kScript User Manual

Philip Keenan

CRPC-TR94537
November 1994

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

kScript User Manual*

Philip T. Keenan†

November 11, 1994

Contents

1 Introduction

This manual is an introduction to `kScript`, a flexible application scripting language. The `kScript` user interface was created with `cmdGen`, one of Phil Keenan's C++ code generation tools. It builds on the Keenan C++ Foundation Class Library. This manual describes version 1.0 of *kScript*, which corresponds to version 4.3 of `cmdGen` and version 2.3 of the KFCL.

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

*This research was supported in part by the Department of Energy, the State of Texas Governor's Energy Office, and project grants from the National Science Foundation. The author was also supported in part by an NSF Postdoctoral Fellowship.

†Department of Computational and Applied Mathematics, Rice University, P.O. Box 1892, Houston, TX 77251-1892.

concatenation, and communication with the UNIX shell. Applications can define additional commands, types and objects which enrich the vocabulary and power of *kScript*. This manual discusses those commands and objects defined by the core `kScript` language and hence available in all applications which support *kScript*.

kScript represents the most recent form of a user interface language I started developing back in 1989 as a C interpreter. Later I decided C syntax was too restrictive to force onto the user, as well as too complex for a user interface language, so I switched to a command driven style which I've successfully used for several years in programs such as `kplot` and `RUF`. Interestingly enough, several other people have been pursuing similar ideas. Microsoft's Visual Basic for Applications, Apple Computer's AppleScript, and the TCL/Tk package for UNIX machines are all examples of similar application scripting languages developed independently over the last couple years. Unlike most of these, however, *kScript* is flexible enough to allow users to define new data types, each with their own syntax, as well as new control structures; this enhances its potential for use in new applications. *kScript* also has "cmdGen", a program which writes most of the necessary C++ code for you when you want to add new *kScript* commands to an application.

For a complete and up-to-date lists of all commands and objects known to a particular application, run the program and access on-line help. Type

```
help
```

to get started.

2 *kScript* Fundamentals

2.1 Commands

This section describes the standard commands available for writing scripts in *kScript*. Applications which support *kScript* can be given such commands interactively or can read them from an input file.

Each command's name is followed by a list of arguments. Most arguments consist of a type name and a descriptive formal argument name, grouped as a pair in angled brackets. These represent required arguments to the command; the actual supplied argument will be parsed according to the syntax rules for the specified type.

Arguments enclosed in single square brackets are optional 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 following three uses of the `set` command are equivalent: they all store 5 in the numeric variable x.

```
set x 5
```

```
set x = 5
set x to 5
```

Whether you use prepositions or not is a matter of taste.

Arguments enclosed in double square brackets are optional keywords which do change the meaning of the script if they are supplied. Arguments enclosed in triple square brackets are lists of alternative keywords, one (and only one) of which must be chosen in any given situation. Occasionally, the triple square brackets notation is generalized to contain type names rather than literal keywords. For example, the `set` command determines the type of the destination variable (numeric or string) and expects the value argument to be of the corresponding type (`mathExpr` or `stringExpr`, respectively).

In *kScript*, a pound sign (`#`) comments out the rest of the line on which it occurs.

Detailed syntax for various common types is presented in Section 2.2. Each argument type is parsed according to its own rules, but in general arguments are delimited by white space (spaces, tabs, line breaks, and so on). For instance, mathematical expressions must be written with no internal spaces. String expressions must either have no internal spaces or be enclosed in curly braces.

`help`
: Provide an overview of the on-line help facilities.

`describe [[[commands|objects|types|stringExpr(name)]]]`
: Provide online descriptions of commands, objects, or types.

`symbolTable [show|print|dump]`
`[[[all|variables|constants|strings]]]`
: List 'all' objects by name and value, or just 'variables', 'constants' or 'strings'.

`quit`
: Ignore the rest of the current input file.

`include [file] <stringExpr filename>`
: Include a file of commands.

`changeIncDir [to] <stringExpr pathName>`
: Change the directory path used with include files.

`echoIncDir`
: Echo the current include file directory path.

`saveAndChangeIncDir [to] <stringExpr pathName>`
: Change the directory path used with include files, saving the old one.

`restoreIncDir`
Restore the directory path used with include files, as saved by the previous 'saveAndChangeIncDir' command.

`define [[string]] [[constant]] <stringExpr name> [as|=|:=] [[[mathExpr|stringExpr]]]`
Define a new variable or constant.

`set <stringExpr name> [=|to|:=] [[[mathExpr|stringExpr]]]`
Change the value of a variable.

`beginComment`
Begin an extended comment, which lasts until a matching 'endComment'. Comments may be nested.

`echo <stringExpr expr>`
Print the value of a string expression on the standard output stream.

`error <stringExpr expr>`
Print the value of a string expression on the standard error stream.

`shellCmd <stringExpr expr>`
Have the shell evaluate the expanded string. The shell's result code is returned in the 'theResult' numeric variable, and it's standard output is returned in the 'theStringResult' string variable.

`eval <stringExpr expr>`
Evaluate the expanded string as if reading commands from an include file.

`if <mathExpr expr> [then] <stringExpr thenCommands> <else stringExpr> <elseCommands [endif]>`
Branching command: if the math expression evaluates to a non-zero value, execute the commands in the string expression following 'then'; otherwise execute the commands in the string expression following 'else', if an else-clause is present. At least one of 'else' or 'endif' must be used.

`repeat <mathExpr repititions> [times] <stringExpr commandList> [endrepeat]`
Simple Looping command: repeat the commands in the string expression 'repititions' times.

`while <mathExpr condition> [do|repeat] <stringExpr commandList> [endwhile]`
General Looping command: repeat the commands in the string expression as long as the condition is non-zero.

`for [each] <stringExpr x> [in] <stringExpr wordlistX> <and [each]> <stringExpr y> [in] <stringExpr wordlistY> [do] <stringExpr commands> [endfor]`

Loop over each word in the list, assigning each in turn to the string variable x and executing the commands. If 'and' if used, a second variable y is assigned from a second list of the same length, in conjunction.

`push <stringExpr varName>`

Save the value of an object on the stack.

`pop <stringExpr varName>`

Restore the value of an object from the stack.

`defineCmd <stringExpr cmdName> [[...formal.argument.names...]] <stringExpr taskDescription> <stringExpr commandList> [enddefn]`

Define a new command. Formal arguments are numeric variables unless their name begins with a percent-sign. Square brackets in the formal argument list enclose an optional preposition. The task description must use the curly brace syntax for string expressions.

`alias [[[command|object]]] <stringExpr oldName> [as|to] <stringExpr newName>`

Define an abbreviation or alternative name for a command or object.

`seed <mathExpr seed>`

Set the random number generator seed. The seed should be a positive integer.

`strLen <stringExpr str>`

Sets theResult to the length of the string.

`strCmp <stringExpr str1> <stringExpr str2>`

Sets theResult to -1, 0, 1 as str1 precedes, equals, or follows str2 alphabetically.

`getChars <mathExpr first> <mathExpr last> <stringExpr sourceString>`

Puts the indicated range of characters (counted from 1) into theStringResult.

`local <stringExpr localnames>`

Within a defineCmd, this declares local variable names. The argument string is a list of names; begin string variable names with two percent signs.

2.2 Argument Types

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

```
describe all types
```

will list all of the type names for which on-line help is available.

The types `int`, `float`, and `double` represent integers, single and double precision floating point numbers. The type `chat*` represents a space delimited string, that is, a single word. Applications can define new C++ types and use them as arguments by defining a text representation and providing an

```
cmdInterpreter& operator>>(cmdInterpreter&, <type>&);
```

function. The `cmdGen` *kScript* user interface generator assumes such an operator is defined for all argument types it encounters.

2.3 Arithmetic and String Variables

Many commands take arithmetic or string expressions as arguments. Math expressions can mix numbers, arithmetic and logical operators, and symbolic names. Math operators are listed in Table 1. Logical operators return 1 for true and 0 for false. The `if` command treats 0 as false and non-zero as true. Parentheses override the standard operator precedences. In addition, many standard mathematical functions can be called, as listed in Table 2. The last three take two arguments (x, y) instead of one. They return x^y, a random number in the range $[x, y]$, and a normal variate with mean x and variance y. The underlying random number generator can be reseeded with the `seed` command. The `msec` function takes no arguments and returns the value of an internal timer, in milliseconds.

String expressions can be a single space delimited word, or arbitrary text enclosed in curly braces. When curly braces are used, internal white space is not ignored (unless the final right brace if followed by a `*`). Curly braces may be nested. Within the top level of curly braces, one can expand references to other string or numeric variables by preceding their names with a percent sign. Several other combinations are recognized as well, as shown in Table 3.

If a variable name to be expanded is followed by white space, the first such white space character is suppressed.

Symbolic names can represent constant or variable values. Predefined ones are listed below; users can define additional ones using the `define` and `set` commands.

Operator	Interpretation
+	addition
−	subtraction
*	multiplication
/	division
^	exponentiation
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!	is not
&	and
|	or

Table 1: Math Expression Operators

abs	round	sqrt	exp	log
sin	cos	tan	atan	msec
pow	random	normal		

Table 2: Mathematical Functions

Construction	Expansion
%%	%
%{	{ without counting toward nesting
%}	} without counting toward nesting
%nX	n repititions of character X
%\n	newline
%\t	tab
% followed by newline	the newline is suppressed

Table 3: String Expression Expansions

`%incDir`
: If an include file is not found in the current directory, the "include" command looks next in this directory. This string variable is initially set to the value of the shell environment variable KSCRIPT_INC_DIR.

`theResult`
: A numeric variable which is intended for use as a return value holder for commands that need to return a numeric value.

`%theStringResult`
: A string variable which is intended for use as a return value holder for commands that need to return a numeric value.

3 Examples

3.1 A Simple Script

The following script can be run by any program which supports *kScript*. It simply illustrates some of the generic programming language features of *kScript*. More interesting scripts can be written for specific applications using application defined commands and objects.

```
# a sample script

define x 1+exp(2)

while x>4 {
  echo {x = %x}
  set x = x-1
}

define string case {}
define string cmd {}
for each case in {first second third} do {
  set cmd {run the `%case' command}
  echo {%cmd}
}

defineCmd sum x y {sets `theResult' to be x+y}
{
  set theResult to x+y
}
```

```
sum 4.5 6.7
echo {The sum is %theResult}
```

When run, this script produces the following output:

```
x = 8.3890561
x = 7.3890561
x = 6.3890561
x = 5.3890561
x = 4.3890561
run the 'first' command
run the 'second' command
run the 'third' command
The sum is 11.2
```

Such a script is very similar to a UNIX shell script; indeed the `shellCmd` command and the `eval` command combine to let *kScript* interact with the UNIX shell in very general ways, thereby enabling powerful extended features like interactive inter-application communication across networks. Yet the full power of *kScript* only becomes apparent when applications define additional commands and objects which enrich the language with concepts and actions specific to the application domain.

For example, my `kplot` graphics program extends *kScript* by defining additional commands for plotting lines, triangles, rectangles and polygons. It uses objects to represent color and font palettes, and to control the coordinate system. Moreover, since *kScript* lets users write their own subroutines and functions, it can even be extended with commands to create custom axis tic mark patterns and labels, without any knowledge of the internals of the program or any need to recompile it.

As another example, the Rice Unstructured Flow code (RUF) solves elliptic partial differential equation on general geometries. It extends *kScript* with commands for specifying the computational grid, defining coefficients, and taking one or more time steps. Other scientific computing applications might define objects such as the time step or an error measurement. In this case, *kScript* would allow the user to implement an adaptive time step selection algorithm, and experiment with modifications of it, all without need to understand or even access the source code for the application.

3.2 Physical Units for Scientific Applications

One very nice application of math expressions is to implement systems of physical units in scientific applications. Users can create include files containing definitions for the units of interest to them, after selecting a set of consistent base units in which

output will be displayed. For instance, one can define CGS units with a script such as the following:

```
define constant cm 1
define constant m 100*cm
define constant km 10^3*m
define constant gm 1
define constant kg 10^3*gm
define constant s  1
define constant min 60*s
define constant hr 60*min
define constant day 24*hr
```

An application which defined a time step and a domain volume could then be controlled by statements such as

```
set theTimeStep to 4*days+2*hours
set theDomainVolume to (4.5*km)^3
```

which are much easier to read and understand than the input files of many scientific applications!

3.3 Extending the *kScript* Programming Language

kScript is designed to be very easy to extend. To begin with, at the scripting level, user defined commands are very flexible. The `eval` command allows evaluating data (a string) as code (commands), much like in LISP and related languages. This means users can design new control structures directly in *kScript*, without access to the C++ implementation.

Programmers can extend *kScript* by defining new C++ classes with `>>` text input operators. For instance, the Keenan C++ Foundation Class Library defines a `doubleArray` class (an array of numbers). It also defines an input operator, so `doubleArray`s can be used by `cmdGen`. While none of the built-in commands in *kScript* take array arguments, it is trivial to define application commands which do. For instance, RUF, the Rice Unstructured Flow code, which uses *kScript* for its user interface, does allow users to input arrays in several situations. Since commands parse their own arguments by calling `>>`, the syntax for new types can be whatever the programmer wishes. In KFCL, array objects can be input in any of 6 different styles including one which reads the data of the array from a separate file, and one which uses curly brace notation much like a string expression.

This example highlights the fact that *kScript* is a context sensitive language. Unlike traditional context-free languages, argument parsing is under the control

of individual commands. This makes left context sensitivity easy to implement; moreover, the command interpreter class can handle one object look-ahead, making limited right context sensitivity also straightforward to use. `cmdGen` takes the place of a traditional parser generator (like `yacc` or `bison`), allowing programmers to quickly and easily define and implement new commands, complete with automatically generated on-line help.

References

[1] Keenan, P. T., *C++ and FORTRAN Timing Comparisons*, Dept. of Computational and Applied Mathematics Tech. Report #93–03, Rice University, 1993.

[2] Keenan, P. T., *RUF 1.0 User Manual: The Rice Unstructured Flow Code*, Dept. of Computational and Applied Mathematics Tech. Report #94–30, Rice University, 1994.