UCS1304 UNIX and Shell Programming

AWK

B.E. CSE B, Semester 3 (2019-2020)

S Milton Rajendram

Department of Computer Science and Engineering

SSN College of Engineering

14 August 2019

1. AWK

- Powerful filter
- Programming language

2. Structure of AWK Program

- AWK program is a sequence of pattern-action statements

```
pattern { action }
pattern { action }
...
```

- `action` is a sequence of statements

```
pattern {
            statement
            statement
            ...
        }
pattern {
            statement
            statement
```

```
        }   ...
...
```

- Input: records and fields

```
awk '$3 > 0 {print $1, $2 * $3}' files/emp.data
```

```
awk '$3 == 0 {print $1}' files/emp.data
```

- Single-quote awk program
  - '$' is also a shell metacharacter.
  - Multiple lines
- Either pattern or action (not both) may be omittted.
  - Omit action

    ```
    awk '$3 == 0' files/emp.data
    ```
  - Omit pattern

```
awk '{print $1}' files/emp.data
```

- Process
```
for each input line:
  for each pattern-action statement:
     if the line matches the pattern:
        apply action on line
```
  - Pattern may be a condition – matching line means condition is true

3. Running an AWK Program

```
awk 'program' files
```

- Multiple input files

  ```
  awk '{print $1}' files/emp.data files/distros.txt
  ```

- stdin

  ```
  awk '{print $1}'
  ```

- Program file = script

  ```
  awk -f scriptfile files
  ```

4. Simple Use

- Two data types: string, numbers
- Each line is a sequence of fields – blanks/tabs is field separator
- Read one line at a time, split it into fields
- Fields are numbered $1, $2, ...
- $0 is the entire line

4.1. Print every line

```
awk {'print}' emp.data
```

```
awk '{ print $0 }' emp.data
```

4.2. Print certain fields

```
awk '{ print $1, $3}' emp.data
```

Fields printed are separated by single blanks, lines by new lines.

4.3. Number of fields

- `NF` built-in variable

```
awk '{ print NF, $1, $NF }' emp.data
```

4.4. Computing and printing

```
awk '{ print $1, $2 * $3 }' emp.data
```

4.5. Line numbers

- `NR` Number of records
- Number every line

```
awk '{ print NR, $0 }'
```

4.6. Putting text in the output

```
awk '{ print "total pay for", $1, "is", $2 * $3 }' emp.data
```

5. Formatted Output

- `printf` similar to C

```
awk '{ printf("total pay for %s is $%.2f\n", $1, $2 * $3) }'   emp
```

- No blanks or newlines are produced automatically. We must explicitly print them.

```
awk '{ printf("%-8s $%6.2f\n", $1, $2 * $3) }' emp.data
```

- Sorting the output

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

6. Selection

- Without patterns, action is on all lines
- Pattern selects lines

6.1. Selection by comparison

```
awk '$2 >= 5' emp.data
```

6.2. Selection by computation

```
awk '$2 * $3 > 50 { printf("$%.2f for %s\n", $2 * $3, $1) }'
```

6.3. Selection by text content

- Select lines in which the first field is "Susie"

```
awk '$1 == "Susie"'
```

- Select "matching" lines by RE – lines that contain "Susie" anywhere

```
awk '$1 == /Susie/'
```

6.4. Combination of patterns

- Logical operators

AND	&&
OR	||
NOT	!

```
awk '$2 >= 4 && $3 >= 20' emp.data
```

-

```
awk -e '$2 >= 4' -e '$3 >= 20' emp.data
```

```
awk '!($2 < 4 || $3 < 20' emp.data
```

6.5. Data validation

6.6. BEGIN and END

- Special patterns
- BEGIN matches before the first line of the first input file is read
- END matches after the last line of the last file has been processed.

```
awk -e 'BEGIN { print "NAME RATE HOURS"; print "" }' -e '{ print
```

7. Computing wih AWK

- Action is a sequence of statements, separated by newlines or semicolons.

7.1. Counting

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

7.2. Computing sums and averages

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

```
    }
```

7.3. Handling text

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

7.4. String concatenation

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

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

7.5. Printing the last line

NR is global, $0 is not.

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

7.6. Built-in functions

- Math: square root, log, random
- String: length

```
{ print $1, length($1) }
```

7.7. Counting lines, words, and characters

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

- $0 does not include newline.

8. Control-Flow Statements

- Modeled on C

8.1. If-Else statement

```
$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"
       }
```

We can break a statement after a comma and continue the statement.

8.2. While statement

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

8.3. For statement

```
# interest2 - compute compound interest
# input: amount rate years
```

```
# output: compounded value at the end of each year
{ for (i=1;i<=$3;i=i+1)
      printf("\t%6.2f\n", $1 * (1 + $2) ^ i)
}
```

9. Arrays

```
# reverse - print input in reverse order by line
    { line[NR] = $0 }
    # print lines in reverse order
END { i = NR
      while (i > 0) {
         print line[i]
         i = i - 1
      }
    }

# reverse - print input in reverse order by line
        { line[NR] = $0 }
    END { for (i = NR; i > 0; i = i - 1)
              print line[i]
```

```
}
```