Basics of “regular expressions”

Srikumar K. S.

19 July 2024

Contents

Regular expressions (abbr. “regex”) are a common pattern description language based on which efficient pattern processing can be done. They are not fully general, but are sufficient for many simple data extraction and transformation needs that every programming language has a regex library included in the standard package. They all have mostly the same concepts, but sometimes differ in syntax in small ways due to the context of usage. We’ll concern ourselves with `grep` here and you can learn the differences on your own once you know this.

`grep` (the “GNU regular expression parser”) command is fully described on the grep man page including all the component constructs of the language, but I offer a commonly used subset here to give you a flavour of how to use it.

1 Searching for words and phrases

To search for the exact phrase “King Kong versus Godzilla” in a movie script text file, you simply do `grep 'King Kong versus Godzilla' FILENAME`. The first argument of `grep` is the regular expression or "pattern" to search for. If the `FILENAME` is omitted, `grep` will search its *stdin* for the pattern line by line.

A few things to note -

1. That entire pattern **must** occur on a line in its entirety and can't be split across multiple lines. Now, it is possible to extend the notion to do multi-line search, but grep is largely useful for problems where the entire pattern is not split across lines since its output is line by line.
2. The pattern as given is case sensitive, meaning it will not match `king kong versus godzilla`. To make it ignore the case, use the `-i` flag before giving the pattern.
3. Every character matters - i.e. the given pattern will not match against `King Kong versus  Godzilla` (note the extra space character between `versus` and `Godzilla`).

2 Wildcard

The "`.`" (period) character in a pattern will match against any character including white space. So the pattern `King Kong.versus.Godzilla` will match against `King Kong-versus-Godzilla` also, `King KongSversusOGodzilla` as well.

So you might already see that your regular expression may match more than the intended set of patterns (termed higher "recall") or match fewer than the intended set (termed higher "precision"). Often, if you manage high recall, you may have sacrificed precision and vice versa. You'll have to test against a known set to ensure you're hitting the right spot for your problem at hand.

So how do you match against a literal period character "`.`" in the text you're searching? For that, you "escape" the special meaning of "`.`" in the pattern by prefixing it with "\". So `King Kong\.versus\.Godzilla` will only match the text "King Kong.versus.Godzilla" in the file. The "\" character can be used to escape the special meaning of any of the other characters we discuss here too. It also **imparts** special meaning to some otherwise ordinary characters as well. For example "`\s`" stands for any white space character including the tab character.

3 Character sets

Supposing I wish to match "King Kong" as well as "Ping pong". Can I describe both using a single regex pattern? Yes. You do it by first noticing the common parts – i.e. "_ing _ong" and then look at what are the options for the "blank" portions. The first blank may be occupied by either "K" or "P" and the second blank may be occupied by either "K" or "p" (lower case). This notion of "set of allowed characters" is specified using square brackets like this – "`[KP]ing`

`[Kp]ong`".

The characters listed within "`[]`" form a "character set" and will match a single character that occurs in the specified set. You can, for example, match "any decimal digit" using "`[0123456789]`". It can get tedious to put that long expression whenever you need to idea of "any decimal digit", so that can also be abbreviated as "`[0-9]`", which means exactly the same thing. You can similarly specify character ranges as well – like "`[A-Za-z]`" which will match any capitalized or lower case alphabetical character. If you want your character set to include the "`-`" character, you'll have to place it at the start like this "`[-0-9\s]`" (which matches "any decimal digit or hyphen or any white space character").

Negation: To express the idea of "any character that is not in this set", you place the "^" character at the start of the character set – i.e. "`[^0-9]`" will match any character **other than** a decimal digit.

4 Alternatives

If you want to match "either the word 'Dolphin' or the word 'Orca' ", you can express that idea using the "or" operator – "`\|`". Note that special meaning is being imparted to "`|`" by the prefix "\".

For example, we could've also solved the problem in the previous section using the (longer) pattern – "`King Kong\|Ping pong`" – which is to be read as " 'King Kong' or 'Ping pong' ".

5 Repeated matches

We're often interested in searching for patterns such as "a decimal number" which may have more than one decimal digit and we don't know in advance how many to expect. There are some constructs that help describe such patterns.

You've seen character patterns so far. Suffixing a character pattern with "`{m,n}`" will cause the pattern to match `m` to `n` consecutive occurrences of the pattern. So if you want to describe the portion of a number that occurs after the decimal digit and restrict is to a minimum of 1 and a maximum of 3 digits, you can specify that as "`\.[0-9]{1,3}`". (Note how we "escaped" the special meaning of "." to turn it into a literal period.)

If you want to express the idea of "at least m occurrences", just omit the "n" part (but keep the comma) – i.e. like "`{m,}`". Similarly to express the idea of "at most n occurrences", omit the "m" part (but keep the comma) – i.e. like "`{,n}`". The latter case also includes "zero occurrences". To match an exact number of times, you can either give "`{m,m}`" or abbreviate it as "`{m}`".

"`{1,}`" describes "one or more occurrences" and since this is a commonly used pattern it has a special character for it “`\+`”.[1]

Similarly, the notion of “zero or more occurrences” is also commonly used and has a special character for it “`\*`”.[2]

These are considered “eager matching operators”, meaning they will match as many characters as possible. So if you have a text “xxxxx” and you’re using “`x\+`” to match it, though the notion of “one or more xs” includes the idea of “3 xs”, it will match the whole run of them **eagerly**.

6 Grouping

You may have come across UUIDs which are identifiers usually written out like - `4a1df6b0-f626-40d2-95ad-f7f2f8f8d1fd`. Can we, and if so how do we, describe such UUIDs using a regular expression?

First we notice that each “part” separated by the hyphen character consists of one or more “hexadecimal” characters. This idea we may describe using the regex pattern “`[0-9a-f]\+`”.

Pause and think: Is the pattern “`[0-9a-f]\+`” precise or a more loose match than we need?

So we may conceive of the UUID pattern as “hexdigits followed by one or more occurrences of (hyphen followed by hexdigits)”, where we’ve placed the group like “-f626” in parentheses to indicate it needs to be considered as a “unit”.

`grep` lets you specify such groupings using “`\(`”/“`\)`” matched pairs. The “” escape character is needed because ”(” and “)” will match literally those characters.

Using that we can express the “(hyphen followed by hexdigits)” part as – “`\(-[0-9a-f]\+\)`”. To now apply the “one or more of” to that entire group, we simply place “`\+`” after the group like this - “`\(-[0-9a-f]\+\)\+`”. So you see that “`\+`” applies to the “unit” that immediately precedes it, be it a single character, or a parenthesized group.

Pause and think: How would you then say “exactly 4 occurrences of the group (hyphen followed by hexdigits)”.

Now we can complete a possible description of a UUID using the regex – “`[0-9a-f]\+\(-[0-9a-f]\+\)\+`”.

Pause and think: That final regex pattern will also match “2-3-4-5-6-7-8-9” which is not a valid UUID. The parts of the UUID are fixed and must have

[1]In other situations, the “+” and “*” is usually not escaped like that. In the context of `grep` though, “+” by itself means the literal character and so “+” is needed to impart the special meaning of “one or more” to the preceding item.

[2]In other situations, the “+” and “*” is usually not escaped like that. In the context of `grep` though, “+” by itself means the literal character and so “+” is needed to impart the special meaning of “one or more” to the preceding item.

exactly the same fixed number of characters. So can you modify the regex to match UUIDs strictly (i.e. with greater precision)?

7 Invisible characters

The special character "^" matches the "start of the line". This isn't visible as a character in the text, but represents the cursor position at the start of the line **before** the first character on the line.

Similarly, the character "$" matches the "end of the line" and represents the cursor position **after** the last character on the line.

So if you want to match against all lines that are only hyphens, you can use a pattern like – "`^-\+$`".

8 There's more

Regex libraries offer a lot more than what I've described here, including special support for character classes, unicode characters, back references, look-ahead and look-behind, other invisible character positions such as "word boundary" and more. However, the above collection should serve you well for many common needs.

Important: While the language is richer than described here, do not underestimate the value of **composing** the above features. For example, you can provides options ("`\|`") within groups ("`\(\)`") and ask for at least 2 occurrences ("`{2,}`"), and so on.

To learn about the other features, you can refer to the grep man page pointed to earlier.