perlsub

Table des matières

1 NAME/NOM

perlsub - Les sous-programmes (ou subroutines) de Perl

2 SYNOPSIS

Pour déclarer des sous-programmes :

```
sub NAME;                    # Une déclaration "en avant".
sub NAME(PROTO);             # idem, mais avec des prototypes
sub NAME : ATTRS;            # avec attributs
sub NAME(PROTO) : ATTRS;     # avec attributs et prototypes

sub NAME BLOCK               # Une déclaration et une définition.
sub NAME(PROTO) BLOCK        # idem, mais avec des prototypes
sub NAME : ATTRS BLOCK       # avec attributs
sub NAME(PROTO) : ATTRS BLOCK # avec attributs et prototypes
```

Pour définir un sous-programme anonyme lors de l'exécution :

```
$subref = sub BLOCK;                 # pas de prototype
$subref = sub (PROTO) BLOCK;         # avec prototype
$subref = sub : ATTRS BLOCK;         # avec attributs
$subref = sub (PROTO) : ATTRS BLOCK; # avec proto et attributs
```

Pour importer des sous-programmes :

```
use MODULE qw(NAME1 NAME2 NAME3);
```

Pour appeler des sous-programmes :

```
NAME(LIST);    # & est optionnel avec les parenthèses.
NAME LIST;     # Les parenthèses sont optionnelles si le
               # sous-programme est prédéclaré ou importé.
&NAME(LIST);   # Court-circuite les prototypes.
&NAME;         # Rend la @_ courante visible par le
               # sous-programme appelé.
```

3 DESCRIPTION

Comme beaucoup de langages, Perl fournit des sous-programmes définis par l'utilisateur. Ils peuvent se trouver n'importe où dans le programme principal. Ils peuvent être chargés depuis d'autres fichiers via les mots-clés `do`, `require` ou `use`, ou être générés à la volée en utilisant `eval` ou des sous-programmes anonymes. Vous pouvez même appeler une fonction indirectement en utilisant une variable contenant son nom ou une référence à du CODE.

Le modèle de Perl pour l'appel de fonction et le renvoi de valeurs est simple : tous les paramètres sont passés aux fonctions comme une simple liste de scalaires, et toutes les fonctions renvoient de la même manière à leur appelant une simple liste de scalaires. Tout tableau ou hachage dans ces listes d'appel et de retour s'effondreront, perdant leur identité – mais vous pouvez toujours utiliser le passage par référence à la place pour éviter cela. Les listes d'appel ou de retour peuvent contenir autant d'éléments scalaires que vous le désirez (une fonction sans instruction de retour explicite est souvent appelée un sous-programme, mais il n'y a vraiment pas de différence du point de vue de Perl).

Tous les arguments passés à la routine se retrouvent dans le tableau `@_`. Ainsi, si vous appelez une fonction avec deux arguments, ceux-ci seront stockés dans `$_[0]` et `$_[1]`. Le tableau `@_` est un tableau local, mais ses éléments sont des alias pour les véritables paramètres scalaires. En particulier, si un élément `$_[0]` est mis à jour, l'argument correspondant est mis à jour (ou bien une erreur se produit s'il n'est pas modifiable). Si un argument est un élément de tableau ou de hachage qui n'existait pas lors de l'appel de la fonction, cet élément est créé seulement lorsque (et si) il est modifié ou si on y fait référence (certaines versions précédentes de Perl créaient l'élément qu'il soit ou non affecté). L'affectation à la totalité du tableau `@_` retire les alias, et ne met à jour aucun argument.

Une instruction `return` peut être utilisée pour sortir d'un sous-programme, en spécifiant éventuellement une valeur de retour, qui sera évaluée dans le contexte approprié (liste, scalaire ou vide) selon le contexte d'appel du sous-programme. Si vous ne spécifiez aucune valeur de retour, le sous-programme retourne une liste vide dans un contexte de liste, la valeur indéfinie dans un contexte scalaire et rien du tout dans un contexte vide. Si vous retournez un ou plusieurs aggrégats (tableau ou table de hachage), ils seront tous applatis en une seule grande liste.

Si aucun `return` n'est trouvé et si la dernière instruction est une expression, sa valeur est retournée. Si la dernière instruction est une structure de contrôle de boucle comme `foreach` ou `while`, la valeur retournée est non spécifiée. Un sous-programme vide retourne une liste vide.

Perl n'a pas de paramètres formels nommés. En pratique tout ce que vous avez à faire est d'affecter à une liste `my()` les contenant. Les variables que vous utilisez dans la fonction et qui ne sont pas déclarées comme privées sont des variables globales. Pour des détails sanglants sur la création de variables privées, voir Variables privées via my() (§3.1) et Valeurs temporaires via local() (§3.3). Pour créer des environnement protégés pour un ensemble de fonctions dans un paquetage séparé (et probablement un fichier séparé), voir Paquetages in *perlmod*.

Exemple :

```
sub max {
    my $max = shift(@_);
    foreach $foo (@_) {
        $max = $foo if $max < $foo;
    }
    return $max;
}
$bestday = max($mon,$tue,$wed,$thu,$fri);
```

Exemple :

```
# obtient une ligne, en y combinant les lignes la continuant qui
# débutent par un blanc

sub get_line {
    $thisline = $lookahead;  # Variables globales !!
    LINE: while (defined($lookahead = <STDIN>)) {
        if ($lookahead =~ /^[ \t]/) {
            $thisline .= $lookahead;
        }
        else {
            last LINE;
        }
    }
    return $thisline;
}

$lookahead = <STDIN>;       # obtient la première ligne
while (defined($line = get_line()) {
    ...
}
```

Affectez à une liste de variables privées pour nommer vos arguments :

```
sub maybeset {
    my($key, $value) = @_;
    $Foo{$key} = $value unless $Foo{$key};
}
```

Puisque l'affectation copie les valeurs, ceci a aussi pour effet de transformer l'appel par référence en appel par valeur. Autrement, une fonction est libre de faire des modifications de `@_` sur place et de changer les valeurs de son appelant.

```
upcase_in($v1, $v2);  # ceci change $v1 et $v2
sub upcase_in {
    for (@_) { tr/a-z/A-Z/ }
}
```

Vous n'avez pas le droit de modifier les constantes de cette façon, bien sûr. Si un argument était en vérité un littéral et que vous essayiez de le changer, vous lèveriez une exception (probablement fatale). Par exemple, ceci ne fonctionnera pas :

```
upcase_in("frederick");
```

Ce serait bien plus sûr si la fonction `upcase_in()` était écrite de façon à renvoyer une copie de ses paramètres au lieu de les changer sur place :

```
($v3, $v4) = upcase($v1, $v2);  # cela ne change pas $v1 et $v2
sub upcase {
    return unless defined wantarray;  # contexte vide, ne fait rien
    my @parms = @_;
    for (@parms) { tr/a-z/A-Z/ }
    return wantarray ? @parms : $parms[0];
}
```

Notez comment cette fonction (non prototypée) ne se soucie pas qu'on lui ait passé de vrais scalaires ou des tableaux. Perl voit tous les arguments comme une grosse et longue liste plate de paramètres dans `@_`. C'est un domaine dans lequel brille le style simple de passage d'arguments de Perl. La fonction `upcase()` fonctionnerait parfaitement bien sans avoir à changer la définition de `upcase()` même si nous lui donnions des choses telles que ceci :

```
@newlist   = upcase(@list1, @list2);
@newlist   = upcase( split /:/, $var );
```

Ne vous laissez toutefois pas tenter de faire :

```
(@a, @b)   = upcase(@list1, @list2);
```

Tout comme la liste plate de paramètres entrante, la liste de retour est aussi aplatie. Donc tout ce que vous êtes parvenu à faire ici est de tout stocker dans `@a` et de vider `@b`. Voir Passage par référence pour savoir comment faire autrement.

Un sous-programme peut être appelé en utilisant un préfixe `&` explicite. Le `&` est optionnel en Perl moderne, tout comme le sont les parenthèses si le sous-programme a été prédéclaré. Le `&` n'est *pas* optionnel lorsque l'on nomme juste le sous-programme, comme lorsqu'il est utilisé en tant qu'argument de defined() ou de undef(). Il n'est pas non plus optionnel lorsque vous voulez faire un appel indirect de sous-programme avec le nom d'un sous-programme ou une référence utilisant les constructions `&$subref()` ou `&{$subref}()`, bien que la notation `$subref->()` résoud ce problème. Voir *perlref* pour plus de détails à ce sujet.

Les sous-programmes peuvent être appelés de façon récursive. Si un sous-programme est appelé en utilisant la forme `&`, la liste d'arguments est optionnelle, et si elle est omise, aucun tableau `@_` n'est mis en place pour le sous-programme : le tableau `@_` au moment de l'appel est visible à la place dans le sous-programme. C'est un mécanisme d'efficacité que les nouveaux utilisateurs pourraient préférer éviter.

```
&foo(1,2,3);        # passe trois arguments
foo(1,2,3);         # idem

foo();              # passe une liste nulle
&foo();             # idem

&foo;               # foo() obtient les arguments courants, comme foo(@_) !!
foo;                # comme foo() SI le sous-programme foo est
                    # prédéclaré, sinon "foo"
```

Non seulement la forme `&` rend la liste d'arguments optionnelle, mais elle désactive aussi toute vérification de prototype sur les arguments que vous fournissez. C'est en partie pour des raisons historiques, et en partie pour avoir une façon pratique de tricher si vous savez ce que vous êtes en train de faire. Voir *Prototypes* ci-dessous.

Les sous-programme dont les noms sont entièrement en majuscules sont réservées pour le noyau de Perl, tout comme les modules dont les noms sont entièrement en minuscules. Une fonction entièrement en majuscules est une convention informelle signifiant qu'elle sera appelée indirectement par le système d'exécution lui-même, habituellement en réponse à un événement. Les sous-programmes qui font des choses spéciales prédéfinies sont entre autres `AUTOLOAD`, `CLONE` et `DESTROY` – plus toutes les fonctions mentionnées dans *perltie* et *PerlIO::via*.

Les sous-programmes `BEGIN`, `CHECK`, `INIT` et `END` sont plutôt des blocs spéciaux, de code, nommés, qui peuvent apparaître plusieurs fois dans un paquetage et qu'il vous est **impossible** d'appeler explicitement. Voir BEGIN, CHECK, INIT et END in *perlmod*.

3.1 Variables privées via my()

Synopsis:

```
my $foo;            # déclare $foo locale lexicalement
my (@wid, %get);    # déclare une liste de variables locale
my $foo = "flurp";  # déclare $foo lexical, et l'initialise
my @oof = @bar;     # déclare @oof lexical, et l'initialise
my $x : Foo = $y;   # similaire, avec application d'un attribut
```

AVERTISSEMENT : l'usage de listes d'attributs sur les déclarations `my` est expérimental. La sémantique précise et l'interface associée sont sujettes à changement. Voir *attributes* et *Attribute::Handlers*.

L'opérateur `my` déclare que les variables listées doivent être confinées lexicalement au bloc où elles se trouvent, dans la conditionnelle (`if/unless/elsif/else`), la boucle (`for/foreach/while/until/continue`), le sous-programme, l'`eval`, ou le fichier exécuté, requis ou utilisé via `do/require/use`. Si plus d'une valeur est listée, la liste doit être placée entre parenthèses. Tous les éléments listés doivent être des lvalues légales. Seuls les identifiants alphanumériques peuvent avoir une portée lexicale – pour l'instant1, les identifiants magiques prédéfinis comme `$/` peuvent, à la place, être localisés par `local`.

Contrairement aux variables dynamiques créées par l'opérateur `local`, les variables lexicales déclarées par `my` sont totalement cachées du monde extérieur, y compris de tout sous-programme appelé. Cela est vrai s'il s'agit du même sous-programme rappelé depuis lui-même ou depuis un autre endroit – chaque appel obtient sa propre copie.

Ceci ne veut pas dire qu'une variable `my()` déclarée dans une portée lexicale englobante statiquement serait invisible. Seules les portées dynamiques sont rétrécies. Par exemple, la fonction `bumpx()` ci-dessous a accès à la variable lexicale `$x` car le `my` et le `sub` se sont produits dans la même portée, probablement la portée du fichier.

```
    my $x = 10;
    sub bumpx { $x++ }
```

Un `eval()`, toutefois, peut voir les variables lexicales de la portée dans laquelle il est évalué, tant que les noms ne sont pas cachés par des déclarations à l'intérieur de l'`eval()` lui-même. Voir *perlref*.

La liste de paramètres de my() peut être affectée si on le désire, ce qui vous permet d'initialiser vos variables (si aucune valeur initiale n'est donnée à une variable particulière, celle-ci est créé avec la valeur indéfinie). Ceci est utilisé couramment pour nommer les paramètres d'entrée d'un sous-programme. Exemples :

```
    $arg = "fred";        # variable "globale"
    $n = cube_root(27);
    print "$arg thinks the root is $n\n";
 fred thinks the root is 3

    sub cube_root {
        my $arg = shift;  # le nom n'importe pas
        $arg **= 1/3;
        return $arg;
    }
```

Le `my` est simplement un modificateur sur quelque chose que vous pourriez affecter. Donc lorsque vous affectez les variable dans sa liste d'arguments, le `my` ne change pas le fait que ces variables soient vues comme un scalaire ou un tableau. Donc

```
    my ($foo) = <STDIN>;                # FAUX ?
    my @FOO = <STDIN>;
```

les deux fournissent un contexte de liste à la partie droite, tandis que

```
    my $foo = <STDIN>;
```

fournit un contexte scalaire. Mais ce qui suit ne déclare qu'une seule variable :

```
    my $foo, $bar = 1;                  # FAUX
```

Cela a le même effet que

```
    my $foo;
    $bar = 1;
```

La variable déclarée n'est pas introduite (n'est pas visible) avant la fin de l'instruction courante. Ainsi,

```
    my $x = $x;
```

peut être utilisé pour initialiser une nouvelle variable $x avec la valeur de l'ancienne $x, et l'expression

```
my $x = 123 and $x == 123
```

est fausse à moins que l'ancienne variable $x ait la valeur `123`.

Les portées lexicales de structures de contrôle ne sont pas précisément liées aux accolades qui délimitent le bloc qu'elles contrôlent ; les expressions de contrôle font aussi partie de cette portée. Ainsi, dans la boucle

```
while (my $line =<>) {
    $line = lc $line;
} continue {
    print $line;
}
```

la portée de $line s'étend à partir de sa déclaration et sur tout le reste de la boucle (y compris la clause `continue`), mais pas au-delà. De façon similaire, dans la conditionnelle

```
if ((my $answer = <STDIN>) =~ /^yes$/i) {
    user_agrees();
} elsif ($answer =~ /^no$/i) {
    user_disagrees();
} else {
    chomp $answer;
    die "'$answer' is neither 'yes' nor 'no'";
}
```

la portée de $answer s'étend à partir de sa déclaration et sur tout le reste de la conditionnelle, y compris les clauses `elsif` et `else`, s'il y en a, mais pas au-delà. Voir Instructions simples in *perlsyn* pour plus d'informations sur la portée des variables dans des instructions avec modificateurs.

La boucle `foreach` a pour défaut de donner une portée dynamiquement a sa variable d'index à la manière de `local`. Toutefois, si la variable d'index est préfixée par le mot-clé `my`, ou s'il existe déjà un lexical ayant ce nom dans la porté, alors un nouveau lexical est créé à la place. Ainsi dans la boucle

```
for my $i (1, 2, 3) {
    some_function();
}
```

la portée de $i s'étend jusqu'à la fin de la boucle, rendant la valeur de $i inaccessible dans `some_function()`.

Certains utilisateurs pourraient désirer encourager l'usage de variables à portée lexicale. Comme aide pour intercepter les utilisations implicites de variables de paquetage, qui sont toujours globales, si vous dites

```
use strict 'vars';
```

alors toute variable mentionnée à partir de là jusqu'à la fin du bloc doit soit se référer à une variable lexicale, soit être prédéclarée via `our` ou `use vars`, soit encore être totalement qualifiée avec le nom du paquetage. Autrement, une erreur de compilation se produit. Un bloc interne pourrait contrer ceci par `no strict 'vars'`.

Un `my` a un effet à la fois à la compilation et lors de l'exécution. Lors de la compilation, le compilateur le remarque. La principale utilité de ceci est de faire taire `use strict 'vars'`, mais c'est aussi essentiel pour la génération de fermetures telle que détaillée dans *perlref*. L'initialisation effective est toutefois retardée jusqu'à l'exécution, de façon à ce qu'elle soit exécutée au moment approprié, par exemple à chaque fois qu'une boucle traversée.

Les variables déclarées avec `my` ne font pas partie d'un paquetage et ne sont par conséquent jamais totalement qualifiée avec le nom du paquetage. En particulier, vous n'avez pas le droit d'essayer de rendre lexicale une variable de paquetage (ou toute autre variable globale) :

```
my $pack::var;      # ERREUR ! Syntaxe Illégale
my $_;              # aussi illégal (pour le moment)
```

En fait, une variable dynamique (aussi connue sous le nom de variable de paquetage ou variable globale) est toujours accessible en utilisant la notation totalement qualifiée `::` même lorsqu'un lexical de même nom est lui aussi visible :

```
package main;
local $x = 10;
my    $x = 20;
print "$x and $::x\n";
```

Ceci affichera `20` et `10`.

Vous pouvez déclarer les variables `my` dans la portée extrême d'un fichier pour cacher tous ces identifants du monde extérieur à ce fichier. Cela est similaire en esprit aux variables statiques de C quand elles sont utilisées au niveau du fichier. Faire cela avec un sous-programme requiert l'usage d'une fermeture (une fonction anonyme qui accède aux lexicaux qui l'entourent). Si vous voulez créer un sous-programme privé qui ne peut pas être appelé en-dehors de ce bloc, il peut déclarer une variable lexicale conenant une référence anonyme de sous-programme :

```
my $secret_version = '1.001-beta';
my $secret_sub = sub { print $secret_version };
&$secret_sub();
```

Tant que la référence n'est jamais renvoyée par aucune fonction du module, aucun module extérieur ne peut voir le sous-programme, car son nom n'est dans la table de symboles d'aucun paquetage. Souvenez-vous qu'il n'est jamais *VRAIMENT* appelé `$some_pack::secret_version` ou quoi que ce soit ; c'est juste $secret_version, non qualifié et non qualifiable.

Ceci ne fonctionne pas avec les méthodes d'objets, toutefois ; toutes les méthodes d'objets doivent se trouver dans la table de symboles d'un paquetage quelconque pour être trouvées. Voir Modèles de Fonctions in *perlref* pour une méthode permettant de contourner ceci.

3.2 Variables privées persistantes

Le fait qu'une variable lexicale ait une portée lexicale (aussi appelée statique) égale au bloc qui la contient, à l'`eval`, ou au FICHIER `do`, ne veut pas dire que cela fonctionne à l'intérieur d'une fonction comme une variable statique en C. Cela marche plutôt comme une variable auto de C, mais avec un nettoyage de la mémoire (un rammasse-miettes) implicite.

Contrairement aux variables locales en C ou en C++, les variables lexicales de Perl ne sont pas nécessairement recyclées juste parce qu'on est sorti de leur portée. Si quelque chose de plus permanent est toujours conscient du lexical, il restera. Tant que quelque chose d'autre fait référence au lexical, ce lexical ne sera pas libéré – ce qui est comme il se doit. Vous ne voudriez pas que la mémoire soit libérée tant que vous n'avez pas fini de l'utiliser, ou gardée lorsque vous en avez terminé. Le ramasse-miette automatique s'occupe de cela pour vous.

Ceci veut dire que vous pouvez passer en retour ou sauver des références à des variables lexicales, tandis que retourner un pointeur sur un auto en C est une grave erreur. Cela nous donne aussi un moyen de simuler les variables statiques de fonctions C. Voici un mécanisme donnant à une fonction des variables privées de portée lexicale ayant une durée de vie statique. Si vous désirez créer quelque chose ressemblant aux vatiables statiques de C, entourez juste toute la fonction d'un niveau de bloc supplémentaire, et mettez la variable statique hors de la fonction mais dans le bloc.

```
{
    my $secret_val = 0;
    sub gimme_another {
        return ++$secret_val;
    }
}
# $secret_val devient désormais impossible à atteindre pour le
# monde extérieur, mais garde sa valeur entre deux appels à
# gimme_another
```

Si cette fonction est chargée par `require` ou `use` depuis un fichier séparé, alors cela marchera probablement très bien. Si tout se trouve dans le programme principal, vous devrez vous arranger pour que le `my` soit exécuté tôt, soit en mettant tout le bloc avant votre programme principal, ou, de préférence, en plaçant simplement un `BEGIN` devant lui pour vous assurer qu'il soit exécutée avant que votre programme ne démarre :

```
BEGIN {
    my $secret_val = 0;
    sub gimme_another {
        return ++$secret_val;
    }
}
```

Voir BEGIN, CHECK, INIT et END in *perlmod* pour en savoir plus sur la mises en oeuvre des blocks spéciaux `BEGIN`, `CHECK`, `INIT` et `END`.

S'ils sont déclarés au niveau de la plus grande portée (celle du fichier), alors les lexicaux fonctionnent un peu comme les variables statiques de fichier en C. Ils sont disponibles pour toutes les fonctions déclarées en dessous d'eux dans le même fichier, mais sont inaccessibles hors du fichier. Cette stratégie est parfois utilisé dans les modules pour créer des variables privées que la totalité du module peut voir.

3.3 Valeurs temporaires via local()

AVERTISSEMENT : en général, vous devriez utiliser `my` au lieu de `local`, parce que c'est plus rapide et plus sûr. Les exceptions à cela incluent les variables globales dont le nom est un caractère de ponctuation, les handles de fichier globaux et les formats globaux et la manipulation directe de la table de symboles de Perl. On utilise `local` lorsqu'on souhaite que la valeur courante d'une variable soit visible par les sous-programmes appelés.

Synopsis :

```
# local-isation de valeurs

local $foo;               # rend $foo dynamiquement locale
local (@wid, %get);       # rend locales toutes les variables d'une liste
local $foo = "flurp";     # rend $foo dynamique, et l'initialise
local @oof = @bar;        # rend @oof dynamique, et l'initialise

local $hash{key} = "val"; # rend local la valeur lié à cette clé
local ($cond ? $v1 : $v2); # la plupart des types de lvalues
                          # acceptent la local-isation

# local-isation de symboles

local *FH;                # rend locales $FH, @FH, %FH, &FH...
local *merlyn = *randal;  # maintenant $merlyn est vraiment $randal,
                          # @merlyn est vraiment @randal, etc.
local *merlyn = 'randal'; # IDEM : 'randal' est promu *randal
local *merlyn = \$randal; # alias juste $merlyn, pas @merlyn etc
```

Un `local()` modifie ses variables listées pour qu'elle soient "locales" au bloc courant, à l'`eval`, ou au FICHIER `do` – et à *tout sous-programme appelé depuis l'intérieur de ce bloc*. Un `local()` donne juste des valeurs temporaires aux variables globales (au paquetage). Il ne crée *pas* une variable locale. Ceci est connu sous le nom de portée dynamique. La portée lexicale est créé par `my`, qui fonctionne plus comme les déclarations auto de C.

Différents types de lvalues peuvent être localisés : les éléments ou les tranches de tableaux ou de tables de hachage, éventuellement de manière conditionnelle (dans la mesure où le résultat est bien localisable), et les références symboliques. Comme pour de simples variables, cela crée de nouvelles valeurs avec une portée dynamqiue.

Si plus d'une variable est fournie à `local`, elles doivent être placées entre parenthèses. Cet opérateur fonctionne en sauvegardant les valeurs courantes de ces variables dans sa liste d'arguments sur une pile cachée et les restaure à la sortie du bloc, du sous-programme ou de l'eval. Ceci veut dire que les sous-programmes appelés peuvent aussi référencer les variables locales, mais pas les globales. La liste d'arguments peut être affectée si l'on veut, ce qui vous permet d'initialiser vos variables locales (si aucun initialiseur n'est donné pour une variable locale particulière, elle est créée avec la valeur indéfinie).

Puisque `local` est une commande réalisée lors de la phase d'exécution, elle est donc exécutée chaque fois que l'on traverse une boucle. Par conséquent il est toujours plus efficace de localiser vos variables en dehors de la boucle.

3.3.1 Note grammatical sur local()

Un `local` est simplement un modificateur d'une expression donnant une lvalue. Lorsque vous affectez une variable `local`isée, le `local` ne change pas selon que sa liste est vue comme un scalaire ou un tableau. Donc

```
local($foo) = <STDIN>;
local @FOO = <STDIN>;
```

fournissent tous les deux un contexte de liste à la partie droite, tandis que

```
local $foo = <STDIN>;
```

fournit un contexte scalaire.

3.3.2 Localisation des variables spéciales

Si vous localisez une variable spéciale, vous lui donnez une nouvelle valeur mais sans lui faire perdre ses fonctionnalités magiques. Cela signifie que tous les effets secondaires de cette magie ont lieu avec cette nouvelle valeur.
Cela permet le fonctionnement d'un code tel que celui-ci :

```
# Lit tout le contenu de FILE dans $slurp
{ local $/ = undef; $slurp = <FILE>; }
```

Notez, en revanche, que cela empêche la localisation de certaines variables. Par exemple, l'instruction suivante déclenchera, dans perl 5.9.0, un appel à 'die' avec l'erreur *Modification of a read-only value attempted* puisque $1 est une variable magique en lecture seule :

```
local $1 = 2;
```

De manière similaire mais moins évidente, le code suivant déclenchera un 'die' en perl 5.9.0 :

```
sub f { local $_ = "foo"; print }
for ($1) {
  # now $_ is aliased to $1, thus is magic and readonly
  f();
}
```

Voir la section suivante pour savoir comment palier cela.
ATTENTION : la localisation de tableaux ou de tables de hachage liés (par tie()) ne fonctionne pas comme cela. Cela sera corrigé dans une prochaine version de Perl. En attendant, évitez de localiser des tableaux ou des tables de hachage liés (la localistion de éléments individuels fonctionnent). Voir Localisation de tableaux et de tables de hachage liés (par tie()) in *perl58delta* pour plus de détails.

3.3.3 Localisation de globs

La construction

```
local $name;
```

crée une nouvelle entrée dans la table de symbole associé au glob `name` dans le paquetage courant. Cela signifie que toutes les variables attachées à cette entrée ($name, @name, %name, &name et le handle de fichier `name`) sont dynamiquement réinitialisées.
En particulier, si vous voulez une nouvelle valeur pour le scalaire par défaut $_, en évitant le problème potentiel évoqué plus haut lorsque $_ était attaché à une valeur magique. vous devriez utiliser `local *_` au lieu de `local $_`.

3.3.4 Localisation des éléments d'un type composé

Une note sur `local()` et les types composés est nécessaire. Quelque chose comme `local(%foo)` fonctionne en plaçant temporairement un hachage tout neuf dans la table des symboles. L'ancien hachage est mis de côté, mais est caché "derrière" le nouveau.
Ceci signifie que l'ancienne variable est complètement invisible via la table des symboles (i.e. l'entrée de hachage dans le typeglob `*foo`) pendant toute la durée de la portée dynamique dans laquelle le `local()` a été rencontré. Cela a pour effet de permettre d'occulter temporairement toute magie sur les types composés. Par exemple, ce qui suit altèrera brièvement un hachage lié à une autre implémentation :

```
tie %ahash, 'APackage';
[...]
{
   local %ahash;
   tie %ahash, 'BPackage';
   [.. le code appelé verra %ahash lié à 'BPackage'..]
   {
      local %ahash;
      [..%ahash est une hachage normal (non lié) ici..]
   }
}
[..%ahash revient à sa nature liée initiale..]
```

Autre exemple, une implémentation personnelle de %ENV pourrait avoir l'air de ceci :

```
{
    local %ENV;
    tie %ENV, 'MyOwnEnv';
    [..faites vos propres manipulations fantaisistes de %ENV ici..]
}
[..comportement normal de %ENV ici..]
```

Il vaut aussi la peine de prendre un moment pour expliquer ce qui se passe lorsque vous localisez un membre d'un type composé (i.e. un élément de tableau ou de hachage). Dans ce cas, l'élément est localisé *par son nom*. Cela veut dire que lorsque la portée du local() se termine, la valeur sauvée sera restaurée dans l'élément du hachage dont la clé a été nommée dans le local(), ou dans l'élément du tableau dont l'index a été nommé dans le local(). Si cet élément a été effacé pendant que le local() faisait effet (e.g. par un delete() dans un hachage ou un shift() d'un tableau), il reprendra vie, étendant potentiellement un tableau et remplissant les éléments intermédiaires avec undef. Par exemple, si vous dites

```
%hash = ( 'This' => 'is', 'a' => 'test' );
@ary  = ( 0..5 );
{
     local($ary[5]) = 6;
     local($hash{'a'}) = 'drill';
     while (my $e = pop(@ary)) {
         print "$e . . .\n";
         last unless $e > 3;
     }
     if (@ary) {
         $hash{'only a'} = 'test';
         delete $hash{'a'};
     }
}
print join(' ', map { "$_ $hash{$_}" } sort keys %hash),".\n";
print "The array has ",scalar(@ary)," elements: ",
      join(', ', map { defined $_ ? $_ : 'undef' } @ary),"\n";
```

Perl affichera

```
6 . . .
4 . . .
3 . . .
This is a test only a test.
The array has 6 elements: 0, 1, 2, undef, undef, 5
```

Le comportement de local() sur des membres inexistants de types composites est sujet à changements à l'avenir.

3.4 Lvalue et sous-programme

AVERTISSEMENT : les sous-programmes en partie gauche (en lvalue) sont encore expérimentaux et leur implémentation est sujette à changements dans les futures versions de Perl.

Il est possible de retourner une valeur modifiable depuis un sous-programme. Pour ce faire, vous devez déclarer que le sous-programme retourne une lvalue.

```
my $val;
sub canmod : lvalue {
  # return $val; cela ne fonction pas, ne dites pas "return"
  $val;
}

sub nomod {
  $val;
}
```

```
canmod() = 5;   # modifie $val
nomod()  = 5;   # ERREUR
```

Le contexte scalaire ou de liste pour le sous-programme et la partie droite de l'affectation est déterminé comme si l'appel au sous-programme était remplacé par un scalaire. Par exemple, si l'on considère :

```
data(2,3) = get_data(3,4);
```

Les deux sous-programmes sont appelées ici dans un contexte scalaire, tandis que dans :

```
(data(2,3)) = get_data(3,4);
```

et dans :

```
(data(2),data(3)) = get_data(3,4);
```

tous les sous-programmes sont appelés dans un contexte de liste.

Les sous-programmes lvalue sont expérimentaux

Cela semble pratique mais il y a plusieurs raisons pour rester circonspect.

Vous ne devez pas utiliser le mot-clé return et vous devez passez la valeur de sortie avant qu'elle soit hors de portée (voir le commentaire dans l'exemple plus haut). Ce n'est pas habituellement un problème mais cela empêche tout de même un return explicite dans une boucle qui est pourtant parfois bien pratique.

Cela ne respecte pas l'encapsulation. Un mutateur normal vérifie la valeur fournie avant de l'affecter à l'attribut qu'il protège. Un sous-programme lvalue ne permet pas cela. Considérez le code suivant :

```
my $some_array_ref = [];  # protégé par un mutateur ??

sub set_arr {             # mutateur normal
  my $val = shift;
  die("expected array, you supplied ", ref $val)
    unless ref $val eq 'ARRAY';
  $some_array_ref = $val;
}
sub set_arr_lv : lvalue { # mutateur par lvalue
  $some_array_ref;
}

# set_arr_lv ne peut pas empêcher cela !
set_arr_lv() = { a => 1 };
```

3.5 Passage d'entrées de table de symbole (typeglobs)

AVERTISSEMENT : le mécanisme décrit dans cette section était à l'origine la seule façon de simuler un passage par référence dans les anciennes versions de Perl. Même si cela fonctionne toujours très bien dans les version modernes, le nouveau mécanisme de référencement est généralement plus facile à utiliser. Voir plus bas.

Parfois, vous ne voulez pas passer la valeur d'un tableau à un sous-programme mais plutôt son nom, pour que le sous-programme puisse modifier sa copie globale plutôt que de travailler sur une copie locale. En perl, vous pouvez vous référer à tous les objets d'un nom particulier en préfixant ce nom par une étoile : `*foo`. Cela est souvent connu sous le nom de "typeglob", car l'étoile devant peut être conçue comme un caractère générique correspondant à tous les drôles de caractères préfixant les variables et les sous-programmes et le reste.

Lorsqu'il est évalué, le typeglob produit une valeur scalaire qui représente tous les objets portant ce nom, y compris tous les handles de fichiers, les formats, ou les sous-programmes. Quand on l'affecte, le nom mentionné se réfère alors à la valeur `*` qui lui a été affectée, quelle qu'elle soit. Exemple :

```
sub doubleary {
    local(*someary) = @_;
    foreach $elem (@someary) {
        $elem *= 2;
    }
}
doubleary(*foo);
doubleary(*bar);
```

Les scalaires sont déjà passés par référence, vous pouvez donc modifier les arguments scalaires sans utiliser ce mécanisme en vous référant explicitement à `$_[0]`, etc. Vous pouvez modifier tous les éléments d'un tableau en passant tous les éléments comme des scalaires, mais vous devez utiliser le mécanisme `*` (ou le mécanisme de référencement équivalent) pour effectuer des `push`, des `pop`, ou changer la taille d'un tableau. Il sera certainement plus rapide de passer le typeglob (ou la référence).

Même si vous ne voulez pas modifier un tableau, ce mécanisme est utile pour passer des tableaux multiples dans une seule LIST, car normalement le mécanisme LIST fusionnera toutes les valeurs de tableaux et vous ne pourrez plus en extraire les tableaux individuels. Pour plus de détails sur les typeglobs, voir Typeglobs et handles de fichiers in *perldata*.

3.6 Quand faut-il encore utiliser local()

Malgré l'existence de `my`, il y a encore trois endroits où l'opérateur `local` brille toujours. En fait, en ces trois endroits, vous *devez* utiliser `local` à la place de `my`.

1. Vous avez besoin de donner une valeur temporaire à une variable globale, en particulier $_.
   Les variables globales, comme `@ARGV` ou les variables de ponctuation, doivent être `local`isées avec `local()`. Ce bloc lit dans */etc/motd*, et le découpe en morceaux, séparés par des lignes de signes égal, qui sont placés dans `@Fields`.

   ```
   {
       local @ARGV = ("/etc/motd");
       local $/ = undef;
       local $_ = <>;
       @Fields = split /^\s*=+\s*$/;
   }
   ```

   Il est important en particulier de `local`iser $_ dans toute routine qui l'affecte. Surveillez les affectations implicites dans les conditionnelles `while`.
2. Vous avez besoin de créer un handle de fichier ou de répertoire local, ou une fonction locale.
   Une fonction ayant besoin de son propre handle de fichier doit utiliser `local()` sur le typeglob complet. Ceci peut être utilisé pour créer de nouvelles entrées dans la table des symboles :

   ```
   sub ioqueue {
       local  (*READER, *WRITER);    # pas my !
       pipe    (READER,  WRITER)     or die "pipe: $!";
       return (*READER, *WRITER);
   }
   ($head, $tail) = ioqueue();
   ```

   Voir le module Symbol pour une façon de créer des entrées de table de symboles anonymes.
   Puisque l'affectation d'une référence à un typeglob crée un alias, ceci peut être utilisé pour créer ce qui est effectivement une fonction locale, ou au moins un alias local.

   ```
   {
       local *grow = \&shrink; # seulement jusqu'à la fin de
                               # l'existence de ce bloc
       grow();                 # appelle en fait shrink()
       move();                 # si move() grandit, il rétrécit
                               # aussi
   }
   grow();                     # récupère le vrai grow()
   ```

   Voir Modèles de fonctions in *perlref* pour plus de détails sur la manipulation des fonctions par leur nom de cette manière.
3. Vous voulez changer temporairement un seul élément d'un tableau ou d'un hachage.
   Vous pouvez `local`iser juste un élément d'un aggrégat. Habituellement, cela est fait dynamiquement :

   ```
   {
       local $SIG{INT} = 'IGNORE';
       funct();                            # impossible à interrompre
   }
   # la possibilité d'interrompre est restaurée ici
   ```

   Mais cela fonctionne aussi sur les aggrégats déclarés lexicalement. Avant la version 5.005, cette opération pouvait parfois mal se conduire.

3.7 Passage par référence

Si vous voulez passer plus d'un tableau ou d'un hachage à une fonction – ou les renvoyer depuis elle – de façon qu'ils gardent leur intégrité, vous allez devoir alors utiliser un passage par référence explicite. Avant de faire cela, vous avez besoin de comprendre les références telles qu'elles sont détaillées dans *perlref*. Cette section n'aura peut-être pas beaucoup de sens pour vous sans cela.

Voici quelques exemples simples. Tout d'abord, passons plusieurs tableaux à une fonction puis faisons-lui faire des `pop` sur eux, et retourner une nouvelle liste de tous leurs derniers éléments :

```
@tailings = popmany ( \@a, \@b, \@c, \@d );

sub popmany {
    my $aref;
    my @retlist = ();
    foreach $aref ( @_ ) {
        push @retlist, pop @$aref;
    }
    return @retlist;
}
```

Voici comment vous pourriez écrire une fonction retournant une liste des clés apparaissant dans tous les hachages qu'on lui passe :

```
@common = inter( \%foo, \%bar, \%joe );
sub inter {
    my ($k, $href, %seen); # locals
    foreach $href (@_) {
        while ( $k = each %$href ) {
            $seen{$k}++;
        }
    }
    return grep { $seen{$_} == @_ } keys %seen;
}
```

Jusque là, nous utilisons juste le mécanisme normal de retour d'une liste. Que se passe-t-il si vous voulez passer ou retourner un hachage ? Eh bien, si vous n'utilisez que l'un d'entre eux, ou si vous ne voyez pas d'inconvénient à ce qu'ils soient concaténés, alors la convention d'appel normale est valable, même si elle coûte un peu cher.

C'est ici que les gens commencent à avoir des problèmes :

```
    (@a, @b) = func(@c, @d);
ou
    (%a, %b) = func(%c, %d);
```

Cette syntaxe ne fonctionnera tout simplement pas. Elle définit juste `@a` ou `%a` et vide `@b` ou `%b`. De plus, la fonction n'a pas obtenu deux tableaux ou hachages séparés : elle a eu une longue liste dans `@_`, comme toujours.

Si vous pouvez vous arranger pour que tout le monde règle cela par des références, cela fait du code plus propre, même s'il n'est pas très joli à regarder (phrase louche, NDT). Voici une fonction qui prend deux références de tableau comme arguments et renvoit les deux éléments de tableau dans l'ordre du nombre d'éléments qu'ils contiennent :

```
($aref, $bref) = func(\@c, \@d);
print "@$aref has more than @$bref\n";
sub func {
    my ($cref, $dref) = @_;
    if (@$cref > @$dref) {
        return ($cref, $dref);
    } else {
        return ($dref, $cref);
    }
}
```

Il s'avère en fait que vous pouvez aussi faire ceci :

```
(*a, *b) = func(\@c, \@d);
print "@a has more than @b\n";
sub func {
    local (*c, *d) = @_;
    if (@c > @d) {
        return (\@c, \@d);
    } else {
        return (\@d, \@c);
    }
}
```

Ici nous utilisons les typeglobs pour faire un aliasing de la table des symboles. C'est un peu subtile, toutefois, et en plus cela ne fonctionnera pas si vous utilisez des variables `my`, puisque seules les variables globales (et les `local`es en fait) sont dans la table des symboles.

Si vous passez des handles de fichiers, vous pouvez habituellement juste utiliser le typeglob tout seul, comme `*STDOUT`, mais les références de typeglobs fonctionnent aussi. Par exemple :

```
splutter(\*STDOUT);
sub splutter {
    my $fh = shift;
    print $fh "her um well a hmmm\n";
}

$rec = get_rec(\*STDIN);
sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}
```

Si vous comptez générer de nouveau handles de fichiers, vous pourriez faire ceci. Faites attention de renvoyer juste le *FH tout seul, et pas sa référence.

```
sub openit {
    my $path = shift;
    local *FH;
    return open (FH, $path) ? *FH : undef;
}
```

3.8 Prototypes

Perl supporte une sorte de vérification des arguments très limitée lors de la compilation, en utilisant le prototypage de fonction. Si vous déclarez

```
sub mypush (\@@)
```

alors `mypush()` prendra ses arguments exactement comme `push()` le fait. La declaration de la fonction doit être visible au moment de la compilation. Le prototype affecte seulement l'interprétation des appels nouveau style de la fonction, où le nouveau style est défini comme n'utilisant pas le caractère `&`. En d'autres termes, si vous l'appelez comme une fonction intégrée, alors elle se comportera comme une fonction intégrée. Si vous l'appelez comme un sous-programme à l'ancienne mode, alors elle se comportera comme un sous-programme à l'ancienne. La conséquence naturelle de cette règle est que les prototypes n'ont pas d'influence sur les références de sous-programmes comme `\&foo` ou sur les appels de sous-programmes indirects comme `&{$subref}` ou `$subref->()`.

Les appels de méthode ne sont pas non plus influencés par les prototypes, car la fonction qui doit être appelée est indéterminée au moment de la compilation, puisque le code exact appelé dépend de l'héritage.

Puisque l'intention de cette caractéristique est principalement de vous laisser définir des sous-programmes qui fonctionnent comme des commandes intégrées, voici des prototypes pour quelques autres fonctions qui se compilent presque exactement comme les fonctions intégrées correspondantes.

```
Déclarées en tant que    Appelé comme

sub mylink ($$)          mylink $old, $new
sub myvec ($$$)          myvec $var, $offset, 1
sub myindex ($$;$)       myindex &getstring, "substr"
sub mysyswrite ($$$;$)   mysyswrite $buf, 0, length($buf) - $off, $off
sub myreverse (@)        myreverse $a, $b, $c
sub myjoin ($@)          myjoin ":", $a, $b, $c
sub mypop (\@)           mypop @array
sub mysplice (\@$$@)     mysplice @array, @array, 0, @pushme
sub mykeys (\%)          mykeys %{$hashref}
sub myopen (*;$)         myopen HANDLE, $name
sub mypipe (**)          mypipe READHANDLE, WRITEHANDLE
sub mygrep (&@)          mygrep { /foo/ } $a, $b, $c
sub myrand ($)           myrand 42
sub mytime ()            mytime
```

Tout caractère de prototype précédé d'une barre oblique inverse représente un véritable argument qui doit absolument commencer par ce caractère. La valeur passée comme partie de `@_` sera une référence au véritable argument passé dans l'appel du sous-programme, obtenue en appliquant \ à cet argument.

Vous pouvez aussi transformer d'un seul coup en référence à plusieurs types d'argument en utilisant la notation `\[]` :

```
sub myref (\[$@%&*])
```

qui autorisera les appels suivant à myref() :

```
myref $var
myref @array
myref %hash
myref &sub
myref *glob
```

et le premier argument de myref() sera une référence à un scalaire, un tableau, une table de hachage, du code ou à un glob.

Les caractères de prototype non précédés d'une barre oblique inverse ont une signification spéciale. Tout `@` ou `%` sans barre oblique inverse mange les arguments restants, et force un contexte de liste. Un argument représenté par `$` force un contexte scalaire. Un `&` requiert un sous-programme anonyme, qui, s'il est passé comme premier argument, ne requiert pas le mot-clé "`sub`" ni une virgule après lui. Un `*` permet au sous-programme d'accepter un bareword, une constante, une expression scalaire, un typeglob, ou une référence à un typeglob à cet emplacement. La valeur sera disponible pour le sous-programme soit comme un simple scalaire, soit (dans les deux derniers cas) comme une référence au typeglob. Si vous désirez toujours convertir de tels arguments en référence de typeglob, utilisez Symbol::qualify_to_ref() ainsi :

```
use Symbol 'qualify_to_ref';

sub foo (*) {
    my $fh = qualify_to_ref(shift, caller);
    ...
}
```

Un point-virgule sépare les arguments obligatoires des arguments optionnels. Il est redondant devant `@` ou `%`, qui engloutissent tout le reste.

Notez comment les trois derniers exemples dans la table ci-dessus sont traités spécialement par l'analyseur. `mygrep()` est compilé comme un véritable opérateur de liste, `myrand()` comme un véritable opérateur unaire avec une précédence unaire identique à celle de `rand()`, et `mytime()` est vraiment sans arguments, tout comme `time()`. C'est-à-dire que si vous dites

```
mytime +2;
```

Vous obtiendrez `mytime() + 2`, et pas `mytime(2)`, qui est la façon dont cela serait analysé sans prototype.

Ce qui est intéressant avec `&` est que vous pouvez générer une nouvelle syntaxe grâce à lui, pourvu qu'il soit en position initiale :

```
sub try (&@) {
    my($try,$catch) = @_;
    eval { &$try };
    if ($@) {
        local $_ = $@;
        &$catch;
    }
}
sub catch (&) { $_[0] }

try {
    die "phooey";
} catch {
    /phooey/ and print "unphooey\n";
};
```

Cela affiche "unphooey" (Oui, il y a toujours des problèmes non résolus à propos de la visibilité de @_. J'ignore cette question pour le moment (Mais notez que si nous donnons une portée lexicale à @_, ces sous-programmes anonymes peuvent agir comme des fermetures... (Mince, est-ce que ceci sonne un peu lispien ? (Peu importe))).

Et voici une réimplémentation de l'opérateur grep de Perl :

```
sub mygrep (&@) {
    my $code = shift;
    my @result;
    foreach $_ (@_) {
        push(@result, $_) if &$code;
    }
    @result;
}
```

Certaines personnes préféreraient des prototypes pleinement alphanumériques. Les caractères alphanumériques ont intentionnellement été laissés hors des prototypes expressément dans le but d'ajouter un jour futur des paremètres formels nommés. Le principal objectif du mécanisme actuel est de laisser les programmeurs de modules fournir de meilleurs diagnostics à leurs utilisateurs. Larry estime que la notation est bien compréhensible pour les programmeurs Perl, et que cela n'interfèrera pas notablement avec le coeur du module, ni ne le rendra plus difficile à lire. Le bruit sur la ligne est visuellement encapsulé dans une petite pilule facile à avaler.

Si vous essayez d'utiliser des séquences alphanumériques dans un prototype, vous aurez droit à un avertissement optionnel - "Illegal character in prototype...". Malheureusement, les anciennes versions de Perl autorisaient l'utilisation de ce type de prototype tant que le début était un prototype valide. L'avertissement actuel pourrait devenir une erreur fatale dans une future version de Perl lorsque la mejorité du code erroné aura été corrigé.

Il est probablement meilleur de prototyper les nouvelles fonctions, et de ne pas prototyper rétrospectivement les anciennes. C'est parce que vous devez être particulièrement précautionneux avec les exigences silencieuses de la différence entre les contextes de liste et les contextes scalaires. Par exemple, si vous décidez qu'une fonction devrait prendre juste un paramètre, comme ceci :

```
sub func ($) {
    my $n = shift;
    print "you gave me $n\n";
}
```

et que quelqu'un l'a appelée avec un tableau ou une expression retournant une liste :

```
func(@foo);
func( split /:/ );
```

Alors vous venez juste de fournir un scalar automatique devant leurs arguments, ce qui peut être plus que surprenant. L'ancien @foo qui avait l'habitude de ne contenir qu'une chose n'entre plus. À la place, func() se voit maintenant passer un 1 ; c'est-à-dire le nombre d'éléments dans @foo. Et le split se voit appelé dans un contexte scalaire et commence à gribouiller dans votre liste de paramètres @_. Aïe !

Tout ceci est très puissant, bien sûr, et devrait être utilisé uniquement avec modération pour rendre le monde meilleur.

3.9 Fonctions constantes

Les fonctions ayant un prototype égal à `()` sont des candidates potentielles à l'insertion en ligne (inlining, NDT). Si le résultat après optimisation et repliement des constantes est soit une constante soir un scalaire de portée lexicale n'ayant pas d'autre référence, alors il sera utilisé à la place des appels à la fonction faits sans `&`. Les appels réalisés en utilisant `&` ne sont jamais insérés en ligne (Voir *constant.pm* pour une façon aisée de déclarer la plupart des constantes).

Les fonctions suivantes seraient toutes insérées en ligne :

```
sub pi ()           { 3.14159 }             # Pas exact, mais proche.
sub PI ()           { 4 * atan2 1, 1 }      # Aussi exact qu'il
                                            # puisse être, et
                                            # inséré en ligne aussi !

sub ST_DEV ()       { 0 }
sub ST_INO ()       { 1 }

sub FLAG_FOO ()     { 1 << 8 }
sub FLAG_BAR ()     { 1 << 9 }
sub FLAG_MASK ()    { FLAG_FOO | FLAG_BAR }

sub OPT_BAZ ()      { not (0x1B58 & FLAG_MASK) }

sub N () { int(OPT_BAZ) / 3 }

sub FOO_SET () { 1 if FLAG_MASK & FLAG_FOO }
```

En revanche, celles qui suivent ne pourront être insérées en ligne ; parce qu'elles contiennent des choses non constantes dans leur propre portée.

```
sub foo_set () { if (FLAG_MASK & FLAG_FOO) { 1 } }

sub baz_val () {
    if (OPT_BAZ) {
        return 23;
    }
    else {
        return 42;
    }
}
```

Si vous redéfinissez un sous-programmes qui pouvait être inséré en ligne, vous obtiendrez un avertissement formel (vous pouvez utiliser cet avertissement pour déterminer si un sous-programme particulier est considéré comme constant ou pas). L'avertissement est considéré comme suffisamment sévère pour ne pas être optionnel, car les invocations précédemment compilées de la fonction utiliseront toujours l'ancienne valeur de cette fonction. Si vous avez besoin de pouvoir redéfinir le sous-programme, vous devez vous assurer qu'il n'est pas inséré en ligne, soit en abandonnant le prototype `()` (ce qui change la sémantique de l'appel, donc prenez garde), soit en contrecarrant le mécanisme d'insertion d'une autre façon, comme par exemple

```
sub not_inlined () {
    23 if $];
}
```

3.10 Surcharges des fonctions prédéfinies

Beaucoup de fonctions prédéfinies peuvent être surchargées, même si cela ne devrait être essayé qu'occasionnellement et pour une bonne raison. Typiquement, cela pourrait être réalisé par un paquetage essayant d'émuler une fonctionnalité prédéfinie manquante sur un système non Unix.

La surcharge ne peut être réalisée qu'en important le nom depuis un module lors de la compilation – la prédéclaration ordinaire n'est pas suffisante. Toutefois, le pragma `use subs` vous laisse, dans les faits, prédéclarer les sous-programmes via la syntaxe d'importation, et ces noms peuvent ensuite surcharger ceux qui sont prédéfinis :

```
use subs 'chdir', 'chroot', 'chmod', 'chown';
chdir $somewhere;
sub chdir { ... }
```

Pour se référer sans ambiguïté à la forme prédéfinie, on peut faire précéder le nom prédéfini par le qualificateur de paquetage spéciale `CORE::`. Par exemple, le fait de dire `CORE::open()` se réfère toujours à la fonction prédéfinie `open()`, même si le paquetage courant a importé d'ailleurs un quelconque autre sous-programme appelé `&open()`. Même si cela a l'air d'un appel de fonction normal, ce n'en est pas un : vous ne pouvez pas en créer de référence, telle que celles que l'incorrect `\&CORE::open` pourrait en produire.

Les modules de bibliothèque ne devraient en général pas exporter de noms prédéfinis comme `open` ou `chdir` comme partie de leur liste `@EXPORT` par défaut, car ceux-ci pourraient s'infiltrer dans l'espace de noms de quelqu'un d'autre et changer la sémantique de façon inattendue. À la place, si le module ajoute ce nom à `@EXPORT_OK`, alors il est possible pour un utilisateur d'importer le nom explicitement, mais pas implicitement. C'est-à-dire qu'il pourrait dire

```
use Module 'open';
```

et cela importerait la surcharge `open`. Mais s'il disait

```
use Module;
```

il obtiendrait les importations par défaut sans les surcharges.

Le mécanisme précédent de surcharge des éléments prédéfinis est restreint, assez délibérément, au paquetage qui réclame l'importation. Il existe une seconde méthode parfois applicable lorsque vous désirez surcharger partout une fonction prédéfinie, sans tenir compte des frontières entre espaces de noms. Cela s'obtient en important un sous-programme dans l'espace de noms spécial `CORE::GLOBAL::`. Voici un exemple qui remplace assez effrontément l'opérateur `glob` par quelque chose qui comprend les expressions rationnelles.

```
package REGlob;
require Exporter;
@ISA = 'Exporter';
@EXPORT_OK = 'glob';

sub import {
    my $pkg = shift;
    return unless @_;
    my $sym = shift;
    my $where = ($sym =~ s/^GLOBAL_// ? 'CORE::GLOBAL' : caller(0));
    $pkg->export($where, $sym, @_);
}

sub glob {
    my $pat = shift;
    my @got;
    local *D;
    if (opendir D, '.') {
        @got = grep /$pat/, readdir D;
        closedir D;
    }
    return @got;
}
1;
```

Et voici comment on pourrait en (ab)user :

```
#use REGlob 'GLOBAL_glob';      # surcharge glob() dans TOUS les
                                # espaces de noms
package Foo;
use REGlob 'glob';              # surcharge glob() dans Foo::
                                # seulement
print for <^[a-z_]+\.pm\$>;     # montre tous les modules pragmatiques
```

Le commentaire initial montre un exemple artificiel, voire même dangereux. En surchargeant `glob` globalement, vous forceriez le nouveau (et subversif) comportement de l'opérateur `glob` pour *tous* les espaces de noms, sans la complète coopération des modules qui possèdent ces espaces de noms, et sans qu'ils en aient même connaissance. Naturellement, ceci doit être fait avec d'extrêmes précautions – si jamais cela doit être fait.

L'exemple `REGlob` ci-dessus n'implémente pas tous le support nécessaires pour surcharger proprement l'opérateur `glob` de perl. La fonction intégrée `glob` a des comportements différents selon qu'elle apparaît dans un contexte scalaire ou un contexte de lites, mais ce n'est pas le cas de notre `REGlob`. En fait, beaucoup de fonctions prédéfinies de perl ont de tels comportements sensibles au contexte, et ceux-ci doivent être supportés de façon adéquate par une surcharge correctement écrite. Pour un exemple pleinement fonctionnel de surcharge de `glob`, étudiez l'implémentation de `File::DosGlob` dans la bibliothèque standard.

Lorsque vous surchargez une fonction prédéfinie, votre surcharge devrait être cohérente (si possible) avec la syntaxe native de l'originale. Vous pouvez le faire en utilisant le prototype approprié. Pour obtenir le prototype d'une fonction prédéfinie surchargeable, utilisez la fonction `prototype` avec comme argument `"CORE::nom_de_la_fonction"` (voir prototype in *perlfunc*).

Remarquez que certaines fonctions prédéfinies utilise une syntaxe non représentable par un prototype (par exemple `system` ou `chomp`). Si vous les surchargez, vous ne pourrez pas simuler leur syntaxe d'origine.

Les "fonctions" prédéfinies `do`, `require` et `glob` peuvent elles-aussi être surchargées, mais comme elles sont magiques, elles conservent leur syntaxe d'origine et vous n'avez pas besoin de définir de prototype pour leurs fonctions de remplacement. (Par contre, vous ne pouvez pas surcharger la syntaxe `do BLOC`).

`require` possède en plus un peu de magie noire ; si vous appelez votre fonction `require` de remplacement par `require Foo::Bar`, vous recevrez en fait l'argument `"Foo/Bar.pm"` dans `@_`. Voir require in *perlfunc*.

Et, en se référant à l'exemple précédent, si vous surchargez `glob`, l'opérateur "glob" `<<*>` sera aussi surchargé.

De manières similaire, la surcharge de la fonction `readline` surcharge aussi l'opérateur d'E/S équivalent `<FILEHANDLE>`.

Et pour finir, certaines fonctions prédéfinies (comme `exists` ou `grep`) ne peuvent pas être surchargées.

3.11 Autochargement

Si vous appelez un sous-programme qui n'est pas défini, vous obtenez ordinairement une erreur fatale immédiate se plaignant que le sous-programme n'existe pas (De même pour les sous-programmes utilisés comme méthodes, lorsque la méthode n'existe dans aucune classe de base du paquetage de la classe). Toutefois, si un sous-programme `AUTOLOAD` est défini dans le paquetage ou dans les paquetages utilisés pour localiser le sous-programme originel, alors ce sous-programme `AUTOLOAD` est appelé avec les arguments qui auraient été passés au sous-programme originel. Le nom pleinement qualifié du sous-programme originel apparaît magiquement dans la variable globale $AUTOLOAD du même paquetage que la routine `AUTOLOAD`. Le nom n'est pas passé comme un argument ordinaire parce que, euh, eh bien, juste parce que, c'est pour ça...

La plupart des routines `AUTOLOAD` chargent une définition du sous-programme requis en utilisant eval(), puis l'exécutent en utilisant une forme spéciale de goto() qui efface la routine `AUTOLOAD` de la pile sans laisser de traces (Voir le module standard `AutoLoader` pour un exemple). Mais une routine `AUTOLOAD` peut aussi juste émuler la routine et ne jamais la définir. Par exemple, supposons qu'une fonction non encore définie doive juste appeler `system` avec ces arguments. Tout ce que vous feriez est ceci :

```
sub AUTOLOAD {
    my $program = $AUTOLOAD;
    $program =~ s/.*::/;
    system($program, @_);
}
date();
who('am', 'i');
ls('-l');
```

En fait, si vous prédéclarez les fonctions que vous désirez appeler de cette façon, vous n'avez mêmes pas besoin des parenthèses :

```
use subs qw(date who ls);
date;
who "am", "i";
ls -l;
```

Un exemple plus complet de ceci est le module Shell standard, qui peut traiter des appels indéfinis à des sous-programmes comme des appels de programmes externes.

Des mécanismes sont disponibles pour aider les auteurs de modules à découper les modules en fichiers autochargeables. Voir le module standard AutoLoader décrit dans *AutoLoader* et dans *AutoSplit*, les modules standards SelfLoader dans *SelfLoader*, et le document expliquant comment ajouter des fonctions C au code Perl dans *perlxs*.

3.12 Attributs de sous-programme

Une déclaration ou une définition de sous-programme peut contenir une liste d'attributs qui lui sont associés. Si une telle liste est présente, elle est découpée aux espaces et aux deux-points et traitée comme si un `use attributes` avait été rencontré. Voir *attributes* pour plus de détails sur les différents attributs actuellement supportés. Contrairement aux limitations de l'`use attrs` obsolète, la syntaxe `sub : ATTRLIST` fonctionne pour associer les attributs à une prédéclaration, et pas seulement dans la définition de sous-programme.

Les attributs doivent être des identifiants simples valides (sans aucune ponctuation à part le caractère "_"). Ils peuvent être suivis d'une liste de paramètres, qui n'est contrôlée que pour voir si ses parenthèses ('(',')') sont correctement imbriquées.

Exemples de syntaxe valide (même si les attributs sont inconnus) :

```
sub fnord (&\%) : switch(10,foo(7,3))  :  expensive ;
sub plugh () : Ugly('\(") :Bad ;
sub xyzzy : _5x5 { ... }
```

Exemples de syntaxe invalide :

```
sub fnord : switch(10,foo() ; # () non équilibrées
sub snoid : Ugly('(') ;       # () non équilibrées
sub xyzzy : 5x5 ;             # "5x5" n'est pas un identifiant valide
sub plugh : Y2::north ;       # "Y2::north" n'est pas un identifiant simple
sub snurt : foo + bar ;       # "+" n'est pas un deux-points ni une espace
```

La liste d'attributs est passée comme une liste de chaînes constantes au code qui les associe au sous-programme. En particulier, le second exemple de syntaxe valide ci-dessus à cette allure en termes d'analyse syntaxique et d'invocation :

```
use attributes __PACKAGE__, \&plugh, q[Ugly('\(")], 'Bad';
```

Pour plus de détails sur les listes d'attributs et leur manipulation, voir *attributes* en *Attributes::Handlers*.

4 VOIR AUSSI

Voir Modèles de fonctions in *perlref* pour plus de détails sur les références et les fermetures. Voir *perlxs* pour apprendre à appeler des sous-programmes en C depuis Perl. Voir *perlembed* pour apprendre à appeler des sous-programmes Perl depuis C. Voir *perlmod* pour apprendre à emballer vos fonctions dans des fichiers séparés. Voir *perlmodlib* pour apprendre quels modules de bibliothèques sont fournis en standard sur votre système. Voir *perltoot* pour apprendre à faire des appels à des méthodes objet.

5 TRADUCTION

5.1 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/.

5.2 Traducteur

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

5.3 Relecture

Personne pour l'instant.

6 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.

Ce document PDF est distribué selon les termes de la license Artistique de Perl. Toute autre distribution de ce fichier ou de ses dérivés impose qu'un arrangement soit fait avec le(s) propriétaire(s) des droits. Ces droits appartiennent aux auteurs du document original (lorsqu'ils sont identifiés dans la rubrique AUTEUR), aux traducteurs et relecteurs pour la version française et à moi-même pour la version PDF.