Serge Fleury Université Paris 3 - ILPGA

PERL

Initiation à la programmation pour le TAL avec Perl

Post-préambule

Contact et hypertoile

- **Courrier électronique**
  - serge.fleury@univ-paris3.fr

  ou
  - fleurys@noos.fr

Retour sommaire

- Site TAL
  - http://tal-p3.no-ip.org
- Portail TAL
  - http://tal.univ-paris3.fr/
- Groupe TAL
  - http://fr.groups.yahoo.com/group/tal-ilpga/
  - Inscription obligatoire : sur le site TAL, inscription en inscrivant son mail dans la zone adéquate
- Site ILPGA-ELEARNING
  - http://e-ilpga.univ-paris3.fr/
  - http://e-ilpga.univ-paris3.fr/campus
    - Un identifient etudiant « user » « password » (me demander)
      - Polys disponibles pour chaque EC
- Intranet ILPGA
  - Sur le site local http://195.221.76.135/~sfleury/
    - Contenus du DVD TAL en ligne

Retour sommaire

Contrôle des connaissances

- **Pour ce module, en contrôle continu, il faudra traiter et rédiger le travail réalisé en TD**
  - **Ce TD doit conduire à la réalisation d'un site WEB regroupant les manipulations et résultats produits en réponse aux questions posées dans le cadre de ce TD**
- **Un partiel en fin de semestre**

Retour sommaire

Le site TAL

Retour sommaire

Page des Enseignements TAL/Info

Accueil
Nouveautés
Sommaire
Cours en ligne
Enseignements
Parcours TAL
Lectures
Liens
PluriTAL
TALeTOILE
Corpus
Logiciels
Etudiants TAL/ILPGA
Travaux Etudiants
Mémoires DL
Examen / Partiel

Secteur TAL Informatique

ILPGA Université Paris 3

Cours TAL/Info

DEUG1 (LMD->Licence niveau 1)

Module SLFD1
Responsable : A. Salem

Module SLMD2
Responsables : A. Salem, S. Fleury

DEUG2 (LMD->Licence niveau 2)

Module SLOD3
(A.Salem)

Module SLOD4
(S. Fleury)

Licence (LMD->Licence niveau 3)

Module SLFE6
Licence Tronc Commun
(A. Salem, P. Samvelian, S.Fleury)

Module SLOU5-6
Licence TAL
(A. Salem)

Module SLOV5-6
Licence TAL
(S. Fleury)

Module SLOW5-6
Licence TAL
(J. Vaissière, A. Salem, P. Renaud)

Retour sommaire

Page des cours en ligne

Accueil
Nouveautés
Sommaire
Cours en ligne
Enseignements
Parcours TAL
Lectures
Liens
PluriTAL
TALetTOILE
Corpus
Logiciels
Etudiants TAL/ILPGA
Travaux Etudiants
Mémoires IDL
Examen/Partiel

Secteur TAL Informatique

ILPGA Université Paris 3

Cours en ligne

Ressources locales

- **Transparents de Cours "Secteur TAL/Informatique"**
  - Cours DEUG1
    - TD Natacha Espinosa SLFD1 : **TD 1 - WORD**.
    - TD Maria Zimina SLFD1 : (1) **(document PDF)** ; 2) VISUAL BASIC **(document PDF)** .
    - **Présentation de PowerPoint** (document Word). **Exercices PowerPoint** (document PDF).
    - **Présentation de CORDIAL : correcteur global et analyseur de la langue française** (document PDF).
    - **Cours Microsoft Excel** : Un aperçu des outils permettant de créer et de gérer des chiffres sous forme de tableaux et de graphiques ...(document PowerPoint)
    - **TD Excel/Cordial** : L'objectif de ces séances est d'utiliser Microsoft Excel pour l'analyse des résultats d'étiquetage morphosyntaxique de textes obtenus

Retour sommaire

Portail e-ILPGA http://e-ilpga.univ-paris3.fr

E-ILPGA

ILPGA Université Paris 3 Sorbonne nouvelle

Université de la Sorbonne Nouvelle - PARIS III

INSTITUT DE LINGUISTIQUE ET PHONETIQUE GENERALES ET APPLIQUEES

19, rue des Bernardins - 75005 PARIS

ILPGA

Directeur : M. Patrick Renaud

Sous-Directeur : M. Serge Fleury

Responsable administrative : Mme Sandrine Kerespars

Tél. : 01 44 32 05 70 **Fax** : 01 44 32 05 73

Nom d'utilisateur
sfleury
Mot de passe
Entrer

Perdu mot de passe

Comment démarrer

Forum de support

Site Paris 3

Site ILPGA

Site TAL P3

Site ED268

L'Institut de Linguistique et Phonétique Générales et Appliquées (ILPGA) se consacre à l'enseignement et à la recherche dans le domaine général du langage (traité comme objet d'études pluridisciplinaires) et des langues (étude synchronique et étude diachronique) et dans certains domaines particuliers (phonétique, phonologie, linguistique africaine, linguistique finno-ougrienne). Il est organisé en sections correspondant aux principaux domaines d'activité et propose un ensemble de cursus conduisant à la fois à des diplômes nationaux, y compris D.E.A. et doctorats, et à des diplômes d'Université.

VOTRE COMPTE : L6T51

MOT de PASSE : xxxxxx

MESSAGE pour les utilisateurs ETUDIANT :
Il est impératif de ne pas modifier le paramétrage (le profil) des comptes ETUDIANT dont les identifiants vous ont été transmis.

En cas de problème de connexion, ne pas hésiter à me contacter par mail (*cf infra*).
S. Fleury.

Description du cours
Agenda
Annonces
Documents et liens
Exercices
Parcours pédagogique
Travaux
Forums
Groupes
Utilisateurs
Discussion
Wiki
Nouveautés

L6T51 - Programmation pour le TAL avec Perl
Coefficient : 1 Volume horaire hebdomadaire : CM/TD 1h30
Responsable : Serge Fleury
Contenu :
"Pourquoi programmer (1) et pourquoi Perl (2)" :

- (1) "Working with language data is nearly impossible these days without a computer. Data are massaged, analyzed, sorted, and distributed on computers. Various software packages are available for language researchers, but to truly take control of this domain, some amount of programming expertise is essential"
- (2) "The Perl programming language may provide an answer. There are a number of reasons why Perl may be an excellent choice. First, Perl was designed for extracting information from text files. This makes it ideal for many of the kinds of tasks language researchers need. Second, there are free Perl implementations for every type of computer. It doesn't matter what kind of operating system you use or computer architecture it's running on. There is a free Perl implementation available. Third, it's free. Again, for any imaginable computer configuration, there is a free Perl implementation. Fourth, it's extremely easy. In fact, it might not be an exaggeration to claim that of the languages that can do the kinds of things language researchers need, Perl may be the easiest to learn. Fifth, Perl is an interpreted language. This means that you can write and run your programs immediately without going through an explicit intermediate stage to convert your program into something that the computer will understand. Sixth, Perl is a natural choice for programming for the web. Finally, Perl is a powerful programming language."

(source : [Hammond 2003])

Le détail des cours est présenté sur le site TAL-ILPGA à l'adresse suivante :
http://www.cavi.univ-paris3.fr/ilpga/ilpga/tal/cours/L6T51.htm

Objectifs pédagogiques : Initiation à la programmation pour le TAL avec PERL

Ressources pour ce cours :

Elles sont principalement disponibles dans la rubrique "*Documents et Liens*"

Bibliographie :

Retour sommaire


Programmation pour le TAL avec Perl

L6T51 - Serge Fleury

Documents et liens

Documents et liens

Remonter | **Rechercher**

Liste des fichiers | **Vignettes**

Nom	Taille	Date
LECTURES		
Des nombreuses lectures, complémentaires au cours		
OUTILS-PROGRAMME		
Des outils et des applications écrits en Perl		
POLYCOPIE		
TP		
TRANSPARENT		

Plan

- Perl pour les linguistes
- Préambule
- Introduction
- Variables Perl : scalaires, tableaux, tableaux associatifs
- Entrées/sorties
- Structures de contrôle
- Expressions régulières et Perl
- Passage de paramètres à un programme
- Gestion de fichiers
- Fonctions et variables prédéfinies
- Exemples de programmes
- Perl en ligne de commandes
- Perl et les objets
- Perl/Tk

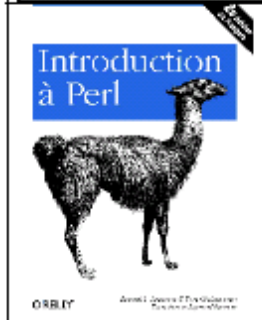
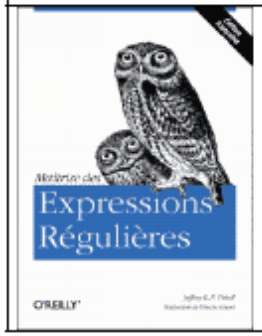
Retour sommaire

Bibliographie (1)

- Brown Martin, *Perl Annotated Archives*, Ed. Osbourne/Mc Grawhill
- Christiansen & Torkington, *Perl en action*, , O'Reilly
- Glover Mike, Humphreys, Ed Weiss, *Perl 5 how-to*, The Wait Group, Inc. 1996
- Habert Benoît, Cécile Fabre, Fabrice Issac, *De l'écrit au numérique (constituer, normaliser et exploiter les corpus électroniques*, InterEditions, 1998.
- Hammond Michael, *Programming for linguists - Perl for Language Researchers*, Blackwell Publishinq, 2003
- Orwant John, (edité par), *Web, Graphics & Perl/Tk : Best of The Perl Journal*, 504 pages, édition O'Reilly, 2002
- Wall L. & al., *Programmation en Perl*, Traduction française, 3ème édition, O'Reilly
- Walsh Nancy, *Introduction à Perl/TK*, O'Reilly, 2000

Retour sommaire

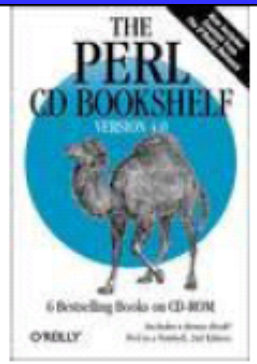
Bibliographie (2)

LIVRE	Titre, auteurs, prix, éditeur (éventuellement URL)	Commentaires
	Programmation Perl (3éme édition) Larry Wall, Tom Christiansen & Jon Orwant Décembre 2001, 1074 pages, 54€ ed. O'Reilly & Associates ISBN: 2-84177-140-7 http://www.oreilly.fr/catalogue/pperl3.html	Le livre de référence sur Perl. Larry Wall (créateur du langage Perl) en est le co-auteur. Parfois trop technique et pas assez pédagogique
	Introduction à Perl (3ème édition) Randal L. Schwartz & Tom Phoenix Janvier 2002, 308 pages, 34€ ed. O'Reilly & Associates ISBN: 2-84177-201-2 http://www.oreilly.fr/catalogue/intro-perl-3ed.html	Un livre plus pédagogique que « programmation Perl », mais moins fournit.
	Maîtrise des expressions régulières Jeffrey E. F. Friedl 2e édition, juin 2003, 486 pages, 40€ ed. O'Reilly & Associates ISBN : 2-84177-236-5 http://www.oreilly.fr/catalogue/2841772365.html	Tout ce que vous avez toujours voulu savoir sur les expressions régulières.

Source : http://www.med.univ-rennes1.fr/~poulique/cours/perl/

Retour sommaire

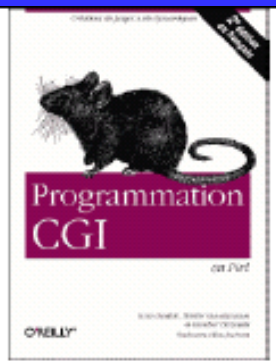
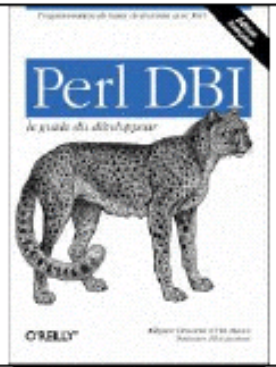
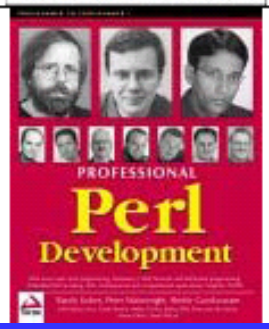
Bibliographie (3)

	**Perl resource kit** (V4.0 prévue pour Janvier 2004) **6 livres sur un CD-ROM**: *"Perl in a Nutshell"*, *"Mastering Regular Expressions"*; *"Learning Perl"*, *"Programming Perl"*, *"Learning Perl Objects, References, and Modules"*, et *"Perl Cookbook"* Janvier 2004 (actuelle version: 3.0), 100$ ISBN 0-596-00622-5 http://www.oreilly.com/catalog/perlcdbs4/	Un CDROM contenant 6 livres (et une version papier de "*Perl in a nutshell*").Certaine ment le meilleur achat (si vous lisez l'anglais)
	**Programmation avancée en Perl** Sriram Srinivasan Juin 1998, 448 pages, 43€ ed. O'Reilly & Associates ISBN : 2-84177-039-7	Pour ceux qui veulent mieux comprendre le fonctionnement de Perl lui-même. Comprend aussi une initiation à PerlTk
	**Perl en action** Tom Christiansen & Nathan Torkington Septembre 1999, 972 pages, 54€ ed. O'Reilly & Associates ISBN : 2-84177-077-X	Recettes et solutions. Une vraie mine d'or.

Source : http://www.med.univ-rennes1.fr/~poulique/cours/perl/

Retour sommaire

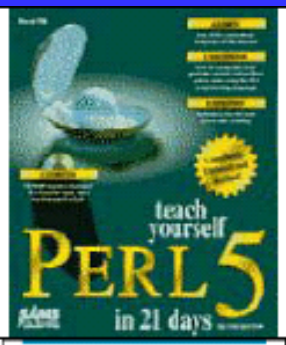
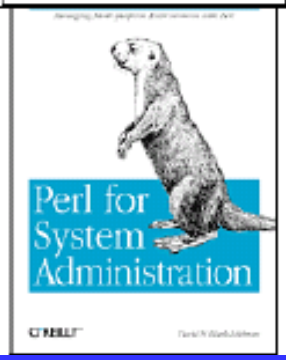
Bibliographie (4)

	**Programmer des CGI en Perl** (2ème édition) Scott Guelich, Shishir Gundavaram et G. Birznieks Juin 2001, 477 pages, 38€ ed. O'Reilly & Associates ISBN: 2-84177-098-2 http://www.oreilly.fr/catalogue/cgi_perl2.html	Livre sur les techniques CGI avec les exemples en Perl. Les différents types de CGI sont abordés, mais pas forcément en profondeur
	**Perl DBI, le guide du développeur** Alligator Descartes & Tim Bunce Décembre 2000, 390 pages, 34€ ed. O'Reilly & Associates ISBN : 2-84177-131-8	Notamment dans le domaine médical, on utilise très souvent Perl avec les bases de données.
	**Professional Perl Development** Randy Kobes, Peter Wainwright, S. Gundavaram Avril 2001, 725 pages, 64€ ed. Wrox Press ISBN : 1-86100-438-9	Non lu, mais l'éditeur a bonne réputation

Source : http://www.med.univ-rennes1.fr/~poulique/cours/perl/

Retour sommaire

Bibliographie (5)

	**Perl 5 en 21 jours** David Till Mai-96, 840 pages,30$ ed. SAMS ISBN: 2-7440-1202-5	Un gros livre qui permet d'apprendre pas-à-pas toutes les fonctionnalités de Perl. **Mais il faut avoir 21 jours devant vous !**
	**Perl & LWP** Sean M. Burke Juin 2002, 264 pages, 35€ ed. O'Reilly & Associates ISBN: 0-596-00178-9	Si vous avez l'intention d'utiliser Perl pour la consultation automatique de sites web, voilà la bible.
	**Perl for System Administration** David N. Blank-Edelman Juillet 2000, 35$ ed. O'Reilly & Associates ISBN: 1-56592-609-9 http://www.oreilly.com/catalog/perlsysadm/	Utile pour l'administration sur Windows / NT (livre décevant pour Unix, encore plus pour Mac OS…)

Source : http://www.med.univ-rennes1.fr/~poulique/cours/perl/

Retour sommaire

Bibliographie (6)

- Nombreuses ressources sur le Web
  - *Cf* voir références sur le site TAL
- En particulier “Perl in a Nutshell”
  - http://www.unix.org.ua/orelly/perl/perlnut/index.htm
- On lira *aussi* et **surtout** le polycopié du cours…

Retour sommaire

Perl pour les linguistes…

Pourquoi programmer et pourquoi Perl ?

- (1) "*Working with language data is nearly impossible these days without a computer. Data are massaged, analyzed, sorted, and distributed on computers. Various software packages are available for language researchers, but to truly take control of this domain, some amount of programming expertise is essential*"

- (2) "*The Perl programming language may provide an answer. There are a number of reasons why Perl may be an excellent choice. First, Perl was designed for extracting information from text files. This makes it ideal for many of the kinds of tasks language researchers need. Second, there are free Perl implementations for every type of computer. It doesn't matter what kind of operating system you use or computer architecture it's running on. There is a free Perl implementation available. Third, it's free. Again, for any imaginable computer configuration, there is a free Perl implementation. Fourth, it's extremely easy. In fact, it might not be an exaggeration to claim that of the languages that can do the kinds of things language researchers need, Perl may be the easiest to learn. Fifth, Perl is an interpreted language. This means that you can write and run your programs immediately without going through an explicit intermediate stage to convert your program into something that the computer will understand. Sixth, Perl is a natural choice for programming for the web. Finally, Perl is a powerful programming language.*"

source : [Hammond 2003]

Retour sommaire

Perl et les linguistes…

- Ouvrez une fenêtre de commandes (DOS ou Cygwin sous Windows)
- Editer un nouveau fichier monpremierprogramme.pl avec un éditeur de texte « simple et puissant » comme Emacs…
- Y écrire :

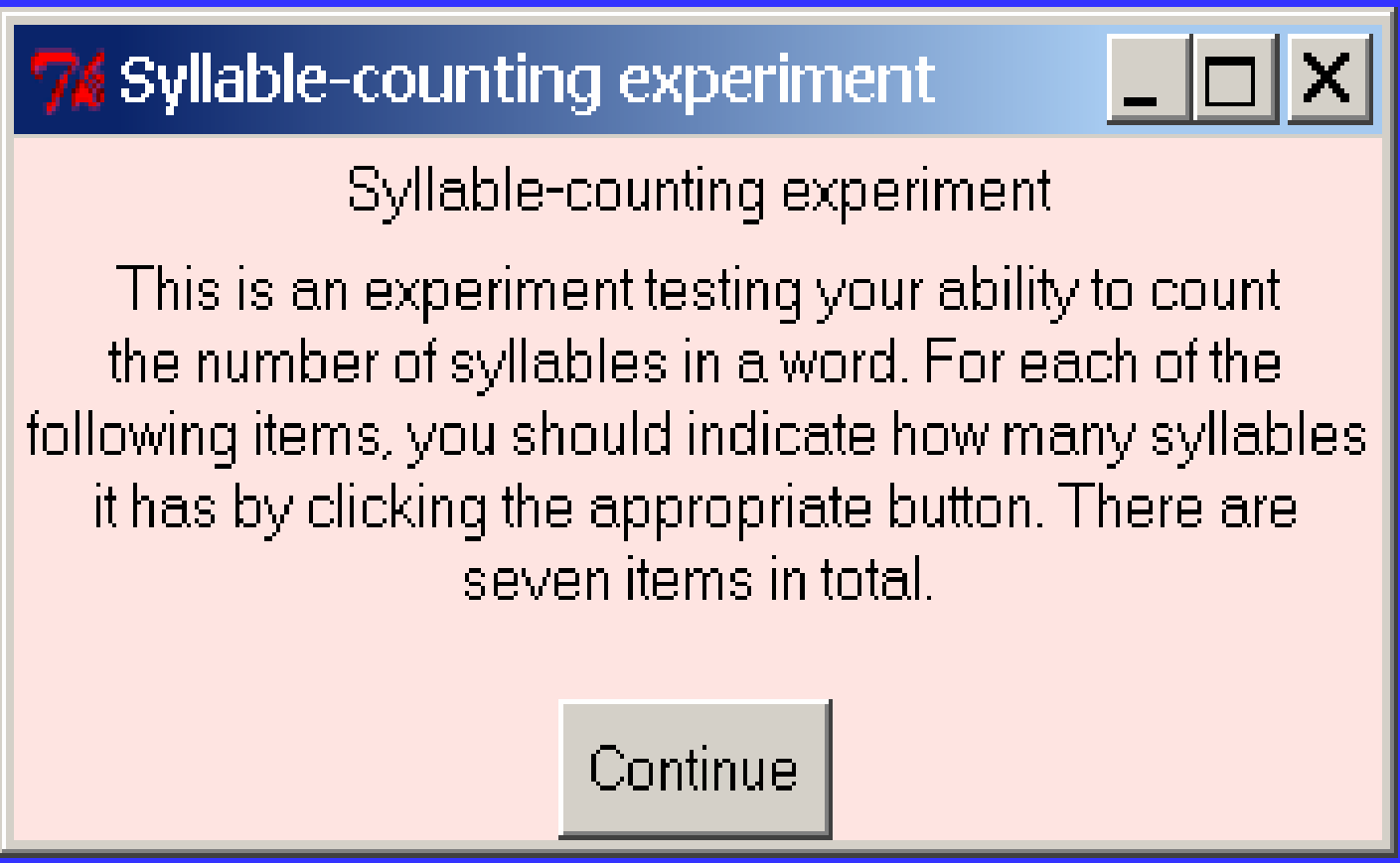
```
print (« Je suis un linguiste »);
```

- Sauvegarder et quitter l'éditeur
- Dans la fenêtre de commande taper :

```
perl monpremierprogramme.pl
```

Retour sommaire

Exemples de programmes pour linguistes (1)



Pour lancer le programme : tkExp.pl

source : [Hammond 2003]

Retour sommaire

Exemples de programmes pour linguistes (2)

- Recherche de verbes dans un texte
  - (*cf* répertoire scripts-hammond)
- Pour lancer le programme, lancer Cygwin ou une fenêtre de commandes, puis taper :

```
perl verbs.pl texte.txt
```

- Le programme `verbs.pl` devrait produire la liste des verbes présents dans le fichier `texte.txt`

source : [Hammond 2003]

Retour sommaire

Exemples de programmes pour linguistes (3)

- **Fréquence des mots d'un texte**
- **Fréquence des bigrammes d'un texte**
- **Fréquence des terminaisons (3 cars)**
- **POS stripping**
  - Etant donné un texte étiquetté avec partie du discours (Parts-Of-Speech) (The/det boy/noun ate/verb . . .) enlever les partie du discours
- **Mots stripping**
  - Etant donné un texte étiquetté avec partie du discours (Parts-Of-Speech) (The/det boy/noun ate/verb . . .) enlever mots
- **Le mot le plus long dans un texte**
- **Trigrammes**
- **4grammes**
- **Etc.**

Retour sommaire

Préambule

Perl

- Perl : *Practical Extraction and Report Language*
- Perl a été conçu par Larry Wall pour développer des programmes dans un environnement UNIX.
- Perl est un langage situé à mi-chemin entre les langages de commande (les shells d'UNIX ou, dans une moindre mesure, MS-DOS) et le langage C

Retour sommaire

EASY Perl

- Perl est disponible sur : UNIX dont il est issu, MACINTOSH, VMS, OS/2, MS-DOS et WINDOWS
- Perl permet d'écrire rapidement de mini-outils de manipulation de données textuelles dont l'implantation à l'aide d'outils classiques, tels sed ou awk, serait malaisée, voir impossible. Il englobe les fonctionnalités de `grep` et `sed`, et va bien au-delà.

Retour sommaire

Un premier exemple (1)

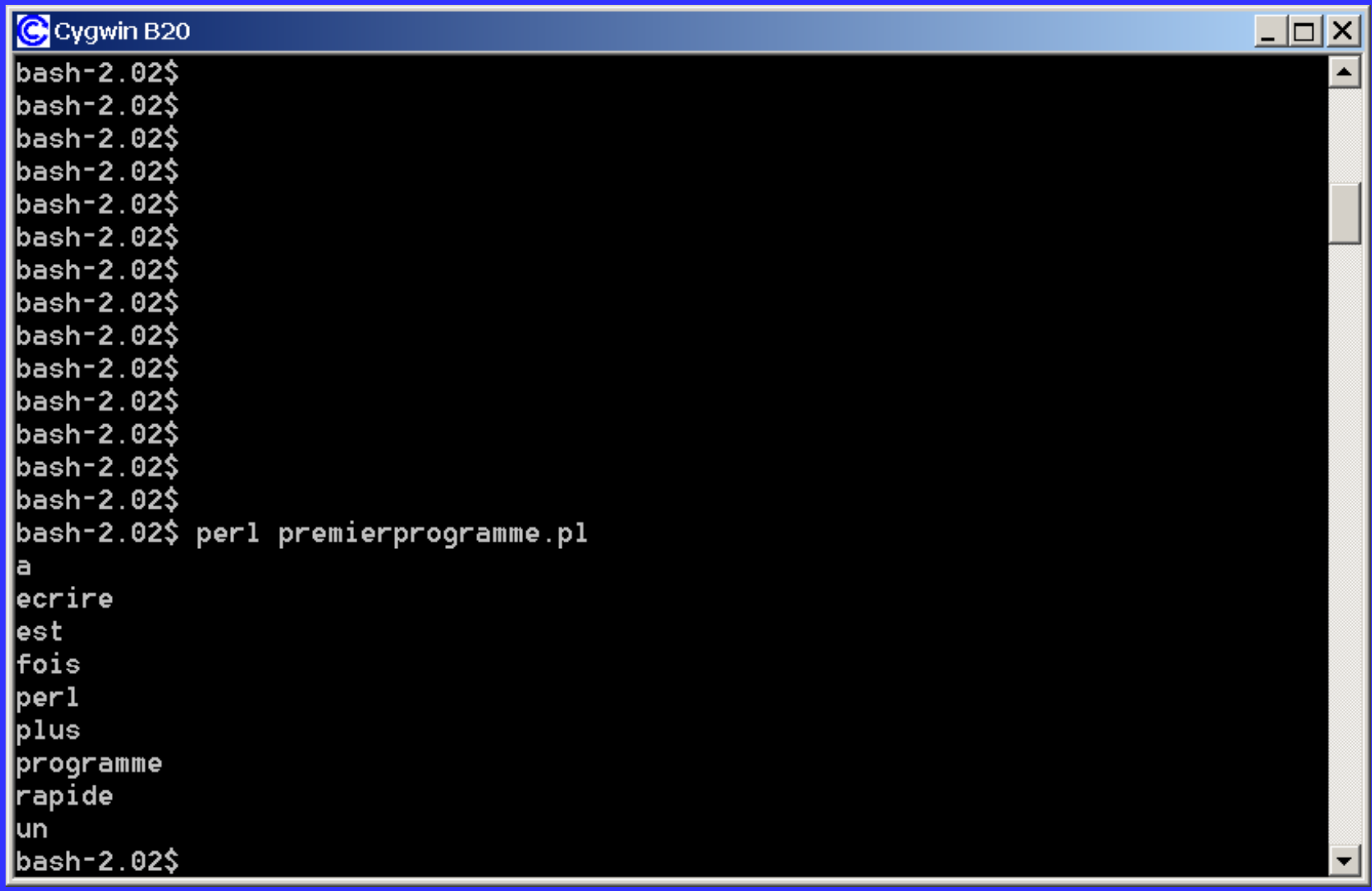
```
my @r = qw(Un programme Perl est 5 fois plus
   rapide a ecrire);
map { tr/A-Z/a-z/; s/\d//g; } @r;
foreach (sort grep !/^$/, @r) { print "$_\n"; }
```

- **Que fait ce programme ?**
  - Ce programme créé une liste de mots (la phrase de la première ligne), transforme les majuscules de ces mots en minuscules, supprime les chiffres appartenant aux mots, supprime les mots vides et affiche la liste des mots ainsi transformés dans l'ordre lexical.
    - Vous aurez en main toutes les notions nécessaires avant la fin de la lecture du document pour comprendre ceci...
  - Le code ci-dessus est contenu dans un fichier nommé ici programme.pl, pour l'exécuter on tape la commande suivante :

```
perl programme.pl
```

Retour sommaire

Un premier exemple (2)



Retour sommaire

Modes d'exécution de Perl

- Première manière : en ligne de commandes (*cf* poly et *infra*)

```
perl -w -e 'print("Salut Larry\n");'
```

- La seconde manière de faire est de créer un fichier `salut.pl` contenant

```
print("Salut Larry\n");
```

  - Puis de lancer la commande suivante depuis un shell :

```
perl -w salut.pl
```

- La troisième façon de faire est de créer un fichier `salut2.pl` contenant :

```
#!/usr/bin/perl –w
print("Salut Larry\n");
```

  - Il faut maintenant rendre ce fichier exécutable (sous Unix/Linux principalement) et le lancer :

```
chmod +x salut2.pl
./salut2.pl
```

Retour sommaire

Programme PERL

- Un programme Perl (que l'on appelle aussi script) consiste en une succession d'instructions dont l'exécution ne nécessite pas d'étape de compilation.
- Sa syntaxe et le nombre important de fonctions dont il dispose en font un langage plus proche de C.

Retour sommaire

Introduction

Ce que Perl est

Perl est :

- un langage de programmation pour l'extraction dans les fichiers textes et l'édition
- un langage interprété
- pas de compilation, donc programme plus simple à executer
- moins rapide qu'un programme compilé
- portable, gratuit, simple, robuste

Retour sommaire

En-tête

- Un programme Perl est un fichier texte, et c'est Perl qui se charge de l'interpréter puis de l'exécuter.
- Sous système UNIX, pour être exécuté un script Perl doit avoir la structure suivante:

```
#!<chemin vers interpréteur>/<nom de l'interpréteur>
<listes de commandes>
```

- Il faut indiquer sur la première ligne l'interpréteur qui va être utilisé pour exécuter la liste de commandes. Pour Perl, on a donc (installation standard) :

```
#!/usr/local/bin/perl
<listes de commandes Perl>
```

Retour sommaire

Perl sous Windows, Mac

- Sur d'autres systèmes, ou l'interaction minimale n'est pas faite via un shell (comme MACINTOSH ou WINDOWS qui sont uniquement graphiques), la première ligne indiquant l'interpréteur n'est pas nécessaire et cette ligne n'est pas prise en compte.

Retour sommaire

Le corps du programme

- Un programme Perl, comme un programme C, est une suite d'instructions terminées par un point-virgule. Pour les structures de contrôle (`while, if, ...`) les blocs de programmes sont obligatoirement délimités par des accolades ouvrantes et fermantes: { et }.

Retour sommaire

Un premier programme Perl


```
#!/usr/local/bin/perl
# programme tout bete qui affiche un message
print " Un premier programme ";
```

Retour sommaire

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

print "Bonjour!"
```

Ceci est un programme Perl

- La première ligne est un commentaire obligatoire qui indique l'endroit où les trouve l'interpréteur Perl
- La deuxième ligne est une commande, qui imprimera Bonjour à l'écran une fois exécuté le programme
- Comment exécuter un programme ?

- Le mettre dans un fichier, par exemple : `premier.pl`
- Sous Linux : rendre le fichier exécutable (`chmod u+x premier.pl`)
- Taper la commande « `premier.pl` » (sous LINUX) ou « `perl premier.pl` » (Windows)

Commentaires et lignes de commandes

- Le caractère # est utilisé pour commenter les programmes. Pour chaque ligne, exceptée la première pour un script écrit sous UNIX, tout ce qui suit ce caractère n'est pas considéré comme faisant partie du langage. La seule manière d'étendre un commentaire sur plusieurs lignes est de commencer chaque ligne par #.
- Les instructions Perl doivent se terminer par un point-virgule, comme la dernière ligne du programme ci-dessus. La fonction print renvoie des informations.

Retour sommaire

Variables Perl

Variables : les scalaires

- Il n'existe qu'un seul type de donnée simple : les scalaires.
- Les variables Perl (les scalaires) permettent d'encoder plusieurs types de données simples. Les identificateurs sont de la forme `$<chaîne>`, où `<chaîne>` est une suite de caractères qui ne doit pas contenir d'opérateur prédéfini de Perl.
- En général, les noms des variables sont composés de chiffres, lettres et souligné (_), mais ils ne doivent pas commencer par un chiffre, et la variable $_ est reservée.
- Perl est sensible à la casse, donc `$a` et `$A` sont deux variables distinctes.

Retour sommaire

Les tableaux

- A partir de ce type simple, les scalaires, on compose deux types complexes:
  - tableau de scalaires;
  - tableau associatif de scalaires.

Retour sommaire

Scalaire

- Un scalaire permet de représenter aussi bien des nombres, entiers ou réels que des chaînes de caractères. Les scalaires commencent par le caractère $. Aux opérateurs classiques de manipulation de nombres, s'ajoute l'opérateur “ . ” (point) qui effectue la concaténation de deux chaînes. Ce sont ces opérateurs qui déterminent le type du contenu de la variable.
- Pour des données numériques, les guillemets ne sont pas nécessaires. On obtient un résultat identique si on écrit `$a="152"` ou `$a=152`. Une variable placée entre deux guillemets (") dans une affectation est remplacée par sa valeur. Si l'on veut éviter cela, on déspécialise le caractère $ à l'aide du caractère “ \ ”.

Retour sommaire

Exemple

```
$numfigure = 12 ;
$p1   =   "...voir   figure   $numfigure
  pour..."
```

$p1 contient :

```
  " . . . voir figure 12 pour . . . "
```

```
$p2   =   "...voir   Figure   \$numfigure
  pour..."
```

$p2 contient :

```
   " . . . voir figure $numfigure pour . . . "
```

Retour sommaire

Perl : constantes et variables

Constantes : 1, -12345, 1.6E16 (signifie 160 000 000 000 000 000), 'cerise', 'a', 'les fruits du palmier sont les dattes'

- **Variables scalaires**

  Les scalaires sont précédés du caractère $

```
$i = 0; $c = 'a';  $mon_fruit_prefere = 'kiwi';
$racine_carree_de_2 = 1.41421;
$chaine = '100 grammes de $mon_fruit_prefere';
```

=> '100 grammes de $mon_fruit_prefere'

```
$chaine = "100 grammes de $mon_fruit_prefere";
```

=> '100 grammes de kiwi'

Attention: Pas d'accents ni d'espaces dans les noms de variables

Par contre un nom peut être aussi long qu'on le veut.

Retour sommaire

```
#!/usr/local/bin/perl
$premier_n   = 10;
$deuxieme_n  = 5;
$troisieme_n = $premier_n + $deuxieme_n;

print "la somme de ", $premier_n, " et ",
$deuxieme_n, " est ",
$troisieme_n,  "\n";
```

En Perl on déclare pas les variables avant de les utiliser.

Les variables Perl ne sont pas typées, on déduit leur type du contexte

```
#!/usr/local/bin/perl
$la_chaine = "Bonjour";
print $la_chaine, "\n";
```

Ou encore, en utilisant le symbole de concaténation

```
#!/usr/local/bin/perl
$bonjour= "Bonjour";
$tout_le_monde = "tout le monde";
$la_chaine = $bonjour . " " . $tout_le_monde .
"\n";
print $la_chaine;
```

Choses importantes à savoir à propos des variables

- Il n'est pas nécessaire d'initialiser les variables, une valeur initiale dépendante du contexte leur est affectée lors de leur création. Mais, cela signifie que toute faute de frappe (par exemple `$ficheir` au lieu de `$fichier`) ne provoquera pas forcément d'erreur mais donnera certainement un mauvais résultat puisqu'une nouvelle variable va être créée et utilisée.
- La gestion de l'allocation de la mémoire est automatique et dynamique. On n'est pas nécessaire de connaître a priori la taille des données que le programme va manipuler.

Retour sommaire

La variable $_

- Variable prédéfinie particulière
- On l'utilise pour stocker des informations fréquemment utilisées (dans la plupart des cas comme variable de boucle) et par défaut quand, dans certaines fonctions ou structures de contrôle, une variable est omise
- *Cf* lecture de fichier dans une boucle `while`

Retour sommaire

Opérations et assignations

- `$a = 1 + 2;` Ajoute 1 à 2, et l'affecte à `$a`
- `$a = 3 - 4;` Soustrait 4 à 3, et l'affecte à `$a`
- `$a = 5 * 6;` Multiplie 5 et 6, et l'affecte à `$a`
- `$a = 7 / 8;` Divise 7 par 8, et affecte 0,875 à `$a`
- `$a = 9 ** 10;` Eleve 9 à la dixième puissance
- `$a = 5 % 2;` Le reste de 5 divisé par deux (division euclidienne)
- `++$a;` Incrémentation de $a, puis on retourne la valeur `$a`
- `$a++;` On retourne la valeur `$a` puis incrémentation de `$a`
- `--$a;` Décrémentation de `$a`, puis on retourne la valeur `$a`
- `$a--;` On retourne la valeur `$a` puis décrémentation de `$a`

Retour sommaire

Chaînes de caractères


```
$a = $b . $c;
```

Concaténation de `$b` et `$c`

```
$a = $b x $c;
```

`$b` répété `$c` fois

Retour sommaire

Assignation d’une valeur à une variable

- `$a = $b;` Assigne `$b` à `$a`
- `$a += $b;` Ajoute `$b` à `$a`
- `$a -= $b;` Soustrait `$b` à `$a`
- `$a .= $b;` Concatène `$b` à `$a`

Retour sommaire

Les opérateurs (1)

Opérateurs arithmétiques

`$a = 1; $b = $a;` les variables a, et b auront pour valeur 1

addition : `$c = 53 + 5 - 2*4;` (=> 50)

Plusieurs notations pour incrémenter une variable

```
$a = $a + 1;
```

ou

```
$a += 1;
```

ou encore

```
$a++;
```

Même chose pour * (multiplication), - (soustraction), / (division), ** (exponentielle)

```
$a *= 3; $a /= 2; $a -= $b;
```

% : modulo

```
17 % 3 =>2
```

Retour sommaire

Les opérateurs (2)

Opérateurs pour chaînes de caractères

concaténation

`$c = 'ce' . 'rise';` (*i.e.* $c devient 'cerise')

`$c .= 's';` (*i.e.* $c devient 'cerises')

réplique

```
$b = 'a' x 5;
```

(*i.e.* 'aaaaa‘)

```
$b = 'jacqu' . 'adi' x 3;
```

(*i.e.* 'jacquadiadiadi‘)

```
$b = 'assez ! '; $b x= 5;
```

(*i.e.* 'assez ! assez ! assez ! assez ! assez ! assez !')

Retour sommaire

Tableau de scalaires (1)

- Un tableau de scalaires est une liste de scalaires indexée par un entier. On fait référence à l'ensemble du tableau en préfixant l’identificateur de la variable par @.
- La référence à un élément particulier du tableau, qui est un scalaire, se fait en préfixant l'identificateur par un $ et en ajoutant entre crochets ([et]) l’indice de l'élément.
- On peut affecter tous les éléments d'un tableau d'un seul coup.

Retour sommaire

Exemple : création d ’un tableau

- Le tableau est une liste d'éléments séparés par des virgules entre parenthèses. Les instructions suivantes :

```
@mots  = ("pommes", "poires", "cerises");
@adjectifs = ("beau", "grand");
```

- affectent une liste de trois éléments à la variable tableau `@mots`, et une liste de deux éléments à la variable tableau `@adjectifs`.
- On accède aux éléments d'un tableau en utilisant des indices entre crochets. Les indices commencent à 0.
- L'expression suivante « `$mots[2]` » renvoie “ cerises ”. On notera que l'@ est devenu un $, étant donné que “ cerises ” est un scalaire.

Retour sommaire

Tableau de scalaires (2)

Etant donnés les deux tableaux

```
@chiffres = (1,2,3,4,5,6,7,8,9,0);
@fruits = ('amande','fraise','cerise');
```

on fait référence à un élément du tableau selon son indice. Les éléments d'un tableaux sont des scalaires, donc ils sont précédés par $

`$chiffres[1]` (=> 2)

`$fruits[0]` (=> 'amande')

REMARQUE: En Perl les tableaux commencent à l'indice 0

Retour sommaire

Tableau de scalaires (3)

En Perl, les tableaux peuvent être utilisés comme des ensembles ou des listes.

On ne doit pas les déclarer au préalable, ni spécifier leur taille.

Toujours précédés du caractère « @ »

```
@chiffres = (1,2,3,4,5,6,7,8,9,0);
@fruits = ('amande','fraise','cerise');
@alphabet = ('a'..'z');
@a = ('a'); @nul = ();
@cartes = ('01'..'10','Valet','Dame','Roi');
```

Les deux points signifient de "tant à tant"

Retour sommaire

Tableau de scalaires (4)

On peut affecter un tableau à un autre tableau

```
@alphanum = (@alphabet, '_', @chiffres);
```

(*i.e.* ('a','b,',...,'z','_','1','2','3','4','5','6','7','8','9','0'))

```
@ensemble = (@chiffres, 'datte', 'kiwi', 12.45);
```

On dispose d'un scalaire spécial : `$#tableau` qui indique le dernier indice du tableau (et donc sa taille - 1) : `$fruits[$#fruits]` (*i.e.* 'cerise')

Il est initialisé à -1

On peut référencer qu'une partie d'un tableau

`@cartes[6..$#cartes]` (*i.e.* ('07','08','09','10','Valet','Dame','Roi'))

`@fruits[0..1]` (*i.e.* ('amande', 'fraise'))

Retour sommaire

Ajout d ’élément

- Le moyen le plus propre pour ajouter un élément à un tableau est d'utiliser l'expression :
  - `push(@mots, "fraise");`
- qui va “ pousser ” fraise à la fin du tableau @mots.
- Pour inclure deux éléments ou plus, on peut utiliser les formes suivantes :
  - `push(@mots,"fraise", "groseille");`
  - `push(@mots, ("fraise", "groseille"));`
- La commande `push` renvoie la longueur de la nouvelle liste.

Retour sommaire

Enlever un élément

- Pour enlever le dernier élément d'une liste, et le renvoyer, il faut utiliser la fonction `pop`.
- A partir de notre tableau original, la fonction `pop` renvoie “ cerises ” et `@mots` n'a plus que deux éléments.
  - `$derniermot = pop(@mots);`

Retour sommaire

Affectations multiples

- Les tableaux peuvent aussi être utilisés pour faire des affectations multiples vers des variables.
- `($a, $b) = ($c, $d);`
  - identique à `$a=$c; $b=$d;`
- `($a, $b) = @mots;`
  - `$a` et `$b` sont les deux premiers éléments de `@mots`

Retour sommaire

Opérateurs spéciaux

- L'opérateur $# donne l'indice du dernier élément du tableau et permet donc de connaître le nombre d'éléments contenus dans un tableau (indice du dernier élément + 1) :
  - `$#mots`

Retour sommaire

Conversion de « chaîne de caractères » à « tableaux » (1)

En Perl les chaînes de charactères et les tableaux sont étroitement liées.

On peut convertir de l'un à l'autre à l'aide des operateurs **split** et **join.**

split accepte une chaîne de caractères en entrée et la décompose en un tableau

split a deux arguments: une spécification de la séquence délimitant la chaîne à décomposer

Exemple

```
$c = "abcAAdefAAghj";
@t = split(/AA/, $c);
```

Résultat: `$t[0]` contient "abc", `$t[1]` contient "def", `$t[2]` contient "ghj"

la valeur de `$c` n'est pas modifiée

Retour sommaire

Conversion de « tableaux » à « chaîne de caractères » (2)

join accepte les éléments d'un tableaux et il le compose en une chaîne de charactères

join a deux arguments: une spécification de la séquence délimitant les éléments à composer

Exemple

```
t[0] = "abc"; t[1] = "def"; t[2] = "ghj";
$c = join("AA", @t);
```

Résultat: `$c` contient "abcAAdefAAghj"

Retour sommaire

Les tableaux associatifs (1)

- Un tableau associatif est une structure qui permet de mettre en relation deux éléments de type scalaire. C'est un ensemble de couples de scalaires où l'un des éléments (la clé) référence l'autre (la valeur). En pratique on considérera un tableau associatif comme un tableau classique où l'indice n'est pas un entier mais un scalaire.
- On accède à l'ensemble du tableau en préfixant l'identificateur par le caractère %. Pour accéder à un élément particulier, qui est un scalaire, on préfixe l'identificateur par un $ et on ajoute, entre accolades la valeur de l'indice.

Retour sommaire

Définition d ’un tableau associatif

- Pour définir un tableaux associatif, on utilise toujours la même notation, mais le tableau lui-même est préfixé par un %.
- Pour faire un tableau avec de noms de personnes et de leurs âges respectifs, on construit le tableau suivant :

```
%ages = ("Michel Dupont", 39, "Jean Daniel",
  34,"Armelle", 27,"Guillaume", "23","Jeanne
  Calment", 121);
```

Retour sommaire

Lecture du tableau associatif

- Pour accéder aux éléments du tableau, on exécute les commandes suivantes :
  - `$ages{"Michel Dupont"};` Renvoie 39
  - `$ages{"Jean Daniel"};` Renvoie 34
  - `$ages{"Armelle"};` Renvoie 27
  - `$ages{"Guillaume"};` Renvoie 23
  - `$ages{"Jeanne Calment"};` Renvoie 108
- Le signe % est remplacé par $ pour accéder aux valeurs scalaires. Et, contrairement aux tableaux, l'index (ici, le nom de la personne) est entre accolades `{}`.

Retour sommaire

Les tableaux associatifs (2)

Les *tableaux* sont des structures qui consistent de paires de clés et valeurs. La clé est toujours un index numérique. Les index sont consécutifs.

Les *tableaux associatifs* sont des structures qui consistent de paires de clés et valeurs où les clés ne sont pas nécessairement numériques (et donc on ne peut pas les ordonner).

Ils sont toujours précédés du caractère % :

```
%prix = ('amande',30, 'fraise',9, 'cerise',25);
```

On référence ensuite un élément du tableau par :

```
$prix{'cerise'}
```

Retour sommaire

Les tableaux associatifs (3)

Exemples:

```
%chiffre = ();
$chiffre{'un'} = 1;=>
print $chiffre{'un'};
$var = 'un'; print $chiffre{$var};
```

Remarque les tableaux associatifs utilisent les accolades, tandis que les tableaux utilisent les crochets.

Donc, `$x{2}` retourne la valeur associée à la clé 2

tandis que `$x[2]` retourne le troisième élément du tableau `x`.

Retour sommaire

Lecture du tableau associatif : keys et values

- Quand on fait un appel à `keys`, on obtient la liste des indices du tableau associatif : `keys %ages`
- Quand `values` est utilisé, on obtient la liste des valeurs des éléments : `values %ages`
- Ces fonctions renvoient leurs valeurs dans le même ordre, mais cet ordre n'a rien à voir avec l'ordre d'affectation des éléments.
- Quand `keys` et `values` sont utilisés dans un contexte scalaire, ils renvoient le nombre d'indice/éléments dans le tableau associatif.

Retour sommaire

Lecture du tableau associatif : each

- Il existe aussi une fonction `each` (chaque) qui renvoie les couples (indice, valeur).

```
while (($personne, $age) = each(%ages)) {
        print "L'âge de $personne est :
  $age \n";
}
```

Retour sommaire

Parcours de tableau associatif (1)

Pour afficher toutes les clefs et les valeurs d'un tableau associatif,

```
while (($key,$val) = each %thearray) {
     print $key, " ", $val, \n";
}
```

ou

```
foreach $key (keys(%prix)) {
     print $key, " ", $prix{$key}, \n";
}
```

Retour sommaire

Parcours de tableau associatif (2)


```
foreach $personne (keys %ages) {
print "L'âge de $personne est $ages{$personne}";
   }

foreach $age (values %ages) {
    print "$age \n";
   }
```

Retour sommaire

Conversion tableau – tableau associatif

- Un tableau associatif peut être converti dans un tableau, par simple affectation.
- Un tableau peut aussi être converti en tableau associatif, en l'affectant à un tableau associatif. Idéalement, le tableau aura un nombre pair d'éléments.

Retour sommaire

Conversion de tableaux

- `@info = %ages;`
  - `@info` est un tableau qui a maintenant 10 éléments
- `$info[5];` Renvoie 23
- `%autres_ages = @info;`
  - `%autres_ages` est un tableau associatif identique à `%ages`

Retour sommaire

Entrées/sorties

Entrées/sorties (1)

- La commande `<PTR_FICHIER>`, où `PTR_FICHIER` est un pointeur de fichier, permet d'accéder à un fichier

- (1) Les éléments du fichier sont lus soit ligne par ligne, en affectant le résultat de la commande à un scalaire:

```
$variable = <PTR_FICHIER>;
```

Retour sommaire

Entrées/sorties (2)

- (2) soit dans leur ensemble, en affectant le résultat de la commande à un tableau:

```
@tableau = <PTR_FICHIER>;
```

- Dans ce cas, chaque élément du tableau correspond à une ligne. On crée le pointeur d'un fichier à l'aide de la commande:

```
open(PTR_FICHIER, "fichier.txt");
```

Retour sommaire

Ouverture, fermeture, écriture

- Ouverture, fermeture

```
open(PTR_FICHIER,<nom du fichier sur disque>);
```

  - Cette commande permet d'ouvrir un fichier en lecture, il ne faut donc pas oublier de refermer le fichier après utilisation:

```
close(PTR_FICHIER);
```

  - Pour créer un fichier ou pour écraser un fichier déjà existant, il faut faire précéder le nom du fichier par un chevron fermant:

```
open(PTR_FICHIER,"><nom du nouveau fichier sur disque>");
```

- On écrit dans un fichier à l'aide de la commande print:

```
print $PTR_FICHIER "cette chaîne est écrite dans un fichier\n";
```

Retour sommaire

Fichiers prédéfinis

- `<STDIN>` Entrée standard.
  - Ce pointeur permet de lire (et uniquement de lire) le flux d'entrée du clavier.
- `<>` Désigne tous les fichiers donnés en argument au programme (ou l'entrée standard si aucun fichier n'est spécifié).
- `<STDOUT>` Sortie standard.
  - Ce pointeur permet d'écrire sur l'écran. C'est le pointeur par défaut de la commande print. L'instruction : “ `print STDOUT "..."` ” est équivalente à “ `print " . . . "` ”.

Retour sommaire

Structures de contrôle

Perl: le bloc

Un bloc est un ensemble de commandes entourées par des crochets ({}), chaque commande étant suivie d'un point-virgule.

Retour sommaire

Structures de contrôle : bloc

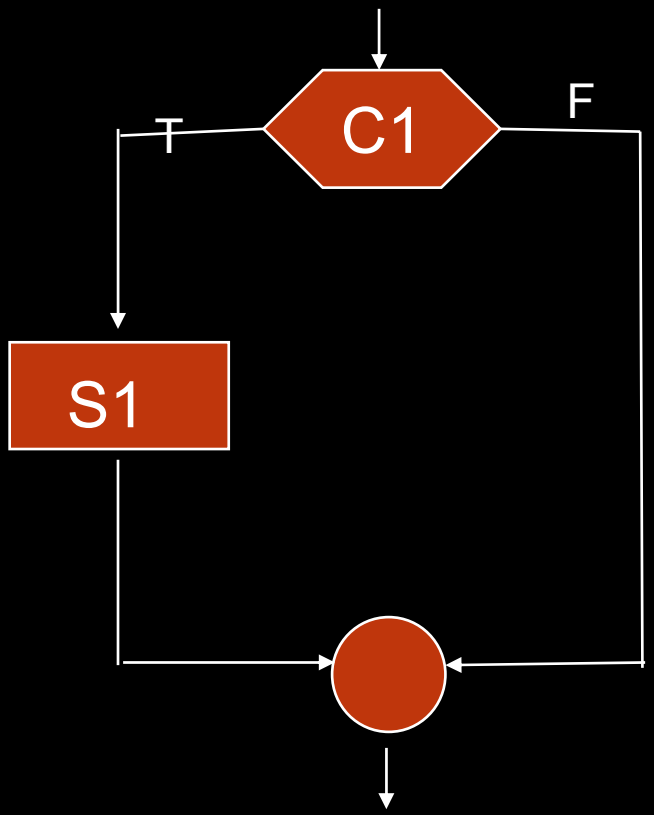
- On appelle bloc une ou plusieurs instructions, suivies par des points-virgules et obligatoirement encadrées par des accolades {. . . }.

```
{
Instruction1;
Instruction2;
Instruction3;
}
```

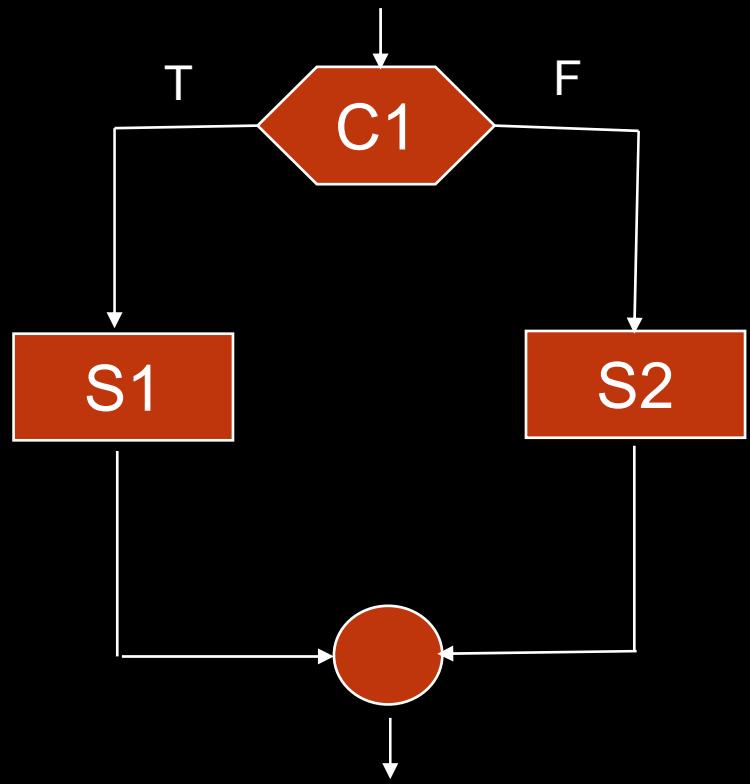
Retour sommaire

Diagramme des instructions conditionnelles

Instruction IF



Instruction IF-ELSE



Expressions conditionnelles (1)

```
if (condition) {    bloc; }
```

```
if ($x == 2) {
  print ''yes\n'';
}
```

```
if (condition) {     bloc; }
else {    bloc; }
```

```
if ($x == 2) {
  print ''yes\n'';
}
else {
  print ''no\n'';
}
```

```
if (condition) {     bloc; }
elsif {    bloc; }
else {    bloc; }
```

```
if ($x == 1) {
        print ''ONE\n'';
}
elsif ($x == 2){
        print ''TWO\n'';
}
else {
        print ''OTHER\n'';
}
```

Expressions conditionnelles (2)

```
$x = ''yes'';
$y = ''no'';
if ($x eq $y) {
        print ''1\n'';
}
else {
        print ''2\n'';
}
```

donne 2

```
$x = ''yes'';
$y = ''no'';
if ($x ne $y) {
        print ''1\n'';
}
else {
        print ''2\n'';
}
```

donne 1

Syntaxe: if (1)

1.

```
if (expression)
        bloc
```

2.

```
if (expression1 )
                    bloc1
      else (expression2)
             bloc2
```

Retour sommaire

Syntaxe: if (2)

3.

```
if (expression1 )
                    bloc1
     elsif (expression2)
           bloc2
.....
     elsif (expressionk)
                    block
     else
                    blocfaux
```

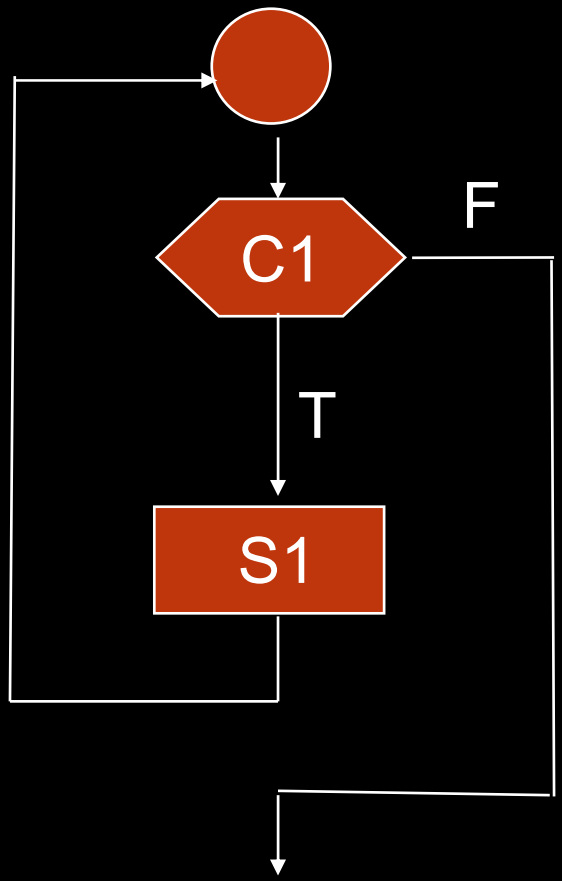
Retour sommaire

Syntaxe: if (3)

1. Si l'expression est évaluée à vrai, alors `bloc` est exécuté;
2. Si l'expression est évaluée à vrai, alors $\texttt{bloc}_1$ est exécuté, sinon $\texttt{bloc}_2$ est exécuté;
3. Si l'expression i est évaluée a vrai alors le $\texttt{bloc}_i$ est exécuté; si aucune expression n'est vraie alors le bloc $\texttt{bloc}_{\texttt{toutfaux}}$ est exécuté.

Retour sommaire

Diagramme de la boucle WHILE



Proprietés de WHILE

Deux chemins pour arriver à la condition C1

1. à partir de l'instruction précédente WHILE
2. de l'intérieur de la boucle

S1 doit changer la valeur de C1,
sinon on boucle à l'infini

Il est possible de ne jamais exécuter une boucle WHILE

Syntaxe: while

```
while (expression)
bloc
```

- Le bloc est exécuté tant que l'expression expression est évaluée à vrai.

Retour sommaire

Boucles while

Les boucles en perl sont très semblable à tout autres langages structurés. Comme d'habitude, il faut definir

la valeur initiale de la condition,

la condition de terminaison

et la fonction d'incrémentation.

```
$x = 0;
while($x < 5) {
    print $x,''\n'';
    ++$x;
}
```

Boucles while pour lire des données

Comme d'habitude, on peut utiliser la boucle pour lire des données.

Voici un programme qui lit le STDIN (la ligne de commande) et il l'affiche, ligne par ligne.

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

$_ est la varible prédéfinie qui contient la dernière ligne lue.

Remarquez qu'il ne faut pas spécifier explicitement qu'il faut lire la prochaine ligne.

Retour sommaire

Affichage du nombre de lignes d'un fichier :

```
#!/bin/perl
open (F, '< monfichier') || die "Problème d'ouverture : $!" ;
my $i=0;
while (my $ligne = <F>) {
     $i++;
}
close F;
print "Nombre de lignes : $i";
```

ouverture d'un fichier en lecture

Déclaration et initialisation du compteur

Détection d'erreur

pour chaque ligne lue...

Incrémentation du compteur

Fermeture du fichier

affichage du contenu du compteur

Parcours de fichiers : exemples

- Ces programmes affichent chaque ligne d'un fichier précédée de son numéro :
  - Dans ces différentes versions du programme, on utilise les mécanismes offerts par Perl pour simplifier l'écriture en utilisant par exemple la variable par défaut `$_` ou le fichier spécial `<>`.
  - On évite ainsi la manipulation de variables : c'est le cas pour la variable `$ligne`, ou pour la variable associée au fichier lu.

Retour sommaire

Parcours n°1


```
#!/usr/local/bin/perl
$numligne = 0;
while ($ligne=<STDIN>) {
  print $numligne."\t".$ligne;
  $numligne++;
  }
```

Retour sommaire

Parcours n°2

```
#!/usr/local/bin/perl
$numligne = 0;
while (<STDIN>) {
  print $numligne."\t".$_;
  $numligne++;
  }
```

Retour sommaire

Parcours n°3

```
#!/usr/local/bin/perl
$numligne = 0;
open(FILEINPUT,"$ARGV[0]");
while($ligne=<FILEINPUT>){
  print $numligne."\t".$ligne;
  $numligne++;
  }
close(FILEINPUT) ;
```

Retour sommaire

Parcours n°4

```
#!/usr/local/bin/perl
$numligne = 0;
open(FILEINPUT,"$ARGV[0]");
while(<FILEINPUT>){
  print $numligne."\t".$_;
  $numligne++;
  }
close(FILEINPUT) ;
```

(détour fichier spécial)

Retour sommaire

Parcours n°5

```
#!/usr/local/bin/perl
$numligne = 0;
while (<>) {
  print $numligne."\t".$_;
  $numligne++;
  }
```

Retour sommaire

Parcours de fichiers (complément 1)

Parcours

```
 open(F,$fichier)|| die " Pb pour ouvrir
$fichier:$!;
 while ($ligne = <F>) {
    print $ligne;
}
close F;
```

Retour sommaire

Parcours de fichiers (complément 2)

Fichier spécial : <> : fichier spécial en lecture qui contiendra ce qui est lu en entrée standard. Lorsque, dans une boucle `while`, on ne spécifie pas dans quelle variable on lit le fichier, la ligne lue se trouvera dans la variable spéciale `$_`.

Ainsi ce programme demandera d'écrire quelques lignes et affichera le nombre de lignes qu'il a lues.

```
#!/bin/perl
$i =0;
while (<>){
        $i++;
}
print "Nombre de lignes:$i";
```

Retour sommaire

Boucle foreach

En Perl, il y a un type de boucle très bien adapté au parcours de tableau, la boucle **foreach**

```
foreach $f (@fruits){
        print $f;
}
```

```
foreach $x (@c) {
        print $x,"\n";
}
```

Retour sommaire

Syntaxe: foreach


```
foreach variable (tableau)
bloc
```

- On affecte ainsi itérativement les valeurs du tableau à la variable `variable`.

Retour sommaire

Boucle for

Comme toujours, dans une boucle for la valeur initiale de la condition, la condition de terminaison, et la fonction d'incrémentation sont specifiés au début de la boucle

```
for ($count=0; $count<5; ++$count) {
   print $x,''\n'';
}
```

Retour sommaire

Boucle for

Exemple : Dans ce programme, on met des valeurs dans un tableaux, et on les affiche, un sur chaque ligne

```
$x[0] = 10;
$x[1] = 15;
$x[2] = 5;
for ($c=0;$c<=$#x;++$c) {
        print ''$x[$c]\n'';
}
```

Retour sommaire

Expressions logiques (1)

- La syntaxe utilisée pour les expressions est presque identique à celle utilisée en C. Cependant, le fait d'utiliser des variables non fortement typées impose des opérateurs différents selon le type de test à effectuer.
- On distingue ainsi deux groupes d'opérateurs selon le type de variable: les variables représentant des chaînes de caractères, celles représentant des nombres, entiers ou non. Pour les valeurs numériques, les opérateurs sont identiques à ceux du C.
- Lecture : les opérateurs Perl

Retour sommaire

Expressions logiques (2)

- Nous indiquons ci-dessous les opérateurs de chaînes de caractères

- Opérateur `eq` :
  - `$a eq $b` est vrai si les deux chaînes sont égales ;
- Opérateur `ne` :
  - `$a ne $b` est vrai si les deux chaînes sont différentes ;
- Opérateur `lt` :
- `$a lt $b` est vrai si `$a` est plus petite que `$b` au sens lexicographique ;
- Opérateur `le` :
  - `$a le $b` est vrai si `$a` est plus petite ou égale que `$b` au sens lexicographique ;
- Opérateur `gt` :
  - `$a gt $b` est vrai si `$a` est plus grande que `$b` au sens lexicographique ;
- Opérateur `ge` :
  - `$a ge $b` est vrai si `$a` est plus grande ou égale que `$b` au sens lexicographique.

Retour sommaire

Expressions logiques (3)

- On peut, comme en C, combiner plusieurs expressions à l'aide de :
  - “&&” (et)
  - “ || ” (ou)
  - “ ! ” (négation
  - Une chaîne vide, ou la valeur 0 (ce qui revient au même pour un scalaire), a pour valeur faux.

Retour sommaire

Comparaison (1)

- **Comparaison de chiffres**

Ce sont les opérateurs habituels : `>, >=, <, <=, ==, !=`

respectivement: supérieur à, supérieur ou égal, inférieur à, inférieur ou égal, égal, différent

Attention: = est une affectation, == est une comparaison

- **Comparaison de chaînes** `gt, ge, lt, le, eq, ne`

respectivement: supérieur à (selon l'ordre alphabétique), supérieur ou égal, inférieur à, inférieur ou égal, égal, différent

Attention: Ne pas confondre la comparaison de chaînes et d'entiers

`'b' == 'a'` => évalué comme étant vrai

il faut écrire : `'b' eq 'a'` => évalué faux

Retour sommaire

Comparaison (2)

- **Comparaison de booléens**

Même si le type booléen n'existe pas en tant que tel, des opérateurs existent :

|| , or (ou inclusif) &&, and (et) !, not (négation)

Exemples `( ! 2 < 1)` => vrai

`(1 < 2) && (2 < 3)` => vrai

`($a < 2) || ($a == 2)` équivaut à `($a <= 2)`

`(!$a && !$b)` équivaut à `!($a || $b)`

(règle de Morgan)

Retour sommaire

Perl et les expressions régulières

Expressions régulières

- Perl offre la possibilité d'utiliser le langage des expressions régulières au travers d'opérateurs ou de fonctions.

Retour sommaire

Alphabet, mot, langage

Définition Un *alphabet* Σ est un ensemble fini des symboles, comme par exemple des lettres, de chiffres et d'autre sigles.

Exemples Σ = {a,b,c} Σ={0,1}

Définition Un *mot* w défini sur un alphabet est une suite ou séquence finie de symbole appartentant à Σ.

Exemple si Σ = {a,b,c}, w = abc est un mot défini sur Σ

Définition Un *langage* L défini sur un alphabet Σ est un ensemble de mots définis sur l'alphabet en question. Cet ensemble de mots peut être infini.

Exemple Soit l'alphabet Σ = {a,b,c}, le langage L ={acbb,accbb,acccbb,…} est le langage de tous le mots qui débutent par le symbole a, suivi d'au moin un symbole c et qui se terminent par deux symboles b.

Retour sommaire

Opérations sur les langages (1)

Union L'union de deux langages L_1 et L_2 définis sur un alphabet Σ (L_1 U L_2) correspondent exactement à l'union d'ensemble.

Concaténation Soient L_1 et L_2 deux langages définis sur un alphabet Σ, l'opération de concaténation est définie comme suit

$$L_1 . L_2 = \{w_1.w_2 , \text{ t.q. } w_1 \text{ appartient à } L_1 \text{ et } w_2 \text{ appartient à } L_2 \}$$

L'élément neutre est donc l'ensemble $L\varepsilon = \{\varepsilon\}$ ($L. L\varepsilon = L$, $L\varepsilon .L = L$)

Exemple

Soit Σ= {a,b,c}, L_1 ={a,b} et L_2 ={bac,b,a}, on a $L_1 . L_2$ = {abac,bbac,ab,bb,aa,ba}

Retour sommaire

Opérations sur les langages (2)

Puissance La puissance d'un langage L, notée L^n où n >= 0, est définie par

1) $L^0 = \{\varepsilon\}$

2) $L^{n+1} = L^n . L$

Kleene star ou fermeture itérative La ferméture itérative (ou de Kleene) d'un langage L, notée L*, est l'ensemble de mots résultant d'une concaténation d'un nombre fini de mots de L. Formellement

$$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \ldots$$

Le mot vide appartient donc à L* mais pas à L^+ qui est défini comme suit

$$L^+ = L^1 \cup L^2 \cup L^3 \ldots = L . L^*$$

Exemples

Soit $\Sigma = \{a\}$, $L^0 = \{\varepsilon\}$, $L^1 = \{a\}$, $L^2 = \{aa\}$, $L^3 = \{aaa\}$,….,

$L^+ = \{a,aa,aaa,aaaa,aaaaa,…..\}$

$L^* = \{\varepsilon, a,aa,aaa,aaaa,aaaaa,…..\}$

Retour sommaire

Formalisme de spécification des langages

Pour spécifier (décrire formellement) un langage différents formalismes sont à disposition. La première solution serait de énumérer de manière exhaustive les composants du langage.

Cela est une définition extensionelle. C'est irréalisable si le langage est infini.

Pour décrire de langage infini on utilise des formalismes plus riches, comme les grammaires, les machine de Turing etc.

Le formalisme le plus simple sont les expressions rationnelles où regulières.

Retour sommaire

Qu'est-ce une expression rationnelle (régulière)?

Définition Soit un alphabet Σ, une *expression régulière* définie sur Σ ainsi que les ensembles qu'elle dénote sont définis récursivement comme suit:

1) *Ø* est une expression régulière dénotant l'ensemble vide,

2) *ε* est une expression régulière dénotant l'ensemble *{ε}*,

3) pour tous *a* dans Σ, *a* est une expression régulière dénotant l'ensemble *{a}*,

4) si *r* et *s* sont des expressions régulières dénotant respectivement les ensembles R et S, alors *(r + s)*, *(rs)*, *(r*)* sont des expressions régulières dénotant respectivement les ensembles R U S, R . S, et R*,

5) rien d'autre n'est consideré comme des expressions régulières

Le langage que dénote une expression régulière *r* est noté *L(r)*

Retour sommaire

Expressions rationnelles - exemples

Soit l'alphabet Σ = {a,b} et l'expression régulière *r* = (a(a+b)*), alors le langage généré par r est L(r) ={a} . {a,b}*

Son extension contient donc {a,aa,aaa,aaaa,…,ab,aba,abbaaababa, ….}

Soit l'alphabet Σ = {0,1}, l'expression régulière s= (0*1)* dénote l'ensemble

L(s) = {{0}*. {1}}* ou bien

L(s) = {x t.q. x appartient à {0,1}* et x représente un nombre impair}

Son extension contient donc {ε,1,01,010101, 100001}

Retour sommaire

Perl et les expressions régulières

Perl offre une grande puissance de manipulation d'expressions régulières. En Perl on utilise les expression régulières en tant que motif de recherche. On utilise l'opérateur conditionnel =~ qui signifie "ressemble à" (matches).

Syntaxe: `chaîne =~/expression/`

Exemple: `if ($nom =~ /^[Dd]upon/) {print "OK !";}`

=> Ok si nom est 'dupont', 'dupond', 'Dupont-Lassoeur'

^ signifie « commence par »

On peut rajouter i derrière l'expression pour signifier qu'on ne différencie pas les majuscules des minuscules.

Le contraire de l'opérateur `=~` est `!~` (ne ressemble pas à ...)

```
if ($nom !~ /^dupon/i) {print "Non...";}
```

Retour sommaire

Recherche de motif

- Cet opérateur peut être comparé à l'effet de la commande UNIX grep sur un fichier d'une seule ligne.

```
$phrase = ~/<expression régulière>/;
```

- a pour valeur vrai si *l'expression régulière* apparaît dans la variable `$phrase`.

Retour sommaire

~///; : exemple

- Par exemple, si

```
$phrase=«<Nom>Fleury</Nom><Pren>Serge</Pren>… »;
$phrase=~/<Nom>(.*)<\/Nom><Pren>(.*)<\/Pren>/;
```

- Cette dernière expression est vraie

Retour sommaire

Perl: expressions régulières (1)

\s = espace ou tabulation ou retour-chariot

\n = retour-chariot

\t = tabulation

^ = début de chaîne

$ = fin de chaîne

a = a

. = un caractère quelconque sauf fin de ligne

Retour sommaire

Perl: expressions régulières (2)

[a-z]	tous caractères minuscules
[aeiouy]	toute voyelle
[a-zA-Z0-9]	tout caractère alphanumérique
^	au début d'un ensemble indique le complément de l'ensemble
[^0-9]	tout caractère non numérique
\w	signifie «une lettre alphanumérique» (sans àé etc)
\W	«tout sauf une lettre alphanumérique»
\S	«tout sauf un \s»

Retour sommaire

Perl: expressions régulières (3)

Quelques opérateurs :

« ? » 0 ou 1 fois, a? = 0 ou 1 a

« * » 0 ou n fois, a* = 0 ou plusieurs a

« + » 1 ou n fois, a+ = 1 ou plusieurs a

$(ab)^+$ = 1 ou plusieurs ab

« | » : ou (inclusif) a|b = a ou b

[^abc] tous caractères qui ne sont pas a ou b ou c

{n,m} de n à m fois a{1,4} = 1 à 4 a

Retour sommaire

Perl: expressions régulières (4)

Remarque: Si l'on veut qu'un caractère spécial apparaisse tel quel, il faut le précéder d'un « back-slash » (\), les caractères spéciaux sont : « `^ | ( ) [ ] { } \ / $ + * ? .` »

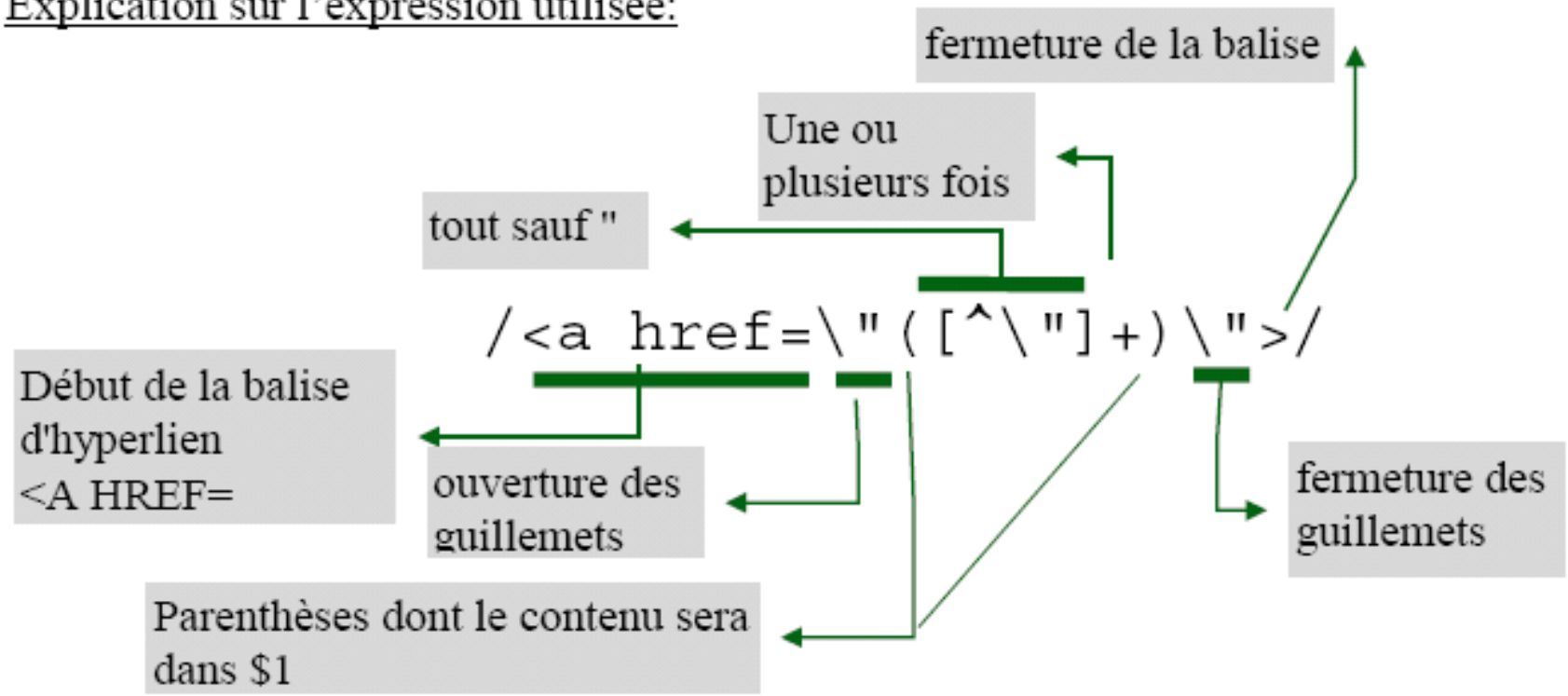
Exemple :

```
if (/\(.*\)/) {print;}
```

(recherche de parenthèses)

Retour sommaire

Explication sur l'expression utilisée:



Perl: expressions régulières - exemples

Pour vérifier si une chaîne $x contient la chaîne``abc''

```
if ($x =~ /abc/) { . . . }
```

Pour vérifier si une chaîne $x commence par la chaîne``abc''

```
if ($x =~ /^abc/) { . . . }
```

Pour vérifier si une chaîne $x commence par une majuscule

```
if ($x =~ /^[A-Z]/) { . . . }
```

Pour vérifier si une chaîne $x ne commence pas par une minuscule `if ($x =~ /^[^a-z]/) { . . . }`

Ici le prémier ^ signifie le début de la chaîne, tandis que le deuxième ^ à l'interieur des crochets indique le complément de l'ensemble indiqué.

Retour sommaire

Recherche et remplacement (~s///)

- La commande :

```
$phrase = ~s/<expression régulière>/<motif de remplacement>/<options>;
```

- remplace dans la variable `$phrase` la ou les occurrences de l'expression régulière par le motif de remplacement.

Retour sommaire

~s/// : exemple

- Par exemple, si

```
$phrase=«<Nom>Fleury</Nom><Pren>Serge</Pren>… »;
$phrase=~s/<Nom>(.*)<\/Nom><Pren>(.*)<\/Pren>/Nom : $1, Prénom : $2/;
```

- `$phrase` contient :
  - `« Nom : Fleury, Prénom : Serge … »`
  - Une variable $i fait référence à la sous chaîne reconnue par la ième partie de l ’expression régulière mise entre parenthèses

Retour sommaire

Perl- Remplacement (1)

Exemples:

```
$fruit =~ s/e$/es/;
```

remplace un e final par es

```
$tel =~ s/^99\./02.99\./;
```

remplace des numéros de téléphone par une nouvelle numérotation

Retour sommaire

Perl- Remplacement (2)

On peux référencer une partie du motif dans le remplacement avec `$1` (`$1` est une variable correspondant au contenu de la première parenthèse).

Exemple : Transformer automatiquement les noms d'arbre par "arbre à fruit"

```
$texte =~ s/([a-z]+)ier /arbre à $1es /;
```

'cerisier' sera traduit par 'arbre à cerises' (contenu de $1 => 'ceris')

'manguier' => 'arbre à mangues'

`$2` correspond à la deuxième parenthèse, `$3` à la troisième etc.

Les options `s/exp/remp/i;` => Indifférenciation minuscules/majuscules

`s/exp/remp/g;` => Remplace toutes les occurrences

(pas seulement la première)

Exemple: Pour remplacer un texte par le même en majuscule (`\U`)

```
s/([a-z])/\U$1/g;
```

Retour sommaire

Perl- Remplacement (3)

Pour remplacer les suites de plusieurs a en un seul a:

```
$x =~ s/a*/a/g;
```

Pour enlever les suites de a:

```
$x =~ s/a*//g;
```

Pour enlever les espaces au début d'une chaîne:

```
$x =~ s/\s+//g;
```

Retour sommaire

(~tr///)

- La commande suivante :

```
$cpt=$phrase=~tr/<classe_de_caractère_initiale>/<nouvelle_classe_de_caractère>/<options>;
```

- remplace dans la variable `$phrase`, tous les caractères appartenant à la `<classe de caractère initiale>` par leur équivalent dans la `<nouvelle classe de caractère>` et retourne le nombre de caractères remplacés.

Retour sommaire

~tr : exemple

- Par exemple : si

```
$phrase=  “ <body><html><head></head>… ”
$cpt = $phrase = ~tr/<>/[]/ ;
```

- `$cpt` est égal à 8, (8 changements ont été effectués)
- `$phrase` contient:

  “ `[body][html][head][/head]…` ”

Retour sommaire

Passage de paramètres

Passage de paramètres au programme

En Unix on peut appeler un programme Perl en lui donnant des paramètres, comme on le fait pour les commandes Unix.

Les paramètres sont stockés dans un tableau spécial: `@ARGV`

Le premier paramètre est donc `$ARGV[0]`

`$#ARGV` est l'indice du dernier element du tableau `@ARGV`, donc

`$ARGV[$#ARGV]` est la valeur du dernier élément du tableau

Exemple

```
open(P, $ARGV[0]) || die "Couldn't open $ARGV[0]: $!\n";
```

Retour sommaire

Gestion des fichiers

Gestion des fichiers (1)

Ouverture L'ouverture consiste (le plus souvent) à associer un descripteur de fichier (filehandle) à un fichier physique

(1) **en lecture** :

```
open(FENT, 'fichier');
```

ouverture d'un fichier, référencé ensuite par FENT

(2) **en écriture**

```
open(FSOR, '> fichier');
```

écriture du fichier, si ce fichier existait auparavant, alors l'ancien contenu est écrasé.

```
open(FSOR,'>>fichier');
```

écriture à la fin du fichier. Le fichier est créé si besoin

Retour sommaire

Gestion des fichiers (2)

Gestion des erreurs (||) Lorsque l'on ouvre un fichier il se peut qu'il y ait une erreur.

En lecture : le fichier n'existe pas, ou ne peut pas être lu (droits d'accès)...

En écriture : Le fichier existe mais on n'a pas le droit d'écrire dessus

Il faut prendre l'habitude, quand on ouvre un fichier, de détecter l'erreur éventuelle. On peut le faire sous la forme suivante :

```
open (F, ...) || die "Problème d'ouverture";
```

On peut, et c'est même conseillé, récupérer le texte de l'erreur contenu dans la variable `$!`

```
open (F, ...) || die "Problème d'ouverture : $!";
```

Retour sommaire

Gestion des fichiers (3)

Fermeture

```
close FENT; close FSOR;
```

Lecture

```
$ligne = <FENT>;
```

```
$reponse = <STDIN>;
```
=> lecture d'une ligne à l'écran

N.B La fin de ligne (retour-chariot) est lue également. Pour enlever cette fin de ligne il suffit d'utiliser la commande **chomp** (enlève le dernier caractère uniquement si c'est un retour-chariot)

On peut lire toutes les lignes d'un fichier dans un tableau (en une seule instruction)

```
@lignes = <FENT>;
```

Retour sommaire

Gestion des fichiers (4)

Ecriture

```
print FSOR 'DUPONT Jean';

print FMAIL 'Comment fait-on pour se
connecter SVP?';

print STDOUT "Bonjour\n";

print STDERR 'Je déteste les oranges !';
```

Retour sommaire

Procédures/Fonctions

Procédures

- Un sous-programme est une partie de code séparée conçue pour exécuter une tâche particulière. Ils se placent n'importe où dans le programme.
- Cependant, il est préférable de les regrouper, soit au début, soit à la fin.

Retour sommaire

Déclaration


```
sub procedure {
        bloc ;
}
```

Retour sommaire

Appel

- `&procedure;`
- L'interpréteur de Perl utilise le caractère & pour indiquer qu'un sous-programme est spécifié dans l'instruction.
- Avec Perl 5, il n'est plus nécessaire d'indiquer ce caractère lors de l'appel du sous-programme si celui-ci a déjà été défini. Si on place les sous-programmes à la fin du programme, il reste nécessaire d'utiliser le caractère & lors de l'appel.

Retour sommaire

Utilisations

- On peut aussi établir une référence préalable à un sous-programme défini en fin de programme et ainsi ne pas utiliser le caractère & lors de l'appel :

```
sub procedure;
...
procedure ;
...
sub procedure {
        bloc ;
}
```

- Avec paramètre(s)
  - `&proc('parametre');`

Retour sommaire

Fonctions

- Une fonction est une procédure qui retourne une valeur.
- Exemple complet
  - un programme « simplissime » qui met au pluriel des mots en leur ajoutant un « s »...

Retour sommaire

- **Exemple 1 : fonction « pluriel »**

```
sub plurielSurTableau {
   my (@mots) = @_ ;
   foreach $mot (@mots) {
    $mot .= ‘s’ ;
    }
return(@mots) ;
}
```

- **Exemple 2 : fonction « pluriel »**

```
sub plurielSurMot {
   my ($mot) = shift ;
   $mot .= ‘s’ ;
   return($mot) ;
}
```

Appel

- l’appel est fait de la manière suivante :

```
@mot_au_pluriel = &plurielSurTableau('mot1 ’,
  'mot2');
$mot_au_pluriel = &plurielSurMot('mot');
```

  - Le tableau `@mot_au_pluriel` contient les chaînes « mot1s » et « mot2s ».
  - La variable `$mot_au_pluriel` contient la chaîne « mots »
  - Le passage de paramètres se fait à l'aide du tableau spécial @_
  - L'instruction « my » réalise une affectation dans des variables locales à la procédure (*cf. infra*) avec les éléments du tableau.
    - Ce type de passage de paramètre est très efficace car le nombre de paramètres n'est pas forcément fixe.

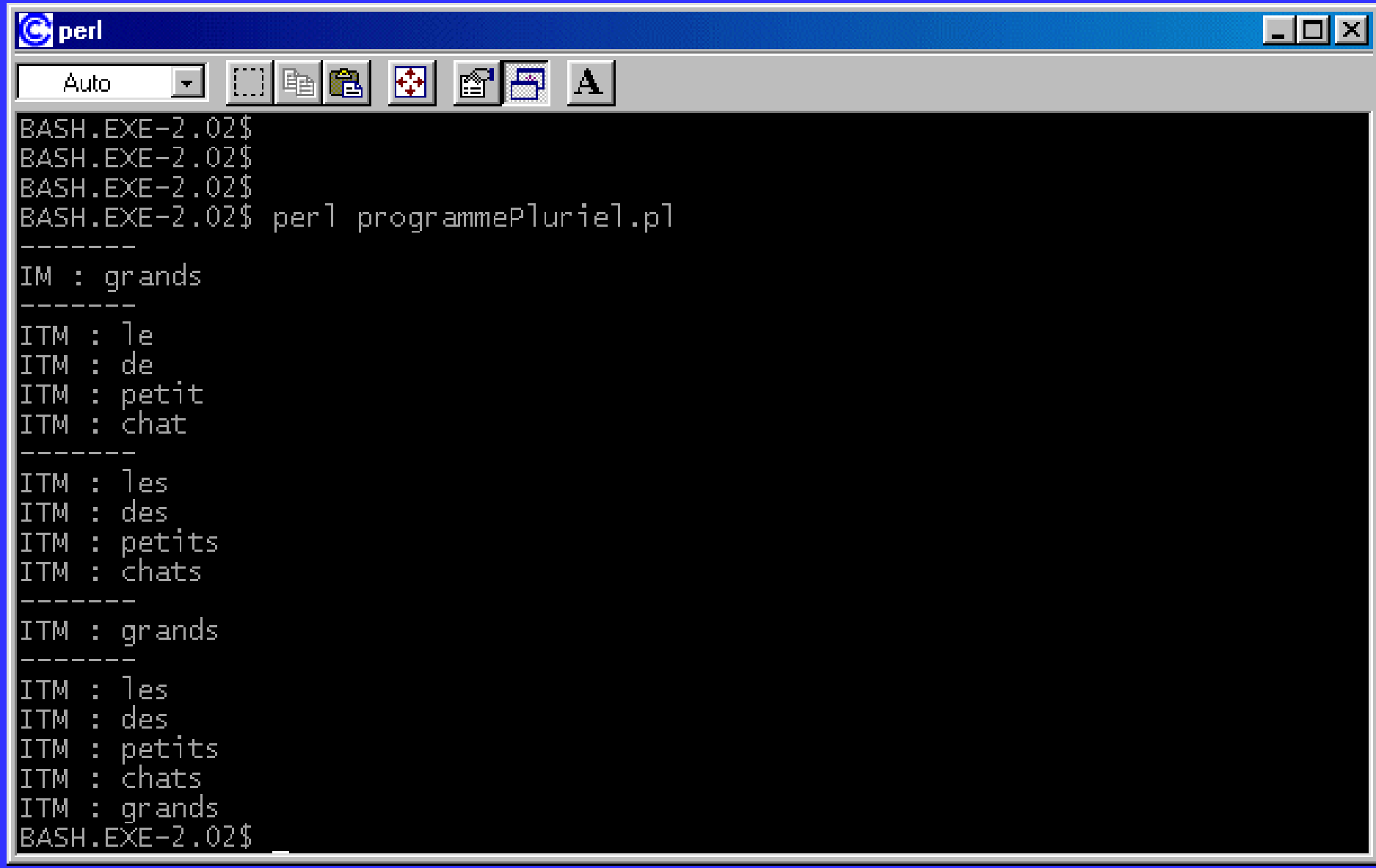
Retour sommaire

- Programme complet
- exécution

Retour sommaire

```
$monmot="grand";
$motaupluriel=&plurielSurMot($monmot);
&imprimeMot($motaupluriel);

@tableaudemots=("le", "de", "petit", "chat");
&imprimeTableauDeMots(@tableaudemots);

@motsaupluriel=&plurielSurTableau(@tableaudem
ots);
&imprimeTableauDeMots(@motsaupluriel);

@motsaupluriel=&plurielSurTableau($monmot);
&imprimeTableauDeMots(@motsaupluriel);

@motsaupluriel=&plurielSurTableau(@tableaudem
ots, "grand");
&imprimeTableauDeMots(@motsaupluriel);
```

```
sub plurielSurTableau {
    my (@mots) = @_;
    foreach $mot (@mots) {
      $mot .= "s";
    }
    return(@mots);
}
sub plurielSurMot {
    my ($mot) = shift;
    $mot .= "s";
    return($mot);
}
sub imprimeTableauDeMots {
    print "-------\n";
    my (@mots) = @_;
    foreach $mot (@mots) {
      print "ITM : $mot\n";
    }
}
sub imprimeMot {
    print "-------\n";
    my ($mot) = shift;
    print "IM : $mot\n";
}
```


perl
Auto
BASH.EXE-2.02$
BASH.EXE-2.02$
BASH.EXE-2.02$
BASH.EXE-2.02$ perl programmePluriel.pl

IM : grands

ITM : le
ITM : de
ITM : petit
ITM : chat

ITM : les
ITM : des
ITM : petits
ITM : chats

ITM : grands

ITM : les
ITM : des
ITM : petits
ITM : chats
ITM : grands
BASH.EXE-2.02$

```
#le programme prend un répertoire en argument
$ARGV[0] =~ s/\\$//;
#début du parcours
Show_Dir($ARGV[0]);

sub Show_Dir {
      my $path = shift(@_);
      opendir(DIR, $path) or die "can't open $path: $!\n";
      my @files = readdir(DIR);
      closedir(DIR);
      foreach my $file (@files) {
            next if $file =~ /^\.\.?$/;
            $file = $path."/".$file;
            if (-d $file) {
                  print "On entre dans le REP : $file\n";
                  Show_Dir($file);  #recursion!
            }
            if (-f $file) {
                  print "$file\n";
                }
      }
}
```

Appel récursif de procédure : parcours d ’une arborescence sur un disque et impression des fichiers uniquement...

Fonctions/variables prédéfinies

Fonctions prédéfinies (1)

- Quelques fonctions offertes par Perl pour manipuler les données.
- Système
  - print : permet d'afficher un message, ou le contenu de variables.
  - Quelques caractères spéciaux affichables avec “ print ” :
    - \n : retour-chariot
    - \t : tabulation
    - \b : bip

Retour sommaire

Fonctions prédéfinies (2)

- `exit`
  - permet d'arrêter le programme en cours

```
if ($erreur) {exit;}
```

- `die`
  - permet d'arrêter le programme en cours en affichant un message d'erreur.

```
if ($mot eq 'arret') {die 'Je m’arrete !'}
```

- `system`
  - permet de lancer une commande système

```
system 'mkdir repertoire';
```

- `sleep n`
  - le programme “ dort ” pendant n secondes

Retour sommaire

Fonctions prédéfinies (3)

- Mathématique
  - Les fonctions mathématiques habituelles existent aussi en Perl : `sin`, `cos`, `tan`, `int` (partie entière d'un nombre), `sqrt`, `rand` (nombre aléatoire entre 0 et n), `exp` (exponentielle de n), `log`, `abs` (valeur absolue).

Retour sommaire

Chaînes de caractères

`chop(ch)`

enlève le dernier caractère de la chaîne

`chomp(ch)`

enlève uniquement un “ retour-chariot ”

`length(ch)`

retourne la longueur de la chaîne (nombre de caractères)

`uc(ch)`

retourne la chaîne en majuscules (Perl 5)

`lc(ch)`

retourne la chaîne en minuscules (Perl 5)

`split('motif', ch)`

sépare la chaîne en plusieurs éléments (le séparateur étant motif). Le résultat est un tableau.

`substr(ch, indicedébut, longueur)`

retourne la chaîne de caractère contenue dans ch, du caractère indicedébut et de longueur longueur.

`index(ch, recherche)`

retourne la position de recherche dans la chaîne ch

Retour sommaire

Fonctions predéfinies – chaînes de caractères

chop(ch) Enlève le dernier caractère de la chaîne

```
$ch='cerises'; chop($ch);
```
=> ch contient 'cerise'

chomp(ch) Même chose que « chop » mais enlève uniquement un "retour-chariot" en fin de chaîne

length(ch) Retourne le nombre de caractères de la chaîne

```
length('cerise')
```
=> 6

uc(ch) Retourne la chaîne en majuscules

```
$ch = uc('poire')
```
=>'POIRE'

lc(ch) Retourne la chaîne en minuscules

```
$ch = lc('POIRE')
```
=>'poire'

Retour sommaire

Fonctions prédéfinies – chaînes de caractères

substr(ch, indicedébut, longueur)

Retourne la chaîne de caractère contenue dans ch, du caractère indicedébut et de longueur longueur

```
$ch=substr('dupond', 0, 3)    => 'dup'
$ch=substr('Les fruits', 4)   => 'fruits'
```

index(ch, souch, début)

Retourne la position de la première instance de souch dans ch

Si début est spécifié, elle donne la position de départ pour la recherche

```
$i=index('Le temps des cerises','cerise');  => 13
```

Retour sommaire

Tableaux, listes

- `grep(/expression/, tableau)` : recherche d'une expression dans un tableau ; grep retourne un tableau des éléments trouvés.
- `join(ch, tableau)` : regroupe tous les éléments d'un tableau dans une chaîne de caractères (en spécifiant le séparateur)
- `pop (tableau)` : retourne le dernier élément du tableau (et l'enlève)
- `push (tableau, element)` : ajoute un élément en fin de tableau (contraire de pop)
- `shift(tableau)` : retourne le premier élément du tableau (et l'enlève)
- `unshift (tableau, element)` : ajoute un élément en début de tableau
- `sort (tableau)` : tri le tableau par ordre croissant
- `reverse (tableau)` : inverse le tableau
- `splice (tableau, début, nb)` : enlève nb éléments du tableau à partir de l'indice début

Retour sommaire

Tableaux indicés

- `each(tableau_indicé)` : les couples clé/valeurs d'un tableau indicé
- `values(tableau_indicé)` : toutes les valeurs d'un tableau indicé (sous la forme d'un tableau)
- `keys(tableau_indicé)` : toutes les "clés" d'un tableau indicé
- `exists(élément)` : indique si un élément a été défini
- `delete(élément)` : supprimer un élément

Retour sommaire

Fonctions prédéfinies – tableaux, listes

pop (tableau) Retourne le dernier élément du tableau (et l'enlève)

`print pop(@fruits);` => affiche 'cerise', @fruits devient ('amande','fraise')

push (tableau, element) Ajoute un élément en fin de tableau (contraire de pop)

`push(@fruits, 'abricot');` => @fruits devient ('amande','fraise','abricot')

shift(tableau) Retourne le premier élément du tableau (et l'enlève)

`print shift(@fruits)` => Affiche 'amande', @fruits devient ('fraise','abricot')

unshift (tableau, element) Ajoute un élément en début de tableau

`unshift ('poire', @fruits);` => @fruits devient ('poire', 'fraise','abricot')

Retour sommaire

Fonctions prédéfinies – tableaux, listes

sort (tableau) Tri le tableau par ordre croissant

```
@fruits = sort (@fruits);
```

=> @fruits devient ('abricot', 'coing', 'fraise')

reverse (tableau) Inverse le tableau

```
@fruits = reverse(@fruits);
```

=> @fruits devient ('fraise', 'coing', 'abricot')

Retour sommaire

Fonctions prédéfinies – tableaux, listes

splice (tableau, début, nb, liste) Remplace nb éléments du tableau à partir de l'indice début avec les éléments dans liste

si @fruits = ('poire', 'fraise', 'abricot', 'cerise') et
@legumes = ('carotte', 'tomate', 'celeri')

```
@fruits2 = splice(@fruits, 1, 2, @legumes);
```

=> @fruits = ('poire', 'carotte', 'tomate', 'celeri', 'cerise')
@fruits2 = ('fraise',abricot')

Retour sommaire

Fonctions prédéfinies – tableaux, listes

splice (tableau, début, nb, liste) Remplace nb éléments du tableau à partir de l'indice début avec les éléments dans liste

On l'utilise beaucoup dans les cas « limite »

```
$a = shift(@abz)
$z = pop(@abz)
unshift (@abz,$a)
push(@abz,$z)
```

```
$a = splice(@abz,0,1)
$z = splice(@abz,$#abz,1)
splice(@abz,0,0,$a)
splice(@abz,$#abz+1,0,$z)
```

Retour sommaire

Gestion de fichiers (1)

- Ouverture
  - L'ouverture consiste (le plus souvent) à associer un descripteur de fichier (`filehandle`) à un fichier physique.
- Lecture
  - `open (FileInput, 'fichier');`
    - Ouverture d'un fichier, référencé ensuite par `FileInput`.

Retour sommaire

Gestion de fichiers (2)

- Ecriture
  - `open (FileOutput, '> fichier');`
    - Ecriture du fichier, si ce fichier existait auparavant : l'ancien contenu est écrasé.
  - `open (FileOutput,'>>fichier');`
    - Ecriture à la fin du fichier, Le fichier est créé si besoin.
- Fermeture
  - Commande close :
  - `close FileInput;`
  - `close FileOutput;`

Retour sommaire

Gestion de fichiers (3)

- Deux fichiers spéciaux: `STDOUT`, `STDERR` (respectivement: sortie standard, et sortie erreur), par défaut l'écran.
- Un fichier spécial: `STDIN`, le clavier (entrée standard).

Retour sommaire

Gestion des erreurs (||)

- Lorsque l'on ouvre un fichier il se peut qu'il y ait une erreur. Il faut prendre l'habitude, quand on ouvre un fichier, de détecter l'erreur éventuelle. On peut le faire sous la forme suivante : (dérivée du C)

```
if (! open (FileInput, ...)) {
die "Problème à l'ouverture du fichier";
        }
```

Retour sommaire

Gestion des erreurs (||)

- Ou sous la forme plus simple et plus usitée en Perl :

```
open(FileInput, ...) || die "Pb
  d'ouverture";
```

- On peut, et c'est même conseillé, récupérer le texte de l'erreur contenu dans la variable $! :

```
open(FileInput, ...) || die "Pb
  d'ouverture : $!";
```

Retour sommaire

Gestion de fichiers (4)

- Lecture
  - `$ligne = <FILEINPUT>;`
    - ⇨ lecture d'une ligne dans le fichier associé au pointeur `FILEINPUT`
  - La fin de ligne (retour-chariot) est lue également.
    - Pour enlever cette fin de ligne il suffit d'utiliser la commande `chop`, ou son équivalent : `chomp` (enlève le dernier caractère uniquement si c'est un retour-chariot).
  - On peut lire toutes les lignes d'un fichier dans un tableau (en une seule instruction) :

```
@lignes = < FILEINPUT >;
```

Retour sommaire

Gestion de fichiers (5)

- Ecriture
  - `print FILEOUTPUT " message ";`
- Fichier spécial : <>
  - Perl offre une fonctionnalité bien pratique : l'utilisation d'un fichier spécial en lecture qui contiendra ce qui est lu en entrée standard. Lorsque, dans une boucle " `while` ", on ne spécifie pas dans quelle variable on lit le fichier : la ligne lue se trouvera dans la variable spéciale " `$_` ".

Retour sommaire

Gestion de fichiers (exemple 1)

```
#!/usr/bin/perl
print ''What file do you want to read?'';
$filename = <STDIN>;
chomp($filename);
open (TEXT, $filename)||die ''Can't open file'';
while ($line = <TEXT>) {
print $line;
}
close(TEXT);
exit;
```

Gestion de fichiers (exemple 2)

```
#!/usr/local/bin/perl
open(OUTPUT,“>out.txt");
print ''What file do you want to read?'';
$filename = <STDIN>;
chomp($filename);
open (TEXT, $filename)||die ''Can't open file'';
while ($inputline = <TEXT>) {
       @line = split(/ /,$inputline);
      foreach $word (@line)       {
               $wordlist{$word}++;
      }
}
while (($key, $value) = each(%wordlist))    {
       print "$key => $value\n";
       print OUTPUT "$key => $value\n";
}
end;
close(TEXT);
close(OUTPUT);
```

Calcul du nb d’occurrence de formes dans un fichier

Variables spéciales (1)

- Ce sont les variables sous la forme `$c` (avec c un caractère non alphabétique) :
- `$_`
  - La dernière ligne lue (au sein d'une boucle “ `while` ”).
- `$!`
  - La dernière erreur, utilisée dans les détections d'erreurs.
- `$$`
  - Le numéro système du programme en cours : parfois utile car il change à chaque appel.

Retour sommaire

Variables spéciales (2)

- `$1, $2, ...`
  - Le contenu de la parenthèse numéro dans la dernière expression régulière.
- `$0`
  - Le nom du programme (à utiliser, cela vous évitera de modifier le programme s'il change de nom).

Retour sommaire

Tableaux spéciaux

- `@_`
  - Contient les paramètres passés à une procédure.
- `@ARGV`
  - Contient les paramètres passés au programme.
- `%ENV`
  - Tableau indicé contenant toutes les variables d'environnement.
- `@INC`
  - Tous les répertoires Perl contenant les “ librairies ”.

Retour sommaire

Variables globales, privées

- Variables globales
  - Les déclarations de sous-programmes et de formats sont globales. Ces déclarations globales peuvent être placées partout où l'on peut placer une instruction. Les déclarations sont mises régulièrement au début ou à la fin du programme, ou dans un autre fichier.
  - Une variable utilisée en un point du programme est accessible depuis n'importe où dans ce programme.

Retour sommaire

Variables globales/privées

Déclarations avec portée

- Comme les déclarations globales, les déclarations à portée lexicale ont un effet au moment de la compilation.
- Les déclaration à portée lexicale ont un effet depuis le point de leur déclaration jusqu'à la fin du bloc le plus interne les encadrant, d'où l'expression "à portée lexicale".

Retour sommaire

MY (1)

- Pour les variables privées à portée lexicale (créées par `my`), il faut donc s'assurer que la définition d'un sous-programme ou d'un format se trouve dans la même portée de bloc que le `my` si l'on désire accéder à ces variables.
- En Perl 4, l'instruction `my` n'est pas définie. On utilise l'instruction `local` pour définir une variable qui n'est pas présente dans le programme principal.

Retour sommaire

MY (2)

- `my` permet de créer des variables privées,
- `local` permet de créer des valeurs semi-privées de variables globales.
  - Un `my` ou un `local` déclare que les variables listées (dans le cas de `my`) ou les valeurs des variables globales listées (dans le cas de `local`) sont confinées au bloc, au sous-programme, à l'`eval` ou au fichier qui les encadre. Si plus d'une variable est listée, la liste doit être placée entre parenthèses. Tous les éléments listés doivent être des `lvalues` légales

Retour sommaire

MY (3)

- Pour `my`, les contraintes sont encore plus strictes: les éléments doivent être de simples variables scalaires, tableaux ou hachages, et rien d'autre.

Retour sommaire

Perl par l’exemple

Des programmes…

Exemple1 – fréquence des mots d'un texte

```
#!/usr/bin/perl
$/ = "";                    #unité de travail le paragraphe
$* = 1;                     #plusieurs lignes à la fois
while (<>){
      s/-\n//g;             #enlever les tirets
      tr/A-Z/a-z/;          #minusculiser
      @words = split (/\W*\s+\W*/,$_);
      foreach $word (@words) {
            $wordcount{$word}++;
      }
}
foreach $word (sort keys (%wordcount)) {
      printf "%20s,%d\n",$word,$wordcount{$word};
}
```

Exemple2 – fréquence des bigrammes d'un texte

```
while(<>) {

      s/^\s+//;
      @words = split(/\W*\s+\W*/,$_);

      for ($count=0;$count<=$#words-1;++$count) {
          $wordcount{$words[$count]. " " . $words[$count+1]}++;
      }
}


foreach $wordpair (sort keys(%wordcount)) {
      printf "%20s,%d\n",$wordpair,$wordcount{$wordpair};
}
```

Exemple 3 – fréquence des terminaisons (3 cars)

```
while(<>) {
 s/^\s+//;
 @words = split(/\s+/,$_);
 for ($count=0;$count<=$#words;++$count) {
     @chars = split(//,$words[$count]);
                  # on split sans séparateur
                  # le résultat est un tableau de caractères
     if ($#chars > 1) {
                  # on vérifie qu'il y ait au moins trois caractères

           $ending{$chars[$#chars-2] . " " .
                 $chars[$#chars-1] . " " .
           $chars[$#chars]}++;
     }
  }
}

foreach $end (sort keys(%ending)) {
      printf "%20s,%d\n",$end,$ending{$end};
}
```

Exemple 4 – POS stripping

Etant donné un texte étiquetté avec partie du discours (Parts-Of-Speech) (Exemple : `The/det boy/noun ate/verb ...`) enlever les partie du discours

```
while(<>) {
        s/^\s+//;
        @words = split(/\s+/,$_);
        for ($count=0;$count<=$#words;++$count) {
           $word = $words[$count];
           # mais il faut enlever l'étiquette
           $word =~ s/\/.*$//;
           # si une chaîne commence avec un slash, et elle comporte
           # une suite de caractères, la remplacer avec la chaîne
           # vide. Remarque: il faut mettre backslash avant la barre oblique
           # dans l'expression régulière
           print $word, " ";
        }
        print "\n";
}
```

Exemple 5 – mots stripping

Etant donné un texte étiquetté avec partie du discours (Parts-Of-Speech)

(exemple : `The/det boy/noun ate/verb ...`), enlever les mots et retourner les parties du discours

```
while(<>) {
  s/^\s+//;
  @words = split(/\s+/,$_);
  for ($count=0;$count<=$#words;++$count) {
       $word = $words[$count];

       # but word has tag on it
       $word =~ s/^.*\///;
       print $word,  " ";
  }
print "\n";
}
```

Exemple 6 – le mot le plus long dans un texte

```
while(<>) {
   s/^\s+//;
   @words = split(/\s+/,$_);
   for ($count=0;$count<=$#words;++$count) {
       @chars = split(//,$words[$count]);
       if ($#chars > $maxlength) {
          $maxlength = $#chars;
       }
   }
}

$maxlength++;

#il faut ajouter 1, car l'indice commence à 0

print $maxlength,''\n'';
```

Exemple 7 – trigrammes

```
#!/usr/local/bin/perl5
# Tallies trigrams in each line Parens are ignored
# Default input from STDIN Default output to STDOUT.
while (<>) {
      # remove parens and tokenize
      s/\(/ /g;    s/\)/ /g;
      @token = split;
      # split on whitespace
      # count trigrams
      for($i=0; $i < @token - 2; $i++) {
            $count{"@token[$i] @token[$i+1] @token[$i+2]"}++;
      };
};
# output
while (($key, $value) = each %count) {
        print "$value $key\n";
};
```

Exemple 8 – 4-grammes

```
#!/usr/local/bin/perl5
# Tallies fourgrams in each line Parens are ignored
# Default input from STDIN Default output to STDOUT.
while (<>) {
      # remove parens and tokenize
      s/\(/ /g;    s/\)/ /g;    @token = split;
      # count 4-grams
      for($i=0; $i < @token - 3; $i++) {
            $count{"@token[$i] @token[$i+1] @token[$i+2]
@token[$i+3]"}++;
      };
};
# output
while (($key, $value) = each %count) {
        print "$value $key\n";
};
```

Exemple 8 – unique

```
#!/usr/local/bin/perl5
# eliminates all but one occurrence of each line in a file, but
#     doesn't change the order otherwise

while (<>) {
    if ($seen{$_}) {
      next;
    } else {
      $seen{$_}++;
      print;
    };
};
```

Exemple 9 – blankwords

```
#!/usr/local/bin/perl5
# blanks out given words
# the words to remove should be in a file, one per line
# check for correct usage
if ($#ARGV < 0) {
      # pas de params
        print "usage:  blankwords <word list> [<base file>]\n";
      exit;
};
open(P, $ARGV[0]) || die "Couldn't open $ARGV[0]: $!\n";
$killwords = "\n";
while (<P>) {
        $killwords .= $_;
      # $var OP= $val equivaut à $var = $var OP $val
};
close(P);
shift;
# s'applique à @ARGV
```

Exemple 10 – blankwords

```
while (<>) {
    @token = split;             # split on whitespace
    @char = split("", $_);      # split on nullspace
    $offset = 0;
    for($i = 0; $i < @token; $i++) {
      $offset = index($_, $token[$i], $offset);
      ($pat = $token[$i]) =~ s/(\W)/\\$1/g;
      if ($killwords =~ /\n$pat\n/) {
          $size = length($token[$i]);
          foreach $ind ($offset .. $offset + $size - 1) {
            splice(@char, $ind, 1, " ");
          };
      };
    };
    print @char;
};
```

Exemple 11 – blankclasses

```
#!/usr/local/bin/perl5
# blanks out words of a given POS in a tagged text the POS to remove
# should be in a file, one per line words and tags are separate by ::
#check for correct usage
if ($#ARGV < 0) {
        print "usage:  blankclasses <POS list> [<base file>]\n";
        exit;
};
open(P, $ARGV[0]) || die "Couldn't open $ARGV[0]: $!\n";
$killclasses = "\n";
while (<P>) {
      $killclasses .= $_;
};
close(P);
shift;
```

Exemple 12 – blankclasses

```
while (<>) {
    s/^ *//;                # enlever les espace au début de ligne
    @token = split;           # split on whitespace
    for($i = 0; $i < @token; $i++) {
      #scalar context
       if ($token[$i] =~ /(.+)::(.+)/) {
          # word::tag and back-referencing
           $word = $1;
           $tag = $2;
           if ($killclasses =~ /\n$tag\n/) {
              $token[$i] = " " x length($word);
           };
       } else {
           die "Word or Tag missing\n";
       };
    };
    print join(' ', @token), "\n";
};
```

Perl en lignes de commandes

Perl en ligne de commande

- Exécution d'un script : -e *ligne de code*
  - -e est le paramètre principal pour l'écriture d'un uniligne.
  - Son argument est une ligne de code. Pour inclure plusieurs lignes de code, il suffit d'utiliser plusieurs fois -e (cela ne dispense pas d'utiliser les points-virgules).
    - [Note : Si `-e` est fourni, perl ne va pas chercher le nom du script à exécuter parmi les arguments en ligne de commande. Classiquement, la liste des options passées à l'interpréteur perl s'arrête avec la dernière option, ou avec l'option spéciale --. La liste des options est suivie par la liste des paramètres. Elle est accessible depuis l'interpréteur par la variable @ARGV]
  - Exemple :

```
$ perl -e 'print "Hello, world!\n"'
Hello, world!
```

Retour sommaire

Relativité des lignes…

- La notion de ligne étant toute relative, on peut réaliser des scripts assez complets qui tiennent sur une ligne !
  (Mais celle-ci pourra faire beaucoup plus de 80 caractères...)

```
$ perl -e 'print "Hello, world!\n" ;' -e 'print "Bonjour, monde
  !\n"'
```

est équivalent à :

```
$ perl -e 'print "Hello, world!\n" ; print "Bonjour, monde !\n"'
```

Retour sommaire

Précautions… (1)

- L'utilisation de l'option -e peut poser quelques problèmes mineurs, selon votre shell.
  - Sous Windows
    - Le fonctionnement des guillemets dans le `command.com` (Windows 95, 98) ou `cmd.exe` (Windows NT, 2000, XP) est différent de celui sous Unix.

```
C:\> perl -e 'print "Hello, world!\n"'
Can't find string terminator "'" anywhere before EOF at -e
  line 1.
```

    - Il faut donc faire quelques contorsions pour utiliser les guillemets attendus par le shell :

```
C:\> perl -e "print qq'Hello, world!\n'" Hello, world!
```

    - L'utilisation de la fonction `qq()` permet ici d'utiliser des apostrophes comme des guillemets (permettant ainsi l'interpolation de variables et l'utilisation de raccourcis comme `\n`).
    - En règle générale, si on ne sait pas comment réagit le shell, il est préférable d'utiliser `qq(), q() et qx()` au lieu de `""`, `''` et ``` `` ``` pour simplifier l'écriture d'unilignes.

Retour sommaire

Exemples sous Cygwin

Cygwin B20

```
bash-2.02$
bash-2.02$
bash-2.02$
bash-2.02$
bash-2.02$
bash-2.02$
bash-2.02$
bash-2.02$ perl -w -e 'print("Salut Larry\n");'
Can't find string terminator '"' anywhere before EOF at -e line 1.
bash-2.02$
bash-2.02$
bash-2.02$
bash-2.02$
bash-2.02$
bash-2.02$
bash-2.02$ perl -w -e "print('Salut Larry\n');"
Salut Larry\nbash-2.02$
bash-2.02$
bash-2.02$
bash-2.02$
bash-2.02$ perl -w -e "print(qq'Salut Larry\n');"
Salut Larry
bash-2.02$
bash-2.02$
bash-2.02$
bash-2.02$
bash-2.02$
bash-2.02$
```

Retour sommaire

Précautions… (2)

- **Sous Unix**
  - Sous la plupart des shells, les chaînes entre guillemets (" ou *double quotes*) subissent une interpolation des variables qui s'y trouvent. Considérons l'exemple suivant :

```
$ perl -e "$A = 12; print ++$A"
syntax error at -e line 1, near "=" Execution of -e aborted due to compilation errors.
```

  - En effet, Perl a reçu le code « `= 12; print ++` », qu'il n'est évidemment pas capable de compiler. Et si la variable `$A` était définie dans votre environnement, cela aurait donné des résultats encore différents.
  - Il faut alors s'engager courageusement sur le chemin difficile des mécanismes de citation du shell :

```
$ perl -e "\$A = 12; print ++\$A"
13
```

  - ou, plus simplement, choisir les guillemets adaptés au contexte d'utilisation :

```
$ perl -e '$A = 12; print ++$A'
13
```

Retour sommaire

Les filtres sur des fichiers : -n et -p

- Ces deux options sont probablement les plus utilisées pour l'écriture d'« unilignes » en Perl
- -n ajoute la boucle suivante autour de votre code :

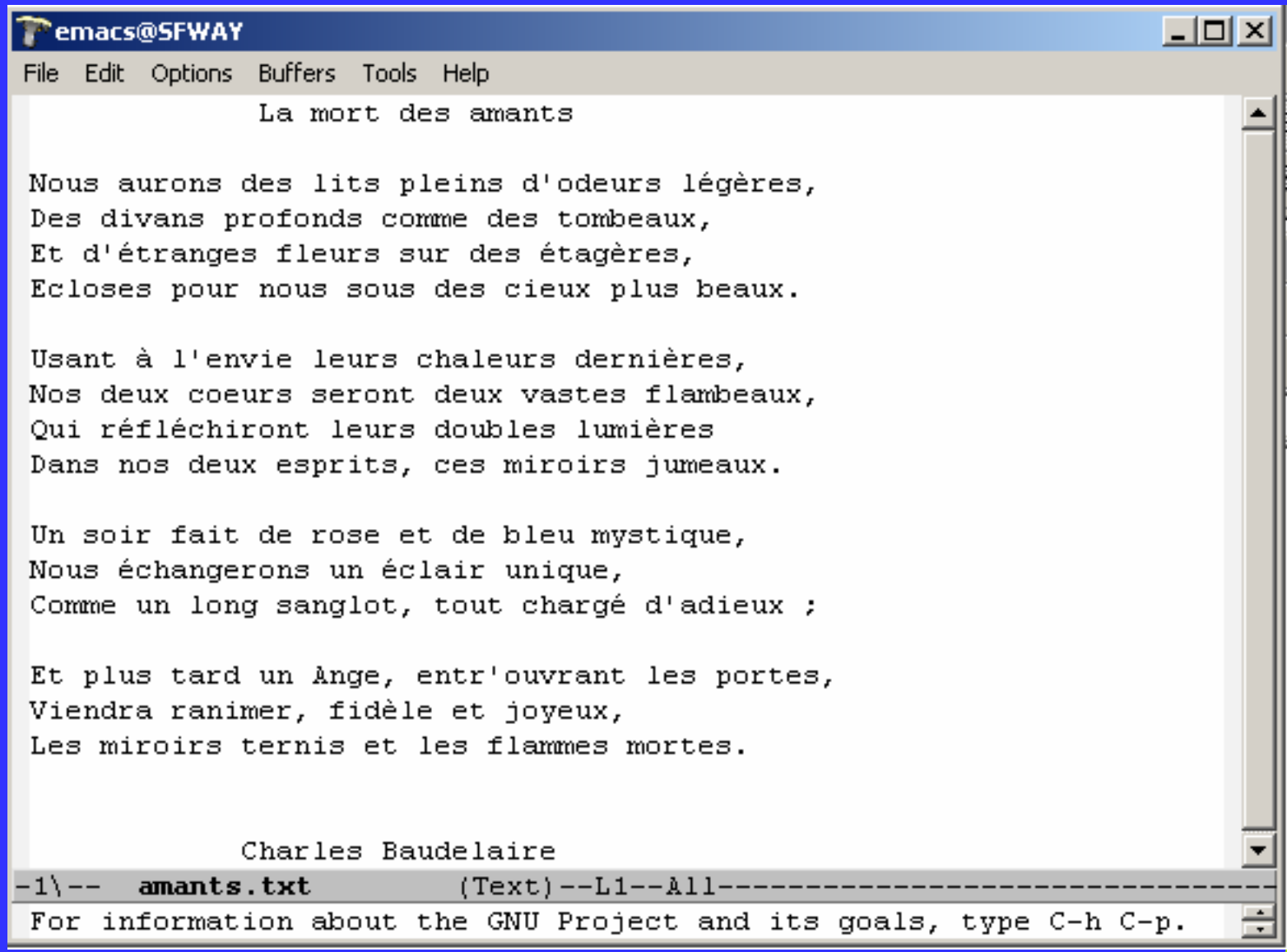
```
LINE: while (<>) {
        ... # votre programme ici
        }
```

- -p ajoute la boucle suivante, qui imprime automatiquement les lignes, autour de votre code :

```
LINE: while (<>) {
    ... # votre programme ici
    }
    continue {
    print or die "-p destination: $!\n";
    }
```

Retour sommaire

Fichier de travail

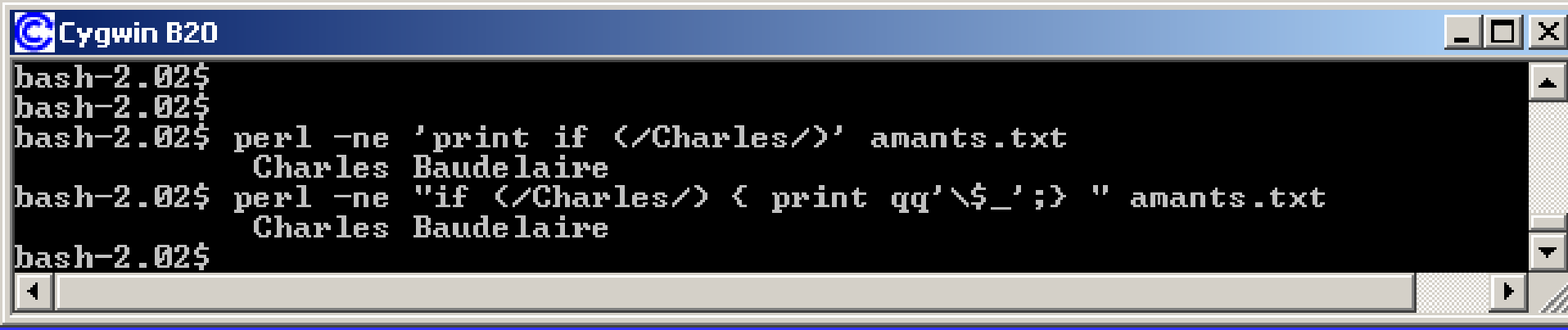


Retour sommaire

Exemple d'utilisation de –n (1)

- # egrep en « uniligne perl »

```
$ perl -ne 'print if /regex perl/' fichier
ou
$ perl -ne "if /regex perl/ { print qq'\$_'} " fichier
```



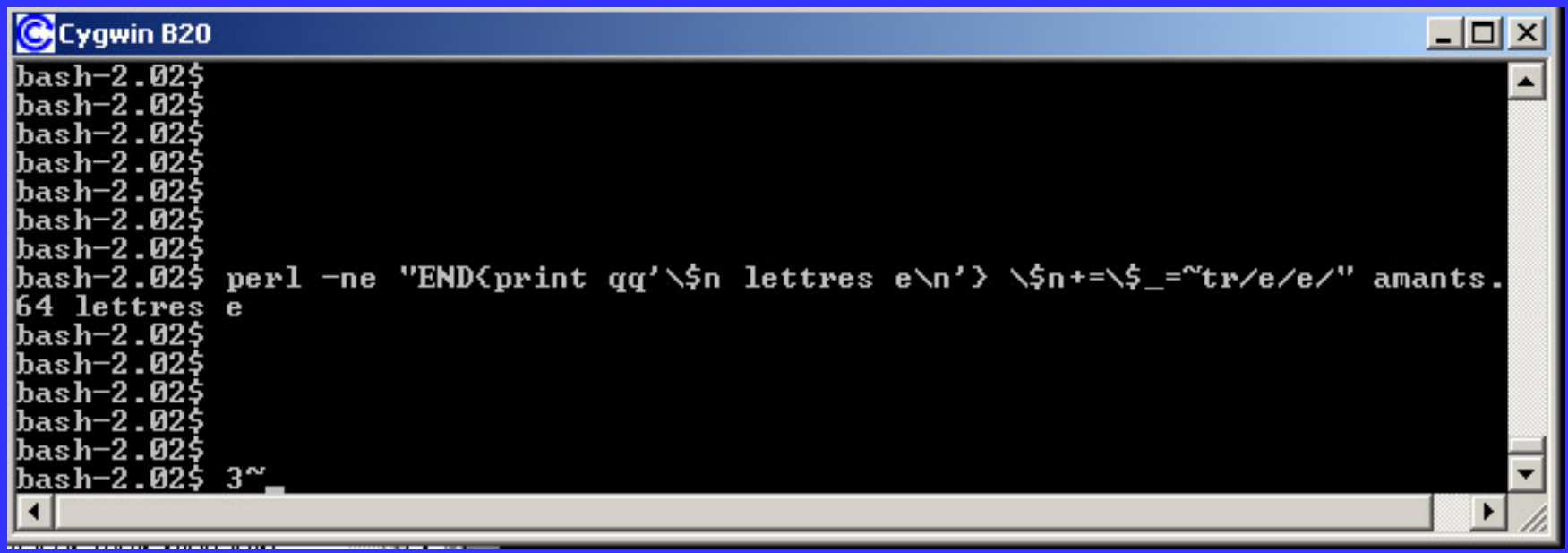
Note : *précautions prises ici pour écrire la commande*

Retour sommaire

Exemple d'utilisation de –n (2)

- # Compter le nombre de "e" dans un fichier

```
$ perl -ne "END{print qq'\$n lettres e\n'} \$n+=\$_=~tr/e/e/"
   fichier
```

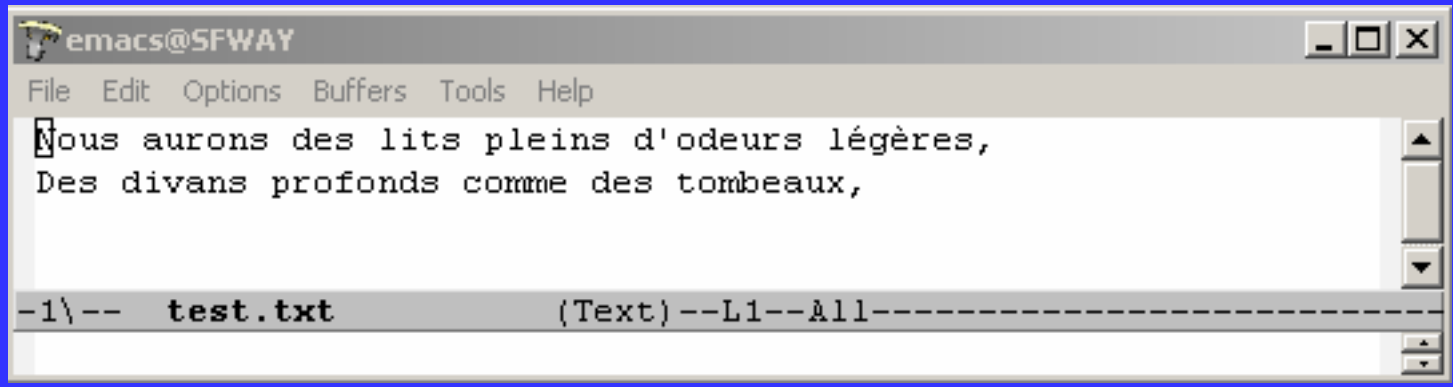


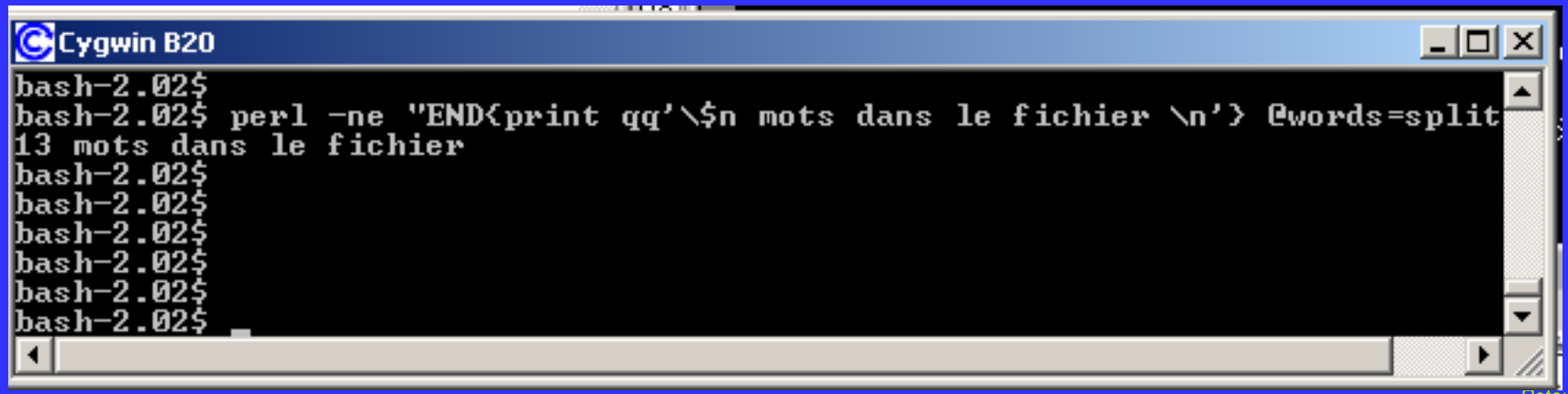
Retour sommaire

Exemple d'utilisation de –n (3)

- # Compter le nombre de mots dans un fichier

```
$ perl -ne "END{print qq'\$n mots dans le fichier\n'}
   @words=split(/\W*\s+\W*/,\$_); \$n+=(\$#words+1) " fichier
```



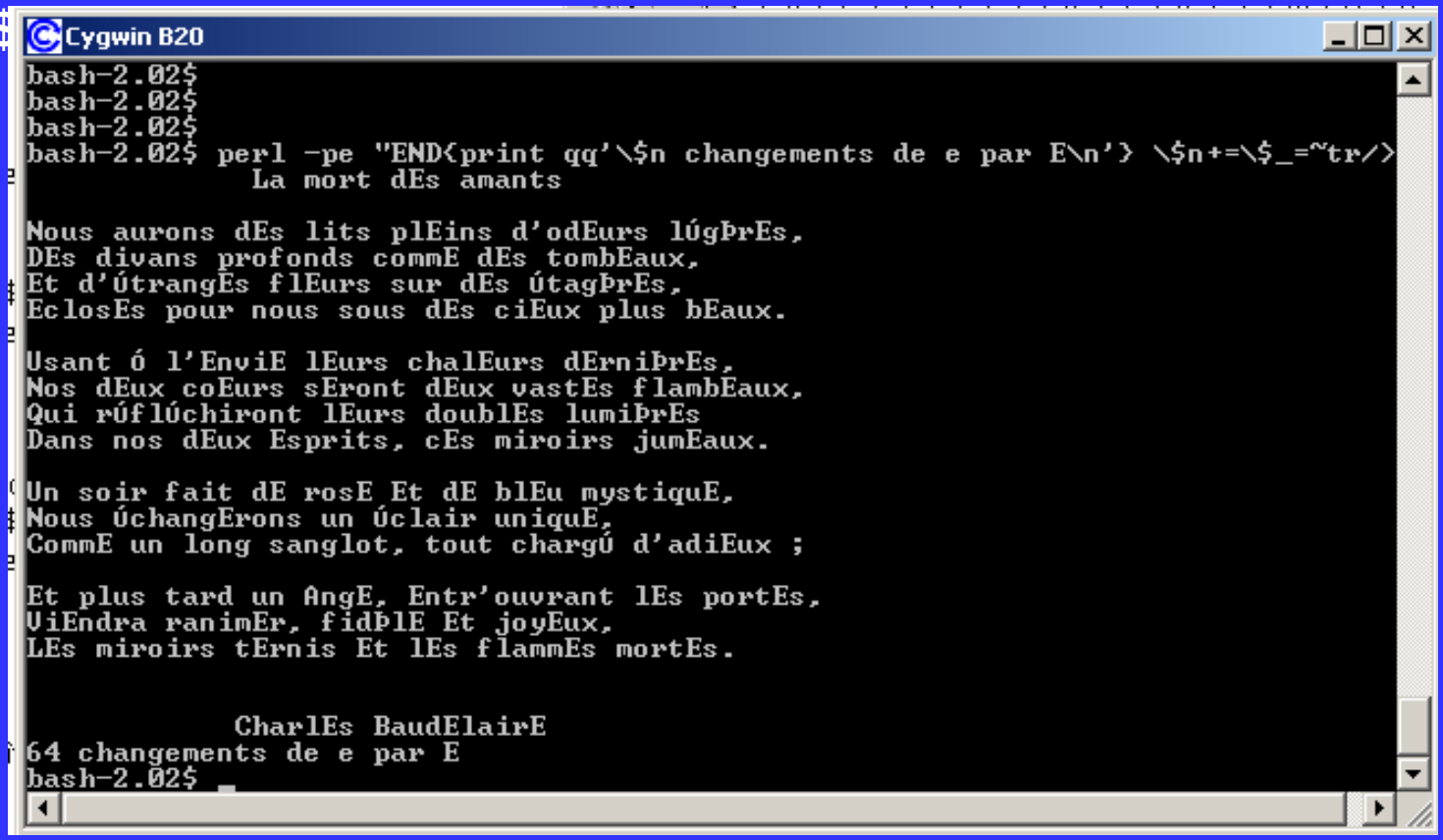


Retour sommaire

Exemple d'utilisation de –p

- # Remplacer tous les e par E dans un fichier

```
$ perl -ne "END{print qq'\$n changements de e par E\n'}
  \$
```



Retour sommaire

Boîte à outils en « *unilignes* »

- Filtres
  - Remplace « machin » par « bidule » (souvent utilisé avec l'option -i) :

```
perl -pe 's/\bmachin\b/bidule/g' fichier
```

  - Imprime les lignes communes à deux fichiers (posté par Randal Schwartz sur perlmonks) :

```
perl -ne 'print if ($seen{$_} .= @ARGV) =~ /10$/' fichier1 fichier2
```

  - Le même, pour trois fichiers (et pour vous aider à comprendre) :

```
perl -ne 'print if ($seen{$_} .= @ARGV) =~ /21+0$/' fichier1 fichier2
   fichier3
```

  - Supprime les lignes en doublon (attention, c'est plus fort qu'uniq) :

```
perl -ne 'print unless $doublon{$_}++' fichier
```

Retour sommaire

Boîte à outils (2) en « *unilignes* »

- just lines 15 to 17

```
perl -ne 'print if 15 .. 17'
```

- just lines NOT between line 10 and 20

```
perl -ne 'print unless 10 .. 20'
```

- lines between START and END

```
perl -ne 'print if /^START$/ .. /^END$/'
```

- lines NOT between START and END

```
perl -ne 'print unless /^START$/ .. /^END$/‘
```

- just lines 15 to 17, efficiently

```
perl -ne 'print if $. >= 15; exit if $. >= 17;'
```

Retour sommaire

Boîte à outils (3) en « *unilignes* »

- command-line that reverses the whole input by lines (printing each line in reverse order)

```
perl -e 'print reverse <>' file1 file2 file3 ....
```

- command-line that shows each line with its characters backwards

```
perl -nle 'print scalar reverse $_' file1 file2 file3 ....
```

- find palindromes in the /usr/dict/words dictionary file

```
perl -lne '$_ = lc $_; print if $_ eq reverse'
   /usr/dict/words
```

- command-line that reverses all the bytes in a file

```
perl -0777e 'print scalar reverse <>' f1 f2 f3 ...
```

- command-line that reverses each paragraph in the file but prints them in order

```
perl -00 -e 'print reverse <>' file1 file2 file3 ....
```

Retour sommaire

« L'édition sur place » : -i[*extension*] (1)

- Supposons que vous traduisez un document en anglais, et que vous voulez transformer tous les *foo* en *toto* et tous les *bar* en *titi* dans les exemples. Une fois que vous avez la nouvelle version, l'ancienne n'a plus d'intérêt pour vous.

```
$ perl -pe 's/\bfoo\b/toto/g;s/\bbar\b/titi/g' monfichier > monfichier.new
$ mv -f monfichier.new monfichier
=> C'était bien la peine de faire un uniligne...
```

- L'option `-i` vous permet de faire de l'édition *sur place*, c'est-à-dire de modifier directement le fichier que vous êtes en train de traiter. Ou plus exactement, le fichier traité par la construction <> (qui est justement la construction utilisée implicitement par `-p` et `-n`).

Retour sommaire

« L'édition sur place » : -i[*extension*] (2)


```
$ perl -i -pe 's/\bfoo\b/toto/g;s/\bbar\b/titi/g' monfichier
```

- En réalité, Perl renomme le fichier original et redirige la sortie du script vers un fichier portant le même nom que l'original. Le fichier renommé peut être conservé comme fichier de sauvegarde, comme nous le voyons au paragraphe suivant.

- Si vous avez peur de vous tromper, vous pouvez fournir une extension à l'option `-i`, qui sera utilisée pour créer un fichier de sauvegarde identique au fichier traité original. L'extension est ajoutée à la fin du nom du fichier traité :

```
$ ls monfichier
$ perl -i.bak -pe 's/\bfoo\b/toto/g;s/\bbar\b/titi/g' monfichier
$ ls monfichier monfichier.bak
```

Retour sommaire

« L'édition sur place » : -i[*extension*] (3)

- En fait, l'option `-i` vous permet de faire plus que d'ajouter une extension à l'ancienne version du fichier. Si l'extension contient un ou plusieurs caractères *, chaque * est remplacé par le nom du fichier courant.

- Cela vous permet d'ajouter un préfixe aux fichiers traités :

```
$ perl -i 'orig_*' -pe 's/\bfoo\b/toto/g;s/\bbar\b/titi/g'
   fichier1 fichier2
```

Ou de sauvegarder les originaux dans un répertoire :

```
$ perl -i 'orig/*.bak' -pe 's/\bfoo\b/toto/g;s/\bbar\b/titi/g'
   fichier1 fichier2
```

Retour sommaire

« L'édition sur place » : -i[*extension*] (4)

- Les séries qui suivent présentent des « unilignes » équivalents :

```
$ perl -pi -e 's/\bfoo\b/toto/g;s/\bbar\b/titi/g'
  fichier1 fichier2
$ perl -pi '*' -e 's/\bfoo\b/toto/g;s/\bbar\b/titi/g'
  fichier1 fichier2
# Copie de sauvegarde dans fichier1.bak
$ perl -pi.bak -e 's/\bfoo\b/toto/g;s/\bbar\b/titi/g'
  fichier1 fichier2
$ perl -pi '*.bak' -e
  's/\bfoo\b/toto/g;s/\bbar\b/titi/g' fichier1
  fichier2
```

Retour sommaire

« L'édition sur place » : -i[*extension*] (5)

- Quelques remarques supplémentaires :
- Perl fait la copie de sauvegarde même si rien n'a été modifié dans le fichier original.
- `-i` ne peut pas être utilisé pour créer un répertoire ou supprimer l'extension des fichiers.
- `-i` ne fait pas non plus l'expansion du ~ dans le nom de fichier (cela est fait par le shell). Ceci est une Bonne Chose (TM), puisque certains utilisent le ~ pour leurs fichiers de sauvegarde :

```
$ perl -pi~ -e 's/\bfoo\b/toto/g;s/\bbar\b/titi/g' fichier1 fichier2
```

- Enfin, -i ne modifie pas l'exécution si aucun fichier n'est donné sur la ligne de commande. Le traitement se fait comme d'habitude de `STDIN` vers `STDOUT` (fonctionnement comme un filtre).

Retour sommaire

Boîte à outils (5) en « *unilignes* »

- in-place edit of *.c files changing all foo to bar

```
perl -p -i.bak -e 's/\bfoo\b/bar/g' *.c
```

- delete first 10 lines

```
perl -i.old -ne 'print unless 1 .. 10' foo.txt
```

- change all the isolated oldvar occurrences to newvar

```
perl -i.old -pe 's{\boldvar\b}{newvar}g' *.[chy]
```

- increment all numbers found in these files

```
perl -i.tiny -pe 's/(\d+)/1+$1/ge' file1 file2 ....
```

- delete all but lines between START and END

```
perl -i.old -ne 'print unless /^START$/ .. /^END$/'
  foo.txt
```

- binary edit (careful!)

```
perl -i.bak -pe 's/Mozilla/Slopoke/g'
  /usr/local/bin/netscape
```

Retour sommaire

Perl et les objets

Perl et les références

- Une des limitations de *perl4* venait du fait qu'il était impossible de créer et de manipuler des structures de données plus évoluées que de simples tableaux sans passer par des astuces plus ou moins efficaces.
- Une des grandes innovations de *perl5* a été l'introduction des références, qui permettent de travailler sur des structures de données plus complexes.

Retour sommaire

Déclaration syntaxe

- Soit `@liste` un tableau. Alors on peut obtenir sa référence avec l'expression suivante :

```
$ref =  @liste
```

- Si on exécute la ligne print $ref; on obtient quelque chose du type :

```
ARRAY(0x91a9c)
```

- Il y a deux manières d'utiliser ensuite cette référence. On peut la déréférencer pour la manipuler comme un tableau normal :

```
@new = @$ref;
print $new[0];
```

- ou bien on peut accéder directement aux données en remplaçant le nom du tableau par la référence :

```
print $$ref[0];
# ou bien
print $ref->[0];
```

- On peut donc ainsi construire facilement des tableaux à plusieurs dimensions :

```
print $table[0]->[2]->[4];
```

- Les flèches étant optionnelles entre les crochets ou les accolades, on peut également l'écrire :

```
print $table[0][2][4];
```

- Ces exemples s'appliquent également aux références sur des tableaux associatifs ou des scalaires.

Retour sommaire

Création de références anonymes (1)

- Voilà par exemple comment créer un tableau de tableaux :

```
@liste = ();
for $i ('A'..'z')
  {
      push(@liste, [ $i, ord($i) ]);
  }
print @{$liste[0]};
# -> affichage de ('A', 65) sous la forme A65
print ${$liste[0]}[0];
# -> affichage de 'A' sous la forme A
# ou bien
@paire = @{$liste[0]};
print $paire[0];
# -> affichage de 'A' sous la forme A
```

- Ceci va créer un tableau à deux dimensions (ou plutôt un tableau de tableaux).
- La notation [ ] renvoie une référence sur un tableau composé des éléments que les crochets renferment

Retour sommaire

Création de références anonymes (2)

- De la même manière, la notation {} renverra une référence sur un tableau associatif composé des éléments encadrés par les accolades :

```
@liste = ();
for $i ('A'..'z')
{
    push(@liste, { "caractere" => $i,
                   "valeur"    => ord($i) } );
}

print %{$liste[0]};
# -> affichage du tableau associatif ("caractere" => "A",
#                                     "valeur" => 65)
#    sous la forme caractereAvaleur65
print ${$liste[0]}{"caractere"};
# -> "A"
# ou bien
%paire = %{$liste[0]};
print $paire{"caractere"};
# -> "A"
```

On peut ainsi imbriquer plusieurs niveaux de références pour créer des structures de données complexes (mais attention à la mémoire utilisée !).

Retour sommaire

Référence sur les fonctions

- On peut de la même manière que pour les tableaux créer des références sur des fonctions de deux façons différentes.
- À partir d'une fonction déjà existante :

```
sub fonc {  ...; }
$ref = &fonc;
```

- ou bien en créant une référence sur une fonction anonyme :

```
$ref = sub { print "hello world !#3021#>n";};
```

- On peut ensuite appeler directement : `&$ref;`

Retour sommaire

Créer des Références

- En mettant "\" devant le nom d'une variable, on créé une référence sur cette variable

```
$aref = \@array; # $aref now holds a reference to @array
$href = \%hash; # $href now holds a reference to %hash
```

- On peut ensuite manipuler la référence comme n'importe quelle variable :

```
$xy = $aref; # $xy now holds a reference to @array
$p[3] = $href; # $p[3] now holds a reference to %hash
$z = $p[3]; # $z now holds a reference to %hash
```

- Remarque :

```
# Ecrire ceci (référence anonyme)
$aref = [ 1, 2, 3 ];
# est identique à ceci :
@array = (1, 2, 3);
$aref = \@array;
```

Retour sommaire

Les objets

- L'introduction des références a permis d'introduire également les concepts de la programmation orientée objets.
- Pour obtenir une documentation plus complète, se référer aux pages de manuel `perlobj` et `perlbot`

- Larry Wall
  Perl reference pages
  Les pages de manuel constituent une référence très bien structurée et facile à consulter. La dernière version se trouve sur les sites CPAN
- Documentation Perl en Français récupéré sur le site PerlGratuit.com : *perlobj*

Retour sommaire

Les bases

- Un objet est simplement une *référence* qui sait à quelle *classe* elle appartient (voir la commande `bless`).
- Une *classe* est un *paquetage* qui fournit des *méthodes* pour travailler sur ces *références*.
- Une *méthode* est une *fonction* qui prend comme premier argument une *référence* sur un *objet* (ou bien un nom de *paquetage*).

Retour sommaire

Un Objet en Perl

- Un objet possède une structure qui lui est propre
  - Cette structure est en fait une collection d'attributs (le type de ces attributs n'est pas prédéfini, tableau de scalaires ou tableau associatif)
- On associe à un objet des données et des méthodes
  - Un objet est un référence
  - une méthode est une procédure
  - une classe d 'objets est un package
- Lecture Poly PERL : "Perl et les objets"

Retour sommaire

Un exemple complet (1)

- Supposons que la classe `Window` soit définie. Alors on crée un objet appartenant à cette classe en appelant son constructeur :

```
$fenetre = new Window "Une fenetre";
# ou dans un autre style
$fenetre = Window->new("Une fenetre");
```

- Si Window a une méthode `expose` définie, on peut l'appeler ainsi :

```
expose $fenetre;
# ou bien
$fenetre->expose;
```

Retour sommaire

Un exemple complet (1)

- Voici un exemple de déclaration de la classe `Window` :

```
package Window;

# La methode de creation. Elle est appelee avec le nom de la
# classe (ie du paquetage) comme premier parametre. On peut passer
# d'autres parametres par la suite

   sub new {
  # On recupere les arguments
  my($classe, $parametre) = @_;

  # L'objet qui sera retourne est ici (et c'est generalement le cas)
  # une reference sur un tableau associatif. Les variables
  # d'instance de l'objet seront donc les valeurs de ce tableau.

     my $self = {};

  # On signale a $self qu'il depend du paquetage Window
  # (ie que sa classe est Window)

      bless $self;

  # Diverses initalisations
  $self->initialize($parametre);

  # On retourne $self
  return $self;
}
```

Retour sommaire

Un exemple complet (2)


```
  # Methode d'initalisation.
  # Le premier parametre est l'objet lui-meme.
  sub initialize {
    my($self, $parametre) = @_;

    $self->{'nom'} = $parametre || "Anonyme";
  }

  # Autre methode.
  sub expose {
    my $self = shift;

    print "La fenetre ", $self->{'parametre'},
          " a recu un evenement expose.\n";
  }
1;
```

Retour sommaire

Package/Paquetage mode d ’emploi

- En pratique
  - on construit un fichier Perl avec l ’extension pm qui contient la classe d ’objet que l ’on veut définir (`maclasse.pm`)
  - Méthodes particulières :
    - `new` : constructeur d ’objet
    - `destroy` : destructeur d ’objet
    - `bless` : rendre l’objet visible de l ’extérieur

Retour sommaire

Exemple: maclasse.pl

```
#!/bin/perl
package maclasse;    # Le nom que l'on donnera à l'objet

sub new {       # constructeur, Méthode appelée à la création
  my ($classe, ...) = @_;# La classe est toujours le 1er paramètre
  ...
  return bless référence;
  # on ne retourne pas une variable, mais sa référence,cf doc Perl
}

sub DESTROY {   # destructeur, appelée à la destruction
     my ($classe) = @_;
     ...
}

sub methode1 {  # On définit ainsi toutes les méthodes
     my ($classe, ...) = @_;
     ...
}
```

La classe de l'objet étant définie, on peut l'utiliser maintenant dans un programme Perl de la manière suivante :

```
use ma-classe;
$mon_objet = new maclasse(paramètres); # Appel du constructeur
$mon_objet->methode1(paramètres);   # Appel d'une méthode

exit;      # fin du programme, appel du destructeur de l'objet
```

Source : http://www.med.univ-rennes1.fr/~poulique/cours/perl/

Programmation Objets : créer des objets (1)

- Créer un objet

```
my $obj= Class->new(@args);
my $obj= new Class(@args);
```

- Exemple

```
my $dom= new XML::DOM::Parser;
my $elt= new XML::Twig::Elt( "p", "text of para
   $para_nb");
```

Retour sommaire

Programmation Objets : créer des objets (2)

- Utiliser un objet (et les méthodes associées)

```
$obj->method(@args);

my $doc = $dom->parsefile ("REC-xml-19980210.xml");
$elt->print( \*FILE);
```

Retour sommaire

Second exemple : un objet patient

- Objet « patient » défini par son nom et l'unité médicale dans laquelle il se trouve
- Deux méthodes seront attribuées au patient :
  - `transfert` : Transfert vers une nouvelle unité médicale
  - `affiche` : Affiche le nom du patient ainsi que l'unité médicale dans laquelle il se trouve

(cf http://www.med.univ-rennes1.fr/~poulique/cours/perl)

Retour sommaire

```
#!/bin/perl
package patient;                    # Déclaration d'un package

# Un nouveau patient consiste à lui donner un nom
# et éventuellement une unité
sub new {       # Constructeur
     my ($class, $nom, $unite) = @_;
     my $patient = {};

     $patient->{'nom'} = $nom;
     if ($unite) { # Si unité définie
          $patient->{'unite'} = $unite;
     } else {
          $patient->{'unite'} = 'Non définie';
     }
     return bless $patient;
}

sub transfert { # Transfert du patient vers nouvelle unité
     my ($patient, $nvunite) = @_;

     $patient->{'unite'} = $nvunite;
}

sub affiche { # Affichage de la situation du patient
     my ($patient) = @_;

     print "Le patient $patient->{'nom'} est dans l'unité
$patient->{'unite'}\n";
}

sub DESTROY {
     my ($patient) = @_;

     print "Le patient $patient->{'nom'} est parti !\n";
}
1; # Une classe d'objet se termine toujours par 1;
```

(cf http://www.med.univ-rennes1.fr/~poulique/cours/perl)

On peut maintenant utiliser cette classe d'objet dans un programme Perl :

```
#!/bin/perl
use sih; # Directive pour dire qu'on utilise le package SIH

# Déclaration de deux nouveaux patients
$patient1 = new patient('Dupont Pierre');
$patient2 = new patient('Durand Albert', 'urgences');

$patient1->affiche;       # Appel d'une méthode pour patient
$patient2->affiche;       # (affichage de sa situation)

$patient1->transfert('cardio');     # transfert vers nelle unite
$patient2->transfert('pneumo');

$patient1->affiche; # Affichage de la situation des 2 patients
$patient2->affiche;
# fin du programme, appel des destructeurs
```

(cf http://www.med.univ-rennes1.fr/~poulique/cours/perl)

Résultat du programme


```
Le patient Dupont Pierre est dans l'unité Non définie
Le patient Durand Albert est dans l'unité urgences
Le patient Dupont Pierre est dans l'unité cardio
Le patient Durand Albert est dans l'unité pneumo
Le patient Durand Albert est parti !
Le patient Dupont Pierre est parti !
```

Retour sommaire

Objets et héritage (1)

- Les objets « *voiture MonoMotorisation* » et « *voiture à BiMotorisation* » sont tous 2 des objets de type « *voiture* »
  - Il est préférable de définir les propriétés communes dans l'objet « *voiture* » puis de différencier les 2 objets-fils uniquement sur ce qui les distingue
  - En pratique, on créé un module générique « *voiture* » puis on créé 2 modules « *Voiture MonoMotorisation* » et «Voiture à BiMotorisation » et dans chacun de ces modules on va utiliser le tableau nommé @ISA dans lequel on peut indiquer des noms de modules pour lesquels on peut utiliser les fonctions qui y sont définies

Retour sommaire

Exemple : héritage (1)

```
package voiture;
sub demarrer {
}
…
package voitureMonoMotorisation;
@ISA="voiture";
…
package voitureBiMotorisation;
@ISA="voiture";
…
```

- De cette manière la méthode « *demarrer* » définie sur la classe « *voiture* » sera accessible par héritage sur les 2 types d'objets disponibles à partir des 2 sous-classes

Retour sommaire

Exemple : héritage (2)

- Rien n'empêche cependant de redéfinir la méthode commune au sein des sous-classes :
  - si on redéfinie la méthode « demarrer » au sein du package **voitureMonoMotorisation**, alors c'est cette méthode qui s 'appliquera sur les objets de cette classe (*i.e.* surcharger une fonction)

```
package voiture;
sub demarrer {
}
…
package voitureMonoMotorisation;
@ISA="voiture";
sub demarrer {
…
$obj->SUPER::demarrer();
}
…
```

- SUPER est une référence qui correspond à la classe mère de la classe actuelle (i.e. la classe « voiture »)

Retour sommaire

Programmation Objets : exemples et exercices

- Un exemple : code patient *cf supra*
  - répertoire TP/Programmes pour récupérer le code patient

- Travail à faire :
  - Tester le code fourni dans le document de travail
    - *Création d'un «package» word.pm*
    - *On complètera ce modèle par des éléments supplémentaires*
    - *Créer un package « Personne » : gestion d ’un carnet d ’adresses*
- Me remettre sur papier une version commentée du code construit

Retour sommaire

D’autres exemples

- **A voir dans le poly…**
- **Construire des objets :**
  - ***Cf* exercices à faire.**

Retour sommaire

Travail personnel/binôme

- **Lecture tutorial PERL OO**
  - **www.perl.com/CPAN/doc/FMTEYEWTK/perltoot.html**
  - **Copyright 1996 Tom Christiansen. All Rights Reserved**
- **Récupérer le code contenu dans le tutorial**
  - **tester et modifier le code présenté dans ce tutorial**

Retour sommaire

Perl et TK

De l’existence de WIDGET(s)

Le terme *widget*, il proviendrait de la contraction anglaise de *window gadget* selon le « Grand dictionnaire terminologique de l'office québécois de la langue française ». Cela se traduit tout simplement par « gadget logiciel » et donc naturellement, ceci n'est jamais utilisé.

Les widgets de Tk, au sens de Perl, sont des méthodes. En revanche, en Perl/Tk, « widget » désigne la classe et « widgets » les classes prédéfinies (`Button`, `Label`, `Frame`, etc.). Les widgets, de par leur construction, ont des options communes et d'autres, bien sûr, qui leur sont propres.

Retour sommaire

Un premier WIDGET : une étiquette


```
use Tk;
$mw=new MainWindow();
$l = $mw->Label(-text=>"Hello Tk!");
$l->pack();
MainLoop();
```

WIDGET LABEL

Programme perl

Retour sommaire

Un second WIDGET : un bouton


```
use Tk;
$mw=new MainWindow();
$b = $mw->Button(
      -text=>"Goodbye, Tk",
      -command=>sub{exit});
$b->pack();
MainLoop();
```

WIDGET Button

Programme perl

Retour sommaire

Huh ? C’est quoi TK

- “Label” et “Button” sont des widgets (parmi d’autres)
- Tk c’est une collection de widgets

- Apprendre TK c’est :
  - Apprendre à découvrir de nombreux widgets.
  - Apprendre à connaître (petit à petit) toutes les fonctionnalités et options associées à tous ces widgets.
  - Apprendre à construire de “jolis” widgets
  - Apprendre à faire coexister plusieurs widgets

Retour sommaire

Entry (1)


```
use Tk;
$mw=new MainWindow();
$l=$mw->Label(-text=>"Enter Your Name:");
$l->pack;
$e = $mw->Entry()->pack;
MainLoop();
```

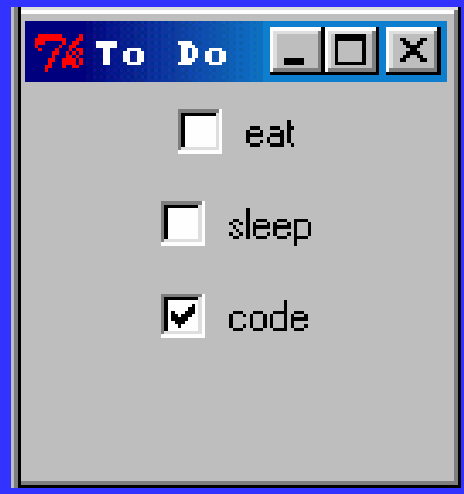
Retour sommaire

Entry (2)


```
use Tk;
$mw=new MainWindow();
$mw->Entry(-textvariable=>\$t)->pack;
$mw->Button(
   -text=>"print",
   -command=>sub{print "$t\n"})
   ->pack;
MainLoop();
```

Retour sommaire

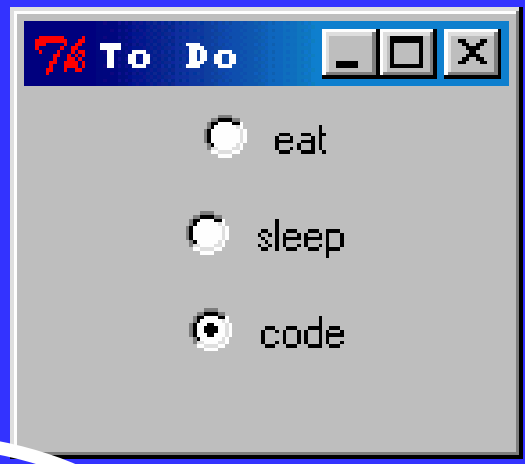
Checkbuttons




```
use Tk;
$mw=new MainWindow(-title=>"To Do");
$mw->Checkbutton(-text=>"eat",-variable=>\$v1)->pack;
$mw->Checkbutton(-text=>"sleep",-variable=>\$v2)->pack;
$mw->Checkbutton(-text=>"code",-variable=>\$v3)->pack;
MainLoop();
```

Retour sommaire

Radiobuttons

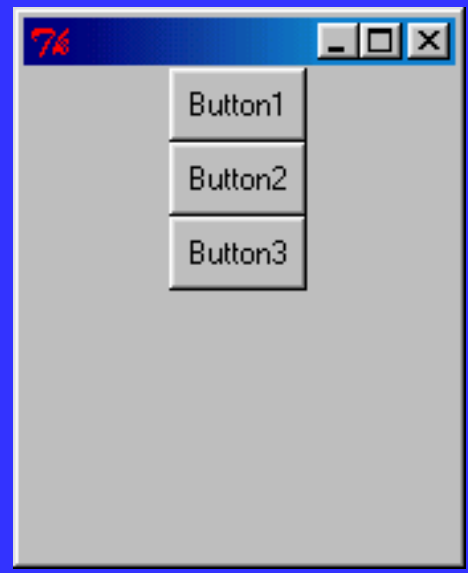



```
use Tk;
$mw=new MainWindow(-title=>"To Do");

$r1=$mw->Radiobutton(-text=>"eat",-variable=>\$v,-value=>"eat");
$r2=$mw->Radiobutton(-text=>"sleep",-variable=>\$v,-value=>"slp");
$r3=$mw->Radiobutton(-text=>"code",-variable=>\$v,-value=>"code");

$_->pack() foreach ($r1,$r2,$r3);
MainLoop();
```

Retour sommaire

Gestion de l'affichage avec `pack` (1)

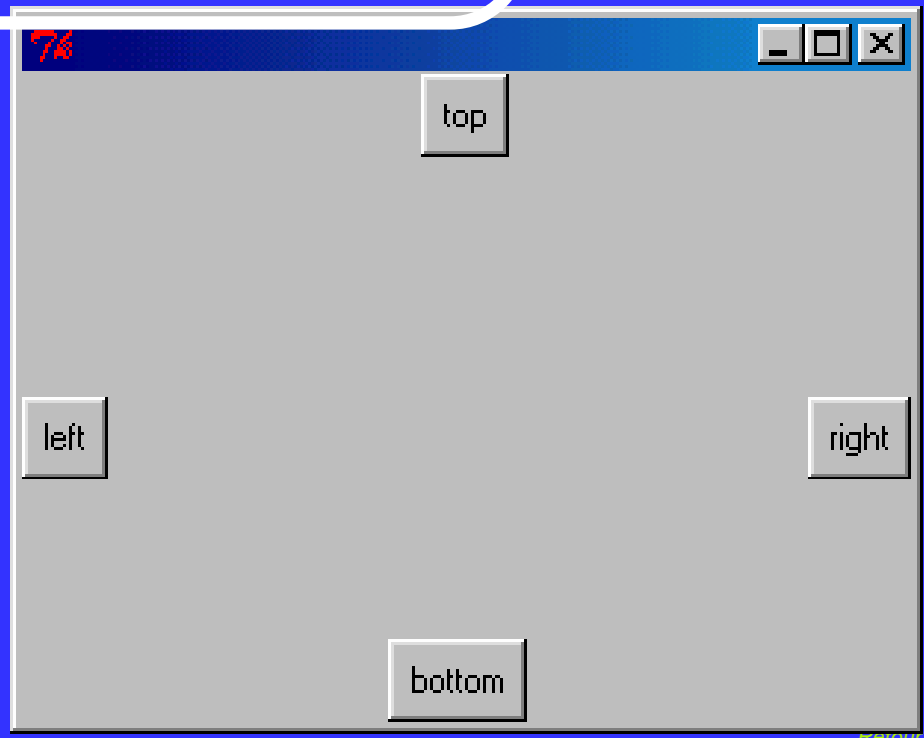
```
use Tk;
$mw=new MainWindow(-title=>"");

$mw->Button(-text=>"Button1")->pack;
$mw->Button(-text=>"Button2")->pack;
$mw->Button(-text=>"Button3")->pack;

MainLoop();
```



Retour sommaire

Gestion de l'affichage avec `pack` (2)

```
use Tk;
$mw=new MainWindow(-title=>"");
$mw->Button(-text=>"top")->pack(-side=>"top");
$mw->Button(-text=>"left")->pack(-side=>"left");
$mw->Button(-text=>"right")->pack(-side=>"right");
$mw->Button(-text=>"bottom")->pack(-side=>"bottom");
MainLoop();
```



Retour sommaire

Gestion de l'affichage avec `pack` (3)

Le gestionnaire de géométrie est certainement un élément essentiel qui simplifie considérablement la vie du programmeur. En effet, il prend en charge les widgets composant de l'application, adapte automatiquement les dimensions et gère le tout de façon dynamique.

Le principe de fonctionnement est simple, vous partez d'un côté (option `-side=>[ 'left' | 'right' | 'top' | 'bottom' ]`) qui, par défaut a la valeur `top`. Les objets sont rangés selon l'ordre de leur déclaration, en fonction de l'espace disponible, en réservant toute la surface perpendiculaire à la direction choisie d'empilage. Si, de plus, l'option `-fill` est précisée alors le widget va remplir l'espace selon la valeur choisie : `x|y|none|both`.

L'option `-anchor` réglera l'ancrage du widget selon la valeur suivante `n|s|e|w|center` (pour *north*, *south*, *east*, *west*). Les combinaisons `sw`, `se`, `ne`, `nw` sont autorisées.

Ainsi, pour bien mettre en relief la différence entre les options '-side' et '-anchor', la première correspond au sens d'empilage tandis que l'autre se réfère au placement du widget dans la surface réservée à cet effet. Cette zone est différente de celle occupée par le widget, mais peut offrir un recouvrement identique avec les options '-expand' et '-fill'.

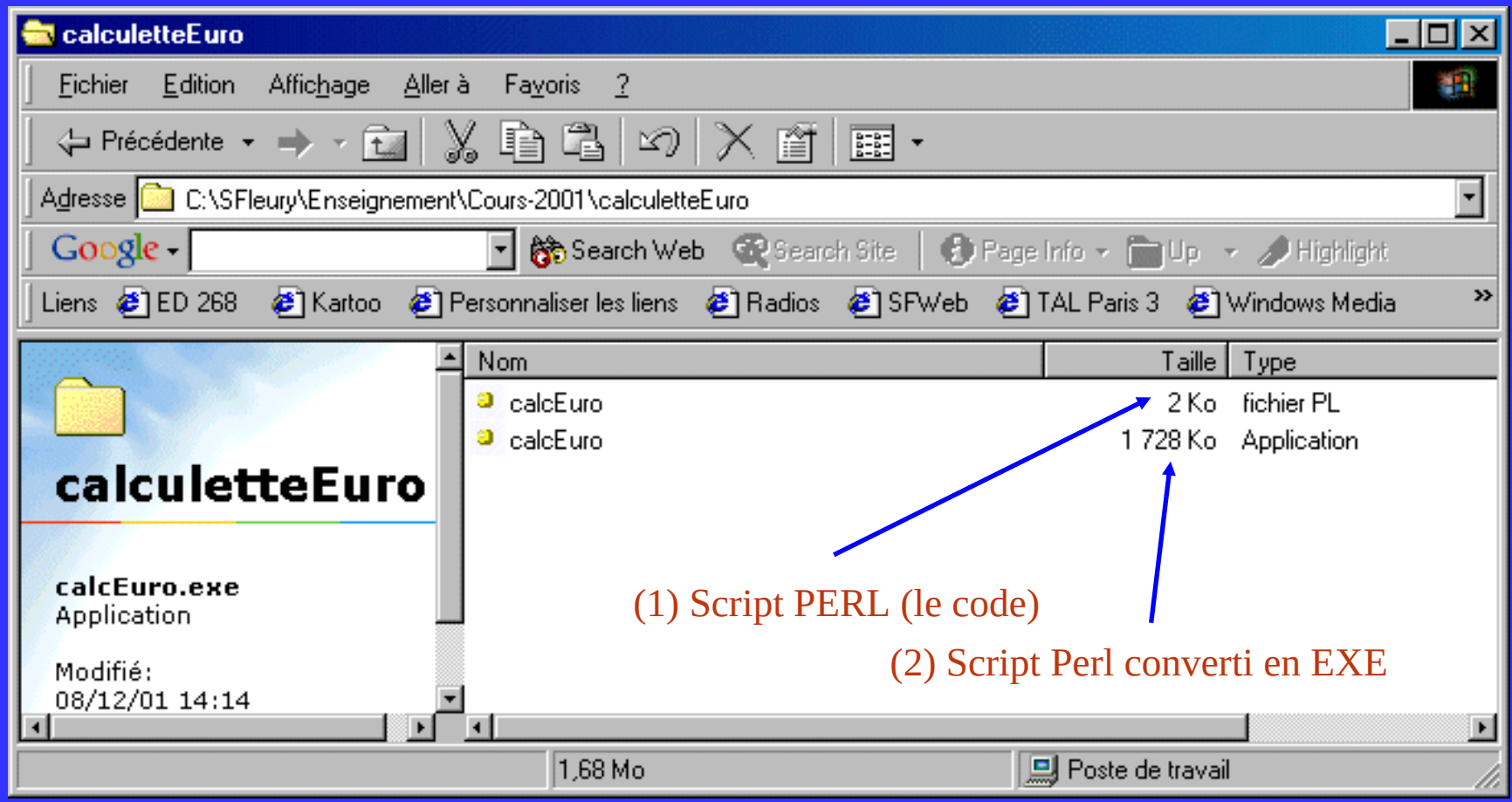
Enfin, l'option `-expand=>y` permet de répartir l'espace restant entre tous les widgets qui partage une donnée de même « nature » (`bottom` et `top`) d'une part et (`left`, `right`) d'autre part. En effet, par défaut, c'est le dernier widget qui récupère toute la surface disponible pour son propre compte.

On comprendra facilement que cette simplicité rend parfois impossible le rangement de widgets dans un ordre donné. C'est là où le widget `Frame` vient à notre secours, en introduisant des cadres intermédiaires pour y placer les éléments. Par exemple, ranger trois « boutons à gauche », l'un en dessous de l'autre, se résout simplement en utilisant un « *frame* » (cadre en français) avec l'option `-side=>'left'` et les boutons avec l'option `-side=>'top'` ou `'bottom'`.

Retour sommaire

Premier programme complet avec Perl Tk

- Une calculette EURO/Franc
- Deux versions disponibles (1) et (2)



Retour sommaire

Lancement du programme

- Solution 1 :
  - pour activer le script `perl calceuro.pl`
    - Démarrer MSDOS ou Cygwin-Beta20
    - Placer vous dans le répertoire où se trouve le programme
    - Lancer la commande :

      ```
      perl calceuro.pl
      ```
- Solution 2 :
  - pour activer le programme `calceuro.exe`
    - Double-clic

Retour sommaire

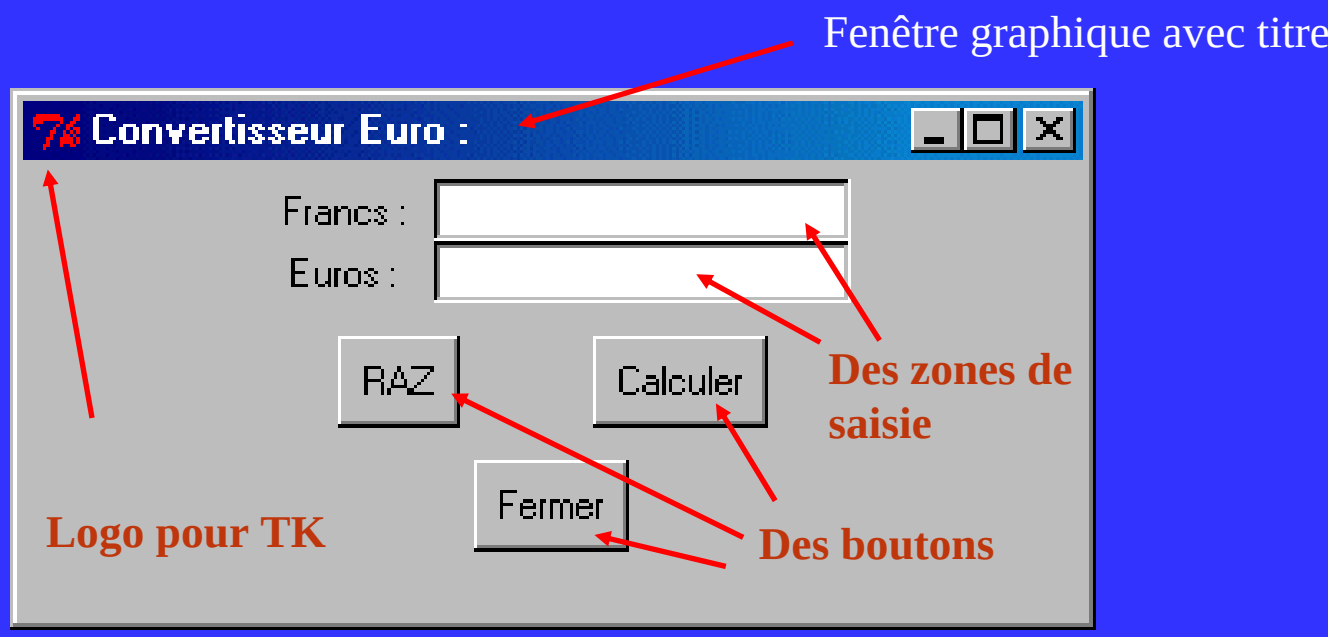
Incises : Perl2exe/PerlApp/Komodo

- Perl2exe est un utilitaire qui permet de convertir des scripts Perl en programmes exécutables
  - i.e. vous pouvez ainsi créer des programmes perl et les activer sans avoir à installer l 'interpréteur Perl
  - Sur le web :
  - http://www.indigostar.com/perl2exe.htm
  - Mode d 'emploi
    - sous MSDOS
      - perl2exe monprogramme.pl
      - en sortie : monprogramme.exe
  - Vous utiliserez cette petite application sur les programmes générés au LaboC pendant le cours
- PerlApp : programme ACTIVESTATE similaire à Perl2exe (mais payant)
- Komodo : IDE pour Perl

Retour sommaire

Lancement de calcEuro.pl

- Après activation du programme, la fenêtre suivante apparaît :



- Nous allons maintenant examiner le code pour voir comment ces composants graphiques ont été définis

Retour sommaire

Premier programme complet avec Perl Tk


```
#! /usr/bin/perl
use Tk;
$main = MainWindow->new(-title=>'Convertisseur Euro :');
$frame1 = $main->Frame()->pack();
$frameG = $frame1->Frame()->pack(-side=>'left', -pady=>5);
$frameD = $frame1->Frame()->pack(-side=>'right', -pady=>5);
$frame2 = $main->Frame()->pack();
$frame3 = $main->Frame()->pack();
$libelF = $frameG->Label(-text=>'Francs :')->pack(-padx=>5);
$montantF = $frameD->Entry()->pack();
$libelE = $frameG->Label(-text=>'Euros :')->pack(-padx=>5);
$montantE = $frameD->Entry()->pack();
$raz = $frame2->Button(-text=>'RAZ', -command=>\&vider)->pack(-
   side=>'left', -padx=>20, -pady=>5);
$calcul = $frame2->Button(-text=>'Calculer', -command=>\&calcul)-
   >pack(-side=>'right', -padx=>20, -pady=>5);
$fin = $frame3->Button(-text=>'Fermer', -command=>sub {exit})-
   >pack(-pady=>5);
MainLoop;
```

Retour sommaire

Les procédures appelées

```
sub calcul {
   $valeur=$montantF->get();
   if ($valeur ne "") {
        $montantE->delete(0, 'end');
        $montantE->insert(0, int($valeur*100/6.55957)/100);
       }
   else {
        $valeur=$montantE->get();
        if ($valeur ne ""){
              $montantF->delete(0, 'end');
              $montantF->insert(0, int($valeur * 655.957)/100);
          }
}}
sub vider
{
   $montantF->delete(0, 'end');
   $montantE->delete(0, 'end');
}
```

Retour sommaire

Premières clés : création d ’une fenêtre


```
#! /usr/bin/perl
use Tk;
$main = MainWindow->new(-title=>'Convertisseur Euro :');
MainLoop;
```

- La première ligne n ’est pas pertinente sous Windows
- La seconde ligne mentionne que l ’on utilise le module Tk
- La troisième ligne génère la fenêtre principale du programme avec un titre donné en argument
- La dernière ligne (une boucle particulière) indique au module d ’attendre une action de l ’utilisateur et de prévenir le programme
  - Le module Tk utilise une programmation événementielle : les actions de l ’utilisateur déclenchent des événements qui correspondent à des fonctions du programme.

Retour sommaire

Widget : une brique de base

- Tk permet de créer et de gérer des widgets et toutes les choses qui ressemblent à une interface graphique
- Un Widget est une brique de base manipulée dans une interface graphique
- Créer une interface graphique avec Perl/Tk c ’est :
  - créer, placer, manipuler des widgets

Retour sommaire

Différents types de widget

- On peut scinder les widgets en 2 types :
  - les conteneurs : ceux qui peuvent contenir d ’autres widgets :
    - fenêtres, cadres, menus, listes de choix…
  - les widgets de base : ceux qui ne contiennent pas d ’autres widgets
    - boutons, cases à cocher, boutons radio, barres de défilement...

Retour sommaire

Création d ’un widget

- Un même principe de base :
  - chaque widget doit avoir un parent qui le surveille et en garde une trace durant son existence
- Un exemple pour commencer
  - on suppose que le widget `$parent` existe déjà
    - Pour créer un widget de type `widgetType` :

```
$fils=$parent->widgetType(|
  -option => valeur,
  …|);
```

Retour sommaire

Ajout de boutons, zone de saisie...

```
#! /usr/bin/perl
use Tk;
$main = MainWindow->new(-title=>'Convertisseur Euro :');
$fin = $main->Button(-text=>'Fermer', -command=>sub
   {exit})
->pack(-pady=>5);
MainLoop;
```

- Perl/tk exploite une structure hiérarchique et tout objet doit posséder un parent
- Pour réaliser cela, on créé une référence à la fenêtre principale à travers une variable (`$main`)
- Pour créer un bouton dans cette fenêtre (fils de l 'objet fenêtre), on utilise la méthode button sur la référence $main
- De plus, la ligne de création du bouton affiche aussi le texte « Fermer » qui est associé à une action (une procédure dont la finalité est de quitter le programme)
- La méthode pack mentionne à Perl de placer le widget Button dans le widget principal

Retour sommaire

Intitulés et zone de saisie

```
...
$libelF = $frameG->Label(-text=>'Francs :')->pack(-padx=>5);
$montantF = $frameD->Entry()->pack();
$libelE = $frameG->Label(-text=>'Euros :')->pack(-padx=>5);
$montantE = $frameD->Entry()->pack();
...
MainLoop;
```

- Le code ci-dessus associe au widget `$frameG` un widget Etiquette (`Label`)
  - ce widget ressemble à un bouton, il contient du texte, il peut être mis en relief, avoir une font différente…
- Le widget de saisie (`Entry`) permet à l ’utilisateur de taper du texte
  - L ’option `-textvariable` permet de savoir ce que l ’utilisateur a tapé dans le widget ; le contenu est associé à la variable associée à cette option

Retour sommaire

Définir des actions sur les widgets


```
$raz = $frame2->Button(-text=>'RAZ', -command=>\&vider)-
   >pack(-side=>'left', -padx=>20, -pady=>5);
$calcul = $frame2->Button(-text=>'Calculer', -
   command=>\&calcul)->pack(-side=>'right', -padx=>20, -
   pady=>5);
```

- Le code ci-dessus associe des actions aux boutons définis :
  - le bouton raz défini dans le widget `frame2` est associé à un label texte « RAZ » et à une action définie par l ’appel de la procédure « `&vider` », le code de cette procédure est défini dans le programme (*cf* code complet)
  - le bouton `calcul` défini dans le widget `frame2` est associé à un label texte « `Calculer` » et à une action définie par l ’appel de la procédure « `&calcul` », le code de cette procédure est défini dans le programme (*cf* code complet)

Retour sommaire

Relecture du programme

- Commencer par relire le programme initial
- Modifier les champs des étiquettes
- Ajouter un `Frame` central avec une conversion en dollar
- Modifier les procédures de calcul

Retour sommaire

Pour aller plus loin

- Perl/Tk : une introduction
- Perl/XML : une introduction aux modules XML pour Perl
- Introductions à Perl/Tk (sur votre CD):
  - « Perl and the Tk Extension », *Steve Lidie*
  - « Getting Started With Perl/Tk », *Steve Lidie*

Retour sommaire