TICAM REPORT 96-09
February 1966

cmdGen 2.5 User Manual

Phillip T. Keenan

`cmdGen` 2.5 User Manual

Philip T. Keenan

February 6, 1996

Contents

1 Introduction

This manual describes the user interface to the `cmdGen` program. The user interface is based on Philip T. Keenan's *kScript* package, version 2.5, a powerful and flexible application scripting language. The user interface was itself created with `cmdGen`, and builds on the Keenan C++ Foundation Class Library version 2.5.

For a complete introduction to *kScript*, see the *kScript User Manual*[1], or the World Wide Web page at `http://www.ticam.utexas.edu/users/keenan/kScript.html`. The *kScript* package can be obtained from the Web site, and used as the front end to other applications, but it is subject to the terms of the *kScript* copyright notice provided with the distribution and is not in the public domain.

The `cmdGen` program is a "user interface compiler". It provides an easy way to build *kScript* user interfaces to application programs. A `cmdGen` script defines the commands, objects and type names that will be used in the interface. The `cmdGen` program converts this script to the C++ source code needed to implement the specified application interface, and also creates a LaTeX manual documenting the interface. The program understands very general command and function descriptions, in which there can be optional arguments or repeated sequences of arguments.

1.1 Copyright and Disclaimer

Copyright (C) Philip Thomas Keenan, 1990-1996.

The following terms apply to all printed documents and computer files distributed with the kScript software package.

The source code and manuals for kScript are COPYRIGHTED, but they may be freely copied as long as all copyright notices are retained verbatim in all copies. In addition, if the source code or derivatives of it are incorporated into another software package, the source code, the documentation and any descriptions of the user interface for that package must contain the acknowledgment "this software uses kScript, an application scripting language developed by Philip T. Keenan", and cite the kScript User Manual [TICAM Tech. Report, The University of Texas at Austin, 1996] and the kScript World Wide Web page [http://www.ticam.utexas.edu/users/keenan/kScript.html]. If the kScript source code is modified, a disclaimer stating the nature of the modifications must also be added to both the source code, the documentation and any descriptions of the user interface.

For instance, if your program uses kScript or a derivative of it, then in any formal setting in which you mention the user interface to your program, you must acknowledge its use of kScript. For instance, a press release, conference presentation or paper about your code which mentions the flexible nature of the user interface as a positive feature of your code MUST CREDIT and site kScript as the basis for that feature. You MAY NOT claim, explicitly or implicitly, that this flexible scripting language was your invention.

The kScript source code, the manuals, and executable programs incorporating kScript are for NON-COMMERCIAL USE only and may not be marketed for profit without a prior written licensing agreement from the author. If you would like to use kScript as part of a research project contact Mary F. Wheeler, Director of the Center for Subsurface Modeling, TICAM, at the University of Texas at Austin (mfw@ticam.utexas.edu) for distribution and collaboration information.

The author makes NO REPRESENTATIONS OR WARRANTIES about the correctness of any source code or documentation, nor about the correctness of any executable program or its suitability for any purpose, and is in no way liable for any damages resulting from its use or misuse.

IN NO EVENT SHALL THE AUTHOR(S) OR DISTRIBUTOR(S) BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHOR(S) HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHOR(S) AND DISTRIBUTOR(S) SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHOR(S) AND DISTRIBUTOR(S) HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

2 Command Line Arguments

To run `cmdGen`, type

```
  cmdGen cmds.k
```

where `cmds.k` is an input script containing commands described in the next section. This will create seven output files with names based on the `project` variable. The files `proj.h` and `projPost.C` declare and post to kScript the generated commands. The files `projObjs.h`, `projObjs.C`, and `projPostObjs.C` declare and post the generated objects. `projDetails.tex` is a LaTeX documentation file. Finally the file `proj.C` contains starter code for the generated actions. It is generally the only file you will need to edit. It contains C++ code to parse each of the generated commands, but you will need to add application specific code to each command to actually do something with the arguments you have parsed. You will want to rename it something like `actions.C` to avoid losing your additions if you re-run `cmdGen` to modify the user interface.

The complete syntax for `cmdGen` is

```
cmdGen [-usage] [-a] [-d] rfile
```

The `-usage` option prints a copyright notice and helpful information about the command line arguments. The `-d` option is for section debugging use by advanced users. The `rfile` argument is the name of the input script file to read.

The `-a` option turns on generation of three additional output files: `projMain.C`, which contains a sample driver program, `projManual.tex`, which contains a sample LaTeX manual, and `makefile`, which is a sample makefile. Generally you use the `-a` option the first time you run `cmdGen` but not subsequently, since you will likely edit these files and do not want them overwritten.

3 Input Commands

3.1 kScript Basics

This manual primarily discusses the extensions to kScript provided by the `cmdGen` program. However, we begin with a brief summary of the core features of kScript.

The user interface reads commands from a *kScript* input file. *kScript* is a complete programming language with comments, numeric and string variables, looping, branching and user defined commands. It includes predefined commands for online help, include file handling, arithmetic calculations and string concatenation, and communication with the UNIX shell. Applications can define additional commands and objects which enrich the vocabulary and power of *kScript*. *kScript* is strongly typed and applications can add new data types as well.

For a complete and up-to-date list of commands, functions, types and objects available to the user interface, run the program to access on-line help. Type

```
help
```

to get started.

Commands specific to the `cmdGen` program are listed below. Each command's name is followed by a list of arguments. Most arguments consist of a type name and a descriptive name, enclosed in angled brackets. These represent required arguments that must be of the stated type.

Arguments enclosed in square brackets are optional literal strings, typically prepositions. They can be used to create English sentence-like scripts which are easy to read, or they can be omitted with no change in the meaning of the script. Sometimes several alternatives are listed, separated by a vertical bar (|). For example, the syntax of the set command is

```
[set] ⟨nameExpr name⟩ [to|=] ⟨expression expr⟩
```

Both the name `set` and the equal sign are optional, so the five commands

```
set x to 3.14
set x = 3.14
```

```
x = 3.14

set x 3.14
x 3.14
```

all assign the same value to a variable named x, but the first three versions are easier for a human reader to understand.

The keywords `optional` and `required` introduce alternative sets of arguments. Each set begins with a string literal which if encountered while parsing the command signals that the remainder of that clause will follow. Multiple cases can be separated by a vertical bar. In the required case, one alternative must be selected; in the optional case, zero or one may be chosen.

The `sequence` keyword introduces an argument pattern which may be repeated multiple times. A sequence argument can be an empty string (`{}`), a curly brace delimited list of one or more instances of the pattern, or, a single instance without the surrounding curly braces.

In *kScript*, a space-delimited sharp or pound symbol (`#`) comments out the rest of the line on which it occurs. Mathematical expressions must be written with no internal spaces. String literals must be enclosed in curly braces, not quote marks. The curly braces can be nested and within them only the percent sign (`%`) is special — all other text is recorded verbatim. In all other contexts, white space (spaces, tabs, line breaks, and so on) serves only to delimit commands and their arguments.

kScript, like C++, is strongly typed. Users can create new objects, called variables, of any built-in type, using the `define` command, while programmers can extend the set of built-in types.

The formal argument types in command descriptions generally correspond to C++ class names. The actual argument must be in the correct format for the specified type. For an explanation of the syntax for a particular type T, use the command `describe T`. The command

```
describe types
```

will list all of the type names for which on-line help is available. Built in types include numbers (`int`, `float`, `double`), character strings (`string`), files (`stream`), and various keyword types such as `category`. A command like `describe category` will list the literal names making up a keyword type.

Many commands take mathematical or string expressions as arguments. Math expressions can mix numbers, arithmetic and logical operators, and symbolic names. String expressions are enclosed in curly braces and can expand references to other variables' values by preceding their names with a percent sign. Both math and string expressions have additional features which are described in complete detail in the *kScript User Manual*.

3.2 cmdGen Extensions to kScript

`project`
: string: You must begin by setting this to a base file name for the project's generated output files.

`verify`
: int: Set to zero to turn off warning messages about quotes, less than and greater than signs in strings.

`inc optional {{obj}} <nameExpr name>`
: Command: Add a C-preprocessor '`#include`' command to the proj.h header file, or optionally to the projObjs.h file. The '`.h`' is automatically appended to the file name.

`cat <nameExpr name> <nameExpr parent> <stringExpr doc>`

Command: Begin a new category for subsequently defined commands, functions, and objects. The parent category can be omitted; it defaults to '`none`'. To return to a previously defined category, omit both the parent and documentation string.

`newType <nameExpr name> <nameExpr cppType> <nameExpr symType> <stringExpr doc>`

Command: Declare a new kScript type name and describe it's input syntax for on-line help. If missing, the C++ type used to declare global (posted) variables defaults to '`name`', and the C++ symbol entry type used to post posted variables defaults to '`nameEntry`'.

`post optional {{constant}} <nameExpr type> <nameExpr name> <nameExpr cppName> <stringExpr doc>`

Command: Declare and post a C++ variable as a constant or variable kScript object. The '`cppType`' and '`symType`' fields are set based on '`type`' and any previous type commands. If omitted, '`cppName`' defaults to '`name`'. The '`name`' expression can include constructor arguments, which are not included in the '`name`' field but are included in the '`constructedName`' field. For special types that require extra information, the '`type`' argument can be extended to the form '`type(specialType)`'. For instance, to post keyword variables use:

```
post keyword(collection) name(literal)
```

To post arrays, use the form: '`post array(kind) name(cppName)`'.

`keyword <nameExpr name> <stringExpr doc> sequence { <nameExpr litName> <nameExpr litCppName> <stringExpr litDoc> }`

Command: Create a new keyword type and give the initial list of literal values, which can be empty.

`optionKinds`

Keyword Type: Special forms recognized by the cmd and fn commands. Literal values are:

`sugar`

introduces syntactic sugar (ignored string literals)

`optional`

introduced optional clauses, each beginning with a string literal

`required`

same as optional, except generated code complains if user fails to choose one of the clauses.

`sequence`

defines a clause pattern; generated code looks for 0 or more instances of the pattern, which must be inside curly braces (unless exactly one instance is supplied)

`cmd <nameExpr name> <nameExpr cppName> optional {{syntax}} <stringExpr syntax> <stringExpr doc> sequence { <nameExpr cppType> <nameExpr argName> }`
Command: Create a new command, giving its name, an optional syntax string, an on-line help string, and an argument list. The '`cppName`' defaults to '`name`' if omitted. The argument list can contain typed arguments as well as repeated or optional arguments. Special type names follow the same syntax used in the post command; for example, use '`keyword(category) cat`' to declare a category name argument.

`fn <nameExpr name> <nameExpr cppName> optional {{syntax}} <stringExpr syntax> <stringExpr doc> sequence { <nameExpr cppType> <nameExpr argName> }`
Command: Create a new function. See the '`cmd`' command for details.

3.3 Specifying Optional Arguments

The `optionKinds` keyword type describes special forms which are recognized by the cmd and fn commands.

`sugar` introduces syntactic sugar (ignored string literals). For example,

```
sugar {{to}|{=}}
```

corresponds to the syntax description `[to|=]` in which either the word "to" or an equal sign is allowed and ignored if encountered.

`optional` introduces optional clauses, each beginning with a string literal. For instance,

```
optional {{row} int nRows | {col} int nCols}
```

`required` is the same as optional, except that the generated code complains if user fails to choose one of the clauses.

`sequence` defines a clause pattern; generated code looks for 0 or more instances of the pattern, which must be inside curly braces (unless exactly one instance is supplied). For instance,

```
sequence {int x stringExpr s}
```

would accept any of the following lines as input:

```
{}
5 word
{ 5 word }
{ 5 word 6 house 7 triangle }
```

3.4 Advanced Features

This section contains advanced information of interest primarily to programmers who wish to customize the output of `cmdGen`.

3.4.1 Output Fields

The `cmdGen` program reads the input file `cmdGen.k`, which may be in the user's current directory or in a master location. This file defines a number of strings, called sections, which represent actions for `cmdGen` to execute for each output section. Generally these actions involve setting the value of the variable `out`.

`out`
: string: The contents of this variable is written to the current output file after evaluating each section string action.

The following field variables are set by `cmdGen` prior to executing the section actions. This gives the action definitions access to the items being defined.

`name`
: string: The name of the thing being defined.

`cppName`
: string: The C++ name for the thing being defined; this defaults to '`name`' if omitted.

`doc`
: string: The documentation string the thing being defined.

`litName`
: string: The kScript name for a keyword literal.

`litCppName`
: string: The C++ name for a keyword literal; this defaults to '`litName`' if omitted.

`litDoc`
: string: The documentation string for a keyword literal.

`parent`
: string: The parent category in the '`cat`' command; this defaults to '`none`' if omitted.

`cppType`
: string: The C++ type for command and function arguments and posted objects.

`symType`
: string: The C++ symbol entry type for posted objects.

`ktype`
: string: The kScript type name for a posted object.

`specialType`
: string: Additional type information for things like keywords and arrays, when posted or when used as command or function arguments.

`isSpecial`
: int: When non-zero, specialType is active.

`constructedName`
: string: The complete '`name(args)`' form of an object name, with constructor arguments, for posting or in command and function argument parsing.

`argName`
: string: The C++ name of an argument in a command and function argument list, without any constructor list. Note that '`name`' and '`cppName`' in this context refer to the name of

the command or function, not the argument.

`syntax`
string: The syntax description of a command for online help.

`tex`
string: The tex description of a command.

`action`
string: The input action of a command or function.

`isConstant`
int: This is nonzero when posting a constant.

`isRequired`
int: This is nonzero inside required options in command or function argument lists.

`optKind`
string: The option kind in an argument list clause.

3.4.2 Keyword Types

The **`keyword`** command can actually generate C++ source code for several different styles of keyword classes, based on the setting of the **`keykind`** kScript string variable at the time the **`keyword`** command is processed. As usual, this behavior can be modified by editing **`cmdGen.k`** without needing to recompile the **`cmdGen`** program. Currently we provide three options:

- `{plain}` The default. The collection name is used as the root object name.
- `{root}` The string `_root` is appended to the collection name to form the root object name, and the plain collection name is typedef'ed to mean keyword.
- `{custom}` This is like the root case, except that the collection name is assumed to define a new class, publicly derived from keyword, rather than being typedef'ed.

After a custom keyword command, one can write additional setup commands to the source file. For instance, in the next section an example cmdGen script creates a keyword category for compass directions. The following script could be used instead, to initialize members of a custom direction class containing **`double x,y`** members and a **`set(x,y)`** member function:

```
keykind {custom}
keyword direction {Compass directions.}
{
  north {This will be (0,1).}
  south {This will be (0,-1).}
}

command dir {Initialize a new direction}
name n mathExpr x mathExpr y
{ echo {  direction_%n.set(%x,%y);} }

outStream = fP
dir north 0 1
dir south 0 -1
outStream = cout
```

One can also edit the cmdGen.k file or override definitions in it in the input file; following the pattern of the keykind option one can add all sorts of useful functionality in special situations, without needing source code to cmdGen itself.

3.4.3 Future Features

Someday I may add a few more features to the cmdGen.k file, such as easier posting of stream and array variables, following the pattern of posting keyword variables, with syntax such as `array(double)`. However, I may not have time to do these things in light of my upcoming career changes.

4 Sample Input Files

Here is a sample input file illustrating many of the features of `cmdGen`.

```
# a sample input file for cmdGen

project = {p}
inc header
cat Primary {The primary category for this example.}

keyword direction {Compass directions.}
{
  north {This will be (0,1).}
  south {This will be (0,-1).}
}

post keyword(direction) defaultDir(north) {The default direction.}*

cmd move {This moves you in the specified direction.}*
{
  keyword(direction) dir
  sugar {{by} | {for}}
  double distance
}

cmd ask {Ask for information}*
{
  required {{about} keyword(direction) dir | {all}}
}
```

Running `cmdGen` with the `-a` option on this file produces several C++ source files, described below, and a LaTeX manual, which is included at the end of this section.

You do not need to edit the generated files, except for `p.C`, which implements the command actions. The source files can be compiled using the generated `makefile` and will build an executable which will parse input, though it won't do anything interesting with it until you edit the command actions.

Still, it is useful to understand what the other files do. The file p.h includes header.h and predeclares the command action functions and keyword literals. The file pMain.C contains a sample driver that initializes kScript and your commands and objects, then calls the parser and processes input. The pObjs.h and pObjs.C files specify and implement the objects in the program, namely a global pointer variable called gptr that points to an object containing all your posted variables, such as defaultDir. The pPost.C and pPostObjs.C files post the commands and objects to the kScript symbol table, so that they will be available to users as extensions to kScript.

To make your new commands do interesting things, you simply edit the skeleton code provided in p.C, which contains sample definitions of your commands that read their arguments and then simply return. You can add code to print the arguments or otherwise make use of them in computations. For example, here is an edited version of the p.C file which prints a message for each action. Only the three lines involving outStream were added in the editing step — all the rest of the code was generated automatically by cmdGen.

```
/* p.C */

#include "p.h"
#include "sequence.h"

void moveCmd(cmdInterpreter& ci)
{
  keyword* dir(kType_direction);
  ci >> dir;
  if(ci.err()) return;
  ci.optional("by","for");
  double distance;
  ci >> distance;
  if(ci.err()) return;

  outStream << "moving " << dir << " for " << distance << " units." << nl;
}

void askCmd(cmdInterpreter& ci)
{
  if(ci.optional("about"))
    {
      keyword* dir(kType_direction);
      ci >> dir;
      if(ci.err()) return;

      outStream << "asking about " << dir << nl;
    }
  else if(ci.optional("all"))
    {
      outStream << "asking about all directions" << nl;
    }
  else SyntaxError(ci) << "required option not found!" << syntax;
  if(ci.err()) return;
}
```

Finally, here is the generated reference manual, formatted by LaTeX.

4.1 Primary

`direction`
Keyword Type: Compass directions. Literal values are:

`north`
This will be (0,1).

`south`
This will be (0,-1).

`defaultDir`
keyword(direction): The default direction.

`move <keyword(direction) dir> [by|for] <double distance>`
Command: This moves you in the specified direction.

`ask required {{about} <keyword(direction) dir> | {all}}`
Command: Ask for information

4.2 Self-Description

The user interface to **cmdGen** was itself generated by running **cmdGen**. Excerpts from the input script are shown below; compare them with the command and variable explanations earlier in this manual, which were automatically generated, for further insight.

```
post string project("") {You must begin by setting this to a base file
name for the project's generated output files.}*

post int verify(1) {Set to zero to turn off warning messages about
quotes, less than and greater than signs in strings.}*

cmd keyword {Create a new keyword type and give the initial list of
literal values, which can be empty.}*
{
  nameExpr name
  stringExpr doc
  sequence {
    nameExpr litName
    nameExpr litCppName
    stringExpr litDoc
  }
}
```

```
keyword optionKinds {Special forms recognized by the cmd and fn
commands.}*
{
  sugar {introduces syntactic sugar (ignored string literals)}*
  optional {introduced optional clauses, each beginning with a string
literal}*
  required {same as optional, except generated code complains if user
fails to choose one of the clauses.}*
  sequence {defines a clause pattern; generated code looks for 0 or
more instances of the pattern, which must be inside curly braces
(unless exactly one instance is supplied)}*
}
```

References

[1] Keenan, P. T., *kScript User Manual, Version 2.5*. TICAM Tech. Report, The University of Texas at Austin, February 1996.