Variable Names and Values

- Variable names
  - Are CASE SENSITIVE
  - May be composed of ANY characters

Use only letters, digits and underscores in variable names to maintain readability!

- Strings can also represent numeric values:
  - Integers: `123` (decimal), `0x7b` or `0x7B` (hexadecimal), `0173` (octal)
  - Real numbers: `3.14159`, `1.23e+2` or `1.23E2`, `123.0`
  - Boolean: `0` (false), `1` (true)

`a` and `A` are different variables

```
% set a 123; set A 456
456
% set a
123
% set !£%^&* "bad idea!"
bad idea!
% set !£%^&*
bad idea!
```

in Tcl this is a valid variable name!

```
% set reg 0173
0173
% set reg 0x7b
0x7b
% set pi 3.14159
3.14159
% set match_found 1
1
```

integer (octal)

integer (hex)

real number

Boolean

Variable Substitution

- Use `${varName}` to indicate variable name when
  - No space is required after the variable's value
  - Name contains characters other than letters, digits or underscores
- `$` followed by a character which is not a letter, digit or underscore is treated as an ordinary character

  ⇒ Constructs such as `$$a` are allowed

```
% puts "Time = $dns"
can't read "dns": no such variable
% puts "Time = ${d}ns or $p%"
Time = 10ns or 2%
% puts "This is $!£%^&*"
This is $!£%^&*
% puts "This is a ${!£%^&*}"
This is a bad idea!
% puts "The price is $$d."
The price is $10.
```

no space after variable's value (`$p%` doesn't need `${}`!)

use `${}` to indicate the variable name

`$` is treated as an ordinary character (doesn't need to be escaped)

Everyone’s Favourite: The Newline

- Inserting newlines

```
% puts "Line 1
Line 2"
Line 1
Line 2
%
```

newlines within `""` or `{}` are reserved

2nd newline inserted by `puts`

```
% puts "Line 1\nLine 2"
Line 1
Line 2
%
```

insert a newline with `\n`

- Avoiding newlines

```
% puts "Line 1\
        Line 2"
Line 1 Line 2
%
```

escaped newline + zero or more spaces or tabs are replaced with a **space!**

  – If you don’t want the space, use `\b`

delete the substituted space with a backspace

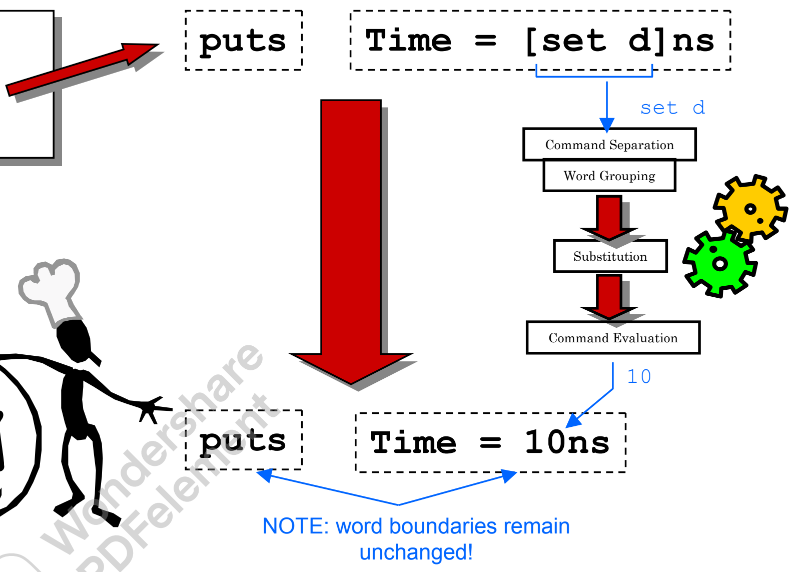
```
% puts "No\
   \bSpace"
NoSpace
%
```

Newlines are very useful for improving readability of user messages, simulation data, serial bitstream, formatting raw data strings, etc. The above examples demonstrate some of the typical uses of newlines in Tcl scripts.

Command Substitution (`[]`)

- Each occurrence of `[<commands>]` is replaced with the value returned from the *last* command executed in `<commands>`
  - Except where `[]` are escaped (`\[` and `\]`)

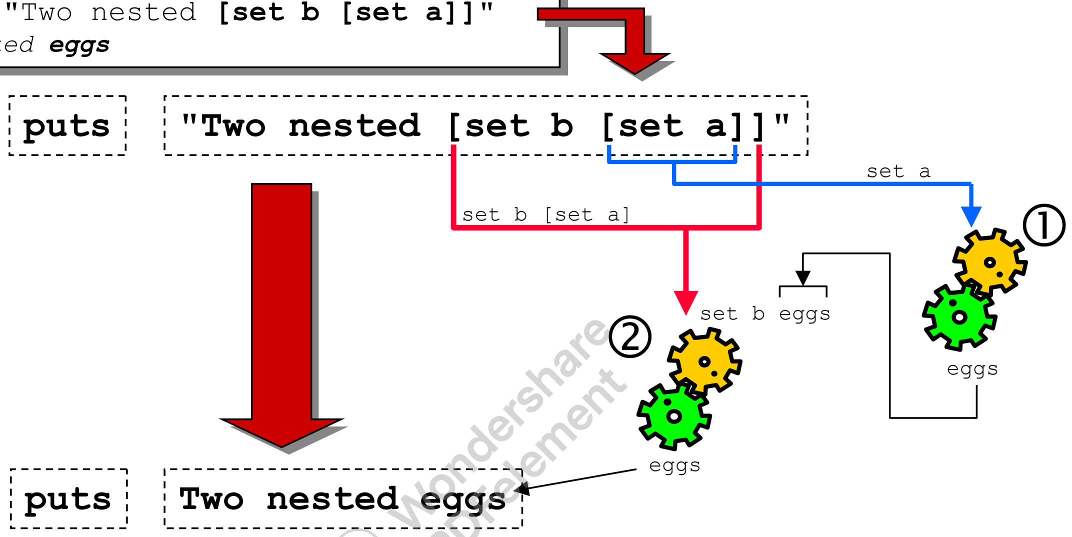
```
% set d 10
10
% puts "Time = [set d]ns"
Time = 10ns
```



Nested Command Substitution

- Command substitution can be nested
  - Each occurrence of `[]` will trigger a new command substitution

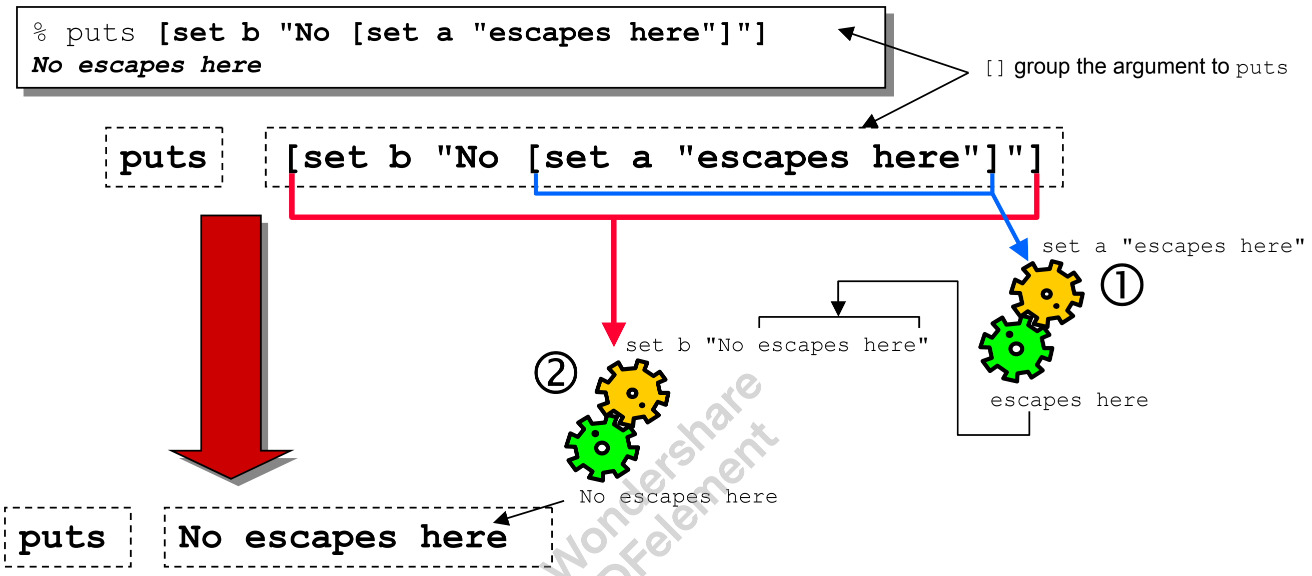
```
% set a "eggs"
eggs
% puts "Two nested [set b [set a]]"
Two nested eggs
```



Word Grouping with `[]`

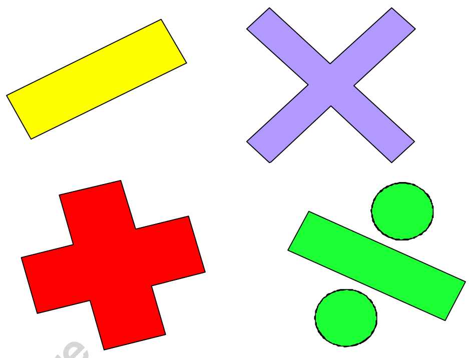
- `[]` can be used for word grouping too!
  - Do not escape word separators and grouping quotes within `[]`

```
% puts [set b "No [set a "escapes here"]"]
No escapes here
```



Mathematical Operations

- Tcl supports all usual operators and functions
- Mathematical
  - Arithmetic
  - Relational
  - Logical
  - Bit-wise
  - Various functions
- But also
  - String pattern matching
  - List, array, string manipulation
  - File I/O, inter-process, network
  - ... (more later)



Tcl as a Calculator

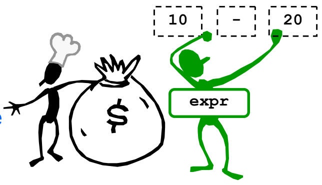
- Use `expr` command to evaluate mathematical expressions

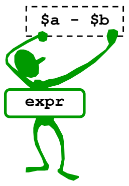
```
% expr 1 + 1
2
% set a 10; set b 20
20
% expr $a - $b
-10
% expr {$a - $b}
-10
```

substitution performed first by the **Tcl interpreter** and then by the **`expr` command**

substitution performed only by the **`expr` command**!

Enclose the expression in braces for efficiency





- Use real number strings to invoke real number operators

```
% expr 9 / 2
4
% expr 9 / 2.0
4.5
```

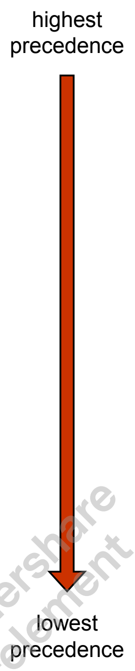
integer division

real number division

If a variable may contain integers AND real numbers, consider adding `+ 0.0` *in the expression, e.g.* `set a [expr {$a + 0.0}]`

Available Operators

- Arithmetic operators accept integers and reals
- Relational operators accept integers, reals and *strings*
- Logical operators accept integers and reals
  - non-zero = true
  - zero = false
- Bit-wise operators accept integers only
- Evaluated from left to right in the order from highest to lowest precedence



```
-a, !a, ~a
a*b, a/b, a%b
a+b, a-b
a<<b, a>>b
a<b, a>b, a<=b, a>=b
a==b, a!=b
a&b
a^b
a|b
a&&b
a||b
a?b:c
```

Operator	Description	Operands

`-a`
`!a`
`~a`
	negative of `a`
	logical NOT: `1` if a is zero, `0` otherwise
	bit-wise complement of `a`
		integer, real
		integer, real
		integer

`a*b, a/b`
`a%b`
	multiply `a` and `b`, divide `a` by `b`
	remainder after integer division of `a` by `b`
		integer, real
		integer

`a+b, a-b`	add `a` and `b`, subtract `b` from `a`	integer, real
`a<<b, a>>b`	left-shift `a` by `b` bits, right-shift `a` by `b` bits	integer

`a<b, a>b,`
`a<=b, a>=b`
	`1` if the comparison is true: `a` less than `b`, `a` greater than `b`, `a` less than or equal to `b`, `a` greater than or equal to `b`	integer, real, string

`a==b, a!=b`	`1` if the comparison is true: `a` is equal to `b`, `a` is not equal to `b`	integer, real, string
`a&b`	bit-wise AND of `a` and `b`	integer
`a^b`	bit-wise exclusive OR of `a` and `b`	integer
`a|b`	bit-wise OR of `a` and `b`	integer
`a&&b`	logical AND: `1` if both `a` and `b` are non-zero, `0` otherwise	integer, real
`a||b`	logical OR: `1` if either `a` or `b` is non-zero, `0` otherwise	integer, real
`a?b:c`	if `a` is non-zero then evaluates to `b`, otherwise evaluates to `c`	integer, real (for `a`)

Examples: Tcl Operators

```
% set a 2
2
% expr $a > 0 && $a <= 3
1
% expr !(($a == 1) || ($a == 2))
0
% expr $a || 0
1
```

3rd, 1st, 2nd

1 = true

0 = false

non-zero evaluates to true

OK because > and <= have higher precedence than &&

Always use parentheses () to ensure readability.

```
% set a 0x07
0x07
% expr $a & 0x04
4
% set a [expr $a | 0x08]
15
% set a_neg [expr ~$a + 1]
-15
```

```
bits  7654 3210
      0000 0111  (7)
AND   0000 0100  (4)
      0000 0100  (4)

      0000 0111  (7)
 OR   0000 1000  (8)
      0000 1111  (15)

INV   0000 1111  (15)
      1111 0000  (-16)
    + 0000 0001  (1)
      1111 0001  (-15)
```

read bit 2

set bit 3 to 1

change the sign (2's complement)

NB: equivalent to `set a_neg -$a`

```
% expr "Tcl" == "Tcl"
syntax error in expression "Tcl == Tcl"
% expr {"Tcl" == "Tcl"}
1
% expr {$a > 2.3 ? "over" : "under"}
under
% expr {"TCL" < "Tcl"}
1
% expr {"0x0" == "000"}
1
```

When using strings in expressions:

- enclose strings within "" or {}
- enclose the entire expression in {}

lexicographical comparison (Tcl is AFTER TCL, i.e. it has GREATER index)

Avoid using ==, !=, >, <, >=, <= for string comparison if the strings could resemble numerical values!

```
% expr $a << 2
60
% expr $a >> 1
7
% expr $a_neg >> 1
-8
```

```
0000 1111 ← 0
0001 1110 ← 0
0011 1100  (60)

0000 1111
0000 0111  (7)

1111 0001
1111 1000  (-8)
```

shift LEFT by 2 bits

left shift always inserts 0

shift RIGHT by 1 bit

right shift always propagates the sign bit!

Incrementing and Decrementing

- Can be done using `expr` command...

```
% set a 10
10
% set a [expr $a + 1]
11
% set a [expr $a - 5]
6
```

- ...but Tcl also provides an efficient replacement: `incr`

```
% set a 10
10
% incr a                  ← increment constant is optional (defaults to 1)
11
% incr a -5               ← decrement by using a negative increment constant
6
```

`incr` syntax

```
incr varName ?increment?
```

Basic Procedures

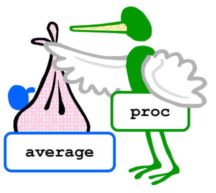
- Use proc command to create new Tcl commands
  - New commands look and behave just like built-in commands

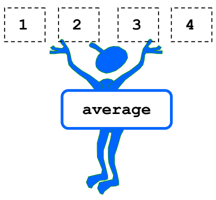
```
proc name arguments body
```

```
% proc average {n1 n2 n3 n4} {
    set sum [expr {($n1+$n2+$n3+$n4)/4.0}]
    return $sum
}
% average 1 2 3 4
2.5
% average -10 10 -50 3
-11.75
```

without return the procedure returns the value returned by **the last executed command**

variables n1, n2, n3, n4 and sum are available only INSIDE the procedure, i.e they are **local variables**





- Why should I use procedures?
  - Store frequently used algorithms (code reuse, productivity gain)
  - Structure your scripts (better readability, easier maintenance)

Global Variables

- Sometimes it is necessary to access variables OUTSIDE the procedure body
- `global` command allows to access variables defined at global scope from inside a procedure body

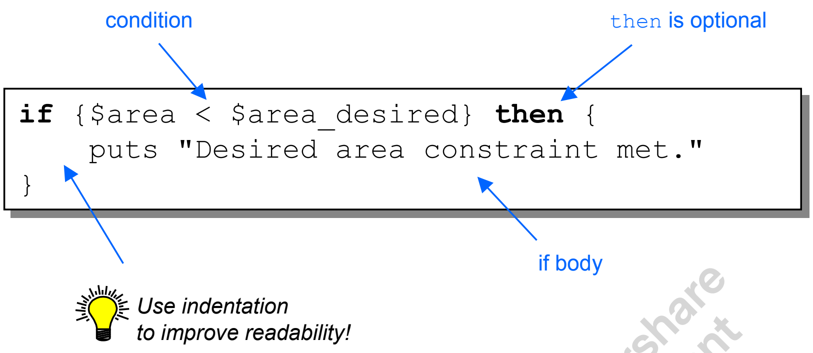
```
% set appname "My script"
My script
% proc print_error {msg} {
    global appname
    puts "$appname: $msg"
}
```

variable `appname` is available globally - both INSIDE and OUTSIDE the procedure.
This is a **global variable**.

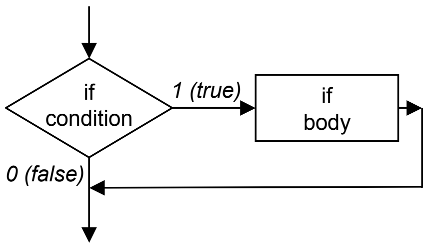
Minimize the use of GLOBAL variables to improve script readability and to ease the maintenance.

Conditional Execution: if

- Execute commands IF the condition is true
  - Condition is evaluated in the same way as `expr` expression
  - Enclose the condition and if command body in `{}` unless you require substitution



```
if {$area < $area_desired} then {
     puts "Desired area constraint met."
}
```



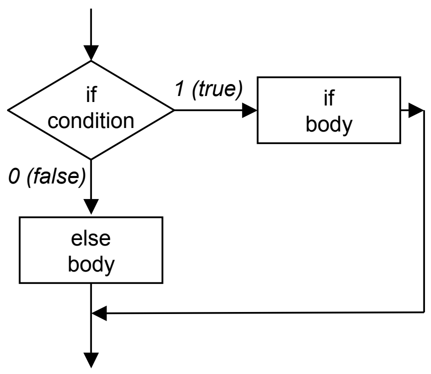
Conditional Execution: if-else

- Execute `if` body commands IF the condition is true, ELSE execute `else` body

```
if {$area < $area_desired} then {
    puts "Desired area constraint met."
} else {
    puts "Area constraint VIOLATED."
}
```

`else` is optional

else body

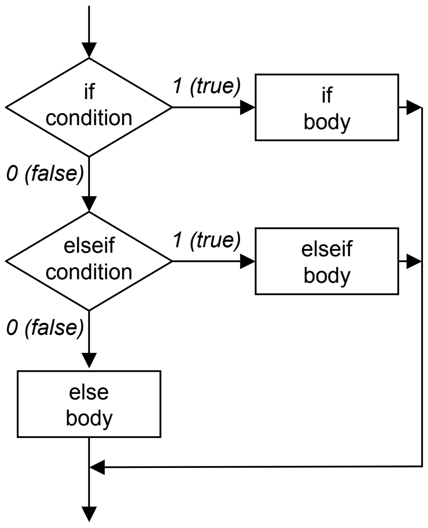


Conditional Execution: if-elseif-else

- Test for more than one condition with `elseif`
  - Any number of `elseif`'s can be used

`then` is optional

```
if {$area < $area_desired} then {
    puts "Desired area constraint met."
} elseif {$area < $area_max} then {
    puts "Maximum area constraint met."
} else {
    puts "Area constraints VIOLATED."
}
```



Complete `if` syntax

```
if expr1 ?then? body1 elseif expr2 ?then? body2 \
elseif ... ?else? ?bodyN?
```

Looping: foreach

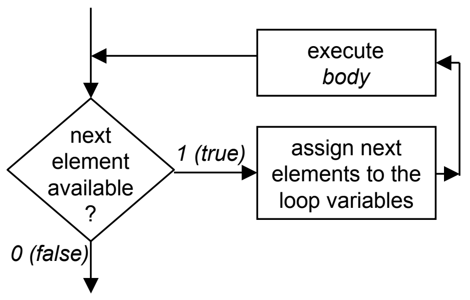
- Use `foreach` to execute loop body for each element in a list
  - Elements are processed from left to right
  - Number of iterations is equal to the number of elements

```
% set lib_cells "INV AND OR"
INV AND OR
```

loop variable name — list of elements to iterate through

```
% foreach cell $lib_cells {
    puts "Found library cell: $cell"
}
Found library cell: INV
Found library cell: AND
Found library cell: OR
%
```

body



Multiple Loop Variables with foreach

- Useful for parsing results returned from other commands

```
% set cell_counts "INV 55 AND 10 OR 20"
INV 55 AND 10 OR 20
```

two loop variables

group loop variables into a single Tcl word

```
% foreach {cell quantity} $cell_counts {
    puts "Found library cell: $cell $quantity times"
}
Found library cell: INV 55 times
Found library cell: AND 10 times
Found library cell: OR 20 times
%
```

`foreach` syntax

```
foreach varName list body
```

or for multiple loop variables

```
foreach varList1 list1 ?varList2 list2 ...? body
```

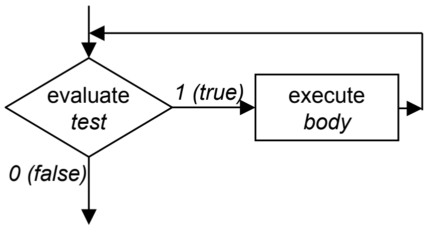
Looping: while

- Use `while` loop to execute commands in a loop which
  - Terminates by meeting a certain condition

```
set number 20
set bits 1
while {(pow(2, $bits) - 1) < $number} {
    incr bits
}
```

test

body



  - Never terminates (indefinite loop)

```
while {1} {
    puts "Tcl forever!"
}
```

`while` syntax

```
while test body
```

Loop Control: break and continue

- Use `break` to terminate the loop
  - Usually used together with conditional commands

```
while {1} {
    puts "Tcl is NOT forever..."
    if {rand() > 0.9} {break}
}
```

when `rand() > 0.9`, terminate the loop

- Use `continue` to advance the loop to the next iteration *immediately*
  - Skips the rest of the current iteration

```
foreach cell $lib_cells {
    if {$cell == "INV"} {continue}
    puts "Found cell with 2+ inputs: $cell"
}
```

skip the loop iteration for `INV` cell (`puts` is not executed for `INV` cell)

`break` and `continue` Tcl commands do not take any arguments.

Control Flow: Syntax Summary

```
if {condition} then {
    body
}

if {condition1} then {
    body1
} elseif {condition2} {
    body2
} elseif {condition3} {
    body3
}
```

Group conditions, tests and bodies with { } unless you require substitution.

```
switch string {
    pattern1 {body1}
    pattern2 {body2}
    pattern3 -
    pattern4 {body4}
    default {default body}
}
```

match `pattern3` OR `pattern4`

if present, `default` must be the LAST pattern

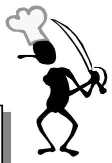
```
for {start} {test} {next} {
    body
}

foreach varName list {
    body
}

while {test} {
    body
}

break

continue
```

`break` — terminate the loop

`continue` — skip the current iteration

Braces & Spaces (Grouping Mistakes)

WRONG



```
if {$a > 0} puts "positive"
```

body NOT grouped!

```
if {$a > 0}{
    puts "positive"
}
```

missing space between } { !

condition NOT grouped!

```
if $a > 0 {puts "positive"}
```

```
if {$a > 0} {
    puts "positive"
}
else {
    puts "negative"
}
```

TWO commands!

RIGHT

```
if {$a > 0} {puts "positive"}

if {$a > 0} {
    puts "positive"
}
```

Use braces and you won't need to worry about spaces!

```
if {$a > 0} {puts "positive"}

if {$a > 0} {
    puts "positive"
} else {
    puts "negative"
}
```

What is a String?

- Any collection of characters
  - Letters, digits, special characters, binary characters, etc.
- In Tcl, everything is a string
  - Universal data type from/to which everything can be converted
  - Easy to manipulate

Constructing Strings

- Strings in Tcl can be constructed in a variety of ways
  - Typing the string characters
  - Substitution and `set`
  - Tcl `append` command

```
% set a "Hello"; set b "world";
world
% set c "$a $b!"
Hello world!
% append d $a " " $b "!"
Hello world!
```

variable `d` will be created if it doesn't exist

- `append` command is optimised for *speed*

```
% set test_files
test/test0.tcl ... test/test9999.tcl
% foreach f $test_files {append test_script "source $f\n"}
% set test_script
source test/test0.tcl
...
source test/test9999.tcl
```

Use `append` when constructing long strings!

`append` syntax

```
append varName ?value value ...?
```

It might seem that there is no justification for having `append`, when the same effect can be achieved using substitution. However, the `append` command is significantly faster when constructing long strings. In the above example, if the body of the `foreach` loop is replaced with

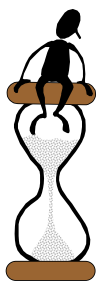
```
set test_script "${test_script}source $f\n"
```

the loop will be **200 times slower**!

Try: `source examples/append_vs_set.tcl`

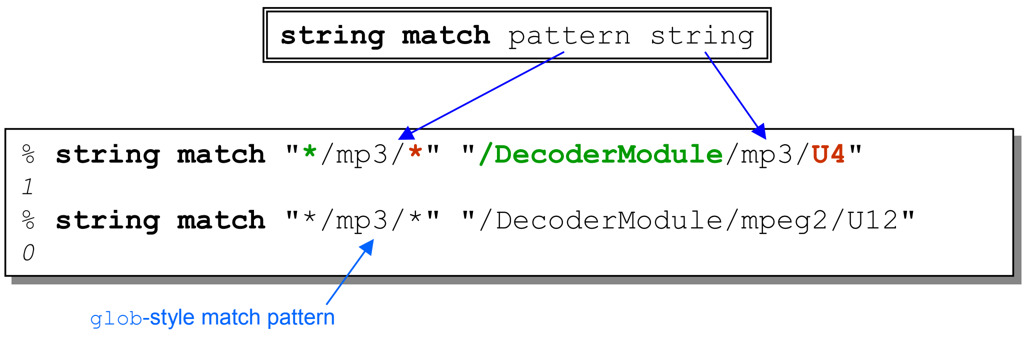
	Execution time	*Relative execution time*
substitution	7.76s	**204**
`append`	38.1ms	1

Tcl 8.0.5 on Pentium III/1GHz PC, 256MB RAM, Windows 2000



String Matching

- Similar to comparison, but allows the first string to contain match patterns



```
string match pattern string
```

```
% string match "*/mp3/*" "/DecoderModule/mp3/U4"
1
% string match "*/mp3/*" "/DecoderModule/mpeg2/U12"
0
```

- Return value:

0	`pattern` does NOT match `string`
1	`pattern` matches `string`

Examples: Character Indices

```
data     =   090FF00FF20001600B7914FF203C899FE
             012345678901234567890123456789012
                       1         2         3
```

end = index of the last character

returns the index of the **first** character in the match

```
% string first "FF2" $data
7
% string last "FF2" $data
22
% string length $data
33
```

```
% string index $data 13
1
% string range $data 10 21
0001600B7914
% string range $data 25 end
03C899FE
```

- Retrieve the data packet between two `FF2` marks

```
% set mark "FF2"
FF2
% set packet [ string range $data \
    [expr [string first $mark $data] + [string length $mark]] \
    [expr [string last $mark $data] - 1] ]
0001600B7914
```

Syntax for the above commands:

```
string first string1 string2
```

```
string last string1 string2
```

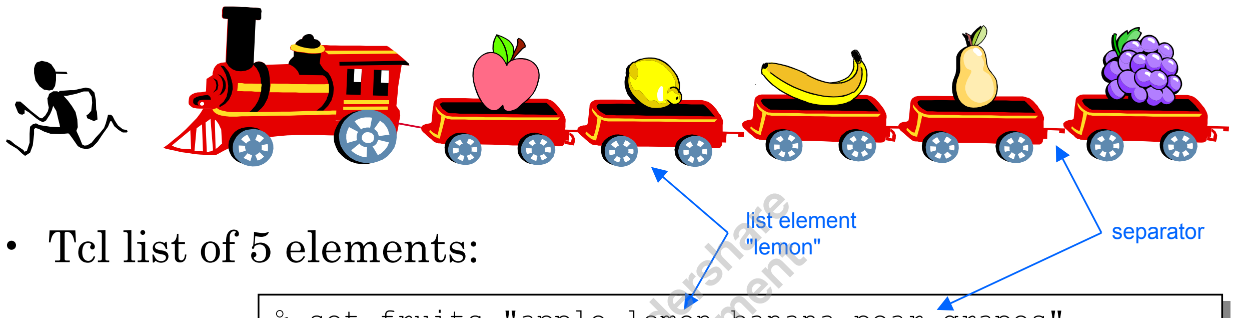
```
string index string charIndex
```

```
string range string firstIndex lastIndex
```

```
string length string
```

What is a List?

- List is a collection of ORDERED elements
  - List elements are strings
    - Can represent anything (string values, other lists, other Tcl data structures, etc.)
  - List elements are separated by whitespaces
    - Spaces, tabs, newlines



- Tcl list of 5 elements:

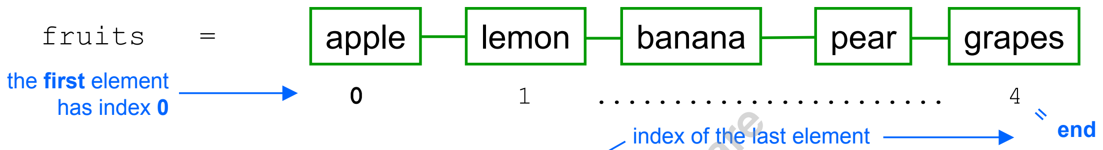
```
% set fruits "apple lemon banana pear grapes"
apple lemon banana pear grapes
```

Manipulating List Elements

- Use `llength` to count the number of elements in a list

```
% llength $fruits
5
```

- Retrieving list elements using indices
  - `lindex` returns an element with a given index
  - `lrange` returns elements within the given index range
  - Both return an empty string if the elements could not be found



```
% lindex $fruits 1
lemon
% lindex $fruits end
grapes
```

```
% lrange $fruits 0 2
apple lemon banana
% lrange $fruits 3 end
pear grapes
```

Syntax for the commands above

```
llength list
```

```
lindex list index
```

```
lrange list firstIndex lastIndex
```

Building Lists (2)

- Use `lappend` to insert new list elements at the END
  - Optimised for *speed*
  - Will create the list variable if it does not exist

name of the list to append
(not `$fruits`!)

one or more elements to append

```
% lappend fruits strawberry orange
apple lemon banana pear grapes strawberry orange
```

- Equivalent to

```
% set fruits "$fruits strawberry orange"
apple lemon banana pear grapes strawberry orange
```

  - Slow, but simple!

Use `lappend` *when building long lists!*

Building Lists (3)

- Join two or more lists together with `concat`
  - Returns a new list
  - Removes one level of list grouping (same effect as with `""`)

```
% set basket "apple pear grapes"
apple pear grapes
% set basket_exotic "lemon banana"
lemon banana
% set fruits [concat $basket $basket_exotic]
apple pear grapes lemon banana                    ← 5 elements
% set fruits [list $basket $basket_exotic]
{apple pear grapes} {lemon banana}                ← 2 elements = lists within a list
```

compare!

  - Eliminates leading and trailing spaces

```
% set fruits [concat $basket_exotic {␣␣orange␣␣}]
lemon banana orange
% set fruits "$basket_exotic {␣␣orange␣␣}"
lemon banana {␣␣orange␣␣}
```

compare!

spaces are removed

`concat` syntax

```
concat ?arg arg ...?
```

Inserting and Replacing List Elements

- Use `linsert` to insert new list elements at ANY position

```
% set fruits "apple lemon banana pear grapes"
apple lemon banana pear grapes
% set fruits [linsert $fruits 1 apricot]
apple apricot lemon banana pear grapes
% set fruits [linsert $fruits 4 cherry plum]
apple apricot lemon banana cherry plum pear grapes
```

insert BEFORE the element with index 1

`linsert` can insert one or more elements

- Use `lreplace` to replace or delete existing list elements

```
% set fruits [lreplace $fruits 0 1 pineapple peach]
pineapple peach lemon banana cherry plum pear grapes
% set fruits [lreplace $fruits 3 end]
pineapple peach lemon
```

replace elements within this index range (from-to)

delete elements within the index range

```
% set fruits [lreplace $fruits 1 1 orange]
pineapple orange lemon
% set fruits [lreplace $fruits 0 0]
orange lemon
```

use IDENTICAL indices if only ONE element is to be replaced (or deleted)

Syntax for the commands above:

```
linsert list index element ?element element ...?
```

```
lreplace list firstIndex lastIndex ?element element ...?
```

Example: Reversing the List

- Tcl does not provide a command which could reverse the order of elements in a list
  - Easy to write using the existing list commands

```
proc lreverse {l} {
    set reversed_l ""
    foreach element $l {
        set reversed_l [linsert $reversed_l 0 $element]
    }
    return $reversed_l
}
```

- Then use as an ordinary Tcl command

```
% set fruits "apple lemon banana pear grapes"
apple lemon banana pear grapes
% set reversed_fruits [lreverse $fruits]
grapes pear banana lemon apple
```

Searching Lists

- Use `lsearch` to search the list for the FIRST element matching the search pattern

```
lsearch ?mode? list pattern
```

- Search mode (`?mode?`) can be

`-exact`	exact matching
`-glob`	glob-style pattern matching, this is the DEFAULT
`-regexp`	regular expression pattern matching (more later)

- Return value

`-1`	`pattern` does NOT match any elements in the `list`
`0-end`	index of the first element matching the `pattern`

Examples: Searching Lists

- Glob-style pattern matching

```
% set cell_list "inv and2 or2 and3 or3 or4"
inv and2 or2 and3 or3 or4
% lsearch $cell_list or*
2
% lsearch -glob $cell_list xor*
-1
```

or2 found at index 2

glob-style search patterns

-1 = not found

optional (-glob is the default)

- Exact matching

```
% lsearch -exact $cell_list or4
5
% lsearch -exact $cell_list and
-1
```

exact strings to be matched

Sorting Lists

- Use `lsort` to sort list elements

returns the list of Tcl commands (UNSORTED)

```
% set sorted_cmds [lsort [info commands]]
after append array auto_execok auto_import auto_load
auto_load_index auto_mkindex auto_mkindex_old auto_qualify
auto_reset binary break case catch cd clock close concat
...
```

SORTED list of commands

- `lsort` is a very powerful sorting command; various features are available through `lsort` options
  - Sort in increasing (default) or decreasing order
  - Sort strings using ASCII (default) or dictionary comparison
  - Sort numbers using integer or real number comparison
  - Sort using a custom comparison command

lsort syntax

```
lsort ?options? list
```

Opening and Closing Files

- Use `open` to open a file for access
  - Returns a channel identifier (also called a file id or descriptor) which is UNIQUE for each opened file

file name   access mode

```
% set fid [open test1.dat w]
file24
```

`file24` is the **channel identifier**

`r`	read only (default)
`r+`	read/write (file must already exist)
`w`	write only
`w+`	read/write
`a`	append only
`a+`	read/append

- After you finish accessing the file, close it with `close`
  - Channel identifier is invalidated after the file was closed

```
% close $fid
```

To improve performance, do not leave files open when they are not being used.

`open` syntax

```
open fileName
```

```
open fileName access
```

```
open fileName access permissions
```

`close` syntax

```
close channelId
```

Writing to Files

- Use `puts` to write to files
  - Returns an *empty string* on success, error message otherwise

```
% puts $fid "Hello world!"
%
% puts -nonewline $fid "Hello world!"
%
```

write `"Hello world!"` + newline into an open file with channel ID `$fid`

write `"Hello world!"` only (no newline)

- Special channels are available for standard terminal I/O
  - `stdout` (standard terminal output), `stderr` (standard terminal error output), `stdin` (standard terminal input)

```
% puts stdout "Hello world!"
Hello world!
% puts "Hello world!"
Hello world!
% puts stderr "Error: invalid input!"
Error: invalid input!
```

write `"Hello world!"` + newline to standard terminal output (`stdout` channel is the default)

write an error message to standard terminal error output

Full `puts` syntax:

```
puts ?-nonewline? ?channelId? string
```

Be warned that stdin/out and stderr exist only when there is a terminal window. On NT stdout and stderr are identical.

Reading Lines

- Use `gets` to read LINES from an open channel
  - There are two alternative forms

```
% set line [gets $fid]
THIS IS LINE 1
% set chars [gets $fid line]
14
```

`gets` **returns the next line read** from an open channel with `$fid` identifier.
(next line = all characters until the end of line)

`gets` stores the next line into variable `line`, and **returns the number of characters read** or `-1` if the end of file was reached

- Typical uses
  - Processing files and communication channel data line-by-line

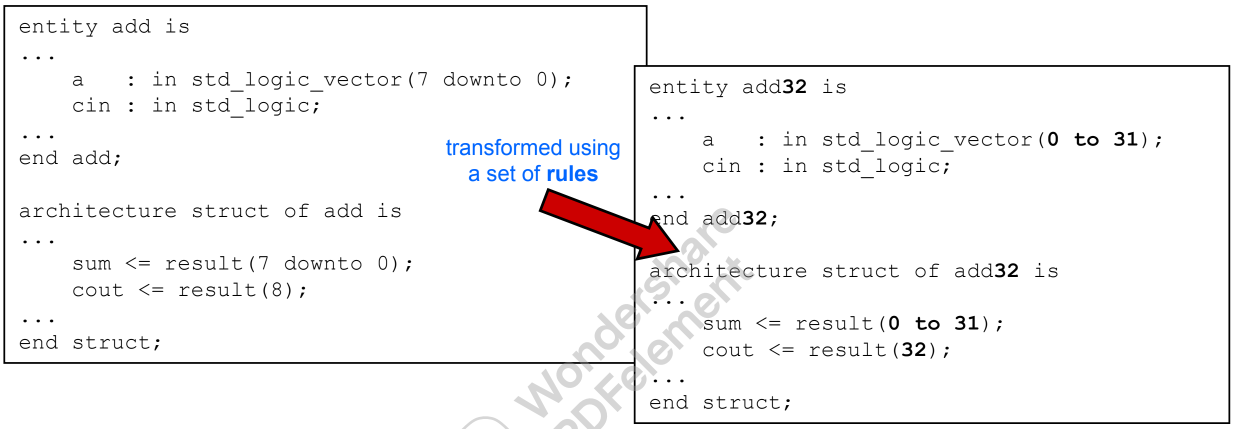
```
set fid [open "test1.dat" r]
while {[gets $fid line] >= 0} {
    puts "test1.dat: $line"
}
close $fid
```

the `while` loop will terminate at the end of file (`gets` will return `-1`)

  - Building INTERACTIVE command-line scripts

Why Regular Expressions?

- String search capabilities already exist in Tcl
  - `lsearch`, `string match`, `switch`, etc.
  - Inefficient for complex string search/replace operations
- Regular Expressions can handle complex and repetitive string manipulation tasks efficiently



```
entity add is
...
    a   : in std_logic_vector(7 downto 0);
    cin : in std_logic;
...
end add;

architecture struct of add is
...
    sum <= result(7 downto 0);
    cout <= result(8);
...
end struct;
```

```
entity add32 is
...
    a   : in std_logic_vector(0 to 31);
    cin : in std_logic;
...
end add32;

architecture struct of add32 is
...
    sum <= result(0 to 31);
    cout <= result(32);
...
end struct;
```

What are Regular Expressions?

- Special string patterns which can match strings using various rules
  - Context-specific
  - Generic (will work for many different strings)
- Example
  - RE for an output port declaration in a VHDL entity:

regular expression →

```
*[a-zA-Z][a-zA-Z0-9_]* *: *out [a-zA-Z][a-zA-Z0-9_]*
```

```
entity add is
port (
    cin : in std_logic;
    a   : in std_logic_vector(7 downto 0);
    b   : in std_logic_vector(7 downto 0);
    y   : out std_logic_vector(7 downto 0);
    cout: out std_logic
);
end add;
```

string matching the regular expression

NOTE: RE string matching is based on **context** (e.g. signal which is a 1-bit output port). Therefore the matching will work for signals of any name or type.

Regular Expression Basics

- Alphabet and digit characters are matched as usual

`a`	matches a SINGLE given character, i.e. character `a`
`VHDL`	matches a SEQUENCE of given characters, i.e. string `VHDL`

- Regular expressions use several special characters

`.`	matches ANY SINGLE character
`[]`	matches a SINGLE character from a sequence, e.g.

`[abc]`	`a` or `b` or `c` (single character)
`[A-Z]`	an uppercase letter (character range)
`[^A-Z]`	a character which is NOT an uppercase letter
`[a-zA-Z0-9_]`	a character which is either a letter (lowercase or uppercase), a digit or an underscore

`*`	matches 0 or more occurrences of a preceding ATOM

`a*`	0 or more characters `a`, e.g `a` or `aaaaa`, but also `''` (no character)
`[a-z]*`	0 or more lowercase letters
`[A-Z][a-z]*`	word with the first letter in uppercase (0 or more lowercase letters)
`.*`	0 or more occurrences of any character, i.e. ALL characters

Searching with Regular Expressions

- Use `regexp` command for RE-based string search

regular expression pattern — string to search — variable which will store the matched string

```
% regexp {[A-Z][A-Z]*} "which is better: VHDL or Verilog?" m_var
1
% set m_var
VHDL
```

Enclose RE patterns within {} to protect patterns from Tcl substitution (unless the substitution is desired)

  - `regexp` returns `1` if the match was found, `0` otherwise

- Use `-nocase` option for case-insensitive search

```
% regexp -nocase -- {[A-Z][A-Z]*} "which is better: ..." m_var
1
% set m_var
which
```

Always use `--` to prevent `regexp` from confusing patterns with options.

- RE patterns always match the LONGEST possible string of characters