

1 NAME

Parse::Eyapp::Base - Miscellaneous support functions for Parse::Eyapp

2 SYNOPSIS

```
use Parse::Eyapp::Base qw(:all)
```

3 INTRODUCTION

Parse::Eyapp::Base holds a set of utility functions that give support to the other modules that made Parse::Eyapp. Several of them are related to the dynamic use of methods and subroutines.

4 SUBROUTINES

Function insert_method

Function `insert_method` receives as arguments a list of class names, the name of the method that will be inserted in such classes and a reference to the code implementing such method.

```
insert_method( qw{CLASS1 CLASS2 ... }, 'subname', sub { ... } )
```

It inserts the method in the specified classes. A second way to call it is without the last argument, the handler:

```
insert_method( qw{CLASS1 CLASS2 ... }, 'subname' )
```

In such case the function is deleted from all the specified classes and it no longer exists. The caller class is assumed if no classes are specified:

```
insert_method('subname', sub { ... } )
```

See the following session with the debugger:

```
pl@nereida:~/src/perl/YappWithDefaultAction/lib/Parse/Eyapp$ perl -wde 0
main:(-e:1): 0
DB<1> use Parse::Eyapp::Base qw(:all)
DB<2> insert_method( qw{PLUS MINUS TIMES }, 'printclass', sub { print "$_[0]\n" } )
DB<3> $_->printclass for qw{PLUS MINUS TIMES }
PLUS
MINUS
TIMES

DB<4> insert_method( qw{PLUS MINUS TIMES }, 'printclass')
DB<5> print $_->can('printclass')?"Yes\n":"No\n" for qw{PLUS MINUS TIMES }
No
No
No
```

Function insert_function

It works as `insert_method` (see section `Function insert_method`), only that instead of classes receives the full names of the functions to install and a reference to the code implementing such function. See an example of call:

```
insert_function(
  qw{ FUNCTIONCALL::type_info VARARRAY::type_info VAR::type_info },
  \&type_info
);
```

When the package is unspecified the caller package is assumed. See the following example:

```

pl@nereida:~/src/perl/YappWithDefaultAction/lib/Parse/Eyapp$ perl -wde 0
main::(-e:1): 0
DB<1> use Parse::Eyapp::Base qw(:all)
DB<2> insert_function('Tutu::tata', 'titi', sub{ print "Inside titi\n"})
DB<3> titi()
Inside titi

DB<4> Tutu::tata()
Inside titi

```

Function empty_method

The call to

```
empty_method(qw{CLASSES ... }, 'subname')
```

is equivalent to

```
insert_method(qw{CLASSES ... }, 'subname', sub {})
```

Consequently `empty_method` replaces the current CODE for function `subname` by an empty subroutine

Function push_method

The call

```
push_method( qw{CLASS1 CLASS2 ... }, 'subname', sub { ... } )
```

saves the current methods `CLASS1::subname`, `CLASS2::subname`, etc. in a stack and proceeds to install the new handler specified through the last argument. See an example:

```

pl@nereida:~/src/perl/YappWithDefaultAction/lib/Parse/Eyapp$ perl -wde 0
main::(-e:1): 0
DB<1> use Parse::Eyapp::Base qw(:all)
DB<2> sub Tutu::titi { print "Inside first Tutu::titi!\n" }
DB<3> push_method('Tutu', 'titi', sub { print "New titi!\n" })
DB<4> Tutu::titi()
New titi!

DB<5> pop_method('Tutu', 'titi')
DB<6> Tutu::titi()
Inside first Tutu::titi!

DB<7> push_method('Tutu', 'titi') # No handler: sub Tutu::titi no longer exists
DB<8> print "Can't titi\n" unless Tutu->can('titi')
Can't titi

DB<9> pop_method('Tutu', 'titi') # Give me the old sub
DB<10> Tutu::titi()
Inside first Tutu::titi!

```

The caller class is assumed if no classes are specified.

In list context the `push_method` function returns an array of pointers to the old versions of the function. In a scalar context returns the first CODE reference. See the following example:

```

pl@nereida:~/src/perl/YappWithDefaultAction/examples$ cat -n returnedbypushmethod.pl
1  #!/usr/local/bin/perl -w
2  use strict;
3  use Parse::Eyapp::Base qw(:all);
4
5  sub tutu { "tutu" }
6  sub Chum::tutu { "chum" }
7
8  my @classes = qw{main Cham Chum};

```

```

 9
10 my %oldf;
11 our $tutu = 5;
12 our @tutu = 9..12;
13 $Cham::tutu = 8;
14 @Cham::tutu = 1..3;
15
16 @oldf{@classes} = push_method(@classes, 'tutu', sub { "titi" });
17
18 print "Calling new function 'tutu':."&tutu()."\n";
19
20 for (@classes) {
21     if (defined($oldf{$_})) {
22         print "Old function 'tutu' in $_ gives: ".$oldf{$_}->()."\n";
23     }
24     else {
25         print "Function 'tutu' wasn't defined in $_\n";
26     }
27 }

```

The following session with the debugger shows that:

- Package variables with the same name like `$tutu` or `@tutu` aren't changed by `insert_method`
- References to the old versions of function `tutu` are returned by `insert_method`

```

pl@nereida:~/LEyapp/examples$ perl -wd returnedbypushmethod.pl
main::(returnedbypushmethod.pl:8):
8:     my @classes = qw{main Cham Chum};
  DB<1> c 18
main::(returnedbypushmethod.pl:18):
18:     print "Calling new function 'tutu':."&tutu()."\n";
  DB<2> n
Calling new function 'tutu':titi
main::(returnedbypushmethod.pl:20):
20:     for (@classes) {
  DB<2> x @tutu
0 9
1 10
2 11
3 12
  DB<3> x @Cham::tutu
0 1
1 2
2 3
  DB<4> p $Cham::tutu
8
  DB<5> c
Old function 'tutu' in main gives: tutu
Function 'tutu' wasn't defined in Cham
Old function 'tutu' in Chum gives: chum

```

Function `pop_method`

The call

```
pop_method(qw{CLASS1 CLASS2 ... }, 'subname' )
```

pops the methods in the tops of the stacks associated with `CLASS1::subname`, `CLASS2::subname`, etc. See the example in the section `push_method` above.

- The caller class is assumed if no classes are specified.

- If the stack for `CLASS::subname` is empty the old specification of `subname` will remain.

```
pl@nereida:~/LEyapp/examples$ cat returnedbypopmethod.pl
#!/usr/local/bin/perl -w
use strict;
use Parse::Eyapp::Base qw(:all);

sub tutu { "tutu" }

my $old = pop_method('tutu');

print "Function 'tutu' is available\n" if main->can('tutu');
print "Old function 'tutu' gives: ".$old->()."\n";
```

When executed gives the following output:

```
pl@nereida:~/LEyapp/examples$ returnedbypopmethod.pl
Function 'tutu' is available
Old function 'tutu' gives: tutu
```

- In list context the `pop_method` function returns an array of pointers to the old versions of the function. In a scalar context returns the first function reference. When the stack is empty the function(s) are deleted.

Examples of `push_method` and `pop_method`

Hiding functions

See the following example:

```
package Tutu;
use Parse::Eyapp::Base qw(:all);

sub tutu {
    print "Inside tutu\n"
}

sub plim {

    # When the stack is empty the old 'tutu' remains ...
    pop_method('tutu');

    &tutu(); # Inside tutu

    push_method('tutu'); # Tutu disappears
}

package main;

Tutu::plim();
# main can't call 'tutu'
print "Can't tutu\n" unless Tutu->can('tutu');
Tutu::plim();
```

When executed the former program produces this output:

```
pl@nereida:~/LEyapp/examples$ localsubbase.pl
Inside tutu
Can't tutu
Inside tutu
```

Changing the Behavior of Method-parametric Methods

A common situation where I need the couple (`push_method`, `pop_method`) is to control the behavior of method `str` when debugging:

```
pl@nereida:~/Lbook/code/Simple-Types/script$ perl -wd usetypes.pl prueba26.c 2
Loading DB routines from perl5db.pl version 1.28
Editor support available.
main::(usetypes.pl:5): my $filename = shift || die "Usage:\n$0 file.c\n";
  DB<1> c Parse::Eyapp::Node::str
1 int f() {
2   int a[30];
3
4   return;
5 }
Parse::Eyapp::Node::str(/home/pl/src/perl/YappWithDefaultAction/lib//Parse/Eyapp/Node.pm:716):
716:     my @terms;
```

Let us assume I want to see the syntax tree for this program. I can see it using `$_[0]->str` but the problem is that nodes `PROGRAM` and `FUNCTION` have defined a `footnote` method that will dump their symbol and type tables producing hundred of lines of output and making difficult to see the shape of the tree. This is because method `str` calls method `footnote` wherever the node being visited *can* do `footnote`. The solution is to use `push_method` to make the `footnote` methods disappear:

```
DB<2> use Parse::Eyapp::Base qw(:all)
DB<3> push_method(qw{PROGRAM FUNCTION}, 'footnote')
```

The use of `push_method` without an explicit code handler eliminates the `CODE` entry for `footnote`:

```
DB<4> p $_->can('footnote')? "1\n" : "0\n" for (qw{PROGRAM FUNCTION})
0
0
```

Now I can see the shape of the tree:

```
DB<5> p $_[0]->str
PROGRAM(
  FUNCTION[f](
    EMPTYRETURN
  )
) # PROGRAM
```

If I want back the `footnote` methods I can use `pop_method`:

```
DB<6> pop_method(qw{PROGRAM FUNCTION}, 'footnote')
DB<7> p $_->can('footnote')? "1\n" : "0\n" for (qw{PROGRAM FUNCTION})
1
1
```

Now the information will be profuse:

```
DB<8> p $_[0]->str
PROGRAM^{0}(
  FUNCTION[f]^{1}(
    EMPTYRETURN
  )
) # PROGRAM
-----
0)
Types:
$VAR1 = {
  'CHAR' => bless( {
```

```

    'children' => []
  }, 'CHAR' ),
  .... etc, etc.
  'A_30(INT)' => bless( {
    'children' => [
      $VAR1->{'INT'}
    ]
  }, 'A_30' )
};
Symbol Table:
$VAR1 = {
  'f' => {
    'type' => 'F(X_0(),INT)',
    'line' => 1
  }
};

```

```

-----
1)
$VAR1 = {
  'a' => {
    'type' => 'A_30(INT)',
    'line' => 2
  }
};

```

You can still do something like this to achieve a similar effect:

```

DB<9> p eval { local (*PROGRAM::footnote, *FUNCTION::footnote) = (sub {}, sub {}); $_[0]->str }
PROGRAM(
  FUNCTION[f](
    EMPTYRETURN
  )
) # PROGRAM

```

but is certainly more verbose and does not eliminate function `footnote` from the `PROGRAM` and `FUNCTION` classes.

Therefore the usefulness of `push_method` is when you either want to temporarily delete your function/methods or localize them not necessarily in a scope basis.

Function `compute_lines`

The call

```
compute_lines(\$text, $filename, $pattern)
```

Substitutes all the occurrences of `$pattern` by `#line $number $filename` in string `$text`. where `$number` is the line number.

Function `slurp_file`

The call

```
my $input = slurp_file($filename, "c");
```

returns a string with the contents of the file `$filename` assuming extension `"c"`.

```

pl@nereida:~/src/perl/YappWithDefaultAction/lib/Parse/Eyapp$ perl -wde 0
main::(-e:1): 0
DB<1> use Parse::Eyapp::Base qw(:all)
DB<2> !!ls *yp # There are two files with extension .yp in this directory
Parse.yp Treeregexp.yp
DB<3> $x = slurp_file('Parse', 'yp') # read the whole file
DB<4> p $x =~ tr/\n// # file Parse.yp has 1038 lines
1038

```

Function valid_keys

The call

```
valid_keys(%hash)
```

Returns a string with the keys of the %hash separated by commas:

```
pl@nereida:~/src/perl/YappWithDefaultAction/lib/Parse/Eyapp$ perl -wde 0
main::(-e:1): 0
DB<1> use Parse::Eyapp::Base qw(:all)
DB<2> %h = ( SCOPE_NAME => 'STRING', ENTRY_NAME => 'STRING', SCOPE_DEPTH => 'STRING')
DB<3> $x = valid_keys(%h)
DB<4> p $x
ENTRY_NAME, SCOPE_DEPTH, SCOPE_NAME
```

Function invalid_keys

It is called with two hash references:

```
DB<5> p invalid_keys(\%h, { SCOPE_NAME => 'a', ENTRY_NAME => 'b', SCOPE_DEPTH => 'c'})
ENTRY_NAME
```

It returns the first key in the second hash that does not appear in the first hash. See a more complete example:

```
pl@nereida:~/src/perl/YappWithDefaultAction/lib/Parse/Eyapp$ head -31 Scope.pm | cat -n
 1 package Parse::Eyapp::Scope;
 2 use strict;
 3 use warnings;
 4 use Carp;
 5 use List::MoreUtils qw(part);
 6 use Parse::Eyapp::Base qw(valid_keys invalid_keys);
 7
 8 my %_new_scope = (
 9     SCOPE_NAME      => 'STRING',
10     ENTRY_NAME      => 'STRING',
11     SCOPE_DEPTH     => 'STRING',
12 );
13 my $valid_scope_keys = valid_keys(%_new_scope);
14
15 sub new {
16     my $class = shift;
17     my %args = @_;
18
19     if (defined($a = invalid_keys(\%_new_scope, \%args))) {
20         croak("Parse::Eyapp::Scope::new Error!\n"
21             ."unknown argument $a. Valid arguments for new are:\n $valid_scope_keys")
22     }
23     $args{ENTRY_NAME} = 'entry' unless defined($args{ENTRY_NAME});
24     $args{SCOPE_NAME} = 'scope' unless defined($args{SCOPE_NAME});
25     $args{SCOPE_DEPTH} = '' unless defined($args{SCOPE_DEPTH});
26     $args{PENDING_DECL} = [];
27     $args{SCOPE_MARK} = 0;
28     $args{DEPTH} = -1; # first depth is 0
29
30     bless \%args, $class;
31 }
```

Function write_file

The call

```
write_file($filename, $textref)
```

simply opens a file with name \$filename writes in it the text referenced by \$textref and closes the file

Function numbered

The call

```
numbered($input)
```

Returns a string like `$input` but with lines numbered and the numbers correctly indented. See an example:

```
DB<1> use Parse::Eyapp::Base qw(:all)
DB<2> $input = "Another line!\n"x12
DB<3> $output = numbered($input)
DB<4> p $output
1 Another line!
2 Another line!
3 Another line!
4 Another line!
5 Another line!
6 Another line!
7 Another line!
8 Another line!
9 Another line!
10 Another line!
11 Another line!
12 Another line!
```

5 SEE ALSO

- *Parse::Eyapp*,

6 CONTRIBUTORS

- Hal Finkel <http://www.halssoftware.com/>
- G. Williams <http://kasei.us/>
- Thomas L. Shinnick <http://search.cpan.org/~tshinnic/>

7 AUTHOR

Casiano Rodriguez-Leon (casiano@ull.es)

8 ACKNOWLEDGMENTS

This work has been supported by CEE (FEDER) and the Spanish Ministry of *Educacion y Ciencia* through *Plan Nacional I+D+I* number TIN2005-08818-C04-04 (ULL::OPLINK project <http://www.oplink.ull.es/>). Support from Gobierno de Canarias was through GC02210601 (*Grupos Consolidados*). The University of La Laguna has also supported my work in many ways and for many years.

A large percentage of code is verbatim taken from *Parse::Yapp* 1.05. The author of *Parse::Yapp* is Francois Desarmenien.

I wish to thank Francois Desarmenien for his *Parse::Yapp* module, to my students at La Laguna and to the Perl Community. Thanks to the people who have contributed to improve the module (see CONTRIBUTORS in *Parse::Eyapp*). Thanks to Larry Wall for giving us Perl. Special thanks to Juana.

9 LICENCE AND COPYRIGHT

Copyright (c) 2006-2008 Casiano Rodriguez-Leon (casiano@ull.es). All rights reserved.

Parse::Yapp copyright is of Francois Desarmenien, all rights reserved. 1998-2001

These modules are free software; you can redistribute it and/or modify it under the same terms as Perl itself. See *perlartistic*.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Index

ACKNOWLEDGMENTS, 8

AUTHOR, 8

Changing the Behavior of Method-parametric Methods, 5

CONTRIBUTORS, 8

Examples of push method and pop method, 4

Function compute lines, 6

Function empty method, 2

Function insert function, 1

Function insert method, 1

Function invalid keys, 7

Function numbered, 8

Function pop method, 3

Function push method, 2

Function slurp file, 6

Function valid keys, 7

Function write file, 7

Hiding functions, 4

INTRODUCTION, 1

LICENCE AND COPYRIGHT, 8

NAME, 1

SEE ALSO, 8

SUBROUTINES, 1

SYNOPSIS, 1