Scripting Languages

COMS W4115

Prof. Stephen A. Edwards
Fall 2005
Columbia University
Department of Computer Science

Scripting Languages

Higher-level languages.

More compact for small programs.

Often not suitable for large systems.

The more popular ones:

- Awk
- Perl
- Python
- Tcl
- Bourne Shell

Awk

Named for its three developers:

Alfred Aho

Peter Weinberger

Brian Kernighan

Good for data file manipulation. I use it for computing grades and other simple tasks.

Simple Awk Program

Input file. Each line a record. Space-separated fields: employee, pay rate, hours worked

```
Beth  10.0   0
Dan    9.75  0
Kathy 14.0  10
Mark  10.0  20
Susie  8.25 18
```

Run on the awk program

```
$3 > 0 { print $1, $2 * $3 }
```

produces

```
Kathy 140
Mark 200
Susie 148.5
```

Simple Awk Program

```
Beth  10.0   0
Kathy 14.0  10
```

3rd field

```
$3  > 0  { print $1, $2 * $3 }
```

pattern — action

```
Kathy 140
```

Awk Program Structure

pattern { action }
pattern { action }
⋮

awk scans an input file one line at a time and, in order, runs each action whose pattern matches.

Patterns:

BEGIN, **END**	True before and after the file.
expression	Condition
/ regular expression /	String pattern match
pattern **&&** *pattern*	
pattern **||** *pattern*	Boolean operators
! *pattern*	

Awk One-Liners

Print every line

```
{ print }
```

Print the first and third fields of each line

```
{ print $1, $3 }
```

Print every line with three fields

```
NF == 3 { print }
```

Print a line number before every line

```
{ print NR, $0 }
```

Statistics in Awk

```
#!/bin/awk -f
BEGIN { n = 0; s = 0; ss = 0;}
NF == 1 {  n++; s += $1; ss += $1 * $1; }
END {
 print n " data points"
 m =  (s+0.0) / n; print m " average"
 sd = sqrt( (ss - n * m * m) / ( n - 1.0))
 print sd " standard deviation"
}
```

Run on

```
1
5
10
3
7
11
```

gives

```
6 data points
6.16667 average
3.92003 standard deviation
```

Associative Arrays: Word Counter

```
{ gsub(/[.,:;!(){}]/, "") # remove punctuation
  for ( i = 1 ; i <= NF ; i++ )
    count[$i]++
}
END { for (w in count)
        print count[w], w | "sort -rn"
    }
```

Run on the Tiger reference manual produces

```
103 the
58 of
51 is
49 and
49 a
35 expression
32 The
29 =
```

Perl

Larry Wall's

Practical Extraction and Report Language

or

Pathologically Eclectic Rubbish Lister

Larger, more flexible language than Awk. Good for text processing and other tasks. Strange semantics. Henious syntax.

Excellent regular-expression support. More complicated data structures possible (even classes).

Wordcount in Perl

```
#!/usr/bin/perl
while(<>) {
    chop;
    s/[.,:;!(){}]//g;
    @words = split;
    foreach (@words) {
        $count{$_}++;
    }
}
open(SORTER, "| sort -nr");
foreach (keys %count) {
    print SORTER
       $count{$_}, " ", $_,"\n";
}
```

Understandable wordcount in Perl

```
#!/usr/bin/perl
while($line = <>) {
    chop($line);
    $line =~ s/[.,:;!(){}]//g;
    @words = split(/\s+/, $line);
    foreach $word (@words) {
        $count{$word}++;
    }
}
open(SORTER, "| sort -nr");
foreach $word (keys %count) {
    print SORTER
       $count{$word}, " ", $word,"\n";
}
```

"There's more than one way to do it"

Perhaps too many. Equivalent ways to print STDIN:

```
while (<STDIN>) { print; }
print while <STDIN>
print while <>
while (defined(($_ = <STDIN>)) { print $_; }
for (;<STDIN>;) { print; }
print $_ while defined($_ = <STDIN>);
```

Many Perl statements come in prefix and postfix form

while (...) ...
... while ...
if (...) ...
... if ...
... unless ...

So Why Perl?

Perhaps the most popular scripting language.

Despite its flaws, it's very powerful.

Almost has a good type system.

Very few things *can't* be done in Perl.

Fast, flexible interpreter.

Ability to make virtually every Unix system call. Binary data manipulation.

Ported everywhere.

Very, very extensive collection of libraries. Database access. CGI/HTML for the web. Math. IPC. Time.

Python

Perl designed by a sane man.

Very clean syntax and semantics.

Large collection of libraries (but not as big as Perl's).

Regular expression support (but not as integrated as Perl's.)

Wordcount in Python

```
#!/usr/bin/env python

import fileinput, re, string, os

count = {}
for line in fileinput.input():
    line = re.sub(r'[.,:;!(){}]',"",line)
    for word in string.split(line):
        if not count.has_key(word):
            count[word] = 1
        else:
            count[word] = count[word] + 1

f = os.popen("sort -nr",'w')
for word in count.keys():
    f.write('%d %s\n' % (count[word], word) )
```

Python Classes

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart

    def add(self, a):
        self.r = self.r + a.r
        self.i = self.i + a.i

    def p(self):
        print "%g + %gi" % (self.r,self.i)

x = Complex(1,2)
y = Complex(2,3)
x.p()
x.add(y)
x.p()
```

Prints

```
1 + 2i
3 + 5i
```

Python's Merits

Good support for programming-in-the-large:

- Packages with separate namespaces; Exceptions; Classes
- Persistent datastructures (pickling)

High-level: lists, strings, associative arrays, iterators

Good collection of libraries:

- Operating-system access (files, directories, etc.); String manipulation; Curses; Databases; Networking (CGI, HTTP, URL, mail/Mime, HTML); Tk; Cryptography; System-specific (Windows, Mac, SGI, POSIX)

Python vs. Perl

Python can be the more verbose language, but Perl can be cryptic.

Regular expression support more integrated with language in Perl.

Perl better-known.

Probably comparable execution speeds.

More “tricks” possible in Perl; Python more disciplined.

Python has the much cleaner syntax and semantics; I know which language’s programs I’d rather maintain.

Tcl

John Ousterhout’s Tool Command Language was originally intended to be grafted on to an application to make it controllable.

Since become a general-purpose scripting language. Its syntax is quite simple, although rather atypical for a programming language.

Tk, a Tcl package, provide graphical user interface widgets. Tcl/Tk may be the easiest way to write a GUI.

Tk has been connected to Perl and Python as well.

Tcl Syntax

Shell-like command syntax:

command argument argument . . .

All data is strings (incl. numbers and lists)

Macro-like variable substitution:

```
set foo "123 abc"
bar 1 $foo 3
```

Command substitution:

```
set foo 1
set bar 2
puts [eval $foo + $bar]; # Print 3
```

Wordcount in Tcl

```
#!/usr/bin/env tclsh

while {[gets stdin line] >= 0} {
  regsub -all {[.,:;!(){}]} $line "" line
  foreach word $line {
     if {![info exists count($word)]} {
        set count($word) 1
     } else {
        incr count($word)
     }
  }
}

set f [open "| sort -rn" w]
foreach word [array names count] {
  puts $f "$count($word) $word"
}
```

Nifty Tcl Features

Associative arrays

```
set count(Stephen) 1
```

Lists

```
lappend foo 1
lappend foo 2
foreach i $foo { puts $i } ; # print 1 then 2
```

Procedures

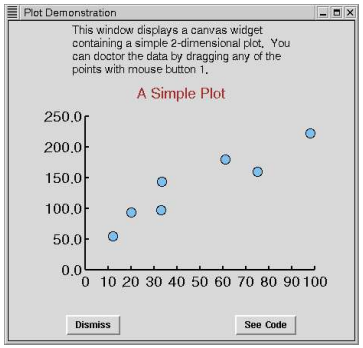
```
proc sum3 {a b c} {
  return [expr $a + $b + $c]
}
```

Tk

“Hello World” in Tk.

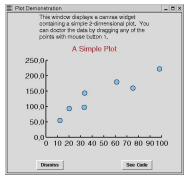
```
button .b -text "Hello World" -command "exit"
pack .b
```

An Editable Graph



An Editable Graph

```
# Set up the main window
set w .plot
catch destroy $w
toplevel $w
wm title $w "Plot Demonstration"
wm iconname $w "Plot"
positionWindow $w
set c $w.c
```



```
# Text description at top
label $w.msg -font $font -wraplength 4i -justify left \
  -text "This window displays a canvas widget containing
  a simple 2-dimensional plot.  You can doctor the data
  by dragging any of the points with mouse button 1."
pack $w.msg -side top
```

An Editable Graph

```
# Set up bottom control buttons
frame $w.buttons
pack $w.buttons -side bottom -fill x -pady 2m
button $w.buttons.dismiss -text Dismiss -command "destroy $w"
button $w.buttons.code -text "See Code" -command "showCode $w"
pack $w.buttons.dismiss $w.buttons.code -side left -expand 1

# Set up graph itself
canvas $c -relief raised -width 450 -height 300
pack $w.c -side top -fill x

# Draw axes
set plotFont Helvetica 18
$c create line 100 250 400 250 -width 2
$c create line 100 250 100 50 -width 2
$c create text 225 20 -text "A Simple Plot" -font $plotFont \
   -fill brown
```

An Editable Graph

```
# Draw axis labels
for {set i 0} {$i <= 10} {incr i} {
    set x [expr {100 + ($i*30)}]
    $c create line $x 250 $x 245 -width 2
    $c create text $x 254 -text [expr 10*$i] \
      -anchor n -font $plotFont
}
for {set i 0} {$i <= 5} {incr i} {
    set y [expr {250 - ($i*40)}]
    $c create line 100 $y 105 $y -width 2
    $c create text 96 $y -text [expr $i*50].0 \
      -anchor e -font $plotFont
}
# Draw points
foreach point {{12 56} {20 94} {33 98} {32 120} {61 180}
               {75 160} {98 223}} {
    set x [expr {100 + (3*[lindex $point 0])}]
    set y [expr {250 - (4*[lindex $point 1])/5}]
    set item [$c create oval [expr $x-6] [expr $y-6] \
    [expr $x+6] [expr $y+6] -width 1 -outline black \
    -fill SkyBlue2]
    $c addtag point withtag $item
}
```

An Editable Graph

```
# Bind actions to events
$c bind point <Any-Enter> "$c itemconfig current -fill red"
$c bind point <Any-Leave> "$c itemconfig current -fill SkyBlue2"
$c bind point <1> "plotDown $c %x %y"
$c bind point <ButtonRelease-1> "$c dtag selected"
bind $c <B1-Motion> "plotMove $c %x %y"

set plot(lastX) 0
set plot(lastY) 0
proc plotDown {w x y} {  # Called when point clicked
    global plot
    $w dtag selected
    $w addtag selected withtag current
    $w raise current
    set plot(lastX) $x
    set plot(lastY) $y
}
proc plotMove {w x y} {  # Called when point dragged
    global plot
    $w move selected [expr $x-$plot(lastX)] \
      [expr $y-$plot(lastY)]
    set plot(lastX) $x
    set plot(lastY) $y
}
```

Bourne Shell

Default shell on most Unix systems (sh or bash).

Good for writing “shell scripts:” parsing command-line arguments, invoking and controlling other commands, etc.

Example: The `cc` command.

Most C compilers built from four pieces:

- Preprocessor (cpp)
- Actual compiler (cc1)
- Assembler (as)
- Linker (ld)

cc in sh

```
#!/bin/sh

# Set up command names
root=/usr/lib
cpp=$root/cpp
cc1=$root/cc1
as=/usr/bin/as
ld=/usr/bin/ld

# Complaint function
usage() {
    echo "usage: $0 [options] files ..." 1>&2
    exit 1
}

# Default output filename
outfile="a.out"
```

cc in sh

```
# Parse command-line options
while [ ! -z "$1" ];
    do case x"$1" in
        x-v) echo "Stephen's cc 1.0"; exit 0 ;;
        x-o) shift; outfile=$1 ;;
        x-c) stopafterassemble=1 ;;
        x-S) stopaftercompile=1 ;;
        x-E) stopafterpreprocess=1 ;;
        x-*) echo "Unknown option $1" 1>&2; usage ;;
        *) break ;;
    esac
    shift
done

# Initialize lists of files to process
cfiles=""
sfiles=""
ofiles="crt1.o"

if [ $# = 0 ]; then
    echo "$0: No input files" 1>&2; exit 1
fi
```

cc in sh

```
# Parse filenames
while [ ! -z "$1" ]; do
  case x"$1" in
    x*.c) cfiles="$cfiles $1" ;;
    x*.s) sfiles="$sfiles $1" ;;
    x*.o | x*.a) ofiles="$ofiles $1" ;;
    *) echo "Unrecognized file type $1" 1>&2; exit 1 ;;
  esac
  shift
done

# Run preprocessor standalone
if [ "$stopafterpreprocess" ]; then
  for file in $cfiles; do
    $cpp $file
  done
  exit 0
fi
```

cc in sh

```
# Preprocess and compile to assembly
for file in $cfiles; do
  asmfile=`echo $file | sed s/.c$/.s/`
  $cpp $file | $cc1 > $asmfile
  sfiles="$sfiles $asmfile"
done
if [ "$stopaftercompile" ]; then exit 0; fi

# Assemble object files
for file in $sfiles; do
  objfile=`echo $file | sed s/.s$/.o/`
  $as -o $objfile $file
  ofiles="$ofiles $objfile"
done
if [ "$stopafterassemble" ]; then exit 0; fi

# Link to build executable
$ld -o $outfile $ofiles
exit 0
```

Scripting Languages Compared

	awk	Perl	Python	Tcl	sh
Shell-like	N	N	N	Y	Y
Reg. Exp.	B	A	C	C	D
Types	C	B	A	B	D
Structure	C	B	A	B	C
Syntax	B	F	A	B	C
Semantics	A	C	A	B	B
Speed	B	A	A	B	C
Libraries	C	A	A	B	C
Power	B	A	A	B	C
Verbosity	B	A	C	C	B

What To Use When

awk: Best for simple text-processing (file of fields)

Perl: Best for legacy things, things requiring regexps

Python: Best all-around, especially for large programs

Tcl: Best for command languages, GUIs

sh: Best for portable “invoking” scripts