Université de Metz

U.F.R. SciF.A.

Nom :_____ **Prénom :**_____

Groupe :_____

T.P. de Système d'Exploitation Unix

M.S.T. Télécom 1re Année

Année Universitaire 2002/2003

Y. Morère

Cette page est laissée blanche intentionnellement

Table des matières

Les buts fixés pour cette série de tps sont les suivants :

- ☑ Se familiariser avec un système d'exploitation professionnel : UNIX.
- ☑ Maîtriser les commandes de base de ce système afin de réaliser des tâches simples (gestion de fichiers, courrier...).

Évaluation et notation :

- ☑ Le ou les compte-rendus comporteront les commandes saisies, les résultats obtenus ainsi que les réponses aux questions.

Cette page est laissée blanche intentionnellement

Chapitre 1

Automatiser les tâches : `make`

1.1 Que fait `make`

Sur les systèmes de la famille Unix, `make` remplit le même rôle que les gestionnaires de projets que l'on retrouve dans la plupart des environnements de développement intégrés (IDE) soit sous windows ou encore MacOS. Quel que soit son nom et sa forme, son objectif est toujours le même : centraliser l'ensemble des fichiers et ressources dont se compose un projet, gérer les dépendances et assurer une compilation correcte.
Ainsi si l'on modifie l'un des fichiers sources, le gestionnaire de projet en tiendra compte et saura qu'il faut recompiler ce fichier et procéder de nouveau à une édition de liens pour obtenir un exécutable à jour.
De plus `make` constitue un puissant langage de programmation, spécialisé dans la gestion des projets.

1.2 Dans le vif du sujet

Partons d'un exemple : `helloworld`. Mais dans notre cas il sera décomposé sur 2 fichiers sources et un fichier d'entête.
Fichier `main.c`

```
#include <stdio.h>
#include "helloworld.h"
int main(int argc, char *argv[])
{
    hello();
    exit(0);
}
```

Fichier `helloworld.h`

```
void    hello();
```

Fichier `helloworld.c`

```
#include <stdio.h>
void hello()
{
    printf("bonjour le monde\n");
}
```

Afin de compiler ce programme il est possible de le faire de trois manières différentes :

1. `gcc helloworld.c main.c -o helloworld`
2. `gcc -c helloworld.c`
   `gcc -c main.c`
   `gcc main.o helloworld.o -o helloworld`
3. ou avec `make`

La solution 1 convient bien s'il s'agit d'un petit exemple. Mais s'il est question d'un projet plus important il devient nécessaire d'écrire un script de compilation. Dans ce cas la solution 3, utilisant `make` est la plus élégante, car le programme utilise un fichier `Makefile` qui gère les dépendances entre les fichiers.
Nous allons donc écrire le fichier `Makefile`, il ressemble à ceci :

```
helloworld: main.o helloworld.o
    gcc -o helloworld main.o helloworld.o
```

```
main.o: main.c
    gcc -c main.c
helloworld.o: helloworld.c
    gcc -c helloworld.c
```

Un `Makefile` contient ainsi un ensemble de règles, dont chacune est constituée d'une "cible", de "dépendances" et de commandes. Il est important de de réaliser l'indentation des lignes ci-dessus avec des tabulations et non des espaces. Ceci occasionnerait des erreurs lors du `make`
La première règle définit la cible `helloworld` ce qui signifie que son rôle réside dans la production d'un fichier `helloworld`. Cette règle possède 2 dépendances `main.o` et `helloworld.o`. Cela indique que pour élaborer le programme `helloworld`, il faut préalablement disposer de ces 2 fichiers. Il vient ensuite la commande shell qui permet de générer `helloworld` à partir des dépendances. Cette commande consiste à appeler la compilation pour obtenir l'exécutable `helloworld` à partir des deux fichiers objets.
La règle suivante est encore plus simple, elle donne le moyen de créer le fichier objet `main.o`. La syntaxe d'un `Makefile` se révèle donc assez simple. On peut alors l'utiliser en vue de la recompilation de notre programme simplement e, lançant la commande `make helloworld` ou encore plus simplement `make`, car l'outil prend par défaut la première cible trouvée.
Que va t'il se passer ? `Make` cherchera à générer `helloworld` : pour cela il vérifiera d'abord si les fichiers requis sont disponibles. S'il manque par exemple `main.o`, il appliquera alors la règle pour produire ce fichier et ainsi de suite si `main.o` nécessitait d'autres dépendances. Une fois toutes les dépendances satisfaites, la commande pour produire `helloworld` sera exécutée afin d'obtenir notre fichier exécutable.

Exercice : Écrire les différents fichiers nécessaires à la mise en œvre du petit projet décrit ci-dessus (`main.c`, `helloworld.c`, `helloworld.h`) sans oublier le `Makefile`. Lancer la compilation (debugger si nécessaire). Ensuite modifier le fichier `helloworld.c` en remplaçant la phrase `bonjour le monde\n` par ce que vous voulez. Que remarquez-vous ?

Réponse :

1.3 Pourquoi passer par `make`

En fait, le principal intérêt de cet outil réside dans le fait qu'il n'effectue que le strict minimum. Ainsi comme vous l'avez fait précédemment, si seul le fichier `helloworld.c` est modifié, lors de la recompilation du projet, `make` constatera que la date de modification de `helloworld.c` est plus récente que la création du fichier `helloworld.o`, donc il le recompilera, par contre dans le cas de `main.c` tout est correct, et il n'a pas besoin de regénérer la fichier objet.
On gagne ainsi un temps considérable lors de la compilation de gros projet, en ne recompilant que ce qui est nécessaire.

1.4 Plus loin avec GNU `Make`

Bien sur si `make` apporte un aide non négligeable poue la compilation de projet, écrire un `Makefile` complet devient très vite agaçant dès que le projet devient important. Heureusement pour nous, tout ceci peut être automatisé.
GNU `make` propose des mécanismes grâce auxquels il peut déduire pratiquement tout seul les règles à appliquer. Comme il s'agit d'un langage, `make` gère les variables dont certaines possèdent une signification particulière. Il est important d'en connaître au moins 8 :
– `CC` définit le compilateur C par défaut,
– `CFLAGS` définit les options à lui transmettre,
– `CXX` et `CXXFLAGS` jouent le même rôle pour le compilateur C++,
– `LIBS` définit les librairies à utiliser pour la compilation,
– `DESTDIR` définit le chemin sur lequel le programme se verra installé un fois compilé,
– `@` et `<` représentent respectivement la cible et la dépendance courante.

De plus GNU `Make` possède des règles prédéfinies : ainsi il sait que par défaut il doit produire un fichier `toto.o` à partir d'un fichier `toto.c` en invoquant le compilateur défini par la variable `CC`, avec les options `CFLAGS`. Ainsi il est possible de simplifier le `Makefile` comme suit :

```
CC = gcc
OBJS = main.o helloworld.o
helloworld : $(OBJS)
    $(CC) -o $(@) $(OBJS)
```

On donne à `CC` la valeur `gcc` et l'on garde également les dépendances dans la variable `OBJS` afin d'éviter de les entrer manuellement. La seule règle que nous avons, indique comment produire `helloworld` à partir des dépendances définie par la variable `OBJS`. On utilise alors le compilateur indiqué par `CC`. On remarque aussi l'emploi de la variable `@` qui, à tout instant, représente la cible de la règle où elle figure ; dans le cas présent sa valeur est donc `helloworld`. Il ne s'avère plus nécessaire d'indiquer les règles pour produire `main.o` et `helloworld.o`.

Exercice : Écrire le nouveau `Makefile` et tester son bon fonctionnement.
Réponse :

1.5 Nouvelles règles prédéfinies

Si les règles prévues dans GNU `make` ne vous suffisent pas, il est possible d'en redéfinir des nouvelles. Il est alors possible de créer un fichier `postscript` à partir d'un fichier `dvi` par exemple lors de la rédaction de document sous LaTeX 2_ε.

```
%.ps: %.dvi
    dvips -ta4 -o $(@) $(<)
```

On exprime ici le faite, que l'on crée un fichier `postscript` à partir d'un fichier portant le même nom avec le suffixe `.dvi` en utilisant la commande `dvips`.

1.6 `make all`, installation et nettoyage

Afin de simplifier la compilation des programmes, la règle `all` est très intéressante. Celle-ci lancera la compilation complète de notre programme.

```
all: helloworld
```

Ce permet de plus d'avoir un terme générique pour lancer la compilation complète de n'importe quel projet.
La seconde chose importante est l'installation du programme. Cette nouvelle règle est dépendante de la compilation complète (`all`). Grâce à la commande `make install`, il sera possible de copier l'exécutable dans le sous-répertoire `bin` de `DESTDIR` ainsi que ses diverses ressources dans `DESTDIR/share`, la documentation dans `DESTDIR/doc` et la page de manuel dans `DESTDIR/man`.

```
install: all
    cp helloworld $(DESTDIR)/bin
    mkdir -p $(DESTDIR)/doc/helloworld
    cp manual.ps $(DESTDIR)/doc/helloworld
    cp README $(DESTDIR)/doc/helloworld
    cp helloworld.1 $(DESTDIR)/man/man1
```

Une fois l'installation faite, il ne reste plus qu'a faire le ménage des fichiers temporaires créés lors de la compilation et qui ne servent plus. La règle `clean` détruira ces fichiers. Il suffira de taper `make clean` pour obtenir une arborescence propre.

```
clean:
    rm -f *.o *~
```

Bien sur cette règle de dépend d'aucune autre.

Exercice : Compléter le fichier `makefile` précédent afin de pouvoir compiler, installer (dans le répertoire `bin` que vous aurez créé dans votre répertoire) et nettoyer le projet `helloworld`.

Réponse :

Chapitre 2

Introduction à la programmation `Perl`

2.1 Introduction

Larry Wall a inventé Perl pour introduire automatiquement des rapports et des états statistiques dans un système interne de forums de discussions. L'intitulé complet de ce langage reflète d'ailleurs cette première orientation : *Practical Extraction and Report Langage*.Perl est l'abréviation de *Practical Extraction and Report Langage* (langage pratique d'extraction et de génération de rapport), même s'il a été également baptisé *Pathologically Eclectic Rubbish Lister* (énumérateur de bétises pathologiquement éclectique).Très vite la structure de Perl, qui se voulait au départ une simple extention de `awk`, évolua sous l'influence de nombreux utilisateurs, pour incorporer des héritages d'autres langages comme `C`, `Sed` ou le `Shell`.

2.2 Généralités

Perl est un langage polyvalent, adaptable à de nombreuses situations, et d'une puissance sans cesse accrue. C'est un langage interprété et il ne convient donc pas pour les applications bas niveau. Par contre pour l'essentiel des tâches logiciels, il convient parfaitement.Le slogan de la communeauté Perl est "TIMTOWTDI – There Is More Than One Way To Do It". C'est la richesse du langage qui permet ceci.

2.3 Utilisation

L'interpréteur Perl se trouve en général dans le répertoire `/user/bin` est est invoqué sous le nom `perl`. L'invocation de l'interpréteur se fait avec la syntaxe suivante :

```
$ perl [options] -e 'commandes' [arguments]
```

ou :

```
$ perl [options] script [arguments]
```

On peut lancer un script directement avec une ligne `shebang`

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

En pratique, on utilisera toujours l'option `-w` (warning) qui affiche des avertissements justifiés lorsque l'interpréteur rencontre des expressions douteuses dans le code.

```
#!/usr/bin/perl -w
```

ou invoquerons diretement l'interpréteur en ligne de commande avec :

```
$perl -w -e 'commandes'
```

2.4 Expressions et variables

Perl introduit la notion de contexte en matière d'évaluation d'expression. Il en existe 2 : le contexte *scalaire* et le contexte *list*. En fonction du contexte d'évaluation, la même fonction peut fournir des résultats différents.On dit qu'une expression est évaluée dans un contexte scalaire quand on lui demande de renvoyer une valeur unique. Cette valeur peut être un nombre (entier ou réel) ou une chaîne de caractères.Dans un contexte de liste, l'évaluation fournit une succession de variables scalaires. Une liste est représentée comme une suite de valeurs scalaires séparées par des virgules, qui est en général encadrée par des parenthèses.Une expression est évaluée dans le

contexte correspondant à ce que l'on attend d'elle. S'il s'agit d'en affecter le résultat à une variable scalaire, elle sera evaluée dans un contexte scalaire. S'il faut l'ajouter à la fin d'une liste, le résultat est demandé dans un contexte de liste.Par exemple, les constantes `1`, `2`, `3e-4`, `"zéro"` sont des scalaires et `(1,2,3,5,7,11,13,17)` est une liste au même titre que `("alpha","bravo","charlie","delta")` ou `("un",2,"deux",4,"trois")`.Il y a un autre type de données scalaires, les références. Il s'agit de pointeurs vers des zones mémoire qui permettent d'accéder indirectement au contenu d'un objet, variable ou constante.Le langage Perl permet de manipuler des variables, structurées sous trois formes distinctes : les variables scalaires, les tableaux classiques et les tables de hachage.

2.4.1 Les variables scalaires

Une variable scalaire de Perl contient une donnée scalaire. N'étant pas typée, la même variable peut contenir successivement une chaîne de caractères, une valeur entière ou réelle, voire une référence qui pointe vers une autre variable. En revanche les opérateurs agissent pour la plupart sur des types de données précis, ce qui conduit l'interpréteur à effectuer des conversion implicites.Une variable scalaire est toujours préfixée par un caractère `$`. De manière générale, en Perl, toutes les variables sont préfixées d'une lettre qui indique leur type (consultation et affectation).Exemple de variables scalaires : `scalaires.pl`
Écrire et tester le programme.

```
#!
/usr/bin/perl -w# Affectations de quelques variables$pi = 3.1415926535;$pi_sur_2
= $pi / 2;$mnemo_pi = "Que j'aime a faire apprendre ce nombre utile aux
sages";print $mnemo_pi;print "\n";
print $pi;
print "\n";
print $pi_sur_2;
print "\n";
```

Quelques remarques :
– l'extention des scripts Perl est `.pl` et celle des modules Perl est `.pm` ;
– les commentaires commencent par le catactère `#` ;
– toutes les instructions doivent se terminer par un point-virgule
Les conversions entre nombre et chaînes sont assurées avec transparence lorsque l'interpréteur en ressent le besoin.

```
yann@yoda:~/script_linux$ perl -e '$chaine="12"; print $chaine+1; print "\n";'
13
yann@yoda:~/script_linux$ perl -e '$chaine="ABCD"; print $chaine+1; print "\n";'
1
yann@yoda:~/script_linux$ perl -e '$chaine="4AB"; print $chaine+1; print "\n";'
5
yann@yoda:~/script_linux$
```

Les opérateurs et les fonctions du langage Perl agissent toujours sur une type bien déterminé,numérique ou chaîne de caractères, ce qui décide des éventuelles conversions.Lorsqu'une liste est évaluée dans un contexte scalaire, elle est remplacée par son dernier élément.

```
yann@yoda:~/script_linux$ perl -e '$i=("aze","qsd","wxc"); print $i; print
"\n";'wxc
yann@yoda:~/script_linux$
```

Si nous utilisons l'option `-w`, l'interpréteur Perl nous avertit que les deux premières constantes de la liste ne seront pas utilisées.

```
yann@yoda:~/script_linux$ perl -we '$i=("aze","qsd","wxc"); print $i; print
"\n";'Useless use of a constant in void context at -e line 1.
Useless use of a constant in void context at -e line 1.
wxc
yann@yoda:~/script_linux$
```

Dans certains cas, les éléments intermédiaires pourraient avoir des effets de bords (`$i++`) pendant l'évaluation. L'évaluation des variables proposées par Perl est très puissante. Supposons que nous comptions une variable scalaire nommée `$i`, qui contienne par exemple le nombre 42. Supposons par ailleurs que nous ayons une variable scalaire nommée `$j` qui contienne la chaîne de caractères `"i"`. Nous pouvons alors écrire :

```
yann@yoda:~/script_linux$ perl -e
'$i=42; $j="i"; print $$j; print "\n";'42yann@yoda:~/script_linux$
```

Ce concept est appelé *référence symbolique*, par analogie avec les liens symboliques que l'on trouve sur les systèmes Unix. Ici une référence symbolique est une variable qui renferme le nom d'une autre variable.Lorsque le référence symbolique est complexe (concaténation de deux noms de variables), il est possible d'employer des accolades pour la délimiter explicitement. L'opérateur `'.'` permet de concaténer deux chaînes de caractères ; nous l'utilisons pour regrouper les deux moitiés du nom de la variable `$deux` et accéder à son contenu.

```
yann@yoda:~/script_linux$ perl -e '$deux=2; $i="de";
$j="ux"; print ${$i.$j}; print "\n";'2yann@yoda:~/script_linux$
```

On a donc reconstruit dynamiquement le nom de la variable pour en obtenir son contenu.

2.4.1.1 Nombres

En Perl, tous les nombres en interne utilisent le même format. Il est possible de spécifier des entiers ou des nombres en virgule flottante (réels, décimaux). Mais en interne Perl ne calcule que des valeurs en virgules flottantes. Il n'y a donc pas de valeurs entières interne à Perl.

Décimaux Le point permet de délimiter les décimaux.

```
1.25
255.000
7.25e45
-6.24e-24
```

Entiers

```
0
2004
-40
61298040283768
61_298_040_283_768
```

Entiers non décimaux Perl vous permet d'indiquer des nombres dans une base différente de la base 10. Les octaux (base 8) commencent par `0`, les héxadécimaux (base 16) commencent par `0x`, les binaires (base 2) commencent par `0b`.

```
perl -e '$toto=0b10 + 0b01;print
$toto\n";'

0377
0xff
0b11111111
0x50_65_72_7C
```

Opérateurs sur les nombres Perl propose les opérateurs ordinaires d'addition, de soustraction, de multiplication et de division, etc. Par exemple :

```
2 + 3
5.1 -2.4
3 * 12
14 / 2
10.2 /0.3
10 / 3
```

Perl supporte aussi l'opérateur modulo (`%`). La valeur de l'expression 10 % 3 représente le reste de la division entière 10 par 3 soit 1. Il existe aussi un opérateur de puissance (élévation à la puissance) de type FORTRAN. Il est représenté par une double astérisque, par exemple 2 ** 3 représente 2^3 soit 8.

2.4.1.2 Chaînes

Les chaînes sont des suites de caractères (comme `hello`). Elles peuvent contenir n'importe quelle combinaison de n'importe quels caractères. La chaîne la plus courte ne contient qu'un caractère. la chaîne la plus longue replit toute votre mémoire disponible. Les chaînes typiques sont des séquences de caractères imprimables de lettres de chiffres et de signes de ponctuation.

Chaînes entre apostrophes simples Une chaîne entre apostrophes simples est une suite de caractères encadrée par des apostrophes simples `'toto'`. Les apostrophes ne font pas partie de la chaîne, elles ne sont là que pour indiquer à Perl le début et la fin de la chaîne.Tout autre caractère autre que l'apostrophe ou une barre oblique est valide. Pour mettre une barre oblique inverse, il faut la faire précéder d'une autre barre oblique. Pour mettre une apostrophe, il faut aussi la faire précéder d'une barre oblique.

```
'fred''''Ne mettez pas qu\'apostrophe cette chaîne''barre
oblique \\''hello\n'
'hello
there'
'\'\\'
```

dans ce cas le `\n` n'est pas interpété.

Chaînes entre guillemets Une chaînes entre guillemets se comporte comme une chaîne d'autres langages. Il s'agit d'une série de caractères encadrée par des guillemets. Dans ce cas la barre oblique remplt son rôle en indiquant certains caractères de contrôle.

```
"barney"
"hello world\n"
"caractère guillemet \""
"caractère \t tabulation"
```

Contenu	Signification
`\n`	Nouvelle ligne
`\r`	Retour chariot
`\t`	Tabulation
`\f`	Saut de page
`\b`	Espace arrière
`\a`	Bell (sonnerie)
`\e`	Escape (caractère escape ASCII)
`\007`	Toute valeur ASCII Octal (ici 007 bell)
`\x7f`	Toute valeur ASCII Héxadécimal (ici 7f delete)
`\cC`	Toute caractère de controle (ici CTRL-C)
`\\`	Barre oblique inverse
`\"`	Guillemet
`\l`	Lettre suivante en minuscule
`\L`	Lettres suivantes en minuscule jusqu `\E`
`\u`	Lettre suivante en majuscule
`\U`	Lettres suivantes en majuscule jusqu `\E`
`\Q`	Tous les signes non alphanumériques sont transformés en caractères d'échappement jusque `\E`
`\E`	Met fin à `\L` `\U` ou `\Q`

TAB. 2.1 – Tableau récapitulatif des caractères de contrôle en Perl

Les chaînes en guillements sont à variables interpolées, ce qui signifie que les noms de certaines variables situées dans les chaînes sont remplacées par leurs valeurs courantes au moment où les chaînes sont utilisées.

Opérateurs de chaînes Les valeurs d'une chaîne peuvent être concaténées par l'opérateur `.`.

```
"hello" . "world" #équivaut à "helloworld"
"hello" . ' ' . "world" #équivaut à "hello world"
'hello world' . "\n" #équivaut à "hello world\n"
```

l'opérateur répétition de chaîne, constitué par la lettre minuscule `x`. Il prend son opérande de gauche (une chaîne) et en effectue autant de copies concaténées que le spécifie son opérande de droite (un nombre).

```
"fred" x 3 #équivaut à "fredfredfred""barney" x (4+1) #équivaut
à "barneybarneybarneybarneybarney"5 x 4 #équivaut à "5" x 4 soit "5555"
```

Conversion automatique entre nombres et chaînes Perl effectue la conversion automatique des nombres en chaînes si nécessaire. Ceci dépend de l'opérateur utilisé sur la valeur scalaire. Quand un opérateur attend un nombre `+`, Perl voit la valeur comme un nombre. Lorsqu'un opérateur attend une chaîne `.`, Perl voit la valeur comme une chaîne.

```
"Z" . 5 * 7 #équivaut à "Z" . 35 ou "Z35"
```

2.4.1.3 Variables scalaires

Une variable scalaire contient une seule valeur scalaire. Le nom des variables scalaire commence par le caractère `$` suivi d'un identificateur Perl. Cette identificateur est constitués de caractère alphanumérique mais ne peut pas être commencé par un chiffre. De plus on distingue les majuscules des minuscules. Par emple la variable `$Fred` est différente de `$fred`.
Afin de bien nommer ses variables, il faut leur donner un nom explicite ni trop court ni trop long.
Une variable qui sera utilisée seulement dans deux ou trois lignes rapprochées portera un nom simple, `$n` par exemple, alors qu'une variable utilisée dans tout un programme aura un nom plus significatif.
La plupart du temps les noms de variable de vos programmes seront en minuscules. Dans certains cas particulier on utilise des majuscules. Le tout majuscule (comme `$ARGV`) indique généralement que cette variable comporte des caractéristiques spéciales.
L'affectation des variables scalaires est l'opération la plus souvent utilisées. L'opérateur d'affectation en Perl est la signe `=` qui prend un nom de variable à gauche et lui affecte la valeur de l'expression située à droite.

```
$fred = 17;
$barney = 'hello';
$barney = $fred +3;
```

Il existe aussi des opérateurs d'affectation binaires qui sont en sorte des raccourcis d'opérations plus affectation. Ces opérateurs permettent de réduire la quatité de code.

```
$fred = $fred + 5;
$fred += 5;
```

Les deux affectations précédentes sont équivalentes. On peut ainsi écrire un opérateur d'élévation à la puissance `**=`

2.4.1.4 Sortie avec `print`

IL est souvent préférable d'imprimer les résultats. C'est l'opérateur `print` qui permet cela. Il prend en argument un scalaire et le place sur la sortie standard. Normalement il 's'agit de l'écran de votre terminal.

```
print "hello world\n";
print "la réponse est ";
print 6 * 7;
print "\n";
```

On peut aussi fournir une série de valeurs séparées par des virgules

```
print "la réponse est ", 6 * 7, ".\n";
```

2.4.1.5 Interpolation de variables scalaires en chaînes

Lorsqu'une chaîne est entre guillemets, elle est sujette à l'interpolation de variables. Cela signifie que tout nom de variable figurant dans la chaîne est remplacé par sa valeur en cours.

```
$meal = "frites dans la purée\n"
$plat = "la fondue belge c'est des $meal";
```

On désigne l'interpolation de variable par interpolation par guillements. Pour placer un vrai `$` il faut le faire précéder d'un `\`.
Si vous voulez accoler des lettres directements à la suite du contenu d'une variable, il faut utiliser les délimiteurs de noms de variables que sont les accolades.

```
$fruit = "pomme";
$phrase = "j\'ai 5 ${fruit}s\n;
```

2.4.1.6 Précédence et associativité des opérateurs

La précédence des opérateur permet de déterminer les cas ambigus dans lesquels deux opérateurs doivent intervenir sur 3 opérandes. Par exemple dans l'opération `2+3*5` le résultat sera `25` ou `17`.
il faut donc définir une précédence des opérateurs pour lever l'ambiguité. Il est possible de passer cette précédence en plaçant des parenthèses au endroits désirés.

Associativité	Opérateurs
gauche	les parenthèses et les opérateurs de listes
gauche	`->` `++ --` (auto-incrémentation, auto-décrémentation)
droite	`**`
droite	`\ ! ~ + -` opérateurs uniaires
gauche	`= ~ !~`
gauche	`* / % x`
gauche	`+ - .` opérateurs binaires
gauche	`<< >>` opérateurs unaires nommés (tests de fichiers -X, rand) `< <= > >= lt le gt ge` (les "non egal") `== != <=> eq ne cmp` (les "égal")
gauche	`&`
gauche	`| ^`
gauche	`&&`
gauche	`||`
droite	`? :` (ternaire)
droite	`= += -= .=` (et les opérateurs d'affectations similaires)
gauche	`, =>` (opérateurs de listes vers la droite
droite	`not` (non logique)
gauche	`and` (et logique)
gauche	`or` (ou logique) `xor` (ou exclusif)

TAB. 2.2 – Associativité et précédence des opérateurs du plus élevé au plus faible

2.4.1.7 Opérateurs de comparaison

Pour comparer les nombres, Perl propose les opérateurs de comparaison logique décrit dans le tableau ci-dessus. Ils renvoient *true* vrai ou *false* faux.

Comparaison	Numérique	Chaîne
Égal à	`==`	`eq`
Différent de	`!=`	`ne`
Inférieur à	`<`	`lt`
Supérieur à	`>`	`gt`
Inférieur ou égal à	`<=`	`le`
Supérieur ou égal à	`>=`	`ge`

TAB. 2.3 – Opérateurs numériques et de comparaison de chaînes

2.4.1.8 Valeurs booléennes

Contrairement à certains langages, Perl ne possède pas de type booléen indépendant, mais respecte quelques règles simples :

1. la valeur spécial `undef` est fausse ;
2. zéro est faux ; tous les autres nombres sont vrais ;
3. la chaîne vide `''` est fausse ; toutes les autres chaînes sont normalement vraies ;
4. seule exception, dans la mesure où les nombres et les chaînes sont équivalents, la forme chaîne de zéro `'0'` possède la même valeur que sa forme numérique fausse.

2.4.1.9 `STDIN` en tant que variable scalaire

Comment obtenir une valeur saisie au clavier dans un programme Perl. Le moyen le plus simple est l'opérateur d'entrée standard `<STDIN>`. Lorsque vous utilisez `<STDIN>` à un endroit où une valeur scalaire est attendue, Perl lit la totalité de la ligne de texte suivante de l'*entrée standard* jusqu'au prochain caractère de nouvelle ligne, et utilise cette chaîne comme valeur de `<STDIN>`.

```
$line = <STDIN>;
if ($line eq "\n")
{
print "Ceci n'était qu'une ligne vide!\n";
```

```
}
else
{
print "Cette ligne de saisie était : $line";
}
```

2.4.1.10 L'opérateur `chomp`

Cet opérateur enlève le dernier caractère d'une variable si celui-ci est un caractère de nouvelle ligne.

```
$text = "une ligne de texte\n";
chomp($text);
```

Il est alors possible de faire directement une affectation dans une variable à partir de `STDIN` en enlevant en même temps le caractère de fin de ligne.

```
chomp($text=<STDIN>);

$text = <STDIN>;
chomp($text);
```

`chomp` est une fonction et renvoie une valeur : le nombre de caractères supprimés (ce qui est rarement utile). Lorqu'une ligne se termine par plusieurs caractère de fin de ligne, la fonction `chomp` n'en supprime qu'un.

2.4.1.11 La valeur `undef`

Que se passe-t-il si vous utilisez une variable scalaire avant de l'avoir affecté. Rien de fatal, les variables possèdent une valeur spéciale `undef` avant d'avoir reçu leur première affectation. Si on essaie d'utiliser `undef` comme un nombre, la valeur vaut 0, comme un chaîne, la valeur vaut la chaîne vide.
Il est possible aussi de fixer une valeur à `undef`.

```
$brel = undef;
```

2.4.1.12 La fonction `defined`

L'opérateur d'entrée standard `<SDTIN>` peut renvoyer `undef` s'il n'y a plus d'entrée (fin de fichier par exemple). Pour savoir si une valeur est `undef`, on utilise la fonction `defined`, qui renvoie faux pour `undef` et vrai pour tout le reste.

```
$brel = <STDIN>;
if (defined($brel))
{
print "La saisie est $brel";
}
else
{
print "Aucune saisie disponible!\n";
}
```

2.4.2 Listes et tableaux

Une liste représente un ensemble de données scalaires ordonnées. Un tableau est une variable contenant une liste. Les termes sont souvent confondus, mais la liste représente les données et le tableau la variable. Une valeur de liste ne peut pas se trouver dans un tableau, mais toute variable tableau contient une liste.
Chaque élément de tableau ou de liste est une variable scalaire indivuduelle comportant une valeur scalaire indépendante.
Ces valeurs sont indexées par des entiers à partir de zéro. Le premier élément d'une liste est l'élément zéro.
Un tableau peut donc contenir une valeur scalaire nombre, chaîne `undef`. Le plus souvent on utilse des tableaux de même type.

2.4.2.1 Accès aux éléments d'un tableau

Perl propose une fonction d'indiçage permettant de faire référence à un élément du tableau par un indice numérique.
Les éléments d'un tableau sont numérotés par une suite d'entiers s'incrémentant à partir de zéro.

```
$toto[0] = "titi";
$toto[1] = "tutu";
$toto[2] = "tata";
```

le nom du tableau `toto` provient d'un espace de nom complètement indépendant de celui utilisé par les scalaires. On peut donc avoir une variable scalaire `toto` dans le même programme ; Perl les traiteras comme des entités différentes sans les confondre.
Il est possible d'employé un élément de tableau tel que `toto[2]` à quasiment tout endroit du programme ou il est possible d'utiliser une variable scalaire (sauf dans les contrôle de boucle). Il est possible modifier la valeur d'un élément d'un tableau.

```
print $toto[0];
$toto[1] = "tututiti";
$toto[2] = "tatatiti";
```

Bien sur il est possible de donner une expression donnant un valeur numérique en guise d'indice de tableau. Si cet valeur n'est pas un entier, elle est automatiquement tronquée à l'entier inférieur le plus proche.

```
$nombre = 1.725;
print $toto[$nombre - 1]; #identique à print $toto[0];
```

si l'indice d'un élément est au delà de la fin du tableau, la valeur correspondante est `undef` comme dans le cas des scalaires ordinaires. Quand une variable n'a jamais stockée de valeur elle est `undef`.

```
$blanc = $toto[127_234];
$blanc = $mel;
```

2.4.2.2 Indices spéciaux d'un tableau

Si vous effectuer un rangement dans un élément situé au delà de la fin d'un tableau ce dernier est automatiquement étendu. En cas de besoin les éléments intermédiaires sont créés en tant que valeur `undef`.

```
$roches[0] = 'sediment';
$roches[1] = 'ardoise';
$roches[2] = 'lave';
$roches[3] = 'minerai';
$roches[99] = 'shiste';
```

On a parfois besoin de l'indice du dernier élément du tableau. Pour le tableau `roches`, il s'agit de la valeur `$#roches`. Elle n'est pas égale au nombre d'élément, car il existe un élément zéro. Il est aussi possible d'affecter cette valeur pour modifier la taille du tableau.

```
$fin = $#roches; # 99, indice du dernier élément
$nombre_de_roches = $fin +1; # 100 éléments
$#roches = 2; # ouble toutes les roches après lave
$#roches = 99; #ajoute 97 élément undef
$roches[$#roches] = 'roches dures'; #la dernière roche
```

Il est assez courant d'utiliser la valeur `$#name` comme indice. Les indices de tableau négatifs comptent à partir de la fin du tableau. Mais ces indices ne bouclent pas. Si un tableau comportent 3 éléments, les indices négatifs sont `-1` (dernier élément), `-2` (l'élément intermédiaire) et `-3` (le premier élément).

```
$roches[-1] = 'roche dure'; # affectation du dernier élément
$dead_rock = $roches[-100]; # donne sédiment
$roches[-200] = 'cristal'; # erreur fatale !
```

2.4.2.3 Littéraux de liste

Un littérale de liste (manière dont vous représentez la valeur d'une liste à l'intérieur de votre programme), est constitués de valeurs séparées par des virgules, entre parenthèses. Ces valeurs constituent les éléments de la liste.

```
(1,2,3) #liste de 3 valeurs
(1, 2, 3,) #liste de 3 valeurs virgule de terminaison
("fred", 4.5) # 2 valeurs
() # liste vide
(1..100) #liste de 100 entiers
```

La dernière utilise l'opérateur étendue .., il crée une liste de valeurs en dénombrant à partir du scalaire de gauche jusqu'au sclaire de droite, une par une.

```
(1..5) #identique à (1,2,3,4,5)
(1.7..5.7) # idem mais les valeurs sont tronquées
(5..1) # liste vide ne dénombre qu'en remontant
(0, 2..6, 10, 12) # identique à (0,2,3,4,5,6,10,12)
($a..$b) # étendue déterminée par des valeurs
(0..$#roches) # les indices du tableau de roches de la section précédente
```

Les élément d'un tableau ne sont pas forcément des constantes. Ils peuvent être des expressions réevaluées à chaque utilisqation du littéral.

```
($a, 17) #deux valeurs : la valeur actuelle de $a et 17
($b+$c, $d+$e) #deux valeurs
```

On peut aussi avoire une liste de chaîne.

```
("toto", "tutu", "titi", "tata")
```

2.4.2.4 Le raccourcis `qw`

On a très souvent recours aux listes de mots simples. Le raccourci `qw` facilite leur génération en épargnant la saisie de nombreuses guillemets :

```
qw/ toto tutu titi tata/ #idem ci dessus mais moins de lettres frappées
```

`qw` signifie *quoted words* (mots entre guillemets) ou *quoted with whitespace* (cités par des blancs). Perl les traîte commeune chaîne entre apostrophes (on ne peut donc pas utiliser `\n` ou `$toto` dans une liste `qw`). Le blanc (espace ou nouvelle ligne) sont écartés.

```
qw/ toto
tutu titi
tata/ #idem ci dessus mais moins de lettres frappées
```

De même il n'est pas possible d'insérer des commantaires dans une liste `qw`. Perl permet de choisir n'importe quel délimiteur de `qw`. Tous les caractères de ponctuation peuvent être utilisé.

```
qw/ toto tutu titi tata/
qw! toto tutu titi tata!
qw{ toto tutu titi tata}
qw( toto tutu titi tata)
qw[ toto tutu titi tata]
qw< toto tutu titi tata>
```

Si vous désirez toujours utiliser le même délimiteur et insérer ce caractère dans la liste, il est toujours possible de la faire en le faisant précéder d'un \.

```
qw! yahoo\! google excite lycos! #comprend l'élément yahoo!
```

2.4.2.5 Affectation de liste

Comme les valeurs scalaires, il est possible d'affecter des valeurs de listes à des variables.

```
($toto, $barney, $dino) = ("silex", "ruine", undef);
```

les trois variables de la liste de gauche obtiennent de nouvelles valeurs, comme si l'on avait fait 3 affectations séparées. La liste étant constituée avant l'affectation, il est facile d'échanger deux variables en Perl :

```
($toto, $barney) = ($barney, $toto); # permute ces valeurs
($betty[0], $betty[1]) = ($betty[1], $betty[0]);
```

Que se passe t'il si le nombre de variables à gauche est différents du nombre de valeurs à droites. dans l'affectation d'une liste, les valeurs en trop sont silencieusement ignorées. Par contre si vous avez trop de variables, les extras obtiennent la valeur `undef`.

2.4.3 Tables classique

Les données scalaires peuvent être regroupées dans des tables indexées par une valeur numérique. Une variable table est toujours préfixée par la lettre '@' en rappel du a de *array*.Une table peut être initialisée par une constante prenant la forme d'une liste de scalaires entre parenthèses.

```
yann@yoda:~/script_linux$ perl -e '@semaine=("dim",
"lun", "mar", "mer", "jeu", "ven", "sam"); print @semaine; print
"\n";'dimlunmarmerjeuvensamyann@yoda:~/script_linux$ perl -e '@semaine=("dim",
"lun", "mar", "mer", "jeu", "ven", "sam"); print $semaine[0]; print "\n";'
dim
yann@yoda:~/script_linux$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu",
"ven", "sam"); print $semaine[1]; print "\n";'
lun
yann@yoda:~/script_linux$
```

On crée alors une table nommée `@semaine`, dont les éléments sont des chaînes de caractères. Le premier élément est d'indice 0 est `"dim"` et le dernier d'indice 6 est `"sam"`. On peut très bien mélanger au sein de la même table des chaînes et des valeurs numériques. Les éléments sont indépendants.Si la table est toujours préfixée par `@`, ses éléments individuels sont des scalaires et sont donc préfixés par un `$`.Pour modifier ou consulter les éléments de la table on utilisera :

```
yann@yoda:~/script_linux$ perl -e '@semaine=("dim",
"lun", "mar", "mer", "jeu", "ven", "sam"); print
$semaine[1];$semaine[2]="tuesday"; print $semaine [4];print $semaine [2]; print
"\n";'lunjeutuesday
yann@yoda:~/script_linux$
```

Il est important de comprendre que les espaces des noms de variables scalaires et des tables sont disjointes. La variable `$i` et la table `@i` peuvent coexister sans conflit. De plus il faut comprendre que `$i[0]` est un élément de `@i` et n'a rien a voir avec `$i`.On peut recopier directement une table dans une autre avec une simple affectation.

```
yann@yoda:~/script_linux$  perl -e
'@t=(1,2,3,4); @s=@t;print $s[3]; print "\n";'4
yann@yoda:~/script_linux$
```

On remarquera que tous les éléments de la table sont dupliqués. Si l'on modifie le contenu de la table initiale, la copie n'est pas modifiée.

```
yann@yoda:~/script_linux$  perl -e '@t=("1","2"); @s=@t;$t[0]="un";
print $s[0]; print " "; print $t[0]; print "\n";'
1 un
yann@yoda:~/script_linux$
```

Un dernière remarque : Lorsque l'on évalue une table dans un contexte scalaire, elle fournit le nombre de ces éléments. Une table est évaluée dans un contexte scalaire quand on essaie de l'affecter dans une variable scalaire.

```
yann@yoda:~/script_linux$ perl -e '@semaine=("dim",
"lun", "mar", "mer", "jeu", "ven", "sam"); $nb_jours=@semaine; print
$nb_jours;print "\n";'7yann@yoda:~/script_linux$
```

De même

```
yann@yoda:~/script_linux$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu",
"ven", "sam"); $nb_jours=@semaine; print @semaine + 0;print "\n";'
7
yann@yoda:~/script_linux$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu",
"ven", "sam"); $nb_jours=@semaine; print @semaine + 2;print "\n";'
9
yann@yoda:~/script_linux$
```

Ici on force l'évaluation de `@semaine` comme une valeur numérique pour l'additionner. Par contre si l'on a :

```
yann@yoda:~/script_linux$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu",
"ven", "sam"); print @semaine;print "\n";'
dimlunmarmerjeuvensam
yann@yoda:~/script_linux$
```

La table est évaluée dans un contexte de liste et est remplacée par la liste de tous ses éléments. Ici `print` accole tous les éléments de la liste qu'on lui transmet en argument.Il faut donc bien distinguer les listes, qui sont une organisation des données, et les tables, qui sont des variables. En particulier l'évaluation d'une liste dans un contexte scalaire en fournit le dernier élément, alors que l'évaluation d'une table dans le même contexte donne le nombre de ses membres.

```
yann@yoda:~/script_linux$ perl -e
'@t=("un", "deux", "trois", "quatre"); $i=@t;print $i;print
"\n";'4yann@yoda:~/script_linux$ perl -e '$i=("un", "deux", "trois", "quatre");
;print $i;print "\n";'quatre
yann@yoda:~/script_linux$
```

Comme les espaces des noms de variables scalaires et des tables sont disjointes, il est possible de créer une variable scalaire avec le même nom qu'une table. En règle générale on y stockera le nombre d'éléments.

```
yann@yoda:~/script_linux$ perl -e '@semaine=("dim",
"lun", "mar", "mer", "jeu", "ven", "sam"); $semaine=@semaine; print
$semaine;print "\n";'7yann@yoda:~/script_linux$
```

L'indice du dernier élément de la table peut être obtenu en préfixant le nom par `#`. Comme cette valeur est scalaire, elle doit être elle-même préfixée par `$`.

```
[yann@ulysse script_linux]$ perl -e
'@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam"); print
$#semaine;print "\n";'6
[yann@ulysse script_linux]$
```

Si l'on utilise un indice négatif pour accéder à un élément, le décompte se fait à rebours à partir de la fin de la table.

```
[yann@ulysse script_linux]$ perl -e '@semaine=("dim", "lun", "mar", "mer",
"jeu", "ven", "sam"); print $semaine[-1];print "\n";'
sam
[yann@ulysse script_linux]$
```

Le fait d'accéder à un élément d'indice supérieur au dernier de la table ajoute automatiquement cet élément.

```
[yann@ulysse script_linux]$ perl -e '@semaine=("dim", "lun", "mar", "mer",
"jeu", "ven", "sam"); $semaine[7]="sun";print $semaine[7];print "\n";'
sun
[yann@ulysse script_linux]$
```

De même si l'on essaie d'insérer un élément dans le dixième case du tableau les éléments intermédiaires seront automatiquement créés mais vides.

```
[yann@ulysse script_linux]$ perl -e '@semaine=("dim", "lun", "mar", "mer",
"jeu", "ven", "sam"); $semaine[7]="sun";$semaine[10]="wed";print $semaine[10];
print "\n";print @semaine+0;print "\n";'
wed
11
[yann@ulysse script_linux]$
```

Le nombre d'élément de la table est donc passé à 11.
On peut accéder à un élément par l'intermédiaire des indices négatifs, mais il n'est pas possible de créer des éléments avant l'indice 0 de la table. Il est possible d'extraire des sous parties d'une table en fournissant une liste des indices souhaités, séparés par des virgules ou en utilisant le symbole `".."` qui indique un intervalle.

```
[yann@ulysse script_linux]$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu", "ven",
"sam");@ouvrable=@semaine[1..5];print @ouvrable;print "\n";'lunmarmerjeuven
[yann@ulysse script_linux]$
```

```
[yann@ulysse script_linux]$ perl -e '@semaine=("dim", "lun", "mar", "mer",
"jeu", "ven", "sam");@jour_pair=@semaine[0,2,4,6];print @jour_pair;print "\n";'
dimmarjeusam
[yann@ulysse script_linux]$
```

Par contre si vous utilisez l'affectation suivante :

```
@nouvelle=@tab[4];
```

`@tab[4]` ne représente pas l'élément numéro 4 `$tab[4]` mais une table ne contenant qu'un seul élément. Il faudra alors consulter cette valeur par :

```
print $nouvelle[0];
```

Une table ne contient que des scalaires, mais un scalaire peut être une référence pointant vers une table, ce qui permet la création de tableaux multidimensionnels. On peut par exemple écrire :

```
@ouvrable=("lun", "mar", "mer", "jeu",
"ven");@semaine=("dim", @ouvrable, "sam");
```

qui est équivalent à :

```
@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
```

Il faut bien comprendre que le contenu de la table `ouvrable` est intégralement insérée au même niveau que les autres éléments de la liste. Les listes sont puissantes en Perl. Il est possible de les utiliser en tant que partie gauche d'une affectation. Par exemple :

```
[yann@ulysse script_linux]$  perl -e '($debut,$sommet,$fin)=(4,6,12);
print $debut." ".$sommet." ".$fin."\n";'
4 6 12
[yann@ulysse script_linux]$
```

Ceci permet des choses très interressantes comme les permutations :

```
[yann@ulysse script_linux]$  perl -e '($debut,$sommet,$fin)=(4,6,12);
print "debut:".$debut." fin:".$fin."\n";($debut,$fin)=($fin,$debut);
print "debut:".$debut." fin:".$fin."\n";'
debut:4 fin:12
debut:12 fin:4
[yann@ulysse script_linux]$
```

Perl assure que les valeurs seront correctement échangées ; il réalise les évaluations des expressions de la liste de droite, puis ensuite les assignations.Lorsque les listes des deux cotés n'ont pas le même nombre d'éléments, Perl agit quand même sans contrainte en ignorant les valeurs surnuméraires dans la liste à gauche du signe égal, ou les éléments en trop dans la liste de droite.Il est aussi possible de placer dans la liste de gauche une variable tableau.

```
[yann@ulysse script_linux]$  perl -e
'($debut,$sommet,$fin)=(4,6,12,15,14); print "debut:".$debut."
fin:".$fin."\n";($debut,$fin)=($fin,$debut);print "debut:".$debut."
fin:".$fin."\n";'debut:4 fin:12
debut:12 fin:4
[yann@ulysse script_linux]$

[yann@ulysse script_linux]$  perl -e '($debut,$fin,@valeurs)=(4,6,12,15,14);
 print "debut:".$debut." fin:".$fin." valeurs:".@valeurs."\n";'
debut:4 fin:6 valeurs:3
[yann@ulysse script_linux]$
```

On remarquera toutefois que le remplissage de la table se fait de manière gloutonne. C'est à dire qu'elle consso-mera toutes les valeurs disponibles dans la liste de droite.

```
[yann@ulysse script_linux]$  perl -e '($debut,@valeurs,$fin)=(4,6,12,15,14); print "debut:".$debut." fi
valeurs:".@valeurs."\n";'debut:4 fin: valeurs:4
[yann@ulysse script_linux]$
```

On voit ici que le valeur `fin` n'a pas été affectée puisque la table à récupérer les 4 dernières valeurs. Lorsque l'on connait bien le nombre d'éléments de la liste, il est possible de restreindre l'affectation à des sous-ensembles des listes contenues dans la partie gauche.

```
[yann@ulysse script_linux]$  perl -e
'(@vecteur_1[0,1], @vecteur_2[0,1], @vecteur_3[0,1])=(1,2,3,4,5,6); print
$vecteur_1[0]." ".$vecteur_1[1]." ".$vecteur_2[0]." ".$vecteur_2[1]."
".$vecteur_3[0]." ".$vecteur_3[1]."\n";'1 2 3 4 5 6
[yann@ulysse script_linux]$
```

Ici les différentes tables ne consomment que le nombre d'éléments correspondant à l'intervalle indiqué. Dans la pratique l'insertion d'une variable table dans une liste qui se trouve en partie gauche d'une affectation se fera presque toujours en dernière position de la liste. Cela est très utile quand on ne connait pas exactement la liste des éléments présents.On est souvent amené à insérer ou à extraire des éléments en tête ou en fin de liste. Même s'il existe des opérateurs spécialisés, il est possible de les remplacer par de simples affectations :

```
@table=($nouveau,@table);@table=(@table, $nouveau);
```

permet d'ajouter un élément en début de liste (resp. en fin de liste).

```
($inutile, @table)=@table;
```

permet de supprimer le premier élément de la liste. Pour supprimer le dernier élément de la liste, le méthode la plus simple est de décrémenter l'indice du dernier élément.

```
$#table--;
```

Il est possible d'utiliser la même stratégie pour éliminer le ième élément d'une liste.

```
@table=(@table[0..$i-1], @table[$i+1..$#table]);
```

```
[yann@ulysse script_linux]$ perl -e '@semaine=("dim", "lun", "mar", "mer",
"jeu", "ven", "sam");$i=3; print
$i."\n";@semaine=(@semaine[0..$i-1],@semaine[$i+1..$#semaine]); print @semaine;
print "\n"; '3
dimlunmarjeuvensam
[yann@ulysse script_linux]$
```

Voici un petit excercice qui va vous permettre de tester tout ce que nous avons vu.

```
#! /usr/bin/perl -w
# fichier tables.pl

@semaine = ("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
for ($i = 0; $i < @semaine; $i ++) {
print "semaine[$i] = $semaine[$i]\n";
}
print "semaine = @semaine\n";

@pairs = @semaine [0, 2, 4, 6];
print "pairs = @pairs\n";

($weekend[0], @ouvrables[0..4], $weekend[1]) = @semaine;
print "weekend = @weekend\n";
print "ouvrables = @ouvrables\n";

($mini, $maxi) = (12, 8);
print "(mini, maxi) = ($mini, $maxi)\n";

if ($mini > $maxi) {
($mini, $maxi) = ($maxi, $mini);
}
print "(mini, maxi) = ($mini, $maxi)\n";
```

Rédigez ce petit script et testez le :

2.4.4 Protection des expressions

D'après le script qui suit et en le testant, expliquer le mécanisme de protection des expressions en Perl :

```
#! /usr/bin/perl -w
#fichier protection.pl

$nombre = 12;
$chaine = "abc def";
@table  = ("un", "deux", "trois");

print "Protection forte avec des apostrophes\n";
print '$nombre $chaine @table \n';

print "\nProtection faible avec des guillemets\n";
print "$nombre $chaine @table \n";

print "\nPas de protection\n";
print 12, $chaine, @table;

print "\n";
```

Ecrivez ce petit script et testez le.

Lorsque

l'on désire délimiter explicitement le nom d'une variable, on peut l'encadrer par des accolades. Si l'on souhaite affiche le contenu de la variable `$t`, suivi de la chaîne de caractère `[0]`, sans que cela soit considérer comme le premier élément de la table `@t` on peut ecrire `${t}[0]`

```
[yann@ulysse script_linux]$ perl -e
'@t=(1,2); $t=3; print "${t}[0]\n";print "$t[0]\n"'3[0]1[yann@ulysse
script_linux]$
```

données important disponible en Perl correspond aux *tables de hachage*. On peut également les trouver sous le nom de tableaux associatifs. Ces tableaux se représentent sensiblement comme des tableaux classiques, mais l'indexation ne se fait plus par nombre entier, mais par une chaîne de caractères que l'on nomme *clé*. Ce type de structure offre un accès efficace aux données.Le nom de variable d'une table de hachage est préfixée par un `%`. Ses éléments, comme ceux d'une table classique, sont des scalaires et donc préfixés par `$`. Pour accéder à un élément, on place la clé entre accolades.Par exemple si `%semaine` est une table de hachage :

```
$semaine{"lundi"}="monday";$semaine{"mardi"}="tuesday";
...
```

Et l'on pourra consulter un élément par :

```
print $semaine{"lundi"};
```

qui affichera `monday`.
On remarquera que la chaîne de caractères qui sert de clé peut contenir des caractère accentués, des caractères spéciaux et même des espaces. Lorsque la clé ne contient que des caractères alphanumériques "classiques" (7bits), les guillemets à l'intérieur des accolades sont facultatives.L'initialisation d'une table de hachage se fait par l'intermédiaire d'une liste qui contient des paires clés/valeurs :

```
%semaine=("lundi", "monday", "mardi", "tuesday",
"mercredi", "wednesday","jeudi", "thursday", "vendredi", "friday", "samedi",
"saturday"),"dimanche", "sunday");
```

Il est donc possible très rapidement de convertir une table de hachage en table standard par l'expression :

```
@semaine=%semaine;
```

L'implémentation d'une table de hachage ne préserve pas l'ordre de saisie des éléments. Aussi dans la table classique, les paires clés/valeurs ne figureront elles pas dans le même ordre que la liste originale. Le script suivant illustre ceci.

```
[yann@ulysse 14]$ more hachages.pl#! /usr/bin/perl -w
# fichier hachages.pl

%semaine = ("lundi", "monday", "mardi", "tuesday", "mercredi", "wednesday",
        "jeudi", "thursday", "vendredi", "friday", "samedi", "saturday",
        "dimanche", "sunday");

@semaine=%semaine;

for ($i = 0; $i < @semaine; $i ++) {
print '$semaine [', $i, '] = ', $semaine[$i], "\n";
}
[yann@ulysse 14]$ ./hachages.pl
$semaine [0] = mercredi
$semaine [1] = wednesday
$semaine [2] = dimanche
$semaine [3] = sunday
$semaine [4] = lundi
$semaine [5] = monday
$semaine [6] = mardi
$semaine [7] = tuesday
$semaine [8] = vendredi
$semaine [9] = friday
$semaine [10] = samedi
$semaine [11] = saturday
$semaine [12] = jeudi
$semaine [13] = thursday
[yann@ulysse 14]$
```

Afin de rendre plus lisible l'association clé/valeur dans les initialisations statiques, le langage Perl propose l'opérateur `=>` :

```
%semaine = (
        "lundi"    => "monday",
        "mardi"    => "tuesday",
```

```
        "mercredi" => "wednesday",
        "jeudi"    => "thursday",
        "vendredi" => "friday",
        "samedi"   => "saturday",
        "dimanche" => "sunday"
);
```

Un certain nombre de variables sont prédéfinies en Perl, ce qui permet de paramétrer le comportement de l'interpréteur. Par exemple à son lancement, l'interpréteur définit automatiquement une variable classique `@ARG` qui contient les arguments passés en paramètre du script. Il définit également une table de hachage `%ENV` qui offre un accès aux variables d'environement du processus. Les clés sont les chaînes de caractères des noms des variables.Le script suivant illustre ce que l'on vient d'exposer :

```
#! /usr/bin/perl -w# fichier getenv.plfor ($i = 0; $i
< @ARGV; $i ++) {        print "$ARGV[$i] : $ENV{$ARGV[$i]}\n";
}
```

```
[yann@ulysse 14]$ ./getenv.pl USER HOSTNAME HOME INEXISTANTE
USER : yann
HOSTNAME : ulysse.lasc.univ-metz.fr
HOME : /home/yann
Use of uninitialized value in concatenation (.) or string at ./getenv.pl line 3.
INEXISTANTE :
[yann@ulysse 14]$
```

2.5 Les opérateurs

Après avoir vu les différents types de données manipulés par Perl, nous allons examiner les opérateurs qui permettent de jouer sur ces expressions.

2.5.1 Opérateurs Numériques

Nous retrouvons les opérateurs arithmétiques et logiques habituels, ainsi que des opérateurs de manipulation de bits.

Symbole	Nom	Exemple d'utilisation
or	OU logique	if(($a == 0) or ($b == 0)){
xor	OU EXCLUSIF logique	if(($a == 0) xor ($b == 0)){
and	ET logique	if(($x < 0) and ($y < 0)){
not	Négation logique	if(not ($valeur < $mini)){
? :	Test	$maxi = ($a > $b) ? $a : $b;
=	Affectation	$i = $j
||	OU logique	if(($a == 0) \|\| ($b == 0)){
&&	ET logique	if(($x < 0) && ($y < 0)){
|	OU binaire	$a = 0x19 | 0x88; # $a contient 0x99
^	OU EXCLUSIF binaire	a = 0x37 ^ 0xFC; # a contient 0xCB !
&	ET binaire	$a = 0x37 & 0xFC; # $a contient 0x34
==	Test d'égalité	if ($i == $y) {
!=	Test de différence	if ($i != $y) {
<=>	Comparaison signée	$a = ($i <=> $y);
<	Test d'infériorité stricte	if ($i < $y) {
<=	Test d'infériorité	if ($i <= $y) {
>	Test de supériorité stricte	if ($i > $y) {
>=	Test de supériorité	if ($i >= $y) {
<<	Décalage binaire	$a = 0x12 << 2; # $a contient 0x48
>>	Décalage binaire	$a = 0xC4 >> 2; # $a contient 0x31
+	Addition	$a = $x + $y;
-	Soustraction	$a = $x - $y;
*	Multiplication	$a = $x * $y;
/	Division	$a = $x / $y;
%	Modulo	$s = $t % 60;
!	Négation logique	$a = ! $resultat;
-	Moins unaire	$s = -$t;
~	Complementation binaire	$s = ~ 0x35013501; # $a contient 0x CAFECAFE
**	Exponentiation	$x = 2 ** $n;
++	Incrémentation	$i ++;
-	Décrémentation	$i --;

L'opérateur de comparaison <=> renvoie -1 si son argument de gauche est inférieur à celui de droite, 0 s'ils sont égaux et +1 si l'opérande de gauche est supérieur à celui de droite.Comme en C, certain opérateur peuvent être combiné avec une opération. On retrouve +=, -=, /=, *=.

2.5.2 Opérateurs de chaîne

Le langage Perl ne néglige pas le traitement de chaîne de caractères. Il offre des opérateurs performants. La première opération proposée est la concaténation de chaîne. L'opérateur est un simple ".".

```
yann@yoda:~/script_linux$ perl -e '$a="abc" . "def"; print $a . "\n";'
abcdef
yann@yoda:~/script_linux$
```

On dispose aussi d'un opérateur de répétition noté "x", pour rappeler le signe multiplier. Il construit la chaîne en multipliant la chaîne de gauche autant de fois qu'on lui indique à droite.

```
yann@yoda:~/script_linux$ perl -e '$a="abc" x 3 . "def";
print $a . "\n";'abcabcabcdefyann@yoda:~/script_linux$
```

Cet opérateur est utile pour faire de la mise en page en mode texte. En guise d'exercice, écrire un petit script qui affiche le texte qu'on lui passe en paramètre en l'encadrant. Vous devriez avoir un affichage de ce type :

```
yann@yoda:~/script_linux/14$ ./encadre_2.pl toto titi tata tutu
+------------------------------+
| toto                         |
| titi                         |
| tata                         |
| tutu                         |
+------------------------------+
yann@yoda:~/script_linux/14$
```

L'opérateur de répétition peut être utilisé pour initialiser une liste.

```
@table = (1) x 15;
```

est équivalent à

```
@table = (1,1,1,1,1,1,1,1,1,1,1,1,1,1,1);
```

Comme une table évaluée en contexte scalaire renvoie son nombre d'élément, on peut l'utiliser en partie droite de l'opérateur `x`. On peut alors réinitialiser tous les éléments d'une table en faisant :

```
@table =
(0) x @table;
```

Il existe des opérateurs de comparaison pour les chaînes de caractères. Ils sont différents des comparateurs numériques et sont représentés sous forme littérale. L'opérateur `eq` *(equal)* vérifie si deux opérandes sont égaux.

```
if (($chaine eq "oui") or ($chaine eq "non"))
{
...
}
```

Si l'on compare deux chaînes de caractères avec l'opérateur `==`, Perl effectue la conversion des chaînes en valeurs numériques et compare les résultats. D'ou l'utilité de l'option `-w`.

```
yann@yoda:~/script_linux/14$ perl -e '$c="ABC";
$d="DEF"; if ($c == sd) {print "OK\n";}'OKyann@yoda:~/script_linux/14$ perl -we
'$c="ABC"; $d="DEF"; if ($c == sd) {print "OK\n";}'Unquoted string "sd" may
clash with future reserved word at -e line 1.Name "main::d" used only once:
possible typo at -e line 1.Argument "sd" isn't numeric in numeric eq (==) at -e
line 1.Argument "ABC" isn't numeric in numeric eq (==) at -e line 1.
OK
yann@yoda:~/script_linux/14$
```

L'opérateur `ne` *(not equal)* teste la différence des chaînes.

```
if (($chaine ne "oui") and ($chaine ne "non"))
{
...
}
```

Les opérateur `gt` *(greater than)* et `lt` *(lesser than)* testent respectivement si une chaîne est supérieure ou inférieure à l'autre. Les comparaisons sont faites caractère par caractère suivant l'ordre des codes Ascii. On dispose également des opérateurs `ge` *(greater or equal)* et `le` *(lesser or equal)* qui sont vrais aussi si les chaînes sont égales. L'opérateur `cmp` se conduit comme `<=>` pour les nombres, en renvoyant les valeurs -1,0 et1 suivant que l'opérande de gauche est supérieur, égal ou inférieur à celui de droite.

2.6 Structures de contrôles

2.6.1 Structure de test

2.6.1.1 Tests avec `if`

Les tests sont réalisés en employant la construction

```
if (condition_1)  { action_1; }
elseif
{
action_2;
}
else
{
action_par_defaut;
}
```

La condition évaluée par un test `if` est une expression du type de celles que nous venons de voir. Il faut noter que les parties actions d'une structure `if` doit toujours être encadrées par des accolades, même si elles ne sont composées que d'une ligne.Un test peu aussi être construit en faisant suivre une instruction simple :

```
action if (condition);
```

l'action n'est réalisée que si la condition est vérifiée. On dit alors que le test est un *modificateur* de l'instruction simple.Par exemple :

```
for ($i=0; $i < @table; $i++)
{
next if ($table[$i]==0);
...
}
```

permet de passer au cas suivant (action `next`) lorsqu'un élément n'est pas intéressant durant le parcours d'une table.

2.6.1.2 Tests avec `unless`

Le mot clé `unless`, qui signifie "sauf si", a le comportement inverse de `if`. Cela signifie qu'une construction

```
unless (condition)
{
action_1;
}
else
{
action_2;
}
```

est équivalente à

```
if (not condition)
{
action_1;
}
else
{
action_2;
}
```

ou encore

```
if (condition)
{
action_2;
}
else
{
action_1;
}
```

La plupart du temps `unless` n'est pas utilisé avec un bloc `else` car le schéma devient difficile à lire. Il est souvent employé comme modificateur d'instructions simples.

2.6.1.3 Tests par court-circuits

Il est possible de rendre dépendantes l'exécution des commandes du shell en les liant par des opérateurs `&&` ou `||`. Le même principe peut être appliqué aux instructions Perl grâce au mécanisme de court circuit.
Exemple :

```
expression_1 or expression_2;
```

La préséance du `or` étant très faible, `expression_1` va être entièrement évaluée. Si cette expression est vrais, Perl n'a pas besoin d'aller plus loin et l'`expression_2` est ignorée.Ceci permet de réaliser des tests *courts-circuits*, que l'on peut lire "*expression_1 doit être vraie sinon essayer expression_2*". Cette structure est très souvent employée avec l'instruction `die` qui permet d'arrêter un script avec une message d'erreur. Voici un exemple qui utilise la fonction `open` pour l'ouverture d'un fichier :

```
if (! open(FIC, $nomp_fichier)) { die
"impossible d'ouvrir $nom_fichier.\n" }
```

ou encore

```
unless(open(FIC, $nomp_fichier))
{
die "impossible d'ouvrir $nom_fichier.\n"
}
```

ou encore

```
die "impossible d'ouvrir $nom_fichier.\n" unless open(FIC, $nomp_fichier);
```

2.6.2 Structures de boucles

Il y a différentes manières de réaliser des itérations en Perl.

2.6.2.1 boucle `while`

La boucle `while` "tant que" se présente comme ceci :

```
while (condition)
{
action;
}
```

Elle exécute le contenu du bloc *action* tant que la *condition* est vérifiée. La commande `next` permet de d'abandonner l'itération en cours et de revenir au test. La commande `last` permet de quitter la boucle. Il est possible d'utiliser l'instruction `while` comme modificateur d'instruction simple :

```
action while(condition);
```

Attention : Si la condition de début est fausse, l'action n'est jamais exécutée.

```
[yann@ulysse yann]$ perl -e 'print "action $i\n" while ($i++ < 4);'
action 1
action 2
action 3
action 4
[yann@ulysse yann]$ perl -e 'print "action $i\n" while (1 == 0);'
[yann@ulysse yann]$
```

Afin que l'action soit exécutée au moins une fois, il faut utiliser le mot clé `do`. *En réalité `do` est une fonction qui exécute le bloc passé en argument et qui renvoie la valeur de la dernière expression évaluée.*

```
do
{
action;
} while (condition);
```

exemple :

```
[yann@ulysse yann]$ perl -e 'do {print "action $i\n"} while (1 == 0);'
action
[yann@ulysse yann]$
```

2.6.2.2 Boucle `until`

Le mot clé `until` "jusqu'à ce que" permet de construire des boucles inverses de `while`, l'action étant exécutée tant que la condition n'est pas vérifiée.

```
until '$fin_demandee)
{
$chaine = $lecture_ligne();
traitement_chaine($chaine);
}
```

ou

```
lecture_chaine() until ligne ne "");
```

ou encore

```
do
{
$resultat = decrypte($message, $cle++);
} until (en_clair ($resultat));
```

2.6.2.3 Boucle `for`

La boucle `for` s'emploie comme en C ou avec Awk.

```
for␣(action_initiale;test;action_iterative)
{
action;
}
```

En réalité les trois arguments de la boucle `for` sont des expressions qui sont évaluées :
- la première expression `action_initiale`, est évaluée avant de démarrer la boucle. On l'emploie en général pour fixer la valeur de départ d'un compteur ;
- la deuxième expression `test` est évaluée avant chaque itération de la boucle. Si elle renvoie une valeur fausse, la boucle `for` se termine ;
- la troisième expression est évaluée à la fin de chaque itération. Elle sert en général de compteur.

l'emploi le plus courant est celui-ci :

```
for ($i = 0; $i < @table; i++)
{
print "$i : table[$i] \n";
}
```

Les expressions sont toutes facultatives ; en particulier l'absence de test dans le second membre permet de créer une boucle infinie dont il faudra sortir par une rupture de séquence explicite :

```
for(;;)
{
$touche = menu_principal();
last if ($touche == 'q');
jeu() if ($touche == 'j');
}
```

On peut aussi cumuler plusieurs expressions dans le même membre grâce à l'opérateur ",".

```
for ($i=0,$j=0; $i + $j < $n;$i += $increment_i, $j += $increment_j)
{
...
}
```

La commande `next` permet de passer à l'itération suivante, alors que `last` permet de sortir de la boucle.

2.6.2.4 Boucle `foreach`

Une autre structure sert à itérer automatiquement les éléments d'une liste. Le mot clé `foreach` est un synonyme de `for`. `foreach` s'utilise ainsi :

```
foreach $variable(liste)
{
action;
}
```

La séquence d'action sera répétée en plaçant dans la variable successivement tous les éléments de la liste. Cette dernière peut être fournie sous la forme de constante entre parenthèses et virgules :

```
[yann@ulysse yann]$
perl -e 'foreach $a ("un","deux","trois") {print "$a\n"};'un
deux
trois
[yann@ulysse yann]$
```

mais cela ne présente que peu d'intérêt. Le plus souvent une liste d'argument sera une variable table :

```
[yann@ulysse yann]$ perl -e '@t=(1,2,3,4,5,6,7,8,9);foreach $a (@t) {print
"$a\n"};'1
2
3
4
5
6
7
8
9
[yann@ulysse yann]$
```

on peut aussi utiliser :

```
for ($i = 0;$i < @t; $i++)
{
print "$t[$i]\n";
}
```

mais ceci est bien moins élégant.

En guise d'exercice, ecrivez ce petit script et testez le. Quelles conclusions tirez-vous.

```
[yann@ulysse 14]$ cat foreach_hachage.pl
#! /usr/bin/perl -w

%semaine = (
        "lundi"    => "monday",
        "mardi"    => "tuesday",
        "mercredi" => "wednesday",
        "jeudi"    => "thursday",
        "vendredi" => "friday",
        "samedi"   => "saturday",
        "dimanche" => "sunday"
);

foreach $jour (%semaine) {
        print "$jour\n";
}
[yann@ulysse 14]$
```

Ecrivez maintenant ce script et testez le. Quelles conclusions tirez-vous.

```
[yann@ulysse 14]$ cat foreach_keys.pl
#! /usr/bin/perl -w

%semaine = (
        "lundi"    => "monday",
        "mardi"    => "tuesday",
        "mercredi" => "wednesday",
        "jeudi"    => "thursday",
        "vendredi" => "friday",
        "samedi"   => "saturday",
        "dimanche" => "sunday"
);

foreach $jour (keys %semaine) {
        print "$jour <=> $semaine{$jour}\n";
}
[yann@ulysse 14]$
```

2.6.2.5 Rupture de séquence

Trois instructions permettent de modifier le comportement des boucles. Il s'agit de `next`, `last` et `redo`. Le mot clé `next` passe à l'expression suivante. Le mot clé `last` fait sortir de la boucle. `redo` possède un comportement légèrement différent. Il reprend l'itération, mais :
- pour les boucles `while`, il ne vérifie pas la condition de boucle ;
- pour les boucles `foreach`, il recommence l'action sans passer à l'élément suivant de la liste ;
- pour les boucles `for`, il n'exécute pas le troisième membre de `for` et ne vérifie pas la condition du second membre.

2.6.2.6 Les étiquettes de bloc

Les boucles `for`, `foreach` et `while/until` peuvent être précédées d'une étiquette. C'est à dire d'un mot -par convention en majuscules- suivi de deux points. Lorsque plusieurs boucles sont imbriquées, les commandes `next`, `last` et `redo` peuvent prendre en argument une étiquette qui indiquent à quelle bouble elle se rapporte. Par défaut l'action de ces arguments s'applique à la boucle la plus interne.

2.7 Définitions de fonctions

2.7.1 Définition et invocation

La définition d'une fonction Perl se fait grâce au mot clé `sub`. Ce dernier doit être suivi du nom de la routine à enregistrer, d'un éventuel prorotype des arguments entre parenthèses, et du bloc représentant la fonction entre accolades.Comme en C, il existe une différence entre la déclaration d'une fonction et sa définition. Cela signifie que l'on peut déclarer une fonction pour permettre à l'interpréteur de connaître la fonction qu'il aura à utiliser par la suite.

```
sub
fonction(arguments);
```

office de déclaration s'il n'y en a pas eu se présente ainsi :

```
sub fonction (arguments){instructions;
}
```

Pour invoquer une fonction, il suffit de citer son nom suivi des arguments :

```
fonction $arg1, $arg2;
```

ou

```
fonction ($arg1, $arg2);
```

Mais la seconde est préférable.

2.7.2 Paramètres et résultat

Les paramètres d'une fonction lui sont toujours transmis dans une table simple, nommée `@_`. Cela signifie que le premier argument sera accessible sous le nom `$_[0]`, le deuxième sous le nom `$_[1]` et ainsi de suite jusqu'au dernier `$_[$#_]`.Une fonction renvoir un résultat par l'intermidiaire de la fonction `return`. Il ne s'agit pas d'un scalaire, mais d'une liste de scalaires.Ecrivez et testez ce petit programme.

```
[yann@ulysse 14]$ cat exemple_sub_1.pl
#! /usr/bin/perl -w

sub somme
{
        $somme = 0;
        foreach $val (@_) {
                $somme += $val;
        }
        return ($somme);
}

print "1+2 = " . somme (1, 2) . "\n";
print "1+2+3+4 = " . somme (1, 2, 3, 4) . "\n";

[yann@ulysse 14]$
```

Ecrivez

et testez ce petit programme.

```
[yann@ulysse 14]$ cat exemple_sub_4.pl
#! /usr/bin/perl -w

sub produit_vectoriel
{
        (@u[0..2], @v[0..2]) = @_;
        $w [0] = $u[1] * $v[2] - $u[2] * $v[1];
        $w [1] = $u[2] * $v[0] - $u[0] * $v[2];
        $w [2] = $u[0] * $v[1] - $u[1] * $v[0];
        return (@w);
}

sub affiche_vecteur
{
        ($x, $y, $z) = @_;
        return ("($x, $y, $z)");
}

@i = (1, 0, 0);
@j = (0, 1, 0);
@k = produit_vectoriel (@i, @j);

print affiche_vecteur (@i);
print " x ";
print affiche_vecteur (@j);
print " = ";
print affiche_vecteur (@k);
print "\n";
[yann@ulysse 14]$
```

2.7.3 Passage des arguments

Le passage des arguments en Perl se fait toujours par référence. C'est à dire que dans sa table `@_`, la fonction a accès à la véritable variable qui se trouve sur la ligne d'invocation de la fonction. Toute modification de la table `@_` aura des répercutions au niveau supérieur d'exécution.Le programme suivant permet de tester ceci. Que remarquez vous :

```
[yann@ulysse 14]$ cat exemple_sub_5.pl #! /usr/bin/perl -w

sub efface
{
        for ($i = 0; $i < @_; $i++) {
                $_[$i] = 0;
        }
}

$a = 4;
$b = 5;
print "a=$a b=$b\n";
efface ($a, $b);
print "a=$a b=$b\n";
efface (12);
[yann@ulysse 14]$
```

Ecrivez et testez ce petit programme. Que remarquez vous ?

```
[yann@ulysse 14]$ cat exemple_sub_6.pl
#! /usr/bin/perl -w

sub efface
{
        @args=@_;
        for ($i = 0; $i < @args; $i++) {
                $args[$i] = 0;
        }
}

$a = 4;
$b = 5;
print "a=$a b=$b\n";
efface ($a, $b);
print "a=$a b=$b\n";
efface (12);
[yann@ulysse 14]$
```

2.7.4 Portée des variables

2.7.4.1 Variables globales

Par défaut en Perl, les variables déclarées dans un bloc d'instructions sont globales. On pourra y accéder depuis n'importe quelle autre partie du script. L'exemple suivant illustre ceci :

```
[yann@ulysse 14]$ cat exemple_variables_1.pl
#! /usr/bin/perl -w

$a = "précédent";

print "Avant : a=$a, b=$b\n";
fonction();
print "Après : a=$a, b=$b\n";

sub fonction
{
        $a="suivant";
        $b="nouveau";
}
[yann@ulysse 14]$
```

Executez ce script. Que remarquez vous ?

2.7.4.2 Variables locales

Deux mots clés, `my` et `local` permettent de définir des variables locales. La différence la plus évidente entre les deux est que `my` définit une variable locale qui n'est accessible que dans le bloc d'instructions auquel elle appartient. Alors qu'avec `local`, la variable sera également visible et modifiable dans les sous fonctions.L'exemple suivant illustre ceci :

```
[yann@ulysse 14]$ cat exemple_variables_2.pl #!
/usr/bin/perl -w
fonction();

sub fonction
{
        my $a = "initiale";
        local $b = "initiale";

        print "fonction() : a=$a, b=$b\n";
        fonction_2();
        print "fonction() : a=$a, b=$b\n";
}

sub fonction_2
{
        $a="modifiée";
        $b = "modifiée";
        print "fonction_2() : a=$a, b=$b\n";
}

[yann@ulysse 14]$
```

Que remarquez vous ?

2.7.5 Référence symbolique de routines

Perl permet d'utiliser les références symboliques sur les noms de fonction. Il autorise aussi l'emploi de références physiques. La notation pour invoquer une routine dont le nom est stocké dans la variable `$nom_fonction` est :

```
&$nom_fonction(argument);
```

L'exemple suivant illustre nos propos :

```
[yann@ulysse 14]$ cat exemple_references.pl
#! /usr/bin/perl -w

my $nom_fonction="factorielle";
```

```
my $resultat=&$nom_fonction(5);

print "$resultat\n";

sub factorielle
{
        my ($val) = @_;
        return 1 if ($val <= 1);
        return $val * factorielle($val -1);
}

[yann@ulysse 14]$
```

Testez ce programme.

2.7.6 Prototypes

Afin de s'assurer qu'une fonction est bien invoquée avec les arguments corrects, il est possible de fournir un prototype dans sa déclaration. Chaque argument est représenté par un caractère qui indique son type.

Caractère	Type d'argument
$	scalaire
@	liste
%	hachage
\$	variable scalaire
\@	variable table
\%	variable table hachage
*	descripteur de fichier
&	sous programme anonyme

Bibliographie

[Agix, 1995] Axis & Agix. *Unix Utilisation : Guide de Formation.* EDITIONS LASER, 1995.

[Blaess, 2002a] CH. Blaess. *Langages de scripts sous Linux.* Eyrolles, 2002.

[Blaess, 2002b] CH. Blaess. *Programmation Système en C sous Linux.* Eyrolles, 2002.

[Champarnaud et Hansel, 1995] J.M. Champarnaud et G. Hansel. *Passeport pour UNIX et C.* Passeport pour l'informatique. Internationnal Thomson Publishing, 1995.

[Charman, 1994] P. Charman. *Unix et X-Window : Guide Pratique.* CÉPADUÈS ÉDITIONS, 1994.

[Poulain, 1994] T. Poulain. Cours unix. 1994.

[Schwartz et Phoenix, 2002] R.L. Schwartz et T. Phoenix. *Introduction à Perl.* O'Reilly, 2002.

[Stoeckel, 1992] Richard Stoeckel. *Filtres et Utilitaires Unix.* ARMAND COLIN, 1992.