

Principles of Programming Languages

<http://www.di.unipi.it/~andrea/Didattica/PLP-15/>

Prof. Andrea Corradini

Department of Computer Science, Pisa

Lesson 30

- Scripting languages

Origin of Scripting Languages

- Modern scripting languages have two principal sets of ancestors.
 - command interpreters or “shells” of traditional batch and “terminal” (command-line) computing
 - IBM’s JCL, MS-DOS command interpreter, Unix **sh** and **cs**
 - various tools for text processing and report generation
 - IBM’s RPG, and Unix’s **sed** and **awk**.
- From these evolved
 - **Rexx**, IBM’s “Restructured Extended Executor,” which dates from 1979
 - Perl, originally devised by Larry Wall in the late 1980s, and now the most widely used general purpose scripting language.
 - Other general purpose scripting languages include Tcl (“tickle”), Python, Ruby, VBScript (for Windows) and AppleScript (for the Mac)

What is a Scripting Language

- Common Characteristics
 - Both batch and interactive use
 - Economy of expression
 - Lack of declarations; simple scoping rules.
 - Flexible dynamic typing
 - Easy access to other programs
 - Sophisticated pattern matching and string manipulation
 - High level data types

Problem Domains

- Some general purpose languages—Scheme and Visual Basic in particular—are widely used for scripting
- Conversely, some scripting languages, including Perl, Python, and Ruby, are intended by their designers for general purpose use, with features intended to support “programming in the large”
 - modules, separate compilation, reflection, program development environments
- For the most part, however, scripting languages tend to see their principal use in ***well defined problem domains***

Problem Domains: Shell Languages

- Shell Languages have features designed for interactive use
- Provide a wealth of mechanisms to manipulate file names, arguments, and commands, and to glue together other programs
 - Most of these features are retained by more general scripting languages
- Typical mechanisms supported:
 - Filename and Variable Expansion
 - Tests, Queries, and Conditions
 - Pipes and Redirection
 - Quoting and Expansion
 - Functions
 - The #! Convention

```
for fig in *.eps
do
    target=${fig%.eps}.pdf
    if [ $fig -nt $target ]
    then
        ps2pdf $fig
    fi
done
```

Problem Domains:

Text Processing and Report Generation

sed: Unix's *stream editor*

- No variables, no state: just a powerful filter
- **s/_/_/** substitution command

```
# label (target for branch):
:top
/<[hH] [123]>.*<\/[hH] [123]>/ {           ;# match whole heading
    h                                     ;# save copy of pattern space
    s/\(<\/[hH] [123]>\).*$/\1/           ;# delete text after closing tag
    s/^\.*\(<[hH] [123]>\)/\1/           ;# delete text before opening tag
    p                                     ;# print what remains
    g                                     ;# retrieve saved pattern space
    s/<\/[hH] [123]>//                   ;# delete closing tag
    b top
}                                           ;# and branch to top of script
/<[hH] [123]>/ {                             ;# match opening tag (only)
    N                                     ;# extend search to next line
    b top
}                                           ;# and branch to top of script
d                                           ;# if no match at all, delete
```

Figure 13.1 Script in `sed` to extract headers from an HTML file. The script assumes that opening and closing tags are properly matched, and that headers do not nest.

Problem Domains:

Text Processing and Report Generation

- **awk**

- adds variables, state and richer control structures

```
/<[hH][123]>/ {
  # execute this block if line contains an opening tag
  do {
    open_tag = match($0, /<[hH][123]>/)
    $0 = substr($0, open_tag)      # delete text before opening tag
                                    # $0 is the current input line

    while (!/<\/[hH][123]>/) {      # print interior lines
      print                        #   in their entirety
      if (getline != 1) exit
    }
    close_tag = match($0, /<\/[hH][123]>/) + 4

    print substr($0, 0, close_tag) # print through closing tag
    $0 = substr($0, close_tag + 1) # delete through closing tag
  } while (/<[hH][123]>/)          # repeat if more opening tags
}
```

Figure 13.2 Script in awk to extract headers from an HTML file. Unlike the sed script, this version prints interior lines incrementally. It again assumes that the input is well formed.

From bash/sed/awk to Perl

- Originally developed by Larry Wall in 1987
- Unix-only tool, meant primarily for text processing (the name stands for “practical extraction and report language”)
- Over the years has grown into a large and complex language, ported to all operating systems: it is the most popular and widely used scripting language
- It is also fast enough for much general purpose use, and includes
 - separate compilation, modularization, and dynamic library mechanisms appropriate for large-scale projects

```
while (>) {                                # iterate over lines of input
next if !/<[hH][123]>/;                    # jump to next iteration
while (!/<\/[hH][123]>/) { $_ .= <>; }      # append next line to $_
s/.*?(/[hH][123]>.*?\/[hH][123]>)//s;
# perform minimal matching; capture parenthesized expression in $1
print $1, "\n";
redo unless eof;    # continue without reading next line of input
}
```


“Force quit” in Perl

```
#!/usr/bin/perl
$#ARGV == 0 || die "usage: $0 pattern\n";
open(PS, "ps -w -w -x -o'pid,command' |"); # 'process status' command
<PS>; # discard header line
while (<PS>) {
    @words = split; # parse line into space-separated words
    if (/ $ARGV[0]/i && $words[0] ne $$) {
        chomp; # delete trailing newline
        print;
        do {
            print "? ";
            $answer = <STDIN>;
        } until $answer =~ /^[yn]/i;
        if ($answer =~ /y/i) {
            kill 9, $words[0]; # signal 9 in Unix is always fatal
            sleep 1; # wait for 'kill' to take effect
            die "unsuccessful; sorry\n" if kill 0, $words[0];
        }
        # kill 0 tests for process existence
    }
}
```

Figure 13.5 Script in Perl to “force quit” errant processes. Perl’s text processing features allow us to parse the output of `ps`, rather than filtering it through an external tool like `sed` or `awk`.

Problem Domains: “Glue” Languages and General Purpose Scripting

- **Tcl**
 - Developed in the late 1980s at UC, Berkeley (Prof. John Ousterhout)
 - The initial motivation for Tcl (“tool command language”) was the desire for an extension language that could be embedded in the tools for VLSI developed by the group, providing them with uniform command syntax and reducing the complexity of development and maintenance
 - Tcl quickly evolved beyond its emphasis on command extension to encompass “glue” applications as well
- In comparison to Perl, Tcl is somewhat more verbose
 - It makes less use of punctuation, and has fewer special cases
- All data represented internally as strings

“Force quit” in Tcl

```
if {$argc != 1} {puts stderr "usage: $argv0 pattern"; exit 1}
set PS [open "|/bin/ps -w -w -x -opid,command" r]

gets $PS                                     ;# discard header line
while {![eof $PS]} {
    set line [gets $PS]                       ;# returns blank line at eof
    regexp {[0-9]+} $line proc
    if {[regexp [lindex $argv 0] $line] && [expr $proc != [pid]]} {
        puts -nonewline "$line? "
        flush stdout                          ;# force prompt out to screen
        set answer [gets stdin]
        while {![regexp -nocase {^[yn]} $answer]} {
            puts -nonewline "? "
            flush stdout
            set answer [gets stdin]
        }
        if {[regexp -nocase {^y} $answer]} {
            set stat [catch {exec kill -9 $proc}]
            exec sleep 1
            if {$stat || [exec ps -p $proc | wc -l] > 1} {
                puts stderr "unsuccessful; sorry"; exit 1
            }
        }
    }
}
}
```

Figure 13.6 Script in Tcl to “force quit” errant processes. Compare to the Perl script of Figure 13.5.

Problem Domains

“Glue” Languages and General Purpose Scripting

- **Rexx** (1979) is considered the first of the general purpose scripting languages
- **Perl** and **Tcl** are roughly contemporaneous: late 1980s
 - Perl was originally intended for glue and text processing applications
 - Tcl was originally an extension language, but soon grew into glue applications
- **Python** was originally developed by Guido van Rossum at CWI in Amsterdam, the Netherlands, in the early 1990s
 - Recent versions of the language are owned by the Python Software
 - All releases are Open Source.
 - Object oriented
- **Ruby**
 - Developed in Japan in early 1990
 - English documentation published in 2001
 - Smalltalk-like object orientation

“Force quit” in Python

```
import sys, os, re, time
if len(sys.argv) != 2:
    sys.stderr.write('usage: ' + sys.argv[0] + ' pattern\n')
    sys.exit(1)

PS = os.popen("/bin/ps -w -w -x -o'pid,command'")
line = PS.readline()          # discard header line
line = PS.readline().rstrip() # prime pump
while line != "":
    proc = int(re.search('\S+', line).group())
    if re.search(sys.argv[1], line) and proc != os.getpid():
        print line + '? ',
        answer = sys.stdin.readline()
        while not re.search('^[yn]', answer, re.I):
            print '? ',          # trailing comma inhibits newline
            answer = sys.stdin.readline()
        if re.search('^y', answer, re.I):
            os.kill(proc, 9)
            time.sleep(1)
            try:
                # expect exception if process
                os.kill(proc, 0) # no longer exists
                sys.stderr.write("unsuccessful; sorry\n"); sys.exit(1)
            except: pass        # do nothing
        sys.stdout.write('')    # inhibit prepended blank on next print
    line = PS.readline().rstrip()
```

Figure 13.7 Script in Python to “force quit” errant processes. Compare to Figures 13.5 and 13.6.

Problem Domains: Extension Languages

- Most applications accept some sort of *commands*
 - these commands are entered textually or triggered by user interface events such as mouse clicks, menu selections, and keystrokes
 - Commands in a graphical drawing program might save or load a drawing; select, insert, delete, or modify its parts; choose a line style, weight, or color; zoom or rotate the display; or modify user preferences.
- An *extension language* serves to increase the usefulness of an application by allowing the user to create new commands, generally using the existing commands as primitives.
- Extension languages are increasingly seen as an essential feature of sophisticated tools
 - Adobe's graphics suite (Illustrator, Photoshop, InDesign, etc.) can be extended (scripted) using JavaScript, Visual Basic (on Windows), or AppleScript

Problem Domains: Extension Languages

- To admit extension, a tool must
 - incorporate, or communicate with, an interpreter for a scripting language
 - provide hooks that allow scripts to call the tool's existing commands
 - allow the user to tie newly defined commands to user interface events
- With care, these mechanisms can be made independent of any particular scripting language
- One of the oldest existing extension mechanisms is that of the **emacs** text editor
 - An enormous number of extension packages have been created for emacs; many of them are installed by default in the standard distribution.
 - The extension language for emacs is a dialect of Lisp called Emacs Lisp.

Problem Domains: Extension Languages

```
(setq-default line-number-prefix "")
(setq-default line-number-suffix ") ")
(defun number-region (start end &optional initial)
  "Add line numbers to all lines in region.
With optional prefix argument, start numbering at num.
Line number is bracketed by strings line-number-prefix
and line-number-suffix (default \"\" and \") \")."
  (interactive "*r\np")      ; how to parse args when invoked from keyboard
  (let* ((i (or initial 1))
         (num-lines (+ -1 initial (count-lines start end)))
         (fmt (format "%%dd" (length (number-to-string num-lines))))
         ; yields "%1d", "%2d", etc. as appropriate
         (finish (set-marker (make-marker) end)))
    (save-excursion
      (goto-char start)
      (beginning-of-line)
      (while (< (point) finish)
        (insert line-number-prefix (format fmt i) line-number-suffix)
        (setq i (1+ i))
        (forward-line 1))
      (set-marker finish nil))))
```

Figure 13.9 Emacs Lisp function to number the lines in a selected region of text.

Scripting the World Wide Web

- CGI Scripts

- The original mechanism for server-side web scripting is the Common Gateway Interface (CGI)
- A CGI script is an executable program residing in a special directory known to the web server program
- When a client requests the URI corresponding to such a program, the server executes the program and sends its output back to the client
 - this output needs to be something that the browser will understand: typically HTML.
- CGI scripts may be written in any language available
 - Perl is particularly popular:
 - its string-handling and “glue” mechanisms are suited to generating HTML
 - it was already widely available during the early years of the web

Scripting the World Wide Web

```
#!/usr/bin/perl

print "Content-type: text/html\n\n";

$host = `hostname`; chop $host;
print "<HTML>\n<HEAD>\n<TITLE>Status of ", $host,
      "\n</TITLE>\n</HEAD>\n<BODY>\n";
print "<H1>", $host, "\n</H1>\n";
print "<PRE>\n", `uptime`, "\n", `who`;
print "\n</PRE>\n</BODY>\n</HTML>\n";
```

Figure 13.10 A simple CGI script in Perl. If this script is named `status.perl`, and is installed in the server's `cgi-bin` directory, then a user anywhere on the Internet can obtain summary statistics and a list of users currently logged into the server by typing `hostname/cgi-bin/status.perl` into a browser window.

Scripting the World Wide Web

- Embedded Server-Side Scripts
 - Though widely used, CGI scripts have several disadvantages:
 - The web server must launch each script as a separate program, with potentially significant overhead
 - Scripts must generally be installed in a trusted directory by trusted system administrators
 - they cannot reside in arbitrary locations as ordinary pages do
 - The name of the script appears in the URI, typically prefixed with the name of the trusted directory, so static and dynamic pages look different to end users
 - Each script must generate not only dynamic content, but also the HTML tags that are needed to format and display it

Scripting the World Wide Web

```
<HTML>
<HEAD>
<TITLE>Status of <?php echo $host = chop('hostname') ?></TITLE>
</HEAD>
<BODY>
<H1><?php echo $host ?></H1>
<PRE>
<?php echo 'uptime', "\n", 'who' ?>
</PRE>
</BODY>
</HTML>
```

Figure 13.13 A simple PHP script embedded in a web page. When served by a PHP-enabled host, this page performs the equivalent of the CGI script of Figure 13.10.

Scripting the World Wide Web

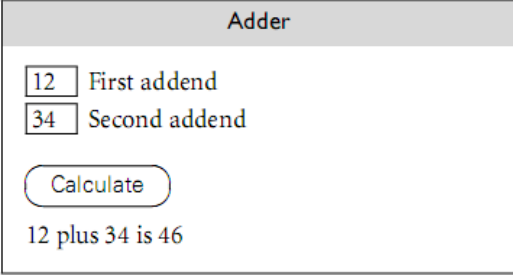
- Client-Side Scripts
 - embedded server-side scripts are generally faster than CGI script, at least when startup cost predominates
 - communication across the Internet is still too slow for interactive pages
 - Because they run on the web designer's site, CGI scripts and, to a lesser extent, embeddable server-side scripts can be written in many different languages
 - All the client ever sees is standard HTML.
 - Client-side scripts, by contrast, require an interpreter on the client's machine
 - there is a powerful incentive for convergence in client-side scripting languages: most designers want their pages to be viewable by as wide an audience as possible
- While Visual Basic is widely used within specific organizations, where all the clients of interest are known to run Internet Explorer, pages intended for the general public almost always use JavaScript for interactive features.

Scripting the World Wide Web

- Client-Side Scripts
 - While Visual Basic is widely used within specific organizations, where all the clients of interest are known to run Internet Explorer, pages intended for the general public almost always use JavaScript for interactive features
- Java Applets
 - An applet is a program designed to run inside some other program
 - The term is most often used for Java programs that display their output in (a portion of) a web page
 - To support the execution of applets, most modern browsers contain a Java virtual machine

Scripting the World Wide Web

```
<HTML>
<HEAD>
<TITLE>Adder</TITLE>
<SCRIPT type="text/javascript">
function doAdd() {
    argA = parseInt(document.adder.argA.value)
    argB = parseInt(document.adder.argB.value)
    x = document.getElementById('sum')
    while (x.hasChildNodes())
        x.removeChild(x.lastChild) // delete old content
    t = document.createTextNode(argA + " plus "
        + argB + " is " + (argA + argB))
    x.appendChild(t)
}
</SCRIPT>
</HEAD>
<BODY>
<FORM name="adder" onsubmit="return false">
<P><INPUT name="argA" size=3> First addend<BR>
    <INPUT name="argB" size=3> Second addend
<P><INPUT type="button" onclick="doAdd()" value="Calculate">
</FORM>
<P><SPAN id="sum"></SPAN>
</BODY>
</HTML>
```



The screenshot shows a web browser window with the title "Adder". Inside the browser, there is a form with two input fields. The first input field contains the number "12" and is labeled "First addend". The second input field contains the number "34" and is labeled "Second addend". Below these input fields is a button labeled "Calculate". Below the button, the text "12 plus 34 is 46" is displayed, representing the result of the calculation.

Figure 13.16 An interactive JavaScript web page. Source appears at left. The rendered version on the right shows the appearance of the page after the user has entered two values and hit the Calculate button, causing the output message to appear. By entering new values and clicking again, the user can calculate as many sums as desired. Each new calculation will replace the output message.

Innovative Features

- Earlier we listed several common characteristics of scripting languages:
 - both batch and interactive use
 - economy of expression
 - lack of declarations; simple scoping rules
 - flexible dynamic typing
 - easy access to other programs
 - sophisticated pattern matching and string manipulation
 - high level data types

Innovative Features

- Most scripting languages (Scheme is the obvious exception) 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

Innovative Features

- 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, JavaScript, in providing first class anonymous local subroutines
 - Nested blocks are statically scoped in Perl
 - In Ruby they are part of the named scope in which they appear
 - Scheme, Perl, Python provide for variables captured in closures
 - PHP and the major glue languages (Perl, Tcl, Python, Ruby) all have sophisticated namespace
 - mechanisms for information hiding and the selective import of names from separate modules

Innovative Features

- String and Pattern Manipulation
 - Regular expressions are present in many scripting languages and related tools employ extended versions of the notation
 - extended regular expressions in sed, awk, Perl, Tcl, Python, and Ruby
 - grep, the stand-alone Unix is a pattern-matching tool
 - 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 Features

- Data Types
 - As we have seen, scripting languages don't generally require (or even permit) the declaration of types for variables
 - Most perform extensive run-time checks to make sure that values are never used in inappropriate ways
 - Some languages (e.g., Scheme, Python, and Ruby) are relatively strict about this checking
 - When the programmer who wants to convert from one type to another must say so explicitly
 - Perl (and likewise Rexx and Tcl) takes the position that programmers should check for the errors they care about
 - in the absence of such checks the program should do something reasonable

Innovative Features

- **Numeric types:** “numeric values are simply numbers”
 - In **JavaScripts** all numbers are double precision floating point
 - In **Tcl** are strings
 - **PHP** has double precision float and integers
 - To these **Perl** and **Ruby** add ***bignums*** (arbitrary precision integers)
 - **Python** also has complex numbers
 - **Scheme** also has *rationals*
 - Representation transparency varies: best in **Perl**, ,minimal in **Ruby**
- Composite types: mainly **associative arrays** (based on hash tables)
 - **Perl** has fully dynamic arrays indexed by numbers, and hashes, indexed by strings. Records and objects are realized with hashes
 - **Python and Ruby** also have arrays and hashes, with slightly different syntax.
 - **Python** also has sets and tuples
 - **PHP** and **Tcl** eliminate distinction between arrays and hashes. Likewise **JavaScript** handles in a uniform way also objects.

Innovative Features

- Object Orientation
 - Perl 5 has features that allow one to program in an object-oriented style
 - PHP and JavaScript have cleaner, more conventional-looking object-oriented features
 - both allow the programmer to use a more traditional imperative style
 - Python and Ruby are explicitly and uniformly object-oriented
 - Perl uses a value model for variables; objects are always accessed via pointers
 - In PHP and JavaScript, a variable can hold either a value of a primitive type or a reference to an object of composite type.
 - In contrast to Perl, however, these languages provide no way to speak of the reference itself, only the object to which it refers

Innovative Features

- Object Orientation (2)
 - Python and Ruby use a uniform reference model
 - Classes are themselves objects in Python and Ruby, much as they are in Smalltalk
 - They are types in PHP, much as they are in C++, Java, or C#
 - Classes in Perl are simply an alternative way of looking at packages (namespaces)
 - JavaScript, remarkably, has objects but no classes
 - its inheritance is based on a concept known as *prototypes*
 - While Perl's mechanisms suffice to create object-oriented programs, dynamic lookup makes both PHP and JavaScript more explicitly object oriented