ECLOP
An Eiffel Command Line Option Parser

Paul Cohen
<paco@enea.se>

October 30, 2003
Covers release 0.1.0 of ECLOP

Contents

Chapter 1

Preface

This document describes ECLOP. ECLOP is a small set of Eiffel classes for parsing command line options. It provides Eiffel programmers with:

- A compact way of specifying valid command line options and a number of properties of the options, such as whether they are required or not, whether they take arguments or not and whether they are mutually exclusive or not.
- An easy way to parse given command line arguments as well as an easy to use interface for checking that all parsed command line arguments are valid options or option arguments and for accessing the parsed options and their arguments.
- Ready to use program invocation error messages as well as messages with information on program usage and program help.

ECLOP 0.1.0 was developed with ISE Eiffel 5.1 and 5.2 and MSVC++ 6.0 on Windows NT and 2000 machines. It is easy to adapt to SmartEiffel and Visual Eiffel and it is indeed the authors intention to do so. It should work on Linux as is. It should also work without modifications with later versions of ISE Eiffel.

The intention is that ECLOP is to support all the features of the POSIX and GNU `getopt` function and also the POSIX Utility Conventions. This is not entirely true of ECLOP 0.1.0. Furthermore ECLOP also has features that are not at all supported by the `getopt` function.

This document was written and prepared using GNU Emacs 20.7.1 and LaTeX2e[1] The BON static diagram in the appendix was created using Kim Waldén's BONsai Visio Solution[2]. The BON diagram was printed to a PostScript file, converted in GSview[3] 4.3 to an Encapsulated PostScript file and then imported with the LaTeX package `graphicx`. The PDF version of this document was created with `dvipdfm`.

[1] MiKTeX distribution. See: http://www.miktex.org/.
[2] See: http://www.bon-method.com
[3] See: http://www.cs.wisc.edu/ ghost/gsview/

Chapter 2

Introduction

Most operating systems provide a mechanism for both human users and programs to provide a number of arguments to a program when it is invoked. These arguments are called *program arguments* or *command line arguments* and are used to control the program's behavior. Two obvious examples of this is invoking programs from a Unix shell or a Windows command prompt:

```
>foo -a -x
```

In this example, `foo` is the name of a program and `-a` and `-x` are two program arguments. The set of program arguments, that are recognized by a program and which affect the behaviour of the program are called *command line options*, *program options* or simply *options* for short. Options are also commonly called *flags*. Program options usually begin with a dash `-`. Options can also be defined to take arguments. For example:

```
> foo -a -x -f bar.dat
```

The `-f` option takes the *option argument* `bar.dat`. In this case, the name of a file.

When a program is invoked, some tokens passed as program arguments may not be recognized options or option arguments. The program can either report these tokens simply as unrecognized options and then terminate, or it can interpretate those tokens as valid input. When interpreted as valid input they are called operands. A common convention is that any program argument following the special token `--` (two dashes) is interpreted as an operand. Any program argument that preceedes that special token are simply reported as being unrecognized options. For example:

```
> foo -a -x -f bar.dat -- x1.in x2.in y7
```

Here the three program arguments `x1.in`, `x2.in` and `y7` are valid operands. Usually, options control the behaviour of the program and operands are used to identify input to the program.

The general problem of describing and implementing software support for the handling of command line options can be divided into three parts:

- How does one specify options and what properties does one want to be able to specify for the options? This is a question of *specification*.
- How are options and arguments given on the command line supposed to be parsed? This is both a question of defining *parsing rules* and actually *implementing* them.
- What are the *recommendations* for option names and associated program behaviour? For example, the "-h" option is usually reserved for telling a program to show help information about its intended use.

Among existing software support for command line options there exists in the POSIX standard a specification of a function for command option parsing called `getopt` and a set of utility conventions that among other things define a valid argument syntax[1]. Among the many GNU utilities there exists the `getopt` functions (`getopt`, `getopt_long` and `getopt_long_only`)[2].

The POSIX specification and the Gnu implementation of the `getopt` function supports the definition and parsing of so called *short options*. A short option is simply a single alphanumeric character prefixed with a dash `-`. Short options can be grouped together as long as none of the short options is defined to take option arguments. For example:

```
> foo -axy
```

Here foo is invoked with the three short options `a`, `x` and `y`.

The GNU `getopt` function also supports some extensions via the two additional functions `getopt_long` and `getopt_long_only`. The most important extension is the support of so called *long options*. A long option is a sequence of two or more characters where each character is either alphanumeric or a dash `-`. For example:

```
> foo --use-compact-encoding
```

In the above example `--use-compact-encoding` is a single option. Long and short options may be used together and intermixed on the command line.

ECLOP works along the same general principles as the `getopt` function. You provide a specification of the valid options and their properties and you can then parse the program arguments passed to you by the operating system. However, there are two areas where ECLOP is more "advanced" than `getopt`. First, it enables you to specify more properties for each option than is possible with `getopt`. Secondly it provides more functionality in the form of features for accessing the actual parsed options and their arguments as well as errors encountered when parsing etc.

[1] See http://www.opengroup.org/onlinepubs/007904975/functions/getopt.html
[2] See http://www.gnu.org/manual/glibc-2.2.3/html_chapter/libc_25.html

Chapter 3

Using ECLOP

3.1 Overview and an example

ECLOP consists of six Eiffel classes of which two are intended for use by clients of the ECLOP library. The two classes are `COMMAND_LINE_SYNTAX` and `COMMAND_LINE_PARSER`. Using ECLOP typically consists of going through the following steps:

1. Create a number of STRING based option specifications and put them in an ARRAY [STRING].
2. Create an instance of COMMAND_LINE_SYNTAX and supply it with the ARRAY [STRING].
3. Create an instance of COMMAND_LINE_PARSER and supply it with your COMMAND_LINE_SYNTAX object. If the COMMAND_LINE_SYNTAX is invalid because it was provided an invalid option specification or inconsistent option specifications the COMMAND_LINE_PARSER will raise a precondition violation.
4. Tell your COMMAND_LINE_PARSER object to parse the ARRAY [STRING] of actual program arguments that your program was invoked with (see the ISE Eiffel class ARGUMENTS).
5. Query your COMMAND_LINE_PARSER object to see if all parsed program arguments were valid. If not, ask the COMMAND_LINE_PARSER object for an error message to present to the user.
6. If all the parsed program arguments were valid you can access them and any associated option arguments via the feature 'valid_options' in the class COMMAND_LINE_PARSER.

To get a feel for how ECLOP is intended to be used let us look at the following example[1].

```
class APPLICATION

creation
   make

feature {NONE} -- Initialization

    make is
            -- Run the program.
        local
            cls: COMMAND_LINE_SYNTAX
            clp: COMMAND_LINE_PARSER
            args: ARGUMENTS
        do
            create cls.make (option_specifications)
            create clp.make (cls)
            create args
            clp.parse (args.argument_array)
            exe_name := clp.executable_without_suffix
            if clp.valid_options.has ("-v") then
                print_version_info
            elseif clp.valid_options.has ("-h") then
                print (cls.program_help (exe_name, Void, Void))
            elseif not clp.invalid_options_found then
                file_names := clp.valid_options @ "-i"
                operate_on_files
            else
                print (clp.error_message)
                print (cls.program_usage (exe_name) + "%N")
                print ("Use -h/--help for more help." + "%N")
            end
        end

feature {NONE} -- Implementation

    print_version_info is
            -- Print version information.

    operate_on_files is
            -- Main body of program. Operates on 'file_names'.

    option_specifications: ARRAY [STRING] is
            -- The recognized options of this program
        once
            Result := <<"-v,--version#Version information.",
                    "-h,--help#Help on using this program.",
                    "-i,--input!=FILE!#Input file(s) to operate on.">>
        end

    file_names: LIST [STRING]
            -- Names of the files to operate on

    exe_name: STRING
            -- Name of this executable

end -- class APPLICATION
```

[1]The full sourcecode and Ace file for this example can be found under the directory `examples/simple_example_1`

3.2 Specifying the options

Specifying options is done by using Eiffel STRINGs and complying with the following syntax rules. An option specification in ECLOP is either an *option name specification* or an *mutual exclusivity specification.* An option name specification has the form:

```
``-x,--option-x!=XARG!#Option x.  XARG defines the X-ness.''
```

The option name specification must contain at least a short option name or a long option name or possibly both. In the last case they must be separated by a single `,` character. The short option name must begin with the `-` character and must be followed by exatly one alpha-numeric character. The long option name must begin with the `--` characters and must then be followed by at least two characters which may be alpha-numeric and/or the `-` character. If both a short and long name are specified they are considered to be *synonym* names for the same option.

After the short or long option name specification a number of optional qualifiers may be used to specify the properties of the option. The possible qualifier combinations are:

- `!` The option is required.
- `!=` The option is required and takes 0 or more option arguments.
- `!=!` The option is required and takes 1 or more (required) option arguments.
- `=` The option is optional and takes 0 or more options.
- `=!` The option is optional and takes 1 or more (required) options.

Immedeately following the `=` qualifier a formal name for the option argument may be given (`XARG` above). This name is used by the class COMMAND_LINE_SYNTAX to generate usage and help messages for the option.

Finally, after the qualifiers, if any, the `#` character may be used to indicate the start of the option description. After the `#` character any printable characters are allowed including the newline `'%N'` character. The option description is also used by the class COMMAND_LINE_SYNTAX to generate the help messages for the option.

A mutual exclusivity specification has the following form:

```
``(-x|-y|-z)''
```

The mutual exclusivity specification is simply a listing of mutually exclusive options. Note the surrounding parenthesis characters and the use of the `|` character as an "or" symbol.

When the options have been specified you can create an instance of a COMMAND_LINE_SYNTAX. The COMMAND_LINE_SYNTAX will parse and validate the array of option specifications and after it is created you can use the query `is_valid` to check that all the option specifications you have given are valid.

Here is a little more extensive example[2] of a valid array of option specifications:

```
option_specifications: ARRAY [STRING] is
        -- The recognized options of this program
    once
        Result := <<''-v,--version#Print version information.'',
                    ''-h,--help#Print help on how to use the program.'',
                    ''-a#Use algorithm A. Can't be used with -b.'',
                    ''-b#Use algorithm B. Can't be used with -a.'',
                    ''(-a|-b)'',
                    ''-i!=INPUTFILE!#Use INPUTFILE as input.'',
                    ''-o=OUTPUTFILE!#Use OUTPUTFILE as ouput. Default is stdout.''>>
    end
```

Once we have a COMMAND_LINE_SYNTAX object we can use it to get usage and help messages. The feature:

```
program_usage (program_name: STRING): STRING
```

will, if invoked with:

```
program_usage (''foo'')
```

return the following string:

```
''Usage: foo -i=INPUTFILE [-o=OUTPUTFILE -a -b -v -h]''
```

The feature:

```
program_help (program_name, usage, program_description: STRING): STRING
```

will, if invoked with:

```
cls.program_help (''foo'', Void, ''Does foo on a file.'')
```

return the following string:

```
''Usage: foo -i=INPUTFILE [-o=OUTPUTFILE -a -b -v -h]%N
  foo -- Does foo on a file.%N
  -a                        Use algorithm A. Can't be used with -b.%N
  -b                        Use algorithm B. Can't be used with -a.%N
  -h, --help                Print help on how to use the program.%N
  -i=INPUTFILE              Use INPUTFILE as input.%N
  -o=OUTPUTFILE             Use OUTPUTFILE as ouput. Default is stdout.%N
  -v, --version             Print version information.''
```

Note that the options are sorted in alphabetical order. If we invoke the same feature, giving a `usage` STRING but no `program_description` with:

```
cls.program_help (''foo'', ''foo [OPTIONS]'', Void)
```

we get:

```
''Usage: foo [OPTIONS]%N
  -a                        Use algorithm A. Can't be used with -b.%N
  -b                        Use algorithm B. Can't be used with -a.%N
  -h, --help                Print help on how to use the program.%N
  -i=INPUTFILE              Use INPUTFILE as input.%N
  -o=OUTPUTFILE             Use OUTPUTFILE as ouput. Default is stdout.%N
  -v, --version             Print version information.''
```

[2]The full sourcecode and Ace file for this example can be found under the directory `examples/simple_example_2`

3.3 Parsing program arguments

Creating a COMMAND_LINE_PARSER is easy. Use the creation feature:

```
make (cls: COMMAND_LINE_SYNTAX) is
        -- Create a new command line parser using the syntax ‘cls’.
    require
        syntax_not_void: cls /= void
        syntax_is_valid: cls.is_valid
    ensure
        syntax_is_set: syntax = cls
    end
```

Note the precondition that requires the COMMAND_LINE_SYNTAX to be valid. Now we can start parsing program arguments with:

```
parse (args: ARRAY [STRING])
```

After calling `parse` we can check that all the parsed program arguments are valid with the query feature `invalid_options_found`. If that query returns `True` we can get an error message to present to the user with the query feature `error_message`. For example, using the option specifications in the previous section:

```
clp.parse (<<''foo'', -i>>)
```

will produce the following `error_message`:

```
''foo: option requires an argument: -i''
```

However, if `invalid_options_found` returns `False` we can access the valid options with:

```
valid_options: HASH_TABLE [LIST [STRING], STRING]
        -- Information on the parsed valid options. The possible
        -- contents is a mix of:
        -- 1) A hash key consisting of a valid option found when
        --    parsing and a hash item that is Void.
        -- 2) A hash key consisting of a valid option found when
        --    parsing and a hash item consisting of a list
        --    arguments found when parsing.
        -- Empty if no valid options were encountered.
```

For example:

```
clp.parse (<<''foo'', ''-i'', ''bar.dat''>>)
```

enables us to check if the `-i` option was given and to access the option arguments for `-i` with:

```
args: LIST [STRING]
...
if clp.valid_options.has (''-i'') then
    args := clp.valid_options. @ ''-i''
...
```

3.4 Parsing conventions

The parser implements the following conventions:

- Command line arguments are options if they begin with a dash `-`.
- Multiple short name options may follow a dash `-` in a single token if the options do not take arguments. Thus, `-abc` is equivalent to `-a -b -c`.
- Short option names are consist of a single dash `-` followed by a single alphanumeric characters.
- A short name option and its argument may or may not appear as separate tokens. (In other words, the whitespace separating them is optional.) Thus, `-o foo` and `-ofoo` are equivalent.
- The token `--` terminates all options; any following arguments are treated as non-option arguments, even if they begin with a dash `-`.
- Any option that does not match a specified option is considered an invalid and unrecognized option, unless that option is a token that comes after the `--` token. In that case it is considered a (valid) operand.
- Options may be supplied in any order, or appear multiple times. The parser will only report the option as chosen or not. Multiple option arguments are merged into a single list of arguments for the option.
- Long options consist of `--` followed by at least two alphanumeric characters and dashes `-`. Option names are typically one to three words long, with dashes `-` to separate words. Users can abbreviate the option names as long as the abbreviations are unique.
- To specify an argument for a long option, write `--name=value`.
- A single dash `-` not followed by any other character (except a whitespace) is reported by the parser as `has_single_dash`. If the program uses operands for specifying file input or ouput, convention dictates that a single dash `-` means that the program is to read from standard input or write to standard output.

3.5 Recommended options

Thera are a number of standard options that all programs should recognize. For example it is good to have an option that makes the program print version information about itself and another option that makes the program print help on how to use it. Obviously it would be nice if all programs used the same naming conventions for these basic options. This would make it much easier for users of the programs to remember the basic options.

Examples of some existing conventions and recommendations are:

- `-h` and `--help`. Print help on using the program.
- `-v` and `--version`. Print version information. Note that -v is often also used to mean that the program is to run in "verbose" mode.
- `-i`. Input file(s).
- `-o`. Output file.

Chapter 4

Design objectives

There already exists two other Eiffel libraries for specifying and parsing command line options – Optimus[1] and gargs[2]. So why write yet another library for this?

Well, it's fun! Seriously, I had different design objectives than those implemented in those libraries. My main objectives were:

- An pure Eiffel solution. I did not originally want to specify the options and their properties in a separate specification language which then is used to generate Eiffel code to be used for the actual parsing. Using compiler compiler tools such as lex and yacc is of course useful for more complex specification languages but in this case only introduces uneccessary complexity in the implementation and also unneccesary dependencies in an automated build environment. Eventually, of course, the ECLOP specification language may become so complicated it may motivate a refactoring using fx. Gobo lex and yacc.
- I wanted a very compact way of specifying the options, along the lines of the POSIX and Gnu getopt function, by simply passing a string or list of strings to the parser. I did not want the user of ECLOP to have to create distinct option specification objects for each option and then having to set attributes on each object etc.
- I wanted support for generating usage and help messages which is especially useful when you have large numbers of options.

I felt that both Optimus and gargs did not meet these objectives to my satisfaction. Futhermore they were written for SmartEiffel and VisualEiffel resepectively and we were using ISE Eiffel in the project where ECLOP was originally developed.

[1] See http://userpages.umbc.edu/ greagl1/optimus/optimus.html
[2] See http://www.object-tools.com/manuals/ve/tools/gargs.html

Chapter 5

Comments on the implementation

Please refer to Appendix A.1 which contains a BON static diagram over the classes in ECLOP when reading this chapter. Please be aware that since this is a beta release, the code and the implemented algorithms are far from optimal - in all senses of the word.

When creating a COMMAND_LINE_SYNTAX object it must be passed an ARRAY of STRINGs containing option and mutual exclusivity specifications. These are then parsed as follows:

1. It first creates a new ARRAY and puts all option specification STRINGs at the beginning and all mutual exclusivity specification STRINGs at the end of the ARRAY.
2. It then iterates over the new ARRAY and creates a HASH_TABLE of OPTION_SPECIFICATIONs hashed by option name. When all option specifications have been dealt with MUTUAL_EXCLUSIVITY_OPTIONs are created. These are used to update each OPTION_SPECIFICATION with information on which other OPTION_SPECIFICATIONs it is mutually exclusive with.

The parsing of command line arguments in the class COMMAND_LINE_PARSER follows a muliple pass scheme as follows:

1. Iterate over the list of actual the command line argument STRINGs and for each argument create a PARSED_COMMAND_LINE_ARGUMENT object. Each such object may represent one of the following:
   - A single option.
   - Multiple options. Eg. `-abc` where `-a`, `-b` and `-c` are separate options.
   - An option and an option argument. Eg. `-ifoo.txt` where `-i` is a short option name and `foo.txt` is its argument or `--input=foo.txt` where `--input` is a long option name and `foo.txt` is its argument.
   - A single option argument.
   - A single dash `-`.

- A double dash `--`
- An operand.

2. Iterate over the list of PARSED_COMMAND_LINE_ARGUMENTs and create a list of PARSED_OPTIONs.

3. Iterate over the list of PARSED_OPTIONs and validate them against the COMMAND_LINE_SYNTAX.

Appendix A

BON diagram and class interfaces

A.1 BON static diagram

Below a static BON diagram over how ECLOP is to be used is shown. The cluster CLIENT_APPLICATION containing the class APPLICATION represents a client program using ECLOP. The important point to note is that the client application only uses the COMMAND_LINE_SYNTAX and the COMMAND_LINE_PARSER classes, the other ECLOP classes are used internally by these two classes.

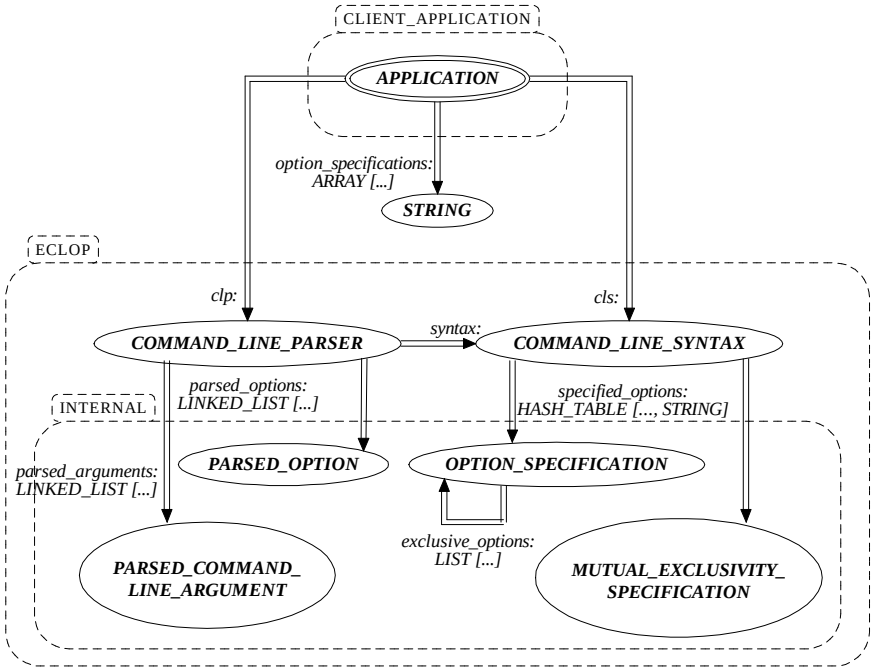

Figure A.1: BON static diagram

A.2 Interface of COMMAND_LINE_SYNTAX

```
class interface
    COMMAND_LINE_SYNTAX

create
    make

feature {NONE} -- Initialization

    make (spec: ARRAY [STRING]) is
            -- Specify a new command line syntax as specified by
            -- the specifications in ‘spec’.
        require
            spec_not_void: spec /= Void

feature -- Status report

    invalid_specifications: HASH_TABLE [STRING, STRING]
            -- Table of invalid option specificationms. Void if
            -- none exists. Hash keys are either a) textual option
            -- specifications or b) mutual exclusivity
            -- specifications. The hash elements are error
            -- descriptions

    is_valid: BOOLEAN
            -- Is this a valid specification?

feature -- Access (Application output messages)

    program_help (program_name, usage, program_description: STRING): STRING
            -- Program help string for presenting to user. The
            -- ‘program_name’ is mandatory. If no ‘usage’ string is
            -- supplied the value of ‘program_usage (program_name)’
            -- is used.
        require
            program_name_not_void: program_name /= void
            program_name_not_empty: program_name.count > 0

    program_usage (program_name: STRING): STRING
            -- Program usage string for presenting to user.
        require
            program_name_not_void: program_name /= void
            program_name_not_empty: program_name.count > 0

end -- class COMMAND_LINE_SYNTAX
```

A.3 Interface of COMMAND_LINE_PARSER

```
class interface
    COMMAND_LINE_PARSER

create
    make

feature {NONE} -- Initialization

    make (cls: COMMAND_LINE_SYNTAX) is
            -- Create a new command line parser using the syntax
            -- `cls'.
        require
            syntax_not_void: cls /= Void
            syntax_is_valid: cls.is_valid
        ensure
            syntax_is_set: syntax = cls
        end

feature -- Access

    ambigous_options: LINKED_LIST [STRING]
            -- List of ambigous options. Empty if no
            -- ambigous options were encountered.

    ambigous_options_found: BOOLEAN
            -- Where any ambigous (long) options found when
            -- parsing?

    error_message: STRING
            -- An error message with information on errors
            -- encountered when parsing
        require
            invalid_options: invalid_options_found

    executable: STRING
            -- The name of the current executable stripped of all
            -- preceding directory names and separators

    executable_path: STRING
            -- The name of the current executable in the form of a
            -- full path including preceding directory names and
            -- directoy name separators

    executable_without_suffix: STRING
            -- The name of the current executable stripped of all
            -- preceding directory names and separators and without
            -- any eventual suffix like ".exe"

    invalid_options_found: BOOLEAN
            -- Where any invalid options found while parsing?
        ensure
            Result implies (required_options_missing or
                            options_with_missing_arguments_found or
                            unrecognized_options_found or
                            mutually_exclusive_options_found or
                            ambigous_options_found or
                            invalidly_grouped_options_found)
```

```
invalidly_grouped_options: LINKED_LIST [STRING]
        -- List of invalidly grouped short options. A short
        -- option may not be grouped with other short options
        -- if it takes option arguments

invalidly_grouped_options_found: BOOLEAN
        -- Where any invalidly grouped short options found when
        -- parsing?

missing_options: LINKED_LIST [STRING]
        -- List of missing options. Empty if no missing options
        -- were encountered.

mutually_exclusive_options: HASH_TABLE [STRING, STRING]
        -- Table of mutually exclusive options. Empty if no
        -- such options were encountered. Hash keys are
        -- individual option names and the items are strings
        -- containing a comma-separated list of all the parsed
        -- options with which the given option is mutually
        -- exclusive.

mutually_exclusive_options_found: BOOLEAN
        -- Where any mutually exclusive options found when
        -- parsing?

operands: LIST [STRING]
        -- List of operands, ie. all arguments following the
        -- "--" argument, if any

options_with_missing_arguments: LINKED_LIST [STRING]
        -- List of options with missing arguments. Empty
        -- if no options with missing arguments were encountered.

options_with_missing_arguments_found: BOOLEAN
        -- Where any options requiring arguments lack arguments
        -- when parsing?

required_options_missing: BOOLEAN
        -- Was any required options missing when parsing?

single_dash_encountered: BOOLEAN
        -- Was a single dash encountered? This means that if
        -- your program uses operands to represent files to be
        -- opened for either reading or writing, you should now
        -- read or write, as the case is, from standard input or
        -- standard output respectively.

unrecognized_options: LINKED_LIST [STRING]
        -- List of unrecognized options. Empty if no
        -- unrecognized options were encountered.

unrecognized_options_found: BOOLEAN
        -- Where any unrecognized options encountered when
        -- parsing?
```

```
    valid_options: HASH_TABLE [LIST [STRING], STRING]
            -- Information on the parsed valid_options. The possible
            -- contents is a mix of:
            -- 1) A hash key consisting of a valid option found when
            --    parsing and a hash item that is Void.
            -- 2) A hash key consisting of a valid option found when
            --    parsing and a hash item consisting of a list
            --    arguments found when parsing.
            -- Empty if no valid_options were encountered.

feature -- Basic operations

    parse (args: ARRAY [STRING])
            -- Parse the ‘args’.
        require
            args_not_void: args /= void
            args_contains_at_least_executable_name: args.count > 0

feature -- Test & Debug

    pretty_print_of_valid_options: STRING
            -- Pretty presentation of all information on the
            -- most recently parsed (valid) options

invariant

    valid_syntax: syntax.is_valid

end -- class COMMAND_LINE_PARSER
```