*Advanced Programming Techniques*

AWK

Christopher Moretti

# The “1-2-3’s” of Testing

In response to questions about writing tests for Assignment 1:

Build upon what you learned in 217 in terms of testing: path testing, boundary testing, stress testing, etc.

Try to enumerate broad categories on several axes — as things get small, as things get big, as things change, as things relate to the heart of the task.

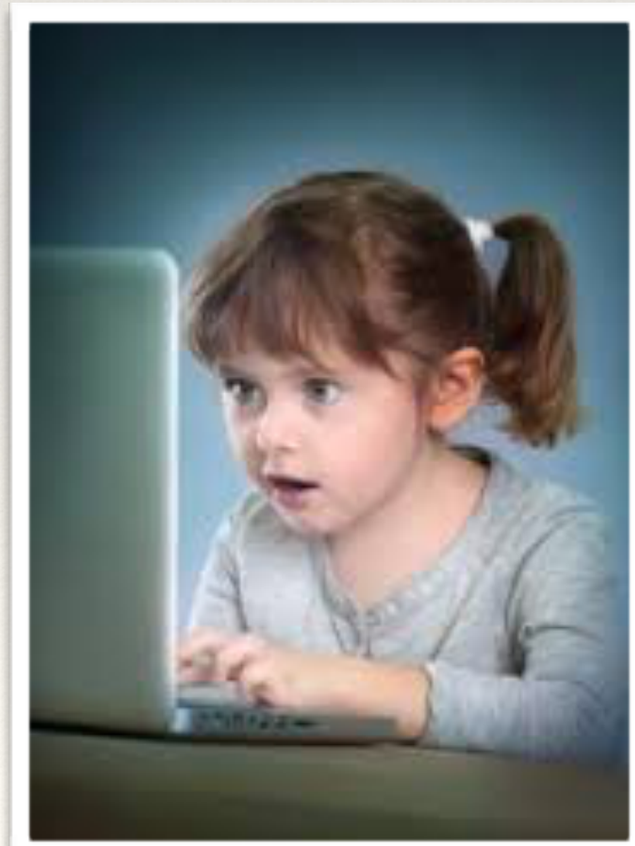
Then permute these possibilities, throwing away the ones that aren’t applicable.

# Exploratory Data Analysis

Data: thousands of lines like this:

```
8/27/1883    Krakatoa         8.8
5/18/1980    Mt St Helens     7.6
3/13/2009    Costa Rica       5.1
```

Task: find all the events with magnitude greater than 6



# Java Version

```
import java.util.Scanner;
import java.util.regex.Pattern;

public class Quake {

    public static void main(String [] args) {
        Scanner in = new Scanner(System.in);
        Pattern tab = Pattern.compile("\t|\n");
        in.useDelimiter(tab);

        while (in.hasNext()) {
            String date = in.next();
            String name = in.next();
            double magn = in.nextDouble();

            if(magn > 6. )
                System.out.println(date+"\t"+name+"\t"+magn);
        }
    }

}
```

# C Version

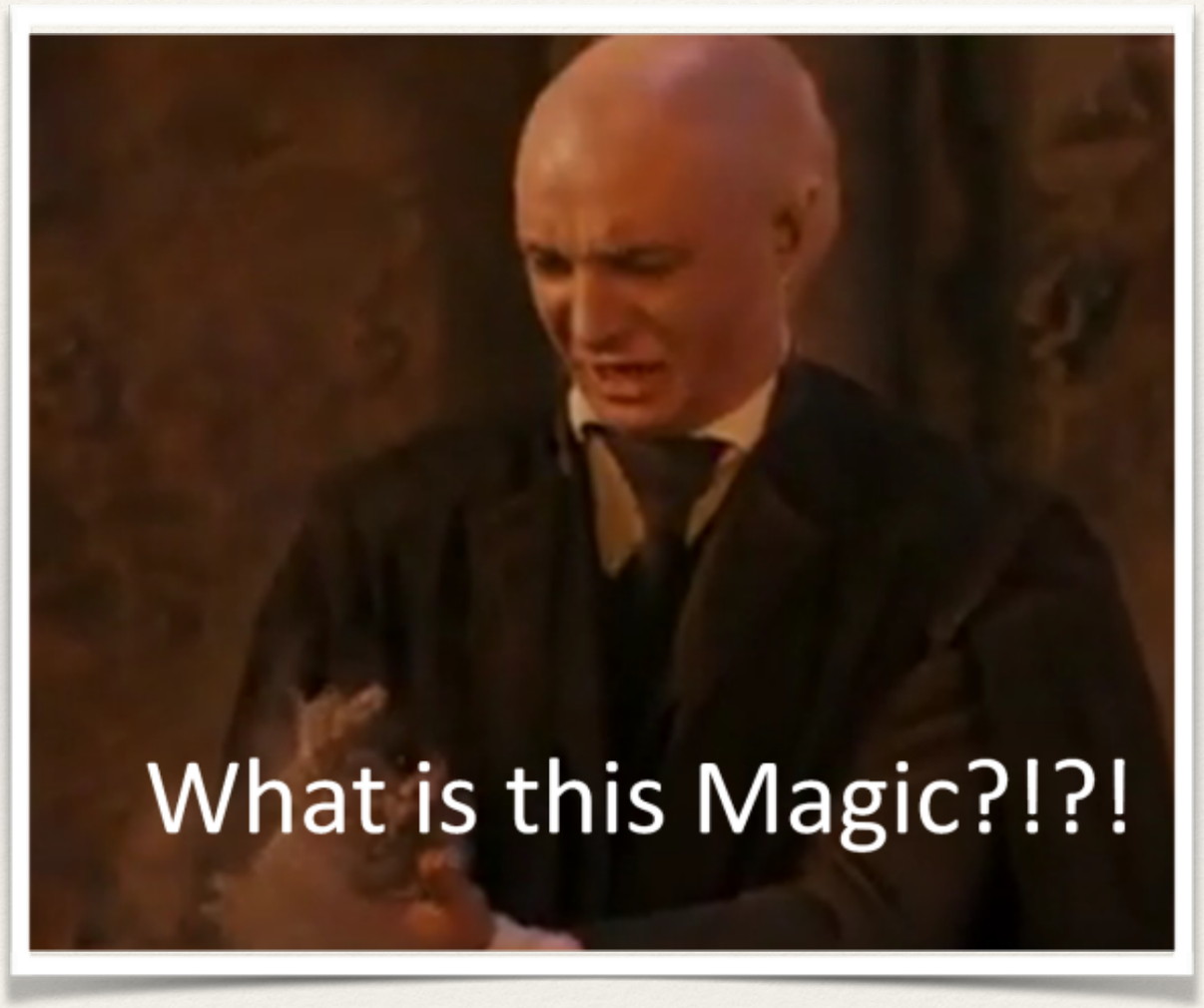
```
#include <stdio.h>
#include <string.h>

int main(void) {
  char line[1000], line2[1000];
  char *p;
  double mag;

  while (fgets(line, sizeof(line), stdin) != NULL) {
    strcpy(line2, line);
    p = strtok(line, "\t");
    p = strtok(NULL, "\t");
    p = strtok(NULL, "\t");
    sscanf(p, "%lf", &mag);
    if (mag > 6)
      printf("%s", line2);
  }
  return 0;
}
```

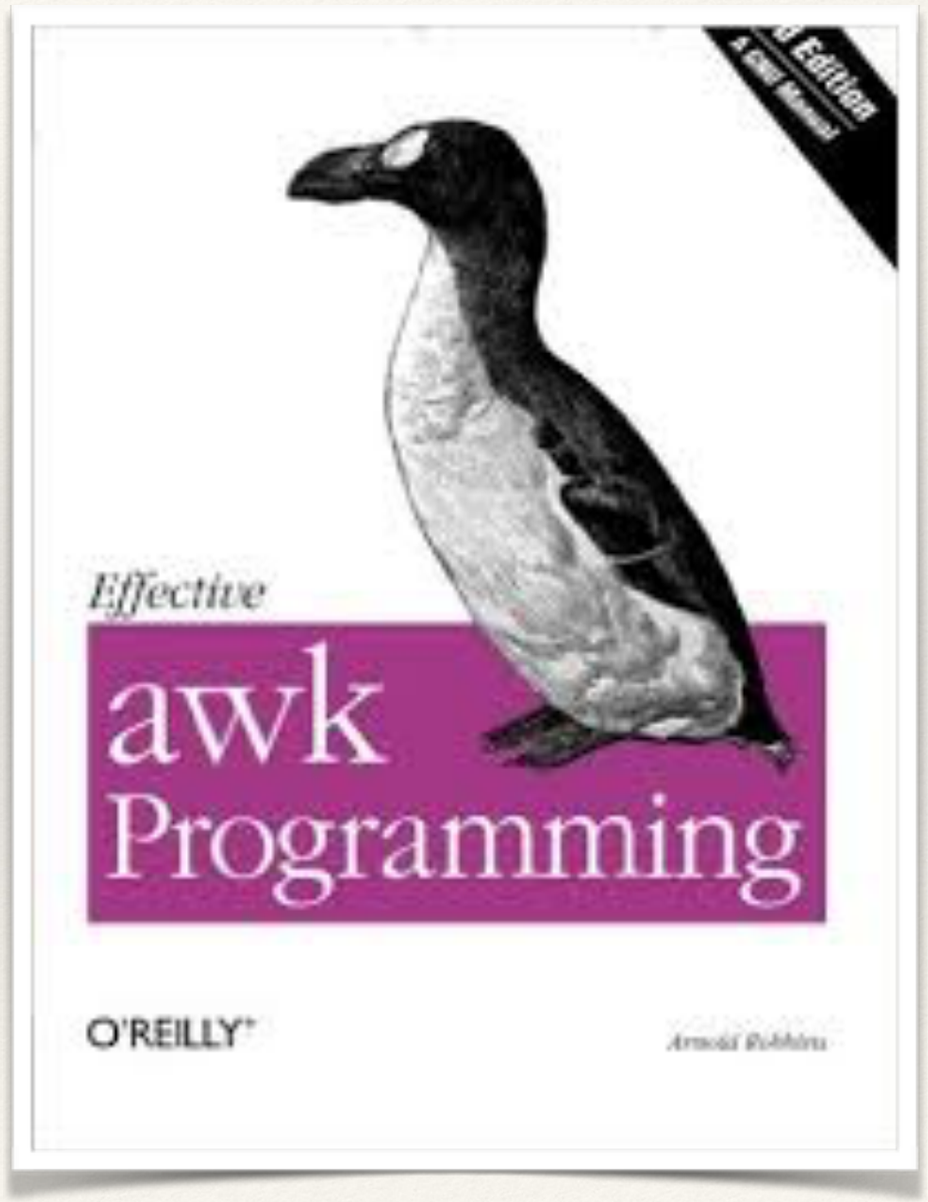
# AWK Version

```
$3 > 6
```



# Not magic. Scripting!

- A language for pattern scanning and processing
- Developed at Bell Labs ca. 1977
  - Al Aho *67, Peter Weinberger, Brian Kernighan *69
- Pattern/Action language
  - pattern is a regexp, string or numeric expression, or combo of those
  - action is executable code



# AWK Features

- Reads input automatically across all command line files
- each line (row, record) is automatically split into fields
- if row matches a pattern, action code is executed on row
  - `BEGIN` (before first line)
  - `END` (after last line)
- control flow statements similar to C
  - `if`, `else`, `while`, `for`, `do`
- Built-in functions for math, string manipulation, I/O, time

```
∀ files
 ∀ line
  ∀ pattern
   if (line ~ pattern)
    if (∃ action) action
    else print
```

# Your First AWK Programs

Usage:

```
awk   ‘literal code' [ file1 file2 ... ]
awk -f file_of_code  [ file1 file2 ... ]
```

Print all lines longer than 80 characters `length() > 80`

Print Reversed First and Second Field `{print $2 , $1 }`

Hello World `BEGIN {print “Hello, world!\n”}`

Print number of lines `END {print NR}`

# AWK Variables

- Variables contain string or numeric values (or both)
  - type is determined by context and use
  - operators coerce type/value according to context
    - initialized to 0 and empty string
- Built-in variables for frequently-used values
- Associative arrays are a first-class type

# AWK Special Variables

- `NR` — Current record # (overall)
- `FNR` — Current record # (current file)
- `NF` — Number of fields in record
- `$0` — Entire record
- `$1 $2 $3 … $NF` — First, second, third, … last field
- `FILENAME` — Current input filename

# More "Patternless" Programs

```
{ print NR, $0 }
```
precede each line by line number

```
{ $1 = NR; print }
```
replace first field by line number

```
{ print $2, $1 }
```
print field 2, then field 1

```
{ temp = $1; $1 = $2;
  $2 = temp; print }
```
flip $1, $2

```
{ $2 = ""; print }
```
zap field 2

```
{ print $NF }
```
print last field

```
{ if(FILENAME != fn) {
  print FNR,FILENAME,$0
  fn = FILENAME
  }
}
```
print first line of each file using filename

```
FNR == 1 {print FNR,FILENAME,$0}
```

# More "Actionless" Programs

| | |
|---|---|
| `NF > 0` | print non-empty lines |
| `NF > 4` | print if more than 4 fields |
| `$NF > 4` | print if last field greater than 4 |
| `NR % 2 == 0` | print even-numbered lines |
| `/regexpr/` | print matching lines (egrep) |
| `$1 ~ /regexpr/` | print lines where first field matches |
| `$1 ~ /regexpr/ && $4 !~ /regexpr/` | print lines where 1st field matches, but the 4th doesn't |

# More AWK Examples

```
        {sum += $1}
END     {print sum sum/NR}
```

Sum first column, print total and average

```
NF > 0 {print $1, $NF}
```

print first, last fields of non-empty lines

```
    { nc += length($0) + 1; nw += NF }
END { print NR, "lines", nw, "words", nc, "characters" }
```

wc

```
length($0) > max { max = length($0); line = $0 }
END      { print max, line }
```

longest line and its length

# Control flow

- `if-else`, `while`, `for`, `do-while`, `break`, `continue`
  - like in C, except no `switch`
- `for (i in array)`
  - like a bash for-in or a Java iterated for loop
- `next` start next iteration of main loop (get next record)
- `exit` leave main loop, go to END block

# Arrays

- common case: array subscripts are integers
- reverse a file:

```
{ x[NR] = $0 }   # put each line into array x
END { for (i = NR; i > 0; i--)
        print x[i] }
```

- make an array:

```
n = split(string, array, separator)
```

  - splits `string` into `array[1]` ... `array[n]`
  - returns number of elements
  - optional `separator` can be any regular expression

# Associative Arrays

- array subscripts can have any value, not just integers
- canonical example: adding up name-value pairs
- input:

```
pizza  200
beer   100
pizza  500
beer   50
```

- output:

```
pizza  700
beer   150
```

- program:

```
        { amount[$1] += $2 }
END { for (name in amount)
        print name, amount[name] | "sort -k2 -nr"
    }
```

# Our Favorite Example

wc.awk

```
BEGIN {
    FS="[^a-zA-Z]+"
}
{
    for (i=1; i<=NF; i++)
        words[tolower($i)]++
}
END {
    for (i in words)
        print i, words[i] | “sort -n -k2”
}
```

```
$awk -f wc.awk mobydick.txt
```

```
the 13967
of 6415
and 6247
a 4583
to 4508
in 4037
that 2911
his 2481
it 2370
i 1940
but 1793
he 1743
as 1709
with 1680
is 1674
was 1600
for 1580
all 1508
this 1368
at 1295
by 1158
not 1136
from 1064
so 1044
him 1044
on 1031
be 1024
whale 1017
```

# Typing/Initialization Gotchas:

```
$awk 'BEGIN { n = 0 + "nancy"; print n}'
nan

$awk 'BEGIN { n = "4chan" + "infinite jest"; print n}'
inf

$awk 'BEGIN { r = 2; s = "fast 2 furious"; n = r s; print n}'
2fast 2 furious
```

```
$awk 'BEGIN { n = "6" + "4"; print n}'
10

$awk 'BEGIN { n = "6" "4"; print n}'
64

$awk 'BEGIN { n = 6 4; print n}'
64

$awk '{ if($1 == foo) print "true foo"; else print "bad bar"}'
bar
bad bar
foo
bad bar
```

# So Isn't A1 Just a Bit of AWK?

REtest.java: split into 3 fields, check if RE matches string, if it matches, see if answer has brackets around the match

```
awk 'BEGIN {FS="@"} ; {sub($1,"[&]",$2);
if($2==$3) print $2; else print “wrong”}’
```

```
$ awk -f REtest.awk
o*@foo@f[oo]
wrong
o*@foo@[]foo
[]foo
o+@foo@f[oo]
f[oo]
o+@baz@baz
baz
o?f?o?o*@foo@[foo]
[foo]
```

# checktests.sh

```
awk '
BEGIN { FS = "@" }
NF > 0 && NF != 3 {
  printf("%d: %d fields: %s\n", NR, NF, $0)
  next
}
$1 ~ /[][]/ || $2 ~ /[][]/ {
  printf("%d: $2 includes brackets: %s\n", NR, $0)
  next
}
$3 !~ /[][]/ && $2 != $3 {
  printf("%d: no brackets but $2 != $3: %s\n", NR, $0)
  next
}
' "$@"
```

# Eyes or Mouth?

- [ ] [ ]

"] ends the bracket expression **if it's not the first item**." But here it is the first item, so it's a character class of 2.

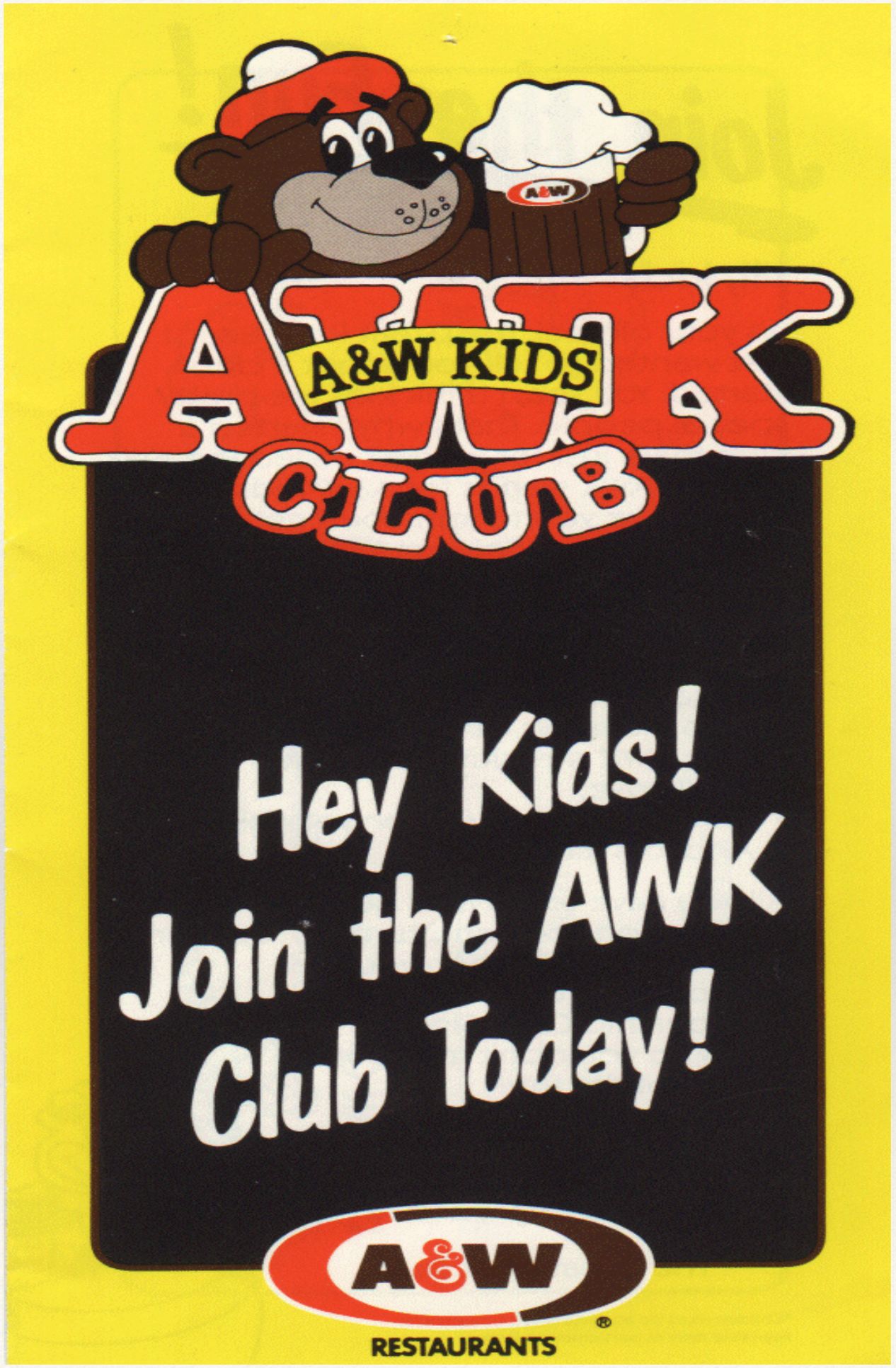
- [ [ ] ]

This matches [ (a single-character character class), then ] (not special if not closing a character class).

http://www.gnu.org/software/grep/manual/html_node/Character-Classes-and-Bracket-Expressions.html

# checktests.sh

```
awk '
BEGIN { FS = "@" }
NF > 0 && NF != 3 {
  printf("%d: %d fields: %s\n", NR, NF, $0)
  next
}
$1 ~ /[][]/ || $2 ~ /[][]/ {
  printf("%d: $2 includes brackets: %s\n", NR, $0)
  next
}
$3 !~ /[][]/ && $2 != $3 {
  printf("%d: no brackets but $2 != $3: %s\n", NR, $0)
  next
}
' "$@"
```


A&W KIDS
AWK
CLUB
Hey Kids!
Join the AWK
Club Today!
A&W
RESTAURANTS

# Well, maybe not everyone …

"I want to store all input lines of a big text file into one individual variable. For example in this sample of my text file:

GATCTGAATCAGGTACCTAGCAATCCGTATCAAC
GTTTTCACGCTATGCTTAAG

AGCGGTGTCGAAGTAGGCATAGCTAGCAACTTA
TCTAATCGAGCGAGCGAGAGA

TTTGTCTATCGATCTAGTCATCGATCCGGTATCAC
GGAGATACAGTCTACCGTT

I want to concatenate all the line into a unique variable. but my file contains about 500000 lines and when I use

var = var $0

the process take too much time to make the concatenation and store the result in var. Is there a specific algorithm in awk to store a big input in var?"