Language Overview

1. Language Overview

2. Substitutions, Quoting, Comments

3. Expressions and Control Commands

4. Lists and Arrays

5. Procedures and Scopes

6. Additional Core Tcl Commands
Why a new command language?

Common problem

- Interactive programs need a command language
- Most programs use a different command language
- Many command languages are limited and ad-hoc
- Some command languages are good but not reusable

Tcl solution:

- Reusable command language interpreter
- Implemented as an embeddable C library
- Procedures, variables, associative arrays, lists, expressions, loops, exceptions, introspection, etc.
- Applications can add new commands to the interpreter
Why should I use Tcl in my Application?

- All applications using Tcl use the same syntax
- Tcl makes applications easily programmable
- Tcl extensions support the integration of simple applications as components into larger applications
- Breaking a complex application into modules written in different languages:
  - Complex data structures and speed ⇒ C components
  - Composition of components ⇒ Tcl scripts
- Tcl is freely available, well documented and very stable
Real-world usage of Tcl

- Web applications (AOL server)
- Testing and automation (see the expect extension)
- Scripting language of some Cisco devices
- Embedded language of some set-top boxes
- Configuration language of the ns-2 network simulator
- Software regression testing
- Scripting of embedded database systems
- Industrial control language (CAD, EDA)
- ...
Design Goals

- Simple syntax, easy to learn
- Supports all constructs of “real” programming languages
- Easy to implement and easy to interpret
- Simple interface to the C programming language
- No distinction between built-in and extension commands
- Usage simplicity is more important than efficiency
- Portability: Win32, Unix, MacOS, Linux, ...
Basic Principles of Tcl

- Commands have a sequence of arguments, separated by white space
- Commands can be separated by line ends or semicolons
- Variables:
  ```tcl
  set a hello
  set b $a
  ```
- Substitution of results of Tcl commands:
  ```tcl
  set b 42
  set a [expr $b+2]
  ```
- Quoting (almost) like in a shell: ", { }, \"
- Simple substitution rules (but sometimes confusing)
Tcl data types:
- Everything can be represented as a character string
- Character strings can be interpreted as lists
- Associative arrays support fast keyed access to data
- Program text and data is interchangeable

Tcl commands:
- All language constructs are commands (no keywords)
- Some command arguments can be arithmetic expressions
- Conditional commands and loop commands
- Procedures are sequences of commands that can be called
- Built-in commands to access file systems and basic operating system services
A Tcl script is a sequence of Tcl commands
- Commands are separated by line ends or semicolons
- The words of a command are separated by white space
- The first word of a command is the command name
- The remaining words are the arguments passed to the command

### Examples

- `set a 23; set b 33`
- `set t Hello`
- `string match *ll* Hello`
- `expr 23 * 2 - 4`
Command Execution in the Interpreter

command

Tcl parser

1. word (command name)

2. word (1. argument)

3. word (2. argument)

4. word (3. argument)

5. word (4. argument)

command procedure

interpretation

result (character string)
The parser does not associate any semantics to command arguments.

Different commands can interpret arguments in differently.

Examples

```
set a "5+9"              -> 5+9
set a                   -> 5+9
expr "5+9"              -> 14
eval "set a 5+9"        -> 5+9
exec ls | wc -l          -> 12
if {5+9<15} {
    puts stdout Abraccadabra
}
        -> ok
```
Substitutions, Quoting, Comments

1. Language Overview
2. Substitutions, Quoting, Comments
3. Expressions and Control Commands
4. Lists and Arrays
5. Procedures and Scopes
6. Additional Core Tcl Commands
Variable Substitution

- A $ in front of a variable name leads to the substitution of the name with the value of the named variable.
- A name of a variable may contain letters, digits, and underscores (actually more if you really want).
- Variable substitution can happen anywhere in a word (unless care has been taken to prevent it).

Examples

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>set b 66</td>
<td>-&gt; 66</td>
</tr>
<tr>
<td>set a b</td>
<td>-&gt; b</td>
</tr>
<tr>
<td>set a $b</td>
<td>-&gt; 66</td>
</tr>
<tr>
<td>set a $b+$b</td>
<td>-&gt; 66+66</td>
</tr>
<tr>
<td>set a $b.3</td>
<td>-&gt; 66.3</td>
</tr>
<tr>
<td>set a 2$b</td>
<td>-&gt; 266</td>
</tr>
<tr>
<td>set a $b2</td>
<td>-&gt; no such variable</td>
</tr>
</tbody>
</table>
A Tcl script enclosed in brackets \([\] \) will be evaluated and replaced by the result of the evaluation.

- Command substitutions can be nested.
- Command substitution can happen anywhere in a word (unless care has been taken to prevent it).

**Examples**

- `set b 66` \(\rightarrow 66\)
- `set a [expr $b+2]` \(\rightarrow 68\)
- `set a b+2=[expr $b+2]` \(\rightarrow b+2=68\)
- `set a [expr $b+[string length $a]]` \(\rightarrow 72\)
Quoting Rules

- Double quotes " prevent word splitting but substitutions take place within the enclosed character string.
- Curly braces { } prevent word splitting and substitutions.
- A backslash \ replaces the following character with a special sequence of characters or the character itself, for example \r \n \{
- Substitutions do not impact word boundaries.

Examples

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>set b 66</td>
<td>-&gt; 66</td>
</tr>
<tr>
<td>set a &quot;b value $b&quot;</td>
<td>-&gt; b value 66</td>
</tr>
<tr>
<td>set a {b value $b}</td>
<td>-&gt; b value $b</td>
</tr>
<tr>
<td>set c &quot;{value}&quot;</td>
<td>-&gt; {value}</td>
</tr>
<tr>
<td>set a &quot;b $c [set b]&quot;</td>
<td>-&gt; b {value} 66</td>
</tr>
</tbody>
</table>
Comments

- A hash character `#` starts a comment when the Tcl parser expects the start of a command.
- A comment ends at the end of the line.
- A `#` character within a comment does not have a special meaning.

Examples

```
# comment
set a one; # set a two  ->  one
set a #                 ->  #
set a one # comment     ->  wrong # args
set a one ; # comment   ->  one
```
Expressions and Control Commands

1. Language Overview
2. Substitutions, Quoting, Comments
3. Expressions and Control Commands
4. Lists and Arrays
5. Procedures and Scopes
6. Additional Core Tcl Commands
The `expr` command evaluates expressions

Expressions are similar to expressions in the C language

Expression may include string comparisons and a number of maths functions

**Examples**

- `expr (5*8)+2` ➞ 42
- `expr (5*8)+2.0` ➞ 42.0
- `expr cos(0)` ➞ 1.0
- `expr 5/3` ➞ 1
- `expr 5/double(3)` ➞ 1.66667
- `expr {"Alf" < "Bert"}` ➞ 1
- `expr "Alf" < "Bert"` ➞ syntax error
Command: expr

- Tcl allows command and variable substitutions in expressions
- Numbers with a leading 0 are treated as octal numbers
- Numbers starting with 0x are treated as hexadecimal numbers
- Precision is controlled by the Tcl variable `tcl_precision`

### Examples

- `set a 22` → 22
- `expr ($a*4)-3` → 85
- `expr {($a * [string length "date"])}` → 88
- `expr 011` → 9
- `expr 09` → syntax error
- `set tcl_precision 12` → 12
- `expr 5/3.0` → 1.66666666667
The if command evaluates the first argument as an expression and, depending on the result, one of the script arguments is evaluated.

The words (arguments) then and else are optional.

Chaining of expressions using elseif is possible.

Examples

```tcl
if {$a == "yes"} then {
    set a 1
} elseif {$a == "no"} {
    set a 0
} else {
    set a ""
}
if $n { set a $n } { set a "" }
```
Command: **switch**

- The **switch** command compares the first argument against a number of pattern and evaluates the corresponding script.
- Options set the type of comparison (exact, glob, regexp).

**Examples:**

```tcl
switch $a {
    yes        { set a 1 }
    no         { set a 0 }
    default    { set a "" }
}

switch -glob $a {
    [yY]*      { set a 1 }
    [nN]*      { set a 0 }
    *          { set a "" }
}
```
The **for** command is implemented as follows:

1. Evaluate the first argument as a Tcl script
2. Evaluates the second argument as an expression and stop if the result is zero
3. Evaluate the fourth argument as a Tcl script.
4. Evaluate the third argument as a Tcl script and repeat from step two

**Example**

```tcl
for {set i 0} {$i < 10} {incr i} {
    puts "$i [expr sqrt($i)]"
}
```
The `while` command is implemented as follows:

1. Evaluate the first argument as an expression and stop if the result is zero
2. Evaluate the second argument as a Tcl script and repeat from step one

Example

```tcl
set i 0
while {$i < 10} {
    puts "$i [expr sqrt($i)]"
    incr i
}
```
The `foreach` command successively assigns the values contained in the (list of the) second argument to the variable named in the first argument and executes the third argument as a Tcl script after each assignment.

**Examples:**

```tcl
foreach letter "a e i o u" {
    puts "$letter is a vocal"
}
foreach file [glob *.c] {
    if {! [file readable $file]} {
        puts "$file not readable"; break
    }
}
```
Command: break and continue

- The break command stops the execution of the enclosing for, while, or foreach command.
- The continue command causes a jump to the next iteration in a for, while, or foreach loop.

Examples:

```tcl
foreach file [glob *.c] {
    if {! [file isfile $file]} {
        continue
    }
    if {! [file readable $file]} {
        break
    }
}
```
A list consists of elements separates by white space

Curly braces can be used to construct lists within a list

Lists can always be split into commands - every list element represents a word

Commands for manipulating lists:

- `list`
- `lappend`
- `linsert`
- `llength`
- `lindex`
- `lsearch`
- `lreplace`
- `lrange`
- `lsort` (for sorting)
- `concat`
- `foreach`

Examples

```
set a "a b" -> a b
set l [list $a b] -> {a b} b
lappend l $l -> {a b} b {{a b} b}
llength $l -> 3
lindex $l 2 -> {a b} b
```
Arrays

- An array is constructed by adding parenthesis enclosing an index value to a variable name.
- Arrays are associative (indexed by strings).
- A variable is either a scalar or an array but never both.
- The array `env` contains the environment variables.
- The array command allows to iterate over arrays.

Examples

```tcl
set a(0) zero        -> zero
set env(SHELL)       -> /bin/sh
set env foo          -> variable is array

foreach index [lsort [array names env]] {
    puts "\$index \t \$env(\$index)"
}
```
Procedures and Scopes

1 Language Overview

2 Substitutions, Quoting, Comments

3 Expressions and Control Commands

4 Lists and Arrays

5 Procedures and Scopes

6 Additional Core Tcl Commands
The \texttt{proc} command creates procedures.

1. The first argument defines the name of the procedure
2. The second argument defines the list of argument names
3. The third argument defines the body of the procedure

Procedures behave like all other Tcl commands.

An argument name consisting of a list of two elements defines a default value.

The special argument name \texttt{args} allows a variable number of arguments.

The \texttt{return} command returns the result produced by the procedure and returns from the procedure call.
Procedures

Examples

proc write {text {file stdout}} { 
    puts -nonewline $file $text
}
proc writeln {text {file stdout}} { 
    puts $file $text
}

proc sum {args} { 
    set sum 0
    foreach num $args { incr sum $num }
    return $sum
}
Variable Scopes

- Every Tcl procedure invocation maintains its own set of variables.
- Variables within a procedure have by default local scope.
- The `global` command marks variables as global.

Examples

```tcl
proc b {b} { return [set a $b] }  
set a 42  
psets "[b 24] $a"

set term $env(TERM)  
proc initialize {} {  
    global term  
    puts $term
}  ```
Variable Scopes

- \texttt{upvar} binds a local variable to a variable in the selected context in the call stack
- \texttt{uplevel} evaluates a script in the selected context in the call stack

Example

\begin{verbatim}
proc ldelete {varName element} {
  upvar 1 $varName list
  set result ""
  foreach e $list {
    if {$e != $element} { lappend result $e }
  }
  set list $result
}
\end{verbatim}
Variable Scopes

Examples

```tcl
proc do { body while expr } {
    if {$while != "while"} {
        error "expected while but got \"$while\"
    }
    uplevel 1 $body
    uplevel 1 [list while $expr $body]
}

proc try {body {foo {}}} {
    if {$foo == ""} {
        catch [list uplevel 1 $body]
    } else {
        if {[catch [list uplevel 1 $body] errorMsg]} {
            catch [list uplevel 1 $foo]
        }
    }
}
```
Additional Core Tcl Commands

1. Language Overview
2. Substitutions, Quoting, Comments
3. Expressions and Control Commands
4. Lists and Arrays
5. Procedures and Scopes
6. Additional Core Tcl Commands
String Manipulation Commands

append   format   scan   string   regexp   regsub
split    join

Examples

scan $date "\%*s \%*s \%*s \%s \%*s \%*s" time -> 1
set $time -> 15:55:30
split $time : -> 15 55 30
join [split $time :] - -> 15-55-30

string compare gross klein -> -1
set name [string tolower Mayer] -> mayer
string match {m[ae][iy]er} $name -> 1
regsub -all {"([aAoOuU])} {j"urgen} {\1e} name -> juergen

Jürgen Schönwälder Short Introduction to the Tool Command Language (Tcl)
File Manipulation Commands

- open
- close
- seek
- tell
- eof
- gets
- read
- puts
- flush
- exec
- file
- glob
- cd
- pwd
- source

Examples

```tcl
foreach file [lsort [glob *]] {
    puts [format "\%8d \%s" [file size $file] $file]
}
```

```tcl
proc ftp {user hostname server remotefile localfile} {
    set f [open "| ftp -n" "r+"
    puts $f "open $server"
    puts $f "user anonymous $user@$hostname"
    puts $f "get $remotefile $localfile"
    puts $f "close"
    flush $f
    close $f
}
```
Errors and Exceptions

- An error stops the execution of a command and returns an error message as the error result up the call stack.
- The global variable `errorInfo` contains the call stack trace.
- The `catch` command evaluates a script and catches any errors so that scripts can handle them.
- The `error` command generates (throws) an error.

Example

```tcl
proc sc {} {
    while {![gets stdin expr] > 0} {
        if [catch {expr $expr} msg] {
            puts stderr "Error : $msg"
            puts stderr "$expr ==> $msg"
        } else {
            puts "$expr ==> $msg"
        }
    }
}
```
References

J. K. Ousterhout.
TCL: An Embeddable Command Language.

J. K. Ousterhout.
An X11 Toolkit Based on the TCL Language.

J. K. Ousterhout.
Tcl and the Tk Toolkit.
Addison-Wesley, April 1994.

J. K. Ousterhout.
Scripting: Higher-level programming for the 21st century.

B. Welch and K. Jones.
Practical Programming in Tcl and Tk.