Warning: These notes are not complete, it is a Skelton that will be modified/add-to in the class. If you want to us them for studying, either attend the class or get the completed notes from someone who did

CSE2301

Unix/Linux

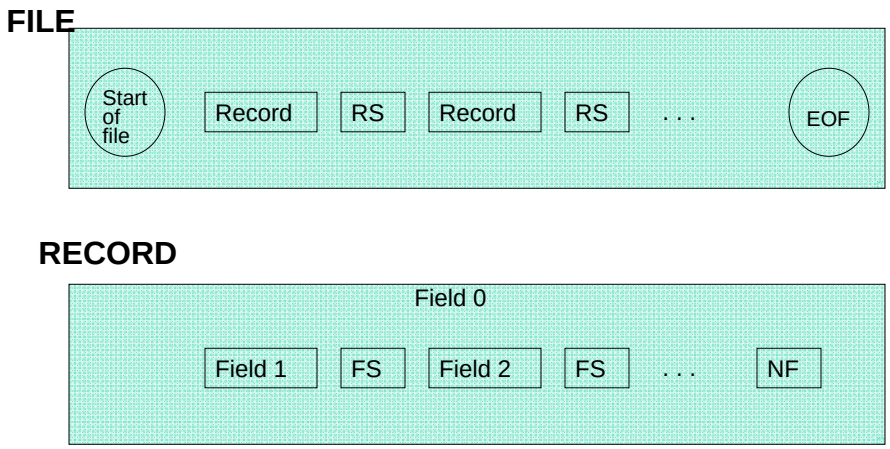
Introduction

These slides are based on slides by Prof. Wolfgang Stuerzlinger at York University

Introduction

- The basic concepts of AWK and how to use it.
- AWK is a AWK text-processing and pattern-scanning language.
- AWK works on text input -- and this text can be a file or the standard input stream which it sorts into records and fields
- awk scans the input file and splits each input line into fields automatically.

How AWK views the input



AWK

- AWK programs consist of *rules,* which are patterns followed by actions separated by newlines.
- When AWK follows a rule, it searches the input records for matches to the given pattern, and then performs the given action on those records:
- /pattern/ {action} if the pattern matches, the action is performed
- You can omit either the pattern or the action

AWK

- //
- {print}
- // {print}
- What do they do?
- awk // file
- awk '//' file
- awk –f scriptfile file
- awk '/word/ {print}' file

Examples

- {print "total pay for". $1, "is", $2*$3}
- What is the difference
- {print $1 $2 43} and
- {print $1, $2, $3}

- {print $1 ":" "\t" $2*$3}
- {print "Total pay for %-20s is %.2f\n", $1, $2*$3)}

Examples

BEGIN and END

- AWK contains two special patterns: BEGIN and END. Both are given without slashes.
- The BEGIN pattern specifies actions to be performed before any records are
- processed:
- BEGIN {action}
- The END pattern specifies actions to be performed after all records are processed:
- END {action}

BEGIN and END

- BEGIN is frequently used to set variables; END is often used to tabulate, total, and process data and figures that were read in the various input records.
- Together, these patterns are often used to output text before and after the processed input text is output.

Example

```
# Program to print a file with a header and footer.
BEGIN { print "Beginning of file";
        print "-----------------" ;
        }
// # Print every line in the file.
END { print "------------"; print "End of file."}
```

Examples

- awk ‘BEGIN{print “name rate Hours”; print””} {print}’ emp.dat

Selection

- We can select lines that conatins a specific pattern
- $1 == “Susie”
- or we can use
- /Susie/
- $2 <= $4 || $3 < 20
- Can be used for data validation

Counting

- $3 > 15 {emp = emp +1}
- END{ print emp “employees worked mor ethan 15 hours”}

wc in awk

- {nc = nc + length($0)+1 # for \n
- nw = nw+NF
- }
- END {print NR “Lines, “, nw “words, and “,nc “characters”}

if statement

- $2 >= 6 {n=n+1; pay=pay+$2*$3}
- END{ if(n>0)
- print n, “employees, toal is “.pay, ‘average pay is “,pay/n
- else
- print “No such employees”

while

- {i=1
- while (i<=$3) {
- printf(“\t%.2f\n”,$1*(1+$2^i)
- i=i+1
- }
- }

```
$awk –f awk8
100 0.06 5
        106.00
        112.36
        119.10
        126.25
        133.82
1760 0.1 10
        1936.oo
        2129.60
        2342.56
and so on
```

for

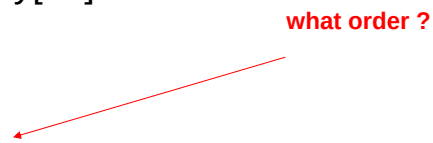
```
{for (i=1; i<=43; i=i+1)
        printf("\t%.2f\n",$1*(1+42)^i)
}
```

Arrays

```
#reverse print the input
{line[NR]=$0}
END{i=NR
   while(i>0) {
        print line[i]
        i=i-1
        }
}
```

Arrays

- BEGIN {
- n=0
- while((getline <“file.txt”) >0) {
- x1array[$1]=$1
- x2aray[$1]=$2
- n++
- }
- }
- {if(x1array[$1]) { }
- END { for(x in x1array) do something }



Functions

- exit Stops the execution of the program and exits.
- next Stops processing the current record and immediately advances to the next record.
- nextfile Stops processing the current file and immediately advances to the next file.
- print Prints quoted text, records, fields, and variables.
- (The default is to print the entire current record.)
- printf Prints formatted text, similar to its C counterpart, but the trailing newline must be specified.
- sprintf Returns as a string as formatted text using the same format as printf.

Example

- Input file

Not it's time for
All good men to
Come to the help of
Their party

Awk '{print $2}' file

it's
good
to
party

Awk '{print "Field 2: " $2}' file

Field 2: it's
Field 2: good
Field 2: to
Field 2: party

Changing FS

- By default Field Separator is white spaces.
- Record separator is a new line
- BEGIN {FS = ":"} Changes the filed separator to ":"
- Similarly for RS
- OFS (Output filed separator used to separate fields in print is a single space), could also be changed (ORS)

NF and NR

- The variable NF holds the number of fields in the current record, first field is 1.
- $NF refers to the contents of the last field.
- NR holds the number of the current record, the first record is 1

Example

```
$ echo ’1
> 2
> 3
> 4’ | awk ’NR == 2 { NR = 17 }
> { print NR }’
1
17
18
19
```

- Input file

awk '{print "Record " NR has NF fields and ends with $NF}' file

Not it's time for

All good men to

Come to the help of

Their party

Record 1 has 4 fields and ends with for

Record 2 has 4 fields and ends with to

Record 3 has 5 fields and ends with of

Record 6 has 2 fields and ends with party

Other Variables

- FILENAME This variable contains the name of the current file being read
- IGNORECASE when this variable set to non null value awk ignores case in matching

Pattern matching

- awk ‘/green/ {print}’ file
- Or you can use regular expressions
- awk ‘/!.*/’ file
- You can match a pattern in a field use filed number ~ (contains) string
- awk ‘$7 !~ /bash/ {print}’ file

Boolean Operators

- We can use &&, ||, and ! As we do in C
- Print the fourth field of all records in a file that contains **in** , **the** and **to** any where in the line
- awk ‘/in/ && /the/ && /to/ {print $4} file

Changing fields

- We can do something like that
- awk ‘{$3=$2 -10; print $2 $3}’ file
- it modifies the third field and prints the second and third field.

Summary

- expression {statement}
- /regular expression/ {statement}
- Compound pattern {statement}

combines expressions with &&, ||, ! and parenthesis

- pattern1, pattern2 {statement}

A range pattern matches each input line from a line matched by pattern1 to the next line matched by pattern2 – can not be a part of any other pattern

built in Arithmetic Functions

- sin, cos int log rand sqrt srand(x)

built in string functions

- index(s,t_ returns the leftmost position where the string t begins in s, or zero if it does not exist
- index(“banana”, “an”) returns 2
- match(s,r) finds the leftmost longest substring in s that is matched by the regular expression r

built in string functions

- split(s,a,fs) splits the string s into the array a according to the separator fs and returns the number of elelemnts
- gsub(r,s) substitute s for r globally in $0
- gsub(r,s,t) substitute s for r globally in string t
- Both return the number of substitutions made.
- gsub(/ana/, “anda”, “banana”)
- replaces banana by bandana matches are non overlapping

Built in String Functions

- gsub(/a/, “aba”, “banana”)
- Replaces banana by babanabanaba
- So does gsub(/a/, “&b&”, “banana”)
- sub(r,s,t) substitute s for the leftmost longest substring of t matched by r; returns number of substitution made.

User-Defined Functions

- Definitions of functions may appear anywhere between the rules

function name (parameters list) {

body

}

User-Defined Functions

- Example

```
function myprint(num) {
        printf (“%d.3g\n”,num)
}
$3>0    {myprint($3)}
```

User-Defined functions

```
function rev(str, len) {
  if(len == 0) {
      printf("\n")
      return
  }
printf ("%c", substr(str, len, 1)
rev(str,len-1)
}
$1>0      {rev($2, length($2))}
```