perlsyn

Table des matières

1 NAME/NOM

perlsyn - Syntaxe de Perl

2 DESCRIPTION

Un script Perl est constitué d'une suite de déclarations et d'instructions qui sont exécutées de haut en bas. Les boucles, les sous-programmes et d'autres structures de contrôles vous permettent de vous déplacer dans le code.

Perl est un langage à syntaxe libre. Vous pouvez donc le présenter et l'indenter comme bon vous semble. Les espaces ne servent qu'à séparer les éléments syntaxiques contrairement à des langages comme Python où ils font partie intégrante de la syntaxe.

La plupart des éléments syntaxiques de Perl sont **optionnels**. Au lieu de vous obliger à mettre des parenthèses pour chaque appel de fonction ou à déclarer toutes les variables, Perl vous laisse libre de le faire ou non et se débrouille pour comprendre ce que vous voulez. Ceci s'appelle **Fait Ce Que Je Pense** ou **FCQJP** (NdT: en anglais **Do What I Mean** ou **DWIM**). Le programmeur peut donc être **paresseux** et peut coder dans le style qui lui plait.

Perl **emprunte sa syntaxe** et ses concepts à de nombreux langages : awk, sed, C, Bourne Shell, Smalltalk, Lisp et même l'Anglais. D'autres langages ont emprunté à la syntaxe de Perl, en particulier ses extensions aux expressions rationnelles. Donc, si vous avez programmé avec d'autres langages, vous rencontrerez des constructions familières en Perl. Souvent elles fonctionnent de la même manière mais lisez tout de même *perltrap* pour savoir quand elles diffèrent.

2.1 Declarations

Les seules choses que vous devez absolument déclarer en Perl sont les formats de rapport et les sous-programmes (parfois ce n'est même pas nécessaire pour les sous-programmes). Une variable contient la valeur indéfinie (`undef`) jusqu'à ce qu'on lui affecte une valeur, qui est n'importe quoi autre que `undef`. Lorsqu'il est utilisé comme un nombre, `undef` est traité comme `0` ; lorsqu'il est utilisé comme une chaîne, il est traité comme la chaîne vide, `""` ; et lorsqu'il est utilisé comme une référence qui n'a pas été affectée, il est traité comme une erreur. Si vous activez les avertissements, vous serez notifié de l'utilisation d'une valeur non initialisée chaque fois que vous traiterez `undef` comme une chaîne ou un nombre. En tout cas, habituellement. Son utilisation dans un contexte booléen, tel que :

```
my $a;
if ($a) {}
```

ne déclenche pas d'avertissement (parce qu'on teste la véracité et non la présence d'une valeur définie). Les opérateurs tels que `++`, `--`, `+=`, `-=` et `.=` lorsqu'ils agissent sur une valeur indéfinies comme dans :

```
my $a;
$a++;
```

ne déclenchent pas non plus de tels avertissements.

Une déclaration peut être mise partout où une instruction peut trouver place, mais n'a pas d'effet sur l'exécution de la séquence d'instructions principale - les déclarations prennent toutes effet au moment de la compilation. Typiquement, toutes les déclarations sont placées au début ou à la fin du script. Toutefois, si vous utilisez des variables privées de portée lexicales créées avec `my()`, vous devrez vous assurez que la définition de votre format ou de votre sous-programme est à l'intérieur du même bloc que le my si vous voulez pouvoir accéder à ces variables privées.

La déclaration d'un sous-programme permet à un nom de sous-programme d'être utilisé comme s'il était un opérateur de liste à partir de ce point dans le programme. Vous pouvez déclarer un sous-programme sans le définir en disant `sub name`, ainsi :

```
sub myname;
$me = myname $0             or die "can't get myname";
```

Notez que myname() fonctionne comme un opérateur de liste, et non comme un opérateur unaire ; faites donc attention à utiliser `or` au lieu de || dans ce cas. Toutefois, si vous déclariez le sous-programme avec `sub myname ($)`, alors `myname` fonctionnerait comme un opérateur unaire, donc `or` aussi bien que || feraient l'affaire.

Les déclarations de sous-programmes peuvent aussi être chargées à l'aide de l'instruction `require` ou bien à la fois chargées et importées dans votre espace de noms via l'instruction `use`. Voir *perlmod* pour plus de détails.

Une séquence d'instructions peut contenir des déclarations de variables de portée lexicale, mais à part pour déclarer un nom de variable, la déclaration fonctionne comme une instruction ordinaire, et est élaborée à l'intérieur de la séquence d'instructions en tant que telle. Cela signifie qu'elle a à la fois un effet à la compilation et lors de l'exécution.

2.2 Commentaires

Dans une ligne, tout texte commençant par le caractère `"#"` et jusqu'à la fin de la ligne est considéré commme un commentaire et est donc ignoré. L'exception à cette règle concerne les `"#"` présents dans les chaînes de caractères ou dans les expressions rationnelles.

2.3 Instructions simples

Le seul type d'instruction simple est une expression évaluée pour ses effets de bord. Chaque instruction simple doit être terminée par un point-virgule, sauf si c'est la dernière instruction d'un bloc, auquel cas le point-virgule est optionnel. (Nous vous encourageons tout de même à placer ce point-virgule si le bloc prend plus d'une ligne, car vous pourriez éventuellement ajouter une autre ligne). Notez qu'il existe des opérateurs comme `eval {}` et `do {}` qui ont l'air d'instructions composées, mais qui ne le sont pas (ce sont juste des TERMES dans une expression), et qui ont donc besoin d'une terminaison explicite s'ils sont utilisés comme dernier élément d'une instruction.

2.4 Le vrai et le faux

Le nombre 0, les chaînes '0' et '', la liste vide () et undef sont considérés comme faux dans un contexte booléen. Tout autre valeur est considérée comme vraie. La négation d'une valeur vraie par les opérateurs ! ou not retourne une valeur fausse spéciale. Lorsqu'elle est évaluée en tant que chaîne, elle vaut '' mais évaluée en tant que nombre, elle vaut 0.

2.5 Modificateurs d'instruction

Toute instruction simple peut être suivie de façon optionelle par un *UNIQUE* modificateur, juste avant le point-virgule de terminaison (ou la fin du bloc). Les modificateurs possibles sont :

```
if EXPR
unless EXPR
while EXPR
until EXPR
foreach LISTE
```

L'expression EXPR qui suit le modificateur s'appelle la "condition". Sa véracité ou sa fausseté détermine le comportement du modificateur.
if exécute l'instruction une fois *si et seulement si* la condition est vraie. unless fait le contraire : il exécute l'instruction *sauf* si la condition est vraie (autrement dit, si la condition est fausse).

```
print "Les bassets ont de longues oreilles" if length $oreilles >= 10;
aller_dehors() and jouer() unless $il_pleut;
```

Le modificateur foreach est un itérateur : il exécute l'instruction une fois pour chacun des items de LISTE (avec $_ qui est un alias de l'item courant).

```
print "Bonjour $_ !\n" foreach ("tout le monde", "Anne", "Maitresse");
```

while répète l'instruction *tant que* la condition est vraie. until fait le contraire : il répète l'instruction *jusqu'à* ce que la condition soit vraie (ou, autrement dit, tant que la condition est fausse).

```
# Dans les deux cas on compte de 0 à 10
print $i++ while $i <= 10;
print $j++ until $j >  10;
```

Les modificateurs while et until ont la sémantique habituelle des "boucles while" (la condition est évaluée en premier), sauf lorsqu'ils sont appliqués à un do-BLOC (ou à la construction désapprouvée do-SOUS_PROGRAMME), auquel cas le bloc s'exécute une fois avant que la condition ne soit évaluée. Ceci afin que vous puissiez écrire des boucles telles que :

```
do {
    $line = <STDIN>;
    ...
} until $line  eq ".\n";
```

Voir do in *perlfunc*. Notez aussi que l'instruction de contrôle de boucle décrite plus tard *ne* fonctionnera *pas* dans cette construction, car les modificateurs n'utilisent pas de labels de boucle. Désolé. Vous pouvez toujours mettre un autre bloc à l'intérieur (for next) ou autour (for last) pour réaliser ce genre de choses. Pour next, il suffit de doubler les accolades :

```
do {{
    next if $x == $y;
    # faire quelque chose ici
}} until $x++ > $z;
```

Pour last, vous devez faire quelque chose de plus élaboré :

```
LOOP: {
        do {
            last if $x = $y**2;
            # faire quelque chose ici
        } while $x++ <= $z;
}
```

NOTE : le comportement d'une instruction my modifié par un modificateur conditionnel ou une construction de boucle (par exemple my $x if ...) est **indéfini**. La valeur de la variable peut être undef, la valeur précédement affectée ou n'importe quoi d'autre. Ne dépendez pas d'un comportement particulier. Les prochaines versions de perl feront peut-être quelque chose de différents que celle que vous utilisez actuellement.

2.6 Instructions composées

En Perl, une séquence d'instructions qui définit une portée est appelée un bloc. Un bloc est parfois délimité par le fichier qui le contient (dans le cas d'un fichier requis, ou dans celui du programme en entier), et parfois un bloc est délimité par la longueur d'une chaîne (dans le cas d'un eval).

Mais généralement, un bloc est délimité par des accolades. Nous appellerons cette construction syntaxique un BLOC.

Les instructions composées suivantes peuvent être utilisées pour contrôler un flux :

```
if (EXPR) BLOC
if (EXPR) BLOC else BLOC
if (EXPR) BLOC elsif (EXPR) BLOC ... else BLOC
LABEL while (EXPR) BLOC
LABEL while (EXPR) BLOC continue BLOC
LABEL until (EXPR) BLOC
LABEL until (EXPR) BLOC continue BLOC
LABEL for (EXPR; EXPR; EXPR) BLOC
LABEL foreach VAR (LIST) BLOC
LABEL foreach VAR (LIST) BLOCK continue BLOCK
LABEL BLOC continue BLOC
```

Notez que, contrairement au C et au Pascal, tout ceci est défini en termes de BLOCs, et non d'instructions. Ceci veut dire que les accolades sont *requises* - aucune instruction ne doit traîner. Si vous désirez écrire des conditionnelles sans accolades, il existe plusieurs autres façons de le faire. Les exemples suivants font tous la même chose :

```
if (!open(FOO)) { die "Can't open $FOO: $!"; }
die "Can't open $FOO: $!" unless open(FOO);
open(FOO) or die "Can't open $FOO: $!";     # FOO or bust!
open(FOO) ? 'hi mom' : die "Can't open $FOO: $!";
                    # ce dernier est un peu exotique
```

L'instruction `if` est directe. Puisque les BLOCs sont toujours entourés d'accolades, il n'y a jamais d'ambiguïté pour savoir à quel `if` correspond un `else`. Si vous utilisez `unless` à la place de `if`, le sens du test est inversé.

L'instruction `while` exécute le bloc tant que l'expression est vraie (son évaluation ne renvoie pas une chaîne nulle (`""`) ou `0` ou `"0"`). L'instruction `until` exécute le bloc tant que l'expression est fausse. Le LABEL est optionnel, et s'il est présent, il est constitué d'un identifiant suivi de deux points. Le LABEL identifie la boucle pour les instructions de contrôle de boucle `next`, `last`, et `redo`. Si le LABEL est omis, l'instruction de contrôle de boucle se réfère à la boucle incluse dans toutes les autres. Ceci peut amener une recherche dynamique dans votre pile au moment de l'exécution pour trouver le LABEL. Un comportement aussi désespérant provoquera un avertissement si vous utilisez le pragma `use warnings` ou l'option **-w**.

S'il existe un BLOC `continue`, il est toujours exécuté juste avant que la condition ne soit à nouveau évaluée. Ce bloc peut donc être utilisé pour incrémenter une variable de boucle, même lorsque la boucle a été continuée via l'instruction `next`.

2.7 Contrôle de boucle

La commande `next` démarre la prochaine itération de la boucle :

```
LINE: while (<STDIN>) {
    next LINE if /^#/;      # elimine les commentaires
    ...
}
```

La commande `last` sort immédiatement de la boucle en question. Le bloc `continue`, s'il existe, n'est pas exécuté :

```
LINE: while (<STDIN>) {
    last LINE if /^$/;      # sort quand on en a fini avec l'en-tete
    ...
}
```

La commande `redo` redémarre le bloc de la boucle sans réévaluer la condition. Le bloc `continue`, s'il existe, n'est *pas* exécuté. Cette commande est normalement utilisée par les programmes qui veulent se mentir à eux-mêmes au sujet de ce qui vient de leur être fourni en entrée.

Par exemple, lors du traitement d'un fichier comme */etc/termcap*. Si vos lignes en entrée sont susceptibles de se terminer par un antislash pour indiquer leur continuation, vous pouvez vouloir poursuivre et récupérer l'enregistrement suivant.

```
while (<>) {
  chomp;
  if (s/\\$//) {
    $_ .= <>;
    redo unless eof();
  }
  # on traite $_
}
```

qui est le raccourci Perl pour la version plus explicite :

```
LINE: while (defined($line = <ARGV>)) {
  chomp($line);
  if ($line =~ s/\\$//) {
    $line .= <ARGV>;
    redo LINE unless eof(); # pas eof(ARGV)!
  }
  # on traite $line
}
```

Notez que s'il y avait un bloc `continue` dans le code ci-dessus, il ne serait exécuté que pour les lignes rejetées par l'expression rationnelle (puisque redo saut le bloc continue). Un bloc `continue` est souvent utilisé pour réinitialiser les compteurs de lignes ou les recherches de motifs `?pat?` qui ne correspondent qu'une fois :

```
# inspire par :1,$g/fred/s//WILMA/
while (<>) {
  ?(fred)?    && s//WILMA $1 WILMA/;
  ?(barney)?  && s//BETTY $1 BETTY/;
  ?(homer)?   && s//MARGE $1 MARGE/;
} continue {
  print "$ARGV $.: $_";
  close ARGV  if eof();           # reinitialise $.
  reset       if eof();           # reinitialise ?pat?
}
```

Si le mot `while` est remplacé par le mot `until`, le sens du test est inversé, mais la condition est toujours testée avant la première itération.

Les instructions de contrôle de boucle ne fonctionnent pas dans un `if` ou dans un `unless`, puisque ce ne sont pas des boucles. Vous pouvez toutefois doubler les accolades pour qu'elles le deviennent.

```
if (/pattern/) {{
  last if /fred/;
  next if /barney/; # même effet que "last", mais en moins compréhensible
  # mettre quelque chose ici
}}
```

Ceci est du au fait qu'un bloc est considéré comme une boucle qui ne s'exécute qu'une fois. Voir BLOCs de base et instruction switch (§2.10).

La forme `while/if BLOC BLOC`, disponible en Perl 4, ne l'est plus. Remplacez toutes les occurrences de `if BLOC` par `if (do BLOC)`.

2.8 Boucles for

Les boucles `for` de Perl dans le style de C fonctionnent de la même façon que les boucles `while` correspondantes ; cela signifie que ceci :

```
for ($i = 1; $i < 10; $i++) {
  ...
}
```

est la même chose que ça :

```
$i = 1;
while ($i < 10) {
  ...
} continue {
  $i++;
}
```

Il existe une différence mineure : si des variables sont déclarées par `my` dans la section d'initialisation d'un `for`, la portée lexicale de ces variables est limitée à la boucle `for` (le corps de la boucle et sa section de contrôle).

En plus du bouclage classique dans les indices d'un tableau, `for` peut se prêter à de nombreuses autres applications intéressantes. En voici une qui évite le problème que vous rencontrez si vous testez explicitement la fin d'un fichier sur un descripteur de fichier interactif, ce qui donne l'impression que votre programme se gèle.

```
$on_a_tty = -t STDIN && -t STDOUT;
sub prompt { print "yes? " if $on_a_tty }
for ( prompt(); <STDIN>; prompt() ) {
    # faire quelque chose ici
}
```

Utiliser `readline` (ou sous sa forme d'opérateur, `<EXPR>`) comme condition d'un `for` est un raccourci d'écriture pour ce qui suit. Ce comportement est le même pour la condition d'une boucle `while`.

```
for ( prompt(); defined( $_ = <STDIN> ); prompt() ) {
  # faire quelque chose
}
```

2.9 Boucles foreach

La boucle `foreach` itère sur une liste de valeurs normale et fixe la variable VAR à chacune de ces valeurs successivement. Si la variable est précédée du mot-clé `my`, alors elle a une portée limitée du point de vue lexical, et n'est par conséquent visible qu'à l'intérieur de la boucle. Autrement, la variable est implicitement locale à la boucle et reprend sa valeur précédente à la sortie de la boucle. Si la variable était précédemment déclaré par `my`, elle utilise cette variable au lieu de celle qui est globale, mais elle est toujours locale à la boucle.

Le mot-clé `foreach` est en fait un synonyme du mot-clé `for`, vous pouvez donc utiliser `foreach` pour sa lisibilité ou `for` pour sa concision (Ou parce que le Bourne shell vous est plus familier que *csh*, vous rendant l'utilisation de `for` plus naturelle). Si VAR est omis, `$_` est fixée à chaque valeur.

Si un élément de LIST est une lvalue (une valeur modifiable), vous pouvez la modifier en modifiant VAR à l'intérieur de la boucle. À l'inverse, si un élément de LIST n'est pas une lvalue (c'est une constante), toute tentative de modification de cet élément échouera. En d'autres termes, la variable d'index de la boucle `foreach` est un alias implicite de chaque élément de la liste sur laquelle vous bouclez.

Si une partie de LIST est un tableau, `foreach` sera très troublé dans le cas où vous lui ajouteriez ou retireriez des éléments à l'intérieur de la boucle, par exemple à l'aide de `splice`. Ne faites donc pas cela.

`foreach` ne fera probablement pas ce que vous désirez si VAR est une variable liée ou une autre variable spéciale. Ne faites pas cela non plus.

Exemples :

```
for (@ary) { s/foo/bar/ }
```

```
for my $elem (@elements) {
    $elem *= 2;
}

for $count (10,9,8,7,6,5,4,3,2,1,'BOOM') {
    print $count, "\n"; sleep(1);
}

for (1..15) { print "Merry Christmas\n"; }

foreach $item (split(/:[\\\n:]*/, $ENV{TERMCAP})) {
    print "Item: $item\n";
}
```

Voici comment un programmeur C pourrait coder un algorithme en Perl :

```
for (my $i = 0; $i < @ary1; $i++) {
    for (my $j = 0; $j < @ary2; $j++) {
        if ($ary1[$i] > $ary2[$j]) {
            last; # ne peut pas sortir totalement :-(
        }
        $ary1[$i] += $ary2[$j];
    }
    # voici l'endroit ou ce last m'emmene
}
```

Tandis que voici comment un programmeur Perl plus à l'aise avec l'idiome pourrait le faire :

```
OUTER: for my $wid (@ary1) {
INNER:   for my $jet (@ary2) {
            next OUTER if $wid > $jet;
            $wid += $jet;
         }
       }
```

Vous voyez à quel point c'est plus facile ? C'est plus propre, plus sûr, et plus rapide. C'est plus propre parce qu'il y a moins de bruit. C'est plus sûr car si du code est ajouté entre les deux boucles par la suite, le nouveau code ne sera pas exécuté accidentellement. Le `next` itère de façon explicite sur l'autre boucle plutôt que de simplement terminer celle qui est à l'intérieur. Et c'est plus rapide parce que Perl exécute une instruction `foreach` plus rapidement qu'une boucle `for` équivalente.

2.10 BLOCs de base et instruction switch

Un BLOC en lui-même (avec ou sans label) est d'un point de vue sémantique, équivalent à une boucle qui s'exécute une fois. Vous pouvez donc y utilisez n'importe quelle instruction de contrôle de boucle pour en sortir ou le recommencer (Notez que ce n'est *PAS* vrai pour les blocs `eval{}`, `sub{}`, ou `do{}` contrairement à la croyance populaire, qui *NE* comptent *PAS* pour des boucles). Le bloc `continue` est optionnel.

La construction de BLOC est particulièrement élégante pour créer des structures de choix.

```
SWITCH: {
    if (/^abc/) { $abc = 1; last SWITCH; }
    if (/^def/) { $def = 1; last SWITCH; }
    if (/^xyz/) { $xyz = 1; last SWITCH; }
    $nothing = 1;
}
```

Il n'y a pas d'instruction `switch` officielle en Perl, car il existe déjà plusieurs façons d'écrire quelque chose d'équivalent.

En revanche, depuis Perl 5.8 pour obtenir les instructions switch et case, ceux qui le souhaitent peuvent faire appel à l'extension Switch en écrivant :

```
use Switch;
```

À partir de là, les nouvelles instructions seront disponibles. Ce n'est pas aussi rapide que cela pourrait l'être puisque elles ne font pas réellement partie du langage (elles sont réalisées via un filtrage des sources) mais elles sont disponibles et utilisables.

Vous pourriez écrire à la place du bloc précédent :

```
SWITCH: {
    $abc = 1, last SWITCH  if /^abc/;
    $def = 1, last SWITCH  if /^def/;
    $xyz = 1, last SWITCH  if /^xyz/;
    $nothing = 1;
}
```

(Ce n'est pas aussi étrange que cela en a l'air une fois que vous avez réalisé que vous pouvez utiliser des "opérateurs" de contrôle de boucle à l'intérieur d'une expression, c'est juste l'opérateur binaire virgule normal utilisé dans un contexte scalaire. Voir Opérateur virgule in *perlop*.)

ou

```
SWITCH: {
    /^abc/ && do { $abc = 1; last SWITCH; };
    /^def/ && do { $def = 1; last SWITCH; };
    /^xyz/ && do { $xyz = 1; last SWITCH; };
    $nothing = 1;
}
```

ou formaté de façon à avoir un peu plus l'air d'une instruction `switch` "convenable" :

```
SWITCH: {
    /^abc/      && do {
                        $abc = 1;
                        last SWITCH;
                   };

    /^def/      && do {
                        $def = 1;
                        last SWITCH;
                   };

    /^xyz/      && do {
                        $xyz = 1;
                        last SWITCH;
                    };
    $nothing = 1;
}
```

ou

```
SWITCH: {
    /^abc/ and $abc = 1, last SWITCH;
    /^def/ and $def = 1, last SWITCH;
    /^xyz/ and $xyz = 1, last SWITCH;
    $nothing = 1;
}
```

or même, horreur,

```
if (/^abc/)
    { $abc = 1 }
elsif (/^def/)
    { $def = 1 }
elsif (/^xyz/)
    { $xyz = 1 }
else
    { $nothing = 1 }
```

Un idiome courant pour une instruction `switch` est d'utiliser l'aliasing de `foreach` pour effectuer une affectation temporaire de `$_` pour une reconnaissance pratique des cas :

```
SWITCH: for ($where) {
            /In Card Names/     && do { push @flags, '-e'; last; };
            /Anywhere/          && do { push @flags, '-h'; last; };
            /In Rulings/        && do {                    last; };
            die "unknown value for form variable where: `$where'";
        }
```

Une autre approche intéressante de l'instruction switch est de s'arranger pour qu'un bloc `do` renvoie la valeur correcte :

```
$amode = do {
    if     ($flag & O_RDONLY) { "r" }       # XXX : n'est-ce pas 0?
    elsif  ($flag & O_WRONLY) { ($flag & O_APPEND) ? "a" : "w" }
    elsif  ($flag & O_RDWR)   {
        if ($flag & O_CREAT)  { "w+" }
        else                  { ($flag & O_APPEND) ? "a+" : "r+" }
    }
};
```

ou

```
    print do {
        ($flags & O_WRONLY) ? "write-only"          :
        ($flags & O_RDWR)   ? "read-write"          :
                              "read-only";
    };
```

Ou si vous êtes certain que toutes les clauses `&&` sont vraies, vous pouvez utiliser quelque chose comme ceci, qui "switche" sur la valeur de la variable d'environnement `HTTP_USER_AGENT`.

```
#!/usr/bin/perl
# choisir une page du jargon file selon le browser
$dir = 'http://www.wins.uva.nl/~mes/jargon';
for ($ENV{HTTP_USER_AGENT}) {
    $page  =    /Mac/            && 'm/Macintrash.html'
             || /Win(dows )?NT/  && 'e/evilandrude.html'
             || /Win|MSIE|WebTV/ && 'm/MicroslothWindows.html'
             || /Linux/          && 'l/Linux.html'
             || /HP-UX/          && 'h/HP-SUX.html'
             || /SunOS/          && 's/ScumOS.html'
             ||                     'a/AppendixB.html';
}
print "Location: $dir/$page\015\012\015\012";
```

Ce type d'instruction switch ne fonctionne que lorsque vous savez que les clauses `&&` seront vraies. Si vous ne le savez pas, l'exemple précédent utilisant `?:` devrait être utilisé.

Vous pourriez aussi envisager d'écrire un hachage de références de sous-programmes au lieu de synthétiser une instruction `switch`.

2.11 Goto

Bien que cela ne soit pas destiné aux âmes sensibles, Perl supporte une instruction `goto`. Il en existe trois formes : `goto`-LABEL, `goto`-EXPR, et `goto`-&NAME. Un LABEL de boucle n'est pas en vérité une cible valide pour un `goto` ; c'est juste le nom de la boucle.

La forme `goto`-LABEL trouve l'instruction marquée par LABEL et reprend l'exécution à cet endroit. Elle ne peut pas être utilisée pour aller dans une structure qui nécessite une initialisation, comme un sous-programme ou une boucle `foreach`. Elle ne peut pas non plus être utilisée pour aller dans une structure très optimisée. Elle peut être employée pour aller presque n'importe où ailleurs à l'intérieur de la portée dynamique, y compris hors des sous-programmes, mais il est habituellement préférable d'utiliser une autre construction comme `last` ou `die`. L'auteur de Perl n'a jamais ressenti le besoin d'utiliser cette forme de `goto` (en Perl, à vrai dire - C est une toute autre question).

La forme `goto`-EXPR attend un nom de label, dont la portée sera résolue dynamiquement. Ceci permet des `goto`s calculés à la mode de FORTRAN, mais ce n'est pas nécessairement recommandé si vous optimisez la maintenance du code :

```
goto(("FOO", "BAR", "GLARCH")[$i]);
```

La forme `goto`-&NAME est hautement magique, et substitue au sous-programme en cours d'exécution un appel au sous-programme nommé. C'est utilisé par les sous-programmes `AUTOLOAD()` qui veulent charger une autre routine et prétendre que cette autre routine a été appelée à leur place (sauf que toute modification de `@_` dans le sous-programme en cours est propagée à l'autre routine). Après le `goto`, même `caller()` ne pourra pas dire que cette routine n'a pas été appelée en premier.

Dans presque tous les cas similaires, une bien, bien meilleure idée est d'utiliser les mécanismes de contrôle de flux structurés comme `next`, `last`, ou `redo` au lieu de s'en remettre à un `goto`. Pour certaines applications, la paire `eval{}` - die() pour le traitement des exceptions peut aussi être une approche prudente.

2.12 POD : documentation intégrée

Perl dispose d'un mécanisme pour mélanger de la documentation avec le code source. Lorsqu'il attend le début d'une nouvelle instruction, si le compilateur rencontre une ligne commençant par un signe égal et un mot, comme ceci

```
=head1 Here There Be Pods!
```

Alors ce texte et tout ce qui suit jusqu'à et y compris une ligne commençant par `=cut` sera ignoré. Le format du texte en faisant partie est décrit dans *perlpod*.

Ceci vous permet de mélanger librement votre code source et votre documentation, comme dans

```
=item snazzle($)

La fonction snazzle() se comportera de la facon la plus
spectaculaire que vous pouvez imaginer, y compris la pyrotechnie
cybernetique.

=cut retour au compilateur, nuff of this pod stuff!

sub snazzle($) {
    my $thingie = shift;
    .........
}
```

Notez que les traducteurs pod ne devraient traiter que les paragraphes débutant par une directive pod (cela rend leur analyse plus simple), tandis que le compilateur sait en réalité chercher des séquences pod même au milieu d'un paragraphe. Cela signifie que le bout de code secret qui suit sera ignoré à la fois par le compilateur et les traducteurs.

```
$a=3;
=truc secret
 warn "Neither POD nor CODE!?"
=cut back
print "got $a\n";
```

Vous ne devriez probablement pas vous reposer sur le fait que le `warn()` sera ignoré pour toujours. Les traducteurs pod ne sont pas tous bien élevés de ce point de vue, et le compilateur deviendra peut-être plus regardant.

On peut aussi utiliser des directives pod pour mettre rapidement une partie de code en commentaire.

2.13 Bons Vieux Commentaires (Non !)

Perl peut traiter des directives de ligne, à la manière du préprocesseur C. Avec cela, on peut contrôler l'idée que Perl se fait des noms de fichiers et des numéros de ligne dans les messages d'erreur ou dans les avertissements (en particulier pour les chaînes traitées par `eval()`). La syntaxe de ce mécanisme est la même que pour pour la plupart des préprocesseurs C : elle reconnaît l'expression régulière :

```
# example: '# line 42 "new_filename.plx"'
/^\#   \s*
  line \s+ (\d+)   \s*
  (?:\s("?)([^"]+)\2)? \s*
$/x
```

avec `$1` uqi est le numéro de ligne pour la ligne suivante et `$3` qui est le nom optionnel du fichier (avec ou sans guillemets).

Voici quelques exemples que vous devriez pouvoir taper dans votre interpréteur de commandes :

```
% perl
# line 200 "bzzzt"
# the `#' on the previous line must be the first char on line
die 'foo';
__END__
foo at bzzzt line 201.

% perl
# line 200 "bzzzt"
eval qq[\n#line 2001 ""\ndie 'foo']; print $@;
__END__
foo at - line 2001.

% perl
eval qq[\n#line 200 "foo bar"\ndie 'foo']; print $@;
__END__
foo at foo bar line 200.

% perl
# line 345 "goop"
eval "\n#line " . __LINE__ . ' "' . __FILE__ ."\"\ndie 'foo'";
print $@;
__END__
foo at goop line 345.
```

3 TRADUCTION

3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez http://perl.enstimac.fr/.

3.2 Traducteur

Traduction initiale : Roland Trique <*roland.trique@free.fr*>. Mise à jour : Paul Gaborit <paul.gaborit at enstimac.fr>.

3.3 RELECTURE

Régis Julié <*Regis.Julie@cetelem.fr*>, Etienne Gauthier <*egauthie@capgemini.fr*>

4 À propos de ce document

Ce document est la traduction française du document orginal distribué avec perl. Vous pouvez retrouver l'ensemble de la documentation française Perl (éventuellement mise à jour) en consultant l'URL <http://perl.enstimac.fr/>.

Ce document PDF a été produit Paul Gaborit. Si vous utilisez la version PDF de cette documentation (ou une version papier issue de la version PDF) pour tout autre usage qu'un usage personnel, je vous serai reconnaissant de m'en informer par un petit message <mailto:Paul.Gaborit@enstimac.fr>.

Si vous avez des remarques concernant ce document, en premier lieu, contactez la traducteur (vous devriez trouver son adresse électronique dans la rubrique TRADUCTION) et expliquez-lui gentiment vos remarques ou critiques. Il devrait normalement vous répondre et prendre en compte votre avis. En l'absence de réponse, vous pouvez éventuellement me contacter.

Vous pouvez aussi participer à l'effort de traduction de la documentation Perl. Toutes les bonnes volontés sont les bienvenues. Vous devriez trouver tous les renseignements nécessaires en consultant l'URL ci-dessus.