Regular expressions

We are here

- Part 1: How computers works
  - Boolean logic, integers
  - Instructions
  - Memory
- Part 2: Software development
  - Compiling (clang, make, …)
  - Code management (regex, git) ⟵ TODAY
- Part 3: Correctness
  - ABIs, specifications
  - Static & dynamic analysis, debugging
- Part 4: Performance
  - CPU pipelines, caches
  - Parallel computation

Regular expressions

Definition

Regular expressions (“regex”) are a mini-language for text pattern matching.

Example

Q: Find all occurences of the word “memory” in the files in this directory.

A:

```
grep 'memory' *
```

Matching

The grep command

```
grep [OPTION...] PATTERNS [FILE...]
```

Options:

- `-E`: “extended” regular expressions (we will use this syntax)
- `-R`: recursive (if a directory is given, look all files in it, incl. subdirectories)
- `-i`: case insensitive (“a” same as “A”)

Patterns:

Use single-quotes (`'`) around `PATTERN` to avoid shell globbing

Files:

if no file provided, `grep` reads its standard input (useful with pipes)

Piping to grep

Q: Find all files in the current directory whose name contains the letter L (lower or upper case)

A:

```
ls | grep -E -i 'L'
```

Introduction to regular expressions

- by default, patterns are looked for in the input **line-by-line**
- strings of “normal” characters are simply matched exactly

```
grep -E 'memory' *
```

Anchors

- the ^ character at the beginning of a regex matches the beginning of a line
- the $ character at the end of a regex matches the end of a line

Examples:

```
grep -E '^int' *
grep -E ' $' *
```

Repetitions

- `?` indicates that the previous character may or may not occur (once)
- `*` indicates that the previous character may occur zero or more times
- `+` indicates that the previous character may occur one or more times
- `{4}` indicates that the previous character must occur 4 times
- `{4,}` indicates that the previous character must occur 4 or more times
- `{4,8}` indicates that the previous character must occur between 4 and 8 times

Examples:

```
grep -E 's?printf' *
grep -E '^ *print' *
grep -E '0b0+' *
grep -E 'e{2,}' *
```

Grouping

Any part of a regex can be grouped using parentheses.
Repetitions then apply to the group instead of a single character.

Examples:

```
grep -E '(Abc)+'                    # matches 'Abc', 'AbcAbc',
                                    #         'AbcAbcAbc', ...
```

Match any character

The dot (“ . ”) matches any character:

Examples:

```
grep -E 'X.Y'                    # matches 'XaY', 'XbY', 'X+Y',
        ...
grep -E 'X.*Y'                   # matches 'XabcY', 'X+-*/Y',
        ...
```

Bracket expressions

- One character can be matched to multiple options using square brackets:

```
grep -E '[abc]XY'              # matches aXY or bXY or cXY
grep -E '0b[01]+'              # matches binary numbers
```

- We can express ranges of characters using a dash:

```
grep -E '[0123456789]+'        # matches decimal numbers
grep -E '[0-9]+'               # ^ equivalent
grep -E '0x[0-9a-fA-F]+'       # matches hexadecimal numbers
grep -E '[A-Z][a-z]*'          # matches words that start with a capital letter
```

- Bracket expressions are negated if the first character is ^:

```
grep -E '[^s]printf'           # matches " printf", "aprintf" ... but not
        "sprintf"
```

Disjunctions

Multiple options can be given using the “ | ” character:

```
grep -E 'system_(startup|shutdown)'    # matches "system_startup" or
        "system_shutdown"
```

Special characters

Special characters can be “escaped” using a backslash (“\”):

```
grep -E 'printf\(.*\)'    # matches "printf("Hello %s", name)"
```

Using regular expressions in `less`

Searching for patterns in the `less` pager is performed by typing “`/`”.

Patterns are specified using regular expressions

Search and replace: sed

```
sed [OPTION...] SCRIPT [FILE...]
```

- Options:
  - `-E`: “extended” regular expressions (we will use this syntax)
  - `-e SCRIPT`: use `-e` when specifying multiple scripts
  - `-i`: edit file in-place (instead of printing)
- Script: Use single-quotes (`'`) around SCRIPT to avoid shell globbing
- Files: if no file provided, `sed` reads its standard input (useful with pipes)

Basic search and replace

```
sed -E 's/REGEX/REPLACEMENT/'
```

- Examples:

```
sed -E 's/python/Python/'                  # replace "python" with "Python"
sed -E 's/printf\(/fprintf\(stderr, /'     # replace "printf(a)" with "fprintf(stderr, a)"
```

- Allow multiple replacements per line:

```
sed -E 's/REGEX/REPLACEMENT/g'             # g stands for global
```

- Use delimiter different from “/”:

```
sed -E 's|REGEX|REPLACEMENT|'
sed -E 's_REGEX_REPLACEMENT_'
```

Advanced search and replace

- In the replacement string, `\1` indicate the first parenthesized group, `\2` the second, etc.:

```
# replace "Hello, World!" with "Bye, World!"
sed -E 's/Hello, ([A-Za-z]*)!/Bye, \1!/'
```

- Groups are numbered in the order of the opening parentheses from the left:

```
sed -E 's/(a(b|z)+)(c+)/{\1}{\3}/g'
#         ^ ^      ^
#         1 2      3
```

Regular expressions in programming languages

Using regular expressions in C

```c
#include <stdio.h>
#include <regex.h>

int main()
{
    regex_t re;

    // REG_EXTENDED: POSIX extended regular expression
    // REG_NOSUB: do not report position of matches
    if (regcomp(&re, "0x[0-9a-fA-F]+", REG_EXTENDED | REG_NOSUB)) {
        error();
        return 1;
    }

    int r = regexec(&re, "Does this contain a hex number, like 0xff ?", 0, NULL, 0);

    if (r == 0) {
        printf("Found\n");
    } else if (r == REG_NOMATCH) {
        printf("Not found\n");
    }

    regfree(&re);

    return r;
}
```

See: `man regex`

Using regular expressions in Python

```
>>> import re
>>> m = re.search(r'0x[0-9a-fA-F]+', 'Does this contain a hex number, like 0xff ?')
>>> m.group(0)
'0xff'
```

> documentation