

# Scripting Languages

- Bash – operating system management
- Awk – advanced text manipulations
- Tcl/Tk – power of substitution
- Python – efficient syntax
- LaTeX – high quality publishing

# Scripting

## Higher Level Programming for the 21st Century

John K. Ousterhout, IEEE Computer magazine, March 1998

<http://www.tcl.tk/doc/scripting.html>

- Scripting languages such as Python, Perl, Tcl, Ruby, etc. represent a very different style of programming than system programming languages such as C or Java™.
- Scripting languages are designed for "gluing" applications;
- They use typeless approaches to achieve a higher level of programming
- Scripting offers more rapid application development than system programming languages.
- Increases in computer speed and changes in the application mix are making scripting languages more and more important for applications of the future.

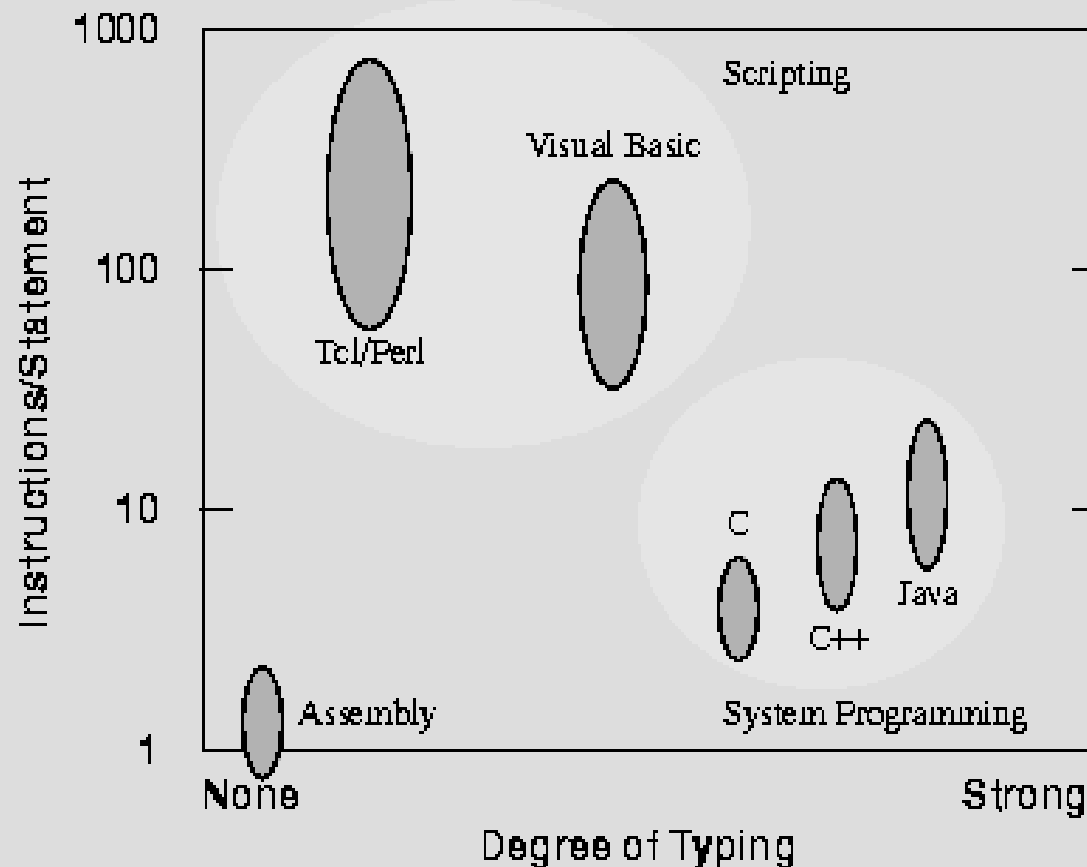
# Features

- Interpreted execution
- Only source code, no compilation & binaries
- Execution of uncompleted programs
- Weak data types
- Advanced data structures
- Collections of ready-to-use components
- Compact code
- Various programming paradigms

# Applications

- Rapid application development
- Ideas and prototypes
- Frequent modifications / adjustments
- Ease of maintenance
- Platform independence
- System integration tasks

# Comparison



**Figure 1.** A comparison of various programming languages based on their level (higher level languages execute more machine instructions for each language statement) and their degree of typing. System programming languages like C tend to be strongly typed and medium level (5-10 instructions/statement). Scripting languages like Tcl tend to be weakly typed and very high level (100-1000 instructions/statement).

Application (Contributor)	Comparison	Code Ratio	Effort Ratio	Comments
Database application (Ken Corey)	C++ version: 2 months Tel version: 1 day		60	C++ version implemented first; Tel version had more functionality.
Computer system test and installation (Andy Belsey)	C test application: 272 Klines, 120 months. C FIS application: 90 Klines, 60 months. Tel/Perl version: 7.7K lines, 8 months	47	22	C version implemented first. Tel/Perl version replaced both C applications.
Database library (Ken Corey)	C++ version: 2-3 months Tel version: 1 week		8-12	C++ version implemented first.
Security scanner (Jim Graham)	C version: 3000 lines Tel version: 300 lines	10		C version implemented first. Tel version had more functionality.
Display oil well production curves (Dan Schenck)	C version: 3 months Tel version: 2 weeks		6	Tel version implemented first.
Query dispatcher (Paul Healy)	C version: 1200 lines, 4-8 weeks Tel version: 500 lines, 1 week	2.5	4-8	C version implemented first, uncommented. Tel version had comments, more functionality.
Spreadsheet tool	C version: 1460 lines Tel version: 380 lines	4		Tel version implemented first.
Simulator and GUI (Randy Wang)	Java version: 3400 lines, 3-4 weeks. Tel version: 1600 lines, < 1 week.	2	3-4	Tel version had 10-20% more functionality, was implemented first.

**Table 1.** Each row of the table describes an application that was implemented twice, once with a system programming language such as C or Java and once with a scripting language such as Tel. The **Code Ratio** column gives the ratio of lines of code for the two implementations (>1 means the system programming language required more lines); the **Effort Ratio** column gives the ratio of development times. In most cases the two versions were implemented by different people. The information in the table was provided by various Tel developers in response to an article posted on the comp.lang.tel newsgroup; see [7] for details.

# Do not use scripting for

- Complex algorithms
- Precise numerical calculations
- Complex data structures
- Huge data sets
- Fastest execution
- Large software projects
- Well defined and matured tasks

# Shell Scripting

# Unix philosophy

- Many small tools performing one task perfectly
- Shell scripts as glue for system commands
- Text oriented processing
- Regular expressions
- Standard input and output
- Common format of input and output data

# Shell categories

Popular unix shells belong to categories:  
Bourne-like & C Shell-like

- Bourne shell compatible

- Bourne shell (sh) -- Written by Steve Bourne, at Bell Labs. First distributed with Version 7 Unix, circa 1978, and enhanced over the years.
- Almquist shell (ash) -- Written as a BSD-licensed replacement for the Bourne Shell; often used in resource-constrained environments.
- Bourne-Again shell (bash) -- Written as part of the GNU project to provide a superset of Bourne Shell functionality.
- Korn shell (ksh) -- Written by David Korn, while at Bell Labs.
- Z shell (zsh) -- considered as the most featured shell

- C shell compatible

- C shell (csh) - Written by Bill Joy, at the University of California, Berkeley. First distributed with BSD, circa 1979.
- TENEX C shell (tcsh)

# Bourne shells

- Compatibility with sh, as its syntax is traditionally used in system initialization
- Programming aids (all modern shells):
  - Command/filename completion
  - History manipulation;
  - Command line editor
  - Aliases/functions
- Linux systems typically choose Bash as the shell used by default
- On most modern Unix-like systems the current shell name is held in the **\$SHELL** environment variable

# C shells

- Csh (tcsh) has syntax similar to C
- It was considerably superior to sh
- **Csh Programming Considered Harmful**
  - <http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/>

*“The csh is seductive because the conditionals are more C-like, so the path of least resistance is chosen and a csh script is written. Sadly, this is a lost cause, and the programmer seldom even realizes it, even when they find that many simple things they wish to do range from cumbersome to impossible in the csh.”*

# Bash invocation

- Login shell (at login or with option `--login`):
  - `/etc/profile`
  - `~/.bash_profile`, `~/.bash_login`, `~/.profile`  
executes commands from the first one that exists and is readable,  
`--noprofile` option inhibits this behavior
  - `~/.bash_logout`
- Not a login shell:
  - `~/.bashrc`
- Bash may be started as non-interactive !
- Bash started as `sh` tries to mimic the historical `sh`

# Shell grammar

- simple command (command + arguments)
- pipelined command ( | )
- lists of commands
  - ; or newline - sequence
  - && - if previous command returns an exit status 0
  - | | - if previous command returns non-0 exit status

# Compound commands

- ( list of commands)
  - executed in subshell environment
  - return status is the exit status of list
- { list of commands }
  - executed in current shell environment
  - return status is the exit status of list
- (( expression ))
  - evaluated as arithmetic expression
  - return 0 for non-0 values of expression
- [[ expression ]]
  - evaluated as conditional expression
  - return 0 or 1 depending on condition result

# Control statements

- **for** name [ **in** word ] ; **do** list ; **done**
- **for** (( expr1 ; expr2 ; expr3 )) ; **do** list ; **done**
- **select** name [ **in** word ] ; **do** list ; **done**
- **case** word **in** [ [(**()** pattern [ | pattern ] ... ) list ;; ] ... **esac**
- **if** list; **then** list; [ **elif** list; **then** list; ] ... [ **else** list; ] **fi**
- **while** list; **do** list; **done**
- **until** list; **do** list; **done**

*brackets [...] indicate optional syntax*

# Shell functions

A shell function is an object that:

- is called like a simple command
- executes a compound command
- has new set of positional parameters

[ **function** ] name () compound-command [redirection]

*brackets [...] indicate optional syntax*

# Comments & quoting

- Word beginning with # causes all remaining characters on that line to be ignored
- Quoting is used to remove the special meaning of certain characters or words to the shell:
  - escape character – non-quoted backslash (\)
  - single quotes – literal value of each character within
  - double quotes – allows expansions within

# Variables

name = value

- Variable has name and value
- Variable is set if it has been assigned a value
- Null string is a valid value
- Variable may be unset by using unset command
- Variables may have attributes (declare command) modifying evaluation of its value (e.g. integer)
- Variables are referenced using \$name

# Bash arrays

`name[subscript]=value`  
`declare -a name`

- Bash provides one-dimensional arrays
- Arrays are indexed using integers, zero-based
- There is no maximum limit on the size of arrays
- The subscript is treated as an arithmetic expression
- Array elements are referenced using `${name[subscript]}`
  - (the braces are required)

# Positional & special parameters

- \$1, \$2, \$3, \$4, \$5, \$6, \$7, \$8, \$9, \${10}, \${11}, ...
  - assigned from the shell's arguments when it is invoked
  - temporarily replaced when a shell function is executed
- \$\* is equivalent to "\$1 \$2 ..."
- @\$@ is equivalent to "\$1" "\$2" ...
- \$# - number of positional parameters (in decimal)
- \$\$ - process ID of the shell
- \$! - process ID of the most recently executed background (asynchronous) command
- \$0 - the name of shell script

# Managing positional parameters

- shift
  - rename positional parameters  
while [ \$# -gt 0 ] do ... shift ... done
- getopt optlist name
  - parse positional parameters according to the list

# Shell variables (quite a lot!)

- Variables set by the shell, e.g.:
  - PWD, RANDOM, SECONDS, UID ...
- Variables used by the shell, e.g.:
  - HOME, IFS, LANG, PATH, PS1..4, SHELL

# Environment

- `export name=value`
  - export to the environment of subsequently executed commands
- `readonly name=value`
  - the values of these names may not be changed by subsequent assignments
- `unset name`
  - remove the corresponding variable or function (except `readonly`)
- `source filename ( . filename )`
  - read and execute commands from `filename` in the current shell environment

# Expansion

- Expansion is transformation performed on the command line after it has been split into words
- There are seven kinds of expansion performed (in order of precedents):
  - brace expansion - { }
  - tilde expansion - ~
  - parameter/ variable expansion - \${name and options}
  - command substitution - \$( ), ` `
  - arithmetic expansion - \$(( ))
  - word splitting – separator IFS
  - pathname expansion - \*, ?, [ ]

# Redirection

- Before a command is executed, its input and output may be redirected
  - /dev/stdin - File descriptor 0
  - /dev/stdout - File descriptor 1
  - /dev/stderr - File descriptor 2
  - files, TCP and UDP ports
- Order is significant: `> xxx 2>&1`
  - Redirecting Input and Output: `< xxx, > xxx`
  - Appending Redirected Output: `>> xxx`
  - Redirecting stdout and stderr: `&>xxx, >xxx 2>&1`
  - Here-Document: `<< xxx`
  - Opening Files for Reading and Writing: `<>xxx`

# Aliases

- Aliases allow a string to be substituted for a word when it is used as the first word of a simple command
- Shell maintains a list of aliases that may be set and unset with the alias and unalias
- For almost every purpose, aliases are superseded by shell functions

# Other subjects ...

- Arithmetic evaluation: operators and order
- Conditional expressions and options
- Internal commands: shell functions, aliases, built-ins
- Command execution: PATH
- Environment management: export
- Exit status: 0 success, non-0 failure
- Signals: kill
- Job control: fg, bg, jobs
- Prompting: PS1, PS2
- History
- ...

**man bash**

# Script debugging

- Use echo liberally
- Conditional echos
  - `[[ -n $debug ]] && echo "debugging"`
- `bash -x filename`
  - Print all executed lines
- `set -x`
  - turn on printing of executed lines
- `set +x`
  - turn off printing of executed lines

# Search & Replace

- `grep` – basic set of reg. expr. (BRE)
  - line-by-line processing
  - `egrep` – extended set of reg. expr. (ERE)
  - `fgrep` – no regular expressions
- `sed`
  - line-by-line processing
  - multiple operations on single line

# BRE

- \ escape special meaning
- . any char
- \* repetition, zero or more
- ^ beginning
- \$ end
- [ ] single char from the list
- {m,n} repetition, from m to n, inclusive
- {m} repetition m-times
- {m,} repetition m-or-more-times
  - {m,n},{m},{m,} repetitions (ERE)
- \( \) record pattern (BRE only)
- \n use recorded pattern (BRE only)

# BRE/ERE Classes

- [:alnum:] letters & digits
- [:alpha:] letters
- [:cntrl:] control characters
- [:digit:] digits
- [:graph:] graphically non-empty characters
- [:lower:] lower case letters
- [:print:] printable characters
- [:punct:] punctuations
- [:space:] separations (space & tab)
- [:upper:] upper case letters
- [:xdigit:] hex digits

# ERE only

- + repetition, one or more
- ? repetition, zero or one
- | pattern alternative
- () grouping
- no recorded patterns

Non standard extension:

- \< \> beginning and end of a word

# BRE/ERE Programs

	grep	sed	vi	egrep	awk	lex
BRE	x	x	x			
ERE				x	x	x
<  >	x	x	x			

# Text fields

- cut
  - field extraction
  - cutting by character or fields columns
  - separator definitions
- join
  - file merging, line-by-line
  - keys

# Other text operations

- sort
- uniq
- fmt
- wc
- head
- tail
- tr
- dd
- file
- od
- strings
- ...

**info coreutils**

# UNIX tools for Windows

Collection of tools which provide Linux look and feel  
Works with x86 32 bit and 64 bit versions of Windows

- Cygwin
  - <http://www.cygwin.com> - free
- UWIN (AT&T)
  - <http://www.research.att.com/sw/tools/uwin/> - free
- MKS Toolkit
  - <http://www.mkssoftware.com> - commercial
- InteropSystems
  - <http://www.interix.com/> - commercial

# Script projects

- Data mining (e.g. with wget)
- Front-ends
- System administration
- File manipulation
- Text/data transformations
- Fun & games
- ...