AWK

A Short Introduction

Remington Furman

Thursday, March 8th, 2019

Outline

Sed

- No talk about Awk can fail to mention `sed`.
- "Stream Editor", from Bell Labs in 1974
- Automated version of the Unix `ed` editor
- One of the earliest tools to use regular expressions

Sed

- Operates on input text line by line
- Simple semantics
  - If line matches some pattern, perform action
- Widely known for the `s/search/replace/g` command
- Has a limited set of single letter commands

Sed

- Worth learning and very useful
- Great for one-line scripts
  - Especially in Unix command pipelines
- Is Turing complete, but that would be painful
- Has no (user-defined) variables
- Switch to AWK if sed can't do your task

AWK

- Another text stream processing language
- Developed at Bell Labs in 1977
  - Alfred **Aho**
  - Peter **Weinberger**
  - Brian **Kernighan**
- These folks knew their compiler theory

Awk

- Has a built-in execution model
  - Makes scripts simpler to write
- Basic idea, for each line:
  - `pattern { action }`
- If a line matches a pattern, then the action is performed
- Patterns are often regular expressions
  - Can also be other tests/conditions

Event-based programming

Another way to look at awk:

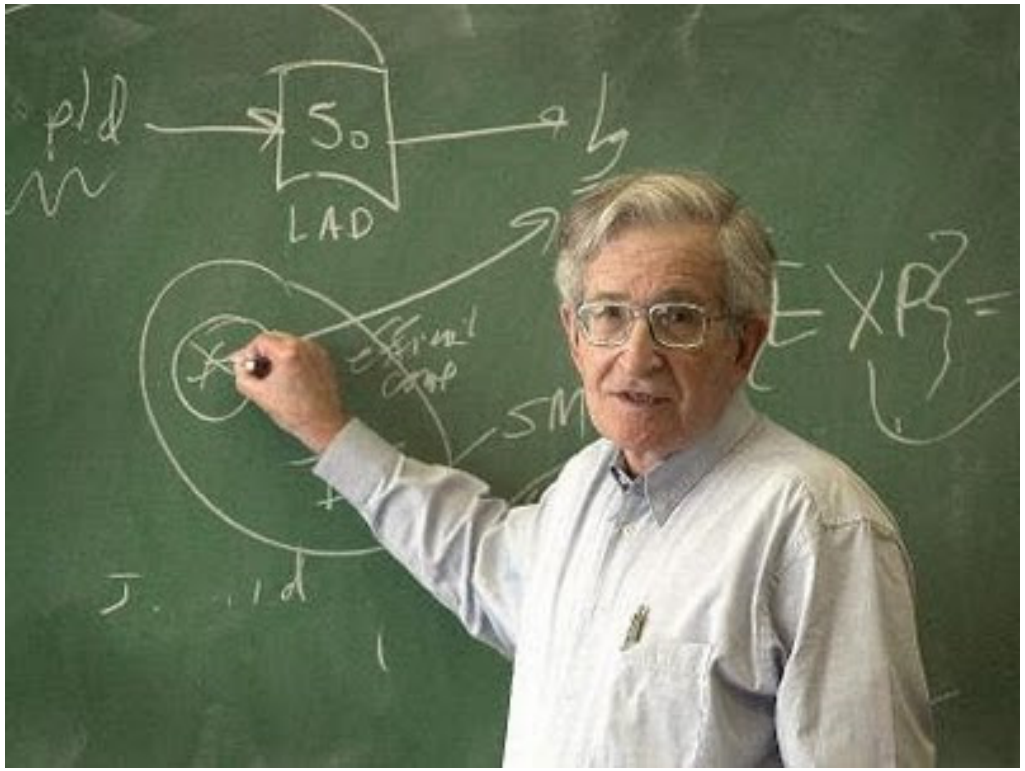
- An event based programming language
- Events are regular expressions (or other conditions)

Obligatory XKCD Reference

Obligatory regexp warning

- Regular expressions are great for parsing bits of text
- Don't try to parse nested structures like:
  - Whole chunks of HTML, XML, YAML, etc
- Why?

Chomsky knows it won't work



People on the internet get mad

21 Answers oldest newest **votes**

1811

You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The cannot hold it is too late. The force of regex and HTML together in the same conceptual space will destroy your mind like so much watery putty. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes. HTML-plus-regexp will liquify the nerves of the sentient whilst you observe, your psyche withering in the onslaught of horror. Regex-based HTML parsers are the cancer that is killing StackOverflow *it is too late it is too late we cannot be saved* the trangession of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) *dear lord help us how can anyone survive this scourge* using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes *using regex* as a tool to process HTML establishes a breach *between this world* and the dread realm of corrupt entities (like SGML entities, but *more corrupt) a mere glimpse* of the world of reg **ex parsers for HTML will ins**tantly transport a *programmer's consciousness* into a world of ceaseless screaming, he comes~~ the pestilent sl~~ithy regex-infection wil**l devour your HT**ML parser, application and existence for all time like Visual Basic only worse *he comes he comes do not fight* h**e comes, h**is unholy radiance *destroying all enlightenment, HTML tags* ***leaking from your eyes like liq****uid pain*, the song of regular expres~~sion parsing~~ will ext*inguish the voices of mor****tal man from the sp****here I can see it can you see it it is beautiful* the f `inal snuf` fing *of the lie****s of Man ALL IS LOST ALL IS*** LOST the *pony he comes* he comes he~~comes~~ **mes the ich**or permeates *all* MY FACE MY FACE *oh god n****o NO NOOOO*** NO stop *the angles* are **not real ZALGO IS** TONY THE **PONY, HE COMES**

Have you tried using an XML parser instead?

link | edit | flag

edited Nov 14 at 0:18 bobince

community wiki

Back to awk

- Basic idea, for each line:
  - `pattern { action }`
- If a line matches a pattern, then the action is performed

Sane defaults

- Either pattern or action can be omitted
  - A blank pattern matches all lines
  - A blank action prints the whole input line
- These commands are the same:
  - `$ awk '// { print }' input.txt`
  - `$ awk '{ print }'    input.txt # No pattern`
  - `$ awk '//'           input.txt # No action`

Special patterns

- `BEGIN` - Action is run before any input is read
  - Initialize variables
  - Print headers
- `END` - Action is run after all input is read
  - Process variables
  - Print footers

Records and Fields

- A core concept in Awk
- Awk will automatically parse input lines for you
- By default
  - Each line is a **record**
  - Each whitespace separated bit of text is a **field**
- You can change the field and record delimiters
  - Useful for comma separated values (CSV files)

Records and Fields

- The entire input line is stored in variable `$0`
- The first field is in `$1`, etc
- These variables can be used in patterns and actions
  - Lines are parsed before patterns are tested
- Note: the delimiters between fields are not saved

Examples

- Simple `grep`:
  - `$ awk '/regexp/' input.txt`
- Simple `cut`:
  - `$ awk '{ print $2 } ' input.txt`
- Simple cat :
  - `$ awk '{print $0}' input.txt input.txt`
- printf :
  - `$ awk '{ printf "%0.2f\n", $2 } ' input.txt`

grep

```
$ cat input.txt
line 1, red,     0xFF0000
line 2, green,   0x00FF00
line 3, blue,    0x0000FF

$ awk '/green/' input.txt
line 2, green,   0x00FF00
```

cut

```
$ cat input.txt
line 1, red,     0xFF0000
line 2, green,   0x00FF00
line 3, blue,    0x0000FF

$ awk '{ print $3 } ' input.txt
red,
green,
blue,
```

cat

```
$ cat input.txt
line 1, red,    0xFF0000
line 2, green,  0x00FF00
line 3, blue,   0x0000FF

$ awk '{print $0}' input.txt input.txt
line 1, red,    0xFF0000
line 2, green,  0x00FF00
line 3, blue,   0x0000FF
line 1, red,    0xFF0000
line 2, green,  0x00FF00
line 3, blue,   0x0000FF
```

printf

```
$ cat input.txt
line 1, red,    0xFF0000
line 2, green,  0x00FF00
line 3, blue,   0x0000FF

$ awk '{ printf "%0.2f\n", $2 }' input.txt
1.00
2.00
3.00
```

printf 2

```
$ cat input.txt
line 1, red,    0xFF0000
line 2, green,  0x00FF00
line 3, blue,   0x0000FF

$ awk '{ print $3": "strtonum($4) }' input.txt
red,: 16711680
green,: 65280
blue,: 255
```

Bash one-liners

- When calling `awk` from a shell, quoting becomes important
- In general, wrap the awk commands in single quotes
  - This prevents the shell from messing with it

Variables

- Variables don't need to be declared before use
- Either **strings** or **floating point numbers**
- Also arrays and associative arrays

Built-in variables

Some commonly used built-in variables:

- `NR` - number of records seen so far
- `NF` - number of fields in current record
- `RS` - record separator (default is newline)
- `FS` - field separator (default is whitespace)

FS example

Parsing comma separated values with custom field separator

```
$ cat input.txt
line 1, red,    0xFF0000
line 2, green,  0x00FF00
line 3, blue,   0x0000FF

$ awk -v FS=', *' \
      '{ print $1, $2, $3, $4 }' input.txt
line 1 red 0xFF0000
line 2 green 0x00FF00
line 3 blue 0x0000FF
```

Variable example

Simple `wc` (word count)

```
$ cat input.txt
line 1, red,     0xFF0000
line 2, green,   0x00FF00
line 3, blue,    0x0000FF

$ awk '{ w += NF; c += length +1 }; \
       END {print NR, w, c} ' input.txt
3 12 75

$ wc input.txt
 3 12 75 input.txt
```

Bash shebang line

For writing longer scripts:

- `#!/usr/bin/awk -f`
- Need to use the `-f` flag to make awk happy
- The shell will expand `./script.awk` to:
  `/usr/bin/awk -f ./script.awk`

POV

Slow rapidly scrolling text by pausing every few seconds

```
#!/usr/bin/awk -f

BEGIN { print_time = 1;
        stall_time = 2;
        start = systime(); }
{
  if (systime() > start + print_time) {
    system("sleep " stall_time);
    start = systime(); }
  print;
}
```

CSV to JSON

How to convert this from CSV to JSON?

```
$ cat morganm.txt
S1,CHERRY RED
S2,PERSIMMON
S3,TANGERINE
```

Large CSV and JSON libraries, or awk?

CSV to JSON

```
#!/usr/bin/awk -f

BEGIN { FS="," ; print "{" }

{ print "\""$1"\": {"
  print "    \"name\": \""$2"\""
  print "  },"
}

END { print "}" }
```

The ugliness comes from having to quote all the quotes.

CSV to JSON

```
$ ./morganm.awk morganm.txt
{
"S1": {
    "name": "CHERRY RED"
  },
"S2": {
    "name": "PERSIMMON"
  },
"S3": {
    "name": "TANGERINE"
  },
}
```

Oops

JSON doesn't like a trailing comma.
Let's fix that while still using the awk programming model.

CSV to JSON 2

```
$ cat morganm2.awk
#!/usr/bin/awk -f

BEGIN { FS="," ; print "{" }

{ if (NR > 1) print ","
  print "\""$1"\": {"
  print "    \"name\": \""$2"\""
  printf "  }"
}

END { print "\n}" }
```

CSV to JSON 2

```
$ ./morganm2.awk morganm.txt
{
"S1": {
    "name": "CHERRY RED"
  },
"S2": {
    "name": "PERSIMMON"
  },
"S3": {
    "name": "TANGERINE"
  }
}
```

Questions?

Any questions?