Regular Expressions

Mechanics

Copyright © Software Carpentry 2010
This work is licensed under the Creative Commons Attribution License
See http://software-carpentry.org/license.html for more information.
# Notebook #1

<table>
<thead>
<tr>
<th>Site</th>
<th>Date</th>
<th>Evil (millivaders)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baker 1</td>
<td>2009-11-17</td>
<td>1223.0</td>
</tr>
<tr>
<td>Baker 1</td>
<td>2010-06-24</td>
<td>1122.7</td>
</tr>
<tr>
<td>Baker 2</td>
<td>2009-07-24</td>
<td>2819.0</td>
</tr>
<tr>
<td>Baker 2</td>
<td>2010-08-25</td>
<td>2971.6</td>
</tr>
<tr>
<td>Baker 1</td>
<td>2011-01-05</td>
<td>1410.0</td>
</tr>
<tr>
<td>Baker 2</td>
<td>2010-09-04</td>
<td>4671.6</td>
</tr>
</tbody>
</table>

---

Regular Expressions

Mechanics
# Notebook #2

<table>
<thead>
<tr>
<th>Site/Date/Evil</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Davison/May 22, 2010/1721.3</td>
<td></td>
</tr>
<tr>
<td>Davison/May 23, 2010/1724.7</td>
<td></td>
</tr>
<tr>
<td>Pertwee/May 24, 2010/2103.8</td>
<td></td>
</tr>
<tr>
<td>Davison/June 19, 2010/1731.9</td>
<td></td>
</tr>
<tr>
<td>Davison/July 6, 2010/2010.7</td>
<td></td>
</tr>
<tr>
<td>Pertwee/Aug 4, 2010/1731.3</td>
<td></td>
</tr>
<tr>
<td>Pertwee/Sept 3, 2010/4981.0</td>
<td></td>
</tr>
</tbody>
</table>

## Regular Expressions

Mechanics
This pattern matches:

- one or more characters
- a slash
- a single upper-case letter
- one or more lower-case letters
- a space
- one or two digits
- a comma if one is there
- a space
- exactly four digits
- a slash
- one or more characters
How?
How?

Using *finite state machines*
Match a single 'a'

\[ \text{Match a single 'a'} \]

Regular Expressions

Mechanics
Match a single 'a'

start here
Match a single 'a'

match this character
Match a single 'a'

must be here
at the end

Regular Expressions
Mechanics
Match a single 'a'

must be here at the end
Match one or more 'a'
Match one or more 'a'

match this
as before
Match one or more 'a'

match again and again

match this

as before
Match one or more 'a'

match this as before
don't have to stop here the first time,
just have to be here at the end

match again and again
Match one or more 'a'

match again and again

don't have to stop here the first time, just have to be here at the end

match this as before

Regular Expressions Mechanics
Match 'a' or nothing
Match 'a' or nothing

transition is "free"
Match 'a' or nothing

transition is "free"

So this is '(a | )'
Match 'a' or nothing

transition is "free"

So this is '(a|)'

Which is 'a'?
Match 'a' or nothing

transition is "free"

So this is '(a | )'
Which is 'a'?
Match zero or more 'a'
Match zero or more 'a'

Combine ideas
Match zero or more 'a'

Combine ideas

This is 'a*'
Match zero or more 'a'

Combine ideas
This is 'a*'
What regular expression is this?
What regular expression is this?

\[ a^+ | (b(c|d)) \]
Action at a node depends only on:
Action at a node depends only on:

- arcs out of that node
Action at a node depends only on:
- arcs out of that node
- characters in target data
Action at a node depends only on:
- arcs out of that node
- characters in target data

Finite state machines have *no memory*
Action at a node depends only on:
- arcs out of that node
- characters in target data

Finite state machines have no memory

Means it is impossible to write a regular expression to check if arbitrarily nested parentheses match
Action at a node depends only on:
- arcs out of that node
- characters in target data

Finite state machines have *no memory*

Means it is impossible to write a regular expression to check if arbitrarily nested parentheses match

"((((....))))" requires memory
Action at a node depends only on:
- arcs out of that node
- characters in target data

Finite state machines have *no memory*

Means it is impossible to write a regular expression to check if arbitrarily nested parentheses match

"((((....))))" requires memory (or at least a counter)
Action at a node depends only on:
- arcs out of that node
- characters in target data

Finite state machines have *no memory*

Means it is impossible to write a regular expression to check if arbitrarily nested parentheses match "(((....)))" requires memory (or at least a counter)

Similarly, only way to check if a word contains each vowel once is to write $5! = 120$ clauses
Why use a tool with limits?
Why use a tool with limits?

They're fast
Why use a tool with limits?

They're fast

- After some pre-calculation, a regular expression only has to look at each character in the input data once
Why use a tool with limits?
They're fast
- After some pre-calculation, a regular expression only has to look at each character in the input data once
It's readable
Why use a tool with limits?
They're fast
- After some pre-calculation, a regular expression only has to look at each character in the input data once
It's readable
- More readable than procedural equivalent
Why use a tool with limits?
They're fast
- After some pre-calculation, a regular expression only has to look at each character in the input data once
It's readable
- More readable than procedural equivalent
And regular expressions can do a lot more than what we've seen so far