AWK Tutorial Guide

中央研究院计算中心
ASPAC 计划
aspac@phi.sinica.edu.tw

技术报告:94011
83 年 12 月 5 日
Version:2.2

Contents

ASPAC 计划版权声明

ASPAC（Academia Sinica PACkage）是中央研究院计算中心关于“软体工具使用”（Software Tools）及“问题解决”（Problem Solving）的计划。在这计划下所发展之软件及文件都属于中央研究院计算中心所有。所有正式公开之电子形式数据（包括软件及文件），在满足下列软件及文件使用权利说明下，都可免费取得及自由使用。

软件及文件使用权利说明如下：

软件的使用权利：将沿用美国 FSF（Free Software Foundation）1991 年 6 月第二版的 GNU General Public License。

文件的使用权利：文件可以自由拷贝及引用，但不得藉以图利。除非必要手续费的收取。

---

aspac@phi.sinica.edu.tw

前言 Preface

- 有关本手册 :

  这是一本 AWK 学习指引, 其重点着重于 :

  AWK 适于解决哪些问题 ?
  AWK 常见的解题模式为何 ?

为使读者快速掌握 AWK 解题的模式及特性, 本手册系由一些较具代表性的范例及其题解所构成; 各范例由浅入深, 彼此间相互连贯,范例中并对所使用的 AWK 语法及指令辅以必要的说明. 有关 AWK 的指令, 函数,...等条列式的说明则收录于附录中, 以利读者往后撰写程序时查阅. 如此编排, 可让读者在短时间内顺畅地学会使用 AWK 来解决问题. 建议读者循着范例上机实习, 以加深学习效果.

- 读者宜先具备下列背景 :

  [a.] UNIX 环境下的简单操作及基本概念.

  例如 : 档案编辑, 档案复制 及 pipe, I/O Redirection 等概念

  [b.] C 语言的基本语法及流程控制指令.

  (AWK 指令并不多, 且其中之大部分与 C 语言中之用法一致, 本手册中对该类指令之语法及特性不再加以繁冗的说明, 读者若欲深究,可自行翻阅相关的 C 语言书籍)

  例如 : printf(), while()...

- 参考书目 :

  本手册是以学习指引为主要编排方式, 读者若需要有关 AWK 介绍详尽的参考书,可参考下列两本书 :

  1. Alfred V. Aho, Brian W. Kernighan and Peter J. Weinberger, The AWK Programming Language'', Addison-Wesley Publishing Company
  2. Dale Dougherty, ″ sed & awk '', O`Reilly & Associates, Inc

---

Overview of AWK

Why AWK

AWK 是一种程序语言. 它具有一般程序语言常见的功能.因 AWK 语言具有某些特点, 如 : 使用直译器(Interpreter)不需先行编译; 变量无型别之分(Typeless), 可使用文字当数组的注标(Associative Array)...等特色. 因此, 使用 AWK 撰写程序比起使用其它语言更简洁便利且节省时间. AWK 还具有一些内建功能, 使得 AWK 擅于处理具数据列(Record), 字段(Field)型态的资料; 此外, AWK 内建有 pipe 的功能, 可将处理中的数据传送给外部的 Shell 命令加以处理, 再将 Shell 命令处理后的数据传回 AWK 程序, 这个特点也使得 AWK 程序很容易使用系统资源.

由于 AWK 具有上述特色, 在问题处理的过程, 可轻易使用 AWK 来撰写一些小工具; 这些小工具并非用来解决整个大问题, 它们只个别扮演解决问题过程的某些角色, 可藉由 Shell 所提供的 pipe 将数据按需要传送给不同的小工具进行处理, 以解决整个大问题. 这种解题方式, 使得这些小工具可因不同需求而被重复组合及使用(reuse); 也可藉此方式来先行测试大程序原型的可行性与正确性, 将来若需要较高的执行速度时再用 C 语言来改写. 这是 AWK 最常被应用之处. 若能常常如此处理问题, 读者可以以更高的角度来思考抽象的问题, 而不会被拘泥于细节的部份. 本手册为 AWK 入门的学习指引, 其内容将先强调如何撰写 AWK 程序,未列入进一步解题方式的应用实例, 这部分将留待 UNIX 进阶手册中再行讨论.

如何取得 AWK

一般的 UNIX 操作系统, 本身即附有 AWK. 不同的 UNIX 操作系统所附的 AWK 其版本亦不尽相同. 若读者所使用的系统上未附有 AWK, 可透过 anonymous ftp 到下列地方取得 :

```
phi.sinica.edu.tw:/pub/gnu
ftp.edu.tw:/UNIX/gnu
prep.ai.mit.edu:/pub/gnu
```

How AWK works

为便于解释 AWK 程序架构, 及有关术语(terminology), 先以一个员工薪资档(emp.dat), 来加以介绍.

```
A125  & Jenny  &100  &210
A341  & Dan    &110  &215
P158  & Max    &130  &209
P148  & John   &125  &220
A123  & Linda  & 95  &210
```

档案中各字段依次为 员工 ID, 姓名, 薪资率,及 实际工时. ID 中的第一码为部门识别码. “A”,’’P’’分别表示”组装”及”包装”部门.本小节着重于说明 AWK 程序的主要架构及工作原理, 并对

一些重要的名词辅以必要的解 释. 由这部分内容, 读者可体会出AWK语言的主要精神及AWK与其它语程序言的差异处. 为便于说明, 以条列方式说明于后.

- 名词定义

1. 资料列: AWK从数据文件上读取数据的基本单位.以上列档案emp.dat为例, AWK读入的
   第一笔资料列是 "A125 Jenny 100 210"
   第二笔资料列是 "A341 Dan 110 215"
   一般而言, 一笔数据列相当于数据文件上的一行资料.
   (参考 : 附录 B 内建变数”RS”)

2. 字段(Field) : 为资料列上被分隔开的子字符串.
   以资料列”A125 Jenny 100 210”为例,

第一栏	第二栏	第三栏	第四栏
“A125”	“Jenny”	100	210

   一般是以空格符来分隔相邻的字段. (参考 : 附录 D 内建
   变数”FS”)

- 如何执行AWK
  于UNIX的命令列上键入诸如下列格式的指令: (“$”表Shell命令列上的提示符号)
  $awk ’AWK程序’ 数据文件文件名
  则AWK会先编译该程序, 然后执行该程序来处理所指定的数据文件.(上列方式系直接把程序写在UNIX的命令列上)

- AWK程序的主要结构 :
  AWK程序中主要语法是 Pattern { Actions}, 故常见之AWK程序其型态如下 :

```
Pattern1  { Actions1 }
Pattern2  { Actions2 }
......
Pattern3  { Actions3 }
```

- Pattern 是什么 ?
  AWK 可接受许多不同型态的 Pattern. 一般常使用 “关系判断式’(Relational expres sion) 来当成 Pattern.例如 : x > 34 是一个Pattern, 判断变量 x 与 34 是否存在大于的关系. x == y 是一个 Pattern, 判断变量 x 与变量 y 是否存在等于的关系. 上式中 x >34 , x == y 便是典型的Pattern.
  AWK 提供 C 语言中常见的关系操作数(Relational Operators) 如 >, <, >=, <=, ==, !=.
  此外, AWK 还提供 ~ (match) 及 !~(not match) 二个关系操作数 (注一). 其用法与涵义如下:
  若 A 表一字符串, B 表一 Regular Expression

A ~ B 判断 字符串 A 中是否 包含 能合于(match)B 式样的子字符串.
A !~ B 判断 字符串 A 中是否 未包含 能合于(match)B 式样的子字符串.
例如 : “banana” ~ /an/ 整个是一个 Pattern. 因为”banana”中含有可 match /an/的子字符串, 故此关系式成立(true),整个 Pattern 的值也是 true.
相关细节请参考 附录 A Patterns, 附录 E Regular Expression [注 一 :] 有少数 AWK 论著, 把 ~, !~ 当成另一类的 Operator,并不视为一种 Relational Operator. 本手册中将这两个操作数当成一种 Relational Operator.

- Actions 是什么?
  Actions 是由许多 AWK 指令构成. 而 AWK 的指令与 C 语言中的指令十分类似.例如 :
  AWK 的 I/O 指令 : print, printf(), getline..
  AWK 的 流程控制指令 : if(...){..} else{..}, while(...){...}...
  (请参考 附录 B --- “Actions”)

- AWK 如何处理 Pattern { Actions } ?
  AWK 会先判断(Evaluate) 该 Pattern 之值, 若 Pattern 判断(Evaluate)后之值为 true(或不为 0 的数字,或不是空的字符串), 则 AWK 将执行该 Pattern 所对应的 Actions. 反之, 若 Pattern 之值不为 true, 则 AWK 将不执行该 Pattern 所对应的 Actions. 例如 : 若 AWK 程序中有下列两指令
  50 > 23 : {print "Hello! The word!!" }
  "banana" ~ /123/ { print "Good morning !" }
  AWK 会先判断 50 >23 是否成立. 因为该式成立, 所以 AWK 将印出"Hello! The word!!". 而另一 Pattern 为 "banana" ~/123/, 因为"banana" 内未含有任何子字符串可 match /123/, 该 Pattern 之值为 false, 故 AWK 将不会印出 "Good morning !"

- AWK 如何处理{ Actions } 的语法?(缺少 Pattern 部分) 有时语法 Pattern { Actions }中, Pattern 部分被省略, 只剩 {Actions}. 这种情形表示 “无条件执行这个 Actions”.

- AWK 的字段变量
  AWK 所内建的字段变量及其涵意如下 :

字段变量	涵意
$0	为一字符串, 其内容为目前 AWK 所读入的资料列.
$1	代表 $0 上第一个字段的数据.
$2	代表 $0 上第二栏个位的资料.
...	其余类推

  读入数据列时, AWK 如何修正(update)这些内建的字段变量. 当 AWK 从数据文件中读取一笔数据列时, AWK 会使用内建变量 $0 予以记录.每当 $0 被异动时 (例如 : 读入新的数据列 或 自行变更 $0,...), AWK 会立刻重新分析 $0 的字段情况, 并将 $0 上各字段的数据用 $1, $2, ..予以记录.

- AWK 的内建变数(Built-in Variables)
  AWK 提供了许多内建变量, 使用者于程序中可使用这些变量来取得相关信息.常见的内建变数有 :

内建变数	涵意
NF	(Number of Fields)为一整数, 其值表$0 上所存在的字段数目.
NR	(Number of Records)为一整数, 其值表 AWK 已读入的资料列数目.
FILENAMEAWK 正在处理的数据文件文件名.	

例如 : AWK 从资料文件 emp.dat 中读入第一笔资料列 "A125 Jenny 100 210" 之后, 程序中:

$0 之值将是 "A125 Jenny 100 210"

$1 之值为 "A125" $2 之值为 "Jenny"

$3 之值为 100 $4 之值为 210

NF 之值为 4 $NF 之值为 210

NR 之值为 1 FILENAME 之值为 “emp.dat”

- AWK 的工作流程 :

执行 AWK 时, 它会反复进行下列四步骤.

a) 自动从指定的数据文件中读取一笔数据列.

b) 自动更新(Update)相关的内建变量之值. 如 : NF, NR, $0...

c) 逐次执行程序中所有 的 Pattern { Actions } 指令.

d) 当执行完程序中所有 Pattern { Actions } 时, 若数据文件中还有未读取的数据, 则反复执行步骤 1 到步骤 4.

AWK 会自动重复进行上述 4 个步骤, 使用者不须于程序中撰写这个循环 (Loop).

打印档案中指定的字段数据并加以计算

AWK 处理数据时，它会自动从数据文件中一次读取一笔记录，并会将该数据切分成一个个的字段；程序中可使用 $1，$2,... 直接取得各个字段的内容. 这个特色让使用者易于用 AWK 撰写 reformatter 来改变数据格式.

[范例 :] 以档案 emp.dat 为例，计算每人应发工资并打印报表.

[分析 :] AWK 会自行一次读入一列数据，故程序中仅需告诉 AWK 如何处理所读入的数据列.

执行如下命令 : ($ 表 UNIX 命令列上的提示符号)

awk '{ print $2, $3 * $4 }' emp.dat

执行结果如下 :

屏幕出现 :

Jenny 21000

Dan 23650

Max 27170

John 27500

Linda 19950

说 明 :

1. UNIX 命令列上，执行 AWK 的语法为:
   awk 'AWK 程序' 欲处理的资料文件文件名.本范例中的程序部分为 {print $2, $3 * $4}.
   把程序置于命令列时，程序之前后须以 ' 括住.
2. emp.dat 为指定给该程序处理的数据文件文件名.
3. 本程序中使用 : Pattern { Actions } 语法.

Pattern	Actions
	print $2, $3 * $4

   Pattern 部分被省略，表无任何限制条件. 故 AWK 读入每笔资料列
   后都将无条件执行这个 Actions.
4. print 为 AWK 所提供的输出指令，会将数据输出到 stdout(屏幕). print 的参数间彼此以 "{ ,}" 隔开，印出数据时彼此间会以空白隔开.(参考 附录 D 内建变量 OFS)
5. 将上述的 程序部分 储存于档案 pay1.awk 中. 执行命令时再指定 AWK 程序文件 之文件名. 这是执行 AWK 的另一种方式，特别适用于程式较大的情况，其语法如下:
   $awk -f AWK 程序文件名 数据文件文件名
   故执行下列两命令,将产生同样的结果.
   $awk -f pay1.awk emp.dat
   $awk ' { print $2, $3 * $4 } ' emp.dat
   读者可使用"-f"参数,让 AWK 主程序使用其它仅含 AWK 函数的档案中的函数
   其语法如下:
   awk -f AWK 主程序文件名 -f AWK 函数文件名 数据文件文件名
   (有关 AWK 中函数之宣告与使用于 7.4 中说明)
6. AWK 中也提供与 C 语言中类似用法的 printf() 函数. 使用该函数可进一步控制数据的输出格式.
   编辑另一个 AWK 程序如下，并取名为 pay2.awk
   { printf("\%6s Work hours: %3d Pay: %5d\", $2,\$3, $3* $4) }
   执行下列命令

```
$awk -f  pay2.awk   emp.dat
```

执行结果屏幕出现:

```
Jenny   Work hours: 100 Pay: 21000
Dan   Work hours: 110 Pay: 23650
Max   Work hours: 130 Pay: 27170
John   Work hours: 125 Pay: 27500
Linda   Work hours:  95 Pay: 19950
```

选印合乎指定条件的记录

Pattern { Action }为 AWK 中最主要的语法. 若某 Pattern 之值为真则执行它后方的 Action. AWK 中常使用”关系判断式” (Relational Expression)来当成 Pattern.

AWK 中除了>, <, ==, != ,...等关系操作数(Relational Operators)外,另外提供 ~(match),!~(Not Match) 二个关系操作数. 利用这两个操作数, 可判断某字符串是否包含能符合所指定 Regular Expression 的子字符串. 由于这些特性, 很容易使用 AWK 来撰写需要字符串比对, 判断的程序.

[范例 :] 承上例, 组装部门员工调薪 5%,(组装部门员工之 ID.系以”A”开头) 所有员工最后之薪资率若仍低于 100, 则以 100 计. 撰写 AWK 程序行印新的员工薪资率报表.

[分析:]这个程序须先判断所读入的数据列是否合于指定条件, 再进行某些动作.AWK 中 Pattern{ Actions }的语法已涵盖这种 “ if (条件) { 动作} ”的架构. 编写如下之程序, 并取名 adjust1.awk

```
$1 ~ /^A.*/ { $3 *= 1.05 } $3<100 { $3 = 100 }
{ printf("%s %8s %d\n", $1, $2, $3)} 执行下列命令 :
$awk -f adjust1.awk emp.dat
结果如下 : 屏幕出现 :
A125 Jenny 105
A341 Dan 115
P158 Max 130
P148 John 125
A123 Linda 100
```

说 明 :

1. AWK 的工作程序是: 从数据文件中每次读入一笔数据列, 依序执行完程序中所有的 Pattern{ Action } 指令

Pattern	Actions
$1~/^A.*/	{ $3 *= 1.05 }
$3 < 100	{ $3 = 100 }
	{printf("%s%8s%d\n",$1,$2,$3)}

   再从数据文件中读进下一笔记录继续进行处理.
2. 第一个 Pattern { Action }是: $1 ~ /^A.*/ { $3 *= 1.05 } $1 ~ /^A.*/ 是一个 Pattern, 用来判断该笔资料列的第一栏是否包含%以”A”开头的子字符串. 其中 /^A.*/ 是一个 Regular Expression, 用以表示任何以”A”开头的字符串. (有关 Regular Expression 之用法 参考 附录 E).
   Actions 部分为 $3 *= 1.05 $3 *= 1.05 与 $3 = $3 * 1.05 意义相同. 运算子”*=” 之用法则与 C 语言中一样. 此后与 C 语言中用法相同的运算子或语法将不予赘述.
3. 第二个 Pattern { Actions } 是: $3 <100 {$3 = 100 } 若第三栏的数据内容(表薪资率)小于 100, 则调整为 100.
4. 第三个 Pattern { Actions } 是: {printf("%s %-8s %d\n",$1, $2, $3)} 省略了 Pattern(无条件执行 Actions), 故所有数据列调整后的数据都将被印出.

AWK 中数组的特色

AWK 程序中允许使用字符串当做数组的注标(index). 利用这个特色十分有助于资料统计工作.(使用字符串当注标的数组称为 Associative Array)

首先建立一个数据文件, 并取名为 reg.dat. 此为一学生注册的资料文件; 第一栏为学生姓名, 其后为该生所修课程.

```
Mary      O.S.          Arch.          Discrete
Steve     D.S.          Algorithm      Arch.
Wang      Discrete      Graphics       O.S.
Lisa      Graphics      A.I.
Lily      Discrete      Algorithm
```

AWK 中数组的特性

1. 使用字符串当数组的注标(index).
2. 使用数组前不须宣告数组名及其大小.

例如 : 希望用数组来记录 reg.dat 中各门课程的修课人数. 这情况,有二项信息必须储存 :

(a) 课程名称, 如 : “O.S.”, “Arch.”.. ,共有哪些课程事前并不明确.

(b)各课程的修课人数. 如 : 有几个人修”O.S.”

在 AWK 中只要用一个数组就可同时记录上列信息. 其方法如下 :

使用一个数组 Number[ ] :

- 以课程名称当 Number[ ] 的注标.
- 以 Number[ ] 中不同注标所对映的元素代表修课人数.

例如 :

有 2 个学生修 “O.S.”, 则以 Number[“O.S.”] = 2 表之. 若修“O.S.”的人数增加一人,则 Number[“O.S.”] = Number[“O.S.”] + 1 或 Number["O.S."]++ .

如何取出数组中储存的信息

以 C 语言为例, 宣告 int Arr[100]; 之后, 若想得知 Arr[ ]中所储存的数据, 只须用一个循环, 如 :

```
for(i=0; i<00; i++) printf("%d\n", Arr[i]);
```

即可. 上式中 :

数组 Arr[ ] 的注标 : 0, 1, 2,..., 99

数组 Arr[ ] 中各注标所对应的值 : Arr[0], Arr[1],...Arr[99]

但 AWK 中使用数组并不须事先宣告. 以刚才使用的 Number[ ] 而言, 程序执行前, 并不知将来有哪些课程名称可能被当成 Number[ ] 的注标.

AWK 提供了一个指令, 藉由该指令AWK会自动找寻数组中使用过的所有注标. 以 Number[ ] 为例, AWK 将会找到 “O.S.”, “Arch.”,... 使用该指令时, 须指定所要找寻的数组, 及一个变量. AWK 会使用该的变量来记录从数组中找到的每一个注标. 例如

```
for(course in Number){....}
```

指定用 course 来记录 AWK 从 Number[ ] 中所找到的注标. AWK 每找到一个注标时, 就用 course 记录该注标之值且执行{....}中之指令. 藉由这个方式便可取出数组中储存的信息.(详见下例)

范例 : 统计各科修课人数,并印出结果. 建立如下程序,并取名为 course.awk:

```
    {for(i=2; i< Number[$i]++ i++)>
        END{for(coursein Number)
        printf("\%-10s %d\n", course, Number[course])
    }
```

执行下列命令 :

```
awk  -f   course.awk    reg.dat
```

执行结果如下 :

```
Discrete    3

D.S.          1

O.S.          2

Graphics   2

A.I.           1

Arch.         2

Algorithm  2
```

说　明 :

1. 这程序包含二个 Pattern { Actions }指令.

Pattern	Actions
	{for(i=2; i< NF; i++) Number[$i]++ }
END	{ for(course in Number) printf("\%-10s \%d\n", course, Number[course])}

2. 第一个 Pattern { Actions }指令中省略了 Pattern 部分. 故随着每笔数据列的读入其 Actions 部分将逐次无条件被执行.以 AWK 读入第一笔资料 “Mary O.S. Arch. Discrete” 为例,因为该笔数据 NF = 4(有 4 个字段), 故该 Action 的 for Loop 中 i = 2,3,4.

i	$i	最初 Number[$i]	Number[$i]++ 之后
2	“O.S.”	AWK default Number[“O.S”]=0	1
3	“Arch.”	AWK default Number[“Arch”]=0	1
4	“Discrete”	AWK default Number[“Discrete”]=0	1

3. 第二个 Pattern { Actions }指令中 * { END}为 AWK 之保留字, 为{ Pattern}之一种. * { END}

成立(其值为 true)的条件是 :[0.3cm] AWK 处理完所有数据，即将离开程序时.
平常读入资料列时， END 并不成立，故其后的 Actions 并不被执行；唯有当 AWK 读完所有数据时，该 Actions 才会被执行 (注意,不管数据列有多少笔，END 仅在最后才成立，故该 Actions 仅被执行一次.)
{ BEGIN} 与 { END} 有点类似，是 AWK 中另一个保留的{Pattern}. 唯一不同的是 : 以 { BEGIN 为 Pattern 的 Actions} 于程序一开始执行时，被执行一次.

4. NF 为 AWK 的内建变量，用以表示 AWK 正处理的数据计列中，所包含的字段个数.AWK 程序中若含有以 $ 开头的自定变量，都将以如下方式解释 : 以 i= 2 为例，$i = $2 表第二个字段数据. (实际上，$ 在 AWK 中为一操作数(Operator)，用以取得字段数据.)

AWK 程序中使用 Shell 命令

AWK 程序中允许呼叫 Shell 指令. 并提供 pipe 解决 AWK 与系统间数据传递的问题. 所以 AWK 很容易使用系统资源. 读者可利用这个特色来撰写某些适用的系统工具.

范例 : 写一 AWK 程序来打印出线上人数.将下列程序建文件, 命名为 count.awk

```
BEGIN{
      while ("who" | getline) n++
      print n
     }
```

并执行下列命令 :

```
awk { -f} count.awk
```

执行结果将会印出目前在线人数

说 明 :

1. AWK 程序并不一定要处理资料文件. 以本例而言, 仅输入程序檔 count.awk, 未输入任何数据文件.
2. BEGIN 和 END 同为 AWK 中之种一 Pattern. 以 BEGIN 为 Pattern 之 Actions,只有在 AWK 开始执行程序,尚未开启任何输入檔前, 被执行一次.(注意: 只被执行一次)
3. “{|}” 为 AWK 中表示 pipe 的符号. AWK 把 pipe 之前的字符串“who”当成 Shell 上的命令, 并将该命令送往 Shell 执行, 执行的结果(原先应于屏幕印出者)则藉由 pipe 送进 AWK 程序中.
4. getline 为 AWK 所提供的输入指令. 其语法如下 :

语法	由何处读取数据	资料读入后置于
getline var < file	所指定的 file	变量 var(var 省略时,表示置于$0)
getline var	pipe	变量 var(var 省略时,表示置于$0)
getline var	见 注一	变量 var(var 省略时,表示置于$0)

注一 : 当 Pattern 为 BEGIN 或 END 时, getline 将由 stdin 读取数据,否则由 AWK 正处理的数据文件上读取数据.

注二 : getline 一次读取一行数据, 若读取成功则 return 1, 若读取失败则 return -1, 若遇到档案结束(EOF), 则 return 0; 本程序使用 getline 所 return 的数据 来做为 while 判断循环停止的条件,某些 AWK 版本较旧,并不容许使用者改变 $0 之值.

注三 : 这种版的 AWK 执行本程序时会产生 Error, 读者可于 getline 之后置上一个变量 (如此, getline 读进来的数据便不会被置于 $0), 或直接改用 gawk 便可解决.

AWK 程序的应用实例

本节将示范一个统计上班到达时间及迟到次数的程序.这程序每日被执行时将读入二个档案 :
员工当日到班时间的数据文件 (如下列之 arr.dat)
存放员工当月迟到累计次数的档案.
当程序执行执完毕后将更新第二个档案的数据(迟到次数), 并打印当日的报表.这程序将分成下列数小节逐步完成, 其大纲如下 :

[7.1] 于到班资料文件 {arr.dat} 之前端增加一列抬头 "ID Number Arrvial Time", 并产生报表输出到档案 today_rpt1 中 (在 AWK 中如何将数据输出到档案)
[7.2] 将 {today_rpt1} 上之数据按员工代号排序, 并加注执行当日之日期; 产生档案 today_rpt2 (AWK 中如何运用系统资源及 AWK 中 Pipe 之特性)
[7.3] (将 AWK 程序包含在一个 shell script 档案中)
[7.4] 于 today_rpt2 每日报表上, 迟到者之前加上"*",并加注当日平均到班时间;产生档案 today_rpt3
[7.5] 从档案中读取当月迟到次数, 并根据当日出勤状况更新迟到累计数.
(使用者于 AWK 中如何读取档案数据)

某公司其员工到勤时间檔如下, 取名为 {arr.dat}. 档案中第一栏为员工代号, 第二栏为到达时间. 本范例中, 将使用该档案为数据文件.

```
1034    7:26
1025    7:27
1101    7:32
1006    7:45
1012    7:46
1028    7:49
1051    7:51
1029    7:57
1042    7:59
1008    8:01
1052    8:05
1005    8:12
```

将数据直接输出到档案

AWK 中并未提供如 C 语言中之 fopen() 指令, 也未有 fprintf() 档案输出之指令. 但 AWK 中任何输出函数之后皆可藉由使用与 UNIX 中类似的 I/O Redirection , 将输出的数据 Redirect 到指定的档案; 其符号仍为 > (输出到一个新产生的档案) 或 >> (append 输出的数据到档案末端).

[例 :]于到班资料文件 arr.dat 之前端增加一列抬头如下 : "ID Number Arrival Time", 并产生报表输出到档案 today_rpt1 中.建立如下档案并取名为 reformat1.awk

```
BEGIN{ print " ID  Number  Arrival Time"  > "today_rpt1"
       print "==========================" > "today_rpt1"
     }
```

{ printf("   %s %s\n", $1,$2) > "today_rpt1" }

执行:

$awk -f reformat1.awk   arr.dat

执行后将产生档案 today_rpt1, 其内容如下 :

```
  ID  Number   Arrival Time
 ============================
     1034         7:26
     1025         7:27
     1101         7:32
     1006         7:45
     1012         7:46
     1028         7:49
     1051         7:51
     1029         7:57
     1042         7:59
     1008         8:01
     1052         8:05
     1005         8:12
```

说 明 :

1. AWK 程序中, 文件名称 today_rpt1 之前后须以" 括住, 表示 today_rpt1 为一字符串常数. 若未以"括住, 则 today_rpt1 将被 AWK 解释为一个变量名称.
   在 AWK 中任何变量使用之前, 并不须事先宣告. 其初始值为空字符串 (Null string) 或 0.因此程序中若未以 " 将 today_rpt1 括住, 则 today_rpt1 将是一变量, 其值将是空字符串, 这会于执行时造成错误(Unix 无法帮您开启一个以 Null String 为档名的档案).

   * 因此在编辑 AWK 程序时, 须格外留心. 因为若敲错变量名称, AWK 在编译程序时会认为是一新的变量, 并不会察觉. 如此往往会造成 RuntimeError.

2. BEGIN 为 AWK 的保留字, 是 Pattern 的一种. 以 BEGIN 为 Pattern 的 Actions 于 AWK 程序刚被执行尚未读取数据时被执行一次, 此后便不再被执行.

3. 读者或许觉得本程序中的 I/O Redirection 符号应使用 “ >>” (append)而非 “ >”.
   \index{ { >} } \index{ { >>} }
   * 本程序中若使用 “>” 将数据重导到 today_rpt1, AWK 第一次执行该指令时会产生一个新档 today_rpt1, 其后再执行该指令时则把数据 append 到 today_rpt1 文件末, 并非每执行一次就重开一个新档. 若采用">>"其差异仅在第一次执行该指令时, 若已存在 today_rpt1 则 AWK 将直接把数据 append 在原档案之末尾. 这一点, 与 UNIX 中的用法不同.

AWK 中如何利用系统资源

AWK 程序中很容易使用系统资源. 这包括于程序中途叫用 Shell 命令来处理程序中的部分数据; 或于呼叫 Shell 命令后将其产生之结果交回 AWK 程序(不需将结果暂存于某个档案). 这过程乃是藉由 AWK 所提供的 pipe (虽然有些类似 Unix 中的 pipe, 但特性有些不同),及一个从 AWK 中呼叫 Unix 的 Shell command 的语法来达成.

[例 :] 承上题, 将数据按员工 ID 排序后再输出到档案 today_rpt2, 并于表头附加执行时的日期.

分析 :

1. AWK 提供与 UNIX 用法近似的 pipe, 其记号亦为 “|”. 其用法及涵意如下 :
   AWK 程序中可接受下列两语法 :
   [a. 语法] AWK output 指令 | “Shell 接受的命令” (如 : print $1,$2 | "sort +1n")
   [b. 语法] “Shell 接受的命令” |AWK input 指令 (如 : "ls " | getline)
   注 : AWK input 指令只有 getline 一个. AWK output 指令有 print, printf() 二个.
2. 于 a 语法中, AWK 所输出的数据将转送往 Shell, 由 Shell 的命令进行处理.以上例而言, print 所印出的数据将经由 Shell 命令 “sort +1n” 排序后再送往屏幕(stdout).
   上列AWK程序中, “print$1, $2” 可能反复执行很多次, 其印出的结果将先暂存于 pipe 中, 等到该程序结束时, 才会一并进行 “sort +1n”.
   须注意二点 : 不论 print \$1, \$2 被执行几次, “sort +1n” 之 执行时间是 “AWK 程序结束时”, “sort +1n” 之 执行次数是 “一次”.

3. 于 b 语法中, AWK 将先叫用 Shell 命令. 其执行结果将经由 pipe 送入 AWK 程序以上例而言, AWK 先令 Shell 执行 “ls”, Shell 执行后将结果存于 pipe, AWK 指令 getline 再从 pipe 中读取资料.
   使用本语法时应留心 : 以上例而言 AWK “立刻”呼叫 Shell 来执行 “ls”, 执行次数是一次. getline 则可能执行多次(若 pipe 中存在多行数据).

4. 除上列 a, b 二语法外, AWK 程序中它处若出现像 "date", "cls", "ls"... 等字符串, AWK 只当成一般字符串处理之.

建立如下档案并取名为 reformat2.awk

```
#  程序 reformat2.awk
#  这程序用以练习 AWK 中的 pipe
BEGIN {
       "date"  |  getline # Shell 执行 “date”. getline 取得结果并以$0 记录
       print " Today is " , $2, $3 >"today_rpt2"
       print "========================" > "today_rpt2"
       print " ID  Number Arrival Time" >"today_rpt2"
       close("today_rpt2")
      }
{printf("%s  \%s\n", $1 ,$2)"sort +2n >>today_rpt2"}
```

执行如下命令:

```
awk -f   reformat2.awk  arr.dat
```

执行后，系统会自动将 sort 后的数据加(Append; 因为使用 “ >>”） 到档案 today_rpt2 末端. today_rpt2 内容如下 :

```
      Today is  Sep 17
=========================
 ID Number   Arrival Time
    1005         8:12
    1006         7:45
    1008         8:01
    1012         7:46
    1025         7:27
    1028         7:49
    1029         7:57
    1034         7:26
    1042         7:59
    1051         7:51
    1052         8:05
    1101         7:32
```

说 明 :

1. AWK 程序由三个主要部分构成 :
   [i.] Pattern { Action} 指令
   [ii.] 函数主体. 例如 : function double(x){ return 2*x }
          (参考第 11 节 Recursive Program)
   [iii.] Comment (以 # 开头识别之)
2. AWK 的输入指令 getline, 每次读取一列数据. 若 getline 之后未接任何变量, 则所读入之资料将以$0 纪录, 否则以所指定的变量储存之.
   以本例而言, 执行 "date" | getline 后, $0 之值为 "Wed Aug 17 11:04:44 EAT 1994". 当 $0 之值被更新时, AWK 将自动更新相关的内建变量, 如 $1,$2,..,NF.故 $2 之值将为"Aug", $3 之值将为"17".
   (有少数旧版之AWK不允许即使用者自行更新(update)$0之值,或者update$0时,它不会自动更新 $1,$2,..NF. 这情况下, 可改用 gawk,或 nawk. 否则使用者也可自行以 AWK 字符串函数 split()来分隔$0 上的资料)
3. 本程序中 printf() 指令会被执行 12 次(因为有 arr.dat 中有 12 笔数据), 但读者不用 担心数据被重复 sort 了 12 次. 当 AWK 结束该程序时才会 close 这个 pipe , 此时才将这 12 笔数据一次送往系统, 并呼叫 "sort +2n >> today_rpt2" 处理之.
4. AWK 提供另一个叫用 Shell 命令的方法, 即使用 AWK 函数 system("shell 命令") 例如 :

```
awk '
    BEGIN{
          system("date  > date.dat")
          getline  <date.dat
          print "Today is ", $2, $3
          }
```

，

但使用 system("shell 命令") 时，AWK 无法直接将执行中的部分数据输出给 Shell 命令. 且 Shell 命令执行的结果也无法直接输入到 AWK 中.

执行 AWK 程序的几种方式

本小节中描述如何将AWK程序直接写在 shell script 之中. 此后使用者执行 AWK 程序时, 就不需要每次都键入 “awk -f program datafile”.script 中还可包含其它 Shell 命令, 如此更可增加执行过程的自动化.建立一个简单的AWK程序 mydump.awk, 如下 :

```
{print}
```

这个程序执行时会把数据文件的内容 print 到屏幕上(与 cat 功用类似).print 之后未接任何参数时, 表示 “print $0”.若欲执行该AWK程序, 来印出档案 today_rpt1 及 today_rpt2 的内容时,必须于 UNIX 的命令列上执行下列命令 :

- 方式一 awk -f mydump.awk today_rpt1 today_rpt2
- 方式二 awk ’ print ’ today_rpt1 today_rpt2 第二种方式系将 AWK 程序直接写在 Shell 的命令列上, 这种方式仅适合较短的AWK程序.
- 方式三 建立如下之 shell script, 并取名为 mydisplay,

```
awk ’      # 注意 , awk 与 ’ 之间须有空白隔开
    {print}
    ’ $*  # 注意 , ’ 与 $* 之间须有空白隔开
```

执行 mydisplay 之前, 须先将它改成可执行的档案(此步骤往后不再赘述). 请执行如下命令:

```
$ chmod  +x mydisplay
```

往后使用者就可直接把 mydisplay 当成指令, 来 display 任何档案.

例如 :

```
$ mydisplay  today_rpt1 today_rpt2
```

说 明 :

1. 在 script 档案 mydisplay 中, 指令”awk”与第一个 ’ 之间须有空格(Shell 中并无” awk’ ”指令).
   第一个 ’ 用以通知 Shell 其后为AWK程序.
   第二个 ’ 则表示 AWK 程序结束.
   故AWK程序中一律以"括住字符串或字符, 而不使用 ’ , 以免Shell混淆.
2. $* 为 shell script 中之用法, 它可用以代表命令列上 “mydisplay 之后的所有参数”.
   例如执行 :

```
$ mydisplay today_rpt1 today_rpt2
```

   事实上 Shell 已先把该指令转换成 :

```
awk ’
    { print}
    ’ today_rpt1 today_rpt2
```

   本例中, $* 用以代表 “today_rpt1 today_rpt2”. 在Shell的语法中,可用 $1 代表第一个参数, $2 代表第二个参数. 当不确定命令列上的参数个数时, 可使用 $* 表之.
3. AWK 命令列上可同时指定多个数据文件. 以 awk -f dump.awk today_rpt1 today_rpt2hf 为例 AWK 会先处理 today_rpt1, 再处理 today_rpt2. 此时若档案无法开启, 将造成错误.
   例如: 未存在档案"file_no_exist", 则执行 :

```
awk -f dump.awk file_no_exit
```

   将产生 Runtime Error(无法开启档案). 但某些 AWK 程序 “仅” 包含以 BEGIN 为 Pattern 的指令. 执行这种 AWK 程序时, AWK 并不须开启任何数据文件.此时命令列上若指定 一个不存在的数据文件,

并不会产生 “无法开启档案”的错误.(事实上 AWK 并未开启该档案)

例如执行:

awk ’BEGIN {print ″Hello,World!!″} ’ file_no_exist

该程序中仅包含以 BEGIN 为 Pattern 之 Pattern {actions}, AWK 执行时并不会开启任何数据文件; 故不会因不存在档案 file_no_exit 而产生 “ 无法开启档案”的错误.

4. AWK 会将 Shell 命令列上 AWK 程序(或 -f 程序文件名)之后的所有字符串, 视为将输入 AWK 进行处理的数据文件文件名.

若执行 AWK 的命令列上 “未指定任何数据文件文件名”, 则将 stdin 视为输入之数据来源, 直到输入 end of file(Ctrl-D)为止.读者可以下列程序自行测试, 执行如下命令 :

$awk -f dump.awk (未接任何资料文件文件名)

或

$ mydisplay (未接任何资料文件文件名)

将会发现 : 此后键入的任何数据将逐行复印一份于屏幕上. 这情况不是机器当机 ! 是因为AWK程序正处于执行中. 它正按程序指示,将读取数据并重新 dump 一次; 只因执行时未指定数据文件文件名, 故 AWK 便以 stdin(键盘上的输入)为数据来源.

读者可利用这个特点, 设计可与 AWK 程序 interactive talk 的程序.

改变 AWK 切割字段的方式 & 使用者定义函数

AWK 不仅能自动分割字段，也允许使用者改变其字段切割方式以适应各种格式之需要. 使用者也可自定函数，若有需要可将该函数单独写成一个档案,以供其它 AWK 程序叫用.
范例 : 承接 6.2 的例子，若八点为上班时间，请加注 “*”于迟到记录之前，并计算平均上班时间.

分析:
1. 因八点整到达者,不为迟到，故仅以到达的小时数做判断是不够的；仍应参考到达时的分钟数. 若“将到达时间转换成以分钟为单位”,不仅易于判断是否迟到，同时也易于计算到达平均时间.
2. 到达时间($2)的格式为 dd:dd 或 d:dd; 数字当中含有一个 ″:″. 但文数字交杂的数据 AWK 无法直接做数学运算. (注: AWK 中字符串″26″与数字 26，并无差异，可直接做字符串或数学运算，这是 AWK 重要特色之一. 但 AWK 对文数字交杂的字符串无法正确进行数学运算).

解决之方法 :

- 方法一.

  对到达时间($2) d:dd 或 dd:dd 进行字符串运算,分别取出到达的小时数及分钟数. 首先判断到达小时数为一位或两位字符,再呼叫函数分别截取分钟数及小时数. 此解法需使用下列 AWK 字符串函数:
  length(字符串) : 传回该字符串之长度.
  substr(字符串,起始位置 ,长度) :传回从起始位置起，指定长度之子字符串. 若未指定长度，则传回起始位置到自串末尾之子字符串.
  所以:
  小时数 = substr($2, 1, length($2) - 3)
  分钟数 = substr($2, length($2) - 2)

- 方法二

  改变输入列字段的切割方式，使 AWK 切割字段后分别将小时数及分钟数隔开于二个不同的字段. 字段分隔字符 FS (field seperator) 是 AWK 的内建变数，其默认值是空白及 tab. AWK 每次切割字段时都会先参考 FS 的内容. 若把″:″也当成分隔字符，则 AWK 便能自动把小时数及分钟数分隔成不同的字段.故令
  FS = ″[\t:]+″ (注 : [\t:]+ 为一 Regular Expression)
  1. Regular Expression 中使用中括号 [...] 表一字符集合， 用以表示任意一个位于两中括号间的字符.故可用”[\t:]”表示 一个 空白 , tab 或 “:”
  2. Regular Expression 中使用 “+” 形容其前方的字符可出现一次或一次以上.故 “[\t:]+” 表示由一个或多个 “空白, tab 或 : ” 所组成的字符串.

  设定 FS =’’[\t:]+’’ 后，资料列如 : “1034 7:26” 将被分割成 3 个字段.

第一栏	第二栏	第三栏
$1	$2	$3
1034	7	26

  明显地,AWK 程序中使用方法一比方法二更简洁方便. 本范例中采用方法二,也藉此示范改变字段切割方式之用途.
  编写 AWK 程序 reformat3, 如下 :

```
awk ’
    BEGIN{
```

```
        {
          FS= "[ \t:]+"  #改变字段切割的方式
          "date"  | getline  # Shell 执行 “date”. getline 取得结果以$0 纪录
          print " Today is " ,$2, $3  > "today_rpt3"
          print "========================">"today_rpt3"
          print  “ ID  Number Arrival Time” > “today_rpt3”
          close("today_rpt3")
        }
        {
          #已更改字段切割方式, $2 表到达小时数, $3 表分钟数
          arrival = HM_to_M($2, $3)
          printf("  %s   %s:%s %s\n", $1,$2, $3, arrival  > 480 ? "*": " " ) | "sort
          +0n>>today_rpt3"
          total += arrival
          END
          {
            close("today_rpt3")   #参考本节说明 5
            close("sort +0n  >> today_rpt3")
            printf(" Average arrival time : %d:%d\n",
            total/NR/60, (total/NR)%60) >> "today_rpt3"
          }
   function HM_to_M(hour, min)
   {
       return  hour*60 + min
   }
   ' $*
```

并执行如下指令 :

```
$ reformat3  arr.doc
```

执行后,档案 today_rpt3 的内容如下:

```
     Today is  Sep 21
   ========================
    ID  Number Arrival Time
        1005      8:12  *
        1006      7:45
        1008      8:01  *
        1012      7:46
        1025      7:27
        1028      7:49
        1029      7:57
        1034      7:26
        1042      7:59
        1051      7:51
        1052      8:05  *
```

```
        1101       7:32
    Average arrival time : 7:49
```

{verbatim}

说 明 :

1. AWK 中亦允许使用者自定函数. 函数定义方式请参考本程序,function 为 AWK 的保留字.HM_to_M() 这函数负责将所传入之小时及分钟数转换成以分钟为单位. 使用者自定函数时,还有许多细节须留心, 如 data scope,…(请参考 第十节 Recursive Program)
2. AWK 中亦提供与 C 语言中相同的 Conditional Operator. 上式 printf()中使用 arrival >480 ? "*" : " "}即为一例若 arrival 大于 480 则 return "*" , 否则 return " ".
3. % 为 AWK 之运算子(operator), 其作用与 C 语言中之 % 相同(取余数).
4. NR(Number of Record) 为 AWK 的内建变数. 表 AWK 执行该程序后所读入的纪录笔数.
5. AWK 中提供的 close()指令, 语法如下(有二种) :
   * close(filename)
   * close(置于 pipe 之前的 command)

   为何本程序使用了两个 close() 指令 :
   - 指令 close("sort +2n >> today_rpt3"), 其意思为 close 程序中置于 "sort +2n >> today_rpt3 " 之前的 Pipe, 并立刻呼叫 Shell 来执行"sort +2n >> today_rpt3". (若未执行这指令, AWK 必须于结束该程序时才会进行上述动作;则这 12 笔 sort 后的数据将被 append 到档案 today_rpt3 中 "Average arrival time : ..." 的后方)
   - 因为 Shell 排序后的数据也要写到 today_rpt3, 所以 AWK 必须先关闭使用中的 today_rpt3 以利 Shell 正确将排序后的数据 append 到 today_rpt3 否则 2 个不同的 process 同时开启一档案进行输出将会产生不可预期的结果.
     读者应留心上述两点,才可正确控制数据输出到档案中的顺序.
   - 指令 close("sort +0n >> today_rpt3")中字符串 "sort +0n >> today_rpt3"须与 pipe | 后方的 Shell Command 名称一字不差, 否则 AWK 将视为二个不同的 pipe.
     读者可于 BEGIN{}中先令变数 Sys_call = "sort +0n >> today_rpt3",程序中再一律以 Sys_call 代替该字符串.

使用 getline 来读取数据

范 例 : 承上题,从档案中读取当月迟到次数, 并根据当日出勤状况更新迟到累计数.(按不同的月份累计于不同的档案)

分 析:
程序中自动抓取系统日期的月份名称, 连接上”late.dat”, 形成累计迟到次数的文件名称(如 : Jullate.dat,...), 并以变数 late_file 纪录该文件名.累计迟到次数的档案中的数据格式为 : “员工代号(ID) 迟到次数”, 例如, 执行本程序前档案 Auglate.dat 的内容为 :

```
1012 0
1006 1
1052 2
1034 0
1005 0
1029 2
1042 0
1051 0
1008 0
1101 0
1025 1
1028 0
```

编写程序 reformat4.awk 如下:

```
awk '
    BEGIN
    {
      Sys_Sort = "sort +0n >> today_rpt4"
      Result = "today_rpt4"
      # 改变字段切割的方式
      # 令 Shell 执行”date”; getline 读取结果,并以$0 纪录
      FS = "[\t:]+"
      "date" |  getline
      print " Today is " , $2, $3 >Result
      print "========================" > Result
      print " ID  Number Arrival Time" > Result
      close(Result)
      # 从文件按中读取迟到数据, 并用数组 cnt[ ]记录. 数组 cnt[ ]中以员工代号为
      # 注标, 所对应的值为该员工之迟到次数.
      late_file = $2 "late.dat"
      while(getline  < late_file >0)  cnt[$1] = $2
      close(late_file)
    }
    {
      # 已更改字段切割方式, $2 表小时数,$3 表分钟数
```

```
    arrival = HM_to_M($2, $3)
    if(arrival > 480)
    {
      mark = "*"  # 若当天迟到,应再增加其迟到次数, 且令  mark 为''*''.cnt[$1]++ }
      else   mark = " "
      # message 用以显示该员工的迟到累计数, 若未曾迟到 message 为空字符串
    message = cnt[$1] ? cnt[$1] " times" : ""
       printf("%s%2d:%2d %5s %s\n", $1, $2, $3, mark,
     message) | Sys_Sort
            total += arrival
  }
   END  {
            close(Result)
            close(Sys_Sort)
            printf(" Average arrival time : %d:%d\n", total/NR/60,
        (total/NR)%60) >> Result
            #将数组 cnt[ ]中新的迟到数据写回档案中
            for(any in cnt)
                  print any, cnt[any] > late_file
           }
     function HM_to_M(hour, min){
                  return hour*60 + min
            }
' $*
```

执行后, today_rpt4 之内容如下 :

```
       Today is  Aug 17
  ================================
    ID Number  Arrival Time
       1005        8:12       * 1 times
       1006        7:45         1 times
       1008        8: 1       * 1 times
       1012        7:46
       1025        7:27         1 times
       1028        7:49
       1029        7:57         2 times
       1034        7:26
       1042        7:59
       1051        7:51
       1052        8: 5       * 3 times
       1101        7:32
   Average arrival time : 7:49
```

说　明：

1. latefile 是一变量，用以记录迟到次数的档案之档名. latefile 之值由两部分构成，前半部是当月月份名称(由呼叫"date"取得)后半部固定为"late.dat" 如 : Junlate.dat.
2. 指令 getline <latefile 表由 latefile 所代表的档案中读取一笔纪录，并存放于$0.若使用者可自行把数据放入$0，AWK 会自动对这新置入 $0 的数据进行字段分割. 之后程序中可用$1，$2,..来表示该笔资料的第一栏,第二栏,..，(注：有少数 AWK 版本不容许使用者自行将数据置于 $0，遇此情况可改用 gawk 或 nawk)执行 getline 指令时，若成功读取纪录,它会传回 1. 若遇到档案结束，它传回 0；无法开启档案则传回-1.
3. 利用 while(getline < filename >0) {....}可读入档案中的每一笔数据并予处理. 这是 AWK 中 user 自行读取档案数据的一个重要模式.
4. 数组 late_cnt[ ] 以员工 ID. 当注标(index)，其对应值表其迟到的次数.
5. 执行结束后，利用 for(Variable in array){..}之语法
   for(any in late_cnt) print any, late_cnt[any]> latefile
   将更新过的迟到数据重新写回记录迟到次数之档案. 该语法于第 5 节中曾有说明.

处理 Multi-line 记录

AWK 每次从数据文件中只读取一笔 Record, 进行处理. AWK 系依照其内建变量 RS(Record Separator) 的定义将档案中的数据分隔成一笔一笔的 Record. RS 的默认值是 "\n"(跳行符号), 故平常 AWK 中一行数据就是一笔 Record. 但有些档案中一笔 Record 涵盖了数行数据, 这种情况下不能再以 "\n" 来分隔 Records. 最常使用的方法是相邻的Records之间改以 一个空白行 来隔开. 在AWK程序中, 令 RS=""(空字符串)后, AWK 把会空白行当成来档案中 Record 的分隔符. 显然 AWK 对 RS = "" 另有解释方式,简略描述如下, 当 RS = "" 时 :

1. 数个并邻的空白行, AWK 仅视成一个单一的 Record Saparator. (AWK 不会于两个紧并的空白行之间读取一笔空的 Record)
2. AWK 会略过(skip)檔首或檔末的空白行. 故不会因为档首或档末的空白行,造成 AWK 多读入了二笔空的数据.

请观察下例,首先建立一个数据文件 week.rpt 如下:

张长弓 GNUPLOT 入门

吴国强 Latex 简介 VAST-2 使用手册 mathematica 入门
李小华 AWK Tutorial Guide Regular Expression
该档案档首有数列空白行, 各笔 Record 之间使用一个或数个空白行隔开. 读者请细心观察, 当 RS = "" 时, AWK 读取该数据文件之方式. 编辑一个 AWK 程序档案 make_report 如下:

```
awk '
    BEGIN
    {
     FS = "\n"
     RS = "" split("一. 二. 三. 四. 五. 六. 七. 八. 九.", C\_Number, " ")
    }
    {
     printf("\n%s 报告人 : %s \n",C_Number[NR],$1)
     for(i=2; i { >}= NF; i++)
     printf(" %d. %s\n", i-1, $i)
    }
    ' $
```

执行 $ make_report week.rpt 屏幕产生结果如下:

一. 报告人 : 张长弓 1. GNUPLOT 入门
二. 报告人 : 吴国强 1. Latex 简介 2. VAST-2 使用手册 3. mathematica 入门
三. 报告人 : 李小华 1. AWK Tutorial Guide 2. Regular Expression

说明:

1. 本程序同时也改变字段分隔字符(FS= "\n"), 如此一笔数据中的每一行都是一个 field. 例如 : AWK 读入的第一笔 Record 为张长弓 GNUPLOT 入门其中 $1 指的是"张长弓", $2 指的是"GNUPLOT 入门"
2. 上式中的 C_Number[ ]是一个数组(array), 用以记录中文数字. 例如 : C_Number[1] = "一", C_Number[2] = "二" 这过程使用 AWK 字符串函数 split() 来把中文数字放进数组 Number[ ]中. 函数 split()用法如下 :

split(原字符串, 数组名, 分隔字符(field separator))
AWK 将依所指定的分隔字符(field separator)分隔原字符串成一个个的字段(field), 并以指定的数组 记录各个被分隔的字段

如何读取命令列上的参数

大部分的应用程序都容许使用者于命令之后增加一些选择性的参数.执行 AWK 时这些参数大部分用于指定数据文件文件名，有时希望在程序中能从命令列上得到一些其它用途的数据. 本小节中将叙述如何在 AWK 程序中取用这些参数.

建立档案如下，命名为 see_arg ：

```
{
  awk '
      BEGIN
      {
       for(i=0; i<ARGC ; i++)
       print ARGV[i]  # 依次印出 AWK 所纪录的参数
      }
 '$*
```

执行如下命令 ：

```
$  see_arg  first-arg  second-arg
```

结果屏幕出现 ：

```
awk
first-arg
second-arg
```

说明 ：

1. ARGC, ARGV[ ] 为 AWK 所提供的内建变量.
   - ARGC : 为一整数. 代表命令列上，除了选项-v, -f 及其对应的参数之外所有参数的数目.
   - ARGV[ ] : 为一字符串数组. ARGV[0],ARGV[1],...ARGV[ARGC-1]. 分别代表命令列上相对应的参数.

   例如，当命令列为 ：

   ```
   $awk  -vx=36  -f program1  data1  data2
   ```

   或

   ```
   awk '{ print $1 ,$2 }'  data1  data2
   ```

   其 ARGC 之值为 3，ARGV[0] 之值为"awk"，ARGV[1] 之值为"data1"，ARGV[2] 之值为 "data2"。命令列上的 "-f program1"," -vx=36", 或程序部分 '{ print $1, $2}' 都不会列入 ARGC 及 ARGV[ ] 中.

2. AWK 利用 ARGC 来判断应开启的数据文件个数.但使用者可强行改变 ARGC; 当 ARGC 之值被使用者设为 1 时; AWK 将被蒙骗,误以为命令列上并无数据文件文件名，故不会以 ARGV[1], ARGV[2],..为文件名来开文件读取数据; 但于程序中仍可藉由 ARGV[1],ARGV[2],..来取得命令列上的资料.

   某一程序 test1.awk 如下 ：

   ```
   BEGIN{
          number = ARGC    #先用 number 记住实际的参数个数.
          ARGC = 2 # 自行更改 ARGC=2, AWK 将以为只有一个资料文件
                   # 仍可藉由 ARGV[ ]取得命令列上的资料.
          for(i=2; i< number; i++) data[i] = ARGV[i]
        }
   ```

.....

于命令列上键入

$awk -f test1.awk data_file apple orange

执行时 AWK 会开启数据文件 data_file 以进行处理. 不会开启以 apple,orange 为档名的档案(因为 ARGC 被改成 2). 但仍可藉由 ARGV[2], ARGV[3]取得命令列上的参数 apple, orange

3. 可以下列命令来达成上例的效果.

$awk -f test2.awk -v data[2]="apple" -v data[3]="orange" data_file

---

撰写可与使用者相互交谈的 AWK 程序

执行 AWK 程序时, AWK 会自动由档案中读取数据来进行处理, 直到档案结束.只要将 AWK 读取数据的来源改成键盘输入, 便可设计与 AWK interactive talk 的程序.本节将提供一个该类程序的范例.

[范例 :] 本节将撰写一个英语生字测验的程序, 它将印出中文字意,再由使用者回答其英语生字.首先编辑一个数据挡 test.dat (内容不拘,格式如下)

```
apple    苹果
orange   柳橙
banana   香蕉
pear     梨子
starfruit 杨桃
bellfruit 莲雾
kiwi      奇异果
pineapple 菠萝
watermelon 西瓜
```

编辑 AWK 程序"c2e"如下:

```
awk '
     BEGIN
     {
       while(getline < ARGV[1])
       {                          #由指定的档案中读取测验数据
         English[++n] = $1        # 最后, n 将表示题目之题数
         Chinese[n] = $2
       }
       ARGV[1] = "-"              # "-"表示由 stdin(键盘输入)
       srand()                    # 以系统时间为随机数启始的种子
       question()                 #产生考题
     }
    {                             # AWK 自动读入由键盘上输入的数据(使用者回答的答案)
      if($1 != English[ind])
         print "Try again!"
      else
      {
        print "\nYou are right !! Press Enter to Continue --- "
        getline
        question()               #产生考题
      }
    }
    function question(){
      ind = int(rand()* n) + 1 #以随机数选取考题
      system("clear")
      print " Press\"ctrl-d\" to exit"
```

```
      printf("\n%s ", Chinese[ind] " 的英文生字是: ")
     }
  '$*
```

执行时键入如下指令 :

```
$c2e  test.dat
```

屏幕将产生如下的画面:

```
Press "ctrl-d " to exit
莲雾 的英文生字是:
```

若输入 bellfruit 程序将产生

```
    You are right !! Press Enter to Continue ---
```

说 明 :

1. 参数 test.dat (ARGV[1]) 表示储存考题的数据文件文件名. AWK 由该档案上取得考题资料后, 将 ARGV[1] 改成 "-"."-" 表示由 stdin(键盘输入) 数据. 键盘输入数据的结束符号 (End of file)是 Ctrl-d. 当 AWK 读到 Ctrl-d 时就停止由 stdin 读取数据.
2. AWK 的数学函数中提供两个与随机数有关的函数.
3. rand() : 传回介于 0 与 1 之间的(近似)随机数值. 0<RAND()<) Functions Built-in 的 AWK C 附录 (参考 seed. 函数起始的 rand() 会以执行时的日期与时间为 则 AWK x, 若省略了 rand(指定以 x 为 : srand(x) 来产生随机数. 为启始, seed 函数都将以同一个内定的 AWK 程序时,rand() 否则每次执行的 seed, 函数起始 除非使用者自行指定>

使用 AWK 撰写 Recusive Program

AWK 中除了函数的参数列(Argument List)上的参数(Arguments)外,所有变量不管于何处出现全被视为 Global variable. 其生命持续至程序结束 --- 该变量不论在 function 外或 function 内皆可使用, 只要变量名称相同所使用的就是同一个变量,直到程序结束. 因 Recusive 函数内部的变量, 会因它呼叫子函数(本身)而重复使用,故撰写该类函数时, 应特别留心.

例如 : 执行

```
awk '
    BEGIN
    {
      x = 35
      y = 45
      test_variable(x)
      printf("Return to  main : arg1= %d, x= %d, y= %d, z= %d\n",arg1, x, y, z)
    }
    function test_variable(arg1)
    {
      arg1++  # arg1 为参数列上的参数, 是 local variable. 离开此函数后将消失.
      y ++    # 会改变主式中的变量 y
      z = 55  # z 为该函数中新使用的变量, 主程序中变量 z 仍可被使用.
      printf("Inside the function: arg1=%d,x=%d, y=%d, z=%d\n", arg1, x, y, z)
    } '
```

结果屏幕印出

```
Inside the function: arg1= 36,x= 35, y= 46, z= 55
Return to main     : arg1= 0, x= 35, y= 46, z= 55
```

由上可知 :

- 函数内可任意使用主程序中的任何变量.
- 函数内所启用的任何变量(除参数外), 于该函数之外依然可以使用.

此特性优劣参半, 最大的坏处是式中的变量不易被保护, 特别是 recursive 呼叫本身, 执行子函数时会破坏父函数内的变量.权变的方法是 : 在函数的 Argument list 上虚列一些 Arguments.

函数执行中使用这些虚列的 Arguments 来记录不想被破坏的数据,如此执行子函数时就不会破坏到这些数据. 此外 AWK 并不会检查,呼叫函数时所传递的参数个数是否一致.

例如 : 定义 recursive function 如下 :

```
       function  demo(arg1)# 最常见的错误例子
       {
         ........
         for(i=1; i< 20 ; i++)
         {
           demo(x)
           # 又呼叫本身. 因为 i 是 global variable, 故执行完该子函数后
           # 原函数中的 i 已经被坏, 故本函数无法正确执行.
```

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

可将上列函数中的 i 虚列在该函数的参数列上, 如此 i 便是一个 local variable, 不会因执行子函数而被破坏. 将上列函数修改如下:

```
function  demo(arg1, i)
{
 ......
 for(i=1; i< 20; i++)
     {
      demo(x)#AWK 不会检查呼叫函数时, 所传递的参数个数是否一致
      .....
      }
 }
```

$0, $1,.., NF, NR,..也都是 global variable, 读者于 recusive function 中若有使用这些内建变量, 也应另外设立一些 local variable 来保存,以免被破坏.

范例 :以下是一个常见的 Recursive 范例. 它要求使用者输入一串元素(各元素间用空白隔开) 然后印出这些元素所有可能的排列.编辑如下的 AWK 式, 取名为 permu.awk ’

```
BEGIN
{
  print "请输入排列的元素,各元素间请用空白隔开"
  getline
  permutation($0, "")
  printf("\n 共 %d 种排列方式\n", counter)
}
function  permutation(main_lst, buffer, new_main_lst, nf, i, j)
{
  $0 = main_lst  # 把 main_lst 指定给$0 之后 AWK 将自动进行  字段分割.
  nf = NF        # 故可用 NF 表示 main_lst 上存在的元素个数.
#  BASE CASE : 当 main_lst 只有一个元素时.
  if(nf == 1)
  {
    print buffer main_lst # buffer 的内容连接(concate)上 main_lst 就
    counter++             # 是完成一次排列的结果
    return
  }
  # General Case : 每次从 main\_lst 中取出一个元素放到 buffer 中
  #  再用 main_lst 中剩下的元素 (new_main_lst) 往下进行排列
  else for(i=1; i< =nf ;i++)
  {
    $0 = main_lst        # $0($1,$2,..$j,,)为 Global variable 已被坏, 故重新
```

```
                        #  把 main\_lst 指定给\$0, 令 AWK 再做一次字段分割
    new_main_lst = ""
    for(j=1; j< =nf; j++) # concate new_main_lst
    if(j != i) new_main_lst = new_main_lst " " $j
               permutation(new_main_lst, buffer " " $i )
  }
}
'$*
```

执行 $ permu, 屏幕上出现请输入排列的元素,各元素间请用空白隔开若输入 1 2 3 结果印出

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

共 6 种排列方式

说 明 :

1. 有些较旧版的 AWK,并不容许使用者指定$0 之值. 此时可改用 gawk, 或 nawk.否则也可自行使用 split() 函数来分割 main_lst.
2. 为避免执行子函数时破坏 new_main_lst, nf, i, j 故把这些变数也列于参数列上. 如此, new_main_lst, nf, i, j 将被当成 local variable,而不会受到子函数中同名的变量影响. 读者宣告函数时,参数列上不妨将这些 “虚列的参数” 与真正用于传递信息的参数间以较长的空白隔开, 以便于区别.
3. AWK 中欲将字符串 concatenation(连接)时, 直接将两字符串并置即可(Implicit Operator). 例如 :

```
awk '
    BEGIN
    {
      A = "This "
      B = "is a "
      C = A B "key." # 变量 A 与 B 之间应留空白,否则''AB''将代表另一新变量.
      print C
    }
   '
```

   结果将印出

   This is a key.

4. AWK使用者所撰写的函数可再reuse, 并不需要每个AWK式中都重新撰写.将函数部分单读编写于一档案中, 当需要用到该函数时再以下列方式 include 进来.

   $ awk -f 函数档名 -f AWK 主程序文件名 数据文件文件名

Appendix A Patterns

AWK 藉由判断 Pattern 之值来决定是否执行其后所对应的 Actions.这里列出几种常见的 Pattern :

1. BEGIN

BEGIN 为 AWK 的保留字, 是一种特殊的 Pattern. BEGIN 成立(其值为 true)的时机是 : “AWK 程序一开始执行, 尚未读取任何数据之前”. 所以在 BEGIN { Actions } 语法中, 其 Actions 部份仅于程序一开始执行时被执行一次. 当 AWK 从数据文件读入数据列后, BEGIN 便不再成立, 故不论有多少数据列, 该 Actions 部份仅被执行一次.

一般常把 “与数据文件内容无关” 与 “只需执行丨次” 的部分置于该 Actions(以 BEGIN 为 Pattern)中.例如 :

```
BEGIN {
        FS = ″[ \t:]″   # 于程序一开始时, 改变 AWK 切割字段的方式
        RS = ″″         # 于程序一开始时, 改变 AWK 分隔数据列的方式
        count = 100     # 设定变量 count 的起始值
        print ″ This is a title line ″  # 印出一行 title
      }
       ..... # 其它 Pattern { Actions } .....
```

有些 AWK 程序甚至''不需要读入任何数据列''. 遇到这情况可把整个程序置于以 BEGIN 为 Pattern 的 Actions 中.例如 :

```
BEGIN {   print ″ Hello ! the Word ! ″ }
```

注意 :执行该类仅含 BEGIN { Actions } 的程序时, AWK 并不会开启任何数据文件进行处理.

2. END

END 为 AWK 的保留字, 是另一种特殊的 Pattern.END 成立(其值为 true)的时机与 BEGIN 恰好相反, 为 : “AWK 处理完所有数据, 即将离开程序时”. 平常读入资料列时, END 并不成立, 故其对应的 Actions 并不被执行; 唯有当 AWK 读完所有数据时, 该 Actions 才会被执行 注意 : 不管数据列有多少笔, 该 Actions 仅被执行一次.

3. Relational Expression

使用像 “ A Relation Operator B” 的 Expression 当成 Pattern. 当 A 与 B 存在所指定的关系(Relation)时, 该 Pattern 就算成立(true). 例如 :

```
length($0)< = 80  { print }
```

上式中 { length($0)<= 80 是一个 Pattern, 当 $0(数据列)之长度小于等于 80 时该 Pattern 之值为 true, 将执行其后的 Action (印出该资料列).

AWK 中提供下列 关系操作数(Relation Operator)

操作数	涵意
>	大于
<	小于
> =	大于或等于
<=	小于或等于

==	等于
!=	不等于
~	match
!~	not match

上列关系操作数除~(match)与!~(not match)外与 C 语言中之涵意一致.~(match) 与!~(match) 在 AWK 之涵意简述如下 :
若 A 表一字符串, B 表一 Regular Expression.
A ~B 判断 字符串 A 中是否 包含 能合于(match)B 式样的子字符串.
A !~B 判断 字符串 A 中是否 未包含 能合于(match)B 式样的子字符串.
例如 :
$0 ~ /program[0-9]+\.c/ \{ print }
$0 ~/program[0-9]+\.c/ }整个是一个 Pattern, 用来判断$0(资料列)中是否含有可 match /program[0-9]+\.c/ 的子字符串, 若$0 中含有该类字符串, 则执行 print (印出该列数据).Pattern 中被用来比对的字符串为$0 时(如本例), 可仅以 Regular Expression 部分表之. 故本例的 Pattern 部分$0 ~/program[0-9]+\.c/ 可仅用/program[0-9]+\.c/表之. (有关 match 及 Regular Expression 请参考 附录 E)

4. Regular Expression
直接使用 Regular Expression 当成 Pattern; 此为 $0 ~ Regular Expression 的简写. 该 Pattern 用以判断 $0(资料列) 中是否含有 match 该 Regular Expression 的子字符串; 若含有该成立(true) 则执行其对应的 Actions.
例如 : /^[0-9]*$/ print "This line is a integer !" 与{ $0 ~/^[0-9]*$/ { print "This line is a integer !" } 相同

5. Compound Pattern
之前所介绍的各种 Patterns, 其计算(evaluation)后结果为一逻辑值(True or False).AWK 中逻辑值彼此间可藉由&&(and),||(or), !(not)结合成一个新的逻辑值. 故不同 Patterns 彼此可藉由上述结合符号来结合成一个新的 Pattern. 如此可进行复杂的条件判断. 例 如 :
FNR > = 23 && FNR <=28 print " " $0
上式利用&& (and) 将两个 Pattern 求值的结果合并成一个逻辑值. 该式 将资料文件中 第 23 行 到 28 行 向右移 5 格(先印出 5 个空白字符)后印出. (FNR 为 AWK 的内建变量, 请参考 附录 D)

6. Pattern1 , Pattern2
遇到这种 Pattern, AWK 会帮您设立一个 switch(或 flag). 当 AWK 读入的资料列使得 Pattern1 成立时, AWK 会打开(turn on)这 switch. 当 AWK 读入的资料列使得 Pattern2 成立时, AWK 会关上(turn off)这个 switch.

该 Pattern 成立的条件是 :
当这个 switch 被打开(turn on)时 (包括 Pattern1, 或 Pattern2 成立的情况)例 如 :
FNR>= 23 && FNR< =28 { print " " $0 }

可改写为

```
FNR == 23 ,  FNR == 28            { print ″      ″ $0 }
```

说 明 :

- 当 FNR >= 23 时, AWK 就 turn on 这个 switch;
- 因为随着资料列的读入, AWK 不停的累加 FNR.
- 当 FNR = 28 时, Pattern2 FNR == 28 便成立, 这时 AWK 会关上这个 switch.当 switch 打开的期间, AWK 会执行 “print ″   ″ $0”

(FNR 为 AWK 的内建变量, 请参考 附录 D)

Appendix B Actions

Actions 是由下列指令(statement)所组成 :

```
expression (function calls, assignments..)
print expression-list
printf(format, expression-list)
if(expression) statement [else statement]
while(expression) statement
do statement while(expression)
for(expression; expression; expression) statement
for(variable in array) statement
delete
break
continue
next
exit [expression]
statement
```

AWK 中大部分指令与 C 语言中的用法一致, 此处仅介绍较为常用或容易混淆之指令的用法.

1. 流程控制指令

- if 指令

  语法 :

  ```
  if (expression) statement1 [else statement2 ]
  ```

  范例 :

  ```
  if($1> 25) print "The 1st field is larger than 25"
  else print "The 1st field is not larger than 25"
  ```

  (a)与 C 语言中相同, 若 expression 计算(evaluate)后之值不为 0 或空字符串, 则执行 statement1; 否则执行 statement2.

  (b)进行逻辑判断的 expression 所传回的值有两种, 若最后的逻辑值为 true, 则传回 1, 否则传回 0.

  (c)语法中 else statement2 以[ ] 前后括住表示该部分可视需要而予加入或省略.

- while 指令

  语法 :

  ```
  while(expression) statement
  ```

  范例 :

  ```
  while(match(buffer,/[0-9]+\.c/)){
    print "Find :" substr(buffer,RSTART, RLENGTH)
    buff = substr(buffer, RSTART + RLENGTH)
  }
  ```

  上列范例找出 buffer 中所有能合于(match) /[0-9]+.c/(数字之后接上 “.c” 的所有子字符串). 范例中 while 以函数 match()所传回的值做为判断条件. 若 buffer 中还含有合于指定条件的子

字符串(match 成功), 则 match()函数传回 1,while 将持续进行其后之 statement.

- do-while 指令

语法 :

```
do statement while(expression)
```

范例 :

```
do{
   print "Enter y or n ! "
   getline data <  "-"
} while(data !~ /^[YyNn]$/)
```

(a) 上例要求使用者从键盘上输入一个字符, 若该字符不是 Y, y, N, 或 n 则会不停执行该循环, 到读取正确字符为止.

(b)do-while 指令与 while 指令 最大的差异是 : do-while 指令会先执行 statement 而后再判断是否应继续执行. 所以, 无论如何其 statement 部分至少会执行一次.

- for Statement 指令(一)

语法 :

```
for(variable in  array) statement
```

范例 : 执行下列命令

```
awk '
    BEGIN{
    X[1]= 50; X[2]= 60; X["last"]= 70
    for(any in X)
                    printf("X[%d] = %d\n", any, X[any])
    }'
```

结果印出 :

```
X[2] = 60
X[last] = 70
X[1] = 50
```

(a)这个 for 指令, 专用以搜寻阵中所有的 index 值, 并逐次使用所指定的变量予以纪录. 以本例而言, 变量 any 将逐次代表 2, 1,及"last".

(b)以这个 for 指令, 所搜寻出的 index 之值彼此间并无任何次续关系.

(c)第 7 节 Arrays in AWK 中有该指令的使用范例, 及解说.

- for Statement 指令(二)

语法 :

```
for(expression1; expression2; expression3) statement
```

范例 :

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

说明 :

(a)上列范例用以计算 1 加到 10 的总合.

(b)expression1 常用于设定该 for 循环的起始条件, 如上例中的 i=1
expression2 用于设定该循环的停止条件, 如上例中的 i<= 10
expression3 常用于改变 counter 之值, 如上例中的 i++

- break 指令

break 指令用以强迫中断(跳离) for, while, do-while 等循环.
范例 :

```
while( getline < "datafile" > 0)
{
    if($1 == 0)         # 所读取的数据置于 $0
        break           # AWK 立刻把 $0 上新的字段数据
    else                # 指定给 $1, $2, ...$NF
    print $2 / $1
}
```

上例中, AWK 不断地从档案 {datafile}中读取数据, 当$1 等于 0 时, 就停止该执行循环.

- continue 指令

循环中的 statement 进行到一半时, 执行 continue 指令来掠过回圈中尚未执行的 statement.
范例 :

```
for(index in X_array)
{
  if(index !~ /[0-9]+/)  continue
  print "There is a digital index", index
}
```

上例中若 index 不为数字则执行 continue, 故将掠过(不执行)其后的指令.
需留心 continue 与 break 的差异 : 执行 continue 只是掠过其后未执行的 statement, 但并未跳离开该循环.

- next 指令

执行 next 指令时, AWK 将掠过位于该指令(next)之后的所有指令(包括其后的所有 Pattern { Actions }), 接着读取下一笔数据列, 继续从第一个 Pattern {Actions}执行起.
范例 :

```
/^[ \t]*$/
{ print "This is a blank line! Do nothing here !"
  next
}
$2 != 0   { print $1, $1/$2 }
```

上例中, 当 AWK 读入的资料列为空白行时(match /\^{}[\]*$/)除打印讯息外且执行 next, 故 AWK 将掠过其后的指令, 继续读取下一笔数据, 从头(第一个 Pattern \{ Actions \})执行起.

- exit 指令

执行 exit 指令时, AWK 将立刻跳离(停止执行)该 AWK 程序.

2. AWK 中的 I/O 指令

- printf 指令

  该指令与 C 语言中的用法相同, 可藉由该指令控制数据输出时的格式.

  语法 :

  ```
  printf("format", item1, item2,..)
  ```

  范 例 :

  ```
  id = "BE-2647";  ave = 89
  printf("ID# : %s   Ave Score : %d\n", id, ave)
  ```

  (a)结果印出 :

  ```
  ID# :BE-647  Ave Score : 89
  ```

  (b)format 部分系由 一般的字符串(String Constant) 及 格式控制字符(Formatcontrol letter, 其前会加上一个\%字符)所构成. 以上式为例 "ID# :" 及 " Ave Score :" 为一般字符串. %s 及 %d 为格式控制字符.

  (c)印出时, 一般字符串将被原封不动地印出. 遇到格式控制字符时,则依序把 format 后方之 item 转换成所指定的格式后印出.

  (d)有关的细节, 读者可从介绍 C 语言的书籍上得到较完整的介绍.

  (e)print 及 printf 两个指令, 其后可使用>或<> 将输出到 stdout 的数据 Redirct 到其它档案, 7.1 Redirect Output to Files 中有完整的范例说明.

- print 指令

  范 例 :

  ```
  id = "BE-267";  ave = 89
  print "ID# :", id, "Ave Score :"ave
  ```

  (a)结果印出 :

  ```
  ID# : BE-267  Ave Score :89
  ```

  (b)print 之后可接上字符串常数(Constant String)或变量. 它们彼此间可用”,” 隔开.

  (c)上式中, 字符串 "ID# :" 与变量 id 之间使用”,”隔开, 印出时两者之间会以自动 OFS(请参考 D 内建变量 OFS) 隔开. OFS 之值一般内定为 "一个空格符"

  (d)上式中字符串 "Ave Score :" 与变量 ave 之间并未以”,”隔开, AWK 会将这两者先当成字符串 concate 在一起(变成”Ave Score :89")后, 再予印出

  (e)print 及 printf 两个指令, 其后可使用> 或> 将输出到 stdout 的数据 Redirct 到其它档案, 7.1 Redirect Output to Files 中有完整的范例说明.

- getline 指令

语法	由何处读取数据	资料读入后置于
getline var> file	所指定的 file	变量 var(var 省略时,表示置于$0)
| getline var	pipe	变量 var(var 省略时,表示置于$0)
getline var	见 注一	变量 var(var 省略时,表示置于$0)

注一 : 当 Pattern 为 BEGIN 或 END 时, getline 将由 stdin 读取数据,否则由 AWK 正处理的数据文件上读取数据.

getline 一次读取一行数据, 若读取成功则 return 1, 若读取失败则 return -1, 若遇到档案

结束(EOF), 则 return 0

- close 指令

  该指令用以关闭一个开启的档案, 或 pipe(见下例)

  范例 :

```
awk '
BEGIN {  print "ID #   Salary" > "data.rpt" }
      {  print $1 , $2 * $3  | "sort +0n > data.rpt" }
END{  close("data.rpt")
      close("sort +0n > data.rpt")
      print " There are", NR, "records processed."
   }
```

  说 明 :

  (a)上例中, 一开始执行 print "ID # Salary" > "data.rpt" 指令来印出一行抬头. 它使用 I/O Redirection(>)将数据转输出到 data.rpt,故此时档案 {data.rpt} 是处于 Open 状态.

  (b)指令 print $1, $2 * $3 不停的将印出的资料送往 pipe(|), AWK 于程序将结束时才会呼叫 shell 使用指令 "sort +0n > data.rpt" 来处理 pipe 中的数据; 并未立即执行, 这点与 Unix 中 pipe 的用法不尽相同.

  (c)最后希望于档案 {data.rpt}之``末尾”处加上一行 "There are.....". 但此时, Shell 尚未执行 "sort +0n > data.rpt" 故各数据列排序后的 ID 及 Salary 等数据尚未写入 data.rpt. 所以得命令 AWK 提前先通知 Shell 执行命令 "sort +0n > data.rpt" 来处理 pipe 中的资料. AWK 中这个动作称为 close pipe. 系由执行 close("shell command")来完成. 需留心 close()指令中的 shell command 需与”|” 后方的 shell command 完全相同(一字不差), 较佳的方法是先以该字符串定义一个简短的变量, 程序中再以此变量代替该 shell command

  (d)为什么要执行 close("data.rpt") ? 因为 sort 完后的资料也将写到 data.rpt,而该档案正为 AWK 所开启使用(write)中, 故 AWK 程序中应先关闭 data.rpt. 以免造成因二个 processes 同时开启一个档案进行输出(write)所产生的错误.

- system 指令

  该指令用以执行 Shell 上 的 command.

  范例 :

```
DataFile = "invent.rpt"
system("rm " DataFile)
```

  说明 :

  (a)system("字符串")指令接受一个字符串当成 Shell 的命令. 上例中, 使用一个字符串常数 "rm " 连接(concate)一个变量 DataFile 形成要求 Shell 执行的命令.Shell 实际执行的命令为 “rm invent.rpt”.

- “|” pipe 指令

  “|” 配合 AWK 输出指令, 可把 output 到 stdout 的数据继续转送给 Shell 上的令一命

令%当成 input 的数据. “|” 配合 AWK getline 指令, 可呼叫 Shell 执行某一命令, 再以 AWK 的 getline 指令将该命令的所产生的数据读进 AWK 程序中.

范例 :

```
{ print $1, $2 * $3  | "sort +1n > result" }
"date" |  getline  Date_data
```

读者请参考 7.2 Using System Resources 其中有完整的范例说明.

3. AWK 释放所占用的内存的指令

AWK 程序中常使用数组(Array)来记忆大量数据. delete 指令便是用来释放数组中的元素所所占用的记忆空间.

范例 :

```
for(any in X_arr)
delete X_arr[any]
```

读者请留心, delete 指令一次只能释放数组中的一个”元素”.

4. AWK 中的数学操作数(Arithmetic Operators)

+(加), -(减), *(乘), /(除), %(求余数), ^(指数) 与 C 语言中用法相同

5. AWK 中的 Assignment Operators

=, +=, -=, *= , /=, %=, ^=

x += 5 的意思为 x = x + 5, 其余类推.

6. AWK 中的 Conditonal Operator

语法 :

判断条件 ? value1 : value2

若判断条件成立(true) 则传回 value1, 否则传回 value2.

7. AWK 中的逻辑操作数(Logical Operators)

&&(and), ||or, !(not)

Extended Regular Expression 中使用 “|” 表示 or 请勿混淆.

8. AWK 中的关系操作数(Relational Operators)

>, >=, <, < =, ==, !=, ~, !~

9. AWK 中其它的操作数

+(正号), -(负号), ++(Increment Operator), - -(Decrement Operator)

AWK 中各操作数的运算优先级(Precedence)

(按优先高低排列)

```
 $        (字段操作数, 例如 : i=3; $i 表示第 3 栏)
 ^        (指数运算)
 + ,- ,! (正,负号,及逻辑上的 not)
 * ,/ ,% (乘,除,余数)
 + ,-     (加,减)
```

>, > =,< , < =, ==, != (大于,大于等于,...,等关系符号)
~, !~ (match, not match)
&& (逻辑上的 and)
|| (逻辑上的 or)
? : (Conditional Operator)
= , +=, -=,*=, /=, %=, ^= (一些指定 Assignment 操作数)

AWK 的内建函数 Built-in Functions

(一). 字符串函数

- index(原字符串, 找寻的子字符串):
  若原字符串中含有欲找寻的子字符串,则传回该子字符串在原字符串中第一次出现的位置,若未曾出现该子字符串则传回 0.
  例如执行 :
  ```
  $awk  'BEGIN{ print index("8-12-94","-" }'
  ```
  结果印出 2

- length(字符串) : 传回该字符串的长度.
  例如执行 :
  ```
  awk  'BEGIN { print length("John") '}
  ```
  结果印出 4

- match(原字符串, 用以找寻比对的 Regular Expression :AWK 会在原字符串中找寻合乎 Regular Expression 的子字符串. 若合乎条件的子字符串有多个, 则以原字符串中最左方的子字符串为准.AWK 找到该字符串后会依此字符串为依据进行下列动作:
  1. 设定 AWK 内建变量 RSTART, RLENGTH :
     RSTART &= 合条件之子字符串在原字符串中之位置.
     &= 0 ; 若未找到合条件的子字符串.
     RLENGTH &= 合条件之子字符串长度.
     &= -1 ; 若未找到合条件的子字符串.
  2. 传回 RSTART 之值.
     例如执行 :
     ```
     awk ' BEGIN {
       match("banana", /(an)+/)
       print RSTART, RLENGTH
     }
     '
     ```
     执行结果印出 2 4

- split(原字符串, 数组名, 分隔字符(field separator):AWK 将依所指定的分隔字符(field separator)来分隔原字符串成一个个的字段(field),并以指定的数组记录各个被分隔的字段.
  例如 :
  ```
  ArgLst = "5P12p89"
  split(ArgLst, Arr, /[Pp]/)
  ```
  执行后 Arr[1]=5, Arr[2]=12, Arr[3]=89

- sprintf(打印之格式, 打印之数据, 打印之数据,...)该函数之用法与 AWK 或 C 的输出函数 printf()相同. 所不同的是 sprintf()会将要求印出的结果当成一个字符串传回一般最常使用 sprintf()来改变数据格式. 如: x 为一数值资料, 若欲将其变成一个含二位小数的数据,可执行如下指令 :

```
x = 28
x = sprintf("%.2f",x)}
```
执行后 x = "28.00"

- sub(比对用的 Regular Expression}, 将替换的新字符串, 原字符串)sub()将原字符串中第一个(最左边)合乎所指定的 Regular Expression 的子字符串改以新字符串取代.
  1. 第二个参数"将替换的新字符串"中可用"&"来代表"合于条件的子字符串"承上例,执行下列指令:
     ```
     A = "a6b12anan212.45an6a"
     sub(/(an)+[0-9]*/, "[&]", A)
     ```
     结果印出 ab12[anan212].45an6a
  2. sub()不仅可执行取代(replacement)的功用,当第二个参数为空字符串("")时,sub()所执行的是”去除指定字符串”的功用.
  3. 藉由 sub()与 match()的搭配使用,可逐次取出原字符串中合乎指定条件的所有子字符串.
     例如执行下列程序:
     ```
     awk '
         BEGIN{
           data = "p12-P34 P56-p61"
           while(match(data ,/[0-9]+/) >0){
             print substr(data,{ RSTART, RLENGTH })
             sub(/[0-9]+/,"")
           }
         }
     ' $*
     ```
     结果印出 :
     ```
     12
     34
     56
     61
     ```
  4. sub()中第三个参数(原字符串)若未指定,则其默认值为$0. 可用 sub(/[9-0]+/,"digital") 表示 sub(/[0-9]+/,"digital",$0)

- gsub(比对用的 Regular Expression}, 将替换的新字符串, 原字符串) 这个函数与 sub()一样,同样是进行字符串取代的函数. 唯一不同点是
  1. gsub()会取代所有合条件的子字符串.
  2. gsub()会传回被取代的子字符串个数.

  请参考 sub().

- substr(字符串,起始位置 [,长度]):传回从起始位置起,指定长度之子字符串. 若未指定长度,则传回起始位置到自串末尾之子字符串.
  执行下例
  ```
  awk {' BEGIN{ print  substr("User:Wei-Lin Liu", 6)}
  ```
  结果印出

Wei-Lin Liu

(二). 数学函数

- int(x) : 传回 x 的整数部分(去掉小数).
  例如 :
  int(7.8) 将传回 7
  int(-7.8) 将传回 -7

- sqrt(x) : 传回 x 的平方根.
  例如 :
  sqart(9) 将传回 3
  若 x 为负数,则执行 sqrt(x)时将造成 Run Time Error

- exp(x) : 将传回 e 的 x 次方.
  例如 :
  exp(1) 将传回 2.71828

- log(x) : 将传回 x 以 e 为底的对数值.
  例如 :
  log(e) = 1
  若 x< 0 ,则执行 sqrt(x)时将造成 Run Time Error.

- sin(x) : x 须以径度量为单位,sin(x)将传回 x 的 sin 函数值.

- cos(x) : x 须以径度量为单位,cos(x)将传回 x 的 cos 函数值

- atan2(y,x) : 传回 y/x 的 tan 反函数之值,传回值系以径度量为单位.

- rand() : 传回介于 0 与 1 之间的(近似)随机数值; 0 < rand()<1.
  除非使用者自行指定 rand()函数起始的 seed,否则每次执行 AWK 程序时, rand()函数都将使用同一个内定的 seed,来产生随机数.

- srand(x) : 指定以 x 为 rand()函数起始的 seed.
  若省略了 x,则 AWK 会以执行时的日期与时间为 rand()函数起始的 seed.

AWK 的内建变数 Built-in Variables

因内建变量的个数不多，此处按其相关性分类说明，并未按其字母顺序排列.

- ARGC 表命令列上除了选项 -F, -v, -f 及其所对应的参数之外的所有参数的个数.若将”AWK 程序”直接写于命令列上，则 ARGC 亦不将该”程序部分”列入计算.

- ARGV 一个数组用以记录命令列上的参数.
  例 : 执行下列命令
  ```
  $awk  -F\t  -v a=8 -f prg.awk  file1.dat file2.dat
  ```
  或
  ```
  $awk  -F\t  -v a=8 '{ print $1 * a }' file1.dat file2.dat
  ```
  执行上列任一程序后
  ARGC = 3
  ARGV[0] = "awk"
  ARGV[1] = "file1.dat"
  ARGV[2] = "file2.dat"
  读者请留心 : 当 ARGC = 3 时，命令列上仅指定 2 个数据文件.
  注 :
  -F\t 表示以 tab 为字段分隔字符 FS(field seporator).
  -v a=8 是用以 initialize 程序中的变量 a.

- FILENAME 用以表示目前正在处理的数据文件文件名.

- FS 字段分隔字符.
  $0 表示目前 AWK 所读入的资料列. $1,$2..分别表示所读入的资料列之第一栏，第二栏,..(参考下列说明)
  当 AWK 读入一笔资料列 “A123 8:15” 时,会先以$0 记载.故 $0 = "A123 8:15"，若程序中进一步使用了 $1，$2.. 或 NF 等内建变数时，AWK 才会自动分割 $0. 以便取得字段相关的数据. 切割后各个字段的数据会分别以 $1，$2，$3...予以记录. AWK 内定(default)的字段分隔字符(FS) 为空格符(及 tab).
  以本例而言，读者若未改变 FS，则分割后，第一栏($1)="A123"， 第二栏($2)="8:15". 使用者可用 Regexp 自行定义 FS. AWK 每次需要分割数据列时，会参考目前 FS 之值. 例如，令 FS = "[:]+" 表示任何由 空白" " 或 ":" 所组成的字符串都可当成分隔字符，则分割后: 第一栏($1) = "A123"，第二栏($2) = "8"，第三栏($3) = "15"

- NR 表从 AWK 开始执行该程序后所读取的数据列数.

- FNR 与 NR 功用类似. 不同的是 AWK 每开启一个新的数据文件,FNR 便从 0 重新累计

- NF 表目前的资料列所被切分的字段数.
  AWK 每读入一笔数据后，于程序中可以 NF 来得知该笔资料包含的字段个数.在下一笔数据被读入之前，NF 并不会改变. 但使用者若自行使用$0 来记录数据

例如 : 使用 getline, 此时 NF 将代表新的 $0 上所记载之数据的字段个数.

- OFS 输出时的字段分隔字符. 默认值 " "(一个空白), 详见下面说明.

- ORS 输出时数据列的分隔字符. 默认值 "\n"(跳行), 见下面说明.

- OFMT 数值数据的输出格式. 默认值 "%.6g"(若须要时最多印出 6 位小数)
  当使用 print 指令一次印出多项数据时,例如 : print $1, $2 印出数据时, AWK 会自动在 $1 与 $2 之间补上一个 OFS 之值(默认值为 一个空白). 每次使用 print 输出(印)数据后, AWK 会自动补上 ORS 之值.(默认值为 跳行) 使用 print 输出(印)数值数据时, AWK 将采用 OFMT 之值为输出格式.
  例如 : print 2/3
  AWK 将会印出 0.666667 ,程序中可藉由改变这些变量之值, 来改变指令 print 的输出格式.

- RS(Record Separator) : AWK 从数据文件上读取数据时, 将依 RS 之定义把资料切割成许多 Records, 而 AWK 一次仅读入一个 Record,以进行处理.RS 的默认值是 "\n". 所以一般 AWK 一次仅读入一行数据. 有时一个 Record 含括了几列资料(Multi-line Record). 这情况下不能再以"\n"来分隔并邻的 Records, 可改用 空白行 来分隔.在 AWK 程序中,令 RS = "" 表示以 空白行 来分隔并邻的 Records.

- RSTART 与使用字符串函数 match()有关之变量,详见下面说明.

- RLENGTH 与使用字符串函数 match()有关之变量.当使用者使用 match(...) 函数后, AWK 会将 match(...) 执行的结果以 RSTART,RLENGTH 记录之.请参考 附录 C AWK 的内建函数 match().

- SUBSEP(Subscript Separator) 数组中注标的分隔字符, 默认值为"\034"实际上, AWK 中的 数组 只接受 字符串 当它的注标, 如 : Arr["John"].但使用者在 AWK 中仍可使用 数字 当数组的注标, 甚至可使用多维的数组(Multi-dimenisional Array)如 :
  Arr[2,79]
  事实上, AWK 在接受 Arr[2,79] 之前, 就已先把其注标转换成字符串 "2\03479", 之后便以 Arr["2\03479"] 代替 Arr[2,79].可参考下例 :

```
awk '
    BEGIN{
      Arr[2,79] = 78
      print  Arr[2,79]
      print  Arr[ 2 , 79 ]
      print  Arr["2\03479"]
      idx = 2 SUBSEP 79
      print Arr[idx]
    }
    ' $*
```

执行结果印出:

```
78
78
78
```

Regular Expression 简介

- 为什么要使用 Regular Expression

  UNIX 中提供了许多 指令 或 tools, 它们具有在档案中 寻找(Search)字符串或置换(Replace)字符串 的功能. 像 grep, vi , sed, awk,... 不论是找寻字符串或置换字符串, 都得先 “告诉这些指令所要找寻(被置换)的字符串为何”.若未能预先明确知道所要找寻(被置换)的字符串为何, 只知该字符串存在的范围或特征时,例如 :

  (一)找寻 “T0.c”, “T1.c”, “T2.c”.... “T9.c” 当中的任一字符串.

  (二)找寻至少存在一个 “A”的任意字符串.

  这情况下, 如何告知执行找寻字符串的指令所要找寻的字符串为何.例 (一) 中, 要找寻任一在 “T” 与 “.c” 之间存在一个阿拉伯数字的字符串;当然您可以列举的方式, 一一把所要找寻的字符串告诉执行命令的指令.但例 (二) 中合于该条件的字符串有无限种可能, 势必无法一一列举.此时, 便需要另一种字符串表示的方法(协议).

- 什么是 Regular Expression

  Regular Expression(以下简称 (Regexp)是一种字符串表达的方式. 可用以指称具有某特征的所有字符串.

  注 : 为区别于一般字符串, 本附录中代表 Regexp 的字符串之前皆加 “Regexp”.注 : AWK 程序中常以/..../括住 Regexp; 以区别于一般字符串.

- 组成 Regular Expression 的元素

  普通字符 除了. * [ ] + ? () \ \^ $ 外之所有字符.由普通字符所组成的 Regexp 其意义与原字符串字面意义相同.例如 : Regexp “the” 与一般字符串的 “the” 代表相同的意义.

  1. Metacharacter : 用以代表任意一字符.

     须留心 UNIX Shell 中使用 “*”表示 Wildcard, 可用以代表任意长度的字符串.而 Regexp 中使用 “.” 来代表一个任意字符.(注意: 并非任意长度的字符串)Regexp 中 “*” 另有其它涵意, 并不代表任意长度的字符串.

  2. 表示该字符串必须出现于行首.
  3. $ 表示该字符串必须出现于行末.

  例如 :

  Regexp /^The/ 用以表示所有出现于行首的字符串 "The".

  Regexp /The$/ 用以表示所有出现于行末字符串 "The".

  4. \ 将特殊字符还原成字面意义的字符(Escape character)

     Regexp 中特殊字符将被解释成特定的意义. 若要表示特殊字符的字面(literal meaning)意义时, 在特殊字符之前加上"\"即可. 例如 :使用 Regexp 来表示字符串 “a.out”时, 不可写成 /a.out/. 因为 “.”是特殊字符, 表任一字符. 可合乎 Regexp / a.out/ 的字符串将不只 “a.out” 一个; 字符串 “a2out”, “a3out”, “aaout” ...都合于 Regexp /a.out/. 正确的用法为 : /a\.out/

  5. [...]字符集合, 用以表示两中括号间所有的字符当中的任一个.

     例如 : Regexp/[Tt]/ 可用以表示字符“T”或“t”. 故 Regexp/[Tt]he/ 表示字符串“The”或“the”. 字符集合[...]内不可随意留空白.

     例如 : Regexp /[Tt]/ 其中括号内有空格符, 除表示“T”, “t” 中任一个字符, 也可代表一

个 “ ”(空格符)

6. - 字符集合中可使用 “-” 来指定字符的区间, 其用法如下 :
   Regexp / [0-9]/ 等于 / [0123456789]/ 用以表示任意一个阿拉伯数字.同理 Regexp /[A-Z]/ 用以表示任意一个大写英文字母.但应留心 :
   Regexp /[0-9a-z]/ 并不等于 /[0-9][a-z]/; 前者表示一个字符, 后者表示二个字符.
   Regexp /[-9]/ 或 /[9-]/ 只代表字符 “9”或 “-”.
7. [^...]使用[^..] 产生字符集合的补集(complement set).其用法如下 :
   例如 : 要指定 “T” 或 “t” 之外的任一个字符, 可用 /[^Tt]/ 表之.同理 Regexp /[^a-zA-Z]/ 表示英文字母之外的任一个字符.须留心 “^” 之位置 : “^”必须紧接于”[″之后, 才代表字符集合的补集
   例如 :Regexp /[0-9\^]/ 只是用以表示一个阿拉伯数字或字符″^″.
8. * 形容字符重复次数的特殊字符.
   “*” 形容它前方之字符可出现 1 次或多次, 或不出现.
   例如 :
   Regexp /T[0-9]*\.c 中 * 形容其前 [0-9] (一个阿拉伯数字)出现的次数可为 0 次或 多次.故 Regexp /T[0-9]*\.c/ 可用以表示 “T.c”, “T0.c”, “T1.c”...”T9.c”
9. +形容其前的字符出现一次或一次以上.
   例如 : Regexp /[0-9]+/ 用以表示一位或一位以上的数字.
10. ? 形容其前的字符可出现一次或不出现.
    例如 : Regexp /[+-]?[0-9]+/ 表示数字(一位以上)之前可出现正负号或不出现正负号.
11. (...)用以括住一群字符,且将之视成一个 group(见下面说明)
    例如 :
    Regexp /12+/ 表示字符串 “12”, “122”, “1222”, “12222”,...
    Regexp /(12)+/ 表示字符串 “12”, “1212”, “121212”, “12121212”....
    上式中 12 以()括住, 故 “+”所形容的是 12, 重复出现的也是 12.
12. | 表示逻辑上的″或″(or)
    例如 :
    Regexp / Oranges? | apples? | water/ 可用以表示 : 字符串 “Orange”, “oranges” 或 “apple”, “apples” 或 “water”

- match 是什么 ?

讨论 Regexp 时, 经常遇到 “某字符串合于(match)某 Regexp”的字眼. 其意思为 : “这个 Regexp 可被解释成该字符串”.

[例如] :

字符串 ″the″ 合于(match) Regexp /[Tt]he/. 因为 Regexp /[Tt]he/ 可解释成字符串 ″the″ 或 ″The″, 故字符串 ″the″ 或 ″The″都合于(match) Regexp /[Th]he/.

AWK 中提供二个关系操作数(Relational Operator,见注一) ~ !~, 它们也称之为 match, not match. 但函义与一般常称的 match 略有不同. 其定义如下 :

A 表一字符串, B 表一 Regular Expression

只要 A 字符串中存在有子字符串可 match(一般定义的 match) Regexp B , 则 A ~B 就算成立, 其值为 true, 反之则为 false.

! ~ 的定义与~恰好相反.

{itemize}

例如 : "another" 中含有子字符串 "the" 可 match Regexp /[Tt]he/ , 所以"another" ~/[Tt]he/ 之值为 true.
[注 一] : 有些论著不把这两个操作数(~, !~)与 Relational Operators 归为一类.

- 应用 Regular Expression 解题的简例
下面列出一些应用 Regular Expression 的简例, 部分范例中会更动$0 之值, 若您使用的 AWK 不容许使用者更动 $0 时, 请改用 gawk.

1. 将档案中所有的字符串 "Regular Expression" 或 "Regular expression" 换成 "Regexp"
```
awk '
   {   gsub(/Regular[ \t]+[Ee]xpression/, "Regexp")
       print
   }
' $*
```
2. 去除档案中的空白行(或仅含空格符或 tab)
```
awk '
      $0 !~ /^[ \t]*$/  {  print   }
' $*
```
3. 在档案中俱有 ddd-dddd(电话号码型态, d 表 digital)的字符串前加上"TEL : "
```
awk '
    {   gsub(/[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]/, "TEL : &")
        print
    }
' $*
```
4. 从档案的 Fullname 中分离出 路径 与 文件名
```
awk '
 BEGIN{
        Fullname = "/usr/local/bin/xdvi"
        match(Fullname, /.*\//)
        path = substr(Fullname, 1, RLENGTH-1)
        name = substr(Fullname, RLENGTH+1)
        print "path :", path,"  name :",name
      }
     ' $*
```
结果印出
```
path : /usr/local/bin   name : xdvi
```
5. 将某一数值改以现金表示法表之(整数部分每三位加一撇,且含二位小数)
```
awk '
 BEGIN {
         Number = 123456789
         Number = sprintf("$%.2f",Number)
         while(match(Number,/[0-9][0-9][0-9][0-9]/))
                sub(/[0-9][0-9][0-9][.,]/, ",&", Number)
         print Number
```

```
        }
    ' $*
```

结果印出

$123,456,789.00

6. 把档案中所有具 “program 数字.f” 形态的字符串改为 “[Ref : program 数字.c]”

```
awk '
    {   while(match($0, /program[0-9]+\.f/) ){
           Replace = "[Ref : " substr($0, RSTART, RLENGTH-2) ".c]"
           sub(/program[0-9]+\.f/, Replace)
         }
        print
     }
  ' $*
```