Command Line Processing

$Revision: 1.20 $

A. M. Sloane

Department of Computer Science
James Cook University
Townsville, QLD, 4811
Australia

Table of Contents

A processor generated by Eli may need to interact with its environment by way of the command line that invokes it. This manual describes the default behaviour and how you can perform more sophisticated command line processing. We will refer to Eli's command line processing support as *CLP*.

1 What is a command line interface?

When a processor is invoked it will be from an interactive or batch shell of some kind. A command line will be used to specify the name of the processor and any inputs that it needs. A few typical Unix command lines are:

```
cc -o fred.exe fred.c
vi fred.c
rlogin prep.ai.mit.edu -l rms
```

Note that a pipe command such as:

```
format doc.troff | lpr -Plaser
```

consists of two command lines because two programs are invoked.

In these examples various options are given to some of the tools via the command line. For example, `-o fred.exe` specifies that the output file of the C compilation should be called `fred.exe` rather than the default `a.out`. A major part of the job of a command line interface is to provide mechanisms for specifying which options are legal and allowing the processor to find out which ones the user actually supplied.

Other information can be provided on the command line in the form of *positional parameters*. For example, `fred.c` in the first two examples and `prep.ai.mit.edu` in the last are positional parameters. A command line interface is also responsible for providing access to positional parameters.

Unix provides access to the components of the command line for C programs via the `argc` and `argv` parameters to the `main` function. The facility described in this manual uses those parameters to provide higher-level access.

2 What happens by default?

The default command line interface provided by Eli assumes that the generated processor will have one input file and no options.

A processor `proc` generated with the default command line interface can be invoked in the following ways:

`proc` *No options or input files.* Input is assumed to come from standard input.

`proc` *input*
: *No options, one input file.* Input comes from *input*.

Any other way
: Signalled as an error.

3 Specifying the command line interface

If the default behaviour is not sufficient you can alter the command line interface using a file whose extension is `.clp`. The following sections show how to specify the varieties of options that the interface may need to handle.

3.1 The general format of the command line

The general format of a command line that can be recognised using a `.clp` specification is:

1. The program name, followed by
2. An arbitrary number of options, followed by
3. An arbitrary number of positional parameters.

A `.clp` specification describes the legal options and positional parameters.

3.2 Options that are either there or not

A boolean option is something like the `-S` (produce assembly code) or `-c` (compile only, don't link) options for the standard Unix compilers. That is, the option string is all that is needed.

A specification line of the form:

name string '`boolean`' '`;`'

describes a boolean option called *name* which is indicated by the command line string *string*. For example:

```
GenAssembly "-S" boolean;
CompileOnly "-c" boolean;
```

describe the compiler options mentioned above.

If a boolean option can appear more than once on the command line you should use the keyword `booleans` instead of `boolean`. Thus, the specification line:

```
WideListing "-w" booleans;
```

says that the user can give as many `-w` options as they like. For example, the processor can check the number provided and produce a listing of the appropriate width. (See the Berkeley Unix `ps` command.)

3.3 Options that have a value given with them

Some options need values. CLP supports two types of values: strings and integers. Typical options of these types would be the `-o` (generate output in the specified file) option of a Unix compiler, or the `-#` (print this many copies) option of a line printing program.

A specification line of the form:

name string type ';'

describes a value option called *name* which is indicated by the command line string *string* and accepts values of the specified type separated from the indication by whitespace. The valid types are `int`, `ints`, `string` and `strings`. The plural versions denote value options that may appear more than once on the command line.

For example:

```
OutputFile "-o" string;
NumCopies "-#" int;
```

describes the options mentioned above and

```
Command "-e" strings;
```

describes a repeatable option (see the Unix command `sed`, for example).

3.4 Options that are joined to their values

Value options as described in the previous section are separated from their values by white space. If this is not desired, *joined* value options can be used and no white space will be expected.

Examples of joined value options are `-` which is used by the Unix `head` program to designate how many lines to print (eg. `head -42 file`), and `-temp=` which is used by some compilers to describe where to put temporary files (eg. `pc -temp=/usr/tmp file.p`).

To describe a joined value option, use the specification line as described in the previous section with the keyword `joinedto` before the type specifier.

For example, the following specification lines describe the options mentioned above:

```
TmpFile "-temp=" joinedto string;
NumLines "-" joinedto int;
```

Joined value options can be repeated in the same way as normal value options. For example,

```
MacroPackage "-m" joinedto strings;
```

3.5 The order of option specification lines

Care must be taken when writing a CLP specification to ensure that the specification lines are ordered correctly. When processing the command line, CLP looks for options in the order that you specify them. A problem can occur if some option indication is a prefix of another option indication specified later.

For example, the code generated from the specification:

```
ModuleOption "-m" joinedto string;
ManOption "-man" string;
```

will never recognize the `-man` option because `ModuleOption` will be tested for first. Putting the specification of `ManOption` first will fix the problem.

3.6 Options that affect usage messages

CLP will automatically arrange for the usage message to be displayed when an erroneous condition is discovered. Sometimes it is nice to be able to implement an option that the user can use on purpose to get the usage message.

A specification line of the form:

`usage` *string* `;`

declares *string* to be such an option. Multiple usage options are allowed.

If a usage option is specified by the user when running the generated processor, the usage message is displayed and execution is terminated. All other options and/or parameters are ignored.

When a positional parameter denotes an input file (always the first parameter in the current version) an error message will be generated in the case that this file cannot be opened. The default message is of the form:

PROG ` : cannot open` *FILE* ` for reading`

where *PROG* is the name of the executable, and *FILE* is the name of the file that could not be opened.

This error message can be customised using a specification of the form:

`open` `error` `format` *string* `;`

String will then be used as the error message if a file cannot be opened. *String* can contain escape sequences of the form `%C` which are replaced before the message is output. The following table gives the valid escape sequences:

`%f`	Replaced by the name of the file that could not be opened.
`%p`	Replaced by the name of the program.
`%%`	Replaced by a single `%`.
`%C`	Where C is not `f`, `p`, or `%`, replaced by nothing.

For example, the default error message is specified using:

open error format "%p: cannot open %f for reading";

Sometimes it is useful to print the usage message when a file cannot be opened. For example, cases like this occur when the user mistypes an option which is then interpreted as a filename. By default, the usage message is not printed when a file cannot be opened. To cause it to be printed, use a specification line of the form:

‘`open`’ ‘`error`’ ‘`usage`’ ‘`;`’

3.7 Terminating the option list

For many processors it is useful to allow the user some way of saying that a command line string that looks like an option isn't really one. For example, this situation may arise when using the Unix `rm` command. If a user wants to remove a file called `-r` they would rather not have the filename interpreted as an option to recursively delete subdirectories.

One way of coping with this is to allow the user to type a special command line string that causes option recognition to terminate. For example, a user could type:

```
rm -i -- -r
```

to interactively (`-i`) delete a file called `-r`.

The termination facility of CLP lets you specify which string (or strings) should cause this behaviour. A specification line of the form:

‘`terminator`’ *string* ‘`;`’

declares *string* to be such a string. Multiple terminator specifications are allowed.

To get the behaviour described above for `rm` the following would be used:

```
terminator "--";
```

3.8 Parameters given by their command line position

Parameters that are interpreted according to their position on the command line are called *positional parameters*. For example, when invoking an editor the file to be edited is usually given as a positional parameter.

A specification line of the form:

name ‘`positional`’ ‘`;`’

describes a situation where a positional parameter is to be recognized and called *name*.

The plural form:

name ‘`positionals`’ ‘`;`’

can be used if a group of positional parameters is to be handled and grouped together.

Multiple positional parameters can be recognized by giving multiple specification lines of these kinds. Parameters will be recognized in the order that they are specified. If a plural form is present, it should be the last positional parameter specification line because it will represent all of the positional parameters from that point on. In that case, the processor generated will accept varying numbers of parameters. If no plural form is given, the processor will accept a fixed number of parameters equal to the number of singular positional parameter specification lines.

3.9 Input parameters

An input parameter is a special kind of positional parameter for use when the file named by the parameter value should be initialised for input by the rest of the program. For example, Eli-generated processors usually take a single input file name as a positional parameter.

A specification line of the form:

name ‘`input`’ ‘`;`’

describes a situation where a positional parameter is to be recognized, called *name* and used as an input file by the program.

Currently, only one input parameter may be specified. Also, if an input parameter is specified but the user does not specify a value for it, standard input is used. Future versions will probably allow this behaviour to be customized. For the time-being positional parameters can be used with custom opening of input files.

3.10 Documentation options and parameters

If the user specifies things incorrectly on the command line the usual practice is to produce a usage message and terminate execution of the processor. CLP will automatically produce a usage message in this fashion. It is possible to attach descriptions to the option and parameter specification lines to make this usage message more helpful to the user.

Each of the types of specification lines described above (except those for termination of option processing) can have a documentation string. Typical examples are:

```
CompileOnly "-c" boolean "Just compile, don't link";
MacroPackage "-m" joinedto strings "Load this macro package";
FileName input "File to be processed";
Others positionals "Other positional parameters";
```

4 Accessing the command line

A CLP specification is turned into code that arranges for command line information to be stored in C variables or a simple database. This code is automatically run by the processor startup code generated by Eli. During attribution of a structure tree you can access the variables or use access functions to obtain the command-line information.

The header file 'clp.h' will contain `extern` declarations for all values defined by the CLP-generated code. It should be included wherever these values must be accessed.

4.1 Accessing boolean options

Since boolean options do not have value information associated with them, all that is needed to represent them is a simple flag rather than a database object. We store the flag as a C integer variable whose name is the option name.

For example, given the specification:

```
GenAssembly "-S" boolean;
```

CLP will generate a variable called `GenAssembly`. Typical C code to test for this option would look like:

```
if (GenAssembly)
  printf ("GenAssembly specified\n");
else
  printf ("GenAssembly not specified\n");
```

4.2 Accessing options with integer values

To provide access to integer value options, CLP generates a database object which has the appropriate value as a property. The object is referred to by a key-valued variable named after the option. For example, given the specification:

```
NumCopies "-#" int;
```

CLP will generate a variable called `NumCopies`. The value of the variable can be used to access the option value using the `GetClpValue` property access function.

```
printf ("NumCopies value is %d\n", GetClpValue (NumCopies, 0));
```

Here `0` will be printed if the option was not specified by the user.

Alternatively, presence of the option can be tested for explicitly by testing the key:

```
if (NumCopies == NoKey)
  printf ("NumCopies not specified\n");
else
  printf ("NumCopies value is %d\n", GetClpValue (NumCopies, 0));
```

In this case the default value parameter in the `GetClpValue` call will never be used.

4.3 Accessing options with string values

Access to string value options is provided via a database object which has the appropriate value as a property. The object is referred to by a key-value variable named after the option. For example, given the specification:

```
TmpFile "-temp=" joinedto string;
```

CLP will generate a variable called `TmpFile`. `GetClpValue` is used to obtain the value which should be interpreted as a string table index (see Section 'Arbitrary-length character strings' in Library Reference).

```
#include "csm.h"

if (TmpFile == NoKey)
  printf ("TmpFile not specified\n");
else
  printf ("TmpFile value is '%s'\n", StringTable (GetClpValue (TmpFile, 0)));
```

4.4 Accessing options that appear more than once

The mechanisms described in the previous three sections only apply to options that can appear at most once on the command line. More complicated mechanisms are needed to access values associated with repeated options.

CLP uses linked lists of definition table keys to provide multiple value access. The lists are implemented using Eli's `List` module. See Specification Module Library.

Given the specification:

```
MacroPackage "-m" joinedto strings;
```

the list module lets you print the multiple values (via keys) as follows:

```
#include "csm.h"
#include "clp.h"

DefTableKey printkey (DefTableKey k)
{
    printf ("%s", StringTable (GetClpValue (k, 0)));
}

(void) MapDefTableKeyList (MacroPackage, printkey);
```

4.5 Accessing positional parameters

For each singular positional parameter specification line CLP generates a variable of the appropriate name which holds the key to a database object which has the string value of the positional parameter as a property. The value can be accessed as for string value options (see Section 4.3 [String value options], page 16).

CLP will always make sure that a positional parameter is specified and will arrange for a usage message to be printed otherwise. Thus there is no need to test that the database object described in the previous paragraph is defined. For example, given the specification line:

```
FileName positional;
```

the following code can be used to print the specified value:

```
#include "csm.h"

printf ("Filename given was '%s'\n", StringTable (GetClpValue (FileName, 0)));
```

When a plural positional parameter specification line is given the mechanisms used for repeated option values are used (see Section 4.4 [Repeated options], page 17). For example, if a processor can take multiple input files a specification like the following might be used:

```
FileNames positionals;
```

The following code can be used to print these parameters See Section 'Linear Lists of Any Type' in Specification Module Library.

```
#include "csm.h"
#include "clp.h"

DefTableKey printkey (DefTableKey k)
{
    printf ("%s", StringTable (GetClpValue (k, 0)));
}

(void) MapDefTableKeyList (FileNames, printkey);
```

Since an input parameter is just a special kind of positional parameter, its value can also be accessed as shown above.

5 Complete Grammar Listing

The following is the complete input grammar for the facilities described in this manual.

clp_spec ::= *params* .

params ::= *param* '`;`' / *param* '`;`' *params* .

param ::=
  identifier string type doc /
  identifier string '`joinedto`' *valtype doc* /
  '`usage`' *string* / '`terminator`' *string*.

type ::= '`boolean`' / '`booleans`' / '`positional`' /
  '`positionals`' / *valtype* .

valtype ::= '`int`' / '`ints`' / '`string`' / '`strings`' .

doc ::= / *string*.

Index

U

V

W