# TCL

## Tool Command Language

Scripting Language

# TCL

## Tool Command Language

Scripting Language

# Why TCL?

Why not Python ? Perl ?

:Single Reason:

# Part of almost every EDA tool

# Why part of every EDA tool?

## General Reasons

○ Rapid Application Development

○ Portability

○ Availability

○ Available Extensions

## Specific Reasons

○ Easy to learn (limited set of rules)

○ Embeddable

○ Interfacing between tools

○ Extendable (Write your own commands)

# What to learn

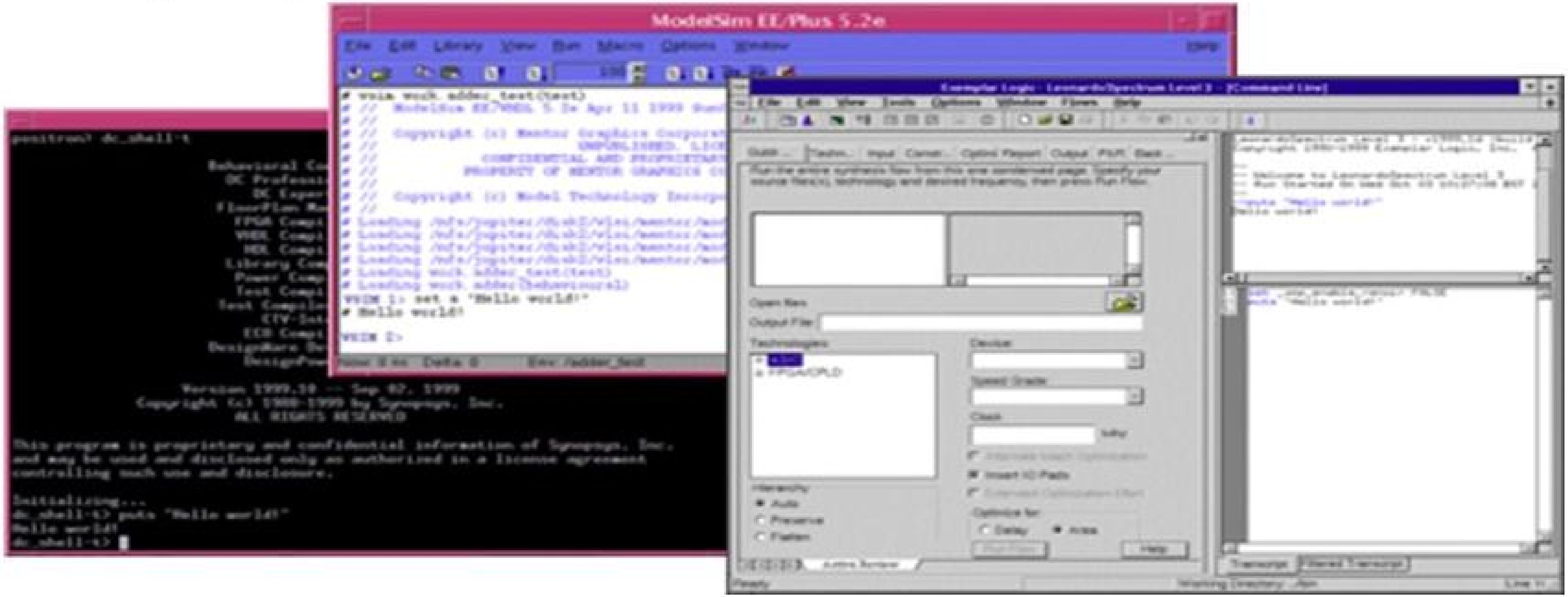
- TCL Strengths
- Understand underlying tenets of TCL
- Be able to understand any TCL script
- Learn Writing Simple TCL scripts
- Demonstrate how to use TCL in EDA tools

# TCL in EDA

- Quickly becoming the standard VLSI scripting language
- Typical uses
  - simulation, synthesis and test automation scripting
  - data analysis and visualixation
  - design flow integration
  - netlist conversion, analysis and hacking
  - linking incompatible tools running on different platforms
  - scripting front-ends for command-line based applications
  - IP core customixation scripting
  - automated test benches, regression testing, HW/SW co-verification
  - portable system demonstrators/applications
  - project/EDA system administration (installation, backup, etc.)

# TCL Interpreter in SoC Design Tools

- TCL interpreter is typically embedded in your SoC Design tool command console (GUI or command line)
  - Try invoking `info commands` in your favorite SoC Design tool!



```
positron? dc_shell-t

                          Behavioral Co
                            DC Professi
                              DC Expert
                         FloorPlan Ma
                             FPGA Compi
                             VHDL Compi
                              HDL Compi
                          Library Com
                             Power Comp
                              Test Compi
                          Test Compile
                                CTV-Int
                               ECO Compi
                          DesignWare De
                              DesignPow

              Version 1999.10 -- Sep 02, 1999
         Copyright (c) 1988-1999 by Synopsys, Inc.
                   ALL RIGHTS RESERVED

This program is proprietary and confidential information of Synopsys, Inc.
and may be used and disclosed only as authorized in a license agreement
controlling such use and disclosure.

Initializing...
dc_shell-t> puts "Hello world!"
Hello world!
dc_shell-t>
```

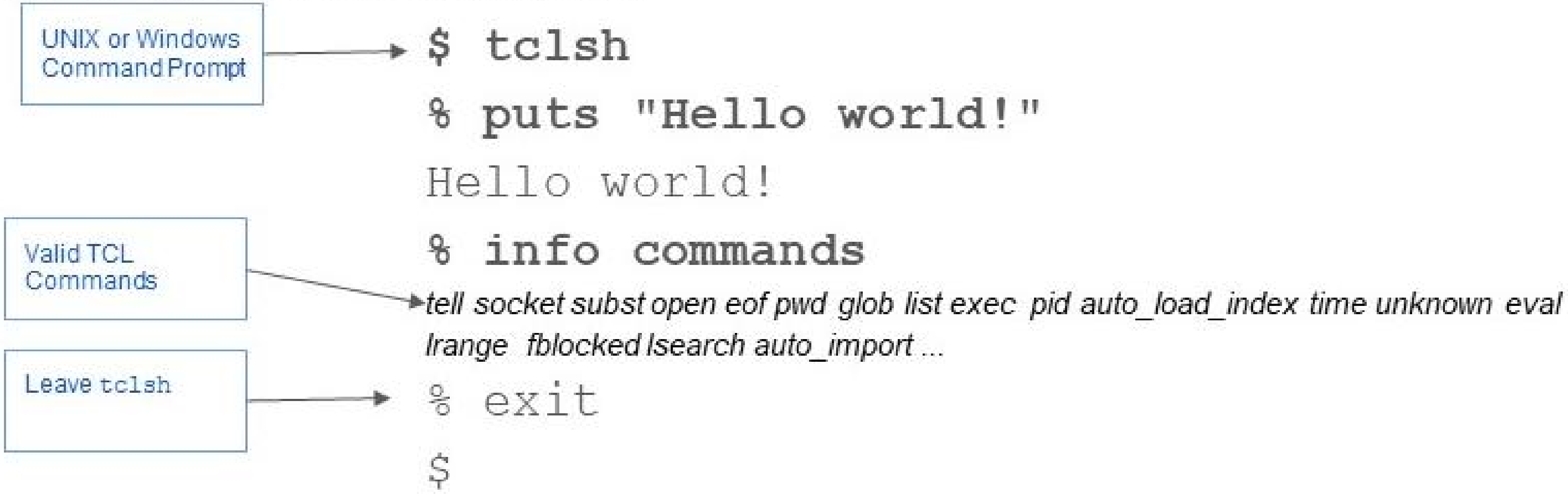
```
# vsim work.adder_test(test)
# //  ModelSim EE/VHDL 5.2e Apr 11 1999 Sun
# //
# //  Copyright (c) Mentor Graphics Corporat
# //                              UNPUBLISHED, LICE
# //                     CONFIDENTIAL AND PROPRIETAR
# //                  PROPERTY OF MENTOR GRAPHICS CO
# //
# //  Copyright (c) Model Technology Incorpo
# //
# Loading /nfs/jupiter/disk2/vlsi/mentor/mod
# Loading /nfs/jupiter/disk2/vlsi/mentor/mod
# Loading /nfs/jupiter/disk2/vlsi/mentor/mod
# Loading /nfs/jupiter/disk2/vlsi/mentor/mod
# Loading work.adder_test(test)
# Loading work.adder(behavioural)
VSIM 1> set a "Hello world!"
# Hello world!

VSIM 2>
```

```
-- puts "Hello world!"
Hello world!
```

# TCL Shell (`tclsh`)

- Command-line interface
  - Interactive incremental testing (try & see)
- Available within many modern SoC Design tools
- Works on Windows/Unix/MacOS



UNIX or Windows Command Prompt

Valid TCL Commands

Leave tclsh

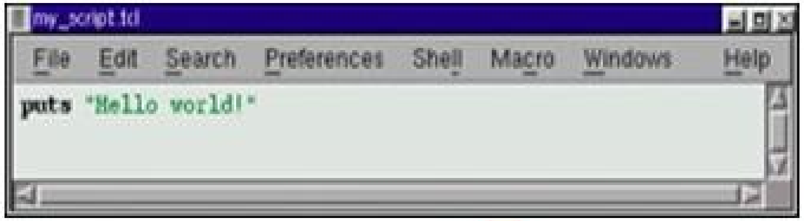
```
$ tclsh
% puts "Hello world!"
Hello world!
% info commands
tell socket subst open eof pwd glob list exec pid auto_load_index time unknown eval
lrange fblocked lsearch auto_import ...
% exit
$
```

# Working with TCL scripts (UNIX)

- Interactive & iterative process

## 1. EDIT

in a file



```
puts "Hello world!"
```

## 2. TEST

From a UNIX Shell

```
$ tclsh
my_script.tcl
Hello world!
$ !!
Hello world!
...
```

OR

From TCL interpreter

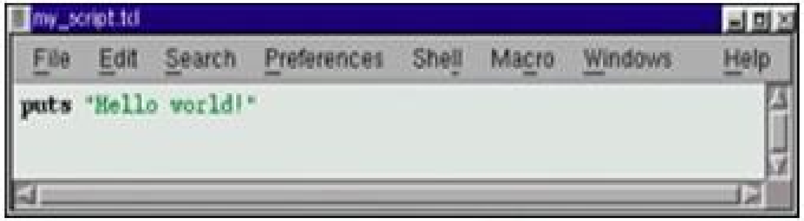
```
% source my_script.tcl
Hello world!
% !!
Hello world!
...
% exit
```

# Working with TCL scripts (Windows)

- Interactive & iterative process

## 1. EDIT

in a file



```
puts "Hello world!"
```

## 2. TEST

From Command/MSDOS Prompt

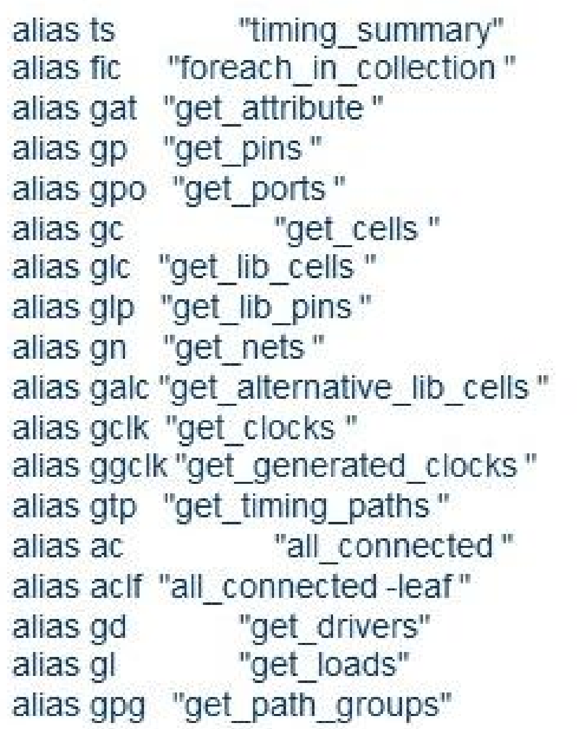
```
C:\> tclsh my_script.tcl
Hello world!
C:\> tclsh my_script.tcl
Hello world!
...
```

OR

From TCL interpreter

```
% source my_script.tcl
Hello world!
% !!
Hello world!
...
% exit
```

# Some Actual Code (IC Compiler Tool)



```
alias ts          "timing_summary"
alias fic     "foreach_in_collection "
alias gat   "get_attribute "
alias gp    "get_pins "
alias gpo   "get_ports "
alias gc             "get_cells "
alias glc   "get_lib_cells "
alias glp   "get_lib_pins "
alias gn    "get_nets "
alias galc "get_alternative_lib_cells "
alias gclk  "get_clocks "
alias ggclk "get_generated_clocks "
alias gtp   "get_timing_paths "
alias ac             "all_connected "
alias aclf  "all_connected -leaf "
alias gd          "get_drivers"
alias gl          "get_loads"
alias gpg   "get_path_groups"
```

# Continuing ...

```
proc gpl {var} {
                foreach_in_collection  pin [get_pins $var] {
                puts "[gon $pin]"
                }
}
proc gpol {var} {
                foreach_in_collection  port [get_ports $var] {
                puts "[gon $port]"
                }
}
proc gcl {var} {
                foreach_in_collection  cell [get_cells $var] {
                puts "[gon $cell]"
                }
}
proc glcl {var} {
                foreach_in_collection  lib_cell [get_lib_cells $var] {
                puts "[gon $lib_cell]"
                }
}
```

# Continuing ...

```
proc glpl {var} {
            foreach_in_collection  lib_pin [get_lib_pins $var] {
            puts "[gon $lib_pin]"
            }
}
proc gnl {var} {
            foreach_in_collection  net [get_nets $var] {
            puts "[gon $net]"
            }
}
proc galcl {var} {
            foreach_in_collection  alternative_lib_cell [get_alternative_lib_cells $var] {
            puts "[gon $alternative_lib_cell]"
            }
}
proc gclkl {var} {
            foreach_in_collection  clock [get_clocks $var] {
            puts "[gon $clock]"
            }
}
```

# Continuing ... with another sample

```
#--------------------------------------------------------------------------------------#
#-- report_slack_distribution
#--------------------------------------------------------------------------------------#
  proc report_slack_distribution { targetClock {marginPoints "0 .5 1 1.5"  } {ListEndPoints 0}}  {
   # Here is a sample report
  #
  #  Timing Information for clock clk with 128 total endpoints:
  #
  #                 Worst Negative Slack:             -1.39365
  #
  #  margin   Num.  of Violators  Total Neg. Slack   percent violators
  #  ------   ---------------   -------------   ---------------
  #  0               16                   -17.9794         13
  #  .5              24                   -27.77           19
  #  1               24                   -39.7698         19
  #  1.5             28                   -53.1804         22
  #
```

# Continuing ...

```
# find all the endpoints clocked by targetClock
  set allRegisterPins [all_registers -clock $targetClock -data]
  set allOutputPorts [all_outputs -clock $targetClock]
  set allEndPoints [concat $allRegisterPins $allOutputPorts]
  foreach_in_collection endPoint $allEndPoints {
                  # get and print the startpoint, arrival and slack
                  set maxTimingSelection  [get_timing_path -nworst $pathsPerEndpoint -to $endPoint -delay max]
                  set firstFor 1
                  foreach_in_collection path $maxTimingSelection {
                  set TE [expr $TE + 1]
                  set slack [get_attribute $path slack]
                  set startpoint [get_attribute $path startpoint]
                  if { $firstFor == 1 } {
                  if { $slack < $WNS } {
                  set WNS  $slack
                  }
```

# Continuing ...

```
  echo ""
  echo "Timing Information for clock $targetClock with $TE total endpoints:"
  echo ""
  echo [format "   Worst Negative Slack:          %-20.2f"  $WNS]
  echo ""
  echo " margin   Num. of Violators  Total Neg. Slack    percent violators"
  echo " ------   ---------------  -------------   ---------------"
  foreach margin $marginPoints {
                 echo [format "%6.2f %13d       %14.2f            %5.1f " \
                 $margin $NVE($margin) $TNS($margin) [expr 100-(100*($TE-$NVE($margin))/$TE) ]]
  }
}
```

# TCL : String Based Command Language

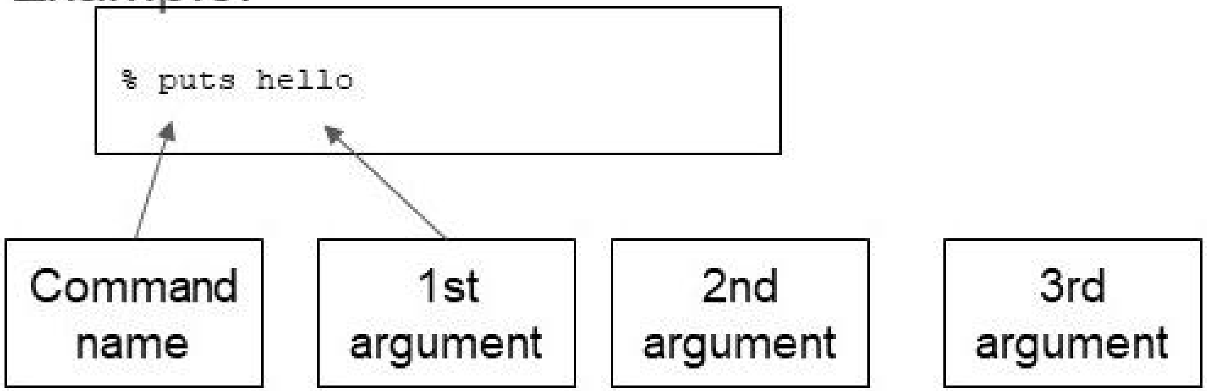
STRING
????

- Very few fundamental constructs
- Very little syntax
- Basic mechanism
  - Related to Strings
  - String substitution
- Just keep in mind: TCL is CASE SENSITIVE

# First thing to understand : COMMAND

**TCL script is composed of commands.**

- Command is most basic unit -> Composed of words
- Be VERY clear what a command is
- Words are separated by space (mostly)
- Example:



```
% puts hello
```

Command name

1st argument

2nd argument

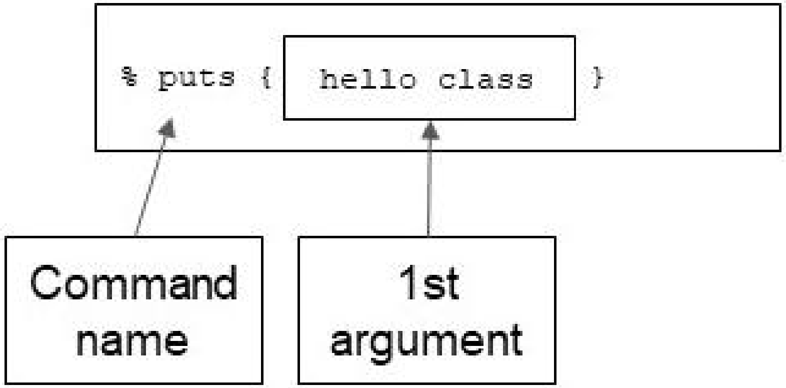
3rd argument

# Continued: Commands

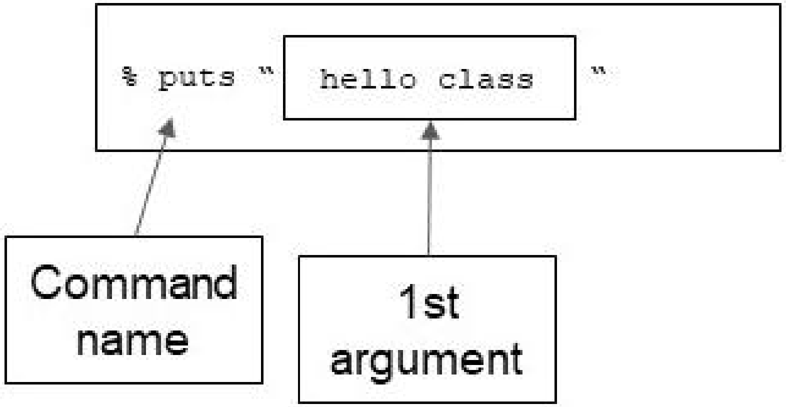
- Command name => Action
- Every command specifies it's own rules
  - Number of arguments
  - Order of arguments
  - Type of argument
- Will give error if requirements are not followed
- `% puts hello tcl class`

# Word Grouping

- Space may not be separating arguments always.
- Words can be grouped as one argument using
    - Braces {}
    - Double Quotes “”
- Examples:



```
% puts { hello class }
```

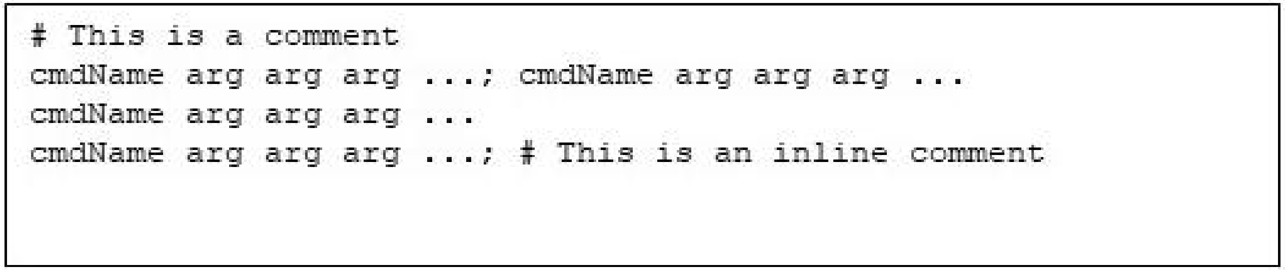


```
% puts " hello class "
```

- There is more to it, then what we see here, but that’s for later.

# Some more things about COMMAND

- Command is/can be terminated by ‘;’ or new line
- Comments start with ‘#’
- Examples :



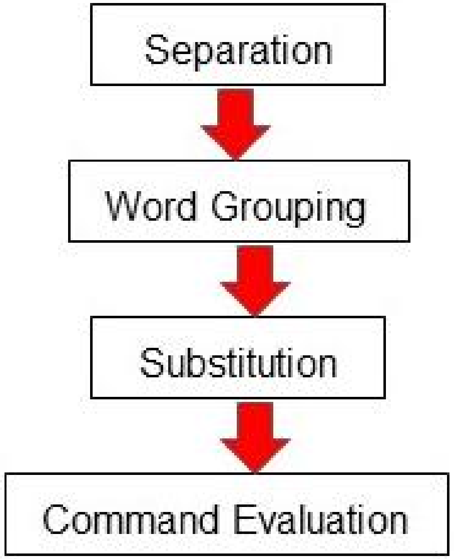
```
# This is a comment
cmdName arg arg arg ...; cmdName arg arg arg ...
cmdName arg arg arg ...
cmdName arg arg arg ...; # This is an inline comment
```

- Not getting into more technicalities of it

# Who takes care of all these?

**TCL Interpreter**

- It follows following order :



Separation

Word Grouping

Substitution

Command Evaluation

- Very important to understand each of these.

# More about TCL Interpreter

- After initial processing by Interpreter, command is executed.
- Treats every argument as string
  - Does not try to see it as number
  - Always a plain string
- Interpreter Errors / Command Errors
  - Are different

# Variable

- Basic construct of any programming language
- Container in MEMORY to hold a value
- Has a NAME
- Can READ value stored in variable
- Can WRITE value to be stored in variable

# Variable in TCL

- Variable Name
  - Case sensitive
  - Can be composed of any characters (But use only _, digits, letters)
- To store value in variable
  - Use `set` command
  - `% set a 123`
  - `% set name tcl`
- To Read Variable
  - Again use `set` command
  - `% set a`
  - `% set name`

# Continued: Variables

- `set` Command
  - Accepts one or two arguments
  - Every argument is string
  - If one argument, then
    1. variable has to be already defined, else error
  - If two argument, then
    1. It can create new variable and assign value
    2. If already defined, then assigns new value
  - Same command for any type of variable

# Substitution

- Different types of substitutions
    - Variable
    - Backslash
    - Command
- Basic flow:
    - All words are searched from left to right (character by character)
    - Searched for special constructs which trigger substitution
    - Each character only processed ONCE

# Continued: Variable Substitution

- How to use variables in a command
  - `$varName`
- It will be replaced with its value by TCL interpreter
  - `% set d 10`
  - `% set b $d   => % set b 10`
  - `% set b`
- No substitution inside braces
  - `% set b {$d}`

# Continued: Variable Substitution

- Substitution inside quotes
  - `% set d 10`
  - `% puts “$d seconds“`
- If no space after or before `$varName`
  - `% set d 10`
  - `% puts “$dns”`
  - `% puts “${d}ns”`

# Backslash Substitution

- What if you want to print “I have $10”
    - Use special characters
    - `% puts "I have $10"`
    - `% puts "I have \$10"`
- Backslash can be used for special characters
- Some other examples:
    - \n : New line, \t : Tab
    - `% puts "I am spread \n on two lines"`
    - `% puts "Name \t Age"`
    - `% puts "Ram \t 10"`

# Command Substitution []

- Each occurrence of `[<commands>]` is
  - Replaced with output of that command
  - `% set d 10`
  - `% puts [set d]`
- Again no subsitution in braces
  - `% puts {[set d]}`
- Substitution allowed in Quotes
  - `% puts "I have [set d]\$"`
- This also can be escaped
  - `% puts \[set d\]`

# Command Evaluation

- Finally actual command is evaluated
  - Interpreter searches for matching TCL command
- Tcl Interpreter passes list of arguments to command
  - Command can interpret strings as numbers too now
- It returns results after execution to interpreter
- Commands can be:
  - built in eg. `put, set`
  - user defined

# Learn one more command

expr

- To process numbers (math expressions)

  ```
  % expr 10 + 5
  % set d 5
  % expr $d * 2
  % set a 10; set b 20
  % expr $a + $b
  ```

- `expr` interprets strings as numbers and not Tcl Interpreter

# Maths : Expression Evaluation

expr

- Operators ?? Operands ??
- What is order of evaluation?
  - Based on precedence
  - `% set a 10; set b 20; set c 5`
  - `% expr $a + $b * $c`
- Difference in below two
  - `% expr $a + $b`
  - `% expr {$a + $b}`
- `% expr 9 / 2  ; # ?? Why ??`

# More on Variables

- Different number formats
  - `% set reg 0173 ;# Octal`
  - `% set reg1 0x7b ;# Hexa Decimal`
  - `% set match_found 1 ;# boolean`
- Tcl Interpreter still sees these as Strings
- But expr will read it as numbers
- How about this?
  - `% set !£%^&* "bad idea!"`
  - `% set !£%^&*`
  - `% set a$b`

# Word grouping examples

- Tell me RIGHT or WRONG
    - `% puts "Hello { world! }} "`
    - `% puts "Hello " world! "" "`
    - `% puts "Hello \" world! \" "`
    - `% puts "Hello "world!" "`
    - `% puts {Hello { world! }} }`
    - `% puts {Hello " world! "" }`
    - `% puts {Hello }world!{ }`

# New Lines

- Inserting New Lines

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

  -
    ```
    % puts "Line 1\nLine 2"
    ```
- Avoiding New Lines

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

# Nested Command Substitution

- Nesting of commands allowed
  - `% set a “eggs”`
  - `% puts "Two nested [set b [set a]]"`

- How about this ?
  - `% puts [set b "No [set a "escapes here"]"]`
  - `% set a 10`
  - `% set b "{$a}\{ #[set c 14]"`
  - `% puts "$b"`

# More on operators

- Arithmetic Operators : `+ - * / %`
- Relational Operators : `<= >= < >`
- Logical Operators : `&& ||`
- Bitwise Operators : `& | ^ << >>`
- Ternary Operator : `a ? b : c`

# Boolean Values

- True/False
  - zero <=> False
  - Non zero <=> True  (output is 1)
  - String can also represent true/false

- Examples

  ```
  % set a 10
  % expr $a && 1
  % expr $a && 0
  % set b false
  % expr $b || 1
  % set c true
  % expr 0 || $c
  ```

# Example : TCL operators

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

- What is happening here?
    - % set a 0x07   ; # Binary: 0000 0111
    - % expr $a & 0x04 ; # ??
    - % set a [expr $a | 0x08] ; # ??
    - % set a_neg [expr ~$a + 1] ; # 2’s complement ?

# Example : TCL Shift operators

- Important to play around with bits
    - `% set a 0`
    - `% expr $a << 2`
    - `% set a 0xf`
    - `% expr $a >> 1`
    - `% set a_neg`
    - `% expr $a_neg >> 1`
- Get 4th bit of a number
    - Use shift operator
    - Use bitwise operators “&” and “ |”

# Continued: TCL operators

- Arithmetic Operators on
    - Integers
    - Reals
- Relational Operators on
    - Integers
    - Reals
    - Strings
- Logical Operators on
    - Integers
    - Reals
- Bitwise Operators on
    - Integers

# Increment / Decrement

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

- More efficient way
  - `% incr a`
  - `% incr a -5`
  - `% incr a - 5 # ??`

# Substitution time again

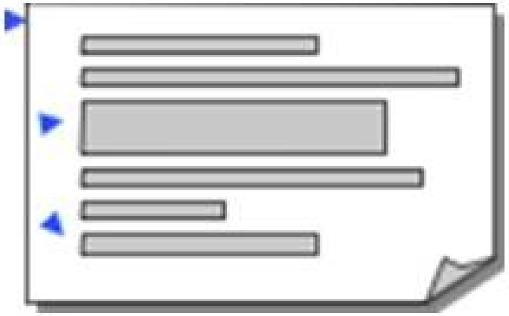
- Quotes enable substitution
  - $ \ [
  - Use this for grouping if substitution required
- Braces disable substitution
  - No substitution of any kind
  - Useful to defere substitution untill later (to be done by command)
  - Better to use if no substitution required

# More dose of substitution

- Check this
  - `% set cost [expr $a*0.1 + $b*0.6 + $c*0.3]`
  - `% set a 100; set b 200; set c 300`
  - `% expr $cost`
  - `% expr {$cost}`

# Writing in a script file

- Open any text editor
  - Write series of commands
  - Save (test.tcl)
- From TCL Shell (interpreter)
  - `% source test.tcl`
- From Linux Shell
  - `$ tclsh test.tcl`
- From Windows Shelll
  - Need to check what binary it is in path
  - Then
    - C:\> <tcl binary> test.tcl
  - NOTE: prefer it running from tcl shell



It sequentially runs each command, one after another.

# Writing a simple script in file

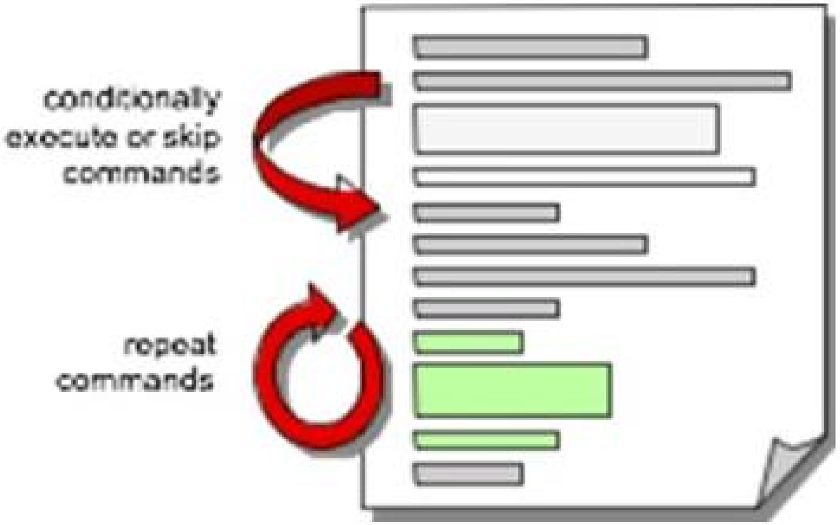
- Open test.tcl, paste following

```
#!/usr/local/bin/tclsh
puts “Hello, I am running script using file”
set num1 10
set num2 20
# This is to demonstrate sum of two numbers
puts “sum of these two numbers is [expr $num1
+ $num2]”
```

- On Linux:
  - `$ chmod 755 test.tcl`
  - `$ ./test.tcl`
- On Windows:
  - Prefer running in tcl interpreter
  - `% source test.tcl`

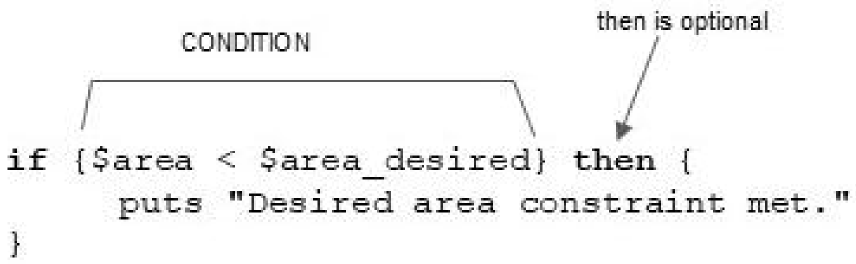
# Control Flow of Programs

- A collection of TCL commands which can be used to control when and how many times commands are executed
- Conditional command execution
  - if
  - switch
- Looping commands
  - for
  - foreach
  - while
- Loop control
  - break
  - continue



# Conditional Execution: if

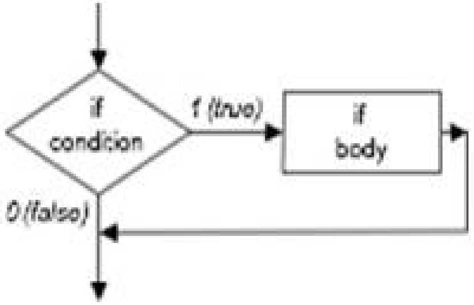
- Execute commands IF the condition is true
  - Condition is evaluated in the same way as **expr expression**
  - Enclose the condition and if command body in {} *unless you require substitution*



CONDITION

then is optional

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



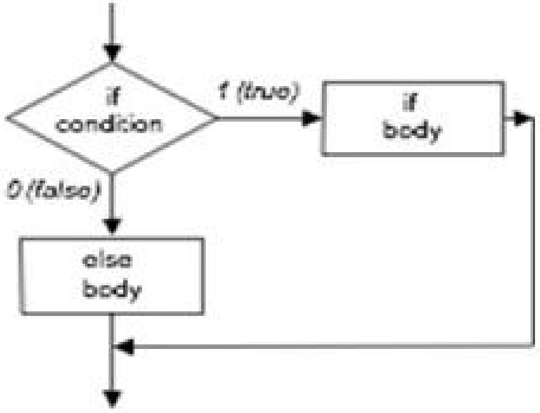
Note: This is also a command only

# 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."
}
```

se is optional

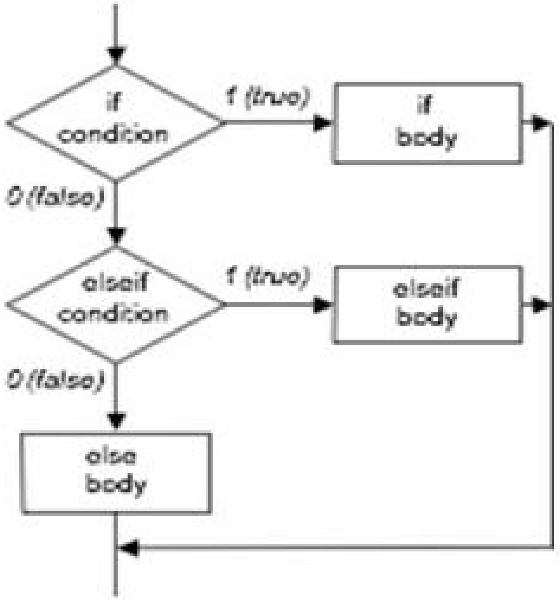


Note: These all are commands

# Conditional Execution: if/elseif/else

- Similarly
  - Test for more than one condition with elseif
  - Any number of elseif's can be used
  - elseif is not optional

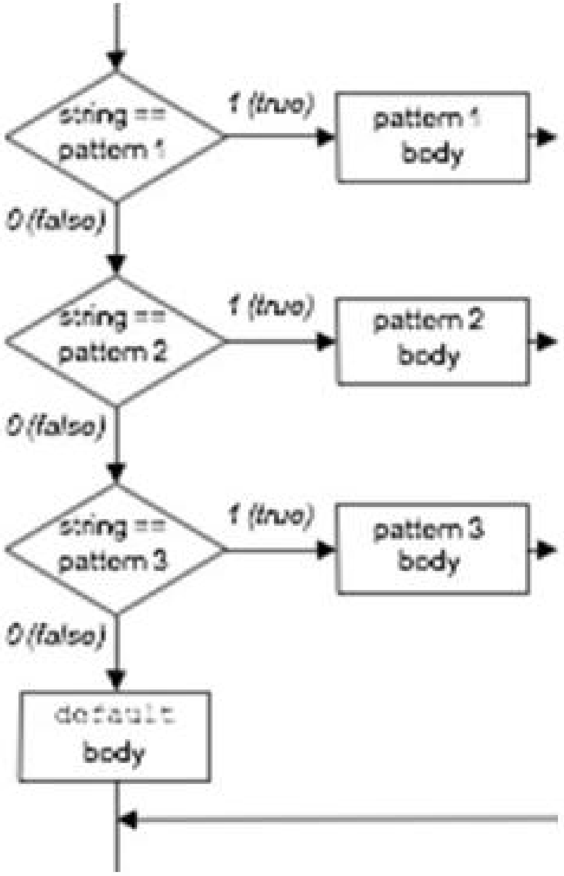
```
if {$area < $area_desired} then {
     puts "Desired area constraint met."
} elseif {$area < $area_max} then {
     puts "Maximum area constraint met."
} else {
     puts "Area constraints VIOLATED."
}
```



# Switch

- Switch
  - String Pattern is matched with options
  - -- is optional
  - default pattern matches all strings (should be last)

```
switch -- $cellName {
  AND {incr and_count}
  OR       {incr or_count}
  INV {incr inv_count}
  default {puts "Unrecognized cell."}
}
```

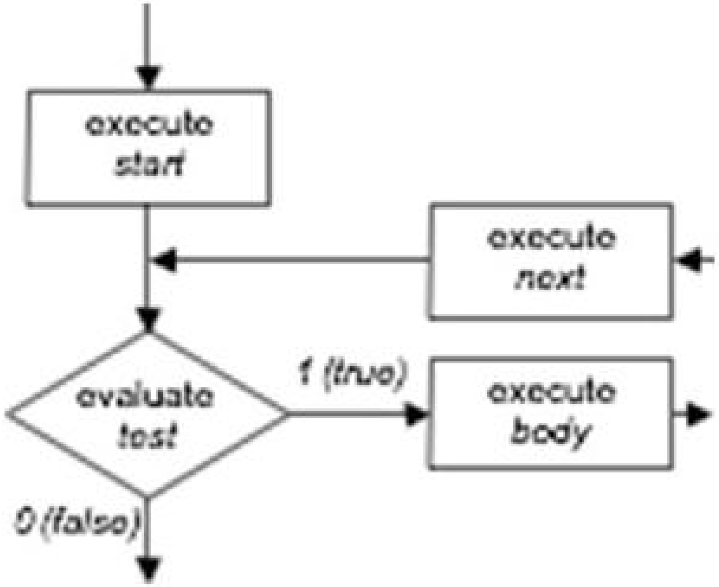


# Looping: For

- Use for to execute commands specified number of times
  - ○ `'test'` : evaluated same way as `'expr'`

```
set a 9; set b ""

       start      test       next
for {set i 0} {$i < 4} {incr i} {
      set bit [expr ($a >> $i) & 0x1]
      set b "$bit$b"
}
```



# Let us write some simple scripts

- Calculate area of rectangle (given length and breadth)
- Declare temperature in °Celsius, then convert it into °Fahrenheit
- Check if a given variable is divisible by 5 and 11 or not.
- Script to sum all numbers
- Script to write table for given number
- Script to find even or odd
- Script to find num is +ve –ve or 0

# Syntax Summary

```
if {condition} then {  body

}
```

```
if {condition1} then {  body1

} elseif {condition2} {  body2

} elseif {condition3} {  body3

}
```

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

}
```

# Right / Wrong ??

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

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

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

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

# Right / Wrong ??

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

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

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

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

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

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

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

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

# STRINGS

# What is String

Collection of characters

EVERYTHING in TCL is STRING

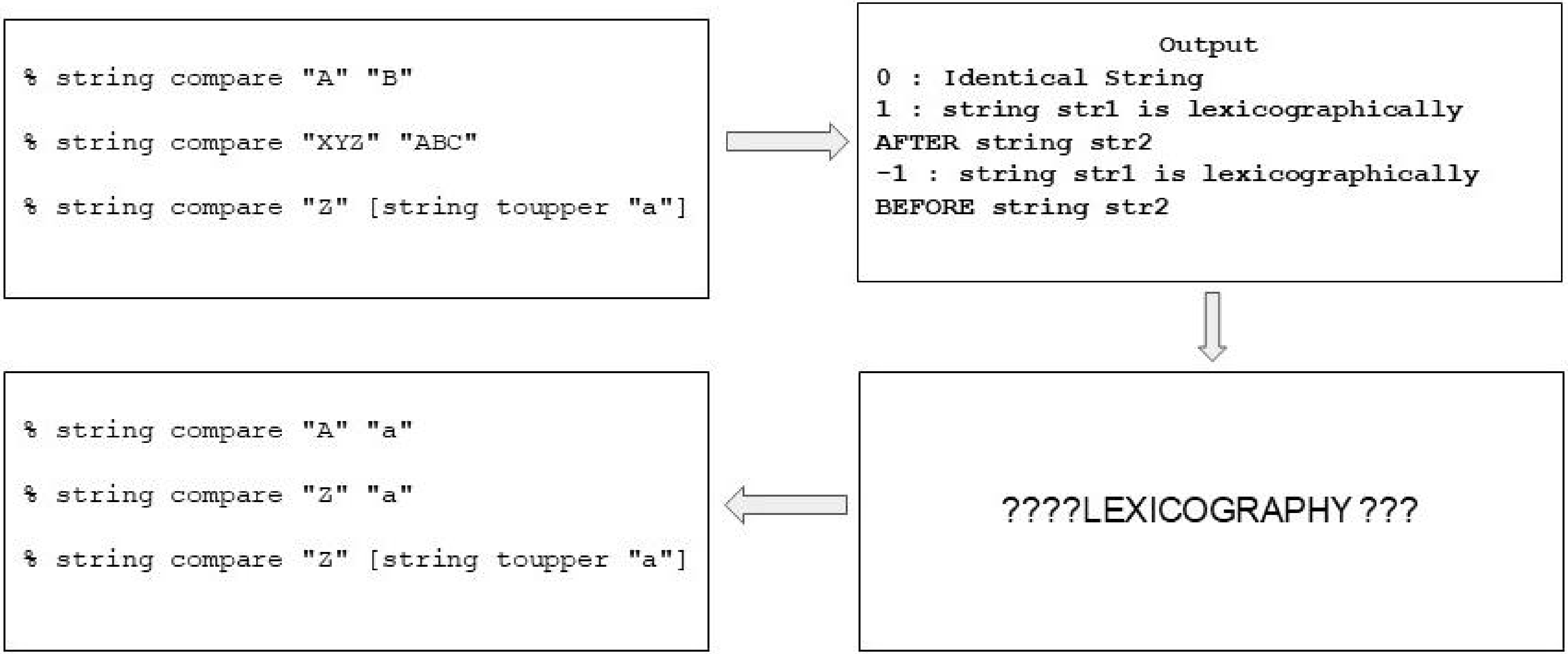
Why ?

Easy to manipulate
Universal data type : can be converted to/from easily

# What all I want to do with STRING?

- Concatenate
- Search
- Compare / Match
- Find character at each index
- Format String
- Convert to upper/lower, trim left/right
- Read values from string (scan)

# Compare two Strings ?



```
% string compare "A" "B"

% string compare "XYZ" "ABC"

% string compare "Z" [string toupper "a"]
```

```
                 Output
0 : Identical String
1 : string str1 is lexicographically
AFTER string str2
-1 : string str1 is lexicographically
BEFORE string str2
```

????LEXICOGRAPHY ???

```
% string compare "A" "a"

% string compare "Z" "a"

% string compare "Z" [string toupper "a"]
```

# Some useful string commands

```
% string toupper "Vhdl Edif TCL"

% string tolower "DECODER.VHD"

% string trim "    Area: 2345

“

% string trimleft “   Left Trim

”

% string trimright “

Right Trim   ”

% string length “length of this string”
```

# Constructing new Strings

```
% set a “string one”

% set b “string two”

% set c “$a $b!”
```

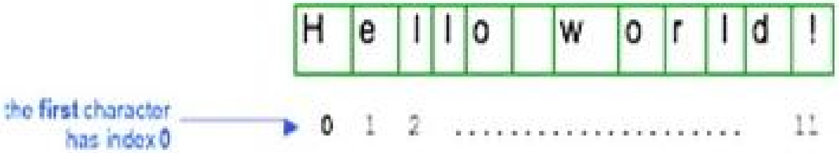
But this is faster way to build big strings:

```
% append d $a " " $b "!"

% # For example, building big string in a loop

% for {set i 0} {$i < 10} {incr i} {
           append d $d
 }
```

# Using Character Indices



```
% string index "Hello World!" 0

% set data "hello world!"

% string range $data 1 4

% string range $data 8 end

% string range $data 25 end
```

# Using Character Indices : Search for String

```
090FF00FF20001600B7914FF203C899FE
```

```
012345678901234567890123456789012
```

```
% set data “090FF00FF20001600B7914FF203C899FE”

% string first “FF2” $data

% string last "FF2" $data

% set mark "FF2"

% set packet [ string range $data \

[expr [string first $mark $data] + [string length $mark]] \

[expr [string last $mark $data] - 1] ]
```

# String match : Not same as compare

**string match** pattern string

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

Return Value

0 : **pattern** does NOT match **string**

1 : **pattern** matches **string**

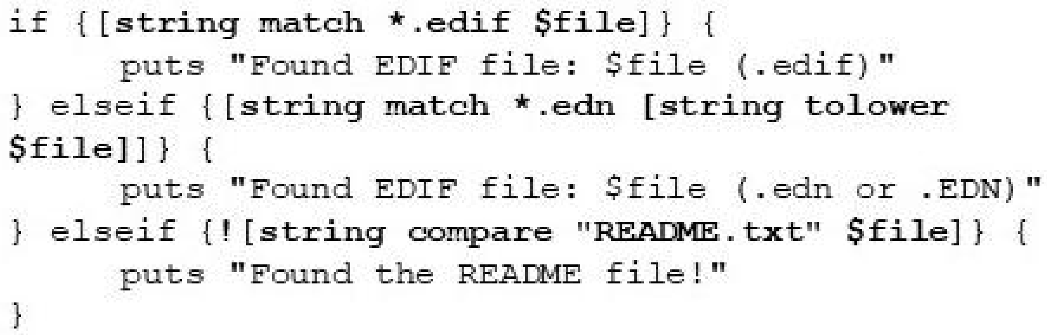
**glob style matching**

? : matches any SINGLE character

* : matches any sequence of xero or more characters

[abc] : matches any SINGLE character in abc

# Script using String Commands



```
if {[string match *.edif $file]} {
     puts "Found EDIF file: $file (.edif)"
} elseif {[string match *.edn [string tolower
$file]]} {
     puts "Found EDIF file: $file (.edn or .EDN)"
} elseif {![string compare "README.txt" $file]} {
     puts "Found the README file!"
}
```

# Script For You

```
set msg "T eciga lsrt"
set code "0 4 9"
set i 0
set j 0

while {$i != ([string length $msg]-1)} {
      foreach k $code {
            set i [expr $k+$j]
            append secret_msg [string index $msg $i]
      }
      incr j
}
puts $secret_msg
```

# Array

# Arrray

- Collection of elements (**Unordered**)
- Each element is given a LABEL (also called key, index)

Array Name: `courses`

| Key | Value |
|---|---|
| dileep | pd |
| ashok | tcl |
| ravi | verilog |

# Read/Write into Array

- Similar to `set` command
- Need to set key before accessing it

```
% set courses(dileep) pd

% set courses(ashok) tcl

% set courses(ravi) verilog

% set courses(dileep)

% set courses(murali)
```

# Basic Array Operations

- Length of array
  ```
  % array size courses
  ```
- List Keys of Array
  ```
  % array names courses
  ```
- Retrieve element’s value
  ```
  % puts “Ashok is taking $courses(ashok) course”
  ```
- Element Key Name can also be Variable
  ```
  % set student_name dileep
  % puts “$student_name is taking $courses($student_name) course”
  ```
- Retrieve entire contents of array
  ```
  % array get courses
  ```

# Set New Array

- One or more elements can be added in one command using `array set`

```
% array set courses { pravin tcl1 “anand raj”
simulation }


% array get courses
```

# Multi-Dimensional Array

- Idea is to use keys which look like multi dimensional indices

2-D Array

```
% set image(0,0) 255
% set image(0,1) 33
% puts "Pixel intensity at (0,1) is $image(0,1)."
% array names image
% array get image
```

3-D Array

```
% set frame_set(10,100,2) 250

% array get frame_set
```

# Some Scripts

- Look Up table

```
% array set ports {  and2 {i1 i2 o1}  or2 {i1 i2 o1}  inv {i1
o1} half_add {i1 i2 o1 cout}  full_add {i1 i2 cin o1 cout}
}
% set cell "and2"
% puts "Cell $cell has ports: $ports($cell)"
```

- 2 D Image Storage

```
% for {set x 0} {$x < 256} {incr x} {
      for {set y 0} {$y < 256} {incr y} {
            set image($x,$y) $y
      }
}
```

# List

# What is List

- Collection of **ordered** elements
    - Elements are strings
        - Can represent anything (other lists, tcl data structures, string values etc)
    - Elements are separate by whitespaces
        - tabs
        - space
        - new lines

**List Examples**

```
% # List of 5 elements
% set fruits “apple lemon banana pear grapes”
% set students {ravi vijay murali dhileep}
```

# Basic List Operations

- Length of list

```
% llength $students
```

- Retrieving list element using index

```
% lindex $students 1
% lindex $students end
```

- Getting Range of Elements (sub list)

```
% lrange $students 0 2
```

- Show Entire List

```
% set students
```

- What about this?

```
% lindex $students 5
% lrange $students 5 8
```

# More list operations

- `lappend` to append elements at the end of list
  - Optimized for speed
  - Will create new if it does not exist
  - Similar to 'append' for string
  - One element for each argument

No $ here

```
% lappend students anand
% set students "$students pawan"
```

- `linsert` to insert new list element at ANY position

  ```
  % set students [linsert $students 1 parth]
  ```

- `lreplace` to replace or delete existing list elements

  ```
  % set students [lreplace $students 4  4 dileep]
  ```

# More ways of making List

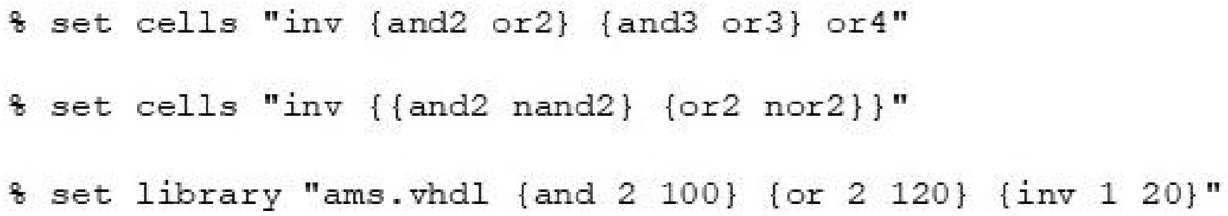
- Use word grouping with “” or {}
- Use list command (all arguments become list elements)

```
% set fruits [list apple lemon banana pear grapes]
% set basket “apple lemon”
% set fruits_list [list $basket banana pear grapes]
% set fruits_quotes "$basket banana pear grapes"
% llength $fruits_list
% llength $fruits_quotes
```

- There is DIFFERENCE in two ways of making list.
  - Word Grouping : Standard white space separation
  - List command: Every argument is one item in list

# Nested Lists

- Use {}'s to define lists within lists



```
% set cells "inv {and2 or2} {and3 or3} or4"

% set cells "inv {{and2 nand2} {or2 nor2}}"

% set library "ams.vhdl {and 2 100} {or 2 120} {inv 1 20}"
```

- How to check if list is nested (confused)?
- Try this:
  - `% llength $cells`
  - `% llength $library`

# Concatenate Lists

- Adds two or more lists to make a new list
- Removes one level of grouping

```
% set basket "apple pear grapes"
% set basket_exotic "lemon banana"

% set fruits [concat $basket $basket_exotic]
% set fruits [list $basket $basket_exotic]
```

- What is the difference in above and below two ?

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

# String <-> Lists

- Easy to convert between list and string
  - `split` splits the string into a set of list elements (based on splitter element)
  - `join` joins the elements of list into String (separator can be any string)

```
% set dir_list [split "/Decoder/mp3/buffer/gnd" "/"]
% set dir_list [lreplace $dir_list end end "GND"]

% set new_path [join $dir_list "::"]
```

# Procedures (`proc`)

# proc <-> Create a new TCL Command

- Would be same as any standard TCL command
- Proc Details :
    - Name : `average`
    - Arguments : `n1 n2 n3 n4` (Are local variables, not available outside the proc)
    - Body : set of tcl commands
    - Return : `$avg` (if no return, then returns value of last executed command)

```
% proc average {n1 n2 n3 n4} {
      set avg [expr {($n1+$n2+$n3+$n4)/4.0}]
      return $avg
}

% average 1 2 3 4
% average -10 10 -50 3
```

# Global Variables

- To access global variables (variables declared outside proc)
- Use `global` command

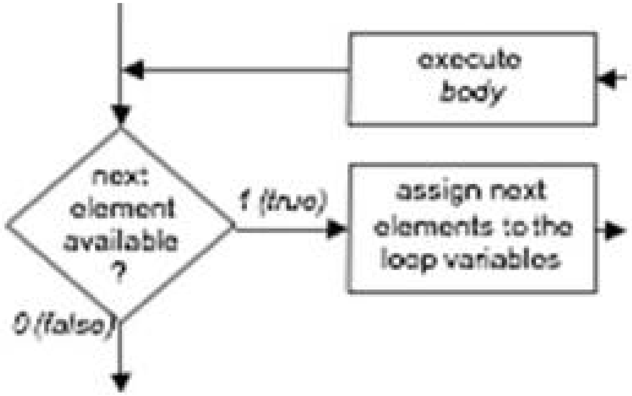
```
% set appname "My script"

% proc print_error {msg} {
    global appname
    puts "$appname: $msg"
}
% print_error "Is a global variable"
```

# Proc to reverse a list

- foreach command
  - Elements processed from left to right
  - Body commands run for each element

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



execute body

next: element available ?

1 (true)

assign next elements to the loop variables

0 (false)

- Reverse a list

```
% proc lreverse {l} {
      set reversed_l ""
      foreach element $l {
            set reversed_l [linsert $reversed_l 0 $element]
      }
      return $reversed_l
}
% set fruits "apple lemon banana pear grapes"
% lreverse $fruits
```

# List vs Arrays

## Lists

- Can store various elements
- Order of elements is preserved
- Elements are retrieved using an INTEGER index
- Lists are manipulated using the list value

```
% set l [list r g b]
% llength $l
```

## Array

- Can store various elements
- Order of elements if NOT preserved
- Elements are retrieved using a STRING key
- Arrays are manipulated using the name of the array variable

```
% array set a {
    r RED g GREEN b BLUE }
% array names a
```

# File Handling

# Simple File Commands

- Works on both Linux and Windows
  - Paths have to be given as per OS

```
% cd
% pwd
% glob
```

```
% file isdirectory ~/project
% file dirname ~/project/README
% file mtime add_v2.vhdl
% file exists add_v3.vhdl
% file readable add_v2.vhdl
```

```
% file mkdir ~/project/daily_backup
% file copy add_v2.vhdl ~/project/daily_backup
% file rename sub_v4.vhdl sub.vhd
% file rename sub.vhd ~/project/archive
% file delete compile.log
```

# Open/Close a file

- Use `open` command to open a file
  - Returns a unique descriptor (also called channel identifier or file id or descriptor)
  - It's unique for each opened file

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

- Command details
  - Name: `open`
  - Args:
    1. file name : `test1.dat`
    2. open mode
       - `r` - read only
       - `w` - write only
       - `a` - append

To close file:

```
% close $fid
```

# Write / Read to/from File

- Use `puts` to write into files
  - ○ Returns empty string on success
  - ○ Error message otherwise

```
% puts $fid “Hello TCL Class”

% puts stdout “this is normal output
on screen

$ puts stderr “for those who
understand stdin, stdout and stderr”
```

- Use `gets` to read lines from file
  - ○ Reads from file line by line

```
% set line [gets $fid]

% set chars [gets $fid line]
```

**Typical Use**

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

# Example Script

```
puts "Welcome to a simple interactive script!"
source to_bits.tcl
while {1} {
    puts "\nCommand:"
    gets stdin line
    if {$line == ""} {continue}

    switch -- [lindex $line 0] {
        quit - exit      {break}
        dec2bits {puts [ to_bits [lindex $line 1] ]}
        dir - ls {puts [ glob * ]}
        default  {puts stderr "Error: unrecognized command"}
    }
}
puts "Thank you!"
```

# Checking for end of file

- Use `eof` command to check for the end of file position
    - Returns 1 at end of file
    - Else 0

```
while {![eof $fid]} {
      gets $fid line
      puts "test1.dat: $line"
}
```

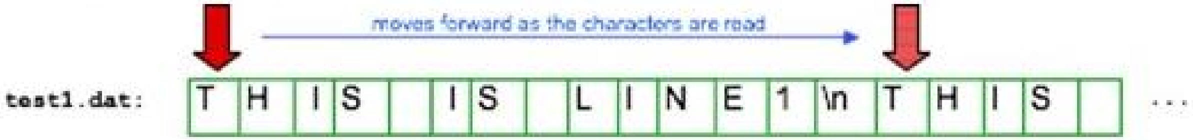
```
set src [open "test1.dat" r]
set dest [open "test2.dat" w]
while {![eof $src]} {
      set c [read $src 1]
      puts -nonewline $dest "$c$c$c"
}
close $src  close $dest
```

Use `read` to read characters only

```
% set seven_chars [read $fid 7]
% set file [read -nonewline $fid]
```

# FILE Pointer Position

- Files are accessed using an “invisible” file pointer
  - Positioned at the beginning when the file is opened
  - Moves forward with each file read


moves forward as the characters are read
test1.dat:
T H I S I S L I N E 1 \n T H I S
...

- Use seek to move pointer to desired position

```
% seek $fid 5 start
% seek $fid 8 current
% seek $fid -10 end
```

# Regular Expressions

# What and Why?

- string already has commands
  - match
  - compare

BUT these can not handle complex string manipulation.

- Regular Expression is:
  - Special string patterns which can match strings using various rules
  - Can be context speicific
  - Generic (will work for many different strings)
- Efficient for complex string search/replace operations
- Regular Expressions can handle complex and repetitive string manipulation tasks efficiently

# Regular Expression (RE) 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`

- Special Characters

| | |
|---|---|
| . | matches ANY SINGLE character |
| [] | matches a SINGLE character from a sequence, e.g.<br>`[abc]`<br>`[A-Z]`<br>`[^A-Z]`<br>`[a-zA-Z0-9_]` |
| * | matches 0 or more occurrences of a preceding ATOM<br>`a*`<br>`[a-z]*`<br>`[A-Z][a-z]*`<br>`.*` |

# Regular Expression (RE) Examples

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

Why use {}

```
% set m_var
```

- RE
  - RE Pattern : `{[A-Z][A-Z]*}`
  - String to search : `"which is better: VHDL or Verilog?"`
  - Variable to store matched string : `m_var`
  - Returns
    - 1 if match is found
    - else 0
  - `-nocase :` to ignore case
  - Matches longest possible string.

# Regular Expression (RE) Script

find_entity_line.tcl

```
set fid [open "adder.vhdl" r]
set add [read $fid]
close $fid
regexp -nocase -- { *entity  *[a-z][a-z0-9_]* *is *} $add e_line

puts "$e_line"
```

```
% source find_entity_line.tcl
```

Adder.vhdl

```
entity add is
port (
...
);
end add;
```

# More RE Symbols

- Alphabet and digit characters are matched as usual
  - ^ matches the BEGINNING of a line, e.g.
    - `^architecture` : string architecture at the beginning of a line
  - `$` matches the END of a line, e.g.
    - `;$` character ; at the end of line

- Alternatives
  - `x|y` matches ONE of the two possible atoms, e.g.
    - `out|in` : string out OR string in
    - `[a-z]|[0-9]` : lowercase letter OR a digit

- Use () to group atoms together, e.g.
  - `([a-z][a-z]*)|[0-9]` : lowercase word of 1 or more letters OR a digit

# Continued.. More RE Symbols

- Sequence matching
  - `+` : matches 1 or more occurrences of a preceding atom e.g.
    - `[a-z]+` : 1 or more lowercase letters ((a word composed of lowercase letters
  - `?` **:** matches 0 or 1 occurrence of a preceding atom
    - `[a-z]?` **:** 0 or 1 lowercase letter

- Meaning of all special RE characters can be escaped with a backslash (\)
  - `((\+?)|-)[0-9]+ :` 1 or more digits preceded by either a character - or optionally character +, i.e. an INTEGER

# Searching for Strings within Strings

- Locate a sub-pattern within a pattern
  - Example: extract VHDL entity name
  - `% regexp -nocase -- { *entity +([a-z][a-z0-9_]*) +is *} $add  e_line e_name`

- Command details:
  - `e_line :` holds the entire matched string
  - `e_name :` holds the 1st matched sub-string
  - `{ *entity +([a-z][a-z0-9_]*) +is *} :` a sub-expression is enclosed within **()**

Adder.vhdl

```
entity add is
port (
...
);
end add;
```

# One more example

file: serr.log

```
...
Synthesizing work.interface.rtl
@W:"c:\lab6\interface.vhd":82:39:82:43|Signal  aver2 in the sensitivity list is not used in the process
Post processing for work.interface.rtl
...
```

```
proc extract_warnings {f} {
     set in_file [open $f r]
     while {[gets $in_file line] >= 0} {
          if {[regexp @W $line]} {
               regexp -nocase -- \
                    {([a-z_]+\.vhd)[0-9:"|]+([a-z_\. 0-9]+)} \
                    $line buf filename msg
               puts "$filename\t\t$msg"
          }
     }
     close $f
}
```

# String Substitution

- Use regsub for RE-based string substitution

```
% regsub -- {[a-z]+} "cin : std_logic;" "carry_in" new_str
```

- Command Details
  - RE Pattern : `{[a-z]+}`
  - Input String : `"cin : std_logic;"`
  - Replace matched pattern with this : `"carry_in"`
  - Variable which will store the result : `new_str`
  - Returns
    - number of matched patterns

# TK : Graphic Toolkit

Toolkit for Window Programming

# Everything is Widget

- Widget
  - button
  - menu
  - text window
- Again commands used to create and manipulate widgets
- Hierarchical windows arrangement
  - Primary Window
  - Children Windows reside in Primary
  - And it goes on
- Every action is event on a widget

# Button Example

- Command : `button`
- Name of widget : `.hello`
- Text on widget : `-text <string>`
- Command to run on click : `-command {puts stdout “Hello, World!”`

```
#!/usr/local/bin/wish -f
button .hello -text Hello \
-command {puts stdout “Hello,
World!”}
pack .hello -padx 20 -pady 10
```

# Other Widgets

- Check Button - `checkbutton`
- Radio Button - `radiobutton`
- Menu Button - `menubutton`
- Menu - `menu`
- Label - `label`
- Entry - `entry`
- List Box - `listbox`
- Text - `text`
- Scale - `scale`
- Scroll Bar - `scrollbar`

# Packages and Namespaces

package require

package provide

namespace, variable

**Other things to cover**:

```
lsearch
(option for pattern matching) in string match, switch etc
lsort
upvar
default variables in proc
```

# Resources

- TclTutor App : www.msen.com/~clif/TclTutor.html
- Tcl Manual Tutorial : https://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html
- Practical Tcl & TK : Book
- http://www.beedub.com/book/3rd/Tclintro.pdf : 3 free chapters from a good book
- Lots of good examples at: http://pleac.sourceforge.net/pleac_tcl/