Capitolo 23

Ricerca, scansione ed elaborazione di file

23.1 Espressioni regolari standard 915
23.1.1 RE: BRE e ERE 915
23.1.2 Problemi di localizzazione 916
23.1.3 Composizione di un'espressione regolare e corrispondenza 916
23.1.4 Espressioni tra parentesi quadre 919
23.1.5 Precedenze 921
23.2 Confronto sintetico tra le espressioni regolari «reali» . 922
23.3 Grep 923
23.3.1 Grep e Gzip integrati 925
23.4 Find 925
23.5 SED 930
23.5.1 Avvio dell'eseguibile 930
23.5.2 Logica di funzionamento 931
23.5.3 Script e direttive multiple 932
23.5.4 Direttive 932
23.5.5 Esempi 936
23.6 Introduzione a AWK 937
23.6.1 Principio di funzionamento e struttura fondamentale 937
23.6.2 Avvio dell'interprete 940
23.6.3 Espressioni 941
23.6.4 Istruzioni 946
23.6.5 Variabili predefinite 951
23.6.6 Esempi 952
23.6.7 Dichiarazione di funzioni 953
23.6.8 Array 954
23.7 Introduzione a M4 957
23.7.1 Principio di funzionamento 958
23.7.2 Convenzioni generali 958
23.7.3 Istruzioni condizionali, iterazioni e ricorsioni 963
23.7.4 Altre macroistruzioni interne degne di nota 964
23.8 Riferimenti 967

`awk` 937 `changecom` 965 `define` 961 `divert` 965 `dnl` 964 `egrep` 923 `fgrep` 923 `find` 925 `forloop` 964 `ifdef` 963 `ifelse` 963 `include` 965 `sed` 930 `shift` 964 `sinclude` 965 `undefine` 963 `undivert` 965 `zegrep` 925 `zfgrep` 925 `zgrep` 925

«a2» 2013.11.11 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://informaticalibera.net

23.1 Espressioni regolari standard

L'espressione regolare è un modo per definire la ricerca di stringhe attraverso un modello di comparazione. Viene usato da diversi programmi di servizio, ma non tutti aderiscono alle stesse regole. Per studiare la grammatica delle espressioni regolari, occorre abbandonare qualunque resistenza, tenendo presente che l'interpretazione di queste espressioni va fatta da sinistra a destra; inoltre, ogni simbolo può avere un significato differente in base al contesto in cui si trova.

> Raramente si può affermare che un'espressione regolare sia «errata»; nella maggior parte dei casi in cui si commettono degli errori, si ottiene comunque qualcosa che può avere un significato (indipendentemente dal fatto che questa possa avere o meno una corrispondenza). È ancora più difficile che una realizzazione in cui si utilizzano le espressioni regolari sia in grado di segnalare un errore grammaticale nella loro scrittura.

23.1.1 RE: BRE e ERE

Un'espressione regolare, come definita nello standard POSIX, può essere espressa attraverso due tipi di grammatiche differenti: le espressioni regolari di base, o elementari, identificate dall'acronimo BRE (*Basic regular expression*), mentre le espressioni regolari estese, identificate dall'acronimo ERE (*Extended regular expression*). Per fare riferimento a espressioni regolari non meglio definite, si usa anche soltanto l'acronimo RE.

23.1.2 Problemi di localizzazione

L'espressione regolare, essendo un mezzo per identificare una porzione di testo, risente di problemi legati alle definizioni locali degli insiemi di caratteri.

Per prima cosa occorre considerare che gli alfabeti nazionali sono differenti da un linguaggio all'altro. Dal punto di vista della localizzazione, gli elementi che compongono gli alfabeti sono degli ***elementi di collazione*** (*collating element*). Questi elementi compongono un insieme ordinato, definito ***sequenza di collazione*** (*collating sequence*), che permette di stabilire l'ordine alfabetico delle parole. In situazioni particolari, alcuni elementi di collazione sono rappresentati da più di un carattere, cosa che può dipendere da motivazioni differenti. Per fare un esempio comune, ciò può essere causato dalla mancanza del carattere adatto a rappresentare un certo elemento, come succede nella lingua tedesca quando si utilizza un insieme di caratteri che non dispone delle vocali con la dieresi, oppure manca la possibilità di indicare la lettera «ß»:

```
ß        --> ss
ä        --> ae
ö        --> oe
ü        --> ue
```

Nella lingua tedesca, nel momento in cui si utilizzano le stringhe «ae», «oe», «ue» e «ss», in sostituzione delle lettere che invece avrebbero dovuto essere utilizzate, queste stringhe vanno considerate come la rappresentazione di tali lettere, costituendo così un elemento di collazione unico. Per esempio, in tedesco la parola «schal» viene prima di «schälen», anche se la seconda fosse scritta come «schaelen».

Ai fini della definizione di un'espressione regolare, questo fatto si traduce nella possibilità di fare riferimento a degli elementi di collazione attraverso la stringa corrispondente, nel momento in cui non è possibile, o non conviene usare il carattere che lo rappresenta simbolicamente in base a una codifica determinata. Tuttavia, il testo su cui si esegue la ricerca attraverso un'espressione regolare, viene interpretato a livello di carattere, per cui non è possibile identificare un elemento di collazione in una sottostringa composta da più caratteri. In pratica, un'espressione regolare non riuscirebbe a riconoscere la lettera «ä» nella parola «schaelen».

Alcuni elementi di collazione possono essere classificati come equivalenti. Per esempio, nella lingua italiana le lettere «e», con o senza accento, rappresentano questo tipo di equivalenza. Gli elementi di collazione «equivalenti» costituiscono una ***classe di equivalenza***.

Infine, i caratteri (e non più gli elementi di collazione) possono essere classificati in base a diversi altri tipi di sottoinsiemi, a cui si fa riferimento attraverso dei nomi standard. In generale si tratta di distinguere tra: lettere maiuscole, lettere minuscole, cifre numeriche, cifre alfanumeriche, ecc.

23.1.3 Composizione di un'espressione regolare e corrispondenza

Un'espressione regolare è una stringa di caratteri, i quali, nel caso più semplice, rappresentano esattamente la corrispondenza con la stessa stringa. All'interno di un'espressione regolare possono essere inseriti dei caratteri speciali, per rappresentare delle corrispondenze in situazioni più complesse. Per fare riferimento a tali caratteri in modo letterale, occorre utilizzare delle tecniche di protezione che variano a seconda del contesto.

I caratteri speciali sono tali solo nel contesto per il quale sono stati previsti. Al di fuori di quel contesto possono essere caratteri normali, o caratteri speciali con un significato differente.

La corrispondenza tra un'espressione regolare e una stringa, quando avviene, serve a delimitare una sottostringa che può andare dalla dimensione nulla fino al massimo della stringa di partenza. È importante chiarire che anche la corrispondenza che delimita una stringa nulla può avere significato, in quanto identifica una posizione precisa nella stringa di partenza. In generale, se sono possibili delle corrispondenze differenti, viene presa in considerazione quella che inizia il più a sinistra possibile e si estende il più a destra possibile.

23.1.3.1 Ancoraggio iniziale e finale

In condizioni normali, un'espressione regolare può individuare una sottostringa collocata in qualunque posizione della stringa di partenza. Per indicare espressamente che la corrispondenza deve partire obbligatoriamente dall'inizio della stringa, oppure che deve terminare esattamente alla fine della stringa stessa, si usano due ancore, rappresentate dai caratteri speciali '`^`' e '`$`', ovvero dall'accento circonflesso e dal dollaro.

Per la precisione, un accento circonflesso che si trovi **all'inizio** di un'espressione regolare identifica la sottostringa nulla che si trova idealmente all'inizio della stringa da analizzare; nello stesso modo, un dollaro che si trovi **alla fine** di un'espressione regolare identifica la sottostringa nulla che si trova idealmente alla fine della stringa stessa. Nel caso particolare delle espressioni regolari BRE, i caratteri '`^`' e '`$`' hanno questo significato anche nell'ambito di una sottoespressione, all'inizio o alla fine della stessa. Una sottoespressione è una porzione di espressione regolare delimitata nel modo che viene mostrato in seguito.

Per fare un esempio, l'espressione regolare '`^ini`' corrisponde alla sottostringa «ini» della stringa «inizio». Nello stesso modo, l'espressione regolare '`ini$`' corrisponde alla sottostringa «ini» della stringa «scalini».

Un'espressione regolare può contenere entrambe le ancore di inizio e fine stringa. In tal caso si cerca la corrispondenza con tutta la stringa di partenza.

23.1.3.2 Delimitazione di una o più sottoespressioni

Una sottoespressione è una porzione di espressione regolare individuata attraverso dei delimitatori opportuni. Per la precisione, si tratta di parentesi tonde normali nel caso di espressioni regolari ERE (estese), oppure dei simboli '`\(`' e '`\)`' nel caso di espressioni regolari BRE.

La delimitazione di sottoespressioni può servire per regolare la precedenza nell'interpretazione delle varie parti dell'espressione regolare, oppure per altri scopi che dipendono dal programma in cui vengono utilizzate. In generale, dovrebbe essere ammissibile la definizione di sottoespressioni annidate.

Per fare un esempio, l'espressione regolare BRE '`\(anto\)logia`' corrisponde a una qualunque sottostringa '`antologia`'. Nello stesso modo funziona l'espressione regolare ERE '`(anto)logia`'.

23.1.3.3 Riferimento a una sottoespressione precedente (solo BRE)

Nelle espressioni regolari di tipo BRE è possibile utilizzare la forma '`\`***n***', dove ***n*** è una cifra numerica da uno a nove, per indicare la corrispondenza con l'***n***-esima sottoespressione precedente. Per esempio, l'espressione regolare '`\(sia\) questo \1`' corrisponde alla

sottostringa '**sia questo sia**' di un testo che può essere anche più lungo.

È importante osservare che la corrispondenza della forma '*n*' rappresenta ciò che è stato trovato effettivamente attraverso la sottoespressione, mentre se si volesse semplicemente ripetere lo stesso modello, basterebbe riscriverlo tale e quale.

23.1.3.4 Sottoespressioni alternative (solo ERE)

Esclusivamente nelle espressioni regolari ERE (estese), è possibile indicare la corrispondenza alternativa tra due modelli utilizzando il carattere speciale '|' (la barra verticale). Di solito si utilizza questa possibilità delimitando espressamente le sottoespressioni alternative, in modo da evitare ambiguità, tuttavia questo non dovrebbe essere necessario, dal momento che si tratta di un operatore con un livello molto basso di precedenza.

Per esempio, l'espressione regolare '**((auto)|(dog))matico**' può corrispondere indifferentemente alla sottostringa '**automatico**' oppure '**dogmatico**'.

23.1.3.5 Corrispondenza con un carattere singolo

In un'espressione regolare, qualsiasi carattere che nel contesto non abbia un significato particolare, corrisponde esattamente a se stesso. Il carattere speciale '**.**' (il punto), rappresenta un carattere qualunque, a esclusione di <*NUL*>. Per esempio, l'espressione regolare '**nuo.o**' corrisponde a '**nuoto**', '**nuovo**' e ad altre sottostringhe simili. Per indicare un punto letterale, occorre utilizzare l'espressione '**\.**' (barra obliqua inversa, punto).

È possibile definire anche la corrispondenza con un carattere scelto tra un insieme preciso, utilizzando una notazione speciale, ovvero un'***espressione tra parentesi quadre***:

```
[elenco_corrispondente]
```

```
[^elenco_non_corrispondente]
```

Come si vede dallo schema sintattico, si distinguono due situazioni fondamentali: nel primo caso si definisce un elenco di corrispondenze; nel secondo si ottiene questa definizione indicando un elenco di caratteri che non si vogliono trovare. Si osservi che per negare l'elenco di corrispondenze si utilizza l'accento circonflesso, il quale assume così un significato speciale, differente dall'ancora di inizio già descritta.

L'elenco tra parentesi quadre può essere un elenco puro e semplice di caratteri (lungo a piacere), per cui, per esempio, l'espressione regolare '**piccol[aieo]**' corrisponde indifferentemente alle sottostringhe «piccola», «piccoli», «piccole» e «piccolo». In alternativa può essere rappresentato attraverso uno o più intervalli di caratteri, ma questo implica delle complicazioni che vengono descritte in seguito.

Per negare un elenco, lo si fa precedere da un accento circonflesso. Per esempio, l'espressione regolare '**aiut[^ia]**' può corrispondere alla sottostringa «aiuto» e anche a molte altre, ma non può corrispondere né ad «aiuti», né ad «aiuta».

Dal momento che l'accento circonflesso ha un significato speciale se appare all'inizio di tale contesto, questo può essere usato in modo letterale solo in una posizione più avanzata.

Infine, per indicare una parentesi quadra aperta letterale in un contesto normale, al di fuori delle espressioni tra parentesi quadre, basta l'espressione '**\[**'.

23.1.3.6 Corrispondenze multiple

Alcuni caratteri speciali fungono da operatori che permettono di definire e controllare il ripetersi di un modello riferito a un carattere precedente, a una sottoespressione precedente, oppure a un riferimento all'indietro delle espressioni regolari BRE.

In tutti i tipi di espressione regolare, l'asterisco ('*') corrisponde a nessuna o più ripetizioni di ciò che gli precede. Per esempio, l'espressione regolare '**aiuto***' corrisponde alla sottostringa «aiut», oppure «aiuto», come anche ad «aiutoooooooo», ecc. Inoltre, è il caso di osservare che l'espressione regolare '**.***' corrisponde a qualunque stringa, di qualunque dimensione.

Per indicare un asterisco letterale in un contesto normale, basta farlo precedere da una barra obliqua inversa: '*****'.

Nel caso di espressioni regolari ERE si possono utilizzare anche gli operatori '**+**' e '**?**', per indicare rispettivamente una o più occorrenze dell'elemento precedente, oppure zero o al massimo un'occorrenza di tale elemento. Per esempio, l'espressione regolare '**aiuto+**' corrisponde alla sottostringa «aiuto», oppure «aiutoo», «aiutooooo», ecc., mentre l'espressione regolare '**aiuto?**' può corrispondere alla sottostringa «aiut», oppure «aiuto».

Le espressioni regolari BRE e ERE permettono l'utilizzo di un'altra forma più precisa e generalizzata per esprimere la ripetizione di qualcosa. Nel caso di BRE si usano i modelli seguenti:

```
\{n\}
```

```
\{n,\}
```

```
\{n,m\}
```

Nel caso di ERE si usano forme equivalenti senza le barre oblique inverse:

```
{n}
```

```
{n,}
```

```
{n,m}
```

Si tenga presente che ***n*** rappresenta un numero non negativo, mentre ***m***, se utilizzato, deve essere un numero maggiore di ***n***.

Nella prima delle tre forme, si intende indicare la ripetizione di ***n*** volte esatte l'elemento precedente; nella seconda si intendono almeno ***n*** volte; nella terza si intendono tante ripetizioni da ***n*** a ***m***. In generale, per garantire che un'espressione regolare sia portabile, occorre che il limite massimo rappresentato da ***m*** non superi 255.

23.1.4 Espressioni tra parentesi quadre

Si è accennato all'uso delle espressioni tra parentesi quadre, per indicare la scelta tra un elenco di caratteri, o tra tutti i caratteri esclusi quelli dell'elenco. Un'espressione del genere si traduce sempre nella corrispondenza con un carattere singolo. All'interno di un'espressione del genere, si possono utilizzare forme particolari per indicare un carattere, attraverso un simbolo di collazione, una classe di equivalenza, oppure attraverso una classe di caratteri. È molto importante anche la possibilità di definire degli intervalli, la quale non è ancora stata affrontata nella descrizione precedente di queste espressioni.

23.1.4.1 Corrispondenza con un elemento di collazione

Se si hanno difficoltà a indicare dei caratteri in un'espressione tra parentesi quadre, potrebbe essere opportuno indicarli attraverso l'elemento di collazione corrispondente. Supponendo che nella localizzazione utilizzata esista l'elemento di collazione '**ä**', identificato dal simbolo di collazione '**<a:>**', mancando la possibilità di usare il carattere corrispondente, questo si potrebbe esprimere nella forma '**[.a:.]**'.

In generale, è possibile indicare un carattere singolo all'interno dei delimitatori '**[.**' e '**.]**', come se fosse un elemento di collazione.

Per esempio, '`[.a.]`' è perfettamente uguale all'espressione '`a`'. In questo modo, si può usare la tecnica di rappresentazione degli elementi di collazione quando il contesto rende difficile l'indicazione di qualche carattere.

È necessario ribadire che il simbolo di collazione può apparire solo all'interno di un'espressione tra parentesi quadre. Per fare un esempio pratico, Se ci si trova in una localizzazione adatta, volendo scrivere un'espressione regolare che corrisponda alla sottostringa «schälen», non potendo rappresentare il carattere «ä» si dovrebbe scrivere: '`sch[[.a:.]]len`', dove '`[.a:.]`' si sostituisce al carattere «ä», avendo definito che il simbolo di collazione per questo è '`<a:>`'.

23.1.4.2 Corrispondenza con una classe di equivalenza

«

Nell'ambito della sequenza di collazione della localizzazione che si usa, alcuni elementi possono essere considerati equivalenti ai fini dell'ordinamento. Questi elementi costituiscono una classe di equivalenza. All'interno di un'espressione tra parentesi quadre, per fare riferimento a un elemento qualunque di una certa classe di equivalenza, basta indicare uno di questi tra i delimitatori '`[=`' e '`=]`'. Per esempio, se si suppone che le lettere «e», «è» ed «é», appartengono alla stessa classe di equivalenza, per indicare indifferentemente una di queste, basta la notazione '`[=e=]`'.

Per indicare effettivamente una classe di equivalenza in un'espressione regolare, occorre ricordare che questa va inserita all'interno di un'espressione tra parentesi quadre. In pratica, l'espressione regolare che corrisponde indifferentemente alla stringa «e», «è» o «é», è '`[[=e=]]`'. Si osservi che in alternativa si potrebbe scrivere anche '`[eèé]`'.

23.1.4.3 Corrispondenza con una classe di caratteri

«

Nell'ambito della localizzazione, sono definiti alcuni gruppi di caratteri, attraverso l'uso di parole chiave standard. Per esempio: '`alpha`' definisce l'insieme delle lettere alfabetiche; '`digit`' definisce l'insieme delle cifre numeriche; '`space`' definisce l'insieme dei caratteri che visivamente si traducono in uno spazio di qualche tipo. Oltre a queste, sono definiti dei raggruppamenti, come nel caso di '`alnum`' che indica l'insieme di '`alpha`' e '`digit`'.

All'interno di un'espressione tra parentesi quadre, per indicare una classe di caratteri, si usa il nome riconosciuto dalla localizzazione, racchiuso tra i delimitatori '`[:`' e '`:]`'. Per esempio, per ottenere la corrispondenza con una sottostringa del tipo '**file*n***', dove ***n*** può essere una cifra numerica qualunque, si può utilizzare l'espressione regolare '`file[[:digit:]]`'.

La tabella 23.2 riepiloga i nomi delle classi di caratteri riconosciuti normalmente dalle localizzazioni (si veda anche la pagina di manuale *locale(5)*).

Tabella 23.2. Elenco dei nomi standard attribuiti alle classi di caratteri.

Classe di caratteri	Descrizione
`upper`	Collezione alfabetica delle lettere maiuscole.
`lower`	Collezione alfabetica delle lettere minuscole.
`alpha`	Lettere alfabetiche: di solito l'unione di '`upper`' e '`lower`'.
`digit`	Cifre numeriche.
`alnum`	Cifre alfanumeriche: di solito l'unione di '`alpha`' e '`digit`'.
`punct`	I caratteri di punteggiatura.
`space`	I caratteri definiti come «spazi bianchi» per qualche motivo.
`blank`	Di solito comprende solo '`<space>`' e '`<tab>`'.
`cntrl`	I caratteri di controllo che non possono essere rappresentati.
`graph`	Caratteri grafici: di solito l'unione di '`alnum`' e '`punct`'.
`print`	Caratteri stampabili: di solito l'insieme di '`alnum`', '`punct`' e di '`<space>`'.
`xdigit`	Cifre numeriche e alfabetiche per rappresentare numeri esadecimali.

23.1.4.4 Intervalli di caratteri

«

All'interno di un'espressione tra parentesi quadre, possono apparire degli intervalli di caratteri, includendo eventualmente anche gli elementi di collazione. Al contrario, non si possono usare le classi di equivalenza e nemmeno le classi di caratteri per indicare degli intervalli, perché non si traducono in un carattere preciso nell'ambito della codifica. La forma per esprimere un intervallo è la seguente:

inizio*-*fine

Questo lascia intendere che il trattino ('`-`') abbia un significato particolare all'interno di un'espressione tra parentesi quadre. Per fare un esempio molto semplice, l'espressione regolare '`[a-d]`' rappresenta un carattere compreso tra «a» e «d», in base alla localizzazione.

Gli intervalli si possono mescolare con gli elenchi e anche con altri intervalli. Per esempio, l'espressione regolare '`[a-dhi]`' individua un carattere compreso tra «a» e «d», oppure anche «h» o «i».

Possono essere aggregati più elenchi assieme, ma tutti questi devono avere un inizio e una fine indipendente. Per esempio, l'espressione regolare '`[a-cg-z]`' rappresenta due intervalli, rispettivamente tra «a» e «c», e tra «g» e «z». Al contrario, l'espressione regolare '`[a-c-z]`' indica l'intervallo da «a» a «c», oppure il trattino (perché è fuori dal contesto previsto per indicare un intervallo), oppure «z».

> Quando si indicano degli intervalli occorre prestare attenzione alla configurazione locale per sapere esattamente cosa viene coinvolto. In generale, per questo motivo, le espressioni regolari che contengono espressioni tra parentesi quadre con l'indicazioni di intervalli, non sono portabili da un sistema all'altro.

23.1.4.5 Protezione all'interno di espressioni tra parentesi quadre

«

Dal momento che in un'espressione tra parentesi quadre i caratteri '`^`', '`-`' e '`]`', hanno un significato speciale, per poterli utilizzare, occorrono degli accorgimenti: se si vuole usare l'accento circonflesso in modo letterale, è necessario che questo non sia il primo; per indicare il trattino si può descrivere un intervallo, in cui sia posto come carattere iniziale o finale. In alternativa, i caratteri che non si riescono a indicare (come le parentesi quadre), possono essere racchiuse attraverso i delimitatori dei simboli di collazione: '`[.[.]`' e '`[.].]`' per le parentesi e '`[.-.]`' per un trattino.

> All'interno di un'espressione tra parentesi quadre, i caratteri che sono speciali al di fuori di questo contesto, qui perdono il loro significato particolare (come nel caso del punto e dell'asterisco ('`*`'), oppure ne acquistano uno nuovo (come nel caso dell'accento circonflesso).

23.1.5 Precedenze

«

Dopo la difficoltà che si affronta per comprendere il funzionamento delle espressioni regolari, l'ordine in cui le varie parti di queste vengono risolte, dovrebbe essere abbastanza intuitivo. La tabella 23.3 riassume questa sequenza, distinguendo tra espressioni BRE e ERE.

Tabella 23.3. Ordine di precedenza, dal più alto al più basso.

<table>
<tr><th>Tipo di componente l' espressione</th><th>Operatore BRE</th><th>Operatore ERE</th></tr>
<tr><td>Contenuto delle espressioni tra parentesi quadre.</td><td>[= =]
[: :]
[. .]</td><td>[= =]
[: :]
[. .]</td></tr>
<tr><td>Caratteri speciali resi letterali.</td><td>\carattere_speciale</td><td>\carattere_speciale</td></tr>
<tr><td>Espressioni tra parentesi quadre.</td><td>[ ]</td><td>[ ]</td></tr>
<tr><td>Sottoespressioni e riferimenti all'indietro (BRE).</td><td>\(\)
\n</td><td></td></tr>
<tr><td>Raggruppamenti (ERE).</td><td></td><td>( )</td></tr>
<tr><td>Ripetizioni.</td><td>*
\{m,n\}</td><td>*
+
?
{m,n}</td></tr>
<tr><td>Concatenamento di espressioni (non si usano simboli).</td><td></td><td></td></tr>
<tr><td>Ancore iniziali e finali.</td><td>^
$</td><td>^
$</td></tr>
<tr><td>Alternanza (solo ERE).</td><td></td><td>|</td></tr>
</table>

23.2 Confronto sintetico tra le espressioni regolari «reali»

Date le diversità notevoli tra tutti i tipi di espressione regolare che si utilizzano in pratica con i programmi che ne fanno uso, vale la pena di riepilogare le differenze fondamentali tra lo standard POSIX e le realtà più importanti. In questo capitolo si raccolgono solo alcune tabelle di comparazione, che mostrano l'abbinamento tra diversi modelli di espressione compatibili. Le descrizioni sono scarne, tuttavia quello che si vede dovrebbe servire per collegare le cose, permettendo di comprendere quali sono le estensioni di ogni realizzazione.

Tabella 23.4. Confronto tra gli operatori fondamentali.

<table>
<tr><th></th><th>BRE POSIX</th><th>ERE POSIX</th><th>BRE GNU</th><th>ERE GNU</th><th>Perl</th></tr>
<tr><td>escape</td><td>\</td><td>\</td><td>\</td><td>\</td><td>\</td></tr>
<tr><td>ancora iniziale</td><td>^</td><td>^</td><td>^</td><td>^</td><td>^</td></tr>
<tr><td>ancora finale</td><td>$</td><td>$</td><td>$</td><td>$</td><td>$</td></tr>
<tr><td>alternativa</td><td></td><td>|</td><td>\|</td><td>|</td><td>|</td></tr>
<tr><td>raggruppamento</td><td>\(\)</td><td>( )</td><td>\(\)</td><td>( )</td><td>( )</td></tr>
<tr><td>elenco</td><td>[ ]</td><td>[ ]</td><td>[ ]</td><td>[ ]</td><td>[ ]</td></tr>
<tr><td>riferimento</td><td>\n</td><td></td><td>\n</td><td>\n</td><td>\n</td></tr>
</table>

Tabella 23.5. Confronto tra gli operatori interni alle espressioni tra parentesi quadre.

	BRE POSIX	ERE POSIX	BRE GNU	ERE GNU	Perl
sequenze	*xy*...	*xy*...	*xy*...	*xy*...	*xy*...
intervalli	*x*-*y*	*x*-*y*	*x*-*y*	*x*-*y*	*x*-*y*
elementi di collazione	[. .]	[. .]			
caratteri equivalenti	[= =]	[= =]			
classi di caratteri	[: :]	[: :]	[: :]	[: :]	[: :]

Tabella 23.6. Simboli speciali.

	BRE POSIX	ERE POSIX	BRE GNU	ERE GNU	Perl
	[[:alnum:]_] [[:word:]]	[[:alnum:]_] [[:word:]]	\w	\w	\w [[:alnum:]_] [[:word:]]
	[^[:alnum:]_] [^[:word:]]	[^[:alnum:]_] [^[:word:]]	\W	\W	\W [^[:alnum:]_] [^[:word:]]
inizio di parola			\<	\<	
fine di parola			\>	\>	
inizio o fine parola			\b	\b	\b
interno di una parola			\B	\B	\B
	[[:space:]]	[[:space:]]	[[:space:]]	[[:space:]]	\s [[:space:]]
	[^[:space:]]	[^[:space:]]	[^[:space:]]	[^[:space:]]	\S [^[:space:]]
	[[:digit:]]	[[:digit:]]	[[:digit:]]	[[:digit:]]	\d [[:digit:]]
	[^[:digit:]]	[^[:digit:]]	[^[:digit:]]	[^[:digit:]]	\D [^[:digit:]]

Tabella 23.7. Operatori di ripetizione.

	BRE POSIX	ERE POSIX	BRE GNU	ERE GNU	Perl
	x*	x*	x*	x*	x*
il minimo di x*					x*?
		x?	x\?	x?	x?
il minimo di x?					x??
		x+	x\+	x+	x+
il minimo di x+					x+?
	x\{n\}	x{n}	x\{n\}	x{n}	x{n}
	x\{n,\}	x{n,}	x\{n,\}	x{n,}	x{n,}
il minimo di x{n,}					x{n,}?
	x\{n,m\}	x{n,m}	x\{n,m\}	x{n,m}	x{n,m}
il minimo di x{n,m}					x{n,m}?

In generale, si può osservare che i programmi GNU e Perl non permettono l'indicazione di simboli di collazione e nemmeno di classi di equivalenza.

A differenza di ciò che si vede di solito, Perl introduce un concetto nuovo: la corrispondenza minima di un'espressione regolare. Questo può essere molto importante in Perl, quando si delimitano delle sottoespressioni per estrapolare delle parti differenti di una stringa.

23.3 Grep

«

Il programma Grep[1] esegue una ricerca all'interno dei file in base a un modello espresso normalmente in forma di espressione regolare. Storicamente sono esistite tre versioni di questo programma: '**grep**', '**egrep**' e '**fgrep**', ognuna specializzata in un tipo di ricerca. Attualmente, Grep comprende tutte queste funzionalità in un solo programma; tuttavia, in alcuni casi, per mantenere la compatibilità con il passato, possono trovarsi sistemi che mettono a disposizione anche i programmi '**egrep**' e '**fgrep**' in forma originale.

L'eseguibile '**grep**' compie una ricerca all'interno dei file indicati come argomento oppure all'interno dello standard input:

```
grep [opzioni] modello [file...]
```

```
grep [opzioni] -e modello [file...]
```

```
grep [opzioni] -f file_modello [file...]
```

Il modello di ricerca può essere semplicemente il primo degli argomenti che seguono le opzioni, oppure può essere indicato precisamente come argomento dell'opzione '**-e**', oppure ancora può essere contenuto in un file che viene indicato attraverso l'opzione '**-f**'. La tabella 23.8 elenca le opzioni principali.

Tabella 23.8. Opzioni principali di '**grep**'.

Opzione	Descrizione
`-G`	Utilizza un'espressione regolare elementare (comportamento predefinito).
`-E`	Utilizza un'espressione regolare estesa. Se si avvia il programma con il nome '**egrep**', è come utilizzare l'eseguibile '**grep**' con l'opzione '**-E**'.
`-F`	Utilizza un modello fatto di stringhe fisse. Se si avvia il programma con il nome '**fgrep**', è come utilizzare l'eseguibile '**grep**' con l'opzione '**-F**'.
`-e` *modello*	Specifica il modello di ricerca.
`-f` *file*	Specifica il nome di un file contenente il modello di ricerca.
`-i`	Ignora la differenza tra maiuscole e minuscole.
`-n`	Aggiunge il numero di riga.
`-c`	Emette solo il totale delle righe corrispondenti per ogni file.
`-h`	Elimina l'intestazione normale per le ricerche su più file.
`-l`	Emette solo i nomi dei file per i quali la ricerca ha avuto successo.
`-L`	Emette solo i nomi dei file per i quali la ricerca non ha avuto successo.
`-v`	Inverte il senso della ricerca: valgono le righe che non corrispondono.

Segue la descrizione di alcuni esempi.

- `$ grep -F -e ciao -i -n *` [*Invio*]

  Cerca all'interno di tutti i file contenuti nella directory corrente la corrispondenza della parola «ciao» senza considerare la differenza tra le lettere maiuscole e quelle minuscole. Visualizza il numero e il contenuto delle righe che contengono la parola cercata.

- `$ grep -E -e "scal[oa]" elenco` [*Invio*]

  Cerca all'interno del file '`elenco`' le righe contenenti la parola «scalo» o «scala».

- `$ grep -E -e \'.*\' elenco` [*Invio*]

  Questo è un caso di ricerca particolare in cui si vogliono cercare le righe in cui appare qualcosa racchiuso tra apici singoli, nel modo '**'...'**'. Si immagina però di utilizzare la shell standard con la quale è necessario proteggere gli apici da un altro tipo di interpretazione. In questo caso la shell fornisce a '**grep**' solo la stringa '**'.*'**'.

- `$ grep -E -e "\'.*\'" elenco` [*Invio*]

  Questo esempio deriva dal precedente. Anche in questo caso si suppone di utilizzare la shell standard, ma questa volta viene fornita a '**grep**' la stringa '**\'.*\'**' che fortunatamente viene interpretata ugualmente da '**grep**' nel modo corretto.

23.3.1 Grep e Gzip integrati

«

Normalmente, i programmi eseguibili (o i collegamenti) che compongono Grep, sono accompagnati da una serie di script che facilitano le ricerche all'interno di file compressi con Gzip o con Compress:

```
zgrep [opzioni] modello [file...]
```

```
zegrep [opzioni] modello [file...]
```

```
zfgrep [opzioni] modello [file...]
```

L'utilizzo è equivalente ai programmi '**grep**', '**egrep**' e '**fgrep**', con la differenza che si intendono scandire file compressi. Di solito, tali script funzionano correttamente anche se in realtà i file da analizzare non sono compressi affatto.

23.4 Find

«

Il programma Find[2] esegue una ricerca, all'interno di uno o più percorsi, per i file che soddisfano delle condizioni determinate, legate alla loro apparenza esterna e non al loro contenuto. Per ogni file o directory trovati, può essere eseguito un comando (programma, script o altro) che a sua volta può svolgere delle operazioni su di essi.

Qui non vengono descritte tutte le funzionalità di Find: una volta appresi i rudimenti del suo funzionamento, conviene consultare il documento *info find* oppure la pagina di manuale *find(1)*.

Find si compone in pratica dell'eseguibile '**find**', il quale ha una sintassi piuttosto insolita, oltre che complessa, anche se dallo schema seguente non sembrerebbe così. È anche indispensabile tenere a mente che molti dei simboli utilizzati negli argomenti dell'eseguibile '**find**' potrebbero essere interpretati e trasformati dalla shell, di conseguenza può essere necessario utilizzare le tecniche che la shell stessa offre per evitarlo.

```
find [percorso...] [espressione]
```

Find esegue una ricerca all'interno dei percorsi indicati per i file che soddisfano l'espressione di ricerca. Il primo argomento che inizia con '**-**', '**(**', '**)**', '**,**' o '**!**' (trattino, parentesi tonda, virgola, punto esclamativo) viene considerato come l'inizio dell'espressione, mentre gli argomenti precedenti sono interpretati come parte dell'insieme dei percorsi di ricerca.

Se non vengono specificati percorsi di ricerca, si intende la directory corrente; se non viene specificata alcuna espressione, o semplicemente se non viene specificato nulla in contrario, viene emesso l'elenco dei nomi trovati.

Il concetto di espressione nella documentazione di Find è piuttosto ampio e bisogna fare un po' di attenzione. Si può scomporre idealmente nello schema seguente:

```
[opzione...] [condizioni]
```

A loro volta, le condizioni possono essere di due tipi: test e azioni. Ma, mentre le opzioni devono apparire prima, test e azioni possono essere mescolati tra loro.

Le opzioni rappresentano un modo di configurare il funzionamento del programma, così come di solito accade nei programmi di servizio. Le condizioni sono espressioni che generano un risultato logico e come tali vanno trattate: per concatenare insieme più condizioni occorre utilizzare gli operatori booleani.

Tabella 23.9. Alcune opzioni importanti. Le opzioni si collocano prima delle condizioni (test e azioni).

Opzione	Descrizione
`-depth`	Elabora prima il contenuto delle directory. In pratica si ottiene una scansione che parte dal livello più profondo fino al più esterno.
`-xdev` `-mount`	Non esegue la ricerca nelle directory contenute all'interno di file system differenti da quello di partenza. Tra i due è preferibile usare '**-xdev**'.
`-noleaf`	Non ottimizza la ricerca. Questa opzione è necessaria quando si effettuano ricerche all'interno di file system che non seguono le convenzioni Unix, come nel caso di CD-ROM senza le estensioni necessario, o partizioni Dos.

L'esempio seguente elenca tutti i file e le directory a partire dalla posizione corrente restando nell'ambito del file system di partenza:

$ **find . -xdev -print** [*Invio*]

Come già accennato, i test sono condizioni che vengono valutate per ogni file e directory incontrati. Il risultato delle condizioni può essere *Vero* o *Falso*. Quando vengono indicate più condizioni, queste devono essere unite in qualche modo attraverso degli operatori booleani per ottenere una sola grande condizione. Se non viene specificato diversamente, viene utilizzato automaticamente l'operatore AND: '**-a**'.

All'interno dei test, gli argomenti numerici possono essere preceduti o meno da un segno:

+*n*	Indica un numero maggiore di *n*.
-*n*	Indica un numero minore di *n*.
n	Indica un numero esattamente uguale a *n*.

Nelle tabelle successive, vengono raggruppati per tipo alcuni testi disponibili.

Tabella 23.11. Test sulla proprietà dei file o delle directory.

Test	Descrizione
`-uid` *n*	Si avvera quando il numero UID (utente) del file o della directory è uguale a *n*.
`-user` *nome_dell'utente*	Si avvera quando il file o la directory appartiene all'utente indicato.
`-nouser`	Si avvera per i file e le directory di proprietà di utenti non esistenti.
`-gid` *n*	Si avvera quando il numero GID (gruppo) del file o della directory è uguale a *n*.
`-group` *nome_del_gruppo*	Si avvera quando il file o la directory appartiene al gruppo indicato.
`-nogroup`	Si avvera per i file e le directory di proprietà di gruppi non esistenti.

Tabella 23.12. Test sui permessi dei file o delle directory.

Test	Descrizione
`-perm` *permessi*	Si avvera quando i permessi del file o della directory corrispondono esattamente a quelli indicati con questo test. I permessi si possono indicare in modo numerico (ottale) o simbolico.
`-perm` -*permessi*	Si avvera quando i permessi del file o della directory comprendono almeno quelli indicati con questo test.
`-perm` +*permessi*	Si avvera quando alcuni dei permessi indicati nel modello di questo test corrispondono a quelli del file o della directory.

Tabella 23.13. Test sulle caratteristiche dei nomi dei file o delle directory.

Test	Descrizione
`-name` *modello*	Si avvera quando viene incontrato un nome di file o directory corrispondente al modello indicato, all'interno del quale si possono utilizzare i caratteri jolly. La comparazione avviene utilizzando solo il nome del file (o della directory) escludendo il percorso precedente. I caratteri jolly ('*****', '**?**', '**[**', '**]**') non possono corrispondere al punto iniziale ('**.**') che appare nei cosiddetti file nascosti.
`-iname` *modello*	Si comporta come '**-name**', ma non tiene conto della differenza tra maiuscole e minuscole ('**iname**' = *insensitive* '**name**').
`-lname` *modello*	Si avvera quando si tratta di un collegamento simbolico e il suo contenuto corrisponde al modello che può essere espresso utilizzando anche i caratteri jolly. Un collegamento simbolico può contenere anche l'indicazione del percorso necessario a raggiungere un file o una directory reale. Il modello espresso attraverso i caratteri jolly non tiene conto in modo particolare dei simboli punto ('**.**') e barra obliqua ('**/**') che possono essere contenuti all'interno del collegamento.
`-ilname` *modello*	Si comporta come '**-lname**', ma non tiene conto della differenza tra maiuscole e minuscole ('**ilname**' = *insensitive* '**lname**').
`-path` *modello*	Si avvera quando il modello, esprimibile utilizzando caratteri jolly, corrisponde a un percorso. Per esempio, un modello del tipo '**./i*no**' può corrispondere al file '`./idrogeno/ossigeno`'.
`-ipath` *modello*	Si comporta come '**-path**', ma non tiene conto della differenza tra maiuscole e minuscole ('**ipath**' = *insensitive* '**path**').
`-regexp` *modello*	Si avvera quando l'espressione regolare indicata corrisponde al file o alla directory incontrati. Per la verifica della corrispondenza, attraverso l'espressione regolare, viene utilizzato anche il percorso e non solo il nome del file o della directory. Quindi, per ottenere la corrispondenza con il file '`./carbonio`' si può utilizzare l'espressione regolare '**.*bonio**' oppure '**.*bo..o**', ma non '**c.*io**'.
`-iregexp` *modello*	Si comporta come '**-regexp**', ma non tiene conto della differenza tra maiuscole e minuscole ('**iregexp**' = *insensitive* '**regexp**').

Tabella 23.14. Test sulla data di modifica dei file o delle directory.

Test	Descrizione
`-mmin` *n*	Si avvera quando la data di modifica del file o della directory corrisponde a *n* minuti fa.
`-mtime` *n*	Si avvera quando la data di modifica del file o della directory corrisponde a *n* giorni fa. Più precisamente, il valore *n* fa riferimento a multipli di 24 ore.
`-newer` *file*	Si avvera quando la data di modifica del file o della directory è più recente di quella del file indicato.

Tabella 23.15. Test sulla data di accesso ai file o alle directory.

Test	Descrizione
`-amin` ***n***	Si avvera quando la data di accesso del file o della directory corrisponde a ***n*** minuti fa.
`-atime` ***n***	Si avvera quando la data di accesso del file o della directory corrisponde a ***n*** giorni fa. Più precisamente, il valore ***n*** fa riferimento a multipli di 24 ore.
`-anewer` ***file***	Si avvera quando la data di accesso del file o della directory è più recente di quella del file indicato.

Tabella 23.16. Test sulla data di creazione o del cambiamento di stato dei file o delle directory.

Test	Descrizione
`-cmin` ***n***	Si avvera quando la data di creazione del file o della directory corrisponde a ***n*** minuti fa.
`-ctime` ***n***	Si avvera quando la data di creazione del file o della directory corrisponde a ***n*** giorni fa. Più precisamente, il valore ***n*** fa riferimento a multipli di 24 ore.
`-cnewer` ***file***	Si avvera quando la data di creazione del file o della directory è più recente di quella del file indicato.

Tabella 23.17. Test sulla dimensione dei file o delle directory.

Test	Descrizione
`-empty`	Si avvera quando il file o la directory sono vuoti.
`-size` ***n***[b|c|k|w]	Si avvera quando la dimensione del file o della directory ha una dimensione pari a ***n***. L'unità di misura è rappresentata dalla lettera che segue il numero: '**b**' blocchi da 512 byte e rappresenta il valore predefinito in mancanza dell'indicazione di questa lettera; '**c**' byte (caratteri); '**k**' blocchi da 1 024 byte (1 Kibyte); '**w**' parole di 2 byte.

Tabella 23.18. Test vari.

Test	Descrizione
`-true`	Sempre vero.
`-false`	Sempre falso.
`-fstype` ***tipo_di_file_system***	Si avvera quando il file o la directory si trova in un file system del tipo indicato (vedere tabella 19.61).
`-inum` ***n***	Si avvera quando il file o la directory ha il numero di inode corrispondente a ***n***.
`-type` ***categoria***	Si avvera se l'elemento analizzato appartiene alla categoria indicata: '**b**' dispositivo a blocchi; '**c**' dispositivo a caratteri; '**d**' directory; '**p**' file FIFO (*pipe* con nome); '**f**' file normale (*regular file*); '**l**' collegamento simbolico; '**s**' socket.

Le condizioni possono essere costruite anche utilizzando alcuni operatori booleani e le parentesi. Quando questi componenti vengono utilizzati, la valutazione delle condizione viene fatta eseguendo il minimo numero indispensabile di operazioni. Ciò significa che di fronte a un operatore AND si verifica la prima condizione e solo se questa risulta vera si passa a verificare la seconda; di fronte a un operatore OR si verifica la prima condizione e solo se questa risulta falsa si passa a verificare la seconda. Infatti, per sapere che il risultato di un'operazione AND è *Falso* basta sapere che almeno una delle due condizioni in ingresso ha un valore *Falso*; per sapere che il risultato di un'operazione OR è *Vero* basta sapere che almeno una delle due

condizioni in ingresso ha il valore *Vero*.

Tabella 23.19. Operatori booleani.

Operatore	Descrizione
`(  )`	Le parentesi tonde stabiliscono la precedenza nell'esecuzione dei test.
`!` `-not`	Davanti a un'espressione si comporta come negazione logica, ovvero è equivalente a NOT.
`-a` `-and`	Tra due espressioni si comporta come l'operatore logico AND. In mancanza dell'indicazione di un operatore logico tra due condizioni si intende AND.
`-o` `-or`	Tra due espressioni si comporta come l'operatore logico OR.

Le azioni sono delle operazioni da compiere per ogni file o directory che si ottiene dalla scansione. Queste azioni generano però un risultato che viene interpretato in maniera logica. Dipende da come vengono concatenate le varie condizioni (test e azioni) se, e in corrispondenza di quanti file, vengono eseguite queste azioni.

Tabella 23.20. Alcune azioni.

Azione	Descrizione
`-exec` ***comando*** `;`	Esegue il comando indicato, nella directory di partenza, restituendo il valore *Vero* se il comando restituisce il valore zero. Tutti gli argomenti che seguono vengono considerati come parte del comando fino a quando viene incontrato il simbolo punto e virgola ('**;**'). All'interno del comando, la stringa '**{}**' viene interpretata come sinonimo del file che è attualmente in corso di elaborazione. Se la shell interpreta questo simbolo occorre utilizzare il meccanismo della protezione per evitarlo.
`-ok` ***comando*** `;`	Si comporta come '**-exec**' ma, prima di eseguire il comando, chiede conferma all'utente. Se il comando non viene eseguito, restituisce il valore *Falso*.
`-print`	Si avvera sempre ed emette il nome completo dei file (e delle directory) che avverano l'insieme delle condizioni. È l'azione predefinita, se non ne vengono indicate delle altre. Il programma Find di GNU considera questa l'azione predefinita, per cui non è necessario indicarla per ottenere un output sullo schermo. Generalmente, Find non ha un'azione predefinita e questo può mettere in crisi un utente di un sistema GNU quando passa a un altro sistema Unix. Questo è il motivo per il quale viene sempre indicata l'azione negli esempi seguenti, anche se non sarebbe necessario.

Segue la descrizione di alcuni esempi.

- `$` **`find / -name "lib*" -print`**[*Invio*]

  Esegue una ricerca su tutto il file system globale, a partire dalla directory radice, per i file e le directory il cui nome inizia per '**lib**'. Dal momento che si vuole evitare che la shell trasformi '**lib***' in qualcosa di diverso, si utilizzano le virgolette.

- `#` **`find / -xdev -nouser -print`**[*Invio*]

  Esegue una ricerca nel file system principale a partire dalla directory radice, escludendo gli altri file system, per i file e le directory appartenenti a utenti non registrati (che non risultano da '`/etc/passwd`').

- `$` **`find /usr -xdev -atime +90 -print`**[*Invio*]

Esegue una ricerca a partire dalla directory '/usr/', escludendo altri file system diversi da quello di partenza, per i file la cui data di accesso è più vecchia di 2160 ore (24 * 90 = 2160).

- `$ find / -xdev -type f -name core -print` [*Invio*]

  Esegue una ricerca a partire dalla directory radice, all'interno del solo file system principale, per i file '`core`' (solo i file normali).

- `$ find / -xdev -size +5000k -print` [*Invio*]

  Esegue una ricerca a partire dalla directory radice, all'interno del solo file system principale, per i file la cui dimensione supera i 5000 Kibyte.

- `$ find ~/dati -atime +90 -exec mv \{\} ~/archivio \;` [*Invio*]

  Esegue una ricerca a partire dalla directory '`~/dati/`' per i file la cui data di accesso è più vecchia di 90 giorni, spostando quei file all'interno della directory '`~/archivio/`'. Il tipo di shell a disposizione ha costretto a usare spesso il carattere di escape ('`\`') per poter usare le parentesi graffe e il punto e virgola secondo il significato che gli attribuisce Find.

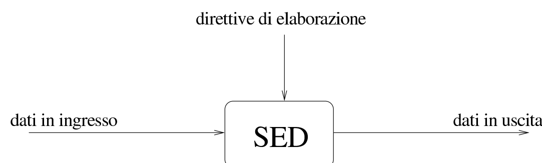
23.5 SED

SED[3] è un programma in grado di eseguire delle trasformazioni elementari in un flusso di dati di ingresso, proveniente indifferentemente da un file o da un condotto. Questo flusso di dati viene letto sequenzialmente e la sua trasformazione viene restituita attraverso lo standard output.

Il nome è l'abbreviazione di *Stream oriented editor*, che descrive istantaneamente il senso di questo programma: *editor* di flusso. Volendo usare altri termini, lo si potrebbe definire come un programma per la modifica sequenziale di un flusso di dati espressi in forma testuale.

Volendo vedere SED come una scatola nera, lo si può immaginare come un oggetto che ha due ingressi: un flusso di dati in ingresso, composto da uno o più file di testo concatenati assieme; un flusso di istruzioni in ingresso, che compone il programma dell'elaborazione da apportare ai dati; un flusso di dati in uscita che rappresenta il risultato dell'elaborazione.

Figura 23.21. Flussi di dati che interessano SED.



In linea di principio, SED non consente di indicare dei caratteri speciali nei suoi comandi attraverso delle sequenze di escape.

23.5.1 Avvio dell'eseguibile

SED è costituito in pratica dall'eseguibile '**`sed`**', il quale interpreta un programma scritto in un linguaggio apposito, che gli viene fornito come argomento della riga di comando, o in un file.

`sed` [*opzioni*] [*programma_di_elaborazione*] [*file*...]

Il testo del programma, o il nome del file che lo contiene, può essere indicato attraverso delle opzioni adatte, oppure, in loro mancanza, può essere indicato come primo degli argomenti che seguono le opzioni. Alla fine possono essere indicati i file da elaborare e in loro mancanza si usa lo standard input.

Opzione	Descrizione
`-e` *istruzioni* `--expression=`*istruzioni*	Questa opzione, che può essere utilizzata anche più volte, permette di specificare delle istruzioni SED che si aggiungono alle altre eventualmente già indicate.
`-f` *file_delle_istruzioni* `--file` *file_delle_istruzioni*	Questa opzione permette di indicare un file contenente una serie di istruzioni SED. Anche questa opzione può essere usata più volte, aggiungendo ogni volta altre istruzioni al programma globale.

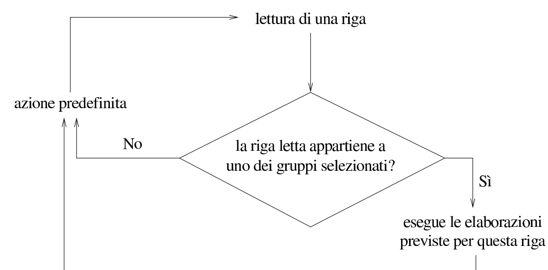
`-n`
`--quiet`
`--silent`
	In condizioni normali, alla fine di ogni ciclo, SED emette il contenuto di quello che viene definito come *pattern space*. In pratica, ogni riga letta ed elaborata viene emessa attraverso lo standard output senza bisogno di un comando apposito. Utilizzando questa opzione, si fa in modo di evitare tale comportamento, così che il programma di elaborazione interpretato da SED deve ordinare quando emettere ogni riga.

23.5.2 Logica di funzionamento

Il primo compito di SED, una volta avviato, è quello di raccogliere tutto ciò che deve andare a comporre il programma di elaborazione: può trattarsi di direttive fornite singolarmente attraverso l'opzione '**`-e`**' e di gruppi di direttive fornite all'interno di file appositi, indicati attraverso l'opzione '**`-f`**'. In particolare, SED si prende cura di mantenerne intatto l'ordine. Successivamente, concatena i dati in ingresso secondo la sequenza indicata dei file posti alla fine della riga di comando, oppure utilizza direttamente lo standard input.

Lo schema che appare nella figura 23.23 si avvicina all'idea del funzionamento di SED: il flusso in ingresso viene letto sequenzialmente, una riga alla volta; ogni volta la riga viene messa in un'area transitoria, nota come *pattern space*; viene confrontata la riga con ogni direttiva del programma di elaborazione e se nessuna di queste direttive coincide, la riga non viene elaborata, compiendo semplicemente l'azione predefinita prima di passare al prossimo ciclo di lettura. Se una o più direttive del programma di elaborazione corrispondono alla riga, vengono eseguite sequenzialmente le elaborazioni previste; poi, alla fine, si passa comunque per l'esecuzione dell'azione predefinita.

Figura 23.23. Struttura semplificata del funzionamento di SED.



L'azione predefinita di SED è l'emissione del contenuto dell'area transitoria, per cui, se non venisse fornita alcuna direttiva a SED, si otterrebbe almeno la riemissione completa dello stesso file ricevuto in ingresso:

`$ sed "" pippo.txt` [*Invio*]

L'esempio mostra proprio l'avvio dell'eseguibile '**`sed`**' allo scopo di interpretare una direttiva nulla, fornendo il file '`pippo.txt`' in ingresso. Il risultato è la riemissione del contenuto di questo file attraverso lo standard output.

Per impedire che questa azione si compia automaticamente, si utilizza l'opzione '**`-n`**' (ovvero '**`--quiet`**' o '**`--silent`**'). In questo modo, è compito delle direttive del programma di elaborazione il richiedere espressamente l'emissione della riga elaborata.

SED dispone di due aree transitorie per le elaborazioni: una che contiene la riga letta, come già descritto; l'altra, definita come *hold space*, viene gestita eventualmente attraverso le direttive del programma di elaborazione interpretato da SED. L'utilizzo di questa seconda area di memoria non viene mostrato qui.

Dal momento che SED è un programma storico dei sistemi Unix, è bene tenere presente le limitazioni che potrebbe avere in questo o quel sistema. In particolare, qualche realizzazione di SED potrebbe porre un limite alla dimensione delle righe. Questo fatto va tenuto presente quando si vogliono realizzare dei programmi «portabili», ovvero, da usare su piattaforme diverse, con sistemi operativi diversi.

23.5.3 Script e direttive multiple

Di solito, si vede utilizzare SED con direttive fornite direttamente attraverso la stessa riga di comando. Volendo realizzare un programmino un po' più complesso, si potrebbe scrivere uno script che deve essere interpretato da SED. Per farlo, occorre iniziare il file in questione con una delle due intestazioni seguenti:

```
#!/bin/sed -f
```

```
#!/bin/sed -nf
```

Nel primo caso, si fa in modo di fornire all'eseguibile '**sed**' (si suppone che si trovi nella directory '/bin/') l'opzione '**-f**', in modo che il file stesso venga inteso correttamente come un programma di elaborazione; nel secondo, oltre a questo, viene aggiunta l'opzione '**-n**', con la quale si inibisce l'emissione predefinita delle righe dopo ogni ciclo di elaborazione.

È utile osservare che in uno script del genere non è possibile fare riferimento alle variabili di ambiente.

Per quanto riguarda le direttive contenute nei file, queste utilizzano una riga per ognuna, dove le righe bianche o vuote vengono ignorate, assieme ai commenti che iniziano con il simbolo '**#**':

```
direttiva_di_elaborazione
direttiva_di_elaborazione
...
```

Le direttive fornite attraverso la riga di comando sono solitamente istruzioni singole; per cui, volendo aggiungerne delle altre, si utilizzano più opzioni '**-e**':

```
sed -e direttiva_di_elaborazione [-e direttiva_di_elaborazione]
... file_in_ingresso...
```

Tuttavia, di solito è possibile indicare più direttive con una sola opzione '**-e**', separandole con un punto e virgola:

```
sed -e direttiva_di_elaborazione[;direttiva_di_elaborazione]
... file_in_ingresso...
```

L'uso di più direttive nella riga di comando, con o senza il punto e virgola, è sconsigliabile in generale, dal momento che dovendo scrivere un programma di elaborazione complesso è preferibile usare un file, trasformandolo eventualmente in uno script come è stato mostrato all'inizio di questa sezione.

23.5.4 Direttive

Ogni direttiva di un programma di elaborazione SED fa riferimento, esplicitamente o implicitamente, a un gruppo di righe, identificate in qualche modo, a cui vengono applicati dei comandi.

```
[selezione_righe]comando
```


Il modello sintattico mostra l'indicazione di un comando dopo la selezione delle righe; questo comando può essere un raggruppamento di comandi, indicato all'interno di parentesi graffe.

La selezione delle righe per una direttiva SED è il primo elemento importante per queste. La mancanza dell'indicazione di questa selezione rappresenta implicitamente la selezione di tutte le righe.

È importante osservare che le righe possono essere indicate anche attraverso la corrispondenza con un'espressione regolare, la quale non deve essere confusa con i comandi che a loro volta possono avere a che fare con altre espressioni regolari.

Inoltre, è necessario ricordare che SED numera le righe a partire dalla prima del primo file, continuando fino alla fine dell'ultimo file, senza interrompere la numerazione.

Tabella 23.26. Selezione delle righe.

Direttiva	Descrizione
n	Un numero puro e semplice, indica precisamente la riga *n*-esima.
$	Un dollaro rappresenta l'ultima riga dell'ultimo file.
/*bre*/	Un'espressione regolare elementare (BRE), racchiusa tra due barre oblique normali, serve a selezionare tutte le righe per cui corrisponde questo modello. Dal momento che la barra obliqua viene usata come delimitatore, se questa deve essere inserita nel modello, occorre proteggerla con una barra obliqua inversa ('\/').
xbrex	Si tratta sempre della selezione delle righe in base alla corrispondenza con un'espressione regolare, con la differenza che questa viene delimitata con un carattere differente, *x*, scelto liberamente, in modo da non interferire con i simboli usati nel modello. Se il modello dell'espressione regolare dovesse contenere anche questo carattere usato per la delimitazione, potrebbe essere protetto con l'aggiunta della barra obliqua inversa all'inizio ('*x*').
riga_iniziale,*riga_finale*	È possibile indicare un intervallo di righe, unendo assieme due riferimenti a righe, sia in forma numerica, sia attraverso le espressioni regolari. Per quanto riguarda l'individuazione della prima riga dell'intervallo, la cosa è abbastanza semplice; in particolare, se si tratta di un'espressione regolare, la prima corrispondenza indica la prima riga. Più complicato è il modo in cui viene preso in considerazione il secondo modello: se si tratta di un numero, questo rappresenta l'*n*-esima riga da raggiungere, che deve essere considerata inclusa nell'intervallo, ma se questo numero indica una riga precedente alla riga iniziale dell'intervallo, allora viene selezionata solo quella iniziale; se si tratta di un'espressione regolare, allora questo modello viene confrontato a partire dalla riga successiva a quella iniziale e alla corrispondenza raggiunta, si ottiene la riga finale dell'intervallo; se si tratta di un'espressione regolare, il confronto avviene a partire dalla riga successiva alla prima che è stata trovata.
riga_iniziale,*riga_finale*!	Se alla fine della selezione delle righe appare un punto esclamativo, questo rappresenta l'inversione della selezione, ovvero tutte le altre righe.

Come accennato, ogni direttiva si compone di una selezione di righe, in modo esplicito o implicito, e di un comando, ovvero di un raggruppamento di comandi racchiuso tra parentesi graffe. Vengono elencati di seguito i comandi più comuni.

Tabella 23.27. Comandi comuni.

Direttiva	Descrizione
#*commento*	Il simbolo '**#**' rappresenta un comando speciale di SED che serve solo a fargli ignorare il testo che segue fino alla fine della riga (fino alla fine della direttiva). Trattandosi di un «comando», si applica a delle righe, che però non possono essere indicate. Di solito, i commenti di questo tipo si inseriscono solo nei file contenenti direttive di un programma di elaborazione SED (eventualmente uno script eseguibile, realizzato nella forma che è già stata mostrata). Se i primi due caratteri di un file del genere corrispondono alla stringa '**#n**', SED funziona come se fosse stata usata l'opzione '**-n**', per cui occorre fare attenzione ai commenti che appaiono nella prima riga di tali file.
s/*bre*/*rimpiazzo*/[*parametri*] s*xbrexrimpiazzox*[*parametri*]	Con questo comando si vuole sostituire ciò che viene delimitato dall'espressione regolare con il testo di rimpiazzo, tenendo conto dei parametri posti eventualmente alla fine. L'espressione regolare e il testo di rimpiazzo sono delimitati e separati attraverso una barra obliqua normale, oppure da un altro simbolo scelto liberamente. Per inserire questa barra obliqua, o qualunque altro simbolo che svolga tale compito nell'espressione regolare, occorre proteggerlo con la barra obliqua inversa ('**\/**', ovvero '*****x***'). L'espressione regolare può essere realizzata in modo da individuare alcune parti, delimitate attraverso '**\(**' e '**\)**' (bisogna ricordare che si tratta di espressioni regolari elementari, ovvero di BRE); in tal caso, nella stringa di rimpiazzo si può fare riferimento a questi blocchi attraverso la forma '*****n***', dove ***n*** è un numero da uno a nove, che indica l'***n***-esimo riferimento a questi raggruppamenti della parte di riga presa in considerazione dall'espressione regolare. Nella stringa di rimpiazzo si può anche utilizzare la e-commerciale ('**&**') per fare riferimento a tutto il blocco di testo a cui corrisponde l'espressione regolare stessa. I parametri in coda al modello, hanno il significato che viene descritto nei modelli successivi.
s/*bre*/*rimpiazzo*/g[*altri_parametri*] s*xbrexrimpiazzox*g[*altri_parametri*]	Con il parametro '**g**' si esegue l'operazione di rimpiazzo per tutte le corrispondenze che si possono avere sulla stessa riga, senza limitarsi alla prima soltanto.
s/*bre*/*rimpiazzo*/p[*altri_parametri*] s*xbrexrimpiazzox*p[*altri_parametri*]	Con il parametro '**p**', se la sostituzione ha avuto luogo, emette la riga risultante (il *pattern space*).

Direttiva	Descrizione
s/*bre*/*rimpiazzo*/*n*[*altri_parametri*] s*xbrexrimpiazzoxn*[*altri_parametri*]	Indicando un numero come parametro, si rimpiazza solo nell'ambito dell'***n***-esima corrispondenza con l'espressione regolare.
s/*bre*/*rimpiazzo*/w *file* ↩ ↪[*altri_parametri*] s*xbrexrimpiazzox*w *file* ↩ ↪[*altri_parametri*]	Con il parametro '**w**', se la sostituzione ha avuto luogo, scrive la riga risultante nel file indicato.
q	Termina il funzionamento di SED senza altre elaborazioni e senza leggere altro dai file in ingresso.
d	Cancella l'area di memoria dove è stata accumulata la riga letta, avviando immediatamente un ciclo nuovo.
p	Emette la riga letta, con le modifiche eventuali che gli fossero state apportate nel frattempo. È importante ricordare che questo è il comportamento predefinito di SED, a meno che venga utilizzata l'opzione '**-n**'. Nella preparazione di script per SED occorre tenere presente che alcune realizzazioni di questo emettono la riga una sola volta, anche se viene usato il comando '**p**' e non è stata usata l'opzione '**-n**', mentre altre, come nel caso di GNU, lo fanno due volte. Questi due comportamenti opposti sono ammissibili secondo lo standard POSIX.
n	Questo comando permette di passare alla prossima riga, immediatamente, tenendo conto che se non è stata usata l'opzione '**-n**', prima di passare alla prossima viene emessa quella precedente (come al solito). Lo scopo di questo comando è fare in modo che le direttive successive si trovino di fronte una riga nuova.
w *file*	Copia le righe nel file indicato, creandolo per l'occasione.
{ *comandi* }	Un raggruppamento di comandi può essere realizzato delimitandolo tra parentesi graffe. Tuttavia, è importante osservare che in questo caso, i comandi vanno indicati ognuno in una riga differente, inoltre la parentesi graffa di chiusura deve apparire da sola in una riga. Di solito, non c'è la necessità di usare un raggruppamento, dal momento che basta ripetere la stessa selezione di righe con un altro comando.

Alcuni comandi che qui non vengono descritti, richiedono una scomposizione in più righe, indicando la continuazione attraverso il simbolo '\'. Dal momento che questi comandi non vengono mostrati, quello che si vuole far notare è che la barra obliqua inversa come simbolo di continuazione ha un significato speciale in SED, pertanto non va usata se non si conosce esattamente il risultato che si ottiene effettivamente.

23.5.5 Esempi

A seconda dei casi e dell'utilità della cosa, si fa riferimento a direttive fornite nella riga di comando (con o senza l'opzione '**-e**'), oppure a uno script vero e proprio.

Elaborazioni banali

- `$ sed "" prova.txt` [*Invio*]

  Legge il file '`prova.txt`' e lo riemette tale e quale, dal momento che non è stata specificata alcuna direttiva per il programma di elaborazione.

- `$ sed -n 'p' prova.txt` [*Invio*]

  Si ottiene lo stesso risultato dell'esempio precedente, perché prima viene usata l'opzione '**-n**' con cui si inibisce la riemissione predefinita delle righe lette, ma poi si specifica una direttiva contenente il comando '**p**' applicato a tutte le righe del flusso in ingresso.

Selezione delle righe

- `$ sed -n '1,10p' prova.txt` [*Invio*]

  Emette solo le prime 10 righe del file.

- `$ sed '/.\{81,\}/d' prova.txt` [*Invio*]

  Elimina le righe più lunghe di 80 caratteri.

- `$ sed '/^$/d' prova.txt` [*Invio*]

  Elimina tutte le righe vuote.

- `$ sed '/^---INIZIO---$/,/^---FINE---$/d' prova.txt` [*Invio*]

  Elimina tutte le righe comprese negli intervalli delimitati da righe contenenti esclusivamente la stringa '**---INIZIO---**' e '**---FINE---**'.

- `$ sed -n '/^---INIZIO---$/,/^---FINE---$/p' prova.txt` [*Invio*]

  Emette tutte le righe comprese negli intervalli delimitati da righe contenenti esclusivamente la stringa '**---INIZIO---**' e '**---FINE---**'.

Sostituzione del contenuto delle righe

- `$ sed 's/andato/venuto/' prova.txt` [*Invio*]

  Sostituisce in ogni riga la prima occorrenza della stringa «andato» con la stringa «venuto».

- `$ sed 's/andato/venuto/g' prova.txt` [*Invio*]

  Sostituisce tutte le occorrenze della stringa «andato» con la stringa «venuto».

- `$ sed 's/^/    /' prova.txt` [*Invio*]

  Aggiunge quattro spazi all'inizio di ogni riga del file.

- `$ sed 's/\(.*\):\(.*\):\(.*\):\(.*\):\(.*\):\(.*\):\(.*\)/\1:\3/' /etc/passwd` [*Invio*]

  Seleziona solo il primo e il terzo campo del file '`/etc/passwd`'; in pratica, preleva il nominativo e il numero UID.

Raggruppamenti

- ```
  #!/bin/sed -nf
  {
  p
  w registro
  }
  ```

  L'esempio mostra l'unione di due comandi riferiti allo stesso gruppo di righe (tutte). Lo scopo è quello di emettere le righe attraverso lo standard output e di annotarle anche in un file denominato '`registro`'.

  > Si osservi il fatto che la parentesi graffa di chiusura deve essere indicata da sola (come si vede nell'esempio) e di conseguenza può essere opportuno fare altrettanto per quella di apertura.

- `$ sed -n -e 'p' -e 'w registro' prova.txt` [*Invio*]

  Questo esempio fa la stessa cosa di quello precedente, con la differenza che i comandi sono stati separati in due direttive riferite allo stesso gruppo di righe, inoltre si elaborano le righe del file '`prova.txt`'.

23.6 Introduzione a AWK

AWK è un linguaggio di programmazione nato fondamentalmente per l'analisi e la rielaborazione di file di testo organizzati in una qualche forma tabellare. AWK potrebbe essere usato per fare anche di più, solo che quando si supera un certo limite di complessità, non è conveniente il suo utilizzo.

AWK è un interprete, nel senso che i programmi fatti secondo questo linguaggio, vengono eseguiti direttamente, senza essere compilati, come nel caso degli script di shell. Un programma AWK può essere scritto in un file di testo normale, oppure può essere fornito come argomento della riga di comando dell'interprete: il binario '**awk**'. Volendo automatizzare l'avvio dell'interprete per l'esecuzione di uno script (che abbia i permessi di esecuzione opportuni), lo si può iniziare con la direttiva comune agli script di shell:

```
#!/usr/bin/awk -f
```

Nei sistemi Unix esistono diversi tipi differenti di interpreti AWK. Con GNU/Linux potrebbe essere disponibile la versione GNU ('**gawk**'),[4] che ha molte estensioni rispetto agli standard, oppure ci potrebbe essere '**mawk**'.

23.6.1 Principio di funzionamento e struttura fondamentale

Il programma AWK tipico, è qualcosa che legge i dati provenienti da uno o più file, li analizza in qualche modo, generando un risultato che viene visualizzato direttamente o indirizzato a un altro file. Questo indica implicitamente due cose: un programma AWK non dovrebbe essere fatto per modificare i file di partenza; inoltre, si dà per scontato che ci sia una lettura dei file di origine, infatti ciò avviene di solito senza una richiesta esplicita.

Dal punto di vista di AWK, un file che viene analizzato è composto da record, corrispondenti normalmente alle righe del file di testo stesso, dove però il codice di interruzione di riga può essere specificato espressamente come qualcosa di diverso rispetto al solito.

Un programma AWK è composto fondamentalmente da ***regole***, le quali stabiliscono il comportamento da prendere nei confronti dei dati in ingresso. I commenti sono introdotti dal simbolo '**#**' e terminano alla fine della riga; inoltre, le righe vuote e quelle bianche vengono ignorate nello stesso modo. La struttura delle regole di un programma AWK si può esprimere secondo lo schema seguente:

> *criterio_di_selezione* { *azione* }

In pratica, ogni regola si suddivide in due parti: un'istruzione iniziale che definisce quali record prendere in considerazione e un'azione (più o meno articolata) indicata all'interno di parentesi graffe (che quindi vanno intese in modo letterale nel modello sintattico), da eseguire ogni volta che si incontra una corrispondenza con il criterio di selezione stabilito. Questa descrizione è solo una semplificazione che per il momento serve a iniziare la comprensione di questo linguaggio.

Una regola di un programma AWK può contenere l'indicazione esplicita del solo criterio di selezione, o della sola azione da compiere. Ciò perché in tal caso si utilizza un'azione o un criterio di selezione predefinito (questo particolare viene ripreso quando si mostrano i primi esempi).

L'azione di una regola AWK è molto simile a un programma C, o Perl, con tante semplificazioni, dove il record selezionato viene passato attraverso dei ***campi***, che ricordano i parametri delle shell comuni; quelli che si espandono con: '**$0**', '**$1**', '**$2**',... In pratica,
```

un’azione di una regola AWK è un programma a sé stante, eseguito ogni volta che il criterio di selezione della regola si avvera.

23.6.1.1 Selezione e azione predefinita

«

Una regola che non contenga l’indicazione del criterio di selezione, fa sì che vengano prese in considerazione tutte le righe dei dati in ingresso. In AWK, il valore booleano *Vero* si esprime con qualunque valore differente dallo zero e dalla stringa nulla, dal momento che entrambi questi rappresentano invece il valore *Falso* in un contesto booleano. In altre parole, una regola che non contenga l’indicazione del criterio di selezione, è come se avesse al suo posto il valore uno, che si traduce ogni volta in un risultato booleano *Vero*, cosa che permette la selezione di tutti i record.

Una regola che non contenga l’indicazione dell’azione da compiere, fa riferimento a un’azione predefinita, che in pratica fa sì che venga emessa attraverso lo standard output ogni riga che supera il criterio di selezione. Praticamente, è come se venisse usata l’azione ‘**{ print }**’. Per essere precisi, dal momento che in AWK il concetto di «predefinito» può riguardare diversi livelli, si tratta dell’azione ‘**{ print $0 }**’.

In pratica, se si unisse il criterio di selezione predefinito e l’azione predefinita, si avrebbe la regola seguente, che riemette attraverso lo standard output tutti i record che legge dai file in ingresso:

```
1 { print }
```

Bisogna ricordare però che almeno una delle due parti deve essere indicata esplicitamente: o il criterio di selezione, o l’azione.

23.6.1.2 Campi

«

Si è accennato al fatto che il testo analizzato da un programma AWK, viene visto generalmente come qualcosa composto da record suddivisi in campi. I record vengono individuati in base a un codice che li separa, corrispondente di solito al codice di interruzione di riga, per cui si ottiene l’equivalenza tra record e righe. I campi sono separati in modo analogo, attraverso un altro codice opportuno.

Eccezionalmente, quando il codice indicato per individuare la suddivisione in campi è <*SP*>, cioè lo spazio normale, diventa indifferente la quantità di spazi utilizzati tra un campo e l’altro; inoltre, è possibile utilizzare anche i caratteri di tabulazione.

Se per il codice che definisce la fine di un record e l’inizio di quello successivo, viene indicata la stringa nulla, (‘**""**’), si intende che i record siano separati da una o più righe bianche o vuote.

Ogni record può avere un numero variabile di campi; al loro contenuto si può fare riferimento attraverso il simbolo ‘**$**’ seguito da un numero che ne indica la posizione: ‘**$*n***’ è il campo ***n***-esimo del record attuale, ma in particolare, ‘**$0**’ rappresenta il record completo. Il numero in questione può anche essere rappresentato da un’espressione (per esempio una variabile) che si traduce nel numero desiderato. Per esempio, se ***pippo*** è una variabile contenente il valore due, ‘**$pippo**’ è il secondo campo.

23.6.1.3 Criterio di selezione e condizioni particolari

«

Il criterio di selezione dei record è generalmente un’espressione che viene valutata per ognuno di questi, in ordine, che, quando si avvera, permette l’esecuzione dell’azione corrispondente. Oltre a queste situazioni generali, esistono due istruzioni speciali da utilizzare come criteri di selezione: ‘**BEGIN**’ e ‘**END**’. Queste due parole chiave vanno usate da sole, rappresentando rispettivamente il momento iniziale prima di cominciare la lettura dei dati in ingresso e il momento finale successivo alla lettura ed elaborazione dell’ultimo record dei dati. Le azioni che si abbinano a queste condizioni particolari servono a preparare qualcosa e a concludere un’elaborazione.

Esiste un altro caso di criterio di selezione speciale, costituito da due espressioni separate da una virgola, come si vede nello schema seguente:

espressione_1 **,** ***espressione_2***

La prima espressione serve ad attivare il passaggio dei record; la seconda serve a disattivarlo. In pratica, quando si avvera la prima espressione, quel record e i successivi possono passare, fino a quando si avvera la seconda. Quando si avvera la seconda espressione (essendosi avverata anche la prima), il record attuale passa, ma quelli successivi non più. Se in seguito si riavvera la prima condizione, la cosa ricomincia.

Tabella 23.31. Schema complessivo dei diversi tipi di criteri di selezione in AWK.

Criterio di selezione	Descrizione
`BEGIN`	Esegue l’azione prima di iniziare a leggere i dati in ingresso.
`END`	Esegue l’azione dopo la lettura dei dati in ingresso.
espressione	Quando si avvera, esegue l’azione per il record attuale.
espr_1 **,** ***espr_2***	Le due espressioni individuano i record a intervalli.

23.6.1.4 Un programma banale per cominciare

«

Per mostrare il funzionamento di un programma AWK viene mostrato subito un esempio banale. Come è già stato descritto, la cosa più semplice che possa fare un programma AWK, è la riemissione degli stessi record letti in ingresso, senza porre limiti alla selezione.

```
1 { print $0 }
```

Come è già stato descritto, la regola mostrata è molto semplice: il numero uno rappresenta in pratica un valore corrispondente a *Vero*, dal punto di vista booleano, per cui si tratta di un’espressione che si avvera sempre, portando così alla selezione di tutti i record; l’azione richiede l’emissione della riga attuale, rappresentata da ‘**$0**’.

Se si realizza un file contenente la regola che è stata mostrata, supponendo di averlo chiamato ‘`banale`’, per avviarlo basta il comando seguente:

$ **awk -f banale** [*Invio*]

Nel comando non è stato specificato alcun file da analizzare, per cui l’interprete ‘**awk**’ lo attende dallo standard input, in questo caso dalla tastiera. Per terminare la prova basta concludere l’inserimento attraverso la combinazione [*Ctrl d*].

Un programma così breve può essere fornito direttamente nella riga di comando:

$ **awk '1 { print $0 }'** [*Invio*]

Per realizzare uno script, basta mettere l’intestazione corretta al file del programma, ricordando poi di rendere eseguibile il file:

```
#!/usr/bin/awk -f
1 { print $0 }
```

Prima di proseguire, è il caso di vedere come funzionano i criteri di selezione ‘**BEGIN**’ e ‘**END**’:

```
BEGIN { print "Inizio del programma" }
1 { print $0 }
END { print "Fine del programma" }
```

In questo modo, prima di iniziare la riemissione del testo che proviene dal file in ingresso, viene emesso un messaggio iniziale; quindi, alla fine di tutto viene emesso un altro messaggio conclusivo.

23.6.1.5 Variabili predefinite

«

AWK ha ereditato dalle shell l’idea delle variabili predefinite, con le quali si può modificarne l’impostazione. Le variabili predefinite si distinguono dalle altre perché sono tutte espresse attraverso nomi con lettere maiuscole.

Due di queste variabili sono fondamentali: ***RS***, *Record separator*, e ***FS***, *Field separator*. La prima serve a definire il carattere da prendere in considerazione per separare i dati in ingresso in record; la seconda serve a definire il codice da prendere in considerazione per separare i record in campi. Per la precisione, nel caso della variabile ***FS***, può trattarsi di un carattere singolo, oppure di un'espressione regolare.

I valori predefiniti di queste variabili sono rispettivamente <*LF*>, ovvero il codice di interruzione di riga dei file di testo normali, e uno spazio normale, che rappresenta una situazione particolare, come è già stato descritto. Questi valori possono essere cambiati: la situazione tipica in cui si deve intervenire nella variabile ***FS*** è quella della lettura di file come '`/etc/passwd`' e simili, dove si assegna generalmente alla variabile ***FS*** il valore '**`:`**', che è effettivamente il carattere utilizzato per separare i campi.

23.6.1.6 Struttura ideale di un programma AWK

Idealmente, un programma AWK potrebbe essere rappresentato in modo più esplicito, secondo lo schema sintattico seguente, dove le parentesi graffe vanno considerate in modo letterale:

```
[function nome_funzione (parametri_formali) { istruzioni }]
...
[BEGIN { azione }]
[BEGIN { azione }]
...
espressione_di_selezione { azione }
...
[END { azione }]
[END { azione }]
...
```

L'ordine indicato non è indispensabile, tuttavia è opportuno. In pratica vengono eseguite nell'ordine le fasi seguenti:

1. vengono eseguite le azioni abbinate alle condizioni '**`BEGIN`**', ammesso che esistano;
2. inizia la lettura del file in ingresso;
3. per ogni record vengono valutate le espressioni di selezione;
4. per ogni espressione che si avvera, viene eseguita l'azione corrispondente (se più espressioni si avverano simultaneamente, vengono eseguite ordinatamente tutte le azioni relative);
5. alla fine, vengono eseguite le azioni abbinate alle condizioni '**`END`**'.

Un programma AWK potrebbe essere composto anche solo da regole di tipo '**`BEGIN`**' o '**`END`**'. Nel primo caso non è nemmeno necessario leggere i dati in ingresso, mentre nel caso ci sia una regola di tipo '**`END`**', ciò diventa indispensabile, perché l'azione relativa potrebbe utilizzare le informazioni generate dalla lettura stessa.

AWK mette a disposizione una serie di funzioni predefinite, consentendo la dichiarazione di altre funzioni personalizzate. L'ordine in cui appaiono queste funzioni non è importante: una funzione può richiamare anche un'altra funzione dichiarata in una posizione successiva.

23.6.2 Avvio dell'interprete

L'interprete di un programma AWK è l'eseguibile '**`awk`**', che di solito è un collegamento alla realizzazione di AWK che risulta installata effettivamente: in un sistema GNU/Linux potrebbe trattarsi di '**`mawk`**' o '**`gawk`**' (il secondo è la versione GNU di AWK). La sintassi standard di un interprete AWK dovrebbe essere quella seguente:

```
awk [-F separazione_campi] [-v variabile=valore] ↩
↪ -f file_contenente_il_programma ↩
↪ [--] [file_in_ingresso...]
```

```
awk [-F separazione_campi] [-v variabile=valore] [--] ↩
↪ 'testo_del_programma' [file_in_ingresso...]
```

I due schemi alternativi riguardano la possibilità di far leggere all'interprete il programma contenuto in un file, indicato attraverso l'opzione '**`-f`**', oppure di fornirlo direttamente nella riga di comando, delimitandolo opportunamente perché venga preso dalla shell come un argomento singolo.

Se non vengono forniti i file da usare come dati in ingresso, l'interprete attende i dati dallo standard input.

Tabella 23.35. Alcune opzioni.

Opzione	Descrizione
`-F` *separazione_campi*	Definisce in che modo devono essere distinti i campi dei record, modificando così il valore predefinito della variabile ***FS***. Può trattarsi di un carattere singolo, oppure di un'espressione regolare estesa (ERE).
`-v` *variabile*`=`*valore*	Assegna un valore a una variabile. La variabile in questione può essere predefinita, oppure una nuova che viene utilizzata nel programma per qualche motivo.
`-f` *file_programma_awk*	Indica espressamente il file contenente il programma AWK del quale deve essere iniziata l'interpretazione.
`--`	Una coppia di trattini dichiara la conclusione delle opzioni normali e l'inizio degli argomenti finali (può essere usato per evitare ambiguità, nel caso ce ne possano essere). Gli argomenti successivi possono essere il programma stesso, se non è stata utilizzata l'opzione '**`-f`**', quindi i file da fornire in ingresso per l'elaborazione.

Segue la descrizione di alcuni esempi.

- `$` **`awk -f programma.awk elenco`**[*Invio*]

  Avvia l'esecuzione del programma contenuto nel file '`programma.awk`', per l'elaborazione del file '`elenco`'.

- `$` **`cat elenco | awk -f programma.awk`**[*Invio*]

  Esattamente come nell'esempio precedente, con la differenza che il file '`elenco`' viene fornito attraverso lo standard input.

- `$` **`awk -f programma.awk -F : /etc/passwd`**[*Invio*]

  Esegue una qualche elaborazione, attraverso il programma '`programma.awk`', sui dati del file '`/etc/passwd`'. Per questo motivo, viene definito l'utilizzo del carattere '**`:`**' come separatore dei campi che compongono i record di quel file.

- `$` **`awk -f programma.awk -v FS=: /etc/passwd`**[*Invio*]

  Esattamente come nell'esempio precedente, intervenendo direttamente sulla variabile predefinita ***FS***.

23.6.3 Espressioni

L'espressione è qualcosa che restituisce un valore. I tipi di valori gestiti da AWK sono pochi: numerici (numeri reali), stringhe e stringhe numeriche. I valori booleani non hanno un tipo indipendente: lo zero numerico e la stringa nulla valgono come *Falso*, mentre tutto il resto vale come *Vero* (anche la stringa '**`"0"`**' vale come *Vero*, a differenza di quanto accade con il linguaggio Perl).

23.6.3.1 Costanti

Le costanti sono espressioni elementari che restituiscono un valore in base a una simbologia convenuta. I valori numerici si esprimono in forma costante nei modi comuni anche agli altri linguaggi di programmazione. I valori interi si possono indicare come una serie di cifre numeriche, non delimitate, che esprimono il valore secondo una numerazione a base decimale; i valori non interi possono essere

espressi utilizzando il punto come separatore tra la parte intera e la parte decimale; sia i valori interi che gli altri, possono essere espressi secondo la notazione esponenziale. Le costanti numeriche che appaiono di seguito, sono esempi di rappresentazione dello stesso valore: 100,5.

```
100.5
1.005e+2
1005e-1
```

Le stringhe sono delimitate da apici doppi, come si vede nell'esempio seguente:

```
"questa è una stringa"
```

Le stringhe possono contenere delle sequenze di escape, come elencato nella tabella 23.38.

Tabella 23.38. Sequenze di escape utilizzabili all'interno delle stringhe costanti.

Escape	Significato
`\\`	\
`\"`	"
`\/`	/
`\a`	<BEL>
`\b`	<BS>
`\f`	<FF>
`\n`	<LF>
`\r`	<CR>
`\t`	<HT>
`\v`	<VT>
`\nnn`	il valore ottale ***nnn***

AWK gestisce anche un tipo speciale di costante, che è da considerare come un tipo speciale di stringa: l'espressione regolare costante. Questa è una stringa delimitata all'inizio e alla fine da una barra obliqua normale. L'esempio seguente è un'espressione regolare che corrisponde alla sottostringa '**`ciao`**':

```
/ciao/
```

Anche le espressioni regolari costanti ammettono l'uso di sequenze di escape e precisamente le stesse che si possono usare per le stringhe.

In generale, un'espressione regolare costante può essere usata alla destra di un'espressione di comparazione, in cui si utilizza l'operatore '**`~`**' o '**`!~`**'. Nelle altre situazioni, salvo i pochi casi in cui un'espressione regolare costante può essere indicata come parametro di una funzione, AWK sottintende che questa esprima la comparazione con il record attuale, ovvero con '**`$0`**'.

23.6.3.2 Campi e Variabili

«

Le variabili sono espressioni elementari che restituiscono il valore che contengono. AWK gestisce una serie di variabili predefinite, che possono essere lette per conoscere delle informazioni sui dati in ingresso, oppure possono essere modificate per cambiare il comportamento di AWK. Oltre a queste si possono utilizzare le variabili che si vogliono; per farlo è sufficiente assegnare loro un valore, senza bisogno di definirne il tipo.

Se in un'espressione si fa riferimento a una variabile che non è mai stata assegnata, questa restituisce la stringa nulla ('**`""`**'), che in un contesto numerico equivale allo zero. In questo senso, non c'è bisogno di inizializzare le variabili prima di usarle, dal momento che è noto il loro valore iniziale.

Eventualmente, una variabile può essere inizializzata a un valore determinato già al momento dell'avvio dell'interprete, attraverso l'opzione '**`-v`**' che è già stata descritta.

I nomi delle variabili sono sensibili alla differenza che c'è tra la collezione alfabetica maiuscola e quella minuscola. In particolare si può osservare che, convenzionalmente, i nomi di tutte le variabili predefinite sono espressi con lettere maiuscole, mentre le variabili definite all'interno del programma tendono a essere espresse utilizzando prevalentemente lettere minuscole.

All'interno di un programma AWK, i riferimenti ai campi del record attuale si fanno attraverso la forma '**`$`*n***', dove ***n*** rappresenta il campo ***n***-esimo. Il riferimento a un campo può essere ottenuto anche utilizzando il risultato di un'espressione, quando questa è preceduta dal dollaro. In particolare, è ammissibile anche l'assegnamento di un valore a un campo, per quanto questo sia una pratica sconsigliabile, dal momento che questo fatto non ha alcun significato nei confronti dei dati originali.

23.6.3.3 Operazioni e operatori

«

Gli operatori usati per le espressioni numeriche sono più o meno gli stessi del linguaggio C. Per quanto riguarda le stringhe, è previsto il concatenamento, che si ottiene senza alcun operatore esplicito, affiancando variabili o costanti stringa. Inoltre, dovendo gestire le espressioni regolari, si aggiungono due operatori speciali per il confronto di queste con delle stringhe. La tabella 23.40 raccoglie l'elenco degli operatori disponibili in AWK.

Tabella 23.40. Riepilogo degli operatori principali utilizzabili nelle espressioni di AWK.

Operatore e operandi	Descrizione
`(`*espressione*`)`	Valuta l'espressione contenuta tra parentesi prima di analizzare la parte esterna.
`++`*op*	Incrementa di un'unità l'operando prima che venga restituito il suo valore.
op`++`	Incrementa di un'unità l'operando dopo averne restituito il suo valore.
`--`*op*	Decrementa di un'unità l'operando prima che venga restituito il suo valore.
op`--`	Decrementa di un'unità l'operando dopo averne restituito il suo valore.
`+`*op*	Non ha alcun effetto dal punto di vista numerico.
`-`*op*	Inverte il segno dell'operando numerico.
op1 `+` *op2*	Somma i due operandi numerici.
op1 `-` *op2*	Sottrae dal primo il secondo operando numerico.
op1 `*` *op2*	Moltiplica i due operandi numerici.
op1 `/` *op2*	Divide il primo operando per il secondo.
op1 `%` *op2*	Modulo: il resto della divisione tra il primo e il secondo operando.
op1 `^` *op2*	Esponente: eleva il primo operando alla potenza del secondo.
var `=` *valore*	Assegna alla variabile il valore alla destra e restituisce lo stesso valore.
op1 `+=` *op2*	*op1* `=` *op1* `+` *op2*
op1 `-=` *op2*	*op1* `=` *op1* `-` *op2*
op1 `*=` *op2*	*op1* `=` *op1* `*` *op2*
op1 `/=` *op2*	*op1* `=` *op1* `/` *op2*
op1 `%=` *op2*	*op1* `=` *op1* `%` *op2*

Operatore e operandi	Descrizione
op1 ^= *op2*	*op1* = *op1* ^ *op2*
op1 && *op2*	AND logico, con cortocircuito.
op1 || *op2*	OR logico, con cortocircuito.
! *op*	NOT logico.
op1 > *op2*	*Vero* se il primo operando è maggiore del secondo.
op1 >= *op2*	*Vero* se il primo operando è maggiore o uguale al secondo.
op1 < *op2*	*Vero* se il primo operando è minore del secondo.
op1 <= *op2*	*Vero* se il primo operando è minore o uguale al secondo.
op1 == *op2*	*Vero* se i due operandi sono uguali.
op1 != *op2*	*Vero* se i due operandi sono diversi.
stringa ~ *regexp*	*Vero* se l'espressione regolare ha una corrispondenza con la stringa.
stringa !~ *regexp*	*Vero* se l'espressione regolare non ha alcuna corrispondenza.
stringa1 *stringa2*	Concatena le due stringhe.

Un tipo particolare di operatore logico è l'operatore condizionale, che permette di eseguire espressioni diverse in relazione al risultato di una condizione. La sua sintassi si esprime nel modo seguente:

```
condizione ? espressione1 : espressione2
```

In pratica, se l'espressione che rappresenta la condizione si avvera, viene eseguita la prima espressione che segue il punto interrogativo, altrimenti viene eseguita quella che segue i due punti.

Per quanto riguarda il confronto tra stringhe ed espressioni regolari, si deve tenere presente che lo scopo è solo quello di conoscere se c'è o meno una corrispondenza tra il modello e la stringa. Inoltre, è molto importante tenere in considerazione il fatto che un'espressione regolare costante, che non si trovi alla destra di un operatore '**~**', o '**!~**', viene interpretata come una forma contratta dell'espressione '**$0 ~/*regexp*/**', ovvero, si considera un confronto con il record attuale.

23.6.3.4 Conversione tra stringhe e numeri

«

Come è già stato descritto, AWK gestisce solo due tipi di dati: stringhe e numeri (reali). In base al contesto, i numeri vengono convertiti in stringhe e viceversa, solitamente in modo abbastanza trasparente. In particolare, una stringa che non possa essere interpretata come un numero, equivale a zero.

In generale, il concatenamento di stringhe, impone una trasformazione in stringa, mentre l'uso di operatori aritmetici impone una trasformazione in numero. Si osservi l'esempio:

```
uno = 1
due = 2
(uno due) + 3
```

Si tratta di tre istruzioni in sequenza, dove le prime due assegnano un valore numerico ad altrettante variabili, mentre l'ultima fa qualcosa di incredibile: concatena le due variabili, che di conseguenza vengono trattate come stringhe, generando la stringa '**"12"**'; quindi, la stringa viene riconvertita in numero, a causa dell'operatore '**+**', che richiede la somma con il numero tre. Alla fine, il risultato dell'ultima espressione è il numero 15.

La conversione da numero a stringa è banale quando si tratta di numeri interi, dal momento che il risultato è una stringa composta dalle stesse cifre numeriche che si utilizzano per rappresentare un numero intero. Al contrario, in presenza di numeri con valori decimali, entra

in gioco una conversione per mezzo della funzione ***sprintf()*** (equivalente a quella del linguaggio C), che utilizza la stringa di formato contenuta nella variabile predefinita ***CONVFMT***. Di solito, questa variabile contiene il valore '**"%.6g"**', che indica una precisione fino a sei cifre dopo la virgola, e una notazione che può essere esponenziale, oppure normale ('***intero*.*decimale***'), in base alla necessità. Le tabelle 23.42 e 23.43 riepilogano i simboli utilizzabili nelle stringhe di formato di ***sprintf()***. Eventualmente, per una descrizione più dettagliata, si può leggere la pagina di manuale *sprintf(3)*.

Tabella 23.42. Elenco dei simboli utilizzabili in una stringa formattata per l'utilizzo con ***sprintf()***.

Simbolo	Corrispondenza
`%%`	Segno di percentuale.
`%c`	Un carattere corrispondente al numero dato.
`%s`	Una stringa.
`%d` | `%i`	Un intero con segno in base dieci.
`%o`	Un intero senza segno in ottale.
`%x`	Un intero senza segno in esadecimale.
`%X`	Come '**%x**', ma con l'uso di lettere maiuscole.
`%e`	Un numero a virgola mobile, in notazione scientifica.
`%E`	Come '**%e**', ma con l'uso della lettera '**E**' maiuscola.
`%f`	Un numero a virgola mobile, in notazione decimale fissa.
`%g`	Un numero a virgola mobile, secondo la notazione di '**%e**' o '**%f**'.
`%G`	Come '**%g**', ma con l'uso della lettera '**E**' maiuscola (se applicabile).

Tabella 23.43. Elenco dei simboli utilizzabili tra il segno di percentuale e la lettera di conversione.

Simbolo	Corrispondenza
spazio	Il prefisso di un numero positivo è uno spazio.
`+`	Il prefisso di un numero positivo è il segno '**+**'.
`-`	Allinea a sinistra rispetto al campo.
`0`	Utilizza zeri, invece di spazi, per allineare a destra.
`#`	Prefissa un numero ottale con uno zero e un numero esadecimale con '**0x**'.
n	Un numero definisce la dimensione minima del campo.
.*n*	Per i numeri interi indica il numero minimo di cifre.
.*n*	Per i numeri a virgola mobile esprime la precisione, ovvero il numero di decimali.
.*n*	Per le stringhe definisce la lunghezza massima.

In generale, sarebbe bene non modificare il valore predefinito della variabile ***CONVFMT***, soprattutto non è il caso di ridurre la precisione della conversione, dal momento che la perdita di informazioni che ne deriverebbe, potrebbe creare anche dei gravi problemi a un programma. In altri termini, il formato di conversione condiziona la precisione dei valori che possono essere gestiti in un programma AWK.

23.6.3.5 Esempi di espressioni

«

Prima di proseguire con la descrizione del linguaggio AWK vengono mostrati alcuni esempi di programmi banali, in cui tutto si concentra sulla definizione delle espressioni per stabilire la selezione dei record. L'azione che si abbina è molto semplice: l'emissione del record selezionato attraverso l'istruzione '**print**'.

```
$ ls -l /etc | awk '$1 == "-rw-r--r--" { print $0 }'[Invio]
```

L'esempio appena mostrato fornisce all'interprete AWK il programma come argomento nella riga di comando. Come si vede, il risultato del comando '**ls -l /etc**' viene incanalato attraverso un condotto, fornendolo in ingresso al programma AWK, che si limita a selezionare i record in cui il primo campo corrisponde esattamente alla stringa '**"-rw-r--r--"**'. In pratica, vengono selezionati i record contenenti informazioni sui file che hanno solo i permessi 0644_8. L'esempio seguente ottiene lo stesso risultato, attraverso la comparazione con un'espressione regolare:

```
$ ls -l /etc | awk '$1 ~ /-rw-r--r--/ { print $0 }' [Invio]
```

I due esempi successivi sono equivalenti e servono a selezionare tutti i record che non corrispondono al modello precedente:

```
$ ls -l /etc ↩
↪| awk '!( $1 == "-rw-r--r--" ) { print $0 }' [Invio]
```

```
$ ls -l /etc | awk '!( $1 ~ /-rw-r--r--/ ) { print $0 }' [Invio]
```

L'esempio seguente utilizza due espressioni, per attivare e disattivare la selezione dei record:

```
$ awk '$0 ~ /\/\*/, $0 ~ /\*\// { print $0 }' prova.c [Invio]
```

In questo caso, i dati in ingresso provengono dal file '`prova.c`', che si intende essere un programma scritto in linguaggio C. Le due espressioni servono a selezionare le righe che contengono commenti nella forma '**/*...*/**'. Si osservi l'uso della barra obliqua inversa per proteggere i caratteri che altrimenti sarebbero stati interpretati diversamente.

La variante seguente è funzionalmente identica all'esempio precedente, dal momento che un'espressione regolare costante da sola, equivale a un'espressione in cui questa si paragona al record attuale:

```
$ awk '/\/\*/, /\*\// { print $0 }' prova.c [Invio]
```

23.6.4 Istruzioni

«

Nel linguaggio AWK, le istruzioni possono apparire nell'ambito della dichiarazione delle azioni abbinate a un certo criterio di selezione dei record, oppure nel corpo della dichiarazione di una funzione.

Le istruzioni di AWK terminano normalmente alla fine della riga, salvo quando nella parte finale della riga appare una virgola ('**,**'), una parentesi graffa aperta ('**{**'), una doppia e-commerciale ('**&&**'), o una doppia barra verticale ('**||**'). Eventualmente, per continuare un'istruzione nella riga successiva, si può utilizzare una barra obliqua inversa esattamente alla fine della riga, come simbolo di continuazione ('****').

Un'istruzione può essere terminata esplicitamente con un punto e virgola finale ('**;**'), in modo da poter collocare più istruzioni in sequenza sulla stessa riga.

Come è già stato descritto, le righe vuote e quelle bianche vengono ignorate; inoltre, ciò che è preceduto dal simbolo '**#**', fino alla fine della riga, è considerato un commento.

Le istruzioni di AWK possono essere delle espressioni di assegnamento, delle chiamate di funzione, oppure delle strutture di controllo.

23.6.4.1 Istruzioni fondamentali

«

Le istruzioni fondamentali di AWK sono quelle che permettono di emettere del testo attraverso lo standard output. Si tratta di due funzioni, che però possono essere usate anche in forma di «operatori»: '**print**' e '**printf**'. La prima di queste due permette l'emissione di una o più stringhe, mentre la seconda permette di definire una stringa in base a un formato indicato, emettendone poi il risultato. In pratica, '**printf**' si comporta in modo analogo alla funzione omonima del linguaggio C.

```
print
```



```
print espressione_1 [, espressione_1 ]...
```

```
print( espressione_1 [, espressione_1 ]...)
```

Quelli che si vedono sono gli schemi sintattici della funzione (o istruzione) '**print**'. Se non vengono specificati degli argomenti (ovvero dei parametri), si ottiene l'emissione del testo del record attuale. Se invece vengono indicati degli argomenti, questi vengono emessi in sequenza, inserendo tra l'uno e l'altro il carattere definito dalla variabile ***OFS*** (*Output field separator*), che di solito corrisponde a uno spazio normale. In tutti i casi, il testo emesso da '**print**' termina con l'inserimento del carattere contenuto nella variabile ***ORS*** (*Output record separator*), che di solito corrisponde al codice di interruzione di riga.

In altri termini, nel primo caso viene emessa la stringa corrispondente al concatenamento '**$0 ORS**'; nel secondo e nel terzo viene emessa la stringa corrispondente al concatenamento '***espressione_1*** **OFS** ***espressione_2*** **OFS** ... ***espressione_n*** **ORS**'.

```
printf stringa_di_formato, espressione_1 [, espressione_2 ]...
```

```
printf( stringa_di_formato, espressione_1 [, espressione_2 ]... )
```

L'istruzione, ovvero la funzione '**printf**', si comporta come la sua omonima del linguaggio C: il primo argomento è una stringa di formato, contenente una serie di simboli che iniziano con il carattere '**%**', che vanno rimpiazzati ordinatamente con gli argomenti successivi. Le tabelle 23.42 e 23.43 riepilogano i simboli utilizzabili nelle stringhe di formato di '**sprintf**'. Eventualmente, per una descrizione più dettagliata, si può leggere la pagina di manuale *sprintf(3)*.

A differenza di '**print**', '**printf**' non fa uso delle variabili ***OFS*** e ***ORS***, dal momento che quello che serve può essere inserito tranquillamente nella stringa di formato (il carattere *<LF>*, corrispondente al codice di interruzione di riga, viene indicato con la sequenza di escape '**\n**').

23.6.4.2 Ridirezione dell'output

«

L'output generato dalle istruzioni '**print**' e '**printf**' può essere ridiretto all'interno del programma AWK stesso, utilizzando gli operatori '**>**', '**>>**' e '**|**'. Ciò permette di ridirigere i dati verso file differenti; diversamente, converrebbe intervenire all'esterno del programma, per mezzo del sistema operativo.

```
print ... > file
```

```
printf ... > file
```

```
print ... >> file
```

```
printf ... >> file
```

```
print ... | comando
```

```
printf ... | comando
```

Utilizzando l'operatore '**>**' si ridirigono i dati verso un file, che viene azzerato inizialmente, oppure viene creato per l'occasione; con l'operatore '**>>**' si accodano dati a un file già esistente; con l'operatore '**|**' si inviano dati allo standard input di un altro comando. È impor-

tante osservare che i file e i comandi in questione, vanno indicati in una stringa. Si osservino gli esempi seguenti:

```
# annota il secondo campo nel file /tmp/prova
print $2 > "/tmp/prova"
```

```
# accoda il secondo campo nel file /tmp/prova
print $2 >> "/tmp/prova"
```

```
# definisce un comando per riordinare i dati e salvarli nel
# file /tmp/prova
comando = "sort > /tmp/prova"
#seleziona alcuni campi e poi invia al comando di riordino
print $2 $4 $5 | comando
```

23.6.4.3 Strutture di controllo di flusso

Il linguaggio AWK offre alcune strutture di controllo di flusso comuni agli altri linguaggi di programmazione. In particolare, come nel linguaggio C, è possibile raggruppare alcune istruzioni delimitandole con le parentesi graffe ('{...}').

Le strutture di controllo permettono di sottoporre l'esecuzione di una parte di codice alla verifica di una condizione, oppure permettono di eseguire dei cicli, sempre sotto il controllo di una condizione. La parte di codice che viene sottoposta a questo controllo, può essere un'istruzione singola, oppure un gruppo di istruzioni. Nel secondo caso, è necessario delimitare questo gruppo attraverso l'uso delle parentesi graffe, a cui si è appena accennato.

Dal momento che è comunque consentito di realizzare un gruppo di istruzioni che in realtà ne contiene una sola, probabilmente è meglio utilizzare sempre le parentesi graffe, in modo da evitare equivoci nella lettura del codice (dato che le parentesi graffe sono usate nel linguaggio AWK, se queste appaiono nei modelli sintattici indicati, queste fanno parte delle istruzioni e non della sintassi).

La tabella 23.47 riassume la sintassi di queste strutture, la maggior parte delle quali dovrebbero essere già note dal linguaggio C, o da altri linguaggi simili.

Tabella 23.47. Istruzioni per le strutture di controllo del flusso in AWK.

Sintassi	Descrizione
`{`*istruzioni*`}`	Raggruppa assieme alcune istruzioni.
`if (`*condizione*`)` *istruzione* [`else` *istruzione*]	Struttura condizionale.
`while (`*condizione*`)` *istruzione*	Ciclo iterativo con condizione iniziale.
`do` *istruzione* `while (`*condizione*`)`	Ciclo iterativo con condizione alla fine.
`for (` *espr_1*`;` *espr_2*`;` *espr_3*`)` *istruzione*	Ciclo enumerativo.
`break`	Interrompe un ciclo iterativo o enumerativo.
`continue`	Riprende un ciclo iterativo o enumerativo.
`exit` [*espressione*]	Termina il programma restituendo il valore dell'argomento.
`next`	legge il prossimo record.

Data la natura di AWK, esiste un'istruzione particolare: '**`next`**'. Questa serve a passare immediatamente al record successivo. Segue la descrizione di alcuni esempi.

- ```
  if ( $1 > 100 ) print $2
  ```
  Se il primo campo del record attuale contiene un valore numerico superiore a 100, emette il contenuto del secondo campo.

- ```
  if ( $1 > 100 ) {
      print $2
      contatore++
  } else {
      print $3
  }
  ```
  Se il primo campo del record attuale contiene un valore numerico superiore a 100, emette il contenuto del secondo campo, incrementando la variabile ***contatore*** di un'unità. Altrimenti, emette solo il contenuto del terzo campo.

- ```
  i = 1
  while (i <= 10) {
      print i
      i++
  }
  ```
  Emette i numeri da 1 a 10.

- ```
  for ( i = 1; i <= 10; i++ ) {
      print i
  }
  ```
  Esattamente come nell'esempio precedente, utilizzando un ciclo enumerativo.

- ```
  for ( i = 1; i <= 20; i++ ) {
      if ( i != 13 ) {
          print i
      }
  }
  ```
  Emette i numeri da 1 a 20, escluso il 13.

- ```
  for ( i = 1; i <= 20; i++ ) {
      if ( i == 13 ) {
          continue
      }
      print i
  }
  ```
  Come nell'esempio precedente, utilizzando una tecnica diversa (l'istruzione '**`continue`**' fa riprendere il ciclo prima di avere completato le altre istruzioni).

- ```
  i = 1
  while (1) {
      if ( i > 10 ) {
          break
      }
      print i
      i++
  }
  ```
  Emette i numeri da 1 a 10, utilizzando un ciclo iterativo perpetuo (il numero uno equivale a *Vero* per AWK), che viene interrotto dall'istruzione '**`break`**'.

23.6.4.4 Chiamata di funzione e funzioni predefinite

La chiamata di una funzione avviene come nel linguaggio C, tenendo conto che per evitare ambiguità, è importante mettere sempre la parentesi iniziale del gruppo dei parametri, attaccata al nome della funzione stessa:

funzione `(`***elenco_parametri***`)`

I parametri sono separati attraverso delle virgole, tenendo conto che in linea di principio si possono omettere quelli finali (si possono omettere tutti i parametri a partire da una certa posizione). I parametri che non vengono forniti sono equivalenti a stringhe nulle; in certi casi ci sono funzioni predisposte per riconoscere la mancata indicazione di tali informazioni, che così gestiscono attribuendo valori predefiniti.

Come nel linguaggio C, il passaggio dei parametri avviene per valore (salvo eccezioni), per cui i parametri in una chiamata possono essere delle espressioni più o meno articolate, che vengono valutate (senza un ordine preciso) prima della chiamata stessa.

Di seguito vengono descritte brevemente le funzioni interne (predefinite) di AWK. In particolare, le funzioni numeriche comuni sono elencate nella tabella 23.55.

Tabella 23.55. Elenco delle funzioni numeriche principali.

Funzione	Descrizione.
`atan2(`***y***`,` ***x***`)`	Arcotangente di ***y***/***x*** in radianti.
  ```

Funzione	Descrizione.
`cos(x)`	Coseno di ***x*** espresso in radianti.
`exp(x)`	Funzione esponenziale (e^x).
`int(x)`	Parte intera di un numero reale.
`log(x)`	Logaritmo naturale (base ***e***).
`rand()`	Numero casuale compreso tra zero e uno.
`sin(x)`	Seno di ***x*** espresso in radianti.
`sqrt(x)`	Radice quadrata di ***x***.

```
index( stringa, sottostringa_cercata )
```

La funzione ***index()*** cerca la stringa indicata come secondo parametro nella stringa indicata come primo, cominciando da sinistra. Se trova la corrispondenza, restituisce la posizione iniziale di questa, altrimenti restituisce zero.

```
index( "Tizio", "zio" )
```

L'espressione mostrata come esempio, restituisce il valore tre, corrispondente al primo carattere in cui si ottiene la corrispondenza della stringa '`zio`' in '`Tizio`'.

```
length( [stringa] )
```

La funzione ***length()*** restituisce la lunghezza della stringa fornita come parametro, oppure, in sua mancanza, la lunghezza di '`$0`', ovvero del record attuale. Si osservino gli esempi.

```
length( "Tizio" )
```

Restituisce il valore cinque, dal momento che la stringa è composta da cinque caratteri.

```
length( 10 * 5 )
```

Dal momento che il parametro della funzione è un'espressione numerica, prima calcola il valore di questa espressione, ottenendo il numero 50, quindi lo trasforma in stringa e restituisce il valore due. In pratica, il numero 50 espresso in stringa è lungo due caratteri.

```
match( stringa, regexp )
```

La funzione ***match()*** cerca una corrispondenza per l'espressione regolare fornita come secondo parametro, con la stringa che appare come primo parametro. L'espressione regolare dovrebbe poter essere fornita in forma costante, senza che questo fatto venga inteso come un confronto implicito con il record attuale.

Se il confronto ha successo, viene restituita la posizione in cui inizia la corrispondenza nella stringa; inoltre, le variabili predefinite ***RSTART*** e ***RLENGTH*** vengono impostate rispettivamente a questa posizione e alla lunghezza della corrispondenza. Se il confronto fallisce, la funzione restituisce il valore zero e così viene impostata la variabile ***RSTART***, mentre ***RLENGTH*** riceve il valore -1.

```
sprintf( stringa_di_formato, espressione [,...] )
```

La funzione ***sprintf()*** restituisce una stringa in base alla stringa di formato indicata come primo parametro, in cui le metavariabili '`%...`' vengono sostituite, nell'ordine, dai parametri successivi. Le metavariabili in questione sono state elencate nelle tabelle 23.42 e 23.43.

```
importo = 10000
sprintf( "Il totale è di EUR %i + IVA", importo )
```

L'espressione finale dell'esempio restituisce la stringa: «Il totale è di EUR 10000 + IVA».


```
sub( regexp, rimpiazzo [, stringa_da_modificare] )
```

La funzione ***sub()***, cerca all'interno della stringa fornita come ultimo parametro, oppure all'interno del record attuale, la prima corrispondenza con l'espressione regolare indicata come primo parametro. Quindi, sostituisce quella corrispondenza con la stringa fornita come secondo parametro. L'espressione regolare dovrebbe poter essere fornita in forma costante, senza che questo fatto venga inteso come un confronto implicito con il record attuale.

L'ultimo parametro deve essere una variabile, dal momento che viene passata per riferimento e il suo contenuto deve essere modificato dalla funzione.

La stringa di sostituzione (il secondo parametro), può contenere il simbolo '`&`', che in tal caso viene sostituito con la sottostringa per la quale si è avverata la corrispondenza con l'espressione regolare. Volendo inserire una e-commerciale letterale, si deve usare la sequenza '`\&`'.

L'indicazione di una e-commerciale letterale può essere un problema. In generale sarebbe meglio evitarlo. In ogni caso, è necessario leggere la documentazione specifica per il tipo di interprete AWK che si utilizza, per sapere come comportarsi esattamente.

La funzione ***sub()*** restituisce il numero di sostituzioni eseguite, pertanto può trattarsi del valore uno o di zero.

```
frase = "ciao, come stai?"
sub( /ciao/, "salve", frase )
```

L'espressione finale dell'esempio restituisce il valore uno, dal momento che la sostituzione ha luogo, mentre la variabile ***frase*** contiene alla fine la stringa: «salve, come stai?».

```
frase = "ciao, come stai?"
sub( /ciao/, "& amico", frase )
```

Questo esempio riutilizza la sottostringa della corrispondenza, attraverso il riferimento ottenuto con la e-commerciale. Alla fine, la variabile ***frase*** contiene: «ciao amico, come stai?».

```
gsub( regexp, rimpiazzo [, stringa_da_modificare] )
```

La funzione ***gsub()***, cerca all'interno della stringa fornita come ultimo parametro, oppure all'interno del record attuale, tutte le corrispondenze con l'espressione regolare indicata come primo parametro. Quindi, sostituisce quelle corrispondenze con la stringa fornita come secondo parametro. In pratica, si tratta di una variante di ***sub()***, in cui la sostituzione avviene in modo «globale». Valgono tutte le altre considerazioni fatte sulla funzione ***sub()***.

```
substr( stringa, inizio [, lunghezza] )
```

La funzione ***substr()*** restituisce una sottostringa di quanto fornito come primo parametro, prendendo ciò che inizia dalla posizione del secondo parametro, per una lunghezza pari al terzo parametro, oppure, fino alla fine della stringa di partenza.

```
substr( "ciao come stai", 6, 4 )
```

L'espressione dell'esempio restituisce la stringa «come».

```
tolower( stringa )
```

La funzione ***tolower()*** restituisce la stringa fornita come parametro trasformata utilizzando solo lettere minuscole.

```
toupper( stringa )
```

La funzione ***toupper()*** restituisce la stringa fornita come parametro trasformata utilizzando solo lettere maiuscole.

23.6.5 Variabili predefinite

La tabella 23.63 riepiloga le variabili predefinite principali di AWK. In particolare, sono state escluse quelle che riguardano la gestione degli array.

Tabella 23.63. Elenco delle variabili predefinite principali di AWK.

Variabile	Descrizione
`CONVFMT`	Formato di conversione da numero a stringa.
`FILENAME`	Nome del file attuale in ingresso, oppure '`-`'.
`FNR`	Numero del record attuale nel file attuale.
`FS`	Separatore dei campi in lettura.
`NF`	Numero totale dei campi nel record attuale.
`NR`	Numero totale dei record letti fino a questo punto.
`OFMT`	Formato di emissione dei numeri (di solito si tratta di '`%.6g`').
`OFS`	Separatore dei campi per '`print`'.
`ORS`	Separatore dei record per '`print`'.
`RS`	Separatore dei record in lettura.
`RSTART`	Utilizzata da ***match()*** per annotare l'inizio di una corrispondenza.
`RLENGTH`	Utilizzata da ***match()*** per annotare la lunghezza di una corrispondenza.

È il caso di ribadire alcuni concetti fondamentali riferiti alle variabili ***FS*** e ***RS***.

- I record in ingresso sono distinti in base al contenuto della variabile ***RS***. Per restare aderenti allo standard POSIX, questa può contenere un carattere, oppure la stringa nulla. Di solito, la variabile ***RS*** contiene il carattere *<LF>*, ovvero il codice di interruzione di riga comune nei sistemi Unix. Nel caso in cui sia indicata la stringa nulla, si è di fronte a una situazione particolare: i record sono separati da una o più righe bianche o vuote.
- I campi dei record in ingresso sono distinti in base al contenuto della variabile ***FS***. Questa variabile può contenere un carattere singolo, oppure un'espressione regolare (senza delimitatori). La corrispondenza con il carattere, o con l'espressione regolare rappresenta ciò che viene considerato il separatore dei campi. Di solito, la variabile ***FS*** contiene il carattere *<SP>*, ovvero lo spazio, che costituisce una situazione particolare: la separazione tra i campi è ottenuta inserendo qualunque spazio orizzontale (*<SP>* o *<HT>*), di qualunque lunghezza. Questa eccezione permette di leggere agevolmente i listati tabellari in cui i dati sono incolonnati in qualche modo, attraverso spaziature più o meno ampie.

23.6.6 Esempi

Gli esempi che vengono mostrati qui sono molto banali e sono tratti prevalentemente da *Effective AWK Programming* di Arnold D. Robbins. Tuttavia, qui sono mostrati come script autonomi, utilizzando una notazione che potrebbe sembrare ridondante, ma che può essere utile per non confondere il principiante. Trattandosi di script autonomi, questi ricevono i dati in ingresso solo attraverso lo standard input.

```
#!/usr/bin/awk -f
1 {
    if (length($0) > max) {
        max = length($0)
    }
}
END {
    print max
}
```


Questo esempio serve a trovare la riga di lunghezza massima di un file di testo normale. In pratica, viene scandito ogni record e viene memorizzata la sua lunghezza se questa risulta superiore all'ultima misurazione effettuata. Alla fine viene emesso il contenuto della variabile che è stata usata per annotare questa informazione.

```
#!/usr/bin/awk -f
length($0) > 80 { print $0 }
```

Questo esempio emette tutte le righe di un file di testo che superano la lunghezza di 80 caratteri.

```
#!/usr/bin/awk -f
NF > 0 { print $0 }
```

In questo caso vengono emesse tutte le righe di un file di testo che hanno almeno un campo. In pratica, vengono escluse le righe bianche e quelle vuote.

```
#!/usr/bin/awk -f
1 { totale += $5 }
END { print "totale:" totale "byte" }
```

Questo programma è fatto per sommare i valori del quinto campo di ogni record. In pratica, si tratta di incanalare nel programma il risultato di un comando '**`ls -l`**', in modo da ottenere il totale in byte.

```
#!/usr/bin/awk -F : -f
1 { print $1 }
```

Questo programma è banale, ma ha qualcosa di speciale: la riga iniziale indica che si tratta di uno script di '`/usr/bin/awk`', che deve essere avviato con le opzioni '**`-F : -f`**'. In pratica, rispetto al solito, è stata aggiunta l'opzione '**`-F :`**', con la quale si specifica che la separazione tra i campi dei record è data dal carattere '**`:`**'. Il programma, di per sé, è fatto per leggere un file composto da righe separate in questo modo, come nel caso di '`/etc/passwd`', allo scopo di emettere solo il primo campo, che, sempre nel caso si tratti di '`/etc/passwd`', corrisponde al nominativo-utente.

```
#!/usr/bin/awk -F : -f
BEGIN { print "Gli utenti seguenti accedono senza parola d'ordine:" }
$2 == "" { print $1 }
```

Si tratta di una variante dell'esempio precedente, dove si presume che i dati in ingresso provengano sicuramente dal file '`/etc/passwd`'. In questo caso, vengono visualizzati i nomi degli utenti che non hanno una parola d'ordine nel secondo campo.

```
#!/usr/bin/awk -f
END { print NR }
```

Legge il file fornito attraverso lo standard input ed emette il numero complessivo di record che lo compongono.

```
#!/usr/bin/awk -f
(NR % 2) == 0 { print }
```

In questo caso, vengono emessi solo i record pari. In pratica, l'espressione '**`(NR % 2) == 0`**' si avvera solo quando non c'è resto nella divisione della variabile ***NR*** per due.

23.6.7 Dichiarazione di funzioni

Un programma AWK può contenere la dichiarazione di funzioni definite liberamente. Queste dichiarazioni vanno fatte al di fuori delle regole normali. La dichiarazione di una funzione avviene in modo simile al linguaggio C, con la differenza che non si dichiara il tipo restituito dalla funzione e nemmeno quello delle variabili che ricevono i valori della chiamata.

```
function nome_della_funzione( elenco_parametri_formali ) {
    istruzioni
}
```

La parentesi tonda aperta che introduce l'elenco dei parametri formali, **deve essere attaccata alla fine del nome della funzione che viene dichiarata**. L'elenco dei parametri formali è in pratica un elenco di nomi di variabili locali, che ricevono il valore dei parametri corrispondenti nella chiamata. Se una chiamata di funzione

utilizza meno parametri di quelli che sono disponibili, le variabili corrispondenti ricevono in pratica la stringa nulla.

È importante osservare che non è possibile dichiarare altre variabili locali, oltre a quelle che appaiono nell'elenco dei parametri formali.

```
function fattoriale(x) {
    i = x - 1
    while ( i > 0 ) {
        x *= i
        i--
    }
    return x
}
```

L'esempio mostra la dichiarazione di una funzione ricorsiva, per il calcolo del fattoriale. Si può osservare l'istruzione '**return**', che permette di stabilire il valore che viene restituito dalla funzione. Naturalmente sono ammissibili anche funzioni che non restituiscono un valore: queste non hanno l'istruzione '**return**'.

```
function somma( x, y, z, i ) {
    z = x
    for ( i = 1; i <= y; i++ ) {
        z++
    }
    return z
}
```

Un altro esempio può servire per comprendere la gestione delle variabili locali in una funzione. In questo caso si tratta di una funzione che calcola la somma dei primi due parametri che gli vengono forniti. I due parametri successivi, ***z*** e ***i***, sono dichiarati tra i parametri formali per essere usati come variabili locali; come si vede, la funzione non tiene in considerazione i valori che potrebbero trasportare.

In effetti, la funzione potrebbe utilizzare ugualmente le variabili ***z*** e ***i***, anche se queste non fossero dichiarate tra i parametri formali. In tal modo, però, queste variabili sarebbero globali, pertanto si potrebbero porre dei problemi di conflitti con altre variabili con lo stesso nome usate altrove nel programma.

```
#!/bin/awk -f
function somma( x, y, z, i ) {
    z = x
    for ( i = 1; i <= y; i++ ) {
        z++
    }
    return z
}
1 { print $1 "+" $2 "=" somma( $1, $2 ) }
```

Questo ultimo esempio mostra un programma completo per ottenere la somma dei primi due campi di ogni record fornito in ingresso.

23.6.8 Array

Il linguaggio AWK può gestire anche gli array, di tipo associativo, simili a quelli del linguaggio Perl. A seconda dell'uso che si vuole fare di questi array, ci si può anche «dimenticare» di questa particolarità di AWK, utilizzando i soliti indici numerici, che però AWK tratta come stringhe.

23.6.8.1 Dichiarazione e utilizzo di un array

La dichiarazione di un array avviene nel momento in cui vi si fa riferimento. In pratica, con l'istruzione seguente si assegna la stringa '**"ciao"**' all'elemento '**"2"**' dell'array '**a**':

```
a[2] = "ciao"
```

Se l'array non esiste già, viene creato per l'occasione. Nello stesso modo, se l'elemento '**"2"**' non esiste, viene creato all'interno dell'array.

In pratica, l'array di AWK è un insieme di elementi a cui si fa riferimento con un indice libero. Il fare riferimento a un elemento che non esiste, anche solo per leggerne il contenuto, implica la creazione di

tale elemento. Come si può intuire, il riferimento a un elemento che non esiste ancora, crea tale elemento assegnandogli la stringa nulla, restituendo pertanto lo stesso valore.

L'esempio seguente crea un array un po' strampalato, con una serie di valori senza un significato particolare:

```
elenco["ciao"] = "Saluti"
elenco["maramao"] = 123
elenco[3] = 345
elenco[2345] = "che bello"
```

Si intuisce che gli elementi di un array AWK non hanno un ordine preciso.

È importante tenere presente che non è possibile riutilizzare una variabile scalare come array; nello stesso modo, non si può riutilizzare un array come se fosse una variabile scalare. Se si tenta di fare una cosa del genere, l'interprete dovrebbe bloccarsi con una segnalazione di errore.

23.6.8.2 Scandire gli elementi di un array

La scansione degli elementi di un array AWK può essere un problema, se si pensa alla sua natura. Per esempio, dal momento che facendo riferimento a un elemento che non esiste, lo si crea implicitamente, si capisce che non si può nemmeno andare per tentativi. Per risolvere il problema, AWK fornisce due strumenti: l'operatore '**in**' e una variante della struttura di controllo '**for**'.

Per verificare che un array contenga effettivamente l'elemento corrispondente a un certo indice, si usa l'operatore '**in**', nel modo seguente:

```
indice in array
```

Per esempio, per verificare che esista l'elemento '**prova[234]**', si può usare un'istruzione simile a quella seguente:

```
if (234 in prova) {
    print "L'elemento prova[234] corrisponde a " prova[234]
}
```

Per scandire tutti gli elementi di un array si usa la struttura di controllo '**for**' in un modo particolare:

```
for (variabile in array) istruzione
```

In pratica, per ogni elemento contenuto nell'array, viene eseguita l'istruzione (o il blocco di istruzioni) che segue, tenendo conto che alla variabile viene assegnato ogni volta l'indice dell'elemento in corso di elaborazione.

È chiaro che l'ordine in cui appaiono gli elementi dipende dall'interprete AWK; in generale dovrebbe dipendere dalla sequenza con cui questi sono stati inseriti. L'esempio seguente, scandisce un array e mostra il contenuto di ogni elemento:

```
for (i in elenco) {
    print "elenco[" i "]  " elenco[i]
}
```

23.6.8.3 Cancellazione di un elemento

L'eliminazione di un elemento di un array si ottiene con l'istruzione '**delete**':

```
delete array[indice]
```

Alcune realizzazioni di AWK sono in grado di eliminare completamente un array, se non si indica l'indice di un elemento. In alternativa, si ottiene questo risultato con la funzione ***split()***, come si vede sotto. L'uso di questa funzione viene mostrato più avanti.

```
split("", array)
```

Considerato che per AWK l'eliminazione di un array è precisamente l'eliminazione di tutti i suoi elementi, si potrebbe fare anche come viene mostrato nello schema seguente:

```
for (variabile in array) {
    delete array[variabile]
}
```

23.6.8.4 Indici numerici e indici «nulli»

Gli indici di un array AWK sono delle stringhe, quindi, se si usano dei numeri, questi vengono convertiti in stringa, utilizzando la stringa di formato contenuta nella variabile ***CONVFMT***. Finché si usano indici numerici interi, non sorgono problemi; nel momento in cui si utilizzano valori non interi, la conversione può risentire di un troncamento, o di un'approssimazione derivata dalla conversione. In altri termini, due indici numerici differenti potrebbero puntare di fatto allo stesso elemento, perché la trasformazione in stringa li rende uguali.

L'indice di un array potrebbe essere anche una variabile mai usata prima. In tal caso, la variabile contiene la stringa nulla. Nel caso in cui questa variabile venga poi trattata in modo numerico, incrementando o decrementando il suo valore, per creare e fare riferimento a elementi dell'array che si vogliono raggiungere con indici pseudo-numerici, bisogna tenere presente che esiste anche l'elemento con indice '`""`'. Se si tenta di raggiungerlo con l'indice '`"0"`', si fallisce nell'intento.

```
1 {
    riga[n] = $0
    n++
}
END {
    for ( i=n-1; i >= 0; i-- ) {
        print riga[i]
    }
}
```

Si intuisce che il programma AWK che si vede nell'esempio serva ad accumulare tutte le righe lette nell'array '`riga`', quindi a scandire lo stesso array per emettere il testo di queste righe. Se si osserva con attenzione, di capisce che la prima riga non può essere ottenuta. Infatti, la variabile ***n*** viene utilizzata subito la prima volta, quando il suo contenuto iniziale è la stringa nulla, '`""`'; successivamente viene incrementata, facendo sì che quella stringa nulla venga intesa come uno zero, ma intanto è stato creato l'elemento '`riga[""]`'. Alla fine della lettura di tutti i record, viene scandito nuovamente l'array, trattandolo come se contenesse elementi da zero a '`n-1`'. Tuttavia, dal momento che l'elemento '`riga[0]`' non esiste, perché al suo posto c'è invece '`riga[""]`' che non viene raggiunto, si perde la prima riga.

23.6.8.5 Trasformare una stringa delimitata in un array

È molto importante considerare la possibilità di convertire automaticamente una stringa in un array attraverso la funzione interna ***split()***.

```
split( stringa, array[, separatore])
```

In pratica, il primo parametro è la stringa da suddividere; il secondo è l'array da creare (nel caso esista già, vengono eliminati tutti i suoi elementi); il terzo, è il carattere, o l'espressione regolare, che si utilizza per separare gli elementi all'interno della lista. Se non viene indicato l'ultimo argomento, viene utilizzato il contenuto della variabile ***FS*** (come si può intuire). Dal momento che questo tipo di operazione è analoga alla separazione in campi di un record, anche in questo caso, se il carattere di separazione è uno spazio (<*SP*>), gli elementi vengono individuati tra delimitatori composti da sequenze indefinite di spazi e tabulazioni.

Il primo elemento dell'array creato in questo modo ha indice '`"1"`', il secondo ha indice '`"2"`', continuando così, di seguito, fino all'elemento ***n***-esimo.

```
split( "uno-due-tre", elenco, "-" )
```

L'esempio che si vede crea (o ricrea) l'array '`elenco`', con tre elementi contenenti le stringhe '`uno`', '`due`' e '`tre`'. In pratica, è come se si facesse quanto segue:

```
elenco[1] = "uno"
elenco[2] = "due"
elenco[3] = "tre"
```

Se non c'è alcuna corrispondenza tra il carattere, o l'espressione regolare, che si utilizzano come ultimo argomento, viene creato solo l'elemento con indice '`"1"`', nel quale viene inserita tutta la stringa di partenza.

23.6.8.6 Array pseudo-multidimensionali

Gli array di AWK sono associativi, pertanto non ha senso parlare di dimensioni, in quanto è disponibile un solo indice. Tuttavia, gestendo opportunamente le stringhe, si possono individuare idealmente più dimensioni, anche se ciò non è vero nella realtà. Supponendo di voler gestire un array a due dimensioni, con indici numerici, si potrebbero indicare gli indici come nell'esempio seguente, dove si assegna un valore all'elemento ideale «1,10»:

```
elenco[1 "s" 10] = 123
```

La lettera «s» che si vede, è solo una stringa, scelta opportunamente, in modo che l'indice che si ottiene non si possa confondere con qualcosa che non si vuole. In questo caso, l'indice reale è la stringa '`1s10`'.

AWK offre un supporto a questo tipo di finzione multidimensionale. Per farlo, esiste la variabile ***SUBSEP***, che viene usata per definire il carattere di separazione. Questo carattere è generalmente <*FS*>, che si esprime in esadecimale come $1C_{16}$ e in ottale come 34_8, corrispondente per AWK alla sequenza di escape '`\034`'.

Quando si fa riferimento a un elemento di un array, in cui l'indice sia composto da una serie di valori separati con una virgola, AWK intende che questi valori debbano essere concatenati con il contenuto della variabile ***SUBSEP***.

```
elenco[1, 10] = 123
```

L'esempio appena mostrato equivale in pratica a quello seguente:

```
elenco[1 SUBSEP 10] = 123
```

In generale, non è opportuno modificare il valore di questa variabile, dal momento che si tratta di un carattere decisamente inusuale, allo scopo di garantire che non si possano formare degli indici uguali per elementi che dovrebbero essere differenti.

Per verificare se un elemento di un array del genere esiste, si può utilizzare lo stesso trucco:

```
(indice_1, indice_2, ...) in array
```

23.7 Introduzione a M4

M4 è un elaboratore di «macro», nel senso che la sua elaborazione consiste nell'espandere le macroistruzioni che incontra nell'input. In altri termini, si può dire che copia l'input nell'output, espandendo man mano le macroistruzioni che incontra. La logica di funzionamento di M4 è completamente diversa dai linguaggi di programmazione comuni; inoltre, le sue potenzialità richiedono molta attenzione da parte del programmatore. Detto in maniera diversa, si tratta di un linguaggio macro molto potente, ma altrettanto difficile da gestire.

L’obiettivo di questa introduzione a M4 è solo quello di mostrarne i principi di funzionamento, per permettere la comprensione, parziale, del lavoro di altri, perché in generale **il suo uso è decisamente sconsigliabile**. Per citare un caso significativo, la configurazione di Sendmail viene gestita attraverso una serie di macroistruzioni di M4, con le quali si genera il file ‘/etc/sendmail.cf’.

23.7.1 Principio di funzionamento

«

M4 è costituito in pratica dall’eseguibile ‘**m4**’, la cui sintassi per l’avvio può essere semplificata nel modo rappresentato dallo schema seguente:

```
m4 [opzioni] [file_da_elaborare]
```

Il file da elaborare può essere fornito come argomento, oppure attraverso lo standard input; il risultato viene emesso attraverso lo standard output e gli errori eventuali vengono segnalati attraverso lo standard error.

Per iniziare a comprendere il funzionamento di M4, si osservi il testo seguente:

```
Ciao, come stai ? dnl Che domanda!

# Questo è un commento ? dnl Sì.

Oggi è una giornata stupenda.
```

Supponendo di avere scritto questo in un file, precisamente ‘prova.m4’, lo si può rielaborare con M4 in uno dei due modi seguenti (sono equivalenti):

$ **m4 prova.m4** [*Invio*]

$ **m4 < prova.m4** [*Invio*]

In entrambi i casi, quello che si ottiene attraverso lo standard output è il testo seguente:

```
Ciao, come stai ?
# Questo è un commento ? dnl Sì.

Oggi è una giornata stupenda.
```

Tutto ciò che M4 non riesce a interpretare come una macroistruzione rimane inalterato. Anche se il simbolo di commento è previsto e corrisponde a ‘**#**’ (a meno che siano state usate opzioni o istruzioni particolari), i commenti non vengono eliminati: servono solo a evitare che il testo sia interpretato da M4.

L’unico commento che funzioni in modo simile a quello dei linguaggi di programmazione comuni è la macro ‘**dnl**’ (è stata usata nella prima riga), con la quale viene eliminato il testo a partire da quel punto fino al codice di interruzione di riga successivo. Dal momento che viene eliminato anche il codice di interruzione di riga, si può vedere dall’esempio che la seconda riga, quella vuota, viene inghiottita; invece, il «dnl» contenuto nella riga di commento non è stato considerato da M4.

23.7.2 Convenzioni generali

«

L’analisi di M4 sull’input viene condotta separando tutto in «elementi» (*token*), i quali possono essere classificati fondamentalmente in tre tipi: ***nomi***, ***stringhe*** tra virgolette e caratteri singoli che non hanno significati particolari.

I nomi sono sequenze di lettere (compreso il trattino basso) e numeri, dove il primo carattere è una lettera. Una volta che M4 ha delimitato un nome, se questo viene riconosciuto come una macroistruzione, allora questa viene espansa (sostituendola al nome).

Le stringhe delimitate da virgolette richiedono l’uso di un apice di apertura e di uno di chiusura (‘**`**’ e ‘**'**’). Il risultato dell’elaborazione di una stringa di questo tipo è ciò che si ottiene eliminando il livello più esterno di apici. Per esempio:

```
`'
```

corrisponde alla stringa nulla;


```
`la mia stringa'
```

corrisponde al testo «la mia stringa»;

```
``tra virgolette''
```

corrisponde a «‘tra virgolette’».

È importante tenere presente che anche i simboli usati per delimitare le stringhe possono essere modificati attraverso istruzioni di M4.

Tutto ciò che non rientra nella classificazione di nomi e stringhe delimitate tra virgolette, sono elementi sui quali non si applica alcuna trasformazione.

I commenti per M4 rappresentano solo una parte di testo che non deve essere analizzato alla ricerca di macroistruzioni. Quello che si ottiene è la riproduzione di tale testo senza alcuna modifica. In linea di principio, i commenti sono delimitati dal simbolo ‘**#**’ fino alla fine della riga, cioè fino al codice di interruzione di riga. M4 permette di modificare i simboli usati per delimitare i commenti, o di annullarli del tutto.

È il caso di soffermarsi un momento su questo concetto. Quando si utilizza M4, spesso lo si fa per generare un file di configurazione o un programma scritto in un altro linguaggio. Questi tipi di file potrebbero utilizzare dei commenti, ma può essere conveniente generare nel risultato dei commenti il cui contenuto cambia in funzione di situazioni determinate. Si immagini di voler realizzare uno script di shell, in cui notoriamente il commento si introduce con lo stesso simbolo ‘**#**’, volendo comporre il commento in base a delle macroistruzioni; diventa necessario fare in modo che M4 non consideri il simbolo ‘**#**’ come l’inizio di un commento.

L’unico tipo di dati che M4 può gestire sono le stringhe alfanumeriche, indipendentemente dal fatto che si usino gli apici per delimitarle. Naturalmente, una stringa contenente un numero può avere un significato particolare che dipende dal contesto.

23.7.2.1 Macro

«

M4 è un linguaggio di programmazione il cui scopo principale è quello di gestire opportunamente la sostituzione di testo in base a delle macroistruzioni. Tuttavia, alcune macroistruzioni potrebbero servire a ottenere qualche funzione in più rispetto alla semplice sostituzione di testo. In generale, per uniformità, si parla sempre di «macro» anche quando il termine potrebbe essere improprio; per la precisione si distingue tra macroistruzioni interne (*builtin*), che pur non essendo dichiarate fanno parte di M4, e macroistruzioni normali, dichiarate esplicitamente.

Una macroistruzione può essere «invocata» attraverso due modi possibili:

```
nome
```

```
nome(parametro_1, parametro_2, ... parametro_n)
```

Nel primo caso si tratta di una macroistruzione senza parametri (ovvero senza argomenti); nel secondo si tratta di una macroistruzione con l’indicazione di parametri. È importante osservare che, quando si utilizzano i parametri, la parentesi aperta iniziale **deve seguire immediatamente il nome della macroistruzione** (senza spazi aggiuntivi); inoltre, se una macroistruzione non ha parametri, non si possono utilizzare le parentesi aperta e chiusa senza l’indicazione di parametri, perché questo sarebbe equivalente a fornire la stringa nulla come primo parametro.

La cosa più importante da apprendere è il modo in cui viene trattato il contenuto che appare tra parentesi, che serve a descrivere i parametri di una macroistruzione; infatti, prima di espandere la macroistruzione, viene espanso il contenuto che appare tra parentesi. Una volta espansa anche la macroistruzione con i parametri ottenuti, viene eseguita un’altra analisi del risultato, con il quale si possono eseguire altre espansioni di macroistruzioni, oppure si può ottenere

la semplice eliminazione delle coppie di apici dalle stringhe delimitate. Le operazioni svolte da M4 per espandere una macroistruzione sono elencate dettagliatamente di seguito.

1. Vengono suddivisi gli elementi contenuti tra parentesi ignorando gli spazi iniziali e includendo quelli finali. Si osservi l'esempio seguente:

```
miamacro(a mio, d)
```

Questo è equivalente a:

```
miamacro(a mio,d)
```

2. Vengono espanse le macroistruzioni contenute eventualmente tra i parametri. Continuando l'esempio precedente, si immagini che '**mio**' sia una macroistruzione che si espande nella stringa:

```
, b, c
```

A causa della sostituzione di '**mio**', si ottiene in pratica quanto segue:

```
miamacro(a , b, c,d)
```

Infine, tutto si riduce a:

```
miamacro(a ,b,c,d)
```

Pertanto i parametri sono esattamente una '**a**' seguita da uno spazio e poi le altre lettere '**b**', '**c**' e '**d**'.

3. Una volta risolti i parametri, viene espansa la macroistruzione.

4. Il risultato dell'espansione viene rianalizzato alla ricerca di stringhe delimitate a cui togliere gli apici esterni e di altre macroistruzioni da espandere.

In un certo senso si potrebbe dire che le stringhe, delimitate come previsto da M4, siano delle macroistruzioni che restituiscono il contenuto in modo letterale, perdendo quindi la coppia di apici più esterni. Questo significa che ciò che appare all'interno di una tale stringa non può essere interpretato come il nome di una macroistruzione; inoltre, nemmeno i commenti vengono presi in considerazione come tali. La differenza fondamentale rispetto alle macroistruzioni normali sta nel fatto che l'espansione avviene una volta sola.

Quando si usano le stringhe delimitate tra le opzioni di una macroistruzione normale, è necessario tenere presente che queste vengono trattate la prima volta nel modo appena descritto, allo scopo di fornire i parametri effettivi alla macroistruzione, ma dopo l'espansione della macroistruzione avviene un'ulteriore elaborazione del risultato.

In generale sarebbe conveniente e opportuno indicare i parametri di una macroistruzione sempre utilizzando le stringhe delimitate, a meno di voler indicare esplicitamente altre macroistruzioni. Ciò facilita la lettura umana di un linguaggio di programmazione già troppo complicato. In ogni caso, non si deve dimenticare il ruolo degli spazi finali che vengono sempre inclusi nei parametri. Per esempio, si osservi la macroistruzione '**miamacro**':

```
miamacro(`a' , `b', `c', `d')
```

Questa ha sempre come primo parametro la lettera '**a**' seguita da uno spazio; a nulla serve in questo caso l'uso degli apici, o meglio, sarebbe stato più opportuno usarli nel modo seguente:

```
miamacro(`a ', `b', `c', `d')
```

È il caso di precisare che le sequenze di caratteri numerici sono comunque delle stringhe per M4, per cui '**miamacro(123)**' è perfettamente uguale a '**miamacro(`123')**'. Tuttavia, dal momento che un nome non può cominciare con un numero, non ci possono essere macroistruzioni il cui nome corrisponda a un numero; pertanto si può evitare di utilizzare gli apici di delimitazione perché sarebbe comunque inutile.

Le stringhe delimitate, oltre che per impedire l'espansione di nomi che corrispondono a delle macroistruzioni, permettono di «unire» due macroistruzioni. Si osservi l'esempio seguente:

```
miamacro_x`ciao'miamacro_y
```


L'intenzione è quella di fare rimpiazzare a M4 le macroistruzioni '**miamacro_x**' e '**miamacro_y**' con qualcosa, facendo in modo che queste due parti si uniscano avendo al centro la parola «ciao». Si può intuire che non sarebbe stato possibile scrivere il testo seguente:

```
miamacro_xciaomiamacro_y
```

Infatti, in tal modo non sarebbe stata riconosciuta alcuna macroistruzione. Secondo lo stesso principio, si può unire il risultato di due macroistruzioni senza spazi aggiuntivi, utilizzando apici che delimitano una stringa nulla.

```
miamacro_x`'miamacro_y
```

L'espansione delle macroistruzioni pone un problema in più a causa del fatto che dopo l'espansione il risultato viene riletto alla ricerca di altre macroistruzioni. Si osservi l'esempio seguente, supponendo che la macroistruzione '**miamacro_x**' restituisca la stringa '**miama**' nel caso in cui il suo unico parametro sia pari a '**1**':

```
miamacro_x(1)cro_z
```

Espandendo la macroistruzione si ottiene la stringa «miama», ma dal momento che viene fatta una scansione successiva, la parola «miamacro_z» potrebbe essere un'altra macroistruzione; se fosse questo il caso, la macroistruzione verrebbe espansa a sua volta. Per evitare che accada una cosa del genere si possono usare gli apici in uno dei due modi seguenti:

```
miamacro_x(1)`'cro_z
```

```
miamacro_x(1)`cro_z'
```

Il problema può essere visto anche in modo opposto, se l'espansione di una macroistruzione, quando questa è attaccata a un'altra, può impedire il riconoscimento della seconda. L'esempio seguente mostra infatti che la seconda macroistruzione, '**miamacro_y**', non può essere riconosciuta a causa dell'espansione della prima.

```
miamacro_x(1)miamacro_y
```

Una considerazione finale va fatta sulle macroistruzioni che non restituiscono alcunché, ovvero che si traducono semplicemente nella stringa nulla. Spesso si tratta di macroistruzioni interne che svolgono in realtà altri compiti, come potrebbe fare una funzione *void* di un linguaggio di programmazione normale. In questo senso, per una macroistruzione che non restituisce alcun valore, viene anche detto che restituisce *void*, che in questo contesto è esattamente la stringa nulla.

23.7.2.2 Definizione di una macroistruzione

«

define(***nome_macro*** [, ***espansione***])

Come si può osservare dalla sintassi mostrata, la creazione di una macroistruzione avviene attraverso una macroistruzione interna, '**define**', per la quale deve essere fornito un parametro obbligatorio, corrispondente al nome della macroistruzione da creare, a cui si può aggiungere il valore in cui questa si deve espandere. Se non viene specificato in che modo si deve espandere la macroistruzione, si intende che si tratti della stringa nulla.

La macroistruzione '**define**' non restituisce alcun valore (a parte la stringa nulla). Si osservi l'esempio seguente:

```
1 define(`CIAO', `Ciao a tutti.')
2 CIAO
```

Se questo file viene elaborato da M4, si ottiene il risultato seguente:

```
1
2 Ciao a tutti.
```

Come già affermato, '**define**' crea una macroistruzione ma non genera alcun risultato, pertanto viene semplicemente eliminata.

Per creare una macroistruzione che accetti delle opzioni, occorre indicare, nella stringa utilizzata per definire la sostituzione, uno o più simboli speciali. Si tratta precisamente di '**$1**', '**$2**',... '**$*n***'. Il numero massimo di parametri gestibili da M4 dipende dalla sua versione.

I sistemi GNU dispongono generalmente di M4 GNU [5] e questo non ha limiti particolari al riguardo, mentre le versioni presenti in altri sistemi Unix possono essere limitate a nove.

Questa simbologia richiama alla mente i parametri usati dalle shell comuni; e con la stessa analogia, il simbolo '**$0**' si espande nel nome della macroistruzione stessa.

```
1 define(`CIAO', `Ciao $1, come stai?')
2 CIAO(`Tizio')
```

L'esempio è una variante di quello precedente, in cui si crea la macroistruzione '**CIAO**' che accetta un solo parametro. Il risultato dell'elaborazione del file appena mostrato è il seguente:

```
1
2 Ciao Tizio, come stai?
```

Prima di proseguire è opportuno rivedere il meccanismo dell'espansione di una macroistruzione attraverso un caso particolare. L'esempio seguente è leggermente diverso da quello precedente, in quanto vengono aggiunti gli apici attorno alla parola «come». Il risultato dell'elaborazione è però lo stesso.

```
1 define(`CIAO', `Ciao $1, `come' stai?')
2 CIAO(`Tizio')
```

Infatti, quando la macroistruzione '**CIAO**' viene espansa, subisce una rianalisi successiva; dal momento che viene trovata una stringa, questa viene «elaborata» restituendo semplicemente se stessa senza gli apici. Questo meccanismo ha comunque una fine, dal momento che non ci sono altre macroistruzioni, come si vede nell'esempio seguente:

```
1 define(`CIAO', `Ciao $1, ``come'' stai?')
2 CIAO(`Tizio')
```

Questo si traduce nel risultato:

```
1
2 Ciao Tizio, `come' stai?
```

23.7.2.3 Simboli speciali

«

All'interno della stringa di definizione di una macroistruzione, oltre ai simboli '**$*n***', si possono utilizzare altri codici simili, in un modo che assomiglia a quello delle shell più comuni.

$#

Rappresenta il numero di parametri passati effettivamente a una macroistruzione:

```
define(`CIAO', `$#')
CIAO
CIAO()
CIAO(primo, secondo)
```

L'esempio si traduce nel risultato seguente (si deve tenere presente che una macroistruzione chiamata con le parentesi senza alcun contenuto ha un parametro costituito dalla stringa nulla):

```
0
1
2
```

$*

Rappresenta tutti i parametri forniti effettivamente alla macroistruzione, separati da una virgola, ma soprattutto senza gli apici di delimitazione:

```
define(`ECHO', `$*')
ECHO(uno,   due  , tre)
```

L'esempio si traduce nel modo seguente; si osservi l'effetto degli spazi prima e dopo i parametri:

```
uno,due  ,tre
```

$@

Rappresenta tutti i parametri forniti effettivamente alla macroistruzione, separati da una virgola, con gli apici di delimitazione. La differenza rispetto a '**$***' è sottile e l'esempio seguente dovrebbe permettere di comprenderne il significato:


```
define(`CIAO', `maramao')
define(ECHO1,`$1,$2,$3')
define(ECHO2,`$*')
define(ECHO3,`$@')
ECHO1(CIAO,`CIAO',``CIAO'')
ECHO2(CIAO,`CIAO',``CIAO'')
ECHO3(CIAO,`CIAO',``CIAO'')
```

Le ultime righe del risultato che si ottiene sono le seguenti:

```
maramao,maramao,CIAO
maramao,maramao,CIAO
maramao,CIAO,`CIAO'
```

23.7.2.4 Eliminazione di una macroistruzione

«

Una macroistruzione può essere eliminata attraverso la macroistruzione interna '**undefine**', secondo la sintassi seguente:

```
undefine(nome_macro)
```

L'esempio seguente elimina la macroistruzione '**CIAO**', per cui, da quel punto in poi, la parola '**CIAO**' mantiene il suo valore letterale:

```
undefine(`CIAO')
```

La macroistruzione '**undefine**' non restituisce alcun valore e può essere usata solo con un parametro, quello che rappresenta la macroistruzione che si vuole eliminare.

23.7.3 Istruzioni condizionali, iterazioni e ricorsioni

«

M4 non utilizza istruzioni vere e proprie, dal momento che tutto viene svolto attraverso delle «macro». Tuttavia, alcune macroistruzioni interne permettono di gestire delle strutture di controllo.

Dal momento che il risultato dell'espansione di una macroistruzione viene scandito successivamente alla ricerca di altre macroistruzioni da espandere, in qualche modo, è possibile anche la realizzazione di cicli ricorsivi. Resta il fatto che questo sia probabilmente un ottimo modo per costruire macroistruzioni molto difficili da leggere e da controllare.

23.7.3.1 Macro «ifdef»

«

```
ifdef(nome_macro, stringa_se_esiste[, stringa_se_non_esiste])
```

La macroistruzione interna '**ifdef**' permette di verificare l'esistenza di una macroistruzione. Il nome di questa viene indicato come primo parametro, mentre il secondo parametro serve a definire la stringa da restituire in caso la condizione di esistenza si avveri. Se si indica il terzo parametro, questo viene restituito se la condizione di esistenza fallisce.

L'esempio seguente verifica l'esistenza della macroistruzione '**CIAO**'; se questa non risulta già definita, la crea:

```
ifdef(`CIAO', `', `define(`CIAO', `maramao')')
```

23.7.3.2 Macro «ifelse»

«

```
ifelse(commento)
```

```
ifelse(stringa_1, stringa_2, risultato_se_uguali[, risultato_se_diverse])
```

```
ifelse(stringa_1, stringa_2, risultato_se_uguali, ... [, ↩
↪    risultato_altrimenti])
```

La macroistruzione interna '**ifelse**' serve generalmente per confrontare una o più coppie di stringhe, restituendo un risultato se il confronto è valido o un altro risultato se il confronto fallisce.

Si tratta di una sorta di struttura di selezione ('**case**', '**switch**' e simili) in cui, ogni terna di parametri rappresenta rispettivamente le

due stringhe da controllare e il risultato se queste risultano uguali. Un ultimo parametro facoltativo serve a definire un risultato da emettere nel caso l'unica o tutte le coppie da controllare non risultino uguali.

Nella tradizione di M4, è comune utilizzare '**ifelse**' con un solo parametro; in tal caso non si può ottenere alcun risultato, pertanto questo fatto viene sfruttato per delimitare un commento.

Segue la descrizione di alcuni esempi.

- ```
  ifelse('Questo è un commento')
  ```
  Utilizzando un solo parametro, '**ifelse**' non restituisce alcunché.
- ```
  ifelse('mio', 'mio', 'Vero', 'Falso')
  ```
  Questa istruzione restituisce la parola '**Vero**'.
- ```
  ifelse('mio', 'mao', 'Vero', 'Falso')
  ```
  Questa istruzione restituisce la parola '**Falso**'.

23.7.3.3 Macro «shift»

shift(***parametro*** [,...])

La macroistruzione interna '**shift**' permette di eliminare il primo parametro restituendo i rimanenti separati da una virgola. La convenienza di utilizzare questa macroistruzione sta probabilmente nell'uso assieme a '**$***' e '**$#**'.

```
shift(mio, tuo, suo)
```

Dall'esempio appena mostrato, eliminando il primo parametro si ottiene il risultato seguente:

```
tuo,suo
```

23.7.3.4 Macro «forloop»

forloop(***indice*** , ***inizio*** , ***fine*** , ***stringa_iterata***)

La macroistruzione interna '**forloop**' permette di svolgere una sorta di ciclo in cui l'ultimo parametro, il quarto, viene eseguito tante volte quanto necessario a raggiungere il valore numerico espresso dal terzo parametro. Nel corso di questi cicli, il primo parametro viene trattato come una macroistruzione che di volta in volta restituisce un valore progressivo, a partire dal valore del secondo parametro, fino al raggiungimento di quello del terzo.

```
forloop('i', 1, 7, 'i; ')
```

L'esempio restituisce la sequenza dei numeri da uno a sette, seguiti da un punto e virgola:

```
1; 2; 3; 4; 5; 6; 7;
```

23.7.4 Altre macroistruzioni interne degne di nota

In questa introduzione a M4 ci sono altre macroistruzione interne che è importante conoscere per comprendere le possibilità di questo linguaggio. Attraverso queste macroistruzione, descritte nelle sezioni seguenti, è possibile eliminare un codice di interruzione di riga, inserire dei file, cambiare i delimitatori dei commenti e deviare l'andamento del flusso di output.

23.7.4.1 Macro «dnl»

dnl [***commento***] *new-line*

La macroistruzione interna '**dnl**' è anomala nel sistema di M4: non utilizza parametri ed elimina tutto quello che appare dopo di lei fino alla fine della riga, comprendendo anche il codice di interruzione di riga. La natura di M4, in cui tutto è fatto sotto forma di macroistruzione, fa sì che ci si trovi spesso di fronte al problema di righe vuote ottenute nell'output per il solo fatto di avere utilizzato macroistruzioni interne che non restituiscono alcun risultato. La macroistruzione '**dnl**' serve principalmente per questo: eliminando anche il codice di interruzione di riga si risolve il problema delle righe vuote inutili.

Teoricamente, '**dnl**' potrebbe essere utilizzata anche con l'aggiunta di parametri (tra parentesi). Il risultato che si ottiene è che i parametri vengono raccolti e interpretati come succederebbe con un'altra macroistruzione normale, senza però produrre risultati. Naturalmente, questo tipo di pratica è sconsigliabile.

```
dnl Questo è un commento vero e proprio
define('CIAO', 'maramao')dnl
CIAO
```

L'esempio mostra i due usi tipici di '**dnl**': come introduzione di un commento fino alla fine della riga, oppure soltanto come un modo per sopprimere una riga che risulterebbe vuota nell'output. Il risultato dell'elaborazione è composto da una sola riga:

```
maramao
```

23.7.4.2 Macro «changecom»

changecom([***simbolo_iniziale*** [, ***simbolo_finale***]])

La macroistruzione interna '**changecom**' permette di modificare i simboli di apertura e di chiusura dei commenti. Solitamente, i commenti sono introdotti dal simbolo '**#**' e sono terminati dal codice di interruzione di riga. Quando si utilizza M4 per produrre il sorgente di un certo linguaggio di programmazione, o un file di configurazione, è probabile che i commenti di questi file debbano essere modificati attraverso le macroistruzioni stesse. In questo senso, spesso diventa utile cancellare la definizione dei commenti che impedirebbero la loro espansione.

L'esempio seguente cambia i simboli di apertura e chiusura dei commenti, facendo in modo di farli coincidere con quelli utilizzati dal linguaggio C:

```
changecom('/*', '*/')
```

L'esempio seguente cancella la definizione dei commenti:

```
changecom
```

23.7.4.3 Macro «include» e «sinclude»

include(***file***)

sinclude(***file***)

Attraverso la macroistruzione '**include**' è possibile incorporare un file esterno nell'input in corso di elaborazione. Ciò permette di costruire file-macro di M4 strutturati. Tuttavia, è necessario fare attenzione alla posizione in cui si include un file esterno (si immagini un file che viene incluso nei parametri di una macroistruzione).

La differenza tra '**include**' e '**sinclude**' sta nel fatto che la seconda macroistruzione non segnala errori se il file non esiste.

L'esempio seguente include il file 'mio.m4':

```
include('mio.m4')
```

23.7.4.4 Macro «divert» e «undivert»

M4 consente l'uso di uno strano meccanismo, detto ***deviazione***, o *diversion*, attraverso il quale parte del flusso dell'output può essere accantonato temporaneamente per essere rilasciato in un momento diverso. Per ottenere questo si utilizzano due macroistruzioni interne: '**divert**' e '**undivert**'.

divert([***numero_deviazione***])
```

```
undivert([numero_deviazione[,...]])
```

La prima macroistruzione, '**divert**', serve ad assegnare un numero di deviazione alla parte di output generato a partire dal punto in cui questa appare nell'input. Questo numero può essere omesso e in tal caso si intende lo zero in modo predefinito.

La deviazione zero corrisponde al flusso normale; ogni altro numero positivo rappresenta una deviazione differente. Quando termina l'input da elaborare vengono rilasciati i vari blocchi accumulati di output, in ordine numerico crescente. In alternativa, si può usare la macroistruzione '**undivert**' per richiedere espressamente il recupero di output deviato; se questa viene utilizzata senza parametri, si intende il recupero di tutte le deviazioni, altrimenti si ottengono solo quelle elencate nei parametri.

Esiste un caso particolare di deviazione che serve a eliminare l'output; si ottiene utilizzando il numero di deviazione '**-1**'. Questa tecnica viene usata spesso anche come un modo per delimitare un'area di commenti che non si vuole siano riprodotti nell'output.

Come si può intuire, queste macroistruzioni non restituiscono alcun valore.

Segue la descrizione di alcuni esempi.

- ```
  1 divert(1)
  2 Questo testo è deviato
  3 divert
  4 Questo testo segue l'andamento normale
  ```
  L'esempio si traduce nell'output seguente, dove le righe sono state numerate per facilitarne l'individuazione. Come si può notare, al termine del file di input viene rilasciato l'output deviato precedentemente:
  ```
  1
  4 Questo testo segue l'andamento normale

  2 Questo testo è deviato
  3
  ```
- ```
  1 divert(1)
  2 Questo testo è deviato
  3 divert
  4 Questo testo segue l'andamento normale
  5 undivert(1)
  ```
  Questo esempio è una variante di quello precedente, con la dichiarazione esplicita della richiesta di recupero dell'output deviato. Aggiungendo la macroistruzione '**undivert(1)**', si inserisce anche un'interruzione di riga ulteriore (anche in questo caso vengono numerate le righe per facilitarne l'individuazione nel risultato):
  ```
  1
  4 Questo testo segue l'andamento normale
  5
  2 Questo testo è deviato
  3
  ```
- ```
  divert(-1)
  Quanto qui contenuto non deve dare alcun
  risultato nell'output.
  Le macro generano regolarmente i loro effetti,
  ma il loro output viene perduto.
  divert
  ```
  Questo esempio, mostra l'uso tipico di '**divert(-1)**'. Dal momento che alla fine appare la macroistruzione '**divert**' (senza altre righe), dall'elaborazione di questo file si ottiene solo un codice di interruzione di riga, cioè una riga vuota (quella in cui appare la macroistruzione '**divert**' finale).
- ```
  divert(1)
  Ciao maramao
  divert(2)
  Ciao Ciao
  divert(-1)
  undivert
  ```
  L'uso di '**divert(-1)**' seguito da '**undivert**' permette di eliminare tutto l'output accumulato nelle varie deviazioni.

23.8 Riferimenti

«

- The Open Group, *The Single UNIX ® Specification, Version 2, Regular Expressions*, 1997, *http://pubs.opengroup.org/onlinepubs/007908799/xbd/re.html*
- *Sed tutorials*, *http://maven.smith.edu/%7Ejfrankli/250f00/sed.html*
- *The GNU Awk User's Guide*, Free Software Foundation, *http://www.gnu.org/software/gawk/manual/gawk.html*

[1] **Grep GNU** GNU GPL

[2] **GNU findutils** GNU GPL

[3] **GNU SED** GNU GPL

[4] **Gawk** GNU GPL

[5] **GNU M4** GNU GPL