CSCI 330
The UNIX System

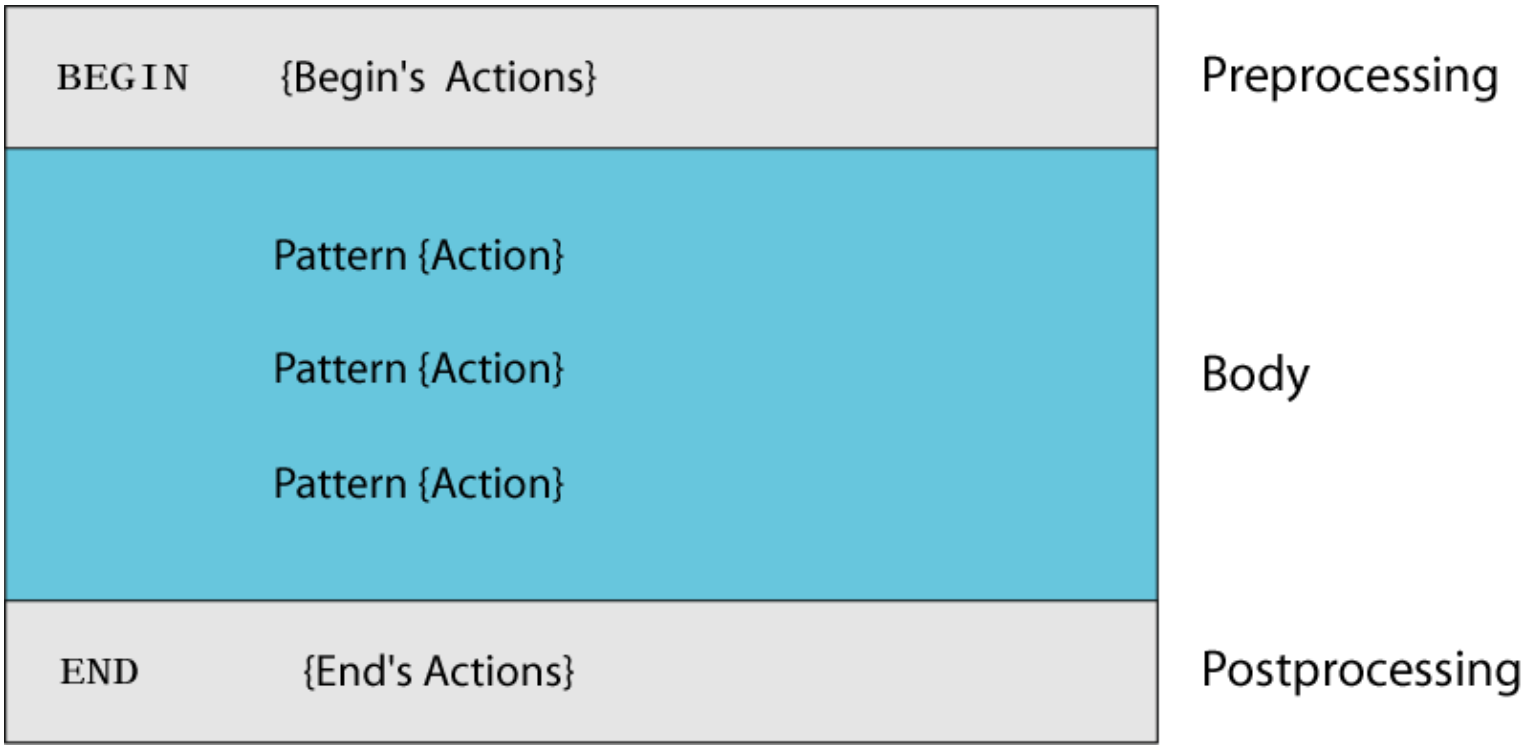
awk

What can you do with awk?

- awk operation:
  - scans a file line by line
  - splits each input line into fields
  - compares input line/fields to pattern
  - performs action(s) on matched lines
- Useful for:
  - transform data files
  - produce formatted reports
- Programming constructs:
  - format output lines
  - arithmetic and string operations
  - conditionals and loops

Typical awk script

- divided into three major parts:



- comment lines start with #

Pattern / Action Syntax

- One action statement:

```
pattern { statement }
```

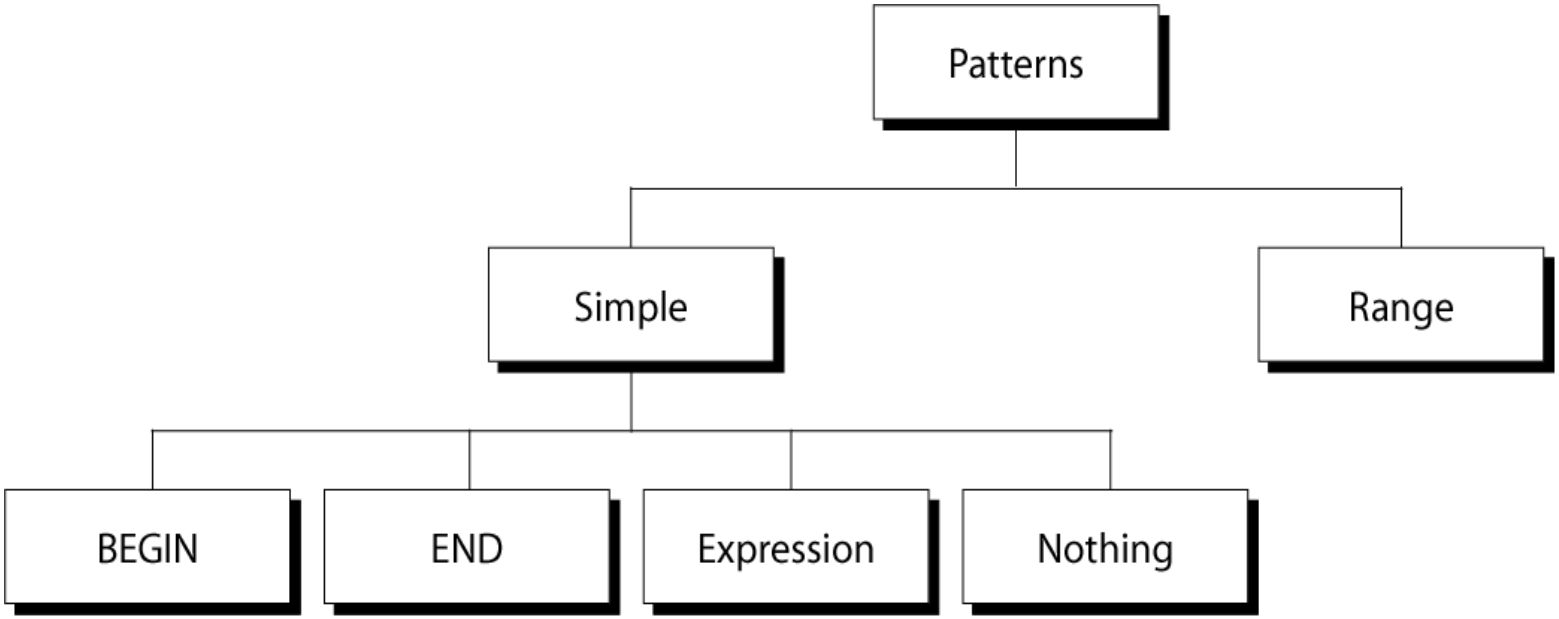
- Multiple statements:

```
pattern { statement1; statement2 }
```

or:

```
pattern {
    statement1
    statement2
    statement3
}
```

Categories of Patterns



Expression Pattern types

- match
  - entire input record

    regular expression enclosed by ‘/’s
  - explicit pattern-matching expressions

    ~ (match), !~ (not match)

- expression operators
  - arithmetic
  - relational
  - logical

Example: match input record

```
% cat emps2
Tom Jones:4424:5/12/66:543354
Mary Adams:5346:11/4/63:28765
Sally Chang:1654:7/22/54:650000
Billy Black:1683:9/23/44:336500

% awk -F: '/00$/' emps2
Sally Chang:1654:7/22/54:650000
Billy Black:1683:9/23/44:336500
```

Example: explicit match

```
% cat datafile
northwest NW     Charles Main         3.0     .98     3    34
western   WE     Sharon Gray          5.3     .97     5    23
southwest SW     Lewis Dalsass        2.7     .8      2    18
southern  SO     Suan Chin            5.1     .95     4    15
southeast SE     Patricia Jones       4.0     .7      4    17
eastern   EA     TB Savage            4.4     .84     5    20
northeast NE     AM Main              5.1     .94     3    13
north     NO     Margot Weber         4.5     .89     5     9
central   CT     Ann Stephens         5.7     .94     5    13

% awk '$5 ~ /\.[7-9]+/' datafile
southwest SW     Lewis Dalsass        2.7     .8      2    18
central   CT     Ann Stephens         5.7     .94     5    13
```

Examples: matching with REs

```
% awk '$2 !~ /E/ {print $1, $2}'
datafile
northwest NW
southwest SW
southern SO
north NO
central CT

% awk '/^[ns]/ {print $1}' datafile
northwest
southwest
southern
```

Arithmetic Operators

Operator	Meaning	Example
+	Add	x + y
-	Subtract	x – y
*	Multiply	x * y
/	Divide	x / y
%	Modulus	x % y
^	Exponential	x ^ y

Example:

```
% awk '$3 * $4 > 500 {print}' file
```

Relational Operators

Operator	Meaning	Example
<	Less than	x < y
< =	Less than or equal	x < = y
==	Equal to	x == y
!=	Not equal to	x != y
>	Greater than	x > y
> =	Greater than or equal to	x > = y
~	Matched by reg exp	x ~ /y/
!~	Not matched by req exp	x !~ /y/

Logical Operators

Operator	Meaning	Example
&&	Logical AND	a && b
||	Logical OR	a || b
!	NOT	! a

Examples:

```
% awk '($2 > 5) && ($2 <= 15)' file
% awk '$3 == 100 || $4 > 50' file
```

Range Patterns

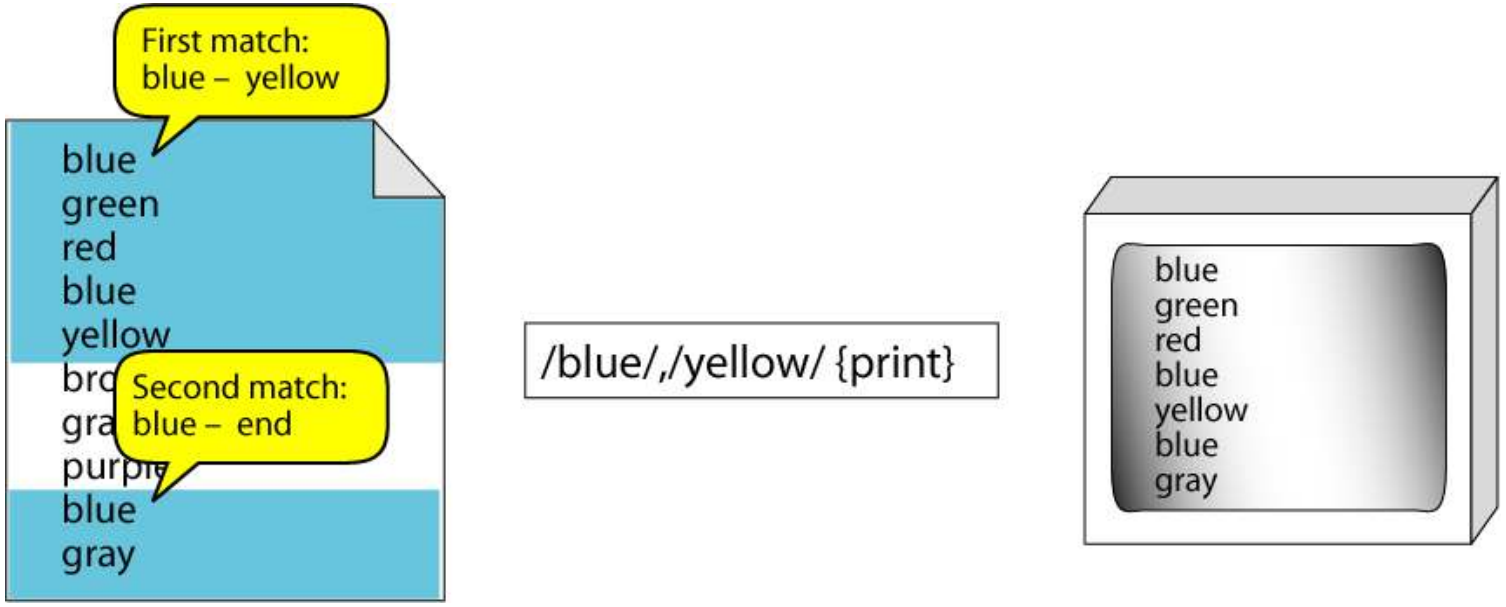
- Matches ranges of consecutive input lines

Syntax:

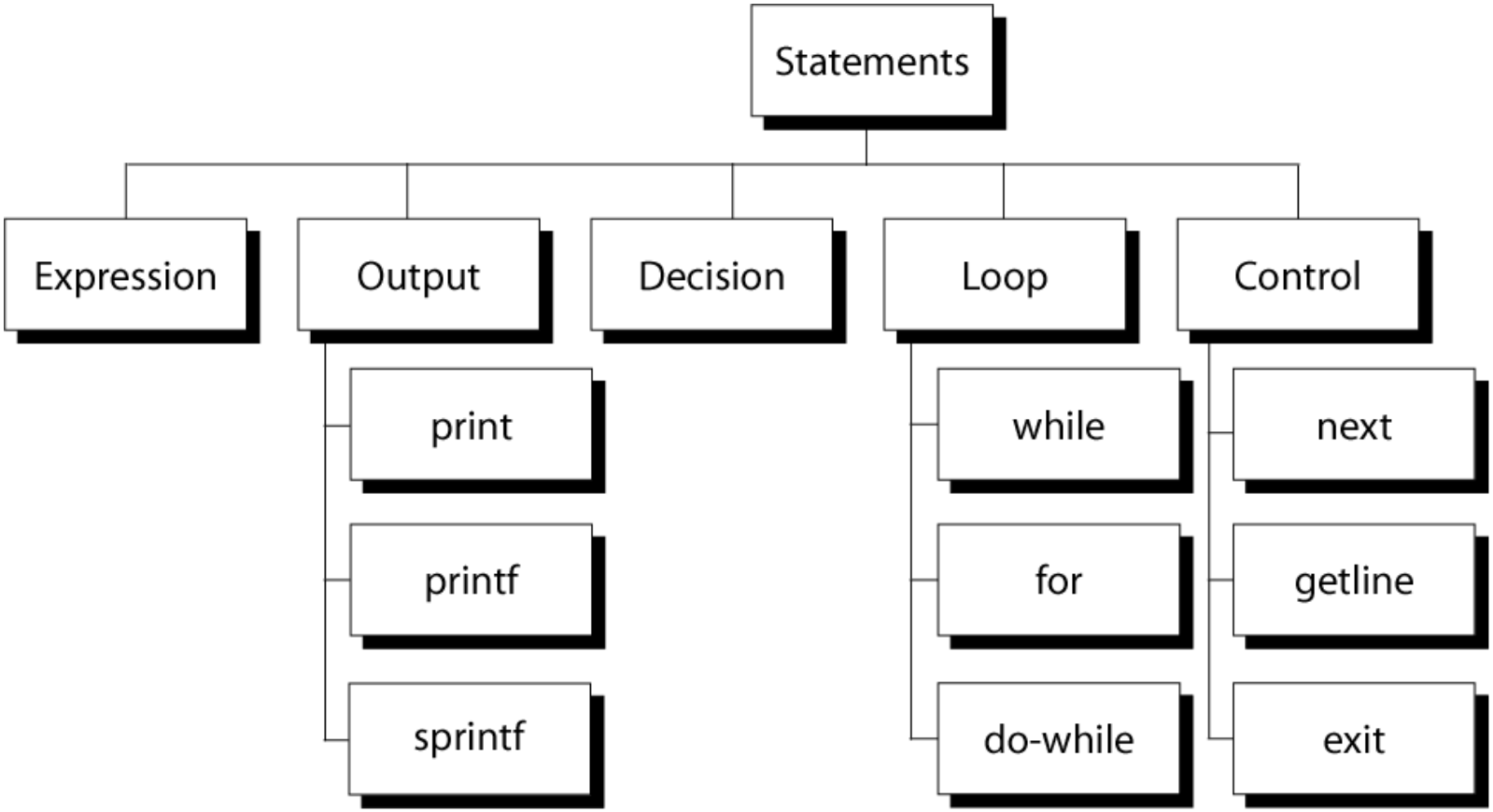
```
pattern1 , pattern2 {action}
```

- pattern can be any simple pattern
- `pattern1` turns action on
- `pattern2` turns action off

Range Pattern Example



awk Actions


Statements
Expression
Output
Decision
Loop
Control
print
printf
sprintf
while
for
do-while
next
getline
exit

awk Expression

- consists of: operands and operators
- operands:
  - numeric and string constants
  - variables
  - functions and regular expression
- operators:
  - assignment: = ++ -- += -= *= /=
  - arithmetic: + - * / % ^
  - logical: && || !
  - relational: > < >= <= == !=
  - match: ~ !~
  - string concatenation: space

awk Variables

- created via assignment:

```
var = expression
```

- types: number (not limited to integer)
  string, array
- variables come into existence when first used
- type of variable depends on its use
- variables are initialized to either 0 or “”

awk Variables

Examples:

```
% awk '$1 ~ /Tom/
          {wage = $3 * $4; print wage}'
        file
% awk '$4 == "CA"
          {$4 = "California"; print $0}'
        file
```

awk Example

- File: grades

```
john 85 92 78 94 88
andrea 89 90 75 90 86
jasper 84 88 80 92 84
```

- awk script: average

```
# average five grades
{ total = $2 + $3 + $4 + $5 + $6
  avg = total / 5
  print $1, avg }
```

- Run as:

```
awk -f average grades
```

awk Example

- File: grades

```
john 85 92 78 94 88
andrea 89 90 75 90 86
jasper 84 88 80 92 84
```

- awk script: average

```
# average five grades
{ total = $2 + $3 + $4 + $5 + $6
  avg = total / 5
  print $1, avg }
```

- Run as:

```
awk -f average grades
```

Function: print

- Writes to standard output
- Output is terminated by ORS
  - default ORS is newline
- If called with no parameter, it will print $0
- Printed parameters are separated by OFS,
  - default OFS is blank
- Print control characters are allowed:
  - \n \f \a \t \b \\ …

print examples

```
% awk '{print}' grades
john 85 92 78 94 88
andrea 89 90 75 90 86
jasper 84 88 80 92 84
% awk '{print $0}' grades
john 85 92 78 94 88
andrea 89 90 75 90 86
jasper 84 88 80 92 84
% awk '{print($0)}' grades
john 85 92 78 94 88
andrea 89 90 75 90 86
jasper 84 88 80 92 84
```

print examples

```
% awk '{print $1, $2}' grades
john 85
andrea 89
jasper 84
% awk '{print $1 "," $2}' grades
john,85
andrea,89
jasper,84
```

print examples

```
% awk '{OFS="-";print $1 , $2}' grades
john-85
andrea-89
jasper-84
% awk '{OFS="-";print $1 "," $2}' grades
john,85
andrea,89
jasper,84
```

Redirecting print output

- Print output goes to standard output

unless redirected via:

> “file”

>> “file”

| “command”

- will open file or command only once
- subsequent redirections append to already open stream

print example

```
% awk '{print $1 , $2 > "file"}' grades
% ls
file  grades
% cat file
john 85
andrea 89
jasper 84
```

print examples

```
% awk '{print $1,$2 | "sort"}' grades
andrea 89
jasper 84
john 85

% awk '{print $1,$2 | "sort -k 2"}' grades
jasper 84
john 85
andrea 89
```

printf: Formatting output

Syntax:

```
printf(format-string, var1, var2, …)
```

- works like C printf
- each format specifier within “format-string” requires additional argument of matching type

Format specifiers

%d, %i decimal integer

%c single character

%s string of characters

%f floating point number

%o octal number

%x hexadecimal number

%e scientific floating point notation

%% the letter “%”

Format specifier modifiers

- between “%” and letter

  %10s

  %7d

  %10.4f

  %-20s

- meaning:
  - width of field, field is printed right justified
  - precision: number of digits after decimal point
  - “-” will left justify

Format specifier examples

Given: x = 'A', y = 15, z = 2.3, and $1 = Bob Smith

Printf Format Specifier	What it Does
%c	*printf("The character is %c\n", x)* **output: The character is A**
%d	*printf("The boy is %d years old\n", y)* **output: The boy is 15 years old**
%s	*printf("My name is %s \n", $1)* **output: My name is Bob Smith**
%f	*printf("z is %5.3f\n", z)* **output: z is 2.300**

sprintf: Formatting text

Syntax:

```
sprintf(format-string, var1, var2, …)
```

- Works like printf, but does not produce output
- Instead it returns formatted string

Example:

```
{
    text = sprintf("1: %d - 2: %d", $1, $2)
    print text
}
```

awk builtin functions

`tolower(string)`

- returns a copy of string, with each upper-case character in the string replaced with its corresponding lower-case character. Nonalphabetic characters are left unchanged.

  Example: tolower("MiXeD cAsE 123")

  returns "mixed case 123“

`toupper(string)`

- returns a copy of string, with each lower-case character in the string replaced with its corresponding upper-case character

awk Example: list of products

```
101:propeller:104.99
102:trailer hitch:97.95
103:sway bar:49.99
104:fishing line:0.99
105:mirror:4.99
106:cup holder:2.49
107:cooler:14.89
108:wheel:49.99
109:transom:199.00
110:pulley:9.88
111:lock:31.00
112:boat cover:120.00
113:premium fish bait:1.00
```

awk Example: output

```
Marine Parts R Us
Main catalog
Part-id name                      price
=====================================
101     propeller                104.99
102     trailer hitch             97.95
103     sway bar                  49.99
104     fishing line               0.99
105     mirror                     4.99
106     cup holder                 2.49
107     cooler                    14.89
108     wheel                     49.99
109     transom                  199.00
110     pulley                     9.88
111     lock                      31.00
112     boat cover               120.00
113     premium fish bait          1.00
=====================================
Catalog has 13 parts
```

awk Example: complete

```
BEGIN {
        FS= ":"
        print "Marine Parts R Us"
        print "Main catalog"
        print "Part-id\tname\t\t\t price"
        print "=================================="
}
{
        printf("%3d\t%-20s\t%6.2f\n", $1, $2, $3)
}
END {
        print "=================================="
        print "Catalog has", NR, "parts"
}
```

awk Array

- awk allows one-dimensional arrays
  to store strings or numbers
- index can be number or string

- array need not be declared
  - its size
  - its elements
- array elements are created when first used
  - initialized to 0 or “”

Arrays in awk

Syntax:

```
arrayName[index] = value
```

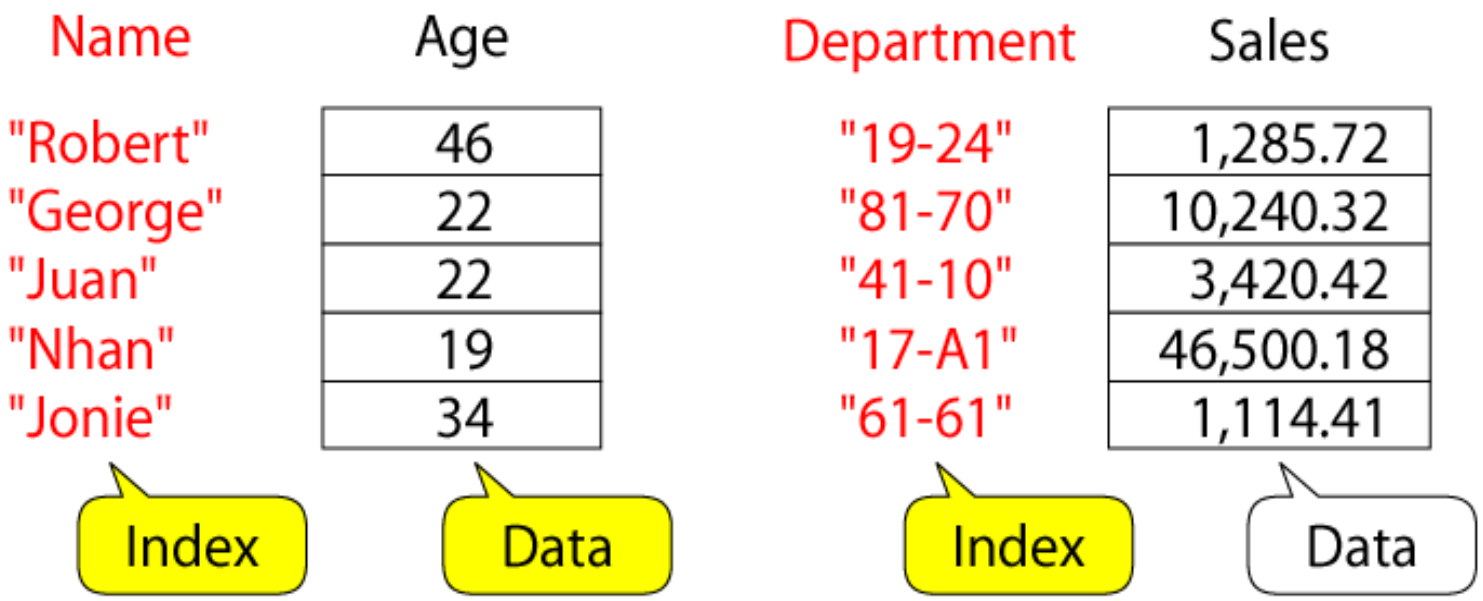
Examples:

```
list[1] = "some value"
list[2] = 123

list["other"] = "oh my !"
```

Illustration: Associative Arrays

- awk arrays can use string as index



Example: process sales data

- **input file:**

Sales

1	clothing	3141
1	computers	9161
1	textbooks	21312
2	clothing	3252
2	computers	12321
2	supplies	2242
2	textbooks	15462

- **output:**
  - **summary of category sales**

Illustration: process each input line

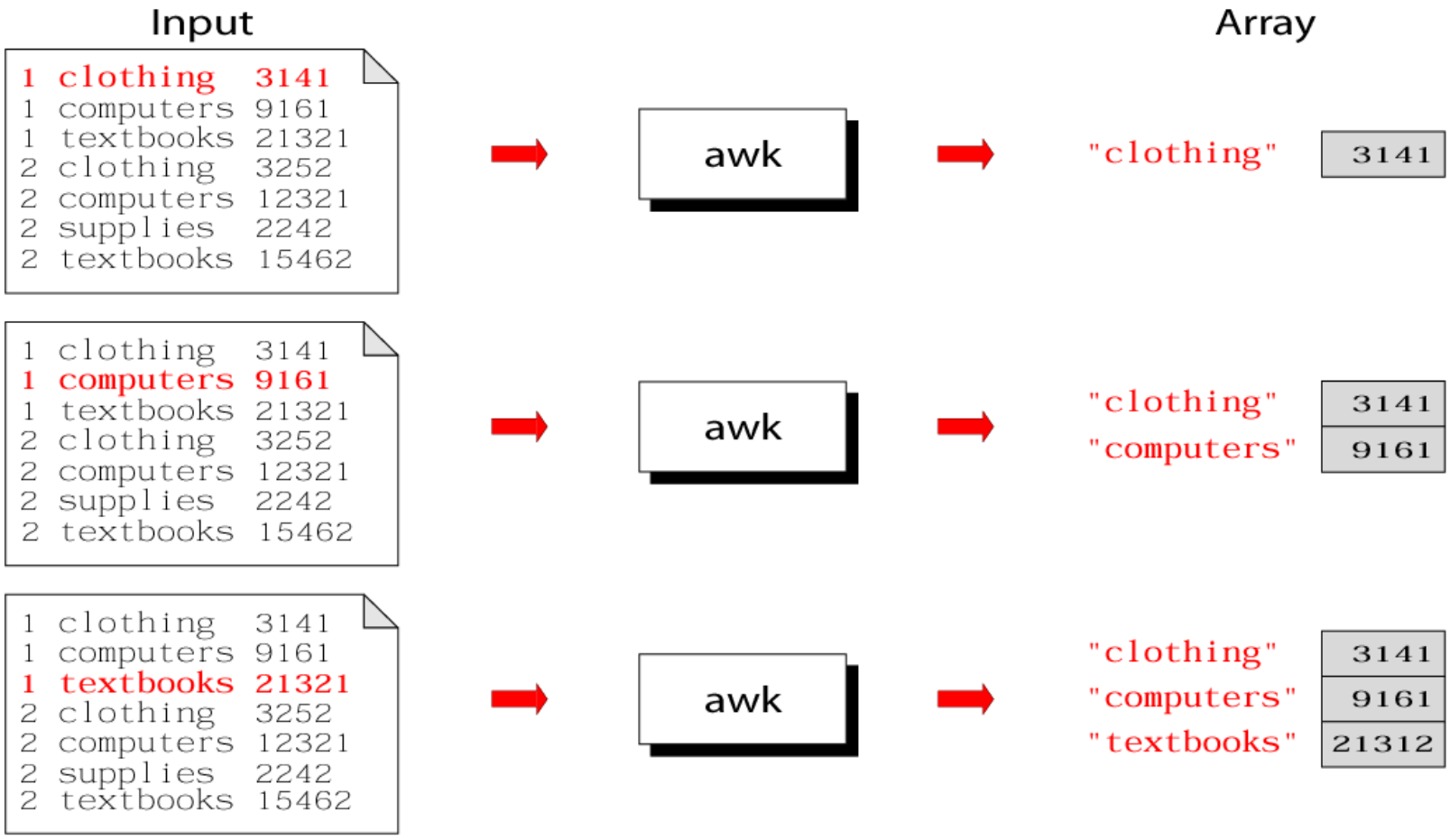
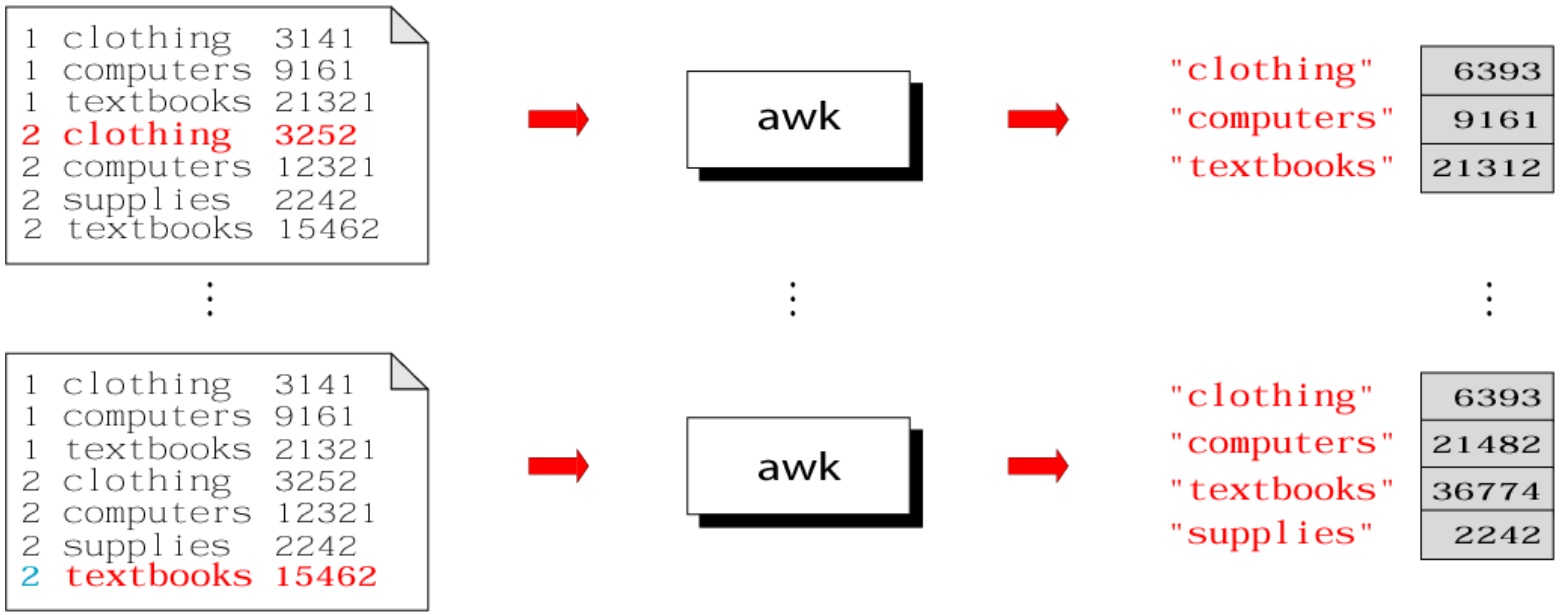


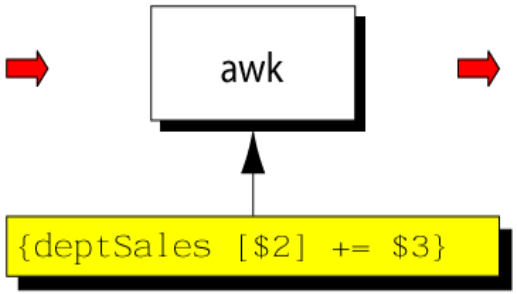
Illustration: process each input line



Summary: awk program

Sales

1	clothing	3141
1	computers	9161
1	textbooks	21312
2	clothing	3252
2	computers	12321
2	supplies	2242
2	textbooks	15462



"clothing"	6393
"computers"	21482
"textbooks"	36774
"supplies"	2242

deptSales

Example: complete program

```
% cat sales.awk
{
        deptSales[$2] += $3
}
END {
        for (x in deptSales)
                print x, deptSales[x]
}
% awk -f sales.awk sales
```

awk built-in functions

- **Arithmetic**

  `sqrt, rand`

- **String**

  `index, length, split, substr, sprintf, tolower, toupper`

- **Misc**

  `system, systime`

awk builtin split function

`split(string, array, fieldsep)`

- divides string into pieces separated by fieldsep,
- stores the pieces in array
- if the fieldsep is omitted, the value of FS is used.

Example:

```
split("26:Miller:Comedian", a, ":")
```

- sets the contents of the array a as follows:

```
a[1] = "26"
a[2] = "Miller"
a[3] = "Comedian"
```

awk control structures

- (Typically used in END section of awk script)
- Conditional
  - if-else
- Repetition
  - for
    - with counter
    - with array index
  - while
  - do-while
  - also: break, continue

if Statement

Syntax:

```
if (conditional expression)
   statement-1
else
   statement-2
```

Example:

```
if ( NR < 3 )
   print $2
else
   print $3
```

If statement for arrays

- **Syntax:**

```
if (value in array)
   statement-1
else
   statement-2
```

- **Example:**

```
if ("clothing" in deptSales)
   print deptSales["clothing"]
else
   print "not found"
```

for Loop

Syntax:

```
for (initialization; limit-test; update)

        statement
```

Example:

```
for (i=1; i <= 10; i++) {

    print "The square of ", i, " is ", i*i

}
```

for Loop for arrays

Syntax:

```
for (var in array)

        statement
```

Example:

```
for (x in deptSales) {

   print x, deptSales[x]

}
```

while Loop

Syntax:

```
while (logical expression)
        statement
```

Example:

```
i=1
while (i <= 10) {
   print "The square of ", i, " is ", i*i
   i = i+1
}
```

do-while Loop

Syntax:

```
do
    statement
while (condition)
```

- statement is executed at least once, even if condition is false at the beginning

Example:

```
i = 1
i=1
do {
  print $0
  i++
} while (i <= 5)
```

```
do {
   print $i
   i++
} while (i <=
```

loop control statements

- `break`

  exits loop

- `continue`

  skips rest of current iteration, continues with next iteration

Example: sensor data file

```
1 Temperature
2 Rainfall
3 Snowfall
4 Windspeed
5 Winddirection
```

- also: sensor readings
- Plan: print average readings in descending order

Example: sensor readings file

```
2008-10-01/1/68
2008-10-02/2/6
2007-10-03/3/4
2008-10-04/4/25
2008-10-05/5/120
2008-10-01/1/89
2007-10-01/4/35
2008-11-01/5/360
2008-10-01/1/45
2007-12-01/1/61
2008-10-10/1/32
```

Report: average readings, sorted

Sensor	Average
Winddirection	240.00
Temperature	59.00
Windspeed	30.00
Rainfall	6.00
Snowfall	4.00

Step 1: print sensor data

```
BEGIN {
   printf("id\tSensor\n")
   printf("-----------------------\n")
}
{
  printf("%d\t%s\n", $1, $2)
}
```

Step 2: print sensor readings

```
BEGIN {
   FS="/"
   printf(" Date\t\tValue\n")
   printf("----------------------\n")
}
{
   printf("%s     %7.2f\n", $1, $3)
}
```

Step 3: print sensor summary

```
BEGIN {
   FS="/"
}
{
   sum[$2] += $3
   count[$2]++
}
END {
 for (i in sum) {
  printf("%d %7.2f\n",i,sum[i]/count[i])
 }
}
```

Next steps: Remaining tasks

- `awk -f sense.awk sensors readings`

```
          Sensor Average
----------------------
  Winddirection  240.00
    Temperature   59.00
      Windspeed   30.00
       Rainfall    6.00
       Snowfall    4.00
```

2 input files

sorted

sensor names

Example: print sensor averages

- Remaining tasks:
  - recognize nature of input data
    use: number of fields in record
  - substitute sensor id with sensor name
    use: associative array
  - sort readings
    use: sort –nr –k 2

Example: sense.awk

```
NF > 1 {
    name[$1] = $2
}
NF < 2 {
    split($0,fields,"/")
    sum[fields[2]] += fields[3]
    count[fields[2]]++
}
END {
    for (i in sum) {
      printf("%15s %7.2f\n", name[i],
    sum[i]/count[i]) | "sort -nr -k 2"
    }
}
```