# GetParameters.pm

Benjamin Bulheller
www.bulheller.com
webmaster.-at-.bulheller.com

July 11, 2009

## Contents

# 1 Introduction

`GetParameters.pm` is a Perl library for parsing command line options given to a script. There are quite a few packages like this out there (`GetOpt`, `GetOpt::EvaP`, `GetOptions`) and some of them may be 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.

For example, one thing that was needed was a single hash structure containing all given parameters instead of assigning the parameters to individual variables (`GetParameters` creates the hash `$Options` containing the parsed command line). This particularly comes in handy when all parameters have to be passed on to a subroutine, avoiding lots of global variables.

A desired type definition (apart from e.g. integer or string) was "file", which would automatically check for existence of the given file(s). This includes a check for multiple extensions if only the base name was given.

Option bundling (using `-abc` instead of `-a -b -c`) was not implemented, with the advantage of not needing double dashes `--` for long option names.

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, file names, integers or real numbers;
- lists (read until the next option is reached, a minimum and maximum number of elements in the list can be defined as well as the type of the elements).

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 use the routine, the references to three variables have to be provided: a hash containing the definitions of the parameters, a hash for the parsed options, and a string with the help message. Optionally it is possible to define default values for the options. It is easier to define these three variables as references in the first place:

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

use GetParameters;

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

my $Options    = {      # create a reference to a 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
- `file`: a file (checked for existence)
- `list`: a list of strings by default
- `stringlist`: a list of strings
- `integerlist`: a list of integers
- `reallist`: a list of real numbers
- `filelist`: a list of (existing) files

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

The command to parse the command line parameters is

```
GetParameters ($Parameters, $Options, $Help)
```

# 3 Usage Example

Assuming the following setup

```
my $Parameters = {      # create a reference to a 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. A plain `list` without type specification is a list of strings by default. If it were defined as a `filelist`, each given file would be checked for existence and rejected if not found. This also means that a `filelist` should only be used for files that must already exist.

All parameters which cannot be attributed to one of the switches are collected in the array `$Options->{rest}` (see section 5).

# 4 Parameter Types

## 4.1 Switches

If a parameter is defined as `switch` like this

```
$Parameters = {
   s => "switch"
}
```

it is true, if given, and false if not given by the user. In a script it could be evaluated like this:

```
if ($Options->{s}) { .... }
               else { .... }
```

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), it can be forced to be false by adding "=0" to the parameter (without a blank!):

```
scriptname -v=0
```

## 4.2 Integers and Real Numbers

The types `integer` and `real` define a number which is checked for being an integer or floating point value.

## 4.3 Files

A `file` parameter defines the name of a file which has to exist. To read the name of a not yet existing output file for example one would simply read a string. In order to additionally check for one or more extensions, these can also be defined in curly braces with a pipe symbol as delimiter:

- `f => "file"` and then `-f myfile`

  Looks for exactly the given string `myfile`

- `f => "file{txt}"` and then `-f myfile`

  looks for `myfile` and `myfile.txt`

- `f => "file{doc|txt}"` and then `-f myfile`

  looks for `myfile`, `myfile.doc` and `myfile.txt`

# 5 The $Options->{rest} Hash

The command line is parsed one parameter after another and the arguments are handled according to their definition. Options starting with a dash (-) are regarded as parameters, unless the dash is recognized as the minus symbol of a numeric value. All items following a parameter defined as a list will be added to this parameter's array until the next parameter is reached. All other items (not following a list parameter) are collected in $Options->{rest}. Consider the following parameter definition

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

and this command line:

```
scriptname stuff1 -t MyString stuff2 -v stuff3 stuff4 -f file1 file2 file3
```

The first item, stuff1 is not preceded by any parameter and is hence added to {rest} right away. -t is recognized as a parameter defined as string and hence only one value, MyString, is read while the one following it goes into {rest}. -v is a switch and only causes this hash key to be set true. The items following -f are all added to it since it is defined as a list. All this would result into this $Options hash:

```
$Options->{f}    => [file1, file2, file3]
$Options->{t}    => "MyString"
$Options->{v}    => 1
$Options->{rest} => [stuff1, stuff2, stuff3, stuff4]
```

The {rest} "parameter" may be given a definition as well. This may be necessary if a script is only given filenames or integer numbers without the need to use a specific option for that. This definition can be e.g. a filelist, reallist, integerlist, etc. but always has to be a list:

```
my $Parameters = {      # create a reference to a hash
  rest => "filelist",
};
```

By default it is assigned the value stringlist. Strings that contain only digits, a leading dash and periods, that is negative numbers, cannot be used as parameters as they would be mistaken for a negative number. In the following command line '-1.2' would therefore be treated as a negative number:

```
scriptname stuff -numbers 3.4 -1.2 7.71 -string perl
```

with `-numbers` being e.g. a `reallist` and `-string` a simple string would result into

```
$Options->{rest}    => [stuff]
$Options->{numbers} => [3.4, -1.2, 7.71]
$Options->{string}  => "perl"
```

# 6 Define Mandatory Parameters

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

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

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

This works with all defined lists:

- `list`: a list of strings by default
- `stringlist`: a list of strings
- `integerlist`: a list of integers
- `reallist`: a list of real numbers
- `filelist`: a list of (existing) files

A file list may also contain one or more possible extensions in curly braces and it does not matter in which order the extensions or range is given, i.e. `filelist[2]{txt}` and `filelist{txt}[2]` are equivalent.

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 in case another parameter follows.

# 8 Differentiate Between Given Parameters and Default Values

Parameters given via the command line overwrite default parameters. 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 and unaltered parameters.

The library itself needs this value if list parameters are split and given in multiple batches to the script:

```
prog  -l as er ty  -f file  -l fg cx
```

would result in a list "`l`" containing the five given elements. When the first three items are read the entry `l` in `$WasGiven` is false and the library will delete the default values, if any are defined. When the last two items are read, the `$WasGiven` entry is true and the three items already present in the array are recognized as 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.