NAME

Switch - A switch statement for Perl

SYNOPSIS

```
    use Switch;

    switch ($val) {
 case 1  { print "number 1" }
 case "a" { print "string a" }
 case [1..10,42] { print "number in list" }
 case (\@array) { print "number in list" }
 case /\w+/ { print "pattern" }
 case qr/\w+/ { print "pattern" }
 case (\%hash) { print "entry in hash" }
 case (\&sub) { print "arg to subroutine" }
 else  { print "previous case not true" }
    }
```

BACKGROUND

[Skip ahead to *DESCRIPTION* if you don't care about the whys and wherefores of this control structure]

In seeking to devise a "Swiss Army" case mechanism suitable for Perl, it is useful to generalize this notion of distributed conditional testing as far as possible. Specifically, the concept of "matching" between the switch value and the various case values need not be restricted to numeric (or string or referential) equality, as it is in other languages. Indeed, as Table 1 illustrates, Perl offers at least eighteen different ways in which two values could generate a match.

```
 Table 1: Matching a switch value ($s) with a case value ($c)

        Switch  Case    Type of Match Implied   Matching Code
        Value   Value
        ======  =====   =====================   =============


        number  same    numeric or referential  match if $s == $c;
        or ref          equality

 object  method result of method call   match if $s->$c();
 ref     name     match if defined $s->$c();
  or ref


        other   other   string equality         match if $s eq $c;
        non-ref non-ref
        scalar  scalar

        string  regexp  pattern match           match if $s =~ /$c/;

        array   scalar  array entry existence   match if 0<=$c && $c<@$s;
        ref             array entry definition  match if defined $s->[$c];
                        array entry truth       match if $s->[$c];

        array   array   array intersection      match if intersects(@$s,
@$c);
```

```
        ref     ref     (apply this table to
                         all pairs of elements
                         $s->[$i] and
                         $c->[$j])

        array   regexp  array grep              match if grep /$c/, @$s;
        ref

        hash    scalar  hash entry existence    match if exists $s->{$c};
        ref             hash entry definition   match if defined $s->{$c};
                        hash entry truth        match if $s->{$c};

        hash    regexp  hash grep               match if grep /$c/, keys
%$s;
        ref

        sub     scalar  return value defn       match if defined $s->($c);
        ref             return value truth      match if $s->($c);

        sub     array   return value defn       match if defined $s->(@$c);
        ref     ref     return value truth      match if $s->(@$c);
```

In reality, Table 1 covers 31 alternatives, because only the equality and intersection tests are commutative; in all other cases, the roles of the `$s` and `$c` variables could be reversed to produce a different test. For example, instead of testing a single hash for the existence of a series of keys (`match if exists $s->{$c}`), one could test for the existence of a single key in a series of hashes (`match if exists $c->{$s}`).

DESCRIPTION

The Switch.pm module implements a generalized case mechanism that covers most (but not all) of the numerous possible combinations of switch and case values described above.

The module augments the standard Perl syntax with two new control statements: `switch` and `case`. The `switch` statement takes a single scalar argument of any type, specified in parentheses. `switch` stores this value as the current switch value in a (localized) control variable. The value is followed by a block which may contain one or more Perl statements (including the `case` statement described below). The block is unconditionally executed once the switch value has been cached.

A `case` statement takes a single scalar argument (in mandatory parentheses if it's a variable; otherwise the parens are optional) and selects the appropriate type of matching between that argument and the current switch value. The type of matching used is determined by the respective types of the switch value and the `case` argument, as specified in Table 1. If the match is successful, the mandatory block associated with the `case` statement is executed.

In most other respects, the `case` statement is semantically identical to an `if` statement. For example, it can be followed by an `else` clause, and can be used as a postfix statement qualifier.

However, when a `case` block has been executed control is automatically transferred to the statement after the immediately enclosing `switch` block, rather than to the next statement within the block. In other words, the success of any `case` statement prevents other cases in the same scope from executing. But see *Allowing fall-through* below.

Together these two new statements provide a fully generalized case mechanism:

```
        use Switch;
```

```
        # AND LATER...

        %special = ( woohoo => 1,  d'oh => 1 );

        while (<>) {
    chomp;
            switch ($_) {
                case (%special) { print "homer\n"; }      # if $special{$_}
                case /[a-z]/i   { print "alpha\n"; }      # if $_ =~ /a-z/i
                case [1..9]     { print "small num\n"; }  # if $_ in [1..9]
                case { $_[0] >= 10 } { print "big num\n"; } # if $_ >= 10
                print "must be punctuation\n" case /\W/;  # if $_ ~= /\W/
    }
        }
```

Note that `switch`es can be nested within `case` (or any other) blocks, and a series of `case` statements can try different types of matches -- hash membership, pattern match, array intersection, simple equality, etc. -- against the same switch value.

The use of intersection tests against an array reference is particularly useful for aggregating integral cases:

```
        sub classify_digit
        {
                switch ($_[0]) { case 0            { return 'zero' }
                                 case [2,4,6,8]    { return 'even' }
                                 case [1,3,5,7,9]  { return 'odd' }
                                 case /[A-F]/i     { return 'hex' }
                               }
        }
```

Allowing fall-through

Fall-though (trying another case after one has already succeeded) is usually a Bad Idea in a switch statement. However, this is Perl, not a police state, so there *is* a way to do it, if you must.

If a `case` block executes an untargeted `next`, control is immediately transferred to the statement *after* the `case` statement (i.e. usually another case), rather than out of the surrounding `switch` block.

For example:

```
        switch ($val) {
                case 1      { handle_num_1(); next }    # and try next
case...
                case "1"    { handle_str_1(); next }    # and try next
case...
                case [0..9] { handle_num_any(); }       # and we're done
                case /\d/   { handle_dig_any(); next }  # and try next
case...
                case /.*/   { handle_str_any(); next }  # and try next
case...
        }
```

If $val held the number `1`, the above `switch` block would call the first three `handle_...` subroutines, jumping to the next case test each time it encountered a `next`. After the third `case` block was executed, control would jump to the end of the enclosing `switch` block.

On the other hand, if $val held `10`, then only the last two `handle_...` subroutines would be called.

Note that this mechanism allows the notion of *conditional fall-through*. For example:

```
switch ($val) {
        case [0..9] { handle_num_any(); next if $val < 7; }
        case /\d/   { handle_dig_any(); }
}
```

If an untargeted `last` statement is executed in a case block, this immediately transfers control out of the enclosing `switch` block (in other words, there is an implicit `last` at the end of each normal `case` block). Thus the previous example could also have been written:

```
switch ($val) {
        case [0..9] { handle_num_any(); last if $val >= 7; next; }
        case /\d/   { handle_dig_any(); }
}
```

Automating fall-through

In situations where case fall-through should be the norm, rather than an exception, an endless succession of terminal `next`s is tedious and ugly. Hence, it is possible to reverse the default behaviour by specifying the string "fallthrough" when importing the module. For example, the following code is equivalent to the first example in *Allowing fall-through*:

```
use Switch 'fallthrough';

switch ($val) {
        case 1      { handle_num_1(); }
        case "1"    { handle_str_1(); }
        case [0..9] { handle_num_any(); last }
        case /\d/   { handle_dig_any(); }
        case /.*/   { handle_str_any(); }
}
```

Note the explicit use of a `last` to preserve the non-fall-through behaviour of the third case.

Alternative syntax

Perl 6 will provide a built-in switch statement with essentially the same semantics as those offered by Switch.pm, but with a different pair of keywords. In Perl 6 `switch` will be spelled `given`, and `case` will be pronounced `when`. In addition, the `when` statement will not require switch or case values to be parenthesized.

This future syntax is also (largely) available via the Switch.pm module, by importing it with the argument `"Perl6"`. For example:

```
use Switch 'Perl6';

given ($val) {
        when 1       { handle_num_1(); }
        when ($str1) { handle_str_1(); }
        when [0..9]  { handle_num_any(); last }
        when /\d/    { handle_dig_any(); }
        when /.*/    { handle_str_any(); }
        default      { handle anything else; }
}
```

Note that scalars still need to be parenthesized, since they would be ambiguous in Perl 5.

Note too that you can mix and match both syntaxes by importing the module with:

```
use Switch 'Perl5', 'Perl6';
```

Higher-order Operations

One situation in which `switch` and `case` do not provide a good substitute for a cascaded `if`, is where a switch value needs to be tested against a series of conditions. For example:

```
sub beverage {
    switch (shift) {
        case { $_[0] < 10 } { return 'milk' }
        case { $_[0] < 20 } { return 'coke' }
        case { $_[0] < 30 } { return 'beer' }
        case { $_[0] < 40 } { return 'wine' }
        case { $_[0] < 50 } { return 'malt' }
        case { $_[0] < 60 } { return 'Moet' }
        else                { return 'milk' }
    }
}
```

(This is equivalent to writing `case (sub { $_[0] < 10 })`, etc.; `$_[0]` is the argument to the anonymous subroutine.)

The need to specify each condition as a subroutine block is tiresome. To overcome this, when importing Switch.pm, a special "placeholder" subroutine named __ [sic] may also be imported. This subroutine converts (almost) any expression in which it appears to a reference to a higher-order function. That is, the expression:

```
use Switch '__';

__ < 2
```

is equivalent to:

```
sub { $_[0] < 2 }
```

With __, the previous ugly case statements can be rewritten:

```
case  __ < 10  { return 'milk' }
case  __ < 20  { return 'coke' }
case  __ < 30  { return 'beer' }
case  __ < 40  { return 'wine' }
case  __ < 50  { return 'malt' }
case  __ < 60  { return 'Moet' }
else           { return 'milk' }
```

The __ subroutine makes extensive use of operator overloading to perform its magic. All operations involving __ are overloaded to produce an anonymous subroutine that implements a lazy version of the original operation.

The only problem is that operator overloading does not allow the boolean operators && and `||` to be overloaded. So a case statement like this:

```
case  0 <= __ && __ < 10  { return 'digit' }
```

doesn't act as expected, because when it is executed, it constructs two higher order subroutines and then treats the two resulting references as arguments to &&:

```
sub { 0 <= $_[0] } && sub { $_[0] < 10 }
```

This boolean expression is inevitably true, since both references are non-false. Fortunately, the overloaded `'bool'` operator catches this situation and flags it as an error.

DEPENDENCIES

The module is implemented using Filter::Util::Call and Text::Balanced and requires both these modules to be installed.

AUTHOR

Damian Conway (damian@conway.org). This module is now maintained by Rafael Garcia-Suarez (rgarciasuarez@gmail.com) and more generally by the Perl 5 Porters (perl5-porters@perl.org), as part of the Perl core.

BUGS

There are undoubtedly serious bugs lurking somewhere in code this funky :-) Bug reports and other feedback are most welcome.

LIMITATIONS

Due to the heuristic nature of Switch.pm's source parsing, the presence of regexes with embedded newlines that are specified with raw `/.../` delimiters and don't have a modifier `//x` are indistinguishable from code chunks beginning with the division operator `/`. As a workaround you must use `m/.../` or `m?...?` for such patterns. Also, the presence of regexes specified with raw `?...?` delimiters may cause mysterious errors. The workaround is to use `m?...?` instead.

Due to the way source filters work in Perl, you can't use Switch inside an string `eval`.

If your source file is longer then 1 million characters and you have a switch statement that crosses the 1 million (or 2 million, etc.) character boundary you will get mysterious errors. The workaround is to use smaller source files.

COPYRIGHT