Le langage PERL

Alain FORCIOLI, aforcioli@april.org Sat Jan 16 17:49:50 CET 1999

Perl est un langage interprété créé par Larry WALL. Perl est optimisé pour scanner des fichiers, en extraire des informations et sortir des rapports à partir de ces informations. C'est pourquoi son nom signifie "Practical Extraction and Report Language". Ce document est le support de cours d'une conférence sur Perl dispensée par l'association APRIL. Nous espérons qu'il pourra aussi vous servir efficament de manuel d'introduction au langage Perl. Toute nouvelle version de *ce document* <http://www.april.org/Travaux/Doc/Html/perl.html> se trouvera sur le serveur web de l'*APRIL* <http://www.april.org>.

Contents

1 Introduction

Larry WALL affirme que son langage regroupe les meilleures caractéristiques des langages C, sed, awk et sh. Il est proche de la vérité. Toutefois son langage manque de succès notamment à cause de la complexité de l´écriture des programmes.

Néanmoins, à partir de Perl 5.003 (mouture orientée objet, dotée d'un système de modules et d'un jeu d'expressions régulières très étendu) Perl a connu un vif succès auprès des webmasters qui écrivent désormais beaucoup de leur programmes CGI en Perl.

2 Les types de variables de PERL

2.1 Introduction

(Se reporter à la section *perldata* du manuel Perl.)

Perl manipule quatre types de variables:

1. les variables scalaires,
2. les tableaux,
3. les tables de hash,
4. les références.

2.2 La variable scalaire

2.2.1 Description

C'est la variable de base. La variable scalaire se préfixe du $ (comme les variables shell). Elle peut prendre pour valeur un entier, une chaîne de caractères ou une référence.

Remarque: dans un code Perl, le caractère # permet de commenter tout ce qui suit jusqu'au retour chariot.

```
$number = 1;              # un entier
$string = "Il fait beau"; # une chaine de caractères
$reference = \\$number;    # une référence sur l'entier 1
```

2.2.2 Exemples d'opérations sur les variables scalaires

```
$i = "1";       # la chaine "1"
$i ++           # 2
$i = "$i"       # la chaine "2"
```

Perl effectue les conversions implicites des valeurs des variables scalaires pour que les opérations qui les affectent puissent être exécutées.

```
$phrase = "2*2 = 4" . "et sqrt(4) = 2" . ".\n"; # concatenation 1
```

Le caractère . permet de concaténer des chaînes entre elles.

```
$phrase = "2*2 = ", 2*2, " et sqrt(4) = ", sqrt (4), ".\n"; # concatenation 2
```

La virgule permet de concaténer les évaluations des routines présentes sur la ligne d'affectation. Ainsi la variable scalaire `$phrase` contient la chaine `"2+2 = 4 et sqrt(4) = 2."`.

2.3 La table ou liste

2.3.1 Description

Préfixé du caractère `@`, le tableau appelé aussi table ou liste est un ensemble de variables scalaires. Donc une table peut contenir des entiers, des chaînes de caractères, des références et même des tables. Pour définir une table on utilise les parenthèses et on sépare les éléments par une virgule.

```
(a, b, c)               # une table de caractères
@tab = ( 1, 2, 3)       # une table d'entiers
@foo = ( 1, b, $var)    # une table avec des éléments de
                        # types différents
```

Une table est indexée par des entiers. Le premier indice d'une table est 0. Le premier élément de la table `@tab` se nomme `$tab[0]`. Le nombre d'éléments de la table est `$#tab`.

```
@tab = (a, b, c);
$tab[0];                # a
$tab[1];                # b
$#tab;                  # 3
$tab[ $#tab -1 ]        # c
```

2.3.2 Les fonctions utilisant les tables

(Se reporter à la section *perlfunc* du manuel Perl.)

push, pop, shift, unshift Ces fonctions permettent de manipuler la table comme une pile. La fonction `pop` retourne le dernier élément de la table et le supprime. `push` ajoute un élément à la fin d'une table. `shift` retourne le premier élément d'une table et le supprime. `unshift` insere un élément au début d'une table.

Voici quelques exemples d'utilisation de ces fonctions.

```
@tab = ('Larry', 'WALL');
$age = '24';
push(@tab, $age);       # @tab = ('Larry', 'WALL', '24')

print shift(@tab);      # affiche 'Larry' et
                        # @tab = ('WALL', '24')

print pop(@tab);        # affiche '24' et
                        # @tab = ('WALL')
```

sort, reverse, split, join La commande `sort` trie les éléments d'une table et retourne la liste triée.

```
@tab = ('b', 'w', 'a', 't');
@tab = sort @tab;               # @tab = ('a', 'b', 't', 'w')
```

La commande `reverse` retourne la liste des éléments dans l'ordre inversé.

```
@tab = reverse @tab;            # @tab = ('w', 't', 'b', 'a')
```

`split` permet de découper une chaîne de caractères. C'est une fonction puissante car c'est l'utilisateur qui spécifie quel caractère ou quelle expression régulière (voir chapitre sur les expressions régulières) permet de délimiter les éléments à découper. La commande retourne une liste contenant les éléments de la chaîne découpée.

```
@tab = split(/\+/, "1 + 2 + 3");        # @tab vaut (1, 2, 3)
```

Ici le caractère de séparation est `+`. Si on avait voulu découper une chaîne contenant des `+` et des `-` on aurait utilisé l'expression régulière suivante : `/+|-/` qui veut dire "`+` ou `-`".

`join` regroupe les éléments d'une table dans une chaîne de caractères en les séparant par la chaîne de caractères donnée en argument. La chaîne de caractères peut être annotée comme une expression régulière ou entre guillements.

```
print join(/ - /, @tab);        # affiche '1 - 2 - 3';
print join(" - ", @tab);        # idem
```

2.4 La table de hash

2.4.1 Introduction

Une fonction de hash permet de calculer une clé (unique si possible) à partir d'une chaîne de caractères. La clé de valeur entière permet d'indexer une table par la suite. Cette technique est appelée *Hash coding*. Son avantage est qu'elle diminue considérablement le nombre de tentatives lors d'une recherche dans une table.

Une table de hash est une liste dont chaque élément est un couple de la forme `(nom, valeur)`. *nom* est une chaîne de caractères (ex: "`123`", "`coucou`", etc...) dont Perl extrait (par une fonction de hash) une clé. Cette fonction est invisible pour le programmeur.

Donc à l'indice correspondant à la clé de `nom` on trouve l'élément `valeur`. Une table de hash est en général utilisée pour construire une structure de données. Dorénavant, et par abus de langage, je dirais que `nom` est la clé.

Remarque: awk (gawk) fournit également ce mécanisme.

2.4.2 Description

La forme générale d'une table de hash est:

```
%hash = ( cle1, val1, cle2, val2, ..., cleN, valN);

%hash = (nom, WALL, prenom, Larry);
```

L'exemple montre une table avec deux éléments. La valeur associée à la clé `prenom` est `Larry`. De même la valeur associée à la clé `nom` est `WALL`.

Cette notation est maintenant obsolète et peu lisible. Une nouvelle écriture permet de ne pas confondre la table de hash avec une liste.

```
%hash = (
        'nom' => 'Larry',
        'prenom' => 'WALL',
        );

$hash{'nom'}    # Larry
$hash{nom}      # Larry
$hash{prenom}   # WALL
```

On remarque que lorsque la clé est constituée d'un seul mot il n'est pas nécessaire de la mettre entre apostrophes (ou *quotes*).

2.4.3 Fonctions usuelles

Les exemples des fonctions que nous allons décrire seront basés sur la table de hash suivante:

```
%hash = (
        'nom' => 'FORCIOLI',
        'prenom' => 'Alain',
        'age' => '24',
        );
```

keys, values `keys` retourne une liste (table) contenant les clés de la table de hash.

```
@k = keys %hash;        # @k = ('nom', 'prenom', 'age')
```

Remarque: les clés sont des valeurs entières. Supposons que `Alain` et `Bob` soient deux noms dont les clés sont `30` et `20`. On s'aperçoit que l'ordonnancement croissant des clés (`20` puis `30`) ne correspond pas à l'ordonnancement alphabétique des noms (`Alain` puis `Bob`). Aussi nous supposerons que la liste (`nom`, `prenom`, `age`) est la liste des noms (clés) retournée par `keys`.

Sachant que `keys` retourne une liste, il est facile de l'avoir de manière ordonnée avec la fonction `sort`.

```
@k = sort keys %hash;    # @k = ('age', 'nom', 'prenom')
```

`values` à l'inverse de `keys` retourne une liste de toutes les valeurs des clés. L'obtention d'une telle liste ordonnée se fait de cette manière.

```
@v = sort values %hash; # @v = ('24', 'Alain', 'FORCIOLI')
```

each, delete, exists `each` retourne le premier couple de la table de hash sous la forme d'une liste. Le couple est ensuite supprimé de la table de hash.

```
($nom, $val) = each %hash;       # $nom = 'nom'
                                 # $val = 'FORCIOLI'
                                 # keys %hash = ('prenom', 'age')
```

`delete` supprime la valeur associée à la clé donnée en argument.

```
delete $hash{'age'};     # keys %hash = ('prenom')
```

`exists` retourne 1 (vrai) si une valeur existe pour une clé donnée.

```
print "Exists\n" if exists $hash{'prenom'};     # affiche
                                                # 'Exists\n'
```

3 Les variables spéciales

(se reporter à la section *perlvar* du manuel de Perl)

Perl gère un ensemble de variables globales. En voici quelques exemples.

```
$/;     # Le caractère séparateur de ligne
        # '\n' par defaut.
$_;     # le contenu de l'entrée standard ou la valeur de
        # chaque élément d'une table dans une boucle foreach
        # ou while.
@_;     # un tableau contenant les arguments d'une fonction;
%ENV;   # une table de hash qui contient les variables
        # d'environnement du shell.
        # Equivalent a la commande 'env'.
@ARGV;  # une table des arguments du script Perl.
```

4 Les structures de contrôle

Comme tout langage de programmation, Perl dispose de structures de contrôle.

4.1 if, else, elseif

```
if(expr)
{
        block1
} else
{
        block2
}
```

ou

```
if(expr)
{
        block1
} elseif(expr2)
{
        block2
}
```

Dans les deux cas `block1` est exécuté si l'évaluation de *expr* retourne une valeur positive. Dans le premier exemple, `block2` est exécuté dans tous les autres cas. Dans le second, `block2` est évalué seulement si *expr2* retourne une valeur positive.

Plusieurs `elseif` peuvent être enchaênés.

```
if($a == "a")
{
        print "$a\n";
} elseif($a == "b")
{
        print "$a$a\n";
} elseif($a == "c")
{
        print "$a$a$a\n";
} ...
```

4.2 foreach, while

```
foreach (table)
{
        block;
}

while(expr)
{
        block;
}
```

La structure de contrôle `foreach` permet d'executer `block` autant de fois qu'il y a d'éléments dans `table`.

```
@tab = (1, 2, 3, 4, 5);
$i = 0;

foreach (@tab)
{
        print "$i\n";
        $i++;
}
```

L'exemple ci-dessus affiche successivement les valeurs 0, 1, 2, 3 et 4. Nous avons vu dans la section `Les variables speciales` que la variable `$_` prenait la valeur de chaque élément d'une table passée dans une boucle `foreach` ou `while`. Illustrons cette particularité en simplifiant l'exemple précédent.

```
@tab = (1, 2, 3, 4, 5);
foreach (@tab)
{
        print "$_\n";
}
```

Cette fois, l'exemple affiche 1, 2, 3, 4 et 5. `$_` prend successivement chaque valeur des éléments de `tab`.

Encore plus simple :

```
foreach (1 .. 5)
{
        print "$_\n";
}
```

(1 .. 5) est un intervalle en Perl.

```
@tab = (1, 2, 3, 4, 5);
while($i = shift(@tab))
{
        print "$i\n";
}
```

`shift` retourne le premier élément de la table `@tab` et le supprime. Ainsi `$i` prend successivement les valeurs 1, 2, 3, 4 et 5. La boucle s'arrête lorsque $i est vide c'est à dire lorsqu'il n'y a plus d'éléments dans `@tab`.

4.3 Présentation du code

Perl dispose de quelques artifices permettant de rendre plus lisible le code écrit.

```
print $value if $value;
```

Le contenu de $value est affiché si la variable est définie. Cela évite d'avoir

```
if($value)
{
        print $value;
}
```

De même:

```
print $value unless ! $value;
```

Littéralement, on affiche la valeur de `$value` A MOINS QUE la variable ne soit pas définie (`!` est l'opérateur non). Pour être encore mieux compris d'un humain, on aurait pu écrire:

```
print $value unless ! defined $value;
```

De la même manière, on peut aussi utiliser `while`.

```
print "/tmp/lock: exists\n" while -f "/tmp/lock";
```

5 Les fonctions

5.1 Présentation

La structure générale d'une fonction est:

```
sub fonction
{
        # block
}
```

`block` est le corps de la fonction, c'est à dire les actions à exécuter. Chaque action est séparée d'une autre par ';'. Le caractère '#' permet de commenter le code écrit. Tout ce qui suit ce caractère jusqu'au prochain retour à la ligne n'est pas interprété.

5.2 Création d'une fonction

```
sub hello
{
        print "Hello World !\n";
}


hello();        # affiche 'Hello World !\n'
hello;          # idem
\&hello;        # idem
do hello;       # idem
```

L'exemple précédent nous montre comment créer une fonction et comment l'exécuter de quatre manières différentes.

5.3 Valeur de retour d'une fonction

```
sub true
{
        return 1;
}

$t = true;              # $t vaut 1;
$t = true();            # idem
```

La fonction suivante est équivalente:

```
sub true
{
        1;
}

$t = true;      # $t vaut 1
```

Les deux exemples ci-dessus sont équivalents. En fait Perl retourne la valeur de la dernière expression de la fonction exécutée. Cette valeur n'est pas seulement entière. Il est possible de retourner des tables ou des tables de hash.

```
sub foo
{
        ...
        ...
        @tab = (1, 2, 3);
        return @tab;    # ou return (1, 2, 3);
}

sub bar
{
        ...
        ...
        %hash = (
                        'i' => 1,
                        'ii' => 2,
                );

        %hash;
}
```

Losque *return* est manquant, la valeur de la dernière expression est retournée.

5.4 Arguments d'une fonction

Les arguments d'une fonction se présentent sous la forme d'une table. Elle est notée `@_`. Donc:

```
$_[0];          # premier argument.
$_[$#_];        # dernier argument :
                # $#_ représente le nombre d'éléments dans le
                # tableau
```

Bien souvent on connait à l'avance le nombre d'arguments d'une fonction. Aussi on privilégiera l'usage de `@_` à `$_[0]`, etc.

```
sub sum
{
        $sum = 0;
        foreach (@_) # parcours de tous les arguments
        {
                $sum += $_;
        }
        $sum;
}


sum(10, 20, 30, "40");          # retourne 100
```

Rappel: Perl fait les conversions implicitement.

```
sub hello
{
        ($nom, $prenom) = @_;

        print "Hello $nom $prenom\n";
}

hello("FORCIOLI", "Alain");     # affiche 'Hello FORCIOLI Alain\n'
```

Cet usage est préférable car il évite une ligne complexe du genre:

```
($nom, $prenom) = ($_[0], $_[1]);
```

5.5 Variables locales

Il existe deux fonctions qui permettent de rendre locale une variable.

```
$a = "a";
{
        local($a) = "b";        # $a est locale au 'scope'
                                # (par exemple: {...})
        print "$a\n";           # affiche 'b\n'
}

print "$a\n";                   # affiche 'a\n'
```

L'exemple précédent montre que les variables `$a` ont des valeurs différentes. La seconde déclaration (utilisant `local()` pour définir `$a`) rend `$a` visible uniquement à l'intérieur des crochets. Nous avons deux espaces de noms en notre présence. Dans le premier nous avons `$a` valant `a`, dans le second nous avons `$a` égale à `b`.

Le manuel (section *perlfunc*) préconise tout de même l'emploi de `my()`, qui apparemment donne un sens plus fort au terme *locale*. En effet dans un contexte orienté objet ou de module, une variable déclarée avec `local`, est connue de tout le *scope* (*scope* : espace de nommage, compris entre crochets) de module ou de la classe. Tandis qu'une variable déclarée avec `my` est connue uniquement à l'intérieur du scope ou elle est définie.

6 Les expressions régulières de Perl (E.R.)

(Voir la section *perlre* du manuel de Perl)

6.1 Description

Les expressions régulières sont des caractères permettant de coder n'importe quelle entité écrite appartenant à un ensemble (ex: les décimales, les majuscules, les mots commencant par *z* ou une minuscule, etc...). Elles sont largement utilisées par les développeurs UNIX et sont implémentées dans des commandes telles que `awk`, `sed`, `ed`, `vi`. Même le langage C dispose d'une librairie d'expressions régulières.

Le principe de ces caractères est de construire un masque codant l'entité cible. Ensuite chaque séquence de caractères est comparée au masque. S'il y a correspondance, la séquence appartient à l'ensemble de l'entité.

Ce mécanisme permet de faire des remplacements de chaînes de caractères dans un fichier sans savoir ou sont placées les chaînes à remplacer. Il permet d'extraire des informations sans en connaître la position dans un ou plusieurs fichiers. C'est de loin le meilleur outil d'extraction et de traitement d'informations dans un fichier et c'est pourquoi Perl l'utilise.

Illustrons l'usage des expressions en codant l'ensemble de tous les mots à caractères minuscules.

```
[a-z]
```

[ ] permet de définir un ensemble de caractères. Le caractère - signifie que l'ensemble contient la lettre a, b, c, ... ou z. On aurait pu écrire:

```
[abcdefghijklmnopqrstuvwxyz]
```

alain appartient appartient à l'ensemble des mots composés de lettres minuscules. Par contre *Alain* non, car le mot contient `A`.

De manière générale une E.R. est préfixée et suffixée du caractère `/`.

```
/[a-z]/
```

Voici une rapide description de quelques caractères d'expression régulières.

```
.       # n'importe quel caractère alphanumérique et
        # d'espacement.

+       # fait correspondre au moins 1 fois ou plusieurs ce
```

```
        # qui précède

?       # 0 ou au plus 1 caractère de ce qui précède
        # a? permet de faire correspondre '' ou 'a'

*       # 0 ou plusieurs caractères de ce qui précède
        # a* peut faire correspondre '', 'a', 'aa', 'aaa', etc...

[]      # un intervalle

[^]     # un complément

^       # qui commence par

$       # qui finit par

()      # Définit un groupe de caractères réutilisable

|       # ou
```

Perl étend le jeu de caractères des expressions.

```
\s      # n'importe quel caractère d'espacements
\S      # le complément de \s
```

Perl permet de modifier le traitement des chaînes par des options, en passant une ou plusieurs lettres après le / final de l'E.R :

```
i       # les majuscules sont traitées comme des minuscules
m       # traite la chaine de caractères commes si elle était
        # constituée de plusieurs lignes
s       # traite la chaine comme si elle était sur une seule
        # ligne
```

Deux opérations sont principalement utilisées par les expressions régulières. La substitution et le *pattern matching* (la mise en correspondance).

```
s/source/target/options
```

`s` signifie substitution. / est le séparateur d'expression. Si une chaîne lue correspond à *source* elle est remplacée par *target*. *options* agit sur le nombre de substitutions. Si `g` est utilisé alors toutes les chaînes de la ligne lue correspondant à *source* sont remplacées (`g` pour global). `e` permet d'évaluer *target*. Ainsi pour convertir un fichier texte en un fichier ne comportant que des majuscules on aurait le code suivant (fichier `uc.pl`):

```
#! /usr/bin/perl

while(<>)
{
        s/^(.*)$/uc($1)/e;
        $_;
}
```

On peut tester cette opération de la façon suivante:

```
$ uc.pl /etc/passwd
```

Le fichier /etc/passwd s'affiche avec uniquement des caractères majuscules.

Remarque: la première ligne `#! /usr/bin/perl` permet de spécifier au shell quelle commande il doit lancer pour interpréter le contenu du fichier uc.pl. `.pl` est le suffixe standard des fichier scripts Perl.

6.2 Exemple: wc.pl

Le programme wc.pl suivant effectue le meme travail que la commande unix, à savoir comptabiliser le nombre de caractères, de lignes et de mots d'un fichier donné en argument (fichier `wc.pl`):

```
#! /usr/bin/perl

# Retourne le nombre de mots dans une ligne par mot on comprendra tous
# ce qui n'est pas 'espace'.
sub compte_mot
{
    my($ligne) = @_;

    $ligne =~ s/\s+/ /g;
    my(@tab) = split( /\s/, $ligne);
    $#tab;
}

sub compte_ligne
{
    my($ligne) = @_;
    return 1 if $ligne !~ /^\s*$/;
}

sub compte_char
{
    my($ligne) = @_;
    $ligne =~ s/\s//g;
    length($ligne);
}

sub main
{
    my($mots, $lignes, $chars);

    while(<STDIN>)
    {
        $mots  += compte_mot($_);
        $chars += compte_char($_);
        $lignes ++ if compte_ligne($_);
```

```
    }

    print "$ARGV[0]: w = $mots, l = $lignes, c = $chars\n";
}

main;
```

Exemple d'utilisation :

```
        $ wc.pl /etc/passwd
```

Remarque: <STDIN> (equivalent à <>) est le flux de l'entrée standard. <STDOUT> et <STDERR> ceux de la sortie standard et d'erreur.

7 Les références

7.1 Description

La référence Perl peut être vue comme le pointeur du langage C. C'est une variable qui en référence une autre. Pour annoter la référence sur la variable scalaire contenant la valeur 1, nous écrirons:

```
        $i = 1;                    # i est une var. scalaire et vaut 1
        $scalar_ref = \\$i;  # scalar_ref référence i.
```

Pour accéder au contenu pointé par la référence, il faut la déréférencer. Perl utilise l'opérateur -> pour déréférencer une référence.

Pour accéder à la valeur de la variable pointée il faut aussi la déréférencer. De même qu'avec le langage C, une référence peut *pointer* une autre référence et ainsi de suite. Il faudra donc déréférencer autant de fois que nécéssaire pour obtenir la valeur.

Le valeur pointée peut être modifiée en manipulant la référence.

```
        int i = 3; int *ref_i = \&i; *(ref_i) ++;

        printf("i = %d\n", i);

        $i = 3;
        $ref_i = \\$i;
        $ref_i ++;
        print $i, "\n";
```

7.2 Référence sur un tableau

La référence d'une table peut être créée de plusieurs façons:

```
        @tab = (1, 2, 3);
        $ref = \@tab;
```

ou

```
$ref = [1, 2, 3];
```

Pour déréférencer la table, il faut la préfixer du caractère qui désigne la table en Perl : `@`

```
@tab1 = @$ref;
```

Pour parcourir les éléments de la table on peut écrire:

```
foreach (@$ref)
{
        # $_ contient successivement 1, 2 et 3
}
```

On peut directement accéder à un élément contenu dans la table pointée.

```
$ref->[0];      # vaut 1
$ref->[1];      # vaut 2
```

Pour obtenir le nombre d'éléments pointés :

```
$#{@$ref};
 | -----
 |   |
 |    retourne la table et non une référence
 |    (1, 2, 3)
 |
 |
 permet d'obtenir le nombre d'éléments d'une table.
```

L'expression précédente est équivalente à:

```
$#(1, 2, 3);
```

7.3 Référence sur une table de hash

De même qu'avec la table il existe divers moyens de créer une référence sur une table de hash.

```
%hash = (
        'un' => 1,
        'deux' => 2,
        'trois' => 3,
        );

$ref = \%hash;
```

ou

```
$ref = {
        'un' => 1,
        'deux' => 2,
        'trois' => 3,
        };
```

Pour déréférencer la référence il faut utiliser le caractère qui désigne la table de hash : `%`.

```
%hash = %$ref;
```

Le parcours d'une telle référence peut se faire de la manière suivante:

```
foreach $k (sort keys %$ref)
{
        # $k vaut 'deux', 'trois' puis 'un'
}

foreach $v (sort values %$ref)
{
        # $v vaut 1, 2 et 3
}
```

On peut modifier un élément pointé comme ceci:

```
$ref->{'un'} = 'UN';
```

On peut étendre une table de hash référencée.

```
$ref->{'quatre'} = 100;
```

On peut effacer un élément référencé:

```
delete $ref->{'un'};
```

7.4 Référence sur une fonction

Cette particularité est appréciable car elle permet entre autres de faire de manière très simple de la génération de code dynamique.

Une référence sur une fonction s'écrit:

```
$func = sub {
                # code de la fonction
            };
```

Pour utiliser la fonction il faut la préfixer du caractère qui désigne (historiquement) la fonction : `&`.

```
&$func();       # lance la fonction
```

7.5 Extension: structure de données complexes

Le référencement/déréférencement permet à Perl de pouvoir créer des structures de données complexes.

```
$a = [1, 2, 3];

$b = [
        $a,
        {
                'un'=>1,
                'deux'=>2,
        },
     ];
```

`$a` référence la table `(1, 2, 3)`. `$b` est une référence sur une table qui contient deux éléments: `$a` et une référence sur une table de hash `('un', 1, 'deux', 2)`.

```
$b->[0];                # vaut $a
@{$b->[0]};             # vaut (1, 2, 3)
${@{$b->[0]}}[0];       # vaut 1
$b->[0]->[0];           # idem

$b->[1];                # une référence sur une table de hash
$b->[1]->{'un'};        # vaut 1
sort keys %{$b->[1]};   # vaut ('deux', 'un')
```

On imagine très bien les immenses possibilités d'un tel mécanisme. C'est d'ailleurs par ce biais que Perl permet de faire de la programmation orientée objet.

8 Remerciements

Merci à :

Benjamin DRIEU `<mailto:bdrieu@april.org>`

pour la mise en page du document,

Charlie SIERRA `<mailto:Charlie.Sierra@RBT.CARSYS.philips.com>`

pour les diverses corrections apportées au document.

Tony BASSETTE `<mailto:tbassette@april.org>`

pour les diverses remarques et corrections apportées au document.

9 Copyright