Programmers spend a lot of time

- doing simple, mechanical, repetitive data manipulation
  - changing the format of data, especially for munging big data
  - checking its validity
  - finding items with some property
  - adding up numbers
  - printing reports
  - managing data (&small databases)
  - Processing system logs...
- AWK is an interpreted programming language designed to handle such tasks with short, simple programs, often only a couple of lines long

AWK : Aho, Weinberger & Kernighan

- something to handle text as easily as numbers…
  - (with no need to declare variables)
- AWK is also a great prototyping language,
  start with a few lines
  and keep adding
  - Jon Bentley "Programming Pearls
- AWK is more like a declarative language,
  - in that it is more data driven than imperative.
- Works on : If there's a match then let's have some action!
  - Easy for sports fans!?

AWK : Applies to the Whole Kabosh

- An AWK program is a sequence of
  - patterns as regex that tell AWK what to look for in each line the input data
  - and actions that tell AWK what to do when it finds the pattern
- BUT THIS CAN HAVE UNEXPECTED CONSEQUENCES :
  - **THE ACTIONS ARE APPLIED TO EACH MATCHING LINE**
  - **AND IF EVERY LINE OF THE FILE MATCHES,**
    THEN IT APPLIES TO EACH LINE OF THE FILE!
    WHICH CAN HAPPEN FOR
    - EITHER NO PATTERN
    - OR A PATTERN THAT MATCHES ALL LINES
- THIS CAN BE A PECULIAR PROGRAMMING CATCH:
  - Normally: for only one program statement => then only one execution;
  - but not here; the program instruction is applied to every matching line!

Some AWK rewards!

- Can handle text as easily as numbers…
  - (with no need to declare variables)
- Is one of the handiest, most powerful Unix tools
- included in all distributions
- Performs basic numeric operations: +, -, *, / , ^
  - And the more common numerical functions:
    int, rand, srand, exp, log, sqrt, sin, cos, atan2
- Has basic programming control structures:
  - if …
  - while …

More AWK rewards!

- Handles fields within regularly patterned lines…
  - E.g. output from ls –l etc.
  - Saves having to delimit words using regular expression \< >\ or –w flag ; instead AWK splits fields/columns
  - Faster since optimised specially in 'C', cache, RAM & disk optimal
  - Basically can act as a simple spreadsheet…
  - probably less error prone than spreadsheet cell refns.

~ 2000AD estimates of £10^9 (a trillion) lost in the city (London Stock Exchange) due to 'fat-finger' & Excel errors !

… but info provided by a 'reliable' competitor!?

Review of Awk Principles

- Awk's purpose: to give Unix a general purpose programming language that handles text (strings) as easily as numbers
  - makes Awk one of the most powerful of the Unix utilities
- Awk process fields while ed/sed process lines
- nawk (new awk) is the new standard for Awk
  - Designed to facilitate large awk programs
- Awk gets it's input
  - directly from standard input
  - So, in Linux it supports redirection, pipes and files

AWK

- A programming language for handling common data manipulation tasks with only a few lines of program
- *Awk* is a *pattern action* language
  - act on match – no match, no act!
- The language looks a little like *C* but automatically handles input, field splitting, initialization, and memory management
  - Built-in string and number data types
  - No variable type declarations
- *Awk* is a great prototyping language
  - Start with a few lines and keep adding until it does what you want
- *nawk* is a newer version
- *mawk* is a POSIX version, allegedly quicker with extensions
- *gawk* is the GNU version –
- On our current lab installation : awk & nawk default to gawk
- mawk is also on Ubuntu – but often an older version!

History

1977 – Implemented by Al Aho, Peter Weinberger, and Brian Kernigan

- In part as an experiment to see how *grep* and *sed* could be generalized to deal with numbers as well as text
- Originally intended for very short programs
- But people started using it
  and the programs kept getting bigger and bigger!

1985, new awk, or *nawk*, was written to

- add enhancements
- facilitate larger program development
- Major new feature is user defined functions

1996 - Gawk is the GNU (open source Unix equivalent) for Linux –

- virtually the Linux standard.
- Proper awk scripts should run with gawk; and frequently is implemented by awk invoking gawk.

AWK – after Aho, Weinberger, Kernighan – at At&T. (K also in – C & Unix!)

- Initially designed for 1-liners for basic processing of text & numeric fields, e.g. in pipes between processes,
- But saves a lot of C or scripting, so it developed as a convenient multiple line programming language.
- Between C & shell in terms of speed, suitable for some systems prog.!

- So we do it before scripting, so you don't try to reinvent the wheel!.

- Drew on Bourne shell & C, influenced Perl, Lua & Korn shell.
- Only universal Unix scripting language in addition to the Bourne shell.
- Also required by Linux Standard Base
- Implementations exist for most other OS'es including Windows
  - Windows Powershell has similar features and vaguely similar syntax!?

new awk (nawk) enhancements

- Some enhancements in nawk include:
  - Dynamic regular expressions
    - Text substitution and pattern matching functions
  - Additional built-in functions and variables
  - New operators and statements
  - Input from more than one file
  - Access to command line arguments

- nawk also improved error messages which makes debugging considerably easier under nawk than awk
- On most systems, nawk has replaced awk, and on some mawk has replaced nawk.

Tutorial

- Program structure
- Running an Awk program
- Error messages
- Output from Awk
- Record selection
- BEGIN and END
- Number crunching
- Handling text
- Built-in functions
- Control flow
- Arrays

Structure of an AWK Program

- An optional BEGIN segment
  - For processing to execute prior to reading input
  - Each statement executed once only before program body
- Program body : pattern - action pairs
  - Processing for input data
  - For each pattern matched, the corresponding action is taken
    - If no action, print line
    - If no pattern, apply to all lines of input file
  - Each statement of the program body, in sequence, is applied to each line, of the input file, in sequence...
  - a line of code can apply any number of times
  - More like declarative or database apps.
- An optional END segment
  - Processing after end of input data
  - Each statement executed once only after program

LOOPY

```
BEGIN
pattern {action}
pattern {action}
  .
  .
  .
pattern { action}
END
```

CATCH => If FACT then ACT!

- If **fact** true for line, then {**act** within brackets}
- UNLIKE NORMAL PROGRAMMING… THIS IS
  - Loopy : no fact, then default loops over all lines
  - Printy : no act, then default is to Print.

LOOPY

- Catch : Easy to overloop / overlook
  - Normally a statement executes once
  - BUT HERE it runs as often as it can
    - (according to the fact-match paradigm!)
    - More like database, declarative languages, etc.

Simple example … shopping

```
# BEGIN .. done before first record is read
BEGIN       { total = 0 }          # initialise total to 0
/Fruit/     {printf "\n\n%s \t\tCost per fruit type\n\n", $0}
/Fruit/     { next }          # skip file header, the first line .. Other ways possible
            { printf "%s \t\t\t %.2f\n", $0,$2*$3}
            { total += $2*$3 }
# END ensures actions are done after last record is read
END         { printf "\n\t\t\t\t\t Total cost is \t$%.2f\n\n", total }
```

Fruit	Item Cost	Number Bought
apples	50	5
oranges	50	10

Running the program above, on the data at the side, gives the result below.

Fruit	Item Cost	Number Bought	Cost per fruit type
apples	50	5	250.00
oranges	50	10	500.00
		Total cost is	$750.00

Actual processing algorithm

```
For each program_line
    if program_line starts with BEGIN
    then      execute BEGIN action first and only once before rest of program
    else if program_line starts with END
        then      execute END action last and only once after rest of program
        else
        for each data_file_line
            if    program_line
                      pattern (regex} or condition {column value}
                  matches
                      data_file_line
            then      execute action
        endfor … each data_file_line
Endfor … each program_line.
```

NB each program line, except for those beginning with BEGIN & END is applied to all matching data lines, in the order encountered in the program.

For ease and clarity of presentation, this omits the absence of either pattern or action

Pattern-Action Structure

- Every program statement has a *pattern*, an *action*, or both
  - Default *pattern* is to match all lines (or records)*
  - Default *action* is to print current record
- Patterns are simply listed; actions are enclosed in { }s
- Awk scans a sequence of input lines, or records, one by one, searching for lines that match the pattern
  - Meaning of match depends on the pattern
  - /put/ matches if the string “put” is in the record
  - $3 > 0 matches if the 3rd field is > 0

* Lines and records are often used interchangeably in IT

Running an AWK Program

There are several ways to run an Awk program

- awk ‘program’ input_file(s)
  - program and input files are provided as command-line arguments
- awk ‘program’
  - program is a command-line argument; input is taken from standard input (yes, awk is a filter!)
  - can use pipes | and file redirection <, >, >>
- awk -f program_file_name input_files
  - program is read from a file

(Clearly the best way to develop non-one-liner programs
For one-liners - history retains one-line code!

Errors

- Awk provides diagnostic error messages

```
awk '$3 == 0 [ print $1 }' emp.data
awk: syntax error near line 1
awk: bailing out near line 1
```

- Or if using nawk

```
nawk '$3 == 0 [ print $1 }' emp.data
nawk: syntax error at source line 1
   context is
   $3 == 0 >>>  [ <<<
      1 extra }
      1 extra [
nawk: bailing out at source line 1
      1 extra }
      1 extra [
```

The traditional advantage of an interpreter is that it tells exactly where it breaks down during the run. Compilers usually produce more optimised convoluted code & hence convoluted alerts!
NB Java bytecode is interpreted on a JVM so is not so bad and compiler and runtime environments have moved on.

Some of the Built-In Variables

- NF - Number of fields in current record
- NR - Number of records read so far
- $0- Entire line
- $*n*- Field *n*
- $NF - Last field of current record

Simple Output From AWK

- Printing Every Line
  - If an action has no pattern, the action is performed for all input lines
    - { print } will print all input lines on stdout
    - { print $0 } will do the same thing
- Printing Certain Fields
  - Multiple items can be printed on the same output line with a single print statement
  - { print $1, $3 }
  - Expressions separated by a comma are, by default, separated by a single space when output

- NF, the Number of Fields
  - Any valid expression can be used after a $ to indicate a particular field
  - One built-in expression is NF, or Number of Fields
  - { print NF, $1, $NF } will, for the current record, print the number of fields, the first field, and the last field
  -
- Computing and Printing
  - You can also do computations on the field values and include the results in your output
  - { print $1, $2 * $3 }

- Printing Line Numbers
  - The built-in variable NR can be used to print line numbers
  - { print NR, $0 } will print each line prefixed with its line number
- Putting Text in the Output
  - You can also add other text to the output besides what is in the current record
  - { print “total pay for”, $1, “is”, $2 * $3 }
  - Note that the inserted text needs to be surrounded by double quotes

Fancier Output

- Lining Up Fields
  - Like C, Awk has a *printf* function for producing formatted output – similar to common older approach in Python
  - *printf* has the form
    - *printf(format, val1, val2, val3, …)*

    { printf(“total pay for %s is $%.2f\n”, $1, $2 * $3) }
  - When using *printf*, formatting is under your control so no automatic spaces or NEWLINEs are provided by Awk. You have to insert them yourself.

    { printf(“%-8s %6.2f\n”, $1, $2 * $3) }

printf Examples

Format conversion descriptors follow standard C format:

%d - decimal

%s – string

%f – floating point

- if preceded by 2 numbers separated by a ‘.’
- the first number is the total fieldwidth for the number
- The second is the decimal digits
- Remember to allow one space for the decimal point!

- printf(“I have %d %s\n”, how_many, animal_type)
- printf(“%-10s has $%6.2f in their account\n”, name, amount)
- printf(“%10s %-4.2f %-6d\n”, name, interest_rate, account_number)
- printf(“\t%d\t%d\t%6.2f\t%s\n”, id_no, age, balance, name)

Formatted Output

- printf provides formatted output
- Syntax is printf("format string", var1, var2, ….)
- Format specifiers
  - %d - decimal number
  - %f - floating point number
  - %s - string
  - \n - NEWLINE
  - \t - TAB
- Format modifiers
  - n column width
  - .n number of decimal places to print
  - - left justify in column

Formatting strings…

- Don't need to include text in strings between vars

  print "total pay for", $1, "is", $2 * $3
- Instead include formatting info within string, indicated by %, followed by list of corresponding vars.

  printf("total pay for %s is $%.2f\n", $1, $2 * $3)
- \n, \t : newline, tab
- %s, %o, %d, %f => string, octal, decimal, floating point etc.
  - %-8s => 8 char string,
    - => left justified (aligned to left of field!),
      otherwise right justified, (aligned to right of field)
  - %.2f => floating point,
    .2 => truncated after 2 digits right of decimal point
    (e.g. for money), and as many as you need for the rest
    (can result in misaligned columns)
  - %9.2f => fixed width field of size 9, for aligned cols, 2 being decimal

Awk as a Filter

- Since Awk is a filter, you can also use pipes with other filters to massage its output even further
- Suppose you want to print the data for each employee along with their pay and have it sorted in order of increasing pay
  - Note sort works on lines, so is primarily affected by the first field

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


Selection

- Awk patterns are good for selecting specific lines from the input for further processing
- Selection by Comparison (hourly rate >= 5)
  - $2 >=5 { print }
- Selection by Computation (pay=rate*hrs > 50)
  - $2 * $3 > 50 { printf("%6.2f for %s\n", $2 * $3, $1) }
- Selection by Text Content (name is Susie)
  - $1 == "Susie" - NB – needs an exact match to $1
  - /Susie/ - string match… can match Susie's
- Combinations of Patterns (rate >=4 or hrs >=20)
  - $2 >= 4 || $3 >= 20

Data Validation

- Validating data is a common operation
- Awk is excellent at data validation

```
NF != 3       { print $0, "number of fields not equal to 3" }
$2 < 3.35     { print $0, "rate is below minimum wage" }
$2 > 10       { print $0, "rate exceeds $10 per hour" }
$3 < 0        { print $0, "negative hours worked" }
$3 > 60       { print $0, "too many hours worked" }
```


BEGIN and END

- BEGIN matches before the first input line is read;
- END matches after the last input line has been read
- Not just restricted to printing, but to initialising variables before processing and wrapping up at the end.. e.g. with a command that does not process any input data file with an awk command
- This allows for initial and wrap-up processing: This prints : a header, a blank, all lines & a footer.

```
BEGIN { print "NAME    RATE    HOURS"; print "" }
      { print }
END   { print "total number of employees is", NR }
```


Computing with AWK

- Counting is easy to do with Awk

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

- Computing Sums and Averages is also simple

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

END delays processing until all records have been read.

Handling Text

- One major advantage of Awk :
  - It can handle strings as easily as numbers
  - conveniently translates between as needed
- This program finds the employee who is paid the most per hour

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

NOTE : this is effectively a 2-line find and print the max!

- String Concatenation
  - New strings can be created by combining old ones

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

  Note : unlikely to need a space before $1, to ensure it is parsed as a separate token…but that space will not be included in string concatenation; however following " " will be, and acts as space separator in concat string!
- Printing the Last Input Line
  - Although NR retains its value after the last input line has been read, $0 does not

```
{ last = $0 }
END { print last }
```


Built-In Functions

- Arithmetic
  - sin, cos, atan, exp, int, log, rand, sqrt
- String
  - length, substitution, find substrings, split strings
- Output
  - print and printf to file
- Special
  - system - executes a Unix command
    - system("clear") to clear the screen
    - Note double quotes around the Unix command
  - exit - stop reading input and go immediately to the END pattern-action pair if it exists, otherwise exit the script

Built-in Functions

- Awk contains a number of built-in functions.
  - *length* is one of them.
- The *length* function can be used to count Lines, Words, and Characters, (an alternative wc!?)
  - Note : +1 is for newline, not for spaces...

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


Operators

- = assignment operator;
  - sets a variable equal to a value or string
- String concatenation - no operator needed on RHS of '='.
- == equality operator; returns TRUE is both sides are equal
- != inverse equality operator; TRUE if both are inequal.
- && logical AND
- || logical OR
- ! logical NOT
- <, >, <=, >= relational operators
- +, -, /, *, %, ^ (% : integer division, ^ : power function)

^ the power or exponentiation operator… 2^3 = 2*2*2 = 8
The number of times the first digit is multiplied by itself.

Break for now

To Follow

Control flow statements

Arrays

Control Flow Statements

- Awk provides several control flow statements for making decisions and writing loops
- If-Else

if (expression is true or non-zero){
statement1
}
else {
statement2
}

where *statement1* and/or *statement2* can be multiple statements enclosed in curly braces { }s

– the *else* and associated *statement2* are optional

Control Flow Statements

- Awk provides several control flow statements for making decisions and writing loops
- If-Else…

(NB pattern condition $2>6 is an implied if; whereas (n>0) following is explicit)

```
$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 "nobody is paid more than $6/hour"
    }
```

Line continuation '\' escapes newline

Often accidentally omitted when copied code is magnified for class presentation, and original source is on a single line, also block braces easily omitted

Loop Control

- While …pre-test (test done before loop)

while (*expression is true or non-zero) {*
statement1
}

- Do While…post-test (test done after loop)

do {
statement1
}
while (expression)

NB Pretests may not execute even once,
Postests will always loop at least once.

Loop Control

- While

```
# interest1 - compute compound interest
#   input: amount rate years
#   output: compound value at end of each year v=a(1+r)^y
{   y = 1
   while (y <= $3) {
            printf("\t%.2f\n", $1 * (1 + $2) ^ y)
            y = y + 1
   }
}
# note '#' is the comment symbol, for us, not computers!
# use comments a lot: Amnesia isn't … what's it again!?
```


- For

for(expression1; expression2; expression3) {
statement1
}

– This has the same effect as:

expression1
while (expression2) {
statement1
expression3
}

– for(;;) is an infinite loop

- For

```
# interest2 - compute compound interest
#   input: amount rate years
#   output: compound value at end of each year
#    v=a(1+r)^y
{ for (y = 1; y <= $3; y = y + 1)
      printf("\t%.2f\n", $1 * (1 + $2) ^ y)
}
```


Arrays

- Awk provides arrays for storing groups of related data values

```
# reverse - print input in reverse order by line
        { line[NR] = $0 }     # remember each line
 END { i = NR     # print lines in reverse order
          while (i > 0) {  # using a countdown loop
              print line[i]  # printing each array el.
              i = i – 1      # in reverse order
          }
}
```

Alternative loop over array:- (increasing index)
for (i in line) print line(i)

Simple examples with lines

- BEGIN {nlines = 0} # initialise to 0!
  /tag/ { nlines = nlines + 1 } # count another line with 'tag'
  END { print nlines } # print lines with /tag/
- $1 > max { max = $1; maxline = $0 } # find max & line no.
  END { print max, maxline } # print max & line no.
- Pseudocode for normal imperative language, needs a loop

```
read (first)     # the first in list
max = first      # which is the max. encountered so far!
while not End of File do# keep going … until the end
    read (next)        # get the next in list
    if next > max then max = next # reset max. if bigger
end_while
```


Simple one-line examples

```
NF > 0              # print every line with at least one field
length($0) > 80     # print lines longer than 80 chars
{ print NF, $0}     # print no. of fields followed by entire line
{ print $2, $1 }    # print first two fields of each line
                    #                    in reverse order
```

- { temp = $1; $1 = $2; $2 = temp; print }

```
# switch the first two fields of every line and print them
# NB if no action (i.e no { } ) is specified, print is the default action,
# but if another action is specified within { }, it won't print unless told
```

- { $2 = ""; print } # print every line after deleting second field

Careful with (shhh!)witches….

- Switching a & b! (Trivial in Python with tuples, but unusual!)

```
a = b; # a now has replaced and lost it's original value with b
b = a # so b is actually taking a, which is actually b.. 2B's
```

- Use a temporary variable store to avoid over-write & loss

```
temp = a;  # kept original a in temp
a = b      # a takes b value
b = temp   # b takes temp, which is original a – OK!
```

- But do it in the right order… or bags again

```
    temp = a
    b = a   # b has lost it's original value, temp has a, no b!
# a can be set only to original value, b is gone!  Only aa's left!
```

- **(FYI regex can do it with backreferences for matched strings !?)**

Fields & lines

- # print the fields of each line in reverse order
  { for (i = NF; i > 0; i = i - 1) printf("%s ", $i); printf("/n") }

? handy for log time analysis, but bash has tac – reverse cat

- # print the sum of the fields for each line

```
{ sum = 0 # sum initialised to 0 on each line
  for (i = 1; i <= NF; i = i + 1) sum = sum + $i
  print sum
}
```

- # sum all the fields of every line..

```
BEGIN {sum = 0}
    { for (i = 1; i <= NF; i = i + 1) sum = sum $i }
  END { print sum }
```

Review of Awk Principles

- Awk's purpose:
  - to give Unix a general purpose programming language that handles text (strings) as easily as numbers
  - This makes Awk one of the most powerful of the Unix utilities
- Awk process fields while ed/sed process lines
- nawk (new awk) is the new standard for Awk
  - Designed to facilitate large awk programs
  - Better debugging if you remember – see slide 18
    - Also next side for convenience

Errors

- Awk provides diagnostic error messages

```
awk '$3 == 0 [ print $1 }' emp.data
awk: syntax error near line 1
awk: bailing out near line 1
```

- Or if using nawk

```
nawk '$3 == 0 [ print $1 }' emp.data
nawk: syntax error at source line 1
   context is
   $3 == 0 >>>  [ <<<
      1 extra }
      1 extra [
nawk: bailing out at source line 1
      1 extra }
      1 extra [
```

The traditional advantage of an interpreter is that it tells exactly where it breaks down during the run. Compilers usually produce more optimised convoluted code & hence convoluted alerts!
NB Java bytecode is interpreted on a JVM so is not so bad and compiler and runtime environments have moved on.

Review of Awk Principles

- Awk is a filter, so it can

  Take input from / send output to

  – Files (file input arguments, and redirection)

  – and pipes

  – directly from standard input

Pattern-Action Pairs

- Both are optional, but at least one is required or empty line!
  - Default pattern is match every record
  - Default action is print record
- Patterns
  - BEGIN and END
  - expressions
    - $3 < 100
    - $4 == "Asia"
  - string-matching
    - /regex/ - /^.*$/
      - Matches any line…any number of any chars
    - string - abc
      - matches first occurrence of regex or string in line

Pattern-Action Pairs

- compound
  - $3 < 100 && $4 == “Asia”
    - && is a logical AND
    - || is a logical OR
- Range – checked , is for range, || for OR
  - NR == 10, NR == 20
    - matches records 10 through 20 inclusive
    - Note this is actually 11 lines not 10!
- Patterns can take any of these forms and for /regex/ and string patterns will match the first instance in the record

AWK’s extended regex manual extract – specifying characters

AWK uses extended regular expressions like egrep, with these metacharacters : ^ $. [ ] | ( ) * + ?

Regular expressions are built up from characters as follows:

c	matches any non-metacharacter c.
\c	matches a character defined by the same escape sequences used in string constants or the literal character c if \c is not an escape sequence.
.	matches any character (including newline).
^	matches the front of a string.
$	matches the back of a string.

[c1c2c3...] matches any character in the class c1c2c3... .
An interval of characters is denoted c1-c2 inside a class [...].
[^c1c2c3...] matches any character not in the class c1c2c3...

Compound regex

Regular expressions are built up …

… from other regular expressions as follows:

r1r2	matches r1 followed immediately by r2 (concatenation).
r1 | r2	matches r1 or r2 (alternation).
r*	matches r repeated zero or more times.
r+	matches r repeated one or more times.
r?	matches r zero or once.
(r)	matches r, providing grouping for e.g. : * + ?

The increasing precedence of operators is:-
alternation, concatenation and the following unary ops:- *, + or ?)

Regular Expressions in Awk

Awk uses the same regular expressions we’ve been using

^ $	- beginning of/end of line
.	- any character
[abcd]	- character class
[^abcd]	- negated character class
[a-z]	- range of characters
(regex1|regex2)	– alternation…i.e. (regex1 OR regex2)
*	- zero or more occurrences of preceding expression
+	- one or more occurrences of preceding expression
?	- zero or one occurrence of preceding expression

NOTE: the min max {m, n} or variations {m}, {m,} syntax may NOT be supported or occurrences of preceding expression (backreferences)

Examples

AWK identifiers /^[_a-zA-Z][_a-zA-Z0-9]*$/ and
AWK numbers /^[-+]?([0-9]+\.?|\.[0-9])[0-9]*([eE][-+]?[0-9]+)?$/

Note that . has to be escaped to be recognized as a decimal point, and that metacharacters are not special inside character classes. [ ]

Any expression can be used on the right hand side of the ~ or !~ operators or passed to a built-in that expects a regular expression. If needed, it is converted to string, and then interpreted as a regular expression. e.g.

```
BEGIN { identifier = "[_a-zA-Z][_a-zA-Z0-9]*" }
$0 ~ "^" identifier
```

prints all lines that start with an AWK identifier.

mawk matches the empty regular expression, //, to the empty string at the front, back and between every character: e.g., using echo, pipe & global substitution:- (compare: space insert **s/␣*/␣/g**; in L11_regex_1_tr (~slide 26) – next slide too!)

```
echo abc | mawk '{ gsub(//, "X") ; print }'
XaXbXcX
```

Awk Variables

- $0, $1, $2, $NF
- NR - Number of records processed
- FNR - Number of records processed in current file
- NF - Number of fields in current record
- FILENAME - name of current input file
- FS - Field separator, space or TAB by default
- OFS - Output field separator, space or TAB default
- ARGC/ARGV - Argument Count, Argument Value array (standard C & Unix approach…)
  - Used to get arguments from the command line

Another easy source for various text

info bash -o- | shuf -n50 | sed 's/␣*/␣/g; s/^␣//' | fmt -w 90

- bash documentation from info
  - is output to stdout using the 'o' flag for info -o- ,
- piped it to shuf which randomly shuffles 50 lines,
- sed then
  - replaces multiple spaces (including none) with one space,
    - so words are s p a c e d o u t ! ! !
    - **s/␣*/␣/g**; => results in one space between non-whitespace chars:-
      - inserting one space where there was none, since * includes zero occurrences
      - and squeezing many spaces, also caught by /␣*/ to one
    - **s/^␣//** strips the leading space at the start of a line
      - Whether remaining from multiple squeezed to 1
      - Or inserted where there was originally none
- finally **fmt -w 90** formats it to lines of max. width of 90.

Command Line Arguments

- Accessed via standard UNIX built-ins ARGC and ARGV
- ARGC : argument count of command line arguments
- ARGV[ ] : array of argument values

  e.g. for the command line

  $ awk 'script' filename
  - ARGC == 2
  - ARGV[0] == "awk"
  - ARGV[1] == "filename
  - Here the 'script' name is not considered an argument
- Demo, loop over number of arguments, omits scriptname:-

```
cs1> mawk '{ for (i=0;i<ARGC;i=i+1) print(ARGV[i]) }' test
mawk
test
```

- ARGC and ARGV can be used like any other variable
- The can be assigned, compared, used in expressions, printed
- They are commonly used for verifying that the correct number of arguments were provided
- To check for proper argument count in an awk script:

```
{if (ARGC != 2) print "I need a filename for this script"; exit}
```

- And from previous demo:-

```
cs1> mawk '{ for (i=0;i<ARGC;i=i+1) print(ARGV[i]) }' test
mawk
test
```


Useful "One(or so)-liners"

```
• END { print NR }          # print total number of lines/records
• NR == 10 { }              # do { } if there are exactly 10 lines
• { print $NF }             # print the last field of each line
• {field = $NF }            # set field equal to the last field
  END { print field }       # print the last field...
• NF > 4{ }                 # do { } if more than 4 fields
• $NF > 4    { }            # do { } if the last field > 4?
• BEGIN {nf = 0}            # initialise nf to zero
   { nf = nf + NF }          # accumulate field count, line by line
  END { print nf }          # print total field count from all lines
```

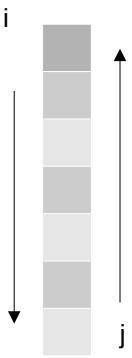
NB... IF NO action is specified (i.e. no { }) then print is the default
... But IF an action is specified, (i.e. something within { })
... then it won't print as well unless it is told to!

Example function written in AWK to sort lines of a file alphabetically

```
{ line[NR] = $0 "" }  # store entire line $0 in array line[line number] NR=Record No.
END {  ssort(line, NR)
for(i = 1 ; i <= NR ; i++) print line[i]   }
```

No whitespace between '(' and function name to avoid ambiguity with concatenation...see manual..

```
#insertion sort by sinking sort
Function ssort( A, n, i, j, hold)
{
      for( i = 2 ; i <= n ; i++)
      {
            hold = A[j = i]
            while ( A[j-1] > hold )
            { j-- ; A[j+1] = A[j] }
            A[j] = hold
      }
# sentinel A[0] = "" will be created if needed
}
```



Presented from the manual as an illustration of AWK programming showing its power and similarity to C with functions and syntax, for those who might be interested.

It's a Bubblesort

It works through the list, from the first to last, index i (using the for loop)
Compares each new element hold
And if lighter has hold bubble back towards the start of the array (using the while loop – index j)
Until it's position in the already sorted part of the array is found.

Sort – just use the bash command

- Sort a file numerically or alphabetically based on the beginning of each line **unless** fields are specified
- Syntax: *sort [-dfnr] [-o filename] [filename(s)]*
  - *d* - Dictionary order, only letters, digits, and whitespace are significant in determining sort order
  - *f* - Ignore case (fold in lower case)
  - *n* - Numeric order, sort by arithmetic value instead of first digit
  - *r* - Sort in reverse order
  - *o filename* - write output to filename, filename can be the same as one of the input files

Don't MAWK AWK

- Fast elegant big data munging language - from 2009 – recent for awk
- NB Ubuntu runs older versions of mawk… ok for us, but not Big Data!

Language	Time (min:sec)	Speed (vs. gawk)	Lines of code	Notes	Type
mawk	1:06	7.8x	3	Mike Brennan's Awk, system default on Ubuntu/Debian Linux.	VM
java	1:20	6.4x	32	version 1.6 (-server didn't matter)	VM+JIT
c-ish c++	1:35	5.4x	42	g++ 4.0.1 with -O3, using stdio.h	Native
python	2:15	3.8x	20	version 2.5, system default on OSX 10.5	VM
perl	3:00	2.9x	17	version 5.8.8, system default on OSX 10.5	VM
nawk	6:10	1.4x	3	Brian Kernighan's "One True Awk", system default on OSX, *BSD	?
c++	6:50	1.3x	48	g++ 4.0.1 with -O3, using fstream, stringstream	Native
ruby	7:30	1.1x	22	version 1.8.4, system default on OSX 10.5; also tried 1.9, but was slower	Interpreted
gawk	8:35	1x	3	GNU Awk, system default on RedHat/Fedora Linux	Interpreted

Is AWK still relevant – is air still relevant!?

- May 2020 – see https://lwn.net/Articles/820829/
- Linux kernel : used to check & reformat objdump files
  - Hard to kill : planned to use Python scripts in 2019 but none bothered
  - It's so embedded everywhere : hard to dump
- Neovim : generates documentation
- Ffmpeg : building and testing
- case for systems programming : between C and shells
  - Apps include
    - Literate programming tool
    - Automatic documentation generators from comments
    - CLI interface to cloud-based translation APIs
- Claims simpler and faster than Spark & Hadoop
  - e.g. 235 x speedup over Hadoop Cluster (cat, mawk, etc.)
  - Distributed processing of distributed big data…access time penalty
    - big data => big cost – bigger comms, time & $$ too!
- Clouds sell spare capacity when not urgently needed by the host!

AWK before sed or the other way around?

- AWK before sed
  - May seem an easier path to sed & editors,
  - but interrupts logical sequence
- AWK can be used in place of sed
  - gsub(r,s,t) gsub(r,s) : Global substitution,
    every match of regular expression r in variable t is replaced by string s. The number of replacements is returned. If t is omitted, $0 is used. An & in the replacement string s is replaced by the matched substring of t. \& and \\ put literal & and \, respectively, in the replacement string.
  - sub(r,s,t) sub(r,s) : Single substitution,
    same as gsub() except at most one substitution.
- Has lots of other string and numeric functions