Lecture 2

Regular Expression Parsing

Awk

Shell Quoting

- Shell Globing: file* and file?
- ls file* (the backslash key escapes wildcards)

Shell Special Characters

~	Home directory
`	backtick (command substitution)
#	Comment
$	Variable expression
&	Background job
*	String wildcard
( )	subshell
\	quote next character
[ ]	character set wildcard
{ }	don't start sub shell
;	command separator echo "hi; echo ho"
'	single (strong) quote
< > >> <<	redirection
?	single character wildcard

Shell Quoting

- ‘ ‘ Single quotes disable all shell interpretation
- “ “ Double quotes disable all shell interpretation except:
  - \ tells shell to specifically ignore the next char
  - $
  - `
    - echo “$HOME”
    - echo ‘$HOME’
- ` ` backticks (command substitution)
  - ls –la `which xclock`
  - ls –l `which emacs`
  - echo “today is `date`”

Regular Expressions are the ‘re’ in grep

- grep (g/*re*/p), fgrep, egrep
- ed, ex, vi, emacs
- sed, awk
- perl
- lex, flex
- perl
- python
- tcl

Chicken Parts

- literals: d, o, g
- Character Class: [abc0-9^&*$]
  - negated CC's: [^abc]
- Anchors: ^, $
- . metacharacter = "any single character [except \n]"
- Alternation: (this|or|that)
- Word boundaries: \< ... \>
- Quantifiers: ?, *, + (work on *preceding* characters)
- Backreferencing: ([abc])\1
- Escaping: \.
- Repeating quantifier: [0-9]{3}, [0-9]{5,6}

Regex fundamentals

- Regular Expressions match parts of lines, and perhaps whole lines
- In this context, a character is defined as ‘a byte represented by a single character encoding, like ASCII, EBCDIC, JIS’
- THINK LITERALLY, forget English concepts of “words” and “sentences”
  - grep ‘put’ WILL match “Lilliputian” every time
  - Read ‘put’ as “the character ‘p’, followed immediately by a ‘u’ character, followed immediately by a ‘t’ character.
  - Don’t say ‘u’ matches ‘put’, but rather the regex matches the ‘u’ part of ‘p-u-t’
  - egrep ‘i[^t]’ reads: “the character ‘i’, followed by a single character that is not the character ‘t’
  - as far as egrep is concerned, the ‘u’ is ‘p-u-t’ is the *number* 117, or 0x75.
  - The *entire* regex is attempted at each position in the string before the ‘pointer’ advances

Regex Engines

- Deterministic Finite Automation (DFA)
  - fast
  - blunt
  - parallel analysis
  - “text” directed, generally no backtracking logic
  - Examples: ed, egrep, awk, lex, flex
- Nondeterministic Finite Automation (NFA)
  - slower, possibly much slower
  - intelligent
  - backtracking logic built in
  - “regex” directed
  - Examples: emacs, vi, perl, sed, grep, python

Friedl's Two Rules

- The earliest match wins:
  - echo "The dragging belly indicates that your cat is much too fat" | egrep 'cat'
  - echo "The dragging belly indicates that your dog is much too fat" | egrep 'cat'
  - 'fat|cat|belly|your' matching process – which matches first? 'fat' or 'belly'?
- Quantifiers (?, *, +, {*min*,*max*}) are "greedy"
  - echo "1999" | showmatch '[0-9]+' (what matches: "1" or "1999"
  - echo "billions and billions and billions and billions" | showmatch '(b.*d)'
  - '.*' will match *any* string *every* time.

echo "Live Danish Dancers" | egrep "Da(nsk | naides | ncers)"

- DFA, being text-directed, examines three regex options in parallel, dropping options as they are excluded. It tries first the n in Da**n**sk, the the n in Da**n**aides, then the n in Da**n**cers. Then it tries the s in Dansk, fails, tries the a in Danaides, fails, and then the c in Dancers, etc.
- NFA, being regex-directed, will first try Da**nsk** until it fails (on the ***s***), then, will try Da**n*a*ides** (on the ***a***), until it fails, then will try Da**ncers** until it succeeds. NFA picks (internal logic) an option, marks its "state", attempts to run with it, and on failure, returns to nearest saved "state", picks another option (internal logic), and attempts to run with it, etc.

Points to Remember

- Does ‘q[^u]’ match the string ‘Iraq’?
- Metacharacters lose their meaning inside character classes: [*?+]
- What does “\<b.*[tk]\>” match?
  - boat
  - book
  - baby back ribs
- grep does NOT substitute, it matches and prints the entire line or nothing:
  - echo 123 | grep "^[1]" prints 123 (entire input line) or nothing, it will NOT print just ‘1’

So you understand this?

- Will the following match or not? If it matches, where does it match?
  - echo "a string with a 0 in it" | grep "[0-9]*"
- Will the following match or not? If it matches, where does it match?
  - echo "a string with absolutely no numbers in it at all" | grep "[0-9]*"

Backtracking Fun with NFA

- Backtracking allows your to refer later to a previous match:
  - echo “hi ho hi ho” |egrep ‘(h?).*\1’
  - echo “hi ho hi ho” |egrep ‘(h. h.)\1’
- Will the following match? If so, what?
  - echo “hi ho hi ho” |egrep ‘(h? h?)’
- Will the following match? If so, what?
  - echo “hi ho hi ho” |egrep ‘(h? h?) \1’

Tie it all together

- Which of the following will match?
  - echo "billions and billions and billions and billions" | egrep '(b.*d) \1'
  - echo "billions and billions and billions and billions" | egrep '(b.*d b.*d) \1'

Common regexes (egrep syntax)

Postal Abbreviation for state	**[A-Z]{2}**
City, ST	**^.*,[A-Z][A-Z]**
City, ST, Zip	**^.*,[A-Z][A-Z] [0-9]{5}(-[0-9]{4})?**
Month, Day, Year	**[A-Z][a-z]{3,9} [0-9]{1,2}, [0-9]{4}**
Social Security Number	**[0-9]{3}-[0-9]{2}-[0-9]{4}**
???	**(\(...\))?.*[0-9]{3}-[0-9]{4}**
???	**\$[0-9]*\.[0-9]{2}**

Introduction to AWK

- Written by Alfred **A**ho, Peter **W**einberger, Brian **K**ernighan in 1977.
- awk is primarily a filter that provides a rich language in which to display and minipulate incoming data
- Whereas grep & Co. allows you to search through a text file and look for something, awk lets you search through a text file and *actually do something* once you've found what you're looking for

awk and C

- awk shares many syntactic similarities with the C programming language (Kernighan was heavily involved in both)
- Whereas a C program requires the program author to open and close files, and move from one line to the next in the input, find and isolate the tokens within a given line, keep track of the total number of lines and the current number of tokens, awk does all this for you automatically
- Therefore, we say that awk is “input-driven”, it must work on lines of input

awk Processing

- awk processes incoming text according to lines which are called *records* and elements within those lines called *fields*.
- awk processes commands called pattern-actions, or rules. If a pattern matches, the associated action is performed
- Actions are enclosed in braces {}
- Patterns, if present, are stated before actions outside of braces
- In an awk rule, either the pattern or the action may be missing, but not both:
  - if the pattern is missing, the action is performed on *every* line of the input
  - if the action is missing, the default action is to print the line out to stdout

awk program structure

- Multiple BEGIN sections (optional)
- Multiple END sections (optional)
- Multiple recursive blocks which will operate on *each* record (line) of the input file

awk Program Flow

- Process optional BEGIN block
- Open the file (either specified during invocation or from STDIN)
- Read each line (record) of the input file and parse records into fields referenced by $*n*
  - $0 denotes the entire record
  - each field is demarked by $1, $2, $3, $4, etc.
- Execute each block defined in the awk program on each record (input line)
- Execute optional END block
- Close the file

awk Patterns

- Patterns may be composed of:
  - /regular expressions/
    - awk '/[2-3]/' five.lines
    - awk '$2 ~ /[2-3]/' five.lines
  - A single expression
    - awk ‘$2 > 3’ five.lines
  - A pair of patterns, separated by a comma indicating a range of records:
    - awk ‘$2 == “2”, $2 == “4”’ five.lines

awk Built-in Variables

- FS: Input field separator (default ‘ ’)
- OFS: Output field separator (default ‘ ’)
- RS: Record Separator (default ‘\n’)
- ARGC: C-style arg count
- ARGV: C-style arg vector (offset 0)
- NF: number of fields in current record
- NR: number of records processed so far
- NOTE: Do NOT put a $ in front of these variables (i.e., don’t say “$NR” but just “NR”)

Example Blocks
What do the following do?

- awk ‘$4 > 0 {print $1,”from”,$6}’ some.data
- awk ‘{print}’ some.data
- awk ‘{print}’
- awk ‘NF > 0’ some.data
- awk '/n/; /e/' five.lines
- awk ‘/text/ {print}’
- awk ‘BEGIN {print “Hello World”}’
- awk '{ $1 = "THE LINE"; print}' five.lines
- ypcat passwd | awk -F: ‘$1 ~ /mark/ { print $1,"is a bozo"}‘
- awk ‘BEGIN {print $3-$4 }’ some.data
- awk '{print "Balance for",$1,"from",$6,"is:",$3-$4}' some.data

A Sample Program

```
ypcat passwd |
awk 'BEGIN{FS=":"}     #could use –F”:” on comand line
{print "Login id:", $1;
print "userid:", $3;
print "group id:", $4;
print "Full Name:", $5;
print "default shell:", $7;
print " " ;}'
```

String-Matching Patterns

- */regex/*
  - matches when the current record *contains* a *substring* matched by *regex*
  - /ksh/ { ... } # process lines that contain the letters ‘ksh’
- *expression* ~ */regex/*
  - matches if the string value of *expression* (can be a field like $3) *contains* a *substring* matched by *regex*
  - $7 ~ /ksh/ { ... } # process records whose 7th field contains the letters ‘ksh’
- *expression* !~ */regex/*
  - matches if the string value of *expression* (can be a field like $3) *does NOT contain* a *substring* matched by *regex*
  - $3 !~ /[4-6]/ { ... } # process records whose 3rd field does not contain a 4, 5, or a 6

awk Functions

- ✓ math functions: cos, int, log, sin, sqrt
- ✓ length(s) returns length of string
- ✓ index(s,t) returns pos of substr s in string t
- ✓ substr(s,p,m) returns substring of string s beginning at p, going length of m
- ✓ split(string, arrayname[, fieldsep])
  split splits *string* into tokens separated by the optional *fieldsep* and stores the tokens in the array *arrayname*
- ✓ gawk C-like extensions:
  - ✓ toupper()
  - ✓ tolower()
  - ✓ sprintf("fmt",expr)
- ✓ Example (what is my regex matching, revisited):
  - ✓ echo '111111' | awk '{sub (/1/, "X"); print }'

awk Arrays

- awk provides functionality for one-dimensional arrays (and by extension, multidimensional arrays)
- Arrays are associative in awk, meaning that a *value* is *associated* with an *index* (as opposed to a memory-based non-associated array scheme in C for example)
- By default, array indices begin at 0 as in C

awk Arrays continued

- This means that indexes (which are always converted to strings) may either be integral or textual (i.e., a string)
  - array[1] may return “un”
  - array[three] may return “trois”

```
awk ‘BEGIN{
for (i in ARGV)
print “Item”,i,“is:”,ARGV[i]
}’ one two three
```

Array Syntax

- To reference an array element:
  - array[*index*]
- To discover if an index exists in an array:
  - if (three in array)
    - print “three in French is”,array[three]
- To walk through an array:
  - for(x in array) print array[x]
- To delete an individual element at an index:
  - delete array[index]

Creating an Array using split()

split1.sh:

```
echo 'un deux trois quatre' |awk
    '{split($0,array)}END{
for (x in array) print "index:",x":",array[x];}‘
```

split2.sh:

```
echo 'un deux trois quatre' |
awk '{split($0,array)}
END{if ( 3 in array )
print "three in French is",array[3]}'
```

Real World Example

- from Aho, Kernighan, Weinberger, *The AWK Programming Lanugage*, chap. 4:
- cat countries
- cat prep.3
- cat form.3
- awk -f prep.3 countries countries | awk -f form.3