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

Shell Programming
Introduction

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

Introduction

- In this part, we introduce
  - The AWK Programming Language

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

The cal program

```
tigger 165 % cal
    February 2009
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
```

Cal 2 gives the calendar for year 2, not month 2

```
#!/cs/local/bin/sh
case $# in
0)      set `date`; m=$2; y=$6;;
1)      m=$1; set `date`; y=$6;;
2)      m=$1; y=$2;;
esac

case $m in
jan*|Jan*)      m=1;;
feb*|Feb*)      m=2;;
mar*|Mar*)      m=3;;
apr*|Apr*)      m=4;;
may*|May*)      m=5;;
jun*|Jun*)      m=6;;
jul*|Jul*)      m=7;;
aug*|Aug*)      m=8;;
sep*|Sep*)      m=9;;
oct*|Oct*)      m=10;;
nov*|Nov*)      m=11;;
dec*|Dec*)      m=12;;
[1-9]|10|11|12) ;;
*)              y=$m;m="";;
esac
/usr/bin/cal $m $y
```

What if *) instead of 2)

The AWK Programming Language

- AWK can be used to manipulate text and numerical values.
- Usually, simple short programs (cluld be just one line).
- The program could be in a file, or could be entered with the command
- Consider the following example

Example

- You have a file →
- Print the name and pay rate for every one who worked more than 0 hours.
- awk '$3 >0 {print $1, $2*$3}' file

```
Person1   4.0     0
Person2   3.75    2
Person 3  2.17    0
Person4   2.25    4
```

```
Person2 7.5
Person4 9
```

```
awk '$3 == 0 {print $1}' file
```

AWK

- 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

```
pattern  {action}
pattern  {action}
```

How to run: `awk 'program' file1 file2`

AWK

- The program could be in a file progfile

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

- tigger 222 % awk '$3 >0 [print $1, $3*$2}' emp
- awk: cmd. line:1: $3 >0 [print $1, $3*$2}
- awk: cmd. line:1: ^ syntax error
- awk: cmd. line:1: $3 >0 [print $1, $3*$2}
- awk: cmd. line:1: ^ syntax error

AWK

- If there is no pattern, the action is executed on every line

```
{print}
{print $0}
```

- Expressions separated by commas in print, are separated by a single blank when printed

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

AWK

- What about
- `{print NR, $0}`
- `{print "Total is", $2*$3}`
- Can use printf (as in C)
- Can pipeline the output

```
awk '{printf("%6.2f
  %s\n,$2*$3,$0)}' file |sort
```

Combination of Patterns

- Patterns could be combined using logical AND, OR, or NOT
- `/name/ #` matches with name in the line
- `$2 >= 4 || $3 >= 20`
- `!($2 <4 && $3 <20)` same as above

```
NF != 3  {print $0, "Number of fields is not 3"}
$2 <8.75 {print %0, "rate below min. wage"}
$2 >20   {print $0, "rate more than $20 dollars"}
$3 <0    {print $0, "Negative pay rate"}
```

Begin and END

- The special pattern BEGIN matches before the first line of the first input file is read.
- The special pattern END matches after the last line of the last input file has been processed.

Introduction

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

awk -f emp.awk emp

Person1	4.0	0
Person2	3.75	2
Person3	2.17	0
Person4	2.25	4

```
NAME    RATE   HOURS

Person1  4.0    0
Person2  3.75   2
Person3  2.17   0
Person4  2.25   4
The total is   16.5
```

Counting and Average

```
$3 > 15 {emp = emp +1}
END {print emp, "Employees worked more than 15 hours"}

{pay = pay + $2 * $3}
END { print NR, "employees"
          print "total pay is ", pay
          print "average pay is", pay/NR
         }

$2 > maxrate {maxrate = $2; maxemp = $1}
END {print "highest hourly rate:", maxrate, "for", maxemp}
```

String manipulation

```
 { names = names $1 " "}
END {print names}

{last = $0
END {print last}

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

NR retains its value in END, but not $0

\n char

Control Flow Statements

```
$2 > 6 {n=n+1; pay = pay + $2 * $3}
END {  if(n > 0)
          print n, "employees, total pay is", pay,
"average pay is  ", pay/n
       else
          print "No employees are making more than $6"
    }

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

```
$ awk -f emp2.awk
1000 0.07 7
     1070.00
     1144.90
     1225.04
     1310.80
     1402.55
     1500.73
     1605.78
```

Control Flow Statements

```
# Another program to calculate the interest
{   for(i=1; i<=$3; i=i+1)¶
       printf("\t%.2f\n", $1*(1+$2)^i)¶
}
```

Arrays

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

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

Arrays

- `{ar[$1]=$2}`
- `END {`
- `    for (x in ar) print x, ar[x]`
- `     }`
- The order of stepping in the array is implementation dependent.

Examples

```
/Beth/ {nlines = nlines +1}
END           {print NLINES}

 NF > 4

{ for(i=NF; i> 0; i=i-1) printf("%s ",$i)
Printf("\n");
}

 Length($0) > 80
```

Patterns

- Again, the rule in AWK programs is
- Pattern Action
- Here are the rules for patterns
- BEGIN {statement} statement is executed before any input is read
- END {statement} statement is executed after all inputs are read.
- Expression {statement} the statement is executed at any line where Expression is true

Patterns

- /regular expression/ {statement} The statement is executed at each input line that contains a string matched by the regular expression.
- Compound pattern {statement} combing expressions with &&, ||, ! And the statement is executed at each line the pattern is true

Patterns

- Pattern1 , patter2 {statement} A range pattern matches each input line from a line matched by pattern1 to the next line matched by pattern2

String Matching Patterns

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

String Matching Patterns

- /Asia/ # short hand for $0 ~ /Asia/
- $4 ~ /Asia/
- $3 !~ /Asia/

Regular Expressions Meta Characters

- 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 Expression

- These operators combine regular expressions.
- 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

- ^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

Regular Expressions

- `^[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

Regular Expressions

- `/^[0-9]+$/` any input line that consists of digits only
- `/^[0-9][0-9][0-9]$/` exactly three digits
- `/^(\+|-)?[0-9]+\.?[0-9]*$/` A decimal number with optional sign and optional fraction
- `/^[+-]?[0-9]+[.]?[0-9]*$/` same as above

Regular Expressions

- /^[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+) ([eE][+-]?[0-9]+)?$/ a floating point number with optional sign and optional exponent
- `/^[A-Za-z][A-Za-z0-9_]$/` a letter followed by any letter of digit variable name in AWK
- /^[A-Za-z][0-9]$/ A letter or a letter followed by a digit

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

Built-in Variables

- **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