ECS 98F - Regular Expressions

Stephen Ott

Agenda

Today's lecture

- How we can robustly look for patterns in strings
- How we can extract data from strings

Problem scenario

Validating email addresses

You have to design a script to tell whether or not an email address is valid.

Rules for a valid email

- Username
  - Can contain any alpahnumeric character and the speical characters `_`, `-`, and `#`
  - Cannot contain `.`
- Username and Doman must be seperated by a `@`.
  - This must be the only `@` in the string
- Domain
  - Can contain any alphanumeric character and `-`
- Top-level domain
  - Must contain between 2 and 4 alphanumeric characters

Problem scenario

What we want

```
$ echo bob_smith123@gmail.com | email_validator.sh
bob_smith123@gmail.com is a valid email address
```

```
$ echo .no@h.rose@@!#.h | email_validator.sh
.no@h.rose@@!#.h is an invalid email address
```

```
$ for i in email_list.txt; do
    echo $i | email_validator.sh
done
```

Intro to Regular Expressions

A regular expression is a string encoding a “pattern” that matches some set of strings

What you can do with regex

- Find strings
- Validate strings
- Extract strings
- Substitute strings

Where to use regex

- The bash language
- `grep`
- `sed` and `awk`
- Standard library of any programming language

Intro to Regular Expressions

A simple bash script

```
#!/bin/bash

pattern=$1

read email

if [[ "$email" =~ $pattern ]]; then
    echo "$email is a valid email address"
else
    echo "$email is an invalid email address"
fi
```

email_validation_test.sh

- `=~` checks if the string on the left matches the regex pattern on the right

```
$ echo foo | ./email_validation_test.sh foobar
foo is an invalid email address
$ echo foobar | ./email_validator_test.sh foobar
foobar is a valid email address
```

Regex Grammar

Bread and butter

Token	Description
`.`	Matches with *any* single character
`a*`	Matches zero or more `a`'s
`a+`	Matches one or more `a`'s

```
$ echo 'bob_smith123@gmail.com' | ./email_validation_test.sh '.*@.*\..*'
bob_smith123@gmail.com is a valid email address
```

```
$ echo '.@.@@.@' | ./email_validation_test.sh '.*@.*\..*'
.@.@@.@ is a valid email address
```

```
$ echo '@.' | ./email_validation_test.sh '.*@.*\..*'
@. is a valid email address
```

Regex Grammar

Character sets

- Matches with any character in the set
- Ranges, i.e. `a-z` and `0-5` are acceptable

```
$ echo 'stephen123@yahoo.com' | ./email_validation_test.sh '[a-zA-Z0-9_#-]+@[a-zA-Z0-9]+\.[a-zA-Z0-9]+'
stephen123@yahoo.com is a valid email address
```

```
$ echo '.@.@@.@' | ./email_validation_test.sh '[a-zA-Z0-9_#-]+@[a-zA-Z0-9]+\.[a-zA-Z0-9]+'
.@.@@.@ is an invalid email address
```

Negated character sets

- Matches with any character not in the set

```
$ echo 'bob_123@gmail.com' | ./email_validation_test.sh '[^@.]+@[^@.]+\.[^@.]'
bob_123@gmail.com is a valid email address
```

```
$ echo '.@.@@.@' | ./email_validation_test.sh '[^@.]+@[^@.]+\.[^@.]'
.@.@@.@ is an invalid email address
```

```
$ echo '!!$tephens mail@yahoo.com' | ./email_validation_test.sh '[^@.]+@[^@.]+\.[^@.]'
!!$tephens mail@yahoo.com is a valid email address
```

Regex Grammar

Meta sequences

Token	Description	Character Set Equivalence
`[:digit:]`	matches any single digit	`[0-9]`
`[:alpha:]`	matches any alphabetic character	`[A-Za-z]`
`[:alnum:]`	matches any letter or digit	`[A-Za-z0-9]`
`[:word:]`	`[:alnum:]` but with _	`[A-Za-z0-9_]`
`[:space:]`	matches any whitespace character	`[ \t\n\r]`

- These belong inside square brackets

```
$ echo '.@.@@.@' | ./email_validation_test.sh '[[:word:]#-]+@[[:alnum:]-]+\.[[:alnum:]]+'
.@.@@.@ is an invalid email address
```

```
$ echo 'bob_smith123@gmail.com' | ./email_validation_test.sh '[[:word:]#-]+@[[:alnum:]-]+\.[[:alnum:]]+'
bob_smith123@gmail.com is a valid email address
```

```
$ echo '@tephens mail@yahoo.com' | ./email_validation_test.sh '[[:word:]#-]+@[[:alnum:]-]+\.[[:alnum:]]+'
@tephens mail@yahoo.com is a valid email address
```

Regex Grammar

Anchors

Tokens that represent the start and end of the string

Token	Description
^	Start of the string
$	End of the string

```
$ echo '@tephens mail@yahoo.com' | ./email_validation_test.sh '^[[:word:]#-]+@[[:alnum:]-]+\.
[[:alnum:]]+$'
@tephens mail@yahoo.com is an invalid email address
```

```
$ echo 'bob_smith123@gmail.commmmmmmmmmmmmmmmmmm' | ./email_validation_test.sh '^[[:word:]#-]
+@[[:alnum:]-]+\.[[:alnum:]]+$'
bob_smith123@gmail.commmmmmmmmmmmmmmmmmm is a valid email address
```

Regex Grammar

Logical or

Token	Description
`a|b`	Matches with either `a` or `b`

```
$ echo 'bob_smith123@gmail.commmmmmmmmmmmmmmmmmm' | ./email_validation_test.sh '^[[:word:]#-]+@[[:alnum:]-]+\.(com|org|edu|gov)$'
bob_smith123@gmail.commmmmmmmmmmmmmmmmmm is an invalid email address
```

```
$ echo 'bob_smith123@gmail.fake_domain' | ./email_validation_test.sh '^[[:word:]#-]+@[[:alnum:]-]+\.(com|org|edu|gov)$'
bob_smith123@gmail.fake_domain is an invalid email address
```

```
$ echo 'bob_smith123@gmail.info' | ./email_validation_test.sh '^[[:word:]#-]+@[[:alnum:]-]+\.(com|org|edu|gov)$'
bob_smith123@gmail.info is an invalid email address
```

Regex Grammar

Quantifiers

Token	Description
`a{n}`	Matches exactly 3 `a`'s
`a{m,n}`	Matches between `m` and `n` `a`'s, inclusive
`a{m,}`	Matches `m` or more `a`'s
`a{,n}`	Matches no more than `n` `a`'s

```
$ echo 'bob_smith123@gmail.commmmmmmmmmmmmmmmmmm' | ./email_validation_test.sh '^[[:word:]#-]+@[[:alnum:]-]+\.[[:alnum:]]{2,4}$'
bob_smith123@gmail.commmmmmmmmmmmmmmmmmm is an invalid email address
```

```
$ echo 'bob_smith123@gmail.com' | ./email_validation_test.sh '^[[:word:]#-]+@[[:alnum:]-]+\.[[:alnum:]]{2,4}$'
bob_smith123@gmail.com is a valid email address
```

Regex Grammar

Revisiting email validation

- Tweak our script that we've been using so far to accept a candidate email

```
#!/bin/bash

read email

if [[ "$email" =~ ^[[:word:]#-]+@[[:alnum:]-]+\.[[:alnum:]]{2,4}$ ]]; then
    echo "$email is a valid email"
else
    echo "$email is not a valid email"
    exit 1
fi
```

email_validator.sh

Extracting strings

- Now, you have been tasked with extracting a user's username, second-level domain, and top-level domain.

Extracting strings

Capture groups

- Parentheses allow specification of a substring to be later referenced
- Stored in `$BASH_REMATCH` as an array after the use `=~`
  - `${BASH_REMATCH[0]}` stores the whole match
  - Indices `1, 2, ... n` store captures in order of appearance

```
#!/bin/bash

phone_number=$1

if [[ "$phone_number" =~ ^([0-9]{3})-([0-9]{3})-([0-9]{4})$ ]]; then
    echo "Phone number: ${BASH_REMATCH[0]}"
    echo "Area code: ${BASH_REMATCH[1]}"
    echo "Prefix: ${BASH_REMATCH[2]}"
    echo "Line number: ${BASH_REMATCH[3]}"
else
    echo "Invalid phone numnber"
    exit 1
fi
```

capture_group_example.sh

```
$ ./capture_group_example.sh 123-456-7890
Phone number: 123-456-7890
Area code: 123
Prefix: 456
Line number: 7890
```

Extracting strings

The final script

```
#!/bin/bash

email=$1

if [[ "$email" =~  ^([[:word:]#-]+)@([[:alnum:]-]+)\.([[:alnum:]]{2,4}$) ]]; then
    echo "Username: ${BASH_REMATCH[1]}"
    echo "Second-level domain: ${BASH_REMATCH[2]}"
    echo "Top-level domain: ${BASH_REMATCH[3]}"
else
    echo "$email is not a valid email"
    exit 1
fi
```

email_extractor.sh

Regex flavors

POSIX Extended Regular Expressions (ERE)

- Strict superset of BRE
- What we have used thus far with bash's `=~` operator
- Special characters are special by default

POSIX Basic Regular Expressions (BRE)

- Default dialect used by many shell utilities, i.e. `sed` and `grep`
- Special characters like `+` and `{}` have to be escaped with `\`

Perl-Compatible Regular Expressions (PCRE)

- Many more powerful and flexible features than however similiar to the other two
- API's exist for use of this in C, Python, and other languages

BRE	`^\([[:word:]#-]\+\)@\([[:alnum:]-]\+\)\.\([[:alnum:]]\{2,4\}\)$`
ERE	`^([[:word:]#-]+)@([[:alnum:]-]+)\.([[:alnum:]]{2,4})$`
PCRE	`^([\w#-]+)@([[:alnum:]-]+)\.([^\W_]{2,4})$`

Regex flavors

More on PCRE

- Comes with shorthands for frequently used character groups
  - Shorthands can be negated with capitalization

Shorthand	BRE/ERE Equivalent	Character Set Equivalent
`\w`	`[:word:]`	`[A-Za-z_]`
`\d`	`[:digit:]`	`[0-9]`
`\s`	`[:space:]`	`[ \n\t\r]`

- Allows for non-capturing groups using `(?:...)`
- Lookaround constructs provide ability to make assertions about strings without assertions being included in the match

More on grep

grep was made for regex

- Use flags to utilize a full regex grammar
  - `-E` for ERE
  - `-P` for PCRE
  - No flag to use BRE

More useful flags

Flag	Function
`-v`	Select line which don't match the given pattern
`-l`	List files which contain a match for the regex
`-i`	Ignore case when doing the match
`-c`	Print the number of matching lines in each input file
`--color`	Prints matches with colors to highlight what part of the string matched
`-n`	Print each matching line with the line number it came from
`-r`	Run recursively for all files

More on grep

Data wrangling with grep

The Ubiquity of Regex

Python

```
#!/usr/bin/python

import re
from sys import argv

email = argv[1]
match = re.match(r'^([\w#-]+)@([[:alnum:]-]+)\.([^\W_]{2,4})$', email)

if match:
  print(f"Username: {match.group(1)}")
  print(f"Second-level domain: {match.group(2)}")
  print(f"Top-level domain: {match.group(3)}")
else:
  print("Invalid email")
```

email_validator.py

C

```c
#define PCRE2_CODE_UNIT_WIDTH 8
#include <stdio.h>
#include <pcre2.h>
#include <string.h>

int main(int argc, char **argv) {
  PCRE2_SIZE erroffset;
  int errcode;
  uint32_t options = 0, ovecsize = 128;
  const char* patt = "^([\\w#-]+)@([[:alnum:]-]+)\\.([^\\W_]{2,4})$";
  const char* email = argv[1];
  size_t patt_size = strlen(patt), subject_size = strlen(email);

  pcre2_code* re = pcre2_compile(patt, patt_size, options, &errcode, &erroffset, NULL);
  pcre2_match_data* match_data = pcre2_match_data_create(ovecsize, NULL);
  int rc = pcre2_match(re, email, subject_size, 0, options, match_data, NULL);

  if (rc > 0) {
    const char *lables[] = {"Username", "Second-level domain", "Top-level domain"};
    PCRE2_SIZE* ovector = pcre2_get_ovector_pointer(match_data);
    for (PCRE2_SIZE i = 1; i < rc; i++) {
      PCRE2_SPTR start = email + ovector[2*i];
      PCRE2_SIZE slen = ovector[2*i+1] - ovector[2*i];
      printf("%s: %.*s\n", lables[i - 1], (int)slen, (char *)start);
    }
  }
  else if (rc <= 0)
    printf("Invalid email\n");

  pcre2_match_data_free(match_data);
  pcre2_code_free(re);
}
```

email_validator.c

Conclusion

- Regex is a powerful tool that is used everywhere
- Regex can greatly many simplify parsing problems

Helpful links

- Regex 101
  - Explains different tokens of the regex grammar and allows you to experiment
- Regex One
  - Interactive tutorial useful for building up intuition for constructing regex
- Regex Cheatsheet
  - Good overall cheatsheet. Also provides a nice comparisson between regex flavors
- A Regular Expression Matcher
  - Not necessary to learn regex, but demonstrates that the implementation of a regex engine can be quite simple
- Regular Expression in C
  - Blog post about using PCRE in C
- David Doty's ECS 120
  - Not a link, but this class provides a mathmatical basis for regular expressions and what they are capable of