perluniintro

Table des matières

1 NAME/NOM

perluniintro - Introduction à l'utilisation d'Unicode en Perl

2 DESCRIPTION

Ce document donne une idée générale d'Unicode et de son utilisation en Perl.

2.1 Unicode

Unicode est un jeu de caractères standardisé qui prévoit de codifier tous les systèmes d'écriture existant de par le monde ainsi que de nombreux autres symboles.

Unicode et ISO/IEC 10646 sont des normes qui attribuent un code pour les caractères de tous les jeux de caractères modernes, couvrant plus de 30 systèmes d'écriture et des centaines de langues, incluant toutes les langues modernes commercialement importantes. Tous les caractères chinois, japonais et cooréens en font aussi parti. Ces standards prévoient de couvrir tous les caractères de plus de 250 systèmes d'écriture et de milliers de langues. Unicode 1.0 est sorti en octobre 1991 et et sa version 4.0 en avril 2003.

Un *caractère* Unicode est une entité abstraite. Il n'est lié à aucune taille d'entier en particulier et encore moins au type `char` du language C. Unicode est neutre vis à vis de la langue et de l'affichage : il ne code pas la langue utilisée par le texte et ne définit pas de polices ou d'autres aspects de présentation graphique. Unicode concerne les caractères et le texte composé de ces caractères.

Unicode définit des caractères comme `LATIN CAPITAL LETTER A` ou `GREEK SMALL LETTER ALPHA` et un nombre unique pour chacun de ces caractères, en l'occurence respectivement 0x0041 et 0x03B1. Ces nombres uniques sont appelés *points de code* ou plus simplement *numéros de caractères*.

Le standard Unicode préfère la notation hexadécimale pour les numéros de caractères. Si des nombres tels que `0x0041` ne vous sont pas familiers, jetez un oeil à la section Notation hexadécimale (§2.14). Le standard Unicode utilise la notation `U+0041 LATIN CAPITAL LETTER A` pour donner le numéro hexadécimal du caractère suivi de son nom normalisé.

Unicode définit aussi différentes *propriétés* pour les caractères, telles que "uppercase" (majuscule) ou "lowercase" (minuscule), "decimal digit" (chiffre décimal) ou "punctuation" (ponctuation) ; ces propriétés étant indépendantes du nom des caractères. En outre, sont définies plusieurs opérations sur les caractères telles que "passer en majuscule", "passer en minuscule" et "trier".

Un caractère Unicode est constitué soit d'un unique caractère, soit d'un *caractère de base* (tel que `LATIN CAPITAL LETTER A`) suivi d'un ou plusieurs *modificateurs* ou *caractères combinatoires* (tel que `COMBINING ACUTE ACCENT`). Cette séquence composée d'un caractère de base suivi de ses modificateurs est appelée une *séquence combinante*.

Le fait de considérer ces séquences comme étant un "caractère" dépend de votre point de vue. En tant que programmeur, vous aurez plutôt tendance à considérer chaque élément de la séquence pris séparément. Alors qu'un utilisateur considérera probablement la séquence complète comme étant un unique "caractère" car c'est de cette manière qu'il est perçu dans sa langue.

Avec la vision globale, déterminer le nombre total de caractères est difficile. Mais avec le point de vue du programmeur (chaque caractère, qu'il soit de base ou combinatoire, est un caractère), le concept de nombre de caractères est plus déterministe. Dans ce document, nous adopterons ce second point de vue : un "caractère" est un point de code Unicode, que ce soit un caractère de base ou un caractère combinatoire.

Pour certaines combinaisons, il existe des caractères *précomposés*. Par exemple `LATIN CAPITAL LETTER A WITH ACUTE` est défini comme un point de code unique. Ces caractères précomposés ne sont disponibles que pour certaines combinaisons et le sont principalement pour faciliter la conversion entre Unicode et les anciens standards (tel que ISO 8859). De manière générale, la méthode par combinaison est plus extensible. Pour faciliter la conversion entre les différentes manières de composer un caractère, plusieurs *formes de normalisation* sont définies dans le but de standardiser les représentations.

Pour des raisons de compatibilité avec les anciens systèmes de codage, l'idée "un numéro unique pour chaque caractère" est un peu remise en cause : en fait, il y a "au moins un numéro pour chaque caractère". Un même caractère peut être représenté différemment dans plusieurs anciens systèmes d'encodage. De même certains points de code ne correspondent à aucun caractère. En effet, d'une part, des blocs déjà utilisés contiennent des points de code non alloués. Et d'autre part, dans Unicode, on trouve des caractères de contrôle (ou de commande) et des caractères spéciaux qui ne représentent pas de vrais caractères.

Un mythe courant concernant Unicode est que celui-ci serait "16 bits" car il n'existe que `0x10000` (ou 65536) caractères entre `0x0000` et `0xFFFF`. **Ceci est faux.** Depuis Unicode 2.0 (juillet 1996), Unicode utilise 21 bits (`0x10FFFF`) et depuis Unicode 3.1 (mars 2001), des caractères ont été définis au-delà de `0xFFFF`. Les `0x10000` premiers caractères sont appellés *plan 0*, ou encore *plan multilingue de base* (PMP – ou BMP en anglais). Avec Unicode 3.1, c'est 17 (oui, dix-sept) plans qui ont été définis mais ils ne sont pas tous entièrement remplis, du moins pour le moment.

Un autre mythe est qu'il existerait une corrélation entre les langues et les blocs de 256 caractères, chaque bloc définissant les caractères utilisés par une langue ou un ensemble de langues. **Ceci est tout aussi faux.** Cette division en bloc existe, mais elle est presque totalement accidentelle, un simple artéfact de la manière dont les caractères ont été et continuent à être alloués. En revanche, il existe le concept d'*écriture* (ou script en anglais) qui est plus utile : il existe une écriture `latine`, une `grecque` et ainsi de suite. Les écritures sont généralement réparties sur plusieurs blocs. Pour plus d'information consultez *Unicode::UCD*.

Les point de code Unicode ne sont que des nombres abstraits. Pour recevoir ou émettre ces nombres abstraits, ils faut les *encoder* ou les *sérialiser* d'une manière ou d'une autre. Unicode définit plusieurs *formes d'encodage*, parmi lesquelles

UTF-8 est peut-être la plus populaire. UTF-8 est méthode d'encodage à taille variable qui encode les caractères Unicode sur 1 à 6 octets (maximum 4 pour les caractères actuellement définis). Les autres méthodes incluent UTF-16, UTF-32 et leur variantes grand-boutistes et petit-boutistes (UTF-8 est indépendant de l'ordre des octets). L'ISO/IEC définit aussi les formes UCS-2 et UCS-4.

Pour plus d'information à propos de l'encodage, par exemple pour savoir à quoi correspondent les caractères de substitution (*surrogates*) et les *marques d'ordre d'octets* (*byte order marks* ou BOMs), consultez *perlunicode*.

2.2 Le support d'Unicode en Perl

À partir de la version 5.6.0, Perl était apte à gérer Unicode nativement. Mais la version 5.8.0 a été la première version recommandée pour pouvoir travailler sérieusement avec Unicode. La version de maintenance 5.6.1 a corrigé un grand nombre des problèmes de l'implémentation initiale, mais par exemple, les expressions rationnelles ne fonctionnaient toujours pas en Unicode dans la version 5.6.1.

À partir de Perl 5.8.0, l'utilisation de `use utf8` n'est plus nécessaire. Dans les versions précédentes le pragma `utf8` était utilisé pour déclarer que les opérations du bloc ou du fichier courant étaient compatibles Unicode. Ce modèle s'avéra mauvais, ou tout du moins maladroit : la caractéristique "Unicode" est maintenant liée aux données et non plus aux opérations. L'utilisation de `use utf8` ne reste nécessaire que dans un seul cas : lorsque le script Perl est lui-même encodé en UTF-8. Cele vous permet alors d'utiliser l'UTF-8 pour vos identifiants ainsi que dans les chaînes de caractères et les expressions rationnelles. Ceci n'est pas le règlage par défaut, car sinon les scripts comportant des données encodées en 8 bits risqueraient de ne plus fonctionner. Consultez *utf8*.

2.3 Le modèle Unicode de Perl

Perl supporte aussi bien les chaînes de caractères utilisant l'encodage pré-5.6 sur huit bits natifs que celles utilisant des caractères Unicode. Le principe est que Perl essaye au maximum de conserver les données avec l'encodage 8 bits, mais dès que Unicode devient indispensable, les données sont converties en Unicode de manière transparente.

En interne, Perl utilise le jeu de caractères natif sur 8 bit de la plateforme (par exemple Latin-1) et pour encoder les chaines de caractères Unicode, il utilise par défaut UTF-8. Plus précisement, si tous les points de codes d'une chaîne sont inférieurs ou égal à `0xFF`, Perl utilisera l'encodage sur 8 bits natif. Sinon, il utilisera UTF-8.

En général, l'utilisateur de Perl n'a besoin ni de savoir ni de se préoccuper de la manière dont Perl encode ses chaînes de caractères en interne, mais cela peut devenir utile lorsqu'on envoie des chaînes Unicode dans un flux n'utilisant pas PerlIO (il utilisera donc l'encodage par "défaut"). Dans ce cas, les octets bruts utilisés en interne (reposant sur le jeu de caractères natif ou sur UTF-8, selon les chaînes) seront transmis et un avertissement "Wide character" sera émis pour les chaînes contenant un caractère dépassant 0x00FF.

Par exemple,

```
perl -e 'print "\x{DF}\n", "\x{0100}\x{DF}\n"'
```

produit un mélange plutôt inutile de caractères natifs et d'UTF-8 ainsi que l'avertissement :

```
Wide character in print at ...
```

Pour afficher de l'UTF-8, il faut utiliser la couche `:utf8`. L'ajout de :

```
binmode(STDOUT, ":utf8");
```

au début de ce programme d'exemple forcera l'affichage à être en totalité en UTF-8 et retirera les avertissements.

Vous pouvez automatiser l'utilisation de l'UTF-8 pour vos entrées/sorties standard, pour les appels à `open()` et pour `@ARGV` en utilisant l'option `-C` de la ligne de commande ou la variable d'environnement `PERL_UNICODE`. Consultez *perlrun* pour la documentation de l'option `-C`.

Notez que cela signifie que Perl espère que les autres logiciels fonctionnent eux-aussi en UTF-8 : si Perl est amené à croire que STDIN est en UTF-8 alors que STDIN provient d'une commande qui n'est pas en UTF-8, Perl se plaindra de caractères UTF-8 mal composés.

Toutes les fonctionnalités qui combinent Unicode et les E/S (I/O) nécessitent l'usage de la nouvelle fonctionnalité PerlIO. La quasi totalité des plateformes Perl 5.8 utilise PerlIO mais vous pouvez le vérifier en lancant la commande "perl -V" et en recherchant `useperlio=define`.

2.4 Unicode et EBCDIC

Perl 5.8.0 supporte aussi Unicode sur les plateformes EBCDIC. Le support d'Unicode y est plus complexe à implémenter car il nécessite des conversions supplémentaires à chaque utilisation. Certains problèmes persistent ; consultez *perlebcdic* pour plus de détails.

Dans tous les cas, le support actuel d'Unicode sur les plateformes EBCDIC est bien meilleur que dans les versions 5.6 (cela ne fonctionnait quasiment pas sur les plateformes EBCDIC). Sur ces plateformes, l'encodage interne est UTF-EBCDIC au lieu d'UTF-8. La différence est que UTF-8 est "ASCII-sûr" dans le sens où les caractères ASCII sont codés tels quels en UTF-8, alors que UTF-EBCDIC est "EBCDIC-sûr".

2.5 Créer de l'Unicode

Dans les littéraux, pour créer des caractères Unicode dont le point de code est supérieur à `0xFF`, il faut utiliser la notation `\x{...}` dans une chaîne entre guillemet :

```
my $smiley = "\x{263a}";
```

De la même façon, cette notation peut être utilisée dans les expressions rationnelles :

```
$smiley =~ /\x{263a}/;
```

À l'exécution vous pouvez utiliser `chr()` :

```
my $hebrew_alef = chr(0x05d0);
```

Consultez Autres ressources (§2.15) pour savoir comment trouver ces codes numériques.

Évidemment, `ord()` fera l'inverse : il transformera un caractère en un point de code.

Notez que `\x..` (sans `{}` et avec seulement deux chiffres hexadécimaux), `\x{...}` ainsi que `chr(...)` génèrent un caractère sur un octet pour les arguments inférieurs à `0x100` (256 en décimal) et ceci pour conserver une compatibilité ascendante avec les anciennes versions de Perl. Pour les arguments supérieurs à `0xFF`, il s'agira toujours de caractères Unicode. Si vous voulez forcer la génération de caractères Unicode quelle que soit la valeur, utilisez `pack("U",...)` au lieu de `\x..`, `\x{...}`, ou `chr()`.

Vous pouvez aussi utiliser le pragma `charnames` pour invoquer les caractères par leur nom entre guillemets :

```
use charnames ':full';
my $arabic_alef = "\N{ARABIC LETTER ALEF}";
```

Et, comme indiquer précédement, vous pouvez aussi utiliser la fonction `pack()` pour produire des caractères Unicode :

```
my $georgian_an  = pack("U", 0x10a0);
```

Notez que `\x{...}` et `\N{...}` sont des constantes : vous ne pouvez pas utiliser de variables dans ces notations. Si vous voulez un équivalent dynamique, utilisez `chr()` et `charnames::vianame()`.

Si vous voulez forcer un résultat en caractères Unicode, utilisez le préfixe spécial `"U0"`. Il ne consomme pas d'argument mais force, dans le résultat, l'utilisation de caractères Unicode à la place d'octets.

```
my $chars = pack("U0C*", 0x80, 0x42);
```

De la même manière, vous pouvez forcer l'utilisation d'octets par le préfixe spécial `"C0"`.

2.6 Manipuler l'Unicode

La manipulation d'Unicode est quasi transparente : il suffit d'utiliser les chaînes de caractères comme d'habitude. Les fonctions comme `index()`, `length()`, et `substr()` fonctionneront avec des caractères Unicode et il en sera de même avec les expressions rationnelles (consultez *perlunicode* et *perlretut*).

Notez que Perl considère les séquences combinantes comme étant composées de caractères séparés, par exemple :

```
use charnames ':full';
print length("\N{LATIN CAPITAL LETTER A}\N{COMBINING ACUTE ACCENT}"), "\n";
```

affichera 2 et non 1. La seule exception est l'utilisation de `\X` dans les expressions rationnelles pour reconnaître une séquence combinante de caractères.

En revanche, les choses ne sont malheureusement pas toujours aussi transparentes que ce soit avec l'encodage natif, avec les E/S ou dans certaines situations spéciales. C'est que nous allons détaillé maintenant.

2.7 Encodages natifs

Lorsque vous utilisez conjointement des données natives et des données Unicode, les donnés natives doivent être converties en Unicode. Par défaut ISO 8859-1 (ou EBCDIC selon le cas) est utilisé. Vous pouvez modifier ce comportement en utilisant le pragma `encoding`, par exemple :

```
use encoding 'latin2'; # ISO 8859-2
```

Dans ce cas, les littéraux (chaînes et expressions rationnelles), `chr()` et `ord()` produiront de l'Unicode en supposant un codage initiale en ISO 8859-2. Notez que le nom de l'encodage est reconnu de manière indulgente : au lieu de `latin2` on aurait pu utiliser `Latin2`, ou `iso8859-2` ou encore d'autres variantes. En utilisant seulement :

```
use encoding;
```

la valeur de la variable `PERL_ENCODING` sera alors utilisée. Si cette variable n'est pas positionnée, le pragma échouera.

Le module `Encode` connait de nombreux encodages et fournit une interface pour faire des conversions entre eux :

```
use Encode 'decode';
$data = decode("iso-8859-3", $data); # convertit du natif vers utf-8
```

2.8 Entrées/Sorties et Unicode

Normalement, l'affichage de données Unicode

```
print FH $une_chaine_avec_unicode, "\n";
```

produit les octets bruts utilisés en interne par Perl pour encoder la chaîne Unicode. Le système d'encodage interne dépend du systèmes ainsi que des caractères présents dans la chaîne. S'il y a au moins un caractère ayant un code suppérieur ou égal à `0x100`, vous obtiendrez un avertissement. Pour être sûr que la sortie utilise explicitement l'encodage que vous désirez, et pour éviter l'avertissement, ouvrez le flux avec l'encodage désiré. Quelques exemples :

```
open FH, ">:utf8", "fichier";

open FH, ">:encoding(ucs2)",      "fichier";
open FH, ">:encoding(UTF-8)",     "fichier";
open FH, ">:encoding(shift_jis)", "fichier";
```

et sur les flux déjà ouverts, utilisez `binmode()` :

```
binmode(STDOUT, ":utf8");

binmode(STDOUT, ":encoding(ucs2)");
binmode(STDOUT, ":encoding(UTF-8)");
binmode(STDOUT, ":encoding(shift_jis)");
```

Les noms d'encodages sont peu regardants : la casse n'a pas d'importance et beaucoup possèdent plusieurs alias. Notez par contre que le filtre `:utf8` doit toujours être spécifiée exactement de cette manière ; il n'est *pas* reconnue de manière aussi permissive que les noms d'encodages.

Consultez *PerlIO* pour en savoir plus sur le filtre `:utf8`, *PerlIO::encoding* et *Encode::PerlIO* pour tout ce qui concerne le filtre `:encoding()` et *Encode::Supported* pour les nombreux encodages reconnus par le module `Encode`.

La lecture d'un fichier que vous savez être encodés en Unicode ou en natif ne convertira pas magiquement les données en Unicode aux yeux de Perl. Pour cela, spécifier la couche désirée à l'ouverture du fichier :

```
open(my $fh,'<:utf8', 'anything');
my $ligne_unicode = <$fh>;

open(my $fh,'<:encoding(Big5)', 'anything');
my $ligne_unicode = <$fh>;
```

Pour plus de flexibilité, le filtre E/S peut aussi être spécifié via le pragma `open`. Consultez *open* ou l'exemple suivant :

```
use open ':utf8'; # les entrees et sorties seront par defaut en UTF-8
open X, ">file";
print X chr(0x100), "\n";
close X;
open Y, "<file";
printf "%#x\n", ord(<Y>); # Cela devrait afficher 0x100
close Y;
```

Avec le pragma `open` vous pouvez utiliser le filtre `:locale` :

```
BEGIN { $ENV{LC_ALL} = $ENV{LANG} = 'ru_RU.KOI8-R' }
# :locale va utiliser les variables d'environnements des locales comme LC_ALL
use open OUT => ':locale'; # russki parusski
open(O, ">koi8");
print O chr(0x430); # Unicode CYRILLIC SMALL LETTER A = KOI8-R 0xc1
close O;
open(I, "<koi8");
printf "%#x\n", ord(<I>), "\n"; # ceci devrais afficher 0xc1
close I;
```

ou vous pouvez aussi utiliser le filtre `':encoding(...)'` :

```
open(my $epic,'<:encoding(iso-8859-7)','iliad.greek');
my $ligne_unicode = <$epic>;
```

Ces méthodes installent, sur le flux d'entrée/sortie, un filtre transparent qui, lors de leur lecture à partir du flux, convertit les données depuis l'encodage spécifié. Le résultat est toujours de l'Unicode.
En instaurant un filtre par défaut, le pragma *open* impacte tous les appels à `open()` qui suivront. Si vous ne voulez modifier que certains flux, utilisez les filtres explicites directement dans les appels à `open()`.
Vous pouvez modifier l'encodage d'un flux déjà ouvert en utilisant `binmode()` ; consultez binmode in *perlfunc*.
Le filtre `:locale` ne peut pas, à l'heure actuelle (comme dans Perl 5.8.0), fonctionner avec `open()` et `binmode()` mais seulement avec le pragma `open`. Par contre `:utf8` et `:encoding(...)` fonctinnent avec `open()`, `binmode()` et le pragma `open`.
De la même manière, vous pouvez utiliser les filtres d'E/S sur un flux en sortie pour convertir automatiquement depuis Unicode vers l'encodage spécifié, lors des écritures dans ce flux. Par exemple, le code suivant copie le contenu du fichier "text.jis" (encodé en ISO-2022-JP, autrement dit JIS) dans le fichier "text.utf8", encodé en UTF-8 :

```
open(my $nihongo, '<:encoding(iso-2022-jp)', 'text.jis');
open(my $unicode, '>:utf8',                  'text.utf8');
while (<$nihongo>) { print $unicode $_ }
```

Les noms des encodages fournis à `open()` ou au pragma `open`, sont similaires à ceux du pragma `encoding` en ce qui concerne la flexibilité : `koi8-r` et `KOI8R` seront interprétés de la même manière.
Les encodages communément reconnus par l'ISO, en MIME ou par l'IANA et divers organismes de normalisation sont aussi Sreconnus ;> pour une liste plus détaillée consultez *Encode::Supported*.
La fonction `read()` lit des caractères et renvoie le nombre de caractère lus. Les fonctions `seek()` et `tell()` utilisent un nombre d'octets ainsi que `sysread()` et `sysseek()`.
Notez qu'à cause du comportement par défaut qui consiste à ne pas faire de conversion si aucun filtre par défaut n'est défini, il est facile d'écrire du code qui fera grossir un fichier en ré-encodant les données déjà encodées :

```
# ATTENTION CODE DEFECTUEUX
open F, "file";
local $/; ## lis l'ensemble du fichier sous forme de caractères 8 bits
$t = <F>;
close F;
open F, ">:utf8", "file";
print F $t; ## conversion en UTF-8 sur la sortie
close F;
```

Si vous exécutez deux fois ce code, le contenu de *file* sera doublement encodé en UTF-8. L'utilisation de `use open ':utf8'` aurait permis d'éviter ce bug ou on aurait pu aussi ouvrir *file* explicitement en UTF-8.
NOTE: Les fonctionnalités `:utf8` et `:encoding` ne fonctionnent que si votre Perl a été compilé avec la nouvelle fonctionnalité PerlIO (ce qui est le cas par défaut sur la plupart des systèmes).

2.9 Afficher de l'Unicode sous forme de texte

Il peut arriver que vous vouliez afficher des scalaires Perl contenant de l'Unicode comme un simple texte ASCII (ou EBCDIC). La sous-routine suivante convertit ses arguments de telle manière que les caractères Unicode dont le point de code est supérieur à 255 sont affichés sous la forme `\x{...}`, que les caractères de contrôle (comme `\n`) sont affichés sous la forme `\x..` et que le reste est affiché sans conversion :

```
sub nice_string {
    join("",
      map { $_ > 255 ?                  # Si caractère étendu ...
            sprintf("\\x{%04X}", $_) :  # \x{...}
            chr($_) =~ /[[:cntrl:]]/ ?  # Si caractère de contrôle ...
            sprintf("\\x%02X", $_) :    # \x..
            quotemeta(chr($_))          # Sinon sous forme quoted
      } unpack("U*", $_[0]));           # Extrait les caractères Unicode
}
```

Par exemple,

```
nice_string("foo\x{100}bar\n")
```

renvoie la chaîne :

```
'foo\x{0100}bar\x0A'
```

qui est prête à être imprimée.

2.10 Cas spéciaux

– Opérateur complément bit à bit ~ et vec()
  L'opérateur de complément bit à bit ~ peut produire des résultats surprenants s'il est utilisé sur une chaîne contenant des caractères dont le point de code est supérieur à 255. Dans un tel cas, le résultat sera conforme à l'encodage interne, mais pas avec grand chose d'autre. Alors, ne le faites pas. Le même problème se pose avec la fonction `vec()` : elle agit sur le motif de bits utilisé en interne pour stocker les caractères Unicode et non sur la valeur du point de code, ce qui n'est probablement pas ce que vous souhaitez.
– Découvrir l'encodage interne de Perl
  Les utilisateurs normaux de Perl ne devraient jamais se préoccuper de la manière dont celui-ci encode les chaînes Unicode (parce que la bonne manière de récupérer le contenu d'une chaîne Unicode, en entrée et en sortie, devrait toujours être via un filtre d'E/S explicitement indiqué). Mais si besoin est, voici deux manières de regarder derrière le rideau.
  Une première manière d'observer l'encodage interne des caractères Unicode est d'utiliser `unpack("C*", ...` pour récupérer les octets ou `unpack("H*", ...)` pour les afficher :
  ```
  # Affiche  c4 80  pour le code UTF-8  0xc4 0x80
  print join(" ", unpack("H*", pack("U", 0x100))), "\n";
  ```
  Une autre manière de faire est d'utiliser le module Devel::Peek :
  ```
  perl -MDevel::Peek -e 'Dump(chr(0x100))'
  ```
  qui affiche, dans FLAGS, le drapeau `UTF8` et, dans `PV`, les deux octets UTF-8 ainsi que le caractère Unicode. Consultez aussi la section concernant la fonction `utf8::is_utf8()` plus loin dans ce document.

2.11 Sujets avancés

– Équivalence des chaînes
  La question de l'équivalence ou de l'égalité de deux chaînes se complique singulièrement avec Unicode : que voulez-vous dire par "égale" ?
  (`LATIN CAPITAL LETTER A WITH ACUTE` est-il égal à `LATIN CAPITAL LETTER A` ?)
  La réponse courte est que par défaut pour les équivalence (`eq`, `ne`) Perl ne se base que sur les points de codes des caractères. Donc, dans le cas ci-dessus, la réponse est non (car 0x00C1 != 0x0041). Mais quelque fois, toutes les CAPITAL LETTER devrait être considérés comme égale, ou même tous les A.
  Pour une réponse détaillée, il faut considérer la normalisation et les problèmes de casse : consultez *Unicode::Normalize*, Unicode Technical Reports #15 and #21, *Unicode Normalization Forms* et *Case Mappings*, http://www.unicode.org/unicode/reports/tr15/ et http://www.unicode.org/unicode/reports/tr21/ .
  Depuis la version 5.8.0. de Perl, le cas "Full" case-folding de *Case Mappings/SpecialCasing* est implémenté.

– Ordre ou tri des chaînes
Le gens aiment que leurs chaînes de caractères soient correctement triées, le terme anglais utilisé pour Unicode est "collated". Mais, que veux dire trier ?
(Doit-on placé `LATIN CAPITAL LETTER A WITH ACUTE` avont ou après `LATIN CAPITAL LETTER A WITH GRAVE` ?)
La réponse courte est que par défaut, les opérateurs de comparaison de chaîne (`lt`, `le`, `cmp`, `ge`, `gt`) de Perl ne se basent que sur le point de code des caractères. Dans le cas précédent, la réponse est "après", car `0x00C1` > `0x00C0`.
La réponse détaillée est "ça dépend", et une bonne réponse ne peut être donnée sans connaître (au moins) le contexte linguistique. Consultez *Unicode::Collate*, et *Unicode Collation Algorithm* http://www.unicode.org/unicode/reports/tr10/

2.12 Divers

– Plages ou intervalles de caractères et classes
Les plages de caractères dans les classes des expressions rationnelles (`/[a-z]/`) et dans l'opérateur `tr///` (aussi connu sous le nom `y///`) ne sont pas magiquement adaptées à Unicode. Cela signifie que `[A-Za-z]` ne correspondra pas automagiquement à "toutes les lettres alphabétiques" ; ce n'est déjà pas le cas pour les caractères 8 bits, et vous devrez utiliser `/[[:alpha:]]/` dans ce cas.
Pour spécifier un classe de caractères comme celle-ci dans les expressions rationnelles, vous pouvez utiliser plusieurs propriétés Unicode (`\pL`, ou éventuellement `\p{Alphabetic}`, dans ce cas particulier). Vous pouvez aussi utiliser un point de code Unicode comme borne de plage, mais il n'y aucune magie associée à la spécification de ces plages. Pour plus d'informations, il existe des douzaines de classes de caractères Unicode, consultez *perlunicode*.
– Conversion de chaînes en nombres
Unicode définit plusieurs caractères séparateurs décimal et plusieurs caractères numériques, en dehors des chiffres habituels 0 à 9, tels que les chiffres Arabes et Indien. Perl ne supporte pas la conversion des chaînes vers des nombres pour les chiffres autres que 0 à 9 ASCII (et a à f ASCII pour l'hexadécimal).

2.13 Questions/Réponses

– Mes anciens scripts vont-ils encore fonctionner ?
Très probablement. À moins que vous ne génériez déjà des caractères Unicode vous-même, l'ancien comportement devrait être conservé. Le seul comportement qui change et qui pourrait générer de l'Unicode accidentellement est celui de `chr()` lorsqu'on lui fourni un argument supérieur à 255 : auparavant, il renvoyait le caractère correspondant à son argument modulo 255. Par exemple, `chr(300)` était égal à `chr(45)` ou "-" (en ASCII) alors que maintenant il produit LATIN CAPITAL LETTER I WITH BREVE.
– Comment rendre mes scripts compatibles avec Unicode ?
Très peu de choses doivent être faites sauf si vous génériez déjà des données Unicode. La plus importante est d'obtenir un flux d'entrée en Unicode ; reportez vous à la section ci-dessus concernant les entrées/sorties.
– Comment savoir si ma chaîne est en Unicode ?
Vous ne devriez pas vous en préoccuper. Non, vous ne devriez vraiment pas. Non, vraiment. Si vous le devez absolument, à part dans les cas décrits précédement, ça signifie que vous n'avez pas enocre atteint la transparence d'Unicode.
Bon, puisque vous insistez :

```
print utf8::is_utf8($string) ? 1 : 0, "\n";
```

Mais notez bien que cela ne signifie pas qu'un caractère de la chaîne est forcément encodé en UTF-8, ou qu'un caractère a un point de code supérieur à 0xFF (255) ou même à 0x80 (128), ou que la chaîne contient au moins un caractère. La seule chose que `is_utf8()` fait, c'est retourner la valeur du drapeau interne "utf8ness" attaché à `$string`. Si ce drapeau n'est pas positionné, les octets du scalaire sont interprétés comme étant encodés sur un octet. S'il est positionné, les octets du scalaires sont interprétés comme les points de code (multi-octets, de taille variable) UTF-8 des caractères. Les octets ajoutés à une chaîne encodées en UTF-8 sont automatiquement convertis en UTF-8. Si un scalaire non-UTF-8 est combiné à un scalaire UTF-8 (par interpolation dans des guillemets, par concaténation explicite ou par substitution des paramètres de printf/sprintf), le résultat sera une chaîne encodée en UTF-8. Par exemple :

```
$a = "ab\x80c";
$b = "\x{100}";
print "$a = $b\n";
```

La chaîne affichée sera `ab\x80c = \x{100}\n` encodée en UTF-8, mais `$a` restera encodé en simple octet.
Il vous faudra quelque fois connaître la taille en octets d'une chaîne et non en caractères. Pour ceci utilisez la fonction `Encode::encode_utf8()` ou encore le pragma `bytes` et `length()`, la seule fonction qu'il définit :

```
my $unicode = chr(0x100);
print length($unicode), "\n"; # Affiche 1
require Encode;
print length(Encode::encode_utf8($unicode)), "\n"; # Affiche 2
use bytes;
```

```
print length($unicode), "\n"; # Affiche 2
                              # (les codes 0xC4 0x80 de l'UTF-8)
```

– Comment puis-je detecter des données non valides pour un encodage particulier ?
Utilisez le package `Encode` pour tenter de les convertir. Par exemple :

```
use Encode 'decode_utf8';
if (decode_utf8($suite_octets_qu_on_pense_etre_utf8)) {
    # valide
} else {
    # invalide
}
```

Si il n'y a que UTF-8 qui vous intéresse, vous pouvez utiliser :

```
use warnings;
@chars = unpack("U0U*", $suite_octets_quon_pense_etre_utf8);
```

Si c'est invalide, un avertissement `Malformed UTF-8 character (byte 0x##) in unpack` est produit. "U0" signifie "n'attends strictement que de l'UTF-8". Sans ça, `unpack("U*",...)` accepterait des données telle que `chr(0xFF)`, comme `pack` ainsi que nous l'avons vu précédemment.

– Comment convertir des données binaires en un encodage particulier et inversement ?
Ce n'est probablement pas aussi utile que vous pourriez le penser. Normalement, vous ne devriez pas en avoir besoin.
Dans un sens, ce que vous demandez n'a pas beaucoup de sens : l'encodage concerne les caractères et des données binaires ne sont pas des "caractères", donc la conversion de "données" dans un encodage n'a pas sens à moins de connaître le jeu de caractères et l'encodage utilisés par ces données, mais dans ce cas il ne s'agit plus de données binaires.
Si vous disposez d'une séquence brute d'octets pour laquelle vous connaissez l'encodage particulier à utiliser, vous pouvez utiliser `Encode` :

```
use Encode 'from_to';
from_to($data, "iso-8859-1", "utf-8"); # de latin-1 à utf-8
```

L'appel à `from_to()` change les octets dans `$data`, mais la nature de la chaîne du point de vue de Perl n'est pas modifiée. Avant et après l'appel, la chaîne `$data` est juste un ensemble d'octets 8 bits. Pour Perl, l'encodage de cette chaîne reste "octets 8 bits natifs du système".
Vous pourriez mettre ceci en relation avec un hypothétique module 'Translate' :

```
use Translate;
my $phrase = "Yes";
Translate::from_to($phrase, 'english', 'deutsch');
## phrase contient maintenant "Ja"
```

Le contenu de la chaîne change, mais pas la nature de la chaîne. Perl n'en sais pas plus après qu'avant sur la nature affirmative ou non de la chaîne.
Revenons à la conversion de données. Si vous avez (ou voulez) des données utilisant l'encodage 8 bits natif de votre système (c-à-d. Latin-1, EBCDIC, etc.), vous pouvez utiliser pack/unpack pour convertir vers/depuis Unicode.

```
$native_string  = pack("C*", unpack("U*", $Unicode_string));
$Unicode_string = pack("U*", unpack("C*", $native_string));
```

Si vous avez une séquence d'octets que vous **savez** être de l'UTF-8 valide, mais que Perl ne le sait pas encore, vous pouvez le convaincre via :

```
use Encode 'decode_utf8';
$Unicode = decode_utf8($bytes);
```

Vous pouvez convertir de l'UTF-8 bien formé en une séquence d'octets, mais si vous voulez convertir des données binaires aléatoires en de l'UTF-8, c'est impossible. **Toutes les séquences d'octets ne forment par une chaîne UTF-8 bien formée**. Vous pouvez utiliser `unpack("C*", $string)` pour la première opération et vous pouvez créer une chaîne Unicode bien formée avec `pack("U*", 0xff, ...)`.

– Comment puis-je afficher de l'Unicode ? Comment puis-je saisir du l'Unicode ?
Consultez http://www.alanwood.net/unicode/ et http://www.cl.cam.ac.uk/~mgk25/unicode.html .

– Comment fonctionne Unicode avec les locales traditionnelles ?
En Perl, ça ne marche pas très bien. Evitez d'utiliser les locales avec le pragma `locale`. Utilisez soit l'un soit l'autre. Mais voyez *perlrun* pour la description de l'argument `-C` et la variable d'environnement équivalente `$ENV{PERL_UNICODE}` pour savoir comment activer les différentes fonctionnalités Unicode, par exemple en utilisant le paramétrage des locales.

2.14 Notation hexadécimale

Le standard Unicode privilégie la notation hexadécimale car cela montre plus clairement la division d'Unicode en blocs de 256 caractères. L'hexadécimal est aussi plus court que le décimal. Vous pouvez toujours utiliser la notation décimale, bien sûr, mais l'apprentissage de l'hexadécimal vous facilitera la vie avec le standard Unicode. La notation `U+HHHH` utilise l'hexadécimal, par exemple.

Le préfixe `0x` indique un nombre hexadécimal, les chiffres étant `0` à `9` *et* `a` à `f` (ou `A` à `F`, la casse n'ayant pas d'importance). Chaque chiffre hexadécimal représente quatres bits, autrement dit la moitié d'un octet. `print 0x..., "\n"` affiche un nombre hexadécimal convertit en décimal et `printf "%x\n", $decimal` affiche un nombre décimal convertit en hexadécimal. Si vous n'avez que les "chiffres" d'un nombre hexadécimal, vous pouvez aussi utiliser la fonction `hex()`.

```
print 0x0009, "\n";    # 9
print 0x000a, "\n";    # 10
print 0x000f, "\n";    # 15
print 0x0010, "\n";    # 16
print 0x0011, "\n";    # 17
print 0x0100, "\n";    # 256

print 0x0041, "\n";    # 65

printf "%x\n",  65;    # 41
printf "%#x\n", 65;    # 0x41

print hex("41"), "\n"; # 65
```

2.15 Autres ressources

– Consortium Unicode
  `http://www.unicode.org/`
– FAQ Unicode
  `http://www.unicode.org/unicode/faq/`
– Glossaire Unicode
  `http://www.unicode.org/glossary/`
– Unicode Useful Resources
  `http://www.unicode.org/unicode/onlinedat/resources.html`
– Support Unicode et multilingue en HTML, fontes, navigateurs web et autres applications.
  `http://www.alanwood.net/unicode/`
– FAQ UTF-8 et FAQ Unicode pour Unix/Linux
  `http://www.cl.cam.ac.uk/~mgk25/unicode.html`
– Jeux de caractères traditionnels
  `http://www.czyborra.com/`
  `http://www.eki.ee/letter/`
– Les fichiers de support de l'Unicode de l'installation de Perl se trouvent dans le répertoire
  `$Config{installprivlib}/unicore`
  en Perl 5.8.0 ou plus récent, et dans
  `$Config{installprivlib}/unicode`
  dans les versions 5.6.x. (Le renommage en *lib/unicore* a été décidé pour éviter les conflits avec lib/Unicode sur les systèmes de fichiers insensible à la casse). Le fichier principale des données Unicode est *UnicodeData.txt* (or *Unicode.301* en Perl 5.6.1.) Vous pouvez obtenir la valeur de `$Config{installprivlib}` via
  `perl "-V:installprivlib"`
  Vous pouvez explorer les informations des fichiers de données Unicode en utilisant le module `Unicode::UCD`.

3 UNICODE DANS LES VERSIONS DE PERL PLUS ANCIENNES

Si vous ne pouvez pas mettre à jour votre Perl en version 5.8.0 ou plus récente, vous pouvez toujours traiter de l'Unicode en utilisant les modules `Unicode::String`, `Unicode::Map8`, et `Unicode::Map`, disponibles sur le CPAN. Si vous avez installé GNU recode, vous pouvez aussi utiliser le front-end Perl `Convert::Recode` pour les conversions de caractères.

Les exemples suivants proposent une conversion rapide d'octets ISO 8859-1 (Latin-1) en octets UTF-8 et inversement, ce code fonctionnant aussi avec les versions plus anciennes de Perl 5.

```
# ISO 8859-1 vers UTF-8
s/([\x80-\xFF])/chr(0xC0|ord($1)>>6).chr(0x80|ord($1)&0x3F)/eg;

# UTF-8 vers ISO 8859-1
s/([\xC2\xC3])([\x80-\xBF])/chr(ord($1)<<6&0xC0|ord($2)&0x3F)/eg;
```

4 REFERENCES

perlunicode, *Encode*, *encoding*, *open*, *utf8*, *bytes*, *perlretut*, *perlrun*, *Unicode::Collate*, *Unicode::Normalize* et *Unicode::UCD*.

5 REMERCIEMENTS

Merci aux lecteurs des listes de diffusion perl5-porters@perl.org, perl-unicode@perl.org, linux-utf8@nl.linux.org et unicore@unicode.org pour leurs commentaires précieux.

6 AUTEUR, COPYRIGHT, ET LICENCE

7 TRADUCTION

7.1 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/.

7.2 Traducteur

Marc Carmier <mcarmier@gmail.com>.

7.3 Relecture

Paul Gaborit (Paul.Gaborit arobase enstimac.fr).

8 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.