PARADIGMS AND COMPUTER PROGRAMMING FUNDAMENTALS (PCPF) ITC305 2022-23

Subject In-charge

Ms. Aaysha Shaikh

Professor Dept. of Information Technology

SFIT Room No. 332

email: aayshashaikh@sfit.ac.in

Module 6

Lecture 3

Alternative Paradigms: Scripting Languages

Contents

- Common characteristics,
- Different Problem domains for using scripting,
- Use of scripting in Web development server and clients side scripting,
- Innovative features of scripting languages –
  - – Names and Scopes,

– string and pattern manipulation,

– data types,

– object orientation.

Innovative Features of Scripting Languages

- Innovative features of scripting languages
  - Names and Scopes,
  - string and pattern manipulation,
  - data types,
  - object orientation.

Innovative feature: Names and Scopes

- Most scripting languages except Scheme do not require variables to be declared
  - Perl and JavaScript permit optional declarations - sort of compiler checked documentation
  - Perl can be run in a mode (use strict ’vars’) that requires declarations
- With or without declarations, most scripting languages use dynamic

typing

– The interpreter can perform type checking at run time, or coerce values when appropriate

– Tcl is unusual in that all values—even lists—are represented internally as strings

- In a Tcl script, everything is a string, and Tcl assigns no meaning to any string, making it a typeless language. Example:

```
set myVariable 18
puts $myVariable
```

Declaration and Scope Example

```
var foo = "I'm global";
var bar = "So am I";

function () {
    var foo = "I'm local, the previous 'foo'
                          didn’t notice a
                          thing";
    var baz = "I'm local, too";
    function () {
        var foo = "I'm even more local, all
        three 'foos' have different values"; baz
        = "I just changed 'baz' one scope
        higher, but it's still not global";
        bar = "I just changed the global
        'bar’ variable";
    }
}
```

- Use of var is optional in JavaScript: • If you **use var** the variable is declared within the scope you are in (e.g. of the function), i.e. local.

- If you use var in the global scope, the variable is truly global and cannot be deleted.
- If you don't use var, the variable bubbles up through the layers of scope until it encounters a variable by the given name or the global object where it then attaches.
- It is then very similar to a global variable.

Dynamic Typing Example

- Python is a dynamically typed language. It means that the type of a variable is allowed to change over its lifetime.
- Other dynamically typed languages are -Perl, Ruby, PHP, JavaScript etc.

•JavaScript example:

var c= 5;

c="I am string now";

- Python Example:
- # variable a is assigned to a string

a ="hello"

print(type(a)) Output: String •# variable a is assigned to an integer a = 5

print(type(a)) Output: Integer •# simple function

```
def add(a, b):
    return a + b
```

• # calling the function with string

print(add('hello ', 'world’))

Output: hello world

•# calling the function with integer

print(add(2, 4))

Output: 6

Innovative feature: Nesting and Scope

- Nesting and scoping conventions vary quite a bit
  - Scheme, Python, JavaScript provide the classic combination of nested subroutines and static (lexical) scope
  - Tcl allows subroutines to nest, but uses dynamic scope
  - Named subroutines (methods) do not nest in PHP or Ruby
  - Perl and Ruby join Scheme, Python, and JavaScript in providing firstclass anonymous local subroutines
    - Nested blocks are statically scoped in Perl

• In Ruby, they are part of the named scope in which they appear

Innovative feature: Nesting and Scope

• In Perl, all variables are global unless otherwise specified. • In Python, all variables are local by default, unless explicitly imported

• In PHP, local unless explicitly imported.

• PHP and the major glue languages (Perl, Tcl, Python, Ruby) all have sophisticated namespace rules

  – mechanisms for information hiding and the selective import of names from separate modules

Innovative feature: Nesting and Scope

- Global scope

```
var greeting = 'Hello World!';
function greet() {
  console.log(greeting);
}
// Prints 'Hello World!'
greet();
```

- Local scope

```
function greet() {
  var greeting = 'Hello World!';
  console.log(greeting);
}
// Prints 'Hello World!'
greet();
```

- Block scope

```
{
  let greeting = 'Hello World!';
  var lang = 'English'; // var not block scoped
console.log(greeting); // Prints 'Hello World!' }
console.log(lang); // Prints 'English'
```

- Nested scope

```
var name = 'Peter';
function greet() {
  var greeting = 'Hello';
  {
    let lang = 'English';
    console.log(`${lang}: ${greeting}
  ${name}`); }
}
greet();
```

Static Scope: JavaScript

Innovative feature: Scope in Python

- Lexical Scope (also known as Static Scope) literally means that scope is determined at the lexing time (compiling) rather than at runtime.

- Here the console.log(number) will always print 42 no matter from where function printNumber() is called.

- In static scoping the compiler first searches in the current block, then in global variables (Top environment)

```
let number = 42;
function printNumber() {
console.log(number);
document.write(number); }
function log() {
let number = 54;
printNumber();
}
log(); // Prints 42
```

```
# Python program to
demonstrate # scope of variable
# In Python, all variables are local
by default, unless explicitly
imported:  a = 1

# Uses global because there is no local
'a' def f():
    print('Inside f() : ', a)

# Variable 'a' is redefined as a
local def g():
    a = 2
    print('Inside g() : ', a)
 #Uses global keyword to modify global
'a' def h():
    global a
    a = 3
    print('Inside h() : ', a)
```

```
‡ Global scope
rint('global : ', a)
)
rint('global : ', a)
()
rint('global : ', a)
()
rint('global : ', a)
```

OUTPUT:

```
global : 1
Inside f() : 1
global : 1
Inside g() : 2
global : 1
Inside h() : 3
global : 3
```

Innovative feature: Scope in Python

```
# Python program to demonstrate
# nonlocal keyword
print ("Value of a using nonlocal is : ", end
="") def outer():
    a = 5
    def inner():
        nonlocal a
        a = 10
    inner()
    print (a)

outer()

# demonstrating without non local
# inner loop not changing the value of outer
a # prints 5
print ("Value of a without using nonlocal is :
",  end ="")
def outer():
    a = 5
    def inner():
        a = 10
    inner()
    print (a)
outer()
```

OUTPUT:

```
Value of a using nonlocal is : 10 Value of
a  without using nonlocal is : 5
```

Innovative feature: Scope in Python

•In Python, all variables are local by default, unless explicitly imported:

```
i=1;
j=3
def outer():
        def middle(k):
                def inner():
                global i #from main program, not outer
        i = 4
        inner()
        return i,j,k #3 element tuple
        i=2
        return middle(j) #old (global) j
print(outer())
print(i,j)
```

• This prints: (2,3,3)

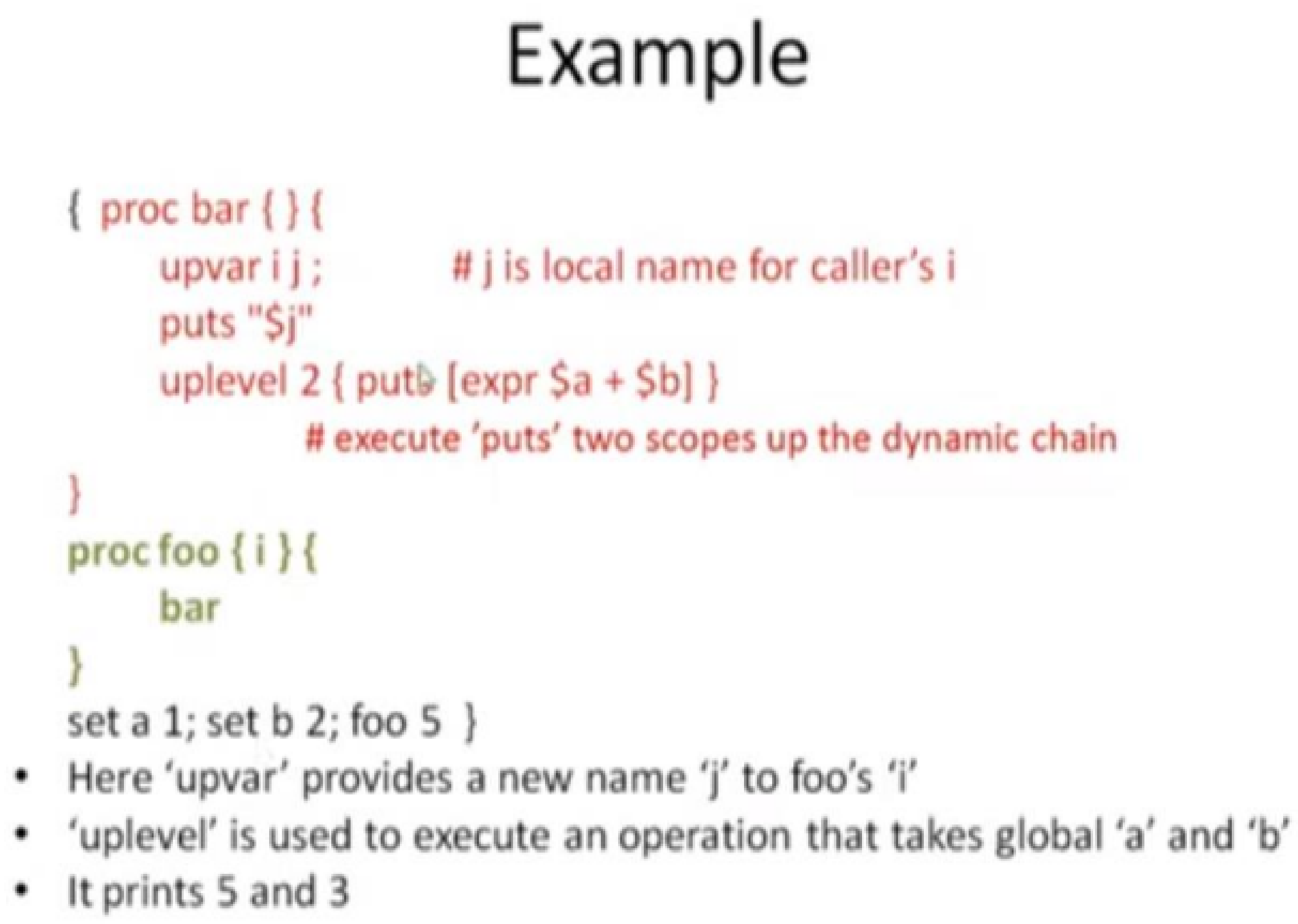
By default, there is no way for a nested scope to write to a non-local or non-global scope - so in previous example, inner could not modify outer’s i variable.

Innovative feature: Scope in Tcl

• Tcl uses dynamic scoping, but in an odd way - the programmer must request other scopes explicitly:

```
upvar i j ;#j is the local name for caller’s I
uplevel 2 {puts [expr $a + $b] }
                    #executes ‘puts’two scopes up on dynamic chain
```

- Employes dynamic scoping.
- Variables are not accessed automatically.
- They must be expilicitly asked by programmers.
- ‘upvar’ and ‘uplevel’ commands are used for this.
- ‘upvar’ command accesses a variable in specified frame and gives it a new name.
- ‘uplevel’ command provides a nested Tcl scripts.
- This script in executed in the context of specified frame using call-by-name mode.



Example

```
{ proc bar { } {
        upvar i j ;          # j is local name for caller's i
        puts "$j"
        uplevel 2 { puts [expr $a + $b] }
                # execute 'puts' two scopes up the dynamic chain
}
proc foo { i } {
        bar
}
set a 1; set b 2; foo 5  }
```

- Here 'upvar' provides a new name 'j' to foo's 'i'
- 'uplevel' is used to execute an operation that takes global 'a' and 'b'
- It prints 5 and 3

Innovative feature: Pattern Matching

- Regular expressions (REs) are present in many scripting languages and related tools employ extended versions of the notation
- Regular Expression provides an ability to match a "string of text" in a very flexible and concise manner.
- A "string of text" can be further defined as a single character, word, sentence or particular pattern of characters.

[ ]: Matches any one of a set characters
[_] with hyphen: Matches any one of a range characters
^: The pattern following it must occur at the beginning of each line ^ with
[ ] : The pattern must not contain any character in the set specified $:
The pattern preceding it must occur at the end of each line . (dot):
Matches any one character
\ (backslash): Ignores the special meaning of the character following
it *: zero or more occurrences of the previous character
(dot).*: Nothing or any numbers of characters.

Innovative feature: Pattern Matching

- grep, the stand-alone Unix is a pattern-matching tool, is another useful program that you might be familiair with
- In general, two main groups.

- The first group includes awk, egrep (the most widely used of several different versions of grep), the regex routines of the C standard library, and older versions of Tcl
    - These implement REs as defined in the POSIX standard
- Languages in the second group follow the lead of Perl, which provides a large set of extensions, sometimes referred to as “advanced REs”

Innovative feature: Pattern Matching with RE

• [ ] : Matches any one of a set characters

Ex1: $grep “New[abc]” filename

It specifies the search pattern as : Newa , Newb or Newc

Ex2: $grep “[aA]g[ar][ar]wal” filename

It specifies the search pattern as: Agarwal , Agaawal , Agrawal , Agrrwal agarwal , agaawal , agrawal , agrrwal

• Use [ ] with hyphen: Matches any one of a range characters

Ex1: $grep “New[a-e]” filename

It specifies the search pattern as: Newa , Newb or Newc , Newd, Newe

Ex2: $grep "New[0-9][a-z]" filename

It specifies the search pattern as: New followed by a number and then an alphabet.: New0d, New4f etc

Innovative feature: Pattern Matching Ex

- Use ^: The pattern following it must occur at the beginning of each line

Ex1: $grep "^san" filename

Search lines beginning with san. It specifies the search pattern as: sanjeev ,sanjay, sanrit , sanchit , sandeep etc.

- Use ^ with [ ]: The pattern must not contain any character in the set specified

Ex1: $grep "New[^a-c]" filename

It specifies the pattern containing the word "New" followed by any character other than an 'a','b', or 'c'

Pattern matching: greedy matches

- If multiple matches are possible, it will take the "left-most longest" possible one.

- For example, in the string **abcbcbcde**, the pattern /(bc)+/ will match abcbcbcde.
- This is knows as the “greedy” match.

Script: PHP

- Using

```
<?php
```

Global variable

Server-side

```
echo "<p>Hello,
World!</p>\n";
$x=15;
$y=30;
$z=$x+$y;
echo "Sum: ",$z; ?>
```

```
<?php
echo "<p>Hello,
World!</p>\n"; $x=15;
$y=30;
$z=$x+$y;
```

```
<?php
```

Local variables

```
echo "<p>Hello,
World!</p>\n";
$x=15;
$y=30;
$z=$x+$y;
```

```
function sayHello(){
echo "Sum: ",$z, "\n"; }
sayHello();
?>
```

```
function sayHello(){
echo "Sum:
",$GLOBALS['z'], "\n"; }
```

```
sayHello(); ?>
```

Server-side Script: PHP

- Using for loop <?php

```
echo "<p>Hello,
World!</p>\n"; $x=15;
$y=30;
$z=$x+$y;
function sayHello(){
    for($n=1;$n<=10;$n++){
      echo "Sum: ",$GLOBALS['z'],
     "\n"; }
 }
sayHello();
?>
```

- Using if-else

```
<?php
   echo "<p>Hello,
World!</p>\n"; $x=15;
$y=30;
$z=$x+$y;
function sayHello(){
  if($GLOBALS['z']>40) {
    for($n=1;$n<=10;$n++) {
       echo "Sum: ",$GLOBALS['z'],
          "\n"; }
   }
   else {
       echo "Hi"; }
  }
```

```
sayHello(); ?>
```

Thank You