CSE2031 Software Tools - UNIX introduction

Summer 2010

Przemyslaw Pawluk

Department of Computer Science and Engineering
York University
Toronto

June 29, 2010

Table of contents

1 The AWK Programming Language

Plan

1 The AWK Programming Language

The AWK Programming Language

- AWK (pron. auk) can be used to manipulate text and numerical values.
- Usually, simple short programs (could be just one line).
- The program could be in a file, or could be entered with the command
- The name AWK is derived from the family names of its authors Alfred Aho, Peter Weinberger, and Brian Kernighan
- Consider the following example

AWK structure

- The structure of an AWK program
- Each AWK program is a sequence of one or more pattern-action statement
- Searches the input file looking for any lines that are matched by any of the patterns and the action is applied

Shell built-in variables

- $# The number of arguments
- $* All arguments to shell
- $- Options supplied to shell
- $? return value of the last command executed
- $$ process ID of the shell
- $ process ID of the last command started with &

Shell pattern Matching Rules

- * Any string, including the null string
- ? Any single character
- [ccc] Any of the characters in ccc [a-d0-3] is equivalent to [abcd0123]
- "..." Matches exactly, the quotes are to protect special characters
- \c c literally; if * it matches the "*" char
- a|b In case expression only, matches a or b

Example

You have a file called in.txt that contains list of cars with distances and rates. We want to print all cars with distance grater than 0.

Structure of the program

- Sequence of one or more pattern-action statements
- Search the input file looking for any lines that are matched by any of the patterns
- if found action is applied
- if there is no pattern action is executed on every line
- expression separated by commas in print are separated by single space when printed
- You can use printf function as in C

```
1 pattern {action}
2 pattern {action}
```

How to run a program?

```
1 awk 'program' file1 file2
```

Program in file

```
1 awk -f progfile file1 file2
```

Combination of Patterns

- patterns can be combined with `&&` (AND), `||`(OR) or `!`(NOT)
- `/name/` matches with name in the line

```
1 NF != 3 {print $0, "Number of fields is not 3"}
2 $2 <8.75 || $2>100 {print %0, "Overflow"}
3 $2 >20 {print $0, "rate more than $20 dollars"}
4 !$3 >0 {print $0, "Negative rate"}
```

Begin and End

- The special pattern BEGIN matches before the first line of the first input file
- The special pattern END matches after the last line of the last input file

Example

```
1 BEGIN{print "NAME RATE HOURS"; print}
2 {print}
3 {total = total + $2 * $3}
4 END{print "The total is ", total}
```

String manipulation

You can use functions such as length(str) to get length of the string str

```
1 { names = names $1 " "}
2 END {print names}
3 {last = $0
4 END {print last}
5 { nc = nc +length($0) +1
6 nw = nw + NF}
7 END {print NR, "lines ", nw,
8 "words , nc , "characters }
```

Control flow

- You can use `if-else` and `while` statements as in C

```
1  $2 > 6 {n=n+1; rate = rate + $2 * $3}
2  END { if(n > 0)
3  print n, cars, total rate is, rate,
4   average rate is  , rate/n
5  else
6  print  No cars are making more than $6
7  }
8  { i=1
9  while (i <= $3) {
10 printf( \t%.2f\n , $1 *(1 + $2) ^i)
11 i=i+1
12 }
13 }
```

Arrays

AWK allows for arrays

- The index of the arrays need not be integer.
- No need for declaration
- Initialized to 0 or ""
- For example, you can say `Ar1[$1] = $2`

```
1 # print the input in a reverse order
2 {line[NR] = $0}
3 END { i=NR
4 while(i > 0) {
5 print line[i]
6 i=i-1
7 }
8 }
```

Arrays

```
1  {ar[$1]=$2}
2  END {
3  for (x in ar) print x, ar[x]
4  }
```

The order of stepping in the array is implementation dependent!

Patterns

- Rule = Pattern+Action
- BEGIN statement is executed befor any input is read
- END statement is executed after all inputs are read
- Pattern statement is executed when Pattern is true (satisfied)
- /regular expression/ statement is executed when line contains string that matches expression
- Compound pattern statement is executed at any line that satisfies pattern
- pattern1, pattern2 statement – is a range pattern that matches each line from line matched by pattern1 to the next line mached by pattern2

Matching Strings

- `/regexpr/` matches when the current input line contains a substring matched by regexpr
- Expression `~/regexpr/` Matches if the string value of the expression contains a substring matched by regespr.
- Expression `!~/regexpr/` matches if the string value of expression does not contain a substring matched by regexpr

```
1 /Asia/ # short hand for $0 ~ /Asia/
2 $4 ~ /Asia/
3 $3 !~ /Asia/
```

Regular expressions

- A non metacharacter that matches itself A, b, D,
- Escape sequence that matches a special symbol `\t`, `\*`
- `^` beginning of a string
- `$` End of a string
- `.` Any single character
- `[ABC]` matches any of A,B,C
- `[A-Za-z]` matches any character
- `[^0-9]` any character except a digit

Regular expressions - combinations

- Alternation: `A|B` matches A or B
- Concatenation: `AB` matches A followed by B
- Closure: `A*` matches zero or more A
- Positive closure `A+` matches 1 or more A
- Zero or one: `A?` matches the null string or A
- Parenthesis: `(r)` matches the same string as r

Regular expressions - string matching

- `^C` matches C at the beginning of a string
- `C$` matches C at the end of a string
- `^C$` matches the string consists of the single character C
- `^.$` any string with exactly one character
- `...` matches any three consecutive characters
- `\.$` matches a string that ends with period
- `^[ABC]` A, B, or C at the beginning of a string
- `^[^ABC]` any character at the beginning of a string except A,B, or C
- `[^ABC]` any character other than A,B, or C
- `^[^a-z]$` any single character string except a lower case character

Built-in variables

- ARGC Number of command lines arguments
- ARGV arra of command line arguments
- FILENAME Name of current input file
- FNR Record number in current file
- FS Input field separator
- NF Number of field in the current record
- NR Number of records red so far
- OFS Output field separator
- ORS Output record separatot
- RLENGTH Length of string matched by matching function
- RS Input record separator

Reading from a File

- getline function can be used to read input from a file, splits the record and sets NF, NR, and FNR
- It returns 1 if there was a record, 0 for end of file, and -1 for error
- `getline < "File"`
- `getline x <"File"` gets the next line and stores it in x (no splitting) NF, NR, and FNR not modified