Tcl Training Class

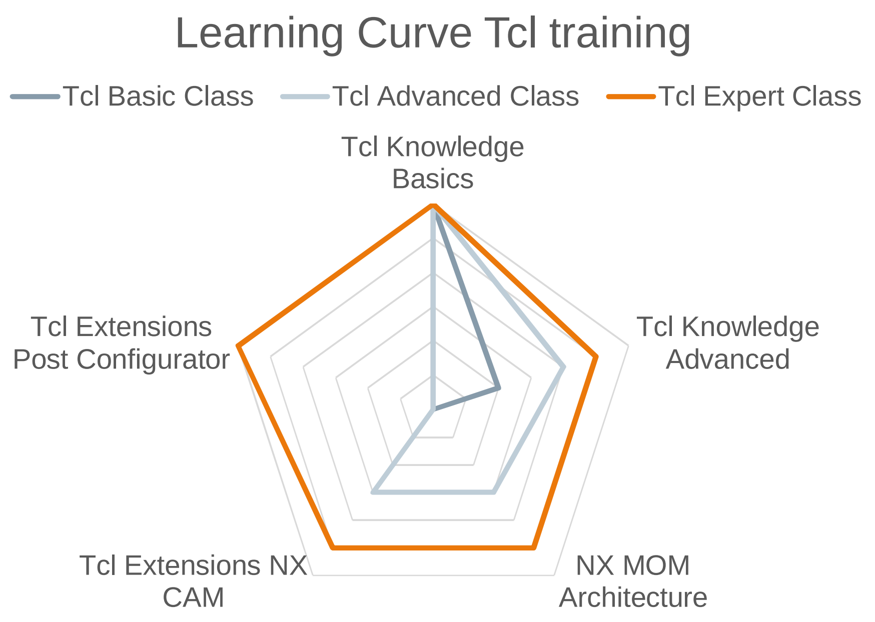
Basic Class

Realize innovation.

Learning goals class

SIEMENS
Ingenuity for life

- ✓Tcl background and history
- ✓Syntax for simplified programming
- ✓Structure of Tcl
- ✓Working with console application
- ✓Set and get variables
- ✓Creating procedures
- ✓Expressions
- ✓If conditions
- ✓Switch conditions
- ✓Looping things
- ✓How to help myself and useful tools



Agenda:

Agenda:

General Introduction Tcl
Helpful Documentation and Tools for the Class
Interpreter and command structure
Variables
Expression
Special Characters
Word Structure
Commands & Procedures
Errorhandling
Control Structures
Tcl standard commands
Summary

Tcl - What is it?

Tcl (pronounced "tickle" or *tee cee ell* /ˈtiː siː ɛl/[6]) is a high-level, general-purpose, interpreted, dynamic programming language. It was designed with the goal of being very simple but powerful.[7] Tcl casts everything into the mold of a command, even programming constructs like variable assignment and procedure definition.[8] Tcl supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles.

It is commonly used embedded into C applications,[9] for rapid prototyping, scripted applications, GUIs, and testing.[10] Tcl interpreters are available for many operating systems, allowing Tcl code to run on a wide variety of systems. Because Tcl is a very compact language, it is used on embedded systems platforms, both in its full form and in several other small-footprint versions.[11]

The popular combination of Tcl with the Tk extension is referred to as **Tcl/Tk**, and enables building a graphical user interface (GUI) natively in Tcl. Tcl/Tk is included in the standard Python installation in the form of Tkinter.

Reference: https://en.wikipedia.org/wiki/Tcl

Tcl History

The Tcl programming language was created in the spring of 1988 by John Ousterhout while working at the University of California, Berkeley.[12][13] Originally "born out of frustration",[9] according to the author, with programmers devising their own languages intended to be embedded into applications, Tcl gained acceptance on its own. Ousterhout was awarded the ACM Software System Award in 1997 for Tcl/Tk.[14]

The name originally comes from **T**ool **C**ommand **L**anguage, but is conventionally spelled "Tcl" rather than "TCL".[15]

Reference: https://en.wikipedia.org/wiki/Tcl

Features

Reference: https://en.wikipedia.org/wiki/Tcl

- All **operations are commands**, including language structures. They are written in prefix notation.
- Everything can be **dynamically redefined and overridden**. Actually, there are no keywords, so even control structures can be added or changed, although this is not advisable.
- All data types can be **manipulated as strings**, including source code. Internally, variables have types like integer and double, but converting is purely automatic.
- Variables are not declared, but assigned to. Use of a **non-defined variable results in an error**.
- Fully dynamic, class-based object system, TclOO
- **Event-driven interface** to sockets and files
- All commands defined by Tcl itself generate **error messages on incorrect usage**.
- Extensibility, via C, C++, Java, Python, and Tcl.
- **Interpreted language** using bytecode, Full Unicode, Regular expressions
- **Cross-platform**: Windows API; Unix, Linux, Macintosh etc.
- cross-platform integration with windowing (GUI) interface Tk.
  - ➢ Full development version (for Windows e.g. ActiveState Tcl)
  - ➢ The Jim Interpreter, a small footprint Tcl implementation
  - ➢ Freely distributable source code under a BSD license.
  - ➢ **NX CAM** MOM **(Manufacturing Output Manager)** extension for post processing

Agenda:

Helpful Documentation and Tools for the Class

SIEMENS
Ingenuity for life

PDF Book for Tcl Basics as reference
http://www.freebookcentre.net/programming-books-download/Tcl-Basics-(PDF-118P).html

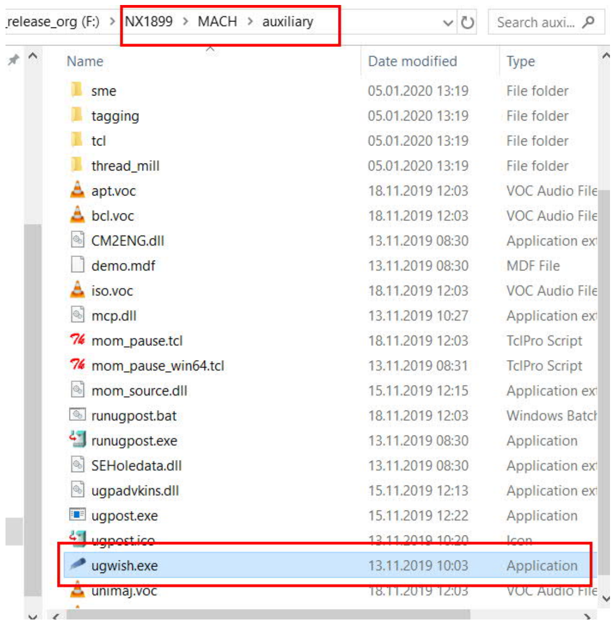
Wikibooks:
https://en.wikibooks.org/wiki/Tcl_Programming/Introduction

Online Tcl-Interpreter (**Option 2** to use for the class):
https://www.jdoodle.com/execute-tcl-online/

Online help Tcl Basics:
https://stackoverflow.com/questions/tagged/tcl

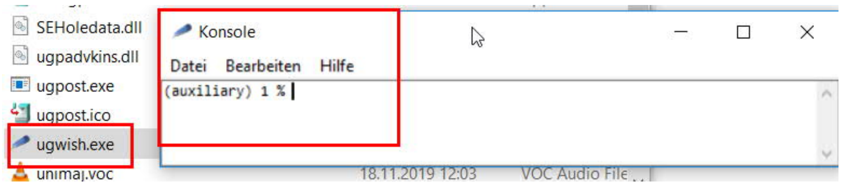
Siemens NX Manufacturing Forum -> NX CAM Postprocessor Group (Post Processor specific topics)
https://community.sw.siemens.com/s/group/0F94O000005TFzSAM/nx-cam-postprocessor-group

Tcl console application integrated in NX CAM (**Option 1** to use for the class)



release_org (F:) > NX1899 > MACH > auxiliary

Name	Date modified	Type
sme	05.01.2020 13:19	File folder
tagging	05.01.2020 13:19	File folder
tcl	05.01.2020 13:19	File folder
thread_mill	05.01.2020 13:19	File folder
apt.voc	18.11.2019 12:03	VOC Audio File
bcl.voc	18.11.2019 12:03	VOC Audio File
CM2ENG.dll	13.11.2019 08:30	Application ex
demo.mdf	13.11.2019 08:30	MDF File
iso.voc	18.11.2019 12:03	VOC Audio File
mcp.dll	13.11.2019 10:27	Application ex
mom_pause.tcl	18.11.2019 12:03	TclPro Script
mom_pause_win64.tcl	13.11.2019 08:31	TclPro Script
mom_source.dll	15.11.2019 12:15	Application ex
runugpost.bat	18.11.2019 12:03	Windows Batch
runugpost.exe	13.11.2019 08:30	Application
SEHoledata.dll	13.11.2019 08:30	Application ex
ugpadvkins.dll	15.11.2019 12:13	Application ex
ugpost.exe	15.11.2019 12:22	Application
ugpost.ico	13.11.2019 10:20	Icon
ugwish.exe	13.11.2019 10:03	Application
unimaj.voc	18.11.2019 12:03	VOC Audio File



SEHoledata.dll
ugpadvkins.dll
ugpost.exe
ugpost.ico
ugwish.exe
unimaj.voc 18.11.2019 12:03 VOC Audio File

```
(auxiliary) 1 %
```

Agenda:

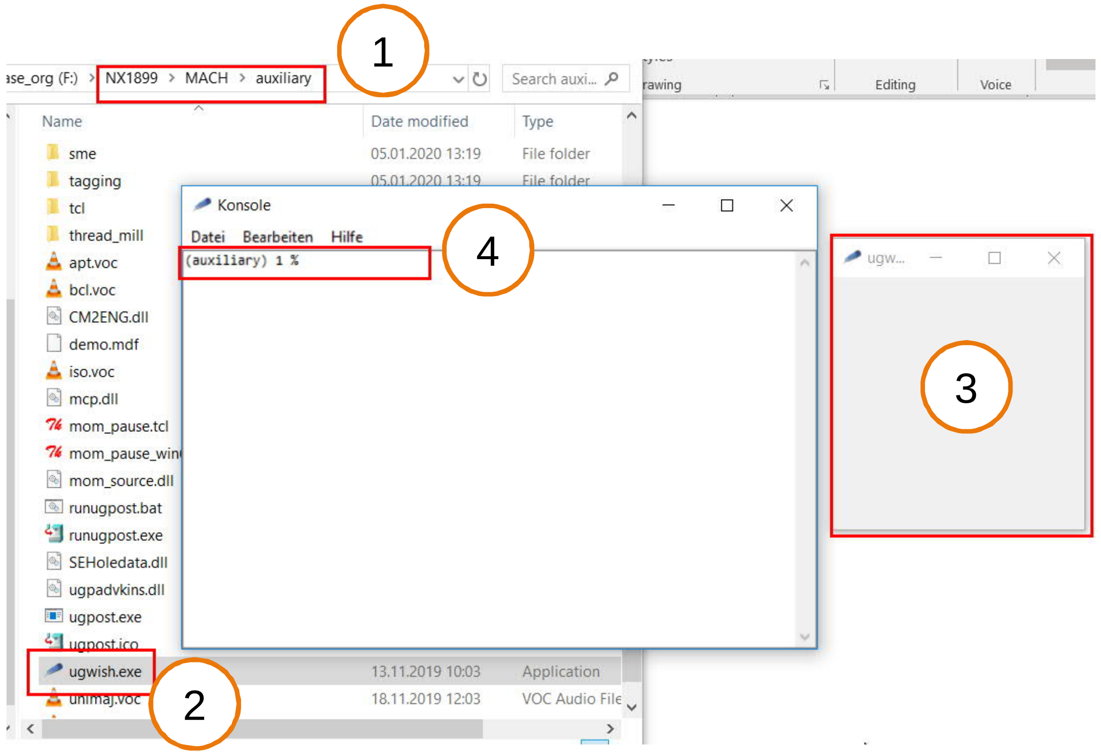
Starting Console application

1 Console application folder

2 Select ugwish.exe to start application

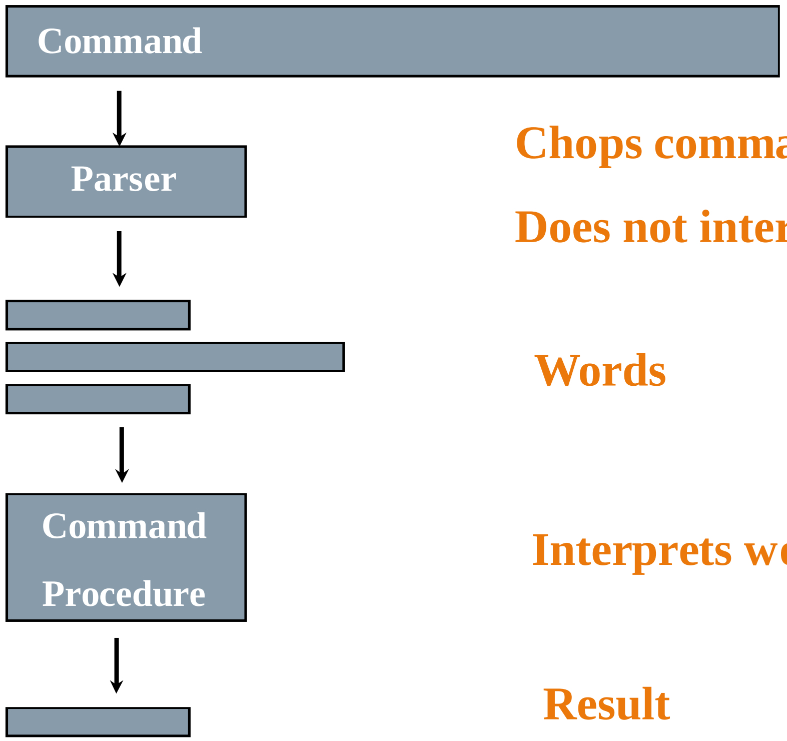
3 Ugwish window is for graphical interface programming (not needed for this training)

4 Console application command line



```
(auxiliary) 1 %
```

Interpretation of a Tcl command



Chops command into words, makes substitutions.

Does not interpret values of words.

Words

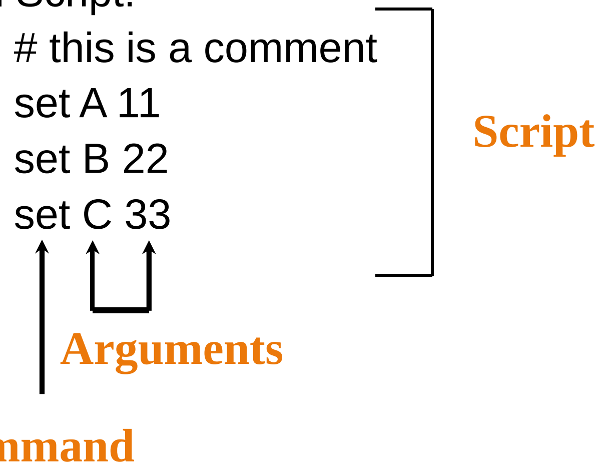
Interprets words, produces string results

Result

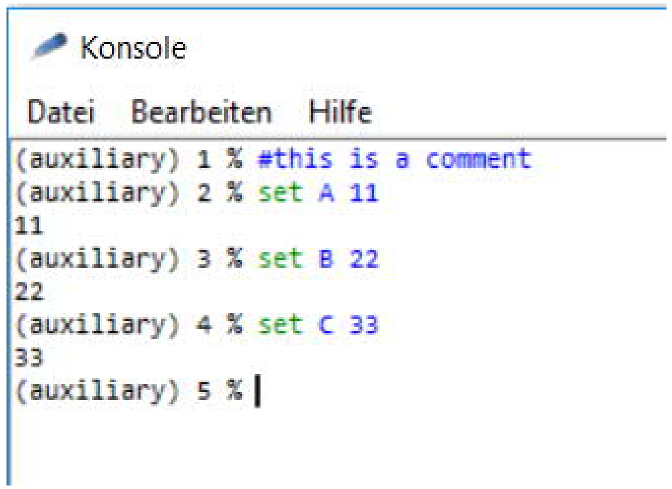
Comment, Script and Command

Example:

- Tcl Script:



```
# this is a comment
set A 11
set B 22
set C 33
```



```
(auxiliary) 1 % #this is a comment
(auxiliary) 2 % set A 11
11
(auxiliary) 3 % set B 22
22
(auxiliary) 4 % set C 33
33
(auxiliary) 5 % 
```

Agenda:

Variables 1/3

Variable name

- Alphanumerical characters, **must start with a letter**
- **no space** in the variable name
- case sensitive
- Tcl is a type-free language

Variable value

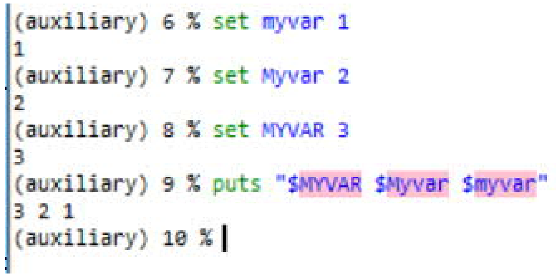
- All **values** are stored as **strings**
- Calculations are performed by special functions/ expressions

Local and global variables

- A variable can be used as a local variable inside a procedure or in a global context

Reading variable value

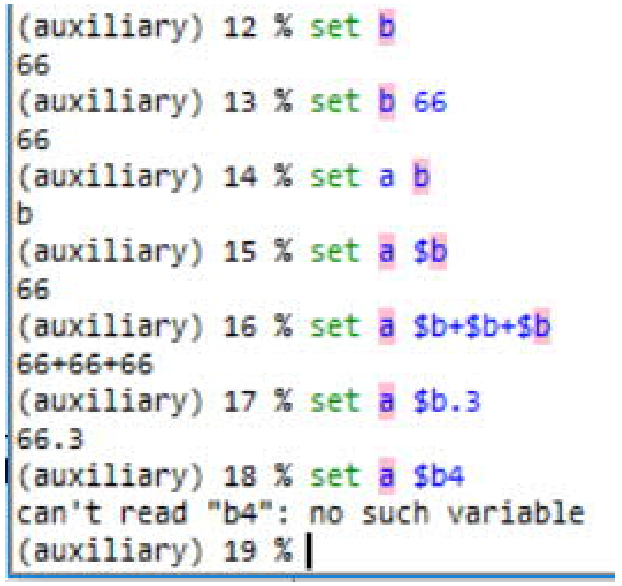
- Variables are read by a prefix of “$”



```
(auxiliary) 6 % set myvar 1
1
(auxiliary) 7 % set Myvar 2
2
(auxiliary) 8 % set MYVAR 3
3
(auxiliary) 9 % puts "$MYVAR $Myvar $myvar"
3 2 1
(auxiliary) 10 %
```

Variables 2/3

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



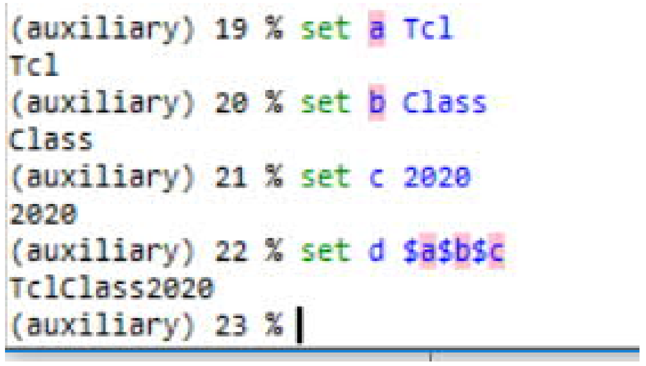
```
(auxiliary) 12 % set b
66
(auxiliary) 13 % set b 66
66
(auxiliary) 14 % set a b
b
(auxiliary) 15 % set a $b
66
(auxiliary) 16 % set a $b+$b+$b
66+66+66
(auxiliary) 17 % set a $b.3
66.3
(auxiliary) 18 % set a $b4
can't read "b4": no such variable
(auxiliary) 19 %
```

Variables 3/3 - Variable Substitution

$*varname*

take the value of the variable with the name *varname*

Sample command	**Result**
set a Tcl	a=Tcl
set b Class	b=Class
set c 2020	c=2020
set d $a$b$c	d=$a$b$c=Tcl Class 2020



```
(auxiliary) 19 % set a Tcl
Tcl
(auxiliary) 20 % set b Class
Class
(auxiliary) 21 % set c 2020
2020
(auxiliary) 22 % set d $a$b$c
TclClass2020
(auxiliary) 23 %
```

Agenda:

Rule 1 to keep

SIEMENS
Ingenuity for life

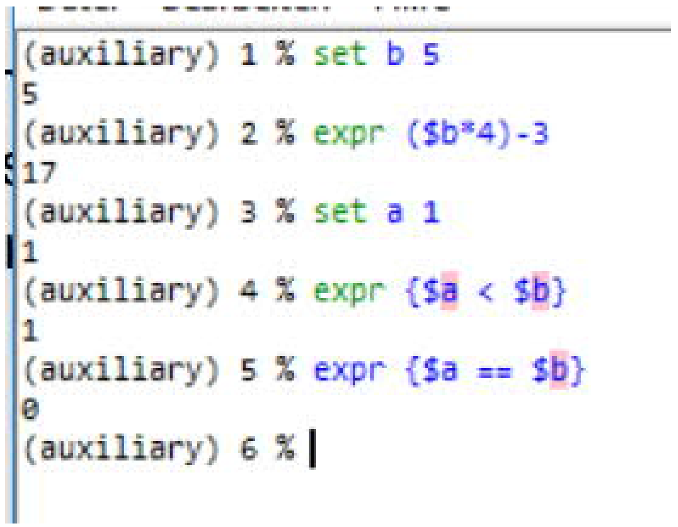
Everything is a string!

Expressions 1/2

SIEMENS
Ingenuity for life

- To do calculations the Tcl command “expr” must be used
- C-like (int and double), extra support for string operations
- Support for command and variable substitution within expressions

Sample command	Result
set b 5	5
expr ($b*4)-3	17
set a 1	1
expr {$a < $b}	1
expr {$a == $b}	0



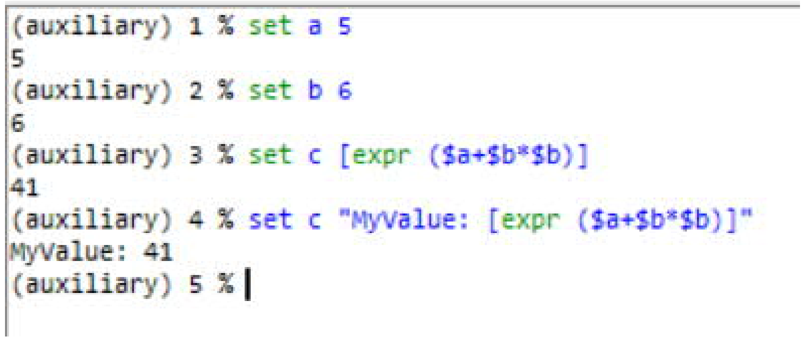
```
(auxiliary) 1 % set b 5
5
(auxiliary) 2 % expr ($b*4)-3
17
(auxiliary) 3 % set a 1
1
(auxiliary) 4 % expr {$a < $b}
1
(auxiliary) 5 % expr {$a == $b}
0
(auxiliary) 6 %
```

Expressions 2/2 – Command Substitution

SIEMENS
Ingenuity for life

- Syntax: [script]
- Execute script, substitute result

Sample command	Result
set a 5	5
set b 6	6
set c [expr ($a+$b*$b)]	41
set c “MyValue: [expr {$a+$b*$b}]”	MyValue: 41



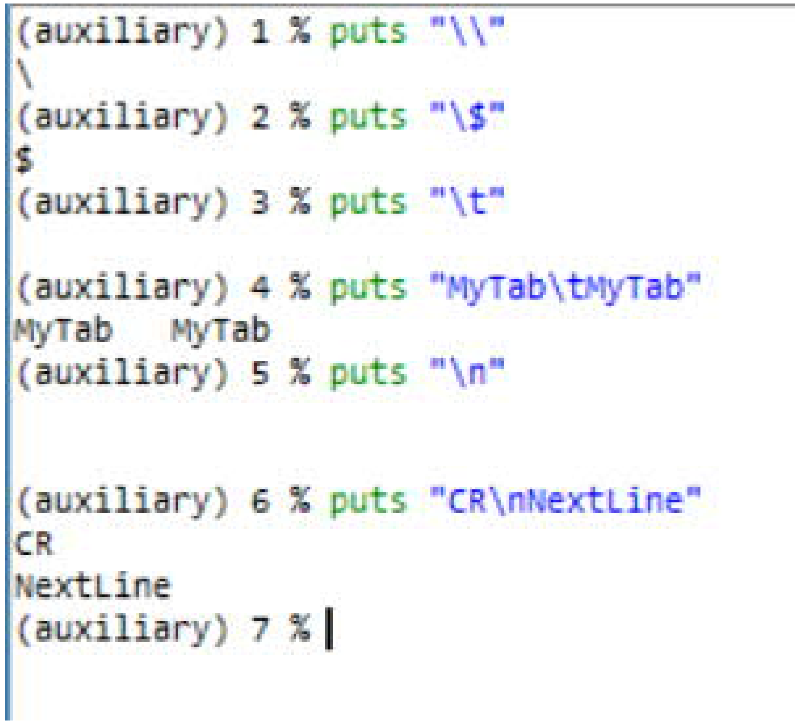
```
(auxiliary) 1 % set a 5
5
(auxiliary) 2 % set b 6
6
(auxiliary) 3 % set c [expr ($a+$b*$b)]
41
(auxiliary) 4 % set c "MyValue: [expr ($a+$b*$b)]"
MyValue: 41
(auxiliary) 5 %
```

Agenda:

Output Special Characters

SIEMENS
Ingenuity for life

Sample command	Result
puts “\\”	\
puts “\$”	$
puts “\t”	Tab
puts “MyTab\tMyTab”	MyTab MyTab
puts “\n”	Carriage Return
puts “CR\nNextLine”	CR NextLine



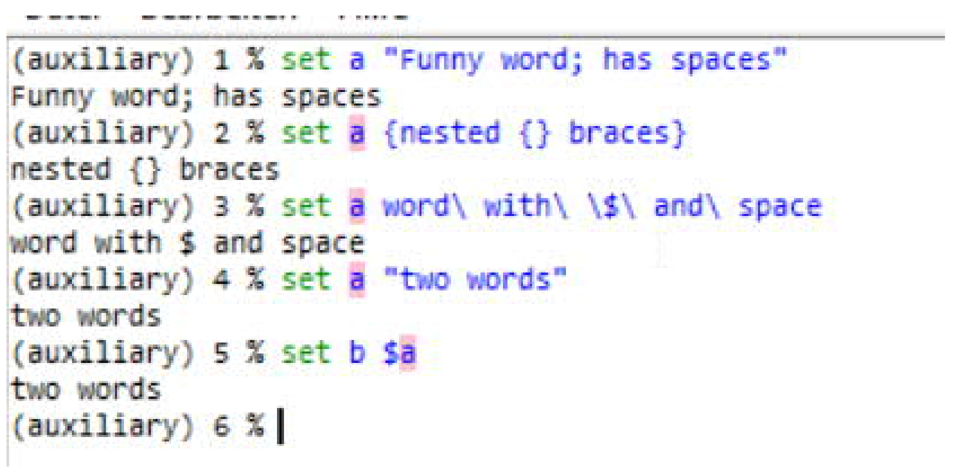
```
(auxiliary) 1 % puts "\\"
\
(auxiliary) 2 % puts "\$"
$
(auxiliary) 3 % puts "\t"

(auxiliary) 4 % puts "MyTab\tMyTab"
MyTab    MyTab
(auxiliary) 5 % puts "\n"


(auxiliary) 6 % puts "CR\nNextLine"
CR
NextLine
(auxiliary) 7 %
```

Agenda:

Word Structure

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

- Double-quotes prevent breaks:
  - set a “Funny word; has spaces”
- Curly braces prevent breaks and substitutions:
  - set a {nested {} braces}
- Backslashes quote special characters:
  - set a word\ with\ \$\ and\ space
- Substitutions don’t change word structure
  - set a “two words”
  - set b $a

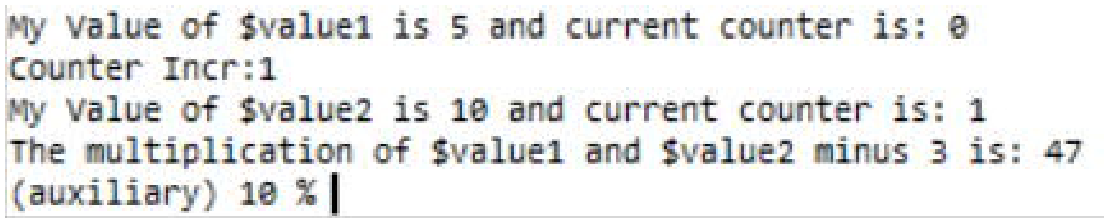


```
(auxiliary) 1 % set a "Funny word; has spaces"
Funny word; has spaces
(auxiliary) 2 % set a {nested {} braces}
nested {} braces
(auxiliary) 3 % set a word\ with\ \$\ and\ space
word with $ and space
(auxiliary) 4 % set a "two words"
two words
(auxiliary) 5 % set b $a
two words
(auxiliary) 6 %
```

Excercise Task

Initial Setup
set a variable named counter to 0
Set a variable named text to My Value
Set a variable named value1 to 5
Set a variable named value2 to 10

Expected Result



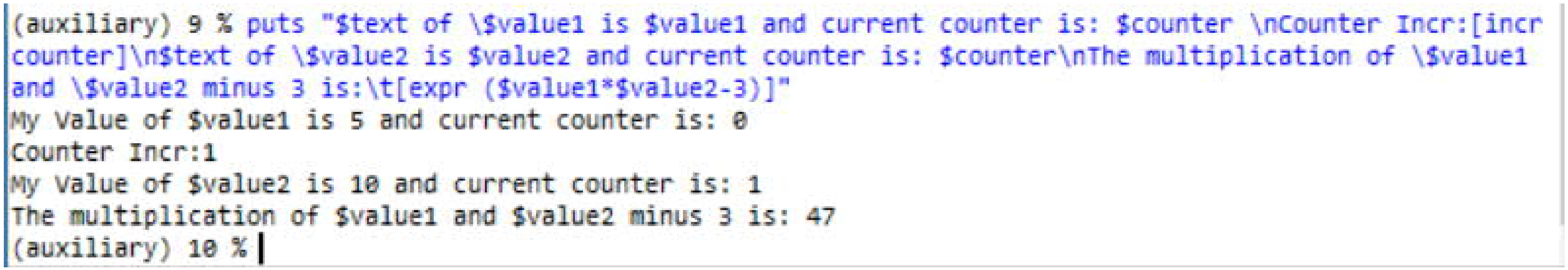
```
My Value of $value1 is 5 and current counter is: 0
Counter Incr:1
My Value of $value2 is 10 and current counter is: 1
The multiplication of $value1 and $value2 minus 3 is: 47
(auxiliary) 10 %
```

Following output is needed in the console:
First line: TextVariable of $value1 is ValueVariable and current counter is: value of counter
Second line: Counter Incr: Increase counter by one
Third line: TextVariable of $value2 is ValueVariable and current counter is: value of counter
Fourth line: The multiplication of $value1 and $value2 minus 3 is: result

Excercise Solution

Tcl code:

puts "$text of \$value1 is $value1 and current counter is: $counter \nCounter Incr:[incr counter]\n$text of \$value2 is $value2 and current counter is: $counter\nThe multiplication of \$value1 and \$value2 minus 3 is:\t[expr ($value1*$value2-3)]"



```
(auxiliary) 9 % puts "$text of \$value1 is $value1 and current counter is: $counter \nCounter Incr:[incr
counter]\n$text of \$value2 is $value2 and current counter is: $counter\nThe multiplication of \$value1
and \$value2 minus 3 is:\t[expr ($value1*$value2-3)]"
My Value of $value1 is 5 and current counter is: 0
Counter Incr:1
My Value of $value2 is 10 and current counter is: 1
The multiplication of $value1 and $value2 minus 3 is: 47
(auxiliary) 10 %
```

Agenda:

Procedures 1/6 – Simple Procedure

SIEMENS
Ingenuity for life

- Procedures behave just like built-in commands
- Scope: local and global variables
- proc command defines procedure

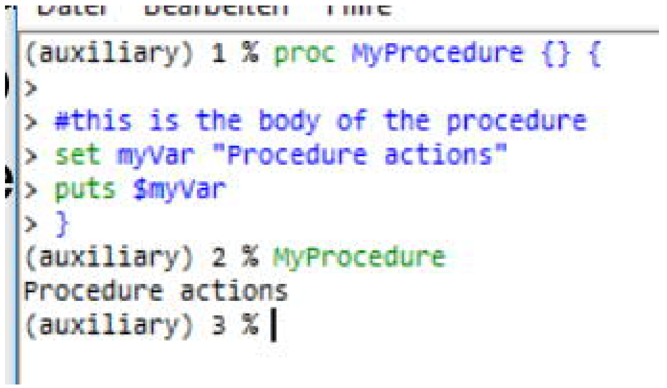
proc MyProcedure {} {
}

Sample procedure

```
proc MyProcedure {} {
#this is the body of the procedure
set myVar "Procedure actions"
puts $myVar
}
#Calling procedure
MyProcedure
```

Result

Procedure actions



```
(auxiliary) 1 % proc MyProcedure {} {
>
> #this is the body of the procedure
> set myVar "Procedure actions"
> puts $myVar
> }
(auxiliary) 2 % MyProcedure
Procedure actions
(auxiliary) 3 %
```

Procedures 2/6 – Local variables

- every procedure has his own variable table
- A variable which is set inside of a procedure is a local variable

proc MyProcedure {} {
#local variable
set localVar 5
}

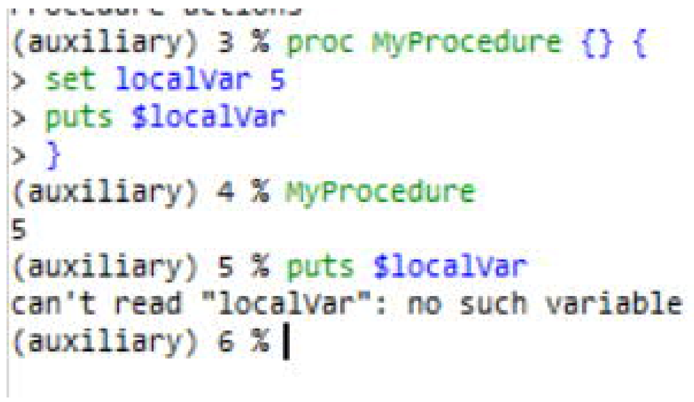
Sample procedure

```
proc MyProcedure {} {
#local variable
set localVar 5
puts $localVar
}
#Calling procedure
MyProcedure
puts $localVar
```

Result

5 -> due procedure call

Error - > no such variable



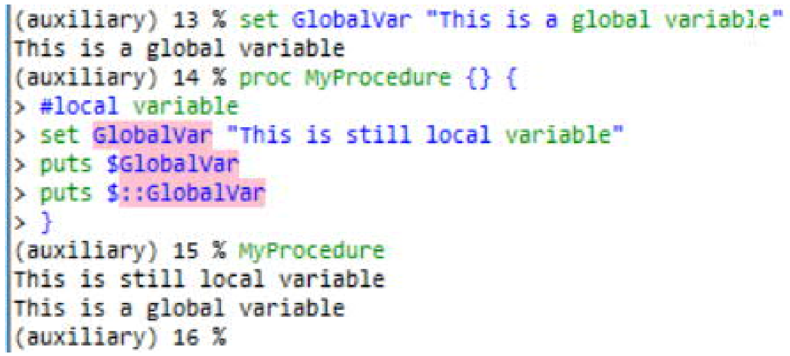
```
(auxiliary) 3 % proc MyProcedure {} {
> set localVar 5
> puts $localVar
> }
(auxiliary) 4 % MyProcedure
5
(auxiliary) 5 % puts $localVar
can't read "localVar": no such variable
(auxiliary) 6 %
```

Procedures 3/6 – Global variables

SIEMENS
Ingenuity for life

- A variable which is set in source level is automatically a global variable
- A variable which is set inside of a procedure is a local variable
- To use the variable in global context “global” or namespace “::” is needed

set GlobalVar “This is global variable”
proc MyProcedure {} {
global GlobalVar
puts $GlobalVar
<u>or</u>
puts $::GlobalVar
}



```
(auxiliary) 13 % set GlobalVar "This is a global variable"
This is a global variable
(auxiliary) 14 % proc MyProcedure {} {
> #local variable
> set GlobalVar "This is still local variable"
> puts $GlobalVar
> puts $::GlobalVar
> }
(auxiliary) 15 % MyProcedure
This is still local variable
This is a global variable
(auxiliary) 16 %
```

Sample procedure

```
	proc MyProcedure {} {
	#local variable
	set GlobalVar “This is still local variable”
	#global variable
	puts $GlobalVar
	puts $::GlobalVar
}
set GlobalVar “This is a global variable”
#Calling procedure
MyProcedure
```

Rule 2 to keep

Use unique names of variables to keep it simple!

Procedures 4/6 – Procedure with arguments

- Procedures can be called with multiple arguments (n-Arguments)
- If a procedure have arguments the call of procedure must contain arguments
  - proc MyProcedure {arg1 arg2} {…

Sample procedure

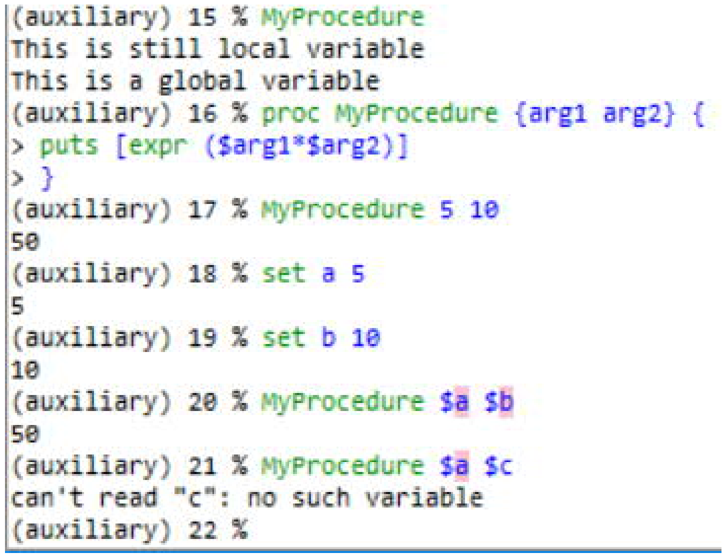
```
        proc MyProcedure {arg1 arg2} {
        puts [expr ($arg1*$arg2)]
}
#Calling procedure
MyProcedure  5 10
set a 5
set b 10
MyProcedure  $a $b
MyProcedure  $a $c
```

Result

50 – with direct values

50 – with assigned variables

Error – no such variable



```
(auxiliary) 15 % MyProcedure
This is still local variable
This is a global variable
(auxiliary) 16 % proc MyProcedure {arg1 arg2} {
> puts [expr ($arg1*$arg2)]
> }
(auxiliary) 17 % MyProcedure 5 10
50
(auxiliary) 18 % set a 5
5
(auxiliary) 19 % set b 10
10
(auxiliary) 20 % MyProcedure $a $b
50
(auxiliary) 21 % MyProcedure $a $c
can't read "c": no such variable
(auxiliary) 22 %
```

Procedures 5/6 – Procedure with arguments and default

- A procedure can have default values for input parameters
- It is an optional argument

  proc MyProcedure {arg1 {arg2 5}} {…

Sample procedure

```
        proc MyProcedure {arg1 {arg2 5}} {
        puts [expr ($arg1*$arg2)]
}
#Calling procedure
MyProcedure 5 10
set a 5
set b 10
MyProcedure $a $b
MyProcedure $a $c
MyProcedure $a
```

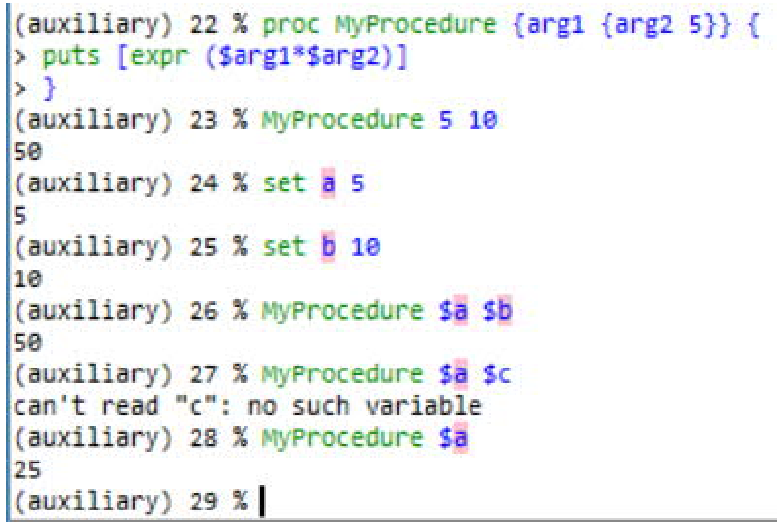
Result

50 – with direct values

50 – with assigned variables

Error – no such variable

25 – second optional argument with default will be used



```
(auxiliary) 22 % proc MyProcedure {arg1 {arg2 5}} {
> puts [expr ($arg1*$arg2)]
> }
(auxiliary) 23 % MyProcedure 5 10
50
(auxiliary) 24 % set a 5
5
(auxiliary) 25 % set b 10
10
(auxiliary) 26 % MyProcedure $a $b
50
(auxiliary) 27 % MyProcedure $a $c
can't read "c": no such variable
(auxiliary) 28 % MyProcedure $a
25
(auxiliary) 29 %
```

Exercise Task

Calculate the area of a rectangle and perimeter based on 2 input variables
2 procedures are needed
First one named **Calculate** is calculation of the area and perimeter with fixed arguments
Second one named **ValidateInputValues** to call by the user with 2 arguments for the values
Condition: If the call is without any arguments standard values will be used 10 and 15

Expected Output in the console after calling ValidateInputValues with **45** and **42.2**
InputValue1: x
InputValue2: x
Calulated Area: x
Calculated Perimeter: x

Exercise Solution

SIEMENS
Ingenuity for life

Calculate is calculation of the area and scope with fixed arguments

```
proc Calculate {arg1 arg2} {
puts „Inputvalue1: $arg1“
puts „Inputvalue2: $arg2“
puts „Calculated Area: [expr ($arg1*$arg2)]“
puts „Calculated Scope: [expr ($arg1*2+$arg2*2)]“
}
```

ValidateInputValues to call by the user with 2 arguments for the values

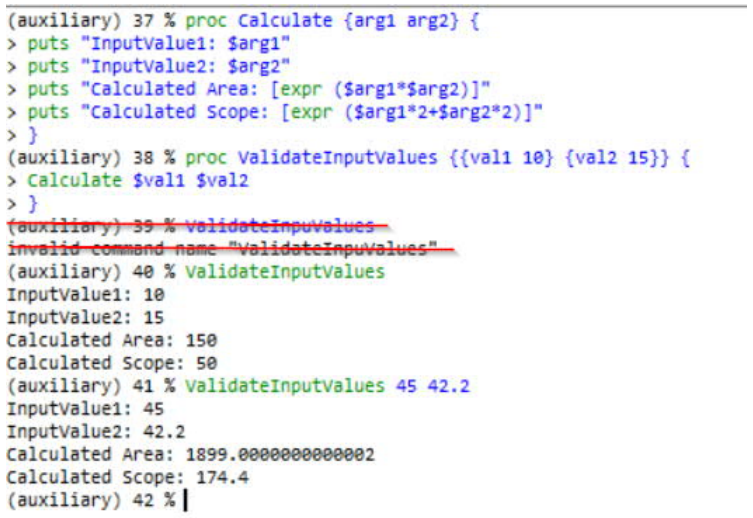
```
proc ValidateInputValues {{val1 10} {val2 15}} {
Calculate $val $val2
}
```

Calling without any arguments

```
ValidateInputValues
InputValue1: 10
InputValue2: 15
Calulated Area: 150
Calculated Scope: 50
```

Output in the console after calling ValidateInputValues with **45** and **42.2**

```
InputValue1: 45
InputValue2: 42.2
Calulated Area: 1899.000…
Calculated Scope: 174.4
```



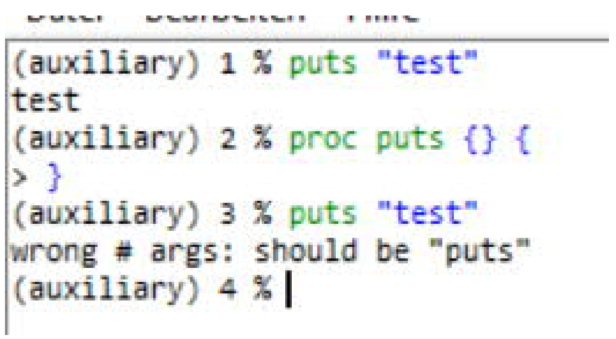
```
(auxiliary) 37 % proc Calculate {arg1 arg2} {
> puts "InputValue1: $arg1"
> puts "InputValue2: $arg2"
> puts "Calculated Area: [expr ($arg1*$arg2)]"
> puts "Calculated Scope: [expr ($arg1*2+$arg2*2)]"
> }
(auxiliary) 38 % proc ValidateInputValues {{val1 10} {val2 15}} {
> Calculate $val1 $val2
> }
~~(auxiliary) 39 % ValidateInpuValues~~
~~invalid command name "ValidateInpuValues"~~
(auxiliary) 40 % ValidateInputValues
InputValue1: 10
InputValue2: 15
Calculated Area: 150
Calculated Scope: 50
(auxiliary) 41 % ValidateInputValues 45 42.2
InputValue1: 45
InputValue2: 42.2
Calculated Area: 1899.0000000000002
Calculated Scope: 174.4
(auxiliary) 42 %
```

Procedures 6/6 – Overwrite existing procedures

Existing procedure can be overwritten
Q: What is Rule 1?
A: Everything is a string!

It‘s not recommend to overwrite existing functionality but sometimes it can be necessary

Overwriting e.g. puts command will result in no more longer possibility to output lines to the console



```
(auxiliary) 1 % puts "test"
test
(auxiliary) 2 % proc puts {} {
> }
(auxiliary) 3 % puts "test"
wrong # args: should be "puts"
(auxiliary) 4 %
```

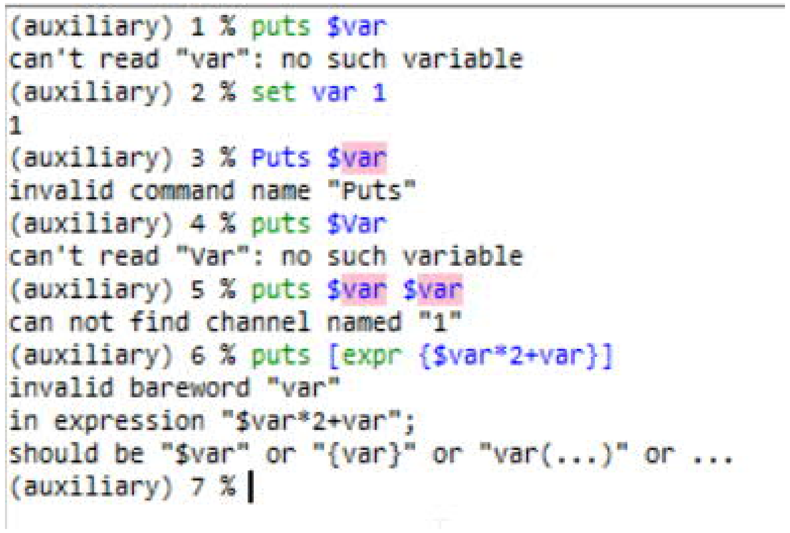
Agenda:

Errorhandling

- Errors normally abort commands in progress
- application displays message (Console or Syslog NX CAM in post processor context)

Typical issues:

- Local/ global variables
  - “no such variable”
- Case sensitive
  - Take care of lower and uppercase
  - If calling standard commands use documentation or Tcl auto complete functionality (if available)
  - Read the call stack to understand the issue
- Invalid command
  - Check command name
- Wrong number of arguments when calling a procedure
- Syntax error in expression



```
(auxiliary) 1 % puts $var
can't read "var": no such variable
(auxiliary) 2 % set var 1
1
(auxiliary) 3 % Puts $var
invalid command name "Puts"
(auxiliary) 4 % puts $Var
can't read "Var": no such variable
(auxiliary) 5 % puts $var $var
can not find channel named "1"
(auxiliary) 6 % puts [expr {$var*2+var}]
invalid bareword "var"
in expression "$var*2+var";
should be "$var" or "{var}" or "var(...)" or ...
(auxiliary) 7 %
```

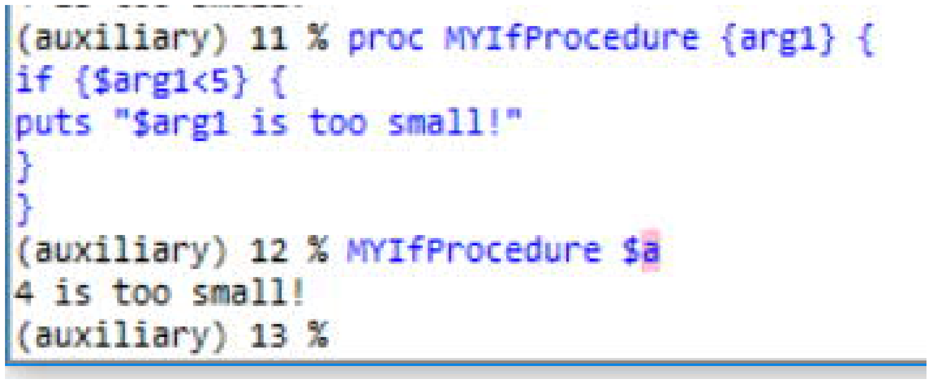
Agenda:

If conditions

- C-like appearance
- Just commands that take Tcl scripts as arguments
- Operators: ==,!=,<,<=,>,>=,&&,||
- Take care of braces

Sample procedure

```
        proc MyIfProcedure {arg1} {
        if {$arg1<5} {
        puts “$arg1 is too small!”
        }
}
set a 4
#Calling procedure
MyIfProcedure  $a
```



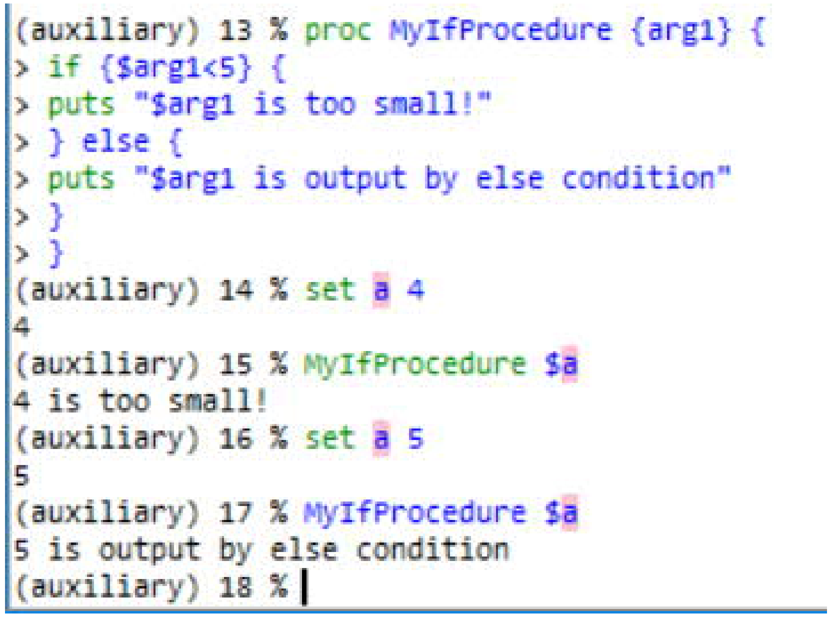
```
(auxiliary) 11 % proc MYIfProcedure {arg1} {
if {$arg1<5} {
puts "$arg1 is too small!"
}
}
(auxiliary) 12 % MYIfProcedure $a
4 is too small!
(auxiliary) 13 %
```

If conditions with else

- C-like appearance
- Just commands that take Tcl scripts as arguments
- Operators: ==,!=,<,<=,>,>=,&&,||
- Take care of braces

Sample procedure

```
        proc MyIfProcedure {arg1} {
        if {$arg1<5} {
        puts "$arg1 is too small!"
        } else {
        puts "$arg1 is output by else condition"
        }
}
set a 4
#Calling procedure
MyIfProcedure $a
set a 5
MyIfProcedure $a
```



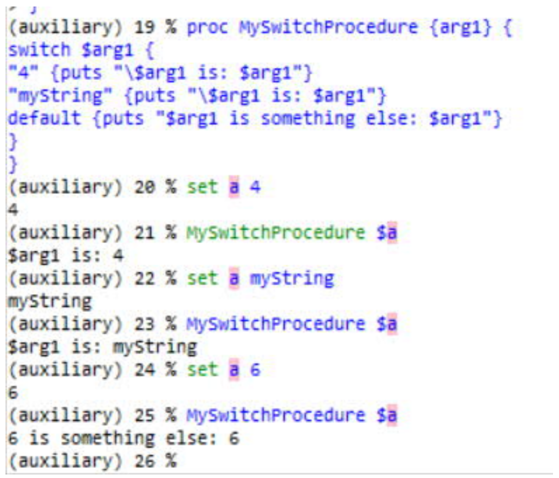
```
(auxiliary) 13 % proc MyIfProcedure {arg1} {
> if {$arg1<5} {
> puts "$arg1 is too small!"
> } else {
> puts "$arg1 is output by else condition"
> }
> }
(auxiliary) 14 % set a 4
4
(auxiliary) 15 % MyIfProcedure $a
4 is too small!
(auxiliary) 16 % set a 5
5
(auxiliary) 17 % MyIfProcedure $a
5 is output by else condition
(auxiliary) 18 %
```

Switch condition

SIEMENS
Ingenuity for life

- C-like appearance
- Just commands that take Tcl scripts as arguments
- Can be variable values or command substitution
- Take care of braces
- Default option

```
Sample procedure
        proc MySwitchProcedure {arg1} {
        switch $arg1 {
        “4” {puts “\$arg1 is: $arg1}
        “myString” {puts “\$arg1 is: $arg1}
        default {puts “\$arg1 is something else: $arg1}
        }
}
set a 4
MySwitchProcedure $a
set a myString
MySwitchProcedure $a
set a 1899
MySwitchProcedure $a
```



```
(auxiliary) 19 % proc MySwitchProcedure {arg1} {
switch $arg1 {
"4" {puts "\$arg1 is: $arg1"}
"myString" {puts "\$arg1 is: $arg1"}
default {puts "$arg1 is something else: $arg1"}
}
}
(auxiliary) 20 % set a 4
4
(auxiliary) 21 % MySwitchProcedure $a
$arg1 is: 4
(auxiliary) 22 % set a myString
myString
(auxiliary) 23 % MySwitchProcedure $a
$arg1 is: myString
(auxiliary) 24 % set a 6
6
(auxiliary) 25 % MySwitchProcedure $a
6 is something else: 6
(auxiliary) 26 %
```

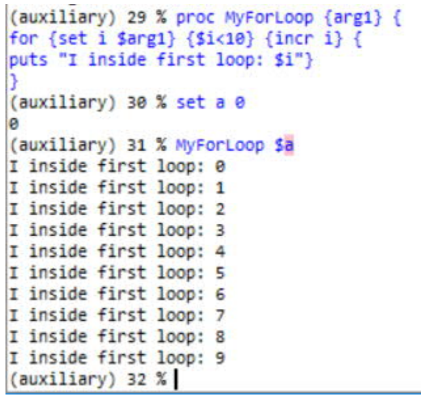
Other Control Structures

SIEMENS

Ingenuity for life

for

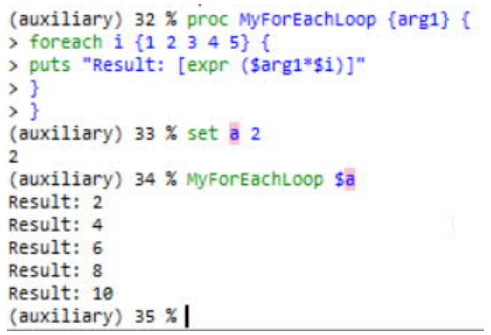
```
proc MyForLoop {arg1} {
for {set i $arg1} {$i < 10} {incr i} {
puts "I inside first loop: $i" }
}
}
set a 0
MyForLoop $a
```



```
(auxiliary) 29 % proc MyForLoop {arg1} {
for {set i $arg1} {$i<10} {incr i} {
puts "I inside first loop: $i"}
}
(auxiliary) 30 % set a 0
0
(auxiliary) 31 % MyForLoop $a
I inside first loop: 0
I inside first loop: 1
I inside first loop: 2
I inside first loop: 3
I inside first loop: 4
I inside first loop: 5
I inside first loop: 6
I inside first loop: 7
I inside first loop: 8
I inside first loop: 9
(auxiliary) 32 %
```

foreach

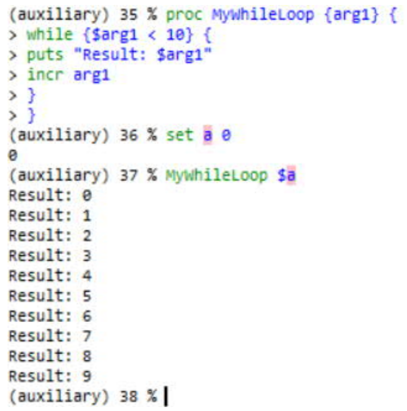
```
proc MyForEachLoop {arg1} {
foreach i {1 2 3 4 5} {
puts „Result: [expr ($arg1*$i)]"
}
}
set a 2
MyForEachLoop $a
```



```
(auxiliary) 32 % proc MyForEachLoop {arg1} {
> foreach i {1 2 3 4 5} {
> puts "Result: [expr ($arg1*$i)]"
> }
> }
(auxiliary) 33 % set a 2
2
(auxiliary) 34 % MyForEachLoop $a
Result: 2
Result: 4
Result: 6
Result: 8
Result: 10
(auxiliary) 35 %
```

while

```
proc MyWhileLoop {arg1} {
while {$arg1 < 10} {
puts „Result: $arg1„
incr arg1
}
}
set a 0
MyWhileLoop $a
```



```
(auxiliary) 35 % proc MyWhileLoop {arg1} {
> while {$arg1 < 10} {
> puts "Result: $arg1"
> incr arg1
> }
> }
(auxiliary) 36 % set a 0
0
(auxiliary) 37 % MyWhileLoop $a
Result: 0
Result: 1
Result: 2
Result: 3
Result: 4
Result: 5
Result: 6
Result: 7
Result: 8
Result: 9
(auxiliary) 38 %
```

Rule 3 to keep

Keep it simple to read!

Exercise Task

Task 1:
- A small program is needed to calculate area/perimeter for rectangle, square and circle
- The user want to call easily a procedure where he can input the option and for rectangle/ square the length and for the circle the diameter
- Dependent on the selected options following output is expected in the console:

    Output Option: circle/ rectangle/ square
    If circle: Entered Diameter
    If square: Entered length
    If rectangle: entered length both sides
    Calculated Area: x
    Calculated perimeter: x

Task 2:
- A procedure is needed called MOM_output_literal and MOM_output_text
- Initially a sequence number variable is set to 1
- If using MOM_output_literal in the console the sequence number will be increased by one, MOM_output_text will output only the string
- Use a loop in the console to output 100 lines

Example:
MOM_output_literal „LineText“
Expected Result: N(n) LineText

Agenda:

Additional Tcl standard commands

- String manipulation commands:

string	**format**	**split**	**regexp**
scan	**join**		

- File I/O commands:

Open	**seek**	**file**	**close**
tell	**glob**	**gets**	**flush**
cd	**read**	**eof**	**pwd**

- Subprocesses with exec command:

```
exec grep foo << $input | wc
```

Agenda:

Summary

SIEMENS
Ingenuity for life

- Script = commands separated by new line, semi-colons
- Command = words separated by white spaces
- $ causes variable substitution
- [] causes command substitution
- “” quotes whitespace and semicolons
- {} quotes all special characters
- \ quotes next character, provides C-like substitutions
- # for comments (must be at the beginning of command)

SIEMENS
Ingenuity for life

Thomas Jenensch

Product Portfolio Lead NX CAM Infrastructure
Siemens Industry Software

thomas.jenensch@siemens.com

SIEMENS

Ingenuity for life

Thank you.