AP-24: Scripting languages
Based on Chapter 13 of Programming Language Pragmatics
by Michael L. Scott, 3rd edition
Origin of Scripting Languages

• Modern scripting languages have two principal sets of ancestors.
  1. command interpreters or “shells” of traditional batch and “terminal” (command-line) computing
     • IBM’s JCL, MS-DOS command interpreter, Unix sh and csh
  2. 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,” ~1979
  – Perl, originally devised by Larry Wall in the late 1980s
  – Other general purpose scripting languages include Tcl (“tickle”), Python, Ruby, VBScript (for Windows) and AppleScript (for Mac)
  – PHP for server-side web scripting (and JSP, VBScript, JavaScript…)
  – And several others....
Scripting Language: Common Characteristics

– Both batch and interactive use
  • Compiled/interpreted line by line

– Economy of expression
  • Concise syntax, avoid top-level declarations

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

```python
print "Hello, world!\n" # Python
```

– Lack of declarations

– Simple default scoping rules, which can be overruled via explicit declarations
Dynamic typing, due to lack of declarations

Flexible typing: a variable is interpreted differently depending on the context (kind of coercion)

```perl
$a = "4";       # Perl
print $a . 3 . "\n";  # "." is concatenation
print $a + 3 . "\n";  # '+' is addition
```

will print

43
7

Easy access to system facilities

- Eg: **Perl** has more than 100 built-in commands for I/O, file/directory manipulation, process management, ...
- **Note, Perl means Perl 5. Perl 6 (now Raku) is different.**
Scripting Language: Common Characteristics

– Sophisticated pattern matching and string manipulation
  • From text processing and report generation roots
  • Based on extended regular expressions

– High level data types
  • Built-in support for associative arrays implemented as hash tables.
  • Storage is garbage collected

– Quicker development cycle than industrial-quality languages (like Java, C++, C#, …)
  • Able to include state-of-the-art features (E.g., Python includes several new constructs seen in Java and Haskell)
Problem Domains

• Some general purpose languages (eg. **Scheme** and **Visual Basic**) are widely used for scripting

• Conversely, some scripting languages (eg. **Perl**, **Python**, and **Ruby**) are intended for general use, with features supporting “programming in the large”
  – modules, separate compilation, reflection, program development environments

• But most scripting languages have principal use in **well defined problem domains**:
  1. Shell languages
  2. Text Processing and Report Generation
  3. Mathematics and Statistics
  4. “Glue” Languages and General-Purpose Scripting
  5. Extension Languages
  6. [*Scripting the World Wide Web – not discussed, see reference*]
The slides from here to page 20 were skipped during the lesson
Problem Domains: Shell Languages

• **Shell Languages** have features designed for interactive use
  – Multics ~1964, Unix ~1973, *sh*, *csh*, *tcsh*, *ksh*, *bash*, …

• Provide many 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

```bash
#!/bin/bash

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

for fig in *; do echo ${fig%.*}; done | sort -u > all_figs
```
Problem Domains: **Text Processing and Report Generation**

**sed**: Unix’s *stream editor*
- No variables, no state: just a powerful filter
- Processes one line of input at a time
- The first matching command is executed
- \`s/_/_/\` substitution command

![Programme](image)

**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** (from Aho, Weinberger & Kernighan)
- adds variables, state and richer control structures
- also *fields* and *associative arrays*

```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.
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: very popular and widely used scripting language
- Also fast enough for much general purpose use, and includes
  - separate compilation, modularization, and dynamic library mechanisms appropriate for large-scale projects

```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]\)/\1/g;  # perform minimal matching; capture parenthesized expression in $1
        print $1, "\n";
    }
    redo unless eof;  # continue without reading next line of input
}
```
Problem Domains:
Mathematics and statistics

- Maple, Mathematica and Matlab (Octave): commercial packages successor of APL (~1960)
  - Extensive support for numerical methods, symbolic mathematics, data visualization, mathematical modeling.
  - Provide scripting languages oriented towards scientific and engineering applications

- Languages for statistical computing: R (open source) and S
  - Support for multidim. Arrays and lists, array slice ops, call-by-need, first-class functions, unlimited extent
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: “a language more powerful than Perl, and more object-oriented than Python”
  - English documentation published in 2001
  - Smalltalk-like object orientation
Example: “Force quit” in Perl

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

$#ARGV == 0 || die "usage: $0 pattern
";
open(PS, "ps w-w-x-o\'pid,command\' ");  # 'process status' command
<PS>;
while ( <$PS> ) {  
    @words = split;
    if (/ARGV[0]/i && $words[0] ne $$) {  # parse line into space-separated words
        chomp;
        print;
        do {
            print "? ";
            $answer = <STDIN>;
        } until $answer =~ /[yn]/i;
        if ($answer =~ /[y]/i) {  # delete trailing newline
            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`.
“Force quit” in Python 2

```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 -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:
            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.
“Force quit” in Ruby

```ruby
ARGV.length() == 1 or begin
  $stderr.print("usage: 

$0 pattern\n"); exit(1)
end

pat = Regexp.new(ARGV[0])
IO.popen("ps -w -w -x -o'pid,command'") { |PS|
  PS.gets
  # discard header line
  PS.each { |line|
    proc = line.split[0].to_i
    if line =~ pat and proc != Process.pid then
      print line.chomp
      begin
        print "? 
        answer = $stdin.gets
      end until answer =~ /\^[yn]/i
      if answer =~ /\^y/i then
        Process.kill(9, proc)
        sleep(1)
        begin
          # expect exception (process gone)
          Process.kill(0, proc)
        rescue
          # handler -- do nothing
        end
      end
    end
  }
}
```

Figure 13.8 Script in Ruby to “force quit” errant processes. Compare to Figures 13.5, 13.6, and 13.7.
Problem Domains: Extension Languages

• Most applications accept some sort of *commands*
  – 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 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 "%%%d" (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.
The next slides were presented
Innovative Features of Scripting Languages

• 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 an 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 \texttt{sed, awk, Perl, Tcl, Python,} and Ruby
    • grep, the stand-alone Unix is a pattern-matching tool
  – Two main groups.
    • The first group includes \texttt{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 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 wants to convert from one type to another he 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](https://developers.mozilla.org) all numbers are double precision floating point
  – In [Tcl](http://www.tcl.tk) are strings
  – [PHP](https://www.php.net) has double precision float and integers
  – To these [Perl](https://www.perl.org) and [Ruby](https://www.ruby-lang.org) add [bignums](https://en.wikipedia.org/wiki/Bignum) (arbitrary precision integers)
  – [Python](https://www.python.org) also has complex numbers
  – [Scheme](https://www.scheme.org) also has [rationals](https://en.wikipedia.org/wiki/Rational_numbers)
  – Representation transparency varies: best in [Perl](https://www.perl.org), minimal in [Ruby](https://www.ruby-lang.org)

• Composite types: mainly [associative arrays](https://en.wikipedia.org/wiki/Hash_table) (based on hash tables)
  – [Perl](https://www.perl.org) has fully dynamic arrays indexed by numbers, and hashes, indexed by strings. Records and objects are realized with hashes
  – [Python](https://www.python.org) and [Ruby](https://www.ruby-lang.org) also have arrays and hashes, with slightly different syntax.
  – [Python](https://www.python.org) also has sets and tuples
  – [PHP](https://www.php.net) and [Tcl](http://www.tcl.tk) eliminate distinction between arrays and hashes. Likewise [JavaScript](https://developers.mozilla.org) 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
  – 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 are more explicitly object oriented
  – Classes are themselves objects in Python and Ruby, much as they are in Smalltalk
  – In Ruby, \( 2 \times 4 + 5 \) is syntactic sugar for \((2 \times (4)) + (5)\), which is in turn equivalent to
    \((2.\text{send}(\text{"*"}, 4)).\text{send}(\text{"+"}, 5)\).
Summary

• Scripting languages evolve quickly
• Able to incorporate latest features of programming language technology
• Quick learning curve
  – Widely used in teaching
• Huge libraries
• Very widely used, but pros and cons should be evaluated carefully...