Introduction to Scripting Languages

What is a scripting language?
- glue
- extension
- text processing
- web (CGI, server-side, client-side, XSLT)

Common characteristics
- economy of expression
- usually interpreted; for both batch and interactive use
- often a single canonical implementation
- lack of definitions; simple scoping rules
- dynamic typing, sometimes lots of coercion
- high-level types: sets, bags, dictionaries, lists, tuples, objects
- easy access to other programs
- pattern matching and string manipulation

Perhaps more accurately termed **dynamic languages**, in reference to the use of dynamic typing.

Ancestors
- shells
  - JCL
  - sh/ksh/bash
  - csh/tcsh
  - DOS shell
- text processing
  - RPG
  - sed
  - awk

Modern categories
- general-purpose
  - Rexx (old but still used on IBM platforms) late '70s
Perl (long the most widely used) late '80s
Tcl (now on the downswing, except for Tk) late '80s
Python (has probably passed Perl) early '90s
Ruby (on the upswing) early '90s*
  * didn't really catch on in the West until good English documentation came out in 2001
AppleScript (Mac platform only)
PowerShell (and, once, Visual Basic — Windows platform only)

extension
most of the general-purpose ones
  Python at Disney and ILM
  Tcl in AOLServer
Lua
  esp. in gaming
Scheme
  Elk
  SIOD — leading extension language for GIMP
    (Tcl, Python, Perl also supported)
Guile
Emacs Lisp
proprietary
  Maya
  Cold Fusion
  AutoCAD
  Macromedia Director, Flash
  Adobe tools w/ JavaScript, AppleScript, or VBScript
  many, many others

math
  APL; S, R; Mathematica, Matlab, Maple

web
  CGI -- all the GP options
  PHP -- leading server-side option; also ASP
  JavaScript -- leading client side option (VB used w/in some orgs.)
  Dart -- leader among several languages designed to compile to
    JavaScript as a way to get around the not-on-all-browsers problem
  XSLT -- for processing XML
names & scopes
what is the scope of an undeclared variable?
Perl: global unless declared otherwise
PHP: local unless explicitly imported
Ruby: foo is local; $foo is global; @foo is instance; @@foo is class

Python and R: local if written; global otherwise
Fig. 14.16, p. 740:

```python
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  # new local i
        return middle(j)

print(outer())
print(i, j)
```

prompts
(2, 3, 3)
4 3

No way historically to write an intermediate-level (non-global, non-local) var in Python. In recent python, replace “global” with “nonlocal”. In R, use foo <<- val (rather than foo <- val). These options avoid creating a new local R; accesses closest existing.

Both static and dynamic scope in Perl:
Fig. 14.17, p. 741:

```perl
sub outer($) {  # must be called with scalar arg
    $sub_A = sub {
        print "sub_A $lex, $dyn\n";
    };
    my $lex = $_[0];  # static local initialized to first arg
```
local $dyn = $_[0];  
    # dynamic local initialized to first arg
$sub_B = sub {  
    print "sub_B $lex, $dyn\n";
    
    print "outer $lex, $dyn\n";
$sub_A->();
$sub_B->();

}$lex = 1; $dyn = 1;
print "main $lex, $dyn\n";
outer(2);
print "main $lex, $dyn\n";

prints
   main 1, 1
   outer 2, 2
   sub_A 1, 2
   sub_B 2, 2
   main 1, 1

---------------------------------------------------------
strings & pattern matching

grep and sed: "basic" REs

awk, egrep, C regex library: "extended" (Posix) REs
quantifiers (generalizations of Kleene closure)
character sets
^ and $, .
backslash

Perl, Python, Ruby, JavaScript, elisp, Java, C#: "advanced" REs
trailing modifiers:
   g  global (all matches)
   i  case insensitive
   s  allow dot to match an embedded newline
   m  allow $ and ^ to match before / after embedded newline
   x  ignore comments and white space in pattern

capture:
$_ = "-3.14e+5";  # default subject of match if =~ not used
if (/^([-+]?)(\d+|\d*\.\d*)(e([-+]?\d+))?$/) {
  # floating point number
  print "sign: ", $1, "\n";
  print "integer: ", $3, $4, "\n";  # only one nonempty
  print "fraction: ", $5, "\n";
  print "mantissa: ", $2, "\n";
  print "exponent: ", $7, "\n";
}

This prints
  sign:  
  integer:  3
  fraction: 14
  mantissa: 3.14
  exponent: +5

greedy (default *) and minimal (*?) matches
  +? matches at least one but no more than necessary
  ?? matches zero or one, with a preference for zero

special escape sequences
  lots of these.  E.g.,
    \n, \r, \t, ...
    \d  digit
    \s  white space
    \w  word character (letter, digit, underscore)
    ...

Implementation
  NFA v. DFA
    tradeoff: DFA requires compilation: good if repeated;
    may also be necessary if there is capture.
    NFA can be emulated immediately.

  compilation?
    qr operator forces (one-time) compilation:
```perl
for (@patterns) {
    my $pat = qr($_);
    for (@strings) {
        if (/\$pat/) {
            print;  # no recompilation required
            print "\n";
        }
        print "\n";
    }
}
```

data types
Perl goes crazy with coercion
```perl
$a = "4";
```
# string
print $a . 3 . "\n";  # concatenation => "34"
print $a + 3 . "\n";  # addition => 7

notion of context in Perl
numeric, string, scalar/array, ...

considerable variety in numeric types
always doubles in JavaScript; doubles by default in Lua
always strings in Tcl (!)
PHP: ints & doubles
Perl, Ruby: ints, doubles, and bignums
Scheme: ints, doubles, bignums, rationals

composites:
where static languages tend to emphasize arrays & structs,
scripting languages typically emphasize mappings
(aka hashes, dictionaries, associative arrays)

Perl, Python, Ruby:
arrays and hashes — both self-expanding (syntax varies)
Python: also tuples & sets
tuples are immutable (and thus faster than arrays)
sets support union, intersection, difference, xor
PHP & Tcl: arrays == hashes
array is just a hash w/ numeric keys
JavaScript: arrays == hashes == objects
multidimensional arrays via tuple keys
not very efficient
much better support in the 3Ms

objects
    hack in Perl 5; supposed to be real in Raku (Perl 6)
"Object-based" approach in JavaScript
    classes added in ECMAScript 6 (backward compatible), TypeScript
pure object orientation in Ruby, ala Smalltalk
executable class declarations
    mentioned under "elaboration" in Chap. 3 lecture
    can be used, e.g., to give the effect of conditional compilation

perl

Note that Perl, unlike Python & Ruby, has no real interactive mode. (You can get some of the functionality from the Perl debugger, but it executes each line in isolation, which is pretty unsatisfying.)

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

    if ($a < $b) { $s = "less"; }
    $s = "less" if ($a < $b);
    $s = "less" unless ($b >= $a);

heavy use of punctuation characters

    #  comment
    #!  convention     script language identifier
    $, @, %, NAKED     scalar, array, hash, filehandle
    <..>               readline of file handle
    =~                  pattern match
    $._                 default input line and loop index
    . and .=           concatenation
Dynamic typing, coercion

```perl
$a = "4";
print $a . 3 . "\n"; # prints 43
print $a + 3 . "\n"; # prints 7
```

subroutines

```perl
sub min {
    my $rtn = shift(@_); # first argument
    # my gives local lexical scope; @_ is list of arguments
    # local gives dynamic scope
    for my $val (@_) {
        $rtn = $val if ($val < $rtn)
    }
    return $rtn;
}
...
$smallest = min($a, $b, $c, $d, @more_vals); # args are flattened
```

context

some things behave differently in array and scalar "contexts".

```perl
@my_array = @_; # @_;
$num_args = @_; # @_
```

you can do this yourself:

```perl
sub abs {
    my @args = @_;  # @_
    for (@args) {  # @_
        $_ = -$_ if ($_ < 0); # $ is a reference;
    }
    return wantarray ? @args : $args[0]; # note: not @args[0]
}
...
print join (", ", abs(-10, 2, -3, 4, -5)), "\n";
print $n = abs(-10, 2, -3, 4, -5), "\n";
```
This prints

```
10, 2, 3, 4, 5
10
```

regular expressions -- discussed in previous lecture

hashes

```
%complements = ("ref" => "cyan",
               "green" => "magenta", "blue" => "yellow");

# NB: => is (almost) an alias for,
# (also forces its left operand to be interpreted as a string)
print $complements{"blue"};    # yellow
```

Examples from book

HTML heading extraction (Example 14.23, Fig. 14.4, p. 716)

```
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
}
```

note:

```
#!
while (<>)
next, redo
implicit matching against $_
update-assignment to $_
s/// -- could have been written $_ =~ s///
minimal matching via *?
character sets in REs: [hH], [123]
backslash escape of /
capture with ( )
trailing s on match allows "." to match embedded \n```
force quit (Example 14.24, Fig. 14.5, p. 719)

```perl
#!/usr/bin/perl

use strict; # use strict mode
use warnings; # use warnings mode

# ARGV[0] is first argument
$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
    }
}

note:
@ARGV, $#ARGV            # latter is last index (one less than length)
die open, file handles
ps command
    -w -w               # print with unlimited width (wide wide)
    -x                 # include processes w/out controlling terminals
    -o'pid,command'    # what to print
split trailing i on match ignores case
$$                   # my process id
ne (strings) vs != (numbers)
beginning of line marker: ^ (and eol marker: $)
built-ins for many common shell commands (kill, sleep)
```