

GetParameters.pm

Benjamin Bulheller

June 22, 2008

Contents

1	Introduction	2
2	Implementation	2
3	Usage Example	3
4	Define the minimum/maximum size of a list	4
5	Set a switch false instead of true	5
6	Define mandatory parameters	5
7	Differentiate between given parameters and default values	5

1 Introduction

`GetParameters.pm` is a Perl library parsing command line options given to a script. There are quite a few packages like this out there (`GetOpt`, `GetOpt::EvaP`, `GetOptions`), some of them more powerful than this. However, this package was created over three years while working on a PhD thesis and at that time a small package was needed which could quickly be extended about needed features and it was easier to write a new package than to change other programmer's code.

The main routine of this package reads in the command line parameters and returns a hash with the options as keys and the arguments as items. The options can be defined as:

- switches (true if given);
- strings, integers or real numbers,
- lists (read until the next option is reached, a minimum and maximum number of elements in the list can also be defined).

The parameters can be defined as optional or mandatory.

Lists can be given in batches, that is for example the items for the list `out` are collected:

```
scriptname -out file1 file2 file3 -a -b -c -out file4
```

The predefined option `-h` displays help screen, if given (unless `-h` is defined as one of the program's parameters). Wild cards (`*.pdb`, `chain?.pdb`) are expanded automatically and the list of files is returned.

2 Implementation

To call the routine, you need to provide references to a hash containing the definitions of the parameters, a hash for the parsed options and optionally predefined values for them and a string with the help message. Set them up like this (it is easier to define them as references in the first place):

```
#!/usr/bin/perl -w

use GetParameters;

my $Parameters = {          # create a reference to an anonymous hash
    f => "string",
};

my $Options     = {          # create a reference to an anonymous hash
    f => "example.txt",
};
```

```

my $Help      = "\n" . # a string holding the usage information
  "Usage:    \n" .
  "\n";

```

This for example defines the parameter `-f` which can contain any value, read in as a string. By default it contains the string `example.txt`. The following parameter types are defined:

- `switch`: this returns true if given
- `string`: just any string
- `integer`: an integer number
- `real`: a real number
- `list`: a list of strings by default
- `stringlist`: a list of strings
- `integerlist`: a list of integers
- `reallist`: a list of real numbers

`$Parameters` contains the possible parameters and their definition (type). `$Options` will after parsing contain the parameters and their values and can be used to define default values for the parameters (which will be overwritten if the parameter is given by the user or used otherwise).

`$Help` is a single string which contains the usage information, displayed if an input error is detected (an undefined option for example) or not parameter is given at all:

```

my $Help = "\nHelp string".
  "Displayed when -h is found\n".
  "or no command line parameters are given\n";

```

To actually parse the command line parameter, use

```

# parse the command line parameters
GetParameters ($Parameters, $Options, $Help)

```

3 Usage Example

Assuming the following setup

```
my $Parameters = {          # create a reference to an anonymous hash
  t => "string",
  v => "switch",
  f => "list",
};
```

“t” is defined as string, “v” as switch and “f” as list. The following command line given to the script `scriptname`:

```
scriptname -t somestring -v -f file1 file2 file3
```

would result into

```
$Options->{f} => [file1, file2, file3]
$Options->{t} => "somestring"
$Options->{v} => 1
```

If the parameter `-h` is found, the help screen is printed

All parameters, which cannot be attributed to one of the switches are collected in the array `$Options->{rest}`.

4 Define the minimum/maximum size of a list

The “list” declaration can be followed by “[min,max]” to define a range for the size of the list or “[value]” to define a required size of it. This also works for `{rest}` to catch a wrong number of parameters in addition to the ones sorted by switches.

Examples:

```
$Parameters{1} = "list[3]";      # exactly 3 elements
$Parameters{1} = "list[3]*";    # exactly 3 elements, mandatory
$Parameters{1} = "list[2,4]";  # 2, 3 or 4 elements
$Parameters{1} = "list[2,]";   # at least 2 elements
$Parameters{1} = "list[,3]";   # at most 3 elements
```

To end the list input and start e.g. with the files to process (which will be stored in `$Options->{rest}`), use the `--` parameter:

```
scriptname -range 150 250 -- *.cd
```

This is not needed of course, in case another parameter follows.

5 Set a switch false instead of true

If for some reason a switch was defined to be true before the actual command line parameters are parsed (e.g. because a configuration file was read), this can be done by adding "=0" to the parameter (without a blank!). Technically that also works with "=1" to set it true:

```
scriptname -v=0
```

6 Define mandatory parameters

An asterisk at the end of the type declaration marks a parameter as mandatory, for example

```
$Parameters{t} = "string*";
```

7 Differentiate between given parameters and default values

Sometimes it is necessary to determine whether a parameter was really given via the command line or whether it was just the default value that had been taken. For this purpose the `$Options->{GivenParams}` key is generated, which contains exactly the given, unaltered parameters.

Parameters given via the command line overwrite default parameters. It needs to be checked, whether default array elements had been declared, which will be erased if this parameter is given. On the other hand, list parameters may also be split, e.g. `prog -l as er ty -f file -l fg cx` would result in a list "l" containing the five given elements. The parameter's hash entry in `$WasGiven` will be true, if it had been passed via the command line before, i.e. then existing elements of this array will be kept. If the entry is false, then existing values were given as default parameters and will be erased.

add the arguments following the list parameter to `@FileList` the `--` parameter to end the list input is caught by the two regexes