

# Lenguajes de scripting

basado en filminas de Michael L. Scott  
Paradigmas de la Programación  
FaMAF-UNC  
2021

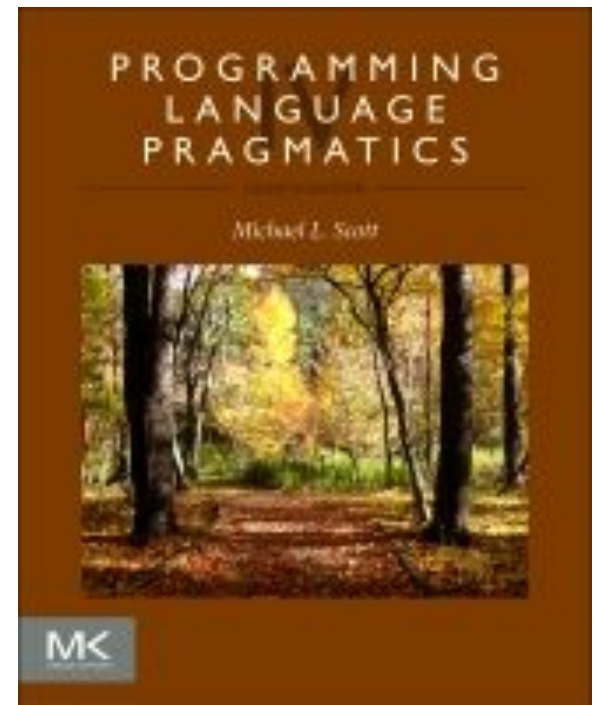
basado en...

Programming Language Pragmatics

Michael L. Scott

<http://booksite.elsevier.com/9780124104099/>

capítulo 14



# Qué es un lenguaje de scripting

dos grandes antepasados:

- intérpretes de comandos o “shells” de la terminal
  - intérprete de comandos de MS-DOS, shell de Unix
- herramientas de procesamiento de texto y generación de reportes
  - IBM: RPG, Unix: sed y awk.

que evolucionaron a...

- Rexx, el “Restructured Extended Executor” de IBM (1979)
- Perl (1987)
- Tcl (1988)
- Python (1991)
- Javascript (1995)
- Ruby (1995)
- VBScript (para Windows) y AppleScript (para Mac)

# diferentes tipos de lenguajes de scripting

- lenguajes específicos de dominio o lenguajes de extensión: para escribir pequeños programas que automatizan la ejecución de tareas en un entorno específico (un navegador, un intérprete de comandos, etc.), combinando llamadas a APIs o tareas elementales
- lenguajes dinámicos de muy alto nivel de propósito general

# características comunes

- hay que escribir muy poco

Java vs Python (o Ruby o Perl):

```
class Hello {  
    public static void main(String[]  
    args) {  
        System.out.println("Hello,  
        world!");  
    }  
}
```

```
print "Hello, world!\n"
```

# características comunes

- sin declaraciones, reglas de alcance simples
  - en Perl, por defecto todo es global, pero se puede convertir en local.
  - en PHP, por defecto todo es local, y las variables globales se tienen que importar explícitamente.
  - en Python, todo es local al bloque en que se hace una asignación.

# características comunes

- tipado dinámico flexible
  - en PHP, Python y Ruby, el tipo de una variable sólo se comprueba justo antes del uso
  - Perl, Rexx o Tcl, son todavía más dinámicos, con casteos:

```
$a = "4"
```

```
print $a . 3 . "\n"
```

```
print $a + 3 . "\n"
```

Outputs the following:

```
43
```

```
7
```

# características comunes

- fácil acceso a otros programas
  - todos los lenguajes tienen acceso al sistema operativo, pero los lenguajes de scripting normalmente proveen un soporte mucho más integrado y con más capacidades: pueden manipular directorios y archivos, módulos de I/O, sockets, acceso a bases de datos, soporte para passwords y autenticación y comunicaciones en redes.



# características comunes

- *pattern matching* y manejo de strings muy sofisticado y eficiente
  - Perl, sed/awk
  - basados en expresiones regulares extendidas

# características comunes

- tipos de datos de alto nivel
  - conjuntos, diccionarios, listas y tuplas por lo menos
  - C++ y Java requieren importarlos
  - optimizaciones como indexar los arrays usando tablas de hash
  - Garbage collection automático

# Lenguajes de dominio: Scripts

## lenguajes de intérprete de comandos (Shell)

- pensados para el uso interactivo
- mecanismos para manejar nombres de archivos, argumentos y comandos, y para “pegar” otros programas (los lenguajes genéricos también tienen estas utilidades)
  - Filename and Variable Expansion
  - Tests, Queries, and Conditions
  - Pipes and Redirection
  - Quoting and Expansion
  - Functions
  - The #! Convention

# Scripts: expansión de nombres de archivos y variables

- expansión de wilcards (o “globbing” por el comando original de unix *glob*) para obtener archivos que se corresponden con patrones.
- ejemplos:
  - `ls *.pdf`
  - `ls fig?.pdf`
  - `ls fig[0-9].pdf`
  - `ls fig3.{eps,pdf}`
- más complejos:
  - `for fig in *eps; do ps2pdf $fig; done`
  - `for fig in *.eps`  
do  
    `ps2pdf $fig`  
done

# Scripts: Tests y Queries

- Can modify the previous to only call ps2pdf on missing pdf files.
- Example: (-nt checks if left file is newer than right, and % removes the trailing .eps from variable \$fig)

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

# Scripts: Pipes and redirection

- Perhaps most significant feature of Unix shell was the ability to chain commands together, “piping” the output of one as the input of another
- Example (run in your homework directory):

```
echambe5@turing:~/.../homework$ ls *.pdf | grep hw
```

```
hw10.pdf
```

```
hw2.pdf
```

```
hw4.pdf
```

```
hw5.pdf
```

```
hw7.pdf
```

```
hw8.pdf
```

# Scripts: Pipes and redirection

- These can get even more complex:

```
for fig in *; do echo ${fig%.*}; done | sort -u | wc -l
```

- Explanation:
  - The for loop prints the names of all files with extensions removed
  - The sort -u removes duplicates
  - The wc -l counts the number of lines
- Final output: the number of files in our current directory with fig in the title, not distinguishing between files with different extensions but the same line (like file1.eps, file1.pdf, and file1.jpeg).

# Scripts: Pipes and redirection

- You can also redirect output to a file with `>` (output to file) or `>>` (append to a file).
- Example: Put a list of figures in a file:

```
for fig in *; do echo "${fig%.*}"; done  
                | sort -u > all_figs
```

-



# Scripts: Functions

- Can define your own functions, as well.
- Example:

```
function ll () {  
    ls -l "$@"  
}
```

- This allows you to type ll instead of ls -l at the prompt.
- In this, \$1 would be first parameter, \$2 the second, etc, so @\$ represents the entire parameter list.

# Scripts: The !# syntax

- To run a script in a file:  
`. my_script`
- This reads the input line by line - but it's not an executable.
- Most version of UNIX can make it a script:
  - Mark it as executable: i.e. `chmod +x my_script`
  - Begin the script with a control sequence telling it how to run it: `#!/bin/bash`
- This syntax is not just for bash - used also for Perl, Python, etc.

# procesamiento de textos

- algunas tareas comunes en edición de textos son difíciles de implementar en shell:
  - Inserción
  - Borrado
  - Búsqueda y reemplazo
  - Correspondencia de paréntesis

# procesamiento de textos

- ejemplo: eliminar los títulos de una página html en sed:
  - encontrar las etiquetas <h1> con pattern matching
  - borrar la línea con <h1>
  - imprimir las líneas que no tienen la etiqueta </h1>
  - borrar las líneas con </h1> y la anterior

# procesamiento de textos con sed

```
# 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.

# procesamiento de textos con sed

- herencia del editor de textos:
  - comandos de un solo caracter
  - no se usan variables más allá de la línea actual
- sed es muy limitado, se suele usar para programas muy simples, de una línea
- ejemplo: leer de input estándar y eliminar líneas en blanco

```
sed -e' /^[[:space:]]*$d'
```

# procesamiento de textos con Awk

- Awk se diseñó en 1977 (Aho, Weinberger y Kernighan) para superar las limitaciones de sed.
- es el paso entre sed y los lenguajes de scripting, sigue leyendo una línea por vez pero tiene mejor sintaxis y funcionalidades
- los programas son patrones con acciones asociadas
- la línea actual es siempre `$0`, tiene funciones como `getline` y `substr(s, a, b)`.
- también tiene loops y otras construcciones, y expresiones regulares

# Awk

```
/<[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.



# Awk: campos y arreglos asociativos

- tiene campos y arreglos asociativos
- por defecto, awk parsea las líneas de entrada en palabras (llamadas campos) separadas por espacios en blanco (aunque eso es parametrizable)
- estos campos se pueden acceder mediante las pseudovariables \$1, \$2, etc.
- ejemplo: `awk '{print $2}'`
  - imprime la segunda palabra de cada línea del input estándar
- los arreglos asociativos son como los diccionarios de Python

# Awk Example:

```
BEGIN { #noise words
    nw["a"] = 1; nw["an"] = 1; nw["and"] = 1; nw["but"] = 1;
    nw["by"] = 1; nw["for"] = 1; nw["from"] = 1; nw["in"] = 1;
    nw["into"] = 1; nw["of"] = 1; nw["or"] = 1; nw["the"] = 1;
    nw["to"] = 1;
}
{
    for (i=1; i<= NF; i++) {
        if (!nw[$i] || i==0 || $(i-1)~/[:-$]/) {
            #capitalize the word
            $i = toupper(substr($i, 1, 1))
        }
        printf $i " ";
    }
    printf "\n";
}
```

# procesamiento de texto: Perl

- intento de combina sed, awk y sh
- originalmente, una herramienta para unix y procesamiento de texto (*“practical extraction and report language”*)
- actualmente, de propósito genérico
- bastante rápido: se puede compilar, modularizar, librerías dinámicas, orientación a objetos...

# ejemplo de perl

```
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
}
```

**Figure 13.4** Script in Perl to extract headers from an HTML file. For simplicity we have again adopted the strategy of buffering entire headers, rather than printing them incrementally.

```

$#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`.

# lenguajes matemáticos

- en los 1960s se diseña APL para elaborar algoritmos matemáticos concisos y elegantes
- sucesores modernos: Matlab, and Mathematica.
- soportan métodos numéricos, matemática simbólica, modelado y aritmética de reales

# lenguajes matemáticos

- S (Laboratorios Bell, 1970s) y R (su equivalente open-source) evolucionaron para la comunidad de estadística.
- Características:
  - arreglos y listas multidimensionales
  - cortes de arreglos
  - pasaje de parámetros call-by-need
  - funciones de primera clase

# lenguajes “pegamento”

- Rexx es el primero
- proveen grandes librerías de control de sistemas operativos: File IO, process management, security, network/socket access, timing, synchronization.
- también proveen tipos de alto nivel, como hashes, tuples, strings, lists, etc.,
- muchos ofrecen hilos, alto orden, iteradores y estructuras más complejas



# Lenguajes pegamento: Tcl

- originalmente se diseña como un lenguaje de extensión para ser embebido en cualquier herramienta, proveyendo una sintaxis de comandos uniformes y reduciendo la complejidad del desarrollo y mantenimiento
- pronto adquiere usos de pegamento

# ejemplo de 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.

# lenguajes pegamento: Python y Ruby

- Perl y Tcl se desarrollan a fines de los 1980s
  - Perl originalmente como pegamento y procesamiento de texto
  - Tcl originalmente un lenguaje de extensión que empezó a usarse para pegamento
- Python y Ruby son posteriores
- sus diseñadores querían “hacerlo bien”, no es que no hubiera otras herramientas disponibles

# Python

- Primer lenguaje de scripting con orientación a objetos
- su librería estándar es tan rica como la de Perl, pero particionada en namespaces como C++

# ejemplo de 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.

# Ruby

- Desarrollado en 1990 por Matsumoto: “I wanted a language more powerful than Perl, and more object-oriented than Python.”
- Todo es un objeto (como en Smalltalk).
- Uso fuerte de bloques e iteradores
- las clases pueden heredar código de módulos, no sólo de otras clases

# 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
  - AOLserver, an open-source web server from America On-Line, can be scripted using Tcl. Disney and Industrial Light and Magic use Python to extend their internal (proprietary) tools

# 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, used to write this book
    - 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.
    - An example script appears in Figure 13.9
      - It assumes that the user has used the standard *marking* mechanism to select a region of text



# Problem Domains

```
(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.