CLIgen Manual

CLIgen version 7.0

Olof Hagsand

May, 2024

Contents

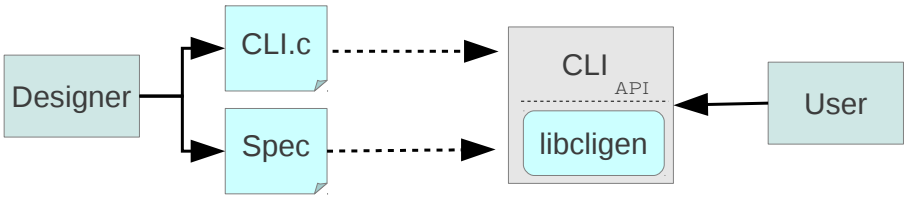

Figure 1: *CLIgen usage: a designer implements a CLI by specificying the syntax in a specification file and the CLI source code using the CLIgen API.*

1 Introduction

CLIgen builds interactive syntax-driven *command-line interfaces* in C from a high-level syntax specification. Interactive CLIs are often used in communication devices such as routers and switches. However, any devices with a textual, syntax-driven, command-based interface can use CLIgen to make CLI programming easy.

CLIgen takes a syntax specification as input, generates a tree representation of the syntax, and provides an interactive command-line tool with completion, help, modes, etc.

A designer formulates the command-line syntax and writes callback functions in C to implement the semantics of the commands.

A good starting point is the hello world example with a simple syntax specification ("hello world") and a callback with a print statement, which produces the following CLI executable:

```
> ./cligen_hello
hello> hello world
Hello World!
hello>
```

The complete `cligen_hello` C application is included in the source code distribution.

Figure 1 shows a typical workflow when working with CLIgen. A designer specifies the CLI syntax by editing a CLIgen specification file and writing a C-program. When users access the CLI at runtime, the specification file is loaded and the CLI uses the API of the CLIgen library to interpret the commands.

An example of a CLIgen specification file of the hello world application is:

```
prompt="hello> "             # Assignment of prompt
hello("Greet the world"){    # 'hello' command with help text
  world, cb("Hello World!"); # 'world' command with callback
}
```

The specification above shows examples of the following key ingredients of a specification:

- *Command syntax* specifies the syntax of the actual commands, and are the main part of a specification. The command syntax is fully described in Section 2.

- *Callbacks* are functions called when a command has been entered by a user. You may specify an argument to a callback. Callbacks are a part of the API described in Section 5.
- *Assignments* are used to set properties of the CLI and its commands, such as prompts, visibility, etc. Assignments are either global or per-command.
- *Help text* provides help text for individual commands.
- *Comments* begin with the '#' sign, at start of a line or at the beginning of a word.

The following sections will describe all aspects of designing CLIgen application. Programming using the CLIgen API in C is described in Section 5.

2 Command syntax

The command syntax consists of a combination of keywords, variables and operators:

- *Keywords* are constant strings representing fixed command words.
- *Variables* are placeholders for user-defined input.
- *Operators* are used to combine keywords and variables in different ways. Operators include 'choice', 'option', 'sequence', etc. Operators are further described in Section 4.

For example, the command syntax `ip tcp <uint16>;` have two keywords: `ip`, and `tcp` and one variable: `<uint16>`. They are combined in a *sequence*, that is, the CLI expects them to be typed one after the other.

2.1 Keywords

The simplest syntax consists only of keywords. Such a syntax can be specified as follows:

```
aa bb{
  ca;
  cb;{
    dd;
    ee;
  }
}
ff;
```

A CLI loaded with the specification above accepts the following strings as input:

```
aa bb ca
aa bb cb
aa bb cb dd
aa bb cb ee
ff
```

Note the following:

- Newlines are not significant, except after comments. This means that another way of specifying the syntax above is: `aa bb{ca;cb;{dd;ee;}}ff;`.
- Keywords specified one after another is a sequence. Example: `aa bb;`. An alternative of expressing the same syntax is: `aa{bb;}`
- Semicolon terminates a complete command. This means that `aa bb cb` is accepted as a complete command, but not `aa bb` in the syntax above.
- Semicolons also act as a choice, you can choose either `dd` or `ee` in the syntax above.
- Keywords can also be specified using variables: `<string keyword:aa>`, there are some advantages with this which may get apparent when programming using the API (see Section 5).
- The syntax above can be written in a more compact way, such as `aa bb (ca|cb [dd|ee]);ff;`. This is described more in Section 4

2.2 Runtime behaviour

A CLI with the syntax above will present the user with a list of commands. On the top-level, only `aa` or `bb` may be chosen when a question mark is entered:

```
> '?'
  aa
  gg
```

If the user prints an 'a', followed by a 'TAB', the CLI performs completion to `aa` since there is only one alternative:

```
> a'TAB'
> aa 'TAB'
> aa bb 'TAB'
> aa bb c'TAB'
ca                  cb
> aa bb c
```

In the example, two more `TAB`s are entered, one for each command level and completion continues until the commands are not unique. In the last `TAB`, the CLI shows the available commands (`ca` and `cb`).

As long as a command is unique it is not necessary to complete the whole string. For example, the following two strings are equivalent from the CLIs perspective:

```
> aa bb cb dd
> a b cb d
```

Before finishing a command (with return), a unique command must be selected. The CLI gives an error code if the command is unknown, ambiguous or incomplete:

```
> a
CLI syntax error in: "a": Incomplete command
> aa bb c
CLI syntax error in: "aa bb c": Ambiguous command
> aa bb dd
CLI syntax error in: "aa bb dd": Unknown command
```

2.3 Escaping

In the CLIgen runtime, some characters have special meaning, such as `'?'`. You can *escape* characters with backslash(`'\'`) so that for example `'?'` can appear in a keyword or value:

```
> a\?b # Gives the string "a?b"
> a\\b # Gives the string "a\b"
```

2.4 Help texts

Help texts are given within parenthesis following a keyword or variable. The help text appears when you invoke the help command '?' in the CLI runtime.

For example, assume the following syntax:

```
ip("The IP protocol"){
  udp("The UDP protocol") <ipaddr>("IPv4 address");
  tcp("The TCP protocol") <uint16>("Port number");
}
```

If a user has typed 'ip ' and '?', the following help text appears:

```
cli> ip '?'
  tcp                     The TCP protocol
  udp                     The UDP protocol
```

2.5 Callbacks

When a unique command has been selected, a callback may be called. Callbacks are typically associated with commands using the file syntax:

```
aa bb{
  ca,fn1("ca");
  cb,fn1("cb");{
    dd,fn2();
    ee,fn3((int)42);
  }
}
```

In the example, the function `fn1` is called with `"ca"` as argument if `aa bb ca`, is selected. The same function is called also if `aa bb cb` is selected, but with another argument. For other commands, `fn2` is called without argument, and `fn3` is called with the integer argument `42`.

Note that callbacks may only be associated with terminal commands. For example, `aa bb` may not have a callback function.

The details on how to write callback functions, such as `fn1` - `fn3` is described in detail in Section 5.

2.6 Assignments

You can assign values to *global* and *local* variables. Global variables are valid for the whole syntax, while local variables only apply to a single command.

In the current release, there are three pre-defined local variables:

- `hide` : specifies that a command is not visible in auto completion, meaning when listing or completing commands with '?' and 'TAB'. This can be useful if there are commands that should be known only by expert users.

In the following example, `aa bb cb dd` and `aa bb cc` are not visible in database and/or auto completion depending on the variable used as mentioned above :

```
aa bb{
  ca;
  cb, hide{
    dd;
  }
  cc, hide;
}
```

A *global* variable is assigned on the top-level. There is currently only one pre-defined global variable (`treename` as described in the next section). But it is easy for a programmer to implement a global variable and define semantics to it.

The tutorial application supports prompt and comment character:

```
prompt="cli> ";              # Assignment of prompt
comment="#";                 # Same comment as in syntax
```

Section 5.4 describes how the C-API can be used to define semantics for a global variable.

2.7 Trees

CLIgen can handle multiple syntax trees. A user can switch between trees (i.e., change CLI mode), or extend a tree with a sub-tree (using tree-references).

A tree is named using a special global `treename` assignment. The following example shows two syntax trees: `tree1` and `tree2`.

```
treename="tree1";
  x{
    y;
  }
treename="tree2";
  z{
    x;
  }
```

When parsing the syntax above, a CLIgen tree-list consisting of two trees will be created. By default, the first parsed tree is the active tree.

```
changetree <tree:string>, changetree("tree2");
```

Note that the `changetree` callback needs to be implemented as a callback function in C to actually change the syntax mode. Such a callback is implemented in the tutorial and is also described in more detail in Section 5.

Note, the `treename` feature is not available in Clixon, only in stand-alone CLIgen.

2.7.1 Tree references

A CLIgen syntax tree may reference another tree as an extension using the reference operator `@`.

The following specification references itself:

```
treename="T";
foo;
recurse @T;
```

which means that the following sentences are valid commands:

```
foo
recurse foo
recurse recurse foo
recurse recurse recurse foo
```

and so on.

Callbacks can be parametrized when using tree references. This means that you can specify which callback to use in the reference of the tree. This means that different callbacks can be called depending on how you reference the tree.

The following examples shows a main tree and a sub-tree.

```
treename="main";
add @sub, addfn();
del @sub, delfn();
treename="sub";
x{
  y, fn("a1");
}
```

The main tree references the subtree twice. In the first reference, `addfn("a1")` is called when invoking the command `add x y`. In the second instance `delfn("a")` is invoked when invoking the command `del x y`.

It may be useful with functional substitution as shown above when the subtree represents a large common data-modeling sub-tree, where the data (x y) is the same but the operation(add/del) is different.

2.7.2 Tree reference callback parameters

In the example above, the tree reference have no parameters (`add();` and the local parameter. If instead the reference does have some parameters a special parameter appending mechanism is used.

Assume the following tree spec:

```
treename="main";
add @sub, addfn("x1","x2");
treename="sub";
x{
  y, fn("a1","a2");
}
```

The `addfn()` callback is the called with the local parameters first, then the ones in the reference.

In other words, if the user types `add x y`, then the following function call is made: `addfn("a1", "a2", "x1", "x2")`. You need to ensure that `addfn()` can handle these arguments.

The reason for this mechanism is the autocli in Clixon, where it is used to generate a CLI from a YANG specification.

2.7.3 Filtering trees

When referencing a tree with the `@tree` syntax, it is possible to filter the tree by removing objects with specific labels. This can be useful if multiple references to a tree is made, but the instanciations differ.

For example, assume the following subtree "T", where one leaf is labelled with flag "local":

```
x {
  y, local;
  z;
}
```

Then this tree can be referenced as: `@T, @remove:local;` in which case the expanded tree has removed all tree nodes labelled with "local" as follows:

```
x {
  z;
}
```

2.7.4 Direct sub-expansion

If a treeref statement is placed directly on the top level of the expanded tree, it will also be expanded.

```
treename="main";
add @sub;
treename="sub";
@extra
x;
treename="extra";
y;
```

which means that both `add x` and `add y` are valid statements.

2.7.5 Tree workpoints

When using tree references it is possible to set an active *workpoint* for that tree, which can change dynamically. In this way, a user can navigate up and down the tree in its references for it. This is useful when implementing automatic modes for example.

To achieve this, a couple of C-API callbacks are available:

- `cligen_wp_set(<tree>)` - Set the workpoint
- `cligen_wp_show(<tree>)` - Show the tree at the active workpoint
- `cligen_wp_up(<tree>)` - Navigate up in the tree
- `cligen_wp_top(<tree>)` - Navigate to top of tree

Assume first a tree "T" is created that will be navigated in using workpoints:

```
treename="T";
a; {
  b <v:int32>; {
    d;
  }
  c;
}
```

Then, some main syntax for navigating in three is added in the main CLIgen spec:

```
edit, cligen_wp_set("T");{
  @T, cligen_wp_set("T");
}
show, cligen_wp_show("T");
up, cligen_wp_up("T");
top, cligen_wp_top("T");
```

It is now possible to traverse the tree using the "T" tree as the following example illustrates (the prompt shows the location of the active workpoint):

```
cli:/> show
a;{
   b <v>;{
         d;
      }
   c;
}
cli:/> edit a
cli:/a> show
b <v>;{
      d;
   }
c;
cli:/a> edit b 23
```

```
cli:/a/b/23> show
d;
cli:/a/b/23> up
cli:/a/b> show
<v>;{
   d;
}
cli> top
cli:/> show
a;{
   b <v>;{
         d;
      }
   c;
}
cli:/>
```

2.8 Output pipes

Output pipes resemble UNIX shell pipes and are useful to filter or modify CLI output. Example:

```
cli> print all | grep parameter
  <parameter>5</parameter>
  <parameter>x</parameter>
cli>
```

Output pipe functions are declared using a special variant of a tree with a name starting with vertical bar. Example:

```
treename="|mypipe";
\| {
   grep <arg:rest>, grep_fn("grep -e", "arg");
   tail, tail_fn();
}
```

where `grep_fn` and `tail_fn` are special callbacks that use stdio to modify output.

Such a pipe tree can be referenced with either an explicit reference, or an implicit rule.

An explicit reference looks something like this:

```
print {
   all, @|mypipe, print_cb("all");
   detail, @|mypipe, print_cb("detail");
}
```

where a pipe tree is added as a tree reference, appending pipe functions to the regular `print_cb` callback.

An example of an implicit rule is as follows:

```
pipetree="|mypipe";
```

```
print {
  all, print_cb("all");
  detail, print_cb("detail);
}
```

where the pipe tree is added implicitly to all commands.

Pipe trees also work for sub-trees, ie a subtree referenced by the top-level tree may also use output pipes.

2.9 Sets

By default, listing several commands within {} gives a choice of commands, but if instead commands are within the set operator `@{}`, the commands can be given on any order, and at least once. For example:

```
x @{
  a;
  b;
  c;
}
```

gives exactly the following allowed CLI commands, and no others:

```
x a
x a b
x a b c
x a c
x a c b
x b
x b a
x b a c
x b c
x b c a
x c
x c a
x c a b
x c b
x c b a
```

3 Variables

Variables are placeholders for user input. They also give support for lexical checking. The `int32` type, for example, only accepts 32-bit integers, while `string` accepts any sequence of characters.

3.1 Basic structure

A variable has the following basic components:

- *name* - How the variable is referenced, such as in a callback.

- *type* - The type of the variable. If no type is given, it is by default the same as *name*.
- *show* - How the variable is displayed in help texts (such as after a '?' or 'TAB'. If no show field is given, it defaults to *name*.

The variable syntax has several forms:

```
<int32>;
<a:int32>;
<a:int32 show:"a number">("A 32-bit number")
```

In the first form, both name, type and show is `int32`. In the second form, the name and show is "a", while type is `int32`. In the last form, all fields are explicitly given, and there is also a help-text.

An example of the last, most explicit form in a CLI:

```
cli> '?'
   <a number>        A 32-bit number
```

3.2 String

The simplest form of a string specification is: `<string>`, which defines a string variable with the name 'string'.

A more advanced string variable specification is the following:

```
address <addr:string>("Address to home");
```

where the name of the string variable is `addr`. The name can be used when referring to the variable in a callback, and is also used in the help text:

```
cli> address '?'
   addr          Address to home
```

A string may contain all characters with some minor exceptions. Most notably, a string can not contain a question mark, since it is used for querying syntax in the CLI. Also, if a string contains spaces, it must be contained within double quotes. The following examples are all valid strings:

```
i_am_a_string
()/&#
"I am a string"
ab"d
```

A string can be constrained by a `length` statement. If given, the number of characters in a string is limited to a min/max interval, or just a max[1].

Example:

```
<addr:string length[8:12]>
<addr:string length[12]>
<addr:string length[2:4] length[8:12]>
```

which means that the `addr` string, if given, must be between 8 and 12 characters long, or just limited to 12 characters, or be between 2-4 and 8-12 charcters respectively.

[1]Note that this differs from YANG types where a single bound means exactly that value, see RFC 7950

3.2.1 Rest

A variant of string is `rest` which accepts all characters until end-of-line, regardless of white space. This is useful for variable argument input.

Example of the difference between `string` and `rest` type:

```
single("System command") <command:string>;
command("System command") <command:rest>;
```

Note the difference where the `string` type accepts a single string, whereas the `rest` type groups all remaining characters into a single string regardless of whitespace:

```
cli> single one two three
CLI syntax error: "single one two three": Unknown command
cli> single "one two three"
cli> command one two three
cli>
```

3.3 Integers

There are several integer variables, signed, unsigned, and 8, 16, 32 or 64-bits. For example, the `int32` variable allows any 32-bit integer, and can be specified in decimal or hex format.

Further, an allowed range of integer can specified, either as an interval or as an upper limit, or a set of ranges:

Examples:

```
<x:uint32>
<x:int8 range[-12:80] range[100:110]>>
<x:int64 range[1000]>
```

3.4 Addresses

CLIgen is often used in communication devices. Therefore, there is support for several pre-defined address types. Special lexical checking is defined for those types:

- `ipv4addr` - An IPv4 address in dotted decimal notation. Example: `1.2.3.4`
- `ipv4prefix` - An IPv4 prefix in 'slash' notation: Example: `1.2.3.0/24`
- `ipv6addr` - An IPv6 address. Example: `2001::56`
- `ipv6prefix` - An IPv6 prefix. Example: `2001:647::/64`
- `macaddr` - A MAC address: Example: `00:E0:81:B4:40:7A`
- `url` - An URL: Example: `http://www.hagsand.se/cligen`

CLIgen performs lexical checking of the address variables, an invalid address is considered as a syntax error.

3.5 Uuid

A variable of type `uuid` accepts uuid according to standard syntax, such as `f47ac10b-58cc-4372-a567-0e02b2c3d479`.

3.6 Time

A `time` variable accepts ISO timestamps on the form

```
2008-09-21T18:57:21.003456
2008-09-21 18:57:21.003456
2008-09-21 18:57:21
```

3.7 Boolean

A variable of `bool` type accepts the values `true`, `false`, `on` and `off`.

3.8 Decimal64

A variable of type `decimal64` defines a subset of floating point numbers that can be obtained by multiplying a 64-bit signed integer with a negative power of ten, ie as can be expressed by $i * 10^{-n}$, where n is between 1 and 18.

The number of fraction-digits can be defined in the specification of the type, if this is not defined explicitly, the default number of decimals is 2.

Two examples of decimal64 are `732848324.2367` (four fraction-digits) and -23.0 (one fraction-digit).

Examples of decimal64 specification is:

```
<d:decimal64 fraction-digits:4>;
<d:decimal64 fraction-digits:4 range[0.1:10]>;
```

which allows numbers with four decimals. The econd example limits the numbers to be between 0.1000 and 10.0000.

Note that the `fraction-digits` statement should come before the `range` statement.

3.9 Keyword

A keyword variable is just an alternative way of specifying command keywords as defined in Section 2. In fact, a syntax with static keywords can just as well be written using keyword variables.

Thus, for example, the two specification lines below are equivalent:

```
aa bb;
<aa:string keyword:aa> <bb:string keyword:bb>;
```

However, a keyword variable can have another name:

```
<myname:string keyword:aa>;
```

Naming of keywords provides for more flexible search functions in callbacks, see Section 5.

Note that a keyword must be of type `string`.

3.10 Choice

The choice variable can take the value from a static list of elements.

Example:

```
interface <ifname:string choice:eth0|eth1>("Interface name");
```

A CLI user will get the following choice:

```
cli> interface '?'
   eth0        Interface name
   eth1        Interface name
cli>
```

The user can only select eth0 or eth1, and thus the value of the `ifname` variable is either `eth0` or `eth1`.

Note the resemblance with choice of strings in Section 4 where the same example could be specified as:

```
interface (eth0|eth1)
```

Again, the former variant allows for naming of the variable which can be better when writing a callback function. In the example, the name of the variable in the first example is `ifname` whereas in the second it is *either* `eth0` or `eth1`.

3.11 Expand

The choice variable specifies a static list of keywords. But what if the list is dynamic and changes over time?

The expansion variable is a *dynamic* list of keywords, where the list may be different each time the CLI command is invoked.

For example, assume a user can select a network interface in the CLI, but the number of interfaces changes all the time. This can be specified as follows:

```
interface <ifname:string interfaces()>("Interface name")
```

The user's choice in the CLI will then be just as in the choice case:

```
cli> interface '?'
   eth0        Interface name
   eth1        Interface name
cli>
```

However, at another point in time, the choice of interfaces may be different:

```
cli> interface '?'
   eth3        Interface name
   lo0         Interface name
cli>
```

There is one catch here: the CLI needs to know in run-time the members of the list. That is, the list members cannot be specified in the syntax. In CLIgen, the application programmer defines a C callback function, `interfaces()` in this example, which computes the list at the time it is needed. This callback is registered and called whenever necessary.

How to write an expand callback is further described in Section 6.3.

3.12 Regular expressions

A string variable may be described using a regular expression. That is, a regular expression defines which values are valid.

For example, a variable may be specified as:

```
<name:string regexp:"(ab|a)b*c">;
<name:string regexp:"[a-z]+[0-8]+\\.[0-9]">;
```

The first rule matches the following strings, for example:

```
ac
abc
abbbbbbbbbc
```

You can also add multiple regexps, in which case the string must match ALL regexps. You can also specify an inverted regex, meaning that a string should NOT match the regex.

In the following example, all strings are matched that do not start with "cli"

```
<name:string regexp:"[a-zA-Z]+" regexp:!"cli.*">;
```

CLIgen by default uses POSIX Extended regular expression syntax. However, XML schema regexps can be used instead by configuring CLIgen with libxml2 as follows:

```
./configure --with-libxml2     # At configure time
cligen_regexp_xsd\_set(h, 1);  # In C init code
```

3.13 Variable translation

CLIgen supports variable translation. A given variable can be translated on-th-fly using a generic translation function. This may be useful for example when hashing or encrypting a value.

In the example below, the variable `var` is translated by the function `incstr`:

```
increment <var:string translate:incstr()>, cb();
```

In this example, `incstr` simply increments every character in the variable.

```
cli> increment HAL
variables:        1 name:var type:string value:IBM
```

In the same way, a clear text password could be translated to an encrypted string. For information on how to implement a translator function, see Section 6.4.

3.14 Variable preference

There are situations when a command matches multiple variables, and CLIgen may return *Ambiguous command*. For this purpose, variable *preference* can resolve tiebreaks.

For example, consider the CLI syntax:

```
mycommand;
<name:string regexp:"[a-z]+">;
<name:string>;
```

If a user types the command `mycommand`, it matches all three rules; typing `foo` matches the two variables; and `foo42` matches only the last.

Variable types have a default preference to resolve tie-breaks like these. The highest preference is the best match. In the example, the command "mycommand" has the highest preference, thereafter the regexp, and finally the unqualified string type.

This means that typing `mycommand` matches the first command rule; `foo` matches the regexp rule; and `foo42` matches the last string type.

The default variable preference table is available in Appendix D.

3.14.1 Resolving tie-breaks

Even with default variable preferences, there may be cases where a command matches several rules with the same same default preference. For example, strings with regular expressions that overlap as follows:

```
<name:string regexp:"[a-b]+">;
<name:string regexp:"[b-c]+">;
```

If a user types the string `bbb`, it matches both rules with equal preference, which results in *Ambiguous command* error in the CLI.

There are two methods in CLIgen to resolve this tie-break:

1. `preference` keyword
2. `cligen_preference_mode_set()` API function

3.14.2 Preference keyword

First, the `preference` keyword alters the preference of a variable, such as in the following example:

```
<name:string regexp:"[a-b]+" preference:9>;
<name:string regexp:"[b-c]+">;
```

The effect is that the default preference (7) is overruled (9) in the first rule and therefore preferred over the second.

3.14.3 Preference API

Second, using the `cligen_preference_mode()` API, a tie-break can be resolved by choosing the first matching command for *all* preference tie-breaks in a system. This may be useful if one knows that this does not pose a problem.

More specifically, if all cases of tie-breaks are on the following form:

```
(<name:string regexp:"[a-b]+"> | <name:string regexp:"[b-c]+">) A();
```

it can be considered safe, since both options lead to the same result `A()`.

However, you need to be careful if effects of the commands are different. For example, given the following syntax:

```
<name:string regexp:"[a-b]+">, A();
<name:string regexp:"[b-c]+"> { sub-command, B(); }
```

In this example, the API method should be avoided, since choosing one of the two options lead to different results, in one case `A()` is called, in the other a sub-menu is selected.

4 Operators

In the regular syntax format, there are (implicit) sequence and choices. For example, the syntax

```
aa bb;
cc;
```

defines a choice between the sequence `aa bb` and `cc`.

It is also possible to explicitly define choices, optional elements and syntactical groupings.

4.1 Choice and grouping

Explit choice between several elements can be made as follows:

```
(aa bb) | cc;
```

which expresses the same syntax as above.

Help strings work as usual, but may *not* be associated with groupings:

```
aa (bb("help b") cc("help c") | dd("help d"));
```

Choices may also be made with variables:

```
values (<int8> | <string> | <int64> | aa);
```

where a pattern matching is made selecting to try to select the most 'specific' variable. For example, the following input will give different matchings:

- `aa` selects the keyword.
- `bb` selects `<string>`.
- `42` selects `<int8>`.
- `324683276487326` selects `<int64>`.

4.2 Optional elements

It is also possible to express an *optional* part of a syntax using brackets:

```
aa [[bb] cc];
```

which accepts the commands: `aa`, `aa bb` and `aa bb cc`.

Any combination of these operations are possible, such as in the line:

```
aa [[(bb|cc <int32>)] dd] ee;
```

Note that the elaborate command specifications above can be combined in a regular syntax, at parsing they are just expanded into a larger syntax tree. Thus for example, the syntax:

```
aa bb (ca("help ca")|cb("help cb")) [dd|ee];
```

is equivalent to:

```
aa bb{
   ca("help ca");{
      dd;
      ee;
   }
   cb("help cb");{
      dd;
      ee;
   }
}
```

which is similar to the syntax used in Section 2.

4.3 Caveat

Note that the choice, groupings and optional elements are only syntactical structures generating the basic constructs described earlier. This means that if you use too many of them, they can generate a large number of states and consume memory.

For example, the optional operator `[]` will increase the number of states with a factor of two. Thus `[a][b][c][d]` will generate 16 states, for example.

As an assistant, you can use the `-p` option to `cligen_file` to see which basic syntax is generated:

```
$ echo "[a][b][c];" | ./cligen_file -p
a;{
   b;{
      c;
   }
   c;
}
b;{
   c;
}
c;
```

As an alternative solution, you can use the set mechanism as described in Sections 2.9.

5 API

This section describes C-programming issues, including types, parsing and callbacks.

Appendix A contains a complete program illustrating many of the topics of this tutorial. More advanced applications can be found in the CLIgen source repository.

5.1 CLIgen variables

Variables in the command syntax (such as `<string>`) described in Sections 3 and 2.6 are translated in runtime into *CLIgen variables* using the `cg_var` datatype. A CLIgen variables is sometimes referred to as a *cv*.

A cv is a handle and its values are accessed using get/set accessors. Two generic fields are `name` and `type`, other fields are accessed via type-specific accessors (see next Section).

Example: get name and type of cligen variable:

```
char *name        = cv_name_get(cv);
enum cv_type type = cv_type_get(cv);
```

5.1.1 Types

CLIgen variables have a simple type-system, essentially following the types introduced in Section 3. Each cv type have get/set operators to access and modify the value.

For example, a command syntax contains `<addr:ipv4addr>`, and the user inputs "12.34.56.78". The CLI will then generate a cv which can be accessed in C. The string "12.34.56.78" is accessed with:

```
struct in_addr addr = cv_ipv4addr_get(cv);
```

Accessors for other types are shown in the table below. There may be several fields for a given type. These are given in the table with the corresponding C-type.

Type	Accessor	C-type
int8	cv_int8_get()	int8_t
int16	cv_int16_get()	int16_t
int32	cv_int32_get()	int32_t
int64	cv_int64_get()	int64_t
uint8	cv_uint8_get()	uint8_t
uint16	cv_uint16_get()	uint16_t
uint32	cv_uint32_get()	uint32_t
uint64	cv_uint64_get()	uint64_t
decimal64	cv_dec64_i_get()	int64_t
	cv_dec64_n_get()	uint8_t
bool	cv_bool_get()	uint8_t
string	cv_string_get()	char*
ipv4addr	cv_ipv4addr_get()	struct in_addr
ipv4prefix	cv_ipv4addr_get()	struct in_addr
	cv_ipv4masklen_get()	uint8
ipv6addr	cv_ipv6addr_get()	struct in6_addr
ipv6prefix	cv_ipv6addr_get()	struct in6_addr
	cv_ipv6masklen_get()	uint8
macaddr	cv_mac_get()	char[6]
uuid	cv_uuid_get()	char[16]
time	cv_time_get()	struct timeval
url	cv_urlproto_get()	char*
	cv_urladdr_get()	char*
	cv_urlpath_get()	char*
	cv_urluser_get()	char*
	cv_urlpasswd_get()	char*

You may also access a value with an unspecified type using:

```
void *v = cv_value_get(cv);
```

5.1.2 Cligen variable vectors

Variables are grouped into vectors whose using `cvec`. Global variables or variables passed to callback functions are always grouped into `cvec` structures.

5.1.3 Finding variables in a vector

Suppose for example that you have the following command syntax:

```
person [male|female] (<age:int32>|<name:string>)
```

A `cvec` is accessed using a handle. Typically an iterator is used to access the individual cv:s within a vector:

```
cvec *vr;
cg_var *cv = NULL;
while ((cv = cvec_each(vr, cv)) != NULL) {
   str = cv_name_get(cv);
}
```

You can also access the variables individually if you know their order, in this example the 3rd element:

```
cvec *vr;
cg_var *cv = cvec_i(cv, 2);
```

A way to find variables using their names is as follows:

```
cg_var *cv = cvec_find(vars, "age");
```

Actually, kewords are also a part of variable vectors. This means that they can also be accessed via their name, although the name of the keyword is the same as its constant value, as described in Section 3.9.

Therefore, you can also check whether a keyword exists or not. Using the same example:

```
if (cvec_find(vars, "male") != NULL)
  printf("male\n");
```

where the conditional evaluates to true only if the user has selected `male` and not `female`.

5.2 Initializing

An application calls the CLIgen init function to initialize the CLIgen library. The function returns a `handle` which is used in most CLIgen API functions.

In the following example, CLIgen is initialized, a prompt is set, and is then terminated:

```
cligen_handle h = cligen_init();
cligen_prompt_set(h, "cli> ");
[...]
cligen_exit(h);
```

5.3 Parsing syntax files

The command syntax as described in Sections 2-4 normally resides in a file which is loaded and parsed by the CLI. The result of the parsing is a parse-tree and a list of global variable assignment. After parsing, the program needs to interpret the result and set up the CLI environment. This includes handling global variable assignments, mapping function callbacks, etc.

Most non-trivial programs handle many syntaxes that are merged into a common parse-tree, while others partition parse-trees into different modes.

An example of parsing syntax file `mysyntax.cli` is the following:

```
cligen_handle h;
FILE         *f;
h = cligen_init();
f = fopen("mysyntax.cli");
cligen_parse_file(h, f,  "mytree", NULL, NULL);
cligen_loop(h);
```

The example code initiates a handle, opens the CLIgen syntax file, parses the syntax into a tree called `mytree` and starts a CLIgen command loop.

The next step is to handle the global variables and to bind callback functions.

5.4 Global variables

A syntax file may contain global variable assignments which can be accessed by the the C-code. Suppose a syntax file contains the following global assignments:

```
prompt="cli> ";              # Assignment of prompt
```

These global variables are parsed and may be read by the C-code as follows:

```
char *prompt;
[...]
cligen_parse_file(h, f,  "mytree", NULL, globals);
prompt = cvec_find_str(globals, "prompt");
cligen_prompt_set(h, prompt);
```

In this way a programmer may define the semantics of global variables by binding their value to actions.

5.5 Command loop

A programmer can use the pre-defined `cligen_loop` function, or create a tailor-made loop as follows:

```
for (;;){
    retval = cliread_eval(h, &line, &ret);
```

The return value of the `cliread_eval` function is as follows:

- `CG_EOF`: end-of-file
- `CG_ERROR`: CLIgen read or matching error, typically if the syntax is not well-defined.
- `CG_NOMATCH`: No match, the input line did not match the syntax. By calling `cligen_nomatch(h)`, the reason for why no match was made is retrieved.
- `CG_MATCH`: Match, the line matched exactly one syntactic node. The variable `ret` contains the return value of the callback (if any).
- > 1: Multiple matches, the line matched several syntax lines.

6 Advanced API

6.1 Writing a callback function

A programmer may write a callback function for every complete command defined in the command syntax. Such a callback is then called every time a user types that command in the CLI.

An example of `CLIgen` callback function from the example in Section 1 with the command syntax `hello world,cb("hello");`is:

```
int cb(cligen_handle h,
       cvec *cvv,
       cvec *argv)
{
   printf("%s\n", cv_string_get(cvec_i(argv, 0)));
   return 0;
}
```

The callback returns zero if everything is OK, and −1 on error. The parameters of a callback function are:

- `handle` - CLIgen handle created by a call to `cligen_init`. The handle is used if the callback makes API calls to CLIgen, such as changing prompt, parse-tree, etc.
- `cvv` - The command line as a list of CLIgen variables. By default, both keys and variables are included in the list.
- `argv` - A vector of CLIgen variables declared in the command syntax.

Regarding a more advanced command syntax from Section 5.1.3:

```
person [male|female] (<age:int32>|<name:string>),cb("person");
```

and an CLI input command such as:

```
cli> person male 67
```

6.1.1 The cvv parameter

In the example, the cligen variable vector `cvv` has four elements and can be accessed via iteration or via the `cvec_i()` function:

1. The complete command string: `person male 67` as the user typed the string.
2. The CLIgen string variable containing the keyword `person`.
3. The keyword `male`.
4. A CLIgen integer variable containing 67.

Two API functions control the format of `cvv` and changes the rules above (should be called initially in the program):

1. `cligen_expand_first_set()`: Instead of showing the first `cvv` argument as the user typed it, the *expanded* variant (after completion) is shown. Example: the user types `per m 67`, whereas the expanded version of that string is `person male 67`.
2. `cligen_exclude_keys_set()`: Do *not* include keywords in the `ccv` argument. With this option enabled, there are only two elements in `cvv` of the example: The complete command string; and the integer, while the two keywords `person` and `male` are omitted.

6.1.2 The argv parameter

The `argv` argument contains the function argument in the command syntax: `person`.

By using the values in the argument and variable vectors, the callback can perform actions by calling CLIgen API functions. In those functions, the handle `h` is usually required and used to make global changes.

6.2 Registering callbacks

A typical syntax contains callback references, such as the following:

```
hello world, callback("arg");
```

The parse-tree created in Section 5.3 contains the function names as strings which need to be mapped to function pointers. This is a typical issue with the C programming language. The problem is essentially the same as finding functions in a symbol-table. Note that this mapping is not a part of CLIgen itself but needs to be made by the application.

There are many ways to solve this issue, including using dynamic libraries and making a lookup in real-time using `dl_open`, `mmap`, or similar C library functions. This is actually the preferred option, the other approaches described here are not as good.

One way is to map each callback specified to a different function. This can be made by defining a function that maps between function name strings and actual functions and calling a mapping funtion, for example:

```
cgv_fnstype_t *
mapper(char *name, void *arg, char **error)
{
  *error = NULL;
  if (strcmp(name, "callback") == 0)
    return callback;
  return callback; /* allow any function (for testing) */
}
cligen_callbackv_str2fn(pt, mapper, NULL);
```

6.2.1 Multiple callbacks

Several callbacks may be associated with a syntax. Example:

```
hello world, callback("arg"), extra();
hello world, extra2();
```

In this case, all three functions: `callback`, `extra` and `extra2` are called, one after the other.

6.3 Completion

If expand variables (see Section 3.11) are used, the application defines a callback to fill in the elements of the dynamic list. Such a callback is invoked every time the CLI asks for a command containing the corresponding expand variable.

That is, the callback may be invoked when a user types a question mark or a TAB as well.

The following example shows the expand function `expand_ifname`. A translator function (`str2fn`) maps name of functions to actual functions. In this case it trivially returns the expand function for all commands. More elaborate mapping functions consult a symbol table or some other way to map the function name supplied in the syntax, with an actual function pointer.

The expand function supplies a list of pointers to strings, in this example a list of interfaces. The example returns a static list of interfaces: "eth0" and "eth1", a real example would dynamically get the list of interfaces. If the helptexts are not given, the helptext in the specification is used.

```
int
expand_ifname(cligen_handle h, char *fn_str, cvec *cvv, cg_var *argv,
              cvec *commands, cvec *helptexts)
{
  cvec_add_string(commands, NULL, "eth0");
  cvec_add_string(helptext, NULL, "The first interface");
  cvec_add_string(commands, NULL, "eth1");
  cvec_add_string(helptext, NULL, "The second interface");
  return 0;
}

expandv_cb *
str2fn(char *name, void *arg, char **error)
{
   return expand_ifname;
}

main()
{
  [...]
  cligen_parse_file(h, f,  "mysyntax", &pt, &globals) < 0)
  if (cligen_expandv_str2fn(pt, str2fn, NULL) < 0)
    return -1;
  [...]
}
```

In other words, as soon as the user selects a line containing the variable `interfaces`, `expand_ifname()` will be called. Therefore, be careful to avoid blocking calls within the callbacks since this may make the CLI less interactive.

6.4 Translation

An example of a variable translator function is as follows:

```
int
incstr(cligen_handle h,
       cg_var       *cv)
{
    char *str;
```

```
    int i;

    if (cv_type_get(cv) != CGV_STRING)
        return 0;
    str = cv_string_get(cv);
    for (i=0; i<strlen(str); i++)
        str[i]++;
    return 0;
}
```

In the function, the CLIgen variable `cv` assumed to be a string, every character is incremented, so that the string HAL would be translated to IBM, for example.

In the same way as expand functions, the translator functions must be registered using `cligen_translate_str2fn()`. See the tutorial example and code for more details.

6.5 History

CLIgen has a command-history API.

Example, initialize a command history with 100 lines, load existing history from a file called ".cli_history" and register history callback `history_cb`:

```
FILE *f;
cligen_hist_init(h, 100);
f = fopen(".cli_history", "r");
cligen_hist_file_load(h, f);
cligen_hist_fn_set(h, history_cb, NULL);
```

The history callback may be used to log CLI commands or in other ways mirror commands. Example of a history callback making syslog calls at LOG_INFO:

```
int
history_cb(cligen_handle h, char *cmd, void *arg)
{
   return syslog(LOG_MAKEPRI(LOG_USER, LOG_INFO), "CLI command:%s", cmd);
}
```

7 Installation

CLIgen is easiest installed from github. Just clone the source, configure it and type make, and try the tutorial program:

```
> git clone https://github.com/clicon/cligen.git
> cd cligen
> ./configure
> make
> sudo make install
> ./cligen_tutorial -f tutorial.cli
hello>
```

CLIgen can be installed on a variety of platforms using configure. Installation installs library and include files in the system. It is also possible to install library only (or include-files only) using `make install-lib` (or `make install-include`).

Appendix A: Tutorial command syntax

Please see the `tutorial.cli` file in the source release.

Appendix B: API functions

To generate CLIgen reference manual, please do `make doc` in the cligen directory.

Appendix C: Control sequences

The control sequences of the runtime CLI is as follows:

Control sequence	Action	Comment
?	Help	
Ctrl + A	Go to beginning of line	
Ctrl + B	One char backwards	
Ctrl + C	Exit CLI	Add extra w cligen_exitchar_add().
Ctrl + D	End-of-file.	Exit if at beginning of line
Ctrl + E	Goto end of line	
Ctrl + F	One char forward	
Ctrl + H	Erase previous character	Backspace
Ctrl + I	Auto completion	TAB
Ctrl + K	Erase line after cursor	
Ctrl + L	Redraw line	
Ctrl + N	Move to next line in history	
Ctrl + O	Toggle overwrite mode	
Ctrl + P	Move to previous line in history	
Ctrl + R	Search history list backward	
Ctrl + S	Search history list forward	
Ctrl + T	Transpose character	
Ctrl + U	Erase line before cursor	
Ctrl + W	Erase word backward	
Ctrl + Y	Insert previously deleted text	'yank'
Ctrl + Z	'Suspend'	Register callback: cligen_susp_hook()
Arrow up	Move to previous line in history	
Arrow down	Move to next line in history	
Arrow left	One char backward	
Arrow right	Once char forward	
ESC + F	Move one word forward	
ESC + B	Move one word backward	

Appendix D: Default variable preference

The full default preference table is as follows:

Type	Preference
command	100
time	74
uuid	73
macaddr	72
ipv6addr	71
ipv4addr	70
decimal64	62
int8 (range)	60
uint8 (range)	59
int16 (range)	58
uint16 (range)	57
int32 (range)	56
uint32 (range)	55
int64 (range)	54
uint64 (range)	53
int8	52
uint8	51
int16	50
uint16	49
int32	48
uint32	47
int64	46
uint64	45
url	20
bool	12
string (expand)	8
string (regexp)	7
string	5
command (partial)	3
rest	1

Note that integers with ranges have higher preference than integers lacking ranges.

Also, expanded strings have higher regexp:s than strings with regexp:s and regular strings.

Note also that “command” occurs twice, first as exact command match with preference 100, second as partial matchwith preference 3.