Lecture10.txt

Parameters and Variables

ตัวแปร

Keyword Variables

เป็นตัวแปรที่เชลล์สร้างและกำหนดค่าเมื่อเชลล์เริ่มทำงาน

ตัวแปรที่ผู้ใช้สร้างขึ้น

- ชื่อตัวแปร ขึ้นต้นด้วย อักษรหรือ underscore, ตามด้วย อักษร ตัวเลข หรือ underscore
- เปลี่ยนค่าได้ตามความต้องการ
- กำหนดให้มีสถานะเป็น read-only :- นำค่าไปใช้ได้อย่างเดียว เปลี่ยนค่าไม่ได้
- กำหนดให้เป็นตัวแปรชนิด global (หรือตัวแปรสิ่งแวดล้อม (environmental variable)) :- ตัวแปรที่ child process ของเชลล์ที่กำหนดตัวแปรนั้นใช้ได้
- **แนวปฏิบัติ (Convention) ตัวแปร global ใช้ชื่อเป็นอักษรตัวใหญ่ล้วน ตัวแปร local ใช้อักษรตัวเล็กตัวใหญ่ผสมกัน**

การกำหนดค่าให้กับตัวแปร

ชื่อตัวแปร=ค่า

เช่น

```
$ NAME=Informatics
```

กำหนดสายอักขระ Informatics ให้กับตัวแปร NAME ในกรณีที่ยังไม่เคยมีตัวแปรนี้มาก่อน ให้ทำการสร้างตัวแปรก่อนกำหนดค่า

```
if   ไม่พบตัวแปร NAME   then
     สร้างตัวแปร NAME
end{if}
กำหนดค่าให้กับตัวแปร
```

ในกรณีที่มีมีวรรคระหว่างชื่อตัวแปรและเครื่องหมาย= และ/หรือ ระหว่างเครื่องหมาย=และค่า จะเป็น error เช่น

```
$ NAME = Informatics
Sorry, command-not-found has crashed! ...
```

ความรู้เดิม - เชลล์จะ fork เชลล์ย่อย เพื่อ run คำสั่งที่ผู้ใช้ป้อนเข้ามาเป็น command line

ความรู้ใหม่ - **ตัวแปรที่สร้างในเชลล์ย่อยใด มีขอบเขตเฉพาะในเชลล์ย่อยนั้น**

```
$ cat my_script
TMPDIR=/home/staff/jira/tmp     # สร้างตัวแปร TMPDIR และกำหนดค่า
echo $TMPDIR                    # แสดงค่าของตัวแปร TMPDIR

$ my_script                     # ให้ my_script ทำงาน
/home/staff/jira/tmp
$ echo $TMPDIR                  # สั่งให้แสดงค่าของตัวแปร TMPDIR จาก command line
```

ไม่มีการแสดงผล เพราะอะไร ?

Keyword variables

Keyword variables คือตัวแปรที่มีความหมายพิเศษสำหรับเชลล์ เช่น

```
HOME    home directory
```

PATH เส้นทางการค้นแฟ้มคำสั่งที่ผู้ใช้ป้อนเข้ามา

Positional and Special parameters

```
---------------------------------
Positional parameters  1, 2, 3, ... , 9       command line argument
                       @              บรรทัดคำสั่ง
                       #              จำนวน argument จาก command line
                       ?              return code ของโพรเซส
```

User-Create variables

```
---------------------
```

ประกาศ(declare) และให้ค่าเริ่มต้น(initialize)

```
$ person=max
$ echo $person
max
$ echo person
person
```

เชลล์จะทำการแทนชื่อตัวแปรด้วยค่าเฉพาะเมื่อมีการใช้ $ นำหน้าชื่อตัวแปร

การ quote กับ $

```
--------------
```

การทดลองต่อไปนี้

```
$ echo $person
max
$ echo "$person"
max
$ echo '$person'
$person
$ echo \$person
$person
```

สรุปได้ว่า ...

เนื่องจาก " " สามารถ quote อักขระพิเศษได้เป็นจำนวนมาก แต่ไม่มีผลต่อ $ จึงเหมาะสำหรับใช้กำหนดค่าที่มีวรรคหรือ tab เช่น

```
$ person="max and zach"
$ echo $person
max and zach
$ person=max and zach
Sorry, command-not-found has crashed! ...
```

เกร็ดความรู้ : เมื่อใช้ตัวแปรเป็น argument ของคำสั่ง เชลล์จะแทนชื่อตัวแปรด้วยค่า และนำค่านั้นไปใช้เป็น argument ของคำสั่ง ในกรณีที่ค่าของตัวแปรมีอักขระพิเศษเช่น * หรือ ?

```
$ ls
max.report
max.summary

$ memo=max*                # เชลล์ไม่ขยาย * เป็น pathname ในตอนกำหนดค่า
$ echo "$memo"             # เครื่องหมายคำพูดคู่ ป้องกันการขยาย * เป็น pathname
max*
$ echo $memo               # เชลล์ขยาย $memo เป็น max* จากนั้นจึงขยาย max เป็น
max.report  max.summary    # pathname
```

รูปแบบพิเศษของ $<ชื่อตัวแปร>
ในกรณีที่ต้องการแยกชื่อตัวแปรออกจากข้อความที่ตาม **ให้กำหนดชื่อตัวแปรไว้ในเครื่องหมาย { }**

```
$ PREF=counter
$ WAY=$PREFclockwise        # ต้องการนำค่าในตัวแปร $PREF และคำว่า clockwise มาต่อกัน
                            # และกำหนดให้ตัวแปร WAY
$ FAKE=$PREFfeit            # ต้องการ $PREF + feit กำหนดให้ WAY
$ echo $WAY $FAKE
```

ไม่มีการแสดงผลใด เพราะก่ารกำหนดค่า WAY=$PREFclockwise เป็นการนำค่าของตัวแปรชื่อ PREFcolockwise กำหนดให้ตัวแปร WAY ซึ่งตัวแปรนี้ยังไม่มีในระบบจึงไม่มีค่า ทำให้ตัวแปร WAY มีค่าว่างตามไปด้วย แก้ปัญหาด้วย

```
$ PREF=counter
$ WAY=${PREF}clockwise      # ต้องการนำค่าในตัวแปร $PREF และคำว่า clockwise มาต่อกัน
                            # และกำหนดให้ตัวแปร WAY
$ FAKE=${PREF}feit                  # ต้องการ $PREF + feit กำหนดให้ WAY
$ echo $WAY $FAKE
counterclockwise counterfeit
```

เชลล์อ้างถึง argument ใน command โดยใช้ตำแหน่ง ตัวแปร $0 เป็น ชื่อของคำสั่ง, ตัวแปร $1 เป็น argument ตัวแรก, ตัวแปร $2 เป็น argument ตัวที่สอง, ... , ตัวแปร $9 เป็น argument ลำดับที่ 9 argument ตั้งแต่ลำดับที่ 10 เป็นต้นไปต้องอ้างอิงโดยใช้วงเล็บปีกกาช่วย เช่น ${10}

unset: Removes a Variables

ตัวแปรที่สร้างขึ้นในสิ่งแวดล้อมของเชลล์ใดจะคงอยู่ตลอดไปจนกว่าเชลล์นั้นจะเลิกการทำงาน หากต้องการยกเลิกค่าของตัวแปร ทำได้โดยการกำหนดให้เป็นค่าว่าง (null)

```
$ person=
```

ค่าของตัวแปร person จะเป็น null แต่ตัวแปร person ยังคงอยู่ในสิ่งแวดล้อมของเชลล์ หากต้องการกำจัดตัวแปรนั้นออกจากสิ่งแวดล้อม ใช้คำสั่ง unset เช่น

```
$ unset person
```

Variable Attributes - การกำหนด attribute (ลักษณะประจำ) ของตัวแปร

---

readonly: Makes the Value of a Variable Permanent

การกำหนดให้ตัวแปรสามารถอ่านค่าได้อย่างเดียว ไม่สามารถเปลี่ยนค่าได้อีก

การใช้งานผู้ใช้ต้องกำหนดค่าให้กับตัวแปรก่อนที่จะกำหนดให้เป็น readonly เพราะเมื่อกำหนดแล้วจะกำหนดค่าใหม่ไม่ได้ และใช้คำสั่ง unset ลบตัวแปรนั้นไม่ได้ เช่น

```
$ person=zach
$ echo $person
zach
$ readonly person
$ person=helen
-bash: person: readonly variable
$ unset person
-bash: unset: person: cannot unset: readonly variable
```

คำสั่ง readonly เป็นคำสั่งของ shell

```
if   ไม่มี argument   then
```

```
        แสดงชื่อตัวแปร readonly ทั้งหมด
    else
        กำหนดให้ตัวแปรนั้นเป็น readonly
    end{if}
```

การกำหนด attribute นอกจากจะใช้คำสั่ง readonly แล้ว ยังสามารถใช้คำสั่ง declare และ typeset ในการกำหนด attribute ได้ด้วย คำสั่ง declare และ typeset เป็นคำสั่งเดียวกัน การใช้งานจริงจะเลือกใช้คำสั่งใดก็ได้ attribute ที่สามารถกำหนดได้เป็นไปตามตารางต่อไปนี้

attribute ของตัวแปร

attribute	ความหมาย
-a	กำหนดให้ตัวแปรเป็น array
-f	กำหนดให้ตัวแปรเก็บชื่อฟังก์ชัน
-i	กำหนดให้ตัวแปรเป็นชนิดจำนวนเต็ม
-r	กำหนดให้ตัวแปรมีสถานะอ่านอย่างเดียว
-x	กำหนดให้ตัวแปรมีสถานะเป็นตัวแปร global และ export

```
    $ declare person=max              # กำหนดตัวแปรตามปกติ จะมีคำสั่ง declare หรือไม่ก็ได้
    $ declare -r person2=zach         # person2 เป็นตัวแปรชนิด readonly
    $ declare -rx person3=helen       # person3 เป็นตัวแปรชนิด readonly และ global
    $ declare -x person4              # person4 เป็นตัวแปรชนิด global ซึ่งทุก subshell สามารถ
                                      # กำหนดค่าได้ และค่าที่กำหนดสามารถใช้งานได้ทุก subshell
```

ในกรณีที่ต้องการถอน attribute จากตัวแปร ทำได้โดยใช้ + เช่น

```
    $ declare +x person3
```

ถอนสถานะความเป็น global ของตัวแปร person3 แต่ person3 ยังคงเป็นตัวแปร readonly ซึ่งเป็น attribute ที่ "ไม่สามารถ" ถอนได้

คำสั่ง declare โดยไม่มี argument จะแสดงรายชื่อตัวแปรและฟังก์ชันเช่นเดียวกับคำสั่ง set หากเพิ่ม option ของ attribute เช่น -r จะแสดงเฉพาะตัวแปรที่มีสถานะเป็น readonly เท่านั้น

```
    $ declare -r                      $ declare -x
    ...                               ...
    declare -r person2                declare -rx person3
    declare -rx person3               declare -x person4
    ...                               ...
```

attribute –a

สำหรับการกำหนดตัวแปรให้เป็น array ทำความเข้าใจจากตัวอย่างดังนี้

```
    $ declare  -a array=([0]="hello" [2]="world" [5]="120")
    $ echo ${array[0]}
    hello
    $ echo ${array[2]}
    world
    $ echo ${array[5]}
    120
```

สมาชิก array ของเชลล์ไม่จำเป็นต้องต่อเนื่องกัน การกำหนดค่าทำเฉพาะสมาชิกที่ต้องการใช้งานก็ได้

attribute –i

โดยปกติค่าของตัวแปรเก็บเป็นสายอักขระเสมอ ในกรณีของตัวเลขเก็บเป็น string of digits เมื่อนำตัวแปรไปใช้ในนิพจน์คณิตศาสตร์ เชลล์จะเปลี่ยนข้อมูลเป็นจำนวน ดำเนินการทางคณิตศาสตร์ และแปลงผลลัพธ์เป็น string of digits หากต้องการเก็บค่าในตัวแปรเป็นจำนวนเต็มสำหรับใช้ในนิพจน์คณิตศาสตร์ทำได้โดยใช้ -i เช่น

```
$ n=6/3
$ echo "n = $n"
n = 6/3

$ declare -i n
$ n=6/3
$ echo "n = $n"
n = 2
```

การตรวจสอบความสมเหตุสมผลของข้อมูล (data validation)

ในวงการคอมพิวเตอร์มีคำพูดติดปากกันอยู่ประโยคหนึ่งคือ

```
"Garbage in, garbage out."
```

หมายความว่าโปรแกรมจะให้ผลลัพธ์ได้ถูกต้อง ก็ต่อเมื่อข้อมูลเข้าถูกต้อง หากข้อมูลเข้าไม่ถูกต้อง หรือเป็นข้อมูลขยะ (garbage) ผลลัพธ์ก็เป็นข้อมูลขยะเช่นเดียวกัน จึงต้องพยายามตรวจสอบข้อมูลเข้าให้ดีที่สุด เรียกว่า data validation ซึ่งสามารถตรวจสอบได้เพียงว่าข้อมูลที่ได้รับอยู่ในพิสัยที่เป็นไปได้หรือไม่ หรือเป็นข้อมูลที่สมเหตุสมผลหรือไม่ สำหรับการตรวจสอบว่าข้อมูลที่ได้รับถูกต้องหรือไม่ เป็นเรื่องที่ไม่สามารถทำได้ เช่น ไม่สามารถตรวจสอบว่าวันเกิดที่ได้รับจากแป้นพิมพ์เป็นวันเกิดของบุคคลนั้นจริงหรือไม่ แต่สามารถตรวจสอบความสมเหตุสมผลได้ เช่นหากวันเกิดเป็น 29 กุมภาพันธ์ 2561 ย่อมไม่สมเหตุสมผล เพราะปี พ.ศ. 2561 ไม่ใช่ปีอธิกสุรทิน

การตรวจสอบความสมเหตุสมผลของข้อมูล จำเป็นจะต้องรู้พิสัยของข้อมูลที่เป็นไปได้ เช่น ส่วนสูงของบุคคลทั่วไป (ไม่รวมเด็กและทารก) มีค่าในช่วง 50 - 275 ซม. อาจมีคำถามว่าตัวเลขนี้ได้มาจากไหน? คำตอบคือสืบค้นจาก World Wide Web ซึ่งได้ส่วนสูงของคนที่เตี้ยที่สุดในโลกเป็น 54.6 ซม. และคนที่สูงที่สุดในโลกคือ 272 ซม. ปรับแต่งเล็กน้อยให้เป็นพิสัยของตัวเลขที่จำได้ง่ายคือ 50 - 275 ซม.

ประเด็นที่สำคัญคือ validation ตรวจสอบได้เพียงว่าข้อมูลเข้าอยู่ในช่วงที่สมเหตุสมผลหรือไม่ ไม่สามารถบอกได้ว่าข้อมูลนั้นถูกต้องหรือไม่ ผู้ที่จะบอกได้ว่าข้อมูล "ถูกต้อง" คือเจ้าของข้อมูลนั้น

การทำ data validation ในกรณีของ shell script ที่รับข้อมูลจาก command line argument ต้องตรวจสอบจำนวน argument,
และตรวจสอบความสมเหตุสมผลของ argument แต่ละตัว ซึ่งต้องใช้ตัวแปร $#, $@, $1, $2, ... ที่เรียนผ่านมาแล้ว ร่วมกับโครงสร้างคุมการทำงานแบบมีทางเลือก (if)

แต่ก่อนที่จะรู้จักโครงสร้าง if ต้องรู้จักคำสั่ง test, ตัวแปร $? ที่ใช้เก็บสถานะการเลิกทำงานของโพรเซส เรียกว่า exit status หรือ return code เสียก่อน ดังต่อไปนี้

Exit status และความหมาย

เมื่อผู้ใช้ป้อนคำสั่ง เช่น ls ทางแป้นพิมพ์ เชลล์จะทำการสร้าง (fork) โพรเซสลูก จากนั้นจึงกำหนดให้โพรเซสลูกอ่านคำสั่งจากแฟ้ม /bin/ls และทำงานตามคำสั่งนั้น เมื่อโพรเซสลูกทำงานเสร็จจะคืนค่าสถานะการเลิกทำงาน หรือ exit status ให้แก่โพรเซสแม่ซึ่งจะเก็บไว้ในตัวแปร พิเศษ คือ $? และใน "ทันที" ที่โพรเซสนั้นเลิกทำงาน ผู้ใช้สามารถอ่านค่าสถานะในตัวแปรนี้ได้โดยใช้
คำสั่ง echo เช่น

```
$ ls
$ echo $?
```

หรือรวมเป็น 'บรรทัดคำสั่ง' เดียว โดยใช้ semicolon คั่นระหว่างคำสั่งเป็น

```
$ ls ; echo $?
```

ผลลัพธ์ที่ได้เป็นเลขจำนวนเต็มบวก โดย 0 หมายถึงโพรเซสทำงานได้สำเร็จตามปกติ (Successful), ค่าที่ "ไมใช่" 0 แสดงว่าเกิดปัญหาขึ้นในระหว่างการทำงาน ค่านี้เป็นรหัสแสดงปัญหาหรือความผิดพลาดที่เกิดขึ้นในการทำงาน ซึ่งระบบปฏิบัติการ Unix กำหนดหมายเลขและความหมายของสถานะการทำงานไว้เป็นมาตรฐาน เรียกดูได้ผ่านคำสั่ง perror

คำสั่ง perror - อธิบายความหมายของ "รหัสความผิดพลาด" (error code) มีรูปแบบการใช้งานเป็น

```
$ perror    <รหัสความผิดพลาด>
```

เช่น ขอดูความหมายของรหัสความผิดพลาดหมายเลข 13

```
$ perror  13
OS error code  13:  Permission denied
```

นั่นคือรหัสความผิดพลาดหมายเลข 13 ของระบบปฏิบัติการคือ Permission denied (ไม่มีสิทธิการใช้งาน)
ในกรณีที่ต้องการดูความหมายของรหัสทั้งหมด ทำได้โดยใช้ตัวเลือก -a ซึ่งจะแสดงรหัสความผิดพลาดทั้งหมดของระบบปฏิบัติการ และในกรณีที่มีการติดตั้งตัวจัดการฐานข้อมูล MySQL จะแสดงรหัสความผิดพลาดของ MySQL ต่อท้าย เช่น

```
$ perror -a
  1 = Operation not permitted
  2 = No such file or directory
  3 = No such process
  ... ... ...
 -30999 = DB_INCOMPLETE: Sync didn't finish
  ... ... ...
```

ตัวอย่างการประยุกต์ใช้งาน exit status ที่เห็นได้ชัด คือการใช้งานร่วมกันระหว่างโปรแกรมภาษา C และ shell script รายละเอียดเป็นตามปฏิบัติการ การใช้งาน exit status โดยทั่วไปมักใช้ในกรณีที่มีการเรียกใช้คำสั่งต่อเนื่องกันหลายคำสั่ง เมื่อคำสั่งใดคำสั่งหนึ่งทำงานผิดพลาด ให้ยุติการทำงานของ script ในทันที เพื่อป้องกันการเกิดความผิดพลาดต่อเนื่องกันโดยไม่จำเป็น และอาจทำให้เกิดข่าวสารแสดงความผิดพลาดมากมายที่ไม่จำเป็น อันจะทำให้ผู้ใช้สับสน อย่าลืมว่าในการทำงานจริง ผู้เขียนโปรแกรมไม่ได้ใช้งานโปรแกรมที่เขียน และผู้ใช้งานก็ไม่ใช่ผู้เขียนโปรแกรม ดังนั้นจึงต้องออกแบบ user interface ให้ดี เพื่อให้ผู้ใช้สามารถใช้งานโปรแกรมได้ง่าย ไม่ยุ่งยาก และไม่สับสน การจะทำเช่นนี้ได้ต้องทดสอบ exit status ด้วยโครงสร้าง if เขียนเป็น "คำสั่งลำลอง" ได้เช่น

```
                                        :: คำสั่งลำลอง ::
เรียกใช้งานโปรแกรมที่ 1
if    โปรแกรมที่ 1 ทำงานสำเร็จ    then
      เรียกใช้งานโปรแกรมที่ 2
      if    โปรแกรมที่ 2 ทำงานสำเร็จ    then
              ... ... ...
              if    ... ... ...    then
                    ... ... ...
               else
                    เลิกการทำงาน
      else
              เลิกการทำงาน
else
       เลิกการทำงาน
end {if}
```

โครงสร้าง if ใน shell script

รูปแบบทั่วไปของคำสั่ง if (สำหรับเชลล์ตามมาตรฐาน POSIX)

```
if   เงื่อนไข
then
     คำสั่งทำงานเมื่อเงื่อนไขเป็นจริง
fi
```

สังเกตคำสั่ง "fi" ที่ใช้ปิดโครงสร้าง หรืออาจเขียนย่อได้เป็น

```
if   เงื่อนไข ; then
     คำสั่งทำงานเมื่อเงื่อนไขเป็นจริง
fi
```

อย่าลืมว่า **เครื่องหมาย semicolon (;) ใช้สำหรับแยกคำสั่งที่อยู่ในบรรทัดเดียวกันออกจากกัน** และในกรณีที่เป็น if .. else if .. else .. เป็นดังนี้

```
if   เงื่อนไข 1 ; then
     คำสั่งทำงานเมื่อเงื่อนไข 1 เป็นจริง
elif   เงื่อนไข 2 ; then
     คำสั่งทำงานเมื่อเงื่อนไข 2 เป็นจริง
else
     คำสั่งทำงานเมื่อเงื่อนไข 1 หรือ 2 เป็นเท็จ
fi
```

เงื่อนไขของ if โดยทั่วไปเป็น "คำสั่ง" ซึ่งเป็นได้ทั้งโปรแกรมที่เป็น binary file (executable file) , shell script, ฟังก์ชันของ shell, หรือตัวแปรที่อ้างอิงถึงโปรแกรมเหล่านี้ หรือเป็นชื่อย่อของโปรแกรม (alias)

คำสั่ง if ของโครงสร้าง

- จะกำหนดให้ "คำสั่ง" ที่ตามมาทำงาน
- ทดสอบว่า "คำสั่ง" นั้นทำงานสำเร็จ (successful) หรือไม่

  หากสำเร็จ "คำสั่ง" นั้นคืนสถานะการทำงานเป็น 0

  หากไม่สำเร็จจะคืนเป็นค่าอื่นที่ไม่ใช่ 0

การคืนค่าเช่นนี้อาจทำให้สับสนบ้าง (เพราะ "ดูเหมือน" 0 ทำให้เงื่อนไขเป็นจริง) แต่เป็นวิธีการในการออกแบบที่ดี เพราะโปรแกรมอาจทำงานไม่สำเร็จได้จากหลายสาเหตุหลายกรณี แต่การทำงานได้สำเร็จมีเพียงกรณีเดียว และการเขียนโปรแกรมส่วนใหญ่จึงต้องใช้ตัวแปร $? (exit status) กำหนดเป็นเงื่อนไข

ในกรณีที่เงื่อนไขของ if เป็นคำสั่งแบบ pipeline ซึ่งมีหลายคำสั่ง, สถานะการทำงาน หรือ exit status ที่ pipeline จะคืนให้คือ exit status ของคำสั่งสุดท้ายใน pipeline

คำสั่ง true และคำสั่ง false

---

shell ตามมาตรฐาน POSIX มีคำสั่ง "true" และคำสั่ง "false" เป็นคำสั่งภายในของ shell ซึ่งจะคืน exit status เป็น 0 และ 1 ตามลำดับ เช่น

```
$ true; echo $?
0
$ false; echo $?
1
```

และมี ! เป็นนิเสธ (negation operator) สำหรับ "กลับ" ค่าของ exit status เช่น

```
$ ! true; echo $?
1
```

ตัวอย่างการประยุกต์ใช้งาน เช่น ใช้เป็นเงื่อนไขของการทำซ้ำตลอดเวลา โดยใช้ if เป็นตัวตรวจสอบภาวะเลิกทำงาน และใช้คำสั่ง break เพื่อเลิกการทำซ้ำ

```
while  true ; do                    # เงื่อนไขของ while เป็นจริงเสมอ
     ...
     if  [ ... ] ; then             # เมื่อมีเหตุการณ์ที่กำหนดเกิดขึ้น,
         break                      # ออกจาก loop
     fi
     ...
done
```

[...] ที่ใช้เป็นเงื่อนไขของโครงสร้าง if เป็นรูปย่อของคำสั่ง test ซึ่งมีรายละเอียดอยู่ตอนท้ายบทความนี้

คำถาม-คำตอบที่น่าสนใจ

จาก script ต่อไปนี้คือ

```
1  if [ false ]; then
2       echo "True"
3  else
4       echo "False"
5  fi
```

Q: เพราะเหตุใดเมื่อให้ทำงาน จึงแสดงผลเป็น "True" ?
A: เพราะ [false] เป็นรูปย่อของคำสั่ง "test false" หรือคำสั่ง test ที่มี false เป็น argument ซึ่งเป็นคำสั่งที่ใช้ทดสอบ ว่า argument ของคำสั่ง test ไม่ใช่สายอักขระว่าง (empty string) ในกรณี "false" ไม่ใช่สายอักขระว่าง test จึงคืนค่าเป็น 0 ทำให้เงื่อนไขของ if เป็นจริงจึงแสดงผลเป็นคำว่า "True" และ ในทำนองเดียวกันหากเปลี่ยนจากคำว่า false เป็นตัวแปร var ก็ให้ผลเช่นเดียวกัน

```
1  var=false
2  if [ $var ]; then
3       echo "True"
4  else
5       echo "False"
6  fi
```

เนื่องจาก false เป็นคำสั่ง หากต้องการให้ผลถูกต้องตามที่ควรเป็น ต้องใช้คำสั่ง false โดยตรงดังนี้

```
1  if  false ; then
2       echo "True"
3  else
4       echo "False"
5  fi
```

ส่งท้าย

"true" *และ* ***"false"*** *เป็น "คำสั่ง" ไม่ใช่ "ค่า"* จึงไม่สามารถนำไปใช้กำหนดค่าตัวแปร เพื่อให้เป็นตัวแปรชนิด boolean ได้
คำสั่ง

```
$ var=false
```

เป็นการกำหนดให้ตัวแปร var มีค่าเป็นสายอักขระ "false" ทดสอบได้โดยการแสดงค่าดังนี้

```
$ echo $var
false                                   # สายอักขระของตัวแปร var ไม่ใช่ค่าตรรกะ (boolean)
```

คำสั่ง test - คำสั่งที่ใช้เป็นเงื่อนไข

คำสั่ง test ใช้สำหรับทดสอบตามที่กำหนดด้วย option และคืนสถานะการทำงานเป็น 0 ในกรณีที่การทดสอบนั้นเป็นจริง หรือค่าที่ไม่ใช่ 1 ในกรณีที่การทดสอบเป็นเท็จ เนื่องจาก test เป็นคำสั่งจึงใช้เป็นเงื่อนไขของ if (และ while) ได้ เช่น

```
if  test -z $var ; then
    # หากตัวแปรมีขนาดเป็นศูนย์ (zero)
    # หรือตัวแปรนั้นไม่มีการกำหนดค่า
    ... ... ...
fi
```

```
if  test -f test.dat ; then
    # แฟ้ม test.dat เป็นแฟ้มธรรมดาและมีจริง
    # ไม่ใช่ ไดเรกทอรี, pipe, หรือ device file
    ... ... ...
fi
```

นอกจากนี้แล้ว test ยังรับ option ที่เป็นสัญลักษณ์บางตัวด้วย เช่น

```
if  test  $var = hello ; then
    # ค่าของตัวแปร var เป็น "hello"
     ... ... ...
fi
```

```
if  test  $var != hello ; then
    # ค่าของตัวแปร var ไม่ใช่ "hello"
    ... ... ...
fi
```

หมายเหตุ

1. option (หรือ operator), = และ != ใช้เปรียบเทียบสายอักขระ (string) หรือตัวเลขก็ได้
2. การเปรียบเทียบจำนวน ใช้ -eq (equal, เท่ากัน), -ne (not equal. ไม่เท่ากัน), -gt (greater, มากกว่า), -ge (greater or equal, มากกว่าหรือเท่ากับ), -lt (less than, น้อยกว่า), -le (less than or equal to, น้อยกว่าหรือเท่ากับ)

และด้วยเหตุที่มีการใช้งานคำสั่ง test เป็นเงื่อนไขของ if และ while, เพื่อให้เขียนโปรแกรมสะดวกและมีรูปแบบสอดคล้องกับภาษาสำหรับเขียนโปรแกรมแบบอื่นๆ จึงกำหนดให้ test มีรูปย่อของคำสั่งเป็น [...] จึงเขียนคำสั่ง if ตามตัวอย่างข้างต้นได้เป็น

```
if  [ -z $var ]; then
if  [ $var = hello ]; then
```

```
if  [ -f test.dat ]; then
if  [ $var != hello ]; then
```

เนื่องจาก [และ] เป็น "คำสั่ง" จึงต้องมีวรรคหลัง [และวรรคก่อน] เพื่อแยกคำสั่ง และ command line argument ออกจากกัน

นิเสธ และ การเชื่อมเงื่อนไขเข้าด้วยกัน

คำสั่ง if มาพร้อมด้วย ! (นิเสธ), && (and) และ || (or) เช่น

```
if  test $x -ge 0 && test $x -le 10; then       # 0 <= x <= 10
```

หรือใช้ option ของ test -a (and), -o (or) เช่น

```
if  test $x -ge 0 -a  $x -le 10; then           # 0 <= x <= 10
if  [ $x -ge 0 -a  $x -le 10 ]; then            # 0 <= x <= 10
```

รายละเอียด(บางส่วน-ไม่ใช่ทั้งหมด)ของคำสั่ง test ศึกษาได้จาก ภาคผนวก ค

Quiz :

ให้เขียน Script สำหรับเป็นเครื่องคิดเลขอย่างง่าย เช่น

```
$ calc
You need 3 more arguments
$ calc add
You need 2 more arguments
$ calc sub 10
You need 1 more argument
$ calc add 10 20
30
$ calc sub 10 20
-10
$ calc mul 10 20
200
$ calc div 10 20
0.5
```

ภาคผนวก ก

ชื่อย่อหรือชื่อแฝง (Alias) ของโปรแกรมหรือคำสั่ง

Note: alias เป็นชื่ออีกชื่อหนึ่ง หรือเป็นนามแฝงที่กำหนดให้กับคำสั่งที่มีอยู่แล้ว สร้างโดยใช้คำสั่ง alias ซึ่งเป็นคำสั่งของ shell และมีรูปแบบการใช้งานทั่วไปดังนี้

```
alias  [name=['command']]
```

ตัวอย่างเช่น

```
$ alias rm='rm -i'
```

กำหนดให้ rm เป็นอีกชื่อหนึ่งของคำสั่ง rm -i ซึ่งจะแสดงข้อความให้ผู้ใช้ยืนยันการลบทุกครั้ง หลังจากที่กำหนดแล้ว เมื่อผู้ใช้เรียกใช้คำสั่ง rm จะมีผลเท่ากับการเรียกใช้คำสั่ง rm -i เช่น

```
$ rm data1.txt
rm: remove regular file data1.txt?
```

ซึ่งผู้ใช้เลือกได้ว่าจะลบหรือไม่ โดยการกดแป้น y หรือ n, และหากแฟ้มที่ต้องการลบเป็นแฟ้มว่าง ระบบจะแจ้งด้วย เช่น

```
$ rm  data2.txt
rm: remove regular empty file data2.txt?
```

การกำหนดลักษณะนี้ช่วยให้ใช้คำสั่งพร้อม option ที่ต้องการได้ง่าย

ตัวอย่างการใช้งาน alias อีกตัวอย่างหนึ่งคือ กำหนดคำสั่งล้างจอภาพของระบบปฏิบัติการ Unix คือคำสั่ง clear ให้มีชื่ออีกชื่อหนึ่งเป็น cls ซึ่งเป็นคำสั่งล้างจอภาพสำหรับ Windows Command Line ของบริษัท Microsoft

```
$ alias cls='clear'
```

เมื่อกำหนดแล้วจะทำให้สามารถใช้คำสั่ง cls ล้างจอภาพได้ การใช้งานลักษณะนี้ เหมาะกับผู้ที่คุ้นเคยกับ Windows Command Line ของบริษัท Microsoft ที่เปลี่ยนมาใช้งาน Unix

หากต้องการดูว่า alias ที่กำหนดไว้มีอะไรบ้าง เรียกดูได้โดยใช้คำสั่ง alias เช่น

```
$ alias
alias cls='clear'
alias rm='rm -i'
```

และสามารถลบ alias ที่ไม่ต้องการได้ โดยใช้คำสั่ง unalias เช่น

```
$ unalias cls
```

การยกเลิก alias โดยการกำหนดค่า alias นั้นให้เป็น null ถึงแม้จะทำได้แต่ไม่ควรใช้ เช่น

```
$ alias rm=
```

กรณีนี้ alias rm ยังคงอยู่ แต่ไม่ได้แทนคำสั่งได้ เมื่อเรียกใช้งาน เช่น ต้องการลบแฟ้มชื่อ xx

```
$ rm xx
-bash: ./xx: Permission denied          # ผู้ใช้ไม่มีสิทธิในการ execute โปรแกรม xx
```

ที่เป็นเช่นนี้เพราะเมื่อ shell ขยายคำสั่ง rm xx จะได้ผลลัพธ์เป็น xx เพราะ rm มีค่าเป็น null จึงเกิด error ดังกล่าวขึ้น โดยเป็นเช่นเดียวกับการเรียกใช้ xx โดยตรงจาก prompt

```
$ xx
-bash: ./xx: Permission denied
```

วิธีแก้ปัญหาคือ ทำการลบ alias rm ออกโดยใช้คำสั่ง unalias จะสามารถใช้คำสั่ง rm ของระบบได้

การกำหนด alias ที่กล่าวมาแล้ว สามารถใช้งานได้เฉพาะใน shell ที่มีการกำหนด และใช้ได้จนกว่าผู้ใช้จะ logout หากต้องการกำหนดไว้ใช้เป็นการถาวร ต้องกำหนดในแฟ้ม .bash_profile

---

ภาคผนวก ข

คำสั่ง test และการประยุกต์ใช้งาน

ชื่อคำสั่ง

test : Evaluates an expression (ประมวลผลนิพจน์) -- โปรแกรมอรรถประโยชน์ของ Unix

รูปแบบการใช้งาน :

```
test  <นิพจน์>              # รูปแบบแรกใช้สำหรับ command line
[ <นิพจน์> ]                # รูปแบบที่สองนิยมใช้ใน shell script
```

คำอธิบาย (Description)

test ใช้สำหรับประมวลผลนิพจน์ (expression) และคืนสถานะของการประมวลผลนิพจน์นั้นว่ามีค่าเป็น "จริง" หรือ "เท็จ" คำสั่ง test มีรูปย่อเป็น [<นิพจน์>] ซึ่งเป็นที่นิยมใช้มากกว่า นิพจน์ที่ใช้งานได้มีหลายกลุ่ม ตัวอย่างเช่น

การทดสอบและการเปรียบเทียบสายอักขระ

---

string	เป็นจริงหาก string ที่ต้องการทดสอบไม่ใช่ string ว่าง
-n string	เป็นจริงหาก string มีความยาวมากกว่าศูนย์
-z string	เป็นจริงหาก string มีความยาวเป็นศูนย์

string1 = string2	เป็นจริงหาก string1 และ string2 เป็นข้อความเดียวกัน
string1 != string2	เป็นจริงหาก string1 และ string2 เป็นข้อความที่ต่างกัน

การเปรียบเทียบจำนวนเต็ม

-eq	เท่ากับ (equal to)
-ge	มากกว่าหรือเท่ากับ (greater than or equal to)
-gt	มากกว่า (greater)
-le	น้อยกว่าหรือเท่ากับ (less than or equal to)
-lt	น้อยกว่า (less than)
-ne	ไม่เท่ากับ (not equal to)

การทดสอบชนิดของแฟ้ม

-b <ชื่อแฟ้ม>	เป็นจริงหากเป็นแฟ้มที่มีอยู่จริงและเป็นแฟ้มแทนอุปกรณ์ชนิด block special file
-c <ชื่อแฟ้ม>	เป็นจริงหากเป็นแฟ้มที่มีอยู่จริงและเป็นแฟ้มแทนอุปกรณ์ชนิด character special file
-d <ชื่อไดเรกทอรี>	เป็นจริงหากเป็นไดเรกทอรีที่มีอยู่จริง
-e <ชื่อ>	เป็นจริงหากชื่อนั้น, ชื่อแฟ้ม หรือชื่อไดเรกทอรีที่กำหนดมีอยู่จริง
-f <ชื่อแฟ้ม>	เป็นจริงหากเป็นแฟ้มธรรมดาที่มีอยู่จริง
-h <ชื่อแฟ้ม>	เป็นจริงหากชื่อเป็น แฟ้มที่มีอยู่จริงและเป็น symbolic link

การทดสอบขนาดของแฟ้ม

-s <ชื่อแฟ้ม>	เป็นจริงหากเป็นแฟ้มที่มีอยู่จริงและมีขนาดมากกว่า 0 ไบต์

การทดสอบสิทธิในการใช้งานแฟ้ม

-r <ชื่อแฟ้มหรือชื่อไดเรกทอรี>	เป็นจริงหากเป็นแฟ้มที่มีอยู่จริงและผู้ใช้มีสิทธิ "read" แฟ้มนั้น
-w <ชื่อแฟ้มหรือชื่อไดเรกทอรี>	เป็นจริงหากเป็นแฟ้มที่มีอยู่จริงและผู้ใช้มีสิทธิ "write" แฟ้มนั้น ในกรณีที่เป็นชื่อไดเรกทอรีหมายถึงผู้ใช้มีสิทธิสร้างและลบแฟ้มในไดเรกทอรีนั้น
-x <ชื่อแฟ้มหรือชื่อไดเรกทอรี>	เป็นจริงหากเป็นแฟ้มที่มีอยู่จริงและผู้ใช้มีสิทธิ "execute" ในกรณีที่เป็นชื่อไดเรกทอรีหมายถึงผู้ใช้มีสิทธิอ่านรายชื่อแฟ้มในไดเรกทอรีนั้น

การเปรียบเทียบแฟ้ม

<ชื่อแฟ้ม> -ef <ชื่อแฟ้ม>	เป็นจริงหากแฟ้ม file1 และ file2 เป็นแฟ้มแทนอุปกรณ์ซึ่งมี i-node เดียวกัน
<ชื่อแฟ้ม> -nt <ชื่อแฟ้ม>	เป็นจริงหากมีการแก้ไข (modified) แฟ้ม file1 "หลัง" แฟ้ม file2
<ชื่อแฟ้ม> -ot <ชื่อแฟ้ม>	เป็นจริงหากมีการแก้ไข (modified) แฟ้ม file1 "ก่อน" แฟ้ม file2

Argument ของคำสั่ง test เป็นนิพจน์ที่ประกอบด้วยเงื่อนไขที่ต้องการทดสอบ ในกรณีที่มีเงื่อนไขตั้งแต่สองเงื่อนไขขึ้นไป กำหนด Logical operator ดังนี้

Logical operator

!	นิเสธ (NOT)
-a	AND
-o	OR

เนื่องจาก AND มีลำดับความสำคัญสูงกว่า OR หากต้องการเปลี่ยนลำดับให้ใช้วงเล็บจัดกลุ่ม แต่เนื่องจากวงเล็บเป็นอักขระพิเศษของเชลล์ เมื่อจะใช้งานจึงต้องกำกับด้วย \ เพื่อป้องกันเชลล์ขยายความหมาย และในทำนองเดียวกันหากมีการใช้อักขระพิเศษอื่นใดต้องทำการกำกับอักขระพิเศษนั้น เพื่อป้องกันเชลล์ทำการขยายความอักขระพิเศษนั้น ก่อนส่งให้ test ดำเนินการ และเนื่องจากองค์ประกอบแต่ละส่วนของนิพจน์ เช่นเงื่อนไข สายอักขระ ตัวแปร ต่างเป็น argument ของคำสั่ง test ดังนั้นจึงต้องใช้วรรค

แยกแต่ละส่วนออกจากกัน

การประยุกต์ใช้งานคำสั่ง test

คำสั่ง test สามารถทำงานได้ทั้งจาก command line และ shell script แต่โดยทั่วไปแล้วคำสั่งนี้มีที่ใช้ใน shell script มากกว่า
โดยใช้เป็นเงื่อนไขสำหรับโครงสร้างควบคุมการทำงานแบบมีทางเลือก และแบบทำซ้ำ

การใช้งานคำสั่ง test จาก command line โดยทั่วไปมักใช้ในการทดสอบว่าเงื่อนไขที่สนใจให้ผลลัพธ์เป็นอย่างไรก่อนที่จะนำไปใช้ใน shell script แต่เนื่องจากคำสั่ง test ไม่มีการแสดงผล ดังนั้นผลการทำงานจึงต้องตรวจสอบจากสถานะการทำงานที่คำสั่งนั้นคืนให้แก่เชลล์ ซึ่งเรียกว่า return code

คำสั่งทุกคำสั่งที่มีการเรียกให้ทำงานในระบบปฏิบัติการ Unix เมื่อจบการทำงานจะคืน return code ให้แก่ parent process (เชลล์ที่เป็นผู้เรียก) ซึ่งเชลล์จะจัดเก็บไว้ในตัวแปร $?, return code เป็นจำนวนเต็ม โดย 0 แทนการทำงานที่เสร็จสิ้นสมบูรณ์
เมื่อเรียกใช้งานคำสั่งใดแล้ว สามารถตรวจสอบสถานะการทำงานได้จากตัวแปรนี้ เช่น

```
$ ls
hello.c  test.c

$ ls -l hello.c           # ขอดูรายละเอียดของแฟ้มที่มี
-rw------- 1 jira staff 89 Jan 11 12:58 hello.c

$ echo $?                 # ตรวจสอบสถานะการทำงานของคำสั่งในตัวแปร $?
0                         # return code = 0 แสดงว่าคำสั่งที่แล้วทำงานเสร็จสมบูรณ์

$ ls -l  xyz              # ขอดูรายละเอียดของแฟ้มที่ไม่มี
ls: cannot access xyz: No such file or directory
$ echo $?                 # return code = 2 แสดงว่ามีปัญหาในการทำงาน
2
```

นำมาใช้ในการศึกษาทำความเข้าใจเงื่อนไข นิพจน์เปรียบเทียบ และนิพจน์ตรรกศาสตร์

```
$ test 4 -gt 3            # 4 > 3?
$ echo  $?
0                         # ผลการเปรียบเทียบเป็น "จริง"

$ test 3 -nq 3            # 3 <> 3?
$ echo  $?
1                         # ผลการเปรียบเทียบเป็น "เท็จ"
```

หมายเหตุ

นอกจากคำสั่ง test ที่เป็นโปรแกรมอรรถประโยชน์ของ Unix แล้ว ยังมีคำสั่ง test ที่เป็นคำสั่งภายในของ bash ด้วย ซึ่งมีการทำงานเหมือนกัน แต่ test ของ bash ไม่มีรูปย่อเป็นวงเล็บก้ามปูเท่านั้น และเนื่องจาก test เป็นคำสั่งภายในของเชลล์ จึงมีลำดับความสำคัญในการทำงานสูงสุด ดังนั้นจึงไม่ควรตั้งชื่อโปรแกรมที่เขียนขึ้นเองเป็น test เพราะจะไม่สามารถเรียกใช้งานได้ เมื่อป้อนชื่อโปรแกรมเป็น test จะได้การทำงานของคำสั่ง test ของเชลล์แทน จึงดูเหมือนโปรแกรมที่พัฒนาขึ้นไม่ทำงาน สร้างความสับสนให้แก่ผู้เริ่มเขียนโปรแกรมได้มาก

