Some random notes:

Using vi/vim to look at files

Use "-R" (readonly) flag to prevent accidental overwriting of the file.

But

Use "-r filename" flag after an editor or system crash. (Recovers the version of filename that was in the buffer when the crash occurred.)

Awk (continuation)

AWK PROGRAMMING LANGUAGE

- NOTE -

We are going to use awk as the generic program name (like kleenex for facial tissue)

Wherever you see awk, you will actually use nawk (or gawk if you are using that on a LINUX box).

AWK Relational Operators

Returns 1 if true and 0 if false

!!! opposite of bash test command

AWK Relational Operators

All relational operators are left to right associative

< : test for less than

<= : test for less than or equal to

> : test for greater than

>= : test for greater than or equal to

== : test for equal to

!= : test for not equal

AWK Boolean (Logical) Operators

Boolean operators return 1 for true and 0 for false

&& : logical AND; tests that both expressions are true, left to right associative.

AWK Boolean (Logical) Operators

|| : logical OR ; tests that one or both of the expressions are true left to right associative.

! : logical negation; tests that expression is true.

Unlike bash, the comparison and relational operators don't have different syntax for strings and numbers.

ie: to test for equality use "==" in awk

rather than "==" to compare strings and "-eq" to compare numbers when using the bash test command.

More Built-in AWK Variables

FS : field separator specifies how to define fields (usually space, maybe also tab [whitespace]), may be modified by resetting the FS built in variable. (we have seen this already)

OFS : output field separator default is " " or a whitespace

More Built-in AWK Variables

RS : record separator specifies when the current record ends and the next begins default is "\n" or newline, useful option is "", or a blank line.

ORS : output record separator default is a "\n" or newline.

More Built-in AWK Variables

NF : number of fields (in line) variable.

NR : number of records (gives current line number).

FILENAME : the name of the file currently being read.

Basic structure of AWK use

The essential organization of an AWK program follows the form:

pattern { action }

The pattern specifies when the action is performed.

Like most UNIX utilities, AWK is line oriented.

That is, the pattern specifies a test that is performed with each line read as input.

If the condition is true, then the action is taken.

The default pattern is something that matches every line.

This is the blank or null pattern.

Two other important patterns are specified by the keywords "BEGIN" and "END."

As you might expect, these two words specify actions to be taken before any lines are read, and after the last line is read.

The AWK program:

```
BEGIN { print "START" }
   { print }
END { print "STOP" }
```

adds one line before and one line after the input file.

This isn't very useful, but with a simple change, we can make this into a typical AWK program:

```
BEGIN { print "File\tOwner"," }
    { print $8, "\t", $3}
END { print " - DONE -" }
```

The characters "\t" Indicates a tab character so the output lines up on even boundries.

The "$8" and "$3" have a meaning similar to a shell script.

Instead of the eighth and third argument, they mean the eighth and third field of the input line.

You can think of a field as a column, and the action you specify operates on each line or row read in.

There are two differences between AWK and a shell processing the characters within double quotes.

AWK understands special characters follow the "\" character like "t".

The Bourne and C UNIX shells do not.

Also, unlike the shell (and PERL) AWK does not evaluate variables within strings.

The second line, for example, could not be written:

{print "$8\t$3" }

As it would print "$8➔$3."

(where ➔ is a usually invisible tab).

Inside quotes, the dollar sign is not a special character. Outside, it corresponds to a field.

Say we want to print out the owner of every file

```
                    Field or column (RS=" ")
1111111111  2 33333333 4444    5555 666 77  8888 9999999999999

-rwxrwxrwx  1 rsmalley user    7237 Jun 12  2006 setup_exp1.sh
```

So we need field 3 and 9.

Example file - create the file owner.nawk and make it executable.

```
#!/bin/awk -f
BEGIN { print "File\tOwner" }
{ print $9, "\t", $3}
END { print " - DONE -" }
```

Now we have to get the input into the program. Pipe in the long directory listing.

```
alpaca.ceri.memphis.edu507:> ls -l | owner.nawk
File     Owner

*CHARGE-2002-107*          rsmalley
022285A.cmt          rsmalley
190-00384-07.pdf           rsmalley
. . .
zreal2.f             rsmalley
zreal2.o             rsmalley
 - DONE -
alpaca.ceri.memphis.edu508:>
```

Say I want to print out a command line argument?

Now we have a little problem.

In the shell, $1 is the first command line argument.

In awk, $1 is the first column of the input line.

How does one "fix" this?

Say I want to print out a shell or environment variable?

Here again there is a little problem.

AWK does not understand $VAR since the $ goes with column numbers.

How do I fix this?

Quotes to the rescue!

```
#!/bin/sh
denom=2
MSG="hello world"
NUMS="0.0,0"
echo msg: $MSG | nawk '{print $0}'
echo nums: $NUMS | nawk '{print $0}'
nawk '{print $1/'$denom' , '$denom' , '$NUMS' , ''"$MSG"'' }'
<<END
1
2
END
SCALE=2
FACTOR=5
RESCALE=`nawk 'BEGIN {print '$SCALE'*'$FACTOR'}'`
echo $RESCALE

alpaca.ceri.memphis.edu619:> Pvar.nawk
msg: hello world
nums: 0.0,0
0.5 2 0 0 hello world
1 2 0 0 hello world
10
alpaca.ceri.memphis.edu620:>
```

```
#!/bin/sh
denom=2
MSG="hello world"
NUMS="0.0,0"
echo msg: $MSG | nawk '{print $0}'
echo nums: $NUMS | nawk '{print $0}'
nawk '{print $1/'$denom' , '$denom' , '$NUMS' , "'"$MSG"'" }'
<<END
1
2
END
SCALE=2
FACTOR=5
RESCALE=`nawk 'BEGIN {print '$SCALE'*'$FACTOR'}'`
echo $RESCALE

alpaca.ceri.memphis.edu619:> Pvar.nawk
msg: hello world
nums: 0.0,0
0.5 2 0 0 hello world
1 2 0 0 hello world
10
alpaca.ceri.memphis.edu620:>
```

There are three ways to figure out the quotes

1) Learn how to think UNIX.

2) Experiment.

3) Ask a UNIX Wizard/Guru.

Many ways to skin a cat with escape/quotes

```
lpaca.581:> nawk 'BEGIN { print "Dont Panic!" }'
Dont Panic!
```

Would be nice to have in correct English

(i.e. with the apostrophe).

BUT

That is also a quote - which means something to the shell!
(Try it by just putting in an apostrophe.)

```
alpaca.581:> nawk 'BEGIN { print "Dont Panic!" }'
Dont Panic!
alpaca.582:> nawk 'BEGIN { print "Don'\''t Panic!" }'
Don't Panic!
alpaca.583:> nawk 'BEGIN { print "Don'"'"'t Panic!" }'
Don't Panic!
Alpaca.584:> echo Don\'t Panic! | nawk '{print}'
Don't Panic!
alpaca.585:> echo Don\'t Panic! | nawk "{print}"
Don't Panic!
```

Look carefully at the 2 lines above - you can (sometimes) use either quote (' or ") to protect the nawk program (depends on what you are trying to protect from the shell).

```
alpaca.586:> echo Don"'"t Panic! | nawk "{print}"
Don't Panic!
alpaca.587:> nawk 'BEGIN { print "\"Dont Panic!\"" }'
"Dont Panic!"
```

accessing shell variables in nawk

3 methods to access shell variables inside a nawk script ...

1. Assign the shell variables to awk variables after the body of the script, but before you specfiy the input

```
awk '{print v1, v2}' v1=$VAR1 v2=$VAR2 input_file
```

Note: There are a couple of constraints with this method;

- Shell variables assigned using this method are not available in the BEGIN section
- If variables are assigned after a filename, they will not be available when processing that filename …

e.g.

```
awk '{print v1, v2}' v1=$VAR1 file1 v2=$VAR2 file2
```

In this case, v2 is not available to awk when processing file1.

Also note: awk variables are referred to by just their name (no $ in front)

```
awk '{print v1, v2, NF, NR}' v1=$VAR1 file1 v2=$VAR2 file2
```

2. Use the -v switch to assign the shell variables to awk variables.

This works with nawk, but not with all flavours of awk.

```
nawk -v v1=$VAR1 -v v2=$VAR2 '{print v1, v2}' input_file
```

3. Protect the shell variables from awk by enclosing them with "'" (i.e. double quote - single quote - double quote).

```
awk '{print "'"$VAR1"'", "'"$VAR2"'"}' input_file
```

Looping Constructs in AWK

awk loop syntax are very similar to C and perl

while: continues to execute the block of code as long as condition is true

```
while ( x==y ) {
    . . .
    block of commands
    . . .
}
```

do/while

do the block of commands, while the test is true

```
do {
    . . .
    block of commands
    . . .
} while ( x==y )
```

The difference between while (last slide) and do/while is when the condition is tested. It is tested prior to running the block of commands for a while loop, but tested after running the block of commands in a do/while loop (at least one trip through block of commands will occur)

for loops

The for loop, allows iteration/counting as one executes the block of code.

It is one of the most common loop structures.

```
for ( x=1; x<=NF; x++) {
    . . .
    block of commands
    . . .
}
```

```
for ( x=1; x<=NF; x++) {
    . . .
    block of commands
    . . .
}
```

This is an extremely useful/important construct as it allows applying the block of commands to the elements of an array

(at least numerical arrays with all the elements "filled-in").

break and continue

break: breaks out of a loop

continue: restarts at the beginning of the loop

```
x=1
while (1) {
    if ( x == 4 ) {
       x++
       continue
    }
    print "iteration",x
    if ( x > 20 ) {
       break
    }
    x++
}
```

if/else/else if blocks

similar to bash but syntax is different (no then or fi, uses brackets { . . . } instead)

```
if ( conditional1 ) {
    . . .
    block of commands
    . . .
} else if ( conditional2 ) {
    . . .
    block of commands
    . . .
} else {
    . . .
    block of commands
    . . .
}
```

else if and else are optional

Simple awk example:

Say I have some sac files with the horrid iris dmc format file names

```
1999.289.10.05.26.0000.IU.KMBO.00.LHZ.SAC
```

and it would rename it to something more "user friendly" like KMBO.LHZ to save on typing while doing one of Chuck's homeworks.

```
alpaca.540:> more rename.sh
#!/bin/sh

#to rename horrid iris dmc file names

#call with rename.sh A x y
#where A is the char string to match, x and y are the field
#numbers in the original file name you want to use in the
#final name, and using the period/dot for the field seperator

#eg if the file names look like
#1999.289.10.05.26.0000.IU.KMBO.00.LHZ.SAC
#and you would ;ike to rename it KMBO.LHZ
#the 8th field is the station name, KMBO
#and the 10th field is the component name, LHZ
#so you would call rename.sh SAC 8 10
#(it will do if for all file names in your directory
#containing the string "SAC")

for file in `ls -1 *$1*`
do
mv $file `echo $file | nawk -F. '{print $'$2'"."$'$3'}'`
done
alpaca.541:>
```

Example

Checkbook balancing program in awk

- Simple tab-delimited text file into which recent deposits and withdrawals are entered.

- The idea is to hand this data file to an awk script that would automatically add up all the amounts and report the balance.

Input file format:

Fields are separated by one or more tabs.

After the date (field 1, $1), there are two fields: "exp field" and "inc field".

When entering an expense, a four-letter nickname is entered in the exp field, and a "-" (blank entry) in the inc field.
When entering a deposit, a four-letter nickname is entered in the inc field, and a "-" (blank entry) in the exp field.

Here's what an expense (debit) looks like:

```
23 Aug 2000   food   -   -   Y   Jimmy's Buffet      30.25
```

Here's what a deposit looks like:

```
23 Aug 2000   -      inco   -   Y      Boss Man       2001.00
```

Fields

```
11111111111→2   →   3333 → 4 → 5 → 66666666 →    7777777
```

Note, there are tabs (not spaces) between the fields, which you can't see in the display.

Now for the code

set up global variables

```
#!/usr/bin/awk -f
BEGIN {
    FS="\t+"
    months="Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec"
}
```

"#!..." allows execution directly from shell.

BEGIN block gets executed before nawk starts processing our checkbook file.

Set FS to "\t+" (one or more tabs).

In addition, we define a string called months.

Set subroutines (aka functions)

Define your own awk function.
Format -- "function", then the name, and then the parameters separated by commas, inside parentheses.

Finally a "{ }" code block contains the code that you'd like this function to execute.

```
function monthdigit(mymonth) {
    return (index(months,mymonth)+3)/4
}
```

nawk provides a "return" statement that allows the function to return a value.

```
function monthdigit(mymonth) {
    return (index(months,mymonth)+3)/4
}
```

This function converts a month name in a 3-letter string format into its numeric equivalent. For example, this:

```
print monthdigit("Mar")
```

....will print this:

What does this do?

```
index(months,mymonth)
```

Built-in string function index, returns the starting position of the occurrence of a substring (the second parameter) in another string (the first paramter), or it will return 0 if the string isn't found.

```
months="Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec"

          000000000111111111122222222223333333334444444444
          123456789012345678901234567890123456789012345678
                                          ↑

print index(months,"Aug")
29
```

To get the number associated with the month (based on the string with the 12 months) add 3 to the index (29+3=32) and divide by 4 (32/4=8, Aug is 8th month).

The string months was designed so the calculation gave the month number.

More functions/subroutines

three basic kinds of transactions, credit (doincome), debit (doexpense) and transfer (dotransfer).

```
function doincome(mybalance) {
    mybalance[curmonth,$3] += amount
    mybalance[0,$3] += amount
}
function doexpense(mybalance) {
    mybalance[curmonth,$2] -= amount
    mybalance[0,$2] -= amount
}
function dotransfer(mybalance) {
    mybalance[0,$2] -= amount
    mybalance[curmonth,$2] -= amount
    mybalance[0,$3] += amount
    mybalance[curmonth,$3] += amount
}
```

The main code block will process each line of the checkbook file sequentially, calling one of these functions so that the appropriate transactions are recorded in an awk array.

All three functions accept one argument, called mybalance.

mybalance is a placeholder for a two-dimensional array, which we'll pass in as an argument.

We will be storing the data in a 2-dimensional "array".

What is an "array"?

An array is a table of values, called elements.

The elements of an array are distinguished by their indices.

Indices in awk may be either numbers or strings.

(as awk maintains a single set of names for naming variables, arrays and functions, you cannot have a variable and an array with the same name in the same awk program.)

Arrays in awk superficially resemble arrays in other programming languages; but there are fundamental differences.

The most fundamental or significant difference is that any number or string may be used as an array index in awk, not just consecutive integers.

(in the end in awk, array indicies, even numerical ones, are strings)

In awk, you also don't need to specify the size of an array before you start to use it.

Arrays in awk are associative.
This means that each array is a collection of pairs: an index, and its corresponding array element value:

```
Element 4       Value 30
Element 2       Value "foo"
Element 1       Value 8
Element 3       Value ""
```

The pairs are shown in jumbled order because the array index order is irrelevant and has nothing to do with storage in memory.

One advantage of associative arrays is that new pairs can be added at any time. Adding a 10th element whose value is "number ten" to our example array.

```
Element 10     Value "number ten"
Element 4      Value 30
Element 2      Value "foo"
Element 1      Value 8
Element 3      Value ""
```

Now the array is sparse, which just means some indices are missing: it has elements 1 through 4 and 10, but doesn't have elements 5 through 9.

Indices of associative arrays don't have to be positive integers.

Any number, or even a string, can be an index.

Here is an array which translates words from English into French:

```
Element "dog" Value "chien"
Element "cat" Value "chat"
Element "one" Value "un"
Element 1     Value "un”
```

We use the number one in each language spelled-out and in numeric form--a single array can have both numbers and strings as indices.

(array subscripts in awk are actually always strings)

The principal way of using an array is to refer to one of its elements.

An array reference is an expression which looks like this:

array[index]

Here, array is the name of an array.

The expression index is the index of the element of the array that you want.

Array elements are assigned values just like awk variables:

array[subscript] = value

array is the name of your array.

subscript is the index of the element of the array that you want to assign a value.

value is the value you are assigning to that element of the array.

mis-indexing of arrays (when they are indexed by integers) is one of the most common bugs in programming.

If you mis-index an array in awk, it makes a new element with that index and a null value.

(Wastes space and does not return value you were trying to obtain.)

To explicitly set an array element, use brackets to specify which index of the array you are setting.

strings - when used as indices or values - have to be in quotes

```
BEGIN {
animals["dog"] = "perro"
animals["cat"] = "gato"
stuff[1]=1
stuff[4]=4
stuff[-1]=-1
stuff[0]=0
print animals["dog"]
print stuff[1]
print stuff[2]
print stuff[3]
print stuff[4]
print stuff[-1]
print stuff[0]
 }
```

Reference to elements that don't exist

Execute the nawk script

```
smalley$ nawk -f arrays.nawk
perro
1

4
-1
0
smalley$
```

Null output for the ones that don't exist

to delete an array element, use the delete command

```
delete myarray[1]
```

Say we have this file and we want to put it into numerical order in an awk array.

```
carpincho:ESCI7205 smalley$ more data.txt
4
1
3
2
a
7
B
carpincho:ESCI7205 smalley$
```

Try this.

(white box - look at raw and sorted file, green box - fill array with sorted elements and numerical index, yellow box print out array indices and values.)

```
carpincho:ESCI7205 smalley$ more awkex1.nawk
#!/bin/bash
cat data.txt
echo ------
sort -n data.txt
echo ------
sort -n data.txt|\
awk 'BEGIN {c=0} {
if ( $0 > 0 ) {
print c, $0
  myarray[c]=$0; c++;
  }
}
END {
    for ( c in myarray ) printf ":: %s %s ",c,myarray[c];
printf "\n";
    }
'
carpincho:ESCI7205 smalley$
```

```
carpincho:ESCI7205 smalley$ awkex1.nawk
4
1
3
2
a
7
b
------
a
b
1
2
3
4
7
------
0 a
1 b
2 1
3 2
4 3
5 4
6 7
:: 2 1 :: 3 2 :: 4 3 :: 5 4 :: 6 7 :: 0 a :: 1 b
carpincho:ESCI7205 smalley$
```

Orginal file

After sort

Store in array: array index plus value

When print out (random order)

you can also set arrays using the split command

```
split(“string”,destination array,separator)
```

split also returns the number of indices

```
numelements=split("Jan,Feb,Mar,Apr,May",mymonths,",”)
```

Splits the string into array elements using the “,” to break the string into elements, and returns numelements=5 and mymonths[1]=“Jan“

A multi-dimensional array is an array in which an element is identified by a sequence of indices, instead of a single index.

For example, a two-dimensional array requires two indices.

The usual way to refer to an element of a two-dimensional array named grid is with grid[x,y].

Back to our checkbook

Record information into "mybalance" as follows.

The first dimension of the array ranges from 0 to 12, and specifies the entire year (0) or month (number of month).

Our second dimension is a four-letter category, like "food" or "inco"; this is the actual category we're dealing with.

(remember that the dimensions are not fixed – we can add categories at will)

So, to find the entire year's balance for the food category, you'd look in

mybalance[0,"food"].

To find June's income, you'd look in

mybalance[6,"inco"].

Arrays are passed by reference.
We also refer to several global variables: curmonth, (numeric value of month of current record), $2 (expense category), $3 (income category).

```
function doincome(mybalance) {
    mybalance[curmonth,$3] += amount
    mybalance[0,$3] += amount
}
function doexpense(mybalance) {
    mybalance[curmonth,$2] -= amount
    mybalance[0,$2] -= amount
}
function dotransfer(mybalance) {
    mybalance[0,$2] -= amount
    mybalance[curmonth,$2] -= amount
    mybalance[0,$3] += amount
    mybalance[curmonth,$3] += amount
}
```

Passing of information between calling routine and subroutine.
Two basic ways.

By reference

Tell subroutine where the information is in the memory and the subroutine uses it.
Changes made by the subroutine are global.

By value

Give the subroutine a copy of the information.
Any changes made by the subroutine are local to its copy of the data.

The main code block contains the code that parses each line of input data.

Remember, because we have set FS correctly, we can refer to the first field as $1, the second field as $2, etc.

When the functions are called, they can access the current values of curmonth, $2, $3 and amount from inside the function.

```
#main program
{
    curmonth=monthdigit(substr($1,4,3))
    amount=$7

    #record all the categories encountered
    if ( $2 != "-" )
       globcat[$2]="yes"
    if ( $3 != "-" )
       globcat[$3]="yes"

    #tally up the transaction properly
    if ( $2 == "-" ) {
       if ( $3 == "-" ) {
          print "Error: inc and exp fields are both blank!"
          exit 1
       } else {
          #this is income
          doincome(balance)
          if ( $5 == "Y" )
             doincome(balance2)
       }
```

```
    } else if ( $3 == "-" ) {
       #this is an expense
       doexpense(balance)
       if ( $5 == "Y" )
       doexpense(balance2)
    } else {
       #this is a transfer
       dotransfer(balance)
       if ( $5 == "Y" )
          dotransfer(balance2)
    }
}
#end of main program
END {
    bal=0
    bal2=0
    for (x in globcat) {
       bal=bal+balance[0,x]
       bal2=bal2+balance2[0,x]
        }
        printf("Your available funds: %10.2f\n", bal)
        printf("Your account balance: %10.2f\n", bal2)
}
```

Input file:

```
23 Aug 2000   food   -   -   Y   Jimmy's Buffet       30.25
23 Aug 2000   -      inco  -   Y   Boss Man             2001.00
```

Output to the screen:

```
Your available funds:     1174.22
Your account balance:     2399.33
```

More string functions

```
print tolower(mystring)

print toupper(mystring)
```

mysub=substr(mystring,startpos,maxlen)

mystring: a string variable or a literal string from which a substring will be extracted.

Startpos: starting character position.
Maxlen: maximum length to extract.
(if length(mystring) is shorter than startpos+maxlen, your result will be truncated.)

substr() won't modify the original string, but returns the substring instead.

match() searches for a regular expression.

match returns the starting position of the match, or zero if no match is found, and sets two variables called RSTART and RLENGTH.

RSTART contains the return value (the location of the first match), and RLENGTH specifies its span in characters (or -1 if no match was found).

string substitution sub() and gsub().

:

Modify the original string.

sub(regexp,replstring,mystring)

sub() finds the first sequence of characters in mystring matching regexp, and replaces that sequence with replstring.

gsub() performs a global replace, swapping out all matches in the string.

AWK patterns (regular expressions)

Print out lines matching "z_max"

```
nawk '/z_max/ {print $5}'
```

Print out 5th field of lines matching "[1]"

```
nawk '/\[1\]/ {print $3, $2, 14,0,1,1,$1 }' samgps.dat
```

Print out stuff from lines matching "[2]", that don't contain the strings "ASLO" and "CHYY"

```
nawk '/\[2\]/&&!/ASLO/&&!/CHYY/ {print $3, $2}' samgps.dat
```

Print out stuff from lines that don't contain "[0" or "[?" or "[-" or "[c" or "[w" or "[1"

```
nawk '!/\[0/&&!/\[\?/&&!/\[-/&&!/\[c/&&!/\[w/&&!/\[1/ {print
NR, $5}' $GPSDATA
```

AWK patterns (regular expressions)

Print out lines where the 4th field squared is <2500

```
nawk '($4*$4)<2500 {print $0}'
```

Print out stuff from lines where LONMIN<=1st field<=LONMAX and LATMIN<=2nd field<=LATMAX and the 10th field is >=MINMTEXP

```
nawk '('$LONMIN'<=$1)&&($1<='$LONMAX')&&('$LATMIN'<=
$2)&&($2<='$LATMAX')&&($10>='$MINMTEXP') {print $1, $2, $3,
$4, $5, $6, $7, $8, $9, $10, '$MECAPRINT' }'
```

Print out lines where the 3rd field is < 60 and the 4th field is > 10, where the pattern is passed using a shell variable

```
nawktst_shal=\(\$3\<60\&\&\$4\>10\)
nawk ''$nawktst_shal' {print $0}'
```

AWK patterns (regular expressions)

If the first 4 characters of the last field is > 1995, print out the whole line and the number of fields.

```
nawk 'substr($NF,1,4)>1995 {print $0, NF}'
```

NF is the awk variable for the number of fields.

The last field is field number NF.

$NF is the value of the last field.

```
$more rtvel.nawk
BEGIN { output=0 }
{ if ( !/Stnm/ ){
if( output == 1 ) print $0;
}
else
{ output =1
}
}

nawk -f $SAMDATA/rtvel.nawk $VELFILE
```

Reads the input file till finds the string "Stnm" and after finding it, prints out records ($0).

```
nawk '{print ($1>=0?$1:360+$1)}'
```

Syntax: (test?stmt1:stmt2)

This will do a test

(in this case: $1>=0)

If true it will output stmt1 ($1)

(does this: `nawk '{print $1}'`

If false it will output stmt2 (360+$1)

(does this: `nawk '{print 360+$1}'`

(in this case we are changing longitudes from the range/format -180<=lon<=180 to the range/format 0<=lon<=360)

Write a file with nawk commands and execute it.

```
#!/bin/sh
#general set up
ROOT=$HOME
SAMDATA=$ROOT/geolfigs
ROOTNAME=$0_ex
VELFILEROOT=`echo $latestrtvel`
VELFILEEXT=report
VELFILE=${SAMDATA}/${VELFILEROOT}.${VELFILEEXT}
#set up for making gmt input file
ERRORSCALE=1.0
SEVENFLOAT="%f %f %f %f %f %f %f "
FORMATSS=${SEVENFLOAT}"%s %f %f %f %f\\\\n"
GMTTIMEERRSCFMT="\$2, \$3, \$4, \$5, ${ERRORSCALE}*\$6, ${ERRORSCALE}*
\$7, \$8”
#make the station list
STNLIST=`$SAMDATA/selplot $SAMDATA/gpsplot.dat pcc`
#now make nawk file
echo $STNLIST {printf \"$FORMATSS\", $GMTTIMEERRSCFMT, \$1, \$9,
$ERRORSCALE, \$6, \$7 } > ${ROOTNAME}.nawk
#cat ${ROOTNAME}.nawk

#get data and process it
nawk -f $SAMDATA/rtvel.nawk $VELFILE | nawk -f ${ROOTNAME}.nawk
```

Notice all the "escaping" ("\" character) in the shell variable definitions (`FORMATSS` and `GMTTIMEERRSCFMT`) and the echo.

Look at the nawk file - it looses most of the escapes.

The next slide shows the nawk file at the top and the output of applying the nawk file to an input data file at the bottom.

```
/ALGO/||/ANT2/||/ANTC/||/ARE5/||/AREQ/||/ASC1/||/AUTF/||/
BASM/||/BLSK/||/BOGT/||/BOR4/||/BORC/||/BRAZ/||/CAS1/||/
CFAG/||/COCR/||/CONZ/||/COPO/||/CORD/||/COYQ/||/DAV1/||/
DRAO/||/EISL/||/FORT/||/FREI/||/GALA/||/GAS0/||/GAS1/||/
GAS2/||/GAS3/||/GLPS/||/GOUG/||/HARB/||/HARK/||/HART/||/
HARX/||/HUET/||/IGM0/||/IGM1/||/IQQE/||/IQTS/||/KERG/||/
KOUR/||/LAJA/||/LHCL/||/LKTH/||/LPGS/||/MAC1/||/MARG/||/
MAW1/||/MCM1/||/MCM4/||/OHI2/||/OHIG/||/PALM/||/PARA/||/
PARC/||/PMON/||/PTMO/||/PWMS/||/RIOG/||/RIOP/||/SALT/||/
SANT/||/SYOG/||/TOW2/||/TPYO/||/TRTL/||/TUCU/||/UDEC/||/
UEPP/||/UNSA/||/VALP/||/VESL/||/VICO/||/HOB2/||/HRA0/||/DAVR/
{printf "%f %f %f %f %f %f %s %f %f %f %f\n", $2, $3, $4,
$5, 1.0*$6, 1.0*$7, $8, $1, $9, 1.0, $6, $7 }

-78.071370 45.955800 -6.800000 -8.600000 0.040000 0.040000
0.063400 ALGO 12.296000 1.000000 0.040000 0.040000⏎
-70.418680 -23.696350 26.500000 8.800000 1.010000 1.010000
-0.308300 ANT2 0.583000 1.000000 1.010000 1.010000⏎
-71.532050 -37.338700 15.000000 -0.400000 0.020000 0.040000
-0.339900 ANTC 8.832000 1.000000 0.020000 0.040000⏎
-71.492800 -16.465520 -9.800000 -13.000000 0.190000 0.120000
-0.061900 ARE5 3.348000 1.000000 0.190000 0.120000⏎
-71.492790 -16.465510 14.100000 3.800000 0.030000 0.020000
-0.243900 AREQ 7.161000 1.000000 0.030000 0.020000⏎...
```