Regular expressions, grep, and awk

Bio724D: Fall 2023

2023-11-13

Regular expressions

Regular expressions are a formal way to define pattern matching in strings.

- Finding patterns that match a query of interest is so common in computing that many programs and programming languages (e.g. Python, R, Perl, Javascript, etc) include regular expression facilities
- In the Unix toolchain, `grep`, `sed`, and `awk` are the three main command line tools we'll explore that exploit regular expressions

Regular expression syntaxes

Different tools and languages often have slightly different syntax for defining regular expressions.

Most Linux command line tools support “Basic Regular Expressions” (BRE) and “Extended Regular Expressions” (ERE). Some tools also support “Perl Compatible Regular Expression” (PCRE).

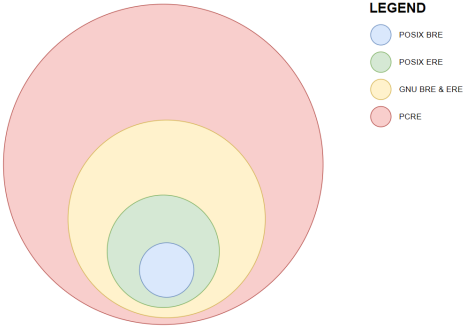

Figure 1: Regular expression syntaxes in Linux. Source link

Regular expressions overview

Typically a regular expression is composed of two classes of characters:

- Regular characters – characters which are interpreted literally. i.e. They match themselves.
- Metacharacters – characters that have special meanings in terms of pattern matching; e.g. anchors, grouping, ranges, etc. Metacharacters must be "escape" if you want to find literal matches.

Examples

- `^Hello` – `^` is a metacharacter (anchor), the letters are regular characters
- `\ttop$` – '`\t`' is the metacharacter for a tab,'$' is the line ending anchor, `the letters in top` are regular characters
- `\([0-9][0-9][0-9]\)` – see discussion in class.

Grep

grep – “global regular expression search and print”

What does grep do?

Applies regular expression matching to one or more files and return lines of text that match the provided regular expression query.

Calling grep with EREs

By default grep uses BRE syntax, but in our examples we’ll prefer ERE syntax:

- `grep -E` – use regular expression syntax
- `egrep` – alias to `grep -E`

Grep metacharacters

- `.` – period matches any single character.
- `?` – preceeding item matches 0 or 1 times
- `*` – preceeding item matches or or more times
- `+` – preceeding item matches 1 or more times
- `^` and `$` – line start and end metacharacters
- `|` – “or” operator;
- Grouping – with parentheses
- Bracket expressions – match any single character in brackets

Some of the most useful grep options

- `-i` – ignore case
- `-v` – inverts match
- `-w` – matches whole words
- `-o` – returns just the matching text
- `-c` – returns the number of lines with that matched
- `--color` – color output to highlight matching text

Grep live coding regex examples

Using the `saccharomyces_cerevisiae_R64-4-1_20230830_trimmed.gff.gz` data set (see wiki):

- Create a new GFF file that includes all the features (except the chromosomes themselves) on chromosomes I and II. How many features are there?
- From your set of features on chromosomes I and II, what is the number of features that are classified as genes or pseudogenes?
- From the gene features on chromosomes I and II, what number include the term “orf_classification=Dubious”? How many genes are not classified as “Dubious”?
- How many of the genes on chromosomes I and II include an entry of the form “ID=XXXX” in the attributesx (column 9) field? How many include the text “gene=XXXX” (e.g. gene=SEO1) in the attributes field?

Awk

Awk is a programming language that is designed to efficiently work on tabular data based on pattern matching.

Kernighan's concise description of Awk, part I

```
Structure of an AWK program:

An awk program is a sequence of pattern-action statements

    pattern { action }
    pattern { action }

A pattern is a regular expression, numeric expression, string expression
or combination; an action is executable code, similar to C.

Operation:
    for each file
       for each input line
          for each pattern
             if pattern matches input line
                do the action

If there is no pattern, the action is performed on each input line.  If
there is no action, the line is printed if it matches the pattern.  The
special pattern BEGIN matches before any input has been read; the special
pattern END matches after all input has been read.

Usage:
    awk 'program' [ file1 file2 ... ]
    awk -f progfile [ file1 file2 ... ]
```

Awk program, basic template

```
# BEGIN rule evaluated before lines are processed
BEGIN {
    initial actions
}

# simple patttern action written as one line
pattern1 {action1}

# more complicated pattern-action statement
# might include flow control like if-else, for loop, etc
pattern2 {
    action2
}

# a pattern without an action (default to printing)
pattern3

# an action without a pattern, applies to all lines
{action4}  # increment count of lines

# END rule evaluated after all lines are processed
END {
    final actions
}
```

Kernighan's concise description of Awk, part II

```
AWK features:

    input is read automatically across multiple files
    lines are split into fields called $1, ..., $NF; $0 is the whole line)
        default split is by white space
        changing FS to some other value (string or RE) affects split
        change FS by assigning to it, or by -F'...' on commandline
    variables contain string or numeric values
        no declarations: using a variable declares it
        initialized to 0 and empty string
        type determined by context and use: the type is set by the
        last operation, and might be string or number or both.  for
        example, x = 1 makes x a number, x = "1" makes it a string.
    operators work on strings or numbers
        coerce the type according to context (e.g., to string for printing)
    built-in variables for frequently-used values; see below
    associative arrays (arbitrary subscripts): x["anything"]
    regular expressions in /.../ (like egrep)

    control flow statements are similar to C
        if-else, while, for, do (but no switch), break, continue
        for (i in array)
           sets i to each subscript of associative array in turn
        next: start next iteration of main loop
        exit: leave main loop, go to END block
```

Awk program, concrete example

```
BEGIN {
    FS = "\t"    # Tab is the field delimiter in input files
    OFS = ","    # Use commas as the field delimiter in output
}

$0 ~ /^#/ { next } # regular expression pattern matching
NF != 9  { next }  # validate number of fields (NF)

# Redirect rows to new CVS files based on their size, add a new field too
$3 == "gene" {
    if ($5 - $4 < 300) {
        $10 = "small"
        print $0 > "small_genes.csv"
        smallct += 1}
    else {
        $10 = "big"
        print $0 > "big_genes.csv"
        bigct += 1}
}

END {
    print "Small genes: " smallct
    print "Big genes: " bigct
}
```

Kernighan's concise description of Awk, part III

```
Basic AWK programs:

Operators include C operators like + - * / % = += -= *= /= %= && || !
  Expressions are almost the same as C.

x ~ /re/, s !~ /re/  string matches/does not match re.

Strings are concatenated by being adjacent:
    hw = "hello" "world" sets hw to "helloworld".
    Watch out; this often has surprising properties.

These are all one-liners:
  { print NR, $0 }  precede each line by its line number
  { $1 = NR; print }    replace first field by the line number
  { print $2, $1 }  print field 2, then field 1 (and nothing else)
  { temp = $1; $1 = $2; $2 = temp; print }   flip $1, $2, print whole line
  { $2 = ""; print }    zap field 2
  { print $NF }     print last field
  NF > 0        print non-empty lines
  NF > 4        print lines with more than 4 fields
  $NF > 4       print line if last field is greater than 4
  NF > 0        {print $1, $2}  print two fields of non-empty lines
  /regexpr/     print lines that match regepxr
  $1 ~ /regexpr/    print lines where first field matches regexpr
  END { print NR }  line count: print number of records at the end
```

Awk built-in functions (from Kernighan's Awk help)

```
Awk strings and string functions are 1-origin; be careful.

    length(s) length of a string
        length(array) returns number of elements
    n = index(s, f)
        returns index of f in s, or 0 if not there
    n = match(s, re)
        index where re matched in s, or 0 if no match
    nsub = sub(re, repl, target)
        replaces first instance of re in target by repl
        returns 0 if no match
    nsub = gsub(re, repl, target)
        replaces all instances of re in target by repl
        returns 0 if no match, number of replacements otherwise
    str = substr(s, start, length)
        returns substring of s starting at start, up to length
        characters (default is rest of string).  works sensibly
        if you go off the ends.  note: origin is 1.
    s = toupper(str)
    s = tolower(str)
        map case
    s = sprintf("...", exprlist)
        formats expressions, returns string result

There are also some of the usual math functions: int, sqrt, exp, log,
sin, cos, atan2, rand (uniform between 0 and 1), srand(new_seed).
```

GNU Awk (gawk) extends/adds built-in functions

See the GNU Awk Manual, Section 9.1 for a full list. Some useful ones include:

String manipulation

- `match(string, regexp [, array])` – extended version of `match` allowing you to capture regex groups into an array
- `split(string, array [, fieldsep [, seps ] ])` – split a string by defininging a separator (`fieldsep`) (can be a regex)
- `patsplit(string, array [, fieldpat [, seps ] ])` – split a string by defining what's between the separators (`fieldpat`)

Others

- `system(command)` – run a command outside of awk and then return to the awk program.
- `strftime([format [, timestamp [, utc-flag] ] ])` – get current system time and return it as a formatted string

Kernighan's Awk Help

See Brian Kernighan's full "Awk help" document at this link:

https://www.cs.princeton.edu/courses/archive/spring19/cos333/awk.help

Redirection and piping in Awk

- Awk has a redirection operator: > (“smart” in that subsequent calls append to the file not overwrite)
  - `print items > output-file` also `print items >> output-file`

```
print $1, $2, $5 > output.txt
```

- Awk has a pipe operator: ‘|’
  - print items | command

```
awk -F"\t" '$3 == "gene" {print $1, $3, $5-$4  | \
"sort -nr -k3,3" }' yeast.gff
```

Note that the command in the pipe call needs to be quoted.

Awk live coding examples

- For each gene feature, output the feature length (start = field 4, end = field 5)
- Calculate the average length of genes in the yeast genome
- Create a table giving the length of each chromosome and the number of genes per chromosome
- For each gene feature, extract the ID and orf_classification from the attributes column (field 9) and create a new file with the ID and orf_classification as a new 10th and 11th columns
- Using Awk, parse the yeast GFF file and for each record, output the record to a file with a name corresponding to its feature type. For example, all genes should be sent to a file `gene.gff`, all mRNAs to a file `mRNA.gff`, etc.
- For each genes, output the gene's ID, gene and Note subfields found in the attribute column. For the Note subfield, replace instances of the string "%20" with spaces. If there is no gene or Note subfields output NA for each of them respectively.