# AWK Tutorial Guide

中央研究院計算中心
ASPAC 計劃
Email: aspac@beta.wsl.sinica.edu.tw
技術報告 ：94011

83 年 12 月 5 日 Version : 2.2

# ASPAC 計畫版權聲明

ASPAC (Academia Sinica PACkage) 是中央研究院計算中心關於“軟體工具使用” (Software Tools) 及 “問題解決” (Problem Solving) 的計畫。在這計畫下所發展之軟體及文件都屬於中央研究院計算中心所有。所有正式公開之電子形式資料（包括軟體及文件），在滿足下列軟體及文件使用權利說明下，都可免費取得及自由使用。

軟體及文件使用權利說明如下：

1. 軟體的使用權利：
   將沿用美國 FSF (Free Software Foundation) 1991年6月第二版的
   GNU General Public License。

2. 文件的使用權利：
   文件可以自由拷貝及引用，但不得藉以圖利。除非必要手續費的收取。

因一般的 PoorMan LaTeX中無法使用中文當成章節名稱,
所以本文中暫以英文來表示章節名稱.

# Contents

# 1 Preface

- 有關本手冊 :

  這是一本AWK學習指引, 其重點著重於 :

  AWK 適於解決哪些問題 ?
  AWK 常見的解題模式為何 ?

  為使讀者快速掌握AWK解題的模式及特性, 本手冊係由一些較具代表性的範例及其題解所構成; 各範例由淺入深, 彼此間相互連貫, 範例中並對所使用的AWK語法及指令輔以必要的說明. 有關AWK的指令, 函數,...等條列式的說明則收錄於附錄中, 以利讀者往後撰寫程式時查閱. 如此編排, 可讓讀者在短時間內順暢地學會使用AWK來解決問題. 建議讀者循著範例上機實習, 以加深學習效果.

- 讀者宜先具備下列背景 :

  a. UNIX 環境下的簡單操作及基本概念.
     例如 : 檔案編輯, 檔案複製 及 pipe, I/O Redirection 等概念
  b. C 語言的基本語法及流程控制指令. [1]
     例如 : printf(), while()...

- 參考書目 :

  本手冊是以學習指引為主要編排方式, 讀者若需要有關AWK介紹詳盡的參考書,可參考下列兩本書 :

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

[1] AWK 指令並不多, 且其中之大部分與 C 語言中之用法一致, 本手冊中對該類指令之語法及特性不再加以繁冗的說明, 讀者若欲深究,可自行翻閱相關的 C 語言書籍

# 2 Overview of AWK

## 2.1 Why AWK

AWK 是一種程式語言. 它具有一般程式語言常見的功能. 因AWK語言具有某些特點, 如 : 使用直譯器(Interpreter)不需先行編譯; 變數無型別之分(Typeless), 可使用文字當陣列的註標(Associative Array)...等特色. 因此, 使用AWK撰寫程式比起使用其它語言更簡潔便利且節省時間. AWK還具有一些內建功能, 使得AWK擅於處理具資料列(Record), 欄位(Field)型態的資料; 此外, AWK內建有pipe的功能, 可將處理中的資料傳送給外部的Shell命令加以處理, 再將Shell命令處理後的資料傳回AWK程式, 這個特點也使得AWK程式很容易使用系統資源.

由於AWK具有上述特色, 在問題處理的過程, 可輕易使用AWK來撰寫一些小工具; 這些小工具並非用來解決整個大問題, 它們只個別扮演解決問題過程的某些角色, 可藉由Shell所提供的pipe將資料按需要傳送給不同的小工具進行處理, 以解決整個大問題. 這種解題方式, 使得這些小工具可因不同需求而被重覆組合及使用(reuse); 也可藉此方式來先行測試大程式原型的可行性與正確性, 將來若需要較高的執行速度時再用C語言來改寫. 這是AWK最常被應用之處. 若能常常如此處理問題, 讀者可以以更高的角度來思考抽象的問題, 而不會被拘泥於細節的部份. 本手冊為AWK入門的學習指引, 其內容將先強調如何撰寫AWK程式, 未列入進一步解題方式的應用實例, 這部分將留待UNIX進階手冊中再行討論.

## 2.2 How to get AWK

一般的UNIX作業系統, 本身即附有AWK. 不同的UNIX作業系統所附的AWK其版本亦不盡相同. 若讀者所使用的系統上未附有AWK, 可透過 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

## 2.3 How AWK works

為便於解釋AWK程式架構, 及有關術語(terminology), 先以一個員工薪資檔( *emp.dat* ), 來加以介紹.

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

檔案中各欄位依次為 員工ID, 姓名, 薪資率,及 實際工時. ID中的第一碼為部門識別碼. "A","P"分別表示"組裝"及"包裝"部門.

本小節著重於說明AWK程式的主要架構及工作原理, 並對一些重要的名詞輔以必要的解釋. 由這部分內容, 讀者可體會出AWK語言的主要精神及AWK與其它語程式言的差異處. 為便於說明, 以條列方式說明於後.

- 名詞定義

  1. 資料列(Record) : AWK從資料檔上讀取資料的基本單位.

     以上列檔案 *emp.dat* 為例, AWK讀入的

     ```
     第一筆資料列是 "A125   Jenny  100  210"
     第二筆資料列是 "A341   Dan    110  215"
     ```

     一般而言, 一筆資料列相當於資料檔上的一行資料.
     (參考 : 附錄 B 內建變數"RS" )

  2. 欄位(Field) : 為資料列上被分隔開的子字串.

     以資料列"A125 Jenny 100 210"為例,

     | 第一欄 | 第二欄 | 第三欄 | 第四欄 |
     |---|---|---|---|
     | "A125" | "Jenny" | 100 | 210 |

     一般是以空白字元來分隔相鄰的欄位. ( 參考 : 附錄 D 內建變數"FS" )

- 如何執行AWK

  於UNIX的命令列上鍵入諸如下列格式的指令: ( "$"表Shell命令列上的提示符號)

  $ awk 'AWK程式' 資料檔檔名

  則AWK會先編譯該程式, 然後執行該程式來處理所指定的資料檔.
  (上列方式係直接把程式寫在UNIX的命令列上)

- AWK程式的主要結構 :

  AWK程式中主要語法是 Pattern { Actions }, 故常見之AWK程式其型態如下 :

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

- Pattern 是什麼 ?

  AWK 可接受許多不同型態的 Pattern. 一般常使用 "關係判斷式"(Relational expression) 來當成 Pattern.

  例如 :

  x > 34 是一個Pattern, 判斷變數 x 與 34 是否存在 大於 的關係.
  x == y 是一個Pattern, 判斷變數 x 與變數 y 是否存在 等於 的關係.

  上式中 x >34 , x == y 便是典型的Pattern.

  AWK 提供 C 語言中常見的關係運算元(Relational Operators) 如 >, <, >=, <=, ==, !=.
  此外, AWK 還提供 ~ (match) 及 !~ (not match) 二個關係運算元(註一). 其用法與涵義如下:

  若 A 表一字串, B 表一 Regular Expression

  A ~ B 判斷 字串A 中是否 包含 能合於(match)B式樣的子字串.
  A !~ B 判斷 字串A 中是否 未包含 能合於(match)B式樣的子字串.

  例如 :

  "banana" ~ /an/ 整個是一個Pattern.

  因為"banana"中含有可match /an/的子字串, 故此關係式

成立(true),
整個Pattern的值也是true.

相關細節請參考 附錄 A Patterns, 附錄 E Regular Expression

註 一：有少數AWK論著, 把 ~, !~ 當成另一類的 Operator, 並不視為一種 Relational Operator. 本手冊中將這兩個運算元當成一種 Relational Operator.

- Actions 是什麼?

  Actions 是由許多AWK指令構成. 而AWK的指令與 C 語言中的指令十分類似. 例如：

  AWK的 I/O指令 : print, printf( ), getline..
  AWK的 流程控制指令 : if(...){..} else{..}, while(...){...}...
  (請參考 附錄 B — "Actions" )

- AWK 如何處理 Pattern { Actions } ?

  AWK 會先判斷(Evaluate) 該 Pattern 之值, 若 Pattern 判斷(Evaluate)後之值為true(或不為0的數字,或不是空的字串), 則 AWK 將執行該 Pattern 所對應的 Actions.
  反之, 若 Pattern 之值不為 true, 則AWK將不執行該 Pattern 所對應的 Actions.

  例如：若AWK程式中有下列兩指令

  ```
        50 > 23          { print "Hello! The word!!" }
  "banana" ~ /123/       { print "Good morning !" }
  ```

  AWK會先判斷 50 >23 是否成立. 因為該式成立, 所以AWK將印出 "Hello! The word!!". 而另一 Pattern 為 "banana" ~/123/, 因為 "banana" 內未含有任何子字串可 match /123/, 該 Pattern 之值為 false, 故AWK將不會印出 "Good morning !"

- AWK 如何處理 { Actions } 的語法? ( 缺少Pattern部分 )

  有時語法 Pattern { Actions }中, Pattern 部分被省略, 只剩 {Actions}. 這種情形表示 "無條件執行這個 Actions".

- AWK 的欄位變數

AWK 所內建的欄位變數及其涵意如下 :

| 欄位變數 | 涵意 |
|---|---|
| $0 | 為一字串, 其內容為目前 AWK 所讀入的資料列. |
| $1 | 代表 $0 上第一個欄位的資料. |
| $2 | 代表 $0 上第二欄個位的資料. |
| ... | 其餘類推 |

讀入資料列時, AWK如何修正(update)這些內建的欄位變數

1. 當 AWK 從資料檔中讀取一筆資料列時, AWK 會使用內建變數 $0 予以記錄.
2. 每當 $0 被異動時 (例如 : 讀入新的資料列 或 自行變更 $0,...)
   AWK 會立刻重新分析 $0 的欄位情況, 並將 $0 上各欄位的資料用 $1, $2, ..予以記錄.

- AWK的內建變數(Built-in Variables)

  AWK 提供了許多內建變數, 使用者於程式中可使用這些變數來取得相關資訊.
  常見的內建變數有 :

| 內建變數 | 涵意 |
|---|---|
| NF | (Number of Fields)為一整數, 其值表$0上所存在的欄位數目. |
| NR | (Number of Records)為一整數, 其值表AWK已讀入的資料列數目. |
| FILENAME | AWK正在處理的資料檔檔名. |

例如 : AWK 從資料檔 *emp.dat* 中讀入第一筆資料列
"A125 Jenny 100 210" 之後, 程式中:

| | | | |
|---|---|---|---|
| $0 | 之值將是 "A125 Jenny 100 210" | | |
| $1 | 之值為 "A125" | $2 | 之值為 "Jenny" |
| $3 | 之值為 100 | $4 | 之值為 210 |
| NF | 之值為 4 | $NF | 之值為 210 |
| NR | 之值為 1 | FILENAME | 之值為 "emp.dat" |

- AWK的工作流程 :

  執行AWK時, 它會反複進行下列四步驟.

  1. 自動從指定的資料檔中讀取一筆資料列.
  2. 自動更新(Update)相關的內建變數之值. 如 : NF, NR, $0...
  3. 逐次執行程式中 所有 的 Pattern { Actions } 指令.

4. 當執行完程式中所有 Pattern { Actions } 時, 若資料檔中還有未讀取的資料, 則反覆執行步驟1到步驟4.

AWK會自動重覆進行上述4個步驟, 使用者不須於程式中撰寫這個迴圈(Loop).

# 3 How to Compute and Print Certain Fields

AWK 處理資料時, 它會自動從資料檔中一次讀取一筆記錄, 並會將該資料切分成一個個的欄位; 程式中可使用 $1, $2,... 直接取得各個欄位的內容. 這個特色讓使用者易於用 AWK 撰寫 reformatter 來改變資料格式.

範例 : 以檔案 *emp.dat* 為例, 計算每人應發工資並列印報表.

分析 : AWK 會自行一次讀入一列資料, 故程式中僅需告訴 AWK 如何處理所讀入的資料列.

執行如下命令 : ( $ 表UNIX命令列上的提示符號 )

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

執行結果如下 :
螢幕出現 :

Jenny 21000
Dan 23650
Max 27170
John 27500
Linda 19950

說 明 :

1. UNIX命令列上, 執行AWK的語法為:

   $ awk 'AWK程式' 欲處理的資料檔檔名.

   本範例中的 程式部分 為 { *print $2, $3 * $4* } . 把程式置於命令列時, 程式之前後須以 ' 括住.

2. *emp.dat* 為指定給該程式處理的資料檔檔名.

3. 本程式中使用 : **Pattern** { **Actions** } 語法.

| Pattern | Actions |
|---|---|
| | *print $2, $3 * $4* |

Pattern 部分被省略, 表無任何限制條件. 故AWK讀入每筆資料列後都將無條件執行這個 Actions.

4. **print** 為AWK所提供的輸出指令, 會將資料輸出到 **stdout**(螢幕). print 的參數間彼此以 "**,**" 隔開, 印出資料時彼此間會以空白隔開.
(參考 附錄 D 內建變數OFS)

5. 將上述的 程式部分 儲存於檔案 *pay1.awk* 中. 執行命令時再指定AWK 程式檔之檔名. 這是執行AWK的另一種方式, 特別適用於程式較大的情況, 其語法如下:
$ awk **-f** AWK程式檔名 資料檔檔名

故執行下列兩命令,將產生同樣的結果.

$ awk **-f** *pay1.awk* *emp.dat*
$ awk *'{ print $2, $3 * $4 }' emp.dat*

讀者可使用"-f"參數,讓AWK主程式使用其它僅含 AWK函數 的檔案中的函數
其語法如下:
$ awk **-f** AWK主程式檔名 **-f** AWK函數檔名 資料檔檔名
(有關 AWK 中函數之宣告與使用於 7.4 中說明)

6. AWK中也提供與 C 語言中類似用法的 **printf()** 函數. 使用該函數可進一步控制資料的輸出格式.

編輯另一個AWK程式如下, 並取名為 *pay2.awk*

*{ printf("%6s Work hours: %3d Pay: %5d\n", $2, $3, $3*$4) }*

執行下列命令
$ awk **-f** *pay2.awk* *emp.dat*

執行結果螢幕出現:

```
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
```

# 4 Selection by Text Content and by Comparsion

**Pattern { Action }** 為AWK中最主要的語法. 若某 **Pattern**之值為真則執行它後方的**Action**. AWK中常使用"關係判斷式"**(Relational Expression)**來當成 **Pattern**.

AWK中除了 >, <, ==, != ,... 等關係運算元( Relational Operators )外, 另外提供 ~ (match), !~ (Not Match) 二個關係運算元. 利用這兩個運算元, 可判斷某字串是否包含能符合所指定 Regular Expression 的子字串.

由於這些特性, 很容易使用AWK來撰寫需要字串比對, 判斷的程式.

範例 : 承上例,

1. 組裝部門員工調薪5%,(組裝部門員工之ID.係以"A"開頭)
2. 所有員工最後之薪資率若仍低於100, 則以100計.
3. 撰寫AWK程式列印新的員工薪資率報表.

分析 :

這個程式須先判斷所讀入的資料列是否合於指定條件, 再進行某些動作. AWK中 Pattern { Actions } 的語法已涵蓋這種 "**if** ( 條件 ) { 動作 } "的 架構.

編寫如下之程式, 並取名 *adjust1.awk*

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

執行下列命令 :

$ awk **-f** *adjust1.awk* *emp.dat*

結果如下 :

螢幕出現 :

```
A125   Jenny     105
A341   Dan       115
P158   Max       130
P148   John      125
A123   Linda     100
```

說 明 :

1. AWK的工作程序是:
   從資料檔中每次讀入一筆資料列, 依序執行完程式中所有的Pattern{ Action }指令

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

   再從資料檔中讀進下一筆記錄繼續進行處理.

2. 第一個 Pattern { Action }是:

   *$1 ~ /^A.*/ { $3 *= 1.05 }*

   $1 ~ /^A.*/ 是一個Pattern, 用來判斷該筆資料列的第一欄是否包含以"A"開頭的子字串. 其中 /^A.*/ 是一個Regular Expression, 用以表示任何以"A"開頭的字串. (有關 Regular Expression 之用法 參考 附錄 E ).

   Actions 部分為 $3 *= 1.05 . $3 *= 1.05 與 $3 = $3 * 1.05 意義相同. 運算子"*=" 之用法則與 C 語言中一樣. 此後與 C 語言中用法相同的運算子或語法將不予贅述.

3. 第二個 Pattern { Actions } 是:

   *$3 <100 { $3 = 100 }*

   若第三欄的資料內容(表薪資率)小於100, 則調整為100.

4. 第三個 Pattern { Actions } 是:

   *{ printf("%s %-8s %d\n",$1, $2, $3 ) }*

   省略了Pattern(無條件執行Actions), 故所有資料列調整後的資料都將被印出.

# 5 Arrays in AWK

AWK程式中允許使用字串當做陣列的註標(index). 利用這個特色十分有助於資料統計工作. (使用字串當註標的陣列稱為 Associative Array)

首先建立一個資料檔, 並取名為 $reg.dat$. 此為一學生註冊的資料檔; 第一欄為學生姓名, 其後為該生所修課程.

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

AWK中陣列的特性

1. 使用字串當陣列的註標(index).

2. 使用陣列前不須宣告陣列名稱及其大小.

   例如 : 希望用陣列來記錄 $reg.dat$ 中各門課程的修課人數.
   這情況,有二項資訊必須儲存 :

   (a) 課程名稱, 如 : "O.S.","Arch.".. ,共有哪些課程事前並不明確.

   (b) 各課程的修課人數. 如 : 有幾個人修"O.S."

   在AWK中只要用一個陣列就可同時記錄上列資訊. 其方法如下 :
   使用一個陣列 $Number[\ ]$ :

   * 以課程名稱當 Number[ ] 的註標.
   * 以 Number[ ] 中不同註標所對映的元素代表修課人數.

   例如 :
   有2個學生修 "O.S.", 則以 Number["O.S."] = 2 表之.
   若修"O.S."的人數增加一人,

   則 Number["O.S."] = Number["O.S."] + 1
   或 Number["O.S."]++ .

3. 如何取出陣列中儲存的資訊

   以 C 語言為例, 宣告 int Arr[100]; 之後, 若想得知 Arr[ ]中所儲存的資料, 只

須用一個迴圈, 如 :

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

即可. 上式中 :

陣列 Arr[ ] 的註標 : 0, 1, 2,..., 99
陣列 Arr[ ] 中各註標所對應的值 : Arr[0], Arr[1],...Arr[99]

但 AWK 中使用陣列並不須事先宣告. 以剛才使用的 Number[ ] 而言, 程式執行前, 並不知將來有哪些課程名稱可能被當成 Number[ ] 的註標.

AWK 提供了一個指令, 藉由該指令AWK會自動找尋陣列中使用過的所有註標. 以 Number[ ] 為例, AWK將會找到 "O.S.", "Arch."",...

使用該指令時, 須指定所要找尋的陣列, 及一個變數. AWK會使用該的變數來記錄從陣列中找到的每一個註標. 例如

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

指定用 *course* 來記錄 AWK 從 *Number[ ]* 中所找到的註標. AWK每找到一個註標時, 就用 *course* 記錄該註標之值且執行{....}中之指令. 藉由這個方式便可取出陣列中儲存的資訊.(詳見下例)

範例 : 統計各科修課人數,並印出結果.

建立如下程式,並取名為 *course.awk* :

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

執行下列命令 :

$ awk -f *course.awk reg.dat*

執行結果如下 :

```
Discrete   3
D.S.       1
O.S.       2
Graphics   2
```

```
A.I.        1
Arch.       2
Algorithm   2
```

說 明 :

1. 這程式包含二個**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. 第一個Pattern { Actions }指令中省略了**Pattern** 部分. 故隨著每筆資料列的讀入其Actions部分將逐次無條件被執行.

   以AWK讀入第一筆資料 "Mary O.S. Arch. Discrete" 為例,
   因為該筆資料 NF = 4(有4個欄位), 故該 Action 的 for Loop中 i = 2,3,4.

| i | $i | 最初 Number[$i] | Number[$i]++ 之後 |
|---|---|---|---|
| 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. 第二個 Pattern { Actions }指令中

   * **END**為AWK之保留字, 為**Pattern**之一種.
   * **END**成立(其值為true)的條件是 :

   "AWK處理完所有資料, 即將離開程式時."

   平常讀入資料列時, **END**並不成立, 故其後的**Actions**並不被執行; 唯有當AWK讀完所有資料時, 該Actions才會被執行 ( 注意,不管資料列有多少筆, END僅在最後才成立, 故該Actions僅被執行一次.)

   **BEGIN** 與 **END** 有點類似, 是AWK中另一個保留的 **Pattern**.
   唯一不同的是 :

   "以 **BEGIN** 為 **Pattern** 的 **Actions** 於程式一開始執行時, 被執行一次."

4. **NF** 為AWK的內建變數, 用以表示AWK正處理的資料計列中, 所包含的欄位個數.

5. AWK程式中若含有以 $ 開頭的自定變數, 都將以如下方式解釋 :

以 $i = 2$ 為例, $\$i = \$2$ 表第二個欄位資料.

( 實際上, $ 在 AWK 中為一運算元(Operator), 用以取得欄位資料.)

# 6 Making Shell Command in an AWK Program

AWK程式中允許呼叫Shell指令. 並提供pipe解決AWK與系統間資料傳遞的問題. 所以AWK很容易使用系統資源. 讀者可利用這個特色來撰寫某些適用的系統工具.

範例 : 寫一AWK程式來列印出線上人數.

將下列程式建檔, 命名為 *count.awk*

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

並執行下列命令 :

```
$ awk  -f   count.awk
```

執行結果將會印出目前線上人數

說 明 :

1. AWK 程式並不一定要處理資料檔. 以本例而言, 僅輸入程式檔*count.awk* , 未輸入任何資料檔.

2. BEGIN 和 END 同為AWK中之種一 Pattern. 以 BEGIN 為 Pattern之Actions,只有在AWK開始執行程式,尚未開啟任何輸入檔前, 被執行一次.(注意: 只被執行一次 )

3. "|" 為 AWK 中表示 pipe的符號. AWK 把 pipe之前的字串"who"當成Shell上的命令, 並將該命令送往Shell執行, 執行的結果(原先應於螢幕印出者)則藉由pipe送進AWK程式中.

4. **getline** 為AWK所提供的輸入指令.

   其語法如下 :

| 語法 | 由何處讀取資料 | 資料讀入後置於 |
|---|---|---|
| getline var< file | 所指定的 file | 變數 var(var省略時,表示置於$0) |
| \| getline var | pipe | 變數 var(var省略時,表示置於$0) |
| getline var | 見 註一 | 變數 var(var省略時,表示置於$0) |

   註一 : 當 Pattern 為 BEGIN 或 END 時, getline 將由 stdin 讀取資料, 否則由AWK正處理的資料檔上讀取資料.

**getline** 一次讀取一行資料,

若讀取成功則return 1,
若讀取失敗則return -1,
若遇到檔案結束(EOF), 則return 0;

本程式使用 getline 所 return 的資料 來做為 while 判斷迴圈停止的條件.
某些AWK版本較舊,並不容許使用者改變 $0 之值. 這種版的 AWK 執行本程式時會產生Error, 讀者可於 getline 之後置上一個變數 (如此, getline 讀進來的資料便不會被置於 $0 ), 或直接改用gawk便可解決.

# 7 A Pratical Example

本節將示範一個統計上班到達時間及遲到次數的程式.

這程式每日被執行時將讀入二個檔案 :

員工當日到班時間的資料檔 ( 如下列之 *arr.dat* )
存放員工當月遲到累計次數的檔案.

當程式執行執完畢後將更新第二個檔案的資料(遲到次數), 並列印當日的報表. 這程式將分成下列數小節逐步完成, 其大綱如下 :

7.1 於到班資料檔 *arr.dat* 之前端增加一列抬頭 "ID Number Arrvial Time" , 並產生報表輸出到檔案 *today_rpt1* 中"

< 在AWK中如何將資料輸出到檔案 >

7.2 將 *today_rpt1* 上之資料按員工代號排序, 並加註執行當日之日期; 產生檔案 *today_rpt2*

< AWK中如何運用系統資源及AWK中Pipe之特性 >

7.3 < 將AWK程式包含在一個shell script檔案中 >

7.4 於 *today_rpt2* 每日報表上, 遲到者之前加上"*", 並加註當日平均到班時間; 產生檔案 *today_rpt3*

<AWK中如何改變欄位切割方式, 定義函數, 及函數呼叫 >

7.5 從檔案中讀取當月遲到次數, 並根據當日出勤狀況更新遲到累計數.

< 使用者於AWK中如何讀取檔案資料 >

某公司其員工到勤時間檔如下, 取名為 *arr.dat* . 檔案中第一欄為員工代號, 第二欄為到達時間. 本範例中, 將使用該檔案為資料檔.

```
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
```

## 7.1 Redirecting Output to Files

AWK中並未提供如 C 語言中之 *fopen()* 指令, 也未有 *fprintf()* 檔案輸出之指令. 但AWK中任何輸出函數之後皆可藉由使用與 UNIX 中類似的 **I/O Redirection**, 將輸出的資料 Redirect 到指定的檔案; 其符號仍為 **>** (輸出到一個新產生的檔案) 或 **>>** ( append 輸出的資料到檔案末端 ).

例 : 於到班資料檔 *arr.dat* 之前端增加一列抬頭如下 : "ID Number Arrival Time", 並產生報表輸出到檔案 *today_rpt1* 中

建立如下檔案並取名為 *reformat1.awk*

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

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

執行:

$ awk -f *reformat1.awk* *arr.dat*

執行後將產生檔案 *today_rpt1* , 其內容如下 :

```
 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
```

說 明 :

1. AWK程式中, 檔案名稱 $today\_rpt1$ 之前後須以 ” 括住, 表示 $today\_rpt1$ 為一字串常數. 若未以” 括住, 則 $today\_rpt1$ 將被AWK解釋為一個變數名稱.

   在AWK中任何變數使用之前, 並不須事先宣告. 其初始值為空字串 (Null string) 或 0. 因此程式中若未以 ” 將 $today\_rpt1$ 括住, 則 $today\_rpt1$ 將是一變數, 其值將是空字串, 這會於執行時造成錯誤(Unix 無法幫您開啟一個以Null String為檔名的檔案).

   * 因此在編輯 AWK程式時, 須格外留心. 因為若敲錯變數名稱, AWK在編譯程式時會認為是一新的變數, 並不會察覺. 如此往往會造成 Runtime Error.

2. **BEGIN** 為AWK的保留字, 是 Pattern 的一種.

   以 BEGIN 為 Pattern 的 Actions 於AWK程式剛被執行尚未讀取資料時被執行一次, 此後便不再被執行.

3. 讀者或許覺得本程式中的I/O Redirection符號應使用 “>>” (append) 而非 “>”.

   * 本程式中若使用 “>” 將資料重導到 $today\_rpt1$ , AWK第一次執行該指令時會產生一個新檔 $today\_rpt1$ , 其後再執行該指令時則把資料append到 $today\_rpt1$ 檔末, 並非每執行一次就重開一個新檔.
   * 若採用”>>”其差異僅在第一次執行該指令時, 若已存在 $today\_rpt1$ 則 AWK 將直接把資料append在原檔案之末尾.
   * 這一點, 與UNIX中的用法不同.

## 7.2 Using System Resources

AWK 程式中很容易使用系統資源. 這包括於程式中途叫用 Shell 命令來處理程式中的部分資料; 或於呼叫 Shell 命令後將其產生之結果交回 AWK 程式(不需將結果暫存於某個檔案). 這過程乃是藉由 AWK 所提供的 pipe (雖然有些類似 Unix 中的 pipe, 但特性有些不同), 及一個從 AWK 中呼叫 Unix 的 Shell command 的語法來達成.

例 : 承上題, 將資料按員工ID排序後再輸出到檔案 *today_rpt2* , 並於表頭附加執行時的日期.

分 析 :

1. AWK 提供與 UNIX 用法近似的 pipe, 其記號亦為 "|". 其用法及涵意如下 : AWK程式中可接受下列兩語法 :

   a. 語法　　AWK output 指令 | "Shell 接受的命令"
   ( 如 : print $1, $2 | "sort +1n" )

   b. 語法　　"Shell 接受的命令" | AWK input 指令
   ( 如 : "ls " | getline )

   註 : AWK input 指令只有 getline 一個.
   AWK output 指令有 print, printf() 二個.

2. 於 a 語法中, AWK所輸出的資料將轉送往 Shell, 由 Shell 的命令進行處理.

   以上例而言, print 所印出的資料將經由 Shell 命令 "sort +1n" 排序後再送往螢幕(stdout).

   上列AWK程式中, "print $1, $2" 可能反覆執行很多次, 其印出的結果將先暫存於 pipe 中, 等到該程式結束時, 才會一併進行 "sort +1n".

   須注意二點 : 不論 print $1, $2 被執行幾次,

   "sort +1n" 之 執行時間是 "AWK程式結束時",
   "sort +1n" 之 執行次數是 "一次".

3. 於 b 語法中, AWK將先叫用 Shell 命令. 其執行結果將經由 pipe 送入AWK程式

   以上例而言, AWK先令 Shell 執行 "ls", Shell 執行後將結果存於 pipe, AWK指令 getline 再從 pipe 中讀取資料.

   使用本語法時應留心 : 以上例而言

AWK “立刻”呼叫 Shell 來執行 “ls”, 執行次數是一次.
getline 則可能執行多次(若pipe中存在多行資料).

4. 除上列 a, b 二語法外, AWK程式中它處若出現像 ”date”, ”cls”, ”ls”... 等字串, AWK只當成一般字串處理之.

建立如下檔案並取名為 *reformat2.awk*

```
# 程式 reformat2.awk
# 這程式用以練習AWK中的pipe

BEGIN {
        "date" | getline       # Shell 執行 “date”. getline 取得結果並 以$0記錄
        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"  }
```

執行如下命令:

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

執行後, 系統會自動將 sort 後的資料加( Append; 因為使用 “>>”) 到檔案 *today_rpt2* 末端. *today_rpt2* 內容如下 :

```
      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
```

說 明 :

1. AWK程式由三個主要部分構成 :

   i. Pattern { Action } 指令

   ii. 函數主體. 例如 : function double( x ){ return 2*x } (參考第11節 Recursive Program )

   iii. Comment ( 以 # 開頭識別之 )

2. AWK 的輸入指令 getline, 每次讀取一列資料. 若getline之後未接任何變數, 則所讀入之資料將以$0 紀錄, 否則以所指定的變數儲存之.

   以本例而言 :
   執行 "date" | getline 後,
   $0 之值為 "Wed Aug 17 11:04:44 EAT 1994"

   當 $0 之值被更新時, AWK將自動更新相關的內建變數, 如 : $1,$2,..,NF.
   故 $2 之值將為"Aug", $3之值將為"17".

   (有少數舊版之AWK不允許即使用者自行更新(update)$0之值,或者update$0時, 它不會自動更新 $1,$2,..NF. 這情況下, 可改用gawk,或nawk. 否則使用者也可自行以AWK字串函數split()來分隔$0上的資料)

3. 本程式中 printf() 指令會被執行12次( 因為有*arr.dat* 中有12筆資料), 但讀者不用擔心資料被重複sort了12次. 當AWK結束該程式時才會 close 這個 pipe , 此時才將這12筆資料一次送往系統, 並呼叫 "sort +2n >> today_rpt2" 處理之.

4. AWK提供另一個叫用Shell命令的方法, 即使用AWK函數

   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" ) 時, AWK無法直接將執行中的部分資料輸出給 Shell 命令. 且 Shell 命令執行的結果也無法直接輸入到AWK中.

## 7.3 Execute AWK Programs

本小節中描述如何將AWK程式直接寫在 shell script 之中. 此後使用者執行 AWK 程式時, 就不需要每次都鍵入 "**awk -f** *program datafile* ". script 中還可包含其它 Shell 命令, 如此更可增加執行過程的自動化.

建立一個簡單的AWK程式 *mydump.awk* , 如下 :

```
{ print }
```

這個程式執行時會把資料檔的內容 print 到螢幕上( 與cat功用類似 ).
print 之後未接任何參數時, 表示 "print $0".

若欲執行該AWK程式, 來印出檔案 *today_rpt1* 及 *today_rpt2* 的內容時, 必須於 UNIX 的命令列上執行下列命令 :

方式一 $ awk -f mydump.awk today_rpt1 today_rpt2

方式二 $ awk '{ print }' today_rpt1 today_rpt2

第二種方式係將AWK程式直接寫在 Shell 的命令列上, 這種方式僅適合較短的AWK程式.

方式三 建立如下之 shell script, 並取名為 *mydisplay* ,

```
awk '          # 注意 , awk 與 ' 之間須有空白隔開
    { print }
' $*           # 注意 , ' 與 $* 之間須有空白隔開
```

執行 *mydisplay* 之前, 須先將它改成可執行的檔案(此步驟往後不再贅述).
請執行如下命令:

**$ chmod +x** *mydisplay*

往後使用者就可直接把 *mydisplay* 當成指令, 來display任何檔案.

例如 :

**$ mydisplay** today_rpt1 today_rpt2

說 明：

1. 在script檔案 *mydisplay* 中, 指令"awk"與第一個 ' 之間須有空格(Shell中並無"awk' "指令).

   第一個 ' 用以通知 Shell 其後為AWK程式.
   第二個 ' 則表示 AWK 程式結束.

   故AWK程式中一律以"括住字串或字元, 而不使用', 以免Shell混淆.

2. $* 為 shell script中之用法, 它可用以代表命令列上 "mydisplay 之後的所有參數".

   例如執行：

   $ mydisplay *today_rpt1 today_rpt2*

   事實上 Shell 已先把該指令轉換成：

```
awk '
        { print }
    '   today_rpt1  today_rpt2
```

   本例中, $* 用以代表 "today_rpt1 today_rpt2". 在Shell的語法中, 可用 $1 代表第一個參數, $2 代表第二個參數. 當不確定命令列上的參數個數時, 可使用 $* 表之.

3. AWK命令列上可同時指定多個資料檔.

   以 $ awk -f *dump.awk today_rpt1 today_rpt2* 為例

   AWK會先處理*today_rpt1*, 再處理 *today_rpt2*. 此時若檔案無法開啟, 將造成錯誤.

   例如: 未存在檔案"file_no_exist", 則執行：

   $ awk -f *dump.awk file_no_exit*

   將產生Runtime Error(無法開啟檔案).

   但某些AWK程式 "僅" 包含以 BEGIN 為Pattern的指令. 執行這種AWK程式時, AWK並不須開啟任何資料檔.
   此時命令列上若指定一個不存在的資料檔,並不會產生 "無法開啟檔案"的錯誤.(事實上AWK並未開啟該檔按)
   例如執行:

   $ awk *'BEGIN { print "Hello,World!!" }' file_no_exist*

   該程式中僅包含以 BEGIN 為 Pattern 之 Pattern {actions}, AWK執行時並不會開啟任何資料檔; 故不會因不存在檔案*file_no_exit* 而產生 "無法開啟檔案"的

錯誤.

4. AWK會將 Shell 命令列上AWK程式(或 -f 程式檔名)之後的所有字串, 視為將輸入AWK進行處理的資料檔檔名.

   若執行AWK的命令列上 "未指定任何資料檔檔名", 則將stdin視為輸入之資料來源, 直到輸入end of file( Ctrl-D )為止.

   讀者可以下列程式自行測試, 執行如下命令 :

   $ awk -f *dump.awk* (未接任何資料檔檔名)

   或

   **$ mydisplay** (未接任何資料檔檔名)

   將會發現 : 此後鍵入的任何資料將逐行複印一份於螢幕上. 這情況不是機器當機 ! 是因為AWK程式正處於執行中. 它正按程式指示,將讀取資料並重新dump一次; 只因執行時未指定資料檔檔名, 故AWK便以stdin(鍵盤上的輸入 )為資料來源.

   讀者可利用這個特點, 設計可與AWK程式interactive talk的程式.

## 7.4 Changing Field Separator & User Define Functions

AWK不僅能自動分割欄位, 也允許使用者改變其欄位切割方式以適應各種格式之需要. 使用者也可自定函數, 若有需要可將該函數單獨寫成一個檔案,以供其它AWK程式叫用.

範例 : 承接 6.2 的例子, 若八點為上班時間, 請加註 "*"於遲到記錄之前, 並計算平均上班時間.

分 析:

1. 因八點整到達者,不為遲到, 故僅以到達的小時數做判斷是不夠的; 仍應參考到達時的分鐘數. 若 "將到達時間轉換成以分鐘為單位",不僅易於判斷是否遲到, 同時也易於計算到達平均時間.
2. 到達時間($2)的格式為 dd:dd 或 d:dd; 數字當中含有一個 ":". 但文數字交雜的資料AWK無法直接做數學運算. (註: AWK中字串"26"與數字26, 並無差異, 可直接做字串或數學運算, 這是AWK重要特色之一. 但AWK對文數字交雜的字串無法正確進行數學運算).

解決之方法 :

方法一. 對到達時間($2) d:dd 或 dd:dd 進行字串運算,分別取出到達的小時數及分鐘數.

首先判斷到達小時數為一位或兩位字元,再呼叫函數分別截取分鐘數及小時數.

此解法需使用下列AWK字串函數:

**length**( 字串 ) : 傳回該字串之長度.

**substr**( 字串,起始位置 ,長度 ) : 傳回從起始位置起, 指定長度之子字串. 若未指定長度, 則傳回起始位置到自串末尾之子字串.

所以:

小時數 = substr( $2, 1, length($2) - 3 )
分鐘數 = substr( $2, length($2) - 2 )

方法二 改變輸入列欄位的切割方式, 使AWK切割欄位後分別將小時數及分鐘數隔開於二個不同的欄位.
欄位分隔字元 FS (field seperator) 是AWK的內建變數, 其預設值是空白及tab. AWK每次切割欄位時都會先參考 FS 的內容. 若把":"也當成分隔字

元, 則AWK 便能自動分把小時數及分鐘數分隔成不同的欄位.
故令

FS = "[ \t:]+"　　(註：[ \t:]+ 為一Regular Expression )

1. Regular Expression 中使用中括號 [ ... ] 表一字元集合, 用以表示任意一個位於兩中括號間的字元.

   故可用"[ \t:]"表示　一個 空白 , tab 或 ":"

2. Regular Expression中使用 "+" 形容其前方的字元可出現一次或一次以上.

   故 "[ \t:]+" 表示　由一個或多個 "空白, tab 或 : " 所組成的字串.

設定 FS ="[ \t:]+" 後, 資料列如 : "1034 7:26" 將被分割成3個欄位.

| 第一欄 | 第二欄 | 第三欄 |
|---|---|---|
| $1 | $2 | $3 |
| 1034 | 7 | 26 |

明顯地, AWK程式中使用方法一比方法二更簡潔方便. 本範例中採用方法二, 也藉此示範改變欄位切割方式之用途.

編寫AWK程式 *reformat3* , 如下 :

```
awk '
BEGIN {
        FS = "[ \t:]+"    #改變欄位切割的方式
        "date" | getline     # Shell 執行 "date". getline 取得結果以$0紀錄
        print "     Today is " , $2, $3 > "today_rpt3"
        print "==========================" > "today_rpt3"
        print " ID Number   Arrival Time" > "today_rpt3"
        close( "today_rpt3" )
        }
```

```
        {
                #已更改欄位切割方式, $2表到達小時數, $3表分鐘數
                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")     #參考本節說明 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
        }
' $*
```

並執行如下指令 :

$ **reformat3** *arr.doc*

執行後,檔案 *today_rpt3* 的內容如下:

```
        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
```

說 明 :

1. AWK 中亦允許使用者自定函數. 函數定義方式請參考本程式, **function** 為 AWK 的保留字. *HM_to_M* ( ) 這函數負責將所傳入之小時及分鐘數轉換成以分鐘為單位. 使用者自定函數時, 還有許多細節須留心, 如 data scope,..( 請參考第十節 Recursive Program)

2. AWK中亦提供與 C 語言中相同的 Conditional Operator. 上式printf()中使用 *arrival* >*480 ? "*" : " "* 即為一例

   若 *arrival* 大於 480 則return "*" , 否則return " ".

3. % 為AWK之運算子(operator), 其作用與 C 語言中之 % 相同(取餘數).

4. NR(Number of Record) 為AWK的內建變數. 表AWK執行該程式後所讀入的紀錄筆數.

5. AWK 中提供的 close( )指令, 語法如下(有二種) :

   1. close( **filename** )
   2. close( 置於pipe之前的**command** )

   為何本程式使用了兩個 close( ) 指令 :

   - 指令 clsoe( "sort +2n >> today_rpt3" ), 其意思為 close 程式中置於 "sort +2n >> today_rpt3 " 之前的 Pipe, 並立刻呼叫 Shell 來執行 "sort +2n >> today_rpt3".
     ( 若未執行這指令, AWK必須於結束該程式時才會進行上述動作; 則這12筆 sort後的資料將被 append 到檔案 *today_rpt3* 中 "Average arrival time : ..." 的後方)
   - 因為 Shell 排序後的資料也要寫到 *today_rpt3* , 所以AWK必須先關閉使用中的 *today_rpt3* 以利 Shell 正確將排序後的資料 append 到*today_rpt3* , 否則2個不同的 process 同時開啟一檔案進行輸出將會產生不可預期的結果.

   讀者應留心上述兩點,才可正確控制資料輸出到檔案中的順序.

6. 指令 close("sort +0n >> today_rpt3")中字串 "sort +0n >> today_rpt3" 須與 pipe | 後方的 Shell Command 名稱一字不差, 否則AWK將視為二個不同的 pipe.
   讀者可於BEGIN{ }中先令變數 Sys_call = "sort +0n >> today_rpt3", 程式中再一律以 Sys_call 代替該字串.

## 7.5 Using getline to Input File

範 例 : 承上題,從檔案中讀取當月遲到次數, 並根據當日出勤狀況更新遲到累計數.(按不同的月份累計於不同的檔案)

分 析 :

1. 程式中自動抓取系統日期的月份名稱, 連接上"late.dat", 形成累計遲到次數的檔案名稱(如 : *Jullate.dat* ,...), 並以變數 *late_file* 紀錄該檔名.

2. 累計遲到次數的檔案中的資料格式為 :

   員工代號(ID) 遲到次數

   例如, 執行本程式前檔案 *Auglate.dat* 的內容為 :

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

編寫程式 *reformat4.awk* 如下:

```
awk '
BEGIN {
        Sys_Sort = "sort +0n >> today_rpt4"
        Result = "today_rpt4"

        # 改變欄位切割的方式
        # 令 Shell執行"date"; getline 讀取結果,並以$0紀錄

        FS = "[ \t:]+"
        "date" | getline

        print "     Today is " , $2, $3 > Result
        print "==========================" > Result
        print " ID  Number   Arrival Time" > Result
```

```
        close( Result )

        # 從檔按中讀取遲到資料, 並用陣列cnt[ ]記錄. 陣列cnt[ ]中以員工代號為
        # 註標, 所對應的值為該員工之遲到次數.

        late_file = $2 "late.dat"
        while( getline < late_file > 0 ) cnt[$1] = $2
        close( late_file )
    }

     {
        # 已更改欄位切割方式, $2表小時數,$3表分鐘數
        arrival = HM_to_M($2, $3)

        if( arrival > 480 ){
           mark = "*"   # 若當天遲到,應再增加其遲到次數, 且令 mark 為"*".
           cnt[$1]++    }
        else mark = " "

        # message 用以顯示該員工的遲到累計數, 若未曾遲到 message 為空字串
        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

        #將陣列cnt[ ]中新的遲到資料寫回檔案中
        for( any in cnt )
              print any, cnt[any] > late_file
    }
function HM_to_M( hour, min ){
        return hour*60 + min
    }
' $*
```

執行後, *today_rpt4* 之內容如下 :

```
          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
```

說 明：

1. *latefile* 是一變數, 用以記錄遲到次數的檔案之檔名. *latefile* 之值由兩部分構成, 前半部是當月月份名稱(由呼叫"date"取得) 後半部固定為"late.dat" 如 : *Jun-late.dat* .

2. 指令 getline < *latefile* 表由*latefile* 所代表的檔案中讀取一筆紀錄, 並存放於$0.

   若使用者可自行把資料放入$0, AWK會自動對這新置入 $0 的資料進行欄位分割. 之後程式中可用$1, $2,..來表示該筆資料的第一欄,第二欄,..,

   (註: 有少數AWK版本不容許使用者自行將資料置於 $0, 遇此情況可改用gawk或nawk)

   執行getline指令時, 若成功讀取紀錄,它會傳回1. 若遇到檔案結束, 它傳回0; 無法開啟檔案則傳回-1.

3. 利用 **while( getline <** *filename* **> 0 )** {....} 可讀入檔案中的每一筆資料並予處理. 這是AWK中user自行讀取檔案資料的一個重要模式.

4. 陣列 late_cnt[ ] 以員工ID. 當註標(index), 其對應值表其遲到的次數.

5. 執行結束後, 利用 **for(** *Variable* **in** *array* **)**{..}之語法 *for( any in late_cnt ) print any, late_cnt[any] > latefile*

   將更新過的遲到資料重新寫回記錄遲到次數之檔案. 該語法於第5節中 曾有說明.

# 8 Multi-line Record

AWK每次從資料檔中只讀取一筆Record, 進行處理. AWK係依照其內建變數 RS(Record Separator) 的定義將檔案中的資料分隔成一筆一筆的Record.
RS 的預設值是 "\n"(跳行符號), 故平常AWK中一行資料就是一筆Record.

但有些檔案中一筆Record涵蓋了數行資料, 這種情況下不能再以 "\n" 來分隔 Records. 最常使用的方法是相鄰的Records之間改以 一個空白行 來隔開.

在AWK程式中, 令 RS = ""(空字串)後, AWK把會空白行當成來檔案中Record的分隔符號. 顯然AWK對 RS = "" 另有解釋方式,簡略描述如下,
當 RS = "" 時 :

1. 數個併鄰的空白行, AWK僅視成一個單一的Record Saparator.(AWK不會於兩個緊併的空白行之間讀取一筆空的Record)
2. AWK會略過(skip)檔首或檔末的空白行. 故不會因為檔首或檔末的空白行, 造成AWK多讀入了二筆空的資料.

請觀察下例,首先建立一個資料檔 *week.rpt* 如下:

```
張長弓
GNUPLOT 入門

吳國強
Latex 簡介
VAST-2 使用手冊
mathematica 入門



李小華
AWK Tutorial Guide
Regular Expression
```

該檔案檔首有數列空白行, 各筆Record之間使用一個或數個空白行隔開. 讀者請細心觀察, 當 RS = "" 時, AWK讀取該資料檔之方式.

編輯一個AWK程式檔案 *make_report* 如下:

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

執行

$ **make_report** *week.rpt*

螢幕產生結果如下:

```
一. 報告人 : 張長弓
          1. GNUPLOT 入門

二. 報告人 : 吳國強
          1. Latex 簡介
          2. VAST-2 使用手冊
          3. mathematica 入門

三. 報告人 : 李小華
          1. AWK Tutorial Guide
          2. Regular Expression
```

說明:

1. 本程式同時也改變欄位分隔字元( FS= "\n"), 如此一筆資料中的每一行都是一個 field.
   例如 : AWK讀入的第一筆 Record 為
   張長弓
   GNUPLOT 入門

其中 $1 指的是"張長弓", $2 指的是"GNUPLOT 入門"

2. 上式中的C_Number[ ]是一個陣列(array), 用以記錄中文數字.

   例如 : C_Number[1] = "一", C_Number[2] = "二"

   這過程使用AWK字串函數 **split( )** 來把中文數字放進陣列Number[ ]中.

   函數 split( )用法如下 :

   **split( 原字串, 陣列名稱, 分隔字元(field separator) )** :

   AWK將依所指定的分隔字元(field separator)分隔 原字串 成一個個的欄位(field), 並以指定的 陣列 記錄各個被分隔的欄位

# 9 Getting Arguments on Command Line

大部分的應用程式都容許使用者於命令之後增加一些選擇性的參數. 執行AWK時這些參數大部分用於指定資料檔檔名, 有時希望在程式中能從命令列上得到一些其它用途的資料. 本小節中將敘述如何在AWK程式中取用這些參數.

建立檔案如下, 命名為 *see_arg* :

```
awk '
BEGIN {
        for( i=0; i<ARGC ; i++)
              print ARGV[i]      # 依次印出AWK所紀錄的參數
      }
' $*
```

執行如下命令 :

$ **see_arg**  *first-arg*  *second-arg*

結果螢幕出現 :

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

說明 :

1. ARGC, ARGV[ ] 為AWK所提供的內建變數.

   - ARGC : 為一整數. 代表命令列上, 除了選項-v, -f 及其對應的參數之外所有參數的數目.
   - ARGV[ ] : 為一字串陣列. ARGV[0],ARGV[1],...ARGV[ARGC-1]. 分別代表命令列上相對應的參數.

   例如, 當命令列為 :

   $ awk -v*x=36*  -f *program1*  *data1*  *data2*

   或

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

   其 ARGC 之值為 3
   ARGV[0] 之值為 "awk"
   ARGV[1] 之值為 "data1"
   ARGV[2] 之值為 "data2"

   命令列上的 "-f program1", "-vx=36", 或程式部分 '{ print $1, $2 }' 都不會列入 ARGC 及 ARGV[ ] 中.

2. AWK 利用 ARGC 來判斷應開啟的資料檔個數.

   但使用者可強行改變 ARGC; 當 ARGC 之值被使用者設為 1 時; AWK將被矇騙, 誤以為命令列上並無資料檔檔名, 故不會以 ARGV[1], ARGV[2],..為檔名來開檔讀取資料; 但於程式中仍可藉由 ARGV[1], ARGV[2],..來取得命令列上的資料.

   某一程式 *test1.awk* 如下 :

```
BEGIN{
        number = ARGC      #先用number 記住實際的參數個數.
        ARGC = 2   # 自行更改 ARGC=2, AWK將以為只有一個資料檔
                   # 仍可藉由ARGV[ ]取得命令列上的資料.
        for( i=2; i<number; i++)   data[i] = ARGV[i]
      }
        ........
```

   於命令列上鍵入

   $ awk -f test1.awk *data_file* *apple* *orange*

   執行時 AWK 會開啟資料檔 *data_file* 以進行處理. 不會開啟以 *apple, orange* 為檔名的檔案(因為 ARGC 被改成2). 但仍可藉由 ARGV[2], ARGV[3] 取得命令列上的參數 *apple, orange*

3. 可以下列命令來達成上例的效果.

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

# 10 Writing Interactive Program in AWK

執行AWK程式時, AWK會自動由檔案中讀取資料來進行處理, 直到檔案結束. 只要將AWK讀取資料的來源改成鍵盤輸入, 便可設計與AWK interactive talk 的程式. 本節將提供一個該類程式的範例.

範例 : 本節將撰寫一個英語生字測驗的程式, 它將印出中文字意,再由使用者回答其英語生字.

首先編輯一個資料擋 *test.dat* (內容不拘,格式如下)

```
apple    蘋果
orange   柳橙
banana   香蕉
pear     梨子
starfruit 楊桃
bellfruit 蓮霧
kiwi      奇異果
pineapple 鳳梨
watermelon 西瓜
```

編輯AWK程式"c2e"如下:

```
awk '
BEGIN {
        while( getline <ARGV[1] ){ #由指定的檔案中讀取測驗資料
              English[++n] = $1     # 最後, n 將表示題目之題數
              Chinese[n] = $2
        }
        ARGV[1] = "-"      # "-"表示由stdin(鍵盤輸入)
        srand()            # 以系統時間為亂數啟始的種子
        question( )        #產生考題

      }
```

```
        { # AWK自動讀入由鍵盤上輸入的資料(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( ) #產生考題
              }
        }

    function question(){
        ind = int(rand( )* n) + 1 #以亂數選取考題
        system("clear")
        print " Press \"ctrl-d \" to exit"
        printf("\n%s ", Chinese[ind] " 的英文生字是: ")
     }
' $*
```

執行時鍵入如下指令 :

$**c2e** *test.dat*

螢幕將產生如下的畫面:

**Press "ctrl-d " to exit**
**漣霧 的英文生字是:**

若輸入 **bellfruit**
程式將產生

**You are right !! Press Enter to Continue —**

說 明 :

1. 參數 *test.dat* (ARGV[1]) 表示儲存考題的資料檔檔名. AWK 由該檔案上取得考題資料後, 將 ARGV[1] 改成 "-".
   "-" 表示由 stdin(鍵盤輸入) 資料. 鍵盤輸入資料的結束符號 (End of file) 是 Ctrl-d. 當 AWK 讀到 Ctrl-d 時就停止由 stdin 讀取資料.

2. AWK的數學函數中提供兩個與亂數有關的函數.

rand( ) : 傳回介於 0與1之間的(近似)亂數值. 0 <rand() <1.
除非使用者自行指定 rand() 函數起始 的seed, 否則每次執行AWK程式時, rand() 函數都將以同一個內定的 seed 為啟始, 來產生亂數.

srand(x) : 指定以x為 rand( ) 函數起始的 seed. 若省略了 x, 則AWK會以執行 時的日期與時間為 rand() 函數起始的 seed. (參考 附錄 C AWK 的 Built-in Functions )

# 11 Recursive Program

AWK 中除了函數的參數列(Argument List)上的參數(Arguments)外, 所有變數不管於何處出現全被視為 Global variable. 其生命持續至程式結束 — 該變數不論在function外或 function內皆可使用, 只要變數名稱相同所使用的就是同一個變數, 直到程式結束. 因 Recusive 函數內部的變數, 會因它呼叫子函數(本身)而重覆使用, 故撰寫該類函數時, 應特別留心.

例如 : 執行

```
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 為參數列上的參數, 是local variable. 離開此函數後將消失.
    y ++          # 會改變主式中的變數 y
    z = 55        # z 為該函數中新使用的變數, 主程式中變數 z 仍可被使用.
    printf("Inside the function: arg1= %d,x= %d, y= %d, z= %d\n",
        arg1, x, y, z)
  } '
```

結果螢幕印出

```
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 :

- 函數內可任意使用主程式中的任何變數.
- 函數內所啟用的任何變數(除參數外), 於該函數之外依然可以使用.

此特性優劣參半, 最大的壞處是式中的變數不易被保護, 特別是recursive呼叫本身, 執行子函數時會破壞父函數內的變數.

權變的方法是 ：在函數的 Argument list 上虛列一些 Arguments.

函數執行中使用這些虛列的 Arguments 來記錄不想被破壞的資料, 如此執行子函數時就不會破壞到這些資料. 此外AWK 並不會檢查, 呼叫函數時所傳遞的參數個數是否一致.

例如 ：定義 recursive function 如下 ：

```
function demo( arg1 ){     # 最常見的錯誤例子
        .....
        for(i=1; i<20 ; i++){
            demo(x)
# 又呼叫本身. 因為 i 是 global variable, 故執行完該子函數後
# 原函數中的 i 已經被壞, 故本函數無法正確執行.

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

可將上列函數中的 i 虛列在該函數的參數列上, 如此 i 便是一個local variable , 不會因執行子函數而被破壞.

將上列函數修改如下:

```
function demo( arg1,   i ){
      ......
      for(i=1; i<20; i++){
       demo(x)   #AWK不會檢查呼叫函數時, 所傳遞的參數個數是否一致
            .....
      }
}
```

$0, $1,.., NF, NR,..也都是 global variable, 讀者於 recusive function中若有使用這些內建變數, 也應另外設立一些 local variable 來保存, 以免被破壞.

範例 ：以下是一個常見的 Recursive 範例. 它要求使用者輸入一串元素(各元素間用空白隔開) 然後印出這些元素所有可能的排列.

編輯如下的AWK式, 取名為 *permu*

```
awk '
BEGIN{
        print "請輸入排列的元素,各元素間請用空白隔開"
        getline
        permutation( $0, "")
        printf("\n共 %d 種排列方式\n", counter)
      }

function permutation( main_lst, buffer,      new_main_lst, nf, i, j ) {
      $0 = main_lst   # 把main_lst指定給$0之後AWK將自動進行欄位分割.
      nf = NF         # 故可用 NF 表示 main_lst 上存在的元素個數.

    #   BASE CASE : 當main_lst只有一個元素時.
      if( nf == 1) {
            print buffer main_lst   # buffer的內容連接(concate)上 main_lst 就
            counter++               # 是完成一次排列的結果
           return
            }

    #  General Case : 每次從 main_lst 中取出一個元素放到buffer中
    #  再用 main_lst 中剩下的元素 (new_main_lst) 往下進行排列
      else for( i=1; i<=nf ;i++){
            $0 = main_lst  # $0($1,$2,..$j,,)為Global variable已被壞, 故重新
                           # 把 main_lst 指定給$0, 令AWK再做一次欄位分割
            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 )
            }
    }
' $*
```

執行　$ permu

螢幕上出現

請輸入排列的元素,各元素間請用空白隔開

若輸入 1 2 3 結果印出

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

共 6 種排列方式

說 明：

1. 有些較舊版的AWK,並不容許使用者指定$0之值. 此時可改用 gawk, 或 nawk. 否則也可自行使用 split() 函數來分割 main_lst.

2. 為避免執行子函數時破壞 new_main_lst, nf, i, j 故把這些變數也列於參數列上. 如此, new_main_lst, nf, i, j 將被當成 local variable,而不會受到子函數中同名的變數影響. 讀者宣告函數時,參數列上不妨將這些 "虛列的參數" 與真正用於傳遞資訊的參數間以較長的空白隔開, 以便於區別.

3. AWK 中欲將字串concatenation(連接)時, 直接將兩字串併置即可(Implicit Operator).

   例如：

   *awk '*

   *BEGIN{*

   *A = "This "*

   *B = "is a "*

   *C = A B "key." # 變數A與B之間應留空白, 否則"AB"將代表另一新變數.*

   *print C*

   *} '*

   結果將印出

   This is a key.

4. AWK使用者所撰寫的函數可再reuse, 並不需要每個AWK式中都重新撰寫.
   將函數部分單讀編寫於一檔案中, 當需要用到該函數時再以下列方式include進來.

   $ awk　-f 函數檔名　-f AWK主程式檔名　資料檔檔名

# Appendix A Patterns

AWK 藉由判斷 Pattern 之值來決定是否執行其後所對應的 Actions.

這裡列出幾種常見的 Pattern :

1. **BEGIN**

   BEGIN 為 AWK 的保留字, 是一種特殊的 Pattern.

   BEGIN 成立(其值為true)的時機是 :

   > "AWK 程式一開始執行, 尚未讀取任何資料之前."

   所以在 BEGIN { Actions } 語法中, 其 Actions 部份僅於程式一開始執行時被執行一次. 當 AWK 從資料檔讀入資料列後, BEGIN 便不再成立, 故不論有多少資料列, 該 Actions 部份僅被執行一次.

   一般常把 "與資料檔內容無關" 與 "只需執行一次" 的部分置於該 Actions(以 BEGIN 為 Pattern)中.
   例如 :

```
BEGIN {
        FS = "[ \t:]"    # 於程式一開始時 , 改變AWK切割欄位的方式
        RS = ""          # 於程式一開始時 , 改變AWK分隔資料列的方式
        count = 100      # 設定變數 count 的起始值
        print " This is a title line "  \# 印出一行 title
      }
       ....... # 其它 Pattern { Actions } .....
```

   有些AWK程式甚至"不需要讀入任何資料列". 遇到這情況可把整個程式置於以 BEGIN 為 Pattern 的 Actions 中.

   例如 :

   BEGIN { print " Hello ! the Word ! " }

   注意 : 執行該類僅含 BEGIN { Actions } 的程式時, AWK 並不會開啟任何資料檔進行處理.

## 2. END

END 為 AWK 的保留字, 是另一種特殊的 Pattern.

END 成立(其值為true)的時機與 BEGIN 恰好相反, 為 :

"AWK 處理完所有資料, 即將離開程式時"

平常讀入資料列時, END並不成立, 故其對應的 Actions 並不被執行; 唯有當 AWK讀完所有資料時, 該 Actions 才會被執行

注意 :不管資料列有多少筆, 該 Actions 僅被執行一次.

## 3. Relational Expression

使用像 "**A Relation Operator B**" 的 Expression 當成 Pattern.

當 A 與 B 存在所指定的關係(Relation)時, 該 Pattern 就算成立(true).

例如 :

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

上式中 *l* ength($0) <= 80 是一個 Pattern, 當 $0(資料列)之長度小於等於 80 時該 Pattern 之值為true, 將執行其後的 Action(印出該資料列).

AWK 中提供下列 關係運算元(Relation Operator)

| 運算元 | 涵意 |
|---|---|
| > | 大於 |
| < | 小於 |
| >= | 大於或等於 |
| <= | 小於或等於 |
| == | 等於 |
| != | 不等於 |
| ~ | **match** |
| !~ | **not match** |

上列關係運算元除 ~(match)與 !~(not match)外與 C 語言中之涵意一致.

~(match) 與 !~(match) 在 AWK 之涵意簡述如下 :

若**A** 表一字串, **B** 表一 **Regular Expression**.

A ~ B 判斷 字串A 中是否 包含 能合於(match)B式樣的子字串.
A !~ B 判斷 字串A 中是否 未包含 能合於(match)B式樣的子字串.

例如：

*$0 ~ /program[0-9]+\.c/ { print }*

**$0 ~ /program[0-9]+\.c/** 整個是一個 Pattern, 用來判斷 $0(資料列)中是否含有可 match /program[0-9]+\.c/ 的子字串, 若 $0 中含有該類字串, 則執行 print (印出該列資料).

Pattern 中被用來比對的字串為 $0 時(如本例), 可僅以 Regular Expression 部分表之. 故本例的 Pattern 部分

$0 ~ /program[0-9]+\.c/ 可僅用 **/program[0-9]+\.c/**表之

(有關 match 及 Regular Expression 請參考 附錄 E )

## 4. Regular Expression

直接使用 Regular Expression 當成 Pattern; 此為 $0 ~ Regular Expression 的簡寫.

該 Pattern 用以判斷 $0(資料列) 中是否含有 match 該 Regular Expression 的子字串; 若含有該成立(true) 則執行其對應的 Actions.
例如：

**/^[0-9]*$/** { *print "This line is a integer !"* }

與 **$0 ~ /^[0-9]*$/** { *print "This line is a integer !"* } 相同

## 5. Compound Pattern

之前所介紹的各種 Patterns, 其計算(evaluation)後結果為一邏輯值(True or False). AWK 中邏輯值彼此間可藉由 &&(and), ||(or), !(not)結合成一個新的邏輯值. 故不同 Patterns 彼此可藉由上述結合符號來結合成一個新的 Pattern. 如此可進行複雜的條件判斷.

例 如：

*FNR >= 23 && FNR <=28 { print " " $0 }*

上式利用 && (and) 將兩個 Pattern 求值的結果合併成一個邏輯值.
該式 將資料檔中 第23行 到 28行 向右移5格(先印出5個空白字元)後印出.
( FNR 為AWK的內建變數, 請參考 附錄 D )

## 6. Pattern1 , Pattern2

遇到這種 Pattern, AWK 會幫您設立一個 switch(或flag).

當AWK讀入的資料列使得 Pattern1 成立時, AWK 會打開(turn on)這 switch.
當AWK讀入的資料列使得 Pattern2 成立時, AWK 會關上(turn off)這個 switch.

該 Pattern 成立的條件是 :

當這個 switch 被打開(turn on)時 (包括 Pattern1, 或 Pattern2 成立的情況)

例 如 :

*FNR >= 23 && FNR <=28* { *print " " $0* }

可改寫為

**FNR == 23 , FNR == 28** { *print " " $0* }

說 明 :

1. 當 FNR >= 23 時, AWK 就 turn on 這個 switch;
2. 因為隨著資料列的讀入, AWK不停的累加 FNR.
3. 當 FNR = 28 時, Pattern2 **FNR == 28** 便成立, 這時 AWK會關上這個 switch.

當 switch 打開的期間, AWK 會執行 "print " " $0"

( FNR 為AWK的內建變數, 請參考 附錄 D )

# Appendix B Actions

Actions 是由下列指令(statement)所組成 :

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 中大部分指令與 C 語言中的用法一致, 此處僅介紹較為常用或容易混淆之指令的用法.

1. 流程控制指令

- if 指令

  語法

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

  範例 :

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

  (a) 與 C 語言中相同, 若 expression 計算(evaluate)後之值不為 0 或空字串, 則執行 statement1; 否則執行 statement2.

  (b) 進行邏輯判斷的expression所傳回的值有兩種, 若最後的邏輯值為true, 則傳回1, 否則傳回0.

  (c) 語法中**else** statement2 以[ ] 前後括住表示該部分可視需要而予加入或省略.

- while 指令

  語法：

  **while( expression )** statement

  範例：

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

  上列範例找出 *buffer* 中所有能合於(match) /[0-9]+ċ/( 數字之後接上".c"的所有子字串).
  範例中 while 以函數 match( )所傳回的值做為判斷條件. 若 *buffer* 中還含有合於指定條件的子字串(match成功), 則 match()函數傳回1, while 將持續進行其後之statement.

- do-while 指令

  語法：

  **do** statement **while(** expression **)**

  範例：

```
do{
        print "Enter y or n ! "
        getline data < "-"

  } while( data !~ /^[YyNn]$/)
```

  (a) 上例要求使用者從鍵盤上輸入一個字元, 若該字元不是 Y, y, N, 或 n 則會不停執行該迴圈, 直到讀取正確字元為止.

  (b) do-while 指令與 while 指令 最大的差異是：do-while 指令會先執行 statement 而後再判斷是否應繼續執行. 所以, 無論如何其 statement 部分至少會執行一次.

- for Statement 指令(一)

  語法：

  **for(** variable **in** array **)** statement

範例 : 執行下列命令

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

結果印出 :

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

(a) 這個 for 指令, 專用以搜尋陣中所有的index值, 並逐次使用所指定的變數 予以紀錄. 以本例而言, 變數 any 將逐次代表 2, 1,及"last".

(b) 以這個 for 指令, 所搜尋出的index之值彼此間並無任何次續關係.

(c) 第7節 Arrays in AWK 中有該指令的使用範例, 及解說.

- for Statement 指令(二)

語法 :

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

範例 :

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

說明 :

(a) 上列範例用以計算 1 加到 10 的總合.

(b) expression1 常用於設定該 for 迴圈的起始條件, 如上例中的 i=1
expression2 用於設定該迴圈的停止條件, 如上例中的 i<= 10
expression3 常用於改變 counter 之值, 如上例中的 i++

- break 指令

break 指令用以強迫中斷(跳離) for, while, do-while 等迴圈.

範例 :

```
while(  getline < "datafile" > 0 ){
        if( $1 == 0 )       # 所讀取的資料置於 $0
             break          # AWK立刻把 $0 上新的欄位資料
        else                # 指定給 $1, $2, ...$NF
        print $2 / $1
     }
```

上例中, AWK 不斷地從檔案 *datafile* 中讀取資料, 當 $1 等於 0 時, 就停止該執行迴圈.

- continue 指令

  迴圈中的 statement 進行到一半時, 執行 continue 指令來掠過回圈中尚未執行的 statement. 範例 :

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

  (a) 上例中若 index 不為數字則執行 continue, 故將掠過(不執行)其後的指令.

  (b) 需留心 continue 與 break 的差異 : 執行 continue 只是掠過其後未執行的statement, 但並未跳離開該迴圈.

- next 指令

  執行 next 指令時, AWK 將掠過位於該指令(next)之後的所有指令(包括其後的所有 Pattern { Actions }), 接著讀取下一筆資料列, 繼續從第一個 Pattern { Actions } 執行起.

  範例 :

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

  上例中, 當 AWK 讀入的資料列為空白行時( match /^[ \]*$/ ) 除列印訊息外且執行 next, 故 AWK 將掠過其後的指令, 繼續讀取下一筆資料, 從頭(第一個 Pattern { Actions })執行起.

- exit 指令

  執行 exit 指令時, AWK將立刻跳離(停止執行)該AWK程式.

2. AWK 中的 I/O 指令

- printf 指令

  該指令與 C 語言中的用法相同, 可藉由該指令控制資料輸出時的格式.

  語法 :

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

  範 例 :

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

(a) 結果印出 :
ID# : BE-647  Ave Score : 89

(b) format 部分係由 一般的字串(String Constant) 及 格式控制字元 (Format control letter, 其前會加上一個%字元)所構成. 以上式為例 "ID# : " 及 " Ave Score : " 為一般字串. %s 及 %d 為格式控制字元.

(c) 印出時, 一般字串將被原封不動地印出. 遇到格式控制字元時, 則依序把 format 後方之 item 轉換成所指定的格式後印出.

(d) 有關的細節, 讀者可從介紹 C 語言的書籍上得到較完整的介紹.

(e) print 及 printf 兩個指令, 其後可使用 > 或 >> 將輸出到stdout的資料 Redirct 到其它檔案, 7.1 Redirect Output to Files 中有完整的範例說明.

- print 指令

  範 例 :

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

(a) 結果印出 : ID# : BE-267 Ave Score :89

(b) print 之後可接上字串常數(Constant String)或變數. 它們彼此間可用 "," 隔開.

(c) 上式中, 字串 "ID# :" 與變數 id 之間使用","隔開, 印出時兩者之間會以自動 OFS(請參考 D 內建變數 OFS) 隔開. OFS 之值一般內定為 "一個空白字元"

(d) 上式中字串 "Ave Score :" 與變數ave之間並未以","隔開, AWK會將這兩者先當成字串concate在一起(變成"Ave Score :89")後, 再予印出

(e) print 及 printf 兩個指令, 其後可使用 > 或 >> 將輸出到stdout的資料 Redirct 到其它檔案, 7.1 Redirect Output to Files 中有完整的範例說明.

- getline 指令

| 語法 | 由何處讀取資料 | 資料讀入後置於 |
|---|---|---|
| getline var< file | 所指定的 file | 變數 var(var省略時,表示置於$0) |
| \| getline var | pipe | 變數 var(var省略時,表示置於$0) |
| getline var | 見 註一 | 變數 var(var省略時,表示置於$0) |

註一：當 Pattern 為 BEGIN 或 END 時, getline 將由 stdin 讀取資料, 否則由AWK正處理的資料檔上讀取資料.

getline 一次讀取一行資料,
若讀取成功則return 1,
若讀取失敗則return -1,
若遇到檔案結束(EOF), 則return 0

- close 指令

該指令用以關閉一個開啟的檔案, 或 pipe(見下例) 範 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."
      }
```

說 明：

(a) 上例中, 一開始執行 print "ID # Salary" > "data.rpt" 指令來印出一行抬頭. 它使用 I/O Redirection( > )將資料轉輸出到 *data.rpt* , 故此時檔案 *data.rpt* 是處於 Open 狀態.

(b) 指令 print $1, $2 * $3 不停的將印出的資料送往 pipe(|), AWK 於程式將結束時才會呼叫 shell 使用指令 "sort +0n > data.rpt" 來處理 pipe 中的資料; 並未立即執行, 這點與 Unix 中pipe的用法不盡相同.

(c) 最後希望於檔案 *data.rpt* 之"末尾"處加上一行 "There are.....". 但此時, Shell 尚未執行 "sort +0n > data.rpt" 故各資料列排序後的 ID 及 Salary 等資料尚未寫入 *data.rpt* . 所以得命令 AWK 提前先通知 Shell 執行命令 "sort +0n > data.rpt" 來處理 pipe 中的資料. AWK中這個動作稱為 close pipe. 係由執行 close( "shell command" )來完成. 需留心 close( )指令中的 shell command 需與 "|"後方的 shell command 完全相同(一字不差), 較佳的方法是先以該字串定義一個簡短的變數, 程式中再以此變數代替該 shell command

(d) 為什麼要執行 close("data.rpt") ? 因為 sort 完後的資料也將寫到 *data.rpt* , 而該檔案正為AWK所開啟使用(write)中, 故AWK程式中

應先關閉 *data.rpt* . 以免造成因二個 processes 同時開啟一個檔案進行輸出(write)所產生的錯誤.

- system 指令

  該指令用以執行 Shell上 的 command. 範 例 :

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

  說明 :

  (a) system("字串")指令接受一個字串當成Shell的命令. 上例中, 使用一個字串常數 "rm " 連接(concate)一個變數 *DataFile* 形成要求 Shell執行的命令.
  Shell 實際執行的命令為 "rm invent.rpt".

- "|" pipe指令

  "|" 配合 AWK 輸出指令, 可把 output 到 stdout 的資料繼續轉送給 Shell 上的令一命令當成input的資料.
  "|" 配合 AWK getline 指令, 可呼叫 Shell 執行某一命令, 再以 AWK 的 getline 指令將該命令的所產生的資料讀進 AWK 程式中.
  範 例 :

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

   "date" |  getline  Date_data
  ```

  讀者請參考 7.2 Using System Resources 其中有完整的範例說明.

3. AWK 釋放所佔用的記憶體的指令

   AWK 程式中常使用陣列(Array)來記憶大量資料. delete 指令便是用來釋放陣列中的元素所所佔用的記憶空間.
   範 例 :

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

   讀者請留心, delete 指令一次只能釋放陣列中的一個"元素".

4. AWK 中的數學運算元(Arithmetic Operators)

   +(加), -(減), *(乘), /(除), %(求餘數), ^(指數) 與 C 語言中用法相同

5. AWK 中的 Assignment Operators

   =, +=, -=, *= , /=, %=, ^=
   x += 5 的意思為 x = x + 5, 其餘類推.

6. AWK 中的 Conditonal Operator
   語 法 :

   判斷條件 ? value1 : value2

   若 判斷條件 成立(true) 則傳回 value1, 否則傳回 value2.

7. AWK 中的邏輯運算元(Logical Operators)

   &&( and ), ||( or ), !(not) Extended Regular Expression 中使用 "|" 表示 or 請勿混淆.

8. AWK 中的關係運算元(Relational Operators)

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

9. AWK 中其它的運算元

   +(正號), -(負號), ++(Increment Operator), - -(Decrement Operator)

10. AWK 中各運算元的運算優先順序( Precedence )
    (按優先高低排列)

```
$        (欄位運算元, 例如 : i=3; $i表示第3欄)
^        (指數運算)
+ ,- ,!  (正,負號,及邏輯上的 not)
* ,/ ,%  (乘,除,餘數)
+ ,-     (加,減)
>, >=,<  , <=, ==, != (大於,大於等於,...,等關係符號)
~, !~    (match, not match)
&&       (邏輯上的 and)
||       (邏輯上的 or )
? :      (Conditional Operator)
= , +=, -=,*=, /=, %=, ^= (一些指定Assignment運算元)
```

# Appendix C Built-in Functions

(一). 字串函數

- **index(** 原字串, 找尋的子字串 **)** :

  若原字串中含有欲找尋的子字串,則傳回該子字串在原字串中第一次出現的位置,若未曾出現該子字串則傳回0.

  例如執行 :

  $ awk *'BEGIN { print index("8-12-94","-" }'*

  結果印出 2

- **length(** 字串 **)** : 傳回該字串的長度.

  例如執行 :

  $ awk *'BEGIN { print length("John") }'*

  結果印出 4

- **match(** 原字串, 用以找尋比對的**Regular Expression** **)** :

  AWK會在原字串中找尋合乎Regular Expression的子字串. 若合乎條件的子字串有多個, 則以原字串中最左方的子字串為準.
  AWK找到該字串後會依此字串為依據進行下列動作:

  1. 設定AWK內建變數 RSTART, RLENGTH :

     RSTART = 合條件之子字串在原字串中之位置.
     = 0 ; 若未找到合條件的子字串.

     RLENGTH = 合條件之子字串長度.
     = -1 ; 若未找到合條件的子字串.

  2. 傳回 RSTART 之值.

     例如執行 :

```
$ awk ' BEGIN {
                match( "banana", /(an)+/ )
                print RSTART, RLENGTH
              }
      '
```

     執行結果印出 2 4

- **split(** 原字串，陣列名稱，分隔字元(field separator) **)**:

  AWK將依所指定的分隔字元(field separator)來分隔原字串成一個個的欄位(field),並以指定的陣列記錄各個被分隔的欄位.

  例如 :

  *ArgLst = "5P12p89"*
  *split( ArgLst, Arr, /[Pp]/)*

  執行後 Arr[1]=5, Arr[2]=12, Arr[3]=89

- **sprintf(** 列印之格式，列印之資料，列印之資料,...**)**

  該函數之用法與AWK或C的輸出函數printf()相同. 所不同的是sprintf()會將要求印出的結果當成一個字串傳回
  一般最常使用sprintf()來改變資料格式. 如: x 為一數值資料, 若欲將其變成一個含二位小數的資料,可執行如下指令 :

  *x = 28*
  *x = sprintf("%.2f",x)*

  執行後 x = "28.00"

- **sub(** 比對用的Regular Expression，將替換的新字串，原字串 **)**

  sub( )將原字串中第一個(最左邊)合乎所指定的Regular Expression 的子字串改以新字串取代.

  1. 第二個參數"將替換的新字串"中可用"&"來代表"合於條件的子字串"
     承上例,執行下列指令:

     *A = "ab12anan212.45an6a"*
     *sub( /(an)+[0-9]*/, "[&]", A)*

     結果印出 ab12[anan212].45an6a

  2. sub()不僅可執行取代(replacement)的功用,當第二個參數為空字串("")時, sub()所執行的是"去除指定字串"的功用.

  3. 藉由 sub()與 match()的搭配使用,可逐次取出原字串中合乎指定條件的所有子字串.

例如執行下列程式:

```
awk '
BEGIN {
        data = "p12-P34 P56-p61"
        while( match( data ,/[0-9]+/) >0) {
              print substr(data,RSTART, RLENGTH )
              sub(/[0-9]+/,"")
              }
              }
    ' $*
```

結果印出 :

12
34
56
61

4. sub( )中第三個參數(原字串)若未指定,則其預設值為$0.

   可用 *sub( /[9-0]+/,"digital" )* 表示 *sub(/[0-9]+/,"digital",$0 )*

- **gsub(** 比對用的Regular Expression, 將替換的新字串, 原字串 **)**

這個函數與 sub()一樣,同樣是進行字串取代的函數. 唯一不同點是

1. gsub()會取代所有合條件的子字串.
2. gsub()會傳回被取代的子字串個數.

請參考 sub().

- **substr(** 字串,起始位置 [,長度] **)** :

傳回從起始位置起,指定長度之子字串. 若未指定長度,則傳回起始位置到自串末尾之子字串.

執行下例

```
$ awk   ' BEGIN{ print substr( "User:Wei-Lin Liu", 6)}'
```

結果印出

Wei-Lin Liu

(二). 數學函數

- **int(x)** : 傳回x的整數部分(去掉小數).

  例如 :
  int(7.8) 將傳回 7
  int(-7.8) 將傳回 -7

- **sqrt(x)** : 傳回x的平方根.

  例如 :
  sqart(9) 將傳回 3
  若 x 為負數,則執行 sqrt(x)時將造成 Run Time Error

- **exp(x)** : 將傳回e的x次方.

  例如 :
  exp(1) 將傳回 2.71828

- **log(x)** : 將傳回x以e為底的對數值.

  例如 :
  log(e) = 1
  若 x<0 ,則執行 sqrt(x)時將造成 Run Time Error.

- **sin(x)** : x 須以徑度量為單位,sin(x)將傳回x的sin函數值.

- **cos(x)** : x 須以徑度量為單位,cos(x)將傳回x的cos函數值

- **atan2(y,x)** : 傳回 y/x 的tan反函數之值,傳回值係以徑度量為單位.

- **rand( )** : 傳回介於 0與1之間的(近似)亂數值; 0 <rand() <1.

  除非使用者自行指定rand()函數起始的seed,否則每次執行AWK程式時,
  rand()函數都將使用同一個內定的seed,來產生亂數.

- **srand(x)** : 指定以x為rand( )函數起始的seed.

  若省略了x,則AWK會以執行時的日期與時間為rand()函數起始的seed.

# Appendix D Built-in Variables

因內建變數的個數不多, 此處按其相關性分類說明, 並未按其字母順序排列.

**ARGC** 表命令列上除了選項 **-F, -v, -f** 及其所對應的參數之外的所有參數的個數.

若將"AWK程式"直接寫於命令列上, 則 ARGC 亦不將該"程式部分"列入計算.

**ARGV[ ]** 一個陣列用以記錄命令列上的參數.

例：執行下列命令

$ 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*

執行上列任一程式後

ARGC = 3
ARGV[0] = "awk"
ARGV[1] = "file1.dat"
ARGV[2] = "file2.dat"

讀者請留心：當 ARGC = 3 時, 命令列上僅指定 2 個資料檔.

註：

-F\t 表示以 tab 為欄位分隔字元 FS(field seporator).

-v a=8 是用以 initialize 程式中的變數 a.

**FILENAME** 用以表示目前正在處理的資料檔檔名.

**FS** 欄位分隔字元.

**$0** 表示目前AWK所讀入的資料列.

**$1,$2..** 分別表示所讀入的資料列之第一欄, 第二欄,..(參考下列說明)

當AWK讀入一筆資料列 **"A123 8:15"** 時, 會先以 $0 記載.

故 $0 = "A123 8:15"

若程式中進一步使用了 $1, $2.. 或 NF 等內建變數時, AWK才會自動分割 $0. 以便取得欄位相關的資料. 切割後各個欄位的資料會分別以 $1, $2, $3...予以記錄.

AWK內定(default)的 欄位分隔字元(**FS**) 為 空白字元(及tab).

以本例而言, 讀者若未改變 FS, 則分割後 :

第一欄($1)="A123", 第二欄($2)="8:15".

使用者可用 Regexp 自行定義 FS. AWK每次需要分割資料列時, 會參考目前FS之值.

例如 :

令 FS = "[ :]+" 表示任何由 空白" " 或 ":" 所組成的字串都可當成分隔

字元, 則分割後 :

第一欄($1) = "A123", 第二欄($2) = "8", 第三欄($3) = "15"

**NR** 表從 AWK 開始執行該程式後所讀取的資料列數.

**FNR** 與 NR 功用類似. 不同的是 AWK每開啟一個新的資料檔, FNR 便從 0 重新累計

**NF** 表目前的資料列所被切分的欄位數.

AWK 每讀入一筆資料後, 於程式中可以 NF 來得知該筆資料包含的欄位個數.

在下一筆資料被讀入之前, NF 並不會改變. 但使用者若自行使用$0來記錄資料

例如 : 使用 getline, 此時 NF 將代表新的 $0 上所記載之資料的欄位個數.

**OFS** 輸出時的欄位分隔字元. 預設值 " "(一個空白), 詳見下面說明.

**ORS** 輸出時資料列的分隔字元. 預設值 "\n"(跳行), 見下面說明.

**OFMT** 數值資料的輸出格式. 預設值 "%.6g"(若須要時最多印出6位小數)

當使用 **print** 指令一次印出多項資料時,
例如 : print $1, $2

印出資料時, AWK會自動在 $1 與 $2 之間補上一個 OFS 之值(預設值為 一個空白)
每次使用 print 輸出(印)資料後, AWK會自動補上 ORS 之值.(預設值為 跳行)
使用 print 輸出(印)數值資料時, AWK將採用 OFMT 之值為輸出格式.

例如 : print 2/3

AWK 將會印出 0.666667

程式中可藉由改變這些變數之值, 來改變指令 print 的輸出格式.

**RS** ( Record Separator) : AWK從資料檔上讀取資料時, 將依 RS 之定義把資料切割成許多Records,而AWK一次僅讀入一個Record,以進行處理.

RS 的預設值是 "\n". 所以一般 AWK一次僅讀入一行資料.

有時一個Record含括了幾列資料(Multi-line Record). 這情況下不能再以 "\n" 來分隔併鄰的Records, 可改用 空白行 來分隔.

在AWK程式中,令 RS = "" 表示以 空白行 來分隔併鄰的Records.

**RSTART** 與使用字串函數 match( )有關之變數,詳見下面說明.

**RLENGTH** 與使用字串函數 match( )有關之變數.

當使用者使用 match(...) 函數後, AWK會將 match(...) 執行的結果以 RSTART,RLENGTH 記錄之.
請參考 附錄 C AWK的內建函數 match().

**SUBSEP** (Subscript Separator) 陣列中註標的分隔字元, 預設值為"\034"

實際上, AWK中的 陣列 只接受 字串 當它的註標, 如 : Arr["John"].
但使用者在 AWK 中仍可使用 數字 當陣列的註標, 甚至可使用多維的陣列 (Multi-dimenisional Array)

如 : Arr[2,79]

事實上, AWK在接受 Arr[2,79] 之前, 就已先把其註標轉換成字串 "2\03479", 之後便以 Arr["2\03479"] 代替 Arr[2,79].

可參考下例 :

```
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]
               }
      ' $*
```

執行結果印出:

```
78
78
78
78
```

# Appendix E Regular Expression

- 為什麼要使用 Regular Expression

  UNIX 中提供了許多 指令 或 tools, 它們具有在檔案中 尋找(Search)字串 或 置換(Replace)字串 的功能. 像 **grep, vi , sed, awk**,...

  不論是找尋字串或置換字串, 都得先 "告訴這些指令所要找尋(被置換)的字串為何".

  若未能預先明確知道所要找尋(被置換)的字串為何, 只知該字串存在的範圍或特徵時, 例如 :

  (一) 找尋 "T0.c", "T1.c", "T2.c".... "T9.c" 當中的任一字串.

  (二) 找尋至少存在一個 "A"的任意字串.

  這情況下, 如何告知執行找尋字串的指令所要找尋的字串為何.

  例 (一) 中, 要找尋任一在 "T" 與 ".c" 之間存在一個阿拉伯數字的字串; 當然您可以列舉的方式, 一一把所要找尋的字串告訴執行命令的指令.

  但例 (二) 中合於該條件的字串有無限種可能, 勢必無法一一列舉. 此時,便需要另一種字串表示的方法(協定).

- 什麼是 Regular Expression

  Regular Expression(以下簡稱 **Regexp**) 是一種字串表達的方式. 可用以指稱具有某特徵的所有字串.

  註 : 為區別於一般字串, 本附錄中代表 Regexp 的字串之前皆加 "Regexp".
  註 : AWK 程式中常以/..../括住 Regexp; 以區別於一般字串.

- 組成 Regular Expression 的元素

普通字元 除了 . * [ ] + ? ( ) \ ^ $ 外之所有字元.

由普通字元所組成的Regexp其意義與原字串字面意義相同.

例如 : Regexp "**the**" 與一般字串的 "the" 代表相同的意義.

- Metacharacter：用以代表任意一字元.

  須留心 UNIX Shell 中使用 “*”表示 Wildcard, 可用以代表任意長度的字串. 而 Regexp 中使用 “.” 來代表一個任意字元.(注意: 並非任意長度的字串)

  Regexp 中 “*” 另有其它涵意, 並不代表任意長度的字串.

^ 表示該字串必須出現於行首.

$ 表示該字串必須出現於行末.

例如 :

Regexp /^**The**/ 用以表示所有出現於行首的字串 "The".
Regexp /**The$**/ 用以表示所有出現於行末字串 "The".

\ 將特殊字元還原成字面意義的字元(Escape character)

Regexp 中特殊字元將被解釋成特定的意義. 若要表示特殊字元的字面(literal meaning)意義時,在特殊字元之前加上"\"即可.

例如 :

使用Regexp來表示字串 “a.out”時, 不可寫成 /**a.out**/.

因為 “.”是特殊字元, 表任一字元. 可合乎 Regexp /**a.out**/ 的字串將不只 “a.out” 一個; 字串 “a2out”, “a3out”, “aaout” ...都合於 Regexp /**a.out**/. 正確的用法為 : /**a\.out**/

[...] 字元集合, 用以表示兩中括號間所有的字元當中的任一個.

例如 : Regexp /[**Tt**]/ 可用以表示字元 “T” 或 “t”.
故 Regexp /[**Tt**]**he**/ 表示 字串 “The” 或 “the”.

字元集合 [... ] 內不可隨意留空白.

例如 : Regexp /[ **Tt** ]/ 其中括號內有空白字元, 除表示 “T”, “t” 中任一個字元, 也可代表一個 “ ”(空白字元)

- 字元集合中可使用 “-” 來指定字元的區間, 其用法如下 :

Regexp /[**0-9**]/ 等於 /[**0123456789**]/ 用以表示任意一個阿拉伯數字.

同理 Regexp /[**A-Z**]/ 用以表示任意一個大寫英文字母.

但應留心 :

1. Regexp /[0-9a-z]/ 並不等於 /[0-9][a-z]/; 前者表示一個字元, 後者表示二個字元.
2. Regexp /[-9]/ 或 /[9-]/ 只代表字元 “9”或 “-”.

**[^...]** 使用[^..] 產生字元集合的補集(complement set).

其用法如下 :

例如 : 要指定 “T” 或 “t” 之外的任一個字元, 可用 /[^**Tt**]/ 表之.
同理 Regexp /[^**a-zA-Z**]/ 表示英文字母之外的任一個字元.

須留心 “^” 之位置 : “^”必須緊接於“[”之後, 才代表字元集合的補集

例如 : Regexp /[0-9^]/ 只是用以表示一個阿拉伯數字或字元”^”.

***** 形容字元重複次數的特殊字元.

“*” 形容它前方之字元可出現 1 次或多次, 或不出現. 例如 : Regexp /**T[0-9]*\.c**/ 中 * 形容其前 **[0-9]** (一個阿拉伯數字)出現的次數可為0次或 多次.
故Regexp /**T[0-9]*\.c**/ 可用以表示 “T.c”, “T0.c”, “T1.c”...“T9.c”

**+** 形容其前的字元出現一次或一次以上.

例如 : Regexp /**[0-9]+**/ 用以表示一位或一位以上的數字.

**?** 形容其前的字元可出現一次或不出現.

例如 : Regexp /**[+-]?[0-9]+**/ 表示數字(一位以上)之前可出現正負號或不出現正負號.

**(...)** 用以括住一群字元,且將之視成一個group(見下面說明)

例如 :

Regexp /**12+**/ 表示字串 “12”, “122”, “1222”, “12222”,...
Regexp /**(12)+**/ 表示字串 “12”, “1212”, “1212”, “1212”....
上式中 12 以 ( ) 括住, 故 “+”所形容的是 12, 重複出現的也是 12.

**|** 表示邏輯上的”或”(or)

例如 :

Regexp /**Oranges?|apples?|water**/ 可用以表示 :

字串 “Orange”, “oranges” 或 “apple”, “apples” 或 “water”

- **match** 是什麼 ?

討論 Regexp 時, 經常遇到 “某字串合於( match )某 Regexp”的字眼.
其意思為 : “這個 Regexp 可被解釋成該字串”.

例如 ：

字串 "the" 合於(match) Regexp /[Tt]he/.
因為 Regexp /[Tt]he/ 可解釋成字串 "the" 或 "The", 故字串 "the" 或 "The" 都合於(match) Regexp /[Th]he/.

AWK 中提供二個關係運算元(Relational Operator,見註一) ~ !~, 它們也稱之為 match, not match 但函義與一般常稱的 match 略有不同.

其定義如下 ：

A 表一字串, B 表一 Regular Expression

只要 A 字串中存在有子字串 可 match( 一般定義的 match) Regexp B , 則 A ~ B 就算成立, 其值為 true, 反之則為 false.

!~ 的定義與 ~ 恰好相反.

例 如 ：

"another" 中含有子字串 "the" 可 match Regexp /[Tt]he/ , 所以 "another" ~ /[Tt]he/ 之值為 true.

註 一 ：有些論著不把這兩個運算元( ~, ~ )與 Relational Operators 歸為一類 .

- 應用 Regular Expression 解題的簡例

下面列出一些應用 Regular Expression 的簡例, 部分範例中會更動 $0 之值, 若您使用的 AWK 不容許使用者更動 $0時, 請改用 gawk.

1. 將檔案中所有的字串 "Regular Expression" 或 "Regular expression" 換成 "Regexp"

```
awk '
   {   gsub( /Regular[ \t]+[Ee]xpression/, "Regexp")
       print
   }
  ' $*
```

2. 去除檔案中的空白行(或僅含空白字元或tab)

```
awk '
      $0 !~ /^[ \t]*$/  {  print   }
 ' $*
```

3. 在檔案中俱有 ddd-dddd(電話號碼型態, d 表digital)的字串前加上"TEL : "

```
awk '
    {   gsub( /[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]/, "TEL : &" )
        print
    }
  ' $*
```

4. 從檔案的 Fullname 中分離出 路徑 與 檔名

```
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
      }
     ' $*
```

結果印出

```
    path : /usr/local/bin   name : xdvi
```

5. 將某一數值改以現金表示法表之(整數部分每三位加一撇,且含二位小數)

```
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

       }
   ' $*
```

結果印出

```
    $123,456,789.00
```

6. 把檔案中所有具 “program數字.f”形態的字串改為 “[Ref : program數字.c]”

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

# Index