OSF DCE SIG **H. Melman (OSF)**
Request For Comments: 42.0 **June 1993**

DCE SHELL FUNCTIONAL SPECIFICATION

1. INTRODUCTION

A goal of DCE 1.1 is to make DCE easier to administer. One of the current problems with DCE is the fact that there are many different control programs for different components, with different syntaxes. Often, more than one is needed to perform certain operations, such as adding a new user to the cell. This Functional Specification describes a new program, known as the *DCE (Administrative) Shell*, abbreviated `dsh`, providing an administrative user interface that will augment the use of the existing programs, at least for commonly performed tasks.

While the licensees and users of DCE 1.0 have found it to be a useful distributed computing platform, they have also found it to be a difficult to administer environment. The feedback we have heard in this area has made the simplification of DCE administration an important goal of DCE 1.1. The long term solution to the problem is to make use of DME, however, this option is not available in the DCE 1.1 timeframe.

The fact that DCE is composed of several different components orginating from different vender companies, is readily apparent when viewing the administrative tools. Many of the difficulties in administering DCE arise from the need to use many different tools that behave inconsistently and operate at a very low level relative to administrative **tasks**[1] (e.g., add user to the cell, configure a client machine, backup, etc.). This functional specification describes a new administrative tool known as the DCE Shell which can perform many of the operations of existing control programs via a new consistent command-line interface. It is expected that DCE **administrators** will use the DCE Shell instead of the existing **control programs** to perform most daily operations.

The DCE Shell will include functionality from the following control programs: `rpccp`, `cdscp`, `rgy_edit`, `acl_edit`, and `dtscp`. That is, it will be able to manipulate data (e.g., objects, directories, entries, groups, principals, accounts, ACL's, etc.) stored in the various databases (e.g., the registry, clearinghouses, ACL managers, etc.). It will also be able to perform certain management operations (e.g, server shutdown, ping, get status, etc.). An important feature of the DCE Shell is that it will provide a script language including but not limited to: variables, control structures, and the ability to create new procedures. This will allow administrators to create their own task **commands**. The first version of DCE Shell will include several such task commands implemented as DCE Shell scripts.

The DCE Shell Language will be based on **Tcl**, an embeddable Tool Control Language developed at the University of California at Berkley. DCE Shell commands will be implemented as Tcl commands in a Tcl interpreter. This document assumes knowledge of Tcl and to a limited extent its syntax. Included in the long Appendix A is the Tcl man page; it can be used as both a tutorial and as a reference.

2. TERMINOLOGY

The following are the ‘‘new’’ terms used in this document.

(a) **Administrator**

The person (or one of several people) responsible for the installation and maintenance of a DCE Cell at an end-user site. They perform various administrative operations and **tasks**. While performing these duties they must have the proper permissions, so they must be members of the appropriate administrative groups.

---

1. This **boldface** convention is used to introduce terminology the first time it appears in this paper. See the Terminology section, below, for definitions.

(b) **Argument**

A parameter given to a command. **Commands** take zero or more arguments followed by zero or more **switches**.

(c) **Command**

Used in this document to mean a Tcl command or DCE Shell script that will be executed if entered as a command (Tcl provides a $PATH-like facility). Most all DCE Shell commands are named for the type of **object** that they operate on (e.g., ‘‘the `acl` command’’). This word is also used to refer to object-operation pairs (e.g., ‘‘the `acl show` command’’).

(d) **Command Shell**

Used to refer to traditional shells such as the Bourne Shell, Korn Shell, and C Shell. Confusion sometimes results with the name of the DCE Shell; this term is used to avoid problems.

(e) **Control Programs**

All of the administrative user interfaces provided with DCE 1.0. This includes but is not limited to: `rpccp`, `cdscp`, `rgy_edit`, `acl_edit`, and `dtscp`.

(f) **Element**

Existing DCE terminology uses this term to refer to elements of NSI profiles. DCE Shell uses **member** for this purpose. In the DCE Shell (and in Tcl), element is used to refer to the contents of lists.

(g) **Member**

Existing DCE terminology uses this term to refer solely to members of groups (both NSI groups and registry groups). Other arbitrary terms are used to refer to objects in other types of containers (e.g., NSI profiles contain **elements**). The arbitrary choice of words has caused confusion for some users. DCE Shell uses the word member to refer to anything stored in any container.

(h) **Operation**

DCE Shell commands are usually named for the type of object they act on. The first argument to the command is usually an operation to perform on that object.

(i) **String Syntax**

A format of a data structure that is understood by DCE Shell and existing control programs. In some cases, data structures have two syntaxs, one this is used in existing control programs, and one that is more natural for DCE Shell. DCE Shell supports the use of both.

(j) **Switch**

Tagged arguments to commands. All switches begin with a − (dash) and are found after all **arguments** in DCE Shell command lines. Some switches are optional and some are required. Some switches take values and some do not.

(k) **Task**

A dictionary defines task as ‘‘a difficult or tedious undertaking.’’ Here it is used to specify a high level operation that an administrator performs, such as add a new user, or configure a new machine. A task is above the level of DCE components (e.g, add principal and account to registry), and usually involves operations in several different components.

(l) **Tcl**

An embeddable Tool Command Language. Tcl refers to both the language and to a library (libtcl) that can be linked into a program which provides it. Commands can be added to the language via both C code and Tcl scripts. The DCE Shell language can be considered a dialect of Tcl, i.e., it is Tcl plus some DCE-specific commands.

(m) **Tcl Syntax**

A format of a data structure that is understood by DCE Shell. In some cases, data structures have two syntaxs, one this is used in existing control programs, and one that is more natural for DCE Shell. DCE Shell supports the use of both.

3. TARGET

The DCE Shell will have a command-line interface and is meant to be used by DCE administrators. The emphasis of the first release will be on the most commonly performed tasks of DCE cell administrators.

Developers will probably find an advantage to using DCE Shell, however they are not the primary target user.[2] For example, the most commonly performed task of a DCE developer is to configure a new cell. However, it is not expected that this will be the same for DCE administrators.

4. GOALS AND NON-GOALS

The following are goals for the DCE Shell:

(a) The major goal is to simplify the administration of a DCE cell by creating a single powerful and consistent administrative user interface.

(b) Emphasis in scheduling development resources must be placed on common everyday administrative duties rather than more esoteric configuration management. Administrators will get more use from a simpler way to add a new user or to modify a CDS entry and its ACL, rather than a new way to create a registry or to initialize a new CDS namespace.

(c) The DCE shell will augment use of existing control programs, it will *not* replace them. Existing control programs will continue to be installed with DCE, and their use will probably remain required for some administrative operations. See Appendix B for a list of existing control programs commands and the DCE Shell commands that replace them.

(d) Commands must be remotely invocable. This is currently true for most existing control program operations. The extension of those commands that are not remotely invocable is desired.

(e) The DCE shell must be portable to a variety of different platforms and management situations. Vendors must be able to integrate it into their own management systems.

The following are not goals of the DCE Shell:

(a) The DCE shell will only be used to administer DCE itself, it will *not* be used to administer applications that are written with DCE.

(b) The DCE shell will not handle extended registry attributes.

(c) A graphical user interface (GUI) is desirable, but a command line interface is: more portable to a wider variety of platforms, scriptable, and usable over telnet lines and on machines not equipped with GUI's. Therefore, the command line interface is more important for the first release of the DCE shell.

(d) This is not DME. The DCE shell is not meant to replace the need for DME, but rather to substitute for its absence in a very limited manner. Nothing in the design of the DCE shell should preclude the use of DME in the future.

---

2. It is expected that during the course of DCE 1.1 development, developers who have access to early versions of DCE Shell, will produce task-oriented scripts that may be of use to DCE administrators. As many of these as is possible and practical will be included in the first release of DCE Shell.

5. REQUIREMENTS

The following are the requirements of the DCE Shell as specified in [RFC 30] and how they are met by the program described in this functional specification:

(a) Existing Control Program Functionality

- (i) Stop servers — Met with `stop` operation.
- (ii) Show/add/modify/delete data — Met with the `show`, `create`, `modify`, `delete`, `addto`, and `rmfrom` operations on various commands.
- (iii) Show status — Met with `status` operation.
- (iv) Remote operations — Met by most commands, see User Interface section for details.

(b) Usability

- (i) Common syntax for all functions — Met with Tcl and designed syntax.
- (ii) Help — Met with `help` and `usage` operations on all commands.
- (iii) Provide command line recall/editing — Met with command-line editing facility.
- (iv) Scriptability — Met with Tcl's `source` command and with the interpretation of an **argument** as a script file to execute.

(c) New Functionality

- (i) Administer serviceability — Met with the `log` command.
- (ii) Administer auditing — Could be met with new commands, see the section *Probable Additions* below.
- (iii) Programmability — Met with use of Tcl.
- (iv) Extensibility — Met with Tcl's `proc` command.
- (v) Bind to different servers or data
- (vi) Shell escape — Met with the `shell` command and Tcl's `exec` command.
- (vii) Administration of little files
- (viii) DCE ping — Met with the `ping` operation.
- (ix) Security

(d) Meet Other DCE 1.1 Goals

- (i) code cleanup — Will followed the DCE Coding Style Guide [RFC 34].
- (ii) internationalization — The new DCE Shell code will follow proper i18n coding practices. Tcl will be internationalized.
- (iii) serviceability — Will use `sams` for message catalogs and messaging API's. There seems to be no need to use `dce_svc_*` interfaces for logging. See [RFC 24] and [RFC 23] for details.

6. FUNCTIONAL DEFINITION

The DCE Shell language is a dialect of Tcl. For a list of base Tcl commands, see Appendix A. The DCE Shell provides command-line recall and editing.

All of the DCE Shell commands will be implemented as Tcl commands either in C code or as Tcl scripts. To the DCE Shell user there is no visible difference. All commands return string values and have been designed so that they may be combined together; the output of one command can be used as input to another.

6.1. Invocation

The DCE Shell is a command-line user interface for administrative commands. There are several methods of invocation:

(a) The user invokes the `dsh` and then sees the DCE Shell prompt:

```
dsh>
```

(b) The user invokes the `dsh` with one argument which is a filename of a DCE Shell script. The script is run and then the DCE Shell exits. This method of invocation allows interpreter files (those with `#!/bin/dsh` as the first line) to work.

(c) The user invokes the `dsh` with the `-c` argument followed by commands (all on one line). The commands are executed and then the DCE Shell terminates. For example:

```
$ dsh -c directory create /.:/foo
```

6.2. Start-up Scripts

When the DCE Shell is invoked, it executes the following scripts in the order shown. Note the syntax used for the script locations is Tcl.

(a) `[info library]/init.tcl`

Contains standard Tcl initialization scripts on a per-host basis. Contains definitions for the `unknown` command and the `auto_load` facility.

(b) `$dsh_library/init.dsh`

Contains DCE Shell scripts on a per-host basis. DCE Shell scripts implementing commands and tasks are stored here.

(c) `$env(HOME)/.dshrc`

This file stores user customizations.

7. DATA STRUCTURES

This section provides a brief introduction to Tcl data types and then presents several DCE Shell data structures. For a complete reference of Tcl data types, see the Tcl man page in Appendix A.

In Tcl, there is only one data type, strings. All commands, expressions, variable values, and procedure return values are strings. Some Tcl commands interpret the strings in different manners. For example, `eval` treats its argument as an expression. The argument it recieves is a string. If the argument is `4 + 5` then the returned result is the string `9`.

Other Tcl commands interpret their string arguments as lists. In Tcl lists consist of elements spearated by white space (space, tab, or newline). The following are three examples of lists with three elements:

```
tom dick harry

a b c

1 2 !
```

There are three meta-characters that are treated specially when parsing lists. They are:

(a) Braces are used for grouping. If a left brace is encountered, then the element is terminated by the matching right brace, instead of the next whitespace. Braces nest. The following are also examples of lists with three elements:

```
tom {dick harry} {sue mary jane}

a {b {c d e}} f
```

(b) Double quotes are also used for grouping, however, there are two differences between them and braces. First they don't nest. Second Tcl will perform command, variable, and backslash substitutions for items within double quotes. Tcl does not perform these substitutions when braces are used.

(c) Backslashes are used to escape metacharacters, and to insert non-printing characters. Standard substitutions occur such as `\t` is a tab character, and `\n` is a newline. Braces and double quotes can be inserted in lists with backslashes as follows, again these example lists all have three elements:

```
a b\ c d

a b \}
```

The net effect of using these metacharacters and tcl commands on a command line, is that lists appear as elements surrounded by braces, and the top level of braces is not present on output. This last point is tricky; it has to do with how commands are parsed. The best way to explain it, is with an example. The Tcl command `lindex` takes two arguments, a list and an integer (remember, both of these are just strings, and it is the `lindex` command that interprets these string arguments specially). It returns the element from the list indicated by the second argument. Elements in lists are numbered from zero. So the following command:

```
lindex {a b c} 1
```

returns the string `b`. The braces in the example are used to group the elements of the list so that they are passed as one argument to the lindex command. Now, the following example returns a list:

```
lindex {a {b c} d} 1
```

The list returned is `b c`, there are no braces around it, but it is a perfectly valid list. To prove this, lets pass it to the `llength` command which takes one arugment, a list, and returns the length as an integer (in Tcl, square brackets perform command substitution, much like backquotes do in command shells):

```
llength [lindex {a {b c} d} 1]
```

This returns the value `2` since the list `b c` has two elements.

In DCE Shell, most data structures are lists. Often two syntaxes are recognized by the DCE Shell for the same data. This is to provide backward compatibility. If present, the two syntaxes are know as the **string syntax** and the **Tcl Syntax**.

7.1. Attributes

In DCE Shell, attributes are represented as a list. The first elelment is the attribute type. This is usually some string name, but in some cases, OID's or UUID's are acceptable. The following elements in the list are the values. If the attribute is single-valued then the attribute list will have two elements, multi-valued attributes will have more. Attribute lists are lists of attributes. The following is an attribute list of two single-valued attributes:

```
{alifetime 2} {plifetime none}
```

7.2. ACLs

An ACL is an Access Control List which consists of a set of ACL Entries. Each ACL Entry is defined as a type, an optional key or value, and a required set of permissions. In access checks, the permissions granted to a user are established by checking the type and key of each ACL Entry and or'ing the permissions described therein.

Each application is free to define new types and permission bits, however, `acl_edit` understands several types which are listed below:[3]

(a) `user_obj`
(b) `group_obj`
(c) `other_obj`
(d) `user`
(e) `group`
(f) `foreign_user`
(g) `foreign_group`
(h) `foreign_other`

---

3. The new ‘‘delegate’’ types are to be added to `acl_edit` during DCE 1.1 development to support delegation.

(i) `any_other`
(j) `mask_obj`
(k) `unauthenticated`
(l) `extended`
(m) `user_obj_delegate`
(n) `group_obj_delegate`
(o) `other_delegate`
(p) `user_delegate`
(q) `group_delegate`
(r) `foreign_user_delegate`
(s) `foreign_group_delegate`
(t) `foreign_other_delegate`
(u) `any_other_delegate`

7.2.1. String syntax

The string syntax of ACL Entries is as follows:

type[:*key*]:*permissions*

Examples of 9 ACL Entries follow:[4]

```
unauthenticated:-r-----

user_obj:crwx---

user:britten:crwx---

user:mahler:-rwx---

foreign_user:/.../C=US/O=OSF/OU=dce/pro/bach:crwxidt

group_obj:-rwx---

group:dds:-rwx---

any_other:-r-----

extended:c417faf8-8340-11c9-ace3-\
08001e5559bb.a.b.c.a1.4.0a0b0c0d:-rwx---
```

7.2.2. Tcl syntax

The Tcl Syntax of ACL Entries is a list of two or three elements. The first element is the type, the optional second element is the key, and the last element is the permission bits. The permission bits are represented as a single character if the permission is granted and as a - (dash) if it is not. An ACL is a list of ACL Entries. An example of an ACL is as follows:

```
{{unauthenticated -r-----}
 {user_obj crwx---}
 {user britten crwx---}
 {user mahler -rwx---}
 {foreign_user /.../C=US/O=OSF/OU=dce/pro/bach crwxidt}
```

4. Line-terminating ''\'' denotes line-folding of long lines, sometimes as a display metacharacter and sometimes as a TCL metacharacter (the distinction will always be clear from context).

```
 {group_obj -rwx---}
 {group dds -rwx---}
 {any_other -r-----}
 {extended c417faf8-8340-11c9-ace3-\
08001e5559bb.a.b.c.a1.4.0a0b0c0d -rwx---}}
```

7.3. Bindings

In the DCE Shell, all bindings are represented as string bindings which contain four pieces of information: a protocol sequence, a network address, an endpoint, and an optional object UUID. Different implementations may support different protocol sequences and different network address formats.

7.3.1. String syntax

The string syntax of a string binding is as follows:

object-UUID`@`*protseq*`:`*address*`[`*endpoint*`]`

The *endpoint* can be preceded by the string `endpoint=`. There can be no whitespace in the string binding. Examples follow:

```
ncadg_udp_ip:130.105.96.3[1234]

001c8234-74ff-1be9-b11c-08002b0f59bb@ncadg_udp_ip:\
130.105.96.3\[1234\]
```

Note that in Tcl square brackets are used to indicate command substitution, so if a string binding in the string syntax is entered the brackets must be preceded by a backslash (i.e., escaped) or enclosed in braces (i.e., quoted). Examples follow:

```
{ncadg_udp_ip:130.105.96.3[1234]}

ncadg_udp_ip:130.105.96.3\[1234\]
```

7.3.2. Tcl syntax

The Tcl Syntax is a list of 3 or 4 elements as follows:

protseq address endpoint

or

protseq address endpoint object-UUID

If an object UUID is to be entered but an endpoint is not, then there must be a null element in the list (specified as `{}`). Examples follow:

```
ncadg_udp_ip 130.105.96.3 1234

001c8234-74ff-1be9-b11c-08002b0f59bb ncadg_udp_ip \
130.105.96.3 1234

ncadg_udp_ip 130.105.96.3 {} 001c8234-74ff-1be9-b11c-\
08002b0f59bb
```

7.4. Interface Identifiers

An interface identifier describes an interface. Specifically, it contains the UUID, major version number, and minor version number of an interface.

7.4.1. String syntax

The string syntax is as follows, where *UUID* is the string syntax of a UUID.

UUID,*major*.*minor*

An example follows:

```
0067b39e-67ac-1be9-abb5-08002b0f59bb,1.2
```

7.4.2. Tcl syntax

The Tcl Syntax is a list of two elements: the UUID in its string syntax and the major and minor version numbers separated by period. An example follows:

```
0067b39e-67ac-1be9-abb5-08002b0f59bb 1.2
```

7.5. Routing Specifiers

Serviceability routing information is specified as a list of routing elements. Each routing element contains three fields containing PCS data:

(a) The first field specifies a message severity. It must be one of the following:

- (i) `FATAL` Fatal error, about to exit
- (ii) `EXIT` Normal exit
- (iii) `WARNING` Error detected, program proceeding
- (iv) `NOTICE` Informational notice
- (v) `VERBOSE` Verbose informational notice

(b) The second field specifies how the messages should be processed. It must be one of the following:

- (i) `BINFILE` Write binary log entry
- (ii) `TEXTFILE` Write human-readable text
- (iii) `FILE` Equivalent to `TEXTFILE`
- (iv) `DISCARD` Do not record this level
- (v) `STDOUT` Write human-readable text to standard output
- (vi) `STDERR` Write human-readable text to standard error

A limited form of log-rolling is available. Any of the file specifiers, above, may be followed by .*gens*.*count* where *gens* is an integer less then 100 indicating the number of files to be kept and *count* is an integer indicating the number of entries per file. No more then *count* entries will be written to a file, before a new file is started. After *gens* files have been written, the files wrap back around to the beginning. The file names are formed by appending .*gen* to the end of the filename, where *gen* runs from one to the last generation number.

Note that if any debug messages (calls to DCE_SVC_DEBUG) that are routed through a `BINFILE` route will lose the types of any arguments.

(c) The third field specifies where the messages should be sent. Presently, this will always be a filename, although it can be blank for the last three specifiers given above. This field may contain a `%ld` string which will be replaced by the process-ID of the currently-running process.

The Tcl syntax of a routing specificiation is a list of three elements. The first element is the severity, the second is the how specification, and the third is the where specification.

```
{severity how where}
```

For example, to send fatal errors to the console, ignore normal exit reports, and send informational messages to a temporary binary log, use the following routing specification:

```
{{FATAL TEXTFILE /dev/console}
 {EXIT DISCARD}
 {NOTICE BINFILE /tmp/log%ld}}
```

7.6. Debug Routing Specifiers

A debug routing specification is a list of debug routing elements. Each debug routing element contains four fields:

(a) The first field is the name of a serviceability component.

(b) The second field is a list of subcomponents and levels separated by a period. If the subcomponent is an asterisk then the level is set for all subcomponents of that component. The level should be a number between zero and nine. This list is parsed in order, so the asterisk may be used to set the initial level for all subcomponets; later entries override earlier ones.

(c) The third field specifies how the messages should be processed. It is identical to the second field of a serviceability routing element.

(d) The fourth field specifies where the messages should be sent. It is identical to the third field of a serviceability routing element.

The Tcl syntax of a debug routing specification is a list of four elements, one for each of the above fields. The second element may itself be a list. For example, to set RPC debugging at a minimal level so that all messages go to standard error, use the following:

```
{rpc *.1 STDERR}
```

The following example sets DTS debugging messages for the general, events, and msgs subcomponents (at the levels indicated for each) to go a logfile:

```
{dts {general.1 events.0 msgs.3} TEXTFILE /tmp/log}
```

7.7. Other Data Types

There are three other data types mentioned in this document that all have a string syntax and no other Tcl Syntax. They are UUID's, UTC times, and Names. For a description of these string syntaxes, see the DCE Documentation.

8. USER INTERFACES

The DCE Shell will present a command-line interface to the user with command-line recall and editing capabilities. The commands and command-line editing interfaces are described below.

The commands available to the user are those known by the Tcl interpreter. In the DCE Shell, the interpreter will know about the Tcl built-in commands and the additional DCE Shell commands described below. If a command name that Tcl does not know about is invoked, then Tcl calls the Tcl built-in command `unknown` with the entered command. The user is free to define this command as they wish to have a variety of things happen. By default, Tcl defines the `unknown` command to use the following heuristic:

(a) See if the autoload facility can locate the command in a Tcl script file. If so, load it and execute it.

(b) See if the command exists as an executable UNIX program. If so, "exec" the command.

(c) If the command was invoked at top-level:

   (i) See if the command requests `csh`-like history substitution in one of the common forms !!, !<number>, or ^old^new. If so, emulate `csh`'s history substitution.

   (ii) See if the command is a unique abbreviation for another command. If so, invoke the command.

This means that from the `dsh>` prompt, a user can type `ls` to see a listing of the files in the current directory, or `rm foo` to delete the file `foo` (see about filename completion in the Command-line Editing Commands section).

8.1. DCE Shell Commands

DCE Shell commands are implemented as procedures in a the Tcl interpreter. Commands return strings, and the main interpreter loop displays the results to the user. Errors are displayed as well. In most cases a Tcl objectoriented approach has been used to define the commands. In this approach commands represent a type of object, and are named the same as the object.

```
object operation name [-switch [value]]...
```

The first argument to the command specifies an operation to perform on an object, the second argument identifies the particular object to operate on (usually by name) and the third and following arguments are **switches** and values. Some commands might require one or more switches. Switches have a leading `-` and are a full word. Some switches take a value which immediately follows the switch. All arguments are strings, and all commands return strings as values. Examples of Tcl commands are the following:

(a) `file exists` *filename*
(b) `file readable` *filename*
(c) `history add` *command*
(d) `string compare` *string1 string2*
(e) `string length` *string*
(f) `string toupper` *string*

The DCE Shell will define commands in a similar manner. Commands will be named for a type of object to operate on, and the operations will be implemented for each object. Operation names will be consistent whenever appropriate (e.g., create will always be used to create an instance of the object, delete will always be used to destroy an instance). This syntax organization will be used whenever possible, but in some cases (such as `quit`, `exit` and `shell`) such a mapping would be too contrived and will be avoided. The syntax is described below.

This section describes the following in various sub-sections:

(a) *Error Handling* — The overall approach to error handling.

(b) *Common Operations* — These are operations that are common to many but not all objects.

(c) *Common Switches* — These are switches that are common to many but not all operations.

(d) *Objects* — These are the object commands that the DCE Shell will support. Each of the objects listed in this section has a command associated with it and the operations supported are described. Variations in the operations of a particular object versus the common operations are described in this section, as well as any non-common operations supported by the object.

(e) *Tasks* — These are the administrative tasks that the DCE Shell will support. They are implemented as object commands as to have similar syntaxes as other DCE Shell commands. They are more complex and are typically implemented in Tcl code.

8.1.1. Error handling

The DCE Shell will make use of Tcl's native error handling facilities. At the C code level, all Tcl commands return a code and a string. The code is usually either `TCL_OK` or `TCL_ERROR`. The returned string is either the normal result of the command or the error message. Tcl stores the ''stack-trace'' of nested errors, and it is accessible to the user. Tcl also provides facilities for scripts to catch errors and invoke error handlers (errors are one of 4 types of exceptions that Tcl supports).

Also there are two global Tcl variables visible to the DCE Shell, `errorInfo` and `errorCode`. The former contains the stack-trace of the error messages, and the later is meant to be machine readable information. The `errorCode` variable is a relatively new addition to Tcl so it's use is limited. Currently the commands that do I/O and such set `errorCode` to be a list where the first element is `UNIX` and the next two elements are the Unix name for the error, such as `ENOENT` and a human-readable message describing the error.

In general, DCE Shell commands will return either a list (e.g., the `show` operation), or the value `0` on success. On error, they will return the message string of the error, the Tcl Error exception (which can be caught with the Tcl `catch` command), and will set `errorCode` as appropriate to a list where the first element is `DCE` and the second is the numeric value of the status code. A DCE Shell script which traps an error is shown below:

```
# This routine returns 1 if the the argument is the name
# of an nsi group, 0 if it is not.  It uses the show operation
# to determine if it is a group.  If the command works then
# it is a valid group, if not, an error is generated and
# caught by the Tcl catch command and the routine returns 0.

proc isnsi-group {name} {
  if {[catch {nsi-group show $name}]==0} {return 1} {return 0}
}
```

8.1.2. Common operations

The following operations are implemented as part of all object commands, except where noted.

(a) `addto`

Adds an object to a container, for example it is used to add a **member** to an NSI group, and is used to add an **element** to an NSI profile. It is to be implemented for all objects that represent containers. Returns `0` on success. The argument is the name of the container. Switches are:

(i) `-member`

Used to specify the name of the member to be added to the container.

(b) `create`

Creates a new instance of the object. Takes one argument which is the name of the instance to be created. Returns `0` on success.

(c) `delete`

Destroys an instance of the object. Takes one argument which is the name of the instance to be deleted. Returns `0` on success.

(d) `help`

Returns a verbose help message about the object and all supported operations on the object.

(e) `list`

Returns a list of the names of all the members of a container. It is to be implemented for all objects that represent containers. Switches are:

(i) `-recurse`

Returns the names of all descendants (not just immediate children) of the specified instance.

(ii) `-all`

Supported by some objects to show the names of all instances of the object. It is used in place of an instance name. For example, `principals show -all`, will list the names of all principals in the registered in the cell.

(f) `operations`

Returns a machine readable list of the operations supported by the object.

(g) `modify`

Allows editing of the instance. Will not be available to all objects. Returns `0` on success. The specific modification is described by the use of one or more of the following switches. If more than one is used, then the whole modify operation is treated atomically in that either it all will work, or non of it will. Also the order of switches matters. They are parsed left to right and the operations are performed in that order.

(i) `-add`

Used to add an attribute to object or merely to add values to an existing attribute. If only an attribute type is specified after the switch, then it is added to the object with no values. If an attribute type and value(s) is given, then they are add to the object, with the attribute being created if necessary.

(ii) `-remove`

Used to remove an entire attribute or merely some values from an object. If only the attribute type is specified after the switch, then the entire attribute is removed. If an attribute type and value is specified then only that value is removed. If it is the only value present in the attribute then the entire attribute is removed.

(iii) `-change`

Used to change one attribute value to another. Logically it is identical to doing a `-remove` followed by an `-add`.

(h) `rmfrom`

Removes an object from a container, for example is used to remove a member from an NSI group, and is used to remove an element from an NSI profile. It is to be implemented for all objects that represent containers. The argument is the name of the container. Switches are:

(i) `-member`

Used to specify the name of the member to be removed from the container.

(i) `show`

Returns the contents of the instance. Invoking this operation on a nsi-profile would return all information about all the the attributes of that object. Some objects use this as a synonym of `list` when container objects have no contents other than the names of the **members**. Takes one argument which is the name of the instance to be shown. Switches are:

(i) `-recurse`

Returns a list of the contents of all descendants of the specified instance.

(ii) `-all`

Supported by some objects to show the contents of all instances of the object. It is used in place of an instance name. For example, `accounts show -all` will list all the data associated with each account in the cell.

(j) `update`

Tells the instance to update with any replicas of itself. In CDS terminology this will perform a skulk on a directory, in DTS it will cause a server to synchronize. It is to be implemented for all objects that support replication. Returns `0` on success.

(k) `usage`

Returns a terse usage message about the object and the supported operations and switches.

8.1.3. Common switches

The following are switches that are implemented for some commands. When used they are behave in a common manner, although there may be slight differences in the implementations.

(a) `-annotation`

Used to specify an annotation string.

(b) `-attribute`

Used to specify an attribute consisting of a type and one or more values. Attributes are represented as a list where the first element is the type and the following elements are the values. Attribute types are specified with either OIDs or their string representation.

(c) `-binding`

Used to specify a string binding.

(d) `-default`

Used to specify that a operation is to be invoked on the default profile member (element). When this switch is specified usually other switches used to identify a particular profile member are ignored.

(e) `-in`

Used to specify that the operation is to be performed in a particular database or place. For example, the `directory create` command uses this switch to specify an alternate clearinghouse to create the CDS directory in.

(f) `-interface`

Used to specify an interface identifier.

(g) `-member`

Used to identify an item in a container. It takes a required value.

(h) `-object`

Used to specify an object UUID.

(i) `-recurse`

Used to have an operation recursively evaluate results. Some operations allow this switch to take a numeric value specifying the maximum depth of the recursion.

(j) `-syntax`

Specifies the naming syntax to use. DCE (and DCE Shell) will only support one syntax (the DCE syntax) for DCE 1.1. This syntax will be the default for all commands so this switch will rarely be used.

(k) `-version`

Used to specify how the version specification of the interface identifier given in the `-interface` switch is used. Requires a value which is one of the following:

(i) `all`

The interface version is ignored, all values are acceptable.

(ii) `exact`

Both the major and minor versions must match the specified versions.

(iii) `compatible`

The major version must match the specified version, and the minor version must be greater than or equal to the specified version.

(iv) `major-only`

The major version must match the specified version; the minor version is ignored.

(v) `upto`

The major and minor versions must be less than or equal to those specified.

8.1.4. Objects

The following are the object commands supported by DCE Shell. The supported operations are listed for each command. The `help`, `usage`, and `operations` operations are not listed as they are supported by all of these commands.

Users can add more objects by using the Tcl `proc` command to define a new command. The tasks described in the next section are examples of doing this.

8.1.4.1. nsi-entry

The `nsi-entry` object represents an NSI server entry. NSI server entries are named in the DCE namespace, so each of these operations takes one argument specifying a name. Each server entry contains a set of object UUIDs, and a set of interface identifier/string binding pairs. Supports the following operations:

(a) `create`

Creates an empty entry. Since an empty entry is the same as an empty group or profile, calling `nsi-entry create` is the same as calling `nsi-group create` or `nsi-profile create`. Recognizes one switch: `-syntax`.

(b) `delete`

Deletes the specified entry. Recognizes the `-syntax` switch.

(c) `export`

Exports binding information to the specified entry (identified by the argument). If the entry does not exist it is created. Uses the `-interface`, `-binding`, `-object` and `-syntax` switches to specify what to export.

This operation will also be able to export codeset and other attributes as specified in the RPC Internationalization specifications.

(d) `import`

Returns a string binding from the specified entry. Uses the `-interface`, `-version`, `-object` and `-syntax` switches to specify matching bindings. Also accepts the `-max` switch to specify a number of string bindings to return. If the value is greater than one, then a list of as many matching bindings less than or equal to the value is returned.

This operation will also be able to distinguish acceptable entries based on codeset and other attributes as specified in the RPC Internationalization specifications.

(e) `show`

Returns a list containing the binding information in the NSI server entry. The list is composed of two lists. The first list is also a list, where the first element is an interface identifier, and the rest of the elements are string bindings. The second list is a list of Object UUID's exported by the server. This operation recognizes the `-interface` switch so that only string bindings matching that interface identifier are returned.

This operation will also return codeset and other attributes as specified in the RPC Internationalization specifications.

(f) `unexport`

Removes binding information from an entry. Uses the `-interface`, `-version`, `-object` and `-syntax` switches to specify what to unexport.

8.1.4.2. nsi-group

The `nsi-group` object represents an NSI group. Each NSI group is named in the DCE namespace; therefore each operation described below takes one argument which is the name of the group to act upon. An NSI group is a container of names of either NSI server entries or other NSI groups; there is no data in the group other than the name of the members. Supports the following operations:

(a) `addto`

Adds a member to the specified group. The value of the `-member` switch is the name of an NSI entry, group, or profile; it does not have to actually exist. Also takes the `-syntax` switch.

(b) `create`

Creates an empty group. Since an empty group is the same as an empty entry or profile, calling `nsi-group create` is the same as calling `nsi-entry create` or `nsi-profile create`. Recognizes one switch: `-syntax`.

(c) `delete`

Deletes the specified group. Recognizes the `-syntax` switch. The `-recurse` switch can be used to delete the group and all members; it does not take a numeric argument limiting the levels of recursion. When using this switch members are removed whether they are nsi-groups, nsi-entries, or nsi-profiles.

(d) `import`

Returns a string binding from the specified group. Uses the `-interface`, `-version`, `-object` and `-syntax` switches to specify matching bindings. Also accepts the `-max` switch to specify a number of string bindings to return. If the value is greater than one, then a list of as many matching bindings less than or equal to the value is returned.

(e) `list`

Returns a list of the names of all members of the specified group. The `-recurse` switch is used to list all the names in sub-groups as well.

(f) `rmfrom`

Removes a member from the specified group. The value of the `-member` switch is the name of an nsi-entry, nsi-group, or nsi-profile; it does not have to actually exist outside of the group. Also takes the `-syntax` switch.

(g) `show`

Is a synonym for the `list` operation.

8.1.4.3. nsi-profile

The `nsi-profile` object represents an NSI profile. Each NSI Profile has a name in the DCE namespace; therefore each operation described below takes one argument which is the name of the profile to be acted upon. An NSI Profile consists of members (also known as elements in other DCE documentation) which can be either NSI server entries, NSI groups, or other NSI profiles; therefore each member of a profile has a name in the DCE namespace. Each profile member also contains an interface identifier, a priority, and an optional annotation. Supports the following operations:

(a) `addto`

Adds a member to the specified profile. The value of the `-member` switch is the name of an nsi-entry, nsi-group, or nsi-profile; it does not have to actually exist. Also takes the `-interface`, `-priority`, `-annotation`, and `-syntax` switch. Also accepts a `-default` switch, in which case, the above switches are ignored and add operation is performed on the default profile member.

(b) `create`

Creates an empty profile. Since an empty profile is the same as an empty entry or profile, calling `nsi-profile create` is the same as calling `nsi-entry create` or `nsi-group create`. Recognizes one switch: `-syntax`.

(c) `delete`

Deletes the specified profile. Recognizes the `-syntax` switch. The `-recurse` switch can be used to delete the profile and all members; it does not take a numeric argument limiting the levels of recursion. When using this switch members are removed whether they are nsi-profiles, nsi-entries, or nsi-groups.

(d) `import`

Returns a string binding from the specified profile. Uses the `-interface`, `-version`, `-object` and `-syntax` switches to specify matching bindings. Also accepts the `-max` switch to specify a number of string bindings to return. If the value is greater than one, then a list of as many matching bindings less than or equal to the value is returned.

(e) `list`

Returns the names of all members of the specified profile. The `-recurse` switch is used to list all the names of sub-profiles as well.

(f) `rmfrom`

Removes one or more members from the specified profile. The members to be removed are those that match the values given in the following switches: `-member`, `-interface`, `-annotation`, `-syntax`. The value of the `-member` switch is the name of an nsi-entry, nsi-group, or nsi-profile; it does not have to actually exist outside of the profile. Also accepts a `-default` switch, in which case, the above switches are ignored and the default profile member is removed.

(g) `show`

Returns a list of all members of the profile. Each member of the profile is shown as a list of 4 items: the interface, the member name, the priority, and the annotation string. Only those members that match the values specified by the following optional switches are returned: `-interface`, `-version`, `-annotation`, `-member`, `-syntax`. Also accepts a `-default` switch, in which case, the above switches are ignored and the default profile member is returned. The `-recurse` switch (with a value) can be used to recursively display the elements of the profile.

8.1.4.4. mapping

The `mapping` object represents RPC endpoint mappings. Since mappings have no names, none of these operations take an instance argument. Mappings contain an interface identifier and one or more string bindings; optionally they contain object UUID's and an annotation. Supports the following operations:

(a) `create`

Creates a new mapping in the endpoint map database. Uses the following switches: `-interface`, `-binding`, `-object`, and `-annotation`.

(b) `delete`

Deletes the specified mapping from the endpoint map database. Uses the following switches: `-interface`, `-version`, `-binding`, and `-object`. Also supports a `-host` switch to specify the use of the endpoint map on another host.

(c) `show`

Returns a list of information about each endpoint that matches the specification of the supported switches: `-interface`, `-version`, and `-object`. Also supports the `-all` switch to return all the mappings in the endpoint map database. Also supports a `-host` switch to specify the use of the endpoint map on another host.

8.1.4.5. principal

The `principal` object represents Registry principals. In the registry, each principal has a primary name and a value for each of the following attributes: `fullname`, `uuid`, `uid`, `quota` and `alias`. The `alias` attribute's value is either yes or no (this is subject to internationalization).

Unless otherwise noted, all of the operations of this object take one argument which is the name of the principal to act on. It must be a principal name, not the name of the database object that contains the registry information about the principal (i.e., it should not begin with `/.:/sec/principal/`). Supports the following operations:

(a) `create`

Creates a new principal. This command returns the name of the created principal. The switches are:

(i) `-alias`

If this switch is present (it takes no value), then the principal name passed in as an argument is added as an alias of the uid specified by the `-uid` switch. When using this switch, the `-uid` switch is required. For example, the following command creates an alias called `postmaster` for the principal with uid 1234:

```
principal create postmaster -uid 1234 \
-alias
```

(ii) `-fullname`

The full name of the principal to be added to the registry. The value is a string, if it contains spaces, it must be in quotes or braces. This switch is required.

(iii) `-quota`

Specifies the pricipal's object creation quota, the total number of registry objects that can be created by the principal. It is either a non-negative number or the string `unlimited`. If you do not specify this option, the object creation quota defaults to `unlimited`.

(iv) `-uid`

The User Identifier for the principal. No two principals can have the same uid, however, aliases can share one uid with a principal. If this switch is not present, then a uid will be assigned to the principal automatically.

(b) `delete`

This operation deletes the principal from the registry. When a principal is deleted, the principals account is deleted as well. This command returns the name of the deleted principal. Note that the name given as an argument to this command can be either the primary name of a principal or an alias name of a principal. In either case, an account associated with that name is deleted.

(c) `list`

Returns a list of names of all the principals in the registry. It takes no arguments.

(d) `modify`

This operation is used to change infomation about a principal. The `-add` and `-remove` switches are not supported, as each attribute associated with a principal must always have a value, however, the `-change` switch can be used to modify the value of any one of the attributes. The value of the `-change` switch is an attribute list describing the new values for the specified attributes. Also supports the `-alias` and `-primary` switches to change the entered name to an alias or primary name respectively.

(e) `show`

This operation returns a list describing the specified principal. The first element in the list is the name of the principal. The following elements are lists of two elements where the first element is the name of the attribute, and the second element is the value of that attribute. In addition to the attributes described above, there is another attribute returned by this command called `groups`, the value of which is a list of group names that the principal is a member of.

8.1.4.6. group

The `group` object represents Registry groups. In the registry, each group has a primary name and a value for each of the following attributes: `fullname`, `gid`, and `inplist`. This last attribute has a value of either yes or no (yes this will be internationalized) and determines if the group is included on the Project List. Groups are collections of principal names.

Unless otherwise noted, all of the operations of this object take one argument which is the name of the group to act on. It must be a group name, not the name of the database object that contains the registry information about the group (i.e., it should not begin with `/.:/sec/group/`). Supports the following operations:

(a) `addto`

This command is used to add a member to the group specified as the argument. The value of the required `-member` switch is a principal name which will be added to the group. To add more than one member to the group, invoke the command with multiple `-member` switches.

(b) `create`

Creates a new group. This command returns the name of the created group. The switches are:

(i) `-fullname`

The full name of the group to be added to the registry. The value is a string, if it contains spaces, it must be in quotes or braces. This switch is required.

(ii) `-gid`

The Group Identifier for the group. No two groups can have the same gid, however, aliases can share one gid with a group. If this switch is not present, then a gid will be assigned to the group automatically.

(iii) `-inplist`

Specifies if the group is to be included in the principal's project list. The value of this field is an internationalized yes or no. If this switch is not included in the command, then the yes value is used as a default.

(iv) `-alias`

If this switch is present (it takes no value), then the group name passed in as an argument is added as an alias of the gid specified by the `-gid` switch. When using this switch, the `-gid` switch is required.

(c) `delete`

This operation deletes the group from the registry. When a group is deleted, any accounts associated with that group are deleted as well.

(d) `list`

The `list` operation is used to list the name of all groups in the registry. It takes no arguments.

(e) `modify`

This command is used to change infomation about a group. The `-add` and `-remove` switches are not supported, as each attribute associated with a group must always have a value, however, the `-change` switch can be used to modify the value of any one of the attributes. The value of the `-change` switch is an attribute list describing the new values for the specified attributes. Also supports the `-alias` and `-primary` switches to change the given name to an alias or primary name respectively.

(f) `rmfrom`

This command is used to remove a member from a group. The value of the required `-member` switch is a principal name which will be removed from the group. To remove more than one member from the group, invoke the command with multiple `-member` switches.

When a member is removed from a group, any accounts associated with that principal and group are deleted. Remember that accounts are associated with a principal, a group, and an organization; so the previous statement means that any accounts where *both* the principal name and group name match those in given to this command are removed, but accounts where only one match are untouched.

(g) `show`

This command returns a list describing a group. The first element in the list is the name of the group. The following elements are lists of two elements where the first element is the name of the attribute, and the second element is the value of that attribute. In addition to the attributes described above, this command returns an attribute called `members` for which the value is a list of principal names which are members of the group.

8.1.4.7. org

The `org` object represents Registry organizations. Organizations are collections of principal names. In the registry, each organization has a primary name and a value for each of the following attributes: `fullname` and `orgid`. Also, each organization has policies associated with them that regulate such things as account and password lifetimes for all accounts associated with a particular organization. In fact, these policies are the main feature that orgainizations provided. There are two types of policies referred to here, those associated with a particular organization, and those that are registry wide. Please note that there are other policies that are associated with accounts (and they also can be registry-wide).

Unless otherwise noted, all of the operations of this object take one argument which is the name of the organization to act on. It must be an organization name, not the name of the database object that contains the registry information about the organization (i.e., it should not begin with `/.:/sec/organization/`). Supports the following operations:

(a) `addto`

This command is used to add a member to the organization specified as the argument. The value of the required `-member` switch is a principal name which will be added to the organization. To add more than one member to the organization, invoke the command with multiple `-member` switches.

(b) `create`

Creates a new organization. This command returns the name of the created organization. The switches are:

(i) `-fullname`

The full name of the organization to be added to the registry. The value is a string, if it contains spaces, it must be in quotes or braces. This switch is required.

(ii) `-orgid`

The Organization Identifier for the organization. No two organizations can have the same orgid. If this switch is not present, then an orgid will be assigned to the organization automatically.

(c) `delete`

This operation deletes the organization from the registry. When an organization is deleted, any accounts associated with that organization are deleted as well. This command returns the name of the deleted organization.

(d) `list`

The `list` operation is used to list the name of all organizations in the registry. It takes no arguments.

(e) `modify`

This command is used to change infomation about an organization. The `-add` and `-remove` switches are not supported, as each attribute associated with an organization must always have a value, however, the `-change` switch can be used to modify the value of any one of the attributes. The value of the `-change` switch is an attribute list describing the new values for the specified attributes.

(f) `policy`

This operation is used to change policies associated with an organization. It is very similar to the `modify` operation in that it takes a `-change` switch which specifies one of the policies listed below and a new value for it. For information on changing registry-wide policies, see the section on the `registry` object. The policies are:

(i) `alifetime`

This policy defines the lifespan of accounts. It is specified as either an integer (for which the units are days) or the string `unlimited`.

(ii) `palpha`

This policy defines whether or not passwords can consist entirely of alphanumeric characters. Its value is an internationalized yes or no string.

(iii) `pexpiration`

This policy defines a date on which a password will expire. The date is entered as an internatinalized date string or the string `none` in which case, there is no expiration date for the password.

(iv) `plifetime`

This policy defines the lifespace of passwords. It is specified as either an integer (for which the units are days) or the string `unlimited`.

(v) `pminlen`

This policy defines the minimum number of characters in a password. Its value is a positive integer or the integer `0` which means there is no minimum length.

(vi) `pspaces`

This policy defines whether or not passwords can consist entirely of spaces. Its value is an internationalized yes or no string.

Note to Reviewers: I18n work needs to be done to see if ''characters'' above should be changed to ''bytes''. Are passwords limited to PCS? Note that this involves internationalizing the registry, there is no i18n work needed for DCE Shell here; this is merely how the registry interprets these values. Also I'm not sure if the current registry protocol will allow a change to one policy or if I have to change them all. If I have to change them all, then either I'll get the default values, or if that is too slow, I'll require all of the above switches. Ideas?

(g) `rmfrom`

This command is used to remove a member from an organization. The value of the required `-member` switch is a principal name or a list of principal names which will be removed from the organization.

When a member is removed from an organization, any accounts associated with that principal and organization are deleted. Remember that accounts are associated with a principal, a group, and an organization; so the previous statement means that any accounts where *both* the principal name and organization name match those given to this command are removed, but accounts where only one match are untouched.

(h) `show`

This command returns a list describing an organization. The first element in the list is the name of the organization. The following elements are lists of two elements where the first element is the name of the attribute (or policy), and the second element is the value of that attribute (or policy). In addition to the attributes described above, this command returns an attribute called `members` for which the value is a list of principal names which are members of the organization. This operation takes a `-effective` switch that returns the polices in effect (they can be different from the organization policies due to conflicts with registry wide policies).

8.1.4.8. account

The `account` object represents Registry accounts. Each account is associated with one principal, one group and one organization. This triple forms the name of each account. Each `account` command takes three arguments which are the principal, group, and organization that name the account instance. They are separate arguments, and therefore are separated by spaces. If any contain spaces in the name (*Is this even legal?*) then they must be surrounded by either double quotes or braces. Aliases are differentiated for principals and groups, so one principal can have multiple accounts under different alias names.

Each account also has a password associated with it. Accounts contain the following attributes:

(a) `avalid`

A flag set to determine account validity. An account with an Account Valid Flag set to no is invalid and cannot log in. The default is yes.

(b) `client`

A flag set to indicate whether or not the account is for a principal that can act as a client. If you set this flag to yes, the principal is able to log in to the account and acquire tickets for authentication. The default is yes.

(c) `description`

A text string (limited to PCS) typically used to describe the use of the account.

(d) `dupkey`

A flag set to determine if tickets issued to the account's principal can have duplicate keys. The default is no.

(e) `expires`

The date on which the account expires. To renew the account, change the date in this field. The default is `none`.

(f) `forwardableticket`

A flag set to determine whether a new ticket-granting ticket with a network address that differs from the present ticket-granting ticket network address can be issued to the account's principal. (The Proxiable flag performs the same function for service tickets.) The default is yes.

(g) `goodsince`

The date and time the account was last known to be in an uncompromised state. Any tickets granted before this date are invalid. Control over this date is especially useful if you know that an account's password was compromised. Changing the password can prevent the unauthorized principal from accessing the system again using that password, but it does not prevent the principal from accessing the system components for which tickets were obtained fraudulently before the password was changed. To eliminate the principal's access to the system, the tickets must be cancelled. If you set the Good Since Date to the date and time the compromised password was changed, all tickets issued before that time become invalid and the unauthorized principal's system access is eliminated. When the account is initially created, the Good Since Date is set to the current date.

(h) `home`

The directory in which the principal is placed at login.

(i) `postdatedticket`

A flag set to determine if tickets with a start time some time in the future can be issued to the account's principal. The default is no.

(j) `proxiableticket`

A flag set to determine whether or not a new ticket with a different network address than the present ticket can be issued to the account's principal. (The Forwardable flag performs the same function for ticket-granting tickets.) The default is no.

(k) `pvalid`

A flag set to determine whether the current password is valid. If this flag is set to no, the next time a principal logs in to the account, the system prompts the principal to change the password. (Note that this flag is separate from the password expiration policy, which sets time limits on password validity.) The default is yes.

(l) `renewableticket`

A flag set to determine if the ticket-granting ticket issued to the account's principal can be renewed. If this flag is set to yes, the Authentication service renews the ticket-granting ticket if its lifetime is valid. The default is yes.

(m) `server`

A flag set to indicate whether or not the account is for a principal that can act as a server. If the account is for a server that engages in authenticated communications, set this flag to yes. The default is yes.

(n) `shell`

The shell that is executed when a principal logs in.

(o) `tgtauth`

A flag set to determine whether or not service tickets issued to the account's principal use the standard DCE ticket-granting ticket authentication mechanism. The default is yes.

Each account also contains information about the following policies.

(a) `maxticketlife`

The maximum amount of time in hours that a ticket can be valid. When a client requests a ticket to a server, the lifetime granted to the ticket takes into account the Maximum Certificate Lifetime set for both the server and the client. In other words, the lifetime, cannot exceed the shorter of the server's or client's Maximum Certificate Lifetime. If you do not specify a maximum certificate lifetime for an account, the Maximum Certificate Lifetime defined as registry authorization policy is used.

(b) `maxticketrenew`

The amount of time in hours before a principal's ticket-granting ticket expires and that principal must log in again to the system to reauthenticate and obtain another ticket-granting ticket. The lifetime of the principal's service tickets can never exceed the lifetime of the principal's ticket-granting ticket. The shorter you make the Maximum Certificate Renewable, the greater the security of the system. However, since principals must log in again to renew their ticket-granting ticket, the time needs to take into consideration user convenience and the level of security required. If you do not specify a maximum certificate renewable lifetime for a account, the maximum certificate renewable lifetime defined as registry authorization policy is used.

Supports the following operations:

(a) `create`

Creates an account with the name specified from the three arguments. Switches are accepted to specify values for each attribute and policy. They may be defaulted. Switches are named the same as the attributes and policies listed above with a leading `-`.

(b) `delete`

Deletes the specified account from the registry. Returns the name of the deleted account.

(c) `list`

The `list` operation is used to list the name of all accounts in the registry.

(d) `modify`

This command is used to change infomation about an account. The `-add` and `-remove` switches are not supported, as each attribute associated with an account must always have a value, however, the `-change` switch can be used to modify the value of any one of the attributes. The value of the `-change` switch is an attribute list describing the new values for the specified attributes.

(e) `policy`

This operation is used to change policies associated with an account. It is very similar to the `modify` operation in that it takes a `-change` switch which specifies one of the policies listed above and a new value for it. For information on changing registry-wide policies, see the section on the `registry` object.

(f) `show`

This command returns a list describing an account. The first element in the list is the name of the account. The following elements are lists of two elements where the first element is the name of the attribute (or policy), and the second element is the value of that attribute (or policy). This operation takes a `-effective` switch that returns the polices in effect (they can be different from the account policies due to conflicts with registry wide policies).

8.1.4.9. registry

The `registry` object represents policies that are registry wide. These refer to policies for organizations (standard policies in `rgy_edit` terminology) and accounts (auth policies), and to properties. For a list of the policies supported, see the `org` and `account` object descriptions.

The registry also contains attributes that affect its overall operation. These are referred to as properties in `rgy_edit`. The list of attributes is as follows:

(a) `defticketlife`

The Default Ticket Lifetime is the default lifetime (in hours) for tickets issued to principals in this cell's registry.

(b) `lowgid`

The low UNIX ID attribute is the starting point for UNIX IDs automatically generated by the Security Service when a group is created. (You can explicitly enter a lower UNIX ID than this number; it applies only to automatically generated numbers.)

(c) `loworgid`

The low UNIX ID attribute is the starting point for UNIX IDs automatically generated by the Security Service when an organization is created. (You can explicitly enter a lower UNIX ID than this number; it applies only to automatically generated numbers.)

(d) `lowuid`

The low UNIX ID attribute is the starting point for UNIX IDs automatically generated by the Security Service when a principal is created. (You can explicitly enter a lower UNIX ID than this number; it applies only to automatically generated numbers.)

(e) `maxuid`

The maximum UNIX ID attribute lets you set the highest number that can be supplied as a UNIX ID when principals are created. This maximum applies to both the system-generated and user-entered UNIX IDs.

(f) `minticketlife`

The Minimum Ticket Lifetime is the minimum amount of time (in minutes) before the principal's ticket must be renewed. This renewal is performed automatically with no intervention on the part of the user. The shorter you make the Minimum Ticket Lifetime, the greater the security of the system. However, extremely frequent renewal can degrade system performance. The Minimum Ticket Lifetime you set needs to take into consideration system performance and the level of security your cell requires. The Minimum Ticket Lifetime can be set only as a registry attribute. It cannot be set for individual accounts.

(g) `pwhidden`

The hidden password attribute determines whether encrypted passwords are displayed or not. If you set this attribute to yes, an asterisk is displayed in place of the encrypted password in command output and files where passwords are displayed.

(h) `readonly`

If the read-only attribute is set to yes, the master registry database, which normally can be updated, is read-only. You can use it to prevent update operations from being performed on the registry.

Supports the following operations:

(a) `modify`

This command is used to change attributes of the registry. The `-add` and `-remove` switches are not supported, as each attribute must always have a value and the list of attributes is not extensible, however, the `-change` switch can be used to modify the value of any one of the attributes. The value of the `-change` switch is an attribute list describing the new values for the specified attributes.

(b) `policy`

This command is used to change policies that are registry-wide. There are no `-add` and `-remove` switches supported, as each policy must always have a value and they are fixed, however, the `-change` switch can be used to modify the value of any one of the policies. The value of the `-change` switch is an attribute list describing the new values for the specified polices.

(c) `show`

Returns a list of all the polices and their registry wide values.

8.1.4.10. keytab

The `keytab` object represents keytab files that store server passwords (and password version numbers) on hosts. Since local file access is required to the keytab files, all `keytab` commands must be performed locally. The instance name given to the command is the name of the keytab file.

Supports the following operations:

(a) `create`

Adds the server's or machine's pasword to the keytab file and to the registry. In `rgy_edit` an explicit switch `-r` had to be given to have the registry updated, but DCE Shell always does this. Switches are:

(i) `-principal`

the name of the principal (server or machine) whose password you are creating. This switch is required.

(ii) `-password`

If this switch is present, then its value is used as the new password. If it is not present, then a password is automatically generated. An inline Tcl script could be invoked as the value of this switch to prompt the user for a password without echoing it to the screen.

```
dsh> keytab create -principal \
> hosts/machine/self -password [getpw]
```

(b) `delete`

Deletes a server or machine principal's password entry from the keytab file. Requires a `-principal` switch to identify the server or machine principal for which the password is to be deleted. Also requires a `-version` switch to specify the version of the password to delete. Versions can be found by using the `show` operation.

(c) `show`

Returns information about a keytab file. Returns a list in which each element is a list of two elements. The first being the principal and the second being the version number of the password (an integer from 1 to 255). Takes a `-principal` switch to limit the returned list to only be for the specified principals.

8.1.4.11. acl

The `acl` object represents ACL's on any object. See the Data Structure section for the syntax of ACLs. The `acl_edit` use model is a batched one. Each `acl_edit` session modifies the ACL on one object. The commands are structured so that individual ACL Entries are modified locally, but nothing is written back to the server until either a `commit` or `exit` command is performed. The DCE Shell will not follow this model. Each `acl` command will write to the server, however, commands will allow more than one modification at once.

Also, `acl_edit` binds to the object once at invocation, and retains that binding throughout the session. DCE Shell should not destroy the binding to the object after the command completes, but rather, it should save it for some amount of time after the command, so that future commands do not have to establish a binding to the object.

See the Data Structures section for a description of the syntax of ACL's and ACL Entries.

Supports the following operations:

(a) `access`

Returns the permissions granted in the specified object's ACL to the principal that invoked the command.

(b) `cell`

Sets the name of the default cell in the ACL.

(c) `delete`

Removes all ACL Entries from the object, except the `user_obj` entry, if it exists.

(d) `modify`

This operation is used to change one or more individual ACL entries. The specific operation to perform is described with one or more of the following switches: `-add`, `-remove`, and `-change`. Either all changes are made to the object, or none are. Multiple actions can be specified on the command line, they will be processed in order. All of the switches take a value which is either one ACL Entry, or a list of ACL Entries. A switch may be specified more than once.

(e) `permissions`

Returns a list describing the permissions associated with the object. Each element of the list is a list of two items, the first is the permission token, the second is a description of the permission. For example:

```
dsh> acl permissions /.:/hosts
{{r read}
 {w write}
 {d delete}
 {t test}
 {c control}
 {i insert}
 {a administer}}
```

(f) `replace`

Replaces the entire ACL on the object specified by the argument, with the value of the `-acl` switch. The syntax of the value of the `-acl` switch is a list of ACL entries.

(g) `show`

Returns a list of the ACL Entries for the specified object.

8.1.4.12. object

The `object` object represents CDS objects. An object in the CDS directory service, has a name in the DCE Cell namespace, and a collection of attributes associated with it. Attributes consist of a type and one or more values. Every object is the child of a directory. See the `directory` command for information on how to get a list of objects beneath a directory.

There is no concept of current working directory, or cd'ing through the CDS namespace. Certainly this is a possibility for future expansion.

The following are the CDS defined attributes that may be present in CDS Objects.

(a) `CDS_Class`

Specifies the class to which an object belongs.

(b) `CDS_ClassVersion`

Contains the version number of the object's class. This allows applications to build in compatibility with entries created by earlier versions.

(c) `CDS_CTS`

Specifies the creation timestamp (CTS) of this object entry.

(d) `CDS_ObjectUUID`

Specifies a unique identifier for the object being referenced.

(e) `CDS_UTS`

Specifies the timestamp of the most recent update to an attribute of the object entry.

Supports the following operations:

(a) `create`

Creates a new object. The name of the new object is given as an argument to the command. Takes one or more `-attribute` switches to specify attributes to be included in the object.

(b) `delete`

Deletes an object. Takes one argument indicating the name of the object to delete.

(c) `modify`

Used to add or remove attributes and their values from the object.

(d) `show`

Returns an attribute list of the attributes in the object.

8.1.4.13. directory

The `directory` object represents CDS directories. CDS directories are containers for other objects, links, and other directories (as well as clearinghouses). If any of these items reside in a directory, they are said to be called children of that directory. Directories also contain attributes that may be viewed or modified.

This object also represents the CDS concept of replicas. Replicas are read-only copies of directories stored in other clearinghouses. Several of the supported operations take switches to indicate that the command is to operate on a replica instead of the master directory.

The following are the CDS defined attributes that may be present in directories and replicas in CDS:

(a) `CDS_AllUpTo`

Indicates the date and time of the last successful skulk on the directory. All replicas of the directory are guaranteed to receive all updates whose timestamps are less than the value of this attribute.

(b) `CDS_Convergence`

Specifies the degree of consistency among replicas. This attribute's value is defined as one of the following:

(i) `low`

CDS does not immediately propagate an update. The next skulk distributes all updates that occurred since the previous skulk. Skulks occur at least once every 24 hours.

(ii) `medium`

CDS attempts to immediately propagate an update to all replicas. If the attempt fails, the next scheduled skulk makes the replicas consistent. Skulks occur at least once every 12 hours.

(iii) `high`

CDS attempts to immediately propagate an update to all replicas. If the attempt fails (for example, if one of the replicas is unavailable), a skulk is scheduled for within one hour. Skulks usually occur at least once every 12 hours. Use this setting temporarily and briefly, because it uses extensive system resources.

By default, every directory inherits the convergence setting of its parent at creation time. The default setting on the root directory is `medium`.

(c) `CDS_CTS`

Specifies the creation timestamp (CTS) of the CDS directory.

(d) `CDS_DirectoryVersion`

Specifies the current version of the directory (derived from the `CDS_DirectoryVersion` attribute of the clearinghouse in which the directory was created). Multiple directory versions are supported in a cell.

(e) `CDS_Epoch`

A UUID that identifies a particular incarnation of the directory.

(f) `CDS_GDAPointers`

A set-valued attribute which is only present in the root directory of a cell. This attribute contains location information about registered Global Directory Agents for that cell, similar to the CDS_Replicas attribute. It is created and only used by a GDA.

(g) `CDS_InCHName`

Indicates whether a directory or any of its descendants can store clearinghouse names. If this value is `true`, the directory can store clearinghouse names. If it is `false`, the directory cannot store clearinghouse names.

(h) `CDS_LastSkulk`

Records the timestamp of the last skulk performed on this directory.

(i) `CDS_LastUpdate`

Records the timestamp of the most recent change to any attribute of a directory replica, or any change to an entry in the replica.

(j) `CDS_ObjectUUID`

Specifies the unique identifier of the directory.

(k) `CDS_ParentPointer`

Contains a pointer to this directory's parent in the namespace.

(l) `CDS_Replicas`

Specifies the address, UUID, and name of every clearinghouse where a copy of this directory is located. This attribute also specifies whether the replica in a particular clearinghouse is a master or read-only replica.

(m) `CDS_ReplicaState`

Specifies whether a directory replica can be accessed.

(n) `CDS_ReplicaType`

Indicates whether a directory replica is a master or read-only replica.

(o) `CDS_ReplicaVersion`

Specifies the version of a replica of the directory.

(p) `CDS_RingPointer`

Specifies the UUID of a clearinghouse containing another replica of this directory.

(q) `CDS_UpgradeTo`

A single-valued attribute used to control the upgrading of a directory from one version of CDS to another. By modifying this attribute, the process of upgrading a directory to a newer version of CDS may be initiated.

(r) `CDS_UTS`

Specifies the timestamp of the most recent update to an attribute of the directory.

Supports the following operations:

(a) `addto`

This operation creates a child pointer in the parent directory. The name of the parent directory is given as an argument to the command, and the name of the child pointer is given as the value of the required `-member` switch. The child pointer name entered should only contain the last RDN of the name. The full name of a clearinghouse that holds a replica of the child directory is given as the value to the required `-clearinghouse` switch.

This command is only needed to recreate a child pointer that was accidently deleted, i.e., in a troubleshooting situation. Normally child pointers are created internally by CDS when creating directories with the `directory create` command.

(b) `create`

Creates a new directory of the specified name. Returns the name of the created directory. Takes an optional `-in` switch to specify a clearinghouse to create the directory in. If this is not specified, the new directory is created in the same clearinghouse as the parent directory. Also takes a `-replica` switch which indicates that the a directory replica is created; when this switch is used, the `-in` switch is required.

(c) `delete`

Used to delete a directory. An optional `-recurse` switch is supported to remove the directory and everything (all directories, objects, links, and clearinghouses) beneath it. Also there is a `-replica` and a `-in` switch which must be used together to delete a replica instead of a directory. The clearinghouse the replica is in is specified as the value the `-in` switch.

(d) `list`

Returns a list of the names of all the descendants of a directory. In this case descendants includes all directories, objects, links, and clearinghouses of the directory. Since only the names are returned there is no way to tell the class of each name unless by convention (e.g., most clearinghouses end with `_ch`).

(e) `modify`

Used to add, remove or change attributes and their values from the directory.

(f) `rmfrom`

This operation deletes a child pointer from the directory specified as an argument. The RDN of the child must be entered as the value to the required `-member` switch.

This command is only needed to delete a child pointer that accidently remains after the child directory is deleted. Normally child pointers are removed internally by CDS when deleting directories with the `directory delete` command.

(g) `show`

Returns a list of attributes in the directory specified as an argument. Also takes a `-replica` and a `-in` switch to specify that the contents of a replica are to be returned. This will return the directory specific and replica specific attributes for the replica. The clearinghouse the replica is in is specified as the value the `-in` switch.

Also takes a `-child` switch with one value which is the last RDN of child pointer in this directory. In this case the returned list describes the attributes stored in the specified child pointer.

(h) `update`

Initiates a directory skulk of the directory specified as an argument to the command. The skulk is begun immediately and the command does not return until it is finished.

8.1.4.14. link

The `link` object represents CDS soft links. A soft link in the CDS directory service contains an attribute that has a name which is the name that the soft link points to. The soft link contains several built-in attributes, but users not free to add their own. Soft links can point to objects, directories, and other soft links.

The following are the CDS defined attributes that may be present in a CDS soft link entry.

(a) `CDS_CTS`

Specifies the creation timestamp (CTS) of this soft link.

(b) `CDS_LinkTarget`

Specifies the full name of the directory, object entry, or other soft link to which the soft link points.

(c) `CDS_LinkTimeout`

Specifies a timeout value after which the soft link is either renewed or deleted. Its value is a list of two elements:

(i) The expiration-time. A date and time after which CDS checks for the existence of the soft link's target and either extends or deletes the soft link. The value is specified as *yyyy-mm-dd-hh:mm:ss*, and portions of it can be defaulted.

(ii) The extension-time. A period of time by which to extend the soft link's expiration time (if the server has validated that the target still exists). The value is specified as *ddd-hh:mm:ss*, and portions of it can be defaulted.

(d) `CDS_UTS`

Specifies the timestamp of the most recent update to an attribute of the soft link.

Supports the following operations:

(a) `create`

Creates a new soft link. The name of the new soft link is given as an argument to the command. There is one required switch, `-linkto` which takes a single value, a name that this link is to point to. Additionally, the `-timeout` switch can be used to specify a value for the `CDS_LinkTimeout` attribute. If this attribute is not included in the command line then the new soft link is permanent and must be explicitly deleted.

(b) `delete`

Deletes the soft link. This command takes one argument which is the name of the soft link to delete.

(c) `modify`

The `modify` operation can be used to change two attributes of a soft link: `CDS_LinkTarget` and `CDS_LinkTimeout`. The value of the later is described above in the description of the `create` operation. The former defines what the soft link points to, its value is a name.

(d) `show`

Returns an attribute list for the softlink.

8.1.4.15. clearinghouse

The `clearinghouse` object represents CDS clearinghouses. Clearinghouses are databases located on CDS server machines that store the data in the CDS directory service. On the server machines are files that contain the actual clearinghouse data. Clearinghouses are also represented in the CDS namespace by an entry that contains information about the clearinghouse.

The following are the CDS defined attributes that may be present in a clearinghouse:

(a) `CDS_AllUpTo`

Indicates the date and time the clearinghouse object has been updated to reflect the `CDS_CHDirectories` attribute.

(b) `CDS_CHDirectories`

Specifies the full name and unique identifier (UUID) of every directory that has a replica in this clearinghouse.

(c) `CDS_CHLastAddress`

Specifies the current reported network address of the clearinghouse.

(d) `CDS_CHName`

Specifies the full name of the clearinghouse.

(e) `CDS_CHState`

Specifies the state of the clearinghouse. The state `on` indicates the clearinghouse is running and available.

(f) `CDS_CTS`

Specifies the creation timestamp (CTS) of the clearinghouse.

(g) `CDS_DirectoryVersion`

Specifies the current version of the directory in the clearinghouse in which the directory was created.

(h) `CDS_NSCellname`

Specifies the name of the cell in which the clearinghouse resides.

(i) `CDS_ObjectUUID`

Specifies the unique identifier of the clearinghouse.

(j) `CDS_ReplicaVersion`

Specifies the current version of the replica in which the directory was created.

(k) `CDS_UpgradeTo`

A single-valued attribute used to control the upgrading of a directory from one version of CDS to another. By modifying this attribute, the process of upgrading a directory to a newer version of CDS may be initiated.

(l) `CDS_UTS`

Specifies the timestamp of the most recent update to an attribute of the clearinghouse.

Supports the following operations:

(a) `create`

Creates a new clearinghouse on the local machine. Takes one argument which is the name of the new clearinghouse. Clearinghouses should only be named in the root directory (i.e., `/.:`). When this operation is invoked, a read-only replica of the root directory is stored in the new clearinghouse. Because the the process that creates the new clearinghouse initiates a skulk of the root directory, all replicas of the root should be reachable when you enter the command. To ensure this, perform an immediate skulk of `/.:` prior to invoking this command, using the `directory update /.:` command.

(b) `delete`

This operation deletes the clearinghouse given as an argument from the *local* machine. Clearinghouses that contain master replicas of directories may not be deleted. This command also automatically deletes all read-only replicas from the clearinghouse, however it is recommended that all read-only replicas be deleted by hand *before* invoking this command since invoking many skulks will cause the command to execute slower.

(c) `disable`

This operation removes knowledge of the specified clearinghouse from the server running on the *local* machine. It is useful when relocating a clearinghouse. Specifically the name of the prefix of the clearinghouse files is removed from the `/opt/dcelocal/var/directory/cds/cds_files` file and the local server is contacted and notified that the clearinghouse is disabled. The clearinghouse entry is *not* removed from the namespace, nor are the datafiles associated with the clearinghouse removed.

(d) `show`

Returns an attribute list showing the attributes of the clearinghouse named as an argument.

(e) `status`

Returns an attribute list showing the values of various counters that CDS keeps for clearinghouses.

(i) `badchs`

Specifies the number of times the `clearinghouse entry missing` event was generated

(ii) `baddata`

Specifies the number of times that the `data corruption` event was generated

(iii) `badskulks`

Specifies the number of times that a skulk of a directory, initiated from this clearinghouse, failed to complete— usually because one of the replicas in the replica set was unreachable

(iv) `badupgrades`

Specifies the number of times that upgrades failed

(v) `disables`

Specifies the number of times that the clearinghouse was disabled since it was last started

(vi) `enables`

Specifies the number of times that the clearinghouse was enabled since it was last started

(vii) `partials`

Specifies the number of requests directed to this clearinghouse that resulted in the return of a partial answer instead of satisfying the client's request

(viii) `reads`

Specifies the number of read operations directed to this clearinghouse

(ix) `rootlost`

Specifies the number of times the `root lost` event was generated

(x) `writes`

Specifies the number of write operations directed to this clearinghouse

8.1.4.16. cds-cache

The `cds-cache` object represents the cds cache on the local node. The CDS cache contains information about servers and clearinghouses known to the local machine, and also contains user data about CDS entries that have been read. Supports the following operations:

(a) `create`

This operation adds information about a CDS server to the local CDS cache. It takes one argument which is the name of the server to add to the cache. This is the simple name of the machine, not the name of entry in the DCE namespace for the server. There is also one required switch, `-binding`. This is the binding information for the server to be added to the CDS cache. The value must be a partial string binding.

(b) `delete`

This operation is used to remove information about a CDS server from the local CDS cache. It can only be used to remove information added with the `cds-cache create` command. It returns the name of the server removed from the CDS cache.

(c) `dump`

Returns all information stored in the local CDS cache. This command meant to be used when debugging CDS problems. It may be removed from a production environment.

(d) `show`

Returns information stored in the CDS cache. The argument is the name of either a clearinghouse or the simple name of a server about whose information from the cache is returned. Must be invoked with one of the two following switches:`-server` or `-clearinghouse` to identify the type of the argument.

8.1.4.17. dtsd

The `dtsd` object represents the `dtsd` process running on the local host. Since DTS is designed not to have stored data, this object represents the information in and about a process rather than some data stored somewhere as other objects do. Also this is not implemented as an instance of the `server` object (described below) sinceb objects, not instances, have operations associated with them. These commands all affect the local `dtsd` entity, they take no arguments.

There are a number of attributes assciated with the `dtsd` object. All of these can be viewed with the `show` operation, and many can be changed with the `modify` operation. Attribute arguments can contain a maximum of 80 characters and are recalculated to a normalized date format. For example if the input value is `0-0025:10:99.99999999`, the result is `1-01:11:39.990`. The attributes are:

(a) `autotdfchange`

Specifies whether automatic changes to the time differential factor are enabled or disabled. The value is either yes or no (subject to internationalization). The value is determined by the operating system (i.e., it cannot be changed with the `modify` operation).

(b) `checkinterval`

Specifies the amount of time between checks for faulty servers. Applicable only to servers that have external time providers. The default is `0-01:30:00.00`, and the range of the possible values is `0-00:00:30.000 - 10675199-02:48:05.478`.

(c) `clockadjrate`

Specifies the rate at which the DTS server or clerk entity adjusts the node's clock during a synchronization. This may not be set by a user, but it built into `dtsd`.

(d) `clockresolution`

Specifies the amount of time between system clock ticks. The value is determined by the operating system (i.e., it cannot be changed with the `modify` operation).

(e) `courier`

Specifies a server's interaction with the set of global servers. Possible values are:

(i) `backup`

The local server becomes a courier if none are available on the LAN.

(ii) `courier`

The local server synchronizes with the global set of servers.

(iii) `noncourier`

The local server does not synchronize with the global set of servers. This is the default.

(f) `epoch`

Specifies the server's epoch number. The default is `0`, and the range of possible values is `0-255`.

(g) `globaltimeout`

Specifies the amount of time the node waits for a response to a WAN synchronization request before sending another request or declaring a global server to be unavailable. The number of attempts made to reach the server is controlled by the `queryattempts` attribute. The default value is `0-00:00:15.000`, and the range of possible values is `0-00:00:00.000 - 0-00:10:00.000`.

(h) `localtimeout`

Specifies the amount of time the node waits for a response to a synchronization request before sending another request or declaring a server to be unavailable. The number of attempts made to reach the server is controlled by the `queryattempts` attribute. The default is `0-00:00:05.000`, and the range of possible values is `0-00:00:00.000 - 0-00:10:00.000`.

(i) `localservers`

Specifies the transport protocol and the local servers known by this DTS entity. This attribute is cannot be changed with the `modify` operation.

(j) `maxdriftrate`

Specifies the worst-case drift rate of the node's clock, in nanoseconds per second, as determined by the manufacturer's specifications (i.e., it cannot be changed with the `modify` operation).

(k) `maxinaccuracy`

Specifies the inaccuracy limit for the node. When the node exceeds the maximum inaccuracy setting, it attempts to synchronize. The default is `0-00:00:00.100`, and the range of possible values is `0-00:00:00.0 - 10675199-02:48:05.478`.[5]

---

5. This will probably be changed during DCE 1.0.3 to be a max of 24 hours. It really makes no sense for this value to be anything other than seconds. See DCE OT 7357.

(l) `minservers`

Specifies the minimum number of servers required for a synchronization. Settings of `1` or `2` may cause unreliable computed times. The default is `3` and the range of possible values is `1-10`.

(m) `nexttdfchange`

Specifies the future time at which the time differential factor is automatically changed. The value is determined by the operating system (i.e., it cannot be changed with the `modify` operation).

(n) `provider`

Specifies whether or not the entity used an external time-provider at the last successful synchronization. This attribute applies to servers only and may not be set by a user.

(o) `queryattempts`

Specifies the number of attempts that a node makes to contact a server before the node considers the server unavailable. The default is `3`, and the range of possible values is `1-10`.

(p) `serverentry`

Specifies a server's ACL entry name; The default setting is the following recommended value `/.:/hosts/`*`hostname`*`/dts-entity`

(q) `servergroup`

Specifies the security group name for the time servers within the cell. The default is `/.:/subsys/dce/dts-servers`.

(r) `serverprincipal`

Specifies a server's principal name for authentication purposes The default setting is the following recommended value `/.:/hosts/`*`hostname`*`/self`.

(s) `syncinterval`

Specifies the interval a node must wait to synchronize. Also specifies synchronization frequency when a node reaches the value specified by the `maxinaccuracy` attribute. For clerks the default is `0-00:10:00.0`, and the range of possible values is `0-00:00:30.0 - 01 00:00:00.00`. For servers the default is `0-00:02.00.0`, and the range of possible values `0-00:00:30.0 - 01 00:00:00.00`.

(t) `tdf`

Specifies the Time Differential Factor (TDF), which is the amount of time the server varies from Greenwich mean time (GMT) or Coordinated Universal Time (UTC). The default is `0-00:00:00.000`, and the range of possible values is `-13-00:00:00 - 13-00:00:00`. This may not be set by a user, but rather is obtained from various timezone information repositories (e.g., the `TZ` environment variable, kernel structures, etc.).

(u) `timerep`

Specifies the timestamp format used by the node. This may not be set by the user, it is built into a `dtsd`.

(v) `tolerance`

Specifies the maximum separation allowed between the local clock and the computed time before synchronizations become abrupt rather than gradual (monotonic). The default is `0-00:10:00.000`, and the range of possible values is `0-00:00:00.500 - 10675199-02:48:05.478`.

(w) `type`

Specifies whether the node is a DTS `server` or `clerk`.

(x) `version`

Specifies the DTS software version installed on the node. This attribute cannot be changed with the `modify` operation.

Supports the following operations:

(a) `config`

Requires one of two switches `-global` `-notglobal` to configure the local `dtsd` as a global server or not. The difference is whether they are listed in `/.:/cell-profile`. Returns the string `global` or `notglobal` to indicate the current (new) state of the `dtsd`.

(b) `modify`

Allows attributes to be changed with the `-change` switch.

(c) `show`

Returns an attribute list giving the values of the attributes listed above.

(d) `status`

Returns an attribute list that gives the values of various status attributes and counters that DTS keeps. The status values are:

(i) `global`

Specifies the set of global servers known by the node.

(ii) `lastsync`

Specifies the computed time at the last synchronization.

(iii) `local`

Specifies the set of local servers known by the node.

(iv) `state`

Specifies the state of the DTS entity.

[a] `off`

The DTS entity is disabled.

[b] `on`

The DTS entity is enabled.

[c] `syncing`

The DTS entity is synchronizing.

[d] `updating`

The DTS entity is updating the time.

(v) `time`

Specifies the current time on the node.

(vi) `uid`

Specifies the entity's unique identifier, which is generated when the entity is created.

The counters are:

(i) `abrupts`

Specifies the number of times the node clock has been set nonmonotonically (abruptly).

(ii) `badprotocols`

Specifies the number of times the local node failed to process a received message containing an incompatible protocol version.

(iii) `badrepversions`

Specifies the number of times the local node failed to process a received message containing an incompatible timestamp format.

(iv) `creation`

Specifies the time at which the DTS entity was created and the counters were initialized.

(v) `diffepochs`

Specifies the number of times the node received time response messages from servers or clerks that had epoch numbers different from its own.

(vi) `disabled`

Specifies the number of times the DTS has been disabled.

(vii) `enabled`

Specifies the number of times the DTS has been enabled.

(viii) `epochchanges`

Specifies the number of times the server's epoch has changed.

(ix) `faultyservers`

Specifies the number of times a server has detected faulty servers (other than itself).

(x) `invalids`

Specifies the number of times the local node failed to process a received message that had an unknown message type or incorrect length.

(xi) `noglobals`

Specifies the number of times the courier server could not contact any global servers.

(xii) `nointersections`

Specifies the number of times the node's time interval failed to intersect with the computed interval of the servers.

(xiii) `nomemory`

Specifies the number of times the node has been unable to allocate virtual memory.

(xiv) `notrespond`

Specifies the number of times the courier server could not contact a specific global server.

(xv) `providererrors`

Specifies the number of times the external time-provider signaled a failure or the node was unable to access the time-provider.

(xvi) `synchs`

Specifies the number of times the node successfully synchronized.

(xvii) `syserrors`

Specifies the number of times a DTS operation detected a system error.

(xviii) `toofewservers`

Specifies the number of times a node failed to synchronize because it could not contact the required minimum number of servers.

(xix) `toomanyservers`

Specifies the number of times the responses to a synchronization request overflowed the assigned system buffer.

(xx) `updates`

Specifies the number of times a server has attempted to update its clock.

(e) `stop`

This stops the `dtsd` process.

(f) `time`

Gradually adjusts the clock to the time specified by the argument.

(g) `update`

Causes the local `dtsd` to synchronize with DTS servers. The machine's clock is adjusted accordingly. By default the clock is adjusted gradually. This command also takes the optional `-abrupt` switch to have the clock set abruptly.

8.1.5. log

The `log` object represents the current state of routing for DCE serviceability messages of a process. There are two types of routing supported, serviceability routing and debug routing. Debug routing may be removed from a production environment.

The commands take one argument which is either a fully bound string binding or the name of a NSI server entry. See the Data Structures section for a description of the syntax of serviceability routing specifications and debug routing specifications.

Supports the following operations:

(a) `list`

Returns a list of serviceability components registered by the server. Also takes an optional `-comp` switch which is used to specify a component name. In this case, a list of sub-components is returned where each element of the list is a list of four elements: the subcomponent name, its level, the message id of the long description of the subcomponent, and the text of the long description of the subcomponent.

Example:

```
dsh> log list /.:/hosts/hostname/cds-server
{svc cds dts rpc sec}
dsh> log list /.:/hosts/hostname/cds-server -comp dts
{{general 0 0x12345671 "General server administration"}
 {events  0 0x12345672 "Events received and acted upon"}
 {arith   0 0x12345673 "Math operations"}
 {ctlmsgs 0 0x12345674 "Control messages received"}
 {msgs    0 0x12345675 "Messages received"}
 {states  0 0x12345676 "Server state transitions"}
 {threads 0 0x12345677 "Thread interactions"}
 {config  0 0x12345678 "Server/cell configuration"}
 {sync    0 0x12345679 "Server/synchronization \
interactions"}}
```

(b) `modify`

Is used to change the routing specifications. The `-add` and `-change` switches are synonymous. The `-remove` switch can be used to specify that messages are to be handled with `DISCARD`. Each switch takes a list of either serviceability or debug routing specifications.

(c) `show`

Returns a list describing the current serviceability routing settings for a server. Takes an optional `-debug` switch in which indicates that debug routing settings are to be returned instead of normal serviceability routing settings.

> *Note to Reviewers: Currently there is no serviceability API call to return the routing information that is currently in effect for a server. A new IDL routine would need to be added to the API that returns this information in order to support this operation.*

8.1.6. server

The `server` object represents instances of running servers. The commands take one argument which is either a fully bound string binding or the name of a NSI server entry. Supports the following operations:

(a) `ping`

Returns `1` if the server specified by the argument is listening for RPC requests, `0` if not. The argument must be a fully bound string binding.

8.1.7. Miscellaneous commands

(a) `dsh_initInterp`

This command will initialize a base Tcl interpreter will all the DCE Shell commands.

(b) `errtext`

Takes a DCE status code as an argument and returns the text of the associated message as found in the message catalogs. The argument can be in decimal, octal (leading `0`), or hexidecimal (leading `0x`) notation.

(c) `expand`

Takes a DCE name as an argument and returns the canonical form of the name.

(d) `login`

Performs a `dce_login`. The rest of the DCE Shell session uses the new login context. Takes the account to login as as an argument. The password is prompted for and not echoed to the screen. Also takes the option `-password` switch to enter a password.

(e) `logout`

Logs you out of the current login context as established with a previous `login` command.

(f) `quit`

Exits from the DCE Shell. A synonym of the Tcl builtin command `exit`.

(g) `resolve`

Takes a partial string binding and returns a fully bound string binding. Takes a `-interface` switch with an interface identifier as an argument to provide enough information for the mapping to occur.

(h) `shell`

Spawns a command shell for the user. The value of the `SHELL` environment variable is used to obtain the name of the shell to spawn. When the command shell terminates, control is returned to the DCE Shell.

8.1.8. Tasks

The following are administrator tasks to be implemented as DCE Shell Scripts. To the user, they will appear exactly like commands.

Note to Reviewers: I need to fill this section in with details of what these commands will do. For example, `user create` *will say something like: adds a principal to registry (and optionally a group and and organization as specified via switches) creates an account in the registry for the user, adds a entry to CDS with the name /.:/blah/blah/user, etc.*

(a) `cell status`

(b) `cell show`

(c) `cell ping`

(d) `cell status`

(e) `cell backup`

(f) `user create`

(g) `user delete`

(h) `user modify`

(i) `user show`

(j) `client config`

(k) `client unconfig`

(l) `server config`

(m) `server unconfig`

(n) `host start`

(o) `host stop`

(p) `host show`

(q) `host ping`

(r) `host status`

8.2. Command-line Editing Commands

A line may be edited before it is sent to the DCE Shell by typing control characters or escape sequences. A control character, shown as a ‘‘C-’’ followed by a letter, is typed by holding down the ‘‘control’’ key while the letter is typed. For example, `C-A` is a control-A. An escape sequence is entered by typing the ‘‘escape’’ key followed by one or more characters. The escape key is abbreviated as `ESC`. Note that unlike control keys, case matters in escape sequences; `ESC F` is not the same as `ESC f`.

An editing command may be typed anywhere on the line, not just at the beginning. In addition, a return may also be typed anywhere on the line, not just at the end.

Most editing commands may be given a repeat count, *n*, where *n* is a number. To enter a repeat count, type the escape key, the number, and then the command to execute. For example, `ESC 4 C-f` moves forward four characters. If a command may be given a repeat count then the text ‘‘[n]’’ is given at the end of its description.

The following control characters are accepted:

(a)	C-A	Move to the beginning of the line
(b)	C-B	Move left (backwards) [n]
(c)	C-D	Delete character [n]
(d)	C-E	Move to end of line
(e)	C-F	Move right (forwards) [n]
(f)	C-G	Ring the bell
(g)	C-H	Delete character before cursor (backspace key) [n]
(h)	C-I	Complete filename (tab key); see below

(i)	C-J	Done with line (return key)
(j)	C-K	Kill to end of line (or column [n])
(k)	C-L	Redisplay line
(l)	C-M	Done with line (alternate return key)
(m)	C-N	Get next line from history [n]
(n)	C-P	Get previous line from history [n]
(o)	C-R	Search backward (forward if [n]) through history for text; must start line if text begins with an uparrow
(p)	C-T	Transpose characters
(q)	C-V	Insert next character, even if it is an edit command
(r)	C-W	Wipe to the mark
(s)	C-XC-X	Exchange current location and mark
(t)	C-Y	Yank back last killed text
(u)	C-[	Start an escape sequence (escape key)
(v)	C-]*c*	Move forward to next character *c*
(w)	C-?	Delete character before cursor (delete key) [n]

The following escape sequences are provided.

(a)	ESC C-H	Delete previous word (backspace key) [n]
(b)	ESC DEL	Delete previous word (delete key) [n]
(c)	ESC SPC	Set the mark (space key); see C-XC-X and C-Y above
(d)	ESC .	Get the last (or [n]'th) word from previous line
(e)	ESC ?	Show possible completions; see below
(f)	ESC <	Move to start of history
(g)	ESC >	Move to end of history
(h)	ESC b	Move backward a word [n]
(i)	ESC d	Delete word under cursor [n]
(j)	ESC f	Move forward a word [n]
(k)	ESC l	Make word lowercase [n]
(l)	ESC u	Make word uppercase [n]
(m)	ESC y	Yank back last killed text
(n)	ESC w	Make area up to mark yankable
(o)	ESC *nn*	Set repeat count to the number *nn*

There is filename completion. Suppose the root directory has the following files in it:

bin vmunix core vmunix.old

If you type `rm /v` and then the tab key, as much of the name as possible will be finished off by adding `munix`. Because the name is not unique, it will then beep. If you type the escape key and a question mark, it will display the two choices. If you then type a period and a tab, the filename will be finished for you:

`rm /v[TAB]`*`munix.`*`[TAB]`*`old`*

The tab key is shown by `[TAB]` and the automatically entered text is shown in italics.

8.3. Probable Additions

There are two areas of functionality that may be added to the first version of the DCE Shell:

(a) The auditing project includes an `auditcp` command to administer the `auditd` server and a basic audit analysis tool which will probably be limited to dumping a binary audit log. Both of these should probably be implemented in DCE Shell. The should probably be two objects implemented to handle the operations, one for setting the event specifications and event objects, and the other to represent a binary audit log file. Further discussions with IBM (the auditing providers) are required. See [RFC 28] and [RFC 29] for details on the auditing design.

(b) HP is planning on adding an RPC interface to allow access to read and modify the `dce_cf.db` file on a DCE host machine. If this is done, then DCE Shell should implement a command line access to this file to allow remote configuration changes. These commands would probably fall under the `host` object.

8.4. Internationalization

Areas that need to be addressed by internationalization include:

(a) DCE Shell scripts will be limited to run in the locale in which they were written. Any scripts shipped by OSF will be in the C Locale.

(b) Portions of Tcl will need to be internationalized:

  (i) Basic parsing routines, since the only datatype in Tcl is a string.

  (ii) Most output strings strings (errors but not prompts) will need to be isolated into message catalogs.

  (iii) The `scan`, `string`, `lsort`, `lsearch`, and `lreplace` commands.

  (iv) The regular expression handling.

  (v) The `glob` command.

9. API'S

All commands in the DCE Shell will be implemented as Tcl commands. Therefore, they are user-visible and can be used in DCE Shell scripts to implement new commands. Some commands are implemented in C since they make DCE library calls; they are also visible to users. However, often there will be another command written in Tcl that will wrap the C command to present a better interface to the user. In this case both commands are available to the user, but only the higher-level command will be documented.

The lower-level commands can be found in the Tcl source, and can be considered in some sense to be DCE Shell API's. They will not typically be called directly by users, but will be visible to the ''Tcl-knowledgable'' (i.e., they will be listed if an `info commands` is performed).

For example, there will be one `create` command implemented in Tcl, it will read its first argument to determine the type of object being created (e.g., a group, a principal, a directory, etc.); it will then call another routine (usually implemented in C) for the lower-level functionality. This lower-level routine will be named the same as the upper-level command with an underscore and the type of object appended to it (e.g., `create_group`, `create_principal`, `create_directory`, etc.).

10. REMOTE INTERFACES

Not applicable

11. MANAGEMENT INTERFACES

Not applicable

12. RESTRICTIONS AND LIMITATIONS

Not applicable

13. OTHER COMPONENT DEPENDENCIES

The DCE Shell will be dependent on the following DCE components:

(a) NSI

(b) CDSPI

(c) The ACL interface

(d) The rgy interface

(e) DTS interfaces

(f) serviceability interfaces

It will also depend on the format of the following datafiles:

(a) `dce_cf.db`

14. COMPATIBILITY

The existing control programs will still be released, DCE Shell will be released in addition to them. As this is the first release of DCE Shell there is nothing for it to be compatible with. It makes use of DCE API's, so as long as they remain compatible with older versions of DCE servers, DCE Shell should be able to contact pre-1.1 DCE services.

15. STANDARDS

Not applicable except in that the DCE Shell code will follow the DCE Coding Style Guide [RFC 34].

16. OPEN ISSUES

The items in this section require some further investigation or decision on their inclusion or exclusion from DCE 1.1 requirements.

16.1. Undo

There is no undo function in the DCE Shell; nor do commands prompt the user with ''Are you sure?'' messages. If such things are desired, then users are free to write Tcl wrappers that do this, however, should this be in the first release?

16.2. Starting Servers

A useful feature of the control program would be the ability to start a server instance on a node. Currently most control programs do not support such a function. It would not be difficult to create such a command in the DCE shell, but its advantage over starting processes from a shell is not terribly clear, unless the DCE shell also has the capability to start servers on other hosts. An implementation of this function would probably require `rpcd` or `dced` to be able to spawn processes (similarly to `inetd`), which is an item currently under consideration for DCE 1.1

16.3. Viewing Seviceability Routing

In order for the `log show` command to be implemeneted there must be a new routine added to the serviceability IDL interface which will return the routing specifications in effect for a server.

APPENDIX A. Base Tcl Commands

Note to Reviewers: This appendix is the Tcl man page [TCL MAN]. It has been reformated for inclusion in this document. Any substantive differences between this version and the original man page are accidental, and the original man page should be considered the official reference.

A.1. Name

Tcl — overview of tool command language facilities

A.2. Introduction

Tcl stands for ''tool command language'' and is pronounced ''tickle''. It is actually two things: a language and a library. First, Tcl is a simple textual language, intended primarily for issuing commands to interactive programs such as text editors, debuggers, illustrators, and shells. It has a simple syntax and is also programmable, so Tcl users can write command procedures to provide more powerful commands than those in the built-in set.

Second, Tcl is a library package that can be embedded in application programs. The Tcl library consists of a parser for the Tcl language, routines to implement the Tcl built-in commands, and procedures that allow each application to extend Tcl with additional commands specific to that application. The application program generates Tcl commands and passes them to the Tcl parser for execution. Commands may be generated by reading characters from an input source, or by associating command strings with elements of the application's user interface, such as menu entries, buttons, or keystrokes. When the Tcl library receives commands it parses them into component fields and executes built-in commands directly. For commands implemented by the application, Tcl calls back to the application to execute the commands. In many cases commands will invoke recursive invocations of the Tcl interpreter by passing in additional strings to execute (procedures, looping commands, and conditional commands all work in this way).

An application program gains three advantages by using Tcl for its command language. First, Tcl provides a standard syntax: once users know Tcl, they will be able to issue commands easily to any Tcl-based application. Second, Tcl provides programmability. All a Tcl application needs to do is to implement a few application-specific low-level commands. Tcl provides many utility commands plus a general programming interface for building up complex command procedures. By using Tcl, applications need not re-implement these features. Third, Tcl can be used as a common language for communicating between applications. Inter-application communication is not built into the Tcl core described here, but various add-on libraries, such as the Tk toolkit, allow applications to issue commands to each other. This makes it possible for applications to work together in much more powerful ways than was previously possible.

This manual page focuses primarily on the Tcl language. It describes the language syntax and the built-in commands that will be available in any application based on Tcl. The individual library procedures are described in more detail in separate manual pages, one per procedure.

A.3. Interpreters

The central data structure in Tcl is an interpreter (C type `Tcl_Interp`). An interpreter consists of a set of command bindings, a set of variable values, and a few other miscellaneous pieces of state. Each Tcl command is interpreted in the context of a particular interpreter. Some Tcl-based applications will maintain multiple interpreters simultaneously, each associated with a different widget or portion of the application. Interpreters are relatively lightweight structures. They can be created and deleted quickly, so application programmers should feel free to use multiple interpreters if that simplifies the application. Eventually Tcl will provide a mechanism for sending Tcl commands and results back and forth between interpreters, even if the interpreters are managed by different processes.

A.4. Data Types

Tcl supports only one type of data: strings. All commands, all arguments to commands, all command results, and all variable values are strings. Where commands require numeric arguments or return numeric results, the arguments and results are passed as strings. Many commands expect their string arguments to have certain formats, but this interpretation is up to the individual commands. For example, arguments often contain Tcl command strings, which may get executed as part of the commands. The easiest way to understand the Tcl interpreter is to remember that everything is just an operation on a string. In many cases Tcl constructs will look similar to more structured constructs from other languages. However, the Tcl constructs are not structured at all; they are just strings of characters, and this gives them a different behavior than the structures they may look like.

Although the exact interpretation of a Tcl string depends on who is doing the interpretation, there are three common forms that strings take: commands, expressions, and lists. The major sections below discuss these three forms in more detail.

A.5. Basic Command Syntax

The Tcl language has syntactic similarities to both the Unix shells and Lisp. However, the interpretation of commands is different in Tcl than in either of those other two systems. A Tcl command string consists of one or more commands separated by newline characters or semi-colons. Each command consists of a collection of fields separated by white space (spaces or tabs). The first field must be the name of a command, and the additional fields, if any, are arguments that will be passed to that command. For example, the command

```
set a 22
```

has three fields: the first, `set`, is the name of a Tcl command, and the last two, `a` and `22`, will be passed as arguments to the `set` command. The command name may refer either to a built-in Tcl command, an application-specific command bound in with the library procedure `Tcl_CreateCommand`, or a command procedure defined with the `proc` built-in command. Arguments are passed literally as text strings. Individual commands may interpret those strings in any fashion they wish. The `set` command, for example, will treat its first argument as the name of a variable and its second argument as a string value to assign to that variable. For other commands arguments may be interpreted as integers, lists, file names, or Tcl commands.

Command names should normally be typed completely (e.g., no abbreviations). However, if the Tcl interpreter cannot locate a command it invokes a special command named `unknown` which attempts to find or create the command. For example, at many sites `unknown` will search through library directories for the desired command and create it as a Tcl procedure if it is found. The `unknown` command often provides automatic completion of abbreviated commands, but usually only for commands that were typed interactively. It's probably a bad idea to use abbreviations in command scripts and other forms that will be re-used over time: changes to the command set may cause abbreviations to become ambiguous, resulting in scripts that no longer work.

A.6. Comments

If the first non-blank character in a command is `#`, then everything from the `#` up through the next newline character is treated as a comment and ignored. When comments are embedded inside nested commands (e.g., fields enclosed in braces) they must have properly-matched braces (this is necessary because when Tcl parses the top-level command it doesn't yet know that the nested field will be used as a command so it cannot process the nested comment character as a comment).

A.7. Grouping Arguments With Double-Quotes

Normally each argument field ends at the next white space, but double-quotes may be used to create arguments with embedded space. If an argument field begins with a double-quote, then the argument isn't terminated by white space (including newlines) or a semi-colon (see below for information on semi-colons); instead it ends at the next double-quote character. The double-quotes are not included in the resulting argument. For example, the command

```
set a "This is a single argument"
```

will pass two arguments to `set`: `a` and `This is a single argument`. Within double-quotes, command substitutions, variable substitutions, and backslash substitutions still occur, as described below. If the first character of a command field is not a quote, then quotes receive no special interpretation in the parsing of that field.

A.8. Grouping Arguments With Braces

Curly braces may also be used for grouping arguments. They are similar to quotes except for two differences. First, they nest; this makes them easier to use for complicated arguments like nested Tcl command strings. Second, the substitutions described below for commands, variables, and backslashes do *not* occur in arguments enclosed in braces, so braces can be used to prevent substitutions where they are undesirable. If an argument field begins with a left brace, then the argument ends at the matching right brace. Tcl will strip off the outermost layer of braces and pass the information between the braces to the command without any further modification. For example, in the command

```
set a {xyz a {b c d}}
```

the `set` command will receive two arguments: `a` and `xyz a {b c d}`.

When braces or quotes are in effect, the matching brace or quote need not be on the same line as the starting quote or brace; in this case the newline will be included in the argument field along with any other characters up to the matching brace or quote. For example, the `eval` command takes one argument, which is a command string; `eval` invokes the Tcl interpreter to execute the command string. The command

```
eval {
    set a 22
    set b 33
}
```

will assign the value `22` to `a` and `33` to `b`.

If the first character of a command field is not a left brace, then neither left nor right braces in the field will be treated specially (except as part of variable substitution; see below).

A.9. Command Substitution With Brackets

If an open bracket occurs in a field of a command, then command substitution occurs (except for fields enclosed in braces). All of the text up to the matching close bracket is treated as a Tcl command and executed immediately. Then the result of that command is substituted for the bracketed text. For example, consider the command

```
set a [set b]
```

When the `set` command has only a single argument, it is the name of a variable and `set` returns the contents of that variable. In this case, if variable `b` has the value `foo`, then the command above is equivalent to the command

```
set a foo
```

Brackets can be used in more complex ways. For example, if the variable `b` has the value `foo` and the variable `c` has the value `gorp`, then the command

```
set a xyz[set b].[set c]
```

is equivalent to the command

```
set a xyzfoo.gorp
```

A bracketed command may contain multiple commands separated by newlines or semi-colons in the usual fashion. In this case the value of the last command is used for substitution. For example, the command

```
set a x[set b 22
expr $b+2]x
```

is equivalent to the command

```
set a x24x
```

If a field is enclosed in braces then the brackets and the characters between them are not interpreted specially; they are passed through to the argument verbatim.

A.10. Variable Substitution With $

The dollar sign (`$`) may be used as a special shorthand form for substituting variable values. If `$` appears in an argument that isn't enclosed in braces then variable substitution will occur. The characters after the `$`, up to the first character that isn't a number, letter, or underscore, are taken as a variable name and the string value of that variable is substituted for the name.

For example, if variable `foo` has the value `test`, then the command

```
set a $foo.c
```

is equivalent to the command

```
set a test.c
```

There are two special forms for variable substitution. If the next character after the name of the variable is an open parenthesis, then the variable is assumed to be an array name, and all of the characters between the open parenthesis and the next close parenthesis are taken as an index into the array. Command substitutions and variable substitutions are performed on the information between the parentheses before it is used as an index. For example, if the variable `x` is an array with one element named `first` and value `87` and another element named `14` and value `more`, then the command

```
set a xyz$x(first)zyx
```

is equivalent to the command

```
set a xyz87zyx
```

If the variable `index` has the value `14`, then the command

```
set a xyz$x($index)zyx
```

is equivalent to the command

```
set a xyzmorezyx
```

For more information on arrays, see ‘‘Variables and Arrays’’ below.

The second special form for variables occurs when the dollar sign is followed by an open curly brace. In this case the variable name consists of all the characters up to the next curly brace. Array references are not possible in this form: the name between braces is assumed to refer to a scalar variable. For example, if variable `foo` has the value `test`, then the command

```
set a abc${foo}bar
```

is equivalent to the command

```
set a abctestbar
```

Variable substitution does not occur in arguments that are enclosed in braces: the dollar sign and variable name are passed through to the argument verbatim.

The dollar sign abbreviation is simply a shorthand form. `$a` is completely equivalent to `[set a]`; it is provided as a convenience to reduce typing.

A.11. Separating Commands With Semi-Colons

Normally, each command occupies one line (the command is terminated by a newline character). However, semi-colon (`;`) is treated as a command separator character; multiple commands may be placed on one line by separating them with a semi-colon. Semi-colons are not treated as command separators if they appear within curly braces or double-quotes.

A.12. Backslash Substitution

Backslashes may be used to insert non-printing characters into command fields and also to insert special characters like braces and brackets into fields without them being interpreted specially as described above. The backslash sequences understood by the Tcl interpreter are listed below. In each case, the backslash sequence is replaced by the given character:

(a) `\b`

Backspace (0x8).

(b) `\f`

Form feed (0xc).

(c) `\n`

Newline (0xa).

(d) `\r`

Carriage-return (0xd).

(e) `\t`

Tab (0x9).

(f) `\v`

Vertical tab (0xb).

(g) `\{`

Left brace (`{`).

(h) `\}`

Right brace (`}`).

(i) `\[`

Open bracket (`[`).

(j) `\]`

Close bracket (`]`).

(k) `\$`

Dollar sign (`$`).

(l) `\`*`space`*

Space ( ): doesn't terminate argument.

(m) `\;`

Semi-colon: doesn't terminate command.

(n) `\"`

Double-quote.

(o) `\`*`newline`*

Nothing: this joins two lines together into a single line. This backslash feature is unique in that it will be applied even when the sequence occurs within braces.

(p) `\\`

Backslash (\).

(q) `\`*`ddd`*

The digits *`ddd`* (one, two, or three of them) give the octal value of the character. Null characters may not be embedded in command fields; if `ddd` is zero then the backslash sequence is ignored (i.e., it maps to an empty string).

For example, in the command

```
set a \{x\[\0yz\141
```

the second argument to `set` will be `{x[yza`.

If a backslash is followed by something other than one of the options described above, then the backslash is transmitted to the argument field without any special processing, and the Tcl scanner continues normal processing with the next character. For example, in the command

```
set \*a \\\{foo
```

The first argument to `set` will be `\*a` and the second argument will be `\{foo`.

If an argument is enclosed in braces, then backslash sequences inside the argument are parsed but no substitution occurs (except for backslash-newline): the backslash sequence is passed through to the argument as is, without making any special interpretation of the characters in the backslash sequence. In particular, backslashed braces are not counted in locating the matching right brace that terminates the argument. For example, in the command

```
set a {\{abc}
```

the second argument to `set` will be `\{abc`.

This backslash mechanism is not sufficient to generate absolutely any argument structure; it only covers the most common cases. To produce particularly complicated arguments it is probably easiest to use the `format` command along with command substitution.

A.13. Command Summary

(a) A command is just a string.

(b) Within a string commands are separated by newlines or semi-colons (unless the newline or semi-colon is within braces or brackets or is backslashed).

(c) A command consists of fields. The first field is the name of the command. The other fields are strings that are passed to that command as arguments.

(d) Fields are normally separated by white space.

(e) Double-quotes allow white space and semi-colons to appear within a single argument. Command substitution, variable substitution, and backslash substitution still occur inside quotes.

(f) Braces defer interpretation of special characters. If a field begins with a left brace, then it consists of everything between the left brace and the matching right brace. The braces themselves are not included in the argument. No further processing is done on the information between the braces except that backslash-newline sequences are eliminated.

(g) If a field doesn't begin with a brace then backslash, variable, and command substitution are done on the field. Only a single level of processing is done: the results of one substitution are not scanned again for further substitutions or any other special treatment. Substitution can occur on any field of a command, including the command name as well as the arguments.

(h) If the first non-blank character of a command is a `#`, everything from the `#` up through the next newline is treated as a comment and ignored.

A.14. Expressions

The second major interpretation applied to strings in Tcl is as expressions. Several commands, such as `expr`, `for`, and `if`, treat one or more of their arguments as expressions and call the Tcl expression processors (`Tcl_ExprLong`, `Tcl_ExprBoolean`, etc.) to evaluate them. The operators permitted in Tcl expressions are a subset of the operators permitted in C expressions, and they have the same meaning and precedence as the corresponding C operators. Expressions almost always yield numeric results (integer or floating-point values). For example, the expression

```
8.2 + 6
```

evaluates to `14.2`. Tcl expressions differ from C expressions in the way that operands are specified, and in that Tcl expressions support non-numeric operands and string comparisons.

A Tcl expression consists of a combination of operands, operators, and parentheses. White space may be used between the operands and operators and parentheses; it is ignored by the expression processor. Where possible, operands are interpreted as integer values. Integer values may be specified in decimal (the normal case), in octal (if the first character of the operand is `0`), or in hexadecimal (if the first two characters of the operand are `0x`). If an operand does not have one of the integer formats given above, then it is treated as a floating-point number if that is possible. Floating-point numbers may be specified in any of the ways accepted by an ANSI-compliant C compiler (except that the `f`, `F`, `l`, and `L` suffixes will not be permitted in most installations). For example, all of the following are valid floating-point numbers: `2.1`, `3.`, `6e4`, `7.91e+16`. If no numeric interpretation is possible, then an operand is left as a string (and only a limited set of operators may be applied to it).

Operands may be specified in any of the following ways:

(a) As an numeric value, either integer or floating-point.

(b) As a Tcl variable, using standard `$` notation. The variable's value will be used as the operand.

(c) As a string enclosed in double-quotes. The expression parser will perform backslash, variable, and command substitutions on the information between the quotes, and use the resulting value as the operand

(d) As a string enclosed in braces. The characters between the open brace and matching close brace will be used as the operand without any substitutions.

(e) As a Tcl command enclosed in brackets. The command will be executed and its result will be used as the operand.

Where substitutions occur above (e.g., inside quoted strings), they are performed by the expression processor. However, an additional layer of substitution may already have been performed by the command parser before the expression processor was called. As discussed below, it is usually best to enclose expressions in braces to prevent the command parser from performing substitutions on the contents.

For some examples of simple expressions, suppose the variable `a` has the value `3` and the variable `b` has the value `6`. Then the expression on the left side of each of the lines below will evaluate to the value on the right side of the line:

```
3.1 + $a                   6.1

2 + "$a.$b"                5.6

4*[llength "6 2"]          8

{word one} < "word $a"     0
```

The valid operators are listed below, grouped in decreasing order of precedence:

(a) `-  ~  !`

Unary minus, bit-wise NOT, logical NOT. None of these operands may be applied to string operands, and bit-wise NOT may be applied only to integers.

(b) `*  /  %`

Multiply, divide, remainder. None of these operands may be applied to string operands, and remainder may be applied only to integers.

(c) `+  -`

Add and subtract. Valid for any numeric operands.

(d) `<<  >>`

Left and right shift. Valid for integer operands only.

(e) `<  >  <=  >=`

Boolean less, greater, less than or equal, and greater than or equal. Each operator produces `1` if the condition is true, `0` otherwise. These operators may be applied to strings as well as numeric operands, in which case string comparison is used.

(f) `==  !=`

Boolean equal and not equal. Each operator produces a zero/one result. Valid for all operand types.

(g) `&`

Bit-wise AND. Valid for integer operands only.

(h) `^`

Bit-wise exclusive OR. Valid for integer operands only.

(i) `|`

Bit-wise OR. Valid for integer operands only.

(j) `&&`

Logical AND. Produces a `1` result if both operands are non-zero, `0` otherwise. Valid for numeric operands only (integers or floating-point).

(k) `||`

Logical OR. Produces a `0` result if both operands are zero, `1` otherwise. Valid for numeric operands only (integers or floating-point).

(l) *x*`?`*y*`:`*z*

If-then-else, as in C. If *x* evaluates to non-zero, then the result is the value of *y* Otherwise the result is the value of *z*. The *x* operand must have a numeric value.

See the C manual for more details on the results produced by each operator. All of the binary operators group left-to-right within the same precedence level. For example, the expression

```
4*2 < 7
```

evaluates to `0`.

The `&&`, `||`, and `?:` operators have ‘‘lazy evaluation’’, just as in C, which means that operands are not evaluated if they are not needed to determine the outcome. For example, in

```
$v ? [a] : [b]
```

only one of `[a]` or `[b]` will actually be evaluated, depending on the value of `$v`.

All internal computations involving integers are done with the C type `long`, and all internal computations involving floating-point are done with the C type `double`. When converting a string to floating-point, exponent overflow is detected and results in a Tcl error. For conversion to integer from string, detection of overflow depends on the behavior of some routines in the local C library, so it should be regarded as unreliable. In any case, overflow and underflow are generally not detected reliably for intermediate results.

Conversion among internal representations for integer, floating-point, and string operands is done automatically as needed. For arithmetic computations, integers are used until some floating-point number is introduced, after which floating-point is used. For example,

```
5 / 4
```

yields the result `1`, while

```
5 / 4.0
```

and

```
5 / ( [string length "abcd"] + 0.0 )
```

both yield the result `1.25`.

String values may be used as operands of the comparison operators, although the expression evaluator tries to do comparisons as integer or floating-point when it can. If one of the operands of a comparison is a string and the other has a numeric value, the numeric operand is converted back to a string using the C `sprintf` format specifier `%d` for integers and `%g` for floating-point values. For example, the expressions

```
"0x03" > "2"
```

and

```
"0y" < "0x12"
```

both evaluate to `1`. The first comparison is done using integer comparison, and the second is done using string comparison after the second operand is converted to the string `18`.

In general it is safest to enclose an expression in braces when entering it in a command: otherwise, if the expression contains any white space then the Tcl interpreter will split it among several arguments. For example, the command

```
expr $a + $b
```

results in three arguments being passed to `expr`: `$a`, `+`, and `$b`. In addition, if the expression isn't in braces then the Tcl interpreter will perform variable and command substitution immediately (it will happen in the command parser rather than in the expression parser). In many cases the expression is being passed to a command that will evaluate the expression later (or even many times if, for example, the expression is to be used to decide when to exit a loop). Usually the desired goal is to re-do the variable or command substitutions each time the expression is evaluated, rather than once and for all at the beginning. For example, the command

```
for {set i 1} $i<=10 {incr i} {...}         *** WRONG ***
```

is probably intended to iterate over all values of `i` from 1 to 10. After each iteration of the body of the loop, `for` will pass its second argument to the expression evaluator to see whether or not to continue processing. Unfortunately, in this case the value of `i` in the second argument will be substituted once and for all when the `for` command is parsed. If `i` was 0 before the `for` command was invoked then `for`'s second argument will be `0<=10` which will always evaluate to 1, even though `i`'s value eventually becomes greater than 10. In the above case the loop will never terminate. Instead, the expression should be placed in braces:

```
for {set i 1} {$i<=10} {incr i} {...}       *** RIGHT ***
```

This causes the substitution of `i`'s value to be delayed; it will be re-done each time the expression is evaluated, which is the desired result.

A.15. Lists

The third major way that strings are interpreted in Tcl is as lists. A list is just a string with a list-like structure consisting of fields separated by white space. For example, the string

```
Al Sue Anne John
```

is a list with four elements or fields. Lists have the same basic structure as command strings, except that a newline character in a list is treated as a field separator just like space or tab. Conventions for braces and quotes and backslashes are the same for lists as for commands. For example, the string

```
a b\ c {d e {f g h}}
```

is a list with three elements: `a`, `b c`, and `d e {f g h}`. Whenever an element is extracted from a list, the same rules about braces and quotes and backslashes are applied as for commands. Thus in the example above when the third element is extracted from the list, the result is

```
d e {f g h}
```

(when the field was extracted, all that happened was to strip off the outermost layer of braces). Command substitution and variable substitution are never made on a list (at least, not by the list-processing commands; the list can always be passed to the Tcl interpreter for evaluation).

The Tcl commands `concat`, `foreach`, `lappend`, `lindex`, `linsert`, `list`, `llength`, `lrange`, `lreplace`, `lsearch`, and `lsort` allow you to build lists, extract elements from them, search them, and perform other list-related functions.

A.16. Regular Expressions

Tcl provides two commands that support string matching using `egrep`-style regular expressions: `regexp` and `regsub`. Regular expressions are implemented using Henry Spencer's package, and the description of regular expressions below is copied verbatim from his manual entry.

A regular expression is zero or more *branches*, separated by `|`. It matches anything that matches one of the branches.

A branch is zero or more *pieces*, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an *atom* possibly followed by `*`, `+`, or `?`. An atom followed by `*` matches a sequence of 0 or more matches of the atom. An atom followed by `+` matches a sequence of 1 or more matches of the atom. An atom followed by `?` matches a match of the atom, or the null string.

An atom is a regular expression in parentheses (matching a match for the regular expression), a *range* (see below), `.` (matching any single character), `^` (matching the null string at the beginning of the input string), `$` (matching the null string at the end of the input string), a `\` followed by a single character (matching that character), or a single character with no other significance (matching that character).

A *range* is a sequence of characters enclosed in `[]`. It normally matches any single character from the sequence. If the sequence begins with `^`, it matches any single character *not* from the rest of the sequence. If two characters in the sequence are separated by `-`, this is shorthand for the full list of ASCII characters between them (e.g., `[0-9]` matches any decimal digit). To include a literal `]` in the sequence, make it the first character (following a possible `^`). To include a literal `-`, make it the first or last character.

If a regular expression could match two different parts of a string, it will match the one which begins earliest. If both begin in the same place but match different lengths, or match the same length in different ways, life gets messier, as follows.

In general, the possibilities in a list of branches are considered in left-to-right order, the possibilities for `*`, `+`, and `?` are considered longest-first, nested constructs are considered from the outermost in, and concatenated constructs are considered leftmost-first. The match that will be chosen is the one that uses the earliest possibility in the first choice that has to be made. If there is more than one choice, the next will be made in the same manner (earliest possibility) subject to the decision on the first choice. And so forth.

For example, `(ab|a)b*c` could match `abc` in one of two ways. The first choice is between `ab` and `a`; since `ab` is earlier, and does lead to a successful overall match, it is chosen. Since the `b` is already spoken for, the `b*` must match its last possibility — the empty string — since it must respect the earlier choice.

In the particular case where no `|`'s are present and there is only one `*`, `+`, or `?`, the net effect is that the longest possible match will be chosen. So `ab*`, presented with `xabbbby`, will match `abbbb`. Note that if `ab*` is tried against `xabyabbbz`, it will match `ab` just after `x`, due to the begins-earliest rule. (In effect, the decision on where to start the match is the first choice to be made, hence subsequent choices must respect it even if this leads them to less-preferred alternatives.)

A.17. Command Results

Each command produces two results: a code and a string. The code indicates whether the command completed successfully or not, and the string gives additional information. The valid codes are defined in `tcl.h`, and are:

(a) `TCL_OK`

This is the normal return code, and indicates that the command completed successfully. The string gives the command's return value.

(b) `TCL_ERROR`

Indicates that an error occurred; the string gives a message describing the error.

In addition, the global variable `errorInfo` will contain human-readable information describing which commands and procedures were being executed when the error occurred, and the global variable `errorCode` will contain machine-readable details about the error, if they are available. See the section ''Built-In Variables'' below for more information.

(c) `TCL_RETURN`

Indicates that the `return` command has been invoked, and that the current procedure (or top-level command or `source` command) should return immediately. The string gives the return value for the procedure or command.

(d) `TCL_BREAK`

Indicates that the `break` command has been invoked, so the innermost loop should abort immediately. The string should always be empty.

(e) `TCL_CONTINUE`

Indicates that the `continue` command has been invoked, so the innermost loop should go on to the next iteration. The string should always be empty.

Tcl programmers do not normally need to think about return codes, since `TCL_OK` is almost always returned. If anything else is returned by a command, then the Tcl interpreter immediately stops processing commands and returns to its caller. If there are several nested invocations of the Tcl interpreter in progress, then each nested command will usually return the error to its caller, until eventually the error is reported to the top-level application code. The application will then display the error message for the user.

In a few cases, some commands will handle certain ''error'' conditions themselves and not return them upwards. For example, the `for` command checks for the `TCL_BREAK` code; if it occurs, then `for` stops executing the body of the loop and returns `TCL_OK` to its caller. The `for` command also handles `TCL_CONTINUE` codes and the procedure interpreter handles `TCL_RETURN` codes. The `catch` command allows Tcl programs to catch errors and handle them without aborting command interpretation any further.

A.18. Procedures

Tcl allows you to extend the command interface by defining procedures. A Tcl procedure can be invoked just like any other Tcl command (it has a name and it receives one or more arguments). The only difference is that its body isn't a piece of C code linked into the program; it is a string containing one or more other Tcl commands. See the `proc` command for information on how to define procedures and what happens when they are invoked.

A.19. Variables — Scalars and Arrays

Tcl allows the definition of variables and the use of their values either through `$`-style variable substitution, the `set` command, or a few other mechanisms. Variables need not be declared: a new variable will automatically be created each time a new variable name is used.

Tcl supports two types of variables: scalars and arrays. A scalar variable has a single value, whereas an array variable can have any number of elements, each with a name (called its ''index'') and a value. Array indexes may be arbitrary strings; they need not be numeric. Parentheses are used refer to array elements in Tcl commands. For example, the command

```
set x(first) 44
```

will modify the element of `x` whose index is `first` so that its new value is `44`. Two-dimensional arrays can be simulated in Tcl by using indexes that contain multiple concatenated values. For example, the commands

```
set a(2,3) 1
```

and

```
set a(3,6) 2
```

set the elements of `a` whose indexes are `2,3` and `3,6`.

In general, array elements may be used anywhere in Tcl that scalar variables may be used. If an array is defined with a particular name, then there may not be a scalar variable with the same name. Similarly, if there is a scalar variable with a particular name then it is not possible to make array references to the variable. To convert a scalar variable to an array or vice versa, remove the existing variable with the `unset` command.

The `array` command provides several features for dealing with arrays, such as querying the names of all the elements of the array and searching through the array one element at a time.

Variables may be either global or local. If a variable name is used when a procedure isn't being executed, then it automatically refers to a global variable. Variable names used within a procedure normally refer to local variables associated with that invocation of the procedure. Local variables are deleted whenever a procedure exits. The `global` command may be used to request that a name refer to a global variable for the duration of the current procedure (this is somewhat analogous to `extern` in C).

A.20. Built-In Commands

The Tcl library provides the following built-in commands, which will be available in any application using Tcl. In addition to these built-in commands, there may be additional commands defined by each application, plus commands defined as Tcl procedures. In the command syntax descriptions below, words in a fixed-width font are literals that you type verbatim to Tcl. Words in italics are meta-symbols; they serve as names for any of a range of values that you can type. Optional arguments or groups of arguments are indicated by enclosing them in question-marks. Ellipses (`...`) indicate that any number of additional arguments or groups of arguments may appear, in the same format as the preceding argument(s).

(a) `append` *`varName value ?value value ...?`*

Append all of the *`value`* arguments to the current value of variable *`varName`*. If *`varName`* doesn't exist, it is given a value equal to the concatenation of all the *`value`* arguments. This command provides an efficient way to build up long variables incrementally. For example, `append a $b` is much more efficient than `set a $a$b` if `$a` is long.

(b) `array` *`option arrayName ?arg arg ...?`*

This command performs one of several operations on the variable given by *`arrayName`*. *`ArrayName`* must be the name of an existing array variable. The *`option`* argument determines what action is carried out by the command. The legal *`options`* (which may be abbreviated) are:

(i) `array anymore` *`arrayName searchId`*

Returns 1 if there are any more elements left to be processed in an array search, 0 if all elements have already been returned. *`SearchId`* indicates which search on *`arrayName`* to check, and must have been the return value from a previous invocation of `array startsearch`. This option is particularly useful if an array has an element with an empty name, since the return value from `array nextelement` won't indicate whether the search has been completed.

(ii) `array donesearch` *`arrayName searchId`*

This command terminates an array search and destroys all the state associated with that search. *`SearchId`* indicates which search on *`arrayName`* to destroy, and must have been the return value from a previous invocation of `array startsearch`. Returns an empty string.

(iii) `array names` *`arrayName`*

Returns a list containing the names of all of the elements in the array. If there are no elements in the array then an empty string is returned.

(iv) `array nextelement` *`arrayName searchId`*

Returns the name of the next element in *`arrayName`*, or an empty string if all elements of *`arrayName`* have already been returned in this search. The *`searchId`* argument identifies the search, and must have been the return value of an `array startsearch` command. Warning: if elements are added to or deleted from the array, then all searches are automatically terminated just as if `array donesearch` had been invoked; this will cause `array nextelement` operations to fail for those searches.

(v) `array size` *`arrayName`*

Returns a decimal string giving the number of elements in the array.

(vi) `array startsearch` *`arrayName`*

This command initializes an element-by-element search through the array given by *`arrayName`*, such that invocations of the `array nextelement` command will return the names of the individual elements in the array. When the search has been completed, the `array donesearch` command should be invoked. The return value is a search identifier that must be used in `array nextelement` and `array donesearch` commands; it allows multiple searches to be underway simultaneously for the same array.

(c) `break`

This command may be invoked only inside the body of a loop command such as `for` or `foreach` or `while`. It returns a TCL_BREAK code to signal the innermost containing loop command to return immediately.

(d) `case` *string ?*`in`*? patList body ?patList body...?*

(e) `case` *string ?*`in`*? {patList body ?patList body...?}*

Match *string* against each of the *patList* arguments in order. If one matches, then evaluate the following *body* argument by passing it recursively to the Tcl interpreter, and return the result of that evaluation. Each *patList* argument consists of a single pattern or list of patterns. Each pattern may contain any of the wild-cards described under `string match`. If a *patList* argument is `default`, the corresponding body will be evaluated if no *patList* matches *string*. If no *patList* argument matches *string* and no default is given, then the `case` command returns an empty string.

Two syntaxes are provided. The first uses a separate argument for each of the patterns and commands; this form is convenient if substitutions are desired on some of the patterns or commands.

The second form places all of the patterns and commands together into a single argument; the argument must have proper list structure, with the elements of the list being the patterns and commands. The second form makes it easy to construct multi-line case commands, since the braces around the whole list make it unnecessary to include a backslash at the end of each line. Since the *patList* arguments are in braces in the second form, no command or variable substitutions are performed on them; this makes the behavior of the second form different than the first form in some cases.

Below are some examples of `case` commands:

(i) The following will return `3`:

```
case abc in {a b} {format 1} default {format 2} a* \
{format 3}
```

(ii) The following will return `1`:

```
case a in {
    {a b} {format 1}
    default {format 2}
    a* {format 3}
}
```

(iii) The following will return `2`:

```
case xyz {
    {a b}
        {format 1}
    default
        {format 2}
    a*
        {format 3}
}
```

(f) `catch` *command ?varName?*

The `catch` command may be used to prevent errors from aborting command interpretation. `Catch` calls the Tcl interpreter recursively to execute *command*, and always returns a TCL_OK code, regardless of any errors that might occur while executing *command*. The return value from `catch` is a decimal string giving the code returned by the Tcl interpreter after executing *command*. This will be `0` (TCL_OK) if there were no errors in *command*; otherwise it will have a non-zero value corresponding to one of the exceptional return codes (see tcl.h for the definitions of code values). If the *varName* argument is given, then it gives the name of a variable; `catch` will set the value of the variable to the string returned from *command* (either a result or an error message).

(g) `cd` *?dirName?*

Change the current working directory to *dirName*, or to the home directory (as specified in the HOME environment variable) if *dirName* is not given. If *dirName* starts with a tilde, then tilde-expansion is done as described for `Tcl_TildeSubst`. Returns an empty string. This command can potentially be disruptive to an application, so it may be removed in some applications.

(h) `close` *`fileId`*

Closes the file given by *`fileId`*. *`FileId`* must be the return value from a previous invocation of the `open` command; after this command, it should not be used anymore. If *`fileId`* refers to a command pipeline instead of a file, then `close` waits for the children to complete. The normal result of this command is an empty string, but errors are returned if there are problems in closing the file or waiting for children to complete.

(i) `concat` *`arg ?arg ...?`*

This command treats each argument as a list and concatenates them into a single list. It permits any number of arguments. For example, the command

```
concat a b {c d e} {f {g h}}
```

will return

```
a b c d e f {g h}
```

as its result.

(j) `continue`

This command may be invoked only inside the body of a loop command such as `for` or `foreach` or `while`. It returns a TCL_CONTINUE code to signal the innermost containing loop command to skip the remainder of the loop's body but continue with the next iteration of the loop.

(k) `eof` *`fileId`*

Returns 1 if an end-of-file condition has occurred on *`fileId`*, 0 otherwise. *`FileId`* must have been the return value from a previous call to `open`, or it may be `stdin`, `stdout`, or `stderr` to refer to one of the standard I/O channels.

(l) `error` *`message ?info? ?code?`*

Returns a TCL_ERROR code, which causes command interpretation to be unwound. *`Message`* is a string that is returned to the application to indicate what went wrong.

If the *`info`* argument is provided and is non-empty, it is used to initialize the global variable `errorInfo`. `errorInfo` is used to accumulate a stack trace of what was in progress when an error occurred; as nested commands unwind, the Tcl interpreter adds information to `errorInfo`. If the *`info`* argument is present, it is used to initialize `errorInfo` and the first increment of unwind information will not be added by the Tcl interpreter. In other words, the command containing the `error` command will not appear in `errorInfo`; in its place will be *`info`*. This feature is most useful in conjunction with the `catch` command: if a caught error cannot be handled successfully, *`info`* can be used to return a stack trace reflecting the original point of occurrence of the error:

```
catch {...} errMsg
set savedInfo $errorInfo
...
error $errMsg $savedInfo
```

If the *`code`* argument is present, then its value is stored in the `errorCode` global variable. This variable is intended to hold a machine-readable description of the error in cases where such information is available; see the section ''Built-In Variables'' below for information on the proper format for the variable. If the *`code`* argument is not present, then `errorCode` is automatically reset to `NONE` by the Tcl interpreter as part of processing the error generated by the command.

(m) `eval` *`arg ?arg ...?`*

`Eval` takes one or more arguments, which together comprise a Tcl command (or collection of Tcl commands separated by newlines in the usual way). `Eval` concatenates all its arguments in the same fashion as the `concat` command, passes the concatenated string to the Tcl interpreter recursively, and returns the result of that evaluation (or any error generated by it).

(n) `exec` *`arg ?arg ...?`*

This command treats its arguments as the specification of one or more UNIX commands to execute as subprocesses. The commands take the form of a standard shell pipeline; | arguments separate commands in the pipeline and cause standard output of the preceding command to be piped into standard input of the next command.

Under normal conditions the result of the `exec` command consists of the standard output produced by the last command in the pipeline. If any of the commands in the pipeline exit abnormally or are killed or suspended, then `exec` will return an error and the error message will include the pipeline's output followed by error messages describing the abnormal terminations; the `errorCode` variable will contain additional information about the last abnormal termination encountered. If any of the commands writes to its standard error file, then `exec` will return an error, and the error message will include the pipeline's output, followed by messages about abnormal terminations (if any), followed by the standard error output.

If the last character of the result or error message is a newline then that character is deleted from the result or error message for consistency with normal Tcl return values.

If an *`arg`* has the value > then the following argument is taken as the name of a file and the standard output of the last command in the pipeline is redirected to the file. In this situation `exec` will normally return an empty string.

If an *`arg`* has the value < then the following argument is taken as the name of a file to use for standard input to the first command in the pipeline. If an argument has the value << then the following argument is taken as an immediate value to be passed to the first command as standard input. If there is no < or << argument then the standard input for the first command in the pipeline is taken from the application's current standard input.

If the last *`arg`* is `&` then the command will be executed in background. In this case the standard output from the last command in the pipeline will go to the application's standard output unless redirected in the command, and error output from all the commands in the pipeline will go to the application's standard error file.

Each *`arg`* becomes one word for a command, except for |, <, <<, >, and `&` arguments, and the arguments that follow <, <<, and >. The first word in each command is taken as the command name; tilde-substitution is performed on it, and the directories in the PATH environment variable are searched for an executable by the given name. No ''glob'' expansion or other shell-like substitutions are performed on the arguments to commands.

(o) `exit` *`?returnCode?`*

Terminate the process, returning *`returnCode`* to the parent as the exit status. If *`returnCode`* isn't specified then it defaults to 0.

(p) `expr` *`arg`*

Calls the expression processor to evaluate *`arg`*, and returns the result as a string. See the section EXPRESSIONS above.

(q) `file` *`option name ?arg arg ...?`*

Operate on a file or a file name. *`Name`* is the name of a file; if it starts with a tilde, then tilde substitution is done before executing the command (see the manual entry for `Tcl_TildeSubst` for details). *`Option`* indicates what to do with the file name. Any unique abbreviation for *`option`* is acceptable. The valid options are:

(i) `file atime` *`name`*

Return a decimal string giving the time at which file *`name`* was last accessed. The time is measured in the standard UNIX fashion as seconds from a fixed starting time (often January 1, 1970). If the file doesn't exist or its access time cannot be queried then an error is generated.

(ii) `file dirname` *`name`*

Return all of the characters in *`name`* up to but not including the last slash character. If there are no slashes in *`name`* then return `.`. If the last slash in *`name`* is its first character, then return `/`.

(iii) `file executable` *`name`*

Return `1` if file *`name`* is executable by the current user, `0` otherwise.

(iv) `file exists` *`name`*

Return `1` if file *`name`* exists and the current user has search privileges for the directories leading to it, `0` otherwise.

(v) `file extension` *`name`*

Return all of the characters in *`name`* after and including the last dot in *`name`*. If there is no dot in *`name`* then return the empty string.

(vi) `file isdirectory` *`name`*

Return `1` if file *`name`* is a directory, `0` otherwise.

(vii) `file isfile` *`name`*

Return `1` if file *`name`* is a regular file, `0` otherwise.

(viii) `file lstat` *`name varName`*

Same as `stat` option (see below) except uses the *`lstat`* kernel call instead of *`stat`*. This means that if *`name`* refers to a symbolic link the information returned in *`varName`* is for the link rather than the file it refers to. On systems that don't support symbolic links this option behaves exactly the same as the `stat` option.

(ix) `file mtime` *`name`*

Return a decimal string giving the time at which file *`name`* was last modified. The time is measured in the standard UNIX fashion as seconds from a fixed starting time (often January 1, 1970). If the file doesn't exist or its modified time cannot be queried then an error is generated.

(x) `file owned` *`name`*

Return `1` if file *`name`* is owned by the current user, `0` otherwise.

(xi) `file readable` *`name`*

Return `1` if file *`name`* is readable by the current user, `0` otherwise.

(xii) `file readlink` *`name`*

Returns the value of the symbolic link given by *`name`* (i.e., the name of the file it points to). If *`name`* isn't a symbolic link or its value cannot be read, then an error is returned. On systems that don't support symbolic links this option is undefined.

(xiii) `file rootname` *`name`*

Return all of the characters in *`name`* up to but not including the last **.** character in the name. If *`name`* doesn't contain a dot, then return *`name`*.

(xiv) `file size` *`name`*

Return a decimal string giving the size of file *name* in bytes. If the file doesn't exist or its size cannot be queried then an error is generated.

(xv) `file stat` *name varName*

Invoke the `stat` kernel call on *name*, and use the variable given by *varName* to hold information returned from the kernel call. *VarName* is treated as an array variable, and the following elements of that variable are set: `atime`, `ctime`, `dev`, `gid`, `ino`, `mode`, `mtime`, `nlink`, `size`, `type`, `uid`. Each element except `type` is a decimal string with the value of the corresponding field from the `stat` return structure; see the manual entry for `stat` for details on the meanings of the values. The `type` element gives the type of the file in the same form returned by the command `file type`. This command returns an empty string.

(xvi) `file tail` *name*

Return all of the characters in *name* after the last slash. If *name* contains no slashes then return *name*.

(xvii) `file type` *name*

Returns a string giving the type of file *name*, which will be one of `file`, `directory`, `characterSpecial`, `blockSpecial`, `fifo`, `link`, or `socket`.

(xviii) `file writable` *name*

Return `1` if file *name* is writable by the current user, `0` otherwise.

The `file` commands that return 0/1 results are often used in conditional or looping commands, for example:

```
if {![file exists foo]} then {error {bad file name}} \
else {...}
```

(r) `flush` *fileId*

Flushes any output that has been buffered for *fileId*. *FileId* must have been the return value from a previous call to `open`, or it may be `stdout` or `stderr` to access one of the standard I/O streams; it must refer to a file that was opened for writing. This command returns an empty string.

(s) `for` *start test next body*

`For` is a looping command, similar in structure to the C `for` statement. The *start*, *next*, and *body* arguments must be Tcl command strings, and *test* is an expression string. The `for` command first invokes the Tcl interpreter to execute *start*. Then it repeatedly evaluates *test* as an expression; if the result is non-zero it invokes the Tcl interpreter on *body*, then invokes the Tcl interpreter on *next*, then repeats the loop. The command terminates when *test* evaluates to 0. If a `continue` command is invoked within *body* then any remaining commands in the current execution of *body* are skipped; processing continues by invoking the Tcl interpreter on *next*, then evaluating *test*, and so on. If a `break` command is invoked within *body* or *next*, then the `for` command will return immediately. The operation of `break` and `continue` are similar to the corresponding statements in C. `For` returns an empty string.

(t) `foreach` *varname list body*

In this command, *varname* is the name of a variable, *list* is a list of values to assign to *varname*, and *body* is a collection of Tcl commands. For each field in *list* (in order from left to right), `foreach` assigns the contents of the field to *varname* (as if the `lindex` command had been used to extract the field), then calls the Tcl interpreter to execute *body*. The `break` and `continue` statements may be invoked inside *body*, with the same effect as in the `for` command. `Foreach` returns an empty string.

(u) `format` *formatString ?arg arg ...?*

This command generates a formatted string in the same way as the C `sprintf` procedure (it uses `sprintf` in its implementation). *FormatString* indicates how to format the result, using `%` fields as in `sprintf`, and the additional arguments, if any, provide values to be substituted into the result. All of the `sprintf` options are valid; see the `sprintf` man page for details. Each *arg* must match the expected type from the `%` field in *formatString*; the `format` command converts each argument to the correct type (floating, integer, etc.) before passing it to `sprintf` for formatting. The only unusual conversion is for `%c`; in this case the argument must be a decimal string, which will then be converted to the corresponding ASCII character value. `Format` does backslash substitution on its *formatString* argument, so backslash sequences in *formatString* will be handled correctly even if the argument is in braces. The return value from `format` is the formatted string.

(v) `gets` *fileId ?varName?*

Reads the next line from the file given by *fileId* and discards the terminating newline character. If *varName* is specified, then the line is placed in the variable by that name and the return value is a count of the number of characters read (not including the newline). If the end of the file is reached before reading any characters then -1 is returned and *varName* is set to an empty string. If *varName* is not specified then the return value will be the line (minus the newline character) or an empty string if the end of the file is reached before reading any characters. An empty string will also be returned if a line contains no characters except the newline, so `eof` may have to be used to determine what really happened. If the last character in the file is not a newline character, then `gets` behaves as if there were an additional newline character at the end of the file. *FileId* must be `stdin` or the return value from a previous call to `open`; it must refer to a file that was opened for reading.

(w) `glob` *?*`−nocomplain`*? filename ?filename ...?*

This command performs filename globbing, using csh rules. The returned value from `glob` is the list of expanded filenames.

If `−nocomplain` is specified as the first argument then an empty list may be returned; otherwise an error is returned if the expanded list is empty. The `−nocomplain` argument must be provided exactly: an abbreviation will not be accepted.

(x) `global` *varname ?varname ...?*

This command is ignored unless a Tcl procedure is being interpreted. If so, then it declares the given *varname*'s to be global variables rather than local ones. For the duration of the current procedure (and only while executing in the current procedure), any reference to any of the *varname*s will be bound to a global variable instead of a local one.

(y) `history` *?option? ?arg arg ...?*

Note: this command may not be available in all Tcl-based applications. Typically, only those that receive command input in a typescript form will support history. The `history` command performs one of several operations related to recently-executed commands recorded in a history list. Each of these recorded commands is referred to as an ''event''. When specifying an event to the `history` command, the following forms may be used:

(i) A number: if positive, it refers to the event with that number (all events are numbered starting at 1). If the number is negative, it selects an event relative to the current event (`−1` refers to the previous event, `−2` to the one before that, and so on).

(ii) A string: selects the most recent event that matches the string. An event is considered to match the string either if the string is the same as the first characters of the event, or if the string matches the event in the sense of the `string match` command.

The `history` command can take any of the following forms:

(i) `history`

Same as `history info`, described below.

(ii) `history add` *command ?*`exec`*?*

Add the *command* argument to the history list as a new event. If `exec` is specified (or abbreviated) then the command is also executed and its result is returned. If `exec` isn't specified then an empty string is returned as result.

(iii) `history change` *`newValue ?event?`*

Replace the value recorded for an event with *newValue*. *Event* specifies the event to replace, and defaults to the *current* event (not event `-1`). This command is intended for use in commands that implement new forms of history substitution and wish to replace the current event (which invokes the substitution) with the command created through substitution. The return value is an empty string.

(iv) `history event` *`?event?`*

Returns the value of the event given by *event*. *Event* defaults to `-1`. This command causes history revision to occur: see below for details.

(v) `history info` *`?count?`*

Returns a formatted string (intended for humans to read) giving the event number and contents for each of the events in the history list except the current event. If *count* is specified then only the most recent *count* events are returned.

(vi) `history keep` *`count`*

This command may be used to change the size of the history list to *count* events. Initially, 20 events are retained in the history list. This command returns an empty string.

(vii) `history nextid`

Returns the number of the next event to be recorded in the history list. It is useful for things like printing the event number in command-line prompts.

(viii) `history redo` *`?event?`*

Re-execute the command indicated by *event* and return its result. *Event* defaults to `-1`. This command results in history revision: see below for details.

(ix) `history substitute` *`old new ?event?`*

Retrieve the command given by *event* (`-1` by default), replace any occurrences of *old* by *new* in the command (only simple character equality is supported; no wild cards), execute the resulting command, and return the result of that execution. This command results in history revision: see below for details.

(x) `history words` *`selector ?event?`*

Retrieve from the command given by *event* (`-1` by default) the words given by *selector*, and return those words in a string separated by spaces. The `selector` argument has three forms. If it is a single number then it selects the word given by that number (`0` for the command name, `1` for its first argument, and so on). If it consists of two numbers separated by a dash, then it selects all the arguments between those two. Otherwise `selector` is treated as a pattern; all words matching that pattern (in the sense of `string match`) are returned. In the numeric forms `$` may be used to select the last word of a command. For example, suppose the most recent command in the history list is

```
format  {%s is %d years old} Alice [expr \
$ageInMonths/12]
```

Then below are some history commands (left column) and the results they would produce (right column):

```
history words $         [expr $ageInMonths/12]

history words 1-2       {%s is %d years  old} Alice

history words *a*o*     {%s is %d years old} [expr \
$ageInMonths/12]
```

`History words` results in history revision: see below for details.

The history options `event`, `redo`, `substitute`, and `words` result in ‘‘history revision’’. When one of these options is invoked then the current event is modified to eliminate the history command and replace it with the result of the history command. For example, suppose that the most recent command in the history list is

```
set a [expr $b+2]
```

and suppose that the next command invoked is one of the ones on the left side of the table below. The command actually recorded in the history event will be the corresponding one on the right side of the table.

```
history redo            set a [expr $b+2]

history s a b           set b [expr $b+2]

set c [history w 2]     set c [expr $b+2]
```

History revision is needed because event specifiers like `-1` are only valid at a particular time: once more events have been added to the history list a different event specifier would be needed. History revision occurs even when `history` is invoked indirectly from the current event (e.g., a user types a command that invokes a Tcl procedure that invokes `history`): the top-level command whose execution eventually resulted in a `history` command is replaced. If you wish to invoke commands like `history words` without history revision, you can use `history event` to save the current history event and then use `history change` to restore it later.

(z) `if` *`test`* ?`then`? *`trueBody`* ?`else`? *`?falseBody?`*

The *`if`* command evaluates *`test`* as an expression (in the same way that `expr` evaluates its argument). The value of the expression must be numeric; if it is non-zero then *`trueBody`* is called by passing it to the Tcl interpreter. Otherwise *`falseBody`* is executed by passing it to the Tcl interpreter. The `then` and `else` arguments are optional ‘‘noise words’’ to make the command easier to read. *`FalseBody`* is also optional; if it isn’t specified then the command does nothing if *`test`* evaluates to zero. The return value from `if` is the value of the last command executed in *`trueBody`* or *`falseBody`*, or the empty string if *`test`* evaluates to zero and *`falseBody`* isn’t specified.

(aa) `incr` *`varName ?increment?`*

Increment the value stored in the variable whose name is *`varName`*. The value of the variable must be integral. If *`increment`* is supplied then its value (which must be an integer) is added to the value of variable *`varName`*; otherwise 1 is added to *`varName`*. The new value is stored as a decimal string in variable *`varName`* and also returned as result.

(ab) `info` *`option ?arg arg ...?`*

Provide information about various internals to the Tcl interpreter. The legal *`option`*’s (which may be abbreviated) are:

(i) `info args` *`procname`*

Returns a list containing the names of the arguments to procedure *`procname`*, in order. *`Procname`* must be the name of a Tcl command procedure.

(ii) `info body` *`procname`*

Returns the body of procedure *procname*. *Procname* must be the name of a Tcl command procedure.

(iii) `info cmdcount`

Returns a count of the total number of commands that have been invoked in this interpreter.

(iv) `info commands` *?pattern?*

If *pattern* isn't specified, returns a list of names of all the Tcl commands, including both the built-in commands written in C and the command procedures defined using the `proc` command. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for `string match`.

(v) `info complete` *command*

Returns 1 if *command* is a complete Tcl command in the sense of having no unclosed quotes, braces, brackets or array element names, If the command doesn't appear to be complete then 0 is returned. This command is typically used in line-oriented input environments to allow users to type in commands that span multiple lines; if the command isn't complete, the script can delay evaluating it until additional lines have been typed to complete the command.

(vi) `info default` *procname arg varname*

Procname must be the name of a Tcl command procedure and *arg* must be the name of an argument to that procedure. If *arg* doesn't have a default value then the command returns `0`. Otherwise it returns `1` and places the default value of *arg* into variable *varname*.

(vii) `info exists` *varName*

Returns `1` if the variable named *varName* exists in the current context (either as a global or local variable), returns `0` otherwise.

(viii) `info globals` *?pattern?*

If *pattern* isn't specified, returns a list of all the names of currently-defined global variables. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for `string match`.

(ix) `info level` *?number?*

If *number* is not specified, this command returns a number giving the stack level of the invoking procedure, or 0 if the command is invoked at top-level. If *number* is specified, then the result is a list consisting of the name and arguments for the procedure call at level *number* on the stack. If *number* is positive then it selects a particular stack level (1 refers to the top-most active procedure, 2 to the procedure it called, and so on); otherwise it gives a level relative to the current level (0 refers to the current procedure, -1 to its caller, and so on). See the `uplevel` command for more information on what stack levels mean.

(x) `info library`

Returns the name of the library directory in which standard Tcl scripts are stored. The default value for the library is compiled into Tcl, but it may be overridden by setting the TCL_LIBRARY environment variable. If there is no TCL_LIBRARY variable and no compiled-in value then and error is generated. See the `library` manual entry for details of the facilities provided by the Tcl script library. Normally each application will have its own application-specific script library in addition to the Tcl script library; I suggest that each application set a global variable with a name like `$`*app*`_library` (where *app* is the application's name) to hold the location of that application's library directory.

(xi) `info locals` *?pattern?*

If *pattern* isn't specified, returns a list of all the names of currently-defined local variables, including arguments to the current procedure, if any. Variables defined with the `global` and `upvar` commands will not be returned. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for `string match`.

(xii) `info procs` *?pattern?*

If *pattern* isn't specified, returns a list of all the names of Tcl command procedures. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for `string match`.

(xiii) `info script`

If a Tcl script file is currently being evaluated (i.e., there is a call to `Tcl_EvalFile` active or there is an active invocation of the `source` command), then this command returns the name of the innermost file being processed. Otherwise the command returns an empty string.

(xiv) `info tclversion`

Returns the version number for this version of Tcl in the form *x.y*, where changes to *x* represent major changes with probable incompatibilities and changes to *y* represent small enhancements and bug fixes that retain backward compatibility.

(xv) `info vars` *?pattern?*

If *pattern* isn't specified, returns a list of all the names of currently-visible variables, including both locals and currently-visible globals. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for `string match`.

(ac) `join` *list ?joinString?*

The *list* argument must be a valid Tcl list. This command returns the string formed by joining all of the elements of *list* together with *joinString* separating each adjacent pair of elements. The *joinString* argument defaults to a space character.

(ad) `lappend` *varName value ?value value ...?*

Treat the variable given by *varName* as a list and append each of the *value* arguments to that list as a separate element, with spaces between elements. If *varName* doesn't exist, it is created as a list with elements given by the *value* arguments. `Lappend` is similar to `append` except that the *value*s are appended as list elements rather than raw text. This command provides a relatively efficient way to build up large lists. For example, `lappend a $b` is much more efficient than `set a [concat $a [list $b]]` when `$a` is long.

(ae) `lindex` *list index*

Treats *list* as a Tcl list and returns the *index*'th element from it (0 refers to the first element of the list). In extracting the element, *lindex* observes the same rules concerning braces and quotes and backslashes as the Tcl command interpreter; however, variable substitution and command substitution do not occur. If *index* is negative or greater than or equal to the number of elements in *value*, then an empty string is returned.

(af) `linsert` *list index element ?element element ...?*

This command produces a new list from *list* by inserting all of the *element* arguments just before the *index*th element of *list*. Each *element* argument will become a separate element of the new list. If *index* is less than or equal to zero, then the new elements are inserted at the beginning of the list. If *index* is greater than or equal to the number of elements in the list, then the new elements are appended to the list.

(ag) `list` *arg ?arg ...?*

This command returns a list comprised of all the *arg*s. Braces and backslashes get added as necessary, so that the `index` command may be used on the result to re-extract the original arguments, and also so that `eval` may be used to execute the resulting list, with *arg1* comprising the command's name and the other *arg*s comprising its arguments. `List` produces slightly different results than `concat`: `concat` removes one level of grouping before forming the list, while `list` works directly from the original arguments. For example, the command

```
list a b {c d e} {f {g h}}
```

will return

```
a b {c d e} {f {g h}}
```

while `concat` with the same arguments will return

```
a b c d e f {g h}
```

(ah) `llength` *`list`*

Treats *`list`* as a list and returns a decimal string giving the number of elements in it.

(ai) `lrange` *`list first last`*

`List` must be a valid Tcl list. This command will return a new list consisting of elements *`first`* through *`last`*, inclusive. *`Last`* may be `end` (or any abbreviation of it) to refer to the last element of the list. If *`first`* is less than zero, it is treated as if it were zero. If *`last`* is greater than or equal to the number of elements in the list, then it is treated as if it were `end`. If *`first`* is greater than *`last`* then an empty string is returned. Note: `lrange` *`list first first`* does not always produce the same result as `lindex` *`list first`* (although it often does for simple fields that aren't enclosed in braces); it does, however, produce exactly the same results as `list [lindex` *`list first`*`]`

(aj) `lreplace` *`list first last ?element element ...?`*

Returns a new list formed by replacing one or more elements of *`list`* with the *`element`* arguments. *`First`* gives the index in *`list`* of the first element to be replaced. If *`first`* is less than zero then it refers to the first element of *`list`*; the element indicated by *`first`* must exist in the list. *`Last`* gives the index in *`list`* of the last element to be replaced; it must be greater than or equal to *`first`*. *`Last`* may be `end` (or any abbreviation of it) to indicate that all elements between *`first`* and the end of the list should be replaced. The *`element`* arguments specify zero or more new arguments to be added to the list in place of those that were deleted. Each *`element`* argument will become a separate element of the list. If no *`element`* arguments are specified, then the elements between *`first`* and *`last`* are simply deleted.

(ak) `lsearch` *`list pattern`*

Search the elements of *`list`* to see if one of them matches *`pattern`*. If so, the command returns the index of the first matching element. If not, the command returns `-1`. Pattern matching is done in the same way as for the `string match` command.

(al) `lsort` *`list`*

Sort the elements of *`list`*, returning a new list in sorted order. ASCII sorting is used, with the result in increasing order.

(am) `open` *`fileName ?access?`*

Opens a file and returns an identifier that may be used in future invocations of commands like `read`, `puts`, and `close`. *`FileName`* gives the name of the file to open; if it starts with a tilde then tilde substitution is performed as described for `Tcl_TildeSubst`. If the first character of *`fileName`* is | then the remaining characters of *`fileName`* are treated as a command pipeline to invoke, in the same style as for `exec`. In this case, the identifier returned by `open` may be used to write to the command's input pipe or read from its output pipe. The *`access`* argument indicates the way in which the file (or command pipeline) is to be accessed. It may have any of the following values:

(i) `r`

Open the file for reading only; the file must already exist.

(ii) `r+`

Open the file for both reading and writing; the file must already exist.

(iii) `w`

Open the file for writing only. Truncate it if it exists. If it doesn't exist, create a new file.

(iv) `w+`

Open the file for reading and writing. Truncate it if it exists. If it doesn't exist, create a new file.

(v) `a`

Open the file for writing only. The file must already exist, and the file is positioned so that new data is appended to the file.

(vi) `a+`

Open the file for reading and writing. If the file doesn't exist, create a new empty file. Set the initial access position to the end of the file.

`Access` defaults to `r`. If a file is opened for both reading and writing, then `seek` must be invoked between a read and a write, or vice versa (this restriction does not apply to command pipelines opened with `open`). When *`fileName`* specifies a command pipeline and a write-only access is used, then standard output from the pipeline is directed to the current standard output unless overridden by the command. When *`fileName`* specifies a command pipeline and a read-only access is used, then standard input from the pipeline is taken from the current standard input unless overridden by the command.

(an) `proc` *`name args body`*

The `proc` command creates a new Tcl command procedure, *`name`*, replacing any existing command there may have been by that name. Whenever the new command is invoked, the contents of *`body`* will be executed by the Tcl interpreter. *`Args`* specifies the formal arguments to the procedure. It consists of a list, possibly empty, each of whose elements specifies one argument. Each argument specifier is also a list with either one or two fields. If there is only a single field in the specifier, then it is the name of the argument; if there are two fields, then the first is the argument name and the second is its default value. braces and backslashes may be used in the usual way to specify complex default values.

When *`name`* is invoked, a local variable will be created for each of the formal arguments to the procedure; its value will be the value of corresponding argument in the invoking command or the argument's default value. Arguments with default values need not be specified in a procedure invocation. However, there must be enough actual arguments for all the formal arguments that don't have defaults, and there must not be any extra actual arguments. There is one special case to permit procedures with variable numbers of arguments. If the last formal argument has the name `args`, then a call to the procedure may contain more actual arguments than the procedure has formals. In this case, all of the actual arguments starting at the one that would be assigned to `args` are combined into a list (as if the `list` command had been used); this combined value is assigned to the local variable `args`.

When *`body`* is being executed, variable names normally refer to local variables, which are created automatically when referenced and deleted when the procedure returns. One local variable is automatically created for each of the procedure's arguments. Global variables can only be accessed by invoking the `global` command.

The `proc` command returns the null string. When a procedure is invoked, the procedure's return value is the value specified in a `return` command. If the procedure doesn't execute an explicit `return`, then its return value is the value of the last command executed in the procedure's body. If an error occurs while executing the procedure body, then the procedure-as-a-whole will return that same error.

(ao) `puts` *`fileId string ?nonewline?`*

Writes the characters given by *`string`* to the file given by *`fileId`*. `Puts` normally outputs a newline character after *`string`*, but this feature may be suppressed by specifying the `nonewline` argument.

Output to files is buffered internally by Tcl; the `flush` command may be used to force buffered characters to be output. *FileId* must have been the return value from a previous call to `open`, or it may be `stdout` or `stderr` to refer to one of the standard I/O channels; it must refer to a file that was opened for writing.

(ap) `pwd`

Returns the path name of the current working directory.

(aq) `read` variants:

(i) `read` *fileId*
(ii) `read` *fileId nonewline*
(iii) `read` *fileId numBytes*

In the first form, all of the remaining bytes are read from the file given by *fileId*; they are returned as the result of the command. If `nonewline` is specified as an additional argument, then the last character of the file is discarded if it is a newline. In the third form, the extra argument specifies how many bytes to read; exactly this many bytes will be read and returned, unless there are fewer than *numBytes* bytes left in the file; in this case, all the remaining bytes are returned. *FileId* must be `stdin` or the return value from a previous call to `open`; it must refer to a file that was opened for reading.

(ar) `regexp ?-indices? ?-nocase? exp string ?matchVar? ?subMatchVar subMatchVar ...?`

Determines whether the regular expression *exp* matches part or all of *string* and returns 1 if it does, 0 if it doesn't. See REGULAR EXPRESSIONS above for complete information on the syntax of *exp* and how it is matched against *string*.

If the `-nocase` switch is specified then upper-case characters in *string* are treated as lower case during the matching process. The `-nocase` switch must be specified before *exp* and may not be abbreviated.

If additional arguments are specified after *string* then they are treated as the names of variables to use to return information about which part(s) of *string* matched *exp*. *MatchVar* will be set to the range of *string* that matched all of *exp*. The first *subMatchVar* will contain the characters in *string* that matched the leftmost parenthesized subexpression within *exp*, the next *subMatchVar* will contain the characters that matched the next parenthesized subexpression to the right in *exp*, and so on.

Normally, *matchVar* and the *subMatchVar*s are set to hold the matching characters from `string`. However, if the `-indices` switch is specified then each variable will contain a list of two decimal strings giving the indices in *string* of the first and last characters in the matching range of characters. The `-indices` switch must be specified before the *exp* argument and may not be abbreviated.

If there are more *subMatchVar*'s than parenthesized subexpressions within *exp*, or if a particular subexpression in *exp* doesn't match the string (e.g., because it was in a portion of the expression that wasn't matched), then the corresponding *subMatchVar* will be set to `-1 -1` if `-indices` has been specified or to an empty string otherwise.

(as) `regsub` *?-all? ?-nocase? exp string subSpec varName*

This command matches the regular expression *exp* against *string* using the rules described in REGULAR EXPRESSIONS above. If there is no match, then the command returns 0 and does nothing else. If there is a match, then the command returns 1 and also copies *string* to the variable whose name is given by *varName*. When copying *string*, the portion of *string* that matched *exp* is replaced with *subSpec*. If *subSpec* contains a `&` or `\0`, then it is replaced in the substitution with the portion of *string* that matched *exp*. If *subSpec* contains a `\`*n*, where *n* is a digit between 1 and 9, then it is replaced in the substitution with the portion of *string* that matched the *n*-th parenthesized subexpression of *exp*. Additional backslashes may be used in *subSpec* to prevent special interpretation of `&` or `\0` or `\`*n* or backslash. The use of backslashes in *subSpec* tends to interact badly with the Tcl parser's use of backslashes, so it's generally safest to enclose *subSpec* in braces if it includes backslashes. If the

`-all` argument is specified, then all ranges in *string* that match *exp* are found and substitution is performed for each of these ranges; otherwise only the first matching range is found and substituted. If `-all` is specified, then `&` and `\n` sequences are handled for each substitution using the information from the corresponding match. If the `-nocase` argument is specified, then upper-case characters in *string* are converted to lower-case before matching against *exp*; however, substitutions specified by *subSpec* use the original unconverted form of *string*. The `-all` and `-nocase` arguments must be specified exactly: no abbreviations are permitted.

(at) `rename` *oldName newName*

Rename the command that used to be called *oldName* so that it is now called *newName*. If *newName* is an empty string (e.g., {}) then *oldName* is deleted. The `rename` command returns an empty string as result.

(au) `return` *?value?*

Return immediately from the current procedure (or top-level command or `source` command), with *value* as the return value. If *value* is not specified, an empty string will be returned as result.

(av) `scan` *string format varname1 ?varname2 ...?*

This command parses fields from an input string in the same fashion as the C `sscanf` procedure. *String* gives the input to be parsed and *format* indicates how to parse it, using `%` fields as in `sscanf`. All of the `sscanf` options are valid; see the `sscanf` man page for details. Each *varname* gives the name of a variable; when a field is scanned from *string*, the result is converted back into a string and assigned to the corresponding *varname*. The only unusual conversion is for `%c`. For `%c` conversions a single character value is converted to a decimal string, which is then assigned to the corresponding *varname*; no field width may be specified for this conversion.

(aw) `seek` *fileId offset ?origin?*

Change the current access position for *fileId*. The *offset* and *origin* arguments specify the position at which the next read or write will occur for *fileId*. *Offset* must be a number (which may be negative) and *origin* must be one of the following:

(i) `start`

The new access position will be *offset* bytes from the start of the file.

(ii) `current`

The new access position will be *offset* bytes from the current access position; a negative *offset* moves the access position backwards in the file.

(iii) `end`

The new access position will be *offset* bytes from the end of the file. A negative *offset* places the access position before the end-of-file, and a positive *offset* places the access position after the end-of-file.

The *origin* argument defaults to `start`. *FileId* must have been the return value from a previous call to `open`, or it may be `stdin`, `stdout`, or `stderr` to refer to one of the standard I/O channels. This command returns an empty string.

(ax) `set` *varname ?value?*

Returns the value of variable *varname*. If *value* is specified, then set the value of *varname* to *value*, creating a new variable if one doesn't already exist, and return its value.

If *varName* contains an open parenthesis and ends with a close parenthesis, then it refers to an array element: the characters before the open parenthesis are the name of the array, and the characters between the parentheses are the index within the array. Otherwise *varName* refers to a scalar variable.

If no procedure is active, then *varname* refers to a global variable. If a procedure is active, then *varname* refers to a parameter or local variable of the procedure, unless the *global* command has been invoked to declare *varname* to be global.

(ay) `source` *fileName*

Read file *fileName* and pass the contents to the Tcl interpreter as a sequence of commands to execute in the normal fashion. The return value of `source` is the return value of the last command executed from the file. If an error occurs in executing the contents of the file, then the `source` command will return that error. If a `return` command is invoked from within the file, the remainder of the file will be skipped and the `source` command will return normally with the result from the `return` command. If *fileName* starts with a tilde, then it is tilde-substituted as described in the `Tcl_TildeSubst` manual entry.

(az) `split` *string ?splitChars?*

Returns a list created by splitting *string* at each character that is in the *splitChars* argument. Each element of the result list will consist of the characters from *string* between instances of the characters in *splitChars*. Empty list elements will be generated if *string* contains adjacent characters in *splitChars*, or if the first or last character of *string* is in *splitChars*. If *splitChars* is an empty string then each character of *string* becomes a separate element of the result list. *SplitChars* defaults to the standard white-space characters. For example,

```
split "comp.unix.misc" .
```

returns `comp unix misc` and

```
split "Hello world" {}
```

returns `H e l l o { } w o r l d`.

(ba) `string` *option arg ?arg ...?*

Perform one of several string operations, depending on *option*. The legal *option*'s (which may be abbreviated) are:

(i) `string compare` *string1 string2*

Perform a character-by-character comparison of strings *string1* and *string2* in the same way as the C `strcmp` procedure. Return -1, 0, or 1, depending on whether *string1* is lexicographically less than, equal to, or greater than *string2*.

(ii) `string first` *string1 string2*

Search *string2* for a sequence of characters that exactly match the characters in *string1*. If found, return the index of the first character in the first such match within *string2*. If not found, return -1.

(iii) `string index` *string charIndex*

Returns the *charIndex*'th character of the *string* argument. A *charIndex* of 0 corresponds to the first character of the string. If *charIndex* is less than 0 or greater than or equal to the length of the string then an empty string is returned.

(iv) `string last` *string1 string2*

Search *string2* for a sequence of characters that exactly match the characters in *string1*. If found, return the index of the first character in the last such match within *string2*. If there is no match, then return `-1`.

(v) `string length` *string*

Returns a decimal string giving the number of characters in *string*.

(vi) `string match` *pattern string*

See if *pattern* matches *string*; return 1 if it does, 0 if it doesn't. Matching is done in a fashion similar to that used by the C-shell. For the two strings to match, their contents must be identical except that the following special sequences may appear in *pattern*:

[a] `*`

Matches any sequence of characters in *string*, including a null string.

[b] `?`

Matches any single character in *string*.

[c] `[`*chars*`]`

Matches any character in the set given by *chars*. If a sequence of the form *x*−*y* appears in *chars*, then any character between *x* and *y*, inclusive, will match.

[d] `\`*x*

Matches the single character *x*. This provides a way of avoiding the special interpretation of the characters `*?[]\` in *pattern*.

(vii) `string range` *string first last*

Returns a range of consecutive characters from *string*, starting with the character whose index is *first* and ending with the character whose index is *last*. An index of 0 refers to the first character of the string. *Last* may be `end` (or any abbreviation of it) to refer to the last character of the string. If *first* is less than zero then it is treated as if it were zero, and if *last* is greater than or equal to the length of the string then it is treated as if it were `end`. If *first* is greater than *last* then an empty string is returned.

(viii) `string tolower` *string*

Returns a value equal to *string* except that all upper case letters have been converted to lower case.

(ix) `string toupper` *string*

Returns a value equal to *string* except that all lower case letters have been converted to upper case.

(x) `string trim` *string ?chars?*

Returns a value equal to *string* except that any leading or trailing characters from the set given by *chars* are removed. If *chars* is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

(xi) `string trimleft` *string ?chars?*

Returns a value equal to *string* except that any leading characters from the set given by *chars* are removed. If *chars* is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

(xii) `string trimright` *string ?chars?*

Returns a value equal to *string* except that any trailing characters from the set given by *chars* are removed. If *chars* is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

(bb) `tell` *fileId*

Returns a decimal string giving the current access position in *fileId*. *FileId* must have been the return value from a previous call to `open`, or it may be `stdin`, `stdout`, or `stderr` to refer to one of the standard I/O channels.

(bc) `time` *`command ?count?`*

This command will call the Tcl interpreter *`count`* times to execute *`command`* (or once if *`count`* isn't specified). It will then return a string of the form

```
503 microseconds per iteration
```

which indicates the average amount of time required per iteration, in microseconds. Time is measured in elapsed time, not CPU time.

(bd) `trace` *`option ?arg arg ...?`*

Cause Tcl commands to be executed whenever certain operations are invoked. At present, only variable tracing is implemented. The legal *`option`*'s (which may be abbreviated) are:

(i) `trace variable` *`name ops command`*

Arrange for *`command`* to be executed whenever variable *`name`* is accessed in one of the ways given by *`ops`*. *`Name`* may refer to a normal variable, an element of an array, or to an array as a whole (i.e. *`name`* may be just the name of an array, with no parenthesized index). If *`name`* refers to a whole array, then *`command`* is invoked whenever any element of the array is manipulated.

`Ops` indicates which operations are of interest, and consists of one or more of the following letters:

[a] `r`

Invoke *`command`* whenever the variable is read.

[b] `w`

Invoke *`command`* whenever the variable is written.

[c] `u`

Invoke *`command`* whenever the variable is unset. Variables can be unset explicitly with the `unset` command, or implicitly when procedures return (all of their local variables are unset). Variables are also unset when interpreters are deleted, but traces will not be invoked because there is no interpreter in which to execute them.

When the trace triggers, three arguments are appended to *`command`* so that the actual command is as follows:

`command name1 name2 op`

`Name1` and *`name2`* give the name(s) for the variable being accessed: if the variable is a scalar then *`name1`* gives the variable's name and *`name2`* is an empty string; if the variable is an array element then *`name1`* gives the name of the array and name2 gives the index into the array; if an entire array is being deleted and the trace was registered on the overall array, rather than a single element, then *`name1`* gives the array name and *`name2`* is an empty string. *`Op`* indicates what operation is being performed on the variable, and is one of `r`, `w`, or `u` as defined above.

`Command` executes in the same context as the code that invoked the traced operation: if the variable was accessed as part of a Tcl procedure, then *`command`* will have access to the same local variables as code in the procedure. This context may be different than the context in which the trace was created. If *`command`* invokes a procedure (which it normally does) then the procedure will have to use `upvar` or `uplevel` if it wishes to access the traced variable. Note also that *`name1`* may not necessarily be the same as the name used to set the trace on the variable; differences can occur if the access is made through a variable defined with the `upvar` command.

For read and write traces, *`command`* can modify the variable to affect the result of the traced operation. If *`command`* modifies the value of a variable during a read or write trace, then the new value will be returned as the result of the traced operation. The return value from *`command`* is ignored except that if it returns an error of any sort then the traced operation is aborted with an error message saying that the access was denied (this mechanism can be used to implement read-only variables, for example). For write traces, *command* is invoked after the variable's value has been changed; it can write a new value into the variable to override the original value specified in the write operation. To implement read-only variables, *command* will have to restore the old value of the variable.

While *command* is executing during a read or write trace, traces on the variable are temporarily disabled. This means that reads and writes invoked by *command* will occur directly, without invoking *command* (or any other traces) again.

When an unset trace is invoked, the variable has already been deleted: it will appear to be undefined with no traces. If an unset occurs because of a procedure return, then the trace will be invoked in the variable context of the procedure being returned to: the stack frame of the returning procedure will no longer exist. Traces are not disabled during unset traces, so if an unset trace command creates a new trace and accesses the variable, the trace will be invoked.

If there are multiple traces on a variable they are invoked in order of creation, most-recent first. If one trace returns an error, then no further traces are invoked for the variable. If an array element has a trace set, and there is also a trace set on the array as a whole, the trace on the overall array is invoked before the one on the element.

Once created, the trace remains in effect either until the trace is removed with the `trace vdelete` command described below, until the variable is unset, or until the interpreter is deleted. Unsetting an element of array will remove any traces on that element, but will not remove traces on the overall array.

This command returns an empty string.

(ii) `trace vdelete` *name ops command*

If there is a trace set on variable *name* with the operations and command given by *ops* and *command*, then the trace is removed, so that *command* will never again be invoked. Returns an empty string.

(iii) `trace vinfo` *name*

Returns a list containing one element for each trace currently set on variable *name*. Each element of the list is itself a list containing two elements, which are the *ops* and *command* associated with the trace. If *name* doesn't exist or doesn't have any traces set, then the result of the command will be an empty string.

(be) `unknown` *cmdName ?arg arg ...?*

This command doesn't actually exist as part of Tcl, but Tcl will invoke it if it does exist. If the Tcl interpreter encounters a command name for which there is not a defined command, then Tcl checks for the existence of a command named `unknown`. If there is no such command, then the interpeter returns an error. If the `unknown` command exists, then it is invoked with arguments consisting of the fully-substituted name and arguments for the original non-existent command. The `unknown` command typically does things like searching through library directories for a command procedure with the name *cmdName*, or expanding abbreviated command names to full-length, or automatically executing unknown commands as UNIX sub-processes. In some cases (such as expanding abbreviations) `unknown` will change the original command slightly and then (re-)execute it. The result of the `unknown` command is used as the result for the original non-existent command.

(bf) `unset` *name ?name name ...?*

Remove one or more variables. Each *name* is a variable name, specified in any of the ways acceptable to the `set` command. If a *name* refers to an element of an array, then that element is removed without affecting the rest of the array. If a *name* consists of an array name with no parenthesized index, then the entire array is deleted. The `unset` command returns an empty string as result. An error occurs if any of the variables doesn't exist.

(bg) `uplevel` *?level? command ?command ...?*

All of the *command* arguments are concatenated as if they had been passed to `concat`; the result is then evaluated in the variable context indicated by *level*. `Uplevel` returns the result of that evaluation. If *level* is an integer, then it gives a distance (up the procedure calling stack) to move before executing the command. If *level* consists of `#` followed by a number then the number gives an absolute level number. If *level* is omitted then it defaults to `1`. *Level* cannot be defaulted if the first *command* argument starts with a digit or `#`. For example, suppose that procedure `a` was invoked from top-level, and that it called `b`, and that `b` called `c`. Suppose that `c` invokes the `uplevel` command. If *level* is `1` or `#2` or omitted, then the command will be executed in the variable context of `b`. If *level* is `2` or `#1` then the command will be executed in the variable context of `a`. If *level* is `3` or `#0` then the command will be executed at top-level (only global variables will be visible). The `uplevel` command causes the invoking procedure to disappear from the procedure calling stack while the command is being executed. In the above example, suppose `c` invokes the command

```
uplevel 1 {set x 43; d}
```

where `d` is another Tcl procedure. The `set` command will modify the variable `x` in `b`'s context, and `d` will execute at level 3, as if called from `b`. If it in turn executes the command

```
uplevel {set x 42}
```

then the `set` command will modify the same variable `x` in `b`'s context: the procedure `c` does not appear to be on the call stack when `d` is executing. The command `info level` may be used to obtain the level of the current procedure. `Uplevel` makes it possible to implement new control constructs as Tcl procedures (for example, `uplevel` could be used to implement the `while` construct as a Tcl procedure).

(bh) `upvar` *?level? otherVar myVar ?otherVar myVar ...?*

This command arranges for one or more local variables in the current procedure to refer to variables in an enclosing procedure call or to global variables. *Level* may have any of the forms permitted for the `uplevel` command, and may be omitted if the first letter of the first *otherVar* isn't `#` or a digit (it defaults to `1`). For each *otherVar* argument, `upvar` makes the variable by that name in the procedure frame given by *level* (or at global level, if *level* is `#0`) accessible in the current procedure by the name given in the corresponding *myVar* argument. The variable named by *otherVar* need not exist at the time of the call; it will be created the first time *myVar* is referenced, just like an ordinary variable. `Upvar` may only be invoked from within procedures. Neither *otherVar* or *myVar* may refer to an element of an array. `Upvar` returns an empty string.

The `upvar` command simplifies the implementation of call-by-name procedure calling and also makes it easier to build new control constructs as Tcl procedures. For example, consider the following procedure:

```
proc add2 name {
    upvar $name x
    set x [expr $x+2]
}
```

`Add2` is invoked with an argument giving the name of a variable, and it adds two to the value of that variable. Although `add2` could have been implemented using `uplevel` instead of `upvar`, `upvar` makes it simpler for `add2` to access the variable in the caller's procedure frame.

(bi) `while` *test body*

The *while* command evaluates *test* as an expression (in the same way that `expr` evaluates its argument). The value of the expression must be numeric; if it is non-zero then *body* is executed by passing it to the Tcl interpreter. Once *body* has been executed then *test* is evaluated again, and the process repeats until eventually *test* evaluates to a zero numeric value. `Continue` commands may be executed inside *body* to terminate the current iteration of the loop, and `break` commands may be executed inside *body* to cause immediate termination of the `while` command. The `while` command always returns an empty string.

A.21. Built-In Variables

The following global variables are created and managed automatically by the Tcl library. Except where noted below, these variables should normally be treated as read-only by application-specific code and by users.

(a) `env`

This variable is maintained by Tcl as an array whose elements are the environment variables for the process. Reading an element will return the value of the corresponding environment variable. Setting an element of the array will modify the corresponding environment variable or create a new one if it doesn't already exist. Unsetting an element of `env` will remove the corresponding environment variable. Changes to the `env` array will affect the environment passed to children by commands like `exec`. If the entire `env` array is unset then Tcl will stop monitoring `env` accesses and will not update environment variables.

(b) `errorCode`

After an error has occurred, this variable will be set to hold additional information about the error in a form that is easy to process with programs. `errorCode` consists of a Tcl list with one or more elements. The first element of the list identifies a general class of errors, and determines the format of the rest of the list. The following formats for `errorCode` are used by the Tcl core; individual applications may define additional formats.

(i) `CHILDKILLED` *pid sigName msg*

This format is used when a child process has been killed because of a signal. The second element of `errorCode` will be the process's identifier (in decimal). The third element will be the symbolic name of the signal that caused the process to terminate; it will be one of the names from the include file signal.h, such as `SIGPIPE`. The fourth element will be a short human-readable message describing the signal, such as ''write on pipe with no readers'' for `SIGPIPE`.

(ii) `CHILDSTATUS` *pid code*

This format is used when a child process has exited with a non-zero exit status. The second element of `errorCode` will be the process's identifier (in decimal) and the third element will be the exit code returned by the process (also in decimal).

(iii) `CHILDSUSP` *pid sigName msg*

This format is used when a child process has been suspended because of a signal. The second element of `errorCode` will be the process's identifier, in decimal. The third element will be the symbolic name of the signal that caused the process to suspend; this will be one of the names from the include file signal.h, such as `SIGTTIN`. The fourth element will be a short human-readable message describing the signal, such as ''background tty read'' for `SIGTTIN`.

(iv) `NONE`

This format is used for errors where no additional information is available for an error besides the message returned with the error. In these cases `errorCode` will consist of a list containing a single element whose contents are `NONE`.

(v) `UNIX` *errName msg*

If the first element of `errorCode` is `UNIX`, then the error occurred during a UNIX kernel call. The second element of the list will contain the symbolic name of the error that occurred, such as `ENOENT`; this will be one of the values defined in the include file `errno.h`. The third element of the list will be a human-readable message corresponding to *errName*, such as ''no such file or directory'' for the `ENOENT` case.

To set `errorCode`, applications should use library procedures such as `Tcl_SetErrorCode` and `Tcl_UnixError`, or they may invoke the `error` command. If one of these operations hasn't been used, then the Tcl interpreter will reset the variable to `NONE` after the next error.

(c) `errorInfo`

After an error has occurred, this string will contain one or more lines identifying the Tcl commands and procedures that were being executed when the most recent error occurred. Its contents take the form of a stack trace showing the various nested Tcl commands that had been invoked at the time of the error.

A.22. Author

John Ousterhout, University of California at Berkeley (ouster@sprite.berkeley.edu).

Many people have contributed to Tcl in various ways, but the following people have made unusually large contributions: Bill Carpenter, Peter Da Silva, Mark Diekhans, Karl Lehenbauer, Mary Ann May-Pumphrey.

APPENDIX B. Mapping of Existing Control Program Commands

In the mappings lists of this appendix, the commands in the left column map to the commands in the right column, unless specified otherwise.

B.1. rpccp

These are commands that are mapped directly:

(a) `add element`	`nsi-profile addto`
(b) `add entry`	`nsi-entry create`
(c) `add member`	`nsi-group addto`
(d) `export`	`nsi-entry export`
(e) `import`	`nsi-{entry,group,profile} \` `import`
(f) `remove element`	`nsi-profile rmfrom`
(g) `remove entry`	`nsi-entry delete`
(h) `remove group`	`nsi-group delete`
(i) `remove mapping`	`mapping delete`
(j) `remove member`	`nsi-group rmfrom`
(k) `remove profile`	`nsi-profile delete`
(l) `show entry`	`nsi-entry show`
(m) `show group`	`nsi-group show`
(n) `show mapping`	`mapping show -all`
(o) `show profile`	`nsi-profile show`
(p) `show server`	`nsi-entry show`
(q) `unexport`	`nsi-entry unexport`

B.2. cdscp

These are commands that are mapped directly:

(a) `add directory`	`directory modify`
(b) `add object`	`object modify`
(c) `clear cached server`	`cds-cache delete`
(d) `clear clearinghouse`	`clearinghouse disable`
(e) `create child`	`directory addto`
(f) `create clearinghouse`	`clearinghouse create`
(g) `create directory`	`directory create`
(h) `create link`	`link create`
(i) `create object`	`object create`
(j) `create replica`	`directory create -replica -in`
(k) `define cached server`	`cds-cache create`
(l) `delete child`	`directory rmfrom`
(m) `delete clearinghouse`	`clearinghouse delete`
(n) `delete directory`	`directory delete`

(o)	delete link	link delete
(p)	delete object	object delete
(q)	delete replica	directory delete −replica −in
(r)	dump clerk cache	cds−cache dump
(s)	list child	directory list
(t)	list clearinghouse	directory list
(u)	list directory	directory list
(v)	list link	directory list
(w)	list object	directory list
(x)	remove directory	directory modify
(y)	remove link	link modify
(z)	remove object	object modify
(aa)	set directory	directory modify
(ab)	set directory to skulk	directory update
(ac)	set link	link modify
(ad)	set object	object modify
(ae)	show cached clearinghouse	cds−cache show −clearinghouse
(af)	show cached server	cds−cache show −server
(ag)	show child	directory show −child
(ah)	show clearinghouse	clearinghouse show
(ai)	show directory	directory show
(aj)	show link	link show
(ak)	show object	object show
(al)	show replica	directory show −replica −in

These are commands which don’t have a mapping:

(a) disable clerk
(b) disable server
(c) set cdscp preferred clearinghouse
(d) set directory to new epoch
(e) show cdscp confidence
(f) show cdscp preferred clearinghouse
(g) show cell
(h) show clerk
(i) show server

B.3. acl_edit

These are commands that are mapped directly:

(a)	cell	acl cell
(b)	delete	acl modify −remove
(c)	get_access	acl access
(d)	kill_entries	acl delete
(e)	list	acl show
(f)	modify	acl modify {−add | −change}
(g)	permissions	acl permissions
(h)	purge	acl purge
(i)	substitute	acl replace

These are commands which don’t have a mapping:

(a) abort
(b) assign
(c) commit
(d) test_access

B.4. rgy_edit

B.4.1. Principal, group, and organization Commands

These are commands that are mapped directly. In the following list *pgo* is used to mean one of: `principal`, `group`, or `organization`. Also, *go* is used to mean one of: `group` or `organization`.

(a) `add`	*pgo* `create`
(b) `change`	*pgo* `modify`
(c) `delete`	*pgo* `delete`
(d) `member`	*go* `[addto` | `rmfrom]`
(e) `view`	*pgo* `show`

These are commands which don't have a mapping:

(a) `adopt`

B.4.2. Account commands

These are commands that are mapped directly:

(a) `add`	`account create`
(b) `change`	`account modify`
(c) `delete`	`account delete`
(d) `view`	`account show`

These are commands which don't have a mapping:

(a) `cell`

B.4.3. Key management commands

These are commands that are mapped directly:

(a) `ktadd`	`keytab create`
(b) `ktlist`	`keytab show`
(c) `ktdelete`	`keytab delete`

B.4.4. Miscellaneous commands

These are commands that are mapped directly:

(a) `auth_policy`	`registry policy`
(b) `login`	`login`
(c) `policy`	`registry policy`
(d) `properties`	`registry modify`

These are commands which don't have a mapping:
(a) `defaults`
(b) `domain`
(c) `scope`

B.4.5. Local registry commands

These are commands which don't have a mapping:

(a) `delete`
(b) `purge`
(c) `view`

B.5. dtscp

These are commands that are mapped directly:

(a) `advertise`	`dtsd config`

(b) `change`	`dtsd modify`
(c) `delete`	`dtsd stop`
(d) `set`	`dtsd modify`
(e) `show`	`dtsd show`
(f) `synchronize`	`dtsd update`
(g) `unadvertise`	`dtsd config`
(h) `update`	`dtsd time`

These are commands which don't have a mapping:

(a) `create`	— should be a switch on `dtsd`
(b) `disable`	
(c) `enable`	— this concept should go away

REFERENCES

[PORTG] DCE Porting and Testing Guide, Open Software Foundation, 1992.

[POSIX.2] IEEE Std 1003.2 — 1988, Portable Operating System Interface — Shells and Utilities.

[RFC 23] R. Mackey, ''DCE 1.1 Internationalization Guide'', January 1993.

[RFC 24] R. Salz, ''DCE 1.1 Serviceability Proposal'', November 1992.

[RFC 28] S. Luan, R. Weisz, ''A DCE Server Auditable-Event Identification and a Proposed Audit Logging API'', December 1992.

[RFC 29] S. Luan, R. Weisz, ''A Design of the DCE Audit Subsystem — An Implementatin Specification'', December 1992.

[RFC 30] H. Melman, ''DCE Administrative Improvement: Requirements for DCE Shell'', December 1992.

[RFC 34] H. Melman, ''DCE 1.1 Coding Style Guide'', April 1993.

[TCL] J. Ousterhout, ''Tcl: An Embeddable command Language'', Proc. USENIX Winter Conference, January 1990, pp. 133-146.

[TCL MAN] J. Ousterhout, ''tcl(3) — overview of tool command language facilities'', unpublished manual page, University of California at Berkeley, December 1992.

AUTHOR'S ADDRESS

Howard Melman
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
USA

Internet email: melman@osf.org
Telephone: +1-617-621-8989