; l'unica differenza è che mentre l'addizione è un'operazione interna sui numeri (cioè che prende due numeri e restituisce un numero), l'operazione di "maggiore" è invece un'operazione esterna (cioè prende due numeri ma restituisce un valore che non è un numero).

Si faccia attenzione a non confondere gli operatori logici di cui sopra con le particelle "e", "o", ecc. dell'italiano o i loro corrispondenti nelle altre lingue naturali, che hanno un uso molto piú ampio. Ad esempio, nelle lingue naturali la congiunzione "e" può essere usata per congiungere due istruzioni, magari indicando l'ordine in cui devono essere eseguite, come nella frase "Si aggiunga un cucchiaio di farina e si mescoli"; in Pascal, invece, per "congiungere" nel senso precedente due istruzioni, non si può usare l'*and*, bisogna usare il punto e virgola: `aggiungiFarina; mescola`.

In Pascal Standard i valori booleani, a differenza dei valori numerici, dei caratteri e delle stringhe, non possono essere argomenti di operazioni di input/output, cioè non possono essere nè direttamente immessi da tastiera attraverso una *read*, nè direttamente visualizzati sullo schermo con una *write*, ma soltanto essere originati ed utilizzati all'interno di esecuzioni di programmi. La ragione di tale scelta è che ben difficilmente in un programma si ha la necessità di richiedere direttamente l'immissione o la scrittura di un booleano senz'altra spiegazione, e in ogni caso si può sempre passare attraverso l'input/output di una stringa, ad esempio:

```
var mostrasommeparziali: boolean;
    s: string;
    n, som: integer;
begin
  writeln('immetti true o false');
  readln(s);
  if s = 'true' then mostrasommeparziali:= true
  else if s = 'false' then mostrasommeparziali:= false
  else begin writeln('errore di input'); halt end;
  writeln('immetti sequenza di interi terminata da CTRL/Z');
  som:= 0;
  while not eof do begin
    if mostrasommeparziali then writeln(som);
    readln(n);
    som:= som+n;
  end;
  writeln(som)
end.
```

Dovendo passare attraverso una stringa, però, tanto vale programmare un dialogo piú intuitivo:

```
...
begin
  writeln('vuoi vedere le somme parziali?');
  repeat
    write('rispondi si o no: ');
    readln(s)
  until (s = 'si') or (s = 'no');
  mostrasommeparziali:= s = 'si';
...
```

In TurboPascal, si può fare l'output di un booleano ma non l'input; ad esempio l'istruzione:

```
writeln('3>5 is', 3>5)
```

visualizza sullo schermo la scritta *`3>5 is FALSE`*.

1.10 Istruzioni condizionali e iterative: una semantica piú precisa.

Avendo introdotto i booleani, l'esecuzione delle istruzioni condizionali e iterative può essere descritta con piú precisione, nel modo seguente.

Istruzione *if then else*:

1) l'espressione booleana costituente la condizione viene valutata, cioè ridotta ad un valore, che sarà *true* o *false*.
2.1) se il valore ottenuto è *true*, viene eseguito il ramo *then*, e poi si prosegue con la prima istruzione successiva all'istruzione condizionale (cioè successiva alla fine del ramo *else*).
2.2) se il valore ottenuto è *false*, viene eseguito il ramo *else*, e poi si prosegue con l'istruzione successiva.

Istruzione *while*:

1) l'espressione booleana costituente la condizione viene valutata, cioè ridotta ad un valore, che sarà *true* o *false*.
2.1) se il valore ottenuto è *false*, non si esegue il corpo del ciclo, l'istruzione *while* è terminata e si passa all'istruzione successiva.
2.2) se il valore ottenuto è *true*, si esegue il corpo del ciclo, poi si ritorna al punto 1.

Analoghe descrizioni si danno per gli altri costrutti: *if-then*, *case*, *for*, e *repeat*.

L'espressione booleana, benchè spesso sia una semplice operazione di confronto (come $i<n$), può essere un'espressione booleana complicata quanto si vuole, o contenere invocazioni di funzioni (vedi *Capitolo 2*) il cui calcolo può richiedere "molto tempo" (o peggio non terminare ...).

È importante non dimenticare che la condizione del *while* viene in ogni caso controllata (o meglio, il valore del'espressione booleana viene calcolato) soltanto in certi istanti ben precisi, cioè prima di ogni iterazione: se durante l'esecuzione delle istruzioni del corpo la condizione diventa falsa, il corpo del ciclo viene comunque eseguito per intero.

Ricordiamo, per finire, che alcuni linguaggi di programmazione del passato avevano un costrutto (variamente denominato, ad es. *loop*) per realizzare un "ciclo perpetuo", ossia la ripetizione indefinita di una sequenza di istruzioni. Un costrutto di tal genere è del tutto superfluo, giacchè basta scrivere:

```
while true do begin
  istruzioni
end;
```

Una qualunque espressione booleana equivalente a *true* andrebbe bene, come `3=3`, o `'ciao'='ciao'`, oppure `n=n`, dove `n` sia una variabile.

Esempi di programmi che in linea di principio non terminano sono i programmi che gestiscono sistemi o risorse in tempo reale, come un sistema operativo, o il software che gestisce una centrale telefonica o una centrale nucleare, ecc.

1.11 Valutazione *pigra* delle espressioni booleane.

Una caratteristica molto utile del TurboPascal, che si trova ormai in tutti i linguaggi moderni, è la valutazione cosiddetta ottimizzata - o "cortocircuitata", o "pigra" (*lazy*) - delle espressioni booleane: per valutare un'espressione della forma *A and B* oppure della forma *A or B*, l'esecutore TurboPascal valuta prima l'espressione *A*, e valuta poi l'espressione *B* solo se necessario: cioè, per valutare *A and B* valuta *B* solo se il risultato della valutazione di *A* è *true* (perchè se *A* vale *false* è inutile valutare *B*, il valore di *A and B* sarà comunque *false*); simmetricamente per valutare *A or B* valuta *B* solo se il risultato della valutazione di *A* è *false* (perchè altrimenti *A or B* vale comunque *true*).

Pertanto, in TurboPascal, la valutazione di un'espressione booleana anche complessa costruita con gli operatori *and* e *or* procede rigorosamente da sinistra a destra, e si arresta non appena il risultato dell'intera espressione risulta determinato. Si possono in tal modo scrivere programmi efficienti senza sacrificare la semplicità e l'eleganza. Ad esempio un ciclo della forma:

```
while (n <> 0) and (m div n <> 1) do Corpo end;
```

è perfettamente corretto in TurboPascal ma logicamente (o semanticamente) errato nello Standard, benchè sintatticamente corretto in entrambi.

In TurboPascal, infatti, se *n* vale 0 non si valuta la seconda componente (perchè si sa che in tal caso la congiunzione vale comunque *false*) ma si esce subito dal ciclo; in Pascal Standard, invece, si va anche in questo caso a valutare la seconda componente, generando cosí un errore al tempo di esecuzione (tentativo di divisione per zero).

Per scrivere un ciclo corretto equivalente si è in generale costretti, nello Standard, ad introdurre una variabile booleana in modo da poter "disaccoppiare" i due test, peggiorando cosí notevolmente la semplicità e chiarezza del programma, perchè l'uscita dal ciclo dovuta alla falsità della seconda espressione viene innaturalmente rimandata all'iterazione successiva:

```
var cond2: boolean;
...
begin
...
  cond2:= true;
  while (n<>0) and cond2 do begin
    cond2:= m div n <> 1;
    if cond2 then
      Corpo
  end;
```

Come vedremo, situazioni di questo genere sono molto frequenti nella scansione di vettori e liste, in cui la regola di valutazione del TurboPascal è quindi molto utile.

Si osservi che adottando la suddetta valutazione ottimizzata si perde la commutatività degli operatori logici *and* e *or*; in TurboPascal, infatti, non è piú vero come in logica che *A and B* è perfettamente equivalente a *B and A*: ad esempio l'istruzione che si ottiene dal precedente ciclo *while* scambiando gli operandi della congiunzione, cioè:

```
while (m div n <> 1) and (n <> 0) do...
```

è evidentemente scorretta logicamente, e può generare un errore a tempo di esecuzione.

Per permettere, a chi lo desideri, di mantenere la compatibilità con lo Standard, il TurboPascal lascia comunque all'utente la possibilità di scelta fra il modello "cortocircuitato" di valutazione e quello standard; tale scelta è operabile attraverso un'opportuno comando al compilatore (si veda il manuale).

1.12 La primitiva *break*.

Come abbiamo visto, dai cicli *for*, *while* e *repeat* è possibile uscire solo in un ben preciso punto, cioè all'inizio (nel *for* e nel *while*) o alla fine (nel *repeat*) dell'esecuzione del corpo.

L'istruzione *break*, cioè *interrompi*, che non esiste in Pascal Standard, nè nelle versioni precedenti del TurboPascal, ma è presente in C e nei suoi discendenti, e nella versione 7 del TurboPascal (a rigore non come istruzione, ma come procedura primitiva del sistema), se eseguita all'interno di un ciclo *for*, *while*, o *repeat*, termina immediatamente il ciclo, cioè fa immediatamente "saltare" alla prima istruzione successiva alla fine del ciclo stesso. La scrittura di un'istruzione *break* non contenuta in alcun ciclo genera un errore di compilazione.

Nota Bene: se l'istruzione *break* si trova in un ciclo annidato in un altro ciclo o in altri cicli, la sua esecuzione fa uscire solo dal ciclo piú interno.

La *break* permette di scrivere cicli con uscita in un punto qualunque e/o con uscite multiple, il che qualche volta può risultare piú naturale. Il principiante è però spesso portato ad usare la *break* in situazioni dove sarebbe perfettamente appropriato un ciclo ordinario. Si ricorra dunque a tale istruzione solo dopo aver tentato le altre soluzioni.

Supponiamo di voler scrivere un ciclo che calcoli il prodotto di una sequenza di interi immessa da tastiera, la cui fine sia segnalata (per qualche ragione che non ci interessa) da 0, che naturalmente non dev'essere moltiplicato per il risultato parziale; si supponga inoltre che la lettura di ogni elemento sia preceduta da un *prompt* (scritta di suggerimento). In Pascal Standard dobbiamo fare una scrittura e lettura iniziale fuori dal ciclo per poter controllare il primo numero:

```
prod:= 1;
write('immetti un intero <> 0 ');
write('oppure zero per terminare: ');
readln(n);
while (n<>0) do begin
  prod:= prod*n;
  write('immetti un intero <> 0 ');
  write('oppure zero per terminare: ');
  readln(n)
end;
```

Usando un *repeat* le cose non migliorano, anzi: per poter eseguire il test sul primo numero si è poi costretti a rifarlo sempre due volte su tutti gli elementi successivi:

```
prod:= 1;
repeat
  write('immetti un intero <> 0 ');
  write('oppure zero per terminare: ');
  readln(n);
```

```
    if n<>0 then prod:= prod*n
  until n=0;
```

oppure a mimare il *while* con il *repeat* ottenendo comunque un programma piú brutto:

```
prod:= 1;
write('immetti un intero <> 0 ');
write('oppure zero per terminare: ');
readln(n);
if n<>0 then
  repeat
    prod:= prod*n;
    write('immetti un intero <> 0 ');
    write('oppure zero per terminare: ');
    readln(n)
  until n=0;
```

Utilizzando il *break* si può invece evitare sia la lettura iniziale che il doppio test:

```
prod:= 1;
while true do begin
    write('immetti un intero <> 0 ');
    write('oppure zero per terminare: ');
    readln(n);
  if n=0 then break;
    prod:= prod*n
end;
```

Il compilatore traduce verosimilmente l'istruzione *while true do* per mezzo di un'istruzione di salto incondizionato (cioè che non va ogni volta a rivalutare la costante *true*); quindi ad ogni iterazione viene effettuato un solo test (il controllo se n è uguale a zero). Si ha cosí un ciclo con una sola uscita, che non è nè all'inizio nè alla fine del corpo, ma nel punto "piú naturale".

Lo svantaggio è che ora per capire quando e perchè si esce dal ciclo bisogna esaminarne tutto il corpo alla ricerca dei *break*; si può cercare di rimediare con opportuni indentamenti, come nell'esempio di sopra.

Supponiamo ora di voler scrivere un programma che cicli chiedendo il nome dell'utente e rispondendo con un saluto; il programma deve terminare quando viene immessa la stringa "fine", oppure dopo aver salutato un utente che si sia presentato col nome "Ultimo". Esso è facilmente realizzabile con un ciclo a due uscite:

```
var s: string;
begin
...
  repeat
      write('Come ti chiami? ');
      readln(s);
    if s='fine' then break;
      writeln('Ciao, ',s)
  until s='Ultimo"
end;
```

Una versione con un ciclo *while* ordinario è:

```
write('Come ti chiami? ');
readln(s);
while (s<>'fine') and (s<>'Ultimo') do begin
  writeln('Ciao, ',s);
```

```
    write('Come ti chiami? ')
    readln(s)
  end;
  if s='Ultimo' then writeln('Ciao, ',s);
```

Dal punto di vista logico, la presenza delle interruzioni di ciclo rende piú complesse le dimostrazioni di correttezza e la progettazione di programmi fondata sulla correttezza; in queste dispense l'uso del *break* sarà perciò riservato a pochissimi casi in cui il significato è evidente, ed accuratamente evitato in tutti gli altri.

1.13 Riepilogo: la nozione di stato.

Come abbiamo visto in questo primo capitolo, l'esecuzione di un programma Pascal può essere descritta come una sequenza di cambiamenti di stato di una virtuale "macchina Pascal". Lo stato della macchina ad un dato istante è costituito dai contenuti delle variabili del programma e dal "punto di esecuzione", cioè da "quale sia la prossima istruzione da eseguire"; un immaginario film del processo di esecuzione di un programma Pascal sarà quindi costituito da una sequenza di fotogrammi ciascuno dei quali non solo mostra i contenuti delle variabili ma anche indica la prossima istruzione da eseguire.

Tale sequenza di fotogrammi può essere visualizzata, nel sistema TurboPascal, mediante l'esecuzione passo-passo dopo aver aperto un *watch* - cioè uno specie di "sportello di osservazione" - per ogni variabile: si vedrà infatti in tal modo via via evidenziarsi l'istruzione in esecuzione, e corrispondentemente cambiare i contenuti delle variabili.

1.14 Stile di programmazione e uso delle variabili.

Fra le capacità che devono essere acquisite da chi inizia a programmare vi è quella di evitare di scrivere programmi e procedure complicati e prolissi per risolvere problemi che invece ammettono soluzioni semplici e concise; procedure inutilmente complicate sono infatti difficili da capire e spesso errate. Nello stesso tempo, bisogna imparare a trovare, per ogni problema, la soluzione piú efficiente, anche se questa non coincide con la soluzione "ingenua", cioè quella che di solito si presenta come la piú naturale.

Un primo duplice principio da seguire è quello di, da una parte, non usare piú variabili del necessario, dall'altra memorizzare i risultati di computazioni intermedie (anzichè rifare le computazioni stesse) quando tali risultati siano riutilizzati o riutilizzabili.

Come esempio banale della prima parte del principio precedente possiamo citare l'inutilità delle variabili per i risultati intermedi di espressioni (nei linguaggi cosiddetti "di alto livello" come il Pascal): se dobbiamo scrivere un pezzo di programma che calcola il rapporto fra la somma e la differenza di due variabili `m` ed `n` e lo memorizza in una variabile `r`, è perfettamente inutile in Pascal introdurre due variabili `somma` e `diff` e scrivere:

```
  somma:= m+n;
  diff:= m-n;
  r:= somma/diff;
```

si può scrivere direttamente

```
r:= (m+n)/(m-n);
```

Analogamente, se dobbiamo scrivere un pezzo di programma che visualizza il rapporto fra la somma e la differenza di due variabili, è inutile assegnare tale rapporto ad una variabile e poi farne l'output:

```
r:= (m+n)/(m-n);
writeln(r);
```

si può scrivere direttamente:

```
writeln((m+n)/(m-n));
```

L'indicazione precedente non è assoluta: se si deve calcolare un'espressione complicata, può essere conveniente, per evitare errori e rendere l'espressione piú comprensibile, calcolarne separatamente delle sottoespressioni assegnandole a variabili con nomi significativi, che vengono poi combinate nel calcolo dell'espressione globale.

Se un'espressione compare invece in piú punti del programma, è sempre opportuno memorizzarla in una variabile piuttosto che ricalcolarla tutte le volte. Non scriveremo quindi:

```
s:= (m+n) + 5*(m+n)*(m+n);
t:= (m+n)/(m-n);
...
u:= 3*(m+n);
...
```

bensí:

```
som:= m+n;
s:= som + 5*som*som;
t:= som/(m-n);
...
u:= 3*som;
...
```

Come esempio leggermente piú sottile consideriamo un ciclo della forma:

```
while (i < m-n+1) and (ris < max) do ris:= ris*i;
```

come si vede, i valori di `m` e di `n` non vengono alterati nel corpo del ciclo, e quindi il valore dell'espressione `m-n+1` rimane costante in tutte le iterazioni; eppure nel programma cosí com'è scritto l'espressione `m-n+1` viene ricalcolata inutilmente ad ogni iterazione. Ciò può essere evitato memorizzando tale valore in una variabile:

```
sup:= m-n+1;
while (i < sup) and (ris < max) do ris:= ris*i;
```

Si noti che se invece di un ciclo *while* abbiamo un ciclo *for*, ad esempio:

```
for i:= 1 to m-n+1 do ...
```

poichè per la semantica del *for* gli estremi inferiore e superiore vengono dall'esecutore Pascal <u>valutati una sola volta</u>, è - contrariamente a prima - perfettamente inutile introdurre una variabile per memorizzare l'espressione `m-n+1`.

1.15 Esercizi.

Esercizio 2. Scrivere un programma che legga da tastiera una sequenza di n interi, con n specificato dall'utente, e ne scriva sullo schermo la somma (senza usare vettori, in questo come in tutti gli esercizi qui di seguito).

Esercizio 3. Scrivere un programma che legga da tastiera una sequenza di interi terminata dal numero 0 e ne scriva sullo schermo la somma.

Esercizio 4. Modificare il programma precedente in modo che legga da tastiera una sequenza di naturali terminata da qualunque numero negativo e ne scriva sullo schermo la somma (non sommando il "numero-tappo").

Esercizio 5. Scrivere un programma che legga da tastiera una sequenza di interi terminata da CTRL/Z e ne scriva sullo schermo la somma.

Esercizio 6. Scrivere un programma che legga da tastiera una sequenza di n interi, con n specificato dall'utente, e ne scriva sullo schermo il massimo.

Esercizio 7. Scrivere un programma che legga da tastiera una sequenza di interi terminata da CTRL/Z e ne scriva sullo schermo il massimo.

Esercizio 8. Modificare il programma precedente in modo che scriva sullo schermo il massimo ed il minimo.

Esercizio 9. Scrivere un programma che legga da tastiera una sequenza di n interi, con n specificato dall'utente, e ne scriva sullo schermo la media aritmetica con due cifre decimali.

Esercizio 10. Scrivere un programma che legga da tastiera una sequenza di interi terminata da CTRL/Z e ne scriva sullo schermo la media aritmetica con due cifre decimali, il massimo, e il minimo.

Esercizio 11. Scrivere un programma che legga da tastiera due numeri naturali m ed n (con m   n) e scriva sullo schermo la somma dei naturali da m ad n compresi. Qual è il risultato più "naturale" quando $m > n$?

Esercizio 12. Scrivere un programma che legga da tastiera una sequenza di n numeri (con n specificato dall'utente) $a_1, a_2, ..., a_n$ e scriva sullo schermo il risultato della somma

$$a_1 + 2a_2 + 3a_3 + ... + na_n.$$

Capitolo 2
Procedure e funzioni.

2.1 Introduzione.

In Pascal un *sottoprogramma* è un "pezzo" di programma che ha la stessa struttura del programma principale, cioè è costituito da un *blocco*: una *sezione delle dichiarazioni* seguita da una *sezione delle istruzioni*. Esso può essere "richiamato" o "invocato" in punti diversi del programma principale in cui si richieda uno stesso tipo di calcolo, evitando cosí ripetizioni di una stessa sequenza di istruzioni (non solo nel sorgente, ma anche nel programma compilato), e realizzando una migliore strutturazione e comprensibilità del programma stesso. Piú in generale, quando un programma è "grosso" e complicato, è sempre opportuno strutturarlo in sottoprogrammi, indipendentemente dal fatto che essi siano richiamati molte volte o una sola. Dal punto di vista della progettazione, ciò equivale a risolvere separatamente ognuno dei sottoproblemi in cui si scompone il problema; cosí come, invece di scrivere in modo indifferenziato una dimostrazione lunga e complicata di un teorema, si dimostra prima un certo numero di lemmi o proposizioni preliminari, e poi nel corpo della dimostrazione del teorema principale semplicemente si invocano (notare l'uso dello stesso verbo) i "sottoteoremi" dimostrati prima.

Naturalmente, per poter "chiamare" qualcuno bisogna che questi abbia un nome: un sottoprogramma deve quindi obbligatoriamente avere un nome; esso può avere inoltre dei *parametri*, necessari per permettere uno scambio di informazioni fra programma chiamante e sottoprogramma chiamato; vi sono due generi di parametri, *per valore* e *per riferimento*; vi sono inoltre due generi di sottoprogrammi, le *procedure* e le *funzioni*. Nelle sezioni seguenti introduciamo gradualmente le varie nozioni per mezzo di esempi.

2.2 Procedura senza parametri e senza variabili.

Supponiamo di voler tenere in un unico programma tutti gli esempietti di programmazione scritti finora, ma facendo in modo che durante l'esecuzione il passaggio ad ogni successivo esempio avvenga per iniziativa dell'utente.

Consideriamo allora il seguente programma:

```
        var max,n,i, n_es: integer;
             x, ris: integer;

        procedure aspettaconferma;
        begin
          writeln('Per procedere pigia ENTER');
          readln;
          n_es:= n_es + 1;
          writeln('********** Esempio n. ',n_es,' **********')
        end;

        begin {corpo del programma principale}
          n_es:= 0;

          aspettaconferma;
  {alfa:} writeln('**** N-esima potenza di X ****')
```

```
         write('immetti la base (reale): ');
         readln(x);
         write('immetti l'esponente, intero non negativo: ');
         readln(n);
         ris:= 1;
         for i:= 1 to n do ris:= x*ris;
         writeln(ris:4:2);  (* scrive due cifre dopo la virg. *)

         aspettaconferma;
{beta:}  writeln('**** massimo di una sequenza ****');
         write('immetti una sequenza di interi');
         writeln('di almeno un elemento: ');
         read(max);
         while ...
         ...
         aspettaconferma;
{gamma:}writeln('**** fattoriale ****');
  ...
  ...
      end.
```

Nella parte dichiarativa del programma vi è la *dichiarazione* (o *definizione*) del sottoprogramma `aspettaconferma`; piú precisamente, di una procedura senza parametri (e priva di variabili locali) che effettua un'elementare interazione con l'utente incrementando inoltre una variabile globale. L'elaborazione della dichiarazione da parte del virtuale esecutore Pascal non consiste nell'esecuzione del corpo della procedura, ma soltanto nel "renderla richiamabile" nel resto del programma.

Si noti che, con l'introduzione dei sottoprogrammi, la parte dichiarativa di un programma viene a contenere delle sezioni di istruzioni (appunto quelle dei sottoprogrammi).

Si osservi inoltre che dall'interno del sottoprogramma è visibile (e modificabile) la variabile globale `n_es`.

Nel corpo del programma principale la procedura viene *richiamata* o *invocata* semplicemente scrivendone il nome. Una *chiamata di procedura* (*procedure call*) è un'istruzione, al pari di un'assegnazione o di un'istruzione iterativa; essa può quindi essere inserita nel programma in qualunque posizione in cui si possa inserire un'istruzione, con le stesse regole sintattiche (punto e virgola per separarla da eventuali istruzioni precedenti o seguenti, ecc.).

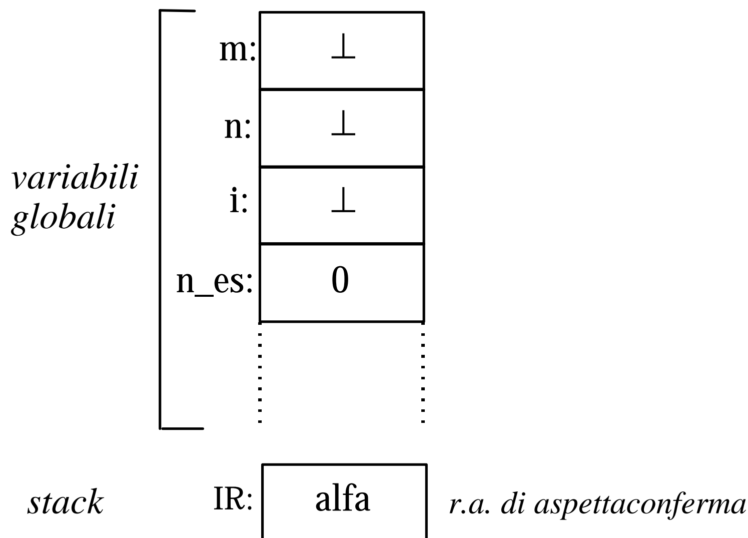
L'esecuzione di una chiamata della procedura `aspettaconferma` da parte dell'esecutore Pascal consiste nelle seguenti azioni:

1) Viene creato, in una particolare area della memoria di lavoro detta *stack* (pila), un insieme di celle di memoria detto *record di attivazione* (abbreviato come r. di a.) della procedura; esso è costituito, nel caso della procedura `aspettaconferma`, da una sola cella detta *indirizzo di ritorno*, o *istruzione di ritorno* (IR); in essa viene memorizzato l'indirizzo della cella di memoria in cui si trova l'istruzione immediatamente successiva alla chiamata di procedura.
   Invece di parlare di "indirizzi", possiamo piú astrattamente pensare che ogni istruzione abbia un nome o etichetta, e affermare che nel r. di a. viene memorizzato tale nome; naturalmente, si tratta di formulazioni perfettamente equivalenti. Per poter rappresentare in modo conveniente il record di attivazione con un disegno, nel programma abbiamo - come si vede - appunto etichettato con nomi di lettere greche le istruzioni che ci interessano.

2) L'esecuzione salta alla prima istruzione del sottoprogramma, e continua poi, con le stesse regole dei programmi principali, fino a raggiungere l'*end* del sottoprogramma; a questo punto l'esecuzione salta all'indirizzo di ritorno precedentemente memorizzato, e il record di attivazione viene "distrutto" (si "ritorna" dalla procedura al programma principale).

Lo stato della memoria-dati dell'esecutore Pascal all'istante dell'esecuzione del *begin* della prima chiamata della procedura *aspettaconferma* è rappresentato nel disegno seguente.

In esso, e in tutti i successivi, il simbolo ⊥ indica che il contenuto di una variabile è indefinito (come ad esempio prima dell'inizializzazione).



L'indirizzo di ritorno sarà in generale diverso per ognuna delle diverse attivazioni della stessa procedura; ma naturalmente, se una stessa istruzione di chiamata di procedura viene eseguita piú volte nel corso del programma (perchè ad esempio si trova dentro un ciclo eseguito piú volte), essa darà origine a dei r. di a. successivi tutti con lo stesso indirizzo di ritorno.

2.3 Procedura con variabili locali.

Modifichiamo il programma precedente in modo da avere la possibilità di interrompere il programma ed evitare l'esecuzione di tutti gli esempi da quel punto in avanti:

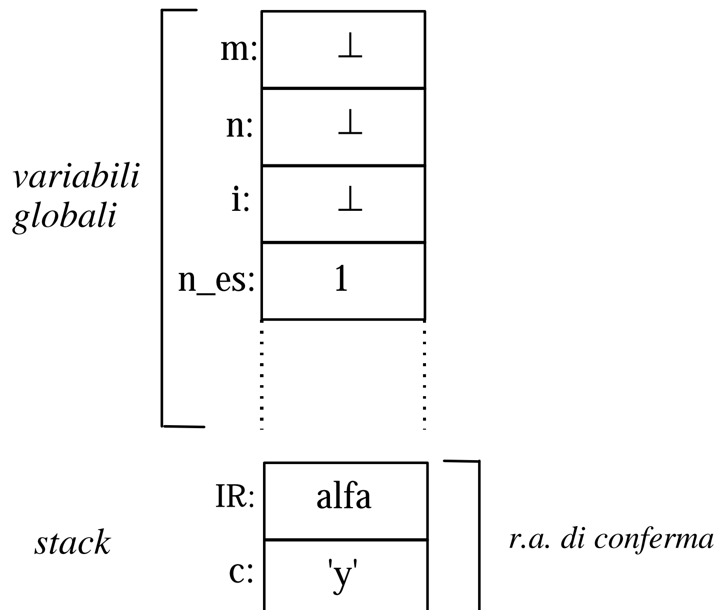
```
uses crt; (* per poter usare la funz. predef. readkey *)
var max,n,i, n_es: integer;
    x, ris: integer;
...

procedure conferma;
var c: char;
begin
  writeln('Per uscire dal programma pigia x');
  writeln('Per continuare pigia qualunque altro tasto');
  c:= readkey;  (* restituisce il carattere immesso *)
  if c = 'x' then halt;
  n_es:= n_es + 1;
  writeln('********** Esempio n. ',n_es,' **********')
```

```
        end;


        begin {corpo del programma principale}
          n_es:= 0;

          conferma;
  {alfa:} writeln('**** N-esima potenza di X ****')
             ...
          conferma;
  {beta:} writeln('**** massimo di una sequenza ****');
             ...
          conferma;
  {gamma:}writeln('**** fattoriale ****');
    ...
        end.
```

La definizione della procedura `conferma` contiene la dichiarazione della variabile `c`: le variabili dichiarate all'interno di un sottoprogramma si dicono *variabili locali* del sottoprogramma; esse vengono create ad ogni attivazione del sottoprogramma, e distrutte ad ogni uscita dal sottoprogramma.

Piú precisamente, il record di attivazione che viene creato ad ogni esecuzione di una chiamata di procedura contiene una cella corrispondente ad ognuna delle variabili locali. Il r. di a. per la procedura `conferma` è quindi costituito da una cella IR e da una cella `c`. Ogni chiamata successiva di `conferma` genera una diversa "incarnazione" della variabile `c`, che "muore" ad ogni ritorno al programma principale; fra un'incarnazione e l'altra, cioè nel tempo intercorrente tra la fine di un'attivazione e l'inizio dell'attivazione successiva, la variabile non esiste, e quindi non può "ricordare" il valore che aveva nella vita precedente.

Nella figura sottostante è rappresentato lo stato (della memoria-dati) dell'esecutore Pascal all'istante immediatamente precedente l'esecuzione dell'*end* della procedura, supponendo che l'utente abbia pigiato il tasto *'y'*.



2.4 Allocazione dinamica e allocazione statica delle aree-dati locali.

Ci si può chiedere per qual motivo la semantica del Pascal richieda che la memoria di lavoro di un sottoprogramma - cioè le variabili locali, la cella per l'indirizzo di ritorno, e inoltre, come vedremo, i parametri - sia creata ad ogni chiamata e distrutta ad ogni ritorno al chiamante, invece di essere sempre esistente come le variabili globali.

In effetti, se ci si limita alle tecniche di programmazione esaminate fin qui, l'unica ragione che potrebbe giustificare la gestione della memoria presentata sopra è quella del risparmio di spazio; infatti, occupando di volta in volta solo la memoria necessaria per i sottoprogrammi attivi in quel momento, si occupa in generale meno spazio che se a tutte le procedure definite nel programma fosse riservata, dall'inizio alla fine dell'esecuzione del programma, la memoria corrispondente alle loro aree-dati locali. Tuttavia, a tale vantaggio in termini di spazio fa riscontro il maggior tempo necessario per creare (cioè *allocare*, rendere disponibile al programma) e distruggere (cioè *deallocare*, liberare) il record di attivazione ad ogni chiamata.

La vera ragione per la gestione della memoria per mezzo dei record di attivazione è, come vedremo, l'ammissibilità in Pascal, come in tutti i linguaggi moderni, delle chiamate cosiddette *ricorsive* di sottoprogramma. In assenza di tale caratteristica, l'allocazione delle aree locali può essere statica, come in Fortran o in Cobol; invece dei record di attivazione che nascono e muoiono, si hanno allora semplicemente, per tutta la durata dell'esecuzione del programma, accanto all'area delle variabili globali tante aree locali quanti sono i sottoprogrammi definiti nel programma. Insomma si avrebbe uno statico *record di sottoprogramma* per ogni definizione di sottoprogramma, invece di un dinamico *record di attivazione* per ogni chiamata di sottoprogramma. La forma e il contenuto di ogni singolo record sarebbe invariato, cosí come i meccanismi di passaggio-parametri che vedremo nelle prossime sezioni.

Le rappresentazioni grafiche precedenti, cosí come quelle che seguono, sono quindi valide anche per i linguaggi ad allocazione completamente statica, con la sola avvertenza che in tal caso i record relativi alle varie procedure devono essere considerati sempre tutti presenti in memoria.

2.5 Parametri per valore.

Si consideri il seguente programma:

```
var a,b,c: integer; r: real;

  procedure scrivisomdif(m,n: integer);
  begin
    writeln('somma: ', m+n);
    writeln('differenza: ', m-n);
  end;

  procedure scriviesponenziale(x: real; n: integer);
  var i, ris: integer;
  begin
    ris:= 1;
    for i:= 1 to n do ris:= x*ris;
    writeln(x:4:2,' elevato a ',n,' = ', ris:4:2);
  end;

begin

  scrivisomdif(a,b); {alfa:} ...
  ...
```

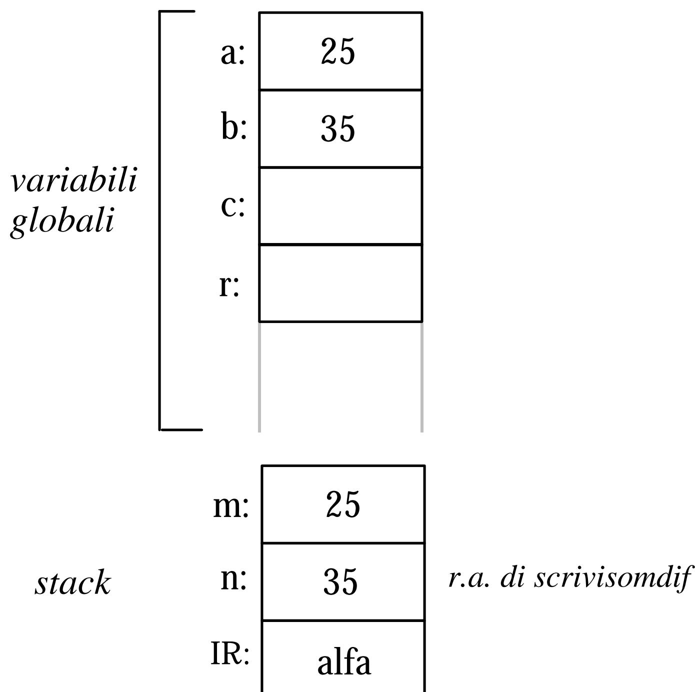
```
    scrivisomdif(b,c); {beta:} ...
    ...
    scrivisomdif(a+10,a+b*c); {gamma:} ...
    ...
    scriviesponenziale(2,10); {delta:} ...
          ...
    scriviesponenziale(r,c);
  {eta:}
  end.
```

Nella definizione della procedura `scrivisomdif` il nome del sottoprogramma è seguito da due identificatori (nomi) di *parametri formali*, `m` ed `n`, qualificati di tipo *integer*. Analoga è la definizione dell'altra procedura.

Parametri di questo genere vengono detti *parametri passati per valore* (o semplicemente *parametri-valore*); essi sono equivalenti a variabili locali che al momento della chiamata vengono automaticamente inizializzate con i valori forniti nella chiamata stessa. Per invocare una procedura definita con parametri occorre infatti passarle gli *argomenti*, detti anche *parametri effettivi*: per ognuno dei parametri formali che compaiono nell'intestazione della procedura, bisogna fornire (nello stesso ordine) un valore dello stesso tipo che qualifica il parametro formale; tale valore può essere una semplice costante, oppure il nome di una variabile (denotante, come al solito, il suo contenuto), oppure un'espressione complicata quanto si vuole.

Il record di attivazione creato a ciascuna chiamata di una procedura contiene, oltre a quanto già visto, una cella per ogni parametro formale.

Nel nostro esempio il r. di a. di `scrivisomdif` contiene quindi due celle `m` ed `n` di tipo *integer*. All'istante della prima chiamata, nella cella `m` viene copiato il contenuto della variabile `a` e nella cella `n` il contenuto della variabile `b`; supponendo che `a` e `b` contengano rispettivamente 25 e 35, lo stato della memoria dell'esecutore Pascal all'istante dell'esecuzione del *begin* della prima chiamata di `scrivisomdif` è quindi rappresentato dal disegno sottostante.



All'uscita dalla procedura il r. di a. viene distrutto, e quindi anche i parametri formali e i loro contenuti.

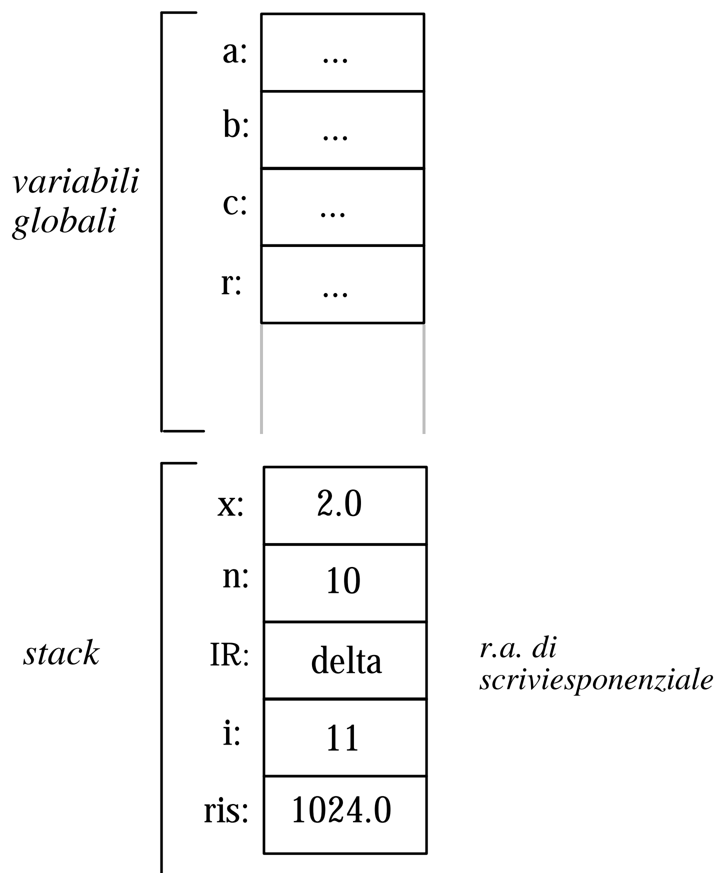
All'istante della seconda chiamata viene ricreato un r. di a. con due celle `m` ed `n`, in cui questa volta vengono copiati i contenuti di `b` e `c`. E cosí via.

Più in generale, al momento della chiamata (ma, come vedremo, prima di creare il r. di a. corrispondente) l'espressione costituente ciascun parametro effettivo viene "valutata", cioè "ridotta" ad un valore, e tale valore viene memorizzato nella cella-parametro-formale corrispondente.

Se il Pascal fosse un linguaggio ad allocazione puramente statica la descrizione precedente sarebbe ugualmente valida, con la sola differenza che lo spazio per il record di `scrivisomdif` sarebbe sempre impegnato, anche negl'intervalli di tempo fra due successive chiamate. Anche in tal caso sarebbe tuttavia ragionevole assumere che le variabili locali e i parametri della procedura non siano accessibili (benchè materialmente esistenti) al di fuori del corpo della procedura; se oltre a ciò si stabilisse che, come il contatore del *for* all'uscita dal ciclo, parametri e variabili locali avessero per definizione valore *indefinito* all'uscita (o all'ingresso) di ogni attivazione, il modello ad allocazione statica sarebbe fin qui indistinguibile, dal punto di vista del programmatore, da quello ad allocazione dinamica.

In generale una procedura ha sia parametri formali che variabili locali; per distinguere gli uni dalle altre nella rappresentazione del r. di a., metteremo sempre nell'ordine dall'alto verso il basso prima i parametri, poi l'IR, per ultime le variabili locali.

All'istante immediatamente precedente all'uscita dal ciclo *for* nella prima chiamata della procedura `scriviesponenziale` lo stato della memoria è:



All'interno di un sottoprogramma i parametri formali possono essere usati esattamente come se fossero delle variabili locali, cioè modificati, passati come argomenti a chiamate di altre procedure, ecc. Ad esempio, la versione seguente della procedura `scriviesponenziale` usa direttamente il parametro `n` come contatore del ciclo:

```
procedure scriviesponenziale(x: real; n: integer);
var ris: integer;
begin
  write(x:4:2,' elevato a ',n,' = ');
  ris:= 1;
  while n >= 1 do begin
    ris:= x*ris;
    n:= n-1
  end;
  writeln(ris:4:2);
end;
```

Se il parametro effettivo passato come esponente è (il contenuto di) una variabile, tale variabile ovviamente non viene modificata dalla procedura; ad esempio, assumendo che `espo` sia una variabile, la chiamata `scriviesponenziale(2, espo)` non modifica `espo`.

Esercizio 1. Un altro semplice esempio di sottoprogramma con parametri e variabili locali è la seguente procedura che calcola e visualizza il prodotto e la somma dei naturali compresi fra due naturali dati M ed N (inclusi).

```
procedure scrivisp(m,n: integer);
var i, somma, prod: integer;
begin
  somma:= 0; prod:= 1;
  for i:= m to n do begin
    somma:= somma+i; prod:= prod*i;
  end;
  writeln(somma);
  writeln(prod)
end;
```

Il lettore disegni il corrispondente record di attivazione in vari istanti di un'immaginaria chiamata della procedura.

2.6 Parametri per riferimento.

Supponiamo di voler modificare le procedure definite nella sezione precedente in modo che i loro risultati, invece di essere scritti sul video, vengano restituiti - cioè resi disponibili - al programma principale.

L'unico meccanismo, fra quelli visti finora, che permetta la comunicazione dei risultati da un sottoprogramma al programma chiamante, è l'uso di variabili globali all'interno della procedura.

L'uso delle variabili globali, però, costringe il programma principale a riservare delle variabili a tale scopo, lega indissolubilmente quella procedura a quel programma, ed è quindi - salvo casi particolari - contrario alle buone regole di scrittura del software:

```
(* ATTENZIONE: ESEMPIO DI PESSIMO STILE DA NON IMITARE! *)
var ris_exp: real;
    somma, prod: integer;

  procedure exp(x: real; n: integer);
  begin
    ris_exp:= 1;
    while n >= 1 do begin
      ris_exp:= x*ris_exp;
```

```
        n:= n-1
      end;
    end;

    procedure sp(m,n: integer);
    var i: integer;
    begin
      somma:= 0; prod:= 1;
      for i:= m to n do begin
        somma:= somma+i; prod:= prod*i;
      end;
    end;

  begin …
```

D'altra parte, sia le variabili locali, come `ris` nella procedura che calcola l'esponenziale, sia i parametri-valore (come `m`, `n`, ecc.) vengono distrutti (insieme ai loro contenuti) all'uscita dalla procedura: non sono dunque nè visibili nè esistenti quando si ritorna al chiamante, e non è pertanto possibile utilizzarli per comunicare dei valori al programma principale.

I parametri per valore sono dunque sempre parametri di ingresso (input): essi permettono di passare informazione dal programma chiamante al sottoprogramma chiamato, ma non viceversa. Per questo secondo scopo sono però disponibili altri meccanismi, quali quello dei parametri-variabile, oppure dei sottoprogrammi-funzione.

Nella definizione della procedura i *parametri-variabile*, o *parametri per riferimento*, sono qualificati dalla parola-chiave *var*.

```
procedure exp(x: real; n: integer; var ris: real);
var i: integer;
begin
  ris:= 1;
  for i:= 1 to n do ris:= x*ris
end;

procedure sp(m,n: integer; var somma, prod: integer);
var i: integer;
begin
  somma:= 0; prod:= 1;
  for i:= m to n do begin
    somma:= somma+i; prod:= prod*i;
  end;
end;
```

Nelle chiamate della procedura, gli argomenti effettivi corrispondenti a tali parametri possono essere soltanto (nomi di) variabili, e non generiche espressioni come nel caso del passaggio per valore (studiando i vettori e i puntatori si vedrà che vi possono essere espressioni aventi come valore una variabile, ma per ora ciò non è possibile).

All'istante della chiamata nel record di attivazione vi è una cella per ogni parametro-variabile, ed in essa viene messo l'indirizzo del corrispondente parametro effettivo; il corpo della procedura utilizza poi tali indirizzi per accedere alle variabili del chiamante. Nel r. di a. vi è quindi una cella per ogni parametro formale, sia esso per valore o per indirizzo; nei parametri per valore vengono messi i valori dei parametri effettivi, nei parametri per indirizzo gl'indirizzi.

All'interno del sottoprogramma ogni riferimento a un parametro-variabile viene automaticamente interpretato come un riferimento alla variabile costituente il parametro effettivo. In particolare, ogni modifica (ad esempio assegnazione) di un parametro-variabile è una modifica del parametro effettivo, e non di una sua inesistente copia locale;

cosí, se nel programma principale compare l'istruzione `exp(bas,esp,pot)`, dove `bas`, `esp`, `pot` siano variabili globali opportunamente dichiarate, durante l'esecuzione della procedura *exp* viene via via modificata - dalla ripetuta esecuzione dell'istruzione `ris:=x*ris` - proprio la variabile globale `pot`.

Nella maggior parte dei casi l'effetto è come se, ad ogni chiamata di procedura, i nomi dei parametri formali fossero letteralmente sostituiti, nel testo del sottoprogramma, dai nomi dei parametri effettivi, cioè diventassero sinonimi dei nomi dei parametri effettivi. Ad esempio, se nel programma principale compare l'istruzione `sp(a,b,s,p)`, dove `a,b,s,p` sono variabili globali opportunamente dichiarate, l'esecuzione di tale istruzione è equivalente alla chiamata di una immaginaria procedura:

```
procedure sp_fittizia(m,n: integer);
var i: integer;
begin
  s:= 0; p:= 1;
  for i:= m to n do begin
    s:= s+i; p:= p*i
  end
end;
```

Se compare la chiamata `exp(b,e,pot)`, la sua esecuzione è equivalente all'esecuzione dell'immaginaria procedura:

```
procedure exp_fittizia;
var i: integer;
begin
  pot:= 1;
  for i:= 1 to n do pot:= x*pot
end;
```

Ciò però non è vero sempre, e in realtà il modello della sostituzione letterale dei nomi qui usato corrisponde in parte ad un altro modo di passaggio parametri, il *passaggio per nome*, ormai abbandonato nei linguaggi moderni, che è stato un po' il predecessore del passaggio per riferimento. Per la correttezza del modello di sostituzione letterale si deve precisare che se il nome di una variabile-argomento-effettivo è uguale al nome di una variabile locale del sottoprogramma, per evitare il conflitto di nomi bisogna ridenominare la variabile locale. Anche con questa precisazione, però, i due modelli non sono del tutto equivalenti, come vedremo piú avanti.

I parametri-variabile (come quelli per nome, del resto) possono essere sia di uscita che di ingresso, o anche di ingresso/uscita (input/output) contemporaneamente, come nella seguente modifica della procedura `sp`:

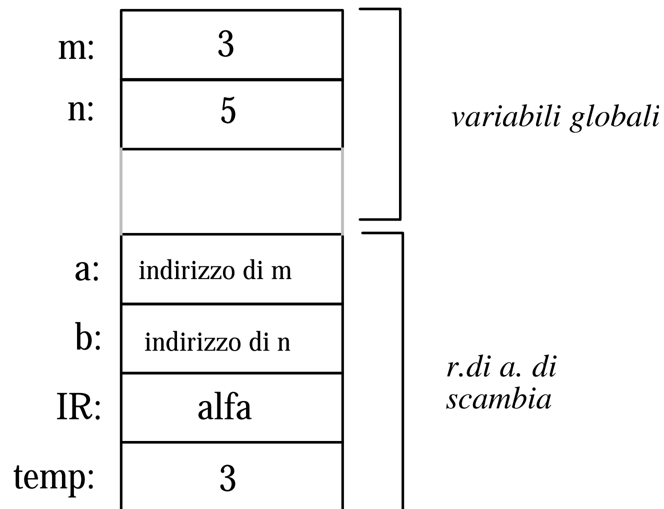
```
procedure sp(var m,n: integer);
var i,somma,prod: integer;
begin
  somma:= 0; prod:= 1;
  for i:= m to n do begin
    somma:= somma + i; prod:= prod * i;
  m:= somma; n:= prod
  end;
end;
```

In essa i parametri `m` ed `n` vengono utilizzati sia per comunicare alla procedura i valori di ingresso, sia per restituire al chiamante i risultati (attraverso le due assegnazioni finali).

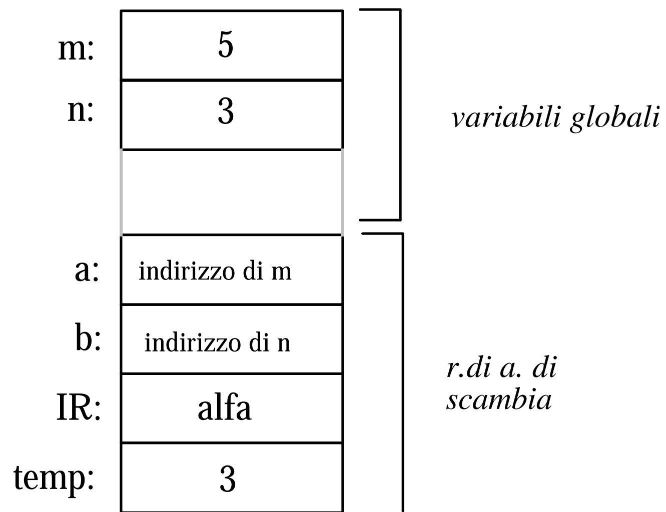
Un esempio classico di procedura con parametri per riferimento (di input/output contemporaneamente) è la procedura che scambia fra loro i contenuti di due variabili:

```
var m,n: integer;
    ...
  procedure scambia(var a,b: integer);
  var temp: integer;
  begin
    temp:= a;
    a:= b;
    b:= temp
  end;

begin
  ...
  scambia(m,n); {alfa:}  ...;
  ...
end.
```

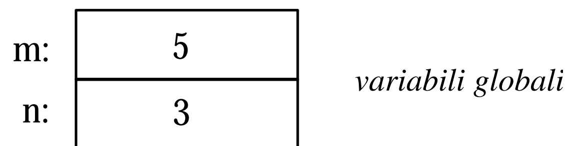
Supponiamo che al momento dell'invocazione della procedura le variabili globali `m` ed `n` contengano rispettivamente i valori 3 e 5. Lo stato della memoria-dati nell'istante immediatamente successivo all'esecuzione dell'istruzione `temp:=a` della procedura *scambia* è allora:



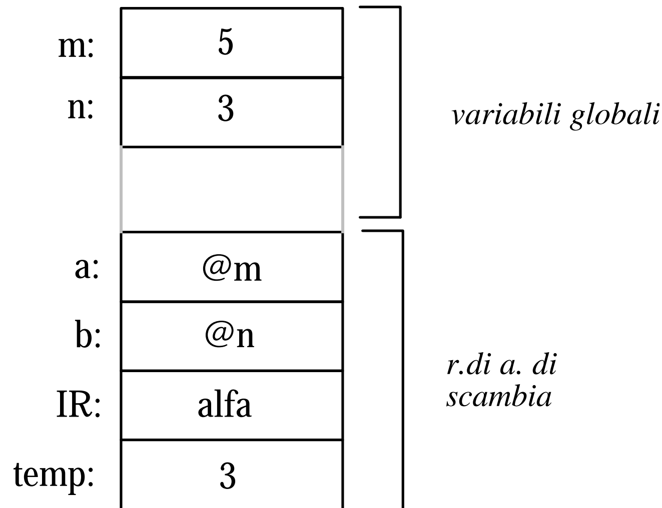
Nell'istante immediatamente prima dell'esecuzione dell'*end* della procedura lo stato della memoria sarà:



Dopo il ritorno nel programma principale la memoria sarà:



Nella rappresentazione grafica, per indicare concisamente l'indirizzo di una variabile `a`, useremo spesso la notazione *@a* (mutuata dal TurboPascal, dove il simbolo @ designa appunto l'operatore *indirizzo di*), come nella figura sottostante.



Omettendo la parola-chiave *var* nella definizione della procedura, e adottando cosí il passaggio parametri per valore, lo scambio viene effettuato sulle copie locali `a` e `b` delle variabili `m` ed `n`, ma queste ultime restano invariate, e quindi la procedura non ha alcun effetto:

```
var m,n: integer;
    ...

  procedure scambia_errato(a,b: integer);
  var temp: integer;
  begin
    temp:= a;
    a:= b;
    b:= temp
  end;

begin
  ...
  scambia_errato(m,n);
alfa:  ...;
...
end.
```

Lo stato della memoria-dati nell'istante immediatamente successivo all'esecuzione dell'istruzione `temp:= a` della procedura *scambia* è allora:

m:	3	*variabili globali*
n:	5	
a:	3	*r. di a. di scambia_errato*
b:	5	
IR:	...	
temp:	3	

Nell'istante immediatamente prima dell'esecuzione dell'*end* della procedura lo stato della memoria sarà:

m:	3	*variabili globali*
n:	5	
a:	5	*r. di a. di scambia_errato*
b:	3	
IR:	alfa	
temp:	3	

Dopo il ritorno nel programma principale la memoria sarà :

m:	3	*variabili globali*
n:	5	

Per finire, citiamo un esempio in cui il modello di passaggio parametri per sostituzione letterale dei nomi non è equivalente al passaggio per riferimento. Si consideri la chiamata:

```
scambia(m, v[m]);
```

dove `m` sia una variabile di tipo intero, e `v` un vettore di interi (vedi *Capitolo 3*).
Con il passaggio parametri per riferimento tale chiamata effettua, come ci si aspetta, lo scambio dei contenuti delle celle di memoria `m` e `v[m]` (se `m` contiene il valore 3 e `v[3]` contiene il valore 5, dopo la chiamata `m` contiene 5 e `v[3]` contiene 3). Se invece si opera una sostituzione letterale si ottiene:

```
temp:= m;
m:= v[m];
v[m]:= temp;
```

la cella `v[m]` nell'ultima assegnazione non è piú quella denotata da `v[m]` nella seconda istruzione!

Nota Bene: Non si confonda l'*input/output* da periferiche con il genere *input* o *output* dei parametri.
E ancora: si osservi che le primitive di input *read*, *readln*, ecc. sono procedure predefinite con un parametro di output, mentre le primitive di output *write*, *writeln*, ecc. sono procedure con un parametro di input!

Esercizio 2. Si trasformino alcuni dei programmi ottenuti come soluzioni degli esercizi del *Capitolo 1* in sottoprogrammi che restituiscono gli opportuni risultati (invece di scriverli sullo schermo); si scriva poi un semplice programma principale che richiama tali sottoprogrammi effettuando l'input/output.

2.7 Funzioni.

Nei casi in cui il risultato da comunicare è uno solo, ad esempio perchè il sottoprogramma realizza il calcolo di una funzione nel senso matematico del termine, cioè associa ad ogni n-upla di valori d'ingresso uno e un solo valore di uscita senza produrre altri effetti, si può (anzi è conveniente) usare il meccanismo della *function*.

Le funzioni, nei linguaggi di programmazione, sono delle forme di sottoprogramma che "restituiscono direttamente un valore" e permettono cosí la scrittura di espressioni analoghe a quelle della matematica.

Se ad esempio volessimo calcolare il valore dell'espressione aritmetica

$$(a \cdot b^4 + c \cdot d^3)^2 + 3 \cdot h$$

richiamando la procedura `exp` per il calcolo dell'esponenziale, definita nella sezione precedente:

```
procedure exp(x:real; n:integer; var ris: real);
var i: integer;
begin
  ris:= 1;
  for i:= 1 to n do ris:= x*ris
end;
```

saremmo costretti ad usare delle variabili intermedie:

```
...
exp(b,4,temp1);
exp(d,3,temp2);
```

```
exp(a*temp1 + c*temp2, 2, temp2);
... temp2 + 3*h ...
```

L'uso di un sottoprogramma-funzione permette di evitare tale inconveniente e di usare anche nel linguaggio di programmazione una notazione analoga a quella matematica.

Una definizione di funzione è del tutto simile ad una definizione di procedura, con la differenza che in essa occorre specificare obbligatoriamente, oltre agli eventuali parametri, il tipo del valore "restituito dalla funzione". All'interno del corpo della funzione, poi, per restituire un valore al chiamante occorre "assegnarlo" al nome della funzione stessa, come nell'esempio seguente:

```
function exp(x: real; n: integer): real;
var i, ris: integer;
begin
  ris:= 1;
  for i:= 1 to n do ris:= x*ris;
  exp:= ris
end;
```

Una chiamata di funzione, diversamente da quella di una procedura, non è un'istruzione (salvo che con l'opzione "sintassi estesa" del TurboPascal, che per ora non consideriamo), bensí un'espressione che denota il valore restituito dalla sua esecuzione, e può quindi comparire a destra del simbolo di assegnazione, o da sola oppure combinata con altre operazioni o funzioni per formare un'espressione piú complicata, ecc.

In questo modo, l'espressione precedente può essere tradotta tal quale in Pascal, nella forma:

```
exp(a*exp(b,4)+c*exp(d,3), 2) + 3*h
```

Si osservi che la definizione della funzione `exp` si è ottenuta dalla definizione della procedura `exp` semplicemente trasformando il parametro di uscita `ris` in una variabile locale, e aggiungendo come ultima istruzione l'assegnazione del valore di `ris` al nome della funzione.

2.8 Particolarità delle funzioni.

Il linguaggio Pascal presenta, per quanto riguarda le funzioni, alcune particolarità (e differenze rispetto ad altri linguaggi) che occorre comprendere bene per poter scrivere programmi che operino correttamente:

1) L'assegnazione di un valore al nome della funzione non fa uscire dal sottoprogramma, cioè non restituisce il controllo al chiamante, come fa invece l'istruzione *return* di molti linguaggi (fra cui il C); l'uscita dal sottoprogramma si ha solo al raggiungimento dell'*end* della funzione.

2) Nel corso dell'esecuzione della funzione possono venire successivamente eseguite assegnazioni del nome della funzione a valori diversi; il valore che sarà restituito al chiamante è l'ultimo che è stato assegnato.

3) La caratteristica precedente potrebbe far pensare che il nome della funzione, all'interno del corpo della funzione stessa, si comporti esattamente come una variabile, cioè come una specie di parametro di output implicito; ad esempio nella

definizione di `exp` si potrebbe pensare di poter eliminare la variabile `ris` e scrivere direttamente:

```
 exp:= 1;
for i:= 1 to n do exp:= x*exp;
```

Invece non è cosí: il nome della funzione nel senso di "variabile contenente il risultato" può essere usato soltanto a sinistra del simbolo di assegnazione: esso si comporta cioè come una strana variabile "a sola scrittura", a cui si possono assegnare valori successivamente diversi ma che non può mai essere letta.

La caratteristica 3 può sembrare una ingiustificata limitazione sintattica; in realtà essa ha una sua motivazione, che cerchiamo di spiegare.

Cominciamo con l'osservare che una funzione, esattamente come una procedura, può anche essere priva di parametri; ad esempio, la seguente banale funzione senza parametri restituisce il giorno del mese, servendosi della primitiva *getdate* del TurboPascal:

```
function oggi: integer;
var anno, mese, giorno, giornosett: word;
begin
  getdate(anno, mese, giorno, giornosett);
  oggi:= giorno
end;
```

Uniformemente con le procedure, l'invocazione di una tale funzione si ottiene semplicemente scrivendone il nome, ad esempio:

```
var ieri: integer;
...
  ieri:= oggi - 1;
  if ieri < 1 then begin
  ...
```

Come si vede, in Pascal (a differenza che in C) la chiamata di una funzione senza parametri è indistinguibile dal nome di una variabile o di una costante.

D'altra parte in Pascal, come in tutti i linguaggi moderni, un sottoprogramma non solo può richiamare un altro sottoprogramma, ma può addirittura chiamare se stesso: si tratta della tecnica di programmazione detta "ricorsiva", che studieremo nella seconda parte del corso. Ad esempio, la versione ricorsiva dell'esponenziale è:

```
function expric(x:real; n: integer): real;
begin
  if n=0 then expric:= 1 else expric:= x*expric(x,n-1)
end;
```

Supponiamo ora di scrivere la seguente definizione di funzione senza parametri:

```
function leggi: integer;
var n: integer;
begin
  leggi:= 0;
  if not eof then begin
    read(n);
    leggi:= leggi + n
  end
end;
```

Se all'interno di una funzione il nome della funzione stessa potesse denotare il risultato, la definizione precedente sarebbe ambigua: l'istruzione `leggi:= leggi + n` incrementa di *n* il risultato, oppure effettua una chiamata ricorsiva della funzione (e poi restituisce la somma di *n* col valore ottenuto)?

L'interpretazione corretta è la seconda, altrimenti le funzioni senza parametri non potrebbero essere ricorsive, e si avrebbe cosí una disuniformità nel linguaggio.

Per evitare un'analoga brutta disuniformità sintattica la regola si applica allora a tutte le funzioni, anche se dotate di parametri: un nome di funzione il quale si trovi in "posizione-espressione", cioè all'interno di un'espressione o comunque nel secondo membro di un'assegnazione, viene sempre interpretato dal compilatore come una chiamata alla funzione stessa, eventualmente con un numero errato di parametri. Ad esempio, se all'interno della funzione *exp* si scrive l'istruzione `exp:= x*exp`, l'`exp` di destra viene interpretato come una chiamata ricorsiva con zero argomenti invece di due, e segnalato quindi come errore.

A tale poco piacevole particolarità sintattica del Pascal viene posto rimedio, nella versione del TurboPascal contenuta nel sistema *Delphi*, attraverso l'introduzione della variabile speciale *result*. Una variabile di tale nome, che non deve essere dichiarata, può essere usata nel corpo di una funzione come una ordinaria variabile del tipo restituito dalla funzione, sia a sinistra che a destra del simbolo di assegnazione; all'uscita dal sottoprogramma il suo contenuto viene automaticamente restituito come risultato della funzione.

Osserviamo infine che le funzioni Pascal non sono soltanto un mezzo per realizzare funzioni in senso matematico; una *function*, a parte la caratteristica di dover "restituire un valore", può per il resto essere un sottoprogramma del tutto analogo ad una procedura, in particolare può avere parametri sia per valore che per riferimento, e può quindi restituire in opportuni parametri di output "altri risultati" oltre a quello "principale" restituito "nel proprio nome".

Ad esempio, la procedura *sp* della *Sezione 2.5*, che restituiva contemporaneamente la somma ed il prodotto dei naturali da *m* ad *n*, potrebbe essere riscritta come una funzione che restituisce il prodotto come "risultato principale" e restituisce la somma, come prima, in un parametro di output:

```
function sp(m,n: integer; var somma: integer): integer;
var i, prod: integer;
begin
  somma:= 0; prod:= 1;
  for i:= m to n do begin
    somma:= somma + i; prod:= prod * i
  end;
  sp:= prod
end;
```

La definizione e l'uso di sottoprogrammi di questo genere, "ibridi" fra procedura e funzione, detti anche *funzioni con effetto collaterale (o side-effect)*, utili in molti casi, devono essere affrontati con cautela, perchè possono indurre ad errori e comunque a realizzare programmi di difficile comprensione.

Esempio: qual è il numero visualizzato sullo schermo dal seguente programma principale che usa la funzione *sp* sopra definita?

```
...
var a,b,c: integer;
begin
```

```
  a:= 3; b:= 5; c:= 1;
  writeln(sp(a,b,c) + c);
end.
```

Risposta: dipende dall'implementazione! la definizione del linguaggio non permette di dare una risposta!

Infatti la definizione del Pascal (e del TurboPascal) non specifica in quale ordine debbano essere valutati gli operandi di un'espressione, ma lascia libertà all'esecutore di valutarli sempre da sinistra a destra o da destra a sinistra oppure secondo strategie piú complesse. Nel caso dell'esempio, se l'esecutore valuta gli operandi dell'addizione da sinistra a destra, esegue prima la chiamata di *sp*, che restituisce il valore 60 ma contemporaneamente mette il valore 12 in `c`, sicchè facendo poi la somma si ottiene 72; se invece gli argomenti vengono valutati da destra a sinistra, si legge prima il contenuto di `c`, che è 1, e poi si invoca la funzione, ottenendo cosí come risultato finale 61.

Esercizio 3. Si provi al calcolatore il programma precedente; si riprovi dopo aver scambiato fra di loro i due addendi: che cosa succede? che cosa si può ricavare riguardo alla strategia dell'esecutore? Si provi ora ad eseguire il programma principale seguente:

```
...
var c: integer;
begin
  c:= 0;
  writeln(sp(1,2,c) + c + sp(3,5,c))
end.
```

Si provino a scambiare fra di loro le due invocazioni di *sp*. Che cosa succede?

Le due porzioni di programma precedenti, pur essendo compilabili ed eseguibili senza errori, sono comunque logicamente scorrette, perchè il loro comportamento non è definito dalla definizione del linguaggio (nemmeno da quella del dialetto Turbo), e quindi può variare da una versione all'altra del compilatore.

2.9 Record di attivazione di funzioni.

Il record di attivazione che si crea sullo *stack* quando viene attivata una funzione è analogo a quello per le procedure, con celle per i parametri e per le variabili locali. È conveniente inoltre immaginare che il record di attivazione di una funzione possieda una cella supplementare per il risultato, corrispondente alla pseudo-variabile avente il nome della funzione stessa (oppure, in *Delphi*, alla variabile speciale *result*). Nella rappresentazione grafica disegneremo tale cella come la prima (la piú in alto) del record di attivazione.

Una differenza significativa fra i record di attivazione di procedure e di funzioni riguarda invece la nozione di indirizzo di ritorno. Come si è visto, nel caso delle chiamate di procedure l'IR è semplicemente quello dell'istruzione successiva alla chiamata, cioè dell'istruzione che nel testo del programma segue immediatamente - separata di solito da un punto e virgola - l'istruzione di chiamata. Una chiamata di funzione, invece, non è un'istruzione, bensí un'espressione, che può comparire all'interno di un'altra espressione, oppure direttamente in un'istruzione (ad es. di assegnazione); al termine della chiamata si ritorna quindi ad eseguire l'operazione rimasta in sospeso.

Esempio:

```
...
a:= m*f(n);
```

L'IR della chiamata di `f` è quello dell'operazione di moltiplicazione, che naturalmente può venire eseguita soltanto dopo che è terminata l'esecuzione di `f`.

Il lettore potrebbe chiedersi che cosa sia mai "l'indirizzo di un'operazione". Una trattazione rigorosa ed esaustiva della nozione di "ritorno" da una chiamata di funzione è al di là degli scopi di queste note; tuttavia, per capire approssimativamente di che si tratta, bisogna non considerare piú un'istruzione Pascal come un'azione atomica, bensí analizzarla in modo piú fine come una sequenza di azioni piú elementari, che utilizzano variabili temporanee "nascoste". Cosí, l'istruzione dell'esempio precedente deve essere pensata tradotta nelle seguenti istruzioni "piú semplici", dove `temp` è una variabile temporanea "generata" dall'esecutore Pascal (magari realizzata con un registro) ma invisibile al programmatore:

```
temp:= f(n);
a:= m*temp;
```

allora l'indirizzo "dell'operazione di moltiplicazione" è in realtà l'indirizzo della seconda delle istruzioni precedenti.

Per poter rappresentare simbolicamente il contenuto della cella IR del record di attivazione di una funzione, adotteremo la soluzione di etichettare nel programma operazioni oltre che istruzioni, inventando per questo soluzioni grafiche intuitive, come nell'esempio seguente. La nozione di "operazione" va intesa in senso molto ampio, come vedremo nella sottosezione seguente.

```
function exp(x: real; n: integer): real;
var i: integer; ris: real;
begin
  ris:= 1;
  for i:= 1 to n do ris:= x*ris;
  exp:= ris;
end;

var r: real; k: integer;

begin
  write('immetti base: ');
  readln(r);
  write('immetti esponente: ');
  readln(k);
  r:= exp(r,k);
{
  alfa   }
  ...
end.
```

L'indirizzo di ritorno della chiamata della funzione *exp* è l'operazione di assegnazione, lo stato della memoria-dati all'istante dell'esecuzione del *begin* della funzione è allora rappresentato dalla figura seguente.

r:	5	*variabili globali*
k:	3	
exp:	⊥	*r. di a. di exp*
x:	5	
n:	3	
ris:	⊥	
IR:	alfa	
i:	⊥	

Se la chiamata della funzione fosse della forma:

```
t:= m + p*exp(r,k) - q;
```

l'indirizzo di ritorno sarebbe quello dell'operazione di moltiplicazione:

```
t:= m + p *_exp(r,k) - q;
       {
         alfa }
```

Poichè, come abbiamo visto, la definizione del linguaggio non stabilisce l'ordine di valutazione degli operandi di un'espressione ma li lascia alla scelta dell'implementatore, può succedere che in base alla definizione non si possa stabilire quale sia esattamente l'operazione di ritorno da una chiamata di funzione. Ad esempio, nell'espressione:

```
a*b + f(c)
```

se l'esecutore Pascal valuta prima l'addendo di sinistra e poi quello di destra, l'operazione da eseguire al ritorno dalla chiamata di *f* sarà l'addizione; se invece l'esecutore Pascal valuta prima l'addendo di destra, al ritorno dalla funzione bisognerà ancora eseguire, prima dell'addizione, la moltiplicazione per valutare il primo addendo, come se il programma fosse:

```
temp1:= f(c);
temp2:= a*b;
temp3:= temp1+temp2;
```

2.10 Chiamate annidate di sottoprogrammi.

Con le funzioni si introduce la possibilità di chiamate annidate di sottoprogrammi; ad esempio:

```
function g(x: integer): integer;
begin
  g:= 3*x+1;
end;

procedure f(x: integer): integer;
begin
  f:= 2*x*x*x - 5
end;

procedure p(n: integer);
begin
  if n<0 then writeln('negativo')
  else writeln(n)
end;

var a: integer;

begin
...
  p(f(g(a)));
...
```

Per capire che cosa succede esattamente quando viene eseguita l'istruzione di chiamata di *p*, bisogna precisare il modello di calcolo relativo alle chiamate di sottoprogramma. In particolare, bisogna precisare che nell'invocazione di un sottoprogramma:

1) prima vengono valutati (in un ordine non specificato) gli argomenti della chiamata;
2) poi viene attivato il sottoprogramma e quindi creato il corrispondente record di attivazione, all'interno del quale i valori degli argomenti vengono memorizzati (nelle celle parametri formali).

Tale fatto può non apparire molto significativo finchè gli argomenti sono variabili o semplici espressioni aritmetiche; con l'introduzione delle funzioni, però, il calcolo di un'espressione - ad esempio di un argomento di una chiamata - può comportare chiamate di sottoprogrammi, e può quindi essere una computazione complessa quanto si vuole, e la precisazione precedente diventa importante.

In base a tale modello di calcolo, l'esecuzione dell'istruzione di chiamata di p avviene nel modo seguente.

Dapprima viene creato un r. di a. per *g*, distrutto il quale - al termine del calcolo di *g(a)* - viene creato un r. di a. per *f*, che sarà poi a sua volta distrutto alla fine del calcolo di *f* e seguito finalmente dalla creazione del r. di a. di *p*. Cioè, in caso di chiamate "annidate" di funzione, le corrispondenti attivazioni vengono eseguite a partire dalla piú interna risalendo verso l'esterno. Ciò equivale a dire che l'istruzione `p(f(g(a))` deve pensarsi tradotta nella sequenza di "istruzioni semplici":

```
        temp1:= g(a);
{alfa:} temp2:= f(temp1);
{beta:} p(temp2);
```

L'indirizzo di ritorno `alfa` dalla chiamata di *g*, memorizzato nella cella IR, è quindi quello della chiamata di *f*, ed a sua volta l'indirizzo di ritorno `beta` dalla chiamata di *f* è quello della chiamata di *p*. Nel programma Pascal originale tali "indirizzi di ritorno" sono indicabili con difficoltà: possiamo cercare di usare frecce che puntano ai nomi delle funzioni nelle chiamate, o artifici grafici simili. Ad esempio:

```
        p(f(g(a)));
```

```
beta alfa
```

L'ordine di valutazione degli argomenti di un sottoprogramma (se da sinistra a destra, oppure ...) è invece, come per gli operandi delle operazioni predefinite, dipendente dall'implementazione.

2.11 Nomi e visibilità.

I nomi globali - di variabili, di costanti, di procedure, e, come vedremo, di tipi - ovviamente non possono essere duplicati, cioè tutti i nomi dichiarati a livello globale devono essere distinti. La stessa cosa vale per ogni sottoprogramma, cioè tutti i nomi di parametri formali e di variabili locali della procedura o funzione devono essere distinti.

Tuttavia, ogni sottoprogramma introduce uno "spazio di nomi" distinto da quello del programma principale (e distinto da quello di ogni altro sottoprogramma), cioè in una procedura o funzione si possono avere parametri o variabili locali aventi lo stesso nome di variabili globali, o di variabili locali di altri sottoprogrammi.

In questo modo possono essere contemporaneamente presenti nella memoria dell'esecutore Pascal variabili diverse aventi lo stesso nome; tuttavia, le regole del Pascal assicurano che in ogni punto del programma una sola di esse è visibile. Si vedano a tale proposito i manuali del linguaggio; per ora basti dire che, se in un sottoprogramma vi è un parametro o variabile locale avente lo stesso nome di una variabile globale, nel corpo del sottoprogramma con tale nome ci si riferisce sempre al parametro o variabile locale. La variabile globale omonima diventa "invisibile" e quindi inaccessibile dal sottoprogramma: si dice che la variabile locale la "scherma" o "le fa ombra".

Esempio:

```
var m,n: integer;
    r,s: real;

  procedure p(n: integer);
  var r,t: real;
  begin
    r:= n/3;
    ...
    m:= m+1;
    ...
  end;

  function f(s: string; n: integer): boolean;
  begin
    if length(s) > n then ...
    ...
  end;

begin
  m:= 0;
  readln(m,n);
  p(m-n);
  ...
end.
```

Nel corpo della procedura `p` i nomi `n` ed `r` si riferiscono rispettivamente al parametro formale `n` ed alla variabile locale `r`; le variabili globali `n` ed `r` sono ivi inaccessibili; la variabile globale `m`, invece, che non è schermata da alcun nome locale, è visibile ed

accessibile dal corpo della procedura, anche se - come vedremo - eccetto che in casi molto particolari è una cattiva pratica di programmazione avere procedure che operino direttamente su variabili globali.

Analogamente, nel corpo della funzione `f` il nome `s` si riferisce al parametro di tipo *string* e non alla variabile globale di tipo *real*: in esso un'assegnazione come `s:=3.14` sarebbe quindi segnalata come errore; eccetera.

Si noti che durante l'esecuzione di un'istruzione di chiamata di sottoprogramma la valutazione degli argomenti avviene nel contesto (o, come si dice meglio, nell'*ambiente*) del chiamante, nel nostro caso il programma principale; ciò si accorda perfettamente con il fatto che l'attivazione del chiamato avvenga, come si è spiegato, solo dopo il calcolo degli argomenti.

Nel testo di un programma Pascal, dunque, non vi può mai essere ambiguità su che cosa un'occorrenza di un nome indichi: se a un certo punto compare il nome `pippo`, anche se nel programma sono definiti dieci `pippo` diversi, in quel punto del programma il nome `pippo` non può che designare uno e un solo di tali dieci enti (ciò che abbiamo detto qui per i nomi di variabile vale infatti in generale per tutti i nomi: di procedura, di tipo, ecc.).

Se invece vogliamo ragionare sul programma, o rappresentare lo stato dell'esecutore, può darsi che si debba parlare contemporaneamente, come abbiamo fatto sopra, di variabili o enti diversi con lo stesso nome; allora, per indicare concisamente a quale ci si riferisce, premetteremo al nome di ente locale (variabile o parametro) il nome della procedura in cui essa è definita, separato da un punto; cosí, nell'esempio precedente, per indicare il parametro *n* rispettivamente di *p* e di *f*, scriveremo semplicemente *p.n* e *p.f*; per indicare la variabile globale *n* scriveremo semplicemente *n*, oppure *main.n*; eccetera.

Attenzione: ribadiamo che tale notazione non fa parte del linguaggio Pascal, ma solo del linguaggio con cui in queste note parliamo di programmi Pascal.

Per finire, non si confonda il *tempo di vita* di una variabile, che è il tempo durante il quale la variabile esiste in memoria, con la sua *visibilità*. Ad esempio le variabili globali `n` ed `r` esistono per tutta la durata dell'esecuzione del programma, esattamente come `m` ed `s`; durante l'esecuzione della procedura `p`, pur non essendo accessibili, continuano ad esistere con i loro contenuti immutati; al ritorno dalla procedura al programma principale ridiventano accessibili.

Le variabili locali `r` e `t` della procedura `p`, invece, cosí come il suo parametro `n`, al di fuori del corpo della procedura - ad esempio nel corpo del programma principale - non sono visibili e non esistono: infatti le variabili locali di un sottoprogramma esistono soltanto durante l'esecuzione del sottoprogramma, nel corrispondente record di attivazione.

Le celle di memoria che realizzano una stessa variabile locale nelle varie chiamate del sottoprogramma devono essere considerate come variabili distinte o, se si preferisce, come "incarnazioni distinte" della stessa variabile; anche nel ragionare sul programma, però, le indichiamo tutte con lo stesso nome, giacchè ad ogni istante dato (finchè non si usano sottoprogrammi ricorsivi) vi può essere una sola "incarnazione" presente in memoria, e quindi non vi può essere ambiguità.

2.12 Lo *stack* (la pila) dei record di attivazione.

Come abbiamo già accennato, un sottoprogramma può essere chiamato non solo dal programma principale, ma da qualunque sottoprogramma, che in tal caso "si sospende" a sua volta in attesa che il chiamato termini il proprio lavoro, proprio come il programma

principale. Si possono quindi avere contemporaneamente in vita piú record di attivazione, corrispondenti a chiamate di sottoprogrammi diversi (o anche, nel caso della ricorsione, a chiamate diverse dello stesso sottoprogramma); naturalmente uno solo di essi, precisamente quello che è stato creato per ultimo, è "attivo"; gli altri corrispondono a sottoprogrammi "sospesi". Si osservi che fra tutti i r. di a. presenti in memoria ad un dato istante, il primo che sarà distrutto è quello attivo, cioè l'ultimo creato; per questa ragione in Pascal (e negli altri linguaggi dello stesso genere) i r. di a. successivamente creati devono essere pensati come "impilati": ogni nuovo r. di a. viene creato o "allocato" sulla cima (*top*) della pila (*stack*); il r. di a. attivo è sempre quello in cima alla pila; il r. di a. che viene di volta in volta distrutto è quello in cima alla pila (cioè un r. di a. non può venire "tolto dalla pila" finchè non sono stati tolti tutti quelli "sopra di lui").

Si vedrà in corsi successivi che il record di attivazione di un sottoprogramma contiene, oltre a IR, parametri e variabili locali, anche altre informazioni che permettono una gestione efficace dello stack, e inoltre lo spazio necessario per il calcolo delle espressioni (cioè le variabili temporanee in cui memorizzare i risultati intermedi).

Riassumendo, la memoria di lavoro dell'esecutore Pascal è costituita da:

- la *memoria di programma* in cui sono memorizzate le istruzioni che compongono il programma;

- la *memoria-dati* su cui le istruzioni operano, che è a sua volta suddivisa in due parti:
  1) l'area delle *variabili globali*, che contiene tutte le variabili dichiarate nel programma principale;
  2) lo *stack* o *pila*, che contiene via via i record di attivazione dei diversi sottoprogrammi.
  (vedremo nella seconda parte del corso che vi è poi un'altra area-dati, detta *heap*).

L'area delle variabili globali rimane immutata, in quanto al numero e al tipo delle variabili, per tutta la durata dell'esecuzione (mentre naturalmente ne varierà in generale il contenuto!); lo spazio della pila invece si estende e si contrae nel corso dell'esecuzione.

Poichè lo stack viene disegnato naturalmente a partire dalla base, mentre la nostra scrittura ordinaria va dall'alto verso il basso, in queste note adotteremo la convenzione di disegnare lo stack come crescente verso il basso, cioè capovolto rispetto alla nozione intutiva di "pila", avendo la cima in basso e il fondo in alto. In molti testi, e nel debugger del TurboPascal, si usa la convenzione opposta, con la cima in alto. Le due rappresentazioni sono naturalmente del tutto equivalenti, l'importante è sceglierne una.

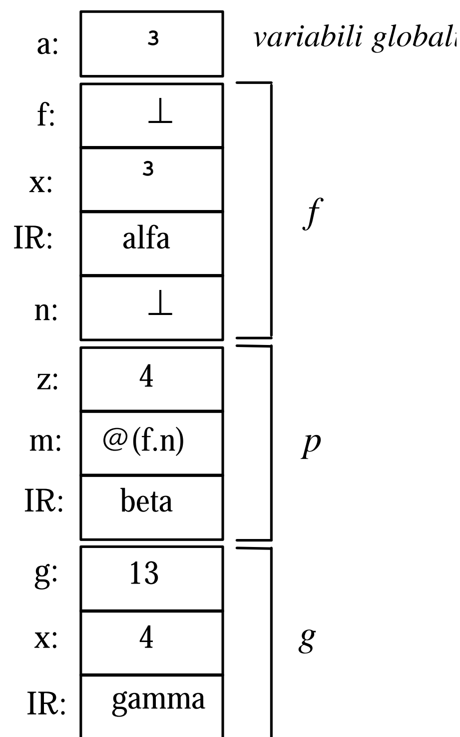
Esempio: si consideri il seguente "artificiale" programma:

```
var a: integer;

  function g(x: integer);
  begin
    g:= 3*x+1;
  end;

  procedure p(z: integer; var m: integer);
  begin
    m:= 2*g(z)
  end;
       gamma

  function f(x: integer): integer;
  var n: integer;
```

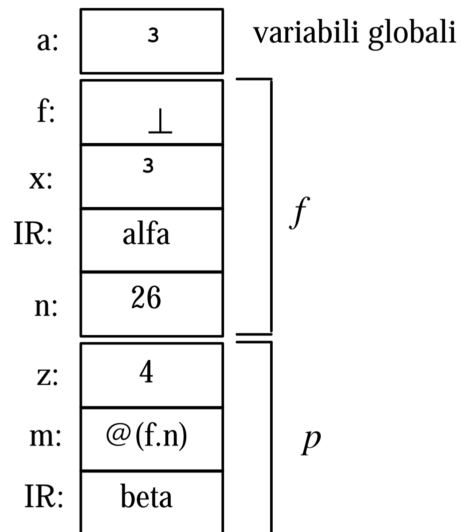
```
      begin
        p(x+1,n);
  beta: f:= n+5
      end;

    begin
      a:= 3; a:=_f(a); writeln(a)
    end.
             alfa
```

Nell'esecuzione di tale programma ogni sottoprogramma viene invocato una sola volta; allora non ci sono ambiguità se parliamo di stato della memoria immediatamente prima dell'esecuzione dell'*end* della funzione *g*, o dell'*end* della procedura *p*, ecc. Alcuni stati durante l'esecuzione sono rappresentati nei disegni seguenti.

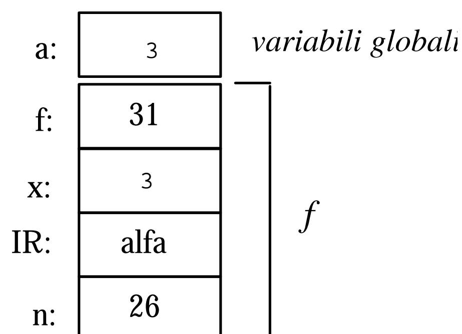
Istante 1) Successiva istruzione da eseguire: l'*end* della funzione *g*.



Istante 2) Successiva istruzione da eseguire: l'*end* della procedura *p*.


a:
3
variabili globali
f:
⊥
x:
3
IR:
alfa
n:
26
f
z:
4
m:
@(f.n)
p
IR:
beta

Istante 3) successiva istruzione da eseguire: l'*end* della funzione *f*.



Istante 4) successiva istruzione da eseguire: il *writeln(a)* alla fine del programma.

Nei linguaggi ad allocazione completamente statica lo stack, inesistente come struttura fisica che si dilata e si contrae, può essere visto astrattamente come la rappresentazione, ad ogni dato istante, della parte *viva* della memoria, cioè della parte di memoria potenzialmente rilevante, in quell'istante, per l'esecuzione del programma Pascal.

Infatti, benchè in tale modello tutte le aree-dati locali siano sempre contemporaneamen-te presenti in memoria, ad ogni istante una sola di esse, quella corrispondente al sottoprogramma in effettiva esecuzione, è *attiva*, mentre un certo numero di altre sono *vive* ma *sospese*, cioè corrispondenti a sottoprogrammi la cui esecuzione non è ancora terminata ma solo sospesa per la chiamata ad un altro sottoprogramma; tutte le altre aree locali, evidentemente corrispondenti a sottoprogrammi che in quel momento non sono nè in esecuzione effettiva nè sospesi, sono aree *morte*, cioè che non sono in alcun modo interessate, in quell'istante, dal processo di calcolo.

2.13 Parametri formali di un sottoprogramma passati come argomenti ad un altro sottoprogramma.

2.13.1 Introduzione.

I parametri di una procedura o funzione, essendo analoghi a variabili locali, possono a loro volta essere passati come argomenti effettivi nell'invocazione di un'altro sottoprogramma, come si vede anche nell'esempio precedente.

Naturalmente si possono avere parametri per valore di un sottoprogramma passati per valore o per indirizzo ad un altro sottoprogramma, parametri per indirizzo passati per valore, ecc. In ognuno di questi casi il comportamento dell'esecutore Pascal, stabilito dalla definizione del linguaggio, è quello che intuitivamente ci si aspetta, che è anche l'unico ragionevolmente possibile in un linguaggio "fortemente tipato" come il Pascal.

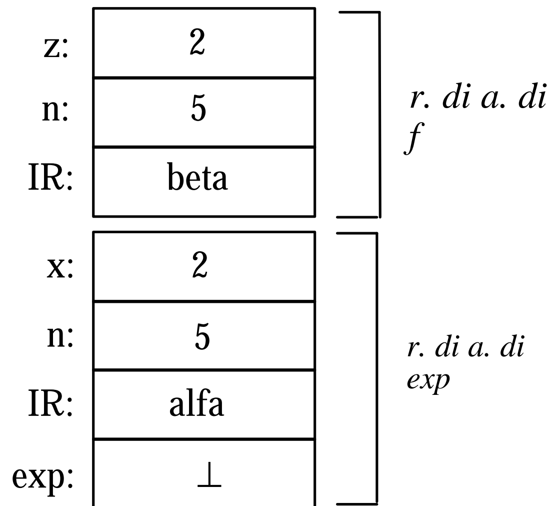
Un caso è già illustrato nell'esempio della sezione precedente. Esaminiamo tuttavia uno per uno i quattro casi possibili per capire bene come tale comportamento sia realizzato.

2.13.2 Parametro per valore passato come argomento per valore.

Questo è il caso piú semplice e non presenta alcuna difficoltà: il valore viene semplicemente copiato.
Esempio:

```
...

function exp(x: real; n: integer): real;
begin
  ...
end;

function f(z: real; n: integer): real;
begin
  f:= exp(z,n) + fact(n)
                 ∈ alfa:∨
end;

begin
  writeln(f(2,5));
  ∈ beta:∨
  ...
end.
```

I due parametri per valore della funzione *f* vengono passati a loro volta per valore alla funzione *exp*.
All'istante d'inizio dell'esecuzione della procedura *exp* lo stato della memoria sarà:

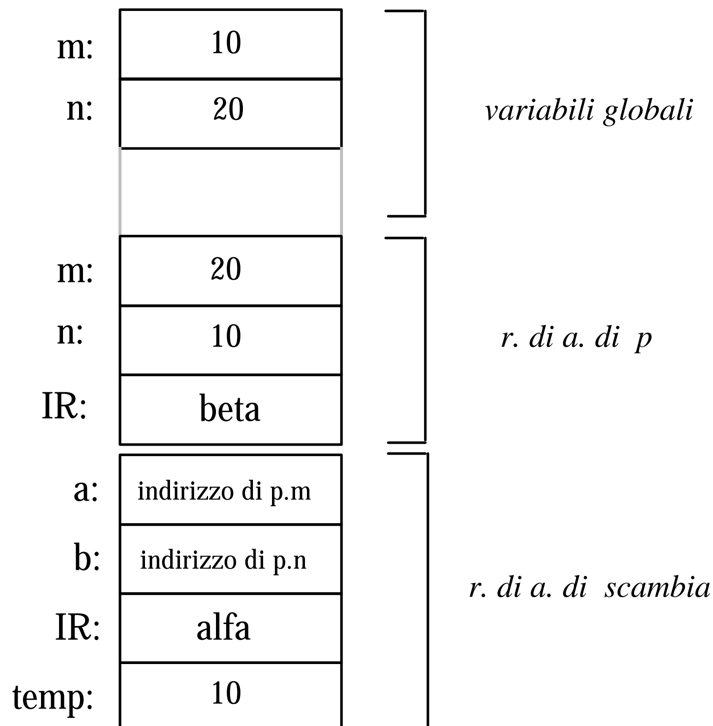


2.13.3 Parametro per valore passato come argomento per riferimento.

Anche questo caso non presenta difficoltà: nelle cella-parametro del r. di a. del sottoprogramma chiamato viene memorizzato l'indirizzo del parametro formale del sottoprogramma chiamante.
Esempio:

```
var m,n: integer;
...
procedure scambia(var a,b: integer);
var temp: integer;
begin
  temp:= a;
  a:= b;
  b:= temp
end;

procedure p(m,n: integer);
begin
  if ... then scambia(m,n) else ...
{alfa:}
end;

begin
  ...
  p(m,n);
{beta:} ...
  ...
end.
```

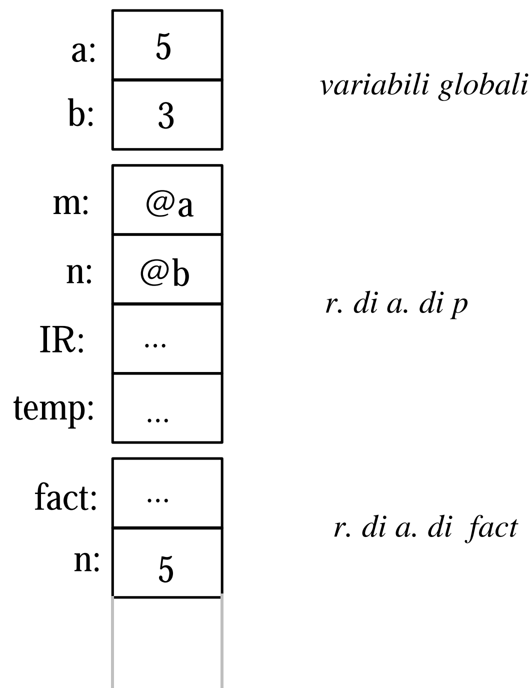
Assumendo che le variabili globali *m* ed *n* contengano i valori 10 ed 20, nell'istante immediatamente precedente all'esecuzione dell'*end* della procedura *scambia* lo stato della memoria è quello rappresentato nel disegno sottostante.



2.13.4 Parametro formale per riferimento passato come argomento per valore.

Come ci si aspetta, nella cella-parametro del r. di a. del sottoprogramma chiamato sarà memorizzato l'argomento effettivo del chiamante, non il contenuto della cella parametro formale del chiamante: infatti il chiamato vuol ricevere un valore, non un indirizzo.
Esempio:

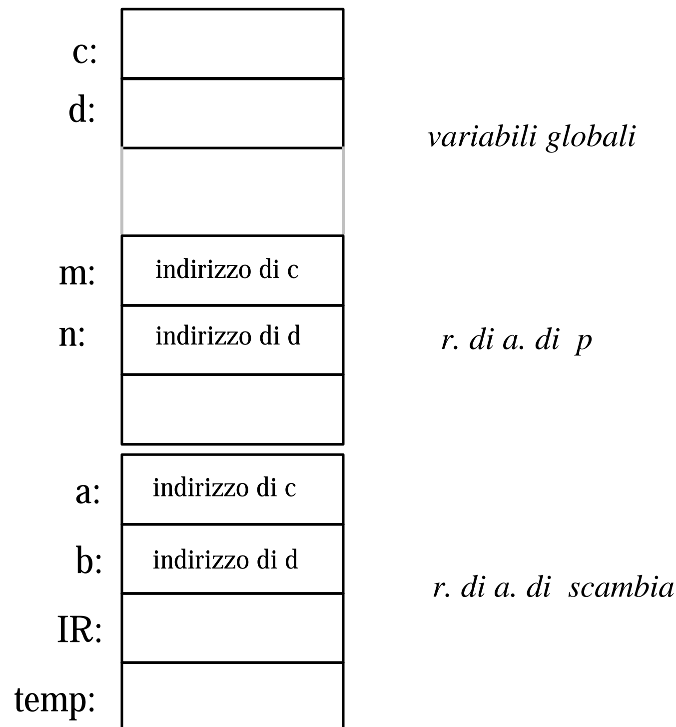
```
var a,b: integer;

  function fact(n; integer): integer;
  var ...
  begin
  ...
  end;

  procedure p(var m,n: integer);
  var temp: integer;
  begin
    temp:= fact(m);
    m:= n;
    n:= temp
  end;

begin
...
  p(a,b);
...
```



2.13.5 Parametro formale per riferimento passato come argomento per riferimento.

Questo è l'unico caso sulla cui realizzazione il lettore potrebbe avere qualche dubbio; tuttavia, passare per riferimento - cioè per indirizzo - qualcosa di cui si ha già soltanto l'indirizzo, vuol dire passare tale indirizzo: nelle celle-parametri-formali del sottoprogramma chiamato vengono copiati gl'indirizzi contenuti nelle celle-parametri-formali del chiamante. Il passaggio di un parametro formale per riferimento come argomento effettivo per riferimento ad un altro sottoprogramma è perciò realizzato esattamente come un passaggio per valore, con una semplice copia! In effetti, esso equivale, per cosí dire, a un passaggio di indirizzi per valore!
Esempio.

```
var c,d: integer;

  procedure scambia(var a,b: integer);
  ...

  procedure p(var m,n: integer);
  begin
    ... scambia(m,n) ...
  end;

begin
  ...
  p(c,d);
  ...
```



2.14 La primitiva *exit*.

La primitiva *exit*, che non esiste nè in Pascal Standard, nè nelle versioni precedenti del TurboPascal, è stata verosimilmente introdotta a partire dalla versione 7 per ovviare alla già notata mancanza, in Pascal, di un'istruzione analoga alla *return* del C. Se eseguita all'interno di un sottoprogramma (procedura o funzione), la *exit* termina immediatamente l'esecuzione del sottoprogramma e restituisce il controllo al chiamante.

L'istruzione *exit* eseguita nel programma principale fa terminare immediatamente l'esecuzione del programma (ed è quindi equivalente ad una *halt*).

Nota Bene: se il sottoprogramma è stato chiamato da un altro sottoprogramma, l'esecuzione dell'istruzione *exit* fa terminare soltanto il sottoprogramma chiamato, e non il chiamante.

Nel corpo di una funzione l'esecuzione di una *exit* subito dopo l'assegnazione di un risultato al nome della funzione permette di simulare l'istruzione di *return(risultato)* dei linguaggi moderni come il C o Modula o Java.

Attenzione: non confondere la *exit* con la *break* !

Capitolo 3.
Tipi.

3.1 Numeri reali.

Come abbiamo visto nel Capitolo 1, in Pascal vi sono diversi tipi primitivi di valori: oltre agl'interi e ai booleani, abbiamo già accennato ai caratteri e alle stringhe. Vi sono poi i numeri reali, o numeri con la virgola; naturalmente un numero reale è, dal punto di vista matematico, un ente infinito, rappresentato ad esempio da una sequenza infinita di cifre dopo la virgola, e in quanto tale non può essere contenuto nella memoria finita del calcolatore. In effetti i numeri con la virgola sono rappresentati in memoria, come si studia nei corsi di Architetture, da un numero finito di cifre significative e dall'esponente della base della numerazione.

In base a ciò il neofita potrebbe pensare che i cosiddetti numeri reali dei linguaggi di programmazione - avendo solo un numero finito di cifre - siano in realtà semplicemente numeri razionali, equivalenti quindi a frazioni. Invece non è proprio cosí, perchè essendo i razionali - a differenza dei reali - degli enti finiti, è implicito - dal punto di vista matematico - che le operazioni su di essi debbano sempre, come per gl'interi, essere "esatte", e non si possano operare approssimazioni.

Al contrario, proprio perchè quando lavoriamo in modo non simbolico (anche se eventualmente con carta e matita!) con i reali abbiamo per forza sempre a che fare con valori approssimati, è perfettamente lecito, nelle operazioni con essi, effettuare degli arrotondamenti per non superare il numero di cifre consentito (beninteso sapendo che in tal modo si introducono errori, ecc., come verrà ampiamente studiato nei corsi di calcolo numerico).

I reali del Pascal o di qualunque altro linguaggio di programmazione sono in questo senso davvero numeri reali e non numeri razionali, perchè i risultati delle operazioni su di essi vengono automaticamente arrotondati.

3.2 Tipi numerici diversi e conversioni.

In matematica i numeri interi possono essere considerati dei particolari numeri reali, ed è perfettamente possibile, ad esempio, sommare un intero con un numero con la virgola, ad es. *5 + 3.141*.

Questo è possibile anche in Pascal e nei linguaggi dello stesso genere; in essi, però, il valore *5* di tipo *integer* è un ente diverso dal valore *5* di tipo *real* (a volte indicato, per distinguerlo dal precedente, come 5.0), anche concretamente: il *5 real* ha nel calcolatore una rappresentazione diversa dal *5 integer*, e le operazioni primitive che operano sui reali (somma, sottrazione, ecc.) sono diverse da quelle che operano sugl'interi (perchè ad es. approssimano automaticamente, come si è detto).

Ciò che in realtà succede, nel calcolo dell'espressione di cui sopra, è che l'intero *5* viene dapprima automaticamente convertito nel reale *5*, e poi viene effettuata la somma fra due reali.

Abbiamo anche visto che ogni variabile Pascal deve essere dichiarata di un certo tipo, e può contenere soltanto valori di quel tipo. Anche qui, se `r` è una variabile dichiarata di tipo *real*, è perfettamente possibile effettuare un'assegnazione `r:= 5`; o anche, se `n` è una variabile di tipo integer, `r:= n`. In entrambi i casi il valore di tipo intero viene prima

convertito nell'omologo valore di tipo reale, e poi depositato nella variabile. Viceversa, non è possibile memorizzare un reale in una variabile di tipo intero.

In quasi tutti i linguaggi moderni, poi, vi sono piú tipi di interi e piú tipi di reali, che differiscono per l'ampiezza dell'intervallo dei valori e - nel caso dei reali - del numero delle cifre significative. Cosí in TurboPascal oltre al tipo *integer*, i cui elementi sono gl'interi fra *-32768* e *+32767* compresi, abbiamo il tipo *longint* ossia degl'interi lunghi, che comprende gl'interi fra *-2.147.483.648* e *2.147.483.647*, ecc. Anche qui l'*integer 5* è un ente distinto dal *longint 5* (che in alcuni linguaggi può essere indicato esplicitamente come *5L*) perchè quest'ultimo è rappresentato da 32 anzichè 16 bit. Sono tuttavia anche qui possibili le ovvie conversioni.

Per una illustrazione accurata dei diversi tipi numerici del TurboPascal e delle regole di conversione automatica fra di essi si rimanda ai manuali del linguaggio.

3.3 Tipi enumerati.

<vedi testo Pascal laboratorio di programmazione>
<inserire materiale>

3.4 Tipi strutturati.

Le nozioni di valore, variabile e tipo sono fra quelle fondamentali dei linguaggi di programmazione del genere del Pascal. Riassumendo, i valori sono ripartiti in diversi insiemi distinti (o meglio, strutture algebriche distinte) che sono i loro tipi; anche le variabili devono essere dichiarate di un dato tipo, e possono poi contenere solo valori di quel tipo. Abbiamo quindi tipi di valori e tipi di variabili: ma le due nozioni - per i tipi semplici visti finora - coincidono nel senso che i tipi attribuibili alle variabili sono esattamente i tipi di valore.

Nel caso dei tipi cosiddetti strutturati, che ora esaminiamo, la situazione è invece un po' piú complessa.

3.4.1 Tipi di valore composti.

Oltre ai valori primitivi semplici, come quelli considerati finora, tanto in matematica quanto in informatica è spesso indispensabile considerare valori "composti", ossia enti costituiti da aggregati di enti (piú) semplici, come sequenze, n-uple, insiemi, ecc.

Se restiamo fedeli al principio che un tipo è costituito da un insieme di elementi con delle operazioni su di esso, un tipo composto o strutturato è allora un tipo tale che:

- i suoi elementi o valori sono degli "aggregati" di elementi piú semplici;
- vi sono operazioni per <u>costruire</u> tali valori strutturati a partire da valori piú semplici, e viceversa operazioni per <u>selezionare</u> ed estrarre componenti di valori composti.

In Pascal, come vedremo, vi è un'aderenza molto imperfetta a tale principio; tuttavia, per capire meglio i concetti in gioco e quindi indirettamente anche il Pascal, immaginiamo per un momento una sua estensione che vi aderisca perfettamente, in cui sarà quindi fissata una sintassi per rappresentare i suddetti valori composti e le loro operazioni.

Ad esempio, possiamo immaginare che nel nostro linguaggio di programmazione vi sia un tipo primitivo *coppia ordinata di integer*, che magari si chiami *intpair*; immaginiamo anche che i valori di tale tipo si rappresentino, come in matematica, scrivendone fra parentesi tonde i componenti separati da una virgola; allora valori di tale nuovo tipo sarebbero ad esempio *(5,3)*, *(2,2)*, *(12, 17)*, ecc., e le parentesi con la virgola sono (o meglio, denotano) l'operazione di costruzione.

Se avessimo tale tipo primitivo, potremmo definire delle variabili di tipo coppia ed effettuare assegnazioni, ad esempio:

```
(* ATTENZIONE: NON È PASCAL! *)
var p: intpair;
    m,n: integer;
begin
  p:= (5,3);
  ...
  p:= (m,n); ... p:= (n,n); ...
...
```

Dobbiamo anche immaginare, come detto sopra, che vi siano due operazioni primitive di selezione che restituiscono rispettivamente il primo ed il secondo elemento della coppia, supponiamo si chiamino *fst* (per *first*) e *snd* (per *second*). Potremmo allora scrivere, facendo riferimento alle dichiarazioni precedenti:

```
(* ATTENZIONE: NON È PASCAL! *)
...
m:= fst(p); ... n:= snd(p); ...
...
```

con gli ovvi significati.

3.4.2 Tipi di variabile composti

Una variabile di un dato tipo composto è un contenitore di valori dell'omonimo tipo; essa è quindi realizzabile in modo naturale come un aggregato di sottocontenitori piú semplici. Sarebbe allora molto comodo poter modificare il contenuto di ciascun sottocontenitore indipendentemente dagli altri. Tornando all'esempio immaginario precedente, se *p* è una variabile di tipo coppia, vorremmo poter modificare il suo secondo elemento lasciando immutato il primo; possiamo allora immaginare che *snd(p)* quando si trova a sinistra del simbolo di assegnazione denoti la seconda cella componente la coppia, e scriviamo:

```
(* ATTENZIONE: NON È PASCAL! *)
snd(p):= 7;
```

In Pascal e nei linguaggi di programmazione dello stesso genere si possono definire tipi composti di variabile, cioè tipi di contenitori composti, i cui componenti sono separatamente modificabili (anche se non esattamente con la sintassi del nostro esempio immaginario). Si tratta dei tipi *array* e dei tipi *record*. È invece un po' discutibile, come vedremo, affermare che in Pascal esistano dei corrispondenti valori composti.

3.4.3 Tipi strutturati in Pascal: i tipi *array*.

Una variabile di un tipo *array di tipo T* è:

- un contenitore costituito da una sequenza di lunghezza fissata di contenitori di tipo *T*, cioè un contenitore composto da un numero fissato di contenitori di tipo *T* etichettati da numeri interi consecutivi (o indici);

- tale che la lettura e modifica dell' i-esimo componente può avvenire con *i* calcolabile dinamicamente, e in tempo costante indipendente da *i*.

Esempio:

```
var a,b: array[1..20] of integer;
    i,k: integer;
begin
  readln(k);
  a[k]:= 5;
  for i:= 1 to 20 do readln(b[i]);
  ...
```

In TurboPascal si può anche assegnare un intero *array* ad un altro *array*, purchè dello stesso tipo:

```
a:= b;
```

in accordo con l'idea che il contenuto di `b` è un "valore composto" che viene ricopiato "in blocco" in `a`. In realtà dal punto di vista realizzativo non ci sono magie; l'istruzione precedente equivale al ciclo:

```
for i:= 1 to 20 do a[i]:= b[i];
```

ed anche dal punto di vista del tempo di calcolo il fatto che si tratti di una sola istruzione non inganni: il tempo necessario per copiare un *array* di *n* elementi è sempre proporzionale a *n*.

Ad un tipo strutturato può essere dato un nuovo nome, che può essere poi utilizzato nel resto del programma:

```
type vettore = array[1..20] of integer;
var a: vettore;
    k: integer;
begin
  readln(k);
  a[k]:= 5;
...
```

Si noti bene che l'elaborazione da parte dell'esecutore Pascal della dichiarazione o definizione di tipo non alloca alcuno spazio di memoria, ma semplicemente associa al nome *vettore* un tipo di variabile, cioè - per cosí dire - uno "stampo" per costruire contenitori. Lo spazio (di 20 celle di memoria) viene creato soltanto con l'elaborazione della dichiarazione della variabile `a`.

In un programma Pascal non si possono esprimere valori immediati di tipo *array*, e le funzioni non possono restituire *array*, ma soltanto valori di tipi semplici. In questo senso i valori composti sono, in Pascal, "cittadini di seconda classe" che non hanno una piena esistenza.

In TurboPascal, come in C, una forma ristretta di valori immediati di tipo *array* esiste, limitatamente all'inizializzazione delle variabili. Ad esempio:

```
const v: vettore = (1,2,3,4,5,6,7,8,9,10,10,9,8,7,6,5,4,3,2,1);
```

(purtroppo in TurboPascal le *variabili inizializzate* si chiamano *costanti*, ed è quindi preferibile non usarle come variabili!).

Gli *array* unidimensionali, come quelli degli esempi precedenti, vengono spesso chiamati *vettori*. Oltre ad essi si possono definire *array* bidimensionali, o *matrici*, che sono semplicemente *array* di *array*, e in generale *array* multidimensionali.

Per maggiori dettagli si vedano i manuali del linguaggio.

3.4.4 Tipi strutturati in Pascal: i tipi *record*.

Una variabile di un *tipo record* può essere considerata come un insieme di sottovariabili - o campi - di tipi non necessariamente uguali, dotato di un nome collettivo, ed entro certi limiti trattabile come un tutto unico. Piú precisamente, un tipo record è:

- un tipo di variabile composto da un numero fisso di "contenitori" o *campi* (di tipi non necessariamente uguali) etichettati da nomi (tutte le variabili di uno stesso tipo *record* hanno perciò gli stessi campi con gli stessi nomi);

- tale che l'accesso ad un componente può avvenire (a differenza che per gli *array*) soltanto per un componente di etichetta staticamente nota, ma - come per gli *array* - in tempo costante indipendente dall'etichetta.

Esempio:

```
var a,b: record
           nome: string; matricola: integer; incorso: boolean
         end;

begin
  a.nome:= 'Temistocle Rossi';
  a.matricola:= 97153;
  a.incorso:= true;
  b:= a;
  writeln(b.matricola);
  ...
```

oppure, dando un nome al tipo:

```
type studente = record
                  nome: string;
                  matricola: integer;
                  incorso: boolean
                end;
var a,b: studente;
...
```

Il tipo "coppia d'interi" dell'esempio immaginario può essere realizzato come *record*:

```
type intpair = record fst, snd: integer end;
var p: pair;
begin
  p.fst:= 5; p.snd:= 3;
  ...
```

Per i valori di tipo *record* valgono regole sintattiche e considerazioni analoghe a quelle per gli *array*.
<inserire materiale>
Per maggiori dettagli sui tipi *record* si vedano i manuali del linguaggio.

3.4.5 Equivalenza fra tipi strutturati.

L'attribuire un nome ad un tipo per mezzo di una dichiarazione di tipo è non solo comodo, ma spesso indispensabile. Infatti in (Turbo)Pascal ciò che distingue un tipo da un altro non è la struttura (cioè la forma) del tipo, ma il nome. Ossia, se con due dichiarazioni di tipo vengono definiti con due nomi (necessariamente) distinti due tipi strutturalmente identici, essi vengono considerati dall'esecutore Pascal come tipi distinti a tutti gli effetti. Ad esempio se si compila il programma seguente:

```
type vettore = array[1..20] of integer;
     vector  = array[1..20] of integer;
var a: vettore;
    b: vector;
    ...
begin
  ...
  a:= b;
  ...
```

si produce un errore di tipo nell'istruzione di assegnazione, perchè le variabili `a` e `b` sono considerate di tipi diversi e quindi non compatibili.

Una dichiarazione di variabile di un tipo strutturato *anonimo* viene considerata equivalente ad una dichiarazione di tipo con nome "segreto" seguita da una dichiarazione di variabile di quel tipo; variabili dichiarate separatamente di tipi anonimi strutturalmente identici sono perciò considerate di tipi distinti non compatibili. Ad esempio, con le dichiarazioni:

```
var a,b: array[1..20] of real;
    c: array[1..20] of real;
```

le variabili `a` e `b` risultano dello stesso tipo, mentre la variabile `c` viene considerata di un tipo diverso.

Una conseguenza di tale *equivalenza per nome* è che per poter scrivere dei sottoprogrammi con parametri di tipi strutturati bisogna nel programma principale assegnare dei nomi espliciti a tali tipi per mezzo di dichiarazioni di tipo (con l'eccezione, in TurboPascal, degli *array aperti*, che esamineremo nel prossimo capitolo).

Se infatti scrivessimo una procedura della forma:

```
(************ ATTENZIONE: NON È PASCAL *************)
procedure p(var a: array[1..20] of integer);
...
```

essa non potrebbe mai essere invocata correttamente, perchè il tipo del suo parametro formale sarebbe un tipo "segreto" di cui non si possono dichiarare variabili.
In realtà, proprio per questo motivo la sintassi del Pascal vieta dichiarazioni di sottoprogrammi come quella sopra riportata, poichè richiede che i tipi dei parametri

formali possno essere soltanto identificatori (cioè nomi) di tipo (con la già citata eccezione degli *array aperti* in TurboPascal).

3.5 Passaggio di parametri di tipi strutturati.

Procedure e funzioni possono avere parametri di tipi strutturati. La loro semantica in Pascal è coerente con l'idea che i valori composti esistano davvero: il passaggio per valore equivale quindi, come per i tipi semplici, a creare nel record di attivazione del sottoprogramma una copia di tutto il valore composto. Il passaggio per riferimento equivale invece a memorizzare soltanto l'indirizzo (cioè, dal punto di vista fisico, l'indirizzo d'inizio) della struttura composta. Nel caso di una struttura "grande", come può essere un *array*, il passaggio per valore richiede perciò la copia di tutti gli elementi dell'*array*, il che richiede quindi un tempo e un'occupazione aggiuntiva di spazio proporzionali alla dimensione dell'*array*.

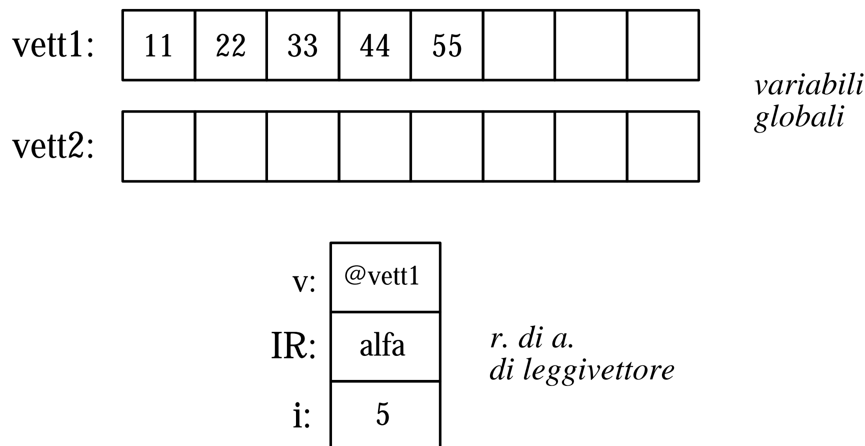
Per tale ragione è bene che i parametri di tipo *array* siano sempre per riferimento, anche quando sono parametri puramente di input, a meno che per qualche ragione non si voglia davvero disporre di una copia locale da manipolare lasciando invariato il parametro effettivo.

In C gli *array* sono automaticamente passati per riferimento. Nei moderni linguaggi ad oggetti sono passati per valore solo i parametri dei tipi primitivi semplici come interi, booleani, ecc., mentre tutti gli altri sono automaticamente per riferimento.

Esempio. La procedura *leggivettore* richiede da tastiera l'immissione di *n* elementi (dove *n* è la dimensione del vettore) e li memorizza nella variabile passatale come parametro; la procedura *scrivivettore* visualizza il contenuto del vettore. Il programma principale permette di provare le due procedure.

```
const n = ...
type vettore = array[1..n] of integer;

var vett1, vett2: vettore;

  procedure leggivettore(var v: vettore);
  var i: integer;
  begin
    writeln('immetti ',n,' interi');
    for i:= 1 to n do read(v[i]);
    readln
  end;

  procedure scrivivettore(var v: vettore);
  var i: integer;
  begin
    for i:= 1 to n do write(i,' ');
    writeln
  end;

begin
  leggivettore(vett1);
{alfa:}
  vett2:= vett1;
  scrivivettore(vett2)
end.
```

Supponendo *n = 8*, e che gli elementi immessi siano *11 22 33 44 55 66 77 88*, lo stato della memoria dopo la *read* del quinto elemento sarà:



Nella figura non abbiamo disegnato una cella di memoria per la costante *n*, perchè in Pascal le costanti (non tipate) non occupano memoria: esse vengono risolte al tempo di compilazione, cioè come se il valore di *n* venisse scritto al posto di ogni occorrenza di *n* nel testo del programma prima dell'esecuzione.

Nel programma seguente, supponendo che la procedura *ordina* effettui l'ordinamento del vettore argomento, la procedura *scriviordinato* visualizza gli elementi del vettore in ordine di grandezza, senza tuttavia ordinare il vettore stesso (ma solo una sua copia locale).

```
...
var vett: vettore;

  procedure scrivivettore(var v: vettore);
  ...
  end;

  procedure ordina(var v: vettore);
  {ordina il vettore v}
  end;

  procedure scriviordinato(v: vettore);
  begin
    ordina(v);
    scrivivettore(v)
{beta:}
  end;

begin
  ...
  scriviordinato(vett);
{alfa:}
  scrivivettore(vett)
end.
```

Assumendo che nel vettore `vett` si trovi memorizzata la sequenza 33 44 88 11 55 22 77 66, lo stato della memoria subito dopo la scrittura del terzo elemento, durante la chiamata della procedura *scrivivettore* all'interno della *scriviordinato*, è raffigurato nella figura seguente (dove, come in quella che precede, i vettori sono per ragioni di spazio disegnati "in

orizzontale" invece che "in verticale" come sarebbe piú coerente con i criteri generali della rappresentazione).

vett:	33	44	88	11	55	22	77	66	*variabili globali*

v:	11	22	33	44	55	66	77	88	*r. di a. di scriviordinato*
IR:	alfa								

v:	@(scriviordinato.v)	
IR:	beta	*r. di a. di scrivivettore*
i:	3	

Il vettore globale non viene modificato dalla procedura di ordinamento, e all'uscita dalla procedura *scriviordinato* la copia locale ordinata del vettore viene distrutta. La successiva invocazione della procedura scrivivettore nel programma principale scriverà quindi la sequenza originale non ordinata.

L'*indirizzo di un vettore* coincide fisicamente con l'indirizzo del suo primo elemento; tuttavia da un punto di vista astratto essi sono due entità distinte, come il numero civico di un complesso residenziale e gli "interni" di tale numero civico; considereremo quindi @*v* e @*v[1]* come indirizzi distinti; ad esempio, nella chiamata di procedura `scambia(v[1],v[2])`, che scambia fra di loro i primi due elementi del vettore `v`, nei parametri formali di `scambia` vengono memorizzati gl'indirizzi di *v[1]* e di *v[2]*, che indichiamo rispettivamente con @*v[1]* e @*v[2]*, entrambi ben distinti concettualmente da @*v*.

Per i parametri di tipo *record* si possono fare considerazioni analoghe a quelle sugli *array*, eccetto il fatto che - trattandosi perlopiú di strutture "piccole" rispetto agli *array* - per i parametri di input è di solito consigliabile il passaggio per valore.

Capitolo 4
Correttezza e complessità.

4.1 Problemi di programmazione.

Come abbiamo visto dai primi esempi, un problema di programmazione - o problema algoritmico - è schematicamente costituito dalla specifica di un insieme di condizioni e di relazioni con i dati in ingresso (o i dati iniziali in memoria centrale) cui si vuole che i dati in uscita (o i dati finali in memoria centrale) soddisfino, se i dati in ingresso (o iniziali) soddisfano a certe condizioni preliminari.
Un problema di programmazione può quindi in generale essere espresso specificando una condizione iniziale (oppure una condizione sui dati in ingresso) e una condizione finale (oppure una condizione sui dati in uscita).

Esempi:

- un problema di ordinamento come relazione fra ingresso e uscita:
  i *dati in ingresso* da tastiera devono essere costituiti da una sequenza, terminata da CTRL/Z, di stringhe (cioè sequenze di caratteri) di lunghezza arbitraria, separate fra loro in qualche modo conveniente;
  l'*uscita* sullo schermo deve essere costituita dalla sequenza delle stringhe di ingresso ordinata lessicograficamente in ordine crescente;
  (naturalmente, è piuttosto raro che ci si trovi a dover risolvere un problema di questo tipo esclusivamente come relazione fra input da tastiera e output su schermo; sarà piú frequente una formulazione come quella che segue);

- un problema di ordinamento di un file:
  i *dati in ingresso* da disco o altro supporto di memoria di massa sono costituiti da un file di stringhe (le cui lunghezze sono tutte inferiori ad una data lunghezza massima "abbastanza grande");
  la *condizione finale* è che tale file risulti ordinato lessicograficamente in ordine crescente;
  (per risolvere tale problema potrà essere necessario ricondursi a un problema di ordinamento nella memoria di lavoro, come il seguente);

- un problema di ordinamento in memoria primaria:
  i *dati iniziali* sono costituiti da una sequenza (ad esempio - come vedremo - un vettore, oppure una lista concatenata) di stringhe in memoria centrale;
  la *condizione finale* è che tale sequenza risulti ordinata lessicograficamente in ordine crescente;

- problemi di controllo di processi:
  i *dati in ingresso* sono costituiti da eventi che si possono verificare su una varietà di periferiche di input (sensori, sonde, tastiere, microfoni, ecc.) e che soddisfano a certe condizioni (anche molto complesse, e con vincoli temporali) imposte dalla realtà naturale o artificiale (ad esempio che un certo sensore di temperatura non registrerà mai una temperatura superiore a 200 gradi centigradi, ecc.);

i *dati in uscita* sono a loro volta costituiti da eventi che si possono verificare su una varietà di periferiche di output come attuatori, comandi di apparecchiature, schermi video, stampanti, altoparlanti, ecc.), i quali devono naturalmente soddisfare a certe condizioni e relazioni con l'input anche molto complesse, spesso includenti dei vincoli temporali.

Trascurando l'input-output, la specifica di un problema di programmazione è allora data formalmente da una coppia di proposizioni: una proposizione CI che esprime la condizione cui si assume soddisfi lo stato iniziale della memoria-dati, ed una proposizione CF esprimente la condizione cui si vuole soddisfi lo stato finale.

Le proposizioni CI e CF possono essere inserite come commenti all'inizio e alla fine del programma o della porzione di programma, a scopo di documentazione. La specifica di un problema di programmazione può allora suggestivamente vedersi come un sorgente Pascal costituito soltanto da una coppia di commenti:

```
{CI: ...}

{CF: ...}
```

Risolvere il problema vuol dire riempire lo spazio fra i due commenti con una porzione di programma la quale, se eseguita a partire da un qualunque stato che soddisfi la condizione CI, termina in uno stato che soddisfa la condizione CF.

```
{CI: ...}
  Prog
{CF: ...}
```

Ad esempio, il problema dell'elevamento di un qualunque reale X (non nullo) alla potenza di un qualunque esponente N intero non negativo può essere specificato nel modo seguente (dove invece del simbolo $\wedge$, che non è un carattere ASCII, usiamo semplicemente la virgola):

```
var x, ris: real;
    n: integer;
{CI: x = X, n = N, X ≠ 0, N >= 0}
{CF: ris = X^N}
```

(dove la condizione X≠0 è stata posta per semplicità per evitare il caso 0^0 che, com'è noto, è di valore indefinito).

Tale uso dei commenti può essere esteso a tutte le porzioni di programma che interessano, fino al limite della singola istruzione; ad esempio il problema di ottenere nella variabile `a` il quoziente della divisione intera del contenuto iniziale di `a` per il contenuto di `b`, con l'istruzione che lo risolve, può essere scritto:

```
{a=A, b≠0}
a:= a div b
{a = A div b}
```

Ciò equivale a concepire il processo di risoluzione di un problema di programmazione - cioè di scrittura di un programma - come un processo di scomposizione del problema in sottoproblemi, fino ai sottoproblemi banali risolubili con una singola istruzione.

I programmi commentati in questo modo semiformale vengono detti *programmi annotati*. Naturalmente, nell'uso corrente si annotano solo gli stati intermedi piú importanti.

Si noti che proposizioni come quelle di cui sopra fanno riferimento implicito allo stato della memoria; cioè la loro verità o falsità dipende dal particolare stato, che però non è nominato esplicitamente nella proposizione stessa. Nell'ambito della programmazione,

proposizioni di questo genere vengono dette *asserzioni*, per distinguerle dalle proposizioni completamente esplicite (come quelle della matematica ordinaria).

Naturalmente ogni asserzione può essere tradotta, nominando esplicitamente lo stato, in una proposizione ordinaria; ad esempio l'asserzione *{a=A, b„ 0}* non è altro che una forma abbreviata della proposizione "nello stato *s* è *a=A* e *b„ 0*" (che è una proposizione "aperta", cioè la cui verità o falsità dipende dal particolare *s*). Analogamente, una porzione di programma compresa fra due asserzioni *{C1} Prog {C2}*può essere letta come la pro-posizione: "se nello stato *s* immediatamente precedente l'esecuzione di *Prog* vale *C1*, allora nello stato *s'* immediatamente successivo al termine dell'esecuzione di *Prog* vale *C2*". In realtà, per il significato che daremo ai programmi annotati, occorrerebbe aggiungere la condizione "se l'esecuzione di *Prog* termina"; ma su questo torneremo piú avanti.

4.2 Il principio di induzione matematica (semplice).

Fra i principi di base che possono aiutare a risolvere correttamente un problema di programmazione vi sono i principi di induzione matematica. Il piú semplice di essi, il principio di induzione matematica semplice, può essere espresso in linguaggio naturale nel modo seguente.

Se una proprietà vale per il numero 0, e se quando vale per un numero naturale k allora vale anche per il numero k+1, allora essa vale per qualsiasi numero naturale.

Con una formula logica: $P(0) \land \forall k . (P(k) \Rightarrow P(k+1)) \Rightarrow \forall n.P(n)$
con k ed n naturali.

Il principio di induzione può essere espresso in modo piú operativo come regola per effettuare dimostrazioni (regola di inferenza):

Se si dimostra che una proprietà P vale per 0, e se si riesce a dimostrare che quando vale per un generico k allora vale anche per k+1, allora si è dimostrato che P vale per qualunque naturale.

Spostandoci dal piano dei fatti (proprietà che valgono o non valgono) al piano linguistico (proposizioni vere e false), possiamo dire che il principio di induzione serve a dimostrare proposizioni della forma $\forall n.P(n)$, dove $P(n)$ è una proposizione contenente la variabile n (variabile nel senso della logica!), ad esempio:

"la somma dei numeri naturali compresi fra 0 ed n (inclusi) è uguale a n(n+1)/2",

o, in notazione matematica informale:

$0+1+2+ \dots + n = n(n+1)/2$

Una dimostrazione per induzione ha pertanto la forma seguente:

BASE DELL'INDUZIONE: $P(0)$ (che naturalmente è la proposizione che si ottiene dalla proposizione P sostituendovi n con 0).

DIMOSTRAZIONE DELLA BASE:
Si dimostra $P(0)$, cioè che la proprietà (espressa dalla proposizione) P vale per 0. (Spesso, anche se non sempre, tale dimostrazione è banale).

PASSO D'INDUZIONE:
Si assume come ipotesi che la proprietà P valga per (un generico) k, e utilizzando tale ipotesi si dimostra la tesi che P vale per $k+1$.
Cioè:

IPOTESI INDUTTIVA: $P(k)$
cioè la proposizione che si ottiene dalla proposizione P sostituendovi n con k.

TESI INDUTTIVA: $P(k+1)$
cioè la proposizione che si ottiene da P sostituendovi n con $k+1$.

DIMOSTRAZIONE DEL PASSO:
Si dimostra la tesi induttiva utilizzando in qualche passaggio logico l'ipotesi induttiva.

Esempio: dimostrazione della formula per la somma dei primi *n* numeri naturali, riportata sopra.

BASE: La somma dei naturali compresi fra 0 e 0 è uguale a 0(0+1)/2.

Dimostrazione della base.
Ovvia: in questo caso la somma è zero, ma anche 0(0+1)/2 è uguale a zero.

PASSO:

Ipotesi: $0 + 1 + ... + k = k(k+1)/2$

Tesi: $0 + 1 + ... + (k+1) = (k+1)((k+1)+1)/2$

cioè

$0 + 1 + ... + (k+1) = (k+1)(k+2)/2$

Dimostrazione del passo.
$0 + 1 + ... + (k+1) =$
$(0 + 1 + ... + k) + (k+1) =$ *per ip.induttiva*
$k(k+1)/2 + (k+1) =$
$k(k+1)/2 + 2(k+1)/2 = (k+2)(k+1)/2$
Come si vede, per passare dalla seconda alla terza riga si è applicata l'ipotesi induttiva.

La variabile logica *n* viene chiamata parametro dell'induzione; di solito, quando si scrive una dimostrazione per induzione, l'ipotesi induttiva viene scritta senza sostituire la variabile *n* con un altro nome *k*; anzi, poichè in tal modo l'ipotesi induttiva diventa formalmente identica al teorema da dimostrare (a parte l'assenza del quantificatore universale), non la si scrive nemmeno, ma si scrive solo la tesi induttiva con $n+1$.

Poichè inoltre nel teorema da dimostrare il quantificatore universale può legittimamente essere lasciato implicito, una dimostrazione per induzione, scritta in modo conciso, ha la forma:

Vogliamo dimostrare per induzione $P(n)$.
Allora dimostriamo prima $P(0)$;
poi assumiamo che valga $P(n)$ e dimostriamo $P(n+1)$.

In questo modo la dimostrazione può sembrare un circolo vizioso: per dimostrare $P(n)$ bisogna assumere che valga $P(n)$! Abbiamo visto che non è cosí, l'apparente paradosso è generato soltanto dall'ambiguità e imprecisione del linguaggio informale; per un esame ed una riflessione piú approfondita su questa come su altre tecniche dimostrative si rimanda ai corsi di logica matematica.

Nota: se si riesce a dimostrare la tesi induttiva senza utilizzare l'ipotesi induttiva vuol dire che la dimostrazione è errata, oppure che non è una dimostrazione per induzione ma per altra via, cioè che non c'era bisogno di invocare il principio di induzione!

Il principio di induzione è facilmente generalizzabile ad una base maggiore di zero:

Se si dimostra che una proprietà P vale per un particolare numero naturale m, e se assumendo che valga per un generico naturale k   m si riesce a dimostrare che vale per k+1, allora si è dimostrato che P vale per qualunque numero naturale maggiore o uguale ad n.

In formula: $P(m) \wedge \forall k \geq m . (P(k) \Rightarrow P(k+1)) \Rightarrow \forall n \geq m . P(n)$

Piú avanti vedremo altre forme, leggermente diverse, del principio di induzione (induzione completa, induzione strutturale, ecc.).

Esercizio 1. Dimostrare per induzione la formula:

$$1^2 + 2^2 + \ldots + n^2 = (2n^3 + 3n^2 + n)/6$$

Esercizio 2. Dimostrare per induzione che (per $n \geq 1$) si ha:

$$1/(1 \cdot 2) + 1/(2 \cdot 3) + 1/(3 \cdot 4) + \ldots + 1/(n \cdot (n+1)) = n/(n+1)$$

4.3 Induzione e progettazione di cicli: introduzione.

4.3.1 Un esempio: la somma di una sequenza di numeri.

Riprendiamo in esame uno dei primi esempi di programma iterativo: la somma di una sequenza di numeri immessa da tastiera, di lunghezza arbitraria passata preliminarmente in input, oppure stabilita implicitamente terminando la sequenza con il fine-riga. Anche chi è completamente digiuno di programmazione non avrà avuto difficoltà, vedendo i relativi programmi, a capirne il funzionamento; tuttavia, se fosse stato posto di fronte ai problemi di cui quei programmi sono le soluzioni, cioè in generale al problema di scrivere un programma che faccia la somma di una sequenza di numeri, avrebbe probabilmente faticato un po' a trovare da solo la soluzione.

La difficoltà è quella di passare da una espressione matematica della forma:

(1) $$x_1 + x_2 + \ldots + x_n$$

all'istruzione Pascal:

```
somma:= somma + x
```

L'espressione (1), infatti, può sembrare non molto diversa da un'espressione della forma:

$$x_1 + x_2 + x_3 + x_4 + x_5$$

per calcolare la quale si può scrivere l'espressione direttamente in Pascal:

```
var x1,x2,x3,x4,x5,somma: integer;
begin
  readln(x1,x2,x3,x4,x5);
  somma:= x1+x2+x3+x4+x5;
  ...
end.
```

In realtà si tratta del passaggio dal finito all'infinito, nel senso che l'espressione (1) indica sí una somma finita, ma in cui il numero degli addendi può essere qualunque; il programma deve cioè essere in grado (in linea di principio) di calcolare la somma di una sequenza di lunghezza n per qualunque n, cioè per tutti gli n finiti, che sono in numero infinito.

Il valore finale dell'espressione (1) non può "apparire improvvisamente", come nel calcolo di un'espressione finita, ma deve essere calcolato un passo alla volta attraverso la ripetizione di una sequenza di istruzioni. Dobbiamo quindi avere una variabile in cui ad ogni passo vi è la somma parziale calcolata fino a quel momento, che ad ogni passo viene aggiornata addizionando un nuovo numero.

Dietro i tre puntini si nasconde il principio d'induzione, o meglio (come vedremo piú avanti) una *definizione induttiva*. Il ragionamento che porta dall'espressione (1) al programma Pascal è appunto di tipo induttivo: vogliamo che "alla fine" in una variabile `somma` ci sia la somma dei valori letti, allora:

- se nella variabile `somma` c'è la somma dei k valori letti fino a quel momento, dopo aver letto un altro valore x_{k+1} si deve venire ad avere in `somma` la somma dei $k+1$ valori letti; ciò si ottiene con le istruzioni: `read(x); somma:= somma+x`, che costituiscono il corpo del ciclo.

- "all'inizio" nella variabile `somma` ci deve essere la somma della sequenza parziale costituita dal solo primo valore (oppure la somma della sequenza vuota, che è 0); ciò si ottiene con l'istruzione `read(somma)` oppure `somma:=0`;

In altre parole, la dimostrazione che il programma funziona correttamente per qualsiasi lunghezza della sequenza è una dimostrazione di tipo induttivo, che può essere scritta nel modo indicato nel seguito.

Riportiamo prima il testo del programma in una delle sue versioni, con la specifica esatta del problema che esso risolve, ossia - in questo caso - la condizione finale CF (la condizione iniziale è vuota):

```
var x, somma: integer;
begin
  somma:= 0;
  while not eoln do begin
    read(x);
    somma:= somma+x
  end;
```

{CF: è stata letta una sequenza di interi $x_1,x_2,...,x_n$ terminata dal fine-riga, e nella variabile somma vi è la somma $0+x_1+x_2+...+x_n$ (intendendo con tale notazione che se la sequenza è vuota la somma vale 0)}

Dimostrazione di correttezza del programma.
Bisogna dimostrare che, nello stato raggiunto alla fine dell'esecuzione del programma, vale la CF. Ciò si ottiene come banale corollario di una proposizione la cui verità, anch'essa evidente, riposa sul principio di induzione.

Proposizione 1.
Qualunque sia n (‡0) dopo n iterazioni del ciclo sono stati letti n valori $x_1, ..., x_n$, e si ha: `somma` $= 0+x_1+x_2+...+x_n$.

DIMOSTRAZIONE PER INDUZIONE.

BASE ($n=0$):
Dopo 0 iterazioni, cioè immediatamente prima di eseguire l'istruzione *while*, si ha:

`somma` $= 0$.

DIMOSTRAZIONE: ovvia, perchè è stata eseguita l'istruzione `somma:=0`.

PASSO INDUTTIVO.

IPOTESI ($n = k$): Dopo k (‡0) iterazioni sono stati letti k valori $x_1, x_2, ..., x_k$, e si ha:

$$\texttt{somma} = 0+x_1+x_2+...+x_k$$

TESI ($n = k+1$):
Dopo $k+1$ iterazioni sono stati letti $k+1$ valori $x_1, x_2, ..., x_{k+1}$, e si ha:

$$\texttt{somma} = 0+x_1+x_2+...+x_{k+1}$$

DIMOSTRAZIONE (ovvia): l'istruzione `read(x)` immette in `x` un nuovo valore x_{k+1}, allora per ipotesi induttiva si ha:

$$\texttt{somma + x} = (0+x_1+x_2+...+x_k) + x_{k+1}$$

Quindi, dopo l'esecuzione dell'istruzione `somma:= somma+x` avremo

$$\texttt{somma} = 0+x_1+x_2+...+x_k+x_{k+1}$$

La proposizione precedente può essere scritta in forma piú concisa senza nominare esplicitamente il parametro n dell'induzione, cioè il numero di ripetizioni del ciclo:

Proposizione 1'.
Alla fine di ogni iterazione è stata letta una sequenza - che può non essere ancora terminata - di numeri, e nella variabile `somma` *c'è la somma di tale sequenza.*
Ciò vale anche dopo 0 iterazioni, cioè "all'inizio"; in tal caso la sequenza letta è la sequenza vuota.

Dalla **1** o dalla sua equivalente **1'** si deduce subito la proposizione esprimente la correttezza:

Proposizione 2 (correttezza). *Al termine (cioè all'uscita definitiva) dell'istruzione while, nella variabile* `somma` *c'è la somma di tutti i valori letti da una riga.*

DIMOSTRAZIONE (ovvia): Per la proposizione precedente, la variabile `somma` contiene alla fine di ogni iterazione la somma di tutti i valori letti fino a quel momento; quindi anche all'uscita definitiva dal *while* conterrà la somma di tutti i valori letti. D'altra parte, se si è usciti dal *while* vuol dire che la riga è terminata: quindi `somma` conterrà la somma di tutti i valori letti su una riga.

Che il programma fosse corretto era ovvio: esplicitare tutti i passi di un ragionamento cosí elementare può sembrare un inutile esercizio retorico. Vedremo tuttavia che lo stesso genere di argomentazione, applicato a problemi di programmazione meno banali, può essere di aiuto per costruire la soluzione o per controllarne la correttezza.

Intanto dall'analisi precedente si possono trarre alcuni insegnamenti che risulteranno di importanza fondamentale nel seguito.

- La progettazione di un ciclo corretto equivale alla costruzione di una dimostrazione per induzione: piú precisamente, le istruzioni del corpo del ciclo devono essere tali che grazie ad esse sia dimostrabile il passo induttivo; le istruzioni di inizializzazione devono essere tali da assicurare la (dimostrabilità della) base dell'induzione.

- Per progettare correttamente un ciclo che risolva un problema non banale, di solito non conviene pensare innanzitutto al primo passo, e poi al secondo passo ... e poi ci si è persi e si è realizzato un programma corretto solo per $n=0$, $n=1$, ed $n=2$.

Conviene invece cercare di porsi mentalmente al passo k-esimo generico, assumendo che in esso valgano certe condizioni, e scrivere il corpo del ciclo in modo che faccia passare correttamente dal passo *k* al passo *k+1*.
Spesso, solo dopo aver cosí ideato il ciclo si potrà stabilire l'inizializzazione in modo che il primo passo sia corretto. Oppure, passo k-esimo generico e inizializzazione saranno pensati insieme. (anche se, naturalmente, una ricetta meccanica non c'è, e si procederà spesso per tentativi ed errori, come in ogni attività creativa).

Ad esempio, nel caso della somma di una sequenza, soltanto dopo che si è già, se non scritto, almeno pensato che nel corpo del ciclo si deve fare `somma:= somma+x`, si capisce che si deve premettere l'inizializzazione `somma:= 0`; o perlomeno le due cose devono essere pensate contemporaneamente.

Nota terminologica.
Nel corso dei ragionamenti su programmi, quando si parla di "esecuzione del ciclo", può non essere chiaro se si intende l'esecuzione di una iterazione del ciclo, oppure l'esecuzione di tutte le iterazioni. Distinguiamo allora bene fra le seguenti due nozioni:

- *esecuzione dell'istruzione while (o for, o repeat)*: è l'esecuzione di tutte le iterazioni del test e del corpo del ciclo; l'istruzione *while*, come le altre iterative, è infatti una istruzione composta, che effettua la ripetizione delle istruzioni componenti;

- *esecuzione del corpo del while (o for, o repeat)*: è una esecuzione dell'istruzione o sequenza di istruzioni seguente il *do* nel caso del *while* o del *for*, compresa fra *repeat* e *until* nel caso del *repeat*;

Quando parliamo di esecuzione di una iterazione intendiamo un'esecuzione del corpo (ovviamente conseguente ad una esecuzione del test con risultato *true*). Naturalmente l'esecuzione di un'istruzione *while* può comportare zero esecuzioni del suo corpo (se il test è subito falso), oppure infinite esecuzioni del suo corpo, cioè l'esecuzione dell'istruzione *while* può non terminare (se il test non diventa mai falso); eccetera.

4.3.2 Un altro esempio: il massimo di una sequenza.

Si consideri il problema, proposto negli esercizi introduttivi, di calcolare il massimo di una sequenza non vuota (cioè contenente almeno un elemento) di interi, terminata dall'indicatore di fine-file (cioè, per la tastiera, dal carattere CTRL/Z).

Vi è una completa analogia con il problema della somma di una sequenza: affinchè alla fine dell'esecuzione una variabile `max` contenga il massimo dei numeri letti, bisogna che dopo ogni iterazione essa contenga il massimo dei numeri letti fino a quel momento. Si ragioni induttivamente: si assuma che `max` contenga il massimo dei valori finora immessi; si tratta allora di leggere un altro valore, confrontarlo con il massimo "provvisorio" o meglio "parziale", e se è maggiore sostituirlo ad esso.

```
read(x);
if x>max then max:= x
```

La base dell'induzione è costituita dal fatto che il massimo della sequenza di un solo elemento è quell'elemento stesso; quindi "all'inizio" `max` dovrà contenere il primo elemento. Il programma completo sarà:

```
var x,max: integer;
begin
  writeln('input sequenza di int. non vuota terminata da CTRL/Z:');
  readln(max);
  while not eof do begin
    readln(x);
    if x>max then max:= x
  end;
  writeln('massimo: ', max)
end.
```

La traccia della dimostrazione di correttezza è del tutto simile a quella della sezione precedente: si dimostra prima una proposizione per induzione, e da essa come banale corollario si deduce la condizione finale.

Proposizione 1.
Per qualunque n (‡0), dopo n iterazioni del ciclo sono stati letti n+1 valori x_0, x_1, *...,* x_n, *e la variabile* `max` *contiene il massimo dei valori letti, cioè, in formule:*
(\$*i*£*n . max =* x_i*)*   *("* *j*£*n . max ‡* x_j*)*

Oppure, senza nominare il numero n di iterazioni:

Proposizione 1'.
Al termine di ogni iterazione è stata letta una sequenza di numeri - che può non essere ancora terminata -, e nella variabile `max` *c'è il massimo di tale sequenza.*

Ciò vale anche dopo 0 iterazioni, cioè immediatamente prima dell'esecuzione dell'istruzione while; in tal caso la sequenza letta è la sequenza costituita da un solo elemento, e il massimo è tale elemento.

DIMOSTRAZIONE PER INDUZIONE.

BASE.
Dopo 0 iterazioni, cioè immediatamente prima di eseguire l'istruzione *while*, `max` contiene il massimo dei valori letti.

DIMOSTRAZIONE: ovvia, perchè è stata eseguita l'istruzione `readln(max)`, quindi è stato letto un (solo) valore, che è anche il massimo.

PASSO INDUTTIVO.

IPOTESI: Dopo *k* (≥0) iterazioni, `max` contiene il massimo dei valori letti.
TESI: Dopo *k+1* iterazioni, `max` contiene di nuovo il massimo dei valori letti (ed è stato letto un numero in piú).

oppure, equivalentemente:

IPOTESI: Si è in uno stato in cui `max` contiene il massimo dei valori letti.

TESI: Se si esegue il corpo del ciclo a partire da tale stato, si perviene ad uno stato in cui `max` contiene di nuovo il massimo dei valori letti (ma avendo letto un numero in piú).

La banale dimostrazione è analoga a quella per la somma.

Proposizione 2 (correttezza). Al termine (cioè all'uscita definitiva) dell'istruzione *while*, nella variabile `max` c'è il massimo di tutti i valori letti prima dell'immissione del carattere di fine-file.

4.4 Correttezza e induzione: invariante di ciclo.

4.4.1 L'invariante negli esempi del massimo e della somma.

Durante l'esecuzione dei programmi della somma o del massimo, riportati nella sezione precedente, la proprietà espressa dalla proposizione "il contenuto della variabile `somma` (o della variabile `max`) contiene la somma (o il massimo) dei valori immessi finora da tastiera" vale immediatamente prima dell'esecuzione dell'istruzione *while* (cioè, come abbiamo detto, prima della prima iterazione), e vale al termine di ogni iterazione; in particolare, vale quindi anche al termine dell'istruzione *while*, cioè all'uscita dal ciclo (se l'istruzione *while* termina, cioè se l'utente pigia a un certo punto rispettivamente il tasto ENTER o il tasto CTRL/Z).
Una tale proprietà o condizione, la cui validità non varia da un'iterazione all'altra, è detta INVARIANTE del ciclo. La dimostrazione che la proprietà citata è un invariante è sempre una dimostrazione per induzione come quelle della sezione precedente.
Si dimostra dapprima la base, cioè che la proprietà vale dopo le istruzioni di inizializzazione: ad esempio, la variabile `max` contiene il solo valore che è stato immesso, che è quindi correttamente il massimo.
Per dimostrare poi il passo d'induzione basta dimostrare che, se si assume che l'invariante valga prima di una generica iterazione, allora vale anche alla fine di tale iterazione. Si noti che, per poter assumere che l'iterazione venga eseguita, cioè che il corpo del *while* venga eseguito, occorre assumere che la condizione testata dal *while* sia vera. Allora si ha, nel caso del massimo:

IPOTESI INDUTTIVA:
Immediatamente prima di eseguire il corpo del while,
la variabile `max` *contiene il massimo dei valori fino ad allora immessi*
Ù
la sequenza da immettere non è terminata.

TESI INDUTTIVA:
Dopo aver eseguito il corpo del while,
la variabile `max` *contiene (di nuovo) il massimo dei valori fino ad allora immessi.*

La dimostrazione informale è naturalmente sempre la stessa, basata sull'esame degli effetti delle istruzioni del corpo.

In generale, quindi, per dimostrare che una certa proprietà INV è un invariante di un ciclo *while* avente come test un'espressione booleana esprimente una condizione B, bisogna dimostrare che:

1(base): *INV vale dopo le istruzioni di inizializzazione;*
2 (passo): *se (ipotesi) immediatamente prima di un'esecuzione del corpo valgono INV e B, allora (tesi) dopo l'esecuzione del corpo vale INV.*

Come nelle Proposizioni **1'** della sezione precedente, nella formulazione di ipotesi e tesi induttiva non abbiamo indicato esplicitamente il parametro dell'induzione; il fatto è che in tutte le dimostrazioni che una data proprietà è un invariante il parametro dell'induzione è ovviamente sempre il numero delle iterazioni; cioè il tipo di induzione è sempre lo stesso, e si riduce allo schema di dimostrazione qui sopra riportato, dove l'induzione non è piú esplicitamente citata. (In termini piú rigorosi: il teorema dimostrato per induzione è sempre derivabile nello stesso modo da due lemmi delle forme 1 e 2).

4.4.2 L'esempio dell' esponenziazione ingenua.

Si riprenda il semplice problema del calcolo dell' N-esima potenza di X, dati N naturale ed X reale (X 0) arbitrari; per risolvere tale problema il principiante deve riconoscerne l'analogia con i problemi della somma e del massimo di una sequenza di numeri:

$$X^N = \underbrace{X \cdot X \cdots X}_{\text{N volte}} \qquad X^0 = 1$$

Anche questo problema, infatti, si risolve con un ciclo, in cui la variabile `ris`, che alla fine conterrà il risultato finale, ad ogni iterazione contiene un risultato parziale che viene aggiornato tramite un'altra moltiplicazione per X.

```
var x, ris: real;
    i, n: integer;

begin
...
{CI: x = X, n = N, X <> 0, N >= 0}

  ris:= 1;
  for i:= 1 to n do ris:= x*ris;

{CF: ris = X^N}
```

Poichè in questa prima soluzione i contenuti di `x` e di `n`, come si vede, non cambiano durante l'esecuzione del ciclo, possiamo nel seguito usare direttamente i nomi delle rispettive variabili per indicare i valori X ed N.

Se immaginiamo di osservare (ad esempio attraverso i *watch* del sistema di sviluppo TurboPascal) lo stato della memoria - cioè i contenuti delle variabili - durante l'esecuzione del programma, abbiamo una successione di stati in cui i contenuti di `ris` e di `i` crescono ad ogni ripetizione del ciclo, ma con una legge ben precisa, che può essere espressa dicendo che alla fine di ogni iterazione i contenuti delle variabili stanno fra di loro nella seguente relazione:

$$\text{contenuto di ris} = (\text{contenuto di x})^{(\text{contenuto di i}) - 1}$$

ossia, scrivendo semplicemente i nomi dei contenitori per indicare i loro contenuti,

INV: $ris = x^{i-1}$

Detto in altro modo: (i contenuti di) `ris` e di `i` cambiano, ma la relazione fra di essi espressa dall'uguaglianza di cui sopra non cambia. Essa è quindi un invariante del ciclo: vale all'inizio (base dell'induzione), cioè prima di eseguire la prima iterazione del *while*; poi ogni esecuzione del corpo del ciclo, pur modificando i contenuti delle variabili, mantiene fra di esse la relazione suddetta (passo induttivo).

Quando si esce dal ciclo, essa quindi vale ancora: ma in più vale anche la condizione di uscita dal for, cioè:

CU: $i = n + 1$

Il fatto che valgano contemporaneamente INV e CU "produce", per cosí dire, la condizione che si ha all'uscita dal ciclo, cioè al termine dell'istruzione *for* :

POSTFOR: $ris = x^n$

che è proprio ciò che volevamo ottenere, e che in questo caso coincide con la condizione finale CF del programma. In termini piú rigorosi, si ha l'implicazione:

$$ris = x^{i-1} \wedge i = n+1 \quad \rightarrow \quad ris = x^n$$

cioè:

$$(INV \wedge CU) \rightarrow POSTFOR$$

In realtà, se vogliamo essere precisi, dobbiamo ricordare che secondo la definizione del Pascal (sia Standard che Turbo) il valore del contatore i all'uscita dal ciclo diventa *indefinito*; tuttavia esso vale n+1 *immediatamente prima* dell'uscita, quindi l'implicazione scritta sopra rimane valida, e la relazione $ris = x^n$ vale anche *immediatamente dopo* l'uscita; al contrario, una relazione che coinvolga esplicitamente la variabile i, come appunto `i = n+1`, anche se valida immediatamente prima dell'uscita, dopo non lo è piú.

L'inizializzazione della variabile `ris`, insieme all'inizializzazione implicita di `i` a 1, stabilisce la relazione invariante, poi ad ogni iterazione l'esecuzione del corpo del ciclo ha l'effetto di "mantenere" l'invariante e contemporaneamente "avvicinarsi" alla condizione di uscita, in modo da prima o poi raggiungerla, e raggiungere cosí automaticamente anche la condizione finale.

NOTA BENE: Le espressioni logiche INV, CU, ecc. scritte sopra, pur contenendo nomi di variabili Pascal, non sono espressioni Pascal, e quindi non fanno parte del programma se non eventualmente come commenti; in particolare non sono delle assegnazioni, ma non sono nemmeno delle operazioni di confronto; si noti infatti, tra l'altro, che la prima e la terza usano l'operazione di elevamento a potenza che non è definita in Pascal, e che anzi è proprio l'operazione che vogliamo realizzare con il nostro programma!

Esse sono, come nel programma che calcola il massimo di una sequenza, asserzioni (cioè affermazioni) intorno allo stato della memoria durante l'esecuzione del programma; esse sono in generale scritte in italiano o altra lingua naturale facendo uso della notazione matematica, dato che i fatti affermati sono relazioni di natura matematica fra i contenuti delle variabili in gioco.

4.4.3 Test del *while* e condizione di uscita.

Il ciclo *for* della sottosezione precedente può essere considerato equivalente ad un ciclo *while* :

```
ris:= 1; i:= 1;
while i <= n do begin
  ris:= x*ris;
  i:= i+1
end;
```

È evidente che anche in questo caso all'uscita del ciclo si ha i = n+1, e quindi si può ancora scrivere:

$$ris = x^{i-1} \wedge i = n+1 \quad \rightarrow \quad ris = x^n$$

Anzi, poichè ora `i` è una variabile ordinaria, essa mantiene tale valore anche dopo l'*end* del *while*, e non sono quindi necessarie le sottili precisazioni sul suo valore fatte nel caso del *for*.

Consideriamo però ora i cicli *while* in generale: essi possono avere le forme piú diverse, e l'unica cosa che si può dire è che alla fine di un'istruzione *while*, cioè all'uscita dal ciclo, vale la negata della condizione-test (se no non si sarebbe usciti!).

Se chiamiamo TEST la condizione testata dal *while*, allora la condizione di uscita CU è semplicemente ¬TEST, cioè la negazione di TEST; se chiamiamo POSTWH la condizione che vogliamo ottenere all'uscita dall'istruzione *while*, l'implicazione che dobbiamo dimostrare è:

(INV ∧ ØTEST) → POSTWH

Nella realizzazione dell'esponenziale con il *while* la negazione del test è `i>n`; ma da `i>n` e dall'invariante $ris = x^{i-1}$ non si deduce $ris = x^n$; bisogna prima "dimostrare" che in realtà si ha proprio i = n+1, il che peraltro si vede subito esaminando il corpo del ciclo. Una dimostrazione piú rigorosa può essere fatta osservando che la disuguaglianza *I£n+1* è anch'essa invariante: è infatti vera all'inizio quando *i=1*, perchè per ipotesi è *n£0*; inoltre viene mantenuta dall'esecuzione del corpo del ciclo, perchè se il test restituisce *true* si ha *i£n*, e quindi dopo l'incremento *i£n+1*. L'invariante "completo" è quindi in realtà:

INV: $ris = x^{i-1} \wedge i \leq n+1$

La condizione di uscita è:

CU (cioè ¬TEST): $i > n$

Cosí si ha: INV ∧ CU → $i = n+1$
perchè $i \leq n+1 \wedge i > n \rightarrow i = n+1$

quindi come prima $ris = x^{i-1} \wedge i = n+1 \rightarrow ris = x^n$

Abbiamo quindi ancora: INV ∧ CU → $ris = x^n$

Osserviamo che i cicli *while*, a differenza dei *for*, possono non terminare; quindi in generale la dimostrazione dell'implicazione

$$INV \wedge CU \rightarrow POSTWH$$

dimostra soltanto che se l'esecuzione del programma esce dal *while*, allora viene raggiunta la condizione POSTWH.

In questo caso la terminazione è ovvia (come del resto tutte le considerazioni svolte finora), e non l'analizziamo ulteriormente.

4.4.4 Osservazioni.

La relazione INV scritta sopra non è l'unica che si mantiene valida per tutta l'esecuzione del programma: ad esempio, la relazione $i+1-1 = i$, o piú semplicemente la relazione $i = i$, sono banalmente vere sempre: sono quindi anch'esse delle relazioni invarianti, ma non molto significative!

Le relazioni invarianti significative sono quelle che fanno sí che il programma si comporti in quel determinato modo; ossia, spostandoci dal piano dei "fatti" (proprietà e relazioni matematiche fra contenuti di variabili) al piano linguistico (le *asserzioni* di tali fatti, e le dimostrazioni), le asserzioni invarianti significative sono quelle che permettono (o permetterebbero) di dimostrare la correttezza del programma. Anche fra queste, poi, alcune giocano un ruolo fondamentale, altre sono secondarie - e spesso, nei ragionamenti informali, trattate in modo implicito; in una dimostrazione formale, però, esse sarebbero scritte esplicitamente.

Esempi di invarianti secondari sono, in questo caso, $x = X$ e $n = N$. Essi sono stati espressi in modo informale dicendo che i contenuti di `x` e di `n` non cambiano, e che possiamo scambiare indifferentemente `x` con X ed `n` con N; nella prossima sottosezione saranno riportati esplicitamente, nel riepilogo della dimostrazione.

In queste note, tuttavia, per "asserzione invariante di un ciclo" intenderemo perlopiú come nell'esempio precedente solo l'asserzione della relazione (o insieme di relazioni) piú significativa.

Simmetricamente, progettare un ciclo *while* che risolva correttamente un dato problema equivale ad individuare una relazione invariante ed una condizione di uscita le quali, congiunte, abbiano come conseguenza la condizione richiesta alla fine del ciclo, e a scrivere delle istruzioni di inizializzazione e di ciclo che mantengano l'invariante e garantiscano il raggiungimento della condizione di uscita; quest'ultima proprietà è fondamentale: infatti la validità dell'invariante, assicurata inizialmente dall'inizializzazione, può essere mantenuta banalmente non facendo niente, cioè con un ciclo *while* con corpo vuoto, ma ciò non servirebbe a molto per risolvere il problema!
Insomma: per scrivere un ciclo *while* corretto una volta individuato l'invariante significativo, bisognerà scrivere delle istruzioni che mantengano l'invariante in modo non banale; e viceversa, per dimostrare la correttezza di un programma dato, occorrerà individuare un invariante significativo, cioè non banale!

4.4.5 Dimostrazione di correttezza dell'esponenziale ingenuo.

Riassumiamo i ragionamenti sul problema dell'esponenziale esposti nelle sottosezioni precedenti, ordinandoli nella forma di una traccia della dimostrazione di correttezza.

```
var x, ris: real;
    i, n: integer;
```

```
begin
...
{CI: x = X, n = N, X <> 0, N >= 0}

ris:= 1; i:= 1;
while i <= n do begin
  ris:= x*ris;
  i:= i+1
end;

{CF: ris = X^N}
...
```

Chiamiamo *Prog* la porzione di programma scritta sopra, compresa fra i due commenti "iniziale" e "finale"; ciò che dimostriamo è (limitandoci per ora alla correttezza cosiddetta parziale):
Proposizione. Se immediatamente prima dell'esecuzione di *Prog* lo stato della memoria soddisfa alla condizione iniziale CI, allora, se l'esecuzione di *Prog* termina, termina in uno stato che soddisfa alla condizione CF.

Per questo, dobbiamo prima dimostrare che:

Proposizione 1. La relazione espressa dalla formula logica

INV: $x = X \land n = N \land ris = X^{i-1} \land i \leq n+1$

è un invariante del ciclo.

Dimostrazione:

Base. INV vale prima dell'esecuzione dell'istruzione *while*.
Dimostrazione: ovvia.

Passo.
Ipotesi: Sia s uno stato della memoria in cui vale INV $\land$ $i \leq n$.
Tesi: Nello stato s' che si ottiene da s dopo aver eseguito il corpo del ciclo, vale INV.

Dimostrazione:
Il corpo del ciclo non altera le variabili x ed n, quindi le prime due componenti della congiunzione restano valide.
Dopo l'esecuzione dell'istruzione `ris:= x*ris` si ha evidentemente $ris = X^i$; dopo l'esecuzione dell'istruzione `i:= i-1` si ha evidentemente $ris = X^{i-1}$.
Per ipotesi è $i \leq n$, quindi dopo aver incrementato `i` si ha $i \leq n+1$.

Ora basta dimostrare che:

Proposizione 2. INV $\land$ $i > n \rightarrow ris = X^N$

La dimostrazione è (una riscrittura di) quella riportata nella sottosezione 4.4.3.

Osserviamo che nella dimostrazione di correttezza di un programma, diversamente dalle dimostrazioni matematiche tradizionali, la nozione di tempo gioca un ruolo privilegiato,

giacchè si devono dimostrare fatti riguardo all'esecuzione di un programma, cioè riguardo ad una storia che si svolge nel tempo.

In quanto precede abbiamo due generi di dimostrazioni: quello della *Proposizione 1*, che coinvolge il tempo, e quello della *Proposizione 2*, che invece è simile a quelle delle formule logiche della matematica tradizionale.

Per la dimostrazione della *1* si devono prendere in considerazione gli effetti dell'esecuzione delle istruzioni. Si faccia bene attenzione al fatto che nella *1* ciò che si dimostra nel passo induttivo non è la formula:

$$\text{INV} \wedge (\texttt{i} \leq \texttt{n}) \rightarrow \text{INV}$$

che sarebbe una tautologia banale, ma qualcosa che ha a che vedere con un prima e un dopo. La dimostrazione della *2* è invece piú simile ad una tradizionale dimostrazione matematica.

Ricordiamo infatti che un'*asserzione*, come *CI* o *INV*, è una proposizione che dipende implicitamente dallo stato. Per indicare esplicitamente tale dipendenza potremmo scrivere *CI(s)* e *INV(s)*, cosí come, per indicare che il contenuto di una variabile `i` dipende dallo stato *s*, potremmo scrivere `i`*(s)*. Cosí la *Proposizione 1* afferma che, se lo stato *s* prima dell'esecuzione delle istruzioni di inizializzazione del ciclo è tale che *CI(s)* è vera, allora *INV(s')* è vera in ogni stato *s'* che si ottiene dopo un qualunque numero di iterazioni del corpo del *while*. La *Proposizione 2* invece semplicemente afferma che in qualunque stato *s* si ha: *INV(s)* Ù *(*`i`*(s) >* `n`*(s))* fi `ris`*(s) =* X^N, cioè:
INV(s) Ù *TEST(s)* fi *CF(s)*.

4.4.6 Prudenza e sicurezza.

Se nel ciclo *while* sostituiamo il test `i<=n` con il test `i<>n+1`, il programma dell'esponenziale funziona ancora perfettamente - purchè la condizione iniziale su `x` e su `n` sia verificata.

In tale realizzazione la negazione del test è proprio $i = n+1$, e la dimostrazione di correttezza parziale non ha quindi bisogno di far intervenire l'invariante *i£n+1*.

La ragione per cui si preferisce la prima versione è che essa, a differenza dell'altra, garantisce che il programma termini (pur se con un risultato errato) anche con input scorretto $n<0$. La prima versione è cioè piú "sicura" (in inglese *safe*) in caso di "incidente", proprio come un'auto dotata di air-bag rispetto ad una che ne è priva.

4.4.7 Un altro esempio: esponenziale ingenuo con ciclo discendente.

Poichè nella soluzione precedente il contatore `i` del *for* non viene usato (esplicitamente) all'interno del ciclo, ma serve solo a contare *n* ripetizioni, essa resta naturalmente valida se invece di un ciclo *for* ascendente si usa un ciclo *for* discendente:

```
ris:= 1;
for i:= n downto 1 do ris:= x*ris;
```

Nel ciclo discendente il valore iniziale di `n` è usato solo per inizializzare `i`, poi non serve più, allora ricordando la semantica del *for* potremmo scrivere, risparmiando la `i`:

```
for n:= n downto 1 do ris:= x*ris;
```

oppure, in modo piú chiaro, usando un ciclo *while*:

```
ris:= 1;
while n > 0 do begin
  ris:= x*ris;
  n:= n-1
end;
```

(nella versione con il *while*, come in quella col *for* che non usa la `i`, alla fine del calcolo si è perso il valore iniziale dell'esponente).

Come per la soluzione precedente, cerchiamo di capire come variano i contenuti delle variabili durante l'esecuzione del ciclo.

Siano X ed N i valori letti da tastiera e memorizzati all'inizio rispettivamente in `x` ed `n` (si richiede, come prima, che sia N≥0). Poichè il contenuto X di `x` non cambia, nella scrittura potremo indifferentemente scambiare `x` con X, e ci basterà descrivere il comportamento soltanto di `ris` e di `n`:

0) Istante iniziale:	`n` = N	`ris` = 1
1) Dopo il 1º ciclo:	`n` = N-1	`ris` = X
2) Dopo il 2º ciclo:	`n` = N-2	`ris` = X^2
...	...	...
m) Dopo l'm-esimo ciclo:	`n` = N-m	`ris` = X^m
...	...	...
N) Dopo l'N-esimo ciclo:	`n` = 0	`ris` = X^N

Osservando la tabella, ci accorgiamo che la quantità

$$x^n \cdot ris$$

(dove con i nomi dei contenitori indichiamo come al solito i loro contenuti) rimane invariata ad ogni iterazione, e precisamente si ha:

$$x^n \cdot ris = X^N$$

dove X^N è il risultato desiderato.

L'uguaglianza precedente è l'asserzione INVARIANTE del ciclo; essa esprime il fatto che, durante il processo di calcolo, in `ris` c'è il risultato parziale via via accumulato, mentre in `n` c'è il numero di volte per cui bisogna ancora moltiplicare `x` per `ris` per ottenere il risultato finale; ossia, il numero x^n è il numero per cui bisogna ancora moltiplicare il risultato parziale per ottenere il risultato finale.

Si badi bene che tale x^n non si trova da nessuna parte nella memoria dell'esecutore Pascal, ma vi è soltanto "implicitamente" o "potenzialmente" presente, codificato - per cosí dire - nella coppia di valori (contenuti in) `x` ed `n`.

Con linguaggio un po' fantasioso potremmo descrivere il processo di calcolo nel modo seguente. Una volta effettuato l'input dei valori X ed N, il numero X^N (che è il secondo membro dell'equazione invariante) è da quel momento in poi, per tutta la durata del calcolo, già potenzialmente presente nel calcolatore, all'inizio sotto forma appunto della coppia di numeri X ed N, poi come prodotto di due fattori, di cui uno (x^n) ancora potenziale, l'altro (il risultato parziale contenuto in `ris`) invece effettivo, cioè calcolato; ad ogni iterazione si riduce il fattore potenziale (decrementando N) e si fa crescere il fattore effettivo (moltiplicandolo per X), finchè alla fine il numero è tutto effettivo, cioè

è stato calcolato ed è presente in forma esplicita nella memoria (e può ad esempio venire visualizzato sullo schermo, in un'altra forma ...).

Insomma, come al solito la relazione invariante, essendo valida alla fine di ogni iterazione, vale in particolare anche all'uscita dal ciclo, nel qual caso però vale anche la condizione di uscita

$$n = 0$$

e quindi si ha:

$$x^0 \cdot ris = X^N$$

cioè

$$ris = X^N$$

che è proprio la condizione finale richiesta .

Anche qui, come nel caso del ciclo ascendente, affinchè la condizione di uscita sia esattamente la negazione della condizione del while bisogna scrivere il ciclo nella forma:

```
while n<>0 do begin ...
```

Se invece usiamo la forma piú "sicura" `while n>0,` allora a rigore dobbiamo inserire nell'invariante la condizione *n‡0*, valida all'inizio per la specifica del problema, ed evidentemente mantenuta dal corpo del ciclo:

INV: $(x^n \cdot ris = X^N) \wedge (n \geq 0)$

La negazione della condizione del test del *while* è:

CU: $n \leq 0$

Dalla congiunzione di *n‡0* e *n£0* si ha *n=0*, sostituendo nell'invariante si ha come prima:

CF: $ris = X^N$

cioè: INV $\wedge$ CU $\rightarrow$ POSTWH

4.5 Tempo di calcolo: introduzione.

Perchè un programma o sottoprogramma sia una soluzione accettabile di un problema, non basta che sia corretto rispetto alla specifica del problema; occorre anche che la risposta sia fornita in un tempo ragionevole, cioè che l'esecuzione del programma abbia una durata "non troppo lunga". Infatti, pur escludendo per ora i programmi con vincoli di "tempo reale", un programma che calcolasse una semplice funzione matematica (ad esempio la sequenza di Fibonacci) in modo corretto ma in tempi di secoli o millenni sarebbe di ben scarsa utilità (non si pensi ad esagerazioni: si vedrà nella seconda parte del corso che proprio una delle realizzazioni piú "facili" della sequenza di Fibonacci ha questa spiacevole proprietà).

Naturalmente, il tempo di esecuzione di un programma dipende da molte cose, fra cui il tipo di calcolatore (cioè la velocità della sua unità centrale e dei suoi accessi alla memoria) e il valore dei dati in ingresso. Non possiamo quindi dare la misura della velocità di un programma fornendo un numero di millisecondi o di secondi; dovremmo invece fornire, per ciascun tipo di calcolatore, il tempo per ognuno degl'infiniti possibili valori di input. Dovremmo fornire, cioè, una funzione: se chiamiamo *n* il valore del

parametro di input (che per ora, per semplicità, supponiamo unico e di tipo numerico), dovremmo fornire una funzione $t = T(n)$.

Come si è detto, la funzione $T(n)$ sarà diversa, per quanto riguarda i suoi valori numerici, da macchina a macchina; tuttavia, poichè ciò che distingue una macchina dall'altra è solo la velocità delle operazioni elementari, ma non la loro natura di base, l'andamento di $T(n)$ al variare (cioè al crescere) di n resterà lo stesso, nel senso che se ad esempio è $T(n) = an^2 + bn$, con a e b costanti, tale formula varrà per qualunque macchina, e muteranno soltanto i valori delle costanti a e b; in ogni caso il tempo crescerà sempre in modo quadratico rispetto ad n, e il suo grafico sarà costituito da una parabola.

In una delle sezioni successive definiremo rigorosamente la nozione di *complessità temporale* di un programma; intanto possiamo esaminare da questo nuovo punto di vista i programmi presentati fin qui.

Quelli per il calcolo dell'esponenziale impiegano, per calcolare l'n-esima potenza di x, un tempo lineare in n (ossia proporzionale a n), poiché (dopo le istruzioni di inizializzazione) ripetono n volte una sequenza di due istruzioni semplici; se chiamiamo a il tempo costante necessario per eseguire il test e il corpo del ciclo, e b il tempo necessario per le istruzioni di inizializzazione, il tempo di calcolo totale è (a parte qualche costante in piú o in meno):

$$T(n) = an + b$$

Si osservi che spesso la soluzione "piú naturale" di un problema è quella piú inefficiente: per scoprirne di migliori, oppure per scoprire che non possono esistere soluzioni migliori, sono di solito necessari ragionamenti di natura matematica.

Un esempio banale è il problema del calcolo della somma dei naturali da 1 a n; l'algoritmo naturale, consistente nel sommare ad uno ad uno tutti gli n numeri, impiega un tempo evidentemente proporzionale ad n; l'algoritmo scoperto da Gauss bambino, consistente nel calcolare l'espressione $n(n+1)/2$, richiede l'esecuzione di solo tre operazioni, che (almeno per n non troppo grande) l'uomo e a maggior ragione la macchina compiono in tempo costante. Si è passati cosí da un andamento lineare ad un andamento costante: ma per questo è stato necessario dimostrare un teorema (per induzione).

4.6 Un'applicazione: l'esponenziale veloce.

4.6.1 Costruzione del programma.

Cerchiamo ora di costruire un algoritmo piú veloce sfruttando una nota proprietà delle potenze:

$$\text{se N è pari allora} \quad X^N = (X^2)^{N \text{ div } 2}$$

Usando una volta tale formula per calcolare una potenza con esponente pari N, invece di N moltiplicazioni ci basta fare una moltiplicazione per ottenere il quadrato, e poi N/2 moltiplicazioni: abbiamo quindi un risparmio di circa metà tempo.

Se ora per semplicità chiamiamo W il quadrato di X, e M la metà di N, se per caso M è ancora pari possiamo applicare la stessa formula anche nel calcolo di W^M; se invece M è dispari, allora M-1 è pari, e possiamo quindi applicare lo stesso metodo per calcolare W^{M-1}, e cosí via ... induttivamente, ottenendo - come vedremo - ben piú di un semplice risparmio di tempo di un fattore costante indipendente da N.

Come si scrive un programma che realizzi correttamente un tale metodo di calcolo? Più in generale, come si passa dall'idea di base di un algoritmo alla realizzazione di un programma corretto? Cerchiamo, per una volta, di esporre in dettaglio un percorso di ragionamento che conduce alla scrittura del programma.

Partiamo dall'analisi, su un paio di esempi, del modo in cui eseguiremmo il calcolo a mano. Prendiamo dapprima con un esponente che sia una potenza di due, ad esempio 16:

$$3^{16} = 9^8 = 81^4 = 6561^2 = 43046721^1 = 43046721.$$

Come si vede, il risultato si ottiene con sole quattro moltiplicazioni invece di sedici:

$$3 \cdot 3 = 9;\ \ 9 \cdot 9 = 81;\ \ 81 \cdot 81 = 6561;\ \ 6561 \cdot 6561 = 43046721.$$

È chiaro che per gli esponenti potenze di due il programma potrebbe essere costituito da un semplice ciclo *while* che ad ogni iterazione moltiplichi per se stesso il risultato precedente e divida per due l'esponente (e termini quando l'esponente arriva ad 1); ma si tratta di un caso troppo particolare.

Proviamo allora con un esponente che non sia una potenza di due, ad esempio 13:

$$3^{13} = \mathbf{3 \cdot 3^{12}} = \mathbf{3 \cdot 9^{6}} = \mathbf{3 \cdot 81^{3}} = 3 \cdot 81 \cdot 81^2 = \mathbf{243 \cdot 81^{2}} = \mathbf{243 \cdot 6561^{1}} = 1594323$$

Come si vede, la traccia del calcolo si può pensare costituita da una sequenza di espressioni numeriche (quelle evidenziate in grassetto): ognuna di esse è un prodotto di due fattori, dove il fattore di sinistra è sempre un valore già calcolato (come risultato dei passi precedenti), mentre il fattore di destra è una potenza ancora da calcolare, data da una coppia base-esponente. Il fattore di sinistra è quindi un risultato parziale; il fattore di destra esprime un calcolo ancora da fare, dove però si noti che la base non è piú in generale quella di partenza, ma è anch'essa il risultato di passi di calcolo precedenti.

Possiamo allora tenere il fattore di sinistra in una variabile `ris`, e tenere base ed esponente del fattore di destra in due variabili rispettivamente `x` ed `n`; l'invariante è quindi lo stesso della versione precedente:

INV: $\quad x^n \cdot ris = X^N \quad \wedge \quad n \geq 0$

Ne consegue che, per avere il risultato finale in `ris`, devono essere le stesse anche l'inizializzazione e la condizione di uscita (quindi il test del *while*):

```
ris:= 1;
while n > 0 do begin ...
```

INV: $\quad x^n \cdot ris = X^N \quad \wedge \quad n \geq 0$

CU: $\quad n \leq 0$

Ciò che è diverso rispetto alla versione precedente è il corpo del ciclo: esso dovrà infatti eseguire un nuovo tipo di passo di calcolo, pur mantenendo lo stesso invariante, cioè lasciando immutato il valore dell'espressione $x^n \cdot ris$. Poniamoci mentalmente all'inizio di una generica iterazione, assumendo che un risultato parziale si trovi in `ris`, e che x^n sia la quantità (non calcolata) "ancora da moltiplicare per `ris`".

Ci sono due casi, a seconda che il contenuto di `n` sia pari o dispari.

Se n è pari si ha l'uguaglianza $x^{n} \cdot ris = (x^{2})^{n \text{ div } 2} \cdot ris$. Se allora al posto del numero contenuto in x mettiamo il suo quadrato e simultaneamente dimezziamo il contenuto di n, il valore di x^{n} non cambia e quindi, lasciando invariato ris, il valore di $x^{n} \cdot ris$ non cambia:

```
if n è pari then begin
  x:= x*x; n:= n div 2  (* l'ordine non importa *)
end
```

Cosí si mantiene anche la seconda parte dell'invariante: se infatti n è (per la condizione del *while*) un intero > 0, n div 2 è un intero ≥ 0.

Nel caso in cui n è dispari, la sopra ricordata proprietà delle potenze non si può applicare; poichè però l'invariante è lo stesso della soluzione ingenua, si può comunque eseguire in tal caso la stessa coppia di istruzioni che si eseguiva là:

```
else {n dispari} begin
  ris:= x*ris; n:= n-1  (* l'ordine non importa *)
end;
```

Rispetto alla soluzione ingenua è diversa solo la situazione in cui le due assegnazioni sono eseguite: come avevamo già notato nell'esempio numerico, x in generale non è piú uguale al valore iniziale della base. Scriviamo in conclusione una versione completa della funzione, corredata di asserzioni iniziale, invariante, e finale:

```
function fastexp(x: real; n: integer): real;
var ris: real;
begin                              {CI: x=X, n=N, X 0, N>=0}
  ris:= 1;                         {INV: x^n*ris = X^N, n>=0}
  while n>0 do
    if n mod 2 = 0 then begin
      x:= x*x;
      n:= n div 2
    end
    else begin
      ris:= x*ris;
      n:= n-1
    end;
  fastexp:= ris;                    {CF: ris = X^N}
end.
```

(si noti che il corpo del *while* è un'unica "grande" istruzione *if-then-else*, pertanto non è necessario racchiuderlo fra un *begin* e un *end*; comunque non è un errore farlo, anzi il programma risulta forse piú chiaro).

Ricordiamo ancora una volta che un programma o sottoprogramma annotato con asserzioni contiene: un'asserzione iniziale o precondizione che si richiede sia soddisfatta dai dati iniziali; un certo numero di asserzioni poste in vari punti del programma, ognuna delle quali risulta soddisfatta - se lo stato iniziale soddisfaceva l'asserzione iniziale - ogni volta che l'esecuzione del programma raggiunge quel punto; un'asserzione finale che è soddisfatta alla fine dell'esecuzione e che costituisce la soluzione del problema.

L'asserzione (o condizione) iniziale e l'asserzione (o condizione) finale non sono altro che la specifica del problema; le asserzioni intermedie (in questo caso l'invariante) descrivono gli stati intermedi piú significativi attraverso cui passa la computazione per arrivare allo stato finale partendo dallo stato iniziale.

Esercizio 3. Riscrivere il contenuto di questa sottosezione nella forma di un'esplicita dimostrazione di correttezza del sottoprogramma.

4.6.2 Terminazione e correttezza totale.

Chi veda per la prima volta, senza le spiegazioni sopra riportate, il (sotto)programma appena scritto, potrebbe chiedersi a che cosa serva un passo di calcolo come quello del caso pari, che non porta alcun contributo all'accumulazione del risultato in `ris`.

Naturalmente la risposta è che un calcolo viene comunque fatto, in `x` invece che in `ris`. Il valore eventualmente accumulato in `x` attraverso una sequenza di passi "di nuovo tipo" verrà "moltiplicato in" `ris` al prossimo passo "di vecchio tipo", cioè non appena a forza di dividere per 2 si ottiene un esponente dispari. Ma non potrebbe succedere che non si ottenga mai un esponente dispari e più in generale che il programma non termini? Si vede subito che no.

Infatti, per l'invariante si ha sempre *n ‡ 0*; d'altra parte ogni volta che il corpo del ciclo viene eseguito si deve avere prima dell'esecuzione *n > 0* (per il test del *while*), ma:

se *n > 0* allora *n div 2 < n* e *n-1 < n*

I valori successivamente assunti da `n` costituiscono perciò una successione <u>strettamente decrescente</u> di interi non negativi, che deve quindi necessariamente terminare con 0 (in parole povere: ad ogni iterazione `n` diventa strettamente piú piccolo, pur rimanendo un intero non negativo; allora prima o poi diventerà zero). Con ciò è dimostrato che la condizione di uscita dal *while* viene sempre raggiunta, e che quindi per qualunque input corretto si ha in tempo finito la risposta corretta.

Ai fini della comprensione di <u>come</u> il programma funziona, si può notare che se il valore iniziale di `n` è maggiore di zero il penultimo valore della successione è sempre il numero dispari 1: infatti ciò è ovvio se il valore iniziale di `n` è 1; d'altra parte, se è *n > 1* si ha *n div 2 > 1* e *n-1 ‡ 1*, e quindi prima o poi si ottiene *n=1*.

La proprietà di terminazione è particolarmente evidente se si pensa alla rappresentazione binaria. In essa i numeri pari e i numeri dispari sono semplicemente quelli rispettivamente con zero e con uno nel bit piú a destra (il meno significativo); il decremento di 1 di un numero <u>dispari</u> equivale a sostituire uno con zero nel suo bit piú a destra, la divisione intera per 2 corrisponde a eliminare il bit meno significativo, cioè ad un'operazione di scorrimento (*shift*) verso destra (con immissione di 0 nel bit piú a sinistra, cioè il piú significativo). La successione di divisioni per 2 e decrementi di 1 equivale perciò ad una successione di *shift* verso destra, che termina necessariamente con tutti i bit a zero, cioè con la rappresentazione binaria del numero 0.

La dimostrazione che <u>quando</u> si raggiunge la condizione di uscita si realizza la condizione finale richiesta è detta di <u>correttezza parziale</u>; la successiva o contemporanea dimostrazione <u>che la condizione di uscita viene (sempre) raggiunta,</u> cioè <u>che il programma termina (sempre)</u>, permette di stabilire la cosiddetta <u>correttezza totale</u> del programma.

Negli esempi precedenti la dimostrazione di terminazione non era stata fatta perchè ovvia, dato che il ciclo *while* era equivalente a un *for*. Non è invece equivalente a un *for* nell'esponenziale veloce, perchè il ciclo non si limita a decrementare un contatore, ma può anche dividerlo; la dimostrazione di terminazione diventa allora importante, anche se in questo caso particolare rimane banale.

Di un programma del quale si possa dire che, se terminasse, calcolerebbe il risultato correttamente, ma in realtà non termina mai, non sapremmo che farcene! Nè sarebbe

molto utile un programma che calcolasse il risultato corretto per certi valori di input, ma non terminasse per altri.

4.6.3 Complessità.

Analizziamo per il nuovo algoritmo l'andamento del tempo di calcolo in funzione del valore iniziale di `n` (che indichiamo semplicemente con n).

Consideriamo dapprima l'andamento di $T(n)$ al crescere di n soltanto per quei valori di n che sono potenze di due (come 2, 4, 8, 16, 32, ...), cioè per i *casi migliori*. Sia dunque $n = 2^k$. Dividendo per due si ottiene di nuovo un numero pari 2^{k-1}, e cosí via fino a 1. Il ciclo *while* viene quindi ripetuto $k+1$ volte: k volte per ridurre il valore di `n` a 1, piú una volta per passare da 1 a 0.

Quindi il tempo di calcolo è proporzionale a $k+1$; ma $k = log_2 n$, dunque:

$T(n) @ log_2 n + 1$

Per un numero pari generico naturalmente non è vero che la sua metà sia necessariamente pari; osserviamo tuttavia che se un numero è dispari, decrementandolo di 1 si ottiene necessariamente un numero pari. Allora a un passo di decremento segue sempre immediatamente un passo di divisione, e alla peggio si avranno per tutta l'esecuzione un passo di decremento e uno di divisione alternati.

Consideriamo l'andamento di $T(n)$ al crescere di n soltanto per quei valori di n costituenti tali *casi peggiori*: cioè quei valori che divisi per due danno un numero dispari che a sua volta decrementato di 1 e diviso per due dà di nuovo un dispari, e cosí via.

Se si pensa di nuovo alla rappresentazione binaria, si vede subito che tali numeri sono quelli in cui dopo ogni *shift* verso destra si ritrova 1 nel bit piú a destra (il meno significativo); sono cioè i numeri formati da tutti 1, ossia quegli n tali che $n = 2^k - 1$ (come 3, 7, 15, 31, ...). Per passare da $2^k - 1$ a 0 sono quindi necessari k decrementi e $k-1$ divisioni, in totale $2k-1$ passi (cioè $2k-1$ iterazioni del ciclo). Ma è $k = log_2(n+1)$, quindi:

$T(n) @ 2log_2(n+1) - 1 @ 2log_2 n$

Per valori generici di n abbiamo evidentemente risultati intermedi fra il caso migliore e il caso peggiore; in generale si ha pertanto:

$$log_2 n + 1 £ T(n) £ 2log_2(n+1) - 1$$

(ossia, in modo piú grossolano: $log_2 n £ T(n) £ 2log_2 n$)

Una funzione di questo genere che, pur essendo rappresentata da una curva "irregolare", è però delimitata inferiormente e superiormente da due funzioni logaritmiche, cioè ha il grafico compreso fra due curve logaritmiche, si dice comunque che ha *andamento (strettamente) logaritmico*; simbolicamente si scrive

$T(n) = Q(n)$ (si legge: *T(n) è "theta grande" di n*)

L'esponenziale veloce ha dunque, rispetto all'esponenziale ingenuo, complessità temporale <u>logaritmica</u> invece che <u>lineare</u>; quindi, per n sufficientemente grande, il tempo di esecuzione dell'algoritmo veloce sarà inferiore a quello dell'algoritmo ingenuo.

Un marginale miglioramento (della costante moltiplicativa, ma non dell'ordine di infinito) può essere ottenuto "compattando due passi in uno" grazie all'osservazione - fatta sopra che a un passo di decremento segue sempre immediatamente un passo di divisione: quando n è dispari, si può nello stesso ciclo fare il passo corrispondente al decremento e quello alla divisione per due, senza rifare il test. Osservando inoltre che per n dispari è $n \ div\ 2 = (n-1)\ div\ 2$, si ha il compatto (sotto)programma seguente:

```
function fastexp(x: real; n: integer): real;
var ris: real;
begin
  write('immetti base ed esponente: ');
  readln(x,n);
  ris:= 1;
  while n>0 do begin
    if n mod 2 = 1 then ris:= x*ris; {solo se n dispari}
    x:= x*x;                         {in entrambi i casi}
    n:= n div 2                      {in entrambi i casi}
  end;
  fastexp:= ris;
end.
```

Esercizio 4. Scrivere un programma che realizzi in modo "ingenuo" la moltiplicazione per mezzo dell'addizione: $n \cdot m = m + m + \dots m$ (*n* volte), e poi un programma che realizzi la "moltiplicazione veloce" per mezzo dell'addizione e dell'operazione *div 2*, analogo all'esponenziale veloce.

4.7 Uno degli algoritmi piú vecchi del mondo: l'algoritmo di Euclide per il MCD.

4.7.1 L'invenzione dell'algoritmo.

Il problema è il seguente: *Dati due numeri naturali (cioè interi non negativi) a e b non entrambi nulli, trovare il loro massimo comun divisore (MCD), cioè il piú grande numero naturale che divide sia a che b (cioè, detto ancora in altre parole, il piú grande numero naturale di cui sia a che b sono multipli).*

Si noti che, poichè tutti i numeri sono divisori di zero, il MCD di due numeri di cui uno sia zero e l'altro maggiore di zero è il numero maggiore di zero, ad esempio il MCD di 0 e 28 è 28; invece il MCD(0,0) non esiste, perchè dovrebbe essere il massimo di tutti i numeri! (mentre il MCD di due numeri coincidenti non nulli è ovviamente quel numero stesso, es. MCD(28,28) = 28). Per questo nelle condizioni iniziali si richiede che *a* e *b* siano non entrambi nulli.

La definizione di MCD di *a* e *b* può essere direttamente tradotta in un metodo "ingenuo" di calcolo: si trovano prima tutti i divisori di *a*, poi tutti i divisori di *b*, poi si prendono quelli comuni, e di essi il piú grande; piú precisamente (ricordando una proprietà elementare dei numeri) si scompongono sia *a* che *b* in fattori primi, e poi il MCD si ottiene dal "prodotto dei fattori primi comuni presi ciascuno con l'esponente minore".

Esempio. Il MCD di 336 e 180:

$336 = 2^4 \cdot 3 \cdot 7$ $\quad$ $180 = 2^2 \cdot 3^2 \cdot 5$ $\quad$ $MCD = 2^2 \cdot 3 = 12$

(i divisori comuni sono infatti 1, 2, 3, 4, 12, e di essi 12 è il piú grande).

Naturalmente, per generare la scomposizione in fattori primi di un numero *m* bisogna generare i successivi numeri primi da 1 ad *m* e per ciascuno di essi trovare l'esponente della sua massima potenza che sia divisore di *m*. Anche supponendo di avere a disposizione una tabella di numeri primi abbastanza grande, le operazioni da fare sono molte... (vedi libro di testo, introduzione).

Euclide, greco di Alessandria (d'Egitto), circa 2300 anni fa' inventò (o almeno descrisse) un metodo migliore, fondato sulla seguente proprietà dei numeri naturali:

Proposizione. Siano q ed r rispettivamente il quoziente ed il resto della divisione intera di a per b. Allora n è divisore comune di a e b se e solo se è divisore comune di b ed r.

Detto in altro modo:
(1) Se n è divisore comune di a e b, allora è divisore comune di b ed r.
(2) Viceversa, se n è divisore comune di b ed r, allora è divisore comune di a e b.

Dimostrazione.
Si ha, per definizione di quoziente e resto:
$a = b \cdot q + r$, con a,b,q,r naturali.

Dimostrazione di (1).
Sia n divisore sia di a che di b; allora vi sono due naturali a_1 e b_1 tali che:

$a = a_1 \cdot n$
$b = b_1 \cdot n$

Sostituendo si ha:

$a_1 n = b_1 n \cdot q + r$

$a_1 n - b_1 n \cdot q = r$

$r = (a_1 - b_1 \cdot q)\, n$

Poichè a_1, b_1, q sono dei naturali, il numero $a_1 - b_1 \cdot q$ è sicuramente un intero; anzi, poichè r ed n sono due naturali (cioè interi non negativi), anche $a_1 - b_1 \cdot q$ è non negativo. Quindi r è uguale al prodotto di n per un naturale, ossia n è divisore anche di r, quindi è divisore comune di b ed r.

Dimostrazione di (2).
Sia n divisore comune di b ed r; allora vi sono due naturali b_1 ed r_1 tali che:

$b = b_1 \cdot n$
$r = r_1 \cdot n$

Sostituendo, si ha:

$a = b_1 n \cdot q + r_1 n = (b_1 \cdot q + r_1)\, n$

quindi, per considerazioni analoghe alle precedenti, a è uguale al prodotto di n per un naturale, ossia n è divisore anche di a, quindi è divisore comune di a e b.

Naturalmente, la proposizione appena dimostrata equivale all'affermazione che l'insieme dei divisori comuni di a e b coincide con l'insieme dei divisori comune di b ed r; il massimo dei divisori comuni di a e b coincide quindi con il massimo dei divisori comuni di b ed r: $MCD(a,b) = MCD(b,r)$.

In parole semplici: per calcolare l'insieme dei divisori comuni (e quindi anche il MCD) di a e b, basta calcolare l'insieme dei divisori comuni (e quindi anche il MCD) di b ed r. Ma la stessa proprietà si può applicare di nuovo alla coppia (b,r), e chiamando r_1 il resto della divisione intera di b per r, passare a calcolare il MCD di r ed r_1, e cosí via.

Che vantaggio c'è? dove ci si ferma (se ci si ferma)? Osserviamo che per definizione il resto della divisione intera di due naturali è strettamente minore del divisore, nel nostro caso $r < b$, e poi, ripetendo il passo, $r_1 < r$, ecc. Abbiamo quindi una successione di coppie:

$$
\begin{array}{c}
(a, b) \\
(b, r) \\
(r, r_1) \\
(r_1, r_2) \\
(r_2, r_3) \\
\ldots
\end{array}
$$

i cui secondi elementi costituiscono una successione strettamente decrescente di naturali:

$$b > r > r_1 > r_2 > r_3 > \ldots$$

essa terminerà necessariamente con 0, quindi il nostro problema iniziale si ridurrà (dopo un numero finito di passi) a quello di trovare il MCD di una coppia di numeri $(R, 0)$; a questo punto la divisione intera di R per 0 non la possiamo piú fare, ma non ne abbiamo bisogno: il problema è risolto! Infatti, come abbiamo ricordato all'inizio, il MCD di R e 0 è semplicemente R.

Abbiamo cosí inventato lo schema generale dell'algoritmo, dimostrandone contemporaneamente (in modo molto informale) la correttezza. Bisogna ora trascriverlo in un programma. Nel far questo, la precedente discussione informale può trarre in inganno: l'aver detto infatti "prendiamo il resto r della divisione intera di a per b; poi consideriamo la nuova coppia (b,r), ecc." può indurre a pensare che si debba comunque fare $a \ mod \ b$, poi controllare il risultato, ecc., e quindi a scrivere un programma ad esempio della forma:

```
repeat
  r:= a mod b;
  a:= b;
  b:= r
until r = 0;
MCD:= a;
```

oppure della forma:

```
r:= a mod b;
while r<>0 do begin
  a:= b;
  b:= r;
  r:= a mod b
end;
MCD:= b;
```

Tali programmi sono però <u>scorretti</u>, perchè non funzionano nel caso in cui il valore iniziale di b sia 0 (*a mod 0* genera un errore al tempo di esecuzione), mentre la specifica del problema è che a e b siano semplicemente dei naturali non <u>entrambi</u> nulli. Ci può essere la tentazione di trattare questo come un caso particolare: *if b = 0 then ...* ma la

soluzione è inelegante, perchè cosí trattiamo come caso particolare quello al quale ci ridurremo in ogni caso!

La soluzione migliore ci viene, ancora una volta, ragionando per induzione su un ciclo *while*. Supponiamo dunque di aver eseguito k ripetizioni del ciclo, con k maggiore o uguale a zero; in che situazione ci troviamo e che cosa dobbiamo fare? L'algoritmo, come abbiamo visto, consiste nel ridurre il problema di calcolare il MCD di a e b a quello di calcolare il MCD di una certa altra coppia di numeri a' e b' (dove $b'<b$). Lo stato è perciò costituito da due variabili a e b, e il corpo del ciclo deve sostituire i loro contenuti con i nuovi valori, che sono rispettivamente b e il resto della divisione di a per b:

```
r:= a mod b;
a:= b;
b:= r
```

Se $b = 0$, il corpo non si può eseguire, ma il problema è risolto: e ciò vale sia che il corpo del ciclo sia stato eseguito tante (o poche) volte, sia che esso non sia stato eseguito nemmeno una volta. La condizione di uscita è quindi $b = 0$, e non c'è altro da fare!

```
function mcd(a,b: integer): integer;
var r: integer;
begin                       {a=A, b=B, (A>0, B>=0) or (A>=0, B>0)}
  while b <> 0 do begin     {INV: MCD(a,b) = MCD(A,B)}
    r:= a mod b;
    a:= b;
    b:= r
  end;
  mcd:= a;                  {CF: a = MCD(A,B)}
end.
```

Esercizio 5. Si espongano le precedenti argomentazioni nella forma ordinata di una dimostrazione di correttezza.

4.7.2 Analisi della complessità.

La proprietà che il resto della divisione è sempre strettamente minore del divisore ci ha permesso di dimostrare che l'algoritmo termina correttamente; nel far questo essa ci dà contemporaneamente un'informazione sul tempo di calcolo: infatti, se ad ogni passo successivo il valore di b diminuisce almeno di 1, l'algoritmo termina al peggio dopo b passi, cioè il tempo di calcolo non può essere peggio che proporzionale a b.

Tale informazione sulla velocità dell'algoritmo naturalmente è corretta, ma non è sufficientemente accurata; utilizzando infatti una meno nota proprietà dei numeri si può vedere (cioè dimostrare) che l'algoritmo termina in realtà piú velocemente, perchè il tempo di calcolo è nel caso peggiore proporzionale non a b ma soltanto al logaritmo di b.

La proprietà in questione è la seguente: *Nella divisione intera, se il dividendo non è minore del divisore, il resto è minore della metà del dividendo.*

Dimostrazione:
Siano a, b, q, r rispettivamente dividendo, divisore, quoziente, e resto.
Allora $a = q \cdot b + r$, cioè $r = a - q \cdot b$, dove $r < b$. Sia inoltre a ‡ b (dividendo non minore del divisore).

Consideriamo separatamente i due casi $b \le a/2$ e $b > a/2$.

caso (1)
$b \le a/2$: allora, essendo $r < b$, si ha $r < a/2$

caso (2)
$b > a/2$: allora è $a < 2b$, ma essendo anche $a \ge b$, abbiamo che b in a "ci sta una volta", cioè il quoziente q è 1; quindi $r = a - b$, ma poichè $b > a/2$, si ha $a - b < a - a/2$, cioè anche in questo caso $r < a/2$.
(fine della dimostrazione).

Consideriamo ora due passi successivi dell'algoritmo:

$$(\ldots, b_k)$$
$$(b_k, b_{k+1})$$
$$(\ldots, b_{k+2})$$

b_{k+2} è il resto della divisione di b_k per b_{k+1}; per la proprietà precedente è quindi:

$$b_{k+2} < b_k/2$$

Cosí sappiamo ora che il valore di b non solo diminuisce ad ogni passo, ma che ogni due passi si riduce almeno della metà. Se si riducesse almeno della metà ad ogni passo, l'algoritmo terminerebbe dopo un numero di passi non piú grande di $\log_2 b$ $(+\ 1)$; essendo invece il doppio piú lento, termina dopo un numero di passi non piú grande di circa $2 \cdot \log_2 b$.

Il tempo di calcolo cresce quindi in modo al piú logaritmico nel minore dei due argomenti.

4.8 Correttezza dei programmi con risultato booleano.

Consideriamo il problema di stabilire se un numero è primo. Stabilire se un ente gode di una certa proprietà, o se fra certi enti vale una certa relazione, vuol dire naturalmente calcolare un valore booleano; una soluzione del nostro non meglio specificato problema è dunque un pezzo di programma con le seguenti condizioni iniziale e finale (avendo scelto dei nomi per le variabili):

{CI: nella variabile `n` vi è un intero positivo $N \ge 2$}
{CF: nella variabile `primo` vi è il valore di verità della proposizione "N è primo"}

oppure, in modo leggermente piú formale:

{CI: (`n` $= N$) $\wedge$ $(N \ge 2)$}
{CF: `primo` $\leftrightarrow$ N è un numero primo}

Tale soluzione può poi, come al solito, essere concretizzata in un programma che legge un numero da tastiera e scrive un'opportuna frase di risposta sullo schermo, oppure - meglio - come un sottoprogramma `function primo(n: integer): boolean`.

Ricordiamo la definizione: un numero naturale maggiore di 1 si dice *primo* se non ha altri divisori all'infuori di se stesso e 1. Se indichiamo con la notazione (mutuata dal Pascal) *m..n* l'insieme dei numeri interi compresi fra *m* ed *n* (estremi inclusi), potremo scrivere: *un naturale n si dice primo se non ha divisori in 2 .. n-1.*

Realizziamo l'algoritmo nel modo piú ingenuo possibile, traducendo alla lettera in TurboPascal la definizione:

```
for i:= 2 to n-1 do
  if n è divisibile per i then begin primo:= false; exit end;
primo:= true  (* se esce normalmente *)
```

Scriviamo ora il corrispondente programma in Pascal Standard, senza uscite forzate; per questo bisogna trasformare il *for* in un *while*: l'inizializzazione `i:= 2` va portata fuori del ciclo, l'estremo superiore `n-1` diventa il test `while i<=n-1`, cioè `while i<n`; ma bisogna uscire anche se `n` è divisibile per `i`, quindi il ciclo è:

```
i:= 2;
while (i<n) and (n mod i <> 0) do i:= i+1;
```

All'uscita dal ciclo bisogna, per dare la risposta corretta, stabilire perchè si è usciti. Se si è usciti con `i=n`, allora non si sono trovati divisori e il numero è primo; se invece si è usciti con `i n`, quindi con `i<n`, allora si è usciti perchè `n mod i = 0` per un `i<n`, e quindi il numero non è primo. Il risultato è allora il valore dell'espressione booleana `i=n`; la soluzione del problema è quindi:

```
i:= 2;
while (i<n) and (n mod i <> 0) do i:= i+1;
primo:= i = n
```

Poichè il programma non altera il contenuto *N* di `n`, ignoriamo la distinzione fra *N* ed `n`. L'invariante è:

INV: *(2 £ i £ n)* Ù *(n non ha divisori in 2 .. i-1)*

Infatti per `i`=2 la proposizione INV è banalmente soddisfatta. Per dimostrare il passo, assumiamo che valgano INV e TEST, si ha allora:

(2 £ i £ n) Ù *(n mod i <> 0)* Ù *(n non ha divisori in 2 .. i-1)*

quindi

(2 £ i < n) Ù *(n non ha divisori in 2 .. i)*

Dopo aver eseguito il corpo del ciclo, cioè dopo aver incrementato `i`, l'invariante vale quindi ancora.

La condizione di uscita dal ciclo è:

CU (ossia ØTEST): *(i ‡ n)* Ú *(n mod i = 0)*

Nei programmi visti finora, come il massimo, o l'esponenziale, o il MCD, all'uscita del ciclo si aveva il risultato numerico in una variabile (`max`, o `ris`, o `mcd`), e quindi non c'erano piú altre istruzioni da eseguire, se non la restituzione del valore; la condizione alla fine del ciclo coincideva con la condizione finale, il problema era risolto.

In questo caso, invece, il booleano costituente il risultato finale non è contenuto in una variabile; è invece il valore di un'espressione booleana (cioè relazionale), la quale deve quindi essere individuata correttamente. Ad esempio, se sostituiamo l'espressione

i=n con l'espressione *n mod i <> 0*, otteniamo un programma errato il cui risultato è sempre false; infatti anche se si esce dal ciclo con *i=n*, si ha *n mod n = 0*, e quindi l'espressione booleana *(n mod i <> 0)* è falsa.

In altre parole, nell'istruzione di restituzione del risultato bisogna porre un'espressione booleana RIS che, <u>nello stato della memoria in quell'istante</u>, sia vera se la proposizione "N è primo" è vera, e sia falsa se la proposizione "N è primo" è falsa, cioè sia <u>logicamente equivalente</u> alla proposizione "N è primo", che è la condizione finale CF del problema. Con una formula logica:

$$(INV \wedge \neg TEST) \Rightarrow (RIS \Leftrightarrow CF)$$

Nel nostro caso si ha:

$$(INV \wedge CU) \Rightarrow (i=n \Leftrightarrow N \text{ è primo})$$

Un tale schema di proposizione può essere dimostrato in molti modi; ad esempio, esso è equivalente a:

$$\begin{array}{c} (INV \wedge CU) \Rightarrow (RIS \Rightarrow CF) \\ \wedge \\ (INV \wedge CU) \Rightarrow (\neg RIS \Rightarrow \neg CF) \end{array}$$

cioè infine:

$$\begin{array}{c} (INV \wedge CU \wedge RIS \Rightarrow CF) \\ \wedge \\ (INV \wedge CU \wedge \neg RIS \Rightarrow \neg CF) \end{array}$$

che è spesso la forma che si dimostra direttamente. Nel nostro caso:

1) $INV \wedge CU \wedge i=n \Rightarrow n$ *è primo*;
   infatti sostituendo *n* ad *i* nella seconda componente dell'invariante si ottiene proprio la proposizione "n non ha divisori in 2 .. n-1.

2) $INV \wedge CU \wedge i \neq n \Rightarrow n$ *non è primo;*
   infatti: $INV \wedge i \neq n \Rightarrow i < n$
   $i < n \wedge CU \Rightarrow n \bmod i = 0$
   quindi esiste un $i < n$ tale che $n \bmod i = 0$, dunque *n* non è primo.

4.9 Dimostrazioni e regole di inferenza.

Nelle dimostrazioni di correttezza abbiamo visto che un ruolo fondamentale giocano - com'è naturale - le dimostrazioni di proposizioni della forma:

se si esegue la sequenza di istruzioni SeqIstruz a partire da uno stato della memoria che soddisfa alla condizione PRE, lo stato della memoria che si ottiene alla fine dell'esecuzione - <u>se essa termina</u> - soddisfa alla condizione POST;

o, in notazione sintetica:

$$\{PRE\}\ \texttt{SeqIstruz}\ \{POST\}$$

Per dimostrare proposizioni di questo genere abbiamo fatto finora ricorso, oltre che alle regole generali della logica (classica), a schemi di ragionamento particolari che abbiamo ricavato dalla semantica del Pascal (e dall'aritmetica); fra questi vi è lo schema per il *while*, che usando la notazione sintetica si può riassumere cosí:
se si è dimostrato che: $\{INV \wedge TEST\}$ *Corpo* $\{INV\}$,
allora si è dimostrato che: $\{INV\}$ *while TEST do Corpo* $\{INV \wedge \neg TEST\}$

oppure, piú formalmente:

$$\{INV \wedge TEST\}\ Corpo\ \{INV\}$$

$$\Downarrow$$

$$\{INV\}\ \textit{while TEST do Corpo}\ \{INV \wedge \neg TEST\}$$

Abbiamo inoltre usato argomentazioni informali sugli effetti delle singole istruzioni, ad esempio: *se prima di eseguire l'istruzione* `i:= i+1` *vale l'uguaglianza* $ris=X^{i,}$ *allora dopo averla eseguita vale l'uguaglianza* $ris=X^{i-1}$. La forma di questa argomentazione, e di qualunque argomentazione analoga sull'istruzione di assegnazione, è evidentemente la seguente:

siano v una variabile (Pascal) e Espr un'espressione; se prima dell'esecuzione dell'istruzione v:= Espr il valore denotato da Espr gode di una proprietà P, cioè la proposizione P(Espr) è vera, allora dopo l'esecuzione dell'istruzione la proprietà P vale per (il contenuto della) variabile v, cioè la proprietà P si è "trasmessa all'indietro" dal lato destro al lato sinistro dell'operazione di assegnazione. In formula:

$$\{P(Espr)\}\ \texttt{v:= Espr}\ \{P(v)\}$$

Si ha cosí una regola di ragionamento sui programmi analoga a quella per il *while* (purchè si precisi bene la nozione di sostituzione di *v* con *Espr*, nozione in apparenza banale, in realtà contenente sottili difficoltà, come abbiamo già notato in altra occasione).

Potremmo, con lo stesso metodo, rendere del tutto espliciti i principi su cui sono fondate le nostre dimostrazioni; potremmo, in altre parole, formalizzare le dimostrazioni di correttezza di programmi come dimostrazioni di teoremi di una teoria - la *teoria dei programmi* - dotata di propri assiomi e regole di inferenza, esattamente come la geometria euclidea o la teoria dei gruppi o qualsiasi altra teoria matematica.

Osserviamo subito che, se da una parte una dimostrazione "informale" di correttezza del genere di quelle delineate finora è sempre - proprio per la sua informalità - soggetta alla possibilità di errore, una dimostrazione formale, cioè in cui tutti i passi del ragionamento siano esplicitati in base alle regole logiche e alle regole di inferenza della teoria, diventa estremamente lunga e noiosa e "meccanica", e perciò difficile da seguire (e quindi di nuovo soggetta ad errore umano) anche per il caso di programmi semplicissimi, come del resto si sarà intuito anche dagli esempi finora riportati. A maggior ragione è impensabile che si possano scrivere a mano dimostrazioni completamente formali di correttezza per programmi di dimensioni realistiche. Ciò vale, del resto, non solo per la teoria dei programmi, ma per qualunque teoria matematica.

Tuttavia, una dimostrazione completamente formale è "meccanicamente" controllabile, proprio nel senso che il suo controllo può essere effettuato molto meglio

da una macchina, cioè dal calcolatore. Si possono allora pensare di costruire dei sistemi interattivi che, in cooperazione con l'utente, dato un programma e la sua specifica cerchino di generarne la dimostrazione di correttezza, oppure che date le specifiche di un problema aiutino a costruire un programma corretto contemporaneamente alla dimostrazione della sua correttezza, ecc.

Ci si può chiedere chi dimostri la correttezza del dimostratore. La risposta può consistere nel ben noto metodo informatico del bootstrap, cioè del sollevarsi da terra tirandosi per i lacci delle scarpe: si scrive un piccolo dimostratore per un piccolo sottoinsieme del linguaggio dimostrandone a mano la correttezza, poi con esso si costruisce o si dimostra la correttezza di un dimostratore piú ampio, eccetera.

Lo studio delle dimostrazioni formali di correttezza esula dagli scopi di questo corso; nella prossima sezione diamo comunque, giustificandole informalmente, le regole per alcuni degli altri principali costrutti del Pascal.

4.10 Regole per altri costrutti del Pascal.

La differenza piú importante fra l'istruzione *while* e l'istruzione *repeat* è che in quest'ultima, a differenza che nella *while*, il corpo del ciclo viene eseguito almeno una volta; ricordiamo che un *repeat* è equivalente a un *while* preceduto da una esecuzione del corpo:

```
repeat Corpo until EsprBool
```

è equivalente alla sequenza di istruzioni:

```
Corpo;
while not EsprBool do Corpo
```

Per dimostrare per induzione che, se l'esecuzione del corpo mantiene l'invariante, allora l'invariante stesso vale all'uscita dell'istruzione *repeat*, non c'è bisogno di assumere che l'invariante valga prima della *repeat*: siccome in ogni caso il corpo viene eseguito almeno una volta, basta assumere che l'invariante valga dopo tale prima esecuzione. In altre parole, la prima esecuzione del corpo della *repeat* può fare le veci - anche solo parzialmente - delle istruzioni di inizializzazione. Nel caso del *while*, invece, ci si deve assicurare che alla fine l'invariante valga anche se il corpo non viene eseguito neppure una volta; esso deve quindi valere già all'inizio.

Insomma, nel caso della *repeat* l'induzione sul numero delle ripetizioni del ciclo è un'induzione con base 1 invece di 0.

Lo schema di ragionamento è allora il seguente (ricordando che il segno logico del test è nel *repeat* invertito rispetto al *while*). Data l'istruzione:

```
repeat Corpo until TEST
```

se dopo la prima esecuzione del *Corpo* vale *INV*, e se *{INV* Ù Ø*TEST} Corpo {INV}*, allora alla fine della *repeat* vale *INV*Ù *TEST*.

Quindi, per dimostrare una proposizione della forma:

{PRE} `repeat Corpo until TEST` {INV∧ TEST}

bisogna dimostrare *{PRE} Corpo {INV}* e *{INV* Ù Ø*TEST} Corpo {INV}*.

La regola è quindi rappresentabile graficamente come una regola di inferenza con due premesse ed una conclusione; la riportiamo insieme alle altre nella tabella riassuntiva sottostante (le regole per l'*if-then-else* e per la sequenza sono evidenti). Si noti che la

regola per l'assegnazione è, a differenza delle altre, una regola senza premesse, cioè un assioma.

ASSEGNAZIONE

$$\{P(Espr)\}\ \texttt{v := Espr}\ \{P(v)\}$$

SEQUENZA

$$\frac{\{PRE\}\ \textit{Istruz1}\ \{Q\} \qquad \{Q\}\ \textit{Istruz2}\ \{POST\}}{\{PRE\}\ \textit{Istruz1; Istruz2}\ \{POST\}}$$

IF-THEN-ELSE

$$\frac{\{PRE \wedge TEST\}\ \textit{RamoThen}\ \{POST\} \qquad \{PRE \wedge \neg TEST\}\ \textit{RamoElse}\ \{POST\}}{\{PRE\}\ \texttt{if}\ \textit{Test}\ \texttt{then}\ \textit{RamoThen}\ \texttt{else}\ \textit{RamoElse}\ \{POST\}}$$

WHILE

$$\frac{\{INV \wedge TEST\}\ \textit{Corpo}\ \{INV\}}{\{INV\}\ \texttt{while}\ \textit{Test}\ \texttt{do}\ \textit{Corpo}\ \{INV \wedge \neg TEST\}}$$

REPEAT

$$\frac{\{PRE\}\ \textit{Corpo}\ \{INV\} \qquad \{INV \wedge \neg TEST\}\ \textit{Corpo}\ \{INV\}}{\{PRE\}\ \texttt{repeat}\ \textit{Corpo}\ \texttt{until}\ \textit{Test}\ \{INV \wedge TEST\}}$$

La regola per il *repeat* può essere espressa in modo equivalente come una regola con una sola premessa:

REPEAT

$$\frac{\{PRE \vee (INV \wedge \neg TEST)\}\ \textit{Corpo}\ \{INV\}}{\{PRE\}\ \texttt{repeat}\ \textit{Corpo}\ \texttt{until}\ \textit{Test}\ \{INV \wedge TEST\}}$$

Abbiamo cosí costruito una "teoria assiomatica" dei programmi: è una teoria matematica i cui teoremi sono proposizioni della forma *{PRE}SeqIstruz{POST}*, le cui dimostrazioni potrebbero venire condotte, come si è detto, in maniera puramente formale "dimenticando il significato" di tali proposizioni, e applicando invece soltanto le regole della teoria elencate sopra (completate con quelle per tutti gli altri costrutti del linguaggio), oltre agli assiomi e alle regole generali della logica matematica e dell'aritmetica.

Affinchè ciò sia possibile, però, manca una regola, che non si riferisce a nessun costrutto Pascal particolare ed è del tutto ovvia, ma non è una regola generale della logica nè dell'aritmetica nè è da esse derivabile, giacchè "analizza" proposizioni della

forma *{PRE}Istruz{POST}*. È una regola generale per tutti i costrutti del linguaggio, la cosiddetta regola della conseguenza, con tre premesse e una conclusione:

CONSEGUENZA

$$\frac{PRE \rightarrow PRE1 \qquad \{PRE1\}\ \mathtt{Istruzioni}\ \{POST1\} \qquad POST1 \rightarrow POST}{\{PRE\}\ \mathtt{Istruzioni}\ \{POST\}}$$

Il lettore inesperto che voglia convincersi che la precedente non è una regola logica generale, basta che dimentichi il significato della forma *{P}Istr{Q}* e la consideri come una proposizione *P* Ñ$_{Istr}$ *Q* costituita con le proposizioni *P* e *Q* tramite un nuovo genere di connettivo Ñ$_{Istr}$ non interpretato.
Allora è evidente che da *P* fi *P1*, *P1* Ñ$_{Istr}$ *Q1*, *Q1* fi *Q*, non si può solo in base alla logica proposizionale classica dedurre *P* Ñ$_{Istr}$ *Q*; infatti il connettivo Ñ$_{Istr}$ non è il connettivo fi .

4.11 Dimostrazioni di correttezza e annotazioni.

Riassumendo, data la specifica di un problema di programmazione attraverso una condizione iniziale CI e una condizione finale CF, risolvere il problema correttamente vuol dire trovare un (sotto)programma *Prog* tale che (con notazione abbreviata intuitiva per la 2):

1) (si possa dimostrare che) {CI} *Prog* {CF}
2) (si possa dimostrare che) {CI} *Prog* {termina}.

La 1) è la proprietà di correttezza parziale, 1) e 2) costituiscono insieme la proprietà di correttezza totale.

Per costruire un tale programma e contemporaneamente la dimostrazione di correttezza parziale, bisogna individuare un percorso dalla condizione iniziale alla condizione finale attraverso condizioni intermedie, e scrivere le istruzioni che realizzano tali passaggi intermedi. Se ad esempio il programma è della forma:

```
istruzioni iniziali;
while test do begin
  corpo
end;
istruzioni finali
```

la dimostrazione di correttezza consiste, al livello piú alto, nel trovare una condizione (o asserzione) INV tale che:

{CI} `istruzioni iniziali` {INV}
{INV $\wedge$ *test*} `corpo` {INV}
{INV $\wedge\neg$ *test*} `istruzioni finali` {CF}

Tale schema di dimostrazione può essere conglobato direttamente nel programma, che cosí diventa (programma annotato):

```
{CI}
istruzioni iniziali;
{INV}
while test do begin
  {INV and test}
  corpo
  {INV}
end;
{INV and not test}
istruzioni finali
{CF}
```

Un programma con ciclo *repeat* si può scrivere nella forma:

```
{CI}
istruzioni iniziali;
{P}
repeat
    {P  or (INV and not test)}
    corpo
    {INV}
until test;
{INV and test}
istruzioni finali
{CF}
```

Volendo scendere nei dettagli della dimostrazione formale, un'istruzione *if-then-else* può essere annotata nel modo seguente:

```
{PRE}
if test then
  {PRE and test}
  ramo_then
  {POST}
else
  {PRE and not test}
  ramo_else
  {POST}
end;
{POST}
```

Una sequenza di istruzioni può essere annotata nel modo ovvio:

```
{PRE} Istruz1; {Q} Istruz2; {R} Istruz3 ... {POST}
```

(la posizione del punto e virgola non importa ...).

Naturalmente nella scrittura concreta, le asserzioni molto lunghe potranno essere scritte "fuori testo" e poi indicate nel testo del programma semplicemente con un nome (come INV per un invariante, o INV1 e INV2 per gl'invarianti di due cicli diversi, ecc.). Insomma la scelta dello stile materiale di un programma annotato è lasciata al lettore.

4.12 Un esempio di programma annotato.

Riportiamo il testo della procedura dell'esponenziale veloce, annotato ad ogni istruzione. Si noti che dove vi sono due commenti consecutivi senza un'istruzione fra di essi, il passaggio dall'uno all'altro è un passaggio puramente logico-aritmetico; dove invece fra

due commenti vi è un'istruzione, il passaggio è un'istanza della regola corrispondente a quel genere di istruzione.

Si osservino in particolare le annotazioni per le assegnazioni, riconoscendovi l'applicazione della relativa regola.

```
function fastexp(x:real; n: integer): real;
var ris: real;
    {CI: x=X, n=N, N>=0, X 0}
begin
    {x=X, n=N, N>=0, X 0, 1=1}
  ris:= 1;
    {x=X, n=N, N>=0, X 0, ris=1}
    {INV: x^n·ris=X^N, n>=0}
  while n>0 do begin
          {x^n·ris=X^N, n>=0, n>0}
    if n mod 2 = 0 then begin
                {x^n·ris=X^N, n>=0, n>0, n mod 2 = 0}
                {(x·x)^(n div 2)·ris = X^N, n>=0}
      x:= x*x; n:= n div 2
                {x^n·ris=X^N, n>=0}
    end
    else begin
                {x^n·ris=X^N, n>=0, n>0, n mod 2   0}
                {x^(n-1)·x·ris=X^N, n-1>=0}
      ris:= x*ris; n:= n-1
                {x^n·ris=X^N, n>=0}
    end
          {x^n·ris=X^N, n>=0}
  end
    {x^n·ris=X^N, n>=0, n<=0}
    {x^n·ris=X^N, n=0}
    {CF: ris=X^N}
```

In ognuno dei due rami del condizionale abbiamo trattato le due assegnazioni come se fossero contemporanee (in alcuni linguaggi di programmazione teorici esiste l'assegnazione contemporanea di variabili); se si volessero trattare una per una in modo sequenziale avremmo, nel ramo *then*:

```
                {x^n·ris=X^N, n>=0, n>0, n mod 2 = 0}
                {(x^2)^n ·ris^2 = X^2N, n>=0, n mod 2 = 0}
                {(x·x)^n ·ris^2 = X^2N, n>=0, n mod 2 = 0}
      x:= x*x;
                {x^n ·ris^2 = X^2N, n>=0, n mod 2 = 0}
                {x^(n div 2) ·ris = X^N, n>=0}
      n:= n div 2;
                {x^n ·ris = X^N, n>=0}
```

L'esempio conferma l'osservazione che un'annotazione cosí fine renderebbe i programmi illeggibili ed è "disumana", cioè non adatta alla progettazione e documentazione umana di programmi di dimensioni realistiche, cosí come, del resto, qualunque dimostrazione fortemente formalizzata in un qualsiasi campo della matematica (vedi *Lolli, Capire una dimostrazione, Il Mulino 1988*)

4.13 Complessità asintotica: definizioni.

4.13.1 La notazione asintotica nell'analisi matematica.

Definiamo rigorosamente le nozioni e la notazione che abbiamo in parte già usato per descrivere l'andamento del tempo di calcolo di un algoritmo.

Siano f e g due funzioni dai naturali ai reali non negativi. Si dice che:

g è O(f), oppure, con notazione abusiva, **g=O(f)** (pronuncia *g è "o grande" di f)*
se e solo se esistono due costanti $c>0$ *e* $n_0 \geq 0$ *tali che per tutti gli* $n \geq n_0$ *sia* $g(n) \leq c\, f(n)$.

Cioè g è $O(f)$ se e solo se, per n sufficientemente grande, è delimitata superiormente da un "multiplo" reale positivo di f, cioè se e solo se (per $n \to \infty$) **g cresce al piú come f**.

g è Ω(f), oppure, abusivamente, **g=Ω(f)** (pronuncia *g è "omega grande" di f)*
se e solo se esistono due costanti $c>0$ *e* $n_0 \geq 0$ *tali che per tutti gli* $n \geq n_0$ *sia* $g(n) \geq c\, f(n)$.

Cioè g è $\Omega(f)$ se e solo se, per n sufficientemente grande, è delimitata inferiormente da un "multiplo" reale positivo di f, cioè se e solo se (per $n \to \infty$) **g cresce almeno come f**.

g è Θ(f), oppure, abusivamente, **g=Θ(f)** (pronuncia *g è "theta grande" di f)*
se e solo se g è $O(f)$ *e g è* $\Theta(f)$*, cioè se esistono tre costanti* $c_1>0$, $c_2>0$, $n_0 \geq 0$ *tali che per tutti gli* $n \geq n_0$ *sia* $c_1 f(n) \leq g(n) \leq c_2 f(n)$.

Cioè g è $\Theta(f)$ se e solo se, per n sufficientemente grande, è delimitata inferiormente e superiormente da due "multipli" reali positivi (in generale distinti) di f, cioè se e solo se (per $n \to \infty$) **g cresce esattamente come f**.

La notazione con l'uguaglianza, adottata in queste dispense perchè piú comoda e di uso universale, è in realtà abusiva, perchè non si tratta di una vera uguaglianza: ad esempio dalle pseudouguaglianze $g = O(f)$ e $h = O(f)$ non si può dedurre $g = h$. La definizione rigorosa di O, Ω, Θ è come insiemi:

O(f) è l'insieme di tutte le funzioni g tali che ...

La notazione $g=O(f)$ va allora letta come $g \in O(f)$, ecc.

4.13.2 Alcune proprietà.

Proprietà transitiva. Se f è $O(g)$, e g è $O(h)$, allora f è $O(h)$; analogamente per Ω e Θ.

Una proprietà dei logaritmi.
Se $f(x) = O(\log_a x)$, oppure $f(x) = \Omega(\log_a x)$, oppure $f(x) = \Theta(\log_a x)$, allora è anche rispet-tivamente $f(x) = O(\log_b x)$, oppure $f(x) = \Omega(\log_b x)$, oppure $f(x) = \Theta(\log_b x)$ per qualunque altro b, con $a > 1$ e $b > 1$.

Per indicare che una funzione f ha crescita non piú che logaritmica possiamo quindi legittimamente scrivere $f(x) = O(\log x)$, senza precisare la base.

Un'analoga proprietà di indifferenza rispetto alla base non vale per l'esponenziale. Il fatto che sia $f(x) = O(a^x)$ o $f(x) = \Omega(a^x)$ <u>non</u> implica che sia rispettivamente $f(x) = O(b^x)$ o $f(x) = \Omega(b^x)$ per $b \neq a$. Anzi, si ha la seguente
Proprietà. Se $f(x)$ è $\Theta(a^x)$, e se $b \neq a$, allora $f(x)$ <u>non è</u> $\Theta(b^x)$.

Osservazione.
Il fatto che una funzione $y=f(x)$ sia $\Theta(\log x)$ non implica che la funzione inversa $x=f^{-1}(y)$, se esiste, sia $\Theta(a^y)$ per qualche a, ma soltanto che esistono a e b (in generale distinti), con $a \leq b$, tali che $f^{-1}(y)=\Omega(a^y)$ e $f^{-1}(y)=O(a^y)$; cioè che $f^{-1}(y)$ è delimitata inferiormente e superiormente da due esponenziali, in generale di basi diverse.

In tal caso non si può, a rigore, scrivere $f^{-1}(y) = \Theta(a^y)$, ma si può comunque dire che la crescita di f è *strettamente esponenziale*, il che è di solito tutto quanto ci interessa (ad esempio, se un algoritmo ha complessità temporale esponenziale nel senso precedente, questo ci basta per concludere che si tratta di un algoritmo di scarsa utilità).

4.13.3 Complessità di algoritmi e problemi: definizioni.

Sia A un algoritmo, espresso in un linguaggio astratto, oppure in particolare un programma scritto in un qualche linguaggio di programmazione.

Sia n la dimensione complessiva (variabile) dei dati in ingresso per A, espressa in unità convenienti (ad es. per un algoritmo avente per input un vettore, n è la lunghezza del vettore).

Sia $T(n)$ il tempo (in unità arbitrarie) di esecuzione dell'algoritmo, per dati in ingresso di dimensione n. Sia $S(n)$ lo spazio usato dall'algoritmo <u>in aggiunta</u> allo spazio occupato dai dati in ingresso.

$T(n)$ (o $S(n)$) di solito non dipende solo da n, ma anche da altri parametri, cioè dalla particolare configurazione dei dati (di dimensione n); la scrittura $T(n)$ (o $S(n)$) per poter denotare un valore univoco deve quindi essere precisata. Siano:

$$T_{peggio}(n)$$

il tempo di esecuzione dell'algoritmo per il caso peggiore fra tutti quelli di dimensione n.

$$T_{meglio}(n)$$

il tempo di esecuzione dell'algoritmo per il caso migliore fra tutti quelli di dimensione n.

$$T_{medio}(n)$$

la media dei tempi di esecuzione dell'algoritmo su tutti i possibili casi di dimensione n. (tale definizione richiede che si assuma una distribuzione di probabilità o di frequenza dei diversi casi possibili).

Definizioni analoghe si danno per la complessità spaziale.

Delimitazione superiore (upper bound).

Si dice che un <u>algoritmo</u> o un <u>programma</u> ha *complessità temporale (o spaziale) asintotica del caso peggiore - o risp. migliore o medio -*

$$O(f(n))$$

se il suo tempo (o spazio) di esecuzione $T_{peggio}(n)$ (o$S_{peggio}(n)$) - o risp. $T_{meglio}(n)$ *o* $T_{medio}(n)$ - è $O(f(n))$.

Si dice che un problema algoritmico (o problema di programmazione) ha *complessità temporale (o spaziale) asintotica del caso peggiore (o risp. migliore o medio)*

$$O(f(n))$$

se esiste (è stato trovato) un algoritmo risolvente di complessità temporale (o spaziale) del caso peggiore (o risp. migliore o medio) $O(f(n))$.

Delimitazione inferiore (lower bound).

Si dice che un algoritmo ha complessità temporale o spaziale del caso peggiore (o risp. migliore o medio)

$$\Omega(f(n))$$

se il suo tempo o spazio di esecuzione $T_{peggio}(n)$ (o risp. $T_{meglio}(n)$ *o* $T_{medio}(n)$) è $\Omega(f(n))$.

Si dice che un problema algoritmico ha complessità del caso peggiore (o risp. migliore o medio)

$$\Omega(f(n))$$

se (si è dimostrato che) qualunque algoritmo che risolva il problema deve avere complessità del caso peggiore (o risp. migliore o medio) $\Omega(f(n))$.

(Definizioni analoghe si danno per la complessità spaziale).

Osservazioni.
Quando si parla di complessità $T(n)$ di un algoritmo senza altra specificazione, di solito si intende la complessità del caso peggiore. In particolare, si usa spesso dire che un algoritmo *ha complessità temporale* $O(f(n))$ per intendere che l'algoritmo ha complessità temporale del caso peggiore $O(f(n))$. La locuzione è giustificata dal fatto che in tal caso $T(n)$ cresce al piú come $f(n)$, qualunque sia il particolare input di dimensione n per ciascun valore di n.

Analogamente, si usa spesso dire che un algoritmo ha complessità temporale $\Omega(f(n))$ per intendere che l'algoritmo ha complessità temporale del caso migliore $\Omega(f(n))$. La locuzione è giustificata dal fatto che in tal caso $T(n)$ cresce almeno come $f(n)$, qualunque sia il particolare input di dimensione n per ciascun valore di n.

Problema algoritmicamente chiuso.
Un problema algoritmico si dice *chiuso* se:
1) esiste (è stato trovato) almeno un algoritmo risolvente di complessità $O(g(n))$;
2) si è dimostrato che qualunque possibile algoritmo risolvente deve avere complessità $\Omega(g(n))$.

L'algoritmo esistente di complessità $O(g(n))$ è quindi, dal punto di vista asintotico, ottimale; rispetto ad esso si possono avere miglioramenti solo per costanti moltiplicative o additive.

Problema algoritmicamente aperto (o con gap algoritmico).
Un problema algoritmico si dice *aperto* se:
1) il miglior algoritmo risolvente conosciuto ha una complessità $O(g(n))$;
2) si è dimostrato che qualunque possibile algoritmo risolvente deve avere complessità $\Omega(f(n))$, con f diverso da g.

In tal caso può darsi che in futuro si riesca a trovare un algoritmo di complessità inferiore, oppure che si riesca a dimostrare che il problema ha una delimitazione inferiore piú grande, o entrambe le cose.

4.14 Un esempio di risoluzione ottima di un problema.

Nel Capitolo 1 è stato enunciato un principio molto generale di buona programmazione: la necessità di evitare calcoli inutili, quindi l'importanza di memorizzare i risultati di computazioni intermedie (anzichè rifare le computazioni stesse) quando tali risultati siano riutilizzati o riutilizzabili. Gli esempi là riportati, non contenendo cicli ma soltanto semplici espressioni, non erano però molto significativi dal punto di vista del risparmio di tempo; d'altra parte l'uso del principio si riduceva al fatto banale di memorizzare (in variabili intermedie) le sottoespressioni incontrate (o meglio, valutate) piú di una volta.

Vogliamo ora far vedere come, nei programmi iterativi, il principio suddetto sia applicabile in casi in cui la ridondanza del calcolo può non essere evidente, e permetta una vera riduzione della complessità asintotica, cioè dell'ordine di infinito della funzione T(n).

Si supponga di aver definito o di avere disponibile la nota funzione Pascal per il calcolo del fattoriale (con restituzione di un risultato di tipo real invece che integer, per consentire il calcolo - anche se approssimato - del fattoriale di numeri anche abbastanza "grandi"):

```
function fact(n: integer): real;
var ris: real; i: integer;
begin
  ris:= 1;
  for i:= 2 to n do ris:= i*ris;
  fact:= ris
end;
```

Si voglia poi definire una funzione Pascal *f* che calcoli la sommatoria:

$$\sum_{i=1}^{n} 1/i!$$

La soluzione piú naturale è quella di scrivere un ciclo che richiami la funzione *fact*:

```
function f(n: integer): real;
var ris: real; i: integer;
begin
  ris:= 0;
  for i:= 1 to n do ris:= ris + 1/fact(i);
  f:= ris;
end;
```

La procedura è corretta; l'invariante del ciclo è:

INV: $ris = 1/1! + 1/2! + ... + 1/(i-1)!$

La condizione di uscita dal ciclo è *CU:* $i = n+1$

La congiunzione di INV e CU ha come conseguenza:

$$ris = 1/1! + 1/2! + ... + 1/n!$$

Analizziamo la complessità temporale della funzione. Il tempo di calcolo di $f(n)$ è proporzionale alla somma degli n tempi di calcolo di $fact(1)$, $fact(2)$, ..., $fact(n)$. Ma il tempo di calcolo di $fact(i)$ è proporzionale ad i. Abbiamo allora:

$$T_f(n) = T_{fact}(1) + T_{fact}(2) + ... + T_{fact}(n) = 1 + 2 + ... + n = n(n+1)/2 = \text{Q}(n^2)$$

Se invece di invocare la funzione *fact* ne espandiamo direttamente le istruzioni nella funzione *f*, si ha un marginale risparmio di tempo dovuto all'eliminazione dei tempi impiegati dal meccanismo della chiamata, ma la complessità asintotica ovviamente non cambia :

```
function f(n: integer): real;
var ris, fact: real; i,j: integer;
begin
  ris:= 0;
  for i:= 1 to n do begin
    fact:= 1;
    for j:= 2 to i do fact:= j*fact;
    ris:= ris + 1/fact;
  end;
  f:= ris;
end;
```

Osservando tali soluzioni, notiamo che in esse il calcolo di ciascun $i!$ viene rifatto per intero ad ogni iterazione del ciclo esterno, pur avendo calcolato $(i-1)!$ all'iterazione precedente, da cui quindi il valore di $i!$ si potrebbe ottenere con una semplice moltiplicazione per i.

Allora si può tenere in una variabile *fact* l'ultimo fattoriale calcolato; l'invariante significativo diventa:

INV: $ris = 1/1! + 1/2! + ... + 1/(i-1)!$ Ù $fact = (i-1)!$

La condizione di uscita è la stessa di prima. Il nuovo invariante viene mantenuto dal seguente corpo del ciclo (*while*):

```
fact:= i*fact; (* cosí si calcola i! *)
ris:= ris + 1/fact; (* si somma il nuovo termine *)
i:= i+1;
```

Naturalmente la variabile `fact` va anch'essa opportunamente inizializzata, per assicurare che l'invariante sia vero prima dell'esecuzione del ciclo (cioè prima della prima esecuzione del corpo del ciclo); in particolare, sostituendo $i=1$ nell'invariante si ottiene $fact = 0! = 1$. Il sottoprogramma completo è quindi (usando un ciclo *for*):

```
function f(n: integer): real;
```

```
var ris, fact: real; i: integer;
begin
  ris:= 0; fact:= 1;
  for i:= 1 to n do begin
    fact:= i*fact;
    ris:= ris + 1/fact;
  end;
  f:= ris;
end;
```

La nuova soluzione è costituita da un unico ciclo *for* nel cui corpo si effettuano soltanto operazioni di durata costante: il tempo di calcolo è quindi proporzionale ad *n*.

Siamo pertanto passati da una soluzione di complessità asintotica quadratica ad una di complessità asintotica lineare ($\Omega(n)$); la seconda soluzione è quindi di gran lunga preferibile.

Esercizio 6.

Si calcoli il valore approssimato della funzione trigonometrica *sin x* per mezzo del suo sviluppo in serie troncato dopo $n+1$ termini, dove *n* sia un parametro della funzione. Cioè:

$$sin(n,x) = x - x^3/3! + x^5/5! - x^7/7! + ... + (-1)^n x^{2n+1}/(2n+1)!$$

Capitolo 5
Programmazione iterativa con i vettori.

5.1 I tipi vettore.

Chiamiamo informalmente *vettori* gli array monodimensionali, per distinguerli dalle "matrici" che sono invece array bi- o pluri-dimensionali.

In questo capitolo studieremo, attraverso un certo numero di esempi, alcune tecniche di programmazione sui vettori; definiremo pertanto dei sottoprogrammi (procedure o funzioni) che operano su vettori risolvendo dati problemi, scrivendo poi (o lasciando come esercizio al lettore) dei programmi principali che ci permettano di "provare" tali sottoprogrammi.

Osserviamo a tal fine che in Pascal un tipo-vettore deve avere una dimensione costante e fissata al momento della scrittura del programma (cioè a "tempo di compilazione").

Naturalmente quando si scrivono programmi che operano su vettori non è conveniente esprimere all'interno del programma la lunghezza del vettore direttamente con il suo valore numerico: infatti se poi si decide di cambiare tale lunghezza (ad esempio perchè prima si è provato il funzionamento con vettori "corti", e poi si vuole usare il programma con vettori piú lunghi) bisogna andare a sostituire ovunque nel programma il vecchio valore con il nuovo, con tutto lo spreco di tempo e le possibilità di errore che ciò comporta.

Conviene invece definire una costante (ad esempio *n*, o un altro nome piú significativo), e usare poi tale costante. Ad esempio:

```
const n = ...
type elemtype = ...
     vectortype = array[1..n] of elemtype;
...
  for i:= 1 to n do ...
```

In questo modo, se si vuol cambiare la dimensione del vettore, basta cambiare la definizione della costante, cioè sostituire la vecchia lunghezza con la nuova in un solo punto!

Poichè in Pascal vettori di lunghezze diverse appartengono per definizione a tipi diversi, se in un programma si vogliono usare contemporaneamente vettori di differenti lunghezze, tutti i sottoprogrammi che operano su vettori devono essere definiti in tante versioni differenti quante sono le possibili lunghezze dei vettori; ad esempio, se vogliamo operare contemporaneamente su vettori di lunghezza 10, 25 e 30, e vogliamo per comodità crearci una procedura *leggivettore* per immettere da tastiera i valori degli elementi, siamo costretti a crearci tre versioni di tale procedura, una per ciascun tipo:

```
const m = 10; n = 25; p = 30;
type tipoelem = ...
     vettore1 = array[1..m] of tipoelem;
     vettore2 = array[1..n] of tipoelem;
     vettore3 = array[1..p] of tipoelem;

  procedure leggivettore1(var v: vettore1);
  var i: integer;
```

```
    begin
      for i:= 1 to m do read(v[i]);
      readln
    end;

    procedure leggivettore2(var v: vettore2);
    var i: integer;
    begin
      for i:= 1 to n do read(v[i]);
      readln
    end;

    procedure leggivettore3(var v: vettore3);
    var i: integer;
    begin
      for i:= 1 to p do read(v[i]);
      readln
    end;

  var a,b: vettore1;
      c: vettore2;
      v3: vettore3;
      ...

  begin
    leggivettore1(a);
    leggivettore1(b);
    leggivettore2(c);
    ...
```

5.1.1 Parametri array aperti.

Nel TurboPascal 7 l'inconveniente sopra segnalato può essere evitato definendo procedure e funzioni con parametri *array aperti* (*open-array parameters*), simili ai parametri di tipo array del linguaggio C: un parametro formale *array aperto* è un parametro di tipo array di cui si dichiara soltanto il tipo (ma non il numero) degli elementi, con la sintassi *array of NomeTipo*. Esempio:

```
procedure leggivettore(var v: array of tipoelem); ...
```

L'argomento effettivo corrispondente ad un parametro array aperto può essere un array dello stesso tipo di elementi ma di qualunque lunghezza; potremo quindi usare un'unica procedura *leggivettore* sia per vettori di tipo *vettore1* (cioè di lunghezza 10), sia per vettori di tipo *vettore2*, ecc.

Attenzione però: all'interno della procedura o funzione, il parametro formale array aperto viene considerato, come in C, un vettore indiciato da 0 a *n-1*, dove *n* è la lunghezza del parametro effettivo, che può essere ottenuta per mezzo della primitiva *SizeOf*, mentre la primitiva *High* restituisce l'indice dell'ultimo elemento, cioè *n-1*; anche i cicli devono perciò essere da 0 a *n-1*.

Esempio:

```
const m = 10; n = 25; p = 30;
type tipoelem = ...
     vettore1 = array[1..m] of tipoelem;
     vettore2 = array[1..n] of tipoelem;
     vettore3 = array[1..p] of tipoelem;
     vettore4 = array[m..n] of tipoelem;
```

```
    procedure leggivettore(var v: array of tipoelem);
    var i: integer;
    begin
      for i:= 0 to High(v) do read(a[i]);
      readln
    end;

  var a,b: vettore1;
      v4: vettore4;
      ...
  begin
    leggivettore(a);
    leggivettore(v4);
    ...
```

Si noti che il primo elemento del vettore *a* è *a[1]*, ma all'interno della procedura *leggivettore* esso viene visto come *v[0]*, il secondo elemento è *a[2]* ma all'interno di *leggivettore* viene visto come *v[1]*, e cosí via. Analogamente, nella seconda chiamata di *leggivettore*, con parametro effettivo *v4*, il primo elemento di *v4* è *v4[10]*, il secondo è *v4[11]*, ecc., ma all'interno di leggivettore essi sono visti rispettivamente come *v[0]*, *v[1]*, ecc.

Si osservi infine che gli array aperti sono solo una particolare forma di parametri, ma non sono tipi nel senso ordinario del termine: non è infatti possibile nè dichiarare variabili di tipo array aperto, nè scrivere definizioni di tipi che contengano array aperti, per la semplice ragione che la loro dimensione non sarebbe determinata al tempo di compilazione, come invece deve essere per tutte le variabili (e per i loro "stampi", ossia i tipi)!

5.1.2 Vettori di lunghezza effettiva variabile o vettori "parzialmente riempiti".

Il meccanismo degli array aperti permette di scrivere sottoprogrammi che possono operare su vettori di dimensione arbitraria, ma non altera il fatto che ogni variabile di tipo array deve avere una dimensione fissata, nota al momento della scrittura del programma.

Ciò può costituire un problema nel caso si voglia utilizzare un vettore per memorizzare una sequenza di elementi la cui lunghezza sarà determinata soltanto durante l'esecuzione, ad esempio una sequenza di elementi immessa da tastiera e terminata da CTRL/Z, oppure la sequenza di tutti quegli elementi di un altro vettore che soddisfano ad una data condizione, ecc.

Il modo per risolverlo è semplice, anche se comporta un piú o meno grande spreco di memoria: si definisce un tipo di vettore con una dimensione *nmax* abbastanza grande da poter contenere quelle che si prevedono essere le sequenze piú lunghe; le sequenze di lunghezza inferiore alla massima riempiranno "solo parzialmente" il vettore. Ad ogni variabile di tipo vettore dovrà quindi essere associata una variabile di tipo intero che contenga la dimensione "effettiva" del vettore, cioè la lunghezza della "parte occupata", ossia - per vettori indiciati a partire da 1 - l'indice dell'ultimo elemento occupato.

Tutte le procedure e funzioni dovranno poi, per ogni parametro di tipo vettore, avere un corrispondente parametro di tipo integer in cui ricevere o restituire o modificare la lunghezza "effettiva" del vettore.

Nelle sezioni seguenti assumeremo, a seconda dei contesti, che sia stato definito un opportuno "tipo vettore" di lunghezza *n*, da considerarsi "tutto occupato", con una dichiarazione della forma:

```
const n = ...
tipoelem = ...
vettore = array[1..n] of tipoelem;
```

oppure che sia stato definito un "tipo vettore" di lunghezza massima *nmax*, con una dichiarazione della forma:

```
const nmax = ...
tipoelem = ...
vettore = array[1..nmax] of tipoelem;
```

e la lunghezza effettiva *n* sia ogni volta passata come parametro.
In questo secondo caso per semplicità ometteremo, all'interno dei programmi e sottoprogrammi, il controllo che la "dimensione effettiva" non superi la dimensione massima.

5.2 Vettori di record.

Nelle applicazioni reali, le sequenze con cui si vuole operare sono spesso non puramente di numeri o altri tipi semplici, ma di elementi contenenti informazioni varie, cioè record formati da vari campi: cosí si potrà avere, ad esempio, un array di studenti in cui ogni elemento è un record contenente due campi *cognome* e *nome* di tipo *string*, un campo numerico contenente l'anno di nascita, magari un vettore con i voti degli esami, ecc.

```
...
type studente = record
                  matr: tipomatr;
                  cognome: string[50];
                  nome: string[50];
                  annonasc: 1900..2000;
            esami: array[1..44] of record
                                         titolo: string[30];
                                         voto: 1..30;
                                         lode: boolean;
                                       end;
                  num_esami: integer;
                  media: real;
                  ...
                end;

     vettstudenti: array[1..n] of studente;
...
```

Spesso, uno dei campi del record è (univocamente) identificativo di tutto il record: un esempio è, nel caso degli studenti, il campo *matr* contenente il numero di matricola. Un campo siffatto è detto "chiave" (in inglese "key", pronuncia "kii"); si può anche pensare che vi siano piú campi che funzionano da chiave, magari uno principale per il quale non sono possibili valori duplicati (ad es. la matricola), ed uno secondario per cui invece questi sono ammessi (ad es. il cognome), ecc.

Per semplicità, e perchè il lettore si possa piú facilmente concentrare sugli aspetti essenziali della programmazione coi vettori, negli algoritmi ed esempi di questo capitolo supporremo quasi sempre di avere a che fare con vettori di elementi di tipo semplice, salvo eventualmente richiamare, ove necessario, le considerazioni di cui sopra.

5.3 Iterazione sui vettori.

I problemi di programmazione sui vettori possono spesso venire risolti per mezzo dell'iterazione, cioè per mezzo di cicli *while* o *for* in cui uno o piú variabili-indice, tradizionalmente chiamate i, j, k, ecc., scandiscono in vario modo il vettore (o i vettori).

Come nel caso dei problemi numerici visti nei capitoli precedenti, per scrivere un programma che risolva correttamente un problema sui vettori è vivamente consigliabile pensare <u>non ai passi iniziali</u>, ma alla situazione del vettore e dei vari indici (cioè allo stato dell'esecutore Pascal) <u>al passo generico</u> del ciclo, aiutandosi con un disegno del vettore in cui l'indice o gl'indici in uso siano chiaramente posizionati; tale disegno non è altro, come vedremo, che una rappresentazione grafica dell'invariante del ciclo; da esso si potrà spesso capire quale dev'essere la condizione di uscita dal ciclo, cioè (per negazione) la condizione del *while* o l'estremo superiore del *for*, e quali istruzioni devono comporre il corpo del ciclo.

All'uscita dal ciclo sarà spesso necessario, per raggiungere la soluzione o restituire il risultato corretto, eseguire ancora qualche istruzione.

5.4 Terminologia e notazione.

Introduciamo la terminologia e notazione che useremo nelle sezioni successive per ragionare su programmi che operano su vettori.
Sia v una variabile di tipo vettore, ad esempio

```
var v: array[1..10] of ...
```

Chiamiamo *segmento* di v, o *sottovettore* di v, una qualunque sequenza, eventualmente vuota, di elementi consecutivi di v, ad esempio la sequenza $v[3], v[4], v[5], v[6]$, che indicheremo sinteticamente con la notazione $v[3..6]$.

In generale il segmento di vettore v costituito dalla sequenza di elementi consecutivi da $v[i]$ a $v[j]$ <u>compresi</u> sarà indicato dalla notazione $v[i..j]$.

Se è $i=j$ il segmento $v[i..j]$ è un segmento contenente un solo elemento; se è $i>j$ il segmento $v[i..j]$ - ad es. $v[5..2]$ - è un segmento vuoto.

Chiamiamo *lunghezza* di un segmento di vettore il numero dei suoi elementi; la lunghezza di un segmento $v[i..j]$ è data perciò da $j-i+1$ se *i£j*, è uguale a 0 altrimenti.
Esempio: la lunghezza di $v[1..n]$ è n, la lunghezza di $v[0..n]$ è $n+1$, ecc.

<u>Nota Bene.</u>

La notazione precedente per i segmenti di vettore non fa parte della sintassi Pascal o TurboPascal, ma soltanto del metalinguaggio informale con cui in queste note esprimiamo ragionamenti su programmi Pascal.

5.5 Ricerca sequenziale in un vettore non ordinato.

Problema: Definire una funzione la quale, presi come argomenti un valore *x* di un dato tipo, ed un vettore *v* (non ordinato) di elementi dello stesso tipo, indiciato da *1* ad *n*, restituisca *true* o *false* a seconda che *x* compaia o no nel vettore.

5.5.1 Considerazioni generali e soluzione TurboPascal.

Nei casi semplici come il precedente la formulazione di un problema sui vettori specifica già esplicitamente o implicitamente la successione di azioni che si devono compiere; il problema consiste nello scrivere un programma la cui esecuzione produca proprio tale successione di azioni. Ad esempio nel nostro caso è chiaro che il valore *x* dovrà essere successivamente confrontato con gli elementi del vettore *v[1]*, *v[2]*, ... finchè si trova un *v[k] = x*, oppure si è arrivati alla fine del vettore senza trovarlo.

In TurboPascal 7 il modo piú semplice per ottenere tale sequenza di azioni consiste nello scrivere un ciclo *for* da *1* ad *n*, nel corpo del quale si esegue il confronto fra *x* e *v[i]*; se *x = v[i]* si interrompe il ciclo e l'esecuzione della funzione restituendo *true*. Se il ciclo termina normalmente si restituisce *false*.

```
function ricerca(var v: vettore; x: tipoelem): boolean;
var i: integer;
begin
  ricerca:= false;
  for i:= 1 to n do if v[i]=x then begin ricerca:= true; exit end;
end;
```

Si può desiderare che la funzione restituisca, invece di un booleano, l'indice dell'elemento cercato, anzi - nel caso che vi possano essere elementi duplicati - l'indice della prima occorrenza dell'elemento cercato, e restituisca un "indice inesistente" - ad esempio 0 - se l'elemento non viene trovato:

```
function ricerca(var v: vettore; x: tipoelem): integer;
var i: integer;
begin
  ricerca:= 0;
  for i:= 1 to n do if v[i]=x then begin ricerca:= i; exit end;
end;
```

Questo può essere utile quando si effettua in un vettore di record una ricerca in base al valore della chiave, per poi effettuare sul record trovato elaborazioni diverse; ad esempio:

```
function ricerca(var v: vettstudenti; x: tipomatr): integer;
var i: integer;
begin
  ricerca:= 0;
  for i:= 1 to n do
    if v[i].matr = x then begin ricerca:= i; exit end;
end;
```

```
...
var m: tipomatr;
       studenti: vettstudenti;
       k, i: integer;
...
begin
...
  k:= ricerca(studenti,m);
  if k <> 0 then
    with studenti[k] do begin
      inc(num_esami);
      esami[num_esami].titolo:= 'Programmazione 1';
      esami[num_esami].voto:= 30;
      esami[num_esami].lode:= false;
      for i:= 1 to num_esami do
      somma:= somma + esami[i].voto;
      media:= somma/num_esami;
    end
...
```

Usando il costrutto *with* abbiamo evitato di ripetere l'espressione che dà accesso al record (e di far ricalcolare all'esecutore il suo indirizzo). Si noti che senza l'uso del *with* l'istruzione `esami[num_esami].titolo:= 'Programmazione 1'` (come le altre per il voto, ecc.) si sarebbe dovuta scrivere:

```
studenti[k].esami[studenti[k].num_esami].titolo:= ...
```

oppure, usando una variabile:

```
n_es:= studenti[k].num_esami;
studenti[k].esami[n_es].titolo:= 'Programmazione 1';
```

5.5.2 Soluzione TurboPascal senza uscite forzate.

Vi è un modo di risolvere il problema che sebbene anch'esso non corretto in Pascal Standard tuttavia - non facendo uso di uscite forzate - è piú aderente allo stile di programmazione Pascal, ed è inoltre facilmente trattabile dal punto di vista della dimostrazione di correttezza.

Ricordiamo che un ciclo *for i:= 1 to n* è equivalente a un ciclo *while i <= n* (attenti alla disuguaglianza debole <= !)

Si esce quindi dal ciclo quando *i > n* (cioè in realtà con *i=n+1*); bisogna però uscire anche se si è trovato il valore cercato, cioè se *v[i] = x*. La condizione di uscita è quindi *(i>n)* Ú *(v[i]=x)*; la condizione (di continuazione) del *while* è allora la sua negata, cioè - applicando la legge di deMorgan - *(i* £ *n)* Ù *(v[i]* „ *x)*.

Inizializzazione e ciclo sono allora:

```
i:= 1;
while (i <= n) and (v[i] <> x) do i:= i+1;
```

In Pascal standard la porzione di programma costituita dalle due righe precedenti, benchè sintatticamente corretta (cioè non generante errori di compilazione) è <u>logicamente</u> errata; poichè infatti nello Standard gli operandi di un'espressione booleana vengono sempre tutti valutati, la disuguaglianza `v[i]<>x` viene valutata anche quando `i` raggiunge il valore *n+1*, il che è non solo inutile, ma scorretto: infatti si accede cosí ad una locazione

$v[n+1]$ logicamente inesistente, e se fra le opzioni del compilatore è stato attivato il controllo del range (*range check*) si genera un errore di esecuzione (*index out of range*). Invece con la valutazione "cortocircuitata" dell'*and* se il primo operando della congiunzione vale *false*, cioè se è $i > n$, non si valuta piú il secondo operando.

Bisogna ora scrivere l'istruzione che, dopo l'uscita dal ciclo, restituisce il corretto risultato della funzione. Consideriamo dapprima la versione che restituisce un booleano: *true* se x è stato trovato, *false* se non è stato trovato.

Osserviamo che se all'uscita dal ciclo è *i* £ *n* allora si deve essere usciti a causa dell'altra condizione, cioè perchè è $v[i] = x$, dove appunto i non è superiore ad n. Quindi il valore x è uguale ad un elemento effettivo del vettore; se invece è $i > n$, allora fino ad $i=n$ (compreso) il valore x non è stato trovato, e quindi l'elemento cercato non c'è.

Il valore booleano che deve essere restituito è quindi semplicemente il valore dell'espressione booleana $i <= n$.

Ecco dunque la procedura completa:

```
function ricerca(var v: vettore; x: tipoelem): boolean;
var i: integer;
begin
  i:= 1;
  while (i <= n) and (v[i] <> x) do i:= i+1;
  ricerca:= i<=n
end;
```

<u>Attenzione</u>: potrebbe forse sembrare equivalente, e piú intuitivo, restituire invece del risultato del confronto $i <= n$ il risultato del confronto $v[i] = x$, cioè:

```
...
  ricerca:= v[i] = x
```

Tale soluzione <u>è però del tutto errata</u> (anche in TurboPascal)! Infatti, se i vale $n+1$ la cella $v[i]$ logicamente non esiste: se il controllo del range è attivato si ha la generazione di un errore al tempo di esecuzione; se il controllo non è attivato, il valore x viene dagli ordinari esecutori (Turbo)Pascal confrontato con il contenuto della cella di memoria fisicamente successiva all'ultimo elemento del vettore, e se per caso esso è uguale al valore cercato la risposta è *true*, anche se x è stato trovato "fuori del vettore"!

Abbiamo cosí il primo esempio di programma errato che tuttavia (se non è attivato il controllo del range) nella stragrande maggioranza dei casi funzionerà perfettamente! Un errore di questo genere è particolarmente subdolo, perchè (senza il controllo del range) molto probabilmente non verrebbe scoperto durante la messa a punto (debugging) del programma, e potrebbe quindi poi rivelarsi durante l'uso reale del programma, ad un istante imprevedibile, magari con conseguenze catastrofiche o almeno dannose.

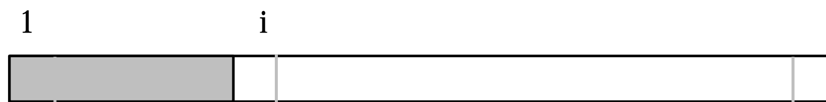
Solo una dimostrazione matematica, del genere di quelle che abbiamo cominciato ad abbozzare nei capitoli precedenti, può garantire la correttezza di un programma; come asserisce il famoso detto di Dijkstra, "il debugging può solo mostrare che un programma è errato, ma non può mai mostrare che è corretto"! (Ma naturalmente la dimostrazione della correttezza di un programma può essere errata ...).

5.5.3 Dimostrazione di correttezza della soluzione precedente (traccia).

Come al solito, i ragionamenti precedenti possono essere espressi in modo leggermente piú rigoroso sotto forma di una dimostrazione di correttezza.

CONDIZIONE INIZIALE: v è un vettore indiciato da 1 ad n, n>0.

CONDIZIONE FINALE: il risultato restituito dalla funzione è il valore booleano della proposizione "x compare nel vettore v".



INVARIANTE (INV): $(1 \le i \le n+1) \wedge (x \text{ non compare in } v[1 \,..\, i-1])$

CONDIZIONE DI USCITA: $(i > n) \vee (v[i] = x)$ *TEST DEL WHILE:* $(i \le n) \wedge (v[i] \ne x)$

Proposizione 1. INV è un invariante del ciclo.

Dimostrazione.

Base: Dopo l'inizializzazione si ha $i=1$; sostituendo nell'invariante si ottiene l'asserzione:

$$(1 \le 1 \le n+1) \wedge (x \text{ non compare in } v[1 \,..\, 0])$$

cioè

$$(1 \le 1 \le n+1) \wedge (x \text{ non compare nel sottovettore vuoto})$$

che è banalmente valida.

Passo:

Ipotesi: $INV \wedge TEST$, cioè
$(1 \le i \le n+1) \wedge (x \text{ non compare in } v[1 \,..\, i-1]) \wedge (i \le n) \wedge (v[i] \ne x)$
cioè
$(1 \le i \le n) \wedge (x \text{ non compare in } v[1 \,..\, i-1]) \wedge (v[i] \ne x)$

Tesi: Dopo aver eseguito il corpo del ciclo, vale di nuovo INV.

Dimostrazione:
Dall' ipotesi si ottiene immediatamente, "mettendo insieme" la seconda e l'ultima componente della congiunzione, che x non compare in $v[1 \,..\, i]$; dopo l'esecuzione del corpo, che consiste nell'incrementare i di 1, si avrà quindi di nuovo che x non compare in $v[1 \,..\, i-1]$. Quanto alla seconda parte dell'invariante, basta osservare che essendo per ipotesi $1 \le i \le n$, incrementando i si otterrà appunto $1 \le i \le n+1$.

Fine della dimostrazione della Proposizione 1.

Bisogna ora dimostrare che l'istruzione *ricerca:= i<=n* fornisce la risposta corretta; cioè che, all'uscita dal ciclo, la condizione $i \le n$ è logicamente equivalente alla proposizione "x compare in v", cioè che è $i \le n$ se e solo se il valore cercato x compare in v; ossia che all'uscita dal ciclo:

1) se è $i \le n$, allora x compare in v (e i è l'indice della prima occorrenza di x in v);
2) altrimenti (cioè $i > n$) x non compare in v.

In modo leggermente piú formale (come nella sezione 4.8):

$$(INV \wedge CU) \Rightarrow (i \leq n \Leftrightarrow x \text{ compare in } v)$$

ossia:

$$(INV \wedge CU) \Rightarrow ((i \leq n \Rightarrow x \text{ compare in } v) \wedge (i > n \Rightarrow x \text{ non compare in } v))$$

ossia anche:

1) $(INV \wedge CU \wedge (i \leq n)) \Rightarrow$ *(x compare in v, e l'indice della sua prima occorrenza è i)*
2) $(INV \wedge CU \wedge (i > n)) \Rightarrow$ *(x non compare in v)*

La dimostrazione è semplice:

2) poichè per *INV* è $i \leq n+1$, se è $i > n$ deve essere $i = n+1$ (in parole povere: se si è usciti per $i > n$, sappiamo che in realtà si è usciti con $i = n+1$, poichè i parte da 1 e viene incrementato di 1 ad ogni passo); sostituendo i con $n+1$ nella seconda componente dell'invariante si ottiene:

x non compare in v[1 .. (n+1)-1]
cioè
x non compare in v[1 .. n]
cioè
x non compare in v

1) poichè $i > n$ è una delle due condizioni di uscita in OR, se all'uscita dal ciclo *(CU)* essa non vale, deve valere l'altra; si ha quindi contemporaneamente $1 \leq i \leq n$ ed $a[i] = x$: le disuguaglianze su i assicurano che x è stato trovato proprio <u>in</u> v, e non eventualmente "fuori"; la prima parte dell'invariante, cioè l'asserzione *x non compare in v[1 .. i-1]* assicura inoltre che x non compare "prima", cioè che i è l'indice della prima occorrenza.

In questo e negli altri esempi di programmi con scansione lineare di vettori tralasciamo l'ovvia dimostrazione di terminazione.

5.5.4 Versione che restituisce l'indice dell'elemento trovato.

È facile modificare la soluzione precedente in modo che restituisca, invece di un booleano, l'indice dell'elemento trovato oppure il valore $n+1$ se l'elemento non è stato trovato.

```
function ricerca(var v: vettore; x: tipoelem): integer;
var i: integer;
begin
  i:= 1;
  while (i <= n) and (v[i] <> x) do i:= i+1;
  ricerca:= i;
end;
```

Se invece in caso di ricerca senza successo si vuol restituire ad esempio il valore 0 si scriverà:

```
function ricerca(var v: vettore; x: tipoelem): boolean;
var i: integer;
begin
  i:= 1;
  while (i <= n) and (v[i] <> x) do i:= i+1;
  if i<=n then ricerca:= i else ricerca:= 0
```

```
end;
```

5.5.5 Soluzioni Pascal Standard.

Per scrivere le versioni in Pascal Standard delle precedenti procedure bisogna togliere il test `(v[i] <> x)` dal test del *while* e portarlo nel corpo del ciclo, in modo che non possa venire eseguito per $i > n$; il risultato di tale test deve quindi essere memorizzato in una variabile booleana in modo da poter controllare la successiva iterazione del ciclo. Inoltre <u>non bisogna dimenticare di inizializzare</u> la variabile booleana stessa!

```
function ricerca(var v: vettore; x: tipoelem): boolean;
var i: integer;
    trovato: boolean;
begin
  i:= 1;
  trovato:= false;
  while (i <= n) and not trovato do begin
    if v[i]=x then trovato:= true else i:= i+1
  end;
  ricerca:= trovato
end;
```

Vi è anche un'altra soluzione Pascal Standard, piú elegante ma un po' piú difficile da trovare o da capire, che non fa uso di variabili booleane. In essa si lasciano nel test del *while* entrambe le condizioni della versione Turbo, ma - per impedire l'esecuzione del test `v[i]<>x` con $i>n$ - si esce dal ciclo "la volta prima" sostituendo alla condizione `i<=n` la condizione piú forte `i<n`. Il ciclo diventa quindi

```
i:= 1;
while (i < n) and (v[i] <> x) do i:= i+1;
```

Osserviamo che all'uscita dal ciclo se è $i<n$ si ha $v[i]=x$, altrimenti è $i=n$. In entrambi i casi il risultato è dato dal valore dell'espressione booleana `v[i]=x`.

```
function ricerca(var v: vettore; x: tipoelem): boolean;
var i: integer;
begin
  i:= 1;
  while (i < n) and (v[i] <> x) do i:= i+1;
  ricerca:= v[i]=x
end;
```

Si noti che se si è usciti dal ciclo perchè $i=n$, bisogna ancora verificare se $v[n]=x$ oppure no, e ciò viene fatto con il test finale $v[i]=x$; se invece si è usciti con $i<n$ si è usciti proprio perchè $v[i]=x$; tuttavia solo la riesecuzione del test permette di stabilire che si è usciti appunto per tale ragione.

Esercizio 1.
Scrivere la dimostrazione di correttezza per la soluzione precedente.

Esercizio 2.
Modificare le soluzioni Pascal Standard precedenti in modo che invece di un booleano restituiscano l'indice dell'elemento.

Per la ricerca in un vettore non vuoto, come quella che abbiamo finora ipotizzato (non ha senso definire un vettore di lunghezza 0, e in Pascal è conseguentemente vietato), una facile ed elegante soluzione in Pascal standard si ottiene con l'istruzione *repeat*:

```
function ricerca(var v: vettore; x: tipoelem): boolean;
var i: integer;
begin
  i:= 0;
  repeat i:= i+1 until (v[i] = x) or (i = n);
  ricerca:= v[i]=x
end;
```

INV: *(1* £ *i* £ *n)* Ù *(x non compare in v[1 .. i-1])*

INV Ù *(v[i]=x)* fi *(1* £ *i* £ *n)* Ù *(v[i]=x)* *cioè x compare in v.*
TEST Ù *(v[i]* „ *x)* fi *(i = n)*
(i = n) Ù *INV* fi *x non compare in v[1 .. n-1]*
(v[n] „ *x)* Ù *(x non compare in v[1..n-1]* fi *x non compare in v*

La soluzione con il *repeat* diventa tuttavia meno elegante se vogliamo disporre di una funzione piú versatile che possa effettuare la ricerca su un sottovettore, e prenda quindi come parametri anche i due indici di inizio e fine (oppure indice d'inizio e lunghezza) del sottovettore, che può essere vuoto. In tal caso bisogna infatti racchiudere il *repeat* dentro un *if-then-else*:

```
function ricerca(var v:vettore; x:tipoelem; inf,sup:integer):boolean;
var i: integer;
begin
  if inf > sup then ricerca:= false
  else begin
    i:= inf
    repeat i:= i+1 until (v[i] = x) or (i = sup);
    ricerca:= v[i]=x
  end
end;
```

La realizzazione Turbo con il *while* è invece direttamente generalizzabile, ad esempio:

```
function ricerca(var v:vettore; x:tipoelem; inf,sup:integer):boolean;
var i: integer;
begin
  i:= inf;
  while (i <= sup) and (v[i] <> x) do i:= i+1;
  ricerca:= i<=n
end;
```

5.5.6 Un esempio di programma principale.

Per finire, scriviamo un programma principale TurboPascal che visualizzi tutti gli elementi di un vettore aventi un dato valore in uno specifico campo (immaginiamo cioè che gli elementi del vettore siano, come nell'esempio del vettore di studenti, dei record costituiti da vari campi, di cui uno - che indichiamo genericamente con l'etichetta *chiave* - sia il campo che ci interessa; ad esempio, tutti i libri di un dato autore presenti in biblioteca).

Definiamo un'ulteriore versione della funzione *ricerca*, consistente un sottoprogramma "ibrido" (funzione con effetto collaterale) che ha fra i parametri d'ingresso gli indici inferiore *inf* e superiore *sup* del sottovettore di ricerca, restituisce *true* o *false* a seconda che la ricerca abbia o no successo, e restituisce inoltre in *inf* (che è quindi un parametro di input/output) l'indice dell'elemento trovato. Assumiamo che il vettore-argomento sia "parzialmente riempito" (vedi sezione 5.1.2); `lun` è la lunghezza della parte occupata di `vett`. Usiamo inoltre, per esercizio, gli array aperti.

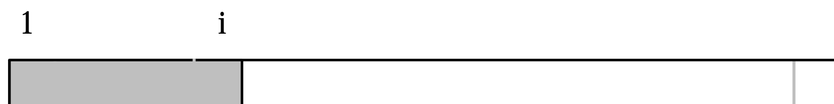
```
const nmax = ...
type tipochiave = ...
     tipoelem = record ...  chiave: tipochiave; ... end;
          ..
var vett: array[0..nmax] of tipoelem;
    iniz, lun: integer;
    val: tipochiave;

  procedure scrivi_elemento(el: tipoelem);
    {visualizza opportunamente tutti i campi di un elemento}
  end;

  procedure leggivettore(var v: array of tipoelem; var n: integer);
    {effettua l'input di una sequenza di elementi, caricandoli nel
     vettore v e restituendo la lunghezza della sequenza}
  end;


  function ricerca(var v:array of tipoelem; x:tipochiave;
                   var inf: integer; sup: integer): boolean;
  begin
    while (inf <= sup) and (v[inf].chiave <> x) do inf:= inf+1;
    ricerca:= inf <= sup
  end;

begin
  leggivettore(vett,lun);
  write('valore della chiave di ricerca: ');
  readln(val);
  iniz:= 0;
  while ricerca(vett,val,iniz,lun-1) do begin
    scrivi_elemento(vett[iniz]);
    iniz:= iniz+1
  end
end.
```

Si noti che se nel ciclo *while* principale non si incrementa `iniz`, il programma non termina.

5.6 Funzione che stabilisce se un vettore è ordinato.

5.6.1 Costruzione della funzione.

Per stabilire se un vettore è ordinato occorre percorrere il vettore verificando che ogni elemento sia maggiore o uguale del precedente, naturalmente fermandosi alla fine del vettore o appena si trova un elemento "fuori posto".



Alla generica iterazione del ciclo avremo allora verificato che una parte iniziale del vettore è ordinata, cioè:

INV: (1 £ i £ n) Ù *il sottovettore v[1..i] è ordinato.*

Come abbiamo detto, vogliamo uscire dal ciclo o perchè abbiamo trovato un elemento "fuori posto", oppure perchè abbiamo stabilito che il vettore è interamente ordinato. La condizione voluta all'uscita dall'istruzione *while* è quindi:
POSTWH: *((i<n)* Ù *(v[1..i] è ordinato)* Ù *(v[i]>v[i+1]))* Ú *(v[1..n] è ordinato)*

La condizione di uscita dal ciclo che congiunta con l'invariante assicura la POSTWH sarà anch'essa come la POSTWH la disgiunzione di due condizioni:

CU: ((i<n) Ù *(v[i]>v[i+1]))* Ú *(i=n)*

Il ciclo è quindi (in TurboPascal, perchè ...):

```
while (i<>n) and (v[i]<=v[i+1]) do i:= i+1;
```

Se all'uscita dal ciclo è *i=n*, il vettore è ordinato; altrimenti si è usciti perchè *v[i]>v[i+1]* e quindi il vettore non è ordinato. Il risultato è pertanto il valore dell'espressione booleana `i=n`. Il sottoprogramma completo è:

```
function ordinato(var v: vettore): boolean;
var i: integer;
begin
  i:= 1;
  while (i<n) and (v[i]<=v[i+1]) do i:= i+1;
  ordinato:= i=n
end;
```

dove il test `i<>n` è stato sostituito dal test `i<n`, il che - come si vede facilmente - non altera il funzionamento del sottoprogramma.

5.6.2 Dimostrazione di correttezza (traccia)

Riordiniamo i ragionamenti precedenti nella traccia della (banale!) dimostrazione di correttezza per la versione finale del sottoprogramma.

Proposizione 1: L'invariante del ciclo è *INV: (1 £ i £ n)* Ù *(v[1..i] è ordinato).*

Dimostrazione.

Base
Dopo l'inizializzazione si ha *i=1*; sostituendo nell'invariante si ottiene la congiunzione della proposizione *1£n*, che è ovviamente vera, e della proposizione *(v[1..1] è ordinato)*

che è anch'essa banalmente vera perchè una sequenza di un solo elemento è banalmente ordinata.

Passo

Ipotesi: *INV* Ù *TEST*, cioè *INV* Ù *(i<n)* Ù *(v[i]<=v[i+1])*

Tesi: Dopo aver eseguito il corpo del ciclo, vale di nuovo INV

Dimostrazione:

Essendo per ipotesi *i<n,* dopo l'esecuzione dell'istruzione `i:= i+1` si ha *i* £ *n*; inoltre, essendo prima dell'incremento *v[i]* £ *v[i+1]*, si ha che dopo l'incremento il nuovo *v[1..i]* è ancora ordinato.

Fine della dimostrazione della Proposizione 1.

Bisogna ora dimostrare che l'istruzione `ordinato:= i=n` fornisce la risposta corretta; cioè che, <u>all'uscita dal ciclo</u>, la condizione *i=n* è <u>logicamente equivalente</u> alla risposta finale desiderata, cioè all'asserzione *"il vettore v è ordinato"*. In formula:

(INV Ù Ø*TEST)* fi *(i = n* « *il vettore v è ordinato)*

Infatti:

1) se è *i* = *n*, allora sostituendo nell'invariante si ha proprio che il vettore *v* è interamente ordinato;
2) altrimenti, cioè se *i* <>*n*, poichè per l'invariante è *i*£*n*, si ha *i*<*n*, allora per la condizione di uscita dal *while* (che è la negata del test) è *v[i]* > *v[i+1]*, dove *v[i]* e *v[i+1]* sono entrambi elementi legittimi del vettore *v* (perchè è *i* < *n*, e inoltre per l'INV è i ≥ 1); quindi il vettore non è ordinato.

5.7 Ricerca del primo elemento uguale alla somma dei *k* precedenti.

Problema: Definire una funzione *f* la quale, presi come parametri un vettore di interi *v* indiciato da *1* ad *n*, e un intero positivo *k<n*, restituisca l'indice *i* del primo (da sinistra) elemento di *v* che è uguale alla somma dei *k* precedenti, cioè tale che

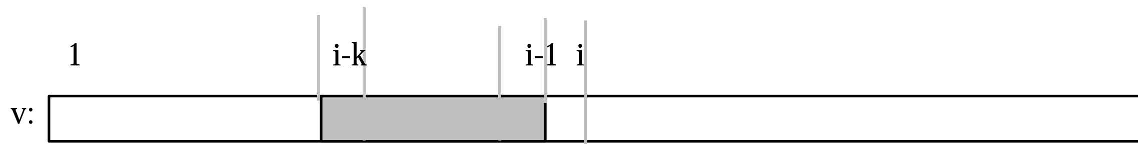
v[i] = v[i-1] + v[i-2] + ... + v[i-k].

Se un tale elemento non esiste, restituisca un valore maggiore di *n* (ad es. il valore *n+1*).

Per risolvere il problema bisognerà percorrere il vettore *v* a partire dal *k+1*-esimo elemento, e per ogni elemento calcolare la somma dei *k* elementi precedenti; la soluzione ingenua è quindi costituita da due cicli annidati, un ciclo *while* o *repeat* esterno che viene ripetuto nel caso peggiore *n-k volte*, e per ogni iterazione del ciclo esterno un ciclo *for* interno di *k* iterazioni per calcolare la somma dei *k* precedenti; il tempo di calcolo dell'algoritmo è quindi proporzionale a *(n-k)k*, cioè è quadratico in *k*, e quindi quadratico in *n* se si assume che, al crescere di *n*, *k* cresca in modo proporzionale ad *n*.

<u>Esiste però una soluzione migliore</u>; si può infatti osservare che per calcolare di volta in volta la somma dei k elementi precedenti non è necessario fare *k* addizioni, ma basta ogni volta togliere dalla somma precedente il primo dei *k* elementi e aggiungere il *k+1*-esimo! In questo modo all'interno del ciclo *while* non si effettuano piú cicli ma soltanto operazioni di durata indipendente da *k* e da *n*, e l'algoritmo diventa lineare in *n*.

Come al solito, per arrivare ad una realizzazione corretta di tale soluzione, rappresentiamo con un disegno la situazione al generico passo di iterazione:



Nel segmento iniziale di vettore fino all'elemento *v[i-1]* compreso non è stato trovato alcun elemento con la proprietà desiderata; *v[i]* è l'elemento da controllare (se è $i \leq n$); nella variabile *somma* vi è già la somma dei *k* elementi precedenti *v[i]*. Abbiamo cosí l'invariante del ciclo.

La condizione di uscita sarà la disgiunzione di due condizioni:

- *v[i] = somma*, nel qual caso l'elemento cercato è proprio l'i-esimo, perchè l'invariante ci assicura che non ne sono stati trovati altri prima con la proprietà voluta;
- $i > n$, nel qual caso l'invariante ci garantisce che l'elemento cercato non esiste nel vettore.

In entrambi i casi il valore che deve essere restituito è *i*.

Il test del while si ottiene negando la condizione di uscita; il corpo del *while*, dovendo mantenere l'invariante, dovrà aggiornare somma e incrementare *i*. Quindi:

```
while (i <= n) and (v[i] <> somma) do begin
  somma:= somma - v[i-k] +v[i];
  i:= i+1
end;
f:= i;
```

In formule:

INV: $1 \leq i \leq (n+1) \wedge$
$\forall j<i .(v[j] \neq v[j-1] + v[j-2] + ... + v[j-k]) \wedge somma = v[i-k] + ... + v[i-2] + v[i-1]$

CU (Condizione di Uscita dal ciclo): $(i > n) \vee (v[i] = somma)$

La congiunzione di invariante e condizione di uscita ha come conseguenza la situazione finale voluta, cioè che il valore di *i* è il valore corretto che deve essere restituito dalla funzione. Infatti:

1) se $i>n$ si ha:
$(INV \wedge (i > n)) \Rightarrow i = n+1$;
allora, sostituendo $n+1$ a i nella seconda componente dell'invariante si ha:
$(INV \wedge (i = n+1)) \Rightarrow \forall j < n+1.(v[j] \neq v[j-1] + v[j-2] + ... + v[j-k])$
cioè
$(INV \wedge (i = n+1)) \Rightarrow \forall j \leq n . (v[j] \neq v[j-1] + v[j-2] + ... + v[j-k])$
cioè l'elemento cercato non esiste.

2) altrimenti, cioè se $i \leq n$, allora $((i \leq n) \wedge CU) \Rightarrow (v[i] = somma)$;
ma, per *INV*, $somma = v[i-k] + ... + v[i-2] + v[i-1]$;

quindi *v[i]* è un elemento legale del vettore (cioè $1 \leq i \leq n$), gode della proprietà voluta, e inoltre - sempre per *INV* - non vi è nessun elemento prima di esso che goda della proprietà.

Affinchè l'invariante valga prima della prima iterazione del *while* - e quindi sia assicurata la base dell'induzione - occorre come al solito un'inizializzazione; in particolare, si dovrà avere:

somma := v[1] + v[2] + ... + v[k];
i:= k+1;

Naturalmente la prima delle due non è un'istruzione di un linguaggio programmativo, ma può essere realizzata per mezzo di un semplice ciclo *for*.

In conclusione, la definizione della funzione è la seguente:

```
function f(var v: vettore; k: integer): integer;
var i, somma: integer;
begin
  somma:= 0;
  for i:= 1 to k do somma:= somma + v[i];
  i:= k+1;                              (* fine inizializzazione *)
  while (i <= n) and (v[i] <> somma) do begin
    somma:= somma - v[i-k] + v[i];
    i:= i+1;
  end;
  f:= i
end;
```

<vedi lucidi pag. 110-111>

5.8 Cancellazione da un vettore (con compattazione) di tutti gli elementi uguali ad un dato valore.

Problema: Definire una procedura la quale, dato un vettore *v* "parzialmente riempito", indiciato da *1* ad *nmax*, di lunghezza effettiva *n*, e dato un valore *x* (dello stesso tipo degli elementi del vettore), modifichi il vettore *v* cancellando in esso tutti gli elementi uguali a *x* (o, come si dice anche, tutte le *occorrenze* di *x*) e compattando il vettore (cioè "eliminando i buchi").

Cominciamo con lo stabilire quali devono essere i parametri della procedura: poichè si è specificato che il vettore *v* è "parzialmente riempito", la procedura dovrà avere come parametro oltre al vettore *v* (e al valore *x*) anche la dimensione effettiva *n* del vettore; poichè inoltre la procedura può modificare tale dimensione effettiva (compattando e quindi accorciando il vettore), il parametro *n* è non solo di input ma anche di output, e deve quindi essere per riferimento. Avremo quindi:

```
procedure cancellatutti(x:tipoelem; var v:vettore; var n:integer);
```

Affrontiamo ora il problema vero e proprio.

La prima soluzione che viene in mente è quella implicitamente contenuta nella specifica del problema: si scorre il vettore *v* controllandone gli elementi, ed ogni volta

che si trova un elemento uguale ad *x* si spostano di una posizione tutti gli elementi seguenti:

```
i:= 1;
while i<= n do
  if v[i] <> x then i:= i+1
  else sposta di una posizione tutti gli elementi seguenti
```

Anche senza scrivere esplicitamente la procedura, è facile rendersi conto che nei casi peggiori - in cui ad esempio vi siano "molti" elementi uguali ad *x* - per la maggior parte delle iterazioni del ciclo esterno verrà effettuato un ciclo interno di compattamento; il numero delle iterazioni del ciclo interno sarà quindi all'incirca *n* per spostare tutti gli elementi successivi al primo, piú *n-1* per spostare tutti gli elementi successivi al secondo, ecc., cioè $n + (n-1) + (n-2) + ... = \Theta(n^2)$.

La complessità temporale asintotica del caso peggiore è quindi quadratica; la maggior parte degli elementi del vettore vengono spostati piú e piú volte prima di raggiungere la loro posizione definitiva.

In termini leggermente piú precisi: se consideriamo *n* tendente ad infinito e il numero *k* di occorrenze di *x* crescente in modo proporzionale ad *n*, otteniamo che il tempo di calcolo cresce in modo proporzionale a n^2. Il risultato non cambia, tranne che in casi particolari, se si percorre il vettore a partire dal fondo.

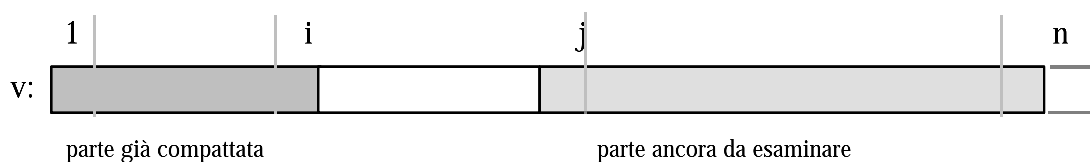
Una soluzione in tempo lineare è però facilmente realizzabile utilizzando un vettore ausiliario *a*: basta infatti scorrere il vettore *v* inserendo uno dopo l'altro nel vettore *a* gli elementi diversi da *x*; alla fine basterà ricopiare il vettore ausiliario nel vettore originale:

```
j:= 0;
for i:= 1 to n do
  if v[i] <> x do begin
    j:= j+1;
    a[j]:= v[i]
  end;
for i:= 1 to j do v[i]:= a[i];
n:= j;
```

Infatti nel caso peggiore - che ora è quello in cui *x* non compare nel vettore - si hanno due cicli di *n* iterazioni ciascuno; il tempo di calcolo è quindi proporzionale a *2n*. Lo spazio necessario per il calcolo in aggiunta a quello occupato dall'input è però ora passato da costante a lineare, poichè si utilizza un vettore ausiliario della stessa dimensione del vettore di ingresso.

Ci si chiede se non esista una soluzione in tempo lineare e spazio costante, cioè una soluzione che - come la seconda soluzione - percorra una sola volta il vettore senza cicli interni, ma non utilizzi un vettore ausiliario. La risposta positiva si ottiene osservando che, nella seconda soluzione, al generico passo di scansione del vettore *v* la parte occupata di *a*, che è il risultato del compattamento della parte già esaminata di *v*, è ovviamente non piú lunga di qust'ultima; la parte via via esaminata del vettore *v* può quindi essere compattata direttamente in *v* stesso.

Rappresentiamo, come al solito, la situazione al generico passo d'iterazione, ed esprimiamo in modo informale l'invariante significativo del ciclo.



Indichiamo con v_0 lo stato iniziale del vettore *v*. Allora abbiamo:

INV: v[1..i] è il sottovettore risultante dall'eliminazione di x nel sottovettore v_0*[1..j-1] e dal suo conseguente compattamento.*

Sostituendo nell'invariante *j* con *n+1* si ottiene la desiderata condizione di eliminazione di *x* da tutto il vettore:

CF: v[1..i] è il sottovettore risultante dall'eliminazione di x nel sottovettore v_0*[1..n] e dal suo conseguente compattamento.*

La condizione di uscita dal ciclo è perciò:

CU: j = n+1

Il ciclo è quindi della forma

```
while j<=n do ...
```

La dimensione finale della parte compattata è `i`, pertanto dopo l'uscita dal ciclo occorrerà eseguire l'istruzione:

```
n:= i
```

per restituire in `n` il valore corretto.

Scriviamo ora il corpo del ciclo. Bisogna prendere in esame il primo elemento del segmento di *v* ancora da scorrere, cioè `v[j]`: se non è uguale ad *x* lo si aggiunge al fondo della parte compattata, poi in ogni caso si incrementa `j`.

```
if v[j] <> x then aggiungi v[j] al fondo del segmento v[1..i];
j:= j+1;
```

Per aggiungere `v[j]` alla parte compattata si deve notare che nel disegno sopra riportato `i` è l'indice dell'ultimo elemento occupato del segmento in questione; occorre quindi prima incrementare `i` e poi copiare `v[j]` in `v[i]`.

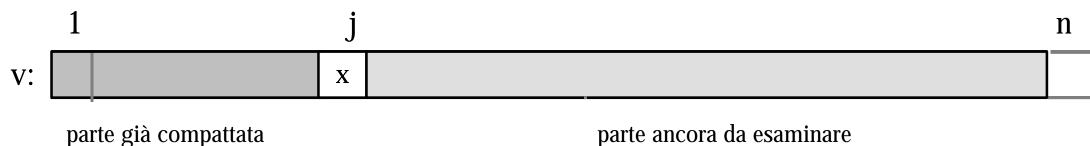
L'inizializzazione necessaria per rendere vero l'invariante prima di ogni azione sul vettore si ricava osservando che all'inizio la parte compattata sarà il sottovettore vuoto, cioè il segmento *v[1..0]*, mentre il segmento ancora da esaminare sarà l'intero vettore, cioè *v[1..n]*; tale affermazione coincide con l'invariante se *i=0* e *j=1*.

In conclusione, il corpo della procedura è:

```
i:= 0; j:= 1;
while j<=n do begin
  if v[j] <> x then begin
    i:= i+1;
    v[i]:= v[j]
  end;
  j:= j+1
end;
n:= i;
```

Il ciclo *while* può essere sostituito da un equivalente ciclo *for*.

Inoltre, si può osservare che la procedura, finchè non ha trovato per la prima volta un elemento uguale ad *x*, ricopia ogni elemento in se stesso. Si può allora introdurre un ciclo iniziale che ricerca tale prima occorrenza di *x*. Alla fine di esso la situazione sarà quella rappresentata nella figura sottostante.



In *v[j]* si ha la prima occorrenza di *x*, oppure, se *x* non compare in *v*, si ha *j = n+1*; in entrambi i casi la parte "compattata" è *v[i..j-1]* (che nel caso in cui *x* non compaia in *v* coincide con *v[1..n]*), la parte da esaminare è *v[j+1..n]* (che nel caso in cui *x* non compaia in *v* diventa *v[n+2..n]*, cioè un segmento vuoto, come dev'essere).

Confrontando con l'invariante (ad es. confrontando i due disegni), si ricavano le inizializzazioni per *i* e *j* prima del ciclo principale:

```
i:= j-1;
j:= j+1;
```

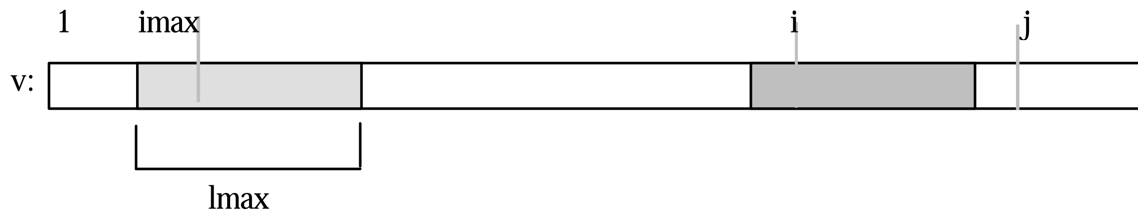
In conclusione, una versione finale della procedura che risolve in modo ottimo il problema è:

```
procedure cancellatutti(x:tipoelem; var v:vettore; var n:integer);
var i,j: integer;
begin
  j:= 1;
  while (j<=n) and (v[j]<>x) do j:= j+1;
  i:= j-1;
  for j:= j+1 to n do
    if a[j]<>x then begin
      i:= i+1;
      v[i]:= v[j]
    end;
  n:= i
end;
```

Si osservi che quando *x* non compare in *v*, o quando *x* compare in *v* soltanto come ultimo elemento, il ciclo *for* non viene eseguito.

5.9 Il piú lungo segmento ordinato.

Problema. Dato un vettore (indiciato da *1* ad *n*) trovare l'indice d'inizio e la lunghezza (oppure l'indice d'inizio e l'indice dell'ultimo elemento) del piú lungo segmento ordinato (cioè non decrescente) del vettore.



Una soluzione.

INV: $1 \le imax < imax+l \le i \le j \le n+1$
$\wedge$
v[imax .. imax+lmax-1] è il piú lungo segmento ordinato contenuto in v[1 .. i-1]
$\wedge$
v[i .. j-1] è ordinato $\wedge$ $\neg\exists k<i$ *. v[k .. j-1] è ordinato*

CU: $j > n$, *cioè* $j = n+1$

INV $\wedge$ *CU*
$\Rightarrow$
se j-i > lmax, allora il segmento piú lungo inizia all'indice i e ha lunghezza j-i,
altrimenti inizia all'indice i e ha lunghezza lmax

INIZIALIZZAZIONE:

i = 1
(perchè il segmento *v[1 .. i-1]* di cui si ha il piú lungo sottosegmento ordinato è vuoto)

lmax = 0, imax = indefinito
(perchè allora anche il piú lungo segmento ordinato nel segmento vuoto è vuoto)

j = 2
(perchè il segmento *v[1..1]* è banalmente ordinato, e $\neg\exists k<1$*. v[k .. 1] è ordinato*).

```
procedure maxord(var v: vettore; var imax, lmax: integer);
var i,j: integer;
begin
  lmax:= 0; i:= 1; j:= 2;
  while j <= n do begin
    if v[j-1] > v[j] then begin   {segmento ordinato termina in j-1}
      if (j-i) > lmax then begin  {è il nuovo massimo}
        lmax:= j-1; imax:= i
      end;
      i:= j    {un nuovo segmento comincia da j, v[j..j] è ordinato}
    end;                          {v[i..j] è ordinato}
    j:= j+1                       {v[i..j-1] è ordinato}
  end{while};
  if (j-i) > lmax then begin      {confronto con l'ultimo segmento}
    lmax:= j-i; imax:= i
  end
end;
```

Possiamo trasformare il ciclo *while* in un *for*, facendo attenzione al fatto che all'uscita del *for* il contatore *j* ha un valore indefinito, e quindi nell'ultimo confronto bisogna sostituire *j* con il valore esplicito $n+1$:

```
procedure maxord(var v: vettore; var imax, lmax: integer);
var i,j: integer;
begin
  lmax:= 0; i:= 1;
  for j:= 2 to n do
    if v[j-1] > v[j] then begin   {segmento ordinato termina in j-1}
      if (j-i) > lmax then begin  {è il nuovo massimo}
        lmax:= j-1; imax:= i
      end;
      i:= j                       {un nuovo segmento comincia da j}
    end;
  if (n+1-i) > lmax then begin    {confronto con l'ultimo segmento}
    lmax:= n+1-i; imax:= i
  end
end;
```

Quando la lunghezza del segmento *v[i .. n]* (i cui segmenti ordinati non sono ancora stati considerati) è minore o uguale alla lunghezza massima finora trovata, è inutile cercare ancora; si ha quindi l'ulteriore condizione di uscita *n-i+1* £ *lmax*, cioè la condizione *i* ‡ *n-lmax+1*, cioè $i > n\text{-}lmax$. Quando si esce per tale ragione, l'ulteriore confronto dopo l'uscita dal ciclo è inutile, allora in TurboPascal si può uscire direttamente dalla procedura con una *exit*.

```
procedure maxord(var v: vettore; var imax, lmax: integer);
var i,j: integer;
begin
  lmax:= 0; i:= 1; j:= 2;
  while j <= n do begin
    if v[j-1] > v[j] then begin   {segmento ordinato termina in j-1}
      if (j-i) > lmax then begin  {è il nuovo massimo}
        lmax:= j-1; imax:= i
      end;
      i:= j;
      if i > n-lmax then exit;    {si esce dalla procedura}
    end;
    j:= j+1;
  end{while};
  if (j-i) > lmax then begin      {confronto con l'ultimo segmento}
    lmax:= j-i; imax:= i
  end
end;
```

Si osservi che sarebbe errato effettuare il nuovo controllo su `j` invece che su `i`, scrivendo semplicemente `while j<=n-lmax`, perchè il segmento ordinato che si sta scandendo ha avuto in generale un inizio precedente a `j` (cioè appunto `i`).

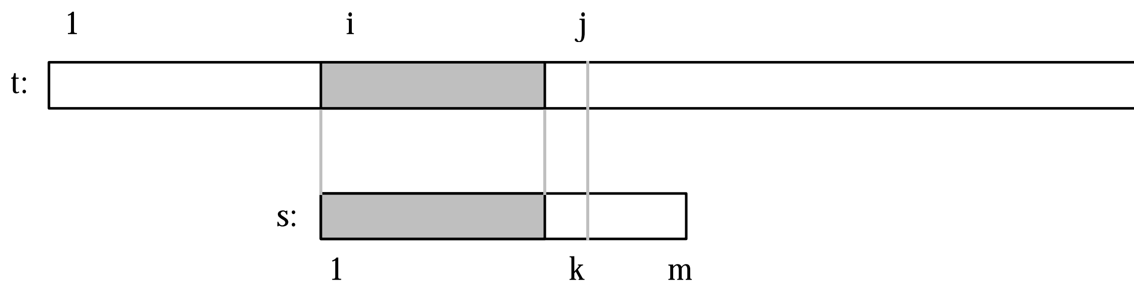
Della soluzione di questo problema sono possibili molte varianti, anche in Pascal standard, il lettore è invitato a scriverne qualcuna dimostrandone la correttezza.

5.10 Ricerca di una stringa in un testo.

5.10.1 Costruzione della soluzione con un solo *while*.

Problema: Definire una funzione *trova* la quale cerchi una stringa *s* in un testo *t* di lunghezza *n*; piú precisamente, scrivere una funzione la quale, presi come argomenti una stringa *s*, un vettore *t* di caratteri indiciato da *1* ad un numero sufficientemente grande, e la lunghezza effettiva *n* del testo *t*, se la stringa *s* è contenuta nel testo *t* restituisce l'indice d'inizio in *t* della prima *occorrenza* della stringa *s*; se la stringa *s* non compare in *t*, restituisce un indice fuori dall'intervallo ammesso, ad es. 0, oppure *n+1*.

Sia *m* la lunghezza della stringa *s* (ed *n* la lunghezza del vettore-testo *t*); la situazione al generico passo di iterazione sarà la seguente:



Si è già stabilito che nessuna delle posizioni di *t* fra *1* e *i-1* (compresi) è l'inizio di una occorrenza della stringa *s* nel testo *t* (*i-1* è quindi compreso nel *range* di indici di *v*); si sta ora cercando di stabilire se la stringa *s* compare in *t* con inizio in *t[i]* (se *i* è ancora nel *range*) e si è già verificato che i caratteri di *t* dalla posizione *i* alla posizione *j-1* sono uguali rispettivamente ai caratteri di *s* dalla posizione *1* alla posizione *k-1*, dove naturalmente *k-1* e *j-1* sono posizioni comprese nei *range* di indici (delle parti "occupate") dei rispettivi vettori, cioè *k-1* £ *m* e j-1 ≤ n, ossia *k* £ *m+1* e *j* £ *n+1*.

Un invariante significativo è quindi:

(i-1 £ *n)* Ù *(k-1* £ *m)* Ù *(j-1* £ *n)*
Ù
(s non compare in t con inizio in t[1..i-1]) Ù *(s[1..k-1] = t[i..j-1])*

Naturalmente, si esce dal ciclo quando si trova la stringa *s*, oppure quando termina il testo, cioè alla fine del vettore *t*; in realtà, la stringa *s* può comparire in *t* non piú a destra della posizione *n-m+1* (perchè dopo non ci sta piú!).
Ponendo *sup = n-m+1*, la condizione di uscita sarà quindi

CU': (k-1 = m) Ú *(i-1 = sup)* cioè *(k = m+1)* Ú *(i = sup+1)*

Infatti:

1) se è *k-1 = m*, sostituendo nell'invariante otteniamo (tralasciando la terza formula della congiunzione):
(s non compare con inizio in t[1..i-1]) Ù *(s[1..m] = t[i..j-1])* Ù *(j -1*£ *n)*

il che vuol proprio dire che *s* compare per la prima volta in *t* nella posizione *i* (la prima formula della congiunzione ci dice che *s* non compare precedentemente, la seconda

formula esprime il fatto che la stringa *s* è stata trovata, e infine la terza garantisce che è stata correttamente trovata tutta entro il vettore-testo).

2) se è $i-1 = sup$, sostituendo nell'invariante otteniamo:

s non compare in t con inizio in t[1..sup]

quindi *s* non compare in *t* (perchè sappiamo che non può comparire a destra di *sup*).

Negando la condizione di uscita si ottiene il test del while:

```
while (k<>m+1) and (i<>sup+1) do ...
```

In realtà, poichè nell'invariante abbiamo $k \leq m+1$, per ottenere all'uscita dal ciclo $k=m+1$ basta porre nella CU $k>m$; analogamente, rafforzando nell'invariante la condizione su *i*, cioè ponendo $i \leq sup+1$, si può indebolire la corrispondente parte della CU, mettendovi semplicemente $i > sup$.

In conclusione, le versioni definitive di invariante e condizione di uscita saranno:

INV:
$(k-1 \leq m) \wedge (j-1 \leq n) \wedge (i-1 \leq sup) \wedge (sup = n-m+1)$
$\wedge$
(s non compare in t con inizio in t[1..i-1]) $\wedge$ $(s[1..k-1] = t[i..j-1])$

CU: $(k > m) \vee (i > sup)$

Il test del while sarà quindi:

```
while (k <= m) and (i <= sup) do ...
```

Per restituire il risultato dopo l'uscita dal ciclo basta osservare che se è $k>m$ allora, come abbiamo visto, la stringa è stata trovata in *t* con inizio in *t[i]*; se invece non è $k>m$, allora deve essere $i>sup$, il che abbiamo visto implica che la stringa non è stata trovata (in tal caso restituiamo 0). Quindi l'ultima istruzione della funzione sarà semplicemente:

```
if k > m then trova:= i else trova:= 0
```

Scriviamo ora il corpo del ciclo: esso deve mantenere l'invariante "avvicinandosi" però alla condizione di uscita, cioè facendo avanzare `i` oppure `k`:

```
if s[k] = t[j] then avanza k e j
else avanza i e riparti con j da i e con k da 1
```

cioè

```
if s[k] = t[j] then begin k:= k+1; j:= j+1 end
else begin
  i:= i+1;
  j:= i; k:= 1
end;
```

Ci rimane infine da scrivere l'inizializzazione. Osserviamo che l'invariante è banalmente vero per $i=1$, $j=1$, $k=1$: infatti sostituendo si ottiene l'asserzione:

(s non compare in t con inizio in t[1..0]) Ù *(s[1..0] = t[1..0])*
Ù *(0* £ *m)* Ù *(0* £ *n)* Ù *(0* £ *sup)*

che è banalmente valida.
Le istruzioni di inizializzazione saranno allora: `i:= 1; j:= 1; k:= 1.`

La funzione completa è allora (preceduta dalla dichiarazione del tipo del vettore testo):

```
type textbuftype = array[1..64000] of char;

function trova(var s:string; var t:textbuftype; n:integer):integer;
var i,j,k,m,sup: integer;
begin
  m:= length(s); sup:= n-m+1;
  i:=1; j:=1; k:=1;
  while (k<=m) and (i<=sup) do
    if s[k]=t[j] then begin k:= k+1; j:= j+1 end
    else begin
      i:= i+1;
      j:=i; k:=1
    end;
  if k>m then trova:= i else trova:= 0
end;
```

La variabile j (con l'istruzione che la incrementa) può essere eliminata osservando che il suo valore è sempre uguale a $i+k-1$:

```
function trova(var s:string; var t:textbuftype; n:integer):integer;
var i,k,m,sup: integer;
begin
  m:= length(s); sup:= n-m+1;
  i:=1; k:=1;
  while (k<=m) and (i<=sup) do
    if s[k]=t[i+k-1] then k:= k+1
    else begin
      i:= i+1;
      k:=1
    end;
  if k>m then trova:= i else trova:= 0
end;
```

5.10.2 Dimostrazione di correttezza.

La (traccia della) dimostrazione di correttezza non è altro, come al solito, che l'esposizione in forma piú ordinata e rigorosa dei ragionamenti che ci hanno condotto a scrivere il sottoprogramma corretto.

Abbreviazioni usate:
Diremo che "*s compare in t con inizio in t[i]*" intendendo naturalmente che l'intera stringa s è contenuta in t con inizio in $t[i]$, cioè:
$s[1] = t[i],\ s[2] = t[i+1], \dots ,\ s[m] = t[i+m-1]$
dove $i, i+1, \dots , i+m-1$ sono compresi nell'intervallo $1..n$.

Analogamente diremo che "*s compare in t con inizio in t[i..i']*" intendendo che *s* compare in *t* con inizio in un certo *t[h]* con *h* compreso fra *i* e *i'* (inclusi).
Poniamo inoltre $sup = n-m+1$.

Proposizione 1. L'invariante del ciclo è:
INV:
$(s \text{ non compare in } t \text{ con inizio in } t[1..i-1]) \land (s[1..k-1] = t[i..j-1])$
$\land (1 \le k \le m+1) \land (1 \le j \le n+1) \land (1 \le i \le sup+1)$

Dimostrazione:

Base: L'invariante vale subito dopo l'inizializzazione.
Dimostrazione:
Per effetto delle istruzioni di inizializzazione si ha: $i=1, j=1, k=1$.
Sostituendo nell'invariante si ha allora:
$(s \text{ non compare in } t \text{ con inizio in } t[1..0]) \land (s[1..0] = t[1..0])$
$\land (0 \le m) \land (0 \le n) \land (0 \le sup)$

Tale asserzione è banalmente vera (perchè i tre sottovettori che compaiono in essa sono vuoti, ecc.).

Passo induttivo:

Ipotesi: Valgono *INV* e la condizione del *while*, quindi:

$(s \text{ non compare in } t \text{ con inizio in } t[1..i-1]) \land (s[1..k-1] = t[i..j-1])$
$\land (k \le m) \land (j \le n) \land (i \le sup)$

Tesi: Dopo aver eseguito il corpo del ciclo, vale di nuovo *INV*, cioè:

$(s \text{ non compare in } t \text{ con inizio in } t[1..i-1]) \land (s[1..k-1] = t[i..j-1])$
$\land (k \le m+1) \land (j \le n+1) \land (i \le sup+1)$

Dimostrazione del passo:

1) Se $s[k] = t[j]$ allora con l'ipotesi $s[1..k-1] = t[i..j-1]$ si ottiene $s[1..k] = t[i..j]$; ma poichè in tal caso poi si incrementano *k* e *j,* si ha di nuovo $s[1..k-1] = t[i..j-1]$; inoltre poichè era $k \le m$ e $j \le n$ si avrà ora $k \le m+1$ e $j \le n+1$. Le altre condizioni su *i* che compongono l'invariante sono ancora valide perchè *i* non è variato.

2) Se $s[k] \ne t[j]$, allora naturalmente la stringa *s* non si trova in *t* con inizio in *t[i]*, e quindi (poichè per ipotesi non si trova in *t[1..i-1])* non si trova in *t* con inizio in *t[1..i]*; ma poichè in tal caso si incrementa *i*, si ha di nuovo che
s non si trova in t con inizio in t[1..i-1].
Inoltre la seconda parte dell'invariante $s[1..k-1] = t[i..j-1]$ diventa, dopo le assegnazioni $j{:}= i$ e $k{:}= 1$, equivalente a $s[1..0] = t[i..i-1]$, che è banalmente vera trattandosi di due sottovettori vuoti; poichè infine era $i \le sup$, ora sarà $i \le sup+1$; ecc.

Fine della dimostrazione della Proposizione 1.

Bisogna ora dimostrare che la condizione di uscita dal ciclo e la successiva istruzione condizionale per la restituzione del risultato garantiscono in ogni caso una risposta corretta. Bisogna cioè dimostrare che, se valgono l'invariante e la condizione di uscita dal ciclo, allora:

1) se $k>m$, allora s compare in t con inizio in $t[i]$, cioè nella posizione i;
2) altrimenti (cioè se $k \le m$) s non compare in t.

Chiamiamo *CU* la condizione di uscita, cioè la negata della condizione del while:

$CU: \quad (j>n) \lor (k>m)$

Dimostriamo separatamente 1 e 2.

1) $INV \land CU \land (k>m) \Rightarrow$ *(s compare in t in posizione i)*

Dimostrazione: se $k>m$, allora la stringa è stata trovata nella posizione i ed è <u>interamente contenuta in *t*</u> perchè j-1 non può superare la fine del vettore t stesso. Un po' piú rigorosamente:
Se $k>m$, poichè l'invariante ci assicura che $k \le m+1$, deve essere $k = m+1$; allora, sostituendo nell'invariante otteniamo:

(s non compare in t con inizio in t[1..i-1]) $\land$ *(s[1..m] = t[i..j-1])*

Per la terza componente dell'invariante si ha $j \le n+1$, cioè $j-1 \le n$, quindi la stringa s compare con inizio in $t[i]$ ed è tutta contenuta in t, e non compare prima in t (cioè si è trovata la <u>prima</u> occorrenza).

2) $INV \land CU \land (k \le m) \Rightarrow$ *(s non compare in t)*

Dimostrazione: se non si è usciti perchè si è trovata *s*, allora si è usciti perchè è finito il testo e *s* non si è trovata. Un po' piú rigorosamente:
Poichè per $INV \land CU$ è $(i=sup+1) \lor (k=m+1)$, se non vale la seconda componente della disgiunzione deve valere la prima, cioè si deve avere $i=sup+1$; sostituendo nell'invariante otteniamo

s non compare in t con inizio in t[1..sup]

e sappiamo che in tal caso *s* non può comparire in *t* (perchè a destra di *sup* "non ci sta piú").

Esercizio 3.
Scrivere in modo dettagliato la dimostrazione di correttezza della soluzione riportata a pag. 87 dei lucidi, utilizzando anche la traccia riportata a pag. 88-sinistra.

5.11 Realizzazione del crivello di Eratostene con uso di un vettore di booleani.

Il crivello di Eratostene per la generazione dei numeri primi è, al pari dell'algoritmo di Euclide, uno dei piú vecchi algoritmi del mondo, tramandatoci dai Greci.

Ricordiamo che un numero naturale maggiore di 1 si dice primo se è divisibile soltanto per 1 e per se stesso; un noto teorema della teoria dei numeri stabilisce che vi sono infiniti numeri primi. I primi numeri primi sono: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ...

Il funzionamento dell'algoritmo del crivello può essere descritto metaforicamente nel modo seguente.

Si vogliano trovare tutti i numeri primi compresi fra 2 ed *n* (con *n* naturale), ad esempio fra 2 e 1000. Allora si dispongono in un crivello tutti gli *n-1* numeri naturali da 2 ad *n* in una sequenza ordinata; si eliminano tutti i multipli di 2 superiori a 2 stesso, poichè sicuramente non sono primi; il primo numero rimasto fra quelli superiori a 2 è 3, che è primo; si eliminano (lasciandoli cadere attraverso i buchi del crivello!) tutti i multipli di 3, poichè sicuramente non sono primi; il primo numero superiore a 3 rimasto è 5, che è primo; allora si eliminano tutti i multipli di 5, e cosí via fino ad *n*. Alla fine i numeri rimasti nel crivello sono tutti e soli i numeri primi compresi fra 2 ed *n*, come le pagliuzze d'oro trattenute nel crivello del cercatore dopo che è stata setacciata via tutta la sabbia.

Per tradurre tale algoritmo in una procedura informatica realizziamo il crivello per mezzo di un vettore *primo* di *n-1* elementi di tipo booleano: i valori *i* dell'indice del vettore, compresi fra 2 ed *n*, rappresentano i numeri naturali inizialmente immessi nel crivello, e per ogni *i* il valore *true* o *false* dell'elemento *primo[i]* rappresenta rispettivamente la presenza o l'assenza (per effetto di un'avvenuta eliminazione) del numero *i* nel crivello.

Inizialmente tutti gli elementi del vettore saranno posti a *true*; ad ogni passata l'eliminazione di un certo insieme di numeri sarà realizzata mettendo a *false* i corrispondenti elementi; alla fine i valori *i* degl'indici per i quali è rimasto *primo[i] = true* saranno i numeri primi cercati.

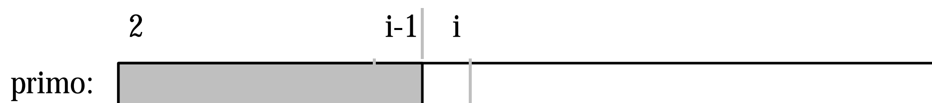
Dalla descrizione del funzionamento dell'algoritmo riportata sopra non è in realtà del tutto chiaro perchè l'algoritmo sia corretto. Pur essendo infatti evidente che i numeri via via eliminati non sono primi, può restare il dubbio che i non-primi (o numeri composti) non vengano eliminati tutti, e che qualcuno possa rimanere nel crivello; in particolare, può non essere chiaro perchè ad ogni passata dichiariamo che il primo numero non cancellato (nella parte di sequenza ancora da esaminare) è primo. Chiariamo allora i dettagli, costruendo un programma Pascal che realizzi l'idea di cui sopra e di cui dimostriamo contemporaneamente la correttezza.

Stabiliamo preliminarmente una terminologia abbreviata ma intuitiva; precisamente, diciamo che:

l'indice *i* è superstite in *2..n* se *primo[i] = true;*

l'indice *i* è cancellato in *2..n* se *primo[i] = false*.

Descriviamo ora la situazione al generico passo d'iterazione, cioè esprimiamo l'invariante del ciclo:



INV:
i superstiti nell'intervallo 2..i-1 sono tutti e soli i primi in tale intervallo
e i superstiti nell'intervallo i..n sono tutti e soli i naturali in tale intervallo che non hanno divisori in 2..i-1.
cioè, in modo leggermente piú formale:

$\forall j .\ 2 \le j \le i-1 \Rightarrow (primo[j] \leftrightarrow j \text{ è primo}) \wedge$
$\quad i \le j \le n \Rightarrow (primo[j] \leftrightarrow (\forall k .\ 2 \le k \le i-1 \Rightarrow k \text{ non divide } j))$

La condizione di uscita è naturalmente *CU:* $i = n+1$; sostituendo infatti nell'invariante $n+1$ ad i si ottiene proprio la condizione finale desiderata, cioè:
CF: i superstiti nell'intervallo 2..n sono tutti e soli i primi in tale intervallo.

L'invariante può essere reso inizialmente vero con l'inizializzazione $i = 2$ (e con i naturali in *2..n* tutti superstiti, cioè con tutti gli elementi del vettore inizializzati a *true*); infatti INV diventa in tal caso:
i superstiti in *2..1* sono tutti e soli i primi in *2..1* e i superstiti in *2..n* non hanno divisori in *2..1*, il che è vacuamente vero perchè l'intervallo *2..1* è vuoto.

Dall'inizializzazione e dalla condizione di uscita ricaviamo allora che il ciclo sarà della forma:

```
for i:= 2 to n do ...
```

Scriviamo ora il corpo del ciclo in modo da mantenere la verità dell'invariante.
Consideriamo l'indice *i*:

- se *i* è cancellato, allora per INV ha divisori in *2..i-1*, quindi non è primo; allora i superstiti in 2..i sono ancora tutti e soli i primi in tale intervallo, e i superstiti in *i+1..n* non hanno divisori in *2..i*, perchè per INV non hanno divisori in *2..i-1*, e d'altra parte se avessero come divisore *i*, che non è primo, avrebbero divisori in *2..i-1*;
  quindi l'invariante continua a valere anche se si incrementa *i* di *1*;

- se *i* è superstite, allora per INV non ha divisori in *2..i-1*, quindi per definizione è primo; allora anche in questo caso i superstiti in *2..i* sono ancora tutti e soli i primi in tale intervallo;
  i superstiti in *i+1..n* non hanno divisori in *2..i-1*; affinchè non abbiano divisori in *2..i* basta cancellare tutti i multipli di *i* (escluso *i* stesso);
  cosí anche in questo caso si può incrementare *i* di 1 mantenendo l'invariante.

Il programma ha allora la forma seguente:

```
for i:= 2 to n do primo[i]:= true;

for i:= 2 to n do
  if primo[i] then begin
    cancella tutti i multipli di i escluso i stesso
  end;
```

I multipli di *i* sono naturalmente dati da *2i*, *2i+i*, *2i+i+i*, ecc.
Osserviamo inoltre che se un numero naturale *m* ha un divisore *d*, allora ha anche come divisore $d_1 = n/d$, e almeno uno di tali due divisori deve essere $\le \sqrt{n}$; infatti se fosse
$d_1 > \sqrt{n}$ e $d_2 > \sqrt{n}$ avremmo $d_1 * d_2 > n$.

La versione completa del programma che visualizza i numeri primi compresi ad esempio fra 1 e 1000 è:

```
const n = 1000;
var primo: array[2..n] of boolean;
    i: integer;

begin
  for i:= 2 to n do primo[i]:= true;

  for i:= 2 to trunc(sqrt(n)) do
    if primo[i] then begin
      j:= 2*i;
      while j<=n do begin
        primo[j]:= false;
        j:= j+i;
      end;
    end;

  for i:= 1 to n do if primo[i] then write(i:4);
  writeln;
  readln
end.
```

Capitolo 6.
Vettori ordinati.

Capitolo 7.
Primi algoritmi di ordinamento.

7.1 Uguali ma diversi: stabilità.

Un algoritmo di ordinamento si dice stabile se non muta la posizione relativa di elementi fra loro uguali, cioè se nell'ordinare una sequenza di elementi mantiene fra elementi uguali l'ordine con cui essi compaiono nella sequenza di ingresso.

Naturalmente la nozione di stabilità è importante soltanto se elementi uguali rispetto all'ordinamento (ad esempio aventi uno stesso valore di una chiave o campo) sono però distinti per altri campi.

Gli algoritmi di ordinamento stabili permettono di ordinare una sequenza di elementi secondo una combinazione di campi nel modo seguente.

Si supponga, ad esempio, di voler ordinare una sequenza di record di persone in base all'ordine alfabetico del cognome, poi - a parità di cognome - per ordine alfabetico del nome, infine - a parità di cognome e nome - per anno di nascita.

Se disponiamo di un algoritmo di ordinamento stabile basterà, per ottenere l'ordinamento desiderato, ordinare la sequenza prima per anno di nascita, poi per nome, e infine per cognome!

Infatti come risultato dell'ordinamento per anno, avremo ovviamente tutti i record disposti in ordine cescente dall'anno minimo all'anno massimo; prendendo poi la sequenza cosí ottenuta e ordinandola per nome se l'algoritmo è stabile esso manterrà, per ogni gruppo di nomi uguali, il precedente ordinamento per anno di nascita; infine, ordinando per cognome, se l'algoritmo è stabile esso manterrà, per ogni gruppo di cognomi uguali, il precedente ordnamento per nome ed anno.