

Introduzione
alla Programmazione in Unix-Bash
con Elementi di Teoria della Calcolabilità

Salvatore Caporaso
Dipartimento di Informatica dell'Università di Bari
caporaso@di.uniba.it

May 11, 2008

Indice

(in progress)

1	Introduzione a Unix	3
1.1	Preliminari	3
1.2	Notazioni	7
1.3	Editing ed Esecuzione di Script Bash	14
1.4	Saper leggere e scrivere	16
1.5	Comandi composti e sub-shell	18
2	Variabili	21
2.1	Array	22
2.2	I parametri della riga di comando	24
2.3	Variabili predefinite	25
2.4	Virgolette	29
2.5	Manipolazione dei parametri posizionali	35
3	Espressioni	37
3.1	Classificazione e priorità	37
3.2	Filename expansion	39
3.3	Espansione delle variabili	41
3.3.1	Gestione di variabili non definite	42
3.3.2	Modifica di stringhe	45
3.4	Aritmetica	46
3.5	L'exit status e i comandi <code>return</code> e <code>source</code>	50
3.6	Quid est veritas?	53
3.7	Se le variabili sono troppe e troppo lunghe	60
3.8	L'interpretazione della riga di comando	61
4	Due forme del teorema della ricorsione	64
4.1	Un programma C che si auto-riproduce	64

4.1.1	Il compilatore non nasce imparato	64
4.1.2	La costruzione del quine	67
4.1.3	L'inserimento di un cavallo di Troia	71
4.2	Una forma debole del teorema di ricorsione	72
4.2.1	In un linguaggio generico	72
4.2.2	La forma debole in C	76
4.3	La forma debole in Bash	83
4.4	La forma forte del teorema	86
4.4.1	Premessa	86
4.4.2	La forma forte per un linguaggio generico	88
4.4.3	La forma forte in Bash	89
4.5	Funzioni universali e programmi totali	93
4.6	Indecidibilità e ricorsività	94
5	Funzioni	102
6	Processi	108
6.1	System call	108
6.2	Notazioni	122

Capitolo 1

Introduzione a Unix

1.1 Preliminari

1 Come è cominciato Ci sono OS piú o meno grandi. Quando nel 1970 Multix (Bell Lab della ATT, industria telefonica, e MIT, università) fu sospeso perché troppo caro, Ritchie e Thompson scrissero il progetto di massima di un nuovo sistema; Thompson se lo implementò in Assembler sul suo PDP7, e Kernighan propose puntigliosamente di chiamarlo Unix. Appena Ritchie inventò il C, Unix vi fu riscritto nel '73. Le idee di programmazione si rivelarono indipendenti dall'architettura, e fu possibile *portare* Unix su altre macchine (Sun, Apple), semplicemente ricompilandone il sorgente C, con pochi aggiustamenti. Berkeley sostituì MIT per un certo tempo. Quando dovette ritirarsi, ancora per mancanza di fondi, AT&T, per prima cosa, depositò la nuova versione 5, e pretese di farsela pagare 500-2000 dollari. Si narra che in un'agitata riunione tra universitari, l'olandese Tanenbaum scommise che in una settimana si sarebbe fatto il suo OS. E così fu. Nacque il sistema a *micro-kernel* Minix. Un kernel da pochi mega si limita a gestire, in modo protetto, l'IPC tra programmi, tutti in spazio utente, per la gestione di: file system, allocazione di memoria, program scheduling etc. Si narra anche che lo studente finlandese Linus Thorvalds volesse fare dei cambiamenti a Minix per collegarsi da casa col suo 386, ma che naturalmente l'anziano prof glielo proibì a colpi di copyright. Così lui ricominciò da zero, ma in logica *macro* kernel.

2 Un kernel non è un OS Invece di "Linux" dovremmo dire "GNU-Linux" perché in realtà Linux è solo il kernel. Il kernel è un SW che realizza un ambiente costituito da *syscall* (system call), rispettando uno standard, che si chiama Posix ¹.

¹anche Windows sostiene di essere *Posix compliant*, dalla versione NT in poi.

Esempi di syscall sono `cd` che cambia la PWD (present working directory), e `echo` che manda in stdout il suo argomento. Sopra di esse ci sono librerie ed applicazioni che formano la personalità del sistema operativo. Tra le applicazioni ci sono `gcc` e sia le GUI che le *shell*. Queste ultime sono interpreti che forniscono un ambiente per le righe di comando e per gli *script*. Bash è la shell di default in Linux. Le shell a loro volta hanno bisogno di applicazioni, per esempio un editor in linea per scrivere gli script, come Emacs. Il kernel è un SW *open source* scritto in ambito GNU (vedi sulla rete per questa sigla). Ma quasi tutto quello che non è Linux è costituito da software GNU (in particolare `bash`, `emacs`, `gcc`²).

3 Shell Le shell sono interpreti che interfacciano utente e OS. Possono

- eseguire in modo interattivo la *command line*, la cui forma è $(N, K \geq 0)$

```
comando -opz1 ... -opzN par1 ... parK
```

prima il comando, poi le *opzioni*, infine i *parametri*. Lo spazio è l'unico separatore. Niente ridondanze: quello che dice

```
COMMAND=CD;ARG=aaa;
```

in Unix da quarant'anni si dice col piú sobrio comando

```
cd aaa
```

- eseguire degli *script*, formati da sequenze di comandi e costrutti di controllo (`if`, `for`, `while`, etc.). I linguaggi di script sono a livello piú alto del C, e sono usati sia per l'amministrazione dei sistemi, sia per la redazione veloce di prototipi di programma, non ancora ottimizzati ma portabili. Si stima che il rapporto tra tempi di redazione come script e in C sia 1:10. Inoltre vi si possono fare cose che in C non sono facili. Per esempio vedremo che lo script

```
for i in *; do
if [ ${i##*.} = mp3 ]; then cp $i ../aaa/$i; fi; done
```

copia i file `.mp3` dalla PWD alla dir `../aaa`, cosa non facile in C.

²La tradizione Unix vuole iniziali maiuscole per i linguaggi, e minuscole per i programmi che li implementano. Quindi `bash` è un interprete Bash e `emacs` è un compilatore Emacs.

Il parsing e l'esecuzione di una riga di comando sono molto complicati, e ne parleremo poi. Cominciamo col dire che quando la shell trova un comando

- se è un alias, una funzione shell (programmata dall'utente) o un built-in, lo esegue, e passa al prossimo;
- se è un comando esterno (script, comando Linux, eseguibile C, compilazione L^AT_EX, etc), lancia un nuovo processo di se stessa con la syscall `fork` e aspetta che questo termini, oppure, se richiestone, lo mette in *background* e continua (anche di questo parliamo dopo). I comandi esterni, si devono trovare in una delle *dir eseguibili*, elencate dalla variabile (personalizzabile) di *environment* `PATH`, e devono avere il permesso di esecuzione (non però i binari C che lo ricevono da `gcc`). Usare `fork` comporta una duplicazione di tutte le variabili, ed altre diseconomie di tempo e spazio.

In genere le priorità sono: alias, funzioni, built-in, comandi esterni. Un utente stupido e cattivo può, per esempio, sostituire il built-in `echo` con

```
alias echo='rm -r / '
```

e cedere innocentemente il posto ad altri, che se lancia il comando

```
echo ciao
```

invece di farsi dire ciao dallo stdout, dice lui ciao a tutta la partizione di root.

Una volta lanciata, la shell comincia un loop senza fine, creando uno stack potenzialmente infinito di sue copie, finché non trova il comando `exit`.

4 Unix si basa su due astrazioni

- Tutto è un *file*
- Se c'è qualche cosa che non è un file, allora è un *processo*

Ogni file è solo una stringa, possibilmente vuota. Esso contiene solo l'info usata dall'applicazione che lo usa, e nessuna info per il FS (file system). Per esempio, non dice la sua lunghezza, e *non* termina con il EOF. I file possono essere *regolari*, *dir*, *link*, *device*, ed altre cose. A ciascuno è associato un numero detto *inode*, usato dal FS. Da un punto di vista logico, il FS è un albero di file. Il *parent* di ogni file è una *dir* che contiene tutte le informazioni meta su di lui (lunghezza, inode, natura) e sui

suoi fratelli³. Ad ogni file si accede fornendo il suo *path* (assoluto a partire dalla radice /, o relativo, a partire dalla PWD ⁴).

Alcune differenze nei confronti di win sono:

- le *extension* hanno poca o nulla importanza (svaniscono le fisime testo contro eseguibile); il sistema capisce da solo quale applicazione chiamare senza bisogno di `.mp3`, `.wav`, `.pdf`, `.jpg` etc. (anche se possono servire a fissare preferenze di apertura diverse: L^AT_EX per i `.tex` e OpenOffice per i `.doc`);
- le dir sono file, non cose scritte in uno zoo di tabelle diverse;
- ogni device è un FS che, per essere accessibile, deve essere *montato* nel FS, divenendone un sotto-albero (la cui radice è una dir detta *mount point*, fissata dalla syscall `mount` lanciata dall'utente; oppure prefissata al momento della installazione di Linux, e vista durante il booting). Sparisce il folklore dei comandi ad hoc per CD, diversi HD e FS (ntfs, FAT, etc). Un FS è sempre e solo un albero di file con *metodi* di apertura, scrittura, etc. non trasparenti neanche per il programmatore di altre parti del sistema ⁵.

L'astrazione che tutto è un file porta con naturalezza alla definizione di programmi, detti, per le ragioni che vedremo subito, *filtri*, che non devono fare nessuna ipotesi sull'I/O perché rispettano lo standard I/O che conosciamo dal C: *stdin* (tastiera), *stdout* e *stderr* (monitor).

Dei processi (programmi in esecuzione, o, piú rigorosamente: *contesti di un programma*) parliamo tra un pò.

5 Fiat session Sappiamo che nelle architetture x86 la presidenza spetta a BIOS. Per default, lui guarda nel MBR del primo HD (oppure dove l'avete settato voi) per sapere a chi deve dare la parola. Se ci trova `lilo` (Li-nux lo-ader) lo esegue. Lilo sa quanti sono gli OS, dove sono le rispettive root, etc. Se te lo sei settato in *dual booting* e scegli sul menu Linux (invece di win), lui avvia il boot del kernel. (Da qualche anno Grub tende a prevalere su Lilo.) Linux non si ricorda lo (o non si fida dello) HW, e lo controlla durante il boot, dando messaggi su video che, finché

³Il fatto che una dir contenga informazione utile per il FS non contraddice ma conferma il principio che i file contengono solo info utile per il programma che li usa. Semplicemente il FS è il programma che usa le dir.

⁴Attenzione allo *slash* ordinario nei path e alle mai/min-uscole: `/bin/bash` e non `\bin\Bash`; attenzione anche al fatto che spesso la *home dir* dell'utente si chiama `/root`, ma non è la *radice* / del FS (la quale è il solo file che non ha parent.)

⁵metodi come nella filosofia degli oggetti, ma rigorosamente in C

tutto va bene, nessuno legge o capisce. Quando si placa, lancia il programma `init`, e si mette a fare il manager, facendo lavorare gli altri. `init` è il primo e l'ultimo programma lanciato da Linux. In genere guarda il FS, l'orologio, la rete, e comincia le `fork` tra processi. Se il sistema è predisposto per cominciare da linea di comando invece del GUI (magari per la buona ragione che non l'avete ancora installato) `init` avvia: `getty` (gestione della console), `login`, e una shell. Da ora in poi è la shell a dare il *prompt*, ad interpretare i comandi da stdin e gli script.

1.2 Notazioni

6 Convenzioni tipografiche Il testo è diviso in paragrafi (questo è il §6; → 99 significa: vedi § 99). In quasi tutti c'è un testo principale nel quale sono inserite delle righe in font `typewriter` come

```
509 ~/bm -> echo buongiorno, Padrone          <return>
buongiorno, Padrone
```

sono trascrizioni dal monitor per le quali mi attengo ad alcune regole

- 509 ~/bm -> è il *prompt* di `bash`, personalizzato per queste dispense in cui
 - il numero iniziale è un identificativo (locale e non progressivo, perché uso macchine e console diverse, ci faccio cose diverse, e la *history* da cui è tratto cicla); *l* 509 sarà nel seguito un riferimento a quella riga;
 - tra il numero e la freccia c'è PWD; Linux abbrevia le dir così
 - ~ per home dir; . per PWD; .. per parent of PWD
 - In questo caso, `/bm` è la sotto-dir di `~`, dedicata a queste dispense.
 - Però qualche volta invece di tutto sto prompt scrivo solo un `$`.
- < ... > dice che sono stati premuti i tasti invisibili ... (quando non ovvio: quindi è difficile che scriva ancora < `return` >). La tradizione vuole che `^C` stia per *ctrl* e `^M` per *alt* (nei primi Unix il tasto alt si chiamava *Meta*).

Esempio importante In Unix l'EOF si dà con `<^C+D>`

- Le righe *non* indentate sono quelle che l'ultimo comando (o `bash` per conto suo) ha mandato in stdout; qui sopra: `Buongiorno, Padrone` (→ 8 per quelle indentate).

7 Font e variabili sintattiche Nel testo principale, uso il *corsivo* per *variabili sintattiche*, ossia generici identificatori scelti dall'utente per variabili, nomi di file, etc. Invece le costanti sintattiche, ossia nomi propri assegnati a particolari e (sperabilmente) unici oggetti, sono in **typewriter**. Però uso il corsivo anche per

- (a) termini in corso di definizione o introduzione;
- (b) espressioni usate in senso traslato, informale, etc.

Per esempio `echo stringa` è un comando preciso seguito da una stringa generica. Siccome il rigore non dovrebbe escludere l'intelligenza, applico questa distinzione solo quando mi sembra che ci sia una qualche possibilità, magari remota, di equivoco. Nel senso che quando dico che $5+7$ è un termine, mi aspetto che si capisca che anche $4+8$ lo è, senza che ci sia bisogno di fare i pignoli, precisando che “per ogni $h \in \mathcal{N}$ e ogni $k \in \mathcal{N}$ l'espressione $h + k$ è, per definizione, un termine”.

Ma gli informatici per sopravvivere devono coniugare rigore e praticità, e hanno inventato due modi molto più semplici del corsivo per fare discorsi generali:

- (a) `foo` e `bar` sono nomi generici (un Tizio e un Caio);
- (b) un prefisso possessivo, come in `myfile` oppure in `yourstring`.

8 Distinzione tra file e suo contenuto `(foo)=bar` significa che nel file `foo` è registrata la stringa `bar`. Se è uno script scrivo su una colonna indentata, applicando l'equivalenza Bash tra `'` e `'newline'`, con delle barre marcatrici a sinistra

```
| (nuovo) =  
| echo buon giorno  
| echo padrone
```

per eseguire questo script, posso usare il comando

```
chmod 755 nuovo
```

oppure dargli un permesso temporaneo di esecuzione, prefiggendo un dot (\rightarrow 143); già che ci siamo, notiamo che `echo` va a capo senza farselo dire (ricordo che `$` è la forma corta del prompt)

```
$ . nuovo  
buon giorno  
padrone
```

Una differenza importante tra Bash e linguaggi come il C, e comunque una cosa di cui ci occuperemo molto, è che uno script può creare un file che, a sua volta, contiene uno script. Per parlare di questi casi dobbiamo complicare le convenzioni del §6.

```

$ command par1 ... parK
  | (foo) =
  | s1
  | ...
  | sM
t1
...
tN

```

significa che la linea di comando `$...` crea il file `foo` che contiene quelle $M \geq 0$ linee *e/o* manda allo stdout $N \geq 0$ linee. Per esempio, il segno `>` nella linea seguente

```
$ comm par1 ... parK > foo
```

provoca una *ridirezione* nel file `foo` di quanto `comm par1 ...parK` manderebbe allo stdout (sostituendo il contenuto precedente ed eventualmente creandolo se non esisteva). Ossia

```

$ echo "s1;...;sK" > foo
  | (foo) =
  | s1
  | ...
  | sK

```

e pertanto

```

503bm -> echo "echo echo ciao > dimmi_ciao" > scrivi_dimmi_ciao
  | (scrivi_dimmi_ciao) =
  | echo echo ciao > dimmi_ciao
505bm -> . scrivi_dimmi_ciao
  | (dimmi_ciao) =
  | echo ciao
507bm -> . dimmi_ciao
ciao

```

9 Comandi Unix essenziali `cat` manda in stdout quanto riceve in stdin fino all'EOF

```

502~/bm -> cat
uno
                <return>

```

```

uno
due                                <return>
due
mo' basta                          <return>
mo' basta                          <^C+D>
503~/bm ->

```

Ci si possono fare cose un pò meno sceme: `stdin/out` sono default. `cat > foo` *ridirige* stdout nel file `foo`; invece `cat bar` sostituisce `stdin` col file `bar`.

```

508~/bm -> mkdir aaa
509~/bm -> cat > aaa/myfile
pi=3.1415...
e=2.718...                          <^C+D>
510~/bm -> cat aaa/myfile
pi=3.1415...
e=2.718...

```

ℓ 508 il comando `mkdir aaa` crea una nuova dir chiamata `aaa`.

ℓ 509 crea il nuovo file `myfile` nella dir `aaa` e ordina di metterci `stdin` fino all'EOF

ℓ 510 manda il file in `stdout`.

Osserviamo che lo stream da `stdin` è interrotto da EOF e non da `return`.

Il comando `ls mydir` lista i file contenuti in `mydir`; il comando `rm foo` rimuove il file `foo`; tutta la dir `bar` col suo eventuale contenuto è rimossa da `rm -r bar` con l'opzione `-r` (= ricorsivamente)

```

512~/bm -> rm myfile
513~/bm -> ls aaa
myfile
514~/bm -> rm aaa/myfile
515~/bm -> ls aaa

```

ℓ513 perché `myfile` sta in `aaa` e non in `PWD`.

ℓ512 non avvisa che il file non esiste.

Osserviamo che se gli dici di mandare qualcosa in un file che non esiste, lo crea. Se fai `cat > foo` e `foo` esiste, il contenuto precedente viene perduto. Se vuoi *appendere* devi scrivere `>>`.

```

517~/bm -> cat > aaa/foo

```

```

pi=3,...
519~/bm -> cat > aaa/foo
e=2,...
520~/bm -> cat aaa/foo
e=2,...
521~/bm -> cat >> aaa/foo
pi=3,14...
522~/bm -> cat aaa/foo
e=2,...
pi=3,14..

```

*ℓ*519 distrugge il pi=3,...; invece *ℓ*521 non distrugge il e=2,...
 Simmetricamente, si può ridirigere lo stdin con <
 Combinando le due ridirezioni, possiamo usare cat per copiare un file

```

512~/bm -> echo usiamo cat per copiare >new
513~/bm -> cat new
usiamo cat per copiare
514~/bm -> cat < new > newer
515~/bm -> cat newer
usiamo cat per copiare

```

Ma la ridirezione dell'input è poco usata. Per copiare e spostare ci sono cp e mv

10 Esercizio Spiegare il comportamento seguente

```

558~/bm -> mkdir aaa
559~/bm -> echo primo > aaa/primo_file
560~/bm -> ls aaa
primo_file
561~/bm -> mkdir aaa/bbb
562~/bm -> ls aaa
bbb primo_file
563~/bm -> cd aaa
564~/bm/aaa -> cp primo_file bbb
565~/bm/aaa -> ls
bbb primo_file
566~/bm/aaa -> echo secondo > aaa/primo_file
bash: aaa/primo_file: No such file or dir

```

```

567~/bm/aaa -> echo secondo > primo_file
568~/bm/aaa -> cat primo_file
secondo
569~/bm/aaa -> mv primo_file bbb/primo_file
570~/bm/aaa -> ls
bbb
571~/bm/aaa -> cat bbb/primo_file
secondo
572~/bm/aaa -> rm bbb
rm: impossibile rimuovere 'bbb': Is a dir
573~/bm/aaa -> rm aaa/bbb
rm: impossibile rimuovere 'aaa/bbb': No such file or dir
574~/bm/aaa -> rm -r bbb
574~/bm/aaa -> ls
574~/bm/aaa ->

```

La mv di ℓ 569 ha sostituito un file precedente? Se cosí fosse, ne dedurremmo che il sistema non avvisa quando uno distrugge accidentalmente un file.

11 Pipe (= *tubo*) Il segno | provoca il *pipe* dell'uscita del programma alla sua sinistra al programma alla sua destra.

```

527~/bm -> cd aaa
528~/bm/aaa -> ls
bbb ccc
529~/bm/aaa -> ls | wc -l
2
530~/bm/aaa -> touch forza_roma
531~/bm/aaa -> ls | wc -l
3
532~/bm/aaa -> touch forza\ roma
533~/bm/aaa -> ls
bbb ccc forza roma forza_roma
534~/bm/aaa -> ls | wc -l
4
535~/bm/aaa -> ls | wc -w
5
536~/bm/aaa -> ls | wc
      4      5      30

```

```

541~/bm -> wc
\
      1      1      3
544~/bm -> wc
\
\
      2      2     74

```

ℓ 529 perché il comando `wc` con l'opzione `-l` restituisce il numero delle righe

ℓ 530 perché il comando `touch myfile` aggiorna i *timestamp* di `myfile` e lo crea se non esiste

ℓ 532 perché `\` protegge il carattere *spazio* e quindi `touch` crea un solo file

ℓ 533 perché `ls` senza l'opzione `-f` elenca i file in ordine Ascii

ℓ 535 perché `wc` con l'opzione `-w` restituisce il numero delle parole; invece per default dà i numeri delle righe, parole e caratteri

ℓ 541 dice che l'input di default per `wc` è `stdin` e dà 3 perché abbiamo un blank protetto, il blank e `<return>` mentre l'EOF non conta

ℓ 544 perché ho messo un sacco di blank prima di andare a capo

Si possono fare catene di pipe

```

45~/bm -> echo ciao | wc
      1      1      5
46~/bm -> echo ciao | wc | wc
      1      3     24
47~/bm -> echo ciao | wc | wc | wc
      1      3     24
52~/bm -> echo ciao | wc | wc | wc
      1  3 24

```

12 Confronto tra ri-direzioni e pipe La ridirezione è tra un comando e un file (output verso un file, oppure input da un file). La pipe è tra due comandi. Abbiamo visto una sequenza di pipe che collegano in sequenza due o più comandi. Invece non si possono avere due ridirezioni, in uscita o due in entrata. La sintassi è

```
comm [-opts] [parms] [< foo] [> bar]
```

cosa manderemmo in `bar1` scrivendo `comm > bar > bar1?`

13 Filtro Un *filter* è un programma che ha stdin e stdout (e magari anche stderr) per default. I filtri si combinano bene con le pipe. L'unica interfaccia tra due vicini separati da una | è lo stream di bytes da sinistra a destra.

Esercizio Elencare quelli, tra i comandi visti fin qui, che sono filtri.

1.3 Editing ed Esecuzione di Script Bash

Bash ⁶ è la shell di default in Linux. È ottenuta dalla shell Unix originaria aggiungendo molte funzionalità, spesso prese dal C.

14 Script Uno script è una sequenza di comandi separati da un ';' o accapo:

- editato e registrato come eseguibile in una dir di eseguibili come `myscript`
- diviene un comando come gli altri e lo esegui con

```
myscript [opzioni] [args]          <return>
```

```
536~/bm -> cat > nuovo
echo -n Forza
echo \ Roma
537~/bm -> 755 nuovo
538~/bm -> nuovo
Forza Roma
```

ℓ 537 è un mio alias per `chmod 755` che dà a me tutti i diritti, e agli altri tutti, meno quello di buttare per aria la roba mia

ℓ 538 `echo` con l'opzione `-n` non va a capo

Tutto bene perché eravamo su una dir eseguibile. Se no devi usare il comando `source` con argomento `myfile`

```
542~/bm -> cd aaa
543~/bm/aaa -> cat > novo
echo Daje Lupi
544~/bm/aaa -> chmod 755 novo
```

⁶“Bourne Again Shell”, ma anche *shell rinata*, gioco di parole col nome del suo creatore Steve Bourne .

```
545~/bm/aaa -> novo
bash: novo: command not found
546~/bm/aaa -> source novo
Daje Lupi
547~/bm/aaa -> . novo
Daje Lupi
548~/bm/aaa -> cd ..
549~/bm -> novo
bash: novo: command not found
550~/bm -> . novo
bash: novo: No such file or dir
551~/bm -> . aaa/novo
Daje Lupi
```

ℓ 547 perché `'.'` è un'abbreviazione Bash per `source`

ℓ 548 perché `'.'` abbrevia pure PWD, e `'..'` sta per “padre della PWD”

ℓ 549 perché le sottodir di una dir eseguibile non è detto che lo siano

15 Emacs è un editor in linea, da usare in mancanza di GUI

```
561~/bm -> rm nuovo
562~/bm -> emacs nuovo
563~/bm -> cat nuovo
echo a
echo b
echo c
                                     <^X ^C yes >
564~/bm -> 755 nuovo
565~/bm -> nuovo
a
b
c
```

ℓ 562 ha chiamato Emacs che ha aperto una finestra sulla quale scrivere. Alla scorciatoia da tastiera `^X ^C` Emacs chiede se vuoi uscire e salvare o no, e pretende che uno risponda “yes/no” (qualche volta gli basta `'y'`). Io ho scritto 3 comandi e 2 accapo, e sono uscito

ℓ 563 et voila

Se invece usate un'interfaccia grafica, pazienza. Continuate pure cosí, non siete sulla strada per divenire programmatori di sistema, ma non è detto che, solo per questo, finiate ad occuparvi di Intelligenza Artificiale. Aprite un editor, scriveteci il vostro script, salvatelo, e guardate se funziona.

16 Sha-bang Bash su un comando esterno prova prima a vedere se è un compilato in una cartella `/bin`; poi se è uno script da interpretare. Se è uno script Perl, Bash, etc. con lo *shah-bang* `#!` gli dai il path del binario di `perl`, `bash` etc. Se il path del binario è `/bin/bash`, uno script Bash `foo.sh` dovrebbe quindi cominciare con

```
#! /bin/bash
# foo.sh
```

Il `#` comincia un commento (come nella seconda riga), anche a metà riga. Lo *shah-bang*, e certi *pattern matching* (\rightarrow `??`) sono eccezioni. In realtà `bash` riconosce benissimo i suoi script anche senza `#!` e senza `.sh`, e io non ce li metto piú.

1.4 Saper leggere e scrivere

17 printf La versione Bash di `printf`

- collabora senza farci perdere tempo con virgole punti-e-virgola e graffe
- ignora le variabili inesistenti
- capisce che deve ciclare sulle variabili della stringa di formato quando gli argomenti sono piú di quelli indicati.

```
~/bm --> a=12345 b=1 c=2 d=dromedario
~/bm --> printf "%d\t %s\n" $a $b $c $dromedario $d $a
12345    1
2        dromedario
12345
```

Si noti che basta uno spazio per separare le assegnazioni.

18 read assegna la stringa da `stdin` alla variabile passata come argomento o a quella di default `REPLY`; opzioni:

- `-n z` al massimo `z` caratteri

- `-p 's'` mostra la stringa `s` e aspetta stdin
- `-t` aspetta `z` secondi, poi esce, e dà il prompt;

la variabile predefinita `TMOU`T ha lo stesso effetto dell'opzione `-t`, ma nei confronti di qualunque prompt

```

502~/bm -> read a
123
503~/bm -> echo $a
123
504~/bm -> read
321
505~/bm -> echo $REPLY
321
506~/bm -> read -p "scrivi un numero " a
scrivi un numero 111          # opzione -p stampa un messaggio
511~/bm -> read -n 3 a        # opzione -n z = max z cifre
123512~/bm -> 45             # ho scritto 1...5 e lui ha continuato
bash: 45: command not found   per conto suo
513~/bm -> echo $a
123
514~/bm -> read -t 5 a        # ti sto ad aspettare per z secondi
123
515~/bm -> read -t 5 a
516~/bm -> echo $a           # nuovo prompt perche' non mi sono spicciato
123                          # almeno non aveva gia' distrutto $a
499~/bm -> TMOU=20          # da ora in poi ti aspetto per 20 secondi
500~/bm -> read a           # ho fatto passare 20 secondi
502~/bm -> echo $a
123
503~/bm -> timed out waiting for input: auto-logout
root@st5a11:~#              # dopo 20 sec pianta pure bash
root@st5a11:~# bash
498~ -> cd bm
499~/bm ->

```

19 Volendo, si possono anche fare dei pasticci

```

511~/bm -> var=prima
512~/bm -> echo poi | read var; echo $var
prima
513~/bm -> echo poi | (read var; echo $var)
poi
514~/bm -> (echo poi | read var; echo $var); echo $var;
prima
prima
515~/bm -> (echo poi | read var); echo $var
prima

```

non ho trovato documentazione soddisfacente per questo fenomeno; qualcuno raccomanda di non usare `read` nelle pipe perché in questo caso `bash` apre un processo nuovo le cui variabili non sono visibili da fuori; se così fosse bisognerebbe poi capire se è un baco o se c'è qualche ragione. Se scopro qualcosa nelle prossime settimane ve lo dico. `ℓ513` sembra confermare questa tesi: siccome `echo` è un built-in all'interno del blocco `(read var; echo $var)` il valore letto di `var` rimane visibile.

Nei prossimi paragrafi vediamo cosa sono queste parentesi.

1.5 Comandi composti e sub-shell

20 Command block = sequenza di comandi racchiusa tra parentesi graffe.

```

534 bm -> test || echo a; echo b;
a
b
535 bm -> test a || echo a; echo b;
b
536 bm -> test a || { echo a; echo b; }
537 bm -> test || { echo a; echo b; }
a
b

```

`ℓ534` Siccome il comando `test` in assenza di opzioni e con un solo argomento *testa* se esso è vuoto o no (ma vedi meglio nel § 72), qui si prova ad eseguire `echo a`; poi si passa all'altra `echo`

`ℓ535` il test è vero e non c'è bisogno di valutare `echo a`; si prosegue con `echo b`

ℓ536 nel caso vero non si valuta l'intero comando composto, e non si stampa nulla
ℓ537 nel caso falso abbiamo la stessa stampa di ℓ534 ma per ragioni diverse.

21 Subshell Invece le tonde definiscono una *sub-shell* che è un ibrido tra blocco e shell. L'altro giorno dovevo spedire molto materiale per una tesi di dottorato, e non mi ricordavo la sintassi del comando `tar`. Su un tabulato per Unix 7 vecchio di 25 anni ho trovato questa cosa, l'ho testata e funziona ancora benissimo

```
522~/bm -> cd ../ScriptVari
523~/ScriptVari -> mkdir ../newbm
524~/ScriptVari -> tar cf - . | (cd ../newbm; tar xpf -)
525~/ScriptVari -> ls ../newbm
bashmetodi kt MiscellaneaScritti
526~/ScriptVari -> ls
bashmetodi kt MiscellaneaScritti
```

mano mano che a sinistra impacchetta, a destra spacchetta, e, quando ha finito ci lascia nella stessa dir. (Ma la sintassi di `tar` in Linux è diversa e sarà meglio seguirla senza fare queste stranezze.)

22 Differenze tra sub-shell e blocchi I blocchi fanno a meno del punto-e-virgola alla fine, ma le graffe li vogliono; e vogliono pure lo spazio tra la graffa e il comando. Il perché è spiegato nel prossimo paragrafo

```
535~/bm -> { sleep 5; echo a }
==> bash: syntax error: unexpected end of file
536~/bm -> ( sleep 5; echo a )
a
537~/bm -> {sleep 5; echo a; }
bash: syntax error near unexpected token `}'
```

23 Keyword Le coppie `if-fi`, `do-done`, `case-esac`, e di parentesi tonde, quadre (→ 72) e graffe sono *keyword*. L'interpretazione di un comando comincia, subito dopo lo splitting in token, col vedere se si ha una di queste keyword: se è iniziale `bash` si organizza per un comando composto da più comandi, se è finale ha buone ragioni per protestare, e lo fa. Con le tonde sta più buono perché per lui non sono keyword (per ragioni storiche: le graffe pongono problemi di portabilità perché già c'erano, le tonde no, essendo più recenti).

24 Variabili nelle sub-shell e nei comandi composti Come le shell, anche le sub-shell hanno le loro variabili, e non possono modificare quelle della shell principale. Invece i comandi composti possono

```
545~/bm -> declare -ix a=5
546~/bm -> (a=10; echo $a)
10
547~/bm -> echo $a
5
548~/bm -> { a=10; echo $a;}
10
549~/bm -> echo $a
10
```

Capitolo 2

Variabili

Come tutti i linguaggi di programmazione, Bash ha le sue variabili, che possono essere dichiarate e inizializzate col comando `declare` e le opzioni `i`, `a`, `r`, `x`

```
428~/bm -> declare -i a=5          # int a
429~/bm -> declare -r b=6          # read-only
430~/bm -> declare -x c=7          # export
431~/bm -> declare -a nm=(s n vic) # array (solo a una dimensione)
```

Per default (ossia se non dichiarate) le variabili sono stringhe invisibili fuori dal processo corrente.

25 L'operatore di referenziazione L'accesso al valore della variabile è un'operazione a sé, chiamata *variable substitution*, e anche, in un contesto diverso (→ 51) *variable expansion*. Si usa il *referencing operator* `$`, che mappa il nome di una variabile nel suo valore, e talora si dice che `$` ha *vestito* la variabile.

```
110~/bm -> ciao=314
111~/bm -> echo ciao
ciao
112~/bm -> echo $ciao
314
```

Il parsing può essere ambiguo: meglio isolare il nome con graffe

```
507~/bm -> a=roma a1=bari b=$a1 c=${a}1 # niente ';'
508~/bm -> echo a=$a b=$b c=$c
a=roma b=bari c=roma1
```

```

332~/bm -> a_b=$a$b # concatenazione senza le storie del C
333~/bm -> echo $a_b
romabari
334~/bm -> a = bari
bash: a: command not found

```

ℓ334 niente spazi prima o dopo l'uguale; Bash capisce `a = roma` come il comando `a` con due arg, e protesta perché `a` non è un comando.

2.1 Array

26 Inizializzazione e accesso Gli elementi di ogni array sono inizializzati con la stringa vuota; poi tu te li puoi assegnare in vari modi

```

370~/bm -> declare -a n
371~/bm -> n[2]=sal n[1]=nick n[0]=vic
372~/bm -> n[5]=sal
374~/bm -> echo 2d=${n[1]} 5th=${n[4]} 6th=${n[5]} 15-th=${n[15]}
2d=nick 5th= 6th=sal 15-th=
375~/bm -> n=(a,b,c,d,e,f,g) # puoi inizializzare con elenco
376~/bm -> echo 2d=${n[1]} 5th=${n[4]} 6th=${n[5]} 15-th=${n[15]}
2d= 5th= 6th= 15-th= # ma non così': che c'entrano le virgole?
377~/bm -> echo ${n[0]}
a,b,c,d,e,f,g # ha creduto che fosse una stringa sola
378~/bm -> n=(a b c d e f g)
379~/bm -> echo 2d=${n[1]} 5th=${n[4]} 6th=${n[5]} 15-th=${n[15]}
2d=b 5th=e 6th=f 15-th=
387~/bm -> n=(a [2]=b c d e f g)
388~/bm -> echo 2d=${n[1]} 5th=${n[4]} 6th=${n[5]} 15-th=${n[15]}
2d= 5th=d 6th=e 15-th=

```

ℓ387 si possono saltare degli elementi, e lui continua ordinatamente.

I caratteri `*` e `@` accedono a tutto l'array assegnato; si comportano in modo diverso tra virgolette: con `@` si ottengono 7 stringhe, con la `*` una stringa ottenuta concatenando 7 stringhe, con un separatore fornito dalla variabile predefinita `IFS` (*internal field separator*) che per default è inizializzata col blank e altri caratteri composti bianchi (`\t`, `\n`, etc.)

```

392~/bm -> echo ${n[*]}
a b c d e f g
393~/bm -> echo ${n[@]}
a b c d e f g
394~/bm -> echo "${n[@]}"
a b c d e f g
400~/bm -> echo "${n[*]}"
abcdefg
401~/bm -> export IFS=,
402~/bm -> echo "${n[*]}"
a,b,c,d,e,f,g
404~/bm -> export IFS=\
405~/bm -> echo "${n[*]}"
a b c d e f g

```

Altra inizializzazione

```

435~/bm -> ar[7]=7 ar[654321]=1
436~/bm -> echo ${ar[*]}
7 1
437~/bm -> echo ${#ar[*]}
2 # numero delle componenti non vuote

```

Comodo per il SW di gestione di strutture sparse.

27 Non omettete le graffe quando volete accedere agli elementi di un array, perché non sono un orpello sintattico, ma chiamano un operatore di espansione (→ 51)

```

533~/bm -> n=(2 3 4)
534~/bm -> echo $n[2]
2[2]
535~/bm -> echo $n[@]
2[@]

```

ℓ534 perché, un pò come in C, il nome dell'array (ossia `n`) è l'indirizzo del primo elemento (ossia `2`), e il parsing dei parametri per `echo` prosegue prendendo alla lettera il seguito (risp. `[2]` e `[@]`).

2.2 I parametri della riga di comando

28 In C Rivediamo come `argc` e `argv` funzionano in C ¹

```
506~/bm -> cat > ciaok.c
main(int argc, char *argv[]){
    int i;
    printf("Buongiorno ");
    if (argc == 1)
        printf("Padrone");
    else
        for (i=1; i<argc; i++)
            printf("%s ",argv[i]);
    printf("\n");
    printf("Cosa comanda%s dopo %s?\n", (argc > 2)?"te":"", argv[0]);
}
508~/bm -> ciaok.bin a b c
Buongiorno a b c
Cosa comandate dopo ciaok.bin?
509~/bm -> ciaok.bin a
Buongiorno a
Cosa comanda dopo ciaok.bin?
510~/bm -> ciaok.bin
Buongiorno Padrone
Cosa comanda dopo ciaok.bin?
```

29 Parametri posizionali Quando si lancia uno script `foo` con `K` parametri, le variabili predefinite `1`, `ldots`, `K` sono automaticamente assegnate con i suoi parametri (per questo i nomi di variabili non devono cominciare con cifre). Inoltre la variabile `0` è assegnata con il cammino completo dello script, e non solo col nome del programma come in C (per forza: un programma C se vuole essere portabile non deve sapere dov'è che sta).

¹In logica un parametro è una variabile il cui valore rimane costante per un certo tempo: quando per studiare una parabola $y = ax^2 + bx + c$, si scelgono a , b e c , si fissano per la durata della discussione i valori di queste variabili; esse vengono trattate come parametri, mentre x e y conservano lo status di variabili. In un programma C si può volere che la sua `main` sia chiamata con dei valori assegnati al momento del lancio del programma, e che poi non vengono più modificati durante l'esecuzione. Essi sono dunque i parametri del programma, che avrà poi delle variabili che lui si assegna da solo. Evidentemente parametri ed input sono cose diverse.

```
584~/bm -> cat > nuovo
echo Buongiorno Padron $1 e Padron $2
echo $0 al vostro servizio
585~/bm -> nuovo Salvatore Nicola
Buongiorno Padron Salvatore e Padron Nicola
/root/bm/nuovo al vostro servizio
```

2.3 Variabili predefinite

30 Variabili di environment si distinguono da quelle di programma perché sono in maiuscolo o cominciano con una cifra decimale (per convenzione). Sono assegnate dal sistema o dall'utente, ma hanno un ruolo pre-definito. Oltre ai parametri posizionali, ce ne sono una cinquantina.

```
97~/bm -> pwd
/root/bm
98~/bm -> echo $PWD
/root/bm
```

`pwd` è un comando che restituisce il valore della variabile di environment `PWD`, la quale è assegnata con la working dir corrente.

31 Qualche variabile di environment

```
470~/bm -> echo PS1 = $PS1
PS1 = \!\w -> # nel linguaggio per definire il prompt \!
# e' il numero della history e w = $PWD
472~/bm -> echo PS2 = $PS2
PS2 = ==> # prompt secondario
473~/bm -> echo PWD = $PWD
PWD = /root/bm
474~/bm -> echo OLDPWD = $OLDPWD
OLDPWD = /root # PWD precedente
475~/bm -> echo PATH = $PATH
PATH = /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:
/root:/root/bm # le dir eseguibili
476~/bm -> echo RANDOM = $RANDOM
RANDOM = 27535
```

475 PATH dà le dir eseguibili, separate da ':'; qui ce ne sono 5 di sistema e 2 mie (l'accapo è mio, solo per ragioni tipografiche).

Le dir in PATH sono separate dal due-punti perché la *tilde-expansion* nel parsing riconosce ~ come \$HOME solo in due casi (secondo me, non ho trovato documentazione)

- dopo il due-punti
- nelle assegnazioni subito dopo l'uguale e lo slash

```
509 bm -> a=roma:~      b=~/bari      c=pi~sa  d=~lecce e=bari/~
510 bm -> echo "a=$a    b=$b    c=$c    d=$d    e=$e"
a=roma:/root    b=/root/bari    c=pi~sa  d=~lecce    e=bari/~
```

(senza le virgolette echo si sarebbe mangiato gli spazi).

UID è assegnata con lo user ID. Lo user *root* ha ID zero. Il comando *su foo* fa lo *switch* all'utente *foo*, e naturalmente passa alle variabili predefinite di *foo*. Quando vuoi ridivenire *root* ti chiede la PW (e ovviamente non la echeggia sullo stdout)

```
177~/bm -> if [ $UID = 0 ] ; then echo sei root;\ <return>
else echo non sei root; fi
sei root
178~/bm -> su capo
capo@st5a11:/root/bm$ if [ $UID = 0 ] ; then echo sei la root;\
else echo non sei root ma mamma ti vuole bene lo stesso; fi
non sei root ma mamma ti vuole bene lo stesso
capo@st5a11:/root/bm$ su root
Password:                                     <sc>
165~/bm ->
```

32 Il file .bashrc si trova nella home dir di ciascun utente, e realizza le sue fisime: alias, opzioni, variabili predefinite. È uno script che viene eseguito tacitamente al momento in cui si lancia *bash*. La cosa più importante è forse la variabile *PATH*. Se voglio cambiare qualcosa mentre sono già in Bash posso

- assegnare una variabile di environment

```
622~/bm -> PATH=$PATH:~/bm/aaa
623~/bm -> cd aaa
625~/bm/aaa -> echo echo roma > nuovo
626~/bm/aaa -> 755 nuovo
627~/bm/aaa -> nuovo
```

```

roma
628~/bm/aaa -> exit
exit
root@st5a11:~# bash
500~ -> cd bm/aaa
501~/bm/aaa -> nuovo
bash: nuovo: command not found
502~/bm/aaa -> echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:
                               /root:/root/bm
503~/bm/aaa -> PATH=$PATH:~/bm/aaa
504~/bm/aaa -> echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:
                               /root:/root/bm:/root/bm/aaa

```

ℓ 627 accetta di eseguire nuovo perché ho messo aaa in \$PATH

ℓ 501 quando esco da bash e rientro .bashrc rimette \$PATH al valore precedente.

- posso chiamare emacs, editare e salvare .bashrc; quindi lanciare il comando di esecuzione . .bashrc

33 Altre personalizzazioni Il mio .bashrc contiene solo

```

export PS1='\!\w -> '
export PS2==='> '
umask 022
PATH=$PATH:~/bm
alias DF='df -HT'
alias DH='dhclient'
alias 755='chmod 755'
set -o ignoreeof

```

degli alias per risparmiare qualche lettera;

la definizione della variabile \$PS1 che dà il prompt principale (\!) è il numero d'ordine del comando corrente nella *history* (avrete notato che si ricomincia sempre da 500: è a causa di un'altra variabile di environment che è inizializzata a 500 per default); segue \w in cui w sta per PWD;

segue la variabile PS2 che dà il prompt secondario che viene dato quando c'è un input da stdio provocato da un comando (→ 38); siccome Bash è un interprete, pensa che un EOF da tastiera sia la fine della sessione ed esce, spegnendo tutto e lasciandoti come un fesso: l'ultima riga disabilita l'EOF di sessione.

34 Cambiamo il prompt Sostituiamo la riga `export PS1='\!\w -> '` con

```
cd(){
command cd "$@"
PS1="\! ${PWD##*/} -> "
}
```

Essa definisce la nuova funzione (→ 140) `cd`. Le funzioni di shell hanno la priorità sui comandi builtin, ma non se il builtin è preceduto dal comando `command` (questo non è il primo bisticcio terminologico di Bash). Per esempio `cd /root/bm/aaa` è eseguito con la nuova funzione. Essa chiama il builtin `cd` dandogli priorità (non per educazione, ma per evitare un loop infinito); quello entra in `/root/bm/aaa`. L'espansione taglia la piú lunga testa del cammino completo assegnato a `$PWD` che finisca con uno slash, e quindi ritorna `aaa`.

```
512 bm -> cd /root/bm/aaa
513 aaa -> cd ~
514 root ->
```

Il neo è che abbiamo il valore ufficiale di `$HOME` invece dell'abbreviazione `~`.

35 Gli alias non sono ricorsivi

```
515 bm -> alias rm='rm -i'
516 bm -> ls aaa
517 bm -> touch aaa/bbb
518 bm -> rm aaa/bbb
rm: remove regular empty file 'aaa/bbb'?
519 bm -> ls aaa
bbb
520 bm -> rm aaa/bbb
rm: remove regular empty file 'aaa/bbb'? y
521 bm -> ls aaa
522 bm ->
```

ℓ515 definisce un alias `rm` alla maniera `windawk`, che mi chiede se voglio veramente rimuovere un file, e non lo fa se non gli rispondo `y(-es)`; l'espansione degli alias non è ricorsiva e quindi non va in paranoia quando trova `ls` all'interno della sua definizione.

2.4 Virgolette

Un programma `P` che gestisce un programma `Q` ha bisogno di distinguere tra le sue variabili (che ha tutti i diritti di assegnare) e quelle di `Q` che, in certi casi, deve lasciare in pace, e in certi altri no. Vedremo che questo secondo caso è cruciale per il teorema principale della seconda parte di questo corso. I livelli di astrazione che corrispondono a questa situazione sono garantiti molto bene dai vari tipi di virgolette (*quotes*) offerti da Bash.

```
516 bm -> cat > scrivi-tratta-come-si-merita
echo "echo $1" '$1' > tratta-come-si-merita
517 bm -> . scrivi-tratta-come-si-merita crepa
518 bm -> . tratta-come-si-merita Bill
crepa Bill
519 bm -> . scrivi-tratta-come-si-merita va al diavolo
520 bm -> . tratta-come-si-merita Bill
va Nicola
521 bm -> . scrivi-tratta-come-si-merita "va al diavolo"
522 bm -> . tratta-come-si-merita Bill
va al diavolo Bill
523 bm -> cat tratta-come-si-merita
echo va al diavolo $1
```

36 Weak vs strong quotes Le virgolette doppie sono più *deboli* delle semplici perché non *proteggono* il segno `$`

```
501~/bm -> a=5
502~/bm -> echo $a
5
503~/bm -> echo "$a"
5
504~/bm -> echo '$a'
$a
```

le virgolette deboli eseguono la vestizione, le forti no.

37 Protezione degli spazi `echo` si ricorda gli spazi inessenziali, ma li fa vedere solo se li chiedi a modo suo

```
512~/bm -> a="Roma  Bari"
513~/bm -> echo $a
Roma Bari
514~/bm -> echo "$a"
Roma  Bari
```

Bisogna fare attenzione perché gli spazi sono delimitatori forti quasi quanto il `'`;

```
520~/bm -> a=5b
521~/bm -> c=6\ d
522~/bm -> echo $a $c
5b 6 d
523~/bm -> a=5 b
bash: b: command not found
524~/bm -> a=5 b=6
525~/bm -> echo $a$b
56
```

ℓ 525 spiega perché ha protestato contro ℓ 523.

38 Here All'interno di un comando, il costrutto `<< key ... key` inserisce *qui* quanto si trova tra le due *key* come se fosse un file

```
513~/bm/aaa -> cat > nuovo << +
===> aaa          <return> (===> e' il "prompt secondario")
===> bbb
===> ccc
===> +           (considera e registra + come EOF)
514~/bm/aaa -> cat nuovo
aaa
bbb
ccc             (per chi non credeva che mettesse l'EOF)
```

la stringa tra le due *key* si chiama *here document* (nome e notazione risalgono agli inizi della rete: le chiavi separavano lo *here document*, che era il messaggio, da numeri magici, protocolli etc.); `echo` è ingordo di accapo come di spazi, ma le virgolette lo frenano

```

516~/bm/aaa -> echo $(cat nuovo)
aaa bbb ccc
517~/bm/aaa -> echo "$(cat nuovo)"
aaa
bbb
ccc

```

che è 'sto \$(cat nuovo)? È una

39 Command substitution Se uno degli arg di

com1 arg1 ... argN

è nella forma

```
$(com2 foo1 ... fooK)
```

allora Bash sostituisce quell'arg con lo stdout di com2 foo1 ...fooK. Questo è quello che ho fatto in *l* 516 prendendo cat come com2 e nuovo come foo.

40 Proteggere le variabili Ho modificato `ciaok.c` aggiungendo

```
printf("ora ho %d padron\n",--argc)
```

Vediamo che `$a` passa tre parametri, mentre ‘`$a`’ passa un solo parametro (una stringa di tre parole).

```

586~/bm -> a="sal nick vic"
587~/bm -> ciaok.bin "$a"
Buongiorno sal nick vic
Cosa comanda dopo ciaok.bin?
ora ho 1 padron
588~/bm -> ciaok.bin $a
Buongiorno sal nick vic
Cosa comandate dopo ciaok.bin?
ora ho 3 padron

```

Vediamo se riusciamo a confonderlo proteggendo una stringa vuota

```

607~/bm -> c=""
608~/bm -> ciaok.bin "$c" $a
Buongiorno sal nick vic
Cosa comandate dopo ciaok.bin?

```

```

ora ho 4 padron
609~/bm ->  ciaok.bin $c $a
Buongiorno sal nick vic
Cosa comandate dopo ciaok.bin?
ora ho 3 padron
611~/bm ->  ciaok.bin $a "$c"
Buongiorno sal nick vic
Cosa comandate dopo ciaok.bin?
ora ho 4 padron

```

ℓ611 crede che una stringa vuota protetta non sia vuota anche se è l'ultima.

41 Numero dei parametri, word splitting, parametri vuoti e inesistenti

La variabile predefinita \$# restituisce il numero dei parametri con cui è stato chiamato uno script

```

~/bm ->  cat > nuovo
echo hai usato $# parametri      # script costituito da un solo comando
~/bm ->  chmod 755 nuovo          # eseguibile
~/bm ->  nuovo u y z
hai usato 3 parametri
~/bm ->  var="queste tre parole"
~/bm ->  nuovo $var
hai usato 3 parametri
~/bm ->  nuovo "$var"             # invece le virgolette prevengono
hai usato 1 parametri            #      il word splitting su blank
~/bm ->  nuovo $u $y $z          # ma variabili inesistenti non sono arg vuoti
hai usato 0 parametri
~/bm ->  nuovo "$u $y $z"
hai usato 1 parametri
~/bm ->  nuovo "$u" "$y" "$z"
hai usato 3 parametri

```

quanto sopra mostra alcune cose

- le virgolette prevengono il word splitting di var in tre parametri distinti;
- le variabili inesistenti non sono prese come parametri vuoti;
- ma le stringhe di variabili inesistenti sono parametri.

42 Variabili strane protezione delle virgolette e IFS (→ 160 per convincersi che le prime due righe dicono che `var_strane` e' uno script di 10 righe + EOF)

```
# var_strane
IFS=' '
var="(]\{$}"
echo $var
echo "$var"
IFS='\ '
echo $var
echo "$var"
IFS='{ '
echo $var
echo "$var"
503~/bm -> var_strane
'(]\{$}
'(]\{$}
'(] {$}
'(]\{$}
'(]\ $}
'(]\{$}
```

43 Interrompere la riga di comando e stampare un apice tra altri due

```
512~/bm -> echo "queste sono<return>
===> due righe"
queste sono
due righe
513~/bm -> echo "qui invece abbiamo \
```

```
522~ -> echo d\'altra parte
d\'altra parte
```

ℓ512 se non pareggi le virgolette lui legge il newline e manda il prompt secondario
ℓ513 ma puoi proteggere l'accapo
ℓ519 il comando `cd ..` e' cortissimo, ma se ti gira puoi dividerlo su due righe proteggendo l'accapo.
ℓ520 capisce il secondo apice come fine protezione del backslash invece del backslash come protezione dell'apice, e pretende il bilanciamento mandando `$PS2`; si è placato quando l'ha visto, ma ha capito e stampato due stringhe, la prima delle quali finisce con un backslash
ℓ521 due stringhe separate da un apice protetto da `\` ma non dentro strong quotes.
ℓ522 ma naturalmente se non c'è altro da proteggere, basta proteggere l'apice

```
538~/bm -> echo -e "\v\v"      # -e abilita le protezioni protette
                                     # due TAB verticali
```

```
539~/bm -> echo  "\v\v"
          \v\v\v\v
```

```
540~/bm -> echo -e "\042"      # anche in ottale \xxx
"                                     # perche' Ascii decimale 34 = "
```

```
541~/bm -> echo $\'42'        # -e superflua nel costrutto '$X'
"
```

```
542~/bm -> echo $\'t\42\t'
"
```

```
543~/bm -> echo $\'t\x22\t'   # anche in esadecimale
"
```

```
544~/bm -> virg8=$'\42'   virg16=$'\x22'
```

```
545~/bm -> echo "$virg8 questo e' virgolettato $virg16 e questo no."
" questo e' virgolettato " e questo no.
```

```
549~/bm -> ABC=$'\102\141\163\150'
```

```
550~/bm -> echo $ABC
```

Bash

44 Ansi C escape expansion è il nome del costrutto '\$X' perché funziona per ogni valore di X che sia una delle *escape sequence* ammesse dal C Ansi, ossia

```
\n  \t  \a          etc.
```

`\nnn` Ascii ottale
`\xnnn` Ascii esadecimale

Questo è la prima dello zoo di *expansion* che ci tocca vedere nel prossimo capitolo.

2.5 Manipolazione dei parametri posizionali

45 I comandi `set` e `shift` sono i soli che possono modificare i parametri posizionali

```
560 bm -> cat > prova_set_shift
set $2 $1 Pisa
echo '$1'=$1     '$2'=$2     '$3'=$3
shift
echo '$1'=$1     '$2'=$2     '$3'=$3
561 bm -> . prova_set_shift Roma Bari
$1=Bari $2=Roma $3=Pisa
$1=Roma $2=Pisa $3=
```

in assenza di opzioni, `set` assegna i suoi argomenti `arg1 ... argK` alle variabili posizionali `1, ..., K`. Il comando `shift` le *shifta* tutte a sinistra di un posto. Entrambi sono builtin di Bash. Nell'esempio, `set` ha invertito i due parametri passati dalla riga di comando, e ne ha aggiunto un terzo, facendo credere a `echo` che `pisa` lo avevo digitato io in `l560`. La seconda riga stampata mostra che `shift` ha distrutto Bari, che è leggermente più di quanto meriti (forse).

46 Differenze nelle protezioni delle variabili `$*`, `$@`, `$#`

```
571 bm -> cat prova_set_shift
set vedo "voi due" adesso
echo ci sono in tutto $# parametri
for i in $*
do echo $i; done
for i in @$@
do echo $i; done
for i in "$*"
do echo $i; done
for i in "$@"
do echo $i; done
```

```
shift
echo adesso ci sono in tutto $# parametri
for i in "$@"
do echo $i; done
572 bm -> . prova_set_shift
ci sono in tutto 3 parametri
vedo
voi
due
adesso
vedo
voi
due
adesso
vedo voi due adesso
vedo
voi due
adesso
adesso ci sono in tutto 2 parametri
voi due
adesso
```

Capitolo 3

Espressioni

Ogni espressione ha un valore. Siccome Bash si occupa piú di stringhe che di numeri, le espressioni aritmetiche sono solo uno dei tipi di espressione. Siccome molte espressioni su stringhe alla fine sono solo abbreviazioni, il processo di assegnare un valore ad un'espressione si chiama *expansion*, invece che *valutazione*. Attenzione che *espandere*, come del resto valutare, non significa *assegnare*.

3.1 Classificazione e priorità

Per interpretare un comando, Bash effettua un parsing iniziale, e i sei tipi di espansione di seguito indicate in ordine di priorità, ma con l'avvertenza che spiegherò l'ordine di valutazione effettivo e dettagliato solo alla fine del capitolo

1. se la riga comincia con una keyword (→ 23) o contiene delle pipe **bash** si organizza per eseguire i comandi uno alla volta; quindi farà queste espansioni
2. filename brace expansion (→ 50);
3. tilde (la home directory);
4. espansione delle variabili, per ogni espressione che comincia con **\$** (→ 44, ma soprattutto → 51), con l'eccezione del prossimo punto 5;
5. sostituzione di comando (→ 39) effettuata da sinistra a destra;
6. word splitting, ossia divisione in *token*, sui valori di IFS (→ 26);
7. globbing (→ 49);

8. valutazione dei comandi `let` e `test`.

Per il momento seguo un ordine logico parzialmente diverso da quello per priorità.

47 Esempi in cui l'ordine delle espansioni è importante

```
70~/bm -> ls ~/bm/bbb
a b c
71~/bm -> a="~/bm/bbb"
72~/bm -> ls $a
ls: ~/bm/bbb: No such file or directory
```

quando deve eseguire ℓ 72, Bash non vede tildi da espandere; quando arriva alla fase 4, vede una variabile, e la espande sostituendole una stringa che comincia con un segno che a quel punto non ha significati particolari; d'altra parte la fase dell'espansione delle tildi è passata

```
75~/bm -> a="/root/bm/bbb"
76~/bm -> ls $a
a b c
```

Per un secondo esempio, anticipiamo che il comando `eval` si comporta come una *funzione universale* nel senso che per Bash il comando `eval comm par1 ... parK` e il comando `comm par1 ... parK` sono del tutto equivalenti ($K \geq 0$); osserviamo anche che assegnare un comando ad una variabile è perfettamente legittimo

```
509 bm -> eval eval eval echo echo
echo
510 bm -> a="echo echo"
511 bm -> $a
echo
```

Ma proviamo a mettere una pipe nel valore di una variabile

```
512 bm -> ls | wc
    191    191    2829
513 bm -> a="ls | wc"
514 bm -> $a
ls: |: No such file or directory
ls: wc: No such file or directory
515 bm -> eval $a
    191    191    2829
```

perché quando tratta il comando `$a` nella fase 1 non vede il segno `|`; quando arriva alla fase 4 espande a `ls | wc`, non vede altre espansioni ed esegue il comando `ls | wc`; e ci crede capaci di chiedere l'elenco dei file che stanno nella dir `|` e nella dir `wc`.

48 I comandi di espansione `test` (\rightarrow 72) e `let` (\rightarrow 60) vanno secondo me considerati come forme di espansione, perché sostituiscono, senza la mediazione di una sostituzione di comando, all'interno di un comando i loro valori in punti in cui uno si aspetterebbe di trovare risp. la risposta ad una domanda, e un numero. La loro esecuzione o espansione conclude il processo di espansione dei comandi. Date l'importanza e la frequenza di queste operazioni, i due comandi

- vengono eseguiti direttamente, senza bisogno del costrutto `$(...)` di sostituzione di comando;
- sono dei *builtin*, e non richiedono quindi il lancio di un nuovo sotto-Bash. Peraltro `test`, ma non `let`, è anche un comando Unix, che però sappiamo avere minore priorità del builtin.

3.2 Filename expansion

I file sono così importanti che una delle prime cose da sapere è il modo di espandere le variabili assegnate con un filepath. Abbiamo due modi

49 Globbing Unix si è inventato questo idiotismo per indicare una sua forma di *pattern recognition* che non segue completamente il metodo delle espressioni regolari (globbing, da glob = globo, significa qualcosa come appallottolare fango, pongo, etc.). Serve per plasmare e selezionare nomi che corrispondono ad un *pattern*. Si usano le *wildcard* `*` e `?`, che sono *meta*-simboli, considerati equivalenti a diversi valori (come il *jolly* in certi giochi). Un pattern `pt` definisce una lista di stringhe `x`, ordinata secondo Ascii. La lista è infinita se in `pt` c'è almeno un `*`. Per questo ho detto che `pt` definisce, e non che genera o produce la lista degli `x`. La lista è ottenuta mediante le equivalenze (*match*) seguenti (nei filepath ci dovrebbero essere solo lettere ordinarie, cifre decimali, i simboli `.`, `-`, `/` e i simboli protetti come il blank `\` `'`, contati come una sola lettera)

- ogni lettera ordinaria di `pt` è equivalente alla stessa lettera in ogni `x`;
- ogni `?` è equivalente in `x` ad una lettera che può occorrere in un filename (comprese `.` e `/`);

- * è equivalente ad una stringa qualsiasi, anche vuota;
- con le parentesi quadre si possono abbreviare alcune cose
 - [bar] è equivalente a una delle lettere b a r;
 - [!bar] è equivalente a una lettera diversa da quelle lí;
 - [x-y] è equivalente ad un simbolo tra (in ordine Ascii) x e y.

```

503~/bm/aaa -> ls
a.bin a.c b.bin b.c d.c
504~/bm/aaa -> rm *c
505~/bm/aaa -> ls
a.bin b.bin
506~/bm/aaa -> rm [ade]*
507~/bm/aaa -> ls
b.bin
508~/bm/aaa -> touch foo.bar
509~/bm/aaa -> ls
b.bin foo.bar
510~/bm/aaa -> rm *
511~/bm/aaa -> ls
512~/bm/aaa ->

```

50 Brace Expansion Il globbing aiuta a *trovare* definendo una lista per poi cercare una corrispondenza tra valori dati ed elementi della lista. L'espansione delle graffe invece *genera* una lista. La forma è (ricorsivamente, ma senza annidamenti)

u {x..y} w

```

569~/bm -> cd aaa
570~/bm/aaa -> rm *
571~/bm/aaa -> touch file{a..b}{1..3}
572~/bm/aaa -> ls
filea1 filea2 filea3 fileb1 fileb2 fileb3
573~/bm/aaa -> rm *a?;ls
fileb1 fileb2 fileb3

```

Il buon senso dice che il globbing può definire insiemi infiniti, la brace expansion no.

3.3 Espansione delle variabili

51 Nel segno del dollaro Come abbiamo detto all'inizio del capitolo, il segno `$` comincia spesso l'espansione di una variabile. L'operatore di assegnazione (da `var` a `$var`) può essere visto come il caso più semplice di espansione (espansione identica). È essenziale notare che l'espansione, non essendo un'assegnazione, può dare un risultato che non è un comando, e quindi può accadere che, da sola, dia errore (`ℓ537`) e anche no

```
534 bm -> a=echo b=" " c=5 d=$a$b$c
536 bm -> $d
5
537 bm -> $c
bash: 5: command not found
538 bm -> a="echo a"
539 bm -> $a
a
```

Abbiamo visto la C escape expansion `$'...'`; si ha una sintassi simile per la *locale expansion* `$"..."` che corrisponde ai `locale` del C. Le espansioni introdotte in questa sezione (ce ne sono molte altre) seguono una delle sintassi

$\{\text{var } X \text{ stringa}\}$ $X = -, :=, :?, +$ (A)

$\{\text{var } X \text{ ptrn}\}$ $X = \#, \#\#, \%, \%\%$ (B)

$\{\text{var } X \text{ ptrn/stringa}\}$ $X = /, //$ (C)

le espansioni (A) gestiscono in vari modi (assegnazione per default, messaggi, interruzione, test semplice) il caso in cui la variabile `var` non sia definita. Le (B) tagliano la più corta/lunga testa/coda di `var` che soddisfi il pattern `ptrn`. Le ultime sostituiscono la prima/ogni sottostringa che soddisfi `ptrn`.


```

612~/bm -> cat > nuovo
echo il valore di a est ${a:? inesistente}; echo ok
613~/bm -> 755 nuovo
614~/bm -> a=bari
615~/bm -> nuovo
/root/bm/nuovo: line 1: a: inesistente
616~/bm -> printf "%s\n" ${!a*}
a
617~/bm -> cat > nuovo
a=roma;echo il valore di a est ${a:? inesistente}; echo ok
618~/bm -> nuovo
il valore di a est roma
ok
619~/bm -> export a=napoli
620~/bm -> cat > nuovo
echo il valore di a est ${a:? inesistente}; echo ok
621~/bm -> nuovo
il valore di a est napoli
ok

```

ℓ612-14 definisco lo script `nuovo` che dovrebbe verificare se `a` è stata definita, definisco `a` e lo lancio

ℓ615 ma lui aggiunge di suo un messaggio nella forma `scriptname : lineno : varname` e scrive il mio messaggio. Perché? ℓ616 mi conferma che `a` esiste. Ma esiste in questo processo `bash`, non nel suo sotto-processo prodotto dal `fork` allo script `nuovo` (→ ??).

ℓ617 riscrivo `nuovo` definendo `ll` la variabile e ora `nuovo` è eseguito fino in fondo e vediamo l'ok.

ℓ619 ma sarebbe bastato dichiarare `export a=...`

Insisto sul fatto che se la variabile non è definita, esce senza terminare lo script, e quindi, in questo caso, senza scrivere ok.

54 stdout oppure stderr? Prima me la sono cavata con un “manda sul monitor”, ma voi volete sapere se come errore, oppure no.

```
622~/bm -> unset a
623~/bm -> nuovo
/root/bm/nuovo: line 1: a: inesistente
624~/bm -> nuovo 2> errore
625~/bm -> cat errore
/root/bm/nuovo: line 1: a: inesistente
```

facciamo fuori a e `ℓ624` ridirige lo stderr sul file `errore`, perché la forma generale della ridirezione è

```
comm_output filedescriptor > filename
```

dove `comm_output` è un comando in una sola riga che dà un output, e `filedescriptor` è per default 1, ma può essere un altro numero di file aperto.

55 Testare se definito Il costrutto `${var+stringa}` restituisce `stringa` se `var` è definita e la stringa vuota altrimenti

```
504~/bm -> printf "%s\n" ${!b*}
b1
505~/bm -> echo ${b1+1}
1
506~/bm -> echo ${b+1}

507~/bm ->
```

56 Riassumendo Se `var` non è definita, puoi

- avere un valore di default con `:-`
- assegnarle un valore di default con `:=`
- intercettare ed uscire con `?:`
- testare senza influenzarla con `+`

```

$ unset a b c d e; a=10 e=20
$ b=${a:-no}; c=${a:=no}; echo a=$a b=$b c=$c; d=${e?:no}; echo d=$d
a=10 b=10 c=10
d=20
$ unset a b c d e
$ b=${a:-no}; c=${a:=no}; echo a=$a b=$b c=$c; d=${e?:no}; echo d=$d
a=no b=no c=no
bash: e: =no
$

```

3.3.2 Modifica di stringhe

L'abbondanza ed eterogeneità delle cose che si possono fare è dovuta al fatto che non si è mai affrontato il metodo delle *regex* (espressioni regolari) e ci si è limitati a dare aria, aggiungendo oggi una finestra, ieri una porta, e domani un abbaino.

57 Sostituzioni Per ogni pattern `pt` (\rightarrow 49) `${varXpt/stringa}` definisce l'espansione di `var` ottenuta sostituendo `stringa`

- se `X=//` a tutti i match di `pt`
- se `X=/` al primo match di `pt`

```

499~/bm -> a=barbari
500~/bm -> echo ${a/ba/}
rbari
501~/bm -> echo ${a//ba/}
rri
502~/bm -> echo ${a//[ia]/e}
berbere

```

58 Tagliare teste e code `${var X pt}` definisce l'espansione di `var` ottenuta tagliando il match di `pt`

- se `X=#` piú corto all'inizio di `var`
- se `X=##` piú lungo all'inizio di `var`
- se `X=%` piú corto alla fine di `var`

- se `X=%%` piú lungo alla fine di `var`

(mnemonica: `#` cerca il match all'inizio e in Ascii viene prima di `%` che cerca il match alla fine; simbolo solo = match corto, simbolo doppio = match lungo)

```
517~/bm -> a=/usr/src/linux-source-2.6.18/linux-source-2.6.18/kernel/irq
518~/bm -> echo ${a#*/}
src/linux-source-2.6.18/linux-source-2.6.18/kernel/irq
519~/bm -> echo ${a##*/}
irq
520~/bm -> echo ${a%/linux-source-2.6.18/*}
/usr/src/linux-source-2.6.18
521~/bm -> echo ${a%%/linux-source-2.6.18/*}
/usr/src
```

59 Gestione del nastro di una TM

```
(tapeparts)=
tape=$1
left=${tape%?X*}
right=${tape##*X??}
echo tape=$tape left=$left right=$right
core1=${tape/$right/}
core=${core1/$left/}
echo $core
```

```
563 bm -> tapeparts 110001X00011
tape=110001X00011 left=11000 right=011
core=1X00
```

3.4 Aritmetica

L'aritmetica di Bash è solo su numeri interi relativi (quindi 5.7 è una stringa, non un float). C'è il modo di rappresentare i numeri in altre basi, oltre 8, 10 e 16.

60 let Il builtin

```
let "var = expr"
```

assegna a `var` il valore dell'espressione `expr`. Le espressioni sono come in C, con le stesse priorità, gli stessi operatori n -ari ($n \leq 3$), compresi quelli *bitwise*, compresi quelli logici (applicati allora a `==`, `!=`, etc.), compresi i *self-referential* `+=`, `*=`, etc., con le stessissime priorità. C'è anche l'operatore ternario `?`. In più c'è la potenza `**`. Ma C accetterebbe quanto segue?

```
554~/bm -> declare -i a=5
555~/bm -> a=a+++++a
556~/bm -> echo $a
12
```

Se la risposta è no, la domanda successiva è perché.

61 Esportare le variabili

```
428~/bm -> declare -i a=5                # int a
429~/bm -> declare -r b=6                # read-only
430~/bm -> declare -x c=7                # export
431~/bm -> declare -a nm=(sal nick vic) # array
432~/bm -> echo $a ${nm[1]} $b $c       # variabili vestite
5 nick 6 7
433~/bm -> bash                          # nuovo processo bash
426~/bm -> echo $a ${nm[1]} $b $c
7                                         # conosce solo le variabili esportate
428~/bm -> let "c += 1"
429~/bm -> echo $c
8
427~/bm -> exit                          # ritorno al processo padre
exit
434~/bm -> echo $a ${nm[1]} $b $c # ritrovo le mie variabili
5 nick 6 7                                # ma il figlio non comunica c al padre
440~/bm -> let "b += 1" # si rifiuta di cambiare le costanti
bash: b: readonly variable
441~/bm -> let "a += 1"
442~/bm -> echo $a
6
```

Il problema della comunicazione dal processo figlio al padre è generale in Linux: il figlio riceve tutte le variabili del padre, ma può mandare info al padre solo per il tramite di un file.

62 Le variabili scalari sono tipate o no? Non c'è bisogno di casting, perché interi e stringhe non sono segregati. In aritmetica Bash tratta le stringhe non numeriche come 0, senza avvisare (questa storia l'ho vista e rivista, ma non mi risulta documentata e non so né se lo fa sempre né se sia portabile). Dichiarare una variabile come intera è un pò piú efficiente perché si risparmia il tempo di andata e ritorno da Ascii ad intero (→ 60 per `let`; → 57 per `${a/12/BB}`)

```
218~/bm -> a=1234
219~/bm -> let "a += 1"
220~/bm -> echo $a
1235
221~/bm -> b=${a/12/BB}
223~/bm -> echo $b
BB35
224~/bm -> let "b += 1"
225~/bm -> echo $b
1
226~/bm -> c=BB34
227~/bm -> d=${c/BB/12}
228~/bm -> let "d += 1"
229~/bm -> echo $d
1235
```

ℓ 219 aritmetica su stringhe numeriche non dichiarate tali; non c'è bisogno di vestire le variabili dentro `let`

ℓ 224 tratta la stringa alfanumerica `b` come 0;

ℓ 228 si accorge che la sostituzione ha trasformato un'alfanumerico in numero

63 In realtà `let a=expr` fa due cose (1) valuta `expr`; *ed inoltre* (2) assegna `var` col valore che si è calcolato. Sembrerebbe insensato permettere un costrutto come `let expr` invece di `let var=expr`. `let` calcola il valore di `expr` e lo butta. Ma il processo di valutazione può avere dei side effect che ci interessano

```
558~/bm -> b=5
559~/bm -> let "a=++b"
560~/bm -> echo $b
6
```

64 Espansione aritmetica Il costrutto `$(expr)` si espande nel valore assegnato da `let a expr`

```
587~/bm -> a=0
588~/bm -> echo $((a?10:20))
20
```

Se poi c'è anche un'assegnazione, tanto meglio; vorrà dire che di cose ne facciamo due (un'espansione *ed anche* un'assegnazione)

```
544~/bm -> a=5 c=100
545~/bm -> echo $((c=a?10:20)); echo $a $c
10 5 10
547~/bm -> echo $((c==a?10:20)); echo $a $c
20 5 10
```

con le solite trappole sceme del C.

Possiamo dire che il costrutto `$((...))` è una *abbreviazione* per `let` Le doppie parentesi possono stare da sole

```
516 bm -> c=5
517 bm -> ((c+=c))
518 bm -> echo $c
10
519 bm -> $((c+=c))
bash: 20: command not found
521 bm -> let "c+=c"
522 bm -> echo $c
40
```

ma senza il `$` perché se no quello espande, passa alla riga di controllo e cerca di eseguire 20.

65 Comma operator valuta ed assegna tutto, ma restituisce l'ultimo

```
519~/bm -> a=$((b=5,c+=b+7,c--,b+c++))
520~/bm -> echo $a $b $c
16 5 12
```

66 La virgola mobile si può fare meglio che in C col linguaggio bc

```
$ cat>nuovo
r=$1 s=$2 t=$3
r=$(echo "scale=$t; $r/$s" | bc)
echo $r
$ nuovo 100 7 0
14
$ nuovo 100 7 4
14.2857
$ nuovo 1997 1999 50
.99899949974987493746873436718359179589794897448724
```

67 Rappresentazioni in base b nella forma b#nnn (come al solito per b=8,16)

```
$ oct17=021 esa17=0x11 ter17=3#122 bin17=2#10001 quin17=5#12
$echo $((oct17+esa17+ter17+bin17+quin17)) $oct17 $esa17 \
==> $ter17 $bin17 $quin17
75 021 0x11 3#122 2#10001 5#12
```

registrati come in input, e convertiti solo per fare aritmetica.

3.5 L'exit status e i comandi return e source

Nell'esecuzione di un algoritmo le decisioni che contano dipendono da proprietà e relazioni numeriche. Invece, in un linguaggio di sistema come Bash, quello che interessa è sapere se i processi sono giunti a termine con successo, o se ci sono state difficoltà, e quali. La questione posta nel §68 è essenziale perché sappiamo che in in Unix i processi di sistema sono, in generale, esecuzioni di funzioni C.

68 Ma che fine fa in C il valore di return della funzione main? In Bash è assegnato (mod 256) a ?, che è una variabile predefinita che si chiama *exit status*

```

514~/bm -> cat > return_where.c
int main(){return 300;}
515~/bm -> gcc -o return_where return_where.c
516~/bm -> return_where
517~/bm -> echo $?
44

```

se `return` manca, `$?` è imprevedibile; se invece il valore è illegittimo, si ha `$?=3`.

```

503~/bm -> cat > return_nil.c
main(){
504~/bm -> gcc -o return_nil return_nil.c; return_nil; echo $?
68
505~/bm -> return_nil; echo $?
36
506~/bm -> cat > return_pi.c
int main(){return 3.14;}
507~/bm -> gcc -o return_pi return_pi.c; return_pi; echo $?
3
508~/bm -> return_pi; echo $?
3

```

69 Exit status In Unix ogni comando assegna *exit status* con un numero ≤ 255 ; per convenzione 0 significa che le cose sono andate bene; siccome le difficoltà possono essere tante, gli altri numeri sono riservati, per intervalli convenzionali ¹, a Linux, Bash, C, e alle esigenze dell'utente; si hanno eccezioni a questa regola quando le cose possono andare bene ma con esiti diversi (per esempio il comando `grep` dà 0, 1, 2 se non si ha un *regex match*, se lo si ha, se c'è un errore). In C, `return z` pone `$?=z`. Negli script, `?` è assegnata (automaticamente) con lo status dell'ultimo comando eseguito (che poi è la stessa cosa perché è spesso un programma C)

¹pia illusione: lo standard esiste ma nessuno lo rispetta, non so perché.

```

517~/bm -> ls aaa
518~/bm -> echo $?          # dir vuota
0
519~/bm -> ls bbb
a b c
520~/bm -> echo $?
0
521~/bm -> ls xyz
ls: xyz: No such file or directory
522~/bm -> echo $?
2
523~/bm -> 5
bash: 5: command not found
524~/bm -> echo $?
127
525~/bm -> echo $?
0

```

525 perché ovviamente ? dà lo statuto dell'*ultimo* comando.

Ci sono comandi che non creano file e non mandano un'uscita a stdout. Se la cavano assegnando uno status *n* a ?. In questo caso, si dice che *restituiscono lo stato n*. Il più importante è *test*.

70 E se uno script vuole ritornare uno stato? Che *myscript* finisca col comando `return n` non ha molto senso, visto che l'exit status di uno script è lo status del suo ultimo comando, e allora quell'*n* sarebbe l'exit status dello stesso `return n`. La logica complessiva del sistema è che Bash, quando arriva al comando *foo*, lancia un nuovo processo *bash* che interpreta *foo*. Quando *foo* è stato eseguito, si torna al processo esterno con l'exit status (dell'ultimo comando) di *foo*. Ci sono due casi in cui *bash* non lancia un nuovo processo *bash*

- quando *foo* esegue una funzione *funct*; vedremo che una funzione Bash è, come in tutti i linguaggi, un pezzo di codice considerato, per qualche ragione, come un tutt'uno isolato. In questo caso è ragionevole che, come *foo* dà un exit status a *bash*, così *funct* ritorni un suo status a *foo* dopo che l'ha chiamata. Per questo, è previsto che, all'interno di una funzione Bash, ci possa essere un `return n`.

Insisto: quando mi scrivo la mia funzione, decido io, sulla base della logica di quella funzione, e di quello che è successo durante la sua esecuzione, quale

status si abbia alla fine. Quando eseguo un comando eseguito da qualcun altro, è lui che decide lo status e me lo passa.

- il comando `source comm [parms]` in una certa riga n provoca
 - la sostituzione (logica, non materiale) della riga n con l'intero file `comm`
 - l'assegnazione di `parms` a `$1*`
 - la restituzione del controllo a `bash`.

`bash` continua come se in n avesse trovato `comm parms`, ed interpreta quindi i comandi di `comm` con i parametri posizionali assegnati in quel modo. Essendo un builtin (ossia una delle funzioni di Bash, non un comando Linux) `source` non causa il lancio di un nuovo processo `bash`. Se anche `command` è fatto di funzioni e builtin si migliora l'efficienza. Un *sourced command* può contenere un `return`, per ragioni analoghe a quelle dette a proposito delle funzioni.

```
527~/bm -> cat > nuovo
return 50
528~/bm -> 755 nuovo
529~/bm -> nuovo
/root/bm/nuovo: line 1: return: can only 'return' from a
                funzione or sourced script [a capo mio]
530~/bm -> source nuovo
531~/bm -> echo $?
50
```

3.6 Quid est veritas?

71 Il comando if

```
519~/bm -> if 0; then echo zero; else echo non zero; fi 2>errore
non zero
```

E perché? Per tre ragioni:

- 1 perché, l'ho già detto, per Bash l'exit status conta più dei valori di verità
- 2 e quindi la sintassi è: (`comJ` è un comando con eventuali opzioni e argomenti; il ramo `else com3`; può essere assente)

```
if com1; then com2; [else com3;] fi
```

e 1 non è un comando;

3 e quindi la semantica è: se `com1` ha avuto successo, ossia se al termine di `com1` si ha `$?=0`, allora `com2` [altrimenti `com3`];].

Non avendosi un successo, si esegue il ramo `else`.

Ed effettivamente abbiamo

```
515~/bm -> 1
bash: /root/bm/1: Permission denied
516~/bm -> echo $?
126
523~/bm -> 1 2>errore
524~/bm -> cat errore
bash: /root/bm/1: Permission denied
```

dove `ℓ523` spiega e `ℓ524` conferma (a chi sappia o ricordi che 2 è il `file descriptor` di `stderr`) perché in `ℓ519` non si è visto il messaggio di errore.

Vediamo meglio

```

602~/bm -> if echo 1; then echo ok; fi
1
ok
603~/bm -> if $(echo 1); then echo ok; fi
bash: /root/bm/1: Permission denied
604~/bm -> if $(echo echo echo); then echo zero; fi 2>errore
echo
zero
605~/bm -> if $(ls zanzibar); then echo 0; else echo de che?; fi 2>errore
de che?
606~/bm -> if (($echo 1)); then echo ok; fi
ok
607~/bm -> if (($echo 0)); then echo ok; else echo no; fi
no
608~/bm -> if eco a; then echo ok; else echo no; fi 2> /dev/null
no
609~/bm -> echo $?
0
610~/bm -> if eco a; then echo ok; else eco no; fi 2> /dev/null
611~/bm -> echo $?
127

```

ℓ602 decide sull'exit status

ℓ603 la sostituzione di comando espande allo stdout del comando stesso, che, in questo caso, non è un comando perché, come già detto, 1 non lo è; manca il ramo **else**, e nulla s'ha da far

ℓ604 qui la sostituzione di comando si è espansa in un comando; il comando viene eseguito con successo (prima riga di output), e si entra nel ramo **then**

ℓ605 non dà nomi turistici ai file nella dir per la didattica

ℓ606/7 ci vuole **let** per trasformare (via doppia parentesi) 1/0 in un comando; e va bene, ma perché adesso ((1)) è un successo e ((0)) no? Un momento (→ 72, 73).

ℓ608/9 si vanta per il successo dell'ultima cosa fatta; ma qual'è ? la deviazione dei rifiuti nella pattumiera universale /dev/null, oppure **echo no**?

ℓ610/11 le ridirezioni non sono comandi e non danno exit status

72 Il comando test restituisce uno stato La sintassi è

```
test expr                                (expr=par1 ... parK)
```

dove

- `test` può essere sostituito da `[` (è una keyword, e quindi deve essere seguita da uno spazio); secondo logica non ci sarebbe bisogno della chiusa; secondo buon senso sí, e, per una volta ha prevalso il buon senso, e quindi l'abbreviazione è lecita purché si aggiunga `parK+1=]`
- versioni recenti di Bash introducono il costrutto `[[...]]` (\rightarrow 76);
- `expr` è atomica o composta con parentesi e/o connettivi `!`, `-a`, `-o`

Parsing e valori come segue

- $K = 0$: sempre 1
- $K = 1$: 0 sse `par1` non è vuoto;
- $K \geq 2$ ed inoltre `par1=!`: l'opposto di `test par2 ... parK`
- $K = 2$ ($K = 3$) e `parK-1` è un operatore unario (binario): 0 se è vero dell'altro (degli altri); per $K = 3$, i connettivi `-a`, `-o` (*and*, *or*) contano come operatori;
- $K = 3$, `par1 = (e par3 =)`: come `test par2`
- tutti gli altri casi: secondo le regole precedenti.

Gli operatori condizionali unari e binari sono pure troppi; tra gli altri

- unari su file: `-e`, `-f`, `-d` (esiste, è regolare, è una dir)
- binari su file `-n`, `-o` (newer, older)
- su stringhe: `-n`, `-z` (non è vuota, è vuota), `=`, `!=`, `<`, `>` (secondo Ascii)
- su interi: `-eq`, `-ne`, `-lt`, `-le`, `gt`, `-ge` (equal, not equal, lower than, lower or equal, grater than, greater or equal)

Ora un pò di esempi per aiutare, ma siccome ci si sbaglia alla grande (autori e lettori) meglio provare prima in modo interattivo, che spulciare dopo uno script erratico.

```

556~/bm -> [ ]
557~/bm -> echo $?
1
558~/bm -> [ 1 ]; echo $?
0
560~/bm -> [ 0 ]; echo $?
0
563~/bm -> if [ ] ; then echo vero; else echo falso; fi
falso
565~/bm -> a=roma b=""
566~/bm -> if [ $a ] ; then echo vero; else echo falso; fi
bash: [a: command not found
568~/bm -> if [ $a ] ; then echo vero; else echo falso; fi
vero
569~/bm -> if [ $b ] ; then echo vero; else echo falso; fi
falso
570~/bm ->

```

nel prossimo esempio vediamo che se `a` è vuota il parse di `test $a = ...` viene fatto su `test = ...` e lui protesta perché non puoi mettere un operatore binario al primo posto. Il trucco è di premunirsi con un prefisso non significativo

```

$ a=
$ if [ $a = xyz ]; then echo y; else echo n; fi
bash: [: =: unary operator expected
n
$ if [ X$a = Xxyz ]; then echo y; else echo n; fi
n

```

73 Invece con il comando `let` e anche con `((...))` ottenere un valore diverso da zero è un successo. Questo comportamento opposto è stato il modo di mettere d'accordo (dopo polemiche cinquantennali) hardwaristi e programmatori di algoritmi ($0=falso$) con logici e programmatori di sistema ($0=successo$): è un successo una *valutazione* (non un comando) che dà *vero* nel senso del C, ossia un non zero

```

571~/bm -> (( a=7-5)); echo $?
0
572~/bm -> (( a=7-7)); echo $?
1

```

```

573~/bm -> (( a=5-7)); echo $?
0
573~/bm -> a=0 b=1; ((a=b)); echo -e "successo=${?}\ta=${a}"
successo=0      a=1
574~/bm -> a=0 b=0; ((a=b)); echo -e "successo=${?}\ta=${a}"
successo=1      a=0

```

e naturalmente anche le relazioni numeriche come in C, con ==, != etc.

```

$ a=0 b=1;((a==0));echo -n $?;((a!=0));echo -n $?;((a==b));echo $?
011

```

Con `test` si possono fare confronti di vario tipo (numeri, stringhe, timestamp dei file, etc.) Il discorso non finirebbe piú. Le relazioni numeriche vogliono le opzioni `-eq`, `-ne`, `-ge`, `lt` etc. (equal, not equal, greater or equal, lower than) perché i segni =, < etc. sono riservati ad altri confronti, e le variabili devono essere vestite (se no si confonderebbero con stringhe costanti)

```

$ [ $a -eq 0 ];echo -n $?;[ $a -eq 0 ];echo -n $?;[ $a -eq $b ];echo $?
011
$ [ a -gt 0 ];echo -n $?;[a -lt 0 ];echo -n $?;[ $a -le $b ];echo $?
bash: [: a: integer expression expected
210

```

Restituisce lo stato 0 se la risposta alla domanda è sí; restituisce lo stato 1 se la risposta è no. Restituisce 2 (e forse anche altri valori, in casi che non conosco) se la domanda è mal posta. Infine, senza opzioni e con un solo argomento, risponde sí sse esso è una stringa non vuota. Le domande sono tante: su file (esiste, è una dir, che diritti c'ha, [`foo -nt bar`] = il file `foo` è piú nuovo del file `bar`, etc.), su stringhe (=, !=, <), su numeri (`-eq`, `-ne`, `-le`, etc.), su carattere (come `isdigit` etc. in C ma con un'altra sintassi)

```

607~/bm -> cat new
if [ -f $1 ]; then printf "il file %s esiste\n" $1;
else return 200; fi
608~/bm -> new new
il file new esiste
609~/bm -> new z37
/root/bm/new: line 2: return: can only 'return' from a funzione or
sourced script [accapo mio]

```

```
610~/bm -> echo $?
1
611~/bm -> . new z37
612~/bm -> echo $?
200
```

ℓ609 ci siamo dimenticati che (→ 69) uno script che non è una funzione non ha il diritto di ritornare uno status

```
74 577~/bm -> if [ 2 -ne 0 ] ; then echo diverso; fi
diverso
578~/bm -> ls xyz
ls: xyz: No such file or directory
579~/bm -> if [ $? -ne 0 ] ; then echo comando fallito; fi
comando fallito
```

ℓ577 fra poche righe vedremo che `-ne` è come `!=` in C, e che le quadre racchiudono una domanda

ℓ579

Siamo riusciti a farci contraddire solo: dandogli una stringa vuota, e omettendo uno spazio

75 Le doppie tonde non piantano grane sugli spazi

```
550~/bm -> ((a+=5))
551~/bm -> echo $a
15
```

76 Il costrutto `[[...]]` si comporta come `[...]`, ma in certi casi è piú affidabile

```
500~/bm -> x=''
501~/bm -> [ $x = roma ]
bash: [: =: unary operator expected
502~/bm -> [[ $x = roma ]]
503~/bm -> echo $?
1
```

ℓ501 dopo aver espanso `$x` in una stringa vuota ha cercato di eseguire `[= roma]`.

3.7 Se le variabili sono troppe e troppo lunghe

77 L'istruzione s dell'editor in linea sed L'editor in linea `sed` modifica e manda in `stdout` le righe che riceve in `stdin`. L'istruzione `s/ptrn/repl` esegue in ogni riga il rimpiazzamento `repl` là dove detto dal pattern `ptrn`; si usano i metasimboli

- `.` per un simbolo qualsiasi
- `*` per un qualsiasi numero di ripetizioni di quanto a sinistra
- `&` per la riga in entrata così com'è

```
536~/bm -> sed 's/./riga\ &/'
```

```
1  
riga 1  
2  
riga 2
```

```
542~/bm -> sed 's/ba*/1/'
```

```
bari  
1ri  
babari  
1bari
```

```
543~/bm -> sed 's/./1/'
```

```
bari  
1  
babari  
1
```

ricordando che `sed` lavora per righe e che `echo -e` interpreta i newline quotati

```
550~/bm -> echo -e "a\nb\nc\n" | sed 's/.a*/riga\ &/'
```

```
riga a  
riga b  
riga c
```

78 Il comando seq n manda in `stdout` `1 ... n`, uno per riga

```
524~/bm -> seq 3
```

```
1  
2
```

```

3
525~/bm -> echo $(seq 3)
1 2 3
535~/bm -> echo $(seq 3 | sed 's/./riga\ &/')
riga 1 riga 2 riga 3

```

79 Troppa roba

```

569~/bm -> eval $(seq 10000 | sed 's/./var&=zanzibar&/')
570~/bm -> echo $var9000
zanzibar9000

```

fin qui va bene, ma

```

576~/bm -> 503~/bm -> eval "$(seq 5000 | sed 's/./export var&=zanzibar/')"
577~/bm ->507~/bm -> ls aaa
a b c
578~/bm -> eval "$(seq 5000 | sed 's/./export var&=zanzibar_dar_es_salaam/')"
579~/bm -> ls aaa
bash: /bin/ls: Lista degli argomenti troppo lunga
580 exit
exit
root@st5a11:~# bash
500~ -> cd bm
501~/bm ->507~/bm -> ls aaa
a b c

```

ℓ579 ha ingoiato 9000 variabili di lunghezza 13 ma non ne ingoia 5000 di lunghezza 23

ℓ580 si blocca al punto che bisogna uscire da `bash` e rientrare (notate che la history ricomincia da 500).

3.8 L'interpretazione della riga di comando

80 L'algoritmo Data la riga `X` esegui (rispettando i `go to`)

- 1) dividi `X` in *token* separati dall'elenco prefissato di metasimboli

```
blank  tab  newline  ;  <  >  |  &  (  )
```

- se X comincia con virgolette semplici (protezione forte) `go to 11`;
- se X comincia con virgolette doppie `go to 5, 6, 7, 8`; dopo 8 `go to 11`;
- 2) se X comincia con keyword (\rightarrow 23) iniziale, non protetta da " o \, , abbiamo un comando composto; separa i comandi, preparane l'esecuzione individuale; `go to 1`; se keyword intermedia (come `then`) o finale (come `fi`) segnala l'errore;
 - 3) se il primo token è nella lista degli alias, espandilo; `go to 1`;
ma, se è un alias già espanso, non lo riespandere (evitando ricorsioni infinite)
 - 4) espansione delle tildi (nei casi detti nel §31)
 - 5) espansioni delle variabili
 - 6) sostituzioni di comando
 - 7) valutazioni aritmetiche (`let` e doppia parentesi)
 - 8) ripete la partizione in token, ma questa volta con i valori di `IFS`; con la differenza che due separatori bianchi consecutivi ne valgono uno, ma due consecutivi non bianchi fanno una stringa vuota (ossia se usi il `:` per separare due campi, poi `a:b::c:d` dà quattro campi, il terzo dei quali è vuoto)
 - 9) filename expansion mediante wildcard
 - 10) ora cerca il comando da eseguire in corrispondenza del primo token, con le priorità già accennate nel §3, ossia
 - built-in speciali
 - funzioni
 - altri built-in
 - comandi rinvenibili nelle `dir` in `PATH`
 - 11) esecuzione di quanto così ottenuto, previo aggiustamento degli I/O secondo ridirezioni e pipeline ed altro

81 Esempio Simuliamo quanto sopra applicato al comando `ℓ542`

```

537bm -> mkdir /aaa
538bm -> cd /aaa
539aaa -> touch f1 f2
540aaa f=f y="a a"
542aaa echo ~+/${f}[12] $y $(echo cmd subst) $((3 + 2)) > out
cat out
/aaa/f1 /aaa/f2 a b cmd subst5
rm -r /aaa

```

- 1) La ridirezione viene vista e conservata per dopo. I 4 blank definiscono 5 token
- 2), 3) `echo` non è una keyword e non è un alias
- 4) espansione delle tildi. `+` è un'abbreviazione per `PWD` che non avevo avuto occasione di dirvi, e si espande il token 2

```
echo /aaa/${f}[12] $y $(echo cmd subst) $((3 + 2))
```

- 5) espansione delle variabili `f` (rimane così, che male c'è) e `y`

```
echo /aaa/f[12] a a $(echo cmd subst) $((3 + 2))
```

- 6) sostituzione di comando (qui non la facciamo lunga, ma poteva richiedere di ricominciare ricorsivamente da 1)

```
echo /aaa/f[12] a a cmd subst $((3 + 2))
```

- 7) valutazioni aritmetiche

```
echo /aaa/f[12] a a cmd subst 5
```

- 8) riconta i token e si accorge che sono 7

```
echo /aaa/f[12] b c b cmd subst 5
```

- 9) wildcard sul token 2

```
echo /aaa/f1 /aaa/f2 b c b cmd subst 5
```

- 10) vede che `echo` è un builtin non speciale
(**Esercizio** verificarlo)

- 11) esegue, tenendo conto della ridirezione stabilita al passo 1

Capitolo 4

Due forme del teorema della ricorsione

4.1 Un programma C che si auto-riproduce

Quando non c'erano i videogiochi, c'era chi trovava divertente programmare. C'era pure chi si sfidava a scrivere il piú corto programma Fortran che desse un'uscita *esattamente* uguale al suo sorgente. Lo chiamavano un *quine* perché un modo per farlo si basava sulla variante di Quine¹ del Mentitore

```
"la frase virgolettata e' falsa se seguita da se stessa" la frase
virgolettata e' falsa se seguita da se stessa
```

Lui voleva chiarire certi aspetti del teorema di Gödel. Ci interessa perché aiuta a chiarire tre fenomeni informatici: cavalli di Troia, ricorsioni, indecidibilità.

4.1.1 Il compilatore non nasce imparato

82 Andare a capo in C comporta almeno tre cose diverse

- (1) qualcuno avrà già detto all'environment (non ancora al compilatore K) se opera in Ascii o in un altro *character set*. Nel caso Ascii, saprà che 82, 83, 92, 110, 10 e 11 codificano risp. R S \ n LF VT;
- (2) K avrà fissato la corrispondenza tra i `char` e gli `int`, facendo in modo che 'R' è una maiuscola, mentre '\n' è il modo C di denotare LF; si spera che lo abbia

¹un logico-matematico di Harvard che forse è stato il massimo filosofo americano del '900.

fatto in modo portabile, contando sulla parte (1) per precisare se tra R ed S c'è solo un onesto nulla, o la mezza dozzina di simboli strani pigiati a caso dal dinosauro di turno quel giorno in fabbrica;

- (3) ma dirgli una volta per tutte che '\n'=LF è una cosa diversa dal riconoscere ogni volta il digrafo. Voglio dire che poi occorrerà metterlo in condizione di stabilire che, per esempio, "ciao\n" è lungo 6, e non 7 o 8. Ogni volta che lo vede. Quando scandisce un file (stdin o no) K riceve un byte alla volta, ed è lui ad interpretare il \ come inizio del digrafo che sub (2) è stato posto equivalente a LF e sub (1) ad Ascii 10. (Un'altra applicazione può capirlo come le pare.)

Schematizzando molto, possiamo dire che si ottiene la parte (3) inserendo in una macro o in una funzione di libreria come `getc()` qualcosa come

```
c = next();
if(c != '\\')
    return(c);
c = next();
if(c=='\\')
    return('\\');
if(c=='n')
    return('\n');
...
```

questo pezzo di codice è istruttivo: sfruttando le parti (1) e (2), esso spiega a K come si va a capo in C, in modo perfettamente portabile in qualsiasi character set. La scansione unificata (via ridirezioni) di ogni file come se fosse lo stdin, permette che questo pezzo di codice sia il solo *posto* dove si fissa \n=LF. Questo codice è usato per ogni file, e quindi anche da K nel momento in cui scandisce un file sorgente. Sorgente che magari è un compilatore K1 compilato da K. Anche perché le varianti come `getchar()`, `fgetc()` etc. usano `getc()`.

83 Nel C pre-Ansi non c'era \v : la soluzione non portabile Se scriviamo

```

c = next();
if(c != '\\')
    return(c);
c = next();
if(c=='\\')
    return('\\');
if(c=='n')
    return('\n');
if(c=='v')
    return('\v');
...

```

ci becchiamo un bel diagnostico perché K, essendo in stile K&R, nella fase (2) non gli ha detto che `\v` è il char VT, mentre gli apici del quarto `return` dicono che il suo argomento è un `char`. Sacrificando la portabilità, poniamo

```

c = next();
if(c != '\\')
    return(c);
c = next();
if(c=='\\')
    return('\\');
if(c=='n')
    return('\n');
if(c=='v')
    return(11);
...

```

84 Ora possiamo renderlo portabile Chiamiamo K2 il risultato di aggiungere a K il secondo codice del §83. Se ora compiliamo il primo codice dello stesso paragrafo col binario di K2, quello lo scandisce e quando trova il `return('\v')` lo interpreta come VT; siccome la fase (2) era portabile, otteniamo un VT in qualunque character set. Il programma K3 ottenuto sostituendo questo codice a quello in K2 è un compilatore portabile che accetta la tabulazione verticale.

85 S/vantaggi di un compilatore che impara Fermiamoci un attimo a riflettere su quello che abbiamo fatto. L'essenza è che è bastato dirglielo una volta, quando era ancora K2, perché, K3 e tutti i discendenti imparassero `\v=VT` una volta per tutte, e per tutti i character set. Questa capacità di impararare è la condizione

essenziale della possibilità di scrivere in C tutte le varianti del C medesimo che si rendano necessarie in rapporto a diverse architetture, e a diversi usi della stessa architettura (per esempio il kernel Linux usa, per ragioni di efficienza, un dialetto del C). Ma non tutto ciò che si impara è angelico.

4.1.2 La costruzione del quine

86 Scrivere *quasi sé stesso* Ponendo (1...K è una parte da cambiare poi)

```
# include<stdio.h>
/* self1.c */
char s[ ]={
    'm',
    'a',
    'i',
    'n',
    '(',
    ')',
    '{',
    '1',
    '.',
    '.',
    '.',
    'K',
    '}',
    0
};
main( ){
    int i;
    printf("# include<stdio.h>\n");
    printf("/* self1.c */\n");
    printf("char\t s[ ]={\n");
    for(i=0;s[i];i++)
        printf("\t%c,\n",s[i]);
    printf("%s\n",s);
}
```

otteniamo

```

538bm -> gcc -o selfmain.bin selfmain.c
539bm -> selfmain.bin
# include<stdio.h>
/* self1.c */
char    s[ ]={
        m,
        a,
        i,
        n,
        (,
        ),
        {,
        1,
        .,
        .,
        .,
        K,
        },
main(){1...K}

```

le indentature sono diverse perché il mio editor è settato a 4, e gcc a 8; per aggiustarle si può provare (io non l'ho fatto) con l'opzione

```
gcc -ftabstop=4 -o selfmain.bin selfmain.c
```

87 Partita in quattro mosse Prima si scrive un programma *P* che stampa una stringa *s* generica ma costante; una volta che ogni dettaglio di *P* (salvo *s*) è definito, si mette *P* al posto di *s*. In *C*

- (1) preambolo e dichiarazione di una stringa *s*, un carattere per riga;
- (2) una *main* che stampa *s* due volte: un carattere per riga, e poi come stringa;
- (3) sostituzione della stessa *main* (che ora è una costante nota) al posto di *s*;
- (4) aggiustamenti per accroccare insieme il tutto.

Le parti (1) e (2) sono fatte nel §86. La parte (3) richiederebbe che uno si mettesse lí a scrivere le 6 righe di quel *main*, un *char* alla volta al posto di 1...*K* (o che c'avesse tempo e voglia di scrivere un programma che lo fa al posto suo). La parte (4) è piú o meno cosí

```

char s[ ]={
    '\t',
    '0',
    '\n',
    '}',
    ';',
    '\n',
    '\n',
    '/',
    '*',
    '\n',

```

(200 righe ca. omesse)

```

0,
};

```

```

/*
 * la stringa s rappresenta tutto
 * il programma da 0 in poi
 */
main( )
{
    int i;
    printf("char\t s[ ]={\n");
    for(i=0;s[i];i++)
        printf("\t%d,\n",s[i]);
    printf("%s",s);
}

```

Chiamiamo P questo programma e Q la sua uscita. Abbiamo $Q = Q_1Q_2$, dove Q_1 è l'output delle righe di P fino allo scope della `for` incluso e Q_2 è prodotto dall'ultima `printf`. Si verifica per simulazione che $Q_1 =$

```

char s[ ]={
    9,
    48,
    10,
    125,
    73,
    10,
    10,
    47,
    42,
    10,

```

(200 righe ca. omesse)

a questo punto il test della `for` diviene falso (perché si ha $s[i] = 0$, stante la differenza tra `'0'` = 48 e 0) e si passa all'ultima `printf`, ottenendo $Q_2 =$

```

    0
};
    /*
    * la stringa s rappresenta tutto
    * il programma da 0 in poi
    */
main( )
{
    int i;
    printf("char\ts[ ]={\n");
    for(i=0;s[i];i++)
        printf("\t%d,\n",s[i]);
    printf("%s",s);
}

```

Q differisce da P solo perché l'enumerazione delle componenti dell'array di caratteri s è fatta con i numeri Ascii, invece che con caratteri espliciti. Siccome la seconda `printf` usa `%d`, l'output di Q è Q stesso.

88 Im/morale Così come Q scrive perfino il suo commento C , si può scrivere un qualunque programma che, non solo si riproduce, ma porta con sé, invece di un innocente commento, tutto il più o meno esplosivo bagaglio B che vuole.

4.1.3 L'inserimento di un cavallo di Troia

89 Non dura un attacco senza quine Al livello piú alto del compilatore `K` le righe di codice sono gestite da una `compile()` che in stile K&R è piú o meno cosí

```
compile(s)
char *s{
    ...
}
```

Uno può manomettere `K` trasformandolo in un `K_lgn` nella forma

```
compile(s)
char *s{
    if(match(s,"pattern")){
        compile("bug");
        return;
    }
    ...
}
```

in cui `pattern` è scritto in modo da intercettare il comando Unix `login`, e `bug` in modo da accettare una password abusiva. Supponiamo che `K` venga in qualche modo sostituito con `K_lgn`. Se e quando qualcuno ri-compila `login` io ottengo il diritto di loggarmi. Però nei posti che vale la pena di attaccare c'è sempre un amministratore serio che prima o poi leggerà il sorgente di `K_lgn` e provvederà ad escludermi.

90 Attaccare il compilatore con un quine Abbiamo visto che un quine può fare qualcosa in piú di scrivere se stesso. Supponiamo che `K` sia stato sostituito da un sorgente `K_lgn_q` nella forma

```

compile(s)
char *s{
    if(match(s,"pattern")){
        compile("bug");
        return;
    }
    if(match(s,"pattern2")){
        compile("quine");
        return;
    }
    ...
}

```

adesso quine scrive non solo se stesso ma i due costrutti `if` indicati. `pattern2` è congegnato in modo da scattare alla prima compilazione. In analogia a quanto fatto per gestire `\v` ora procediamo in tre fasi

- i due costrutti vengono compilati generando un binario `K_lgn_q.bin`
- gli stessi costrutti vengono cancellati dal sorgente `K_lgn_q` che quindi ridiventa uguale a `K`
- ma `K_lgn_q.bin` viene rinominato `K.bin` e messo al posto del binario autentico.

Tu hai un bello scrivere un compilatore sorgente pulito OK. Finché usi il mio binario fasullo, o un suo discendente, esso entra nella sua parte (in binario) di `quine`, e aggiunge al tuo compilato `OK.bin` le due parti abusive, permettendomi il login. Le colpe del padre ricadranno per sempre su tutta la sua schiatta.

4.2 Una forma debole del teorema di ricorsione

4.2.1 In un linguaggio generico

91 Notazioni² Un qualsiasi linguaggio di programmazione imperativo può essere visto come una coppia

$$\mathcal{L} = \langle \mathcal{P}, \mathcal{D} \rangle$$

²Questa sezione 4.2 è stata inserita all'ultimo momento, dopo che avevo finito il testo principale, per aggiungere delle cose dette a lezione; quindi, a parte il fatto che è probabile che ci siano più errori del solito, andrebbe studiata come un testo supplementare, separato dal resto.

in cui \mathcal{P} è una classe di programmi P , e \mathcal{D} è una classe di dati x, \dots, z . Ad ogni programma può essere associato un'arità, ossia un numero di posti od argomenti, non necessariamente costante. \vec{z} denoterà una n -pla di dati per un programma n -ario. Supporremo che ogni P per ogni \vec{y} *diverga*, oppure dia in uscita un solo $z \in \mathcal{D}$ (che naturalmente può essere formato da più *sotto-dati*, come nel caso di una stringa divisa in campi da opportuni separatori). La notazione

$$P : \vec{y} = z$$

significa: P per input \vec{y} diverge oppure dà in uscita z . Invece $P : \vec{y} \uparrow$ significa che P diverge in \vec{y} . Una conseguenza insoddisfacente di questa comoda convenzioni è che se $P : \vec{y} \uparrow$ allora $P : \vec{y} = z$ è vero per ogni (invece che per nessuna) z .

92 Funzione associata Il trattamento di programmi da parte di programmi è il sale dell'informatica. Questo può richiedere che i programmi siano trasformati in dati. Non in C, i cui dati sono stringhe, ma sempre in linguaggi che trattino solo numeri o array di numeri. Per discutere il teorema di ricorsione astraendo dal particolare linguaggio, assumiamo che sia data una iniezione $\Gamma(\cdot)$

$$\Gamma : \mathcal{P} \mapsto \mathcal{D}$$

Notazione P_x è l'unico programma P tale che $\Gamma(P) = x$.

(Vale dunque l'identità $\Gamma(P_x) = x$.)

Per ogni programma n -ario P_x , φ_x denota la funzione parziale³ tale che ($\Gamma^* = \bigcup_h \Gamma^h$ invece di Γ^n se n non è una costante)

$$\begin{aligned} \varphi_x : \mathcal{D}^n &\mapsto \mathcal{D} \\ \varphi_x(\vec{y}) \uparrow &\text{ sse } P_x : \vec{y} \uparrow \\ \varphi_x(\vec{y}) = z &\text{ sse } P_x : \vec{y} = z \quad \text{e non } P_x : \vec{y} \uparrow \end{aligned}$$

Osserviamo che una funzione può essere calcolata da diversi programmi⁴ sicché si può benissimo avere $x \neq y$ nonché $\varphi_x = \varphi_y$ ⁵.

³ossia non definita in tutto il suo dominio

⁴I programmi che calcolano una stessa funzione sono in realtà infiniti, a causa di diversi algoritmi, ma anche solo perchè possono includere divagazioni, come scrivere una stringa e poi cancellarla. Cosa stupida dalla quale i complessisti traggono conseguenze che stupide non sono.

⁵Siccome le funzioni sono casi particolari di insiemi (precisamente: insiemi costituiti da coppie), dire che due funzioni sono uguali significa dire che sono lo stesso insieme (di coppie).

93 Nel caso del C possiamo prendere l'identità come Γ . Stabiliamo inoltre che ogni volta che parliamo di qualche riga di codice come di un *programma* P vogliamo dire che esse sono il corpo (body) B di una funzione $\text{main}()\{B\}$ preceduta da tutte le direttive e dichiarazioni che le buone maniere sintattiche pretendono.

Esempio Ponendo $P=$

```
scanf("%s%s", a, b);
b=strcat(b, a)
printf("%s", b)
```

si ha

$$\varphi_{\text{scanf}("%s%s", a, b); \text{b}=\text{strcat}(\text{b}, \text{a}); \text{printf}("%s", \text{b})}(y, z) = zy$$

94 Una delle pietre angolari della teoria della ricorsività è noto come Teorema S - m - n (in onore della notazione originale di Kleene, che però lo aveva chiamato *Iteration Theorem*) o come *Parameter Theorem*. Il suo scopo (evocato dall'iniziale S) è di consentire sostituzioni uniformi di parametri. Le dimostrazioni matematiche usano trucchi di teoria dei numeri un pò faticosi, tanto che

- (a) sono pochi i libri che lo dimostrano per intero;
- (b) quando hai finito di capire tutti i passaggi, capisci che non hai capito la dimo.

Con tipi di dati un pò meno rudimentali dei numeri la dimostrazione è invece agevole ed istruttiva, come vedremo ripetendola in dettaglio in C e in Bash.

95 Teorema (Teorema S - m - n) Per ogni m, n , esiste un programma s_n^m tale che per ogni funzione $m + n$ -aria $\varphi_x(\vec{y}, \vec{z})$ si ha

$$\varphi_x(\vec{y}, \vec{z}) = \varphi_{s_n^m(x, \vec{y})}(\vec{z})$$

Dimostrazione. L'idea è semplice, anche se la sua realizzazione in alcuni casi può essere anche molto complicata. In termini di procedura possiamo dire che un programma P_p tale che (ponendo $m = n = 1$ per semplicità) $\varphi_p(x, y) = s_1^1(x, y)$ e quindi

$$P_p : x, y = u \quad \text{implica} \quad P_u : z = P_x : y, z$$

sarà piú o meno cosí

scrivi y
concatena z
applica x

Ma qui si ferma il discorso generale, perché *scrivere, concatenare, applicare*

- dipende troppo dalle caratteristiche del singolo linguaggio
- deve essere fatto in modo uniforme in x e in y
- comporta il mettere insieme un dato per P_p con uno per P_u
- se i dati sono numeri, implica impacchettare i dati y e z in un numero solo.

96 Teorema (Forma debole ⁶ o di Kleene del teorema della ricorsione). Per ogni $\varphi_x(y, x)$ esiste un *punto fisso* u tale che

$$\varphi_u(z) = \varphi_x(u, z)$$

Dimostrazione. Costruzione. Data x definiamo una nuova funzione

$$\varphi_{x_0}(y, z) = \varphi_x(s_1^1(y, y), z) \tag{A}$$

In sostanza (ossia ignorando le quattro difficoltà menzionate alla fine della dimo precedente) P_{x_0} per input y e z

- (a) seleziona e duplica y
- (b) lo passa al programma P_p della dimo precedente
- (c) passa a P_x l'uscita $s_1^1(y, y)$ di P_p

Possiamo ora definire il punto fisso menzionato nell'asserto

$$u = s_1^1(x_0, x_0) \tag{B}$$

Verifica della costruzione. Abbiamo

$\varphi_u(z)$	$= \varphi_{s_1^1(x_0, x_0)}(z)$	Definizione (B)
	$= \varphi_{x_0}(x_0, z)$	Teorema <i>S-m-n</i>
	$= \varphi_x(s_1^1(x_0, x_0), z)$	Definizione (A)
	$= \varphi_x(u, z)$	Definizione (B) ancora.

⁶debole perché nella sua dimo non si usa un elemento universale, ossia un interprete per \mathcal{P} , scritto in \mathcal{P} (approfondiremo nel §?? questa questione).

97 Esempio In C, per la x del §93, tale che $\varphi_x(y, z) = zy$, il teorema dice che c'è una stringa u tale che il programma P_u per input qualunque stringa z , dà in uscita la stessa z , seguita da u (più precisamente, seguita dal sorgente di P_u). Tuttavia, il modo troppo generico in cui abbiamo costruito sia P_p che P_{x_0} non ci permette di capire bene come sia fatto P_u : per ora sappiamo solo che esiste. I dettagli concreti della costruzione saranno studiati nella prossima sezione.

4.2.2 La forma debole in C

Puoi costruire un quine o un cavallo di Troia se sei un buon programmatore, o perché conosci il metodo. Il teorema di Kleene, o della ricorsione, dà il metodo: (1) per farlo; (2) per realizzare ricorsioni di forma qualunque; (3) per dimostrare l'indecidibilità di alcuni problemi.

Prima di attaccare in Bash la dimo completa e la discussione delle sue conseguenze, ridimostriamo in C la versione debole del §96; dobbiamo limitarci a questo perché il C funziona benissimo quando si tratta di *scrivere* funzioni, ma pone alcuni problemi quando si cerca di manipolare e *trasformare* funzioni:

- (a) Non c'è l'equivalente di `$0`; il nome della funzione va estatto dal suo codice; anche dei parametri il nome non è noto; con `<stdarg.h>` puoi gestire un numero non prefissato di variabili (come in `printf("format", x1, ..., xk)`) ma non sai il nome e il numero di quelle che vengono prima dei puntini.
- (b) Puoi chiamare funzioni scritte a compile time, non a run time.

Sub (a) ho delle difficoltà, non degli impedimenti; le aggiro con gli standard di calcolo che seguono. Per (b) dovrei usare un interprete; oppure uscire con `system()`, chiamare il compilatore, e poscia rientrare. Cosa non meno grottesca dell'avverbio appena usato.

98 Notazione (1) Prendiamo la classe delle funzioni (non dei programmi) C come la \mathcal{P} del §91. E prendiamo come \mathcal{D} la classe Σ di tutte le stringhe nell'alfabeto dei caratteri che possono figurare in un sorgente C.

(2) Nel nostro discorso una *funzione* può essere due cose: un sottoinsieme di Σ^{n+1} oppure una parte di un programma C. L'ambiguità è risolta per contrasto tra greco, e *fixed-width font*, oltre che, si spera, dall'intelligenza del contesto.

(3) `r, ..., z`, talora seguite da cifre decimali, sono generiche stringhe in Σ .

99 Standard di calcolo (1) Nel seguito abbiamo bisogno di passare dal codice di una funzione al nome (*function designator*) con cui viene chiamata. Stabiliamo che i nomi di tutte le nostre funzioni finiscono con un underscore; `foo` è una metanotazione per un codice nella forma `foo_(){...}` (\rightarrow 101). In certi casi `foo` denoterà, oltre al sorgente di `foo_()`, anche i sorgenti di qualche funzione da lei chiamata (vedi il Commento alla fine del §105).

(2) Concentreremo il nostro interesse su funzioni `x`

(a) descritte da stringhe `x` nella forma

$$x_(){...} \tag{C}$$

(b) senza prototipo, e senza parametri in entrata;

(c) nelle quali possono figurare ed essere modificate (oltre ad eventuali variabili interne) solo le tre variabili globali `a b c`. Esse sono stringhe dichiarate all'esterno, che possono essere assegnate sia dall'esterno, che dalla funzione.

(d) Per quanto stabilito alla fine della parte (1), non è detto che quella `}` chiuda quella `{`, che può essere invece chiusa da una `}` nascosta nei puntini.

(3) **Notazione** `a=y` significa che `y` è stata assegnata ad `a`; scriviamo poi (cfr. §91)

$$x:y,z=u$$

se dopo una chiamata di `x_()` con `a=y` e con `b=z` si ha divergenza, oppure si ha `c=u`. Assumo che le funzioni siano chiamate da programmi corretti, che includano tutte le necessarie dichiarazioni e direttive (tra cui le `include` per `stdio.h` e `string.h`).

(4) **Definizione** La funzione `x_()` *calcola in modo standard* (*s-calcola*) $\varphi^{(n)}$ se

$$x:y,z=u \quad \text{se e solo se} \quad \varphi(y,z) = u$$

(`z` assente per $n = 1$; `u` inesistente se e solo se $\varphi(y,z) \uparrow$). Una *s-funzione* è una funzione C che rispetta questo standard.

Nota Lo standard è solo un modo per gestire l'interfaccia con l'esterno, in assenza di analoghi dei parametri posizionali del Bash. Non c'è dunque ragione di proibire che una s-funzione chiami una funzione che non rispetti lo standard, purché non scambi informazione con quanto è esterno della s-funzione che l'ha chiamata.

Notazione φ_x è la funzione s-calcolata da `x_()`.

100 Richiamo di alcune funzioni di stringa della libreria standard `string.h`

<code>strcpy(s,t)</code>	<code>s = t</code>	(t viene copiata in s)
<code>strcat(s,t)</code>	<code>s = st</code>	(concatenazione)
<code>strchr(s,'c')</code>	puntatore	al primo c di s
<code>strncpy(s,t,j)</code>	<code>s = primi j char di t</code>	(non aggiunge '\0')

101 Esempio Ponendo

```
id_(){
  strcpy (c,a);
}
```

```
swap_(){
  strcpy(c,b);
  strcat(c,a);
}
```

si ha `id:y=y` nonché `swap:y,z=zy`. Lo si può anche scrivere nella forma

$$\varphi_{id}(y) = y \quad \varphi_{swap}(y, z) = zy$$

102 Nota La differenza concettuale tra questa notazione φ_{swap} e la notazione piú lunga usata nel §93 è che qui, avendo dato il nome `swap` alla stringa `swap_(){...}`, possiamo usare il nome (che è piú corto) invece dell'intera stringa (che è piú lunga). Si usano i nomi perché sono piú maneggevoli delle cose.

Osserviamo inoltre che stabilendo che il nome dell'intera stringa è ottenuto togliendo l'underscore al designatore della funzione, ossia ponendo `x=x_...` , abbiamo dato una regola per passare in modo uniforme da una classe di oggetti (che vengono scritti prima) alla classe dei loro nomi (a posteriori), così come si forma il codice fiscale a partire da un record anagrafico, scegliendo tre lettere del cognome, tre del nome, etc.

103 Estrazione del nome di una funzione standard La funzione

```

fn_(){
    char *p=c;
    strcpy(c,a);
    p=strchr(c,'(');
    *p='\0';
}

```

estrae in modo standard il nome della funzione assegnata ad `a` (l'ultima riga perché, come detto nel §100, `strchr()` non mette lo 0). Per esempio

```

$ cat > fn.c
# include<stdio.h>
# include<string.h>
char a[]="swap_(){strcpy(c,b);strcat(c,a);";
char c[80];
main(){
    fn_();
    printf("function name is %s\n",c);
}
fn_(){
char *p=c;
    strcpy(c,a);
    p=strchr(c,'(');
    *p='\0';
}
$ gcc -o fn.bin fn.c;fn.bin
function name is swap_

```

104 Notazione Per ogni $\tau : \Sigma \mapsto \Sigma$

$$\varphi_{\tau(x)}(y, z) \quad \text{sta per} \quad \tau(x) = u \quad \text{nonché} \quad \varphi_u(y, z)$$

105 Lemma della sostituzione diagonale (Variante del teorema *S-m-n*) Esiste una funzione `ds_()` che s-calcola la funzione $\sigma^{(1)}$ tale che per ogni $\varphi_x^{(2)}(y, z)$

$$\varphi_{\sigma(x)}(y) = \varphi_x(x, y)$$

Ossia esiste una funzione `ds_()` che assegna in ogni `x_()` il sorgente `x` ad `a`.

Dimostrazione. La funzione `ds_()` che adesso definiamo assegna il nome `sx_()` al valore in `x` di σ (qui `s` è un `char` fisso, non una stringa come altrove). Poniamo

```

ds_(){
    fn_();                // c=x_
    strcpy(b,c);          // b=x_
    strcpy(c,"s");        // c=s
    strcat(c,b);          // c=sx_
    strcat(c,"(){strcpy(b,a);}"); // c=sx_(){strcpy(b,a);}
    strcat(c,"strcpy(a,\""); // c=sx_(){strcpy(b,a);strcpy(a,"
    strcat(c,a);          // c=sx_(){strcpy(b,a);strcpy(a,"x
    strcat(c,"\"");}); // c=sx_(){strcpy(b,a);strcpy(a,"x");
    strcat(c,b);          // c=sx_(){strcpy(b,a);strcpy(a,"x");x_
    strcat(c,"()};}); // c=sx_(){strcpy(b,a);strcpy(a,"x");x_();}
    strcat(c,a);          // c=sx_(){strcpy(b,a);strcpy(a,"x");x_();}x
}

```

Dai commenti a destra dei // si vede passo passo che `ds` per `x=x_(){...}` produce (a meno di accapo e indentazioni) una stringa `sx` consistente nella definizione di una funzione `sx_()`, seguita da una copia di `x`. Abbiamo `sx=`

```

sx_(){
    strcpy(b,a);
    strcpy(a,"x");
    x_();
}
x

```

L'asserto segue osservando che la funzione `sx_()`, per `a=y`

- (1) trasferisce `y` in `b`
- (2) assegna `x` ad `a`
- (3) chiama `x` con `a=x` e `b=y`.

Commento La chiamata (3) richiede una copia di `x`. A ciò aveva provveduto l'ultima chiamata di `strcat` nella definizione di `ds`.

106 Esempio Applichiamo il Lemma ad un programma che chiamo `SMN.c` che

- assegna la costante `swap` ad `a`;
- chiama da `main` la funzione `ds_()`
- ne stampa l'uscita `sswap`

```

# include<stdio.h>
# include<string.h>
/* SMN.c */
char a[]="swap_(){strcpy(c,b);strcat(c,a);}"; char b[80], c[80];
main(){
    ds_();
    printf("function sswap_() is\n %s\n",c);
}
ds_(){
    fn_();
    strcpy(b,c);
    strcpy(c,"s");
    strcat(c,b);
    strcat(c,"(){strcpy(b,a);}");
    strcat(c,"strcpy(a,\"\");");
    strcat(c,a);
    strcat(c,"\\");");");
    strcat(c,b);
    strcat(c,"();}");");
    strcat(c,a);
}
fn_(){
char *p=c;
    strcpy(c,a);
    p=strchr(c,'(');
    *p='\0';
}
609 bm -> gcc -o SMN.bin SMN.c; SMN.bin
function sswap_() is
sswap_(){ // a capo aggiunti
strcpy(b,a);
strcpy(a,"swap_(){strcpy(c,b);strcat(c,a);}");
swap_();
}
swap_(){
strcpy(c,b);
strcat(c,a);
}

```

107 Teorema (Rivisitazione in C del teorema 96) Per ogni $\varphi_x^{(2)}(y, z)$ esiste un punto fisso u tale che

$$\varphi_u(z) = \varphi_x(u, z).$$

Dimostrazione. Data x nella forma (C) del §99, definiamo una nuova funzione x_0

```
x0_() {
  ds_();
  strcpy(a, c);
  x_();
}
ds
x
```

Si ha

$$\varphi_{x_0}(y, z) = \varphi_x(\sigma(y), z) \tag{D}$$

perché

- (1) la chiamata `ds_()` con `a=y` dà `c = $\sigma(y)$` ;
- (2) la chiamata `x_()` con `a = $\sigma(y)$` (via `strcpy(a, c)`) e con `b=z` dà

$$c = \varphi_x(\sigma(y), z)$$

Definiamo ora

$$u = \sigma(x_0) \tag{E}$$

L'asserto segue perché abbiamo

$$\begin{aligned} \varphi_u(z) &= \varphi_{\sigma(x_0)}(z) && \text{per la definizione (E)} \\ &= \varphi_{x_0}(x_0, z) && \text{per il lemma 105} \\ &= \varphi_x(\sigma(x_0), z) && \text{per la (D)} \\ &= \varphi_x(u, z) && \text{per la (E) un'altra volta} \end{aligned}$$

4.3 La forma debole in Bash

108 Anche se non ce ne dovrebbe essere bisogno ricordiamo (\rightarrow 45) che

```
set parms
```

assegna i $k \geq 1$ parametri `parms` alle variabili posizionali $\$1, \dots, \K

```
$ cat > set_echo
set bari $*
echo $*
$ . set_echo roma pisa
bari roma pisa
```

E ricordiamoci pure (\rightarrow 13) che un filtro è uno script che legge lo stdin, lavora e manda un'uscita allo stdout.

109 Teorema (La forma debole in Bash) Esiste un filtro `kleene` che per ogni filtro `x` fornisce un *punto fisso* `kx` tale che per ogni `z` si ha

$$kx z = x kx z$$

Dimostrazione. Ponendo

```
| (kleene) =
| echo "set k$1 \${*};$(cat $1)">k$1
| chmod 755 k$1
```

otteniamo

```
$ kleene foo
| (kfoo) =
| set kfoo $*;
| (foo)
```

perché la prima riga di questo script manda nel file `kfoo`

- (a) il comando `set kfoo $*`
- (b) seguito dall'intero file `foo`, grazie alla sostituzione di comando indicata.

L'asserto segue perché `set bar $*`; `(bar)` si comporta come `bar bar $*`.

110 Esempio Cominciamo col porre

```
541bm -> cat > idnt          # def. di uno script che
echo $1                      # si comporta da identita'
539bm -> cat > cat2          # def.e di uno script che manda allo stdout i
cat $1 $2                    # contenuti di due file passati come parametri
540bm -> . cat2 idnt cat2    # idnt e cat2 stesso come parametri per cat2
echo $1                      # (idnt)
cat $1 $2                    # (cat2)
```

Prendendo `cat2` come `foo` otteniamo

```
543bm -> kleene cat2
544bm -> kcat2 idnt
set kcat2 $*;cat $1 $2
echo $1
545bm -> . cat2 kcat2 idnt
set kcat2 $*;cat $1 $2
echo $1
```

*ℓ*543 diamo `cat2` in entrata allo script che dimostra il teorema

*ℓ*544 applicando lo script prodotto da `kleene` all'unico parametro `idnt` otteniamo due righe: la seconda è uguale a `(idnt)`;

*ℓ*545 dice che la prima riga di *ℓ*544 è proprio `(kcat2)` come pomesso dal teorema.

Togliendo il `$2` da `cat2` otteniamo uno script che si auto-riproduce ed è piuttosto compatto

```
552bm -> cat > cat1
cat $1
553bm -> kleene cat1
554bm -> kcat1
set kcat1 $*;cat $1
555bm -> cat kcat1
set kcat1 $*;cat $1
```

Naturalmente, prima di auto-riprodursi può fare varie cose

```
559bm -> cat>nuovo
echo $((9**9)); cat $1
560bm -> kleene nuovo
```

```
561bm -> knuovo
387420489
set knuovo $*;echo $((9**9)); cat $1
```

111 Esempio: calcolo del fattoriale Ponendo

```
| (fact) =
| arg=$2;
| if ((arg)); then
| ((b=arg, --b));
| b=${$1 $b};
| echo $((b*arg));
| else echo 1;
| fi
```

supponiamo ora che il filtro `f` calcoli la funzione numerica f , in modo che $f(n) = h$ implichi

```
$ f n
h
```

In tal caso otteniamo

```
$ fact f n
k
```

dove $k = f((n - 1) * n)$. Infatti si assegna $n - 1$ a `b`, quindi si applica `f` a `b`; e infine si manda in `stdout` $b * n$. Per esempio se si prende l'identità come `f` e si prende 5 come secondo parametro, si assegna 4 a `b`, si applica l'identità e si moltiplica per 5

```
532bm -> fact idnt 5
20
```

A questo punto si intuisce che, per calcolare il fattoriale, dobbiamo passare come primo parametro uno script che si comporti come `fact`. A ciò provvede il teorema

```
537bm -> kleene fact
538bm -> kfact 5
120
539bm -> cat kfact
set kfact $*;arg=$2; if ((arg)); then ((b=arg, --b));      # a capo mio
b=${$1 $b}; echo $((b*arg)); else echo 1; fi
```

ℓ539 mostra che lo script `kfact` differisce da `fact` solo per la riassegnazione preliminare delle variabili posizionali.

112 Autoriferimento, non uguaglianza Il teorema di Gödel costruisce una formula G ed una seconda formula che dice che G non è derivabile, e che è equivalente (nell'interpretazione voluta) a G stessa. Sarebbe sciocco dire che G occorre in G . Così qui non abbiamo uno script che occorre in se stesso, cosa ovviamente impossibile, ma uno script `foo` che chiama un altro scritto il quale contiene al suo interno una copia di `foo`. Per questo, insisto, ho detto poche righe più su che dovevamo "passare come primo parametro uno script *che si comporti come*" lo script dato.

4.4 La forma forte del teorema

4.4.1 Premessa

113 Caratterizzazioni delle funzioni ricorsive⁷ Possiamo dire che due linguaggi $\mathcal{L}_i = \langle \mathcal{P}_i, \mathcal{D}_i \rangle$ ($i = 1, 2$) sono *altrettanto potenti* se in ciascuno si può definire un interprete o un compilatore per l'altro (tali sono ad esempio il C e il Pascal). Questa relazione è ovviamente una equivalenza. Essa astrae dal tipo dei dati (stringhe, numeri, alberi). Le funzioni calcolate da un programma di un qualsiasi linguaggio equivalente al C costituiscono la classe \mathcal{R} delle funzioni *ricorsive* (*generali*, o anche *parziali*). Di ogni classe equivalente al C si dice che è una *caratterizzazione di \mathcal{R}* . C'è sostanziale unanimità sulla Tesi di Chirch, secondo cui

- (a) ogni funzione che attualmente può essere calcolata è ricorsiva;
- (b) la dimostrazione (per esempio, per assurdo) che una data funzione non è ricorsiva dovrebbe convincere che essa *non è calcolabile*; non nel senso che non la sappiamo calcolare *adesso*, ma nel senso che non poltrà *mai* essere calcolata se non con metodi attualmente *inconcepibili*.

Quanto sub (a) è una semplice evidenza empirica. Quanto sub (b) si basa anche sugli interpreti. \mathcal{R} contiene tutti gli elementi di base (assegnazioni e valutazioni) dei vari linguaggi ed è chiusa rispetto a tutti i loro costrutti (composizioni, ricorsioni, etc.). Quindi il calcolo in futuro di una funzione non ricorsiva si baserebbe su operazioni o su composizioni oggi inesistenti, anche a livello teorico.

⁷Nella versione finale questo paragrafo troverà un più logico inserimento all'inizio del capitolo.

114 Fatto: esistono funzioni universali Con gli interpreti si può fare un *giro completo*: andata da un dato \mathcal{L}_1 ad un altro \mathcal{L}_2 e ritorno da questo a quello. In ogni caratterizzazione di \mathcal{R} esiste dunque una funzione *universale*, tale che φ_{univ} (notazioni come nel §92)

$$\varphi_{\text{univ}}(\mathbf{x}, \mathbf{y}) = \varphi_{\mathbf{x}}(\mathbf{y}).$$

Le più antiche caratterizzazioni di \mathcal{R} risalgono agli anni '30, e sono dovute (indipendentemente) a Turing e a Church. Il primo⁸ è considerato il padre dell'informatica perché in un primo tempo, in quanto matematico, introdusse i concetti fondamentali, ed in particolare quello di *macchina universale*; in un secondo momento, da ingegnere, progettò e realizzò macchine decrittatrici e un computer. L'analogia progettuale tra quella idea e quel prodotto industriale colpisce.

115 Diagonalizzazione Si chiama *diagonalizzazione* un metodo di dimo che si basa sul rendere uguali (spesso mediante una sostituzione) due cose. Il nome viene dalla retta $x = y$. Per esempio si dimostra che esiste un numero reale non razionale nell'intervallo $]0, 1[$ osservando che

(1) tutti i razionali nell'intervallo sono enumerati dalla sequenza

$$x_i = b_1^i b_2^i \dots b_j^i \dots \quad (i, j \geq 1; b_j^i = 0, 1)$$

(2) definendo prima per diagonalizzazione e poi per bit-negation

$$\bar{x} = \bar{b}_1^1 \bar{b}_2^2 \dots \bar{b}_h^h \dots \quad (\bar{b} = 1 - b)$$

si ottiene un numero $\bar{x} \in]0, 1[$ ma diverso da ogni b^i per almeno un bit.

116 Forma debole e forma forte La forma debole del teorema di ricorsione permette di *diagonalizzare* solo sugli argomenti. Con $\varphi_{\mathbf{u}}(\mathbf{z}) = \varphi_{\mathbf{x}}(\mathbf{u}, \mathbf{z})$ si diagonalizza rispetto ad un argomento, stabilendo una relazione tra una funzione data e un valore di punto fisso per un suo argomento; si fissa uno degli argomenti per la funzione data $\varphi_{\mathbf{x}}$ senza creare veramente una nuova funzione. La forma forte del teorema sfrutta l'esistenza di una funzione universale per costruire una nuova funzione. La sua dimo non vale dunque per quei linguaggi (meno potenti del C) che non ammettano un elemento universale. Nella sez. ?? ne vedremo le conseguenze positive e negative.

⁸Bisogna ricordare con sdegno che fu perseguitato fino al suicidio dallo stato inglese perché gay.

4.4.2 La forma forte per un linguaggio generico

117 Notazioni per programmi che scrivono programmi Per discutere di programmi che producono per qualche argomento y (la rappresentazione nei dati di) un altro programma P_u , conveniamo che (cf. §104)

$$\varphi_{\varphi_x(y)}(z) \text{ sta per } \varphi_u(z) \text{ nonch  } u = \varphi_x(y)$$

Esempio La funzione σ del Teorema 105   ora denotata da φ_{ds} , e l'enunciato del teorema pu  essere riscritto nella forma

$$\varphi_{\varphi_{ds}(x)}(z) = \varphi_x(x, z)$$

118 Fatto Ogni caratterizzazione di \mathcal{R}   uniformemente chiusa rispetto a ogni tipo di composizione. Pertanto in ciascuna di esse esiste un programma P_{hd-cmp} per la *composizione in testa*, tale che

$$\varphi_{\varphi_{hd-cmp}(u,w)}(y, z) = \varphi_u(\varphi_w(y), z)$$

Per esempio, in una caratterizzazione con programmi e dati numerici si avr 

$$\varphi_{sum}(y, z) = y + z \text{ nonch  } \varphi_{sq}(y) = y^2 \text{ implica } \varphi_{\varphi_{hd-cmp}(sum, sq)}(y, z) = y^2 + z$$

119 Teorema (Forma forte o di Rogers del teorema della ricorsione). In ogni caratterizzazione di \mathcal{R} , per ogni programma P_x che produce programmi, esiste un punto fisso u tale che

$$\varphi_{\varphi_x(u)}(z) = \varphi_u(z)$$

Dimostrazione. Dato P_x , poniamo

$$w = hd-cmp(univ, x) \tag{F}$$

Prendendo come u il punto fisso associato a φ_w dalla forma debole del teorema otteniamo (giustificazioni accanto a ciascun $=$, salvo l'ultima che ci viene dal §114)

$$\varphi_u(z) = (\S 96) \varphi_w(u, z) = (F) \varphi_{hd-cmp(univ, x)}(u, z) = (\S 118) \varphi_{univ}(\varphi_x(u), z) = \varphi_{\varphi_x(u)}(z)$$

Commento. Per una migliore comprensione del senso del teorema si veda il § ??

4.4.3 La forma forte in Bash

120 Ritorno da Bash al linguaggio generico; scriptmaker Prendiamo ora come classi \mathcal{P} e \mathcal{D} del §91 gli script Bash e le stringhe Σ , e stabiliamo di denotare con φ_x la funzione (parziale) $\varphi : \Sigma^K \mapsto \Sigma$ tale che x è un filtro e si ha

$$x : y_1 \dots y_K = u \quad \text{sse} \quad \varphi_x(y_1, \dots, y_K) = u$$

Osservazione A differenza di quanto fatto per il C, in cui la x di φ_x denotava l'intero codice $x_()\{\dots\}$, questa notazione non è del tutto conforme allo spirito del §92 perché non si può risalire dal nome `foo` di uno script al suo testo (`foo`).

Definizione Chiamiamo *script maker* un filtro x (a) eseguibile; (b) che produce, per ogni o qualche input, un file $x_$, contenete uno script eseguibile.

```
-> x y
    | (x_) =
    | ...
                                [script dipendente da y]
x_
```

Al solito, $\varphi_{\varphi_x(y)}(z)$ sta per $\varphi_u(z)$ dove $u = \varphi_x(y)$

121 Ricorsione senza istruzioni di controllo Osserviamo che nel frammento di Bash studiato finora

- (a) ci sono assegnazioni, valutazioni, e simili
- (b) ci sono istruzioni per mandare il controllo in avanti (*if-then-else*, composizioni per concatenazione e per sostituzione di comando)
- (c) non ci sono istruzioni per mandarlo indietro, così consentendo di ripetere righe di codice scritte più su
- (d) ma si possono creare file eseguibili

Per produrre uno script che calcola ricorsivamente il fattoriale, abbiamo fatto leva sul punto (d). Lo si vede analizzando il modo in cui lo script `fact` dà, per ogni script x

$$\varphi_{\text{fact}}(x, k) = \begin{cases} 1 & \text{qualora } k = 0 \\ \varphi_x(k-1) * k & \text{altrimenti} \end{cases}$$

e il modo in cui con il teorema di Kleene⁹ si risolve l'equazione

$$\varphi_{\mathbf{kfact}}(\mathbf{k}) = \varphi_{\mathbf{fact}}(\mathbf{kfact}, \mathbf{k}) \quad (= \varphi_{\mathbf{kfact}}(\mathbf{k} - 1) * \mathbf{k} \quad (\mathbf{k} > 0))$$

La forma forte del teorema ci dà il metodo generale per fare ricorsioni.

122 Teorema (La forma forte in Bash) Esiste uno script `rogers` che per ogni script-maker `x` produce un *punto fisso* `krx` tale che, per ogni `z` si ha

$$\mathbf{rogers\ x} = \mathbf{krx} \quad \text{nonché} \quad \varphi_{\mathbf{krx}}(\mathbf{z}) = \varphi_{\varphi_x(\mathbf{krx})}(\mathbf{z})$$

Proof. Poniamo (\rightarrow 109 per lo script `kleene`)

```
| (rogers) =
| echo "$1 \ $1;${1}_ \ $2" > r$1
| kleene r$1
```

la ridirezione della prima riga e l'applicazione di `kleene` producono due file

```
-> rogers x
| (rx) =
| x $1
| x_ $2
| (krx)
```

se `x` è uno script-maker, il comando `rx y z`

- (a) lancia il comando `x y` che produce l'eseguibile `x_` (\rightarrow 120(b));
- (b) lancia il comando `x_ z`

Abbiamo dunque

$$\varphi_{\mathbf{rx}}(\mathbf{y}, \mathbf{z}) = \varphi_{\varphi_x(\mathbf{y})}(\mathbf{z})$$

Il teorema 109, applicato a `rx`, produce lo script `krk` promesso dall'asserto.

⁹Studenti frequentanti a.a. 2007-8: seguendo l'idea di uno di voi, ho modificato la dimo del teor. 109 (la prima volta dopo May 11, 2008 che leggete questa nota, scaricate anche da pag.83 a qui); in particolare adesso lo script `kleene` non mette l'underscore tra `k` e il nome dello script; ora quindi `kleene x` genera il file `kx` (invece di `k_x`) e si ha `kx z=k kx z` (invece di `k_x z = x k_x z`).

123 Funzioni ricorsive primitive unarie Il fattoriale è un caso particolare di un tipo di ricorsione chiamato *ricorsione primitiva* (PR) (che non significa *rozza*, visto che se l'è inventata Gödel). Si dice che $f(n)$ è definita per PR nella *funzione base* (costante) c e nell'*invariante* (o passo) $h(p, q)$ se si ha

$$\begin{cases} f(0) = c \\ f(n+1) = h(f(n), n) \end{cases}$$

Per esempio il resto della divisione per due è PR nella base 1 e nel passo $h(p, q) = 1 - p$; invece $[n/2]$ lo è in 0 e nella funzione $h = p + \text{even}(q)$

$$\begin{cases} \text{even}(0) = 1 \\ \text{even}(n+1) = 1 - \text{even}(n) \end{cases} \quad \begin{cases} \text{half}(0) = 0 \\ \text{half}(n+1) = \text{half}(n) + \text{even}(n+1) \end{cases}$$

124 Ancora ! Il fattoriale è PR in 1 e in $p * q$. Applichiamo il teorema di Rogers, *parametrizzandone* la base e il passo in vista di una generalizzazione a PR qualsiasi

```
| (fct_base)=
| echo 1
| (fct_step)=
| echo $(( $1 * $2 ))
| (fct)=
| echo 'arg=$1;
| if ((arg)); then ((b=arg, --b));
| b=$(( $1 $b ));
| echo $(fct_step $b $arg);
| else echo $(fct_base); fi' > fct_
| chmod 755 fct_
592bm -> fct idnt
593bm -> cat fct_
arg=$1; if ((arg)); then ((b=arg, --b));
b=$((idnt $b)); echo $(fct_step $b $arg);
else echo $(fct_base); fi
594bm -> fct_ 5
20
595bm -> rogers fct
596bm -> krfct
1
597bm -> krfct 5
120
```

125 Uso di sed per passare da fct a even Poniamo

```
| (even_base) =  
| echo 1  
| (even_step) =  
| echo $((1-$1))
```

trasformiamo ora l'intero file che abbiamo chiamato `fct`, in un nuovo file, che chiamiamo `even`, usando `sed` per sostituire, in ogni riga, la (sotto-)stringa `fct` con la stringa `even`

```
528 bm -> cat > even_base  
echo 1  
529 bm -> cat > even_step  
echo $((1-$1))  
530 bm -> cat fct | sed s/fct/even/g > even  
531 bm -> cat even  
echo 'arg=$1;  
if ((arg));then ((b=arg,--b));  
b=$(' $1' $b);  
echo $(even_step $b $arg);  
else echo $(even_base);fi'>even_  
chmod 755 even_  
546 bm -> 755 even_base even_step even even_  
547 bm -> rogers even  
548 bm -> kreven 9  
0
```

l530 il file `fct` è passato dalla pipeline a `sed`; questo, riga per riga, *s*-ostituisce *g*-lobalmente stringa `fct` con `even`; lo `stdout` di `sed` è ridiretto al nuovo file `even`.

126 Una funzione PR unaria qualsiasi Il prossimo script aspetta in entrata il nome `fn` di una funzione PR unaria, e assume che gli script `fn_base` e `fn_step` siano stati già scritti. Compila uno script-maker `fn` per `fn`, dà un pò di permessi e chiama `rogers`.

```
| (sedpr) =  
| a=$(echo "sed s/fct/$1/g")  
| cat fct | $a > $1  
| chmod 755 $1_base $1_step $1
```

```
| rogers $1
| chmod 755 kr$1
```

Si ottiene lo script `krfn` che calcola `fn`. Per esempio

```
530bm -> cat half_base
echo 0
531bm -> cat half_step
a=$(kreven $2)
b=$1
echo $((a+b))
532bm -> sedpr half
533bm -> krhalf 11
5
534bm -> krhalf 12
6
535bm -> krhalf
0
```

4.5 Funzioni universali e programmi totali

127 Teorema I linguaggi di programmazione che definiscono solo funzioni totali non ammettono funzione universale.

Dimostrazione. Ad absurdum. In un linguaggio $\mathcal{L} = \langle \mathcal{P}, \mathcal{D} \rangle$ si abbia dunque

$$\varphi_{\text{univ}}(\mathbf{x}, \mathbf{y}) = \varphi_{\mathbf{x}}(\mathbf{y}) \tag{G}$$

per ogni \mathbf{x}, \mathbf{y} e un certo `univ`. Assumiamo per semplicità $\mathcal{D} = \Sigma$. Trattandosi di un linguaggio di programmazione, sarà possibile duplicare un dato, modificare un output per esempio concatenandogli un `-` e sarà possibile comporre programmi, in modo che per ogni `p` ci sia un `p-` tale che per ogni `q`

$$\varphi_{p-}(q) = \varphi_p(q, q)- \tag{H}$$

Ma allora, scegliendo `univ` come `p` otteniamo la contraddizione

$$\varphi_{\text{univ-}}(\text{univ-}) \stackrel{(H)}{=} \varphi_{\text{univ}}(\text{univ-}, \text{univ-})- \stackrel{(G)}{=} \varphi_{\text{univ-}}(\text{univ-})-$$

128 Solo simulazioni Il programma P_{univ} può (a) procedere per simulazione, oppure (b) può essere basato sull'analisi del programma oggetto, basata su criteri molto intelligenti di analisi, scomposizione o chissà che. Nel caso (a), la sua *time complexity* $T(\text{univ}, \mathbf{x}, \mathbf{z})$ (tempo di CPU, numero di passi, etc.) sarà maggiore di $T(\mathbf{x}, \mathbf{z})$; siccome \mathbf{x} è generica, la complessità di P_{univ} dovrà maggiorare strettamente le complessità di tutti i programmi del linguaggio. Compresa la sua, se appartenesse al linguaggio; il che non può essere. Questa è una passabile spiegazione intuitiva della dimo. Può essere resa più precisa spiegando meglio chi è T , come si misurano gli input, etc. Ma la dimo del teorema

- ci risparmia questa seccatura;
- ci dice una cosa in più: il metodo (b) non esiste, e non c'è niente di meglio della simulazione brutta.

129 Conseguenze Naturalmente si può scrivere il programma universale per un linguaggio \mathcal{L}_1 in un linguaggio \mathcal{L}_2 più potente. Ma non lo puoi usare per fare ricorsioni senza uscire da \mathcal{L}_1 . Quindi non si può pretendere un linguaggio \mathcal{L}

- con programmi che non divergano
- con ricorsioni qualunque, o con un interprete per \mathcal{L} all'interno di \mathcal{L} .

4.6 Indecidibilità e ricorsività

130 Classi decidibili Una sotto-classe \mathcal{C} di \mathcal{R} è (per definizione) *decidibile* se c'è un programma P_c che ne calcola la funzione caratteristica ossia

$$P_c(\mathbf{x}) = 0 \quad \text{qualora} \quad \varphi_{\mathbf{x}} \in \mathcal{C}; \quad P_c(\mathbf{x}) = 1 \quad \text{qualora} \quad \varphi_{\mathbf{x}} \notin \mathcal{C} \quad (\text{I})$$

\mathcal{A} è una *sottoclasse non banale* di \mathcal{B} se non è vuota, e non la esaurisce.

131 Teorema (Rice) Tutte le sottoclassi non banali di \mathcal{R} sono indecidibili.

Dimostrazione. Ad absurdum. Si abbia (I) per una $\mathcal{C} \subset \mathcal{R}$ con \mathbf{v}, \mathbf{w} tali che

$$\varphi_{\mathbf{v}} \in \mathcal{C}; \quad \varphi_{\mathbf{w}} \notin \mathcal{C} \quad (\text{J})$$

Modifichiamo (I) in modo che si abbia

$$P_c(\mathbf{x}) = \mathbf{w} \quad \text{qualora} \quad \varphi_{\mathbf{x}} \in \mathcal{C}; \quad P_c(\mathbf{x}) = \mathbf{u} \quad \text{qualora} \quad \varphi_{\mathbf{x}} \notin \mathcal{C} \quad (\text{K})$$

Per il teorema di Rogers (con c come x) esiste una v tale che

$$\varphi_{\varphi_c(u)} = \varphi_u \tag{L}$$

Ma questa φ_u non può essere, perché

$$\begin{array}{ll} \varphi_u \in \mathcal{C} \text{ implica } \varphi_c(u) = w & \text{per la (K) con } u \text{ come } x \\ \varphi_{\varphi_c(u)} \notin \mathcal{C} & \text{per la (J)} \\ \varphi_u \notin \mathcal{C} & \text{per la riga precedente e per la (L)} \end{array}$$

mentre con un simmetrico argomento si giunge a contraddizione dall'ipotesi $\varphi_u \in \mathcal{C}$.

132 Commento Sono quindi indecidibili problemi come (per input x, y, z)

- $\varphi_x(y) \downarrow$ (halting problem)
- $\forall u \exists w \varphi_x(u) = w$ (funzioni totali)
- $\forall u \varphi_x(u) = c$ (funzioni costanti, un problema per ogni c)
- $\exists u \varphi_x(u) = c$ (appartenenza al codominio di una costante)
- $\forall u \varphi_x(u) \leq c$ (codominio limitato da una costante)
- $\forall u \varphi_x(u) = u$ (funzioni identità)
- etc.

133 Problema Poniamo $x =$

```
read; echo $REPLY
```

È ovviamente decidibile che φ_x è l'identità. Questo contraddice in qualche modo la penultima riga del Commento precedente?