Introduction to Awk

ComS 252 — Iowa State University

Andrew Miner

Motivation

Suppose we want to list files owned by `bob`

Motivation

Suppose we want to list files owned by `bob`

First cut: `ls -l | grep bob`

- Good: Easy to remember
- Good: Will show files owned by `bob`

Motivation

Suppose we want to list files owned by `bob`

First cut: `ls -l | grep bob`

- Good: Easy to remember
- Good: Will show files owned by `bob`
- Bad: might also show files not owned by `bob`
  - E.g., a file named `bob` or `turboboost.wav`
  - E.g., a file with *group* `bob`

Motivation

Suppose we want to list files owned by `bob`

First cut: `ls -l | grep bob`

- Good: Easy to remember
- Good: Will show files owned by `bob`
- Bad: might also show files not owned by `bob`
  - E.g., a file named `bob` or `turboboost.wav`
  - E.g., a file with *group* `bob`

There may be "false positives". So what?

Motivation

Suppose we want to list files owned by `bob`

First cut: `ls -l | grep bob`

- Good: Easy to remember
- Good: Will show files owned by `bob`
- Bad: might also show files not owned by `bob`
  - E.g., a file named `bob` or `turboboost.wav`
  - E.g., a file with *group* `bob`

There may be “false positives”. So what?

- If this output is for human consumption — no big deal

Motivation

Suppose we want to list files owned by `bob`

First cut: `ls -l | grep bob`

- Good: Easy to remember
- Good: Will show files owned by `bob`
- Bad: might also show files not owned by `bob`
  - E.g., a file named `bob` or `turboboost.wav`
  - E.g., a file with *group* `bob`

There may be “false positives”. So what?

- If this output is for human consumption — no big deal
- But what if this is inside a pipeline or script?

Motivation

Suppose we want to list files owned by `bob`

First cut: `ls -l | grep bob`

- Good: Easy to remember
- Good: Will show files owned by `bob`
- Bad: might also show files not owned by `bob`
    - E.g., a file named `bob` or `turboboost.wav`
    - E.g., a file with *group* `bob`

There may be "false positives". So what?

- If this output is for human consumption — no big deal
- But what if this is inside a pipeline or script?
    - Bad — might remove / backup / process an incorrect file

Motivation

Suppose we want to list files owned by `bob`

First cut: `ls -l | grep bob`

- Good: Easy to remember
- Good: Will show files owned by `bob`
- Bad: might also show files not owned by `bob`
  - E.g., a file named `bob` or `turboboost.wav`
  - E.g., a file with *group* `bob`

There may be "false positives". So what?

- If this output is for human consumption — no big deal
- But what if this is inside a pipeline or script?
  - Bad — might remove / backup / process an incorrect file

We need something stronger than `grep`

AWK

- Small scripting language
  - POSIX now specifies a standard for the language
  - Programs are often very short (and cryptic), e.g.:

    ```
    $3 ~ /bob/ {print $9}
    ```

    You will understand this program by the end of lecture
- Named for its inventors
  - Aho, Weinberger, Kernighan
  - The same Kernighan of “Kernighan and Ritchie” C
- Great for editing streams
- There are multiple implementations of the AWK language
  - This lecture uses a generic “awk”
- Often used in pipelines
  - E.g., `crazy | pipeline | awk ... | other | things`

AWK input stream

AWK assumes the input stream is structured as follows:

- The input files are divided into one or more records
  - Default: each line of a file is a record
  - The "record separator" may be changed (default is "newline character")
- Each record is divided into one or more fields
  - The number of fields may be different, for each record
  - Default: fields are separated by "whitespace"
  - The "field separator" may be changed

AWK input stream

AWK assumes the input stream is structured as follows:

- The input files are divided into one or more records
    - Default: each line of a file is a record
    - The "record separator" may be changed (default is "newline character")
- Each record is divided into one or more fields
    - The number of fields may be different, for each record
    - Default: fields are separated by "whitespace"
    - The "field separator" may be changed

You can imagine the input stream as a table

- Rows of the table are records
- Columns of the table are fields

Running AWK programs

```
awk 'program' file1 ... filen
```

- Pass the entire program as the first argument
  - Programs can be short, remember?
  - Single quotes: otherwise need escapes
- The remaining arguments: input files
  - Processed by the program
  - If none: reads from standard input

```
awk -f progfile file1 ... filen
```

- Use `-f` to read the program from `progfile`
  - Good for complex, multi–line programs

AWK scripts

- # is a comment line in AWK programs. So. . .
- We can make an AWK script

AWK scripts

- # is a comment line in AWK programs. So. . .
- We can make an AWK script

For example, if we have an executable text file:

progfile

```
#!/usr/bin/awk -f
# AWK program here
```

then running

```
prompt$ ./progfile file
```

is the same as

```
prompt$ /usr/bin/awk -f ./progfile file
```

AWK programs

- Programs follow a different model than “general” languages
- Programs operate on the input files
    - One record at a time
- AWK programs are a sequence of statements
- Statements specify
    - Which records they apply to
    - Instructions to execute for those records

AWK statements

Generic syntax

which-records { instructions }

- For the cryptic 1–line example: "`$3 ~ /bob/ {print $9}`"

  `$3 ~ /bob/` specifies which records

  `{print $9}` says what to do with the matching records
- "which-records" is optional
  - Default is: apply to all records
- "instructions" are optional, if "which-records" is given
  - Default is: print the record
- To be a proficient AWK programmer, need to know
  - How to specify "which-records"
  - What instructions can be written

Field specifiers

$1 : field 1 of the current record

$2 : field 2 of the current record

⋮

$9 : field 9 of the current record

Field specifiers

$1 : field 1 of the current record

$2 : field 2 of the current record

⋮

$9 : field 9 of the current record

$10 : field 10 of the current record

$11 : field 11 of the current record

⋮

Field specifiers

`$1` : field 1 of the current record

`$2` : field 2 of the current record

⋮

`$9` : field 9 of the current record

`$10` : field 10 of the current record

`$11` : field 11 of the current record

⋮

`$0` :

Field specifiers

$1 : field 1 of the current record

$2 : field 2 of the current record

⋮

$9 : field 9 of the current record

$10 : field 10 of the current record

$11 : field 11 of the current record

⋮

$0 : the entire record

`print` instruction

`print item item item ...`

- Prints text (to standard output)
- Items to print are concatenated
- No items? Prints the current record

What are possible items?

- Fields
  - E.g., `print $9`
    prints field 9 of the current record
- Literal strings, in double quotes
  - E.g, `print "Hello world!"`
    prints "Hello, world!"
- Others . . .

Example AWK program

```
{print "Hello, world!"}
```

What does this do?

Example AWK program

```
{print "Hello, world!"}
```

What does this do?

- What records are selected?

Example AWK program

```
{print "Hello, world!"}
```

What does this do?

- What records are selected?
  - All records

Example AWK program

```
{print "Hello, world!"}
```

What does this do?

- What records are selected?
    - All records
- What do we do for each record?

Example AWK program

```
{print "Hello, world!"}
```

What does this do?

- What records are selected?
  - All records
- What do we do for each record?
  - Print the string “Hello, world!”

Example AWK program

```
{print "Hello, world!"}
```

What does this do?

- What records are selected?
  - All records
- What do we do for each record?
  - Print the string “Hello, world!”

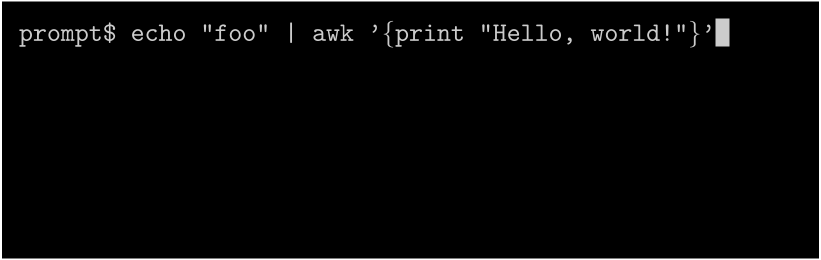
```
prompt$ █
```

Example AWK program

```
{print "Hello, world!"}
```

What does this do?

- What records are selected?
  - All records
- What do we do for each record?
  - Print the string “Hello, world!”



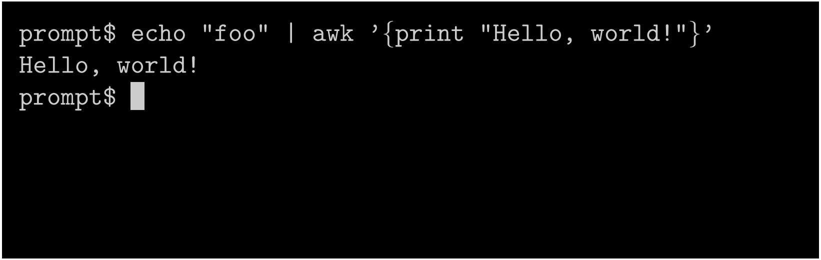
```
prompt$ echo "foo" | awk '{print "Hello, world!"}'
```

Example AWK program

```
{print "Hello, world!"}
```

What does this do?

- What records are selected?
  - All records
- What do we do for each record?
  - Print the string “Hello, world!”



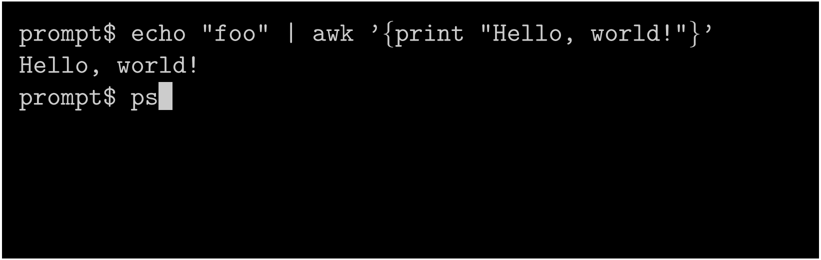
```
prompt$ echo "foo" | awk '{print "Hello, world!"}'
Hello, world!
prompt$ █
```

Example AWK program

```
{print "Hello, world!"}
```

What does this do?

- What records are selected?
    - All records
- What do we do for each record?
    - Print the string “Hello, world!”



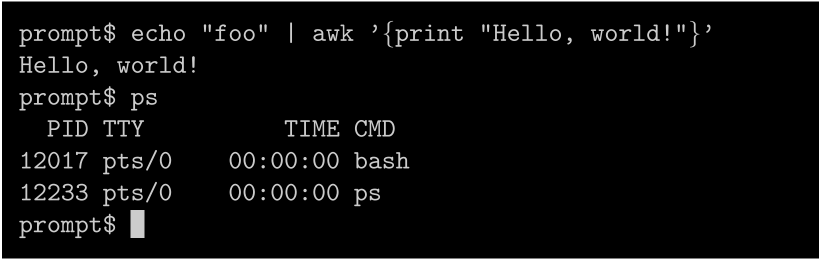
```
prompt$ echo "foo" | awk '{print "Hello, world!"}'
Hello, world!
prompt$ ps
```

Example AWK program

```
{print "Hello, world!"}
```

What does this do?

- What records are selected?
  - All records
- What do we do for each record?
  - Print the string “Hello, world!”



```
prompt$ echo "foo" | awk '{print "Hello, world!"}'
Hello, world!
prompt$ ps
  PID TTY          TIME CMD
12017 pts/0    00:00:00 bash
12233 pts/0    00:00:00 ps
prompt$ █
```

Example AWK program

```
{print "Hello, world!"}
```

What does this do?

- What records are selected?
  - All records
- What do we do for each record?
  - Print the string “Hello, world!”

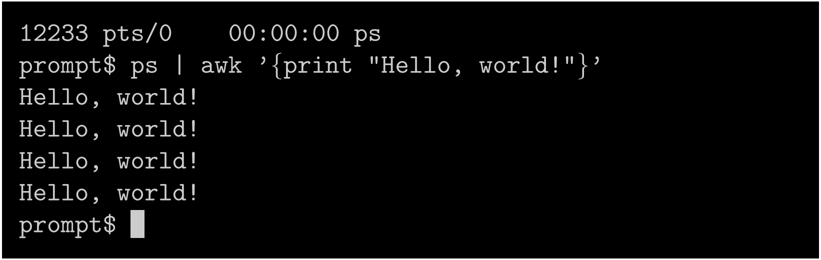
```
prompt$ echo "foo" | awk '{print "Hello, world!"}'
Hello, world!
prompt$ ps
  PID TTY          TIME CMD
12017 pts/0    00:00:00 bash
12233 pts/0    00:00:00 ps
prompt$ ps | awk '{print "Hello, world!"}'
```

Example AWK program

```
{print "Hello, world!"}
```

What does this do?

- What records are selected?
  - All records
- What do we do for each record?
  - Print the string “Hello, world!”



```
12233 pts/0    00:00:00 ps
prompt$ ps | awk '{print "Hello, world!"}'
Hello, world!
Hello, world!
Hello, world!
Hello, world!
prompt$ █
```

Another example AWK program

```
{print "Process " $1 " is " $4 "."}
```

What does this do?

Another example AWK program

```
{print "Process " $1 " is " $4 "."}
```

What does this do?

- For every record,
- Print. . .

Another example AWK program

```
{print "Process " $1 " is " $4 "."}
```

What does this do?

- For every record,
- Print. . .

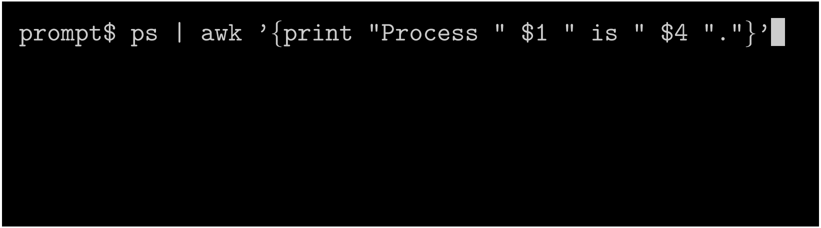
```
prompt$ █
```

Another example AWK program

```
{print "Process " $1 " is " $4 "."}
```

What does this do?

- For every record,
- Print...



```
prompt$ ps | awk '{print "Process " $1 " is " $4 "."}'
```

Another example AWK program

```
{print "Process " $1 " is " $4 "."}
```

What does this do?

- For every record,
- Print. . .

```
prompt$ ps | awk ’{print "Process " $1 " is " $4 "."}’
Process PID is CMD.
Process 12017 is bash.
Process 12237 is ps.
Process 12238 is awk.
prompt$ █
```

Using a program file

prog1.awk

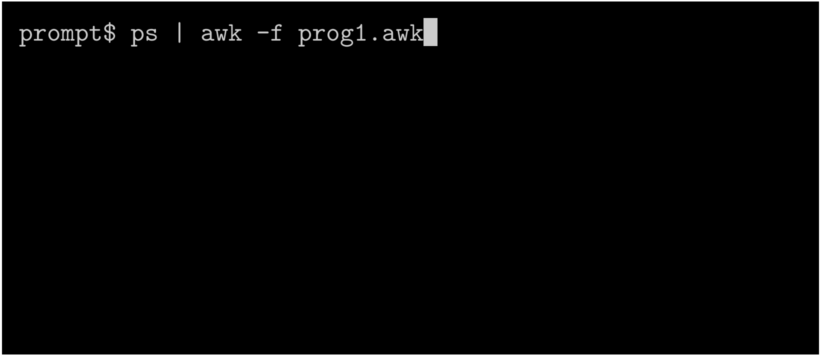
```
{ print "Process " $1 " is " $4 "." }
```

```
prompt$ 
```

Using a program file

prog1.awk

```
{ print "Process " $1 " is " $4 "." }
```

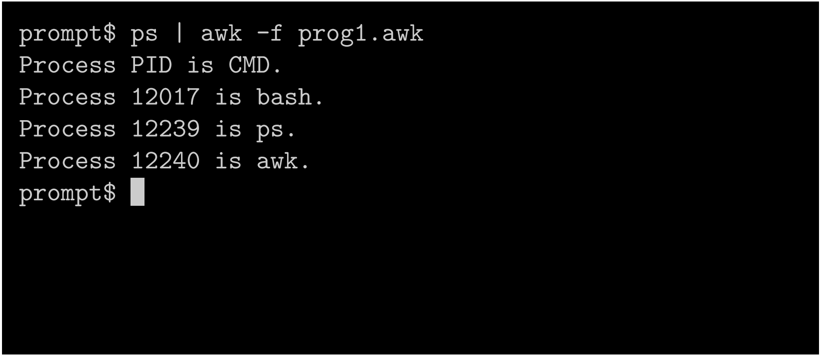


```
prompt$ ps | awk -f prog1.awk
```

Using a program file

prog1.awk

```
{ print "Process " $1 " is " $4 "." }
```

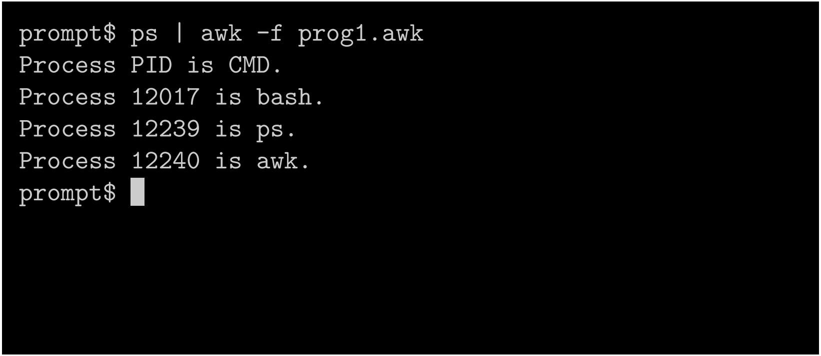


```
prompt$ ps | awk -f prog1.awk
Process PID is CMD.
Process 12017 is bash.
Process 12239 is ps.
Process 12240 is awk.
prompt$ 
```

Using a program file

prog2.awk

```
{ print "Process " $1 " is " $4 "."
  print "    And it is running on terminal " $2
}
```

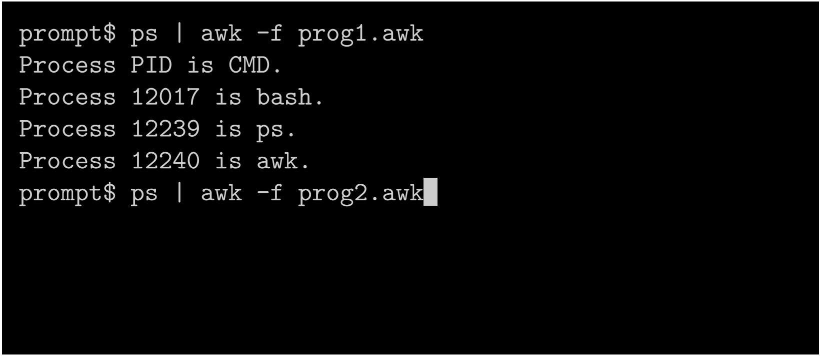


```
prompt$ ps | awk -f prog1.awk
Process PID is CMD.
Process 12017 is bash.
Process 12239 is ps.
Process 12240 is awk.
prompt$ █
```

Using a program file

prog2.awk

```
{ print "Process " $1 " is " $4 "."
  print "    And it is running on terminal " $2
}
```



```
prompt$ ps | awk -f prog1.awk
Process PID is CMD.
Process 12017 is bash.
Process 12239 is ps.
Process 12240 is awk.
prompt$ ps | awk -f prog2.awk
```

Using a program file

prog2.awk

```
{ print "Process " $1 " is " $4 "."
  print "    And it is running on terminal " $2
}
```

```
prompt$ ps | awk -f prog2.awk
Process PID is CMD.
    And it is running on terminal TTY
Process 12017 is bash.
    And it is running on terminal pts/0
Process 12041 is ps.
    And it is running on terminal pts/0
Process 12042 is awk.
    And it is running on terminal pts/0
prompt$ █
```

Ways to specify “which records”

- So far, we have used the default of “all records”
- What are some other ways to select records?

Ways to specify "which records"

- So far, we have used the default of "all records"
- What are some other ways to select records?

BEGIN

- The "virtual record" before the first real record
- Allows initialization before processing input
- Requires instructions

Ways to specify "which records"

- So far, we have used the default of "all records"
- What are some other ways to select records?

`BEGIN`

- The "virtual record" before the first real record
- Allows initialization before processing input
- Requires instructions

`END`

- The "virtual record" after the last real record
- For instructions to execute after processing all input
- Requires instructions

Examples

hello.awk: proper "Hello, world!" AWK script

Examples

hello.awk: proper “Hello, world!” AWK script

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
```

hibye.awk

Examples

hello.awk: proper “Hello, world!” AWK script

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
```

hibye.awk

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
END   { print "Goodbye, world!" }
```

Examples

hello.awk: proper "Hello, world!" AWK script

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
```

hibye.awk

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
END   { print "Goodbye, world!" }
```

```
prompt$ █
```

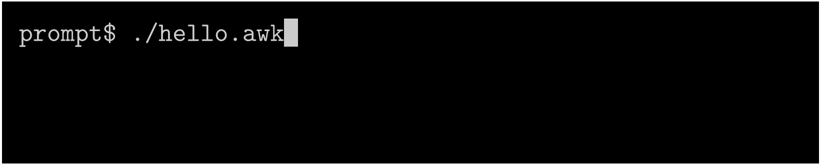
Examples

hello.awk: proper “Hello, world!” AWK script

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
```

hibye.awk

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
END   { print "Goodbye, world!" }
```



```
prompt$ ./hello.awk
```

Examples

hello.awk: proper “Hello, world!” AWK script

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
```

hibye.awk

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
END   { print "Goodbye, world!" }
```

```
prompt$ ./hello.awk
Hello, world!
prompt$ █
```

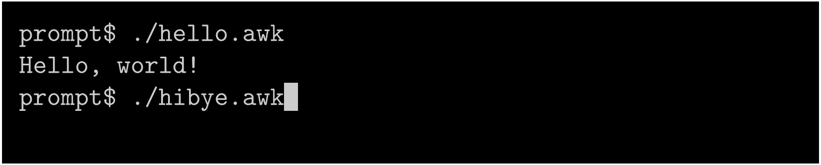
Examples

hello.awk: proper “Hello, world!” AWK script

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
```

hibye.awk

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
END   { print "Goodbye, world!" }
```



```
prompt$ ./hello.awk
Hello, world!
prompt$ ./hibye.awk
```

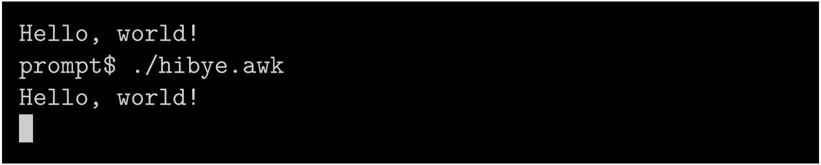
Examples

`hello.awk`: proper “Hello, world!” AWK script

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
```

`hibye.awk`

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
END   { print "Goodbye, world!" }
```



```
Hello, world!
prompt$ ./hibye.awk
Hello, world!
█
```

Examples

hello.awk: proper "Hello, world!" AWK script

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
```

hibye.awk

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
END   { print "Goodbye, world!" }
```

```
Hello, world!
prompt$ ./hibye.awk
Hello, world!
I am typing this
```

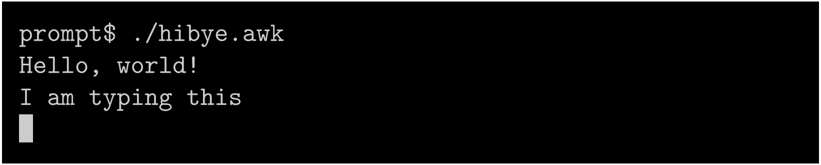
Examples

hello.awk: proper “Hello, world!” AWK script

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
```

hibye.awk

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
END   { print "Goodbye, world!" }
```



```
prompt$ ./hibye.awk
Hello, world!
I am typing this
█
```

Examples

hello.awk: proper "Hello, world!" AWK script

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
```

hibye.awk

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
END   { print "Goodbye, world!" }
```

```
prompt$ ./hibye.awk
Hello, world!
I am typing this
On the next line I will press Ctrl-D
```

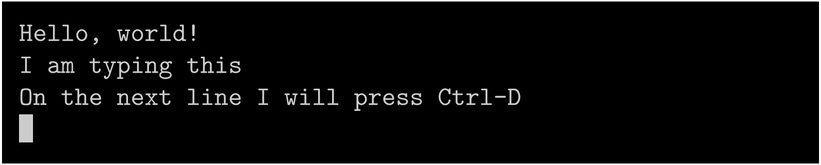
Examples

hello.awk: proper "Hello, world!" AWK script

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
```

hibye.awk

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
END   { print "Goodbye, world!" }
```



```
Hello, world!
I am typing this
On the next line I will press Ctrl-D
█
```

Examples

hello.awk: proper “Hello, world!” AWK script

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
```

hibye.awk

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
END   { print "Goodbye, world!" }
```

```
I am typing this
On the next line I will press Ctrl-D
Goodbye, world!
prompt$ █
```

Matching patterns

`/pattern/`

- Selects records that contain "pattern" somewhere
- Same pattern language as `grep`
  - I.e., "regular expressions"

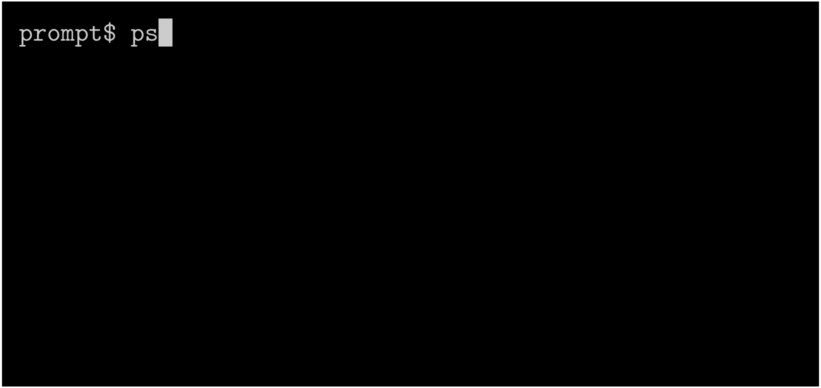
`!/pattern/`

- Selects records that do not contain "pattern"

Example: matching patterns

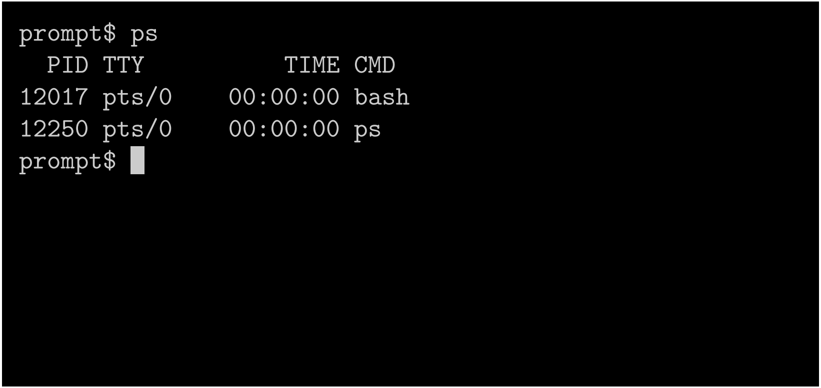
```
prompt$ █
```

Example: matching patterns



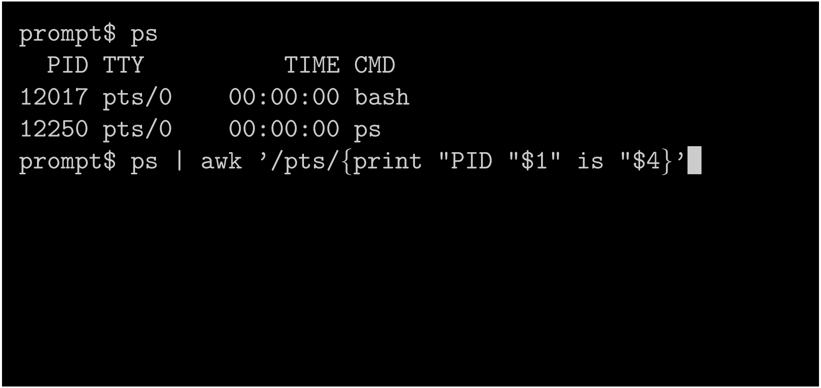
```
prompt$ ps
```

Example: matching patterns



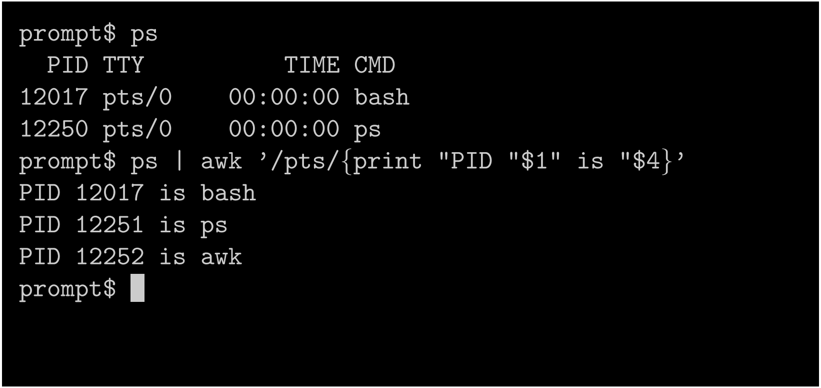
```
prompt$ ps
  PID TTY          TIME CMD
12017 pts/0    00:00:00 bash
12250 pts/0    00:00:00 ps
prompt$ 
```

Example: matching patterns



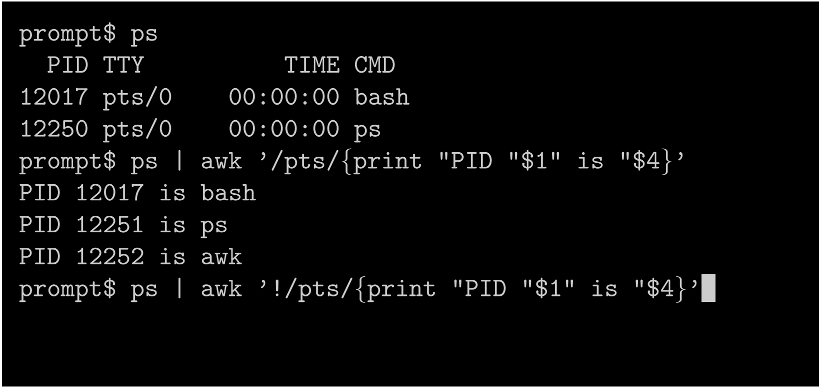
```
prompt$ ps
  PID TTY          TIME CMD
12017 pts/0    00:00:00 bash
12250 pts/0    00:00:00 ps
prompt$ ps | awk ’/pts/{print "PID "$1" is "$4}’
```

Example: matching patterns



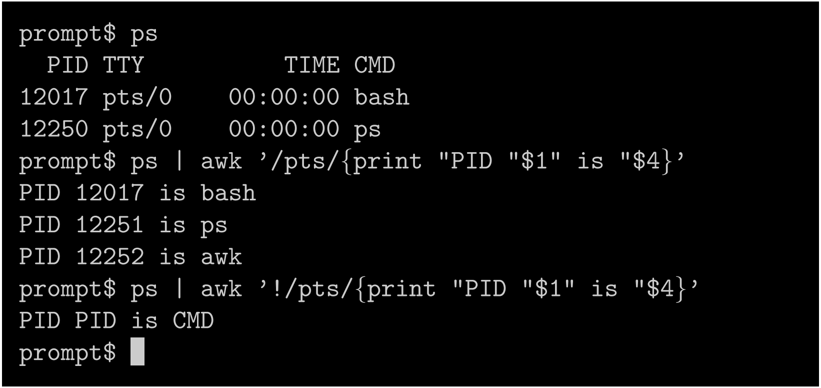
```
prompt$ ps
  PID TTY          TIME CMD
12017 pts/0    00:00:00 bash
12250 pts/0    00:00:00 ps
prompt$ ps | awk ’/pts/{print "PID "$1" is "$4}’
PID 12017 is bash
PID 12251 is ps
PID 12252 is awk
prompt$ █
```

Example: matching patterns



```
prompt$ ps
  PID TTY          TIME CMD
12017 pts/0    00:00:00 bash
12250 pts/0    00:00:00 ps
prompt$ ps | awk ’/pts/{print "PID "$1" is "$4}’
PID 12017 is bash
PID 12251 is ps
PID 12252 is awk
prompt$ ps | awk ’!/pts/{print "PID "$1" is "$4}’
```

Example: matching patterns



```
prompt$ ps
  PID TTY          TIME CMD
12017 pts/0    00:00:00 bash
12250 pts/0    00:00:00 ps
prompt$ ps | awk ’/pts/{print "PID "$1" is "$4}’
PID 12017 is bash
PID 12251 is ps
PID 12252 is awk
prompt$ ps | awk ’!/pts/{print "PID "$1" is "$4}’
PID PID is CMD
prompt$
```

Simple quiz

What does `awk '/pattern/'` do?

Simple quiz

What does `awk ’/pattern/’` do?

```
awk ’/pattern/’  ≡  awk ’/pattern/{print}’
                 ≡  awk ’/pattern/{print $0}’
```

- Print input lines that contain “pattern”

Simple quiz

What does `awk '/pattern/'` do?

```
awk '/pattern/'  ≡  awk '/pattern/{print}'
                 ≡  awk '/pattern/{print $0}'
```

- Print input lines that contain "pattern"
- `awk '/pattern/'` ≡ `grep 'pattern'`
- `awk '!/pattern/'` ≡ `grep -v 'pattern'`

More ways to select records

relational–expression

- Select records based on the criteria in the expression

What can go in the "relational–expression"?

- Terms
  - Literals: `"Bob"`, `"50"`, `50`
  - Fields: `$4`
  - Variables (in a few slides . . .)
- Relational operators
  - The usual C/Java ones: `>`, `<`, `>=`, `<=`, `==`, `!=`
  - Are clever about string vs. numerical comparisons
- Parentheses for grouping
- Logical operators
  - The usual C/Java ones: `!`, `&&`, `||`

More ways to select records

relational–expression

- Select records based on the criteria in the expression

What can go in the "relational–expression"?

- Terms
  - Literals: `"Bob"`, `"50"`, `50`
  - Fields: `$4`
  - Variables (in a few slides . . .)
- Relational operators
  - The usual C/Java ones: `>`, `<`, `>=`, `<=`, `==`, `!=`
  - Are clever about string vs. numerical comparisons
- Parentheses for grouping
- Logical operators
  - The usual C/Java ones: `!`, `&&`, `||`
- Matching operators. . .

Matching operators?

What are the matching operators?

str ˜ /pattern/

- Check if string `str` matches `pattern`
- Again, pattern is a regular expression

str !˜ /pattern/

- Check if string `str` does not match `pattern`

Example: prog1.awk revisited

prog1.awk

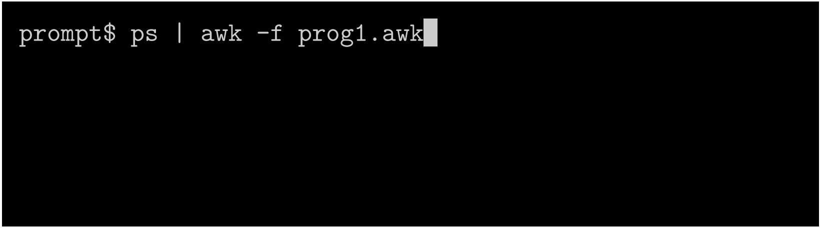
```
{ print "Process " $1 " is " $4 "." }
```

```
prompt$ 
```

Example: prog1.awk revisited

prog1.awk

```
{ print "Process " $1 " is " $4 "." }
```

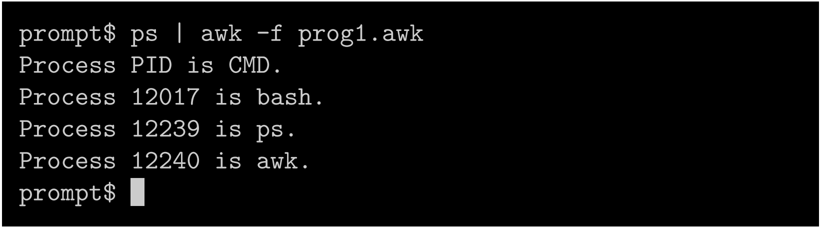


```
prompt$ ps | awk -f prog1.awk
```

Example: prog1.awk revisited

prog1.awk

```
{ print "Process " $1 " is " $4 "." }
```



```
prompt$ ps | awk -f prog1.awk
Process PID is CMD.
Process 12017 is bash.
Process 12239 is ps.
Process 12240 is awk.
prompt$ █
```

Example: prog1.awk revisited

prog1.awk

```
{ print "Process " $1 " is " $4 "." }
```

```
prompt$ ps | awk -f prog1.awk
Process PID is CMD.
Process 12017 is bash.
Process 12239 is ps.
Process 12240 is awk.
prompt$
```

- How can we avoid printing “Process PID is CMD”?

Example: prog1.awk revisited

prog1.awk

```
{ print "Process " $1 " is " $4 "." }
```

```
prompt$ ps | awk -f prog1.awk
Process PID is CMD.
Process 12017 is bash.
Process 12239 is ps.
Process 12240 is awk.
prompt$
```

- How can we avoid printing “`Process PID is CMD`”?
  - One way — select records where field 1 is not PID

Example: prog1.awk revisited

prog1.awk

```
{ print "Process " $1 " is " $4 "." }
```

```
prompt$ ps | awk -f prog1.awk
Process PID is CMD.
Process 12017 is bash.
Process 12239 is ps.
Process 12240 is awk.
prompt$
```

- How can we avoid printing “Process PID is CMD”?
  - One way — select records where field 1 is not PID

prog3.awk

```
$1 != "PID" { print "Process " $1 " is " $4 "." }
```

Cryptic 1–line example program

```
$3 ~ /bob/ {print $9}
```

Cryptic 1–line example program

```
$3 ~ /bob/ {print $9}
```

- For all records where field 3 matches the pattern /bob/, . . .

Cryptic 1–line example program

```
$3 ~ /bob/ {print $9}
```

- For all records where field 3 matches the pattern `/bob/`, . . .
- Print field 9

Cryptic 1–line example program

```
$3 ~ /bob/ {print $9}
```

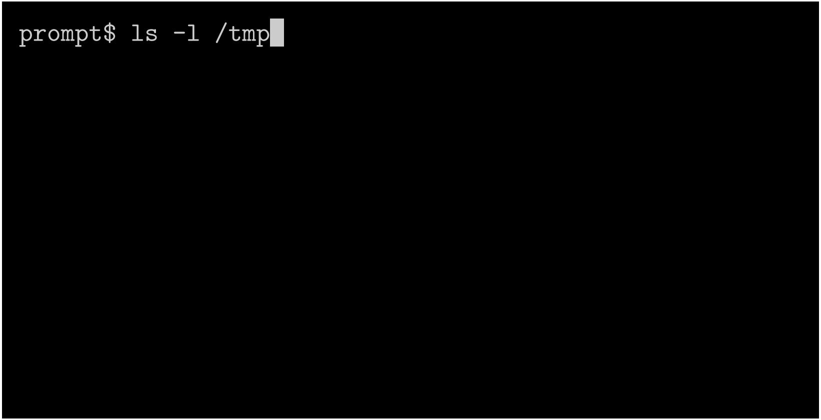
- For all records where field 3 matches the pattern /bob/, . . .
- Print field 9

```
prompt$
```

Cryptic 1–line example program

`$3 ~ /bob/ {print $9}`

- For all records where field 3 matches the pattern `/bob/`, . . .
- Print field 9

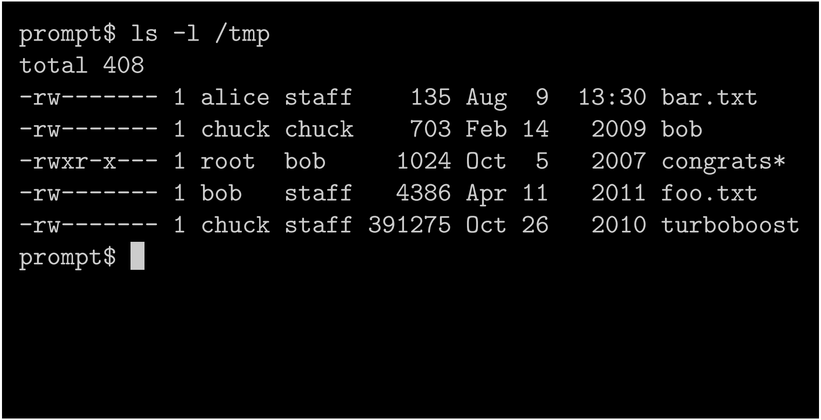


```
prompt$ ls -l /tmp
```

Cryptic 1–line example program

```
$3 ~ /bob/ {print $9}
```

- For all records where field 3 matches the pattern /bob/, . . .
- Print field 9

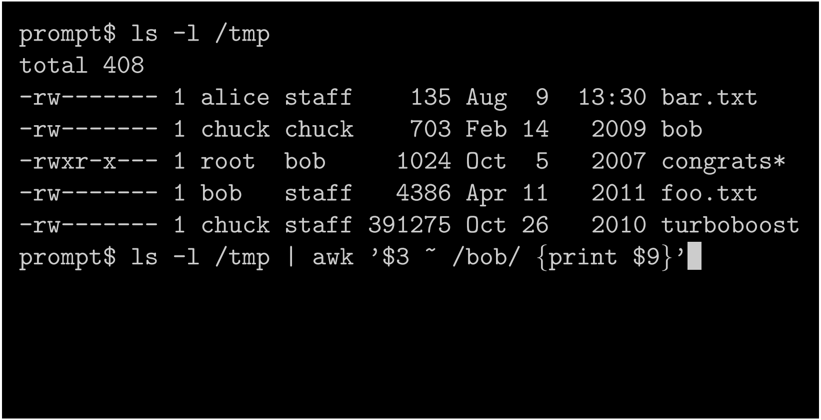


```
prompt$ ls -l /tmp
total 408
-rw------- 1 alice staff    135 Aug  9  13:30 bar.txt
-rw------- 1 chuck chuck    703 Feb 14   2009 bob
-rwxr-x--- 1 root  bob     1024 Oct  5   2007 congrats*
-rw------- 1 bob   staff   4386 Apr 11   2011 foo.txt
-rw------- 1 chuck staff 391275 Oct 26   2010 turboboost
prompt$ █
```

Cryptic 1–line example program

`$3 ~ /bob/ {print $9}`

- For all records where field 3 matches the pattern /bob/, . . .
- Print field 9

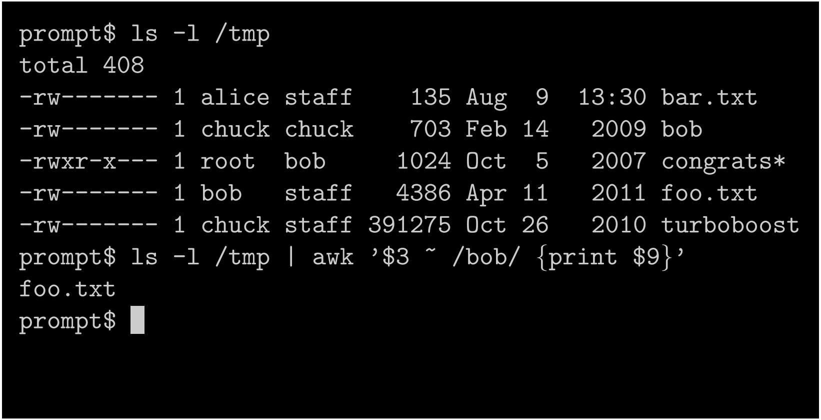


```
prompt$ ls -l /tmp
total 408
-rw------- 1 alice staff    135 Aug  9  13:30 bar.txt
-rw------- 1 chuck chuck    703 Feb 14   2009 bob
-rwxr-x--- 1 root  bob     1024 Oct  5   2007 congrats*
-rw------- 1 bob   staff   4386 Apr 11   2011 foo.txt
-rw------- 1 chuck staff 391275 Oct 26   2010 turboboost
prompt$ ls -l /tmp | awk ’$3 ~ /bob/ {print $9}’
```

Cryptic 1–line example program

`$3 ~ /bob/ {print $9}`

- For all records where field 3 matches the pattern /bob/, . . .
- Print field 9

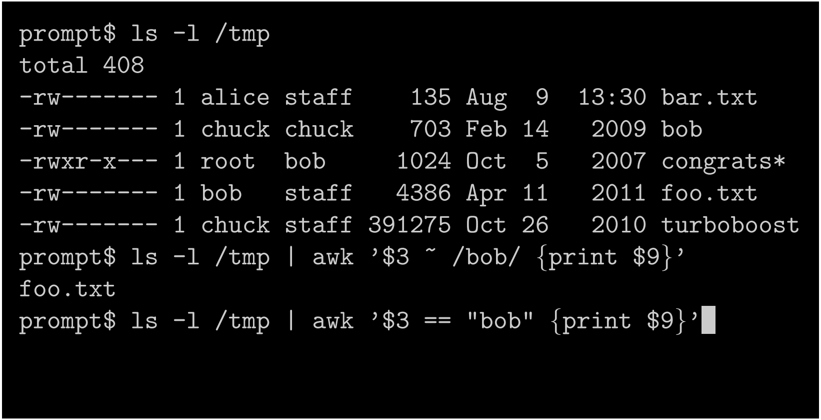


```
prompt$ ls -l /tmp
total 408
-rw------- 1 alice staff    135 Aug  9  13:30 bar.txt
-rw------- 1 chuck chuck    703 Feb 14   2009 bob
-rwxr-x--- 1 root  bob     1024 Oct  5   2007 congrats*
-rw------- 1 bob   staff   4386 Apr 11   2011 foo.txt
-rw------- 1 chuck staff 391275 Oct 26   2010 turboboost
prompt$ ls -l /tmp | awk '$3 ~ /bob/ {print $9}'
foo.txt
prompt$ 
```

Cryptic 1–line example program

```
$3 ~ /bob/ {print $9}
```

- For all records where field 3 matches the pattern /bob/, . . .
- Print field 9

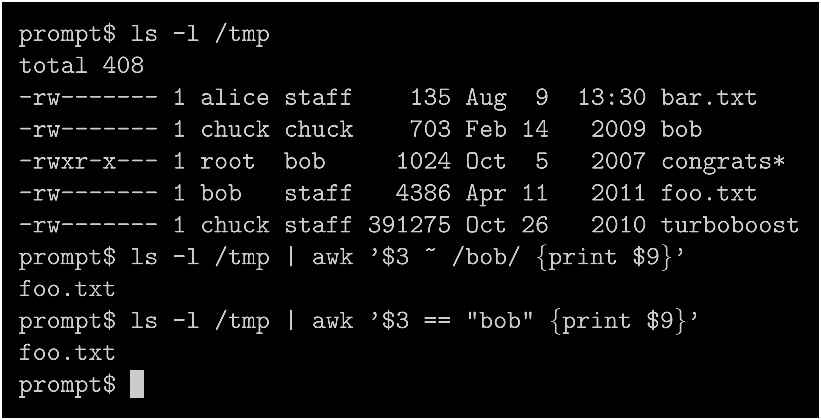


```
prompt$ ls -l /tmp
total 408
-rw------- 1 alice staff    135 Aug  9  13:30 bar.txt
-rw------- 1 chuck chuck    703 Feb 14   2009 bob
-rwxr-x--- 1 root  bob     1024 Oct  5   2007 congrats*
-rw------- 1 bob   staff   4386 Apr 11   2011 foo.txt
-rw------- 1 chuck staff 391275 Oct 26   2010 turboboost
prompt$ ls -l /tmp | awk ’$3 ~ /bob/ {print $9}’
foo.txt
prompt$ ls -l /tmp | awk ’$3 == "bob" {print $9}’
```

Cryptic 1–line example program

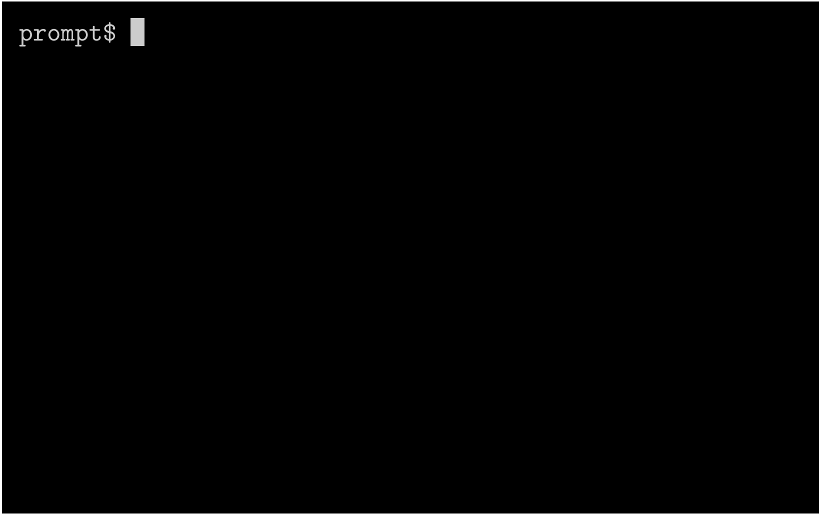
`$3 ~ /bob/ {print $9}`

- For all records where field 3 matches the pattern /bob/, . . .
- Print field 9



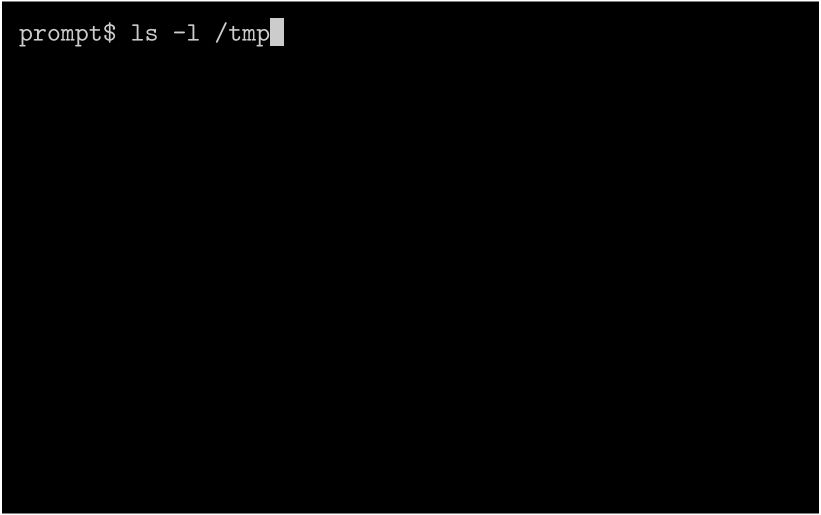
```
prompt$ ls -l /tmp
total 408
-rw------- 1 alice staff    135 Aug  9  13:30 bar.txt
-rw------- 1 chuck chuck    703 Feb 14   2009 bob
-rwxr-x--- 1 root  bob     1024 Oct  5   2007 congrats*
-rw------- 1 bob   staff   4386 Apr 11   2011 foo.txt
-rw------- 1 chuck staff 391275 Oct 26   2010 turboboost
prompt$ ls -l /tmp | awk '$3 ~ /bob/ {print $9}'
foo.txt
prompt$ ls -l /tmp | awk '$3 == "bob" {print $9}'
foo.txt
prompt$ █
```

Numeric comparisons: example



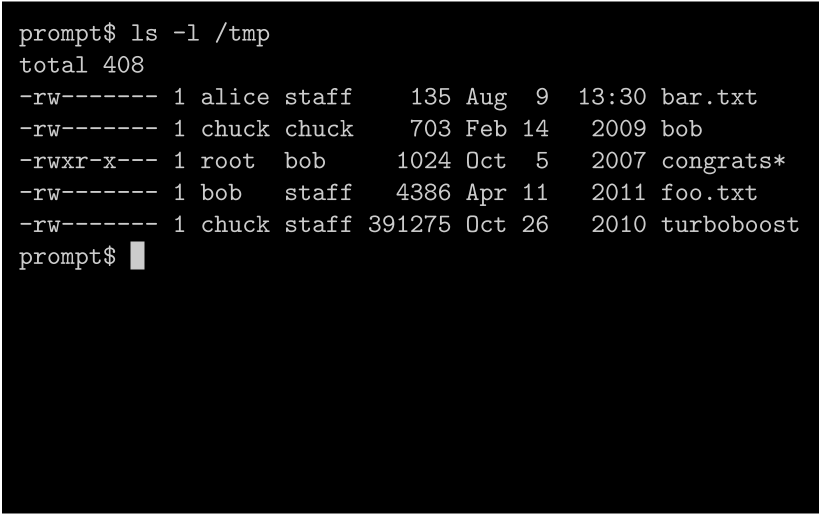
```
prompt$
```

Numeric comparisons: example



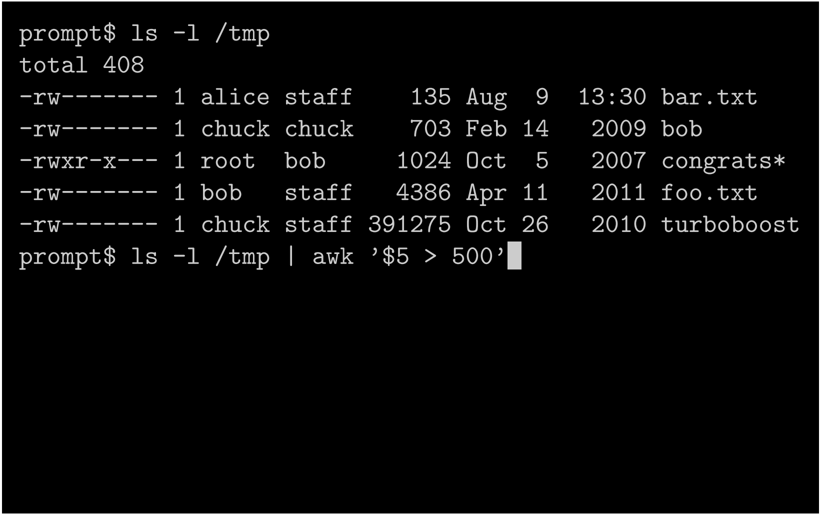
```
prompt$ ls -l /tmp
```

Numeric comparisons: example



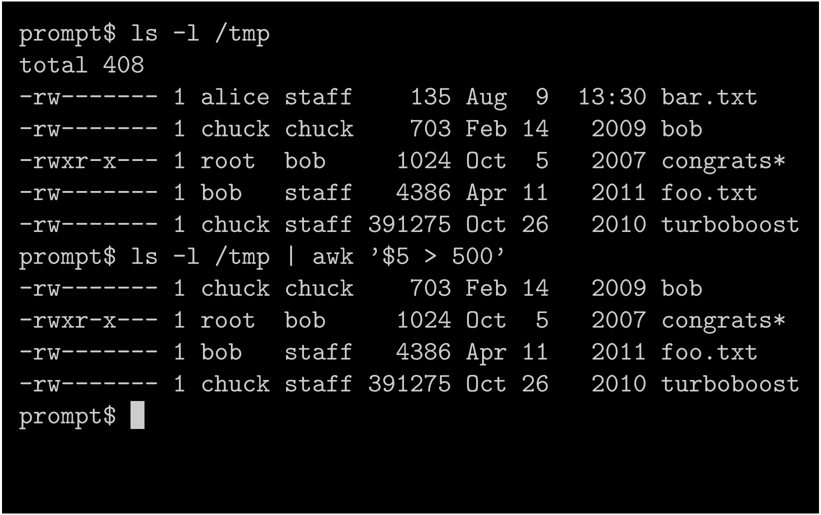
```
prompt$ ls -l /tmp
total 408
-rw------- 1 alice staff    135 Aug  9  13:30 bar.txt
-rw------- 1 chuck chuck    703 Feb 14   2009 bob
-rwxr-x--- 1 root  bob     1024 Oct  5   2007 congrats*
-rw------- 1 bob   staff   4386 Apr 11   2011 foo.txt
-rw------- 1 chuck staff 391275 Oct 26   2010 turboboost
prompt$ 
```

Numeric comparisons: example



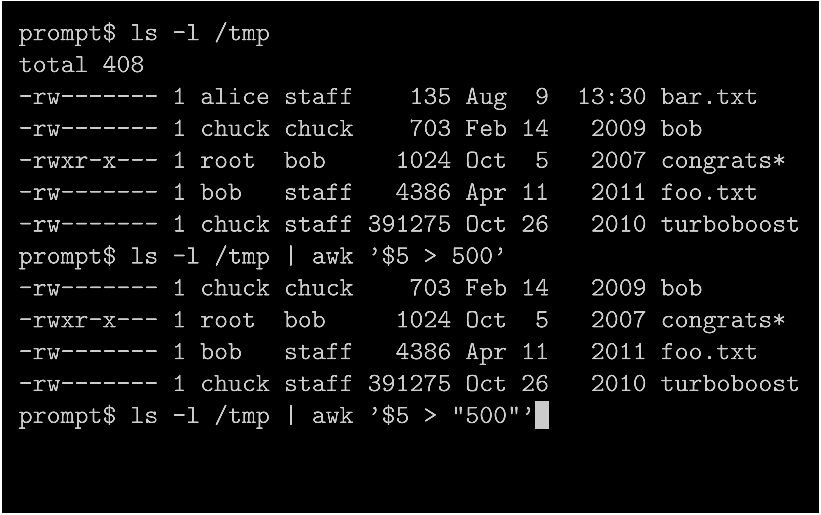
```
prompt$ ls -l /tmp
total 408
-rw------- 1 alice staff    135 Aug  9  13:30 bar.txt
-rw------- 1 chuck chuck    703 Feb 14   2009 bob
-rwxr-x--- 1 root  bob     1024 Oct  5   2007 congrats*
-rw------- 1 bob   staff   4386 Apr 11   2011 foo.txt
-rw------- 1 chuck staff 391275 Oct 26   2010 turboboost
prompt$ ls -l /tmp | awk ’$5 > 500’
```

Numeric comparisons: example



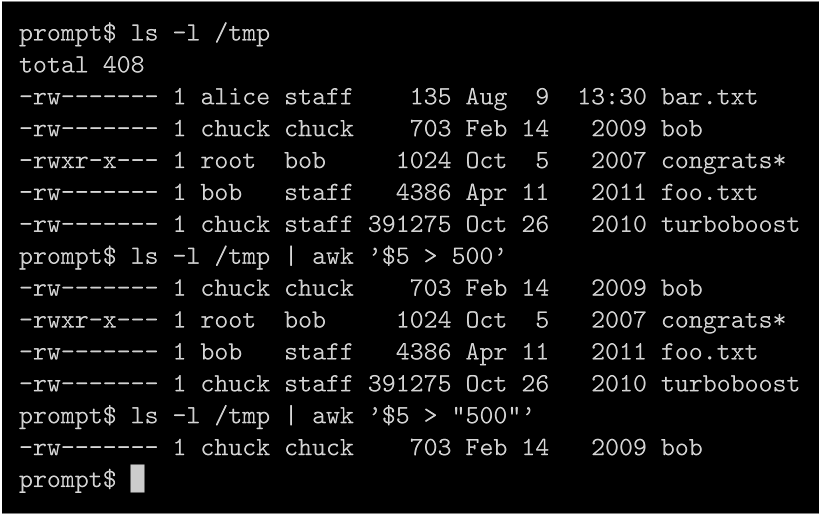
```
prompt$ ls -l /tmp
total 408
-rw------- 1 alice staff    135 Aug  9  13:30 bar.txt
-rw------- 1 chuck chuck    703 Feb 14   2009 bob
-rwxr-x--- 1 root  bob     1024 Oct  5   2007 congrats*
-rw------- 1 bob   staff   4386 Apr 11   2011 foo.txt
-rw------- 1 chuck staff 391275 Oct 26   2010 turboboost
prompt$ ls -l /tmp | awk '$5 > 500'
-rw------- 1 chuck chuck    703 Feb 14   2009 bob
-rwxr-x--- 1 root  bob     1024 Oct  5   2007 congrats*
-rw------- 1 bob   staff   4386 Apr 11   2011 foo.txt
-rw------- 1 chuck staff 391275 Oct 26   2010 turboboost
prompt$ █
```

Numeric comparisons: example



```
prompt$ ls -l /tmp
total 408
-rw------- 1 alice staff    135 Aug  9  13:30 bar.txt
-rw------- 1 chuck chuck    703 Feb 14   2009 bob
-rwxr-x--- 1 root  bob     1024 Oct  5   2007 congrats*
-rw------- 1 bob   staff   4386 Apr 11   2011 foo.txt
-rw------- 1 chuck staff 391275 Oct 26   2010 turboboost
prompt$ ls -l /tmp | awk ’$5 > 500’
-rw------- 1 chuck chuck    703 Feb 14   2009 bob
-rwxr-x--- 1 root  bob     1024 Oct  5   2007 congrats*
-rw------- 1 bob   staff   4386 Apr 11   2011 foo.txt
-rw------- 1 chuck staff 391275 Oct 26   2010 turboboost
prompt$ ls -l /tmp | awk ’$5 > "500"’
```

Numeric comparisons: example



```
prompt$ ls -l /tmp
total 408
-rw------- 1 alice staff    135 Aug  9  13:30 bar.txt
-rw------- 1 chuck chuck    703 Feb 14   2009 bob
-rwxr-x--- 1 root  bob     1024 Oct  5   2007 congrats*
-rw------- 1 bob   staff   4386 Apr 11   2011 foo.txt
-rw------- 1 chuck staff 391275 Oct 26   2010 turboboost
prompt$ ls -l /tmp | awk ’$5 > 500’
-rw------- 1 chuck chuck    703 Feb 14   2009 bob
-rwxr-x--- 1 root  bob     1024 Oct  5   2007 congrats*
-rw------- 1 bob   staff   4386 Apr 11   2011 foo.txt
-rw------- 1 chuck staff 391275 Oct 26   2010 turboboost
prompt$ ls -l /tmp | awk ’$5 > "500"’
-rw------- 1 chuck chuck    703 Feb 14   2009 bob
prompt$ █
```

AWK variables

- Same variable naming rules as bash / C / Java
  - May contain letters, digits, underscores
  - May not start with a digit
  - Are case sensitive
- Like bash: no need to declare them
- Like bash: no type information
- Unlike bash: each variable has two values
  - A string value
  - A numeric value
    - Non-numeric strings have numeric value 0
  - `awk` uses the appropriate value based on expression
- Using a variable is like C / Java
  - Do not need the annoying “$” of `bash`

AWK operators

Same as C / Java but some extras

+	addition
−	subtraction or unary minus
∗	multiplication
/	division
%	modulo

AWK operators

Same as C / Java but some extras

+	addition
−	subtraction or unary minus
*	multiplication
/	division
%	modulo
^	exponentiation
**	exponentiation (not POSIX compliant)

AWK assignment operators

Includes the usual C / Java operators

= Assignment

+= Add value to a variable

-= Subtract value from a variable

*= Multiply variable by a value

/= Divide variable by a value

%= “Mod” variable by a value

^= “Exponentiate” variable by a value

**= “Exponentiate” variable by a value

- Not POSIX compliant

++ Increment operator (pre or post)

-- Decrement operator (pre or post)

Example: display a file with line counts

catn.awk

```
#!/usr/bin/awk -f
```

Example: display a file with line counts

catn.awk

```
#!/usr/bin/awk -f
BEGIN  { n = 0 }
```

Example: display a file with line counts

catn.awk

```
#!/usr/bin/awk -f
BEGIN  { n = 0 }
       { n++; print n ":\t" $0 }
```

Example: display a file with line counts

catn.awk

```
#!/usr/bin/awk -f
BEGIN  { n = 0 }
       { print ++n ":\t" $0 }
```

Example: show files owned by bob

With a byte total at the end

We will use

```
ls -l | ./bobtotal.awk
```

Need to write the AWK program `bobtotal.awk`

Example: show files owned by bob

With a byte total at the end

We will use

```
ls -l | ./bobtotal.awk
```

Need to write the AWK program bobtotal.awk

bobtotal.awk

```
#!/usr/bin/awk -f
```

Example: show files owned by bob

With a byte total at the end

We will use

```
ls -l | ./bobtotal.awk
```

Need to write the AWK program `bobtotal.awk`

bobtotal.awk

```
#!/usr/bin/awk -f
BEGIN  { total = 0 }
```

Example: show files owned by bob

With a byte total at the end

We will use

```
ls -l | ./bobtotal.awk
```

Need to write the AWK program `bobtotal.awk`

`bobtotal.awk`

```
#!/usr/bin/awk -f
BEGIN  { total = 0 }
$3 == "bob"  { print; total += $5 }
```

Example: show files owned by bob

With a byte total at the end

We will use

```
ls -l | ./bobtotal.awk
```

Need to write the AWK program `bobtotal.awk`

bobtotal.awk

```
#!/usr/bin/awk -f
BEGIN  { total = 0 }
$3 == "bob"  { print; total += $5 }
END    { print "Total is " total " bytes." }
```

Parameters to AWK programs

- A bit of a hack. . .

Parameters to AWK programs

- A bit of a hack. . .
- Variables can be initialized *on the command line*
  - E.g., “`var=value`”
  - Cannot have spaces around equals sign

Parameters to AWK programs

- A bit of a hack. . .
- Variables can be initialized *on the command line*
  - E.g., “`var=value`”
  - Cannot have spaces around equals sign
- Can initialize variables *in between* files
- The `BEGIN` block executes before any arguments are processed

Example: generalizing bobtotal.awk to any user

With a default user of root

We will use something like

```
ls -l | ./total.awk user=bob
```

Need to write the AWK program total.awk

total.awk

```
#!/usr/bin/awk -f
```

Example: generalizing `bobtotal.awk` to any user

With a default user of `root`

We will use something like

```
ls -l | ./total.awk user=bob
```

Need to write the AWK program `total.awk`

`total.awk`

```
#!/usr/bin/awk -f
BEGIN  { user="root"; total = 0 }
```

Example: generalizing bobtotal.awk to any user

With a default user of root

We will use something like

```
ls -l | ./total.awk user=bob
```

Need to write the AWK program total.awk

total.awk

```
#!/usr/bin/awk -f
BEGIN  { user="root"; total = 0 }
$3 == user  { print; total += $5 }
END    { print "Total is " total " bytes." }
```

AWK “system” variables (1)

NF number of fields in the current record

- Can be changed, but POSIX does not specify behavior

FS field separator

- Can be changed
- Or use command–line switch `-F`
- Default is “space” for “chunk of whitespace”

AWK "system" variables (1)

NF number of fields in the current record

- Can be changed, but POSIX does not specify behavior

FS field separator

- Can be changed
- Or use command–line switch `-F`
- Default is "space" for "chunk of whitespace"
- Can be set to a regular expression

AWK "system" variables (1)

NF number of fields in the current record

- Can be changed, but POSIX does not specify behavior

FS field separator

- Can be changed
- Or use command–line switch `-F`
- Default is "space" for "chunk of whitespace"
- Can be set to a regular expression

Example: print the number of search directories in your `PATH`

AWK “system” variables (1)

NF number of fields in the current record

- Can be changed, but POSIX does not specify behavior

FS field separator

- Can be changed
- Or use command–line switch `-F`
- Default is “space” for “chunk of whitespace”
- Can be set to a regular expression

Example: print the number of search directories in your `PATH`

```
echo $PATH | awk -F: '{print NF}'
```

AWK “system” variables (2)

FILENAME name of the current input file

FNR total number of records seen in the current input file

NR total number of records seen so far

RS record separator

- Can be changed
- Default is “newline character”

AWK "system" variables (2)

FILENAME name of the current input file

FNR total number of records seen in the current input file

NR total number of records seen so far

RS record separator

- Can be changed
- Default is "newline character"
- Can be set to a regular expression

AWK "system" variables (2)

FILENAME name of the current input file

FNR total number of records seen in the current input file

NR total number of records seen so far

RS record separator

- Can be changed
- Default is "newline character"
- Can be set to a regular expression

Easy version of `catn`:

catn.awk

```
#!/usr/bin/awk -f
{ print NR ":\t" $0 }
```

Fun use of NR

- How to print the first 17 lines of a file?

Fun use of NR

- How to print the first 17 lines of a file?

```
head -n 17 file
```

Fun use of NR

- How to print the first 17 lines of a file?

```
head -n 17 file
```

- How to print the first 17 lines of a file, using awk?

Fun use of NR

- How to print the first 17 lines of a file?

```
head -n 17 file
```

- How to print the first 17 lines of a file, using awk?

```
awk 'NR <= 17' file
```

Fun use of NR

- How to print the first 17 lines of a file?

```
head -n 17 file
```

- How to print the first 17 lines of a file, using awk?

```
awk 'NR <= 17' file
```

- How to print from line 42 onward, using awk?

```
awk 'NR >= 42' file
```

Fun use of NR

- How to print the first 17 lines of a file?

```
head -n 17 file
```

- How to print the first 17 lines of a file, using awk?

```
awk 'NR <= 17' file
```

- How to print from line 42 onward, using awk?

```
awk 'NR >= 42' file
```

- How to print lines 17 through 42, using awk?

Fun use of NR

- How to print the first 17 lines of a file?

```
head -n 17 file
```

- How to print the first 17 lines of a file, using awk?

```
awk ’NR <= 17’ file
```

- How to print from line 42 onward, using awk?

```
awk ’NR >= 42’ file
```

- How to print lines 17 through 42, using awk?

```
awk ’(NR >= 17) && (NR <= 42)’ file
```

Cool Trick #1

When specifying a field $\$n$,

Cool Trick #1

When specifying a field $\$n$, n can be an expression (with variables)

Cool Trick #1

When specifying a field $n, *n* can be an expression (with variables)

Example: print the *last* word on each line

Cool Trick #1

When specifying a field $\$n$, n can be an expression (with variables)

Example: print the *last* word on each line

```
{ print $NF }
```

Cool Trick #1

When specifying a field $$n$, n can be an expression (with variables)

Example: print the *last* word on each line

```
{ print $NF }
```

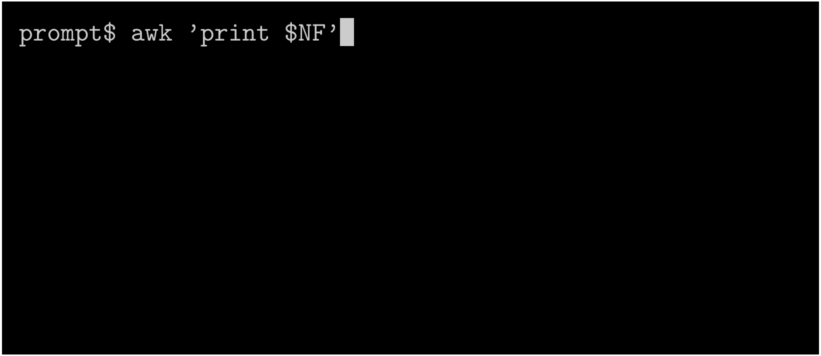
```
prompt$ █
```

Cool Trick #1

When specifying a field $$n$, n can be an expression (with variables)

Example: print the *last* word on each line

```
{ print $NF }
```



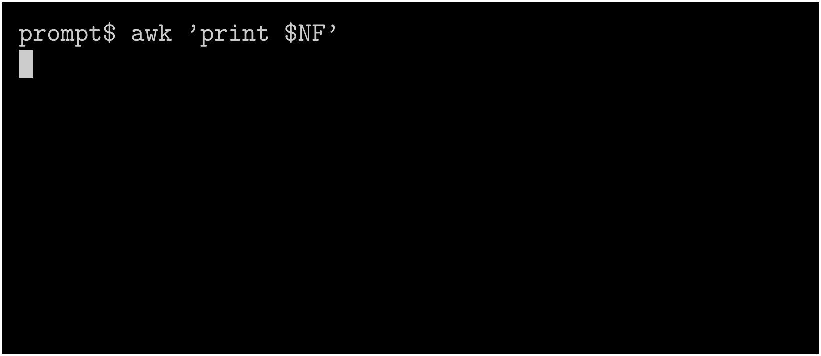
```
prompt$ awk ’print $NF’
```

Cool Trick #1

When specifying a field $\$n$, n can be an expression (with variables)

Example: print the *last* word on each line

```
{ print $NF }
```



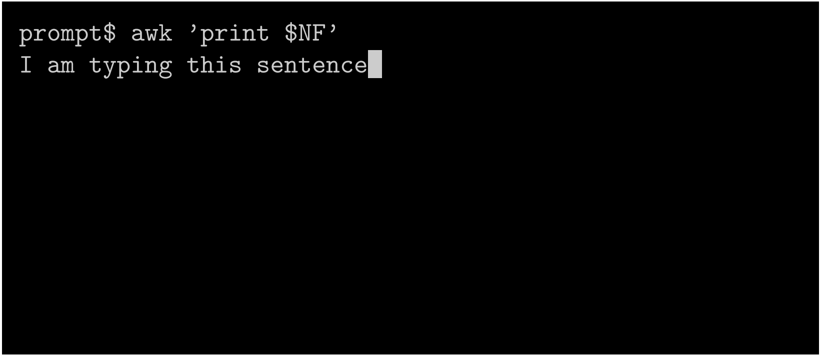
```
prompt$ awk 'print $NF'
```

Cool Trick #1

When specifying a field $n, *n* can be an expression (with variables)

Example: print the *last* word on each line

```
{ print $NF }
```



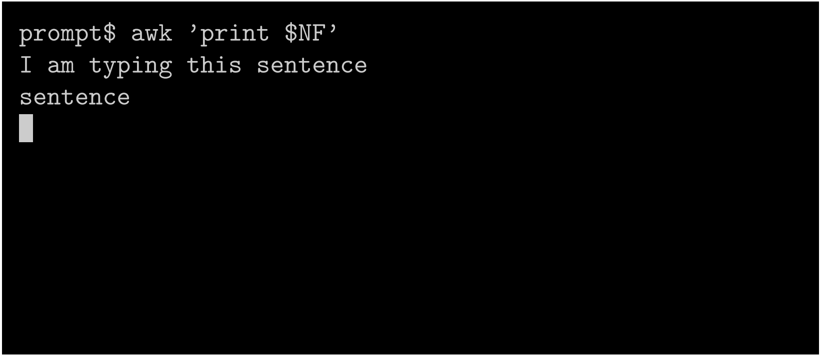
```
prompt$ awk 'print $NF'
I am typing this sentence
```

Cool Trick #1

When specifying a field $\$n$, n can be an expression (with variables)

Example: print the *last* word on each line

```
{ print $NF }
```



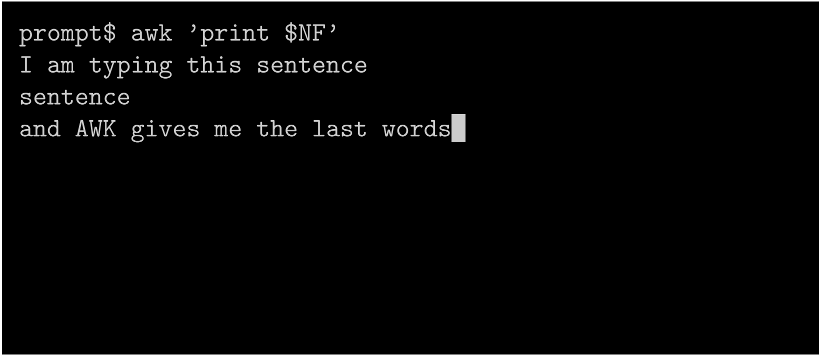
```
prompt$ awk 'print $NF'
I am typing this sentence
sentence
```

Cool Trick #1

When specifying a field $*n*, *n* can be an expression (with variables)

Example: print the *last* word on each line

```
{ print $NF }
```



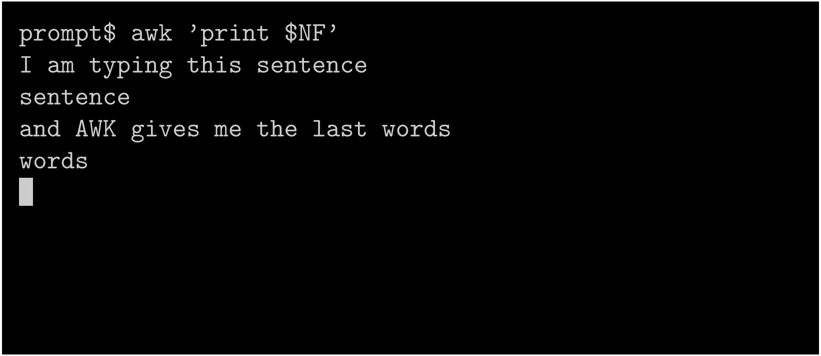
```
prompt$ awk ’print $NF’
I am typing this sentence
sentence
and AWK gives me the last words
```

Cool Trick #1

When specifying a field $*n*, *n* can be an expression (with variables)

Example: print the *last* word on each line

```
{ print $NF }
```



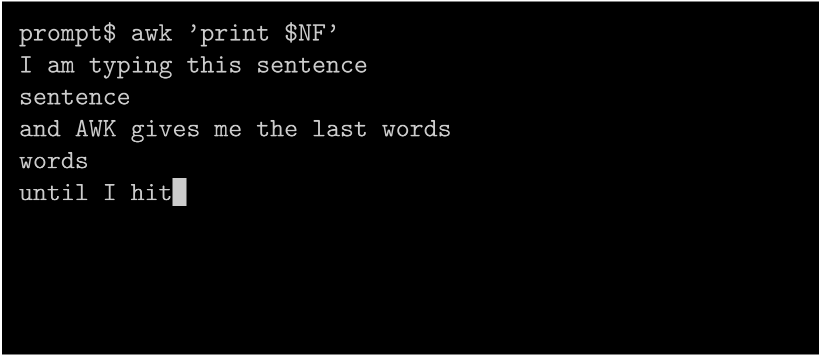
```
prompt$ awk 'print $NF'
I am typing this sentence
sentence
and AWK gives me the last words
words
```

Cool Trick #1

When specifying a field \$*n*, *n* can be an expression (with variables)

Example: print the *last* word on each line

```
{ print $NF }
```



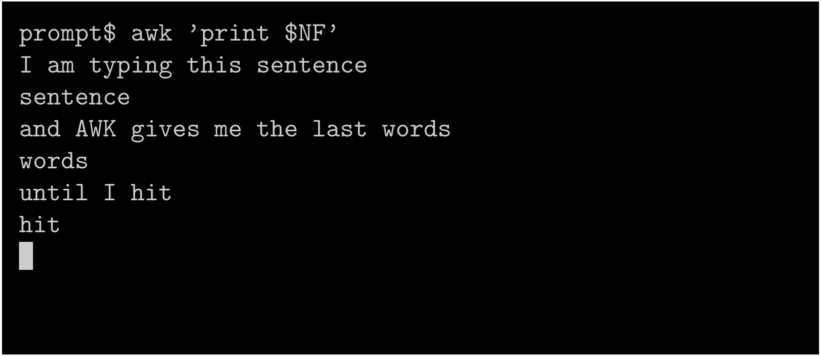
```
prompt$ awk 'print $NF'
I am typing this sentence
sentence
and AWK gives me the last words
words
until I hit
```

Cool Trick #1

When specifying a field $n, *n* can be an expression (with variables)

Example: print the *last* word on each line

```
{ print $NF }
```



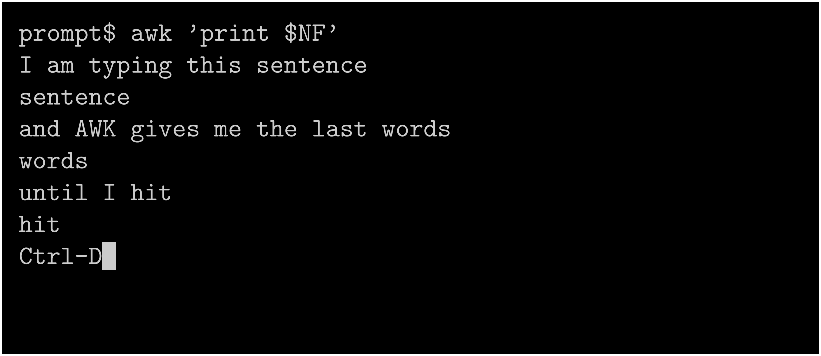
```
prompt$ awk ’print $NF’
I am typing this sentence
sentence
and AWK gives me the last words
words
until I hit
hit
```

Cool Trick #1

When specifying a field $\$n$, n can be an expression (with variables)

Example: print the *last* word on each line

```
{ print $NF }
```



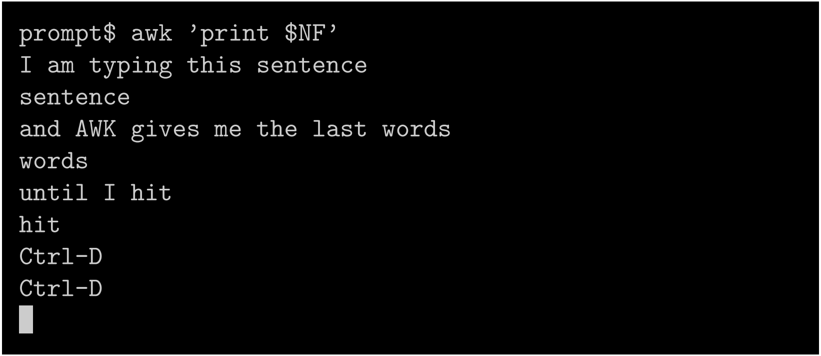
```
prompt$ awk 'print $NF'
I am typing this sentence
sentence
and AWK gives me the last words
words
until I hit
hit
Ctrl-D
```

Cool Trick #1

When specifying a field $n, n can be an expression (with variables)

Example: print the *last* word on each line

```
{ print $NF }
```



```
prompt$ awk 'print $NF'
I am typing this sentence
sentence
and AWK gives me the last words
words
until I hit
hit
Ctrl-D
Ctrl-D
```

Cool Trick #1

When specifying a field \$*n*, *n* can be an expression (with variables)

Example: print the *last* word on each line

```
{ print $NF }
```

```
prompt$ awk 'print $NF'
I am typing this sentence
sentence
and AWK gives me the last words
words
until I hit
hit
Ctrl-D
Ctrl-D
prompt$ █
```

Cool Trick #2

Cool Trick #2

Fields are not fixed

Cool Trick #2

Fields are not fixed

- We can assign new values to a field as usual
  - E.g., `$1 = "New thing"`
- This does not change the input file
- This does change the record in memory
  - I.e., $0 will be updated

Cool Trick #2

Fields are not fixed

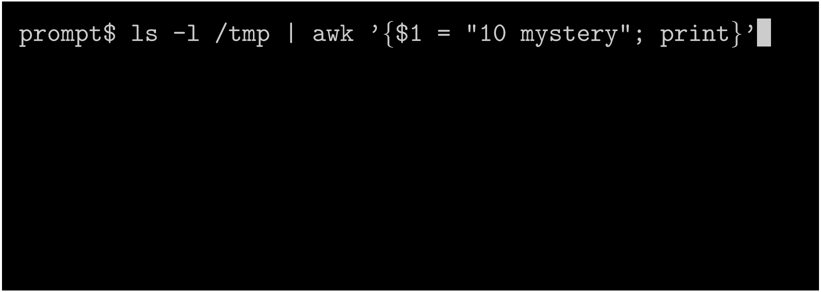
- We can assign new values to a field as usual
  - E.g., `$1 = "New thing"`
- This does not change the input file
- This does change the record in memory
  - I.e., $0 will be updated

```
prompt$ █
```

Cool Trick #2

Fields are not fixed

- We can assign new values to a field as usual
  - E.g., `$1 = "New thing"`
- This does not change the input file
- This does change the record in memory
  - I.e., $0 will be updated



```
prompt$ ls -l /tmp | awk '{$1 = "10 mystery"; print}'
```

Cool Trick #2

Fields are not fixed

- We can assign new values to a field as usual
  - E.g., `$1 = "New thing"`
- This does not change the input file
- This does change the record in memory
  - I.e., $0 will be updated

```
prompt$ ls -l /tmp | awk '{$1 = "10 mystery"; print}'
10 mystery 408
10 mystery 1 alice staff    135 Aug  9  13:30 bar.txt
10 mystery 1 chuck chuck    703 Feb 14   2009 bob
10 mystery 1 root  bob     1024 Oct  5   2007 congrats*
10 mystery 1 bob   staff   4386 Apr 11   2011 foo.txt
10 mystery 1 chuck staff 391275 Oct 26   2010 turboboost
prompt$ █
```

C and AWK

- AWK borrows lots of things from C
  - Kernighan's influence?
- For example, in AWK we can use:

C and AWK

- AWK borrows lots of things from C
  - Kernighan's influence?
- For example, in AWK we can use:

`printf`

- *Exactly* like in C
- Usage: `printf(` *format-string* [, arg[, arg. . .]])
- See `man 3 printf` for what can go in a format string

Conditionals

- We can use conditionals in the list of instructions
- Syntax is the same as C:

If–then–else syntax

```
if ( Relational-expression )
  instruction-or-block
[ else
  instruction-or-block ]
```

- Can be all on one line, or split
- The `else` part is optional
- “instruction-or-block” is either:
  1. One instruction (may need a semicolon)
  2. Several instructions, grouped in braces

Examples with conditionals

- Another way to print the first 17 lines:

Examples with conditionals

- Another way to print the first 17 lines:

```
awk ’{ if (NR <= 17) print }’ file
```

Examples with conditionals

- Another way to print the first 17 lines:

```
awk ’{ if (NR <= 17) print }’ file
```

- Insert a blank line every 5 lines:

Examples with conditionals

- Another way to print the first 17 lines:

```
awk ’{ if (NR <= 17) print }’ file
```

- Insert a blank line every 5 lines:

```
awk ’{ print; if (!(NR%5)) print ""}’ file
```

  - A zero value means false in AWK (and in C)
  - A non–zero value means true in AWK (and in C)

Conditionals (2)

- We can use the "if-then-else" operator from C

if-then-else operator

```
( Relational-expression ) ? then-expr : else-expr
```

- This goes inside an expression
- The `else` part is required
- Example expressions:

```
(x>1) ? "bob" : "alice"
```

```
2* ( (x>1) ? x : 1 ) + 7
```

Fancy example

Pretty printer

- First line should be bold text
- Remaining lines alternate blue and purple
- Use “escape sequences” to change text style in the terminal
    - Below, these are written as `<bold>`, etc.

Fancy example

Pretty printer

- First line should be bold text
- Remaining lines alternate blue and purple
- Use “escape sequences” to change text style in the terminal
  - Below, these are written as `<bold>`, etc.

pretty.awk

```
#!/usr/bin/awk -f
BEGIN       { printf "<bold>" }
{ print $0 ( (NR%2) ? "<blue>" : "<purple>" ) }
END         { printf "<normal>" }
```

Loops

- We can use `while` loops and `for` loops in AWK
  - The syntax is the same as C
  - Sorry, no deep discussion here

Loops

- We can use `while` loops and `for` loops in AWK
    - The syntax is the same as C
    - Sorry, no deep discussion here

`commafy.awk`: Convert lists into "," separated lists

```
#!/usr/bin/awk -f
{ for (i=1; i<NF; i++) printf("%s,",$i); print $NF }
```

Loops

- We can use `while` loops and `for` loops in AWK
  - The syntax is the same as C
  - Sorry, no deep discussion here

`commafy.awk`: Convert lists into "," separated lists

```
#!/usr/bin/awk -f
{ for (i=1; i<NF; i++) printf("%s,",$i); print $NF }
```

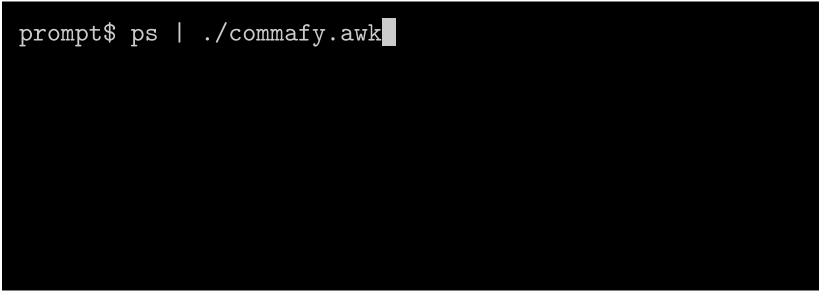
```
prompt$ █
```

Loops

- We can use `while` loops and `for` loops in AWK
    - The syntax is the same as C
    - Sorry, no deep discussion here

commafy.awk: Convert lists into "," separated lists

```
#!/usr/bin/awk -f
{ for (i=1; i<NF; i++) printf("%s,",$i); print $NF }
```



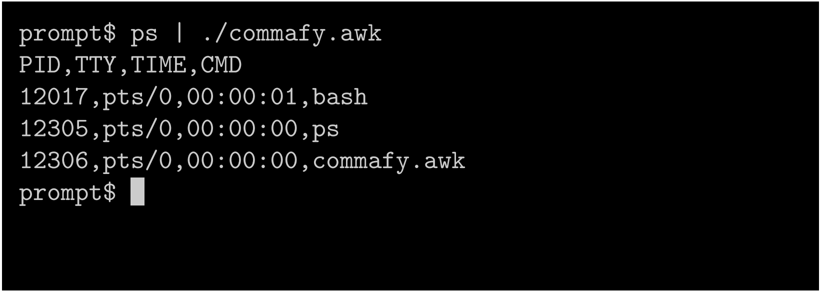
```
prompt$ ps | ./commafy.awk
```

Loops

- We can use `while` loops and `for` loops in AWK
  - The syntax is the same as C
  - Sorry, no deep discussion here

`commafy.awk`: Convert lists into “,” separated lists

```
#!/usr/bin/awk -f
{ for (i=1; i<NF; i++) printf("%s,",$i); print $NF }
```



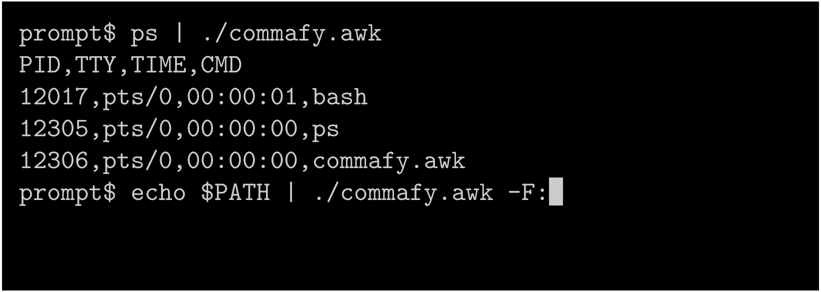
```
prompt$ ps | ./commafy.awk
PID,TTY,TIME,CMD
12017,pts/0,00:00:01,bash
12305,pts/0,00:00:00,ps
12306,pts/0,00:00:00,commafy.awk
prompt$ █
```

Loops

- We can use `while` loops and `for` loops in AWK
  - The syntax is the same as C
  - Sorry, no deep discussion here

`commafy.awk`: Convert lists into "," separated lists

```
#!/usr/bin/awk -f
{ for (i=1; i<NF; i++) printf("%s,",$i); print $NF }
```



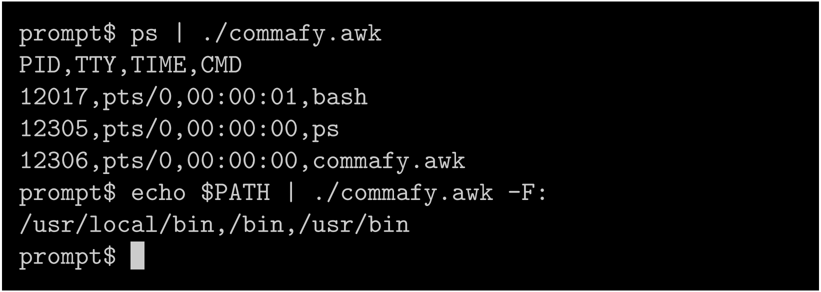
```
prompt$ ps | ./commafy.awk
PID,TTY,TIME,CMD
12017,pts/0,00:00:01,bash
12305,pts/0,00:00:00,ps
12306,pts/0,00:00:00,commafy.awk
prompt$ echo $PATH | ./commafy.awk -F:
```

Loops

- We can use `while` loops and `for` loops in AWK
  - The syntax is the same as C
  - Sorry, no deep discussion here

`commafy.awk`: Convert lists into “,” separated lists

```
#!/usr/bin/awk -f
{ for (i=1; i<NF; i++) printf("%s,",$i); print $NF }
```



```
prompt$ ps | ./commafy.awk
PID,TTY,TIME,CMD
12017,pts/0,00:00:01,bash
12305,pts/0,00:00:00,ps
12306,pts/0,00:00:00,commafy.awk
prompt$ echo $PATH | ./commafy.awk -F:
/usr/local/bin,/bin,/usr/bin
prompt$ 
```

Summary

- `awk` can make your life easier

Summary

- `awk` can make your life easier
- This is not all there is with AWK

Summary

- awk can make your life easier
- This is not all there is with AWK. For example:
  - Are there arrays in AWK?

Summary

- awk can make your life easier
- This is not all there is with AWK. For example:
  - Are there arrays in AWK? YES

Summary

- awk can make your life easier
- This is not all there is with AWK. For example:
  - Are there arrays in AWK? YES
  - Are there functions other than printf?

Summary

- awk can make your life easier
- This is not all there is with AWK. For example:
  - Are there arrays in AWK? YES
  - Are there functions other than printf? YES

Summary

- awk can make your life easier
- This is not all there is with AWK. For example:
  - Are there arrays in AWK? YES
  - Are there functions other than printf? YES
  - Can I write my own functions in AWK?

Summary

- awk can make your life easier
- This is not all there is with AWK. For example:
  - Are there arrays in AWK? YES
  - Are there functions other than printf? YES
  - Can I write my own functions in AWK? YES

End of lecture