Expresii regulate în Perl

Caracteristici, operatori, exemple

– Sabin Corneliu Buraga, Victor Tarhon-Onu

Vom încerca în acest articol să tratăm spinosul (considerat de unii) domeniu al expresiilor regulate... Astfel, una dintre cele mai interesante facilități oferite de limbajul Perl este reprezentată de *expresiile regulate* (pentru fundamentele teoretice ale expresiilor regulate, cititorul interesat poate parcurge cartea profesorului Toader Jucan, *Limbaje formale și automate*, Editura Matrix Rom, București, 1999 sau unul dintre articolele, din edițiile PC Report mai vechi, avându-l ca autor pe Mihai Budiu). În fapt, există un număr larg de utilitare și aplicații care încorporează expresiile regulate ca parte a funcționalității interne a respectivelor programe: comenzile UNIX/Linux de procesare a liniilor (`grep`, `sed` sau `awk`), diversele utilitare de sistem sau chiar *shell*-urile din sistemul Linux (*e.g.* `bash`), dar și aplicații rulând pe alte platforme (de exemplu, `UltraEdit`). În afară de Perl, și alte limbaje oferă suport direct pentru expresii regulate, putem da ca exemple Python ori Tcl. De asemenea, există suport (direct sau nu) pentru prelucrarea expresiilor regulate și în limbaje/medii de programare precum C, C++, PHP, Delphi, .NET (prin clasa `Regex`) etc. De asemenea, diverse construcții folosite pentru validarea documentelor XML, via DTD (Document Type Definition) sau schemele XML, folosesc expresiile regulate.

O *expresie regulată* reprezintă un *șablon* (*pattern*) căruia, pe baza unor reguli precise, i se poate asocia unui text.

Pentru lucrul cu expresiile regulate, limbajul Perl pune la dispoziție mai mulți operatori care, pe lângă rolul de delimitare, oferă un set de opțiuni pentru căutare și/sau substituție în cadrul unui text.

Variabila implicită în care se realizează diferite acțiuni implicând expresii regulate este „`$_`“, iar specificarea altei variabile se realizează prin intermediul operatorului „`=~`“.

De notat faptul că, în primele exemple de utilizare pe care le vom da, se vor folosi drept expresii regulate simple șiruri de caractere. Pentru a manipula expresiile regulate, ne vom sluji de o serie de operatori descriși în continuare.

Operatorul m//

Acest operator se folosește pentru a căuta un șablon în cadrul unui text dat. Deoarece de cele mai multe ori nu există nici un pericol de confuzie, caracterul „`m`“ care precedă expresia este opțional. Se returnează valoarea logică „adevărat“ în cazul în care căutarea se încheie cu succes, „fals“ în rest (putem așadar să-l folosim în cadrul expresiilor logice).

```
# atita timp cit se introduce ceva de la tastatura
while (<STDIN>) {
  print "Am gasit subsirul \"Victor\" in $_"
    if m/Victor/;
}
```

Câteva posibile opțiuni ale acestui operator sunt:

- `i` – căutare *case-insensitive* (majusculele nu diferă de minuscule):

```
while (<STDIN>) {
  print "Am gasit tag-ul \"<b>\" sau \"<B>\" in $_"
    if /<b>/i;
}
```

- `s` – tratează textul ca fiind stocat pe o singură linie;
- `m` – caută pe mai multe linii, astfel începutul de șir (desemnat de simbolul „`^`“) și sfârșitul de șir (desemnat de „`$`“) se transformă în început, respectiv sfârșit de linii.
- `g` – caută în șir toate secvențele care se potrivesc șablonului:

```
my $sir = "a b r a c a d a b r a";
my $num_a = 0;
while ($sir =~ /a/g) {
  $num_a++;
}
print "Am gasit de $num_a ori caracterul \'a\'.\n";
# se va afisa "Am gasit de 5 ori caracterul 'a'."
```

- `o` – evaluează șablonul doar o singură dată. Folosirea lui în căutări succesive în același șablon are ca efect creșterea vitezei de căutare.

```
while (<STDIN>) {
  print if /$ARGV[0]/o;
}
```

Vom considera acum același exemplu puțin modificat care primește două argumente diferite în linia de comandă și care le va interschimba succesiv la fiecare parcurgere a buclei:

```
my $nargum = 0;
while (<STDIN>) {
  print if /$ARGV[($nargum++) % 2]/o;
}
```

Rulând aceste două exemple cu aceeași intrare standard și același prim argument vom obține aceeași ieșire, ceea ce demonstrează că șablonul din cadrul expresiei regulate este același încă de la intrarea în bucla `while`.

- `x` – permite utilizarea de spații albe și comentarii în cadrul expresiei regulate cu scopul de a face codul mai lizibil:

```
while (<STDIN>) {
  print if /^e x\ tins # tipareste daca linia incepe
                       # cu 'ex tins'
   /sx;
}
```

În acest exemplu spațiul dinaintea caracterului „`#`“ care precede un comentariu va fi ignorat, la fel ca și spațiul dintre „`e`“ și „`x`“, dar nu și spațiul alb precedat de un caracter backslash „`\`“ dintre „`x`“ și „`t`“.

Operatorul ??

Operatorul `?șablon?` este echivalent cu `/șablon/` cu mențiunea că el nu va returna valoarea „adevărat“ decât la prima căutare reușită dintre două apelări succesive ale operatorului/funcției `reset()`. Rulând următoarea secvență de la *prompt*-ul unui *shell*:

```
(infoiasi)$ perl -e 'print "sablon\n" x 20;' |
            perl -e 'my $i = 1;
              while (<STDIN>) {
                print "$i - $_" if ?sablon?;
                reset if ((++$i) % 7 == 0);
              }'
```

se va obține ieșirea:

```
————-
1 - sablon
7 - sablon
14 - sablon
————-
```

Compararea s-a încheiat cu succes la prima intrare în buclă, atunci când variabila `$i` avea valoarea 1, și mai apoi când numărul liniei era multiplu de 7.

Operatorul s///

Operatorul `s/șablon/text/` permite căutarea și substituția unui șablon cu un text. Un exemplu simplu:

```
while (<STDIN>)
  {
  s/netreprot/NETReport/i;
  print;
}
```

Se va înlocui cuvântul `netreprot` dat la intrarea standard cu `NETReport` și se vor scrie la ieșirea standard toate liniile citite, indiferent dacă substituția s-a efectuat sau nu.

Operatorul qr//

Acest operator primește drept parametru de intrare un șir de caractere pe care îl precompilează ca expresie regulată. Expresia regulată precompilată poate fi memorată într-o variabilă și refolosită în construcția altor expresii regulate sau poate fi utilizată direct:

```
my $expr = qr/(netreprot|netrepotr)/i;
# exemplu de folosire directa
while (<STDIN>) {
  s/$expr/NETReport/;
  print;
}
```

Delimitarea expresiei regulate și, dacă este cazul, a șirului substituitor se poate realiza cu alte caractere speciale decât „/“. Astfel, secvența de cod de mai jos este perfect valabilă:

```
my $text = "csc.exe /Help";
# inlocuim "/Help" cu "-help"
print "$text\n" if $text =~ s!/Help!-help!;
```

Desigur, în loc de „!“ putem folosi oricare alt caracter (*e.g.* „#“). Un alt exemplu, care va afișa conținutul tuturor elementelor `<pre>` dintr-un document HTML (caracterul „/“ nu mai putea fi folosit ca delimitator de expresie regulată):

```
while (<>) {
  print if m#<pre>#i .. m#</pre>#i;
}
```

În acest exemplu, remarcăm și folosirea operatorului „..“ care se dovedește extrem de util aici pentru extragerea unui interval de linii fără a se reține explicit aceste informații.

Secvențe pentru identificarea unui caracter. Multiplicatori

Cel mai simplu mod de a identifica un anumit caracter în cadrul unui text este cel de a căuta exact acel caracter. Scrierea unui caracter „a“ într-o expresie regulată presupune existența aceluiași caracter în șirul căruia i se aplica. De multe ori însă am dori să folosim diverse meta-caractere pentru a identifica un set de caractere.

Meta-caracterele sunt acele caractere din codul ASCII care nu se identifică pe ele însele în cadrul unei expresii regulate sau chiar al unei secvențe de cod-sursă (scrisă în C/C++, Perl etc). În cadrul unei expresii regulate meta-caracterele sunt folosite pentru alcătuirea unor construcții cu rol de a identifica diferite secvențe de caractere dintr-un text.

Putem considera specificatorii de fișier UNIX/Linux drept expresii regulate folosind un set restrâns de meta-caractere.

Următoarele caractere ASCII sunt considerate meta-caractere în cadrul unei expresii regulate:

- *Caracterul* „`.`“ este utilizat în cadrul unei expresii regulate să identifice orice caracter, exceptând caracterul *newline* „`\n`“.
- *Construcția* „`[...]`“ reprezintă o clasă de caractere. De exemplu, expresia regulată `/[a-z]/` se poate asocia oricărui șir de caractere care conține cel puțin o literă minusculă. Bineînțeles, se pot concepe construcții mai complexe, cum ar fi `/[a-z][ATX][0-7]\-[^1-378]/` care va identifica orice text conținând o succesiune de caractere formată, în ordine, de o literă minusculă, una dintre majusculele „`A`“, „`T`“ sau „`X`“, o cifră între 0 și 7, semnul minus „`-`“ și oricare cifră diferită de 1, 2, 3, 7 sau 8.
- *Meta-caracterul* „`^`“ joacă două roluri:
- Folosit în cadrul unei secvențe de caractere are rolul de negare. Astfel, `[^2-5]` va identifica oricare cifră aparținând mulțimii {1, 6, 7, 8, 9, 0}, iar `[^a-m]` va identifica oricare caracter cu excepția minusculelor de la „`a`“ la „`m`“.
- Desemnează începutul unei linii, fiind un caracter de poziționare în rest. De exemplu, `^[2-5]` va identifica orice șir care începe cu o cifră cuprinsă între 2 și 5. Precedat de un „`\`“ va desemna caracterul „`^`“.
- *Simbolul* „`$`“ folosit într-o expresie regulată identifică sfârșitul unei linii.
- *Caracterele* „`(`“ *și* „`)`“ au rolul de a grupa atomi în cadrul expresiei regulate și de a memora valoarea subșirurilor din text corespunzătoare acestor atomi, fără însă a modifica valoarea expresiei regulate (această construcție se mai numește și *operator de memorare*). Un caracter lipsit de semnificație sau un meta-caracter de poziționare (*e.g.* „`^`“ sau „`$`“) se numește *atom* al unei expresii regulate. Putem grupa atomi pentru a forma fragmente ale unei expresii regulate mai complexe.

Exemple:

```
my $text = "acesta
este
un text
pe
mai multe
linii";

while ($text =~ /^([a-k].*)$/gm) {
  print "Linia \"$1\" incepe cu un caracter de la \'a\' la
        \'k\'\n";
}
```

Fie *script*-ul Perl:

```
my ($LOG, $i);
open(LOG, ">>/tmp/word_switch.log") ||
  die "Nu pot deschide fisierul: $!\n";
while (<STDIN>) {
  $i++;
  s/^(\S+)\s+(\S+)/$2 $1/;
  print;
  print LOG "linia $i: s-a inversat \"$1\" cu \"$2\"\n";
}
close(LOG);
```

Acest program va prelua linii de text de la intrarea standard și va afișa la ieșirea standard inversând primele două cuvinte între ele, scriind într-un fișier modificările efectuate.

- Caracterul „|“ va da posibilitatea de a alterna între două sau mai multe forme posibile ale unei secvențe dintr-un text.

```
while (<STDIN>) {
  print if (/[0-9]|[A-Z][a-z]/);
}
```

- Meta-caracterele „?“, „*“, „+“, „{“ și „}“ au rolul de *multiplicatori* ai unui atom. Un atom al unei expresii regulate urmat de „?“ poate identifica de zero sau maxim o singură dată un atom. Simbolul „*“ poate identifica zero, una sau mai multe apariții consecutive ale aceluiași atom, iar un atom urmat de „+“ poate să identifice măcar o apariție a atomului.

  Un exemplu:

```
while (<STDIN>) {
  print "Cel putin o aparitie a cuvintului \'web\' la
         inceputul liniei\n"
    if (/^(web)+/);
}
```

Multiplicatorul „{}“ are o sintaxă ceva mai complexă decât „*“ și „+“, astfel:

- `atom{m, n}` va identifica într-o expresie cel puțin `m` atomi, dar nu mai mulți de `n`;
- `atom{m,}` va identifica `m` sau mai mulți atomi;
- `atom{,n}` va identifica `n` atomi cel mult;
- `atom{n}` va identifica exact `n` atomi.

Putem face aici observația că `{1,}` este echivalent cu „+“, `{0,1}` cu „?“, iar construcția `{0,}` este echivalentă cu „*“.

Limbajul Perl pune la dispoziție un set de construcții predefinite pentru identificarea unor clase de caractere, după cum se poate remarca din tabelul „Clase de caractere“.

Alți identificatori de caractere

Limbajul Perl mai pune la dispoziție, alături de construcțiile predefinite pentru identificarea unor clase de caractere, și construcții conforme cu standardul POSIX, de forma `[[:clasa_de_caractere:]]`, utilizate, de exemplu, și de funcțiile PHP.

Clasele de caractere (care pot fi utilizate ca mai sus) sunt: `alpha`, `alnum`, `ascii`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `word` și `xdigit`. Caracterele incluse în aceste clase de caractere sunt cele pentru care funcțiile C cu numele `isclasa_de_caractere()` returnează „adevărat“ (pentru amănunte, vezi antetul C standard `ctype.h`). Astfel, `[[:alnum:]]` este echivalentă cu `[0-9a-zA-Z]`, construcția `[[:word:]]` este echivalentă cu `[0-9a-zA-Z_]`, iar `[[:digit:]]` cu `[0-9]` etc.

De asemenea, limbajul Perl definește construcții cu lungime nulă (*zero-width assertions*) care identifică doar contexte, nu caractere, în următorul mod:

- `\b` identifică limitele unui cuvânt;
- `\B` identifică orice alt context decât limitele unui cuvânt (interiorul unui cuvânt);
- `\A` desemnează începutul unui șir;
- `\Z` identifică sfârșitul unui șir sau înaintea caracterului *newline* de la finalul șirului;
- `\z` identifică sfârșitul unui șir;
- `\G` va identifica locul unde are loc ultima potrivire a șablonului în text, în cazul folosirii opțiunii `/g` a operatorilor `m//` sau `s///`.

De exemplu „`/Report\b/`“ poate identifica „`Report`“, „`NETReport`“, dar nu și „`Reportul`“.

Variabile speciale și expresii regulate

După cum s-a putut observa în unele dintre exemplele precedente, la evaluarea expresiilor regulate se setează variabilele speciale `$1`, `$2` etc., atunci când șablonul conține referințe anterioare (*back-references*). Astfel de referințe sunt utile pentru specificarea de sub-expresii care au satisfăcut un șablon de mai multe ori. Astfel, pentru a avea acces la părți ale șirului sau chiar la întregul șir care satisface o anumită expresie regulată se folosesc parantezele rotunde „`()`“, șirurile de caractere care reprezintă rezultatul fiind stocate în variabilele speciale numerotate. Ca regulă generală, putem specifica referințele anterioare prin construcții precum „`\1`“, „`\2`“ și așa mai departe, până la „`\9`“. În Perl, putem folosi mai lejer variabilele speciale `$1`, `$2`, al căror număr nu este limitat.

Următorul program va afișa, pentru o adresă *e-mail*, numele de cont și adresa simbolică a mașinii (am folosit apostrofuri în loc de ghilimele pentru ca simbolul „@“ să nu fie interpretat drept prefix al unui tablou):

```
$_ = 'Contactati-ne la <cgi@infoiasi.ro>';
if (/<(.*)\@(.*)>/) {
  print "Numele de cont:   $1\n";
  # desemneaza prima sub-expresie (.*)
  print "Adresa simbolica: $2\n";
  # desemneaza a doua sub-expresie (.*)
}
```

De asemenea, programatorii Perl mai au la dispoziție următoarele variabile speciale asociate expresiilor regulate:

- `$+` va conține șirul care corespunde ultimii paranteze evaluate din expresia regulată;
- `$&` va desemna întrgul șir corespunzând expresiei regulate;
- `$`` va conține toate caracterele care precedă șirului care se potrivește expresiei regulate;
- `$'` va conține toate caracterele care urmează șirului corespunzător expresiei regulate.

Funcții predefinite folosind expresii regulate

În conjuncție cu expresiile regulate se pot utiliza următoarele funcții predefinite:

- `tr///` realizează translatarea caracter cu caracter a unui text și are forma următoare:
- `tr/caractere_de_căutare/caractere_de_înlocuire/`

Clase de caractere

Construcție	Echivalent	Construcție complementară	Echivalentul construcției complementare
\d (o cifră)	[0-9]	\D (orice exceptând cifre)	[^0-9]
\w (un caracter alfanumeric)	[0-9_a-zA-Z]	\W (un caracter ne-alfanumeric)	[^0-9_a-zA-Z]
\s (un spațiu alb)	[\t\r\n\ \f]	\S (orice exceptând spații albe)	[^\t\n\r\ \f]

Această funcție mai poartă numele și de funcție de *transliterare*.

Șirul de intrare este parcurs caracter cu caracter, înlocuindu-se fiecare apariție a unui caracter de căutare cu corespondentul lui din mulțimea caracterelor de înlocuire.

Se pot folosi opțiunile:

- `c` – va realiza translatarea utilizând complementara mulțimii de caractere de căutare;
- `d` – va șterge toate caracterele din mulțimea caracterelor de căutare care nu au corespondent în setul caracterelor de înlocuire;
- `s` – va reduce secvențele de caractere care au fost înlocuite folosindu-se același caracter la o apariție unică a caracterului respectiv.

Câteva exemple:

```
# majusculele devin minuscule
tr/A-Z/a-z/
# http: devine ftp:
tr/http:/ftp:/
# caracterele diferite de alfanumerice devin spatii
tr/A-Za-z0-9/ /cs
```

- `split()` împarte un șir de caractere în funcție de o expresie regulată și returnează un tablou conținând subșirurile care nu satisfac acea expresie regulată.

Pentru a afișa numele de cont și directorul *home* al utilizatorilor din sistem vom putea scrie următorul *script* (folosim fișierul `/etc/passwd`):

```
open (FIS, "/etc/passwd") ||
  die "Eroare la deschiderea fisierului\n";
while (<FIS>) {
  $linie = $_;
  chomp($linie);
  @date = split(':', $linie);
  ($cont, $dir) = @date[0, 6];
  print "Directorul home al lui $cont este $dir\n";
}
close (FIS);
```

Elementele returnate de `split()` se pot memora și în variabile scalare separate. Astfel, pentru a stoca data sistemului local vom putea scrie următoarele linii de cod:

```
$data_sistem = localtime(time);
($ziua, $luna, $num, $timp, $an) = split(/\s+/,
   $data_sistem);
```

Funcția `split()` poate fi utilă, de asemenea, în cadrul *script*-urilor CGI la analiza numelor și valorilor parametrilor obținuți în urma completării unui formular Web.

- `join()` este complementară funcției mai sus amintite, în sensul că reunește mai multe șiruri de caractere în unul singur, delimitate de o valoare având ca tip un scalar.

Un exemplu:

```
$perl = "Perl";
$cpp  = "C++";
$java = "Java";
$tcl  = "Tcl";
print "Limbaje: ", join(" ", $perl, $cpp, $java, $tcl), "\n";
print "Limbaje: ", join(", ", $perl, $cpp, $java, $tcl), "\n";
```

- `eval()` poate fi folosită pentru evaluarea/execuția unei expresii Perl. Contextul execuției expresiei Perl este contextul curent al programului. Putem considera expresia ca o subrutină sau bloc de instrucțiuni în care toate variabilele locale au timpul de viață egal cu cel al subrutinei ori blocului. Dacă nu se specifică o expresie ca argument al funcției, se va utiliza în mod firesc variabila specială `$_`. Valoarea returnată de `eval()` reprezintă valoarea ultimei expresii evaluate, fiind permisă și folosirea operatorului `return`.

Posibilele erori vor cauza returnarea unei valori nedefinite și setarea variabilei `$@` cu un mesaj de eroare. Astfel, `eval()` poate fi de ajutor în verificarea corectitudinii unui șablon:

```
sub este_sablonul_valid {
     my $sablon = shift;
     return eval { "" =~ /$sablon/; 1 } || 0;
}
```

Putem preîntâmpina afișarea unui mesaj de eroare la apariția excepției de împărțire la zero a unui număr astfel:

```
print "Impartire la zero"
  unless eval { $rezultat = $nr1 / $nr2 };
```

La final...

Desigur, multe alte detalii referitoare la manipularea expresiilor regulate în Perl nu au fost dezvăluite... Cititorul interesat poate parcurge documentația pusă la dispoziție de orice distribuție Perl actuală (ne referim mai ales la secțiunea dedicată expresiilor regulate – încercați în Linux, de exemplu, `man perlre`) sau referințele bibliografice. Nu ne mai rămâne acum decât să vă urăm succes!

Sabin-Corneliu Buraga este doctorand în Computer Science la Facultatea de Informatică, Universitatea „Al.I.Cuza" din Iași și poate fi contactat la adresa busaco@infoiasi.ro. Victor Tarhon-Onu este student la Facultatea de Automatică și Calculatoare, Universitatea Tehnică „Gh. Asachi" din Iași și administrator de rețea la RDS (filiala Iași), putând fi contactat la adresa mituc@ac.tuiasi.ro. ■ 98

Referințe bibliografice

- M. Budiu, **Expresii regulate**, PC Report, vol.9, 04 (91), apr. 2000
- S. Buraga, V. Tarhon-Onu, Ș. Tanasă, **Programare Web în bash și Perl**, Editura Polirom, Iași, 2002: `http://www.infoiasi.ro/~cgi/`
- S. Buraga, **Script-uri CGI în Perl**, NET Report, vol.11, 02 (113), feb. 2002
- T. Jucan, **Limbaje formale și automate**, Editura Matrix Rom, București, 1999
- L. Wall **et al.**, **Programming Perl** (Third Edition), O'Reilly & Associates, Cambridge, 2000
- R. Schwartz, T. Christiansen, **Learning Perl** (Second Edition), O'Reilly & Associates, Cambridge, 1997
- Ș. Trăușan-Matu **et al.**, **Prelucrarea documentelor folosind XML și Perl**, Editura Matrix Rom, București, 2001
- ***, *Learning to Use Regular Expressions*: `http://gnosis.cx/publish/programming/regular_expressions.html`
- ***, *Perl Monks*: `http://www.perlmonks.com/`
- ***, *Perl Month*: `http://www.perlmonth.com/`
- ***, *The Perl Journal*: `http://www.tpj.com/`
- ***, *Scripts*: `http://www.worldwidemart.com/scripts/`