Praktikum 6
Expression

Tujuan Pembelajaran

Mahasiswa dapat memahami dan menggunakan berbagai ekspresi dalam bahasa pemrograman awk.

Dasar Teori

Expression (ekspresi) adalah bagian yang mendasar dari suatu blok dari *pattern* dan *action* program awk. Sebuah *expression* mengevaluasi suatu nilai yang dapat dicetak, dites, atau dilimpahkan ke fungsi. Selain itu, sebuah *expression* dapat memberi nilai pada suatu variabel atau field dengan menggunakan operator *assignement*.

Sebuah *expression* dapat berperan sebagai *pattern* maupun *action* pada statement itu sendiri. Kebanyakan jenis statement berisi satu atau lebih *expression* yang menentukan data mana yang akan dioperasikan. Sebagaimana dalam bahasa pemrograman lain, *expression* dalam awk meliputi variable, array references, contants, dan function call, atau kombinasi dengan berbagai operasi yang beragam.

Percobaan 1: *Constants, Variables and Conversions*

Expression dibangun dari nilai dan operasi yang dilakukan pada nilai tersebut. Bagian ini menjelaskan bagian mendasar yang mengatur nilai-nilai yang digunakan dalam *expression*.

a. *Constant Expression*

Tipe yang paling sederhana dari *expression* adalah *constant* (konstanta) yang nilainya selalu tetap. Terdapat tiga tipe konstanta, yaitu: numerik, string, and *regular expression*. Masing-masing dapat digunakan dalam konteks yang tepat saat membutuhkan sebuah nilai data yang tidak ingin diubah. Setiap konstanta numerik dapat memiliki bentuk yang berbeda, tapi secara internal tersimpan dengan cara yang sama.

1) Konstanta Numerik dan String

Konstanta numerik terdiri dari angka-angka. Angka tersebut dan berupa integer, pecahan desimal, atau sebuah bilangan dalam notasi scientific (eksponensial). Berikut ini contoh ekspresi numerik yang memiliki nilai sama:

```
105        1.05e+2        1050e-1
```

Cobalah program berikut ini:

```
$ awk 'BEGIN { print 105, 1.05e+10, 1050e-1}'
```

Sebuah konstanta string berisi rangkaian karakter yang diapit oleh tanda double-quotation seperti pada contoh berikut ini:

```
"apel"     "melon"        "semangka"
```

Cobalah program berikut ini:

```
$ awk 'BEGIN { print "apel", "melon", "semangka"}'
```

Merepresentasikan string yang isinya adalah 'semangka'. String dalam awk dapat memiliki panjang berapapun, dan dapat berisi karakter ASCII 8-bit termasuk karakter ASCII NUL (karakter kode nol). Implementasi awk lainnya mungkin Anda menemui kesulitan dalam beberapa kode karakter.

2) Bilangan Octal dan Hexadecimal

Dalam awk, semua bilangan adalah dalam desimal (basis 10). Pemrograman awk mengijinkan programmer untuk menentukan basis bilangan lain. Bilangan oktal

dimulai dengan ‘0’, dan bilangan heksadesimal dimulai dengan ‘0x’ atau ‘0X’, seperti pada contoh berikut ini:

11	Decimal value 11.
011	Octal 11, decimal value 9.
0x11 atau 0X11	Hexadecimal 11, decimal value 17.

Cobalah program berikut ini:

```
$ awk ‘BEGIN { printf “%d, %d, %d\n”, 011, 11, 0x11 }’
```

Jika berkerja dengan data yang tidak dapat direpresentasikan dengan mudah sebagai karakter atau regular expression, penggunaan konstanta dalam oktal dan hexadecimal akan lebih mudah. Tidak seperti implementasi pada bahasa C, karakter ‘8’ dan ‘9’ adalah konstanta oktal yang tidak valid. Contoh program berikut ini memperlakukan ‘018’ seperti desimal 18 dengan perintah print:

```
$ awk ‘BEGIN { print “021 is”, 021 ; print 018 }’
```

Jika nilai oktal yang diberikan tidak valid, maka akan direpresentasikan dalam integer basis 10. Tampak 021 tetap menjadi 021 yaitu 17 desimal sedangkan 018 menjadi 18 desimal.

Setelah sebuah konstanta numerik sudah diubah secara internal ke dalam angka, awk tidak ingat lagi apa bentuk asli dari konstanta tersebut (nilai internal yang biasanya digunakan). Hal ini memiliki konsekuensi tertentu untuk konversi bilangan ke string. Berikut ini program untuk menampilkan bilangan heksadesimal yang disimpan dalam variabel string dengan menggunakan perintah printf:

```
$ awk ‘BEGIN { printf “0x11 is <%s>\n”, 0x11 }’
```

Tampak basis nilai input user tidak berpengaruh terhadap nilai internalnya.

3) Konstanta Regular Ekspression

Konstanta regex adalah regular ekspression yang ditulis di dalam slash, seperti

```
/^beginning and end$/
```

Kebanyakan regex yang digunakan pada program awk adalah termasuk dari konstanta, tetapi karakter ‘~’ dan ‘!~’ sebagai operator pembanding dapat juga digunakan untuk mencocokkan regex yang dihitung atau yang dinamis (meskipun hanya string biasa atau variabel yang terdiri dari regex). Selanjutnya akan dibahas lebih dalam lagi pada bab berikut ini.

b. Menggunakan Konstanta *Regular Expression*

Bila digunakan pada sisi sebelah kanan dari operator '~' atau '!~', sebuah konstanta regex artinya adalah regex yang akan dicocokkan. Namun, konstanta regex (seperti `/foo/`) dapat digunakan seperti kalimat sederhana. Ketika konstanta regex muncul dengan sendirinya, itu memiliki arti yang sama seperti muncul dalam pola lainnya seperti '`($0 ~ /foo/)`'. Artinya memiliki 2 bagian kode sebagai contoh baris program berikut:

```
$ awk '{
if ($0 ~ /barfly/ || $0 ~ /camelot/)
      print "found"
}' BBS-list
```

dan

```
$ awk '{
if (/barfly/ || /camelot/)
      print "found"
}' BBS-list
```

Kedua format di atas menghasilkan output yang sama, karena secara default expresi diatas akan membaca setiap record atau sama dengan $0. Program tersebut menggunakan regular expression yang diapit oleh dua karakter slash (‘/’). Regular expression tersebut akan cocok dengan record yang berisi ‘barfly’ atau ‘camelot’. Salah satu konsekuensi yang agak aneh dari peraturan ini adalah bahwa ekspresi Boolean berikut ini berlaku, tetapi tidak melakukan apa yang user maksudkan. Contohnya seperti pada program berikut:

```
$ awk '{
# Catatan /foo/ ada di sebelah kiri dari operator '~'
      if (/foo/ ~ $1)
      print "found foo"
}' BBS-list
```

Penggunaan regular expression di sebelah kiri operator ‘~’ tersebut menyebabkan error, karena penggunaan regular expression harus berada di sebelah kanan dari operator’~’.

c. Menggunakan Variabel

Variabel adalah cara menyimpan nilai-nilai pada program Anda untuk digunakan nanti pada bagian lain dari program Anda. Variabel dapat dimanipulasi seluruhnya dalam teks program, dan variabel juga dapat diisi nilainya pada baris perintah awk. Variabel membebaskan Anda untuk memberikan nama (*initialization*) pada nilai-nilai tertentu dan merujuk (*refers*) kepada mereka nanti. Variabel telah digunakan dalam banyak contoh. Nama variabel harus menjadi urutan huruf, angka, atau garis bawah, dan mungkin tidak dimulai dengan digit.

Hal yang juga penting adalah dalam penamaan variabel, karakter ‘a’ dan ‘A’ adalah variabel yang berbeda.

Nama variabel adalah ekspresi yang valid dengan sendirinya, nama tersebut mewakili nilai variabel saat ini. Variabel diberi nilai-nilai baru dengan operator *assignment*, operator *increment* dan operator *decrement.* Beberapa jenis variabel sudah dipesan (*built-in)* dengan nilai yang khusus, misalnya FS (*field separator*) dan NF (*number of field*). Variabel *built-in* dapat digunakan dan ditugaskan seperti pada variabel lainnya, tetapi nilai-nilainya juga digunakan atau diubah secara otomatis oleh awk. Nama semua variabel built-in adalah huruf besar.

Variabel dalam awk dapat diinisialisasi dengan nilai baik numerik atau string. Jenis nilai variabel dapat berubah selama ada dalam sebuah program. Secara default, variabel diinisialisasi ke string kosong, misalnya zero jika dikonversi ke angka. Tidak perlu secara eksplisit "menginisialisasi" variabel dalam awk, seperti apa yang Anda lakukan di C dan dalam kebanyakan bahasa pemrograman lainnya. Penggunaan variable dalam program awk tidak jauh berbeda dengan penggunaan variabel pada bahasa pemrogaman lainnya. Nilai variabel dapat diberikan dengan cara assigning menggunakan operator assingment(‘=’), seperti contoh berikut ini:

```
variable = text
```

Dengan demikian, variabel dapat diatur baik pada jalannya program awk atau antara input file. Ketika *assignment* didahului dengan opsi “-v”, seperti berikut:

```
-v variable=text
```

Disisi lain, assignment variabel dilakukan pada waktu di antara argumen input file, setelah proses argumen input file selesai. Berikut ini adalah program untuk mencetak nilai dari jumlah field (n) untuk semua input record.

```
$ awk ‘{ print $n }’ n=4 inventory-shipped n=2 BBS-list
```

Nilai n sebelum memproses file inventory-shipped diberi nilai 4 maka ketika diproses oleh fungsi print nilai $n = $4, sehingga yang dicetak adalah field ke-4. Setelah itu nilai n diberi nilai 2, maka saat memproses file BBS-list dalam fungsi print nilai field yang dicetak adalah field ke-2. Argumen command-line dibuat tersedia untuk pemeriksaan eksplisit oleh program awk dalam array ARGV. awk memproses *assignment* nilai-nilai pada command-line untuk *escape sequence.*

d. Konversi dari String ke Angka

String dapat dikonversi ke angka dan begitu juga angka dikonversi ke string, jika konteks program awk menuntut hal itu. Misalnya, jika string “foo” atau “bar”

dalam ekspresi “foo+bar” akan menjadi satu string, itu akan diubah menjadi angka dahulu sebelum penambahan dilakukan. Jika nilai numerik muncul dalam *string concatenation*, maka akan dikonversi ke string. Coba program berikut ini:

```
$ awk 'BEGIN {
  two = 2; three = 3;
  print (two three) + 4}'
```

Pada program di atas, nilai dari bilangan yang tersimpan dalam variabel two dan three saat diproses oleh fungsi print akan dikonversi ke menjadi string. Sehinga (two three) menjadi (23) bukan 2+3 atau 2*3. Namun ketika dioperasikan secara matematika dengan menggunakan operator aritmatika, maka string akan dikonversi menjadi angka (number). Sehingga hasilnya 23+4=27.

```
$ awk 'BEGIN {
  var1=2.3; var2=4
  print (var1 var2) }'

$ awk 'BEGIN {
  var1=2.3; var2=4
  print (var1 + var2) }'

$ awk 'BEGIN {
  var1=2.3; var2=4
  print (var1 var2) - 2 }'

$ awk 'BEGIN {
  var1=2xx; var2=4
  print (var1 var2) }'

$ awk 'BEGIN {
  var1=2xx; var2=4
  print (var1 var2) - 2 }'

$ awk 'BEGIN {
  var1=x2x; var2=4
  print (var1 var2) }'

$ awk 'BEGIN {
  var1=x2x; var2=4
  print (var1 var2) - 2}'
```

Tampak konversi untuk mengubah string menjadi angka menggunakan metode yang sama dengan parsing angka pada bahasa pemrograman lain. String akan sintesis nilainya dengan memilah karakter yang valid sebagai bilangan. Jika ditemukan karakter yang tidak valid sebagai angka (number) pada awal string maka tidak dikonversi ke angka, sedangkan jika ditemukan pada tengah atau akhir maka akan dikonversi sejauh karakter yang valid sebagai angka.

```
$ awk 'BEGIN {
  CONVFMT="%2.2f"
  a=12.11111111111
  b=a ""
```

```
  print b
}'

$ awk 'BEGIN {
  a=12.11111111111
  b=a ""
  print b
}'
```

CONVFMT adalah built-in variable yang mengontrol aturan pasti dalampeng-konversian angka ke string. Nilai defult dari CONVFMT adalah "%.gg", yang mencetak nilai dengan enam digit yang paling signifikan. Nilai CONVFMT dapat diubah dengan menginisialisasi menggunakan assignment. Format "%2.2f" menghasilkan dua digit yang paling signifikan

Standar POSIX mengatakan bahwa awk selalu menggunakan periode sebagai titik desimal ketika membaca kode sumber program awk, dan assignment variabel pada command-line. Namun, ketika menafsirkan input data, untuk output perintah *print* dan maupun *printf*, dan untuk konversi angka ke string, karakter titik desimal lokal yang digunakan. Berikut ini adalah beberapa contoh yang menunjukkan perbedaan perilaku, pada sistem GNU / Linux:

```
$ gawk 'BEGIN { printf "%g\n", 3.1415927 }'
= 3.14159

$ LC_ALL=en_DK gawk 'BEGIN {
  printf "%g\n", 3.1415927 }'
= 3,14159

$ echo 4,321 | gawk '{ print $1 + 1 }'
= 5

$ echo 4,321 | LC_ALL=en_DK gawk '{ print $1 + 1 }'
= 5,321
```

Bentuk lokal '*en_DK*' adalah untuk bahasa Inggris di Denmark, di mana koma bertindak sebagai titik pemisah desimal. Dalam bentuk lokal "C" yang normal, awk memperlakukan '4,321 'sebagai '4', sementara di lokal Denmark, itu diperlakukan sebagai jumlah penuh yaitu '4,321'.

Beberapa versi gawk sebelumnya memenuhi aspek standar. Namun, banyak pengguna pada bentuk lokal non-English mengeluh tentang perilaku ini, karena data mereka menggunakan titik sebagai tanda desimal, sehingga perilaku default dikembalikan menggunakan karakter titik sebagai tanda desimal. Anda dapat menggunaka *lc-numerik* untuk memaksa gawk menggunakan bentuk lokal karakter titik sebagai desimal. (gawk juga menggunakan bentuk lokal karakter titik sebagai desimal ketika berada di mode POSIX, baik melalui "–posix", atau POSIXLY_CORRECT *environtment variabel*).

Tabel berikut ini menggambarkan beberapa kasus di mana bentuk lokal untuk karakter titik sebagai desimal digunakan dan ketika period digunakan. Beberapa fitur ini belum dapat dijelaskan.

Feature	Default	--posix or --use-lc-numeric
`%'g`	Use locale	Use locale
`%g`	Use period	Use locale
`Input`	Use period	Use locale
`strtonum()`	Use period	Use locale

Standar formal modern dan standar representasi floating point IEEE dapat memiliki efek yang tidak biasa namun penting dalam perjalanan gawk mengkonversi beberapa nilai string khusus untuk angka.

Percobaan 2: Operator, bekerja dengan nilai (values)

Bagian berikut ini menjelaskan penggunaan operator yang menggunakan nilai – nilai yang tersimpan dalam konstanta atau variabel.

a. Operator Aritmetika

Bahasa pemrograman awk,menggunakan operator aritmetika yang biasa digunakan ketika mengevaluasi ekspresi. Semua operator aritmetik mengikuti aturan *precedence* yang normal dan bekerja seperti apa yang Anda harapkan. Berikut ini contoh file bernama *grades,* yang berisi daftar nama mahasiswa dan 3 jenis nilai testnya.

```
Pat          100    97     58
Sandy        84     72     93
Chris        72     92     89
```

Program berikut ini membaca file *grades* tersebut dan mencetak nilai rata-ratanya.

```
$ awk '{ sum = $2 + $3 + $4 ; avg = sum / 3
         print $1, avg }' grades
```

Berikut ini daftar operator aritmetika dalam awk, dengan urutan yang paling tinggi *precedence*-nya sampai yang paling rendah.

Operator	Keterangan
`- x`	Negation.
`+ x`	Unary plus; the expression is converted to a number.
`x ^ y` `x ** y`	Exponentiation; x raised to the y power. '2 ^ 3' has the value eight; the character sequence '**' is equivalent to '^'. The POSIX standard only specifies the use of '^' for exponentiation. For maximum portability, do not use the '**' operator.

`x * y`	Multiplication.
`x / y`	Division; because all numbers in awk are floating-point numbers, the result is not rounded to an integer—‘3 / 4’ has the value 0.75.
`x % y`	Remainder.
`x + y`	Addition.
`x - y`	Subtraction.

Operator unary plus dan minus memiliki presedence yang sama, operator perkalian dan pembagian memiliki presedence yang sama, begitu juga dengan operator penambahan dan pengurangan. Biasa terjadi kesalahan (khususnya pada programmer C) bahwa mereka lupa semua angka dalam awk adalah *floating-point* dan pembagian bilangan integer menghasilkan bilangan real (bukan integer).

Ketika menghitung *remainder* (sisa atau modulus) ‘x%y’, quotient tersebut dibulatkan menjadi nol ke integer dan dikalikan dengan y. Hasil ini dikurangi dari x; operasi ini kadang-kadang dikenal sebagai "trunc-mod." Hubungan berikut selalu berlaku:

```
b * int(a / b) + (a % b) == a
```

Kemungkinan lain yang tidak biasa adalah bagaimana jika nilai *x*-nya negatif.

```
-17 % 8 = -1
```

Dalam implementasi awk lainnya, *signed-ness* (tanda) sisanya mungkin berdasarkan jenis mesinnya (*machine-dependent*).

b. String Concatenation

Hanya ada satu operasi string yaitu “*concatenation”* (penggabungan). Operasi ini tidak memiliki operator tertentu untuk mewakilinya. Sebaliknya, *concatenation* dilakukan dengan menulis ekspresi samping satu sama lain, tanpa operator. Sebagai contoh coba program berikut ini:

```
$ awk ’{ print "Field number one: " $1 }’ BBS-list
```

Jika tanpa ada spasi setelah tanda ‘:’ pada string diatas, baris berjalan bersamaan. Contohnya pada program berikut ini:

```
$ awk ’{ print "Field number one: " $1 }’ BBS-list
```

Karena penggabungan string tidak memiliki operator yang eksplisit, perlu untuk memastikan bahwa hal itu terjadi pada waktu yang tepat dengan menggunakan tanda kurung untuk menyertakan item dengan tujuan menggabungkan. Sebagai contoh, Anda mungkin berharap bahwa kode bagian “file” berikut dan merangkai “name” :

```
$ print "something meaningful" > (file name)
```

Tanda kurung harus digunakan sekitar penggabungan string dalam semua tapi konteks yang paling umum, seperti pada sisi sebelah kanan dari '='. Hati-hati tentang jenis ekspresi yang digunakan dalam penggabungan string. Secara khusus, urutan evaluasi ekspresi yang digunakan untuk penggabungan string tidak didefinisikan dalam bahasa awk. Coba lihat contoh berikut ini:

```
$ awk 'BEGIN {
      a = "don't"
      print (a " " (a = "panic"))
}'
```

Hal diatas tidak dapat didefinisikan apakah *assignment* ke 'a' terjadi sebelum atau sesudah nilai yang diambil untuk memproduksi nilai penggabungan. Hasilnya bisa berupa '*don't panic*', atau '*panic panic*'. Precedence (urutan) dari *concatenation,* waktu digabungkan dengan operator biasanya *kontra-intuitif.* Coba program berikut ini:

```
$ awk 'BEGIN { print -12 " " -24 }'
```

Contoh diatas tidak menampilkan karakter spasi. Kemana spasi menghilang ? Jawabannya terletak pada kombinasi precedences operator dan aturan konversi otomatis awk itu. Bagaimana jika ingin mendapatkan hasil yang diinginkan menampilkan karakter spasi diantara dua karakter angka tersebut, cobalah contoh berikut ini:

```
$ awk 'BEGIN { print -12 " " (-24) }'
```

Contoh diatas memaksa (force) awk untuk memperlakukan karakter minus pada '-24' sebagai karakter unary. Mekanisme parsing ditunjukkan seperti berikut ini:

> –12 (" " – 24)
> –12 (0 – 24)
> –12 (–24)
> –12– 24

Seperti disebutkan sebelumnya, ketika melakukan penggabungan *parenthesize* dengan tanda kurung (…). Namun, Anda belum tentu yakin apa yang Anda dapatkan.

c. Ekspresi Assignment

Assignment adalah sebuah ekspresi yang menyimpan nilai ke dalam sebuah variabel. Sebagai contoh, mari kita memberikan sebuah nilai pada variabel z seperti contoh berikut ini:

```
z = 1
```

Setelah ekspresi diatas dieksekusi, variabel z memiliki nilai satu. Nilai z yang lama sebelum diberikan nilai yang baru akan dihapus. Assignment dapat menyimpan nilai string seperti contoh berikut ini:

```
$ awk 'BEGIN{ thing = "food"; predicate = "good";
       message = "this " thing " is " predicate;
       print message;
}'
```

Contoh diatas mengilustrasikan *string concatenation*. Tanda '=' disebut dengan operator assignment. Ini adalah cara yang sederhana untuk menunjukkan cara kerja operator assignment, karena nilai operand disimpan tanpa perubahan. Kebanyakan operator (*addition*, *concatenation*, dan lainnya) tidak memiliki efek kecuali menghitung sebuah nilai. Jika nilai tersebut tidak digunakan, maka tidak ada alasan untuk menggunakan operator. Operator *assignment* itu berbeda, ia tidak menghasilkan sebuah nilai, tetapi meskipun Anda menghiraukannya, *assignment* akan membuat dirinya dapat dirasakan melalui adanya perubahan variabel. Inilah yang disebut dengan efek samping (*side effect*).

Righthand operand (operand sebelah kiri) pada sebuah assignment tidak memerlukan variabel, tetapi juga dapat menjadi *field* atau elemen *array*. Ini semua disebut *Lvalues*, yang berarti mereka dapat muncul di sisi kiri dari operator *assignment*. *Righthand operand* bisa saja ekspresi apapun, menghasilkan nilai baru yang oleh assignment simpan ke dalam variabel yang sudah ditentukan, *field*, atau elemen array. (Nilai-nilai tersebut disebut *Rvalues*.)

Penting untuk dicatat bahwa di dalam awk, variabel tidak memiliki tipe data permanen. Tipe variabel adalah tipe yang sederhana yang berisi nilai apa yang terjadi saat ini. Pada program berikut ini, variabel 'foo' memiliki nilai numerik pada awalnya, dan nilai string pada akhirnya:

```
$ awk 'BEGIN{ foo = 1
              print foo
              foo = "bar"
              print foo }'
```

Ketika assignment kedua 'foo' diisi dengan nilai string, faktanya nilai angka sebelumnya sudah dilupakan. Nilai string yang tidak dimulai dengan digit atau angka, nilainya adalah nol. Setelah eksekusi kode berikut ini, nilai 'foo' adalah 5.

```
$ awk 'BEGIN{ foo = "a string"
              print foo
              foo = foo + 5
              print foo  }'
```

Catatan: menggunakan variabel sebagai nomor dan kemudian sebagai string dapat membingungkan dan seperti gaya pemrograman pemula. Dua contoh sebelumnya menggambarkan bagaimana awk bekerja, bukan bagaimana Anda harus menulis program Anda ☺.

Assignment adalah sebuah ekspresi, sehingga memiliki nilai-nilai yang sama seperti nilai yang diinisialisasikan. Dengan demikian, 'z=1' adalah sebuah ekspresi dengan satu nilai. Salah satu konsekuensi dari ini adalah bahwa Anda dapat menulis beberapa assignment secara bersama-sama, seperti contoh berikut:

```
$ awk 'BEGIN{ x = y = z = 5
              print x, y, z  }'
```

Contoh diatas menyimpan nilai 5 kedalam tiga variabel sekaligus (x, y, dan z). Hal ini dilakukan karena nilai dari 'z=5' disimpan ke dalam 'y' dan kemudian nilai adri 'y=z=5' disimpan ke dalam variabel 'x'.

Assignment dapat digunakan di mana saja ekspresi tersebut dipanggil. Sebagai contoh, dibenarkan untuk menulis 'x! = (y=1)' untuk memberikan nilai 'y' dengan 1, dan kemudian menguji apakah 'x' sama dengan 1. Tapi gaya pemrograman ini cenderung membuat program sulit dibaca, seperti *assignment* dalam *assignment* yang harus dihindari, kecuali mungkin dalam *one-shot* program.

```
$ awk 'BEGIN { print x!=(y=1) }'
```

Selain '=', ada beberapa operator *assignment* lain yang melakukan operasi aritmatika dengan nilai dari variabel sebelumnya. Sebagai contoh, operator '+=' menghitung nilai baru dengan menambahkan nilai sebelah kanan untuk nilai dari variabel sebelumnya. Dengan demikian, *assignment* berikut ini menambahkan lima dengan nilai foo:

```
$ awk 'BEGIN { foo = 1
               foo += 5
               print foo }'
```

Ada sebuah situasi dimana dapat menggunakan operator '+=' tidak sama seperti perulangan *lefthand operand* pada *righthand operand*. Seperti contoh dibawah ini:

```
$ awk 'BEGIN{
      foo[rand()] += 5
      for (x in foo)
            print x, foo[x]

      bar[rand()] = bar[rand()] + 5
      for (x in bar)
            print x, bar[x]
}'
```

Indeks dari ‘bar’ hampir pasti berbeda, karena fungsi *rand()* mengembalikan nilai yang berbeda setiap kali dipanggil. Array dan fungsi *rand()* belum tercakup. Contoh berikut ini menggambarkan fakta penting tentang operator *assignment*: ekspresi *left-hand* hanya dievaluasi sekali. Bergantung pada implementasinya seperti pada ekspresi yang dievaluasi pertama, *left-hand* atau *right-hand*. Coba amati contoh berikut ini, nilai dari a[3] bisa bernilai 2 atau 4:

```
$ awk ‘BEGIN { i = 1
               a[i += 2] = i + 1
               print a
               print i
}’
```

Berikut ini adalah Tabel operator assignment aritmatika. Dalam setiap kasus, sebelah kanan (*righthand*) operand adalah ekspresi yang nilainya dikonversi ke angka.

Operator	Effect
`lvalue += increment`	Adds increment to the value of lvalue.
`lvalue -= decrement`	Subtracts decrement from the value of lvalue.
`lvalue *= coefficient`	Multiplies the value of lvalue by coefficient.
`lvalue /= divisor`	Divides the value of lvalue by divisor.
`lvalue %= modulus`	Sets lvalue to its remainder by modulus.
`lvalue ^= power` `lvalue **= power`	Raises lvalue to the power power. (c.e.)

Cobalah beberapa contoh program dibawah ini:

```
$ awk ‘BEGIN { x=5; x+=5; print x }’
$ awk ‘BEGIN { x=5; x-=5; print x }’
$ awk ‘BEGIN { x=5; x*=5; print x }’
$ awk ‘BEGIN { x=5; x/=5; print x }’
$ awk ‘BEGIN { x=5; x%=5; print x }’
$ awk ‘BEGIN { x=5; x^=5; print x }’
$ awk ‘BEGIN { x=5; x**=5; print x }’
```

d. Operator Increment dan Decrement

Operator *increment* dan *decrement* berfungsi untuk menambah atau mengurangi nilai variabel setiap satu satuan. Sebuah operator assignment dapat melakukan hal yang sama, sehingga operator increment tidak menambahkan kekuatan kepada bahasa pemrogramanawk, namun operator tersebut merupakan operator yang sangat singkat dan nyaman untuk operasi yang sangat umum. Operator decrement ‘--’ bekerja seperti operator increment ‘++’, kecuali
Berikut ini tabel dari ekspresi operator *increment* dan *decrement.*

Operator	Keterangan
`++lvalue`	Increment lvalue, returning the new value as the value of the expression.
`lvalue++`	Increment lvalue, returning the old value of lvalue as the value of the expression.
`--lvalue`	Decrement lvalue, returning the new value as the value of the expression. (This expression is like '++lvalue', but instead of adding, it subtracts.)
`lvalue--`	Decrement lvalue, returning the old value of lvalue as the value of the expression. (This expression is like 'lvalue++', but instead of adding, it subtracts.)

Cobalah beberapa contoh program dibawah ini:

```
$ awk 'BEGIN { x=5; ++x; print x }'
$ awk 'BEGIN { x=5; x++; print x }'
$ awk 'BEGIN { x=5; --x; print x }'
$ awk 'BEGIN { x=5; x--; print x }'
```

Percobaan 3: *Truth Values and Conditions*

Dalam beberapa konteks tertentu, ekspresi juga dapat berfungsi sebagai "nilai kebenaran", dimana ekspresi tersebut yang menentukan apa yang akan terjadi selanjutnya setelah program dijalankan. Bagian ini menjelaskan bagaimana awk mendefinisikan nilai kebenaran yang terdiri dari "benar" dan "salah" dan bagaimana nilai-nilai tersebut dibandingkan.

a. True – False dalam awk

Banyak bahasa pemrograman yang memiliki konsep benar atau salah. Beberapa diantaranya menggunakan konstanta *true* atau *false*. Berbeda dengan awk yang mengambil konsep yang sederhana dari C, dimana angka yang bernilai tidak nol atau string yang bernilai tidak kosong, keduanya bernilai *true.* Program berikut ini mencetak string jika isi parameter dari blok if bernilai *true.*

```
$ awk 'BEGIN {
            if (3.1415927)
                print "A strange truth value"
            if ("Four Score And Seven Years Ago")
                print "A strange truth value"
            if (j = 57)
                print "A strange truth value"
}'
```

Dari kondisi "non-zero" atau "non-null" di atas, ada konsekuensi yang harus diambil, yaitu konstanta string yang bernilai "0" bernilai *true* karena tidak null. Namun, tidak seperti bahasa pemrograman lainnya, variabel awk tidak memiliki bentuk yang tetap. Bisa jadi variabel tersebut berisi angka atau string, tergantung

nilai apa yang di-*assign* ke variabel tersebut. Sekarang kita melihat bagaimana variabel tersebut ditulis dan bagaimana awk membandingkannya.

b. Menulis Variabel dan *Comparison Expression*

Tidak seperti bahasa pemrograman lainnya, variabel awk tidak memiliki bentuk yang tetap. Ia dapat berbentuk angka maupun string, tergantung dari nilai yang dimasukkan didalamnya. Dalam bagian ini kita mengetahui bagaimana variabel ditulis dan bagaimana awk membandingkan variabel-variabel tersebut.

1) String Type vs Numeric Type

Standar POSIX 1992 mengenalkan konsep dari "numeric string", dimana konsep tersebut tidak lain adalah string yang seakan-akan seperti sebuah karakter angka. Contohnya "+2".Konsep ini digunakan untuk menentukan tipe dari variabel. Tipe variabel sangat penting, karena tipe dari 2 variabel yang menentukan bagaimana variabel tersebut dibandingkan.

Standard POSIX 2008 menyempurnakan konsep "numeric string" yang sebelumnya dengan aturan sebagai berikut:

- Konstanta numerik / hasil dari operasi numerik memiliki atribut numerik.
- Konstanta string / hasil dari operasi string memiliki atribut string.
- Field, getline input, FILENAME, ARGV elements, ENVIRON elements dan elemen-elemen sebuah array yang dibuat oleh *patsplit()*, *split()* dan *match()* adalah *numeric string* yang memiliki atribut *strnum.* Tetapi, mereka memiliki atribut string. Variabel yang belum diinisialisasi juga memiliki atribut *strnum.*
- Atribut perlahan-lahan berjalan melalui assignment tetapi tidak merubah penggunaannya.

Aturan yang terakhir yang secara khusus dianggap penting. Program berikut ini menunjukkan variabel 'a' merupakan tipe numerik namun dapat digunakan dalam operasi string.

```
$ awk 'BEGIN {
      a = 12.345
      b = a " is a cute number"
      print b
}'
```

Ketika 2 operand dibandingkan, masing-masing *string comparison* atau *numeric comparison* mungkin digunakan. Hal ini tergantung pada atribut dari operand tersebut berdasarkan matriks berikut ini:

	STRING	NUMERIC	STRNUM
STRING	string	string	string
NUMERIC	string	numeric	numeric
STRNUM	string	numeric	numeric

Ide dasarnya adalah ketika user memberikan input numerik, awk akan memperlakukannya sebagai numerik, meskipun sebenarnya terdiri dari *character* dan atau string. Contoh konstanta string “+3.14” misalnya, ketika muncul pada source code program itu adalah string (meskipun terlihat numerik), dan tidak akan diperlakukan sebagai angka untuk tujuan perbandingan. Singkatnya, ketika satu operand adalah benar-benar string “aseli” seperti konstanta string, lalu dilakukan *string comparison*. Begitu juga dengan *numeric comparison.*

Pada bagian ini ditambahkan penekanan, bahwa: Semua input dari user adalah karakter dan atau string. Input string yang terlihat seperti numerik adalah secara otomatis ditambahkan atribut strnum. Dengan demikian, enam karakter dari -input string ' +3.14' menerima atribut strnum. Sebaliknya, delapan-karakter literal " +3.14" yang muncul dalam teks program adalah konstanta string.

Contoh program berikut ini mencetak '1' ketika perbandingan antara dua konstanta yang berbeda bernilai benar, dan '0' jika bernilai salah :

```
$ echo ’ +3.14’ | gawk ’{ print $0 == " +3.14" }’ True
= 1
$ echo ’ +3.14’ | gawk ’{ print $0 == "+3.14" }’ False
= 0
$ echo ’ +3.14’ | gawk ’{ print $0 == "3.14" }’ False
= 0
$ echo ’ +3.14’ | gawk ’{ print $0 == 3.14 }’ #True
= 1
$ echo ’ +3.14’ | gawk ’{ print $1 == " +3.14" }’ #False
= 0
$ echo ’ +3.14’ | gawk ’{ print $1 == "+3.14" }’ #True
= 1
$ echo ’ +3.14’ | gawk ’{ print $1 == "3.14" }’ #False
= 0
$ echo ’ +3.14’ | gawk ’{ print $1 == 3.14 }’ #True
= 1
```

2) Comparison Operator

Comparison expressions (ekspresi perbandingan) membandingkan string atau angka dengan tujuan untuk mencari hubungan seperti kesetaraan. Ditulis menggunakan operator relasional, yang merupakan superset dari mereka di dalam bahasa C. Tabel berikut menunjukkan operator relasional.

Expression	Result
`x < y`	True if x is less than y.
`x <= y`	True if x is less than or equal to y.
`x > y`	True if x is greater than y.

`x >= y`	True if x is greater than or equal to y.
`x == y`	True if x is equal to y.
`x != y`	True if x is not equal to y.
`x ~ y`	True if the string x matches the regexp denoted by y.
`x !~ y`	True if the string x does not match the regexp denoted by y.
`subscript in array`	True if the array array has an element with the subscript subscript.

Ekspresi perbandingan bernilai 1 jika benar dan bernilai 0 jika salah.Waktu dilakukan perbandingan operands dari tipe-tipe yang berbeda, operand numerik akan dikonversi menjadi string menggunakan nilai dari CONVFMT.

String akan dibandingkan dengan karakter pertama masing-masing, kemudian karakter kedua masing-masing, dan seterusnya. Misalnya “10” apakah lebih kecil dari “9”. Jika ada 2 string dimana yang satu adalah awalan (*prefix*) dari yang lain, string terpendek lebih pendek dari yang lainnya. Oleh karenanya “abc” lebih kecil dari “abcd”. Kadang-kadang kita terlupa menuliskan operator “==” dengan “=” saja. Awk masih membetulkan, tetapi jalannya program belum tentu. Berikut ini contoh program yang mengilustrasikan beberapa jenis perbandingan gawk, begitu juga hasil dari perbandingannya:

```
$ gawk 'BEGIN { print 1.5 <= 2.0 }'
$ gawk 'BEGIN { print "abc" >= "xyz"}'
$ gawk 'BEGIN { print 1.5 != " +2"}'
$ gawk 'BEGIN { print "1e2" < "3"}'
$ gawk 'BEGIN { print 1.5 <= 2.0 }'
$ gawk 'BEGIN { a = 2; b = "2"; print (a == b)}'
$ gawk 'BEGIN { a = 2; b = " +2"; print (a == b)}'
```

Contoh yang lainnya bisa dalam bentuk seperti dibawah ini:

```
$ echo 1e2 3 | awk '{ print ($1 < $2) ? "true" : "false" }'
= false
```

Hasilnya ‘false’ karena keduanya $1 dan $2 adalah input dari user. Keduanya *numeric string*, oleh karenanya keduanya memiliki atribut *strnum* yang mendikte perbandingan numerik. Tujuan dari aturan perbandingan dan penggunaan string numerik adalah untuk mencoba untuk menghasilkan perilaku yang paling sedikit kejutannya, sementara masih "melakukan hal yang benar."

String comparison dan *regex comparison* sangat berbeda. Contohnya: x == “foo”, memiliki nilai 1 jika variabel x berisi tepat ‘foo’. Hal ini akan berbeda dengan x ~ /foo/, memiliki nilai 1 jika variabelnya berisi string ‘foo’ seperti ‘What a fool am I!’.

Righthand operand (operand sebelah kanan) seperti operator ‘~’ dan ‘!~’ dapat berupa sebuah konstanta regex (/.../) atau ekspresi biasa. Dalam kasus terakhir, nilai dari ekspresi sebagai string digunakan sebagai regex dinamis. Pada

implementasi awk yang modern, konstanta regular ekspression dilakukan dengan memberikan slash pada tiap ekspresinya. Salah satu tempat khusus di mana ‘/foo/’ bukan merupakan singkatan untuk '$ 0 ~ / foo /' adalah ketika menjadi operand sebelah kanan dari '~' atau '! ~'.

3) String Comparison dengan Aturan POSIX

Standar POSIX mengatakan bahwa perbandingan string dilakukan berdasarkan urutan *local collating*. Hal ini biasanya sangat berbeda dari hasil yang diperoleh ketika melakukan perbandingan satu karakter dengan karakter lainnya. Karena perilaku ini berbeda jauh dari praktek yang ada, gawk hanya mengimplementasikannya pada mode POSIX. Berikut ini adalah contoh untuk menggambarkan perbedaan, dalam bentuk lokal *'en_US.UTF-8'*:

```
$ gawk ’BEGIN { printf("ABC < abc = %s\n",
  ("ABC" < "abc" ? "TRUE" : "FALSE")) }’
= ABC < abc = TRUE

$ gawk --posix ’BEGIN { printf("ABC < abc = %s\n",
  ("ABC" < "abc" ? "TRUE" : "FALSE")) }’
= ABC < abc = FALSE
```

c. Ekspresi Boolean

Ekspresi Boolean adalah kombinasi dari ekspresi perbandingan atau ekspresi pencarian yang cocok, menggunakan operator Boolean "atau" ('||'), "dan" ('&&'), dan "tidak" ('!'), bersama dengan tanda kurung untuk mengatur *nesting*. Nilai kebenaran dari ekspresi Boolean dihitung dengan menggabungkan nilai-nilai kebenaran dari ekspresi komponen. Ekspresi Boolean juga disebut sebagai ekspresi logis.

Ekspresi boolean dapat digunakan untuk membandingkan dan mencocokkan ekspresi. Dapat menggunakan perintah if, while, do dan for. Semuanya memiliki nilai numerik (satu jika benar, nol jika salah) dan dapat dijalankan jika hasil dari ekpresi boolean disimpan dalam sebuah variabel atau digunakan dalam operasi aritmetika. Setiap ekspresi boolean adalah pola yang valid (benar), jadi Anda dapat menggunakannya untuk mengkontrol aturan eksekusinya. Operator-operator boolean dapat dilihat pada tabel berikut ini:

Boolean expression	Keterangan
`boolean1 && boolean2`	True if both boolean1 and boolean2 are true.
`boolean1 || boolean2`	True if at least one of boolean1 or boolean2 is true.
`! boolean`	True if boolean is false

Program berikut ini memberikan contoh penggunaan boolean expression.

```
$ awk ‘{ if ($0 ~ /2400/ && $0 ~ /foo/)
                  print }’ BBS-list
```

Pernyataan diatas mencetak input record jika terdiri dari ‘2400’ dan ‘foo’. Sub-ekspresi boolean-2 dievaluasi hanya jika boolean-1 benar. Hal ini dapat membuat berbeda ketika boolean-2 terdiri dari ekspresi yang memiliki efek samping.

```
$ awk ‘BEGIN{bar = 0} { if ($0 ~ /foo/ && ($2 == bar++)) print;
                  print bar}’ BBS-list
```

Pada contoh diatas variabel ‘bar’ tidak di-increment jika tidak ada substring ‘foo’ pada record.

```
$ awk ‘{ if ($0 ~ /2400/ || $0 ~ /foo/)
                  print }’ BBS-list
```

Contoh diatas menampilkansemua record pada input yang mengandung ‘2400’ atau ‘foo’ atau keduanya. Sub-ekspresi boolean-2 dievaluasi hanya jika boolean-1 salah. Hal ini dapat membuat berbeda ketika boolean-2 terdiri dari ekspresi yang memiliki efek samping.

Operator ‘&&’ dan ‘||’ disebut dengan *short-circuit* karena cara kerja operator tersebut yang cepat. Evaluasi dari ekspresi tersebut, jika hasilnya dapat ditentukan maka akan melalui bagian evaluasi. Pernyataan yang menggunakan operator ‘&&’ dan ‘||’ dapat dilanjutkan dengan meletakkan *newline* setelahnya. Tetapi Anda tidakdapat meletakkan *newline* di depan operator ini tanpa menggunakan *backslash continuation*.

Program berikut mencetak ‘no home!’ pada event yang tidak biasa ketika variabel HOME environtment tidak didefinisikan.

```
$ awk ‘BEGIN { if (! ("HOME" in ENVIRON))
      print "no home!" }’
```

Nilai sebenarnya dari ekspresi yang menggunakan operator ‘!’ adalah salah satu diantara satu atau nol, tergantung pada nilai kebenaran dari ekspresi yang dipakai. Operator ‘!’ biasanya digunakan untuk merubah variabel *flag* dari salah ke benar dan kembali lagi. Contohnya, program berikut ini adalah salah satu cara mencetak baris diantara *special bracketing lines*:

```
$ awk ‘BEGIN {
      $1 == "START" { interested = ! interested; next }
      interested == 1 { print }
      $1 == "END" { interested = ! interested; next }
}’
```

Variabel *interested* diatas, termasuk juga semua variabel pada awk, dimulai dari inisialisasi dengan nol yang juga definisinya adalah “*false*”. Ketika sebuah baris dilihat dari field pertama adalah ‘START’, nilai dari variabel ‘*interested*’ dirubah (toggle) menjadi “*true*” menggunakan ‘!’. Aturan berikutnya mencetak baris yang menunjukkan bahwa nilai dari ‘*interested*’ adalah benar.

d. Ekspresi Kondisional

Ekspresi kondisional adalah jenis ekspresi yang spesial, karena ekspresi ini memiliki 3 operand. Ekspresi ini mengijinkan Anda menggunakan satu nilai ekspresi untuk memilih satu dari dua ekspresi. Ekspresi kondisional hampir sama seperti pada Bahasa C :

```
selector ? if-true-exp : if-false-exp
```

Ada 3 sub-ekspresi, yang pertama selector (selalu dihitung lebih dulu). Jika benar (*not zero or not null*), lalu *if-true-exp* dihitung berikutnya dan nilainya menjadi nilai dari akhir ekpresi. Jika tidak, *if-false-exp* dihitung kemudian dan nilainya menjadi nilai dari akhir ekpresi. Contohnya ekspresi berikut ini menghasilkan nilai absolute dari x:

```
x >= 0 ? x : -x
```

Setiap waktu, ekspresi kondisional dihitung, hanya sekali dari *if-true-exp* dan *if-false-exp* digunakan, lainnya diabaikan. Hal ini penting ketika ekspresi memiliki efek samping. Contohnya ekspresi kondisional berikut ini memerikas elemen *'i'* begitu juga 'a' dan 'b' dan increment dari 'i':

```
x == y ? a[i++] : b[i++]
```

Hal ini memberikan kepastian untuk melakukan increment 'i' tepat sekali, karena setiap waktu hanya satu dari dua ekspresi increment yang dieksekusi, dan yang lainnya tidak. Lebih lanjut akan dijelaskan pada bab array. Sebagai ekspresi yang jarang digunakan pada awk, pernyataan yang menggunakan ':?' dapat dilanjutkan hanya dengan menempatkan baris baru setelah karakter berikutnya. Namun, menempatkan baris baru di depan karakter setelahnya hal ini tidak bekerja tanpa menggunakan *backslash continuation*.

Percobaan 4: *Function Calls*

Fungsi (*function*) adalah sebuah nama untuk perhitungan tertentu. Dengan fungsi memungkinkan Anda untuk memanggil perhitungan tersebut pada setiap baris dalam program. Misalnya, fungsi *sqrt()* menghitung akar kuadrat dari angka.

Ada beberapa fungsi yang tetap atau *built-in,* yang tersedia dalam setiap program awk. Contohnya fungsi *sqrt()* adalah salah satunya. Selain itu, Anda dapat menentukan fungsi untuk digunakan dalam program Anda. Selanjutnya akan dijelaskan pada bab khusus tentang fungsi.

Cara untuk menggunakan fungsi adalah dengan ekspresi pemanggilan fungsi, yang terdiri dari nama fungsi yang diikuti oleh daftar argumen dalam tanda kurung (mirip seperti bahasa C). Argumen adalah ekspresi yang menyediakan "bahan baku" untuk perhitungan fungsi itu. Bila ada lebih dari satu argumen, mereka dipisahkan dengan koma (,). Jika tidak ada argumen, hanya menulis kurung tanpa isi, seperti ini '()' setelah nama fungsi. Contoh-contoh berikut menunjukkan pemanggilan fungsi dengan dan tanpa argumen:

```
sqrt(x^2 + y^2)           # one argument
atan2(y, x)          # two arguments
rand()               # no arguments
```

Setiap fungsi memiliki sejumlah argumen tertentu. Misalnya, fungsi *sqrt()* harus dipanggil dengan argumen yang jumlahnya satu yaitu angka untuk dioperasikan akar-kuadrat kepadanya.

```
sqrt(argumen)
```

Beberapa fungsi *built-in* memiliki satu atau lebih pilihan argumen. Jika argumen-argumen tersebut tidak diberikan, fungsi menggunakan nilai standar yang default. Jika argumen-argumen dimasukkan untuk dipanggil oleh fungsi yang sudah didefinisikan, kemudian argumen tersebut diperlakukan sebagai varibel lokal dan diinisialisasi dengan string kosong (*empty string*). Sebagai fitur lanjut, gawk menyediakan *indirect function calls*, yaitu sebuah cara memilih fungsi untuk memanggil pada saat runtime, meskipun pada saat Anda menulis *source code* program. Lebih lanjut, fitur ini akan dijelaskan pada bab khusus.

Seperti ekspresi lainnya *function call* juga memiliki sebuah nilai, dimana dihitung oleh fungsi berdasarkan argumen yang Anda berikan. Contohnya, nilai dari 'sqrt(argument)' adalah akar kuadrat dari argumen. Berikut ini contoh program yang membaca angka, satu angka tiap barisnya, dan mencetak nilai akar kuadrat dari masing-masing:

```
$ awk '{ print "The square root of", $1, "is", sqrt($1) }'
1
a The square root of 1 is 1
3
a The square root of 3 is 1.73205
5
a The square root of 5 is 2.23607
Ctrl-d
```

Sebuah fungsi juga memiliki efek samping, seperti pemberian nilai kepada variabel tertentu atau melakukan aktivitas input-output (I/O). Program berikut ini menunjukkan bagaimana fungsi *match()* merubah variabel RSART dan RLENGTH:

```
$ awk `{ if (match($1, $2))
```

```
            print RSTART, RLENGTH
      else
            print "no match"
}'
> aaccff    c+
> foo       bar
> abcdefg   e
```

Varibel RSTART berisi nomor urutan karakter yang sama, sedangkan variabel RLENGTH berisi berapa banyak karakter yang sama.

Percobaan 5: *Operator Precedence*

Operator precedence (kedudukan operator) menunjukkan bagaimana operator dikelompokkan dan diurutkan ketika operator yang berbeda muncul dalam satu ekspresi. Contohnya operator '*' memiliki *precedence* lebih tinggi dari pada '+'. Jadi 'a+b*c' artinya adalah kalikan dulu 'b' dengan 'c' kemudian hasilnya jumlahkan dengan 'a' seperti pada operasi berikut 'a+(b*c)'.

Precedence yang normal dari operator dapat diatur dengan menggukan tanda kurung (*parentheses*). Aturan *precedence* mengatakan bahwa dimana letak tanda kurung diasumsikan. Bahkan, adalah kebiasaan yang baik untuk selalu menggunakan tanda kurung jika ada kombinasi yang tidak biasa dari operator, karena orang lain yang membaca program mungkin tidak mengetahui operator mana yang didahulukan. Bahkan programmer berpengalaman sekalipun melupakan aturan ini, sehingga dapat mengarah kepada terjadinya kesalahan. Tanda kurung secara explicit akan membantu mencegah kesalahan tersebut.

Ketika operator yang memiliki precedence yang sama digunakan bersamaan ada beberapa aturan pembacaan dari yang paling kiri maupun yang paling kanan. Hampir semua operator mulai dari paling kiri yang dieksekusi pertama kali, kecuali operator *assignment, conditional,* dan *exponential*. Jadi, 'a-b+c' dapat dikelompokkan menjadi '(a-b)+c' dan 'a=b=c' dapat dikelompokkan menjadi 'a=(b=c)'. Tabel berikut menunjukkan operator pada awk, dengan urutan dari yang paling tinggi ke yang paling rendah.

Operator	Keterangan
`(...)`	*Grouping.*
`$`	*Field reference.*
`++ --`	*Increment, decrement.*
`^ **`	*Exponentiation. These operators group right-to-left.*
`+ - !`	*Unary plus, minus, logical "not."*
`* / %`	*Multiplication, division, remainder.*
`+ -`	*Addition, subtraction.*
`String Concatenation`	*There is no special symbol for concatenation. The operands are simply written side by side.*
`< <= == != > >= >> | |&`	*Relational and redirection. The relational operators and the*

	redirections have the same precedence level. Characters such as '>' serve both as relationals and as redirections; the context distinguishes between the two meanings.
~ !~	*Matching, nonmatching.*
in	*Array membership.*
&&	*Logical "and".*
||	*Logical "or".*
?:	*Conditional. This operator groups right-to-left.*
= += -= *= /= %= ^= **=	*Assignment. These operators group right-to-left.*

Catatan:
Operator '|&', '**', and '**=' adalah operator yang tidak dikenali oleh POSIX.

Percobaan 6: Dimana Anda Membuat Perbedaan.

Sistem yang modern mensupport notasi *locales*, yaitu sebuah cara untuk memberitahu sistem tentang *local character set* dan *language* (bahasa). Sekali waktu, pengaturan lokal digunakan untuk mempengaruhi pencocokan regex, tetapi hal ini tidak begitu lama.

Lokal dapat mempengaruhi pembagian *record*. Untuk kasus normal 'RS = "\ n"', lokal sebagian besar tidak relevan. Sebagai *record separator* yang berupa karakter tunggal lainnya, pengaturan 'LC_ALL=C' pada environtment pemrograman awk akan memberikan kinerja yang jauh lebih baik ketika membaca record. Jika tidak, awk harus membuat beberapa fungsi yang dapat dipanggil tiap karakter input, untuk menemukan *record terminator* (akhir dari *record*).

Menurut POSIX, perbandingan string juga dipengaruhi oleh lokal (mirip dengan *regular expression*). Terakhir, lokal akan mempengaruhi nilai karakter titik desimal yang digunakan ketika awk mem-parsing input data.