ハードリファレンスとシンボリックリファレンスの違い

見ために違いはない。リファレンスをデリファレンスしようとすると、それはハードリファレンスの展開となる。

リファレンスでないものをデリファレンスしようとすると、シンボリックリファレンスを展開しようとしていると解釈される。Perlっぽい Interpretation ですな。

デリファレンス

デリファレンスとは、リファレンスの参照先の値をそこに展開すること。

デリファレンスを記述するには、デリファレンス演算子を、リファレンスを格納している変数の前にくっつける。デリファレンス演算子とは、$, @, &, %, * である。

たとえば、$$hoge は、リファレンス $hoge をデリファレンスしている。つまり、${$hoge}。

$$hoge[0] は？ ...${$hoge}[0] と見なされる。$hoge が指している配列の [0] 番目。${$hoge[0]} ではない。デリファレンス演算子は、結び付きが強いから。

コーディング･ポリシーとしては。この構文は、$hoge->[0] と書かなければならない。そうでないと、保守できなくなるのは明らか。
ハードリファレンスの展開に、$$hoge[0] などという記法を使ってはならない。

型グロブリファレンス

型グロブとは。

シンボル・テーブル上のすべての同名のエントリを表す、抽象的なコンテナ。

昔（Perl4 まで）、参照渡しに使われていたというが、今はリファレンスができたので、ほとんど使わなくてもよい。

今でも型グロブが必要な場面は、ファイルハンドルのリファレンスが欲しいとき。

```
open(HOGE, "hoge.txt");
$hoge = \HOGE;
```

とかやると、

```
$hoge = \"HOGE";
```

とみなされる。つまり、裸のワード "HOGE" へのリファレンスになってしまう。ファイルハンドルへのリファレンスが欲しいときは、

```
$hoge = \*HOGE;
```

と、型グロブへのリファレンスとして得なければならない。じゃあ、デリファレンスはどうするのか！？

型グロブは、コンテキストに応じて展開されるので、ファイルハンドル・コンテキストにそのまま置いてやれば、ちゃんとデリファレンスしてくれる。たとえば、<$hoge> とする。

■ use 文の挙動

use A_MODULE @list;

use 文の挙動は、すべて本文コンパイル前に行われることに留意。

まず、@INC パスをサーチする。

モジュールが無事ロードできたら、モジュールのソースコードを eval する（括目！）。

最後に、モジュール内に import という名前のサブルーチンがあったら、それを実行する。

'use A_MODULE' の後ろにリストが記述してあれば、それはサブルーチン 'import' の引数として渡される。

サブルーチン 'import' は、呼び出し側の名前空間内で、クラスメソッドとして実行される。

import についての覚書：
これは、Perl の変数はすべてグローバルであるという性質を利用して、呼び出し側の名前空間を侵食するために使用できる。

モジュールロード時の eval では、モジュール内の名前空間にしかアクセスできない。その時点では、呼び出し側の名前空間名はわからないから。import なら、呼び出し側の名前空間内で実行されることが保証されている。

ただ、本当に単なる import 目的なら、Exporter モジュールに任せた方が楽でしょう。

モジュールのロード前に @INC の内容を変更する：
標準モジュール lib を使う。

なお、use は require の上に構築されているらしい。ということは、同じモジュールを 2 度ロードしても、2 度目は無効になる。

■ pack と unpack

pack は、配列を受け取って文字列を返す？

unpack は文字列を整形する？

本来の使い方は、固定長のデータを整形すること（なのかなあ）。

固定長データから人間が必要な部分だけを取り出したり、入力データを保存するときに固定長に整形したりする（のだろうか）。

テンプレートの'C', 'c' を使えば、文字を文字コードに変換することができる (unpack)。文字コードを同じテンプレートで pack すれば、元の文字を得ることができる。

URI エンコーディングとデコーディング：
クッキーや GET メソッドでの文字列のやりとりでは、アルファベットと数字以外の特定の文字は、その文字の 16 進数コードに変換するという規約がある。

それらの記号類は、http が特別な目的のために使用する可能性があるからである。おおざっぱに言って、[^a-zA-Z0-9] を対応する文字コードの数値に変換する。それを 16 進数で表す。これがエンコード。

そして、それがエンコードされた 16 進文字コードであることを明示するために、頭に % 記号をつける。Perl でこの過程を実装する場合。まず、

```
$target_string =~ s/([^a-zA-Z0-9])/'%' . sprintf('%x', unpack('C', $1))/ge;
```

とする。これでエンコード完了。ごちゃっとしているが、unpack は単に [^a-zA-Z0-9] を対応する ASCII コードに直しているだけ。

sprintf は、それを単に 16 進数に直しているだけ。というか、Perl で 10 進数を他の進数に変換するには、sprintf を使う。ASCII コードは 0-255 なので、2 桁の 16 進数で表しきれることが保証されている。

これで、対象文字の中で、アルファベットでも数字でもないものは、16 進文字コードの頭に % 識別記号がついたものになった。

デコードは、

```
$encoded_string_to_be_decoded =~ s/%([0-9a-zA-Z]{2})/pack('C*', hex($1))/ge;
```

となる。エンコードされた文字列の中では、% 記号の後には必ず、2 桁の 16 進数が続いていることが保証されている。これが、デコードされるべき文字である。

そこで、% 記号と 16 進文字列の並びを正規表現でマッチさせる。16 進数部分を 10 進数に変換する（hex 関数）。それをテンプレート 'C*' で pack すれば、元の文字が得られる。

■ sort の比較ルーチン

sort に比較ルーチンを指定すると、暗黙の変数操作が行われる。

$a, $b の怪である。

sort 比較ルーチン sort 対象配列

こうするとまず、sort 対象配列から二つの要素が呼ばれて、比較ルーチンへ渡される。

しかも、何も言わなくても比較ルーチン内の local($a, $b) に渡される。

勝手に渡される。

比較ルーチンを書く側は、このローカル変数を宣言する必要は全くない。ただ書いておけば、勝手に Perl が宣言して渡すのである。

■未定義値、0、"0"

未定義値と 0 の関係は。
数値として比較すると、同値。
文字列として比較すると、同値ではない。
未定義値、0、"0" どれも単独で評価すると、偽。

defined：対象が未定義値 undef なら偽。0、"0" は真と判断される。
exists：ハッシュ表にキーが登録されていれば、値が undef だろうと 0 だろうと真。

Perl は、ブールコンテキストでは、最終的には "0" のみを偽と認定すると、どっかに書いてあったような気がするが、どこだったけ？

文字列："0" でなければすべて真
数字：小数点のゼロを取った後、文字列に変換して、"0" なら偽、それ以外は真。
0.00 -> 0 -> "0" -> False
空リスト、未定義値は数字にすると 0 に変換される。その後は上に同じ。

なんてことが、どっかに書いてあったような気がするのだが。

■処理失敗時の関数の戻り値

スカラーコンテキストでは未定義値が返り、リストコンテキストでは空リストが帰る（ラクダ , pp. 163）。

ていうか、みんなドキュメント読もうね。

■プロセス操作

exec：引数となる文字列を、システム・コマンドとして実行する。実行した Perl プロセス自身がコマンドとなって消えうせる。

system：fork して、子プロセスに、引数をシステム・コマンドとして実行させる。子プロセスの実行が終われば、元の Perl プロセスに制御が戻る。

fork：下記は簡単すぎるので、本当はもっと条件厳しくしなきゃいけない。

親 = 子プロセスのプロセス ID が戻る
子 = 0 が戻る。
失敗 = 未定義値が戻る。

```
if ($pid = fork) {
    wait;
} elsif (defined($pid)) {
    $ppid = getppid();
    print "I'm a child of $ppid.\n";
} else {
    die "Couldn't fork.\n";
}
```

■ localtime

$year 値、つまり (localtime)[5] は、西暦年号から 1900 を引いた数が返されるので、注意。

それから、スカラー・コンテキストで評価すると、英語風に整形された文字列が帰ってくるって知ってた？

■コンテキストの明示

これはけっこう忘れるので、要メモ。

リスト：()

なんか、ラクダ 2.2.3 じゃ、リスト･コンテキストを明示しなきゃいけない場面はないだろうとか言ってるが、コードというのは人間も読む以上、見えるようにしておいた方がいいだろ。

print (localtime)[5] + 1900;
とか。

wantarray なんてものもあるらしい。

スカラー：scalar()

■ map と grep

map、grep のブロック中で $_ を破壊すると、元の配列も破壊される。これは直感的だろうか？

あと、map と grep のブロック内で、リストの各要素を $_ 以外で受ける方法はないようだ。今のところ。