Lecture 5

Awk and Shell

Sed Drawbacks

- Hard to remember text from one line to another
- Not possible to go backward in the file
- No way to do forward references like **`/..../+1`**
- No facilities to manipulate numbers
- Cumbersome syntax

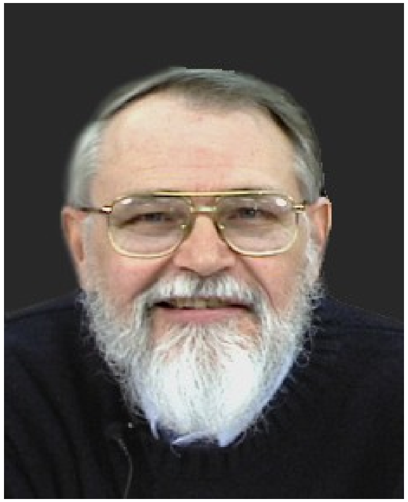
Awk

Programmable Filters

Why is it called AWK?

Aho

Weinberger



Kernighan

Awk Introduction

- **awk**'s purpose: A general purpose programmable filter that handles text (strings) as easily as numbers
  - This makes **awk** one of the most powerful of the Unix utilities
- **awk** processes *fields* while **sed** only processes lines
- **nawk** (new **awk**) is the new standard for **awk**
  - Designed to facilitate large **awk** programs
  - **gawk** is a free **nawk** clone from GNU
- **awk** gets it’s input from
  - files
  - redirection and pipes
  - directly from standard input

AWK Highlights

- A programming language for handling common data manipulation tasks with only a few lines of code
- **awk** is a *pattern-action* language, like **sed**
- 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

Structure of an AWK Program

- An **awk** program consists of:
  - An optional BEGIN segment
    - For processing to execute prior to reading input
  - pattern - action pairs
    - Processing for input data
    - For each pattern matched, the corresponding action is taken
  - An optional END segment
    - Processing after end of input data

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

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!)
  - ***awk -f program_file input_files***
    - program is read from a file

Patterns and Actions

- Search a set of files for *patterns*.
- Perform specified *actions* upon lines or fields that contain instances of patterns.
- Does not alter input files.
- Process one input line at a time
- This is similar to **sed**

Pattern-Action Structure

- Every program statement has to have a *pattern* **or** an *action* **or** both
- Default *pattern* is to match all lines
- Default *action* is to print current record
- Patterns are simply listed; actions are enclosed in **`{ }`**
- **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

Patterns

- Selector that determines whether *action* is to be executed
- *pattern* can be:
  - the special token **BEGIN** or **END**
  - regular expressions (enclosed with //)
  - arithmetic relation operators
  - string-valued expressions
  - arbitrary combination of the above
    - **`/NYU/`** matches if the string “NYU” is in the record
    - **`x > 0`** matches if the condition is true
    - **`/NYU/ && (name == "UNIX Tools")`**

BEGIN and END patterns

- **BEGIN** and **END** provide a way to gain control before and after processing, for initialization and wrap-up.
  - **BEGIN**: actions are performed before the first input line is read.
  - **END**: actions are done after the last input line has been processed.

Actions

- *action* may include a list of one or more C like statements, as well as arithmetic and string expressions and assignments and multiple output streams.
- *action* is performed on every line that matches *pattern*.
  - If *pattern* is not provided, *action* is performed on every input line
  - If *action* is not provided, all matching lines are sent to standard output.
- Since *patterns* and *actions* are optional, *actions* must be enclosed in braces to distinguish them from *pattern*.

An Example

```
ls | awk '
  BEGIN { print "List of html files:" }
  /\.html$/ { print }
  END { print "There you go!" }
  '
```

```
List of html files:
index.html
as1.html
as2.html
There you go!
```

Variables

- **awk** scripts can define and use variables

```
BEGIN { sum = 0 }
{ sum ++ }
END { print sum }
```

- Some variables are predefined

Records

- Default record separator is **newline**
  - By default, **awk** processes its input a line at a time.
- Could be any other *regular expression*.
- **RS**: record separator
  - Can be changed in **BEGIN** action
- **NR** is the variable whose value is the number of the current record.

Fields

- Each input line is split into fields.
  - **FS**: field separator: default is whitespace (1 or more spaces or tabs)
  - **`awk -F`***c* option sets **FS** to the character *c*
    - Can also be changed in BEGIN
  - **$0** is the entire line
  - **$1** is the first field, **$2** is the second field, ….
- Only fields begin with **$**, variables are unadorned

Simple Output From AWK

- Printing Every Line
  - If an action has no pattern, the action is performed to all input lines
    - **`{ print }`** will print all input lines to standard out
    - **`{ 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

Output (continued)

- **NF**, the Number of Fields
  - Any valid expression can be used after a **`$`** to indicate the contents of a particular field
  - One built-in expression is **NF**, or Number of Fields
  - **`{ print NF, $1, $NF }`** will print the number of fields, the first field, and the last field in the current record
  - **`{ print $(NF-2) }`** prints the third to 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 }`**

Output (continued)

- 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
  - *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 ) }
```

Selection

- Awk patterns are good for selecting specific lines from the input for further processing
  - Selection by Comparison
    - `$2 >= 5 { print }`
  - Selection by Computation
    - `$2 * $3 > 50 { printf(“%6.2f for %s\n”, $2 * $3, $1) }`
  - Selection by Text Content
    - `$1 == "NYU"`
    - `/NYU/`
  - Combinations of Patterns
    - `$2 >= 4 || $3 >= 20`
  - Selection by Line Number
    - `NR >= 10 && NR <= 20`

Arithmetic and variables

- **awk** variables take on numeric (floating point) or string values according to context.
- User-defined variables do not need to be declared.
- By default, user-defined variables are initialized to the null string which has numerical value 0.

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
    }
```

Handling Text

- One major advantage of Awk is its ability to handle strings as easily as many languages handle numbers
- Awk variables can hold strings of characters as well as numbers, and Awk conveniently translates back and forth as needed
- This program finds the employee who is paid the most per hour:

```
# Fields: employee, payrate
$2 > maxrate { maxrate = $2; maxemp = $1 }
END { print “highest hourly rate:”,
        maxrate, “for”, maxemp }
```

String Manipulation

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

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

- 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

- **awk** contains a number of built-in functions. length is one of them.
- Counting Lines, Words, and Characters using length (a poor man’s **wc**)

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

- **substr(s, m, n)** produces the substring of *s* that begins at position *m* and is at most *n* characters long.

Control Flow Statements

- **awk** provides several control flow statements for making decisions and writing loops
- If-Then-Else

```
  $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 paid more
  than $6/hour"
    }
```

Loop Control

- While

```
# interest1 - compute compound interest
#   input: amount, rate, years
#   output: compound value at end of each year
{  i = 1
  while (i <= $3) {
           printf(“\t%.2f\n”, $1 * (1 + $2) ^ i)
           i = i + 1
  }
}
```

Do-While Loops

- Do While

do {

statement1

}

while (expression)

For statements

- For

```
# interest2 - compute compound interest
#   input: amount, rate, years
#   output: compound value at end of each year

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

Arrays

- Array elements are not declared
- Array subscripts can have ***any*** value:
  - Numbers
  - Strings! (***associative arrays***)
- Examples
  - `arr[3]="value"`
  - `grade["Korn"]=40.3`

Array Example

```
# reverse - print input in reverse order by line

{ line[NR] = $0 }    # remember each line

END {
              for (i=NR; (i > 0); i=i-1) {
                  print line[i]
              }
    }
```

Useful One (or so)-liners

- `END { print NR }`
- `NR == 10`
- `{ print $NF }`
- ```
  { field = $NF }
  END { print field }
  ```
- `NF > 4`
- `$NF > 4`
- ```
  { nf = nf + NF }
  END { print nf }
  ```

More One-liners

- **/Jeff/ { nlines = nlines + 1 }**
  END { print nlines }
- **$1 > max { max = $1; maxline = $0 }**
  END { print max, maxline }
- **NF > 0**
- **length($0) > 80**
- **{ print NF, $0}**
- **{ print $2, $1 }**
- **{ temp = $1; $1 = $2; $2 = temp; print }**
- **{ $2 = ""; print }**

Even More One-liners

- ```
  { for (i = NF; i > 0; i = i - 1)
  printf(“%s “, $i)
      printf(“\n”)
  }
  ```
- ```
  { sum = 0
      for (i = 1; i <= NF; i = i + 1)
  sum = sum + $i
      print sum
   }
  ```
- ```
  { for (i = 1; i <= NF; i = i + 1)
       sum = sum $i }
      END { print sum }
  }
  ```
  ```

Awk Variables

- $0, $1, $2, $NF
- NR - Number of records processed
- 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 by default
- ARGC/ARGV - Argument Count, Argument Value array
  - Used to get arguments from the command line

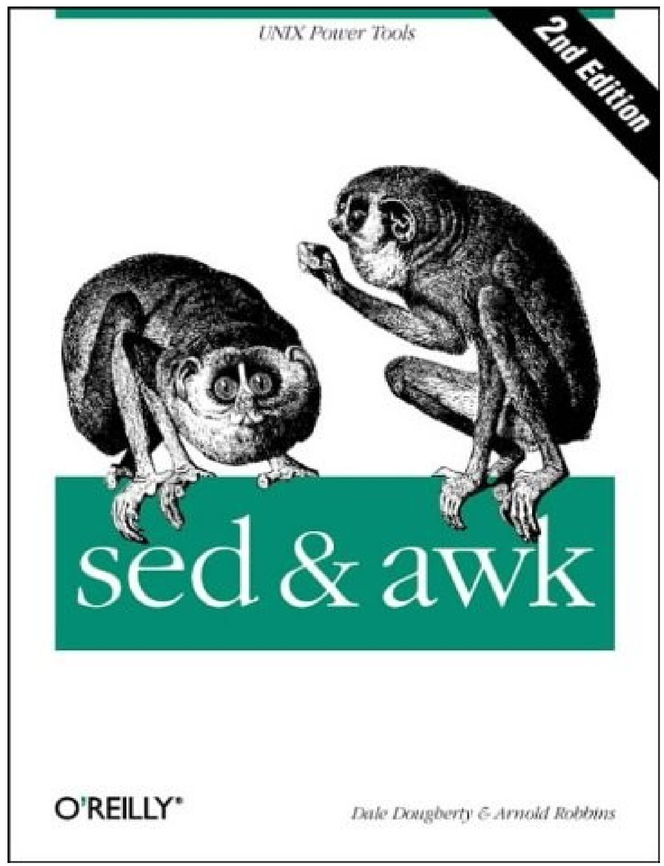
Operators

- = assignment operator; sets a variable equal to a value or string
- == equality operator; returns TRUE is both sides are equal
- != inverse equality operator
- && logical AND
- || logical OR
- ! logical NOT
- <, >, <=, >= relational operators
- +, -, /, *, %, ^
- String concatenation

Built-In Functions

- Arithmetic
  - **sin**, **cos**, **atan**, **exp**, **int**, **log**, **rand**, **sqrt**
- String
  - **length**, **substitution**, find substrings, split strings
- Output
  - **print**, **printf**, 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

More Information



on the website

Lecture 5

Shell Scripting

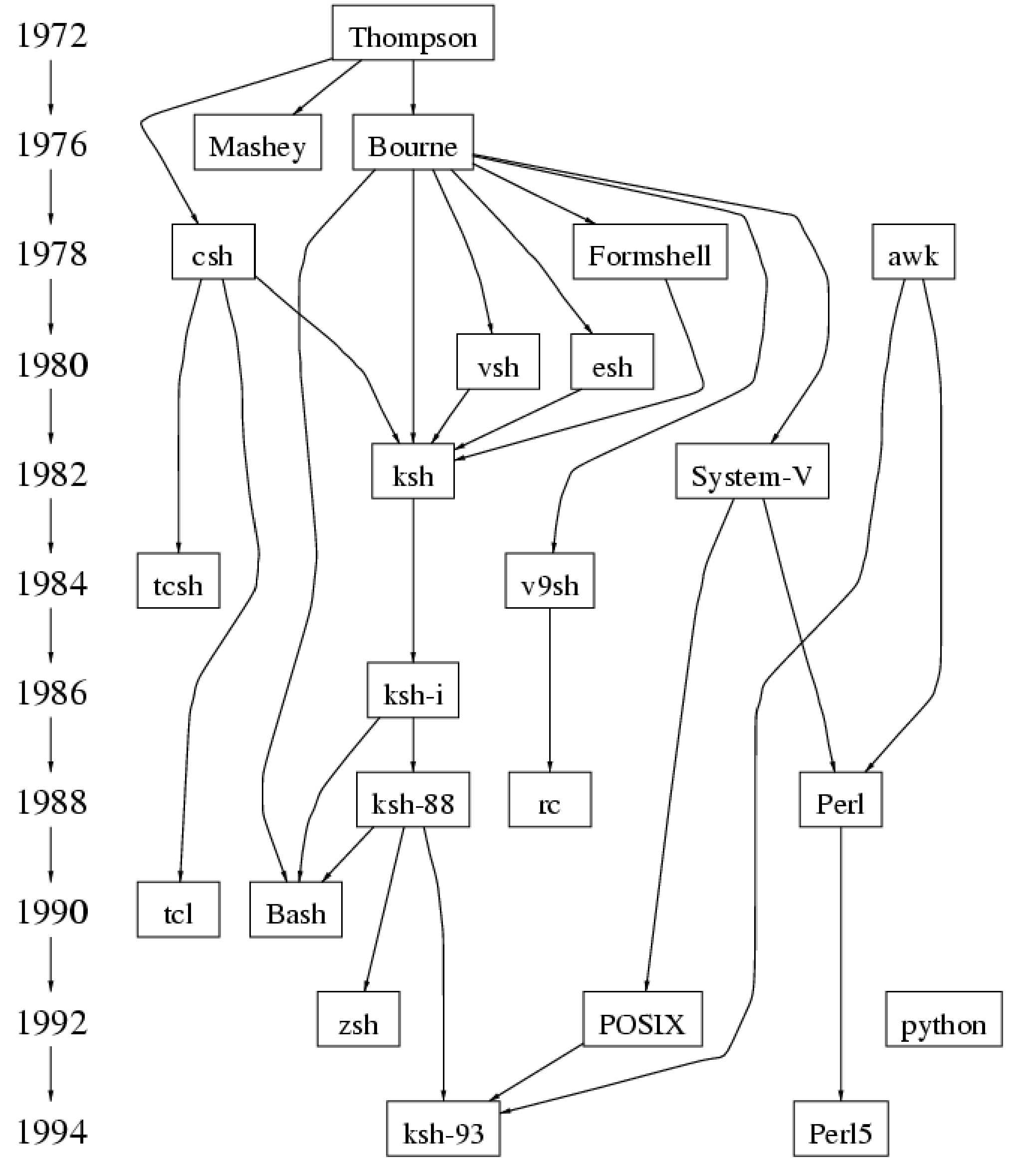
What is a shell?

- The user interface to the operating system
- Functionality:
  - Execute other programs
  - Manage files
  - Manage processes
- Full programming language
- A program like any other
  - This is why there are so many shells

Shell History

- There are many choices for shells

- Shell features evolved as UNIX grew



Shell Scripts

- A shell script is a regular text file that contains shell or UNIX commands
  - Before running it, it must have execute permission:
    - **chmod +x *filename***
- A script can be invoked as:
  - **ksh name *[arg …]***
  - **ksh < name *[args …]***
  - **name *[arg …]***

Shell Scripts

- When a script is run, the **kernel** determines which shell it is written for by examining the first line of the script
  - If 1st line starts with **`#!`*`pathname-of-shell`***, then it invokes *pathname* and sends the script as an argument to be interpreted
  - If **`#!`** is not specified, the current shell assumes it is a script in its own language
    - leads to problems

Simple Example

```
#!/bin/sh

echo Hello World
```

Scripting vs. C Programming

- Advantages of shell scripts
  - Easy to work with other programs
  - Easy to work with files
  - Easy to work with strings
  - Great for prototyping. No compilation
- Disadvantages of shell scripts
  - Slow
  - Not well suited for algorithms & data structures

The C Shell

- C-like syntax (uses `{ }`'s)
- **Inadequate for scripting**
  - Poor control over file descriptors
  - Can't mix flow control and commands
  - Difficult quoting `"I say \"hello\""` doesn't work
  - Can only trap SIGINT
- Survives mostly because of interactive features.
  - Job control
  - Command history
  - Command line editing, with arrow keys (**tcsh**)

The Bourne Shell

- Slight differences on various systems
- Evolved into standardized POSIX shell
- Scripts will also run with **ksh**, **bash**
- Influenced by ALGOL

Simple Commands

- ***simple command***: sequence of non blanks arguments separated by blanks or tabs.
- 1st argument (numbered zero) usually specifies the name of the command to be executed.
- Any remaining arguments:
  - Are passed as arguments to that command.
  - Arguments may be filenames, pathnames, directories or special options

Complex Commands

- The shell's power is in its ability to hook commands together
- We've seen one example of this so far with pipelines:

```
cut –d: -f2 /etc/passwd | sort | uniq
```

- We will see others

Redirection of input/ouput

- Redirection of output: >
  - example:`$ ls -l > my_files`
- Redirection of input: <
  - example: `$ cat <input.data`
- Append output: >>
  - example: `$ date >> logfile`
- Arbitrary file descriptor redirection: *fd*>
  - example: `$ ls –l 2> error_log`

Multiple Redirection

- **`cmd 2>file`**
  - send standard error to file
  - standard output remains the same
- **`cmd > file 2>&1`**
  - send both standard error and standard output to file
- **`cmd > file1 2>file2`**
  - send standard output to file1
  - send standard error to file2

Here Documents

- Shell provides alternative ways of supplying standard input to commands (an *anonymous file*)
- Shell allows in-line input redirection using << called here documents
- **<u>format</u>**

```
command [arg(s)] << arbitrary-delimiter
command input
  :
  :
arbitrary-delimiter
```

- `arbitrary-delimiter` should be a string that does not appear in text

Here Document Example

```
#!/bin/sh

mail steinbrenner@yankees.com <<EOT
  You guys really blew it in
  yesterday.  Good luck tomorrow.
  Yours,
  $USER
  EOT
```

Shell Variables

- Write

`name=value`

- Read: **`$var`**

- Turn local variable into environment:

    `export variable`

Variable Example

```
#!/bin/sh

MESSAGE="Hello World"
echo $MESSAGE
```

Environmental Variables

NAME	MEANING
$HOME	Absolute pathname of your home directory
$PATH	A list of directories to search for
$MAIL	Absolute pathname to mailbox
$USER	Your login name
$SHELL	Absolute pathname of login shell
$TERM	Type of your terminal
$PS1	Prompt

Parameters

- A parameter is one of the following:
  - A variable
  - A *positional parameter*, starting at 1 (next slide)
  - A *special* parameter
- To get the value of a parameter: **`${param}`**
  - Can be part of a word (**`abc${foo}def`**)
  - Works in double quotes
- The **`{}`** can be omitted for simple variables, special parameters, and single digit positional parameters.

Positional Parameters

- The arguments to a shell script
  - **`$1, $2, $3 …`**
- The arguments to a shell function
- Arguments to the **set** built-in command
  - **`set this is a test`**
    - **$1=this, $2=is, $3=a, $4=test**
- Manipulated with **shift**
  - **`shift 2`**
    - **$1=a, $2=test**
- Parameter 0 is the name of the shell or the shell script.

Example with Parameters

```
#!/bin/sh

# Parameter 1: word
# Parameter 2: file
grep $1 $2 | wc –l
```

```
$ countlines ing /usr/dict/words
3277
```

Special Parameters

- **$#** Number of positional parameters
- **$-** Options currently in effect
- **$?** Exit value of last executed command
- **$$** Process number of current process
- **$!** Process number of background process
- **$*** All arguments on command line
- **"$@"** All arguments on command line individually quoted **"$1" "$2" ...**

Command Substitution

- Used to turn the output of a command into a string
- Used to create arguments or variables
- Command is placed with grave accents **\`  \`** to capture the output of command

```
$ date
Wed Sep 25 14:40:56 EDT 2001
$ NOW=`date`

$ sed "s/oldtext/`ls | head -1`/g"

$ PATH=`myscript`:$PATH
$ grep `generate_regexp` myfile.c
```

File name expansion

- Wildcards (patterns)

*	matches any string of characters

?	matches any single character

[**list**] matches any character in **list**

[lower-upper] matches any character in range **lower-upper** inclusive

[!list] matches any character not in list

File Expansion

- If multiple matches, all are returned and treated as separate arguments:

```
$ /bin/ls
file1 file2
$ cat file1
a
$ cat file2
b
$ cat file*
a
b
```

- Handled by the shell (*exec never sees the wildcards*)
  - argv[0]: /bin/cat
  - argv[1]: file1
  - argv[2]: file2

NOT

  - argv[0]: /bin/cat
  - argv[1]: file*

Compound Commands

- Multiple commands
  - Separated by semicolon
- Command groupings
  - pipelines
- Boolean operators
- Subshell
  - `( command1; command2 ) > file`
- Control structures

Boolean Operators

- Exit value of a program (**exit** system call) is a number
  - 0 means success
  - anything else is a failure code
- *cmd1* **&&** *cmd2*
  - executes cmd2 if cmd1 is successful
- *cmd1* **||** *cmd2*
  - executes cmd2 if cmd1 is not successful

```
$ ls bad_file > /dev/null && date
$ ls bad_file > /dev/null || date
Wed Sep 26 07:43:23 2001
```

Control Structures

```
if expression
then
    command1
else
    command2
fi
```

What is an expression?

- Any UNIX command. Evaluates to true if the exit code is 0, false if the exit code > 0
- Special command **`/bin/test`** exists that does most common expressions
  - String compare
  - Numeric comparison
  - Check file properties
- **`/bin/[`** is linked to **`/bin/test`** for syntactic sugar
- Good example UNIX tools working together

Examples

```
if test "$USER" = "kornj"
then
      echo "I hate you"
else
      echo "I like you"
fi
```

```
if [ -f /tmp/stuff ] && [ `wc –l < /tmp/stuff` -gt 10 ]

then
      echo "The file has more than 10 lines in it"
else
      echo "The file is nonexistent or small"
fi
```

test Summary

- **String based tests**

`-z string`	Length of string is 0
`-n string`	Length of string is not 0
`string1 = string2`	Strings are identical
`string1 != string2`	Strings differ
`string`	String is not NULL

- **Numeric tests**

`int1 –eq int2`	First int equal to second
`int1 –ne int2`	First int not equal to second
`-gt`, `-ge`, `-lt`, `-le`	greater, greater/equal, less, less/equal

- **File tests**

`-r file`	File exists and is readable
`-w file`	File exists and is writable
`-f file`	File is regular file
`-d file`	File is directory
`-s file`	file exists and is not empty

- **Logic**

`!`	Negate result of expression
`-a`, `-o`	and operator, or operator
`( expr )`	groups an expression

Control Structures Summary

- **`if … then … fi`**
- **`while … done`**
- **`until … do … done`**
- **`for … do … done`**
- **`case … in … esac`**