Notas sueltas - Mastering Regex

Alonso Martinez

Introduction

Searching Text Files: Egrep

El utility *egrep* viene incluído en MacOS y otros! El comando `egrep` toma el primer argumento "entrecomillado" como un regex, y los argumentos restantes como los archivo(s) en los cuales buscar.

```
$ egrep 'regex' <file1> <file2>
```

Nota importante: *egrep* remueve los *newlines* de cada línea al buscar. Un *newline* se escribe literalmente como `\n`.

Algunos metacaracteres:

Ejemplos simples: `^` (*caret*) & `$` (*dollar*). *caret* representa el inicio de una línea, y *dollar* representa el fin de la línea. Actúan de cierta forma como anclas. Son especiales en el sentido que hacen *match* (coincide,empareja) con posiciones en la línea, no con texto particular

Clases de caracteres

El bloque regex `[<chars>]` deja listar caracteres con los que quieres coincidir. Por ejemplo, la expresión `[ea]` hace match con las letras `e` o{.underline} `a`. Por ejemplo, la expresión `gr[ea]y` hace match con las palabras *gray* y *grey*, las dos posibles ortografías[1]. El orden en el que están los caracteres dentro de las clases no interesa. La implicación es más como $char1 \vee char2$. En ese sentido se pueden pensar más como *conjuntos* y no clases de caracteres, pero en ese caso los elementos del conjunto no están separados por comas.

Dentro de las clases de caracteres podemos usar otro metacaracter: el guión (`-`). Dentro de una clase de caracteres, el guión indica un rango. Por ejemplo, `[1-6]`

[1] si se dice asi?

hace match con cualquier número entre 1 y 6. La expresión `[a-z]` hace match con cualquier letra del alfabeto, pero en minúsculas{.underline}. Vale la pena decir que el guión solo se considera metacaracter en el contexto de una clase de caracteres, fuera de ella se interpreta como un literal. Dicho esto, si no indica un rango no se considera metacaracter. Por ejemplo `[-a]` hace match con los caracteres `-` o `a`.

Otro ejemplo de este fenómeno, son los (usually) metacaracteres `?`, `.`, que dentro del contexto de una clase de caracteres (clase o charclass por brevedad) se interpretan como literales; van a hacer match con texto que contenga "?" o ".". No obstante, fuera de clases, son metacaracteres que tiene su propio significado. Por ejemplo, `?` se interpreta como *uno o más* del patrón anterior.

> Consider character classes theri own mini language. The rules regarding wich metacharacters are supported (and what they do) are completely different inside and outside character classes.

Existe un patrón similar a las clases de caracteres, las clases negadas. En vez de hacer match con los caracteres listados en la clase, hace match con cualquier caracter *salvo* los que están en la clase. Volviendo a la analogía de conjuntos, es el complemento de un conjunto. Ese patrón se escibe como `[^<chars>]`.[2]

Una vez más, este es un buen ejemplo de cómo el significado o estatus de un caracter depende de dónde se usa. En este caso el *caret* no quiere decir inicio de línea, sino que se interpreta como un negador de clase, puesto que se está utilizando en contexto de clases.

Dot & Pipe

El metacaracter `.` (dot, punto) se interpreta como "cualquier caracter". Una vez más, un punto dentro de una charclass pierde estatus de metacaracter, y se vuelve un literal.

Ya habíamos discutido que una expresión del tipo `cat` hace match con texto que contiene una `c`seguida de una `a`, y asi. Por ejemplo, "cat", "vacation", etc... Es decir, es como pedir encontrar `c` *&*, `a`, *&* `t`. El operador lógico es *and*. Para el operador lógico *or* tenemos un nuevo metacaracter: el *pipe* (`|`). Usualmente nos referemimos a esto como alternancia, donde cada una de las opciones se conoce como subexpresión o alternativa. El *pipe* es otro metacaracter que se interpreta como literal dentro de una clase, lo cual tiene sentido ya que los caracteres dentro de las clases tienen un *or* implícito.

Vale la pena destacar que hay otro metacaracter que podemos usar, y su uso no está tan alejado de su significado en el lenguaje natural. Para encapsular

[2] es buen momento para hacer notar que usamos la notación de corchetes `<>`para denotar que dentro de los corchetes va lo de interés.

alternancias, podemos usar paréntesis `(, )`. Volviendo al ejemplo de las diferentes maneras de escribir "gray", la expresión que usamos para ejemplificar charclasses `gr[ae]y` es equivalente a la expresión `gr(a|e)y`. Lo cual también es un buen ejemplo de a qué nos referíamos con "or implícito" dentro de las clases.

Una distinción importante entre clases y alternancias. Cuando usamos una alternancia, las subexpresiones pueden ser expresiones regulares completas por sí mismas, mientras que una clase de caracteres solo hace match {con un solo caracter}{.underline}

En el caso de *egrep*, hay un flag opcional, `-i` que hace las expresiones *case-insensitive*

```
$ egrep -i '<expression>' <file1> <file2>
```

Word Boundaries

Las expresiones del tipo `cat` hacen match no solo con la palabra, sino con strings que tienen el patrón incrustado, como "vacation". Para resolver este problema *egrep* tiene (o por lo menos, mi versión) herramientas llamadas *metasecuencias* que actúan como fronteras de palabras. Estas metasecuencias son `\<` y `\>`, que actúan como `^` y `$` para marcar inicio y fin de la línea, pero para palabras[3]. Y por palabras, léase secuencias alfanuméricas.

Por si solos `<` y `>` no son metacaracteres, son literales que dan match con "<" y ">" respectivamente. Les llamamos metasecuencias porque solo combinadas con otro caracter, el *backslash* `\`, se convierten en no-literales.

Table 1: Resumen de metacaracteres vistos hasta ahora.

Metacharacter	Name	Matches
`.`	dot	Any one character
`[ ]`	character class	Any character listed
`[^ ]`	negated character class	Any character not listed
`^`	caret	The position at start of the line
`$`	dollar	The position at end of line
`\<`		Start of word
`\>`		End of word
`|`	pipe	Matches eiter expression it separates
`( )`	parentheses	limiter

[3]esta es la primera implementación de regex que veo que hace esto. Quizás sea específico a *egrep*. Por ejemplo RegExr no tiene esto

Quantifiers

Los cuantificadores sirven para marcar cantidades de caracteres. Por ejemplo, uno o ninguno, varios, todos los que puedas.

El metacaracter `?` quiere decir opcional. Actúa haciendo al caracter que lo precede opcional en los strings que hacen match con la expresión. Por ejemplo `colou?r` hace match con "color y con"colour".

El metacaracter `+`quiere decir "uno o más del caracter al que precede", y `*`quiere decir "cualquier número del caracter precedente, incluido cero".

En conjunto, `+`, `?`, `*` se llaman cuantificadores.

Table 2: Summary of quantifiers.

	Minimum Required	Maximum to try
`?`	none	One allowed; none required
`*`	none	Unlimited allowed; none required
`+`	one	Unlimited allowed; one required

Asi como hay cuantificadores para al menos uno, ilimitados, etc... hay una forma de especificar entre n y m coincidencias (*matches*). Eso se logra con la metasecuencia `{<n>,<m>}`.[4]

Parentheses and Backreferences

Más allá de su papel de "agrupadores", los paréntesis tienen más usos. En algunas implementaciones de regex los paréntesis "tienen memoria" de el texto con el cual coincide la expresión contenida en ellos. A estos a veces se les dice *capturing groups*.

Para el cookbook:

```
\<([A-Za-z]+) +\1\>
```

Busca palabras repetidas[5]

A la metasecuencia `\1` se llama *backreference* y como su nombre sugiere, hace referencia a el texto capturado. Si se tienen más paréntesis de captura las metasecuencias como `\2`, `\3`, etc... están disponibles. Los grupos de paréntesis se van numerando de izquierda a derecha.

[4] nótese que no hay espacio entre `<n>` y `<m>`

[5] pero *egrep* solo las ve si están en la misma línea.

En algunas implementaciones, como RegExr, las backreferences se hacen con `$1, $2`, etc... [6]

Para incluír metacaracteres como literales en un patrón, se "escapan". Es decir, se preceden con un *backslash* (\).

Otro ejemplo para el cookbook

```
"[^"]*"
```

Encuentra cosas contenidas entre comillas

Más ejemplos. Adios *egrep*

Otras implementaciones (como la de Perl, que es medio estándar) otras construcciones y tienen un conjunto más ámplio de metacaracteres.

Table 3: Nuevos metacaracteres

Metachar	Matches
`\t`	Tab character
`\n`	Newline
`\r`	Carriage-return
`\s`	Any whitespace
`\S`	Anything not in `\s`
`\w`	Short for `[a-aA-Z0-9_]` *i.e* word
`\W`	Negated `\w`
`\d`	Short for `[0-9]` *i.e* a digit
`\D`	Anythong not `\d`

Modifying text with regex

En Perl, el comando

```
$var =~ m/regex/
```

Busca coincidencias en el texto. Hay una manera de hacer más que eso, y además de buscar, reemplazar texto con con un patrón como este

[6] El libro menciona que hay ciertas versiones de *egrep* que tienen un bug con la opción `-i` y backreferences. Por eso no puede hacer ambos, y solo hace backreferencing. Es decir, puede encontrar "cat cat", pero no "The the". La versión instalada en mi computadora tiene ese problema.

```
$var =~ s/regex/replacement/
```

Nótese como cambió el comando que se le da a Perl de `m`, que podemos leer como *match*, a `s`, que podemos leer como substitute.

Como siempre, las diagonales `/` delimitan la expresión, y también el patrón de reemplazo.

Nota: Mientras *egrep* tiene *word boundaries*, los metacaracteres `\<` y `\>`, Perl y otras implementaciones inspiradas en Perl tienen el *catch-all* `\b` que sirve como comodín entre `\<` y `\>` y coincide con tanto inicio como fin de palabra.

Lookaround

Los *lookarounds* son como los conceptos de *line start*, *word-boundary*, etc... en el sentido de que no coinciden con texto sino con posiciones. De hecho, los *line start* y constructos similares son casos particulares de *lookarounds*. Vemos dos tipos particulares: *lookahead* y *lookbehind*. Ambos tiene versiones positivas y negativas

El *lookahead* se fija en el texto de adelante (hacia la derecha) para verificar si coincide con la subexpresión que contiene, y tiene éxito si dicho texto coincide con la subexpresión. Un *lookahead* positivo se hace con la expresión `(?=)`. Decimos positivo para referirnos a que tendrá éxito si la subexpresión contenida entre `(?=` y el paréntesis de cierre `)` coincide con el texto al que se está mirando; en este caso a la derecha.

El *lookbehind* se fija en el texto "de atrás" es decir, el de la izquierda. El patrón para un *lookbehind* positivo es el patrón `(?<=)` donde una vez más, `(?<=` actúa como paréntesis de apertura, y `)` como paréntesis de cierre.

Algo muy importante de los *lookarounds* es que no consumen texto. Es decir, si coinciden con algún string o una parte de el, el texto de coincidencia no forma parte de lo que se extrae al final. Justo como *caret* o *dollar*, no coinciden con texto sino con posiciones. Por ejemplo, teniendo el string "200 pounds" la expresión `\d+` coincidiría con uno o más dígitos, en este caso coincidiría con todo el número "200". Ahora bien, con el mismo texto, la expresión `(?=\d+)` coincidiría con el *inicio* del número "200". Ahora el *lookbehind* de la expresión `(?<=\d+)` coincidiría con el *final* del número "200". Trataremos de ilustrarlo mejor

```
 200 pounds
^   ^
|   L Coincide con (?<=\d+)
L Coincide con (?=\d+)
```

Claro que la magia del *lookbehing* es cuando se combina con expresiones que si consumen texto. El hecho de que coinciden con posiciones dentro del texto y no consumen texto por si solos, las hace una herramienta como *caret* o *dollar*, muy útiles y una buena adición al arsenal.