

Sistemi Operativi, Secondo Modulo, Canale A-L
e Teledidattica
Riassunto della lezione dell'11/03/2019

Igor Melatti



Le espressioni regolari

- Un po' di contesto: parliamo di *linguaggi formali*
 - linguaggio formale = insieme di parole
 - i linguaggi “interessanti” sono infiniti, ovvero con un numero infinito di parole
 - ogni parola è di lunghezza finita, ovvero costituita da un numero finito di *caratteri*
 - * esistono in realtà anche linguaggi dove le parole sono di lunghezza infinita, ma vanno al di là dei nostri scopi

- le parole dei linguaggi formali possono *non* essere come quelle dei linguaggi “naturali”
 - * ad esempio, se si parla del linguaggio C/Java/Python, una parola del linguaggio è *un intero programma C/Java/Python*
 - * cose come **while**, **for** etc, che si sarebbe tentati di chiamare “parole”, sono l’equivalente di singoli caratteri (tecnicamente, fanno parte dei cosiddetti *simboli terminali*)
- ogni linguaggio formale è definito da un insieme di regole, detto *grammatica* del linguaggio
 - * di nuovo, si tratta di grammatica “formale”: è possibile darne una ben precisa (ovvero, definita con linguaggio matematico) sintassi e semantica
 - * la grammatica di un linguaggio naturale, invece, oltre ad ammettere numerose eccezioni è tipicamente “raccontata” in un libro discorsivo (e spesso ambiguo)
 - * da notare che una grammatica *naturale* può dare diverse regole sullo stesso linguaggio:
 - *morfologiche*: come si costruiscono le parole, ad esempio usando le coniugazioni per i verbi o le desinenze per i sostantivi
 - *sintattiche*: come si costruisce una frase (in questo caso, una parola è una frase...), ad es. una regola è soggetto+verbo+complemento, o anche come si costruisce un periodo (in questo caso, una parola è un periodo...), ad es. proposizione principale+subordinata
 - *fonetiche*: essenzialmente la pronuncia delle parole
 - *semantiche*: data una parola (o frase, o periodo) ben formata, come dargli un significato
 - * invece, una grammatica formale (di quelle classiche) si deve necessariamente limitare a sintassi e semantica
 - * formalmente, si dice anche che una grammatica *genera o riconosce* un linguaggio, e che un linguaggio è *generato o riconosciuto* da una grammatica
 - * a differenza dei linguaggi informali (ad es., l’italiano), queste regole sono ferree
 - * a seconda della tipologia di queste grammatiche (completamente generali vs. qualche limitazione vs. ulteriori limitazioni, ...), vengono fuori particolari tipi di grammatiche e quindi di linguaggi riconosciuti da esse
 - * le classi più importanti dal nostro punto di vista sono 2:
 - grammatiche libere dal contesto: sono quelle che definiscono la sintassi dei linguaggi di programmazione (C, Java, Python, ...)

- espressioni regolari: usate in svariati contesti, fondamentali nei sistemi operativi
 - uso tipico: un'espressione regolare può rappresentare un insieme di nomi di files; si può dire al sistema operativo di applicare un'operazione solo a quei files che, facendo *pattern matching* con l'espressione regolare, rientrano in quest'insieme
 - altro uso tipico: mentre le grammatiche libere dal contesto definiscono la sintassi, le espressioni regolari possono descrivere delle sotto-parti del linguaggio come nomi di identificatori o costanti in virgola fissa o mobile
- Cominciamo col dare la *sintassi* delle espressioni regolari; poi passeremo alla *semantica*
 - sintassi: come si scrive un'espressione regolare (in pratica, tramite una sorta di espressione regolare diciamo come si scrivono le espressioni regolari)
 - semantica: cosa vuol dire un'espressione regolare “scritta bene” (ovvero, che rispetti la sintassi)
 - “cosa vuol dire”: qual è l'insieme riconosciuto da un'espressione regolare di un certo tipo
 - nel seguito useremo la tipica espressione “*pattern matching*”: una parola w fa *pattern matching* con una espressione regolare E se e solo se w appartiene al linguaggio generato da E
 - La sintassi delle espressioni regolari cambia (anche se non di molto) da applicazione ad applicazione
 - qui ci concentreremo sulla sintassi prevista dalla Bash, o meglio, da alcuni comandi della Bash
 - altre applicazioni (Python, Perl, Java, Prompt di Windows...) possono usare delle varianti
 - Anche concentrandoci sulla Bash, ci sono (almeno) 2 sintassi per le espressioni regolari
 - basic (BRE), dette anche *obsolete*, usate da programmi “vecchi” come `ed`
 - extended (ERE), dette anche *moderne*, usate dagli altri programmi (`grep`, `sed`, ...)
 - standardizzazione: *POSIX* (Portable Operating System Interface for Unix)
 - è parte integrante di GNU: specifica tutti gli standard cui occorre assoggettarsi per poter essere GNU-compliant

- `man 7 regex`
 - http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html#tag_09_04_03
 - bisogna poi considerare anche il *globbing* (vedere più avanti)
- Iniziamo con le BRE
 - un’espressione regolare è costituita da caratteri stampabili (quelli sulla tastiera “normale”, senza tasti funzione, tabulazione, invio, shift)
 - * si possono aggiungere anche i caratteri non stampabili: vedere http://www.gnu.org/software/bash/manual/html_node/ANSI_002dC-Quoting.html#ANSI_002dC-Quoting
 - alcuni di questi caratteri sono speciali e sono detti *metacaratteri* (*metacharacters*), e sono quelli elencati di seguito: `[,], ., *, ^, $, \`
 - in più, sono metacaratteri anche le seguenti sequenze di caratteri: `\<, \>`
 - si possono raggruppare parti di BRE con `\(, \)`
 - ovviamente, le parentesi `[]` e `\(\)` devono essere aperte e chiuse nel modo corretto
 - tutti gli altri sono detti *letterali* (*literal*)
 - Queste invece sono le ERE (escludendo alcune caratteristiche poco usate, come i collating elements)
 - i metacaratteri sono gli stessi delle BRE, con in più i seguenti: `+, ?, |, (,), {, }`
 - per essere ben formata, le parentesi devono essere correttamente bilanciate
 - inoltre, le parentesi graffe possono essere usate solo nei seguenti modi: `{n}, {n, m}, {n, }`, dove $n \leq m$ sono interi non negativi
 - infine, il backslash può essere usato anche così: `\n`, dove n è un intero positivo (maggiore di 0)
 - *Semantica*: data un’espressione regolare, quali stringhe fanno parte del linguaggio generato? La risposta si ottiene concatenando la semantica di ogni singolo simbolo di un’espressione regolare; vedere Tabelle 1 e 3
 - c’è un ordine definito sui letterali, che viene usato per i range: cifre < lettere minuscole < lettere maiuscole
 - all’interno di ogni categoria, vale il normale ordinamento: $2 < 4$, $a < c$, $D < E$
 - quindi, $9 < a$ e $z < A$

- a rigore, la semantica di un'espressione regolare E sarebbe l'insieme delle stringhe (linguaggio) generato da E
 - per rendere le cose più semplici, descriveremo invece la semantica di un'espressione regolare E sulla base delle stringhe che possono fare pattern matching con E
 - ad esempio, la semantica della BRE $.$ sarebbe l'insieme (finito) $\{a, b, c, d, \dots, 0, 1, 2, \dots\}$; tuttavia, noi diremo più semplicemente che qualsiasi singolo carattere può fare pattern matching con $.$
- Come controllare cosa fa pattern matching: comando `grep`

```

[--color=[always|never|auto]] [-E] [-F] [-i] [-f file_re]
[-r] [-c] [-x] [-n] [-v] [-w] [-o] pattern [file...]

```

 - se non si specifica alcun file, legge da tastiera
 - **pattern** è la BRE (se non c'è l'opzione **-E**) o la ERE (con l'opzione **-E**; talmente comune che esiste **egrep**)
 - normalmente, ritorna un'intera linea (presa o dai file specificati, o da tastiera) se e solo se *contiene* un match con **pattern**
 - con **-v** la ritorna solo se *non contiene* un match con **pattern**
 - con **-x** la ritorna solo se è *esattamente* un match con **pattern** (utile per cercare una riga senza specificare \wedge , $\$$)
 - con **-w** la ritorna solo se il match è una parola intera (utile per cercare una parola senza specificare $\<$, $\>$)
 - con **--color=always** evidenzia la parte di testo che fa match (utile senza **-v** e **-x**)
 - con **-o** ritorna solo la parte di testo che fa match (quindi, solo quella evidenziata con **--color=always**)
 - da notare che **--color=always** e **-o** mostrano *tutti* i match, se ce n'è più d'uno (**-o** li mostra in righe consecutive)
 - quindi, è sufficiente scrivere in un **file** di testo una sequenza di righe, ognuna contenente ciò che si vuole controllare che faccia pattern matching o no
 - poi, eseguire il comando `grep [-E] -x 'pattern' file` (dando **-E** se si vuole testare una ERE)
 - altre opzioni: con **-F** il pattern va matchato esattamente, senza interpretarlo come BRE o ERE; con **-f file_re** il pattern si prende dal file **file_re** (se contiene più righe, ogni riga è un pattern); con **-i** si può fare pattern matching anche se le minuscole/maiuscole sono diverse; con **-r**, se ci sono delle directory negli argomenti di **grep**, allora considera ricorsivamente i files contenuti in tali directory; con **-c** stampa solo il numero di righe che contengono pattern matching; con **-n** stampa anche i numeri delle righe che fanno match

Num	Caratteri	Semantica: con cosa fa pattern matching
1	Letterale l	Solo l
2	.	Qualsiasi singolo carattere
3	$[S]$, dove S è una sequenza (concatenazione) di caratteri dove non compare -, oppure - compare solo all'inizio o solo alla fine	Uno qualsiasi dei caratteri in S ; se il primo carattere di S è \wedge , allora uno qualsiasi dei caratteri <i>non</i> in S
4	$[R]$, dove R è un range l_1-l_2	Uno qualsiasi dei letterali compresi nel range, quindi un qualsiasi $l_1 \leq l \leq l_2$
4bis	$[\wedge R]$, dove R è un range l_1-l_2	Uno qualsiasi dei caratteri <i>non</i> in R , quindi un qualsiasi $l < l_1 \vee l > l_2$
5	$[M]$, dove M è un misto di sequenze S_1, \dots, S_n e di range R_1, \dots, R_k	Uno qualsiasi dei caratteri che fanno pattern matching con una sequenza S_i o con un range R_j . Alcuni di questi M sono predefiniti da POSIX: vedere Tabella 2
6	$[\wedge M]$, dove M è come sopra	Uno qualsiasi dei caratteri che <i>non</i> fanno pattern matching con alcuna sequenza S_i né con alcun range R_j
7	E^*	Si può fare pattern matching con E per 0, 1 o più volte (consecutive); però, se l'asterisco $*$ si trova all'inizio di una BRE, diventa un letterale
8	$\wedge E$	Si può fare pattern matching con E solo se la stringa che fa pattern matching si trova all'inizio del testo
9	$E\$$	Si può fare pattern matching con E solo se la stringa che fa pattern matching si trova alla fine del testo
10	$\backslash < E$	Si può fare pattern matching con E solo se la stringa che fa pattern matching si trova all'inizio di una parola (ovvero, dopo alcuni spazi)
11	$E \backslash >$	Si può fare pattern matching con E solo se la stringa che fa pattern matching si trova alla fine di una parola (ovvero, prima di alcuni spazi)
12	$\backslash m$	Se m è un metacarattere, forza a considerarlo come se fosse un letterale
13	$E_1 E_2$	Si può fare pattern matching con una stringa che fa pattern matching con E_1 , concatenata (ovvero, seguita immediatamente) da una che fa pattern matching con E_2
14	$\backslash (E \backslash)$	Si può fare pattern matching con E ; utile per applicare uno degli operatori di cui sopra ad un'intera sottoespressione

Table 1: Semantica delle BRE

Classe	Semantica: con cosa fa pattern matching
[[:alnum:]]	come la BRE A-Za-z0-9
[[:alpha:]]	come la BRE A-Za-z
[[:blank:]]	Spazio o tab
[[:cntrl:]]	Caratteri di controllo
[[:graph:]]	Caratteri grafici stampabili ASCII 33-126
[[:lower:]]	come la BRE a-z
[[:print:]]	Tutti i caratteri stampabili
[[:space:]]	Tutti i caratteri di spaziatura
[[:xdigit:]]	Cifre esadecimali, come la BRE 0-9A-Fa-f
[[:digit:]]	Cifre decimali, come la BRE 0-9

Table 2: Classi predefinite da POSIX; per usarle, occorre metterle dentro un range

Num	Caratteri	Semantica: con cosa fa pattern matching
1	E^+	Si può fare pattern matching con E per 1 o più volte (consecutive)
2	$E^?$	Si può fare pattern matching con E per 0 o 1 volta
3	$E_1 E_2$	Si può fare pattern matching o con E_1 o con E_2
4	(E)	Si può fare pattern matching con E : utile per occorre sovvertire le precedenze (nelle ERE, la concatenazione ha precedenza sul), per raggruppare delle sottoespressioni cui applicare un operatore, o per il backreferencing
5	$\backslash n$	Esattamente la stessa sottostringa che ha fatto match con l' n -esima sottoespressione regolare tra parentesi (backreferencing)
6	$\{n\}$	Si deve fare pattern matching con la sottoespressione regolare immediatamente precedente per esattamente n volte (consecutive)
7	$\{n, \}$	Si deve fare pattern matching con la sottoespressione regolare immediatamente precedente per almeno n volte (consecutive)
8	$\{n, m\}$	Si deve fare pattern matching con la sottoespressione regolare immediatamente precedente per almeno n volte e per al più m volte (consecutive)

Table 3: Semantica delle ERE

- Esempi a gogò: data una RE, con cosa può far match

1. `.A`

- è una BRE, ottenuta concatenando il metacarattere `.` con il letterale `A`; occorre applicare prima la regola 13 e poi (sostanzialmente insieme) le regole 1 e 2
- quindi: un qualsiasi carattere concatenato (ovvero, seguito immediatamente) dal letterale `A`
- esempi di stringhe che fanno pattern matching: `aA`, `AA`, `3A`, `[A`
- esempi di stringhe che non fanno pattern matching *esatto* (opzione `-x` di `grep`): `AB`, `Aa`, qualsiasi stringa non lunga 2 caratteri...
- esempi di stringhe che non *contengono* un pattern matching: `AB`, `Aa`

2. `0*`

- è una BRE, ottenuta applicando il metacarattere `*` al letterale `0`; occorre applicare la regola 7
- quindi: lo `0` ripetuto quante volte si vuole
- esempi di stringhe che fanno pattern matching: (stringa vuota), `0`, `0000`, ...
- esempi di stringhe che non *contengono* pattern matching: nessuna! perché?

3. `^Un` (da qui in poi per **esercizio**: capire com'è formata sintatticamente, quindi quali regole adottare, e quali sono le stringhe che vengono/non vengono “matchate”)

4. `lo$`

5. `ross[oa]`

6. `[Ll][ue]i`

7. `3*2`

8. `3*2`

9. `[PSps]ed[ai]l[ei]*`

10. che differenza c'è tra `a[bc]*` e `\(a[bc]\)*?`

11. `ab{2}` (da qui sono estese; nota: i costrutti di ripetizione `*`, `+`, `?`, `{}` hanno precedenza sulla concatenazione, che a sua volta ha precedenza sull'alternanza `|`)

12. `ab{2,}`

13. `ab{2,4}`

14. `ab|cd`

15. `([0-9]\1)`

16. `Un[oa]+`

17. Books?

18. [0-9], (132)+

- Esempi a gogò: dato con cosa si vuol far match, creare la RE, specificando se debba essere ERE o possa essere BRE
 1. un numero palindromo di 5 cifre
 - è un numero le cui prime 3 cifre possono essere qualsiasi (ma la prima diversa da 0...), e le ultime 2 invece sono uguali alle prime 2 (ma in ordine inverso)
 - ovvero, se $c_i \in \{0, \dots, 9\}$, $c_1 c_2 c_3 c_2 c_1$, con $c_1 \neq 0$
 - può andare bene la ERE $([[:alnum:]])([[:alnum:]])([[:alnum:]]\2\1?$
 2. un numero in notazione mantissa-esponente, così come lo accetta il programma bc
 - quindi una cosa del tipo
$$c_1 \dots c_k \cdot c_{k+1} \dots c_{k+n} * 10^{c_{k+n+1} \dots c_{k+n+m}}$$
dove k, n non possono essere *entrambi* zero, mentre $m > 0$
 - potrebbe andare la ERE
$$[[:digit:]]* \cdot [[:digit:]]* * 10^{[[:digit:]]+?}$$
 - attenzione: ci sono 2 (o 3) errori nella ERE di cui sopra
 3. un identificatore di Java o Python (e C...; da qui in poi, per **esercizio**)
 4. un URL, completo di protocollo (a scelta tra `http`, `https`, `ftp`, `ftps`); supporre che siano al massimo 4 campi (ed almeno 2), e che gli unici caratteri ammessi siano quelli alfanumerici più - e _
 5. un indirizzo email, con le stesse assunzioni di cui sopra
 6. un numero in virgola fissa
 7. un numero in virgola fissa o in notazione mantissa-esponente
 8. tutte le coniugazioni del verbo *amare*
 9. tutte le coniugazioni del verbo *essere*
- Ultimo appunto teorico: le BRE sono un sottoinsieme di quelli che, nella teoria dei linguaggi formali, sono chiamati *linguaggi regolari*; le ERE, senza l'operatore $\backslash n$ (*back referencing*), sono equivalenti ai linguaggi regolari; le ERE, con il backreferencing, riconoscono una classe di linguaggi che *contiene* quelli regolari
- *Globbering* o *wildcarding*: altre (e diverse) espressioni regolari usate *direttamente* da Bash
 - usata dalle shell (quindi, nel nostro caso, da Bash) per specificare insiemi di file in modo compatto
 - come per le BRE, ci sono i letterali e alcuni metacaratteri: *, ?, [], !, \, ^

- \ è usato per far diventare letterali i caratteri
- * non si riferisce più ad una espressione regolare precedente: semplicemente, dove c'è *, ci può essere una qualsiasi sequenza (anche vuota) di caratteri
 - * quindi è equivalente al .* delle BRE/ERE
- lo stesso vale per ?, ma questa volta il match è con un solo carattere
 - * quindi è equivalente al . delle BRE/ERE
 - * per contro, il . del globbing è un letterale
- [] ha lo stesso significato che ha per le BRE, solo che si può specificare un solo range o una sola sequenza
- [!] fa match con tutto ciò che *non* fa match con il range o la sequenza dati; lo stesso vale per [^]
- ad esempio: `ls [a-z][1-4]*` mostra tutti i files il cui nome inizia con una lettera minuscola seguita da un numero
- piccola eccezione: i file che *iniziano* con il . (file nascosti) non fanno match né con * né con ?
- si può fare wildcarding anche attraverso le directory: `ls [1-3]/*.txt` mostra tutti i file che finiscono in .txt e si trovano in una directory il cui nome è 1, 2 oppure 3
- quello che succede è che la Bash *espande* l'espressione con le wildcard (trovando tutti i file corrispondenti) e li passa al comando dato
 - * ritorneremo con maggiore dettaglio sul concetto di espansione di un comando Bash nelle prossime lezioni
 - * se nessun file o directory fa pattern matching con l'espressione data, allora non avviene alcuna espansione, e l'espressione viene passata al comando Bash così com'è
- ad esempio, se in una directory ci sono 4 files: `.nome1.txt`, `nome1.txt`, `nome2.tgz` e `nome3.txt`, allora scrivendo `ls *.txt` quello che succede è che la Bash espande `*.txt` in `nome1.txt` `nome3.txt`, e poi passa questa stringa ad `ls`; pertanto, in quest'esempio scrivere `ls *.txt` e scrivere `ls nome1.txt nome3.txt` è la stessa cosa
- questo crea non pochi conflitti con le espressioni regolari viste sopra: provare a creare i 4 files detti sopra, e poi dare il comando `grep -E *.txt file...`
- per inibire il wildcarding, usare il *quoting*: `grep -E '*.txt' file`
- **esercizio:** scrivere un'espressione wildcard che venga espansa in tutti i file (o directory) che si trovino in un path del tipo `dir1/dir2/nomefiledir`. Il nome della directory `dir1` è un numero di esattamente 3 cifre decimali, seguito da un trattino e da una stringa qualsiasi, ma non vuota; `dir2` è un numero tra 20 e 29 seguito da un

qualsiasi numero di altri caratteri; ed infine `nomefiledir` è un file non nascosto con estensione `txt` o `tgz`