Praktikum 7

Patterns, Actions, and Variables

Tujuan Pembelajaran

Mahasiswa dapat memahami dan menggunakan *pattern*, *actions* dan *variable* dalam bahasa pemrograman awk.

Dasar Teori

Seperti Anda ketahui bersama, bahwa setiap statement pada bahasa pemrograman awk terdiri dari sebuah pola (*pattern*) yang secara langsung diiringi dengan aksi (*actions*). Pada Bab ini akan menjelaskan bagaimana anda membangun *patterns* dan *actions*. Apa yang dapat Anda buat dengan *actions* dan *built-in variabel* pada awk. Aturan *pattern-action* dan statement tersedia untuk digunakan pada bentuk *actions* dari inti program awk. Semuanya disini merupakan dasar dari program yang akan dibangun diatasnya. Sekarang saatnya membuat sesuatu yang lebih berguna.

Pattern merupakan sebuah bentuk statement. Pattern bisa terdisi dari regular expression, BEGIN-END, dan empty. Setiap pattern mempunyai bertuk dan fungsi tersendiri. Di dalam sebuah pattern terdapat *action*. *Action* tersebut yang mengontrol jalannya program. Statement kontrol yang terdapat dalam awk antara lain: if-else, while, do-while, dan for. Masing-masing kontrol memiliki cara yang berbeda dalam mengontrol jalannya program, namun pada dasarnya sama yaitu melakukan sesuatu berdasarkan kondisi yang diberikan apakan sesuai (bernilai benar) atau tidak (bernilai salah). *Variabel* program awk dapat diambil dari shell atau command-line. Terdapat dua cara yaitu dengan menggunakan metode shell qouting dan mengunakan argumen -v. Metode singgle quoting menggunakan bagian program tersendiri yang dimasukkan dalam tanda double-

quote. Penggunaan argumen -v memungkinkan untuk mendefinisikan variabel yang nilainya dapat diambil dari coommand line.

Percobaan 1: Pattern Elements

Semua expression pada awk merupakan sebuah *pattern*. Pattern akan cocok jika nilai ekspresinya tidak nol (jika merupakan angka) dan tidak null (jika sebuah string). Expression ini dievaluasi setiap menemukan input record baru. Berikut ini beberapa rangkuman jenis-jenis pattern di awk:

/regular expression/ : Sebuah regular expression akan cocok jika teks dari input record berada dalam lingkup regular expression.

ekspression : Single expression cocok jika nilainya tidak nol (jika merupakan angka) dan tidak null (jika sebuah string).

pat1, pat2 : Pasangan pola dipisahkan oleh sebuah coma, yang menunjukkan jangkauan (*range*) dari records. Jangkauannya dituliskan pada kedua record awal (*initial record*) yang cocok dengan *pat1* dan (*final record*) yang cocok dengan *pat2*.

BEGIN … END : Pattern yang spesial untuk memberikan *start up* atau *clean up actions* pada program awk.

BEGINFILE … ENDFILE : Pattern yang spesial untuk memberikan *start up* atau *clean up actions* untuk dikerjakan pada tiap dasar (*file basis*).

empty : Pola kosong yang cocok dengan setiap input record.

1. Regular expression sebagai pattern.

Regular expression adalah salah satu jenis pola yang dijelaskan sebelumnya. Polanya merupakan konstanta regex yang memiliki aturan pola yang sederhana. Artinya pernyataan berikut:

```
$0 ~ /pattens/
```

Menunjukkan pola input yang cocok dengan regex. Contohnya:

```
/foo|bar|baz/ { buzzwords++ }
END { print buzzwords, "buzzwords seen" }
```

…

2. Expression sebagai pattern.

Kebanyakan ekspresi awk adalah benar (valid) sebagai pattern awk. Pattern akan mencocokkan jika nilai dari ekspresi tikda nol (jika angka) atau tidak *null* (jika string). Ekspresi tersebut akan di evaluasi kembali setiap aturannya dicoba dengan input record yang baru. Jika ekspresi tersebut menggunakan *field* seperti $1, nilainya tergantung secara langsung pada teks input record yang baru. Jika tidak, hal ini tergantung hanya pada yang terjadi saat eksekusi program awk.

Membandingkan ekspresi, menggunakan pembandingan operator (yang dijelaskan pada bab sebelumnya) adalah pola yang sangat umum. Pencocokkan dengan menggunakan regex juga merupakan ekspresi yang sangat umum. Operand kiri (*left operand*) dari operator '~' dan '!~' adalah sebuah string. Operand kanan (*right operand*) adalah konstanta regex lainnya yang ditutup dengan slash (/regex/), atau ekspresi lainnya yang nilai string digunakan sebagai dynamic regular expression.

Program berikut akan mencetak field ke-2 dari setiap input record yang field pertamanya adalah tepat 'foo':

```
$ awk '$1 == "foo" { print $2 }' BBS-list
```

Program diatas menggunakan *field* sebagai *expression* notasi '$1'. Tidak ada ouput yang dihasilkan karena expression pada percobaan ini '$1 == "foo"' hanya cocok ketika field pertama bernilai "foo" dan pada file BBS-list tidak ada field pertama yang bernilai "foo". Coba kita bandingkan dengan ekpresi regex yang menggunakan '/foo/' :

```
$ awk '$1 ~ "/foo/" { print $2 }' BBS-list
```

Program diatas menunjukkan bahwa *regular expression* juga merupakan expression. Operand yang berada di sebelah kiri dari '~' atau '!~' adalah operator yang berupa string, sedangkan operand yang berada di sebelah kanannya adalah regex yang berupa konstanta string yang diapit karakter slash (/foo/).

Boolean expression juga merupakan pattern yang umum. Kapan saat pattern cocok dengan input record, tergantung pada subekspresi-nya cocok. Program di bawah ini membandingkan input record dengan sub-expression dan akan mencetak semua record pada input file 'BBS-list' yang berisi '2400' dan 'foo'.

```
$ awk '/2400/ && /foo/' BBS-list
```

Contoh berikut akan mencetak semua record yang mengandung '2400' atau 'foo'.

```
$ awk '/2400/ || /foo/' BBS-list
```

Sedangkan program di bawah ini mencetak semua record pada input file 'BBS-list' yang di dalamnya tidak terdapat 'foo'.

```
$ awk '! /foo/' BBS-list
```

Sub-ekspresi dari operator boolean dalam sebuah pattern dapat berupa konstanta regex, pembanding (comparison) atau jenis ekspresi awk lainnya. Jangkauan pattern adalah bukan ekspresi, jadi mereka tidak dapat muncul dalam boolean pattern. Begitu juga, spesial pattern seperti BEGIN, END, BEGINFILE dan ENDFILE yang tidak pernah cocok dengan semua input record yang bukan ekspresi dan tidak muncul didalam boolean pattern.

3. Mendefinisikan jangkauan record (*record ranges*) dengan pattern

Program di bawah ini menggunakan pattern dengan range tertentu. Range pattern terdiri dari dua pattern yang dipisahkan dengan koma (,) dengan aturan sebagai berikut:

```
begpat , endpat
```

Bentuk di atas digunakan untuk mencocokkan jangkauan dari input record yang berturut-turut. Pola pertama *begpat* mengkontrol dimana jangkauan (*range*) mulai, sedangkan *endpat* mengatur dimana pattern berakhir. Berikut ini adalah sebuah file yang bernama “myfile.txt”

```
online: 1 user
offline: 5 users
on my computer
```

Kemudian jalankan program berikut ini:

```
$ awk '$1 == "on", $1 == "off"' myfile
```

Program diatas akan mencetak semua record diantara *begpat* yaitu $1==“on” dan $1==“off” dari file input “myfile.txt” yang cocok dengan jangkauan pattern tersebut.

Range pattern dimulai dengan mencocokkan *begpat* dengan setiap input record. Ketika sebuah record cocok dengan *begpat*, maka *range pattern* diaktifkan dan mencocokkan recordnya. Sepanjang *range pattern* aktif, secara

otomatis mencocokkan setiap input record yang dibaca. Sambil *range pattern* juga mencocokkan *endpat* dengan setiap input record yang dibaca. Ketika hal ini berhasil, range pattern di-non-aktif-kan untuk record setelahnya. Kemudian range pattern kembali untuk melakukan cek terhadap *begpat* pada setiap recordnya dan begitu seterusnya.

Record yang aktif pada *range pattern* dan record yang menonaktifkannya, keduanya dicocokkan dengan *range pattern*. Jika Anda tidak ingin bekerja dengan record ini, Anda dapat menulis *if-statement* dalam aturan *action* untuk membedakan mereka dari record yang Anda maksud.

4. Special Pattern BEGIN dan END

Semua pattern yang dijelaskan sejauh ini digunakan untuk mencocokkan input record. BEGIN dan END adalah pola khusus yang berbeda dengan yang lainnya. Keduanya digunakan untuk persiapan (*startup*) sebelum program dijalankan dan pembersihan (*clean-up*) setelah program awk dijalankan. BEGIN dan END harus memiliki *action*, tidak ada standar untuk *action* pada aturan ini, karena tidak ada record saat ini ketika keduanya jalankan. BEGIN dan END biasanya diartikan yang ssebagai "BEGIN dan END blok" oleh programmer awk yang lama.

a) Startup and Cleanup Actions

Aturan BEGIN dieksekusi hanya satu kali, sebelum input record pertama dibaca. Begitu juga dengan aturan END, dieksekusi satu kali setelah semua input dibaca. Contohnya program berikut ini yang akan mencari jumlah record pada file ‘BBS-list’ yang berisi string ‘foo’.

```
$ awk 'BEGIN {
       print "Analysis of \"foo\"" }
       /foo/ { ++n }
  END { print "\"foo\" appears", n, "times." }' BBS-list
```

Blok BEGIN mencetak judul report. Dalam awk, inisialisasi awal variabel n sama dengan nol tidak perlu dilakukan karena sudah secara otomatis nilai awalnya bernilai nol. Ketika menemukan record yang sesuai dengan pattern 'foo' maka nilai n ditambah satu. END mencetak nilai akhir dari n yang menunjukkan jumlah record yang berisi 'foo'. BEGIN dieksekusi hanya satu kali, sebelum record input pertama dibaca. Begitu juga dengan END hanya dieksekusi satu kali, setelah semua input dibaca.

b) Input/Output Actions

Ada beberapa hal yang perlu diingat ketika melakukan action I/O dari aturan BEGIN atau END.

- Yang pertama yang harus diperhatikan adalah penggunaan nilai $0 dalam aturan BEGIN. Karena aturan BEGIN dijalankan sebelum input dibaca, tidak ada input record dan karena itu tidak ada field, ketika menjalankan aturan BEGIN.
- Yang kedua sama seperti yang pertama, tetapi dari arah lainnya. Kebiasaanya, untuk penggunaan skala besar, $0 dan NF adalah tidak didefinisikan didalam aturan END. Standar POSIX mendefinisikan bahwa NF ada di dalam aturan END yang isinya adalah jumlah field dari input record terakhir, secara logika begitu juga $0 dianggap ada pada aturan END. Kenyataannya gawk tidak menyediakan $0 pada aturan END.
- Yang ketiga, arti dari 'print' di dalam aturan BEGIN atau END adalah sama dengan 'print $0'. Jika $0 adalah null string, lalu ini akan mencetak record kosong.
- Yang terakhir pernyataan 'next' dan 'nextfile' tidak diijinkan di dalam aturan BEGIN, karena secara implisit melakukan perulangan dalam membaca record dan mencocokkannya dengan aturan (*read-a-record-and-match-against-the-rules*) tidak akan dimulai. Hal ini sama untuk aturan END, untuk semua input yang sudah dibaca.

5. Spesial Pattern BEGINFILE dan ENDFILE

Aturan BEGINFILE dan ENDFILE memberikan Anda "*hooks*" cara untuk mengaitkan *ke command-line file processing* pada gawk. Seperti pada aturan BEGIN dan END, semua aturan BEGINFILE pada sebuah program digabungkan dengan tujuan dibaca oleh gawk, dan semua aturan di dalam ENDFILE digabungkan juga. Bagian dari aturan BEGINFILE dieksekusi sebelum gawk membaca record pertama dari sebuah file. FILENAME di-set dengan nama file sekarang, dan FNR di-set ke nol.

Aturan BEGINFILE menyediakan Anda kesempatan untuk menyelesaikan 2 tugas yang sulit atau tidak mungkin untuk melakukan hal-hal berikut:

- Jika file dapat dibaca, maka Anda dapat mencobanya. Biasanya, fatal error terjadi jika nama file pada command line tidak bisa dibuka/dibaca. Tetapi, Anda dapat melakukan melewati (*bypass*) fatal error tersebut dan melanjutkan ke file selanjutnya pada command-line.
- Jika Anda telah menulis *extensions* yang memodifikasi penanganan record, Anda dapat membangunkan mereka pada titik ini, sebelum gawk semulai melanjutkan pemrosesan pada file.

Aturan ENDFILE dipanggil ketika gawk telah menyelesaikan proses pada record terakhir pada input file. Untuk input file terakhir, akan dipanggil sebelum aturan END. Aturan ENDFILE tetap dieksekusi meskipun untuk input file yang kosong. Selanjutnya akan dibahas pada subbab berikutnya.

5. Pattern Kosong (*Empty Pattern*)

Sebuah pattern yang kosong adalah dipertimbangkan untuk cocok dengan setiap input record. Contohnya program berikut ini:

```
$ awk '{ print $1 }' BBS-list
```

Program diatas akan mencetak field pertama pada setiap record.

Percobaan 2: Menggunakan *Shell Variables* pada Program

Program awk biasanya sering menggunakan komponen untuk program yang besar, yang ditulis di dalam shell. Contohnya, sangat umum untuk menggunakan varibel shell untuk memegang (*hold*) sebuah pattern yang akan dicari oleh program awk. Ada 2 cara untuk mendapatkan nilai dari varibel shell ke dalam program awk.

Cara yang paling umum adalah dengan menggunakan *shell quoting.* Shell quoting adalah metode yang paling sering digunakan untuk mengganti nilai variabel menjadi program dalam sebuah script. Contohnya program berikut ini:

```
$ printf "Enter search pattern:"
$ read pattern
  foo
$ awk "/$pattern/ "'{ nmatches++ }
  END { print nmatches, "found" }' BBS-list
```

Program diatas tadi menggunakan metode shell-quoting. Program awk di atas terdiri dari dua bagian. Bagian pertama yang diapit oleh *double quote*, mengijinkan variabel *pattern* untuk masuk dan menggantikan variabel tersebut. Bagian kedua adalah yang diapit tanda *single-quote* yang merupakan program utama.

Sedangkan pada program berikut ini hanya terdiri dari satu bagian. Dengan menggunakan argumen '-v' yang memungkinkan untuk mendefinisikan variabel di luar dari program utama awk.

```
$ printf "Enter search pattern:"
$ read pattern
  foo
$ awk -v pat="$pattern" '$0 ~ pat { nmatches++ }
  END { print nmatches, "found" }' BBS-list
```

Sekarang program awk hanya terdiri dari satu *sigle-quoted string*. Assignment '-v pat="$pattern"' masih membutuhkan *double-quotes*. Pada kasus tertentu juka ada karakter *whitespace* sebagai nilai dari $pattern. Variabel awk *pat* dapat diberina nama pattern juga (tetapi akan membingungkan). Menggunakan sebuah varibel dapat membuat lebih fleksibel, dengan catatan bahwa variabel tersebut dapat digunakan didalam program (printing, array subscript, atau lainnya) tanpa membutuhkan model quote pada program.

Percobaan 3: Pernyataan *Control* pada *Actions*

Pernyataan *control* seperti *if, while* dan sebagainya mengatur alur dari eksekusi program pada awk. Kebanyakan pernyataan *control* mirip seperti bahasa pemrograman C. Semua pernyataan control dimulai dengan keyword seperti *if* dan *while* untuk membedakan dengan ekspresi yang lainnya. Banyak pernyataan control terdiri dari pernyataan-pernyataan lainnya. Contohnya pernyataan *if* terdiri dari banyak pernyataan lainnya yang mungkin dieksekusi, mungkin juga tidak dieksekusi. Pernyataan di dalam statement disebut dengan "*body*", termasuk jika lebih dari satu pernyataan di dalam "*body*", dengan demikian beberapa pernyataan dapat dikelompokkan ke dalam satu pernyataan yang dipisahkan oleh *newlines* atau *semicolon*.

a) *if – statements*

Berikut ini akan dibahas mengenai statement *if-else*. Satement *if-else* tersebut melakukan kontrol berdasarkan nilai kondisi yang diberikan. Jika kondisi bernilai *true*, maka bagian *then-body* akan dieksekusi, sedangkan jika *false* maka bagian *else-body* yang akan dieksekusi. Kondisi bernilai *false* jika nilainya adalah nol atau null (string), selain itu kondisi bernilai benar. Pada program berikut ini, expression 'x%2==0' bernilai true, karena nilai x adalah 16 sehingga bagian then-body dieksekusi yaitu menampilkan pesan "x is even".

```
$ awk 'BEGIN {
```

```
        x=16
        if(x%2==0) print "x is even"
        else print "x is odd"
}'
```

b) *while – statements*

Berikut ini akan dibahas mengenai statement *while*. while merupakan statement looping yang paling sederhana pada awk. Statement tersebut akan melakukan pengulangan selama kondisinya bernilai *true*. Isi dari *while* satement pada program di bawah ini adalah gabungan dari beberapa statement yang berada dalam tanda kurung kurawal. Statement pertama mencetak *field* ke-*i* dan statement ke-dua menambah nilai *i* sebesar satu. Selama *i* kurang dari atau sama dengan 3 maka statement tersebut akan di loop.

```
$ awk '{ i=1
          while (i <= 3)
              print $i
              i++ }
}' inventory-shipped
```

c) *do while – statements*

Berikut ini akan dibahas mengenai statement *do-while*. Bentuk *do-while* hampir sama dengan bentuk perulangan *while*. Hanya saja loop pertama dari *do-while* langsung dieksekusi tanpa melihat kondisi, dan baru akan dilakukan looping jika kondisi terpenuhi (bernilai benar). Program berikut ini akan mencetak setiap record input sebanyak 10 kali.

```
$ awk '{ i=1
          do { print $1 i++ }
          while (i <= 10)
}' inventory-shipped
```

d) *for – statements*

Berikut ini akan dibahas mengenai statement *for*. Satement *for* dimulai dari mengeksekusi inisialisasi. Kemudian, selama kondisinya bernilai benar, maka looping akan dilakukan serta melakukan increment. Program berikut ini akan mencetak tiga field pertama dari setiap record input secara perbaris.

```
$ awk '{ i=1
         for (i=1;i<=3;i++)
             print $i
}' inventory-shipped
```

e) *switch case – statements*

Berikut ini akan dibahas mengenai pernyataan *switch case*. Pernyataan ini mengijinkan untuk melakukan eksekusi berdasarkan *case* yang sama. Dilakukan pengecekan terhadap semua pernyataan *case* sesuai urutan, jika ada yang cocok maka akan dieksekusi. Jika tidak ada yang cocok, bagian *default* akan dieksekusi. Setiap *case* dapat berisi konstanta baik numerik, string, maupun regex. Contoh dibawah ini menunjukkan penggunaan switch-case pada NR.

```
$ awk '{switch (NR * 2 + 1) {
            case 3:
            case "11":
                  print NR - 1
                  break
            case /2[[:digit:]]+/:
                  print NR
            default:
                  print NR + 1
            case -1:
                  print NR * -1
}' inventory-shipped
```

Pada contoh di atas ada *case -1* dieksekusi ketika *default* tidak dapat berhenti eksekusi.

f) *break – statements*

Berikut ini akan dibahas mengenai pernyataan *break.* Pernyataan ini digunakan untuk keluar dari perulangan seperti *for, while*, *do-while* atau *switch*. Pernyataan *break* tidak ada artinya jika ditempatkan di luar perulangan. Contoh berikut ini menunjukkan cara untuk mencari pembagi paling kecil dari bilangan integer dan mengidentifikasi bilangan prima.

```
$ awk '{# find smallest divisor of num
            num = $1
            for (div = 2; div * div <= num; div++) {
                  if (num % div == 0)
                        break
            }
            if (num % div == 0)
                  printf "Smallest divisor of %d is %d\n", num, div
            else
                  printf "%d is prime\n", num
}'
```

g) *continue – statements*

Berikut ini akan dibahas mengenai pernyataan *continue.* Seperti pada pernyataan *break*, pernyataan *continue* digunakan di dalam perulangan seperti *for, while*, *do-while*. Jika break keluar dari looping, pernyataan *continue* ini digunakan untuk menjalankan looping berikutnya tanpa harus menjalankan seluruh body loop saat itu, hal ini menyebabkan perulangan berikutnya dilakukan secara langsung. Berikut ini adalah contohnya :

```
$ awk 'BEGIN {
      for (x = 0; x <= 20; x++) {
            if (x == 5)
                  continue
                  printf "%d ", x
      }
      print ""
}'
```

Atau contoh berikut ini:

```
$ awk 'BEGIN {
      x = 0
      while (x <= 20) {
            if (x == 5)
                  continue
                  printf "%d ", x
                  x++
      }
      print ""
}'
```

h) *next* dan *next file – statements*

Pernyataan *next* memaksa awk agar segera menghentikan proses pada record saat itu dan menuju ke record berikutnya. Dengan demikian tidak ada lagi pernyataan yang dieksekusi pada record yang dijalankan saat ini. Berikut ini contohnya :

```
$ awk 'NF != 4 {
      err = sprintf("%s:%d: skipped: NF != 4\n", FILENAME, FNR)
      print err > "/dev/stderr"
      next
}'
```

Selain pernyataan *next* ada pernyataan "*nextfile"* yang digunakan untuk menghentikan proses pada file yang saat itu sedang dijalankan, lalu berpindah menuju ke file berikutnya.

i) *exit - statement*

Pernyataan exit dapat menyebabkan awk berhenti mengeksekusi aturan yang sedang dijalankan saat ini, menghentikan pemrosesan input, sehingga input lainnya diabaikan. Berikut ini contohnya :

```
$ awk 'BEGIN {
      if (("date" | getline date_now) <= 0) {
            print "Can't get system date" > "/dev/stderr"
            exit 1
      }
      print "current date is", date_now
      close("date")
}'
```

Percobaan 4: Menggunakan Built-in Variables

Kebanyakan variabel awk digunakan untuk tujuan yang spesifik, dan tidak pernah dirubah kecuali program yang dibuat memberikan nilai tertentu. Hal ini tidak akan berefek apa-apa kecuali program memang dibuat untuk memeriksa variabel tersebut. Namun, awk memiliki beberapa variabel yang memiliki arti yang spesial, biasa disebut dengan "Built-in Variabel". Awk memeriksa variabel ini secara otomatis, jadi variabel ini dapat memungkinkan kita memberitahukan kepada awk untuk melakukan hal-hal tertentu. Yang lainnya di-set secara otomatis oleh awk. Sehingga dapat membawa informasi internal ketika program awk kita bekerja. Berikut ini akan dibahas beberapa "Built-in Variabel" yang ada pada awk :

a) "Built-in Variabel" untuk control awk

Berikut ini adalah "Built-in Variabel" yang dapat Anda gunakan untuk mengatur bagaimana awk melakukan beberapa hal. Jika menggunakan gawk ditandai dengan tanda pagar ('#').

- BINMODE # (*On non-POSIX systems, this variable specifies use of binary mode for all I/O.*).
- CONVFMT, (*This string controls conversion of numbers to strings*).
- FIELDWIDTHS # (*This is a space-separated list of columns that tells gawk how to split input with fixed columnar boundaries.*).
- FPAT # (*This is a regular expression (as a string) that tells gawk to create the fields based on text that matches the regular expression*).

- FS, (*This is the input field separator*).
- IGNORECASE # (*If IGNORECASE is nonzero or non-null, then all string comparisons and all regular expression matching are case independent.*).
- LINT # (*When this variable is true (nonzero or non-null), gawk behaves as if the –lint command-line option is in effect.*)
- OFMT (*This string controls conversion of numbers to strings for printing with the print statement*)
- OFS (*This is the output field separator*)
- ORS (*This is the output record separator*)
- RS (*This is awk's input record separator*)
- SUBSEP (*This is the subscript separator*)
- TEXTDOMAIN # (*This variable is used for internationalization of programs at the awk level*)

b) "Built-in Variabel" untuk memberikan informasi

Berikut ini adalah "Built-in Variabel" yang sudah di set secara otomatis sesuai dengan kondisinya untuk memberikan informasi pada program yang dibuat. Jika menggunakan gawk ditandai dengan tanda pagar ('#').

- ARGC, ARGV (*The command-line arguments available to awk programs are stored in an array called ARGV. ARGC is the number of command-line arguments present*.)
- ARGIND # (*The index in ARGV of the current file being processed*.)
- ENVIRON (*An associative array containing the values of the environment*.)
- ERRNO # (*If a system error occurs during a redirection for getline, during a read for getline, or during a close() operation, then ERRNO contains a string describing the error*.)
- FILENAME (*The name of the file that awk is currently reading*.)
- FNR (*The current record number in the current file*.)
- NF (*The number of fields in the current input record*.)
- NR (*The number of input records awk has processed since the beginning of the program's execution*)

- PROCINFO # (*The elements of this array provide access to information about the running awk program.)*
- RLENGTH (*The length of the substring matched by the match() function*)
- RSTART (*The start-index in characters of the substring that is matched by the match() function*)
- RT # (*This is set each time a record is read. It contains the input text that matched the text denoted by RS, the record separator*)

c) Penggunaan ARGC dan ARGV

CONFMT adalah built-in variabel yang mengontrol konversi akngka ke string. Default nilainya adalah "%.6g". Pada percobaan ini nilai CONFMT ditentukan "%2.2f". Variabel a adalah numerik. Ketika digabungkan dengan string "", maka hasilnya akan menjadi string.

```
$ awk 'BEGIN {
       CONVFMT = "%2.2f"
       a = 12
       b = a ""
       print b
}'
```

Di dalam awk, terdapat variabel ARGC dan ARGV. ARGV merupakan array yang menyimpan argumen command-line yang tersedia untuk program awk. ARGC adalah jumlah argumen command-line yang tersimpan pada saat ini. Pada perogram berikut ini, looping dimulai dari nol, karena ARGV berbeda dari array yang umum dalam awk. ARGV dimulai dari indeks nol. ARGV[0] berisi "awk", ARGV[1] berisi "inventory-shipped", dan ARGV[2] berisi "BBS-list". Nilai ARGC adalah 3.

```
$ awk 'BEGIN {
       for (i=0; i <ARGC; i++)
           print ARGV[i]
}' inventory-shipped BBS-list
```