Ficha 4

Scripting no Processamento de Linguagem Natural

15 de Março de 2015

1 Expressões Regulares

Esta ficha introduz um dos componentes mais poderosos das linguagens ditas de *scripting* tal como o Perl: as expressões regulares. As expressões regulares permitem uma manipulação bastante sofisticada de texto.

2 Metacaracteres

`\`	Tira o significado especial ao próximo caracter
`^`	Procura o início da linha
`.`	Procura qualquer caracter (excepto `\n`)
`$`	Procura o fim da linha
`|`	Alternância
`()`	Agrupamento
`[]`	Classe de caracteres

No caso das classes de caracteres, `[a-z]` são todos os caracteres entre `a` e `z` enquanto que `[^a-z]` são todos os caracteres *excepto* os caracteres entre `a` e `z`

3 Quantificadores

`*`	Zero ou mais vezes
`+`	Uma ou mais vezes
`?`	Zero ou uma vez
`{n}`	n vezes
`{n,}`	Pelo menos n vezes
`{n,m}`	Entre n e m vezes

A quantificação é *avarenta*. Isto é, tenta apanhar o máximo possível desde que o resto da expressão regular ainda possa funcionar. Assim, quando se usa a expressão regular `a.+c` para apanhar uma parte da string `abbbbbcdcf` o padrão encontrado é `abbbbbcdc`. Para não ser avarenta é necessário usar o metacaracter `?` a seguir ao quantificador. Assim, a expressão regular `a.+?c` para apanhar uma parte da string `abbbbbcdcf` encontra o padrão `abbbbbc`.

`*?`	Versão não avarenta de zero ou mais vezes
`+?`	Versão não avarenta de uma ou mais vezes
`??`	Versão não avarenta de zero ou uma vez
`{n}?`	Versão não avarenta de n vezes
`{n,}?`	Versão não avarenta de pelo menos n vezes
`{n,m}?`	Versão não avarenta de entre n e m vezes

Quando um subpadrão quantificado não deixa o resto da expressão regular funcionar o Perl volta atrás e tenta outra possibilidade. Existe também uma versão *possessiva* dos operadores que impede esse comportamento. Nesta nova versão, o padrão quantificado apanha o máximo possível *mesmo que* o resto não possa funcionar.

`*+`	Versão possessiva de zero ou mais vezes
`++`	Versão possessiva de uma ou mais vezes
`?+`	Versão possessiva de zero ou uma vez
`{n}+`	Versão possessiva de n vezes
`{n,}+`	Versão possessiva de pelo menos n vezes
`{n,m}+`	Versão possessiva de entre n e m vezes

Assim, o comando

```
'aaaa' =~ /a++a/;
```

não pode funcionar porque a expressão regular `a++` apanha a cadeia inteira `aaaa`.

Existe um conjunto de classes de caracteres que são tão utilizadas que até possuem atalhos específicos.

`\w`	Caracteres alfanuméricos, o mesmo que `[a-zA-Z0-9_]`
`\W`	O oposto de `\w`, isto é `[^a-zA-Z0-9_]`
`\s`	Espaçamentos, o mesmo que `[ \t\r\n\v\h]`
`\S`	O oposto de `\s`
`\d`	Dígitos, i.e., `[0-9]`
`\D`	Oposto de `\d`
`\1 até \9`	Referência a um grupo capturado
`\b`	Limite de uma palavra
`\B`	Interior de uma palavra

O último precisa de explicação adicional: quando uma expressão regular contém parentesis, a string que é contemplada por essa expressão é armazenada numa variável especial (`$1` até `$9`). A numeração é dada pela ordem que aparece o parentesis esquerdo. O exemplo a seguir usa um conceito que só veremos mais à frente para imprimir (no ciclo `for`) as nove variáveis (`$1` até `$9`).

```
"-90.017" =~ /((-?)(\d+)(\.(\d+))?)/;

for $x (1..9) {
       print "\$$x\t$$x\n";
}
```

Agora imagine que quer verificar quais de entre um conjunto de strings são um número repetido. Para isso poderia utilizar o seguinte programa.

```
@palavras = qw{1010 101 91919 919919};
for $palavra (@palavras) {
```

```
    print "$palavra\n" if $palavra =~ /^(\d+)\1$/;
}
```

4 Variáveis

Para além das variáveis `$1` até `$9` existem mais algumas interessantes e por isso vamos apresentá-las:

`$&`	A string que fez match
`$‘`	A string antes da parte que fez match
`$’`	A string depois da parte que fez match
`$1 até $9`	Referência a um grupo capturado

O exemplo seguinte demonstra a utilização dessas variáveis.

```
"num= -90.017 metros" =~ /((-?)(\d+)(\.(\d+))?)/;

for $x (1..9) {
       print "\$$x\t$$x\n";
}

print "antes: <$‘>\nmatch: <$&>\ndepois :<$’>\n";
```

5 Match em Ambiente Lista

Outra forma útil de utilizar os grupos tem a ver com usar o operador `m//` em ambiente de lista.

```
@exemplos = qw{-90.017e39 104 73.12 2.07E-8};

for $_ (@exemplos) {
       @grupos = /((-?)(\d+)(\.(\d+))?((e|E)(-?)(\d+))?)/;

       print join ":", @grupos;
       print "\n";

       ($tudo, $sinal, $int, $resto, $frac, $exp, $e, $sinal_exp, $exp_sem_sinal) = /((-?)
           (\d+)(\.(\d+))?((e|E)(-?)(\d+))?)/;

       print "Tudo: $tudo\n";
       print "Sinal: $sinal\n";
       print "Parte inteira: $int\n";
       print "Parte fracionaria: $frac\n";
       print "Expoente: $exp\n";
       print "Sinal Expoente: $sinal_exp\n";
       print "Valor do expoente: $exp_sem_sinal\n\n";
}
```

6 Opções da procura

O operador `m//` também conhecido como *match* tem várias opções:

s Tratar a string como uma única linha;

m Trata a string como múltiplas linhas;

i Ignorar as minúsculas/maiúsculas;

g Procura global;

c Guarda a posição actual mesmo que a procura falhe;

p Preserva as três variáveis `${^PREMATCH}`, `${^MATCH}`, e `${^POSTMATCH}` depois da procura;

Segue-se um exemplo da opção `g`. Esta pode ser utilizada de várias maneiras tal como se pode constatar.

```
@a = "ser ou nao ser eis a questao" =~ /(\S+)/g;
print join ":", @a;
print "\n";

print "$&\n" while("ser ou nao ser eis a questao" =~ /(\S+)/g);
```

7 Opções Específicas de Substituição

O operador `s///` tam algumas opções específicas (i.e., para além das opções que já vimos antes):

r Não destructivo, em vez de modificar a string retorna a string modificada;

e Avalia o lado da substituição;

ee Pode ser usado mais do que uma vez, por cada e avalia o resultado;

Seguem-se alguns exemplos para ajudar a perceber as opções.

```
$_ = 'abc123xyz';
$a=s/abc/def/r; # depende da versao do perl, pode nao funcionar
s/\d+/$&*2/e; # resultado: 'abc246xyz'

# o operador x replica o primeiro argumento varias vezes, por exemplo
# "xyz" x 3 da "xyzxyzxyz"
s/\w/$& x 2/eg; # resultado: 'aabbcc224466xxyyzz'
```

8 Exercícios

1. Escreva um programa (usando o operador de substituição) que leia uma frase e para cada palavra coloque só a inicial seguida de um ponto, isto é, para a frase `viva o perl` o programa imprima `v. o. p.`

2. Escreva um programa que se ler

```
8144658695
812 673 5748
812  453   6783
812-348 7584
(617) 536 6584
834-674-8595
```

   Escreva

```
814 465 8695
812 673 5748
812 453 6783
812 348 7584
617 536 6584
834 674 8595
```

3. Escreva um programa que reformate as datas do formato `DD/MM/YYYY` para `MM/DD/YYYY` no seguinte texto

```
Zaphod, Beeble, 23/12/1994
Dent, Arthur, 05/04/1993
Feynman, Richard, 18/02/1956
```

4. Faça um programa para extrair endereços de email de um texto. a.b@c.d and a@b.com são endereços válidos mas a.com, a@b e @b.com não são. Experimente o seu programa com o seguinte texto (que tem quatro endereços):

```
A towel, it says, is about the most massively useful thing an
interstellar hitchhiker can have. Partly it has great practical value
- you can wrap it around you dent@vogon.com for warmth as you bound across the cold
moons of Jaglan Beta; you can lie on it on the brilliant marble-sanded
beaches of Santraginus V, foo@bar.bar.com inhaling the heady sea vapours; you can
sleep under it beneath the stars which shine so redly on the desert
world of Kakrafoon; use it john.smith@blah.org to sail a mini raft down the slow heavy
river Moth; wet it for use in hand-to- hand-combat; wrap it round your
head to ward off glom@flop.net noxious fumes or to avoid the gaze of the Ravenous
Bugblatter Beast of Traal (a mindboggingly stupid animal, it assumes
that if you can't see it, it can't see you - daft as a bush, but very
ravenous); you can wave your towel in emergencies as a distress
signal, and of course dry yourself off with it if it still seems to be
clean enough.
```

5. Faça um programa que leia um texto com linhas do género `identificador=valor` e crie um hash com a informação correspondente;

6. Reescreva o programa que torna maiúscula a primeira letra de todas as palavras de uma frase usando só uma substituição (e sem usar a função `ucfirst`);

7. Escreva uma script que leia ficheiros BibTeX e que produza uma estrutura de dados com os dados correspondentes aos artigos. Por exemplo, para o texto:

```
@article{AscRuuSpi97,
  title =       {Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations},
  author =        {Ascher and Ruuth and Spiteri},
  journal =      "Applied Numerical Mathematics",
  volume =       "25",
  year =         "1997",
  pages =        "151--167"
}
@article{storn:de-article,
  author  = "R. Storn and K. Price",
  title   = "Differential Evolution - A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces",
  journal = {Journal of Global Optimization},
  volume  = {11},
  pages   = {341-359},
  year    = {1997}
}

@techreport{storn:de-techreport,
  author = "R. Storn and K. Price",
  title  = "Differential Evolution - A Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Space",
  institution = "The International Computer Science Institute",
  year = "1995"
}


@InProceedings{storn:usage,
  author       = "R. Storn",
  title        = "On the Usage of Differential Evolution for Function Optimization",
  booktitle    = "1996 Biennial Conference of the North American Fuzzy Information Processing Society (NAFIPS 1996)",
  pages        = "519-523",
  publisher    = "IEEE",
  year         = "1996"
}
```

A estrutura de dados deveria ser algo do género:

```
$VAR1 = {
 'storn:de-article' => {
    'volume' => '11',
    'title' => 'Differential Evolution - A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces',
    'author' => 'R. Storn and K. Price',
    'type' => 'article',
    'pages' => '341-359',
    'journal' => 'Journal of Global Optimization',
    'key' => 'storn:de-article'
  },
 'AscRuuSpi97' => {
     'volume' => '25',
     'author' => 'Ascher and Ruuth and Spiteri',
     'title' => 'Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations',
     'type' => 'article',
     'year' => '1997',
     'journal' => 'Applied Numerical Mathematics',
     'key' => 'AscRuuSpi97'
   },
 'storn:usage' => {
     'booktitle' => '1996 Biennial Conference of the North American Fuzzy Information Processing Society (NAFIPS 1996)',
     'title' => 'On the Usage of Differential Evolution for Function Optimization',
     'author' => 'R. Storn',
     'type' => 'inproceedings',
     'publisher' => 'IEEE',
     'pages' => '519-523',
     'key' => 'storn:usage'
   },
 'storn:de-techreport' => {
    'title' => 'Differential Evolution - A Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Space',
    'author' => 'R. Storn and K. Price',
    'type' => 'techreport',
    'institution' => 'The International Computer Science Institute',
    'key' => 'storn:de-techreport'
     }
 };
```