CS 241: Systems Programming
Lecture 23. Regular Expressions I

Spring 2020

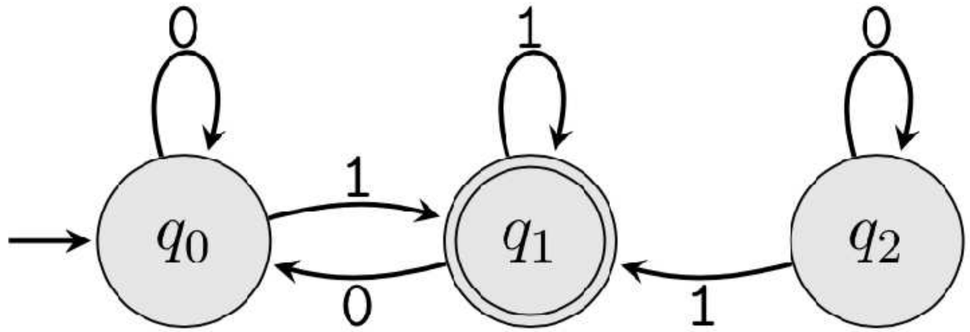
Prof. Stephen Checkoway

Theory of regular languages

Mathematical theory of sets of strings

- You'll see this in CS 383

Connection to finite state machines

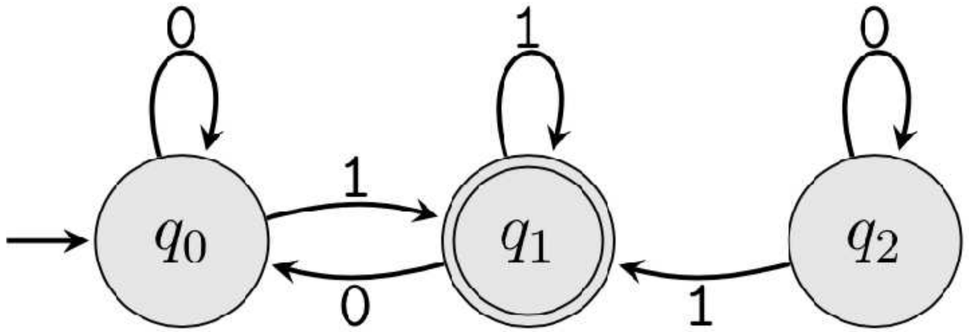


Theory of regular languages

Mathematical theory of sets of strings

- You'll see this in CS 383

Connection to finite state machines



We're going to skip all of this for this course!

Problem we want to solve

Identify and/or extract text that matches a given **pattern**

Examples

- Determine if a text string matches the pattern
- Find all lines of text in a file containing a given word
- Extract all phone numbers from a file
- Extract fields from structured text
- Classify types of text (e.g., compilers need to determine if some text is a number like `0x7D2` or symbols like == or keywords like `double`)
- Find all of the tags in an HTML file

Approach: Use a **regular expression** to specify the **pattern**

grep(1)

grep matches lines of input against a given regular expression (regex), printing each line that matches (or does not match)

```
$ grep 'Computer Science' file
```

- prints each line of `file` that contains the string "`Computer Science`"

More generally,

```
$ grep regex file
```

will print each line of `file` that matches the regular expression `regex`

What is a regular expression?

Text that describes a **search pattern**

Comes in a variety of "flavors"

- Basic Regular Expression (**BRE**)
- Extended Regular Expression (**ERE**)
- Perl-Compatible Regular Expressions (**PCRE**)

Be careful not to confuse with file globbing which uses similar special characters like * and ? but with slightly different meanings

Baseline regex characters

Baseline regex characters

- (period) any single character except newline

Baseline regex characters

- `.` (period) any single character except newline
- `*` 0 or more of the preceding item (greedy)

Baseline regex characters

- `.` (period) any single character except newline
- `*` 0 or more of the preceding item (greedy)
- `^` start of a line

Baseline regex characters

- `.` (period) any single character except newline
- `*` 0 or more of the preceding item (greedy)
- `^` start of a line
- `$` end of the line

Baseline regex characters

- `.` (period) any single character except newline
- `*` 0 or more of the preceding item (greedy)
- `^` start of a line
- `$` end of the line
- `[ ]` match one of the enclosed characters
  - `[a-z]` matches a range
  - `[^ ]` reverses the sense of match
  - put `]` or `–` at start to be a member of the list

Baseline regex characters

- `.` (period) any single character except newline
- `*` 0 or more of the preceding item (greedy)
- `^` start of a line
- `$` end of the line
- `[ ]` match one of the enclosed characters
  - `[a-z]` matches a range
  - `[^ ]` reverses the sense of match
  - put `]` or `–` at start to be a member of the list

Every other character just matches itself; precede any of the above with \ to treat as a normal character that must literally match

Examples

Examples

a	Anything with the letter 'a'

Examples

`a`	Anything with the letter 'a'
`abc`	Anything with the string 'abc'

Examples

`a`	Anything with the letter 'a'
`abc`	Anything with the string 'abc'
`a.c`	'a' followed by any char then 'c'

Examples

`a`	Anything with the letter 'a'
`abc`	Anything with the string 'abc'
`a.c`	'a' followed by any char then 'c'
`^a`	Line starting with 'a'

Examples

`a`	Anything with the letter 'a'
`abc`	Anything with the string 'abc'
`a.c`	'a' followed by any char then 'c'
`^a`	Line starting with 'a'
`a$`	Line ending with 'a'

Examples

`a`	Anything with the letter 'a'
`abc`	Anything with the string 'abc'
`a.c`	'a' followed by any char then 'c'
`^a`	Line starting with 'a'
`a$`	Line ending with 'a'
`^a$`	Line consisting of a single 'a' on it

Examples

`a`	Anything with the letter 'a'
`abc`	Anything with the string 'abc'
`a.c`	'a' followed by any char then 'c'
`^a`	Line starting with 'a'
`a$`	Line ending with 'a'
`^a$`	Line consisting of a single 'a' on it
`a.*b`	'a' then anything else, then 'b' (includes 'ab')

Examples

`a`	Anything with the letter 'a'
`abc`	Anything with the string 'abc'
`a.c`	'a' followed by any char then 'c'
`^a`	Line starting with 'a'
`a$`	Line ending with 'a'
`^a$`	Line consisting of a single 'a' on it
`a.*b`	'a' then anything else, then 'b' (includes 'ab')
`[abc]`	One of 'a', 'b', or 'c'

Examples

`a`	Anything with the letter 'a'
`abc`	Anything with the string 'abc'
`a.c`	'a' followed by any char then 'c'
`^a`	Line starting with 'a'
`a$`	Line ending with 'a'
`^a$`	Line consisting of a single 'a' on it
`a.*b`	'a' then anything else, then 'b' (includes 'ab')
`[abc]`	One of 'a', 'b', or 'c'

Valid identifiers in C (things like variable or function names)
1. start with either a letter or an underscore; and
2. consist of letters, numbers, or underscores.

E.g., `main`, `foo_bar`, `_Okay123XY` are valid identifiers;
but `32x`, `foo-bar`, and `&blah` are not

Which regular expression describes valid C identifiers?

A. `[a-zA-Z0-9_]*`

B. `[a-zA-Z0-9_][a-zA-Z0-9_]*`

C. `[a-zA-Z_][a-zA-Z0-9_]*`

D. `[^0-9][a-zA-Z0-9_]*`

Basic regex (obsolete)

`\{m,n\}` match previous item at least `m` times, but at most `n` times

`\{m\}` match previous item exactly `m` times

`\{m,\}` match previous item at least `m` times

`\( \)` group and save enclosed pattern match

- `\1` the first saved match
- `\5` the fifth saved match
- Using such "back references" makes it not a real regular expression and should be avoided

Extended regex (modern)

`{m,n}` match previous item at least `m` times, but at most `n` times

`( )` group and save enclosed pattern match

`+` match 1 or more of the previous `{1,}`

`?` match previous 0 or 1 time `{0,1}`

`|` match RE either before or after

- `apple|banana`

`(ab|c+){2}` 'abab', 'abc', 'abcccc', 'cab', 'cccab' 'ccccccccc'

Examples

Examples

`(ab|c){2}` 'abab', 'abc', 'cab', 'cc' (ERE)

POSIX character classes

Within brackets [ ], we can use character classes corresponding to those in ctype.h by surrounding the name with `[:` and `:]`

- `alnum`, `digit`, `punct`, `alpha`, `graph`, `space`, `blank`, `lower`, `upper`, `cntrl`, `print`, `xdigit`
- E.g., `[[:digit:][:blank:]]`

Shortcuts (needs "enhanced" regular expressions):

- `\d` is `[[:digit:]]` `\D` is `[^[:digit:]]`
- `\s` is `[[:space:]]` `\S` is `[^[:space:]]`
- `\w` is `[[:alnum:]_]` `\W` is `[^[:alnum:]_]`

Which string does the ERE

```
\([[:digit:]]{3}\) [[:digit:]]{3}-[[:digit:]]{4}
```

match?

A. `([1]{3}) [2]{3}-[3]{4}`

B. `123 456-7890`

C. `(123) 456-7890`

D. `\(123\) 456-7890`

grep(1)

Name comes from ed(1) program command `g/re/p`

```
 grep -E re files
egrep -l re files
egrep -c re files
egrep -n re files
egrep -i re files
egrep -v re files
```

use extended regex (or use egrep)

just list file names

just list count of matches

just list line numbers

ignore case

show non-matching lines

awk(1)

Named after the developers

- A. Aho
- P. Weinberger
- B. Kernighan

Programming language for working on files

Consists of a sequence of pattern-action statements of the form

- `pattern { action }`
- Each line of the input is matched compared to each `pattern` in order; each matching pattern has its associated `action` run

Running AWK

Running

- `$ awk -f foo.awk files` # foo.awk contains the program
- `$ awk prog files` # pattern-action separated by `;`

Understands whitespace separated fields (can change this via `-F` option)

- `$1`, `$2`, `$3`
- `$0` is the whole line

Other variables, just use their names

Patterns

`/re/`	matches the regular expression `re`
`BEGIN`	matches before any input is used (can be used to set variables)
`END`	matches after all input is used (e.g., can print things)
`expr`	matches if the expression is nonzero
`p1,p2`	matches all lines between the line matching p1 and the line matching p2 (including those lines)
	(empty pattern) matches every line

Simple AWK program

Prints the lines of a file with START and END

```
BEGIN { print "START" }
      { print }
END   { print "END"}
```

Actions

An action is a sequence of statements inside `{ }` separated by `;`

- assignment statements var = value
- conditionals/loops: **`if`**, **`while`**, **`for`**, **`do-while`**, **`break`**, **`continue`**,
- **`for`** `(var` **`in`** `array) stmt`
- **`print`** `expr-list`
- **`printf`** `format, expr-list`

A missing action means to print the line

Simple AWK program

Prints lines longer than 72 characters

```
length($0) > 72 { print }
```

Missing action block means print

```
length($0) > 72
```

Sum up a list of numbers

```
BEGIN { SUM = 0 }
      { SUM += $1 }
END   { print "Total is", SUM }
```

Print size and owner from ls -l

```
$ ls -l | awk '{ print $5, "\t", $3 }'
```

Given pop.txt with lines containing zip code, county, population, e.g.,

```
44001, Lorain, 20769
44011, Lorain, 21193
```

what is the awk command to print out the population of Oberlin (zip code 44074)?

A.
```
$ awk -F ', ' '/44074/ { print $3 }'
```

B.
```
$ awk -F ', ' '$0 == 44074 { print $2 }'
```

C.
```
$ awk -F ', ' '$1 == 44074 { print $3 }'
```

D.
```
$ awk -F ', ' '44074 { print $2 }'
```

In-class exercise

https://regex.sketchengine.co.uk Do the four interactive exercises

Grab a laptop and a partner and try to get as much of that done as you can!