

---

# **CS 35L- Software Construction Lab 3**

Fall 18

TA: Guangyu Zhou

---

# Announcement

---

- PTE
    - Waitlist
    - Lab switch
  - Switch of lab session:
    - Please proceed to contact the TA in other session for enrollment. Each session has it's own exam. You are welcomed to auditing.
  - BeagleBones
-

# Presentation

---

## Useful pointers

- Presentation

- Topic on recent research in computer science
  - **Technical** content is required
- Please think about topics from now on!
- ~10 minutes talk in class
- 1 or 2 people
- Participation in Q&A
- Sign-up sheet in week 3-4. (FCFS!)
- Brief Research report (due in the last week)

- News sources
  - [ACM TechNews](#), for example:
    - [2018-09-09](#)
    - [2018-09-21](#)
    - [2018-09-24](#)
  - [;login: The USENIX Magazine](#)
  - [Computing Research News](#)
  - [Linux Today](#)
- Index for research in computer science
  - [Google Scholar](#)
- Computing research and study organizations
  - [Association for Computing Machinery](#) and the [UCLA ACM Student Chapter](#)
  - [IEEE Computer Society](#) and the [UCLA IEEE student chapter](#)
  - [Linux Users Group at UCLA](#)
  - [USENIX](#)
  - [Computing Research Association](#)
  - [SCaLE](#)
- Academic study and research
  - [CRA for students](#)
  - Joel Spolsky, [Advice for computer science college students](#) (2005)
  - Phil Agre, [Advice for undergraduates considering graduate school](#) (2001)
  - Mor Harchol-Balter, [Applying to Ph.D. Programs in Computer Science](#) (2014)
  - [UC Berkeley Computer Science Division](#)
  - [Carnegie Mellon School of Computer Science](#)
  - [MIT Department of Electrical Engineering & Computer Science](#)
  - [Stanford Computer Science Department](#)
- Industrial research and development
  - [Bell Labs](#)
  - [Cisco Research Center](#)
  - [Facebook Research](#)
  - [Research at Google](#)
  - [HP Labs](#)
  - [IBM Computer Science Research](#)

---

# Shell Scripting and Regular Expression

Week 2

---

# Outline

---

- **Advanced Linux Commands**
  - **Regular Expression**
  - **The Shell Scripting**
-

# Locale

---

- Set of parameters that define a user's cultural preference
  - Language
  - Country
  - Other area-specific things

`locale` command:

prints information about the current locale environment to standard output

---

# Environmental Variables

---

- Variables that can be accessed from any child process
- Common ones:
  - HOME: path to user's home directory
  - PATH: list of directories to search in for command to execute
- Change value:
  - **export** VARIABLE=...

# LC\_\* Environment Variables

---

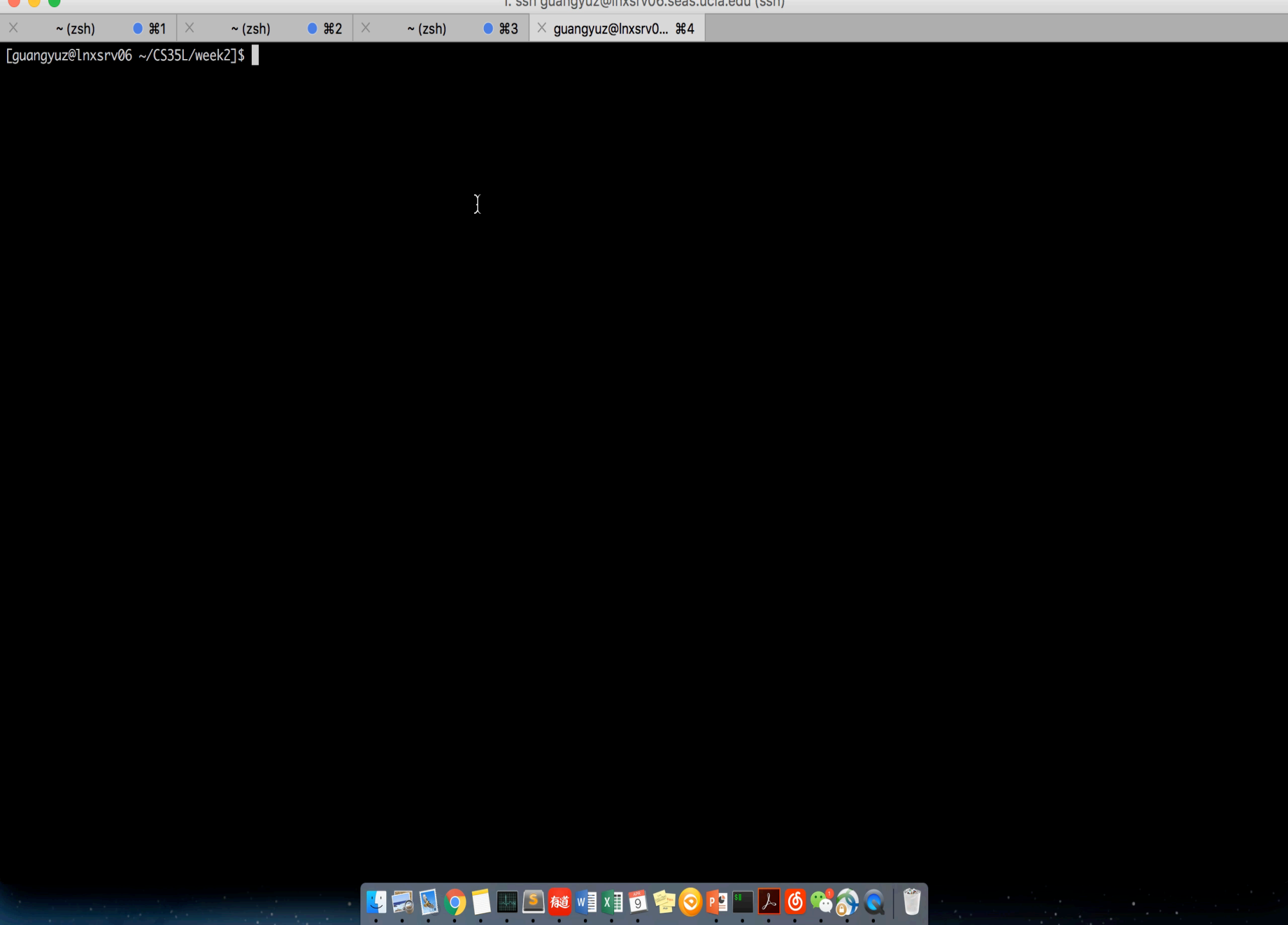
- locale gets its data from the LC\_\* environment variables
  - Examples:
    - LC\_TIME  
Date and time formats
    - LC\_NUMERIC  
Non-monetary numeric formats
    - LC\_COLLATE  
Order for comparing and sorting
-



# Locale Settings Can Affect Program Behavior

---

- Default sort order for the sort command depends:
  - LC\_COLLATE='C': sorting is in ASCII order
  - LC\_COLLATE='en\_US': sorting is case insensitive except when the two strings are otherwise equal and one has an uppercase letter earlier than the other.
- Other locales have other sort orders!



# Text Processing Tools

---

- `sort`: sorts text
  - `wc`: outputs a one-line report of lines, words, and bytes
  - `head`: extract top of files
  - `tail`: extracts bottom of files
-

# Sorting words

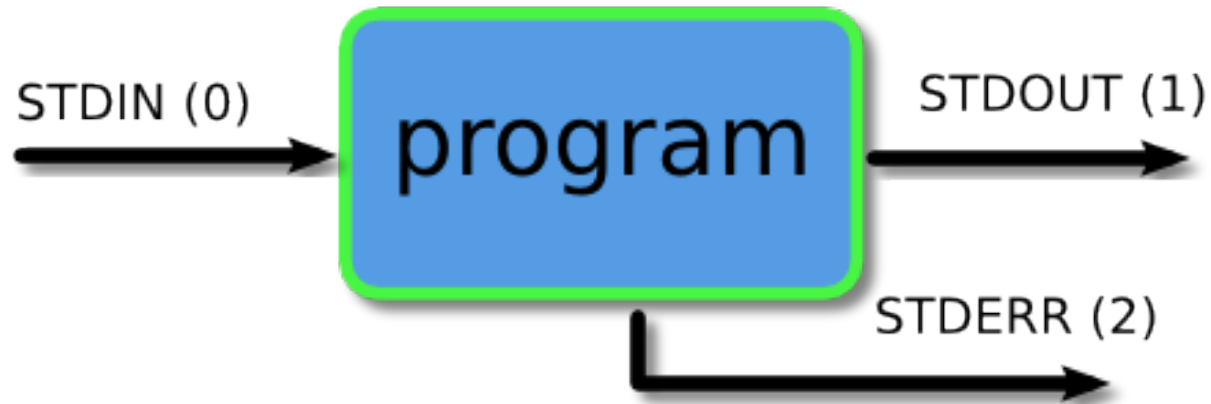
---

- Investigate the 'sort' command – man sort
    - Usage: sort [options] [filename]
    - -b: ignore leading blanks
    - -d: consider only blank and alphabets
    - -r: reverse the results of comparison
    - -u: unique, for duplicate records, output only the first one
  - Sort all words in /usr/share/dict/words
  - Save to your home folder
-

# Basic I/O Redirection

---

- I/O of most programs
  - read from standard input (stdin)
  - Write to standard output (stdout)
  - Send error messages to standard error (stderr)



# Redirection and Pipelines

---

- Redirection
    - Use `command < file` to make program's standard input be file
    - Use `command > file` to make program's standard output be file
    - Use `command >> file` to append program's standard output to file
    - Use `command 2> file` to redirect STDERR to the file specified.
  - Pipeline
    - Use `command1 | command2` to make the standard output of program1 become the standard input of program2
    - Simple output: `echo`
-

# Example

```
[guangyuz@lnxsrv06 ~/CS35L]$ ls -l
total 24
-rw-r--r-- 1 guangyuz csgrad 10 Jan 8 13:38 test.txt
drwxr-xr-x 2 guangyuz csgrad 4096 Apr 4 15:57 week1
drwxr-xr-x 2 guangyuz csgrad 4096 Apr 9 20:58 week2
drwxr-xr-x 4 guangyuz csgrad 4096 Jan 24 15:20 week3
drwxr-xr-x 4 guangyuz csgrad 4096 Feb 5 13:37 week4
drwxr-xr-x 2 guangyuz csgrad 4096 Feb 5 13:35 week5
drwxr-xr-x 6 guangyuz csgrad 4096 Mar 12 21:36 week6
```

What does *ls | head -3* return?

*test.txt*

*week1*

*week2*

How to list out week1 through week3?

• *ls | head -4 | tail -3*

# Search for Text

---

- `grep`
    - Use basic regular expression
    - Usage: `grep [option] [pattern]`
    - Can be integrated to other commands with `|`
  - `egrep`
    - Extended `grep` that uses extended regular expressions
    - These are equal: `grep -E` `egrep` `sed -r`
  - `fgrep`
    - Fast `grep` that matches fixed strings instead of regular expressions
    - These are equal: `fgrep` `grep -F`
-



# Simple grep

---

```
$ who                                Who is logged on
tolstoy tty1 Feb 26 10:53
tolstoy pts/0 Feb 29 10:59
tolstoy pts/1 Feb 29 10:59
tolstoy pts/2 Feb 29 11:00
tolstoy pts/3 Feb 29 11:00
tolstoy pts/4 Feb 29 11:00
austen pts/5 Feb 29 15:39 (mansfield-park.example.com)
austen pts/6 Feb 29 15:39 (mansfield-park.example.com)
```

```
$ who | grep -F austen             Where is austen logged on?
austen pts/5 Feb 29 15:39 (mansfield-park.example.com)
austen pts/6 Feb 29 15:39 (mansfield-park.example.com)
```

---

# Compare difference between files

---

- diff
    - usage:  
*diff original\_file new\_file*  
*diff -u original\_file new\_file*  
*diff -y original\_file new\_file* (output in two columns)
    - function: compare files line by line
  - comm
    - usage: `comm [option] [file1] [file2]`
    - function: compare **sorted files** line by line
  - cmp
    - Compare two files byte by byte. When the files differ, by default, '**cmp**' outputs the byte offset and line number where the first difference occurs.
-

# wget & curl

- A computer program that retrieves content from web servers
- Usage
  - `wget <URL>`, `wget -O new_name <URL>`

```
[guangyuz@lnxsrv07 ~/CS35L/fa18/week2]$ wget -O new https://stringdb-static.org/download/protein.links.v10.5/9606.protein.links.v10.5.txt.gz
--2018-10-07 23:35:25-- https://stringdb-static.org/download/protein.links.v10.5/9606.protein.links.v10.5.txt.gz
Resolving stringdb-static.org (stringdb-static.org)... 104.25.69.109, 104.25.70.109, 2606:4700:20::6819:456d, ...
Connecting to stringdb-static.org (stringdb-static.org)|104.25.69.109|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 65880584 (63M) [application/x-gzip]
Saving to: 'new'

38% [=====>] 25,448,294 7.70MB/s eta 5s
```

- `curl -O <URL>`: Write output to <file> instead of stdout.

```
[guangyuz@lnxsrv07 ~/CS35L/fa18/week2]$ curl -O https://stringdb-static.org/download/protein.links.v10.5/9606.protein.links.v10.5.txt.gz
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           % Dload  % Upload   Total   Spent    Left   Speed
0 62.8M    0 15571    0     0  23692      0  0:46:20 --:--:--  0:46:20 23664
```

# tr: command for translate or transliterate

---

- Usage
    - `tr [options] [set1] [set2]`
    - Function: replace the elements in set1 with corresponding elements from set2
  - Options:
    - `-c`: complement
    - `-d`: delete
    - `-s`: Replace each input sequence of a repeated character that is listed in set1 with a single occurrence of that character
-

# Example: tr

---

Example:

```
echo "abc" | tr [:lower:] [:upper:]
```

ABC

```
echo "password a1b2c3" | tr -d [:digit:]
```

password abc

```
echo "aaa123334" | tr -s a3
```

a1234

Without using |:

```
tr [:lower:] [:upper:]
```

abc

ABC

---

# sed: stream editor

---

- Modifies the input as specified by the command(s)

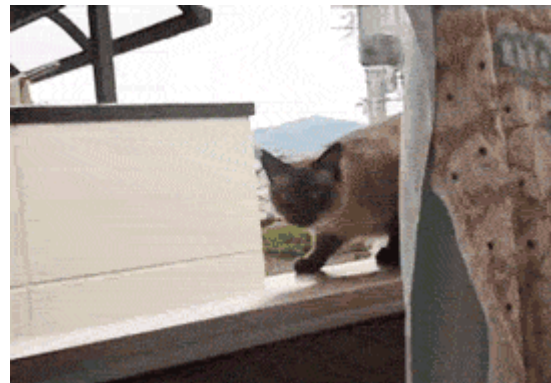
## Usages:

- Printing specific lines or address ranges
    - `sed -n '1p' file.txt`
    - `sed -n '1,5p' file.txt`
    - `sed -n '1~2p' file.txt`
  - Deleting text
    - `sed '1~2d' file.txt`
  - Substituting text - `s/regex/replacement/flags`
    - `sed 's/cat/dog/' file.txt`
    - `sed 's/cat/dog/g' file.txt`
-

# Regular Expression

---

- Notation that lets you search for text that fits a particular criterion, such as “starts with the letter a”
- Easy to learn, but difficult to handle



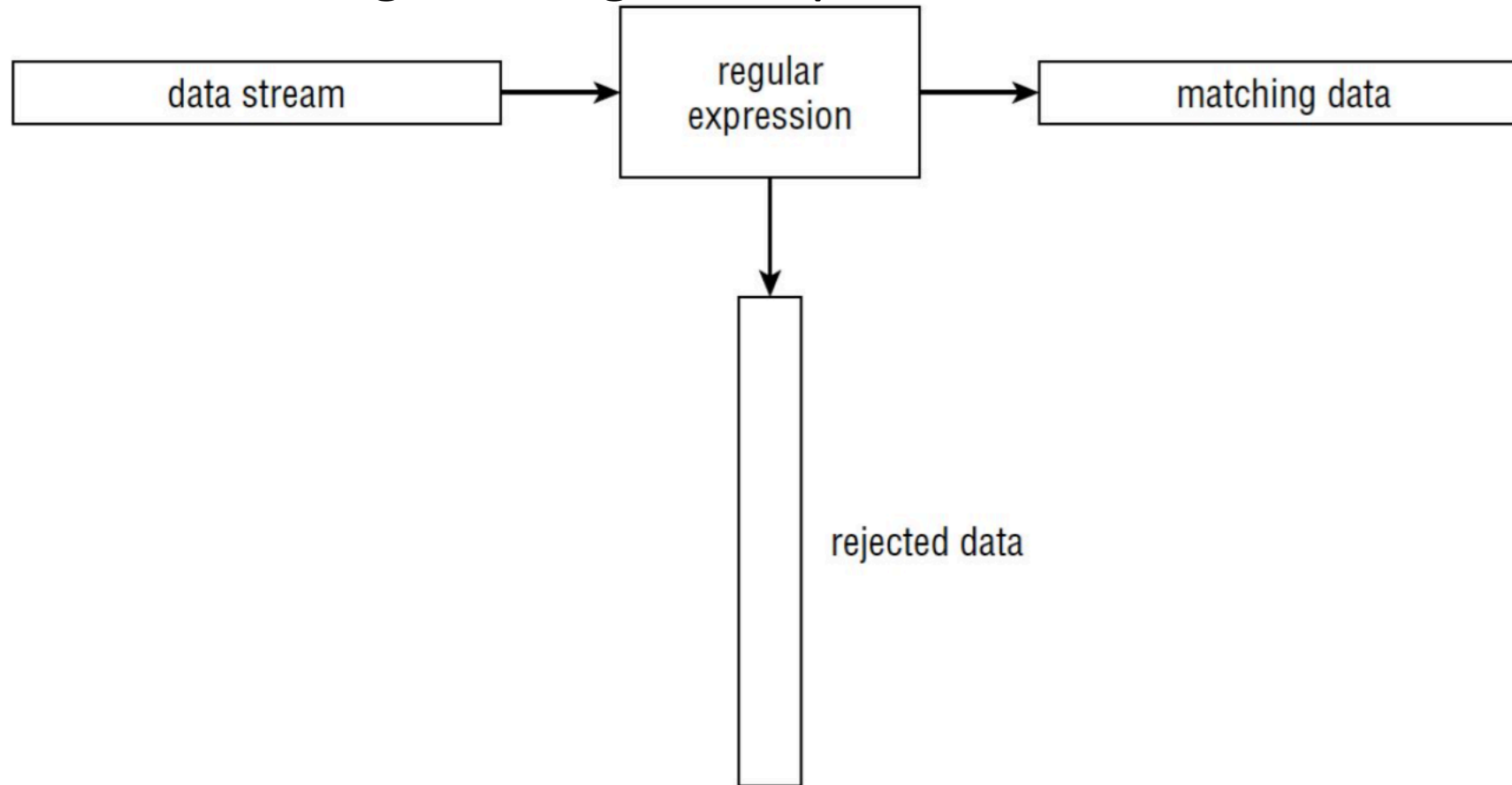
The programmer learns about regular expression, and try to use it in practical project.

---

# Regular Expression $\approx$ Query

---

Match/Filter data against regular expression





# Regular Expression

---

- Useful tools
    - Online test your regex expression  
<http://regexpal.com>
    - Simple regex tutorial  
[https://www.icewarp.com/support/online\\_help/203030104.htm](https://www.icewarp.com/support/online_help/203030104.htm)
    - References in Supplement materials
-

# Regular Expression

---

- Different applications use different types of RE:
    - Programming languages (Python, Perl, Java)
    - Linux (sed, grep)
    - SQL
  - Regular Expression Engine
    - Interprets regular patterns and use them to match texts
    - Two types:
      - BRE: Basic Regular expression
      - ERE: Extended Regular expression
    - BRE and ERE work together. ERE adds ?, +, and |, and it removes the need to escape the metacharacters ( ) and { }, which are *required* in BRE.
-

# Special characters in Regular Expression

---

- Quantification (the number of previous occurrences)
    - `?` (0 or 1)
    - `*` (0 or more)
    - `+` (1 or more)
    - `{}` (specified number)
  - Alternation
    - `[]` (any character in the range)
    - `|` (one case or another)
  - Anchors
    - `^` (beginning of a line)
    - `$` (end of a line)
  - Group
    - `()`
-

# Regular Expressions

Character	BRE / ERE	Meaning in a pattern
\	Both	Usually, turn off the special meaning of the following character. Occasionally, enable a special meaning for the following character, such as for <code>\(...\)</code> and <code>\{...\}</code> .
.	Both	Match any single character except NULL. Individual programs may also disallow matching newline.
*	Both	Match any number (or none) of the single character that immediately precedes it. For EREs, the preceding character can instead be a regular expression. For example, since <code>.</code> (dot) means any character, <code>.*</code> means "match any number of any character." For BREs, <code>*</code> is not special if it's the first character of a regular expression.
^	Both	Match the following regular expression at the beginning of the line or string. BRE: special only at the beginning of a regular expression. ERE: special everywhere.

# Regular Expressions (cont'd)

---

<code>\n</code>	BRE	Replay the nth subpattern enclosed in <code>\(</code> and <code>\)</code> into the pattern at this point. n is a number from 1 to 9, with 1 starting on the left.
<code>{n,m}</code>	ERE	Just like the BRE <code>\{n,m\}</code> earlier, but without the backslashes in front of the braces.
<code>+</code>	ERE	Match <b>one or more</b> instances of the preceding regular expression.
<code>?</code>	ERE	Match <b>zero or one</b> instances of the preceding regular expression.
<code> </code>	ERE	Match the regular expression specified before or after.
<code>( )</code>	ERE	Apply a match to the enclosed group of regular expressions.

---

# Regular Expressions (cont'd)

\$	Both	Match the preceding regular expression at the end of the line or string. BRE: special only at the end of a regular expression. ERE: special everywhere.
[...]	Both	Termed a bracket expression, this matches any one of the enclosed characters. A hyphen (-) indicates a range of consecutive characters. (Caution: ranges are locale-sensitive, and thus not portable.) A circumflex (^) as the first character in the brackets reverses the sense: it matches any one character not in the list. A hyphen or close bracket (]) as the first character is treated as a member of the list. All other metacharacters are treated as members of the list (i.e., literally). Bracket expressions may contain collating symbols, equivalence classes, and character classes (described shortly).
{n,m}	BRE	Termed an <i>interval expression</i> , this matches a range of occurrences of the single character that immediately precedes it. {n} matches exactly n occurrences, {n,} matches at least n occurrences, and {n,m} matches any number of occurrences between n and m. n and m must be between 0 and RE_DUP_MAX (minimum value: 255), inclusive.
( \)	BRE	Save the pattern enclosed between \( and \) in a special <i>holding space</i> . Up to nine subpatterns can be saved on a single pattern. The text matched by the subpatterns can be reused later in the same pattern, by the escape sequences \1 to \9. For example, (ab).*\1 matches two occurrences of ab, with any number of characters in between.

# Examples

Expression	Matches
tolstoy	The seven letters tolstoy, anywhere on a line
^tolstoy	The seven letters tolstoy, at the beginning of a line
tolstoy\$	The seven letters tolstoy, at the end of a line
^tolstoy\$	A line containing exactly the seven letters tolstoy, and nothing else
[Tt]olstoy	Either the seven letters Tolstoy, or the seven letters tolstoy, anywhere on a line
tol.toy	The three letters tol, any character, and the three letters toy, anywhere on a line
tol.*toy	The three letters tol, any sequence of zero or more characters, and the three letters toy, anywhere on a line (e.g., tolstoy, tolWHOttoy, and so on)

# Example

---

"ab*c"	matches a string that has an a followed by zero or more b's ("ac", "abc", "abbc", etc.)
"ab+c"	same, but there's at least one b ("abc", "abbc", etc., but not "ac")
"ab?c"	there might be a single b or not ("ac", "abc" but not "abbc").
"a?b+\$"	a possible 'a' followed by one or more 'b's at the end of the string: Matches any string ending with "ab", "abb", "abbb" etc. or "b", "bb" etc. but not "aab", "aabb" etc.

---



# Example

---

"ab{2}"	matches a string that has an a followed by exactly two b's ("abb")
"ab{2,}"	there are at least two b's ("abb", "abbbb", etc.)
"ab{3,5}"	from three to five b's ("abbb", "abbbb", or "abbbbbb")

# POSIX Bracket Expressions

---

Class	Matching characters	Class	Matching characters
<code>[:alnum:]</code>	Alphanumeric characters	<code>[:lower:]</code>	Lowercase characters
<code>[:alpha:]</code>	Alphabetic characters	<code>[:print:]</code>	Printable characters
<code>[:blank:]</code>	Space and tab characters	<code>[:punct:]</code>	Punctuation characters
<code>[:cntrl:]</code>	Control characters	<code>[:space:]</code>	Whitespace characters
<code>[:digit:]</code>	Numeric characters	<code>[:upper:]</code>	Uppercase characters
<code>[:graph:]</code>	Nonspace characters	<code>[:xdigit:]</code>	Hexadecimal digits

---

# Matching Multiple Characters with One Expression

---

*	Match zero or more of the preceding character
$\{n\}$	Exactly n occurrences of the preceding regular expression
$\{n,\}$	At least n occurrences of the preceding regular expression
$\{n,m\}$	Between n and m occurrences of the preceding regular expression

# Operator Precedence (High to Low)

---

Operator	Meaning
[ . ] [= =] [ : :]	Bracket symbols for character collation
<i>\metacharacter</i>	Escaped metacharacters
[]	Bracket expressions
<i>\( \) \digit</i>	Subexpressions and backreferences
<i>* \{ \}</i>	Repetition of the preceding single-character regular expression
no symbol	Concatenation
<i>^ \$</i>	Anchors

---

# Demo

---

- <https://github.com/ziishaned/learn-regex>
-

# Examples of tr command with regex

---

- Usage: as a part of pipeline
  - e.g. `cat assign2.html | tr -cs 'A-Za-z' '[\n*]' > pre`
- Eliminate everything except alphabet characters, also duplicate words
  - `tr -cs 'A-Za-z' '[\n*]'`
- Transform all upper cases characters to lower cases
  - `tr '[:upper:]' '[:lower:]'`
- Delete all left-over blanks
  - `tr -d '[:blank:]'`

# Examples of sed command with regex

---

- Usage: similar to tr
- Replace \* with +
  - sed s/\\*/\+/g
- Separate words in a sentence
  - sed 's/ /\n/g'
- Format: sed `s/**regExpr**/**replText**/`
- Example

```
echo $PATH | sed 's/::.*//\' #Display the first directory in PATH
```

# Laboratory -- Spell-checking Hawaiian

---

- Build a spelling checker for the Hawaiian language (Get familiar with sort, comm and tr commands)
  - Steps
    - Download a copy of web page containing basic English-to-Hawaiian dictionary
    - Extract only the Hawaiian words from the web page to build a simple Hawaiian dictionary. Save it to a file called hwords (site scraping)
    - Automate site scraping: *buildwords* script  
Usage: **cat hwnwdseng.htm | buildwords > hwords**
    - Modify the command in the lab assignment to act as a spelling checker for Hawaiian
    - Use your spelling checker to check hwords and the lab web page for spelling mistakes
-



# Laboratory -- Spell-checking Hawaiian

---

- The script *buildword*
    - Preprocess
      - Delete whatever before/after the html <table> tag
      - Eliminate html tags, extract words
    - Change upper case characters to lower case
    - Treat ` as `
    - Remove any misspelled Hawaiian language
    - Hints: **don't leave unnecessary information behind** (e.g. duplication, empty lines, spaces, html tags)
-

# Laboratory -- Spell-checking Hawaiian

---

- Hints:

- Run your script on seasnet servers before submitting to CCLE
  - `sed '/patternstart/,/patternstop/d'`
  - delete patternstart to patternstop, works across multiple lines  
will delete all lines starting with patternstart to patternstop
  - The Hawaiian words html page uses `\r` and `\n` for new lines
    - `od -c hwnwdseng.htm` # see the ASCII characters
  - `sed 's/<[^>]*>//g' a.html` # remove all HTML tags
  - You can delete blank white spaces such as tab or space using
    - `tr -d '[:blank:]'`
    - Use `tr -s` to squeeze multiple new lines into one
-