Un pèl de Perl

Perl per lingüistes

Laura Alonso i Alemany
alemany@famaf.unc.edu.ar

http://www.cs.famaf.unc.edu.ar/~alemany/un_pel_de_Perl

Resum

Aquest és un curs instrumental orientat a lingüistes que volen aprendre Perl per a tractar textos de manera automàtica.

Perl és un llenguatge imperatiu que resulta especialment eficaç per al processament del llenguatge natural principalment per dues raons: el seu gran poder expressiu pel que fa a *expressions regulars* i les seves estructures de dades predefinides, especialment els arrays associatius o *hashos*. Com a contrapartida, no podem exercir un control directe sobre l'eficiència dels programes, i tampoc no resulta especialment adequat a la orientació a objectes.

Al final del curs, els lingüistes podran:

- formalitzar problemes en llenguatges de programació imperatius
- dominar estructures de dades bàsiques i les seves aplicacions al tractament de corpus
- dominar expressions regulars
- desenvolupar utilitats bàsiques d'exploració i descripció de corpus textuals

Índex

1 Formalització de problemes

Els llenguatges imperatius, com *C* o Perl, prenen una orientació algorísmica a la resolució de problemes, és a dir, estan orientats a la **realització d'accions**. Un algorisme no és res més que una seqüència de passos orientada a la consecució d'un objectiu.

La major part de problemes que volem solucionar amb programes informàtics es poden abordar tant amb llenguatges imperatius o declaratius (com per exemple els llenguatges lògics com ProLog o els funcionals com Lisp o Haskell). Però alguns tipus de problemes es resolen amb més eficàcia i/o eficiència amb llenguatges imperatius, sobretot quan es tracta de realitzar accions concretes i senzilles, com per exemple:

---

imprimeix *"hola!!"*

---

cerca *en aquest text tots els mots que comencin per* **t** *i acabin per* **ava** *i tinguin menys d'11 lletres* **;**
compta*'ls* **;**
desa*'ls en un fitxer*

Com a conseqüència de la immediatesa amb què es realitzen les accions, els llenguatges imperatius són especialment indicats per manipular dades, com per exemple gestionar fitxers, modificar textos, etc.

Com ja hem dit, els llenguatges imperatius es caracteritzen per una aproximació algorísmica a la resolució de problemes, mitjançant l'ús de **variables**, **estructures de dades** simples (unitàries) i complexes (de conjunt) i **estructures de control** (úniques o iteratives).

Distingirem dos passos en la resolució d'un problema:

1. **especificació** del problema en forma d'algorisme (preferiblement mitjançant un *pseudocodi*), amb els següents requisits:
   - definir les dades d'entrada, dades de sortida,
   - definir els passos de l'algorisme i les estructures de dades necessàries per dur a terme les accions especificades *sense ambigüitat*,
   - ha de ser *finit*, és a dir, s'ha d'acabar un cop s'han executat un nombre finit de passos, cadascun dels quals ha de ser executable en un temps finit.
2. **implementació** de l'algorisme en un programa informàtic que pot ser executat per un ordinador, mitjançant un *llenguatge de programació*.

1.1 Estructures de control bàsiques: `si, sino, sinosi`

Les estructures de control estableixen un conjunt de condicions que poden controlar de manera diversa les accions que l'algorisme realitza. S'organitzen en condició i acció, de manera que l'acció especificada tan sols es realitza si s'acompleix la condició estipulada.

if (A) { B } si s'acompleix la condició A es realitza l'acció B.

---

if `( trobes una paraula que comença amb la lletra` *a* `)`
`{ imprimeix ‘‘A’’ i la paraula }`

---

if (A) { B } elsif (C) { D } else { E } si s'acompleix la condició A es realitza l'acció B i se salta al final de l'estructura de control. Si no s'acompleix A, s'avalua la condició C, si s'acompleix la condició C, es realitza D i se salta al final de l'estructura de control. Si no s'acompleix C, es realitza E.

---

if `( trobes una paraula que comença amb la lletra` *a* `)`
`{ imprimeix-la }`
elsif `( trobes una paraula que acaba amb la lletra` *t* `)`
`{ imprimeix ‘‘T’’ i la paraula}`
else `{ imprimeix la paraula}`

1.2 Estructures de dades bàsiques: unitats i llistes

Les variables són estructures de dades que serveixen per emmagatzemar temporalment informació que volem manipular en el programa, i que no és prèviament coneguda sinó que és, precisament, variable. En l'exemple anterior, si apliquem el programa a un text, "la paraula" seria una variable el valor de la qual aniria canviant a mesura que anéssim recorrent un text i anéssim avaluant les diferents paraules que conté.

En Perl, les variables van immediatament precedides d'un símbol que indica el tipus de variable de què es tracta: $ per a escalars, @ per a llistes ordenades, % per a llistes associatives. Els noms de les variables no poden començar per un nombre, ni poden contenir espais en blanc o signes de puntuació, excepte el guió baix "_". Hi ha diferència entre majúscules i minúscules: $pou i $pOU no són la mateixa variable.

A diferència de llenguatges com C, no cal declarar les variables. No obstant, és recomanable fer-ho, per garantir la coherència del programa. La instrucció "`my`" serveix per declarar-les. Per forçar la declaració de variables, es pot usar el mode estricte, escrivint al principi del programa `use strict ;` .

variables unitàries (escalars) les variables simples, altrament dites escalars, tenen un valor atòmic. Aquests valors poden ser molt variables, l'única cosa en comú que tenen és que es tracten com a una unitat. Una variable escalar pot representar un nombre, un caràcter o una seqüència de caràcters de qualsevol longitud, i també la referència a una estructura de dades més complexa. En Perl, aquestes variables van precedides del símbol $, com per exemple "`$nom`", "`$cognom`".

variables de conjunt (llistes) les llistes són una agrupació d'un conjunt de variables escalars, de manera que quan actuem sobre un membre concret de la llista el tractem com un escalar, anomenant-lo pel seu número d'ordre en la llista (el seu *índex*). Les llistes ordenades van precedides del símbol @, i els elements unitaris s'anomenen `$nom_llista[index]`, on `index` és el nombre d'ordre de l'element en la llista. Si no es força el contrari, el primer element d'una llista té índex 0, el segon té índex 1, i així successivament.

1.3 Accions bàsiques: assigna valor, avalua, imprimeix, gestió d'entrada i sortida

Les diverses accions d'un algorisme se separen mitjançant un punt-i-coma. Darrera del símbol } que tanca una estructura de control no cal posar un punt-i-coma, però si ho fem no trobem error.

1.3.1 Assignació

Les variables prenen el seu valor per assignació explícita en l'algorisme, a través de l'operador =. Per exemple:

```
$nom = Pepa ; $cognom = Lopez ;
```

En el cas de variables de llista, l'assignació de valor pot ser conjunta o particular per cada element:

- *assignació conjunta:* `@llista = (‘‘pomes’’, ‘‘peres’’, ‘‘plàtans’’, ‘‘ous’’)`
- *assignació particular:* `$llista[0] = ‘‘pomes’’; $llista[1] = ‘‘peres’’;`
  (fixeu-vos: és idèntica a l'assignació escalar)

Encara que una variable tingui un valor, podem assignar-li'n un altre, fins i tot utilitzant la mateixa variable en l'assignació. Per exemple:

operador	efecte	sintaxi
=	assignació de valor (numèric, textual,...)	$mot = “hola”
==	avaluació de valor numèric	**if** ($nombre == “5”) { ... }
`eq`	avaluació de valor textual	**if** ($mot eq “hola”) { ... }
+, −, /, ∗	suma, resta, divisió, multiplicació	$total = $subtotal1 + $subtotal2
>, <, >=, <=	més gran/petit, més gran/petit o igual	**if** ($nombre1 <= $nombre2) { ... }
.	concatenació (variables textuals)	$mot = $arrel.“ava”

Taula 1: Operadors per a variables unitàries o escalars.

```
$resultat = 2 + 3;
```
→ *la variable* `$resultat` *té el valor* “`5`”
```
$resultat = 2 + 3; $resultat = 4;
```
→ *la variable* `$resultat` *té el valor* “`4`”
```
$resultat = 2 + 3; $resultat = $resultat * 2;
```
→ *la variable* `$resultat` *té el valor* “`10`”

1.3.2 Avaluació

Sovint, en les condicions (**if**) que controlen una acció, voldrem avaluar el valor d'una variable. Això es fa amb l'operador ==. Per exemple:

```
if ( $resultat == 5 )  imprimeix $resultat
```

A la taula 1.3.2 resumim alguns operadors que poden actuar sobre variables escalars.

1.3.3 Impressió

Aquesta és l'acció més fàcil, s'executa mitjançant la comanda `print`:

```
if ( $resultat == 5 ) { print $resultat }
```

Ara bé, si voleu imprimir les coses amb format, haureu de fer servir cometes dobles i símbols especials per a salts de línia (‘‘`\n`’’) o tabulacions (‘‘`\t`’’). Per exemple:

```
print ‘‘un\tdos\ntres\tquatre\n\tcinc\n’’ ;
```

→ *imprimeix això:*

```
un      dos
tres    quatre
     cinc
```

1.3.4 Gestió d'entrada i sortida (estàndard, fitxers, directoris)

Una manera molt fàcil d'introduir dades en un programa i assignar-les a les variables és a través de l'entrada estàndard. Hi ha dos tipus d'entrada estàndard: interactiva o per defecte. La interactiva és la que demana explícitament a l'usuari que introdueixi les dades en un moment determinat, i s'invoca amb la comanda <STDIN>. Per exemple:

```
print ‘‘escriu el que vols imprimir:  ’’  ;
$inputline =<STDIN> ;
print ‘‘$inputline\n’’ ;
```

Ara bé, si el que voleu és que es tracti un fitxer complet, haureu de fer servir les expressions <> i $_, que contenen, respectivament, el fitxer per defecte i la línia per defecte. Per exemple, quan invoquem el següent programa, que anomenarem `imprimeix.pm`[1] s'imprimeix cada línia del fitxer en la sortida estàndard (la pantalla):

```
while ( <> ) {
     print ‘‘$_’’ ;
}
```

Així, si tenim el següent fitxer, anomenat `llista.txt`:

```
pomes
peres
ous
plàtans
```

I escrivim:

```
perl imprimeix.pm < llista.txt
```

[1] L'extensió típica dels fitxers d'emacs és ".pm".

Obtindrem el següent resultat:

```
pomes
peres
ous
plàtans
```

Si volem desar els resultats en un fitxer, en lloc de la sortida estàndard, podem invocar el programa de la següent manera:

```
perl imprimeix.pm < llista.txt > llista_impresa.txt
```

Si no voleu escriure els noms dels fitxers d'entrada i sortida en invocar el programa, haureu de treballar amb els noms dels fitxers dins del programa, utilitzant el que s'anomena *filehandles*. Per obtenir el mateix resultat que en l'exemple immediatament anterior amb *filehandles* farem servir el següent programa `imprimeix_filehandles.pm`, que s'invoca sense cap nom de fitxer:

```
open E, ‘‘llista.txt’’ ;
open S, ‘‘>llista_impresa.txt’’ ;
while ( <E> ) {
    print S ‘‘$_’’ ;
}
close E;
close S;
```

Els *filehandles* són qualsevol seqüència de lletres majúscules que associem a un fitxer en el moment d'obrir-lo. Fixeu-vos que quan obrim un fitxer per llegirlo diem `open F, ‘‘nom_fitxer’’`, en canvi, si hi volem escriure diem `open F, ‘‘>nom_fitxer’’`. I recordeu que tot el que s'obre s'ha de tancar!

2 Implementació de problemes bàsics en Perl

2.1 Instal·lació de Perl en Microsoft windows

Totes les distribucions de Linux i les de MacOS a partir de la versió X porten Perl instal·lat per defecte. En Windows, Perl no està instal·lat i per tant cal instal·lar-lo.

Us podeu baixar una distribució de Perl per windows a ActivePerl. És gratuïta i molt fàcil d'instal·lar.

Un cop instal·lat, la manera clàssica d'invocar Perl és a través de la terminal, de la següent manera:

```
terminal$ perl programa.pm < fitxer_entrada > fitxer_sortida
```

Es pot evitar d'escriure la paraula "perl" si en la primera línia del programa s'escriu el camí on està situat el perl en l'ordinador que esteu utilitzant, per exemple:

```
#!/usr/local/bin/perl
```

No obstant, si utilitzeu la paraula perl en invocar el programa, el sistema mateix buscarà el perl en l'ordinador, independentment d'on sigui.

2.2 Exemples d'implementació d'accions, estructures de control i estructures de dades bàsiques

```
print ("Enter a number:\n");
$number1 = <STDIN>;
chop ($number1);
print ("Enter another number:\n");
$number2 = <STDIN>;
chop ($number2);
if ($number1 == $number2) {
       print ("The two numbers are equal.\n");
}
else {
       print ("The two numbers are not equal.\n");
}
print ("This is the last line of the program.\n");
```

```
print ("Enter a number:\n");
$number1 = <STDIN>;
chop ($number1);
print ("Enter another number:\n");
$number2 = <STDIN>;
chop ($number2);
if ($number1 == $number2) {
       print ("The two numbers are equal.\n");
}
elsif ($number1 == $number2 + 1) {
       print ("The first number is greater by one.\n");
```

```
}
elsif ($number1 + 1 == $number2) {
       print ("The second number is greater by one.\n");
}
else {
       print ("The two numbers are not equal.\n");
}
print ("This is the last line of the program.\n");
```

-- Intenteu resoldre els següents problemes:

1. substituïu el nom de les variables en els programes anteriors, sense canviar el resultat
2. entreu mots en lloc de nombres, i compareu-ne la igualtat
3. utilitzeu *filehandles*

3 Estructures de dades i control complexes

3.1 Llistes associatives

Les llistes associatives (o `hash`) són semblants a les llistes ordenades (o `array`), amb la diferència que els elements no estan ordenats mitjançant un número, sinó que el seu *índex* és una paraula. Les llistes associatives van precedides del símbol %, i els seus elements unitaris s'anomenen `$nom_llista{index}`, on `index` és un índex estipulat en el moment en què s'assignen els valors:

L'assignació de valor a llistes associatives es fa de la següent manera:

- *assignació conjunta:*
  `%llista = { pomes => ‘‘golden’’, peres => ‘‘blanquilla’’ }`
- *assignació particular:*
  `$llista{pomes} = ‘‘golden’’; $llista{peres} = ‘‘blanquilla’’`
  (fixeu-vos: és idèntica a l'assignació escalar)

3.2 Estructures de control iteratives: `mentre`, `fins que`, `per`, `per cada`

Les estructures de control iteratives també s'organitzen en condició i acció, però, a diferència de les estructures de control simples, el procés de comprovació de la condició i realització de l'acció es repeteix mentre la condició no sigui falsa.

while (A) { B } si s'acompleix A, es realitza B, i es torna a avaluar A. Aquest procés es repeteix mentre A s'acompleixi, és a dir, mentre A sigui cert.

```
$comptador = 10 ;
while  ( $comptador > 0 ) {
     print ‘‘$comptador\n’’;
     $comptador = $comptador - 1 ;
}
```

until (A) { B } si **no** s'acompleix A, es realitza B, i es torna avaluar A. Aquest procés es repeteix mentre A no s'acompleixi, és a dir, mentre A sigui fals.

```
$comptador = 10 ;
until  ( $comptador == 0 ) {
     print ‘‘$comptador\n’’;
     $comptador = $comptador - 1 ;
}
```

foreach $element (@A) { B } per cada element de la llista A, es realitza B, fins que s'hagin exhaurit tots els membres de la llista.

```
@llista = (1,2,3,4,5,6,7,8,9,10);
foreach $elem ( @llista ) {
     $comptador = scalar @llista;
     print ‘‘$comptador\n’’;
}
```

for ($i = INICI ; $i < FINAL ; $i = $i+INCREMENT) { B } en aquesta estructura s'utilitza una variable anomenada *dummy* que s'utilitza per anomenar els

tipus d'estructura de dades	notació de conjunt	notació unitària
variable escalar	–	$escalar
llista ordenada (*array*)	@llista	$llista[$index]
llista associativa (*hash*)	%hash	$hash{$index}

Taula 2: Sintaxi de les estructures de dades bàsiques en Perl.

tipus d'estructura de control	sintaxi de la condició	sintaxi de l'acció
si	if (*condició*)	{ *acció* }
sino	–	else { *acció* }
sinosi	elsif (*condició*)	{ *acció* }
mentre	while (*condició*)	{ *acció* }
fins que	until (*condició*)	{ *acció* }
per	for (*rang*)	{ *acció* }
per cada	foreach *$membre* (@llista)	{ *acció* }

Taula 3: Sintaxi de les estructures de control bàsiques en Perl.

punts d'un rang que va d'INICI fins a FINAL en els intervals determinats per INCREMENT. El següent exemple té el mateix efecte que els exemples anteriors.

```
for ( $i=0 ; $i<10 ; $i = $i+1 ) { print ``$i\n'' }
```

En el següent exemple utilitzem un `for` per recórrer un subsegment d'una llista ordenada (sense oblidar que el primer element d'una llista té índex 0!):

```
@llista = (``peres'', ``pomes'', ``plàtans'', ``ous'', ``nous'');
for ( $i=2 ; $i<5 ; $i = $i+1 ) { print ``$llista[$i]\n'' }
```

A les taules 3.2 i 3.2 trobareu resumides les estructures de dades i de control més comunes en Perl.

3.3 Exemples d'implementació d'accions, estructures de control i estructures de dades complexes

```
foreach $m (@text) {
```

```
    $n = $n + 1;
    print $n.'' ''.$mot.''\n'';
}
```

```
$done = 0;
$count = 1;
print ("This line is printed before the loop starts.\n");
while ($done == 0) {
        print ("The value of count is ", $count, "\n");
        if ($count == 3) {
                $done = 1;
        }
        $count = $count + 1;
}
print ("End of loop.\n");
```

```
print ("What is 17 plus 26?\n");
$correct_answer = 43;     # the correct answer
$input_answer = <STDIN>;
chop ($input_answer);
until ($input_answer == $correct_answer) {
        print ("Wrong! Keep trying!\n");
        $input_answer = <STDIN>;
        chop ($input_answer);
}
print ("You've got it!\n");
```

```
@array1 = (14, "cheeseburger", 1.23, -7, "toad");
@array2 = @array1;
$count = 1;
while ($count <= 5) {
        print("element $count: $array1[$count-1] ");
        print("$array2[$count-1]\n");
        $count++;
}
```

```
@innerlist = " never ";
@outerlist = ("I", @innerlist, "fail!\n");
print @outerlist;
```

```
@input = <STDIN>;
chop (@input);
# first, reverse the order of the words in each line
$currline = 1;
foreach $currline ( @input ) {
       @words = split (/ /, $input[$currline-1]);
       @words = reverse(@words);
       $input[$currline-1] = join(" ", @words, "\n");
       $currline++;
}
# now, reverse the order of the input lines and print them
@input = reverse(@input);
print (@input);
```

-- Intenteu resoldre els següents problemes:

1. llistar tots els nombres parells menors de 100
2. comptar totes les vegades que ocorre la paraula ‘‘de’’ en un text

4 Expressions regulars

Una expressió regular és una abstracció sobre diverses seqüències de caràcters que permet especificar trets que caracteritzen aquest conjunt de seqüències i alhora ignorar les diferències que no són rellevants per caracteritzar aquest conjunt.

Per exemple, per tractar només els mots acabats en `-ava` i `-ia` d'un cert text, caldrà crear una expressió regular que requereixi la presència de qualsevol d'aquestes dues seqüències a final de mot, però que permeti ignorar la resta de caràcters del mot. Les expressions regulars resulten especialment adequades per tractar seqüències de tot tipus, i, en el nostre cas, seqüències lingüístiques.

4.1 Correspondència de patrons

El mecanisme bàsic amb què s'apliquen les expressions regulars és la **correspondència de patrons**: donada una seqüència de caràcters, s'avalua si aquesta seqüència acompleix

les característiques especificades per l'expressió regular, i, si és així, s'estableix la correspondència de patrons.

En Perl, la correspondència de patrons entre una seqüència de caràcters i una expressió regular s'avalua a través de l'operador `=~` [2], que té a un costat la seqüència a avaluar i a l'altre costat l'expressió regular, entre barres (/ `expressió` /).

```
if ( $mot =~ /ava/ )  print ``$mot\n''
```

Perl disposa d'operadors de correspondència de patrons que representen de manera condensada una condició basada en correspondència de patrons i una acció de substitució de la seqüència amb què s'estableix la correspondència. El més utilitzat d'aquests operadors és "s///", que té la següent sintaxi: `s/patro_per_substituir/patro_amb_que_substituir/g`.

```
$mots[0] = ``casa'';
$mots[1] = ``cosa'';
$mots[2] = ``barcassa'';
$mots[3] = ``cas'';
$mots[4] = ``sac'';

foreach $mot ( @mots ) {
    $mot =~ s/cas/cos/;
    print ``$mot\n'';
}
```

RESULTAT:
```
cosa, cosa, barcossa, cos, sac
```

```
Proveu d'expressar la semàntica dels operadors de substitució
mitjançant l'estrucutra clàssica de condició - acció
```

4.2 Símbols per a conjunts de caràcters

Per tal d'expressar abstraccions sobre seqüències, les expressions regulars ofereixen símbols que agrupen conjunts determinats de caràcters. Per exemple, el símbol "." representa qualsevol caràcter, mentre que el símbol "\w" representa qualsevol lletra. En la taula 4.2 trobareu un resum dels principals agrupaments de caràcters.

[2] També es pot recórrer a la negació de l'operador de correspondència de patrons, `!~`

símbol	equivalència
a	a
b	b
`\d`	qualsevol nombre
`\D`	qualsevol caràcter excepte un nombre
`\w`	qualsevol lletra o nombre
`\W`	qualsevol caràcter excepte una lletra o nombre
`\s`	qualsevol espai blanc (espai, tabulador, salt de línia)
.	qualsevol caràcter

Taula 4: Símbols que representen conjunts de caràcters en les expressions regulars de Perl.

També podeu crear els vostres propis agrupaments de caràcters mitjançant llistes expressades entre `[ ]`, o bé `[^ ]` si voleu donar el conjunt complementari del que expresseu entre els claudàtors, tal com es veu en el següent exemple.

```
a,z,O,0,3 = [azO03]
a,b,c,d,e,f,g,h,i,j,k,A,B,C,D,7,8,9 = [a-kA-D7-9]
a,b,c,d,e,f,g,h,i,j,k,A,B,C,D,7,8,9 = [^ l-zE-ZO-6]
```

```
doneu els equivalents en llistes dels símbols propis de Perl que
agrupen conjunts de caràcters.
```

Els anomenats metacaràcters (vegeu taula 4.2) són utilitzats per Perl per especificar la seva sintaxi i per tant no representen el caràcter que veiem, sinó una instrucci'o del llenguatge de programació. Per tal d'establir una correspondència de patrons amb aquests caràcters, cal indicar que no s'han d'interpretar, mitjançant el símbol \ precedint immediatament el caràcter.

4.3 Explotant el context

Per caracteritzar conjunts de seqüències, sovint no és suficient disposar d'agrupacions de caràcters, sinó que també cal especificar la seva posició relativa en una seqüència. Per això disposem de símbols que indiquen inici ^ i final $ de seqüència.

També podem quantificar fragments de les seqüències que treballem, ja siguin símbols unitaris, tant caràcters simples com símbols que representen un conjunt de caràcters, o també subseqüències, prèviament agrupades entre ( ). Trobareu un resum dels quantificadors de Perl a la taula 4.3.

Les subseqüències agrupades entre ( ) poden tractar-se de manera separada, per exemple, per imprimir tan sols aquella part de la subseqüència que trobem entre ( ), o per

símbol	instrucció que expressa
\	evita la interpretació del símbol que el segueix el qual tindrà una interpretació literal
()	agrupen subseqüències d'una seqüència de manera que es pugui tractar com una unitat
[]	estableixen una llista de caràcters equivalents
{}	delimita l'acció controlada per una condició
^	indica l'inici d'una seqüència
?	restringeix l'ocurrència d'una seqüència a cap o una vegada
+	restringeix l'ocurrència d'una seqüència a una o més vegades
*	restringeix l'ocurrència d'una seqüència a cap o més vegades
.	qualsevol caràcter
$	identifica com a variable escalar el mot que el segueix (també indica el final d'una seqüència)
@	identifica com a array el mot que el segueix
%	identifica com a hash el mot que el segueix
&	identifica com a subrutina el mot que el segueix
#	precedeix els comentaris (no s'interpreta res al seu darrera)

Taula 5: Símbols reservats per Perl per a expressar instruccions.

símbol	quantificació
{n,m}	l'expressió pot ocórrer de n a m vegades
*	l'expressió pot ocórrer 0 o més vegades
+	l'expressió pot ocórrer 1 o més vegades
?	l'expressió pot ocórrer 0 o 1 vegades

Taula 6: Expressions de quantificació en Perl.

modificar-la. Per referir-nos a aquestes subseqüències utilitzem les variables pròpies de Perl *$1, $2, $3... $n*, de la següent manera:

a(bc)d(e) $1=bd, $2=e, $3=“”

(ab(cd))(e) $1=abcd , $2=cd , $3=e

(ab)(cd(e)) $1=ab , $2=cde , $3=e

```
Creeu expressions regulars que representin els següents conjunts
de seqüències de la manera més restrictiva possible:

  -- amor, casa, rosa, pota (seqüències de quatre lletres)
  -- pa, casa, rosa, pota, nineta (seqüències formades per un
     nombre indeterminat de parells consonant + vocal)
  -- anna, otto (palíndroms)
```

Mitjançant aquests atributs i les agrupacions de caràcters que ja coneixeu, podreu expressar gairebé qualsevol seqüència lingüística.

4.4 Exemples d'implementació d'expressions regulars

```
print ("Ask me a question politely:\n");
$question = <STDIN>;
if ($question =~ /please/) {
        print ("Thank you for being polite!\n");
}
else {
        print ("That was not very polite!\n");
}
```

```
$wordcount = 0;
$line = <STDIN>;
while ($line ne "") {
        chop ($line);
        @words = split(/[\t ]+/, $line);
        $wordcount += @words;
        $line = <STDIN>;
}
print ("Total number of words: $wordcount\n");
```

```
print ("Enter a variable name:\n");
$varname = <STDIN>;
chop ($varname);
if ($varname =~ /^\$[A-Za-z][_0-9a-zA-Z]*$/) {
        print ("$varname is a legal scalar variable\n");
}
elsif ($varname =~ /^@[A-Za-z][_0-9a-zA-Z]*$/) {
        print ("$varname is a legal array variable\n");
```

```
}
elsif ($varname =~ /^[A-Za-z][_0-9a-zA-Z]*$/) {
       print ("$varname is a legal file variable\n");
}
else {
       print ("I don't understand what $varname is.\n");
}
```

```
$thecount = 0;
print ("Enter the input here:\n");
$line = <STDIN>;
while ($line ne "") {
        if ($line =~ /\bthe\b/) {
                $thecount += 1;
        }
       $line = <STDIN>;
}
print ("Number of lines containing 'the': $thecount\n");
```

```
print ("Enter the search pattern:\n");
$pattern = <STDIN>;
chop ($pattern);
$filename = $ARGV[0];
$linenum = $matchcount = 0;
print ("Matches found:\n");
while ($line = <>) {
       $linenum += 1;
       if ($line =~ /$pattern/) {
               print ("$filename, line $linenum\n");
               @words = split(/$pattern/, $line);
               $matchcount += @words - 1;
       }
       if (eof) {
               $linenum = 0;
               $filename = $ARGV[0];
       }
}

if ($matchcount == 0) {
        print ("No matches found.\n");
}
```

```
else {
        print ("Total number of matches: $matchcount\n");
}
```

```
@input = <STDIN>;
$count = 0;
while ($input[$count] ne "") {
        $input[$count] =~ s/^[ \t]+//;
        $input[$count] =~ s/[ \t]+\n$/\n/;
        $input[$count] =~ s/[ \t]+/ /g;
        $count++;
}
print ("Formatted text:\n");
print (@input);
```

```
while ($line = <STDIN>) {
         $line =~ tr/A-Z/a-z/;
         print ($line);
}
```

---

5 Problemes de Processament del Llenguatge Natural

Tot seguit plantejarem una sèrie de problemes de processament del llenguatge natural. Per resoldre'ls, haureu d'especificar l'algorisme i implementar-lo en Perl.

- Calcular la freqüència dels mots d'un text, i de patrons de dos o tres mots
- Comparació de les freqüències de mots en dos textos diferents
- Extracció dels contextos d'ocurrència d'un mot
- Reconeixement de l'idioma en què està escrit un text
- Formatar un text perquè ressaltin algunes característiques en `html`

```
Se us acuden altres problemes que podrien ser resolts de manera
eficient mitjançant un llenguatge imperatiu?
```

6 Per saber-ne més

6.1 D'algorísmica

– Joan Vancells i Enric Lòpez. *Programació: introducció a l'algorísmica*, Barcelona, Eumo, 1992.

– Joëlle Biondi i Gilles Clavel. *Introducción a la programación*, Barcelona, Masson, 1988.

– Andrés Marzal i Isabel Gracia. Apunts de *Metodología y Tecnología de la Programción* `http://marmota.act.uji.es/MTP/teoria.shtml`.

6.2 De Perl

– Teach Yourself Perl 5 in 21 days és un llibre de gran èxit que ha estat piratejat en aquesta web russa...

– Larry Wall, Tom Christiansen i Jon Orwant. Programming Perl, O'Reilly, (3a Edició) 2000.

– Randal Schwartz i Tom Christiansen. *Learning Perl*, O'Reilly, 1997.

– Jon Orwant, Jarkko Hietaniemi i John Macdonald. *Mastering Algorithms with Perl*, O'Reilly, 1999.

– `http://www.Perl.com`