
# FOSSIL SCRIPTING LANGUAGE

## TH version 1

## MANUAL AND REFERENCE

**May 2009**

**CONTENTS**

# AN INTRODUCTION TO TH

## TH is a Tcl-like language

TH is a string-based command language closely based on the Tcl language. The language has only a few fundamental constructs and relatively little syntax which is meant to be simple. TH is designed to be the scripting language of the “Fossil” source code management system. TH is an interpreted language and it is parsed, compiled and executed when the script runs. In Fossil, TH scripts are typically run to build web pages, often in response to web form submissions.

The basic mechanisms of TH are all related to strings and string substitutions. The TH/Tcl way of doing things is a little different from some other programming languages with which you may already be familiar, so it is worth making sure you understand the basic concepts.

## The “Hello, world” program

The traditional starting place for a language introduction is the classic "Hello, World" program. In TH this program has only a single line:

```
puts “Hello, world\n”
```

The command to output a string in TH is the `puts` command. A single unit of text after the puts command will be printed to the output stream. If the string has more than one word, you must enclose the string in double quotes or curly brackets. A set of words enclosed in quotes or curly brackets is treated as a single unit, while words separated by white space are treated as multiple arguments to the command.

## TH STRUCTURE AND SYNTAX

### Datatypes

TH has at its core only a single data type which is string. All values in TH are strings, variables hold strings and the procedures return strings. Strings in TH consist of single byte characters and are zero terminated. Characters outside of the ASCII range, i.e. characters in the 0x80-0xff range have no TH meaning at all: they are not considered digits or letters, nor are they considered white space.

Depending on context, TH can interpret a string in four different ways. First of all, strings can be just that: text consisting of arbitrary sequences of characters. Second, a string can be considered a list, an ordered sequence of words separated by white space. Third, a string can be a command. A command is a list where the first word is interpreted as the name of a command, optionally followed by further argument words. Fourth, and last, a string can be interpreted as an expression.

The latter three interpretations of strings are discussed in more detail below.

### Lists

A list in TH is an ordered sequence of items, or words separated by white space. In TH the following characters constitute white space:

```
' '    0x20
'\t'   0x09
'\n'   0x0A
'\v'   0x0B
'\f'   0x0C
'\r'   0x0D
```

A word can be any sequence of characters delimited by white space. It is not necessary that a word is alphanumeric at all: “`.%*`” is a valid word in TH. If a word needs to contain embedded white space characters, it needs to be quoted, with either a double quotes or with opening/closing curly brackets. Quoting has the effect of grouping the quoted content into a single list element.

Words cannot start with one of the TH special characters `{ } [ ] \ ;` and `"`. Note that a single quote `‘` is not a special character in TH. To use one of these characters to start a word it must be escaped, which is discussed further later on.

TH offers several built-in commands for working with lists, such as counting the number of words in a list, retrieving individual words from the list by index and appending new items to a list. These commands are discussed in the section “Working with lists”.

## Commands

TH casts everything into the mold of a command, even programming constructs like variable assignment and procedure definition. TH adds a tiny amount of syntax needed to properly invoke commands, and then it leaves all the hard work up to the command implementation.

Commands are a special form of list. The basic syntax for a TH command is:

```
command arg1 arg2 arg3 ...
```

The command is either the name of a built-in command or a TH procedure.

White space is used to separate the command name and its arguments, and a newline character or semicolon is used to terminate a command. TH comments are lines with a "#" character at the beginning of the line, or with a "#" character after the semicolon terminating a command.

## Grouping & substitution

TH does not interpret the arguments to the commands except to perform **grouping**, which allows multiple words in one argument, and **substitution**, which is used to deal with special characters, to fetch variables and to perform nested command calls. Hence, the behavior of the TH command processor can be summarized in three basic steps:

1. Argument grouping.
2. Value substitution of backslash escapes, variables and nested commands
3. Command invocation.

Note that there is no step to evaluate the arguments to a command. After substitution, arguments are passed verbatim to the command and it is up to the command to evaluate its arguments as needed.

## Argument grouping

TH has two mechanisms for grouping multiple words into a single item:

- Double quotes, `" "`
- Curly brackets, `{ }`

Whilst both have the effect of grouping a set of words, they have different impact on the next phase of substitution. In brief, double quotes only group their content and curly brackets group and prevent all substitution on their content.

Grouping proceeds from left to right in the string and is not affected by the subsequent substitution. If a substitution leads to a string which would be grouped differently, it has no effect, as the grouping has already been decided in the preceding grouping phase.

**Value substitutions**

TH performs three different substitutions (see the `th.c/thSubstWord` code for details)

- Backslash escape substitution
- Variable substitution
- Nested command substitution

Like grouping, substitution proceeds from left to right and is performed only once: if a substitution leads to a string which could again be substituted such this not happen.

*Backslash escape substitution*.

In general, the backslash (\) disables substitution for the single character immediately following the backslash. Any character immediately following the backslash will stand as literal. This is useful to escape the special meaning of the characters `{ } [ ] \ ;` and `"`.

There are two specific strings which are replaced by specific values during the substitution phase. A backslash followed by the letter 'n' gets replaced by the newline character, as in C. A backslash followed by the letter 'x' and two hexadecimal digits gets replaced by the character with that value, i.e. writing "`\x20`" is the same as writing a space. Note that the \x substitution does not "keep going" as long as it has hex digits as in Tcl, but insists on two digits. The word `\x2121` is not a single exclamation mark, but the 3 letter word `!21`.

*Variable substitution*

Like any programming language, TH has a concept of variables. TH variables are named containers that hold string values. Variables are discussed in more detail later in this document, for now we limit ourselves to variable substitution.

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 curly brackets 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 "`puts $foo.c`" is equivalent to the command:

"`puts 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

```
puts xyz$x(first)zyx
```

is equivalent to the command

```
puts xyz87zyx
```

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

```
puts xyz$x($index)zyx
```

is equivalent to the command

```
puts xyzmorezyx
```

See the section Variables and arrays below for more information on arrays.

The second special form for variables occurs when the dollar sign is followed by an open curly bracket. In this case the variable name consists of all the characters up to the next curly bracket. Array references are not possible in this form: the name between curly brackets 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
```

A dollar sign followed by something other than a letter, digit, underscore, or left parenthesis is treated as a literal dollar sign. The following prints a single character $.

```
puts x $
```

*Command Substitution*

The last form of substitution is command substitution. A nested command is delimited by square brackets, `[ ]`. The TH interpreter takes everything between the brackets and evaluates it as a command. It rewrites the outer command by replacing the square brackets and everything between them with the result of the nested command.

Example:

```
puts [string length foobar]
=> 6
```

In the example, the nested command is: `string length foobar.` This command returns the length of the string foobar. The nested command runs first. Then, command substitution causes the outer command to be rewritten as if it were:

```
puts 6
```

If there are several cases of command substitution within a single command, the interpreter processes them from left to right. As each right bracket is encountered, the command it delimits is evaluated. This results in a sensible ordering in which nested

commands are evaluated first so that their result can be used in arguments to the outer command.

**Argument grouping revisited**

During the substitution phase of command evaluation, the two grouping operators, the curly bracket and the double quote are treated differently by the TH interpreter.

Grouping words with double quotes allows substitutions to occur within the double quotes. A double quote is only used for grouping when it comes after white space. The string `a″b″` is a normal 4 character string, and not the two character string `ab`.

```
puts a″b″
=> a″b″
```

Grouping words within curly brackets disables substitution within the brackets. Again, A opening curly bracket is only used for grouping when it comes after white space. Characters within curly brackets are passed to a command exactly as written, and not even backslash escapes are processed.

Note that curly brackets have this effect only when they are used for grouping (i.e. at the beginning and end of a sequence of words). If a string is already grouped, either with double quotes or curly brackets, and the curly brackets occur in the middle of the grouped string (e.g. `"foo{bar"`), then the curly brackets are treated as regular characters with no special meaning. If the string is grouped with double quotes, substitutions will occur within the quoted string, even between the brackets.

The square bracket syntax used for command substitution does not provide grouping. Instead, a nested command is considered part of the current group. In the command below, the double quotes group the last argument, and the nested command is just part of that group:

```
puts "The length of $s is [string length $s]."
```

If an argument is made up of only a nested command, you do not need to group it with double-quotes because the TH parser treats the whole nested command as part of the group. A nested command is treated as an unbroken sequence of characters, regardless of its internal structure. It is included with the surrounding group of characters when collecting arguments for the main command.

**Summary**

The following rules summarize the fundamental mechanisms of grouping and substitution that are performed by the TH interpreter before it invokes a command:

- Command arguments are separated by white space, unless arguments are grouped with curly brackets or double quotes as described below.
- Grouping with curly brackets, `{}`, prevents substitutions. Curly brackets nest. The interpreter includes all characters between the matching left and right brace in the

group, including newlines, semicolons, and nested curly brackets. The enclosing (i.e., outermost) curly brackets are not included in the group's value.

- Grouping with double quotes, " ", allows substitutions. The interpreter groups everything until another double quote is found, including newlines and semicolons. The enclosing quotes are not included in the group of characters. A double-quote character can be included in the group by quoting it with a backslash, (i.e. \").
- Grouping decisions are made before substitutions are performed, which means that the values of variables or command results do not affect grouping.
- A dollar sign, $, causes variable substitution. Variable names can be any length, and case is significant. If variable references are embedded into other strings, or if they include characters other than letters, digits, and the underscore, they can be distinguished with the ${varname} syntax.
- Square brackets, [], cause command substitution. Everything between the brackets is treated as a command, and everything including the brackets is replaced with the result of the command. Nesting is allowed.
- The backslash character, \, is used to quote special characters. You can think of this as another form of substitution in which the backslash and the next character or group of characters is replaced with a new character.
- Substitutions can occur anywhere unless prevented by curly bracket grouping. Part of a group can be a constant string, and other parts of it can be the result of substitutions. Even the command name can be affected by substitutions.
- A single round of substitutions is performed before command invocation. The result of a substitution is not interpreted a second time. This rule is important if you have a variable value or a command result that contains special characters such as spaces, dollar signs, square brackets, or curly brackets. Because only a single round of substitution is done, you do not have to worry about special characters in values causing extra substitutions.

**Caveats**

- A common error is to forget a space between arguments when grouping with curly brackets or quotes. This is because white space is used as the separator, while the curly brackets or quotes only provide grouping. If you forget the space, you will get syntax errors about the wrong number of arguments being applied. The following is an error because of the missing space between } and {:

```
if {$x > 1}{puts "x = $x"}
```

- When double quotes are used for grouping, the special effect of curly brackets is turned off. Substitutions occur everywhere inside a group formed with double quotes. In the next command, the variables are still substituted:

```
set x xvalue
set y "foo {$x} bar"
=> foo {xvalue} bar
```

- Spaces are not required around the square brackets used for command substitution. For the purposes of grouping, the interpreter considers everything between the square brackets as part of the current group.

## TH EXPRESSIONS

The TH interpreter itself does not evaluate math expressions. TH just does grouping, substitutions and command invocations. However, several built-in commands see one of more of their arguments as expressions and request the interpreter to calculate the value of such expressions.

The **`expr` command** is the simplest such command and is used to parse and evaluate expressions:

```
puts [expr 7.4/2]
=> 3.7
```

Note that an expression can contain white space, but if it does it must be grouped in order to be recognized as a single argument.

Within the context of expression evaluation TH works with three datatypes: two types of number, integer and floating point, and string. Integer values are promoted to floating point values as needed. The Boolean values True and False are represented by the integer values 1 and 0 respectively. The implementation of expr is careful to preserve accurate numeric values and avoid unnecessary conversions between numbers and strings.

Before expression evaluation takes place, both variable and command substitution is performed on the expression string. Hence, you can include variable references and nested commands in math expressions, even if the expression string was originally quoted with curly brackets. Note that backslash escape substitution is not performed by the expression evaluator.

A TH 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. If an operand is not in integer format, 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. 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:

- As a numeric value, either integer or floating-point.
- As a string enclosed in curly brackets. The characters between the opening bracket and matching closing bracket are used as the operand without any substitutions.
- As a string enclosed in double quotes. The expression parser performs variable and command substitutions on the information between the quotes, and uses the resulting value as the operand.
- As a TH variable, using standard $ notation. The variable's value is used as the operand.

- As a TH command enclosed in square 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 curly brackets to prevent the command parser from performing substitutions on the contents.

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

| | |
|---|---|
| **+ - ~ !** | Unary plus, 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. |
| *** / %** | Multiply, divide, remainder. None of these operands may be applied to string operands, and remainder may be applied only to integers. |
| **+ -** | Add and subtract. Valid for numeric operands only. |
| **<< >>** | Left and right shift. Valid for integer operands only. |
| **< > <= >=** | 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. |
| **== !=** | Boolean equal and not equal. Each operator produces a zero/one result as per above. Valid for all operand types. |
| **eq ne** | Compare two strings for equality (eq) or inequality (ne). These operators return 1 (true) or 0 (false). Using these operators ensures that the operands are regarded exclusively as strings, not as possible numbers:<br>`expr { "9" == "9.0"}`<br>`=> 1`<br>`expr { "9" eq "9.0"}`<br>`=> 0` |
| **&** | Bit-wise AND. Valid for integer operands only. |
| **^** | Bit-wise XOR. Valid for integer operands only |
| **\|** | Bit-wise OR. Valid for integer operands only |
| **&&** | Logical AND. Produces a 1 result if both operands are non-zero, 0 otherwise Valid for integer operands only. Note: there is no “shortcut evaluation”: the right hand is evaluated even if the left hand evaluated to false. |

| | |
|---|---|
| `\|\|` | Logical AND. Produces a 1 result if either operand is non-zero, 0 otherwise Valid for integer operands only. Note: there is no “shortcut evaluation”: the right hand is evaluated even if the left hand evaluated to true. |

All of the binary operators group left-to-right within the same precedence level. For example, the expression `4*2 < 7' evaluates to 0.

All internal computations involving integers are done with the C type `int`, and all internal computations involving floating-point are done with the C type `double`. 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.

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.

## TH VARIABLES

Like almost all programming languages TH has the concept of variables. TH variables bind a name to a string value. A variable name must be unique in its scope, either the global scope or a local scope. TH supports two types of variables: scalars and arrays.

TH 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.

### Working with variables

TH has two key commands for working with variables, set and unset:

```
set varname ?value?
unset varname
info exists varname
```

The **`set` command** returns the value of variable `varname`. If the variable does not exist, then an error is thrown. If the optional argument `value` is specified, then the set command sets the value of varname to `value`, creating a new variable in the current scope if one does not already exist, and returns its value.

The **`unset` command** removes a variable from its scope. The argument `varname` is a variable name. If `varname` refers to an element of an array, then that element is removed without affecting the rest of the array. If `varname` 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 the variable doesn't exist.

The **`info exists` command** returns `1' if the variable named `varname` exists in the current scope, either the global scope or the current local scope, and returns `0' otherwise.

### Scalar variables and array variables

TH 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. TH arrays are one-dimensional associative arrays, i.e. the index can be any single string.

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.

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 TH by using indices that contain multiple concatenated values. For example, the commands

```
set a(2,3) 1
set a(3,6) 2
```

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

In general, array elements may be used anywhere in TH 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.

**Variable scope**

Variables exist in a scope. The TH interpreter maintains a global scope that is available to and shared by all commands executed by it. Each invocation of a user defined command creates a new local scope. This local scope holds the arguments and local variables of that user command invocation and only exists as long a the user command is executing.

If not in the body of user command, then references to varname refer to a global variable, i.e. a variable in the global scope. In contrast, in the body of a user defined command references to varname refer to a parameter or local variable of the command. However, in the body of a user defined command, a global variable can be explicitly referred to by preceding its name by `::`.

TH offers a special command to access variables not in the local scope of the current command but in the local scope of the call chain of commands that leas to the current command. This is the upvar command:

```
upvar ?frame? othervar myvar ?othervar myvar ...?
```

The **`upvar` 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. If `frame` is an integer, then it gives a distance (up the command calling stack) to move. The argument `frame` may be omitted if `othervar` is not an integer (`frame` then defaults to `1'). For each `othervar` argument, the upvar command makes the variable by that name in the local scope identified by the `frame` offset 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. The upvar command is only meaningful from within user defined command bodies. Neither `othervar` nor `myvar` may refer to an element of an array. The upvar command 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 TH commands. For example, consider the following procedure:

```
proc incr {name} {
    upvar $name x
    set x [expr $x+1]
}
```

`incr` is invoked with an argument giving the name of a variable, and it adds one to the value of that variable.

## TH COMMANDS, SCRIPTS AND PROGRAM FLOW

In TH there is actually no distinction between commands (often known as ‘statements’ and 'functions' in other languages) and "syntax". There are no reserved words (like if and while) as exist in C, Java, Python, Perl, etc. When the TH interpreter starts up there is a list of built-in, known commands that the interpreter uses to parse a line. These commands include for, set, puts, and so on. They are, however, still just regular TH commands that obey the same syntax rules as all TH commands, both built-in, and those that you create yourself with the proc command.

### Commands revisited

Like Tcl, TH is build up around commands. A command does something for you, like outputting a string, computing a math expression, or generating HTML to display a widget on the screen.

Commands are a special form of list. The basic syntax for a TH command is:

```
command arg1 arg2 arg3 ...
```

The command is either the name of a built-in command or a TH procedure.

White space is used to separate the command name and its arguments, and a newline character or semicolon is used to terminate a command. TH comments are lines with a “#” character at the beginning of the line, or with a “#” character after the semicolon terminating a command.

### Scripts

Normally, control in TH flows from one command to the next. The next command is either in the same list (if the current command is terminated with a semicolon) or in the next input line. A TH program is thus a TH list of commands.

Such a list of commands is referred to as a “script”. A script is hence a self contained code fragment containing one or more commands. The commands in a script are analogous to statements in other programming languages.

Some commands take one or more scripts as arguments and run those scripts zero or more times depending on the other arguments. For example, the if command executes either the then script or the else script once, depending on the if expression being true or false. The command that takes a script will perform the normal grouping and substitution as part of executing the script.

> Note that the script always needs to be enclosed in curly brackets to prevent substitution taking place twice: once as part of the execution of the top level command and once again when preparing the script.

| |
|---|
| Forgetting to enclose a script argument in curly brackets is common source of errors. |

A few commands (return, error, break and continue) immediately stop execution of the current script instead of passing control to the next command in the list. Control is instead returned to the command that initiated the execution of the current script.

**Command result codes**

Each command produces two results: a result 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 `th.h', and are:

| Name | Value | Meaning |
|---|---|---|
| **TH_OK** | 0 | This is the normal return code, and indicates that the command completed successfully. The string gives the command's return value. |
| **TH_ERROR** | 1 | Indicates that an error occurred; the string gives a message describing the error. |
| **TH_RETURN** | 3 | 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. |
| **TH_BREAK** | 2 | Indicates that the break command has been invoked, so the innermost loop should abort immediately. The string contains “break” or the argument of break, if any |
| **TH_CONTINUE** | 4 | Indicates that the continue command has been invoked, so the innermost loop should go on to the next iteration.. The string contains “break” or the argument of break, if any |

TH programmers do not normally need to think about return codes, since TH_OK is almost always returned. If anything else is returned by a command, then the TH interpreter immediately stops processing commands and returns to its caller. If there are several nested invocations of the TH 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 TH_BREAK code; if it occurs, for stops executing the body of the loop and returns TH_OK to its caller. The for command also handles TH_CONTINUE codes and the procedure interpreter handles TH_RETURN codes. The catch command allows TH programs to catch errors and handle them without aborting command interpretation any further.

**Flow control commands**

The flow control commands in TH are:

```
if expr1 body1 ?elseif expr2 body2? ? ?else? bodyN?
for init condition incr script
break    ?value?
continue ?value?
error    ?value?
catch script ?varname?
```

Below each command is discussed in turn

The **`if` command** has the following syntax:

```
if expr1 body1 ?elseif expr2 body2? ? ?else? bodyN?
```

The `expr` arguments are expressions, the `body` arguments are scripts and the `elsif` and `else` arguments are keyword constant strings. The `if` command optionally executes one of its body scripts.

The `expr` arguments must evaluate to an integer value. If it evaluates to a non-zero value the following `body` script is executed and upon return from that script processing continues with the command following the if command. If an `expr` argument evaluates to zero, its `body` script is skipped and the next option is tried. When there are no more options to try, processing also continues with the next command.

The if command returns the value of the executed script, or “0” when no script was executed.

The **`for` command** has the following syntax:

```
for init condition incr body
```

The init, incr and body arguments are all scripts. The condition argument is an expression yielding an integer result. The `for` command is a looping command, similar in structure to the C for statement.

The `for` command first invokes the TH interpreter to execute `init`. Then it repeatedly evaluates `condition` as an expression; if the result is non-zero it invokes the TH interpreter on `body`, then invokes the TH interpreter on `incr`, then repeats the loop. The command terminates when test evaluates to zero.

If a `continue` command is invoked within execution of the body script then any remaining commands in the current execution of body are skipped; processing continues by invoking the TH interpreter on `incr`, then evaluating `condition`, 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.

The for command returns an empty string.

The **break command** has the following syntax:

```
break ?value?
```

The `break` command returns immediately from the current procedure (or top-level command), with `value` as the return value and `TH_BREAK` as the result code. If `value` is not specified, the string “break” will be returned as result.

The **continue command** has the following syntax:

```
continue ?value?
```

The `continue` command returns immediately from the current procedure (or top-level command), with `value` as the return value and `TH_CONTINUE` as the result code. If `value` is not specified, the string “continue” will be returned as result.

The **error command** has the following syntax:

```
error ?value?
```

The `error` command returns immediately from the current procedure (or top-level command), with `value` as the return value and `TH_ERROR` as the result code. If `value` is not specified, the string “error” will be returned as result.

The **catch command** has the following syntax:

```
catch script ?varname?
```

The catch command may be used to prevent errors from aborting command interpretation. The catch command calls the TH interpreter recursively to execute `script`, and always returns a `TH_OK` code, regardless of any errors that might occur while executing `script`.

The return value from catch is a decimal string giving the code returned by the TH interpreter after executing `script`. This will be `0' (TH_OK) if there were no errors in command; otherwise it will have a non-zero value corresponding to one of the exceptional result codes. If the `varname` argument is given, then it gives the name of a variable; catch sets the value of the variable to the string returned from running `script` (either a result or an error message).

**Creating user defined commands**

The proc command creates a new command. The syntax for the proc command is:

```
proc name args body
```

The `proc` command creates a new TH 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 TH interpreter.

The parameter `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. Curly brackets and backslashes may be used in the usual way to specify complex default values.

The `proc` command returns the null string.

**Execution of user defined commands**

If a command is a user defined command (i.e. a command created with the proc command), then the TH interpreter creates a new local variable context, binds the formal arguments to their actual values (i.e. TH uses call by value exclusively) and loads the body script. Execution then proceeds with the first command in that script. Execution ends when the last command has been executed or when one of the returning commands is executed. When the script ends, the local variable context is deleted and processing continues with the next command after the user defined command.

More in detail, when a user defined command is invoked, a local variable is created for each of the formal arguments to the procedure; its value is 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 be accessed by using the `::` syntax.

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.

The syntax for the return command is:

```
return ?-code code? ?value?
```

The optional argument pair `–code code` allows to change the return status code from the default of TH_OK to another status code. This code has to be specified with its numeric value.

**Special commands**

TH includes three core commands that assist with working with commands. They are:

```
breakpoint args
rename oldcmd newcmd
uplevel ?level? script
```

The **`breakpoint` command** does nothing. It is used as placeholder to place breakpoints during debugging.

The **`rename` command** renames a user defined or a built-in command. The old name is removed and the new name is inserted in the interpreter’s command table.

The **`uplevel` command** executes a command in the variable scope of a command higher up in the call chain. The `script` argument is evaluated in the variable scope indicated by `level`. The uplevel command 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` is omitted then it defaults to `1'.

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 omitted, then the command will be executed in the variable context of b. If level is `2' then the command will be executed in the variable context of a. If level is `3' then the command will be executed at top-level (i.e. 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 TH procedure. The set command will modify the variable x in the context of b, 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 the context of b context: the procedure c does not appear to be on the call stack when d is executing.

## WORKING WITH STRINGS

TH provides the **`string` command** to facilitate working with strings. The string command is a single command with seven subcommands, identified by the first argument. The first argument serves no purpose other than to identify the subcommand. If the first argument does not match a subcommand, an error is thrown.

The seven string subcommands are:

```
string length string
string compare string1 string2
string first needle haystack ?startindex?
string last needle haystack ?startindex?
string range string first last
string repeat string count
string is alnum string
```

The **`string length` subcommand** takes one parameter, which is a string. It returns the decimal string with the length of the string. As TH uses a single byte character encoding the string size is both the size in characters and in bytes.

The **`string compare` subcommand** performs a character-by-character comparison of argument strings `string1` and `string2` in the same way as the C *strcmp* procedure. It returns a decimal string with value -1, 0, or 1, depending on whether string1 is lexicographically less than, equal to, or greater than string2.

The **`string first` subcommand** searches argument `haystack` for a sequence of characters that exactly match the characters in argument `needle`. If found, it returns a decimal string with the index of the first character in the first such match within haystack. If not found, it returns return -1. The optional integer argument `startindex` specifies the position where the search begins; the default value is 0, i.e. the first character in `haystack`.

The **`string last` subcommand** searches argument `haystack` for a sequence of characters that exactly match the characters in argument `needle`. If found, it returns a decimal string with the index of the first character in the last such match within haystack. If not found, it returns return -1. The optional integer argument `startindex` specifies the position where the search begins; the default value is 0, i.e. the first character in `haystack`.

The **`string range` subcommand** returns a range of consecutive characters from argument `string`, starting with the character whose index is `first` and ending with the character whose index is `last`. An index of zero refers to the first character of the string. last may be `end` to refer to the last character of the string. If first is less than zero then it is treated as if it was 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.

The **`string repeat` subcommand** returns a string that is formed by repeating the argument `string` for count times. The argument `count` must be an integer. If `count` is zero or less the empty string is returned.

The **`string is alnum` subcommand** tests whether the argument string is an alphanumeric string, i.e. a string with only alphanumeric characters. It returns a decimal string with value 1 if the string is alphanumeric, and with value 0 it is not.

## WORKING WITH LISTS

The list is the basic TH data structure. A list is simply an ordered collection of items, numbers, words, strings, or other lists. For instance, the following string is a list with four items. The third item is a sub-list with two items:

```
{first second {a b} fourth}
```

TH has three core commands to work with lists:

```
list ?arg1 ?arg2? ...?
lindex list index
llength list
```

The **`list` command** returns a list comprising all the args. Braces and backslashes get added as necessary, so that the lindex command may be used on the result to re-extract 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}}
```

The **`lindex` command** treats argument `list` as a TH list and returns the element with `index` number index from it. The argument `index` must be an integer number and zero refers to the first element of the list. In extracting the element, the lindex command observes the same rules concerning braces and quotes and backslashes as the TH 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, an empty string is returned.

The **`llength` command** treats argument `list` as a list and returns a decimal string giving the number of elements in it.