Agenda: Day One

DAY 1

Unit	Register to Register Paths	Lab
0i	Welcome	
1	Introduction to Static Timing Analysis	
2	Writing Basic Tcl Constructs in PT	
3	Reading Data	
4	Constraining Internal Reg-Reg Paths	

Unit Objectives 2-2

After completing this unit, you should be able to:

- **Use PT-Tcl variables**
- **Embed PT-Tcl commands**
- **Describe basic control structures**
- **Define a simple Tcl procedure**
- **Describe how to use the Tcl Syntax Checker**

What is PT Tcl? 2-3

Tcl = Tool Command Language (*tickle*):

- **PT-Tcl is the command interface to PrimeTime**
- **Built on the "open" industry-standard shell programming language Tcl**
- **PT-Tcl an interpreted, fully programmable and fully scriptable language**

Tcl was originally developed by John K. Ousterhout at UCA Berkeley.

There are many books on the topic of Tcl programming, here a few:

Tcl and the Tk Toolkit, John K. Ousterhout

Practical Programming in Tcl and Tk, Brent B. Welch

Visual Tcl, David Young

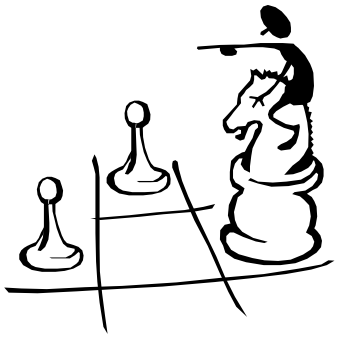
Running PT Tcl Commands 2-4

- **Commands can be typed interactively in PT Tcl:**

```
pt_shell> echo "Running my.tcl..."
pt_shell> source my.tcl
```

- Or executed in batch mode

```
UNIX% pt_shell -f my.tcl | tee my.log
```



Tcl Basics 2-5

- **Commands:**
  - One or more words separated by white space
  - First word is command name, others are arguments
  - Returns **string** result
- **Script:**
  - Sequence of commands
  - Commands separated by newlines and/or semi-colons

Examples:

```
set a 22                    set the variable 'a' to 22
echo "Hello, World!"        world's shortest program
```

Arguments 2-6

- **Parser assigns no meaning to arguments:**

  C: `x = 4; y = x+10` *y is 14*

  Tcl: `set x 4; set y x+10` *y is "x+10"*

- **Different commands assign different meanings to their arguments**
- **"Type-checking" must be done by commands themselves**

```
set a 122
expr 24/3.2
read_file -format verilog foo.v
string length Abracadabra
```


"Type-checking" means that the command itself, not the "Tcl Parser", has to figure out whether the arguments passed to it are of the correct type, e.g. the `expr` function would make sure it is receiving numbers and arithmetic operators.

`set a 122`

Assigns the ***string*** "122" to the variable named "a"

`expr 24/3.2`

The Tcl parser calls the `expr` function with the "string" "24/3.2". The command `expr` (mathematical expression calculator) interprets/type-checks "24/3.2", calculates and returns the result, in this case the ***string*** "7.5". The result is a "floating point" number because one of the arguments was a floating point as well. Division of two "integers" yields an integer result, e.g. [expr 10/3] would return 3, **not** 3.333!

`read_file -format verilog foo.v`

This is a DC/PT-command that reads the verilog file foo.v. The Tcl parser just passes the arguments to the command "read_file" without interpreting the arguments.

`string length Abracadabra`

The `string` function can perform many operations on strings. In this example the function will return the length (again as a ***string***), here "11".

Variable Substitution 2-7

Syntax: *$varName:*

- **Variable name is:**
  - letters
  - digits
  - underscores *
- **May occur anywhere in a word**

Sample command	Result
`set b 66`	`66`
`set a b`	`b`
`set a $b`	`66`
`set a $b+$b+$b`	`66+66+66`
`set a $b.3`	`66.3`
`set a $b4`	*`no such variable`*

- **Variables do not need to be declared:**
  - All are type “string” and of arbitrary length

* Actually, Variables in Tcl can have any shape and form, e.g.:

```
set {*&#2$3rdt} 333
  333
puts ${*&#2$3rdt}
  333
```

Using a combination of letters, digits and underscores is recommended; this will make variable substitution a lot easier.

To remove a variable, use the command unset e.g.:

```
unset b
```

Variables can be concatenated with strings in many ways, e.g. to get the contents of the variable b concatenated with the string “test”, you type:

```
set a ${b}test        -> “66test”
```

Variables do not need declaration as in languages like C, Pascal, etc., since there is only one “type” of variable – a string.

Command Substitution 2-8

Syntax: [*script*]:

- **Evaluate script, substitute result**
- **May occur anywhere in a word**

Sample command	Result
`set b 8`	`8`
`set a [expr $b+2]`	`10`
`set a "b-3 is [expr $b-3]"`	`b-3 is 5`

"`expr`" is a Tcl function that performs math operations.

Controlling Word Structure 2-9

Words end or break at white space and semi-colons, except:

- **Double-quotes prevent breaks:**

```
set a "x is $x; y is $y"
```

- **Curly braces prevent breaks and substitutions:**

```
set a {[expr $b*$c]}
```

- **Backslashes quote special characters:**

```
set a word\ with\ \$\ and\ space
```

- **Backslashes can escape newline (line-continuation):**

```
report_constraint \
  -all_violators
```

Writing Basic Tcl Constructs in PT

Synopsys 34000-000-S16

PrimeTime: Introduction to Static Timing Analysis

```
set a "x is $x; y is $y"
```
Sets the variable a to "x is 3; y is 5"

```
set a {[expr $b*$c]}
```
Sets the variable a to "[expr $b*$c]"

```
set a word\ with\ \$\ and\ space
```
Sets the variable a to "word with $ and space."

```
report_constraint \
  all_violators
```
Make sure that there is no space after the backslash.
"Line-continuation" means "backslash – newline."

Notes on Substitution and Parsing 2-10

Tcl substitution rules are simple and absolute.

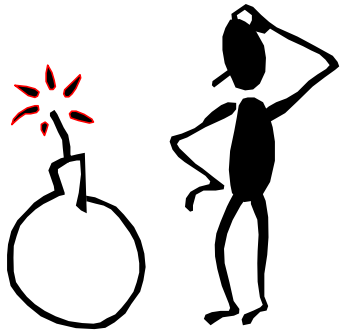
Example comments:

```
set a 22; set b 33                  OK
# this is a comment                 OK
set a 22  # same thing?             Wrong!
set a 22 ;# same thing              OK
```

Parser looks at a command just once!

```
set a 7
set b a
echo $b             a

echo $$b            $a
expr $$b+5          12
```



Writing Basic Tcl Constructs in PT

`echo $$b`

This command returns “$a”. The Tcl parser looks at the command only once, it substitutes $b by its contents “a”

`expr $$b+5`

Here, the Tcl parser does the first substitution, replacing $b with “a”. The `expr` command performs a second round of substitution, replacing $a with its contents “7”

Tcl Lists 2-11

- **Data can be arranged as lists**

  Example: `set colors {red green blue}`

- **Lists are accessed through special commands**

  Example:

```
llength $colors
  3

lappend colors white yellow
 red green blue white yellow

set colors [lsort $colors]
  blue green red white yellow
```


Other list manipulation commands include:

`lindex list index`	Returns value of element at *index* in *list*
`linsert list index element [element…]`	Returns new list formed by inserting given new elements at *index* in *list*
`lrange list first last`	Returns new list from slice of list at indices *first* through *last* inclusive
`lsearch list pattern`	Returns index of first element in *list* that matches *pattern* (-1 for no match)
`join list [joinString]`	Returns string created by joining all elements of *list* with *joinString*
`split string [splitChars]`	Returns a *list* formed by splitting string at each character in *splitChars*

Note: List indices start at `0` *and the word* **`end`** *may be used to reference the last element in the list. To echo the first and last element of a list you would use:*

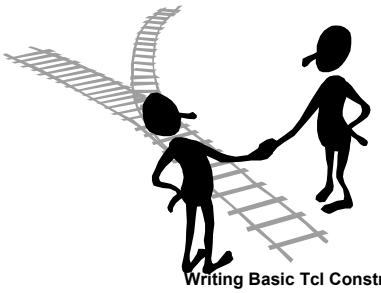
```
echo "First: [lindex $colors 0], Last: [lindex $colors end]"
```

Control Flow 2-12

There are many commands to control the flow of a Tcl script.

Example:

```
if [file exists postlayout_design.db] {
   read_db postlayout_design.db
} elseif [file exists prelayout_design.db] {
   read_db prelayout_design.db
} else {
   echo "Could not read design!"
}
```



Writing Basic Tcl Constructs in PT

Other `file` command options are:

`executable`	returns a 1 if file is executable by current user
`exists`	returns a 1 if file name exists
`extension`	returns characters after and including the last dot
`isdirectory`	returns a 1 if file name is a directory
`isfile`	returns a 1 if file name is a file
`mtime`	returns time the file was last modified
`owned`	returns a 1 if file name is owned by the current user
`readable`	returns a 1 if the file name is readable
`size`	returns the size of file name
`type`	returns a string giving the type of the file name
`writable`	returns 1 if file name is writable

For a complete description of the options see the man page of `file`.

Looping Constructs 2-13

- A foreach loop iterates through members of a list:

```
set a "red green blue"
foreach color $a {
  set s "$color is a nice color..."
  echo $s
}
```

- Other looping commands:
  - for
  - while

Synopsys 34000-000-S16

Writing Basic Tcl Constructs in PT
PrimeTime: Introduction to Static Timing Analysis

```
for {set i 1} {$i <= 10} {incr i} {
  echo "$i potato"
}
```

The following example reverses a list:

```
set a "red green blue"
set b ""
set i [expr [llength $a] - 1]
while {$i >= 0} {
  lappend b [lindex $a $i]
  incr i -1
}
```

Procedures 2-14

- `proc` command defines a procedure:

```
proc sub1 {x} {expr $x-1}
```

- Procedures behave just like built-in commands:

```
sub1 3          2
```

- Arguments can have default values:

```
proc decr {x {y 1}} {
    expr $x-$y
}
```

Writing Basic Tcl Constructs in PT

Synopsys 34000-000-S16 PrimeTime: Introduction to Static Timing Analysis

In a procedure, the last command's return value is the procedure's return value as well. For more control over what is returned, use "`return`" and have a look at the example below.

As stated earlier, brackets {} can be used to control word structure. This means that if the procedure body is multiple lines, the opening bracket still **needs** to be on the first line, since the command `proc` requires 3 arguments.

A procedure can also have a variable number of arguments.

Here is an example procedure:

```
proc add args {       the argument name "args" is FIXED – you cannot use a different name!
  set result 0
  foreach value $args {
    incr result $value
  }
  return $result
}
```

Now the procedure can be called in these ways:

```
add 1 2 3 4         10
add 7 8             15
```

Procedures and Scope 2-15

- **Scoping: local and global variables:**
  - Interpreter knows variables by their name and scope
  - Each procedure introduces a new scope
- **`global` procedure makes a global variable local:**

```
> set x 10
> proc DELTAX {d} {
    set x [expr $x-$d]
  }
> DELTAX 1    can't read "x": no such variable
> proc DELTAX {d} {
    global x
    set x [expr $x-$d]
  }
> DELTAX 1    9
```


You can use the command **`info`** to get information on procedures created in PT memory.

```
pt_shell> info procs;   # Returns all procedures defined in current session.
```

(By using UPPERCASE letters for the procedure names, it is easier to locate the procedure name as returned by "info procs").

```
pt_shell> info args DELTAX
d
pt_shell> info body DELTAX
    global x
    set x [expr $x-$d]
```

Recommendations 2-16

- Use procedures to simplify scripts
- Prefer UPPER case letters for procedure names and user defined variable names
- Avoid using aliases and abbreviating command names in scripts
- Use common extensions:

  `my_script.pt` or `foo.tcl`
- Use full option names in commands:

```
create_clock -period 5 clk
```

- Avoid “snake scripts”
- Perform syntax checking

“Snake scripts” are scripts that call scripts that call scripts: Very hard to debug.

Avoid sourcing scripts from your `.synopsys_pt.setup` file, since these scripts will be executed automatically every time you start the tool. This of course excludes scripts that only define procedures for later use.

Syntax Checking 2-17

Check your script in PrimeTime

```
pt_shell> package require snpsTclPro
1.0

pt_shell> check_script my_script.tcl

Synopsys Tcl Syntax Checker - Version 1.0

Loading snps_tcl.pcx...
Loading primetime.pcx...
scanning: /home/.../my_script.tcl
checking: /home/.../my_script.tcl
my_script.tcl:6 (warnUndefProc) undefined procedure: set_ouput_delay

set_ouput_delay 4 -clock clk [all_outputs]

^
```


The Syntax checker has some limitations:

- Cannot check abbreviated command names
- Cannot understand aliased command name
- Does not use PT's search_path variable to find a script file

Need Help?

- **Help on PT Tcl Commands:**

```
help create*
help -verbose create_clock
create_clock -help
man create_clock
```

- **Help on PT variables:**

```
printvar *_path
echo $link_path
man link_path
```

Lab Overview and Review 2-19

45 min

During this lab, you will:

- Use the “transcript” program to translate a given specification (in DC shell format) into PT Tcl format
- Write a Tcl procedure to convert a given frequency (in MHz) into a clock period (in ns)
- Use the Syntax checker to debug your Tcl script
- Find any reusable information from the command log file

Review 2-20

❶ The command needed to display the value of all **"*_path"** variables is?

a) help *_path
b) help –verbose *_path
c) printvar *_path
d) echo $*_path
e) C and D

❷ The result of the command **"echo 25 divided by 3 is [expr {25/3}]"** is?

a) 25 divided by 3 is 8.333
b) 25 divided by 3 is 8
c) 25 divided by 3 is 25/3
d) 25 divided by 3 is [expr 25/3]

❸ Is the following Tcl statement for PrimeTime correct?

- lappend link_path "CORE.db RAMS.db"