CS 241: Systems Programming
Lecture 31. Regular Expressions I

Fall 2025

Prof. Stephen Checkoway

Announcement

- SETs are available

Today - Regular Expressions

- A way to specify patterns used in many programming languages and tools

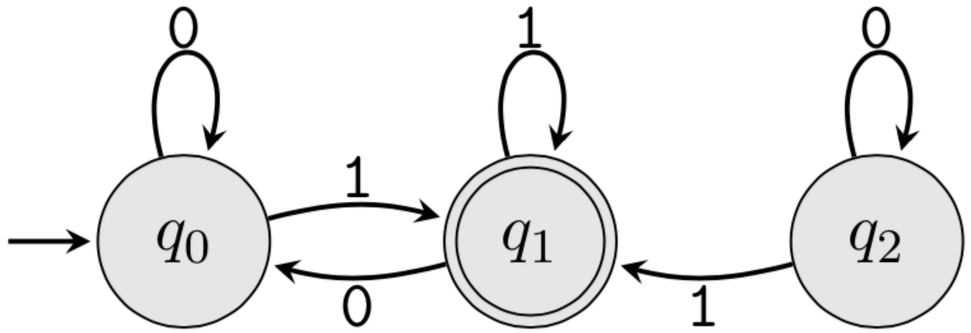
- Used in Rust, Python, Java, C, etc. and command-line tools like grep, sed, and awk

Theory of regular languages

Mathematical theory of sets of strings

- You'll see this in CS 383

Connection to finite state machines

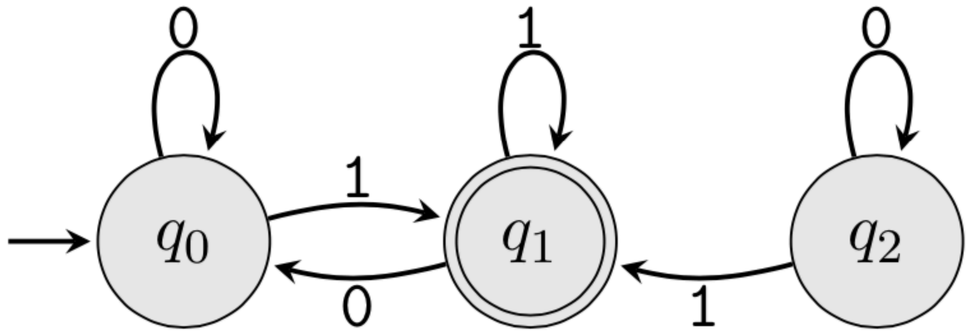


Theory of regular languages

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 `fn`)
- ~~Find all of the tags in an HTML file~~ (Don't use regex for this please)

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

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 Expression (**PCRE**)

Be careful not to confuse with file globbing which uses similar special characters like * and ? but with slightly different meanings (we talked about this with bash)

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'
`[a-zA-Z0-9]`	Anything containing a letter or number

Valid identifiers in Rust* (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 Rust 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_]*`

*Not totally true. Rust has “raw” identifiers as well, ignore those

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 the regex either before or after the pipe

- `apple|banana`

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

What are some expressions that match this extended regex?

```
(ab|c){2}
```

A. Select any option on your clicker

`{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 regex either before or after

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

POSIX character classes

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:][:space:]]`

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:][:space:]]`

Shortcuts (needs "enhanced" basic or extended 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)

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`

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

Example

```
mhogan@mcnulty:~/cs241/F24/lab6$ egrep -r process src/*
src/bin/ps.rs:use process::{Process, Result, user, device_map};
src/bin/ps.rs:#[command(author, version, about = "ps - report process status", long_about = None)]
src/bin/ps.rs:    ///Write information for all processes with controlling terminals. Omit session
leaders.
src/bin/ps.rs:    /// Write information for all processes.
src/bin/ps.rs:    /// Write information for all processes, except session leaders
src/bin/ps.rs:    let vec = Process::all_processes()?;
src/bin/ps.rs:        std::process::exit(1);
src/bin/runnable.rs:use process::Result;
src/bin/runnable.rs:/// (processes and threads) and  the total number of kernel scheduling entities.
src/bin/runnable.rs:            std::process::exit(1);
src/bin/whoami.rs:use process::user;
src/proc.rs:/// Models a Linux process.
src/proc.rs:    /// Look up information about a running process with the given PID.
src/proc.rs:    /// Look up information for the current process.
src/proc.rs:        let pid: i32 = std::process::id() as i32;
src/proc.rs:    /// Returns a list of all running processes.
src/proc.rs:    pub fn all_processes() -> Result<Vec<Self>> {
src/proc.rs:    /// Returns `true` if the process is a session leader.
src/proc.rs:    /// Returns `true` if the process has a controlling terminal.
```

Example

```
mhogan@mcnulty:~/cs241/F24/lab6$ egrep -r "(P|p)rocess" src/*
src/bin/ps.rs:use process::{Process, Result, user, device_map};
src/bin/ps.rs:#[command(author, version, about = "ps - report process status", long_about = None)]
src/bin/ps.rs:    ///Write information for all processes with controlling terminals. Omit session leaders.
src/bin/ps.rs:    /// Write information for all processes.
src/bin/ps.rs:    /// Write information for all processes, except session leaders
src/bin/ps.rs:fn print_full(p: &Process, full: bool, long: bool, ticks:i64, tty:&str) {
src/bin/ps.rs:    let me = Process::for_self()?;
src/bin/ps.rs:    let vec = Process::all_processes()?;
src/bin/ps.rs:        std::process::exit(1);
src/bin/runnable.rs:use process::Result;
src/bin/runnable.rs:/// (processes and threads) and  the total number of kernel scheduling entities.
src/bin/runnable.rs:            std::process::exit(1);
src/bin/whoami.rs:use process::user;
src/lib.rs:pub use proc::Process;
src/proc.rs:/// Models a Linux process.
src/proc.rs:pub struct Process {
src/proc.rs:    /// Process ID.
src/proc.rs:impl Process {
src/proc.rs:    /// Look up information about a running process with the given PID.
src/proc.rs:    /// Look up information for the current process.
src/proc.rs:        let pid: i32 = std::process::id() as i32;
src/proc.rs:    /// Returns a list of all running processes.
src/proc.rs:    pub fn all_processes() -> Result<Vec<Self>> {
src/proc.rs:    /// Returns `true` if the process is a session leader.
src/proc.rs:    /// Returns `true` if the process has a controlling terminal.
```

Which command will return only “oberlin” and “snow”

```
oberlin
winter
snow
Rust
Ferris
turtle
```

241.txt

A. `egrep o 241.txt`

B. `egrep “[a-z]o[a-z]*” 241.txt`

C. `egrep “[a-z]*o[a-z]*” 241.txt`

D. `egrep o* 241.txt`

E. More than one of the above

awk(1)

Named after the developers

- Alfred Aho
- Peter Weinberger
- Brian Kernighan

Programming language for working on files - used for text processing and data extraction

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` # prog is patterns-actions separated by `;`

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

Awk programs can manipulate the fields in a line with

- `$1`, `$2`, `$3` are the first three fields
- `$0` is the whole line

Other variables, just use their names

Simple AWK program

Simple AWK program

Prints the lines of a file with START and END

Simple AWK program

Prints the lines of a file with START and END

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

Simple AWK program

Prints the lines of a file with START and END

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

The first line has the special pattern **`BEGIN`** whose action runs before looking at any lines

Simple AWK program

Prints the lines of a file with START and END

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

The first line has the special pattern **`BEGIN`** whose action runs before looking at any lines

The second line does not have a pattern so its action (print the line) runs for each line

Simple AWK program

Prints the lines of a file with START and END

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

The first line has the special pattern **`BEGIN`** whose action runs before looking at any lines

The second line does not have a pattern so its action (print the line) runs for each line

The final line has the special pattern **`END`** whose action runs after all lines

Sum up a list of numbers

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

Sum up a list of numbers

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

```
$ cat nums
10
39
48
22
51
$ awk -f sum.awk nums
Total is 170
```

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

Expressions in patterns

Examples:

- $3 == "foo" { … } Matches when field 3 is the string foo
- $2 ~ /re/ { … } Matches when field 2 matches the regex re

You can use relational operators: <, <=, ==, !=, >, and >=

You can use match operators: expr ~ /re/ and expr !~ /re/

A bunch of builtin functions including substr, length, and sub (substitute)

The action(s) are performed when the pattern expression evaluates to true

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

AWK example

Prints lines longer than 72 characters

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

Missing action block means print

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

Print size and owner from ls -l

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

```
mhogan@mcnulty:~$ ls -l
total 48
drwx------ 17 mhogan mhogan 4096 Nov 21 22:39 cs241
drwxr-xr-x  2 mhogan mhogan 4096 Sep 10 15:32 Desktop
drwxr-xr-x  2 mhogan mhogan 4096 Sep 10 15:32 Documents
drwxr-xr-x  2 mhogan mhogan 4096 Sep 10 15:32 Downloads
…

mhogan@mcnulty:~$ ls -l | awk '{ print $5, "\t", $3 }'

4096    mhogan
4096    mhogan
4096    mhogan
4096    mhogan
```

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 '/44074/ { print $3 }'
```

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

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

D.
```
$ awk '44074 { print $2 }'
```