NAME

synopsis – parse parameters and print an interpretable code
usage, match – parse parameters updating the Bash environment

SYNOPSIS

synopsis [*options*] *usages* **parse** *parameters*
usage [*options*] *usages* [**parse** *parameters*]
match [*parameters* **with**] [*options*] *usages*

DESCRIPTION

The purpose of this set of commands is to help the developer in the tedious and repetitive task of parsing the actual parameters of a shell command. They significantly reduce the effort that is required to implement a set of features commonly needed when implementing shell commands (programmed in Bash or other scripting languages). Specifically:

- to accept options and arguments independently of their ordering;
- to test the well-formedness of arguments either with the Bash built-in **test**, or with regular expressions or with the external command **file**;
- to rearrange arguments into separate variables according to their meaning, instead of having all parameters (either arguments or options) stuffed in a single structure $@ as in Bash;
- to automatically generate a standard help message when one of '-h', '--help', or '-?' is given as an option or when the command is invoked with malformed arguments;
- to automatically expand lists of options written in the condensed syntax (e.g. 'tar -cvzf' reads 'tar -c -v -z -f').

While the external command **synopsis** is a generic compiler, the shell functions **usage** and **match** are specialized for the target language Bash. Both encapsulate and manage a call of **synopsis**. Using these functions the output is not printed on stdout, but is directly interpreted by the internal command **source**, affecting in this way the current shell environment.

Please notice that except for the difference outlined above, the behaviour of **usage** and **match** is the same of **synopsis**. Therefore, the examples given in the rest of this manual are equally applicable to all the commands.

Description of synopsis

In the Unix world, the actual parameters of a command are a sequence of **options** or **arguments**. If the parameter begins with a dash is an option, otherwise is an argument. Arguments are themselves commonly separated in two sub-classes: the **option-related** arguments or the **stand-alone** arguments. For instance, in the following call of the well-known command tar:

tar -c -f archive.tar etc usr var

the first two actual parameters '-c' and '-f' are options, 'archive.tar' is an option-related argument (it is related to the option '-f'), while 'etc', 'usr' and 'var' are stand-alone arguments. An option-related argument immediately follow the option which is related.

Supposing only this kind of parameters, the command **synopsis** parses the given *parameters* according to the set of specified *usages*, then it builds a source code in the target language (by default Bash) and prints the generated code on stdout. The generated code will fit the given *parameters* into the appropriate capture variables defined by the *usages*. For instance, the call:

synopsis 'a?' 'b?' 'c?' '!ARG' **parse** -b -a /etc/profile

prints the following code on stdout:

OPTION_b=1
OPTION_a=1
ARG="/etc/profile"

Additionally, the command **synopsis** is able to automatically generate a minimal help about the usage of the script or the function where synopsis is used. Actually, synopsis reads the given *usages* in order to automatically generate the help message whenever one of the options '-h', '--help', or '-?' appears among the given *parameters*. The sole exception to this behaviour is when the developer explicitly define a different treatment of these options. For instance, the call:

synopsis -n mycommand 'o?' '-yes?' '?ARG1:d' '!ARG2:f' **parse** -h

prints the following message on stderr:

Usage: mycommand [-o] [--yes] [<dir>] <file>

Description of usage

The function **usage** is intended to be called either at the beginning of Bash function, simply using "$@" in place of *parameters*:

function foo {
usage 's?' 'o?OUT' '*ARGS' **parse** "$@"
...
}

or at the beginning of a Bash script (in this case the suffix 'parse "$@"' may be omitted):

#!/bin/bash
usage 's?' 'o?OUT' '*ARGS'
...

In both cases, the well-formedness of the actual parameters of the script or the function will be checked w.r.t. to the expected usage. When this matching succeed, the actual arguments are fitted into variables which will be used in the rest of the script or function.

For convenience, the developper may provide to **usage** additional informations for the user. Actually, **usage** appends its stdin on stderr in order to complete the short description about the correct usage of the script or the function. The suggested way to take advantage of this feature is to combine it with the here-document bash redirection (see 'man bash'):

```
#!/bin/bash
usage 's?' 'o?OUT' '*ARGS' <<EOF
Options:
  -s             something strange will happen activating this option
  -o filename    the output will be put in this file instead of stdout
EOF
...
```

Supposing that the name of the script is 'foo', when the user will call 'foo -h', the result printed on stderr will be:

```
Usage: foo [-s] [-o <arg>] [<arg>]*
Options:
  -s             something strange will happen activating this option
  -o filename    the output will be put in this file instead of stdout
```

Description of match

The meaning of **match** is very close to **usage**. Abstracting from their respective syntax, the main difference between them is the role of the stdin. While **usage** reads its stdin only for providing help informations to the user, **match** searches the actual parameters in the stdin when *parameters* and the keyword **with** are both omitted in the command line. In this way, **match** may be directly used in a Bash pipe, while **usage** needs to be used with the command **xargs**. For instance, the following pipe stores in the variable NEF the names of non-empty files (and in TRASH the other things):

find | match '*NEF:s' '*TRASH'

EXIT STATUS

The commands **synopsis**, **usage** and **match** exit with status 0 if *usages* is correctly defined and if the *parameters* successfully fit into the expected *usages*. They exit with a status greater than 0 if errors occur.

OPTIONS

The options of **synopsis**, **usage** and **match** which no need an argument, must be intended as boolean switches and they are disable by default.

Switches : [**−a**] [**−l**] [**−e**] [**−p**] [**−r**] [**−s**] [**−x**] [**−u**]
With argument : [**−d** *delimiter*] [**−n** *name*] [**−t** *language*]

Behaviour tuning

The following options alter the behaviour of the command.

−n *name*

Set the name of the script or function for which we are defining the usage. This name will appear in the help message explaining the correct synopsis. The commands **usage** and **match** automatically set this value to the name of the script or function in which they are contained.

−p When errors occur, print on the stderr a human readable message explaining briefly the expected *usages*.

−r Reverse both the *parameters* and the *usages* description. Activating this option, the sequence of actuals parameters will be read and interpreted from the last to the first. See the EXAMPLES section below.

−s Suppress parsing error messages.

−t *language*
Set the target programming language (only for **synopsis**). The output fitting the arguments into the capture variables will be written in this language. Currently, Bash is the unique supported language.

−x Exit the current shell on error (only for **usage** and **match**).

Bash code tuning

The following options may be used only when the target language is Bash. Otherwise are not meaningful. They allow to improve the Bash code resulting from the parsing process.

−a Use Bash arrays instead of simple variables for containing sequences of arguments (separated by a delimiter).

−l First declare local all capture variables. This option is suitable in writing **functions** in order to avoid conflicts with global homonymous variables.

−d *delimiter*
Delimiter for all Bash variables expected to contain a sequence of arguments. This kind of variables are related with a capture directive with the operators '*' or '+' (see the USAGE section below). The delimiter by default is the blank character, and the behaviour by default is to escape all blank characters occurring in the arguments fitting the variables. Defining a new delimiter one disable this behaviour: arguments will be placed just as they are in the variables and will be separated by the new delimiter. This option is not meaningful when the switch -a (arrays) is enable.

−e Export all capture variables in the environment (by the internal command **export**).

−u First unset all capture variables. This option is suitable in writing **scripts** in order to avoid conflicts with homonymous variables in the shell environment.

USAGE

This section describe the syntax and the meaning of the *parameters* and the *usages*.

Parameters specification

The *parameters* are simply an arbitrary sequence of options and arguments. However, one make here the most of the expansion mechanisms provided by Bash. Typically parameters are actually "$@" (especially for **usage**) or * (especially for **match**).

Usages specification

The *usages* formal specification determines:

- the allowed/requested options,
- the allowed/requested option-related arguments,
- the allowed/requested stand-alone arguments.

Additionally, it determines the name of variables which will be affected with arguments. We will call these variables **capture** variables.

DIRECTIVES

A capture variable may be defined by a capture *directive* which have the following three general forms:

directive ::= *optname operator*
 | *optname operator varname* [:*testexpr*]
 | *operator varname* [:*testexpr*]

The first case specify an allowed/required stand-alone option (i.e. an option which no requires to be followed by an argument). This kind of directive is implicitely associated to the variable OPTION_*optname* in order to register the occurrence of the option among the actual parameters. For instance, the directive 'y?' allows (because ?) the presence of the option '-y'. If this option occurs among the actual parameters, the variable OPTION_y will be affected to 1 in the generated code. Note that if the *optname* contains dashes, they will be converted in underscores (_). For instance, if the directive '-yes?' is given and the option '--yes' occurs among the actual parameters, the variable OPTION__yes will be affected to 1 in the generated code.

The second case specify an allowed/required option which requires to be followed by a argument. If given, the boolean expression *testexpr* will be applyed to the related argument. The directive capture the argument and generate the assignment of the *varname* with the related argument only if this argument pass the test. For instance, the directive 'o?DEST:f' define a variable DEST appointed to capture the argument related to the allowed (because ?) option -o, when both the option and its argument will be found among the actual parameters and when the argument is a regular file (the builtin 'test -f' is actually applied here).

The third case specify an allowed/required stand-alone argument. For instance, the directive '!DEST:d', requires (because !) the presence of a stand-alone directory pathname among the actual parameters.

OPERATORS

The operator of a directive specify if the presence of the stand-alone option, the option and its related argument, or the stand-alone argument is optional (?), mandatory (!), multiple (*), or multiple but almost one (+):

operator ::= **?** | **!** | ***** | **+** | *****(*range*) | **+**(*range*)
range ::= *n*,*m* | *n*, | ,*n* | *n*

A *range* may be provided to specify exactly the number of required matches of the directive. It may be a pair of integers separated by a comma (as for instance in the directive 'o*(1,3)DEST'), an integer followed or precedeed by a comma (as for instance in the directives 'o*(1,)DEST' and 'o*(,3)DEST'), or a single integer (as for instance in the directive 'o*(3)DEST') In the first case it specify both the minimal (1) and maximal (3) number of requested matches of the directive. In the second case, it specify only the minimal number of matches. In the third and forth cases, which are two equivalent forms, it specify only the maximal number of matches.

Note that the following equivalences hold:
? is equivalent to *(0,1)
! is equivalent to *(1,1)
+ is equivalent to *(1,)
+(n) is equivalent to *(1,n)
*(n) is equivalent to *(0,n)
+(n,m) is equivalent to *(n,m)

TEST EXPRESSIONS

Three kind of tests are available and may be combined with the standard boolean operators (and, or, not):

- tests based on the Bash built-in command **test**
- tests based on regular expressions
- tests based on the mime type returned by the external command **file**

testexpr may be provided to test the nature of an option-related or a stand-alone argument. The *testexpr* is a simple boolean expression built from the letters used in the syntax of the Bash built-in test (see 'man test'). These letters, which belong to the set [bcdefgGhkLOprsStuwx], may be combined in a boolean expression using the simple catenation (and), the symbol + (or), the symbol ! (not), and parenthesis. For instance, the directive '*DEST:!e+wf' matches only if the stand-alone argument doesn't exists (!e) or (+) it is a writable regular file (fw).

regexpr may be provided to check the syntax of an option-related or a stand-alone argument. In the syntax of the directive, the regular expression may be preceeded by '=' or '/'. The symbol '=' stands for a complete match, while the symbol '/' stands for a substring match. Note that the directive suffix =*regexpr* is equivalent to the suffix /^*regexpr*$.

For instance, the capture directive 'o?DEST:f=[a-z]*.txt' define a variable DEST appointed to capture the argument related to the option -o, if both (the option and its argument) will be found among the actual parameters. The symbol ? means that the presence of the pair -o <argument> among the actual parameters is optional, not mandatory. Because the directive contains a *varname*, if the option -o is encoutered, it must be followed by an <argument> which will be stored precisely in this *varname*. Moreover, this argument must be checked. Firstly, it must satisfy the Bash command 'test -f' (see 'man test' for more details). Secondly, it must exactly (because =) match the regular expression [a-z]*.txt. The directive **matches**, and the variable DEST is affected, only if the argument satisfy all conditions.

KIT SYNTAX

A directive is the basic form of *usages*. For instance, the following call of **usage** preclude all options allowing only stand-alone arguments for the script or function in which it appears:

usage '*ARGS' **parse** "$@"

Other relevant examples using an unique directive may be represented by scripts or functions with a unique option (for instance 's?') or a unique argument (for instance '?ARG' or '!ARG') or a sequence of arguments of the same nature (for instance '*DEST:d'). But combining directives into a sequence we obtain more actractive possibilities. A sequence of directives separated by blanks constitutes a **kit** of capture directives:

kit ::= *directive* [*kit*]

A kit allow to define a set of allowed and/or request options, option-related and stand-alone arguments abstracting the order of appearance in the command line. For instance, suppose to call the command **usage** at the beginning of a script or function named 'mycommand':

usage '-o?OUT' '!ARG' **parse** "$@"

With this specification, the following calls of mycommand will be equivalent (and correct):

mycommand -o output.ps input.dvi
mycommand input.dvi -o output.ps

In both cases, the variable OUT will be affected to the value 'output.ps' and the variable ARG to the value 'input.dvi'.

A kit is an ordered set of directives. When several directives refer to the same option (for instance 'o?D:d' and 'o?F:f') or refer to standalone arguments (for instance '*D:d' and '*F:f'), we will say that they have the same **domain.** In this case, the order of appearence in the kit became relevant because it express the priority between the directives of the same domain. Actually, parsing actual parameters with a kit, we will try to match a directive from the left to the right of the kit. For instance, the call:

usage '!DEST:d' '*ARGS' **parse** "$@"

fits the first directory pathname in the DEST variable and the rest of actual parameters in the variable ARGS. Both directives have the domain of stand-alone arguments. Inversing they in the call, we obtain an unsatisfiable usage, because the '*ARGS' directives will capture all actual parameters prevent '!DEST' to be matched.

Given an actual argument, we will say that a kit **captures** the argument if it contains a directive that capture it. We will say that a kit is **satisfied** by a sequence of actual parameters if it capture all the parameters of the sequence satisfying the directives as many time as requested by the operators ! (one time), + (almost one time), *(range) and +(range) (a number of times in the range). Otherwise we will say that the kit **fails**.

USAGE SYNTAX

An *usage* is a sequence of kits, separated by the keyword **then**:

usage ::= *kit* [**then** *usage*]

The comma (**,**) may be used instead of the keyword **then**. The purpose of an *usage* is to define a structure for the actual parameters: a first block of parameters is expected, then another block and so on. For instance, suppose the following call at the beginning of a command named 'mycommand' :

usage '+ORIG:f' then '!DEST:d' **parse** "$@"

This call means to start capturing parameters with the first kit '*ORIG:f' then, when this kit became unable to capture an argument but has been satisfied by the previous (almost an existing regular file will be required), continue trying to satisfy the second kit. This implies that the following calls:

mycommand /etc/fstab /etc/services /tmp
mycommand /etc/fstab /tmp /etc/services

will be not equivalent: only the first one will be correct w.r.t. the specified *usage*.

We will say that an *usage* is **satisfied** by a sequence of actual parameters if the sequence may be divided into sub-sequences wich satisfy the sequence of kits constituting the *usage*. Otherwise we will say that the *usage* **fails**.

USAGES SYNTAX

Several usages may be defined using the keyword **or**:

usages ::= *usage* [**or** *usages*]

The symbol **+** may be used instead of the keyword **or**. Given a sequence of actual parameters, the or-separated list of usages are tryed from the left to the right, searching for the first *usage* wich can be satisfied by the actual parameters.

EXAMPLES

(1) Forbidding options

Suppose a script requiring only stand-alone arguments. We can begin writing:

```
#!/bin/bash
usage '*ARGS' parse "$@"
...
```

In this way, all the given actual parameters will be put into the variable ARGS. Options, which are parameters beginning with a dash, will be forbidden because there isn't any directive starting with an option name. In the rest of the code, the developer will use the variable ARGS, for instance, as follow:

```
...
for i in $ARGS; do <something with $i>; done
...
```

(2) Classifying names in a directory hierarchy

The following command fits the sequences of the regular file names and directory names of the current directory hierarchy respectively into the variables F and D:

```
find | match '*F:f' '*D:d' '*TRASH'
```

ADVANCED USAGE

directive ::= *varname*

Example: a wrapper for find

This wrapper enable the user to call the command find with an unique argument. If this name doesn't exist in the current working directory, find returns an error. Instead, with this wrapper, this unique and non existing argument will be interpreted as the name to find in the directory hierarchy starting from the current working directory:

```
#!/bin/bash
usage '!NAME:!e' or 'CMDLINE'
[[ -z $NAME ]] && exec find $CMDLINE
exec find . -name "$NAME"
```

SEE ALSO

bash(1) - for the builtin test
getopt(1)

BUGS

The external command **synopsis** is a software implemented in OCaml, a functional and strictly typed programming language (see http://caml.inria.fr/), in order to minimize the number of bugs. A large number of bugs has been already discovered and corrected at the compile-time. Anyway, you are encouraged to use this program, which is delivered without any warranty, in order to discover inevitable survivors.

COPYRIGHT NOTICE