ĐẠI HỌC QUỐC GIA TP HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

Tài liệu hướng dẫn thực hành

HỆ ĐIỀU HÀNH

Biên soạn: ThS Phan Đình Duy
ThS Nguyễn Thanh Thiện
KS Trần Đại Dương
KS Trần Hoàng Lộc

LƯU HÀNH NỘI BỘ
Thành phố Hồ Chí Minh – Tháng 09/2014

MỤC LỤC

NỘI QUY THỰC HÀNH

1. Sinh viên tham dự đầy đủ các buổi thực hành theo quy định của giảng viên hướng dẫn (GVHD) (6 buổi với lớp thực hành cách tuần hoặc 10 buổi với lớp thực hành liên tục).
2. Sinh viên phải chuẩn bị các nội dung trong phần "Sinh viên viên chuẩn bị" trước khi đến lớp. GVHD sẽ kiểm tra bài chuẩn bị của sinh viên trong 15 phút đầu của buổi học (nếu không có bài chuẩn bị thì sinh viên bị tính vắng buổi thực hành đó).
3. Sinh viên làm các bài tập ôn tập để được cộng điểm thực hành, bài tập ôn tập sẽ được GVHD kiểm tra khi sinh viên có yêu cầu trong buổi học liền sau bài thực hành đó. Điểm cộng tối đa không quá 2 điểm cho mỗi bài thực hành.

Bài 2. LẬP TRÌNH TRONG MÔI TRƯỜNG SHELL

2.1 Mục tiêu

Bài thực hành này giúp sinh viên:

- Làm quen và tìm hiểu cách sử dụng ngôn ngữ shell để lập trình
- Thực hành lập trình một số cầu trúc điều khiển cơ bản của shell
- Viết được một số chương trình đơn giản bằng ngôn ngữ shell

2.2 Nội dung thực hành

- Thực hành các lệnh tương tác với hệ điều hành Linux thông qua shell
- Tìm hiểu và thực hành ngôn ngữ shell

2.3 Sinh viên chuẩn bị

- Đọc phần hướng dẫn thực hành trước ở nhà
- Viết lại các đoạn code trong phần 2.4 theo hướng dẫn

2.4 Hướng dẫn thực hành

2.4.1 Sử dụng shell như ngôn ngữ lập trình

Shell có thể được sử dụng như một ngôn ngữ lập trình. Có hai cách để viết chương trình điều khiển shell. Cách thứ nhất là gõ chương trình ngay từ dòng lệnh, đây cũng là cách đơn giản nhất. Tuy nhiên một khi đã thành thạo có thể gộp các lệnh vào một tệp để chạy (chúng tương đương với cách DOS gọi tệp *.bat), cách này hiệu quả và tận dụng triệt để tính năng tự động hóa của shell.

2.4.1.1 Điều khiển shell từ dòng lệnh

Chúng ta hãy bắt đầu với ví dụ đơn giản sau. Giả sử trên đĩa cứng có rất nhiều file mã nguồn .c, bạn muốn truy tìm và hiển thị nội dung của các tệp nguồn chứa chuỗi main(). Thay vì dùng lệnh *grep* để tìm ra từng file sau đó quay lại dùng lệnh *more* để hiển thị file, ta có thể dùng lệnh điều khiển shell tự động như sau:

```
$ for file in *
do
 if grep -l 'main( ) ' $file
 then
 more $file
 fi
 done
```

Khi gõ một lệnh chưa hoàn chỉnh từ dấu nhắc của shell, shell sẽ chuyển dấu nhắc thành >, shell chờ nhập đầy đủ các lệnh trước khi thực hiện tiếp. Shell tự động hiểu được khi nào thì lệnh bắt đầu và kết thúc. Trong ví dụ trên lệnh *for* . . . *do* sẽ kết thúc bằng *done*.

Khi gõ xong *done*, shell sẽ bắt đầu thực thi tất cả những gì đã gõ vào bắt đầu từ *for*. Ở đây, *file* là một biến của shell, trong khi * là một tập hợp đại diện cho các tên tệp tìm thấy trong thư mục hiện hành.

Bất tiện của việc điều khiển ngôn ngữ shell từ dòng lệnh là khó lấy lại khối lệnh trước đó để sửa đổi và thực thi một lần nữa. Nếu ta nhấn phím up/down thì shell có thể trả lại khối lệnh như sau:

```
$ for file in * ; do ; if grep -1 'main( )' $file;
then ; more $file; fi; done
```

Đây là cách các shell Linux vẫn thường làm để cho phép thực thi nhiều lệnh cùng lúc ngay trên dòng lệnh. Các lệnh có thể cách nhau bằng dấu (;). Ví dụ:

```
$ mkdir myfolder; cd myfolder;
```

sẽ tạo thư mục myfolder bằng lệnh mkdir sau đó chuyển vào thư mục này bằng lệnh cd. Chỉ cần gõ Enter một lần duy nhất để thực thi hai lệnh cùng lúc. Tuy nhiên sửa chữa các khối lệnh như

vậy không dễ dàng và rất dễ gây lỗi. Chúng chỉ thuận tiện khi kết hợp khoảng vài ba lệnh. Để dễ bảo trì, bạn có thể đưa các lệnh vào

một tập tin và yêu cầu shell đọc nội dung tập tin để thực thi lệnh. Những tập tin như vậy gọi là tập tin kịch bản (shell script).

2.4.1.2 Điều khiển shell bằng tập tin kịch bản (script file)

Trước hết bạn dùng lệnh $cat > first.sh hay các trình soạn thảo như vi hay emacs (hoặc mc) hay gedit để soạn nội dung tập tin first.sh như sau:

```
 # !/bin/ sh
 # first.sh
 # Script này sẽ tìm trong thư mục hiện hành các chuỗi mang nội dung
# main( ), nội dung của file sẽ được hiển thị ra màn hình nếu tìm thấy.
 for file in *
 do
 if grep -l 'main( ) ' $file
 then
more $file
fi
 done
```

exit 0

Không như chú thích của C, một dòng chú thích (comment) trong ngôn ngữ shell bắt đầu bằng ký tự #. Tuy nhiên, ở đây có

một chú thích hơi đặc biệt đó là #!/bin/sh. Đây thực sự không phải là chú thích. Cặp ký tự #! là chỉ thị yêu cầu shell hiện tại triệu gọi shell sh nằm trong thư mục /bin. Shell sh sẽ chịu trách nhiệm thông dịch các lệnh nằm trong tập tin script được tạo.

Có thể chỉ định #!/bin/bash làm shell thông dịch thay cho sh, vì trong Linux thật ra sh và bash là một. Tuy nhiên như đã nêu, trên các hệ Unix vẫn sử dụng shell sh làm chuẩn, vì vậy vẫn là một thói quen tốt cho lập trình viên nếu sử dụng shell sh. Khi tiếp cận với UNIX, ta sẽ cảm thấy quen và thân thuộc với shell này hơn. Nên chạy script trong một shell phụ (như gọi sh chẳng hạn), khi đó mọi thay đổi về môi trường mà script gây ra không ảnh hưởng đến môi trường làm việc chính.

Chỉ thị #! còn được dùng để gọi bất kỳ chương trình nào ta muốn chạy trước khi script tiếp theo được dịch. Lệnh *exit* bảo đảm rằng script sau khi thực thi sẽ trả về mã lỗi, đây là cách mà hầu hết các chương trình nên làm, mặc dù mã lỗi trả về ít khi được dùng đến trong trường hợp thực hiện tương tác trực tiếp từ dòng lệnh. Tuy nhiên, việc nhận biết mã trả về của một đoạn script sau khi thực thi lại thường rất có ích nếu bạn triệu gọi script từ trong một

script khác. Trong đoạn chương trình trên, lệnh *exit* sẽ trả về 0, cho biết script thực thi thành công và thoát khỏi shell gọi nó. Mặc dù khi đã lưu tập tin script với tên ***.sh***, nhưng UNIX và Linux không bắt buộc điều này. Hiếm khi Linux sử dụng phần đuôi mở

rộng của tập tin làm dấu hiệu nhận dạng, do đó tên tệp script có thể là tùy ý. Tuy vậy .sh vẫn là cách chúng ta nhận ngay ra một tập tin có thể là script của shell một cách nhanh chóng.

2.4.1.3 Thực thi script

Chúng ta vừa tạo ra tập tin script first.sh, nó có thể được gọi thực thi theo hai cách. Cách đơn giản nhất là gọi trình shell với tên tập tin script làm đối số. Ví dụ:

$ **/bin/sh** first.sh

Cách gọi trên là bình thường, nhưng vẫn quen thuộc hơn nếu ta có thể gọi first.sh ngay từ dòng lệnh, tương tự các lệnh Linux thông thường. Để làm được điều này, trước hết cần chuyển thuộc tính thực thi (x) cho tập tin script bằng lệnh *chmod* như sau:

$ **chmod +x** first.sh

Sau đó có thể triệu gọi script theo cách thứ hai tiện lợi hơn:

$ first.sh

Có thể lệnh trên không thực hiện thành công và ta sẽ nhận được thông báo lỗi 'command not found' (không tìm thấy lệnh).

Điều này xảy ra bởi vì biến môi trường PATH thường không chứa đường dẫn hay vị trí thư mục hiện hành. Để khắc phục, ta có thể thêm vào biến môi trường PATH đường dẫn thư mục hiện hành như sau:

$ PATH=$PATH: .

Nếu muốn Linux tự động nhớ thư mục hiện hành mỗi khi đăng nhập bạn có thể thêm lệnh PATH=$PATH : . vào cuối tệp **.bash_profile** (file được triệu gọi lúc hệ thống đăng nhập - tương tự autoexec.bat của DOS). Tuy nhiên cách ngắn gọn và đơn giản nhất mà ta vẫn thường làm là định rõ dấu thư mục hiện hành **./** ngay trên lệnh. Ví dụ:

$ **.** / first.sh

Lưu ý: Đối với tài khoản root, không nên thay đổi biến môi trường PATH (bằng cách thêm dấu chỉ định **.**) cho phép truy tìm thư mục hiện hành. Điều này không an toàn và dễ tạo ra lỗ hổng bảo mật. Ví dụ, một quản trị hệ đăng nhập dưới quyền root, triệu gọi chương trình của Linux mà họ tưởng ở thư mục quy định như /bin, nếu biến PATH cho phép tìm ở thư mục hiện hành thì rất có thể nhà quản trị thực thi chương trình của ai đó thay vì chương trình Linux ở /bin. Vì vậy, nên tạo thói quen đặt dấu **./** trước một tập tin để ám chỉ truy xuất ở thư mục hiện hành.

Một khi bạn tin rằng first.sh chạy tốt, có thể di chuyển nó đến thư mục khác thích hợp hơn thư mục hiện hành. Nếu lệnh script

chỉ dành cho mục đích riêng của bạn, bạn có thể tạo ra thư mục /bin trong thư mục chủ (home) mà nhà quản trị quy định cho người dùng, sau đó thêm đường dẫn này vào biến môi trường PATH. Nếu muốn script được gọi bởi người dùng khác, hãy đặt nó vào thư mục /usr/1ocal/bin. Thông thường, để cho phép một script

hay chương trình thực thi, cần được người quản trị cho phép. Nếu bạn là nhà quản trị, cũng cần cẩn thận xem xét các script do các người dùng khác (hacker chẳng hạn) đặt vào hệ thống. Ngôn ngữ script rất mạnh, nó có thể làm được hầu như là mọi chuyện, kể cả hủy diệt hệ thống!

Để ngăn script bị sửa đổi bởi người dùng khác, có thể sử dụng các lệnh thiết lập quyền (thường phải đăng nhập với tư cách root để làm công việc này):

```
#cp first.sh /usr/1ocal/bin
#chown root /usr/local/bin/first.sh
#chgrp root /usr/1ocal/bin/first.sh
#chmod u=rwx go=rx /usr/1ocal/bin/firsc.sh
```

Đoạn lệnh trên mang ý nghĩa: chuyển quyền sở hữu tập tin cho root, root được toàn quyền đọc sửa nội dung và thực thi tập tin, trong khi nhóm và những người dùng khác root chỉ được phép đọc và thực thi. Nên nhớ mặc dù bạn loại bỏ quyền ghi w trên tập

tin, UNIX và Linux vẫn cho phép bạn xoá tập tin này nếu thư mục chứa nó có quyền ghi w. Để an toàn, với tư cách là nhà quản trị, nên kiểm tra lại thư mục chứa script và bảo đảm rằng chỉ có root mới có quyền w trên thư mục chứa các tệp .sh.

2.4.2 Cú pháp ngôn ngữ shell

Chúng ta đã thấy cách viết lệnh và gọi thực thi tập tin scirpt. Phần tiếp theo này sẽ giúp bạn khám phá sức mạnh của ngôn ngữ lập trình shell. Trái với lập trình bằng trình biên dịch khó kiểm tra lỗi và nâng cấp, lập trình script cho phép bạn dễ dàng sửa đổi lệnh bằng ngôn ngữ văn bản. Nhiều đoạn script nhỏ có thể kết hợp lại thành một script lớn mạnh mẽ và rất hữu ích. Trong thế giới UNIX và Linux, đôi lúc khi gọi thực thi một chương trình bạn khó mà biết được chương trình được viết bằng script hay thực thi theo mã của chương trình nhị phân, bởi vì tốc độ thực thi và sự uyển chuyển của chúng gần như ngang nhau.

Trong phần này chúng ta sẽ học về:

- Biến: kiểu chuỗi, kiểu số, tham số và biến môi trường.
- Điều kiện: kiểm tra luận lý boolean bằng shell.
- Điều khiển chương trình: if, elif, for, while, until, case.

2.4.2.1 Sử dụng biến

Thường bạn không cần phải khai báo biến trước khi sử dụng. Thay vào đó biến sẽ được tự động tạo và khai báo khi lần đầu tiên tên biến xuất hiện, chẳng hạn như trong phép gán. Mặc định, tất cả các biến đều được khởi tạo và chứa giá trị kiểu chuỗi (string). Ngay cả khi dữ liệu mà bạn đưa vào biến là một con số thì nó cũng

được xem là định dạng chuỗi. Shell và một vài lệnh tiện ích sẽ tự động chuyển chuỗi thành số để thực hiện phép tính khi có yêu cầu. Tương tự như bản thân hệ điều hành và ngôn ngữ C, cú pháp của shell phân biệt chữ hoa, chữ thường: biến mang tên foo, Foo và FOO là ba biến khác nhau.

Bên trong các script của shell, bạn có thể lấy về nội dung của biến bằng cách dùng dấu $ đặt trước tên biến. Để hiển thị nội dung biến, bạn có thể dùng lệnh **echo**. Khi gán nội dung cho biến, bạn không cần phải sử dụng ký tự $. Ví dụ trên dòng lệnh, bạn có thể gán nội dung và hiển thị biến như sau:

```
xinchao=Hello
echo $xinchao
```

⇨ Kết quả in ra màn hình: Hello

```
xinchao="I am here"
echo $xinchao
```

⇨ Kết quả in ra màn hình: I am here

xinchao=12+l

echo $xinchao

⇨ Kết quả in ra màn hình: 12+1

Lưu ý: sau dấu = không được có khoảng trắng. Nếu gán nội dung chuỗi có khoảng trắng cho biến, cần bao bọc chuỗi bằng dấu " ".

Có thể sử dụng lệnh **read** để đọc dữ liệu do người dùng đưa vào và giữ lại trong biến để sử dụng. Ví dụ:

read yourname

nhập vào: XYZ

echo "Hello " $yourname

⇨ Kết quả in ra màn hình: Hello XYZ

Lệnh **read** kết thúc khi bạn nhấn phím Enter (tương tự scanf của C hay readln của Pascal).

2.4.2.2 Các ký tự đặc biệt (metacharacters của shell)

a. Chuyển hướng vào/ra

Một tiến trình Unix/Linux bao giờ cũng gắn liền với các đầu xử lý các dòng (stream) dữ liệu: đầu vào chuẩn (stdin hay 0), thường là từ bàn phím qua chức năng getty(); đầu ra chuẩn (stdout hay 1), thường là màn hình và cơ sở dữ liệu lỗi hệ thống (stderr hay 2). Tuy nhiên các hướng vào/ra có thể thay đổi được bởi các thông báo đặc biệt:

Ký hiệu	Ý nghĩa (… tượng trưng cho đích

	đổi hướng)
>	Đầu ra hướng tới …
>>	Nối vào nội dung của …
<	Lấy đầu vào từ < …
<< word	Đầu vào là ở đây …
2>	Đầu ra báo lỗi sẽ hướng vào …
2>>	Đầu ra báo lỗi hướng và ghi thêm vào …

Ví dụ:

$date > login.time

Lệnh date không kết xuất ra đầu ra chuẩn (stdout) mà ghi vào tệp login.time. *>login.time* không phải là thành phần của lệnh date, mà đơn giản mô tả tiến trình tạo và gởi kết xuất ở đâu (bình thường là màn hình). Nhìn theo cách xử lý thì như sau: cả cụm lệnh trên chứa hai phần: lệnh *date*, tức chương trình thực thi và thông điệp *(>login.time*) thông báo cho shell biết kết xuất lệnh sẽ được xử lý như thế nào (khác với mặc định, bản thân date cũng không biết chuyển kết xuất đi đâu, shell chọn mặc định).

Ví dụ:

$cat < file1

Bình thường cat nhận và hiển thị nội dung tệp có tên (là đối đầu vào). Với lệnh trên cat nhận nội dung từ file1 và kết xuất ra màn hình. Thực chất không khác gì khi gõ:

$cat file1

Hãy xem:

$cat < file1 > file2

Lệnh này thực hiện như thế nào? Theo trình tự sẽ như sau: cat nhận nội dung của file1 sau đó ghi vào tệp có tên file2, không đưa ra stdout như mặc định. Lệnh trên cho thấy ta có thể thay đổi đầu vào và đầu ra cho lệnh như thế nào. Những lệnh cho phép đổi đầu

ra/vào gọi chung là quá trình lọc (filter).

Ví dụ:

$cat file1 < file2

Lệnh này chỉ hiển thị nội dung của file1, không gì hơn. Tại sao? cat nhận đối đầu vào là tên tệp. Nếu không có đối, nó nhận từ stdin (bàn phím). Có đối thì chính là file1 và đầu ra là stdout. Trường hợp này gọi là *bỏ qua đổi hướng*. Cái gì ở đây là quan trọng? Đầu ra/vào của lệnh đã đổi hướng cũng không có nghĩa bảo đảm rằng sự đổi hướng sẽ được sử dụng. Một lần nữa cho thấy lệnh bản thân nó không hiểu rằng đã có sự đổi hướng và có lệnh chấp nhận đổi hướng vào/ra, nhưng không phải tất cả. Ví dụ:

$date < login.time

date khác cat, nó không kiểm tra đầu vào, nó biết phải tìm đầu vào ở đâu. Đổi hướng ở đây không có tác dụng.

Ví dụ

$cat < badfile 2> errfile

Thông thường các lỗi hệ thống quản lý đều ở stderr và sẽ in ra màn hình. Tuy nhiên có thể chuyển hướng báo lỗi, ví dụ vào một tệp (chẳng hạn logfile) mà không đưa ra màn hình. Ví dụ trên là như vậy. Ta biết stderr là tệp có mô tả tệp = 2, do vậy 2>errfile có nghĩa đổi đầu ra của stderr vào một tệp, tức ghi báo lỗi vào tệp xác

định.

Những gì vừa đề cập ở trên đều có tác động trên tệp vào/ra. Ta cũng có cách xử lý ngay trong một dòng của tệp, gọi là *đổi hướng trong dòng (in-line Redirection)*. Loại này bao gồm hai phần: đổi hướng (<<) và *dấu hiệu đánh dấu* (là bất kỳ ký tự nào) của dòng dữ liệu vào.

Ví dụ:

$cat << EOF # *dấu hiệu đánh dấu* chọn ở đây là EOF

> Xin chao

> …

> EOF (và gõ ENTER)

Ngay lập tức trên màn hình sẽ là:

Xin chao

…

Ở đây EOF là *dấu hiệu đánh dấu*, hay còn gọi là thẻ bài (token). Điều đáng lưu ý là:

1. Cùng một dòng dữ liệu, phải được kết thúc.
2. Token phải đứng ngay ở đầu dòng.

Ví dụ trên có một chú ý: dấu > gọi là dấu nhắc thứ cấp, nó cho biết dòng lệnh đưa vào dài hơn là 1 dòng và cũng là dấu hiệu shell thông báo nó nhận nhiều thông tin ở đầu vào.

b. Các ký tự đặc biệt kiểm soát tiến trình

1. & (Ampersand): đặt một tiến trình (chương trình) vào chế độ chạy nền (background process). Bản thân Unix không có khái niệm gì về tiến trình chạy nền hay tiến trình tương tác (foreground), mà shell điều khiển việc chạy các tiến trình. Với & chương trình sẽ tự chạy và shell quay ngay về tương tác với người dùng, trả lại dấu nhắc ngay. Tiến trình nền có nhiều cách để kiểm soát.

Ví dụ:

$sort huge.file > sorted.file & // huge.file là một file thật lớn

$

Bằng lệnh ps sẽ thấy lệnh sort đang chạy kèm với số ID của tiến trình đó.

Bằng lệnh

$ jobs

[1]

sẽ thấy số hiệu của lệnh đang chạy ngầm.

Để kết thúc thực thi, dùng

$ kill 1234 #1234 là ID của tiến trình sort

Để quay lại chế độ tương tác:

$ fg 1

2. **Ngoặc đơn (;)** Dùng để nhóm một số lệnh lại, phân cách bởi **;**

Ví dụ:

$ (date ; who) > system.status

$ cat system.status

(Hãy xem kết xuất trên màn hình)

3. Dấu nháy ` ` (backquotes) (là dấu ở phím đi cùng với dấu ~)

Hay còn gọi là dấu thay thế. Bất kỳ lệnh nào xuất hiện bên trong dấu nháy sẽ được thực hiện trước và kết quả của lệnh đó sẽ thay thế đầu ra chuẩn (stdout) trước khi lệnh trong dòng lệnh thực hiện.

Ví dụ:

$ echo Logged in `**datè** > login.time

sẽ ra lệnh cho shell đi thực hiện **date** trước tiên, trước khi thực hiện các phần khác còn lại của dòng lệnh, tức sau đó mới thực hiện lệnh **echo.** Vậy cách diễn đạt dòng lệnh trên như sau:

echo Logged in Fri May 12:52:25 UTC 2004 >

login.time

Tức là: 1. thực hiện date với kết quả *Fri May 12:52:25 UTC 2004* không hiện ra stdout (màn hình), nhưng sẽ là đầu vào của echo.

2. sau đó lệnh echo sẽ echo *Logged in Fri May 12:52:25 UTC 2004*, nhưng không đưa ra màn hình (stdout) mà đổi hướng vào tệp login.time.

Nếu gõ $ cat login.time, ta có kết xuất từ tệp này ra màn hình:

Logged in Fri May 12:52:25 UTC 2004

4. Ống dẫn (Pipelines)

Shell cho phép kết quả thực thi một lệnh (đầu ra của lệnh), kết hợp trực tiếp (nối vào) đầu vào của một lệnh khác, mà không cần xử lý trung gian (lưu lại trước tại tệp trung gian).

Ví dụ:

$who | ls –l

Đầu ra (stdout) của *who* (đáng lẽ sẽ ra màn hình) sẽ là đầu vào (stdin) của *ls –l*.

Ví dụ:

$ (date ; who) | ls -l

Tóm tắt:

cmd & *đặt lệnh cmd chạy nền (background)* ***cmd1 ; cmd2*** *chạy cmd1 trước, sau đó chạy cmd2* ***(cmd)*** *thực hiện cmd trong một shell con (subshell)* ***`cmd`*** *đầu ra của cmd sẽ thay cho đầu ra của lệnh trong dòng lệnh*

cmd1 | cmd2 *nối đầu ra của cmd1 vào đầu vào của cmd2*

c. Dấu bọc chuỗi (quoting)

Shell có một tập các ký tự đặc biệt mà hiệu lực của chúng là để vô hiệu hóa ý nghĩa của các ký tự đặc biệt khác. Khi một ký tự đặc biệt bị giải trừ hiệu lực, ta gọi ký tự đó là bị *quoted*.

Trước khi tiếp tục chúng ta cần hiểu một số tính chất của dấu bọc chuỗi mà shell quy định. Thông thường, tham số dòng lệnh thường cách nhau bằng khoảng trắng. Khoảng trắng có thể là ký tự spacebar, tab hoặc ký tự xuống dòng. Trường hợp muốn tham số

của mình chứa được cả khoảng trắng, cần phải bọc chuỗi bằng dấu nháy đơn ' hoặc nháy kép " .

Dấu nháy kép được dùng trong trường hợp biến chuỗi của bạn có khoảng trắng. Tuy nhiên với dấu nháy kép, ký hiệu biến $ vẫn có hiệu lực. Nội dung của biến sẽ được thay thế trong chuỗi. Dấu nháy đơn sẽ có hiệu lực mạnh hơn. Nếu tên biến có ký tự $ đặt trong chuỗi có dấu nháy đơn, nó sẽ bị vô hiệu hóa. Có thể dùng dấu

\ để hiển thị ký tự đặc biệt $ trong chuỗi.

Backslash (\ )

Ví dụ:

$cat file1&2: lệnh này gây ra nhiều lỗi, bởi có sự hiểu nhầm & trong khi nó đơn giản là thành phần của tên tệp (file1&2). Để được như ý, cần sử dụng thêm \

$cat file1\&2 sẽ cho kết quả như mong muốn: đưa nội dung của tệp có tên file1&2 ra màn hình. Dấu \ đã giải trừ ý nghĩa đặc biệt của &.

Các ví dụ khác về " " hay ' ':

Ví dụ 2-1: *variables.sh*

```
#!/bin/sh
myvar="Hi there"
```



```
echo $myvar
echo "message : $myvar"
echo 'message : $myvar'
echo "messgae : \$myvar"

echo Enter some text
read myvar
```

```
echo '$myvar' now equals $myvar

exit 0
```

Kết xuất khi thực thi script:

```
Hi there
message : Hi there
message : $myvar
message : $myvar
Enter some text
Hello World
$myvar now equals Hello World
```

Cách chương trình làm việc:

Biến myvar được tạo ra và khởi gán giá trị là chuỗi Hi there. Nội dung của biến sau đó được hiển thị bằng lệnh echo trong các trường hợp bọc chuỗi bằng nháy kép, nháy đơn và dấu hiển thị ký tự đặc biệt. Từ các kết quả trên, ta thấy nếu muốn thay thế nội dung biến trong một chuỗi, cần bọc chuỗi bằng nháy kép. Nếu muốn hiển thị toàn bộ nội dung chuỗi, hãy dùng nháy đơn.

2.4.2.3 Biến môi trường (environment variable)

Khi trình shell khởi động nó cung cấp sẵn một số biến được khai báo và gán trị mặc định. Chúng được gọi là các biến môi trường. Các biến này thường được viết hoa để phân biệt với biến do người dùng tự định nghĩa (thường là ký tự không hoa). Nội dung các biến này thường tùy vào thiết lập của hệ thống và người quản trị cho phép người dùng hệ thống sử dụng. Danh sách của các biến môi trường là khá nhiều, nhưng nhìn chung nên nhớ một số biến môi trường chủ yếu sau:

Biến môi trường	*Ý nghĩa*
$HOME	Chứa nội dung của thư mục chủ. (Thư mục đầu tiên khi người dùng đăng nhập)
$PATH	Chứa danh sách các đường dẫn (phân cách bằng dấu hai chấm :). Linux thường tìm các chương trình cần thi hành trong biến

	$PATH.

$PS1	Dấu nhắc (prompt) hiển thị trên dòng lệnh. Thông thường là $
cho	User không phải *root*.
$SP2	Dấu nhắc thứ cấp, thông báo người dùng nhập thêm thông tin trước khi lệnh thực hiện. Thường là dấu >.
$IFS	Dấu phân cách các trường trong danh sách chuỗi. Biến này chứa danh sách các ký tự mà shell dùng tách chuỗi (thường là tham số trên dòng lệnh). Ví dụ $IFS thường chứa ký tự Tab, ký tự trắng hoặc ký tự xuống hàng.
$0	Chứa tên chương trình gọi trên dòng lệnh.
$#	Số tham số truyền trên dòng lệnh
$$	Mã tiến trình (process id) của shell script thực thi. Bởi số process id của tiến trình là duy nhất trên toàn hệ thống vào lúc script thực thi nên thường các lệnh trong script dùng con số này để tạo các tên file tạm. Ví dụ /tmp/tmpfile_$$.

Mỗi môi trường mà user đăng nhập chứa một số danh sách biến môi trường dùng cho mục đích riêng. Có thể xem danh sách này bằng lệnh **env**. Để tạo một biến môi trường mới, có thể dùng lệnh **export** của shell (một số shell sử dụng lệnh **setenv**).

2.4.2.4 Biến tham số (parameter variable)

Nếu cần tiếp nhận tham số trên dòng lệnh để xử lý, có thể dùng thêm các biến môi trường sau:

Biến tham số	*Ý nghĩa*
$1, $2, $3 . . .	Vị trí và nội dung của các tham số trên dòng lệnh theo thứ tự từ trái sang phải.
S*	Danh sách của tất cả các tham số trên dòng lệnh. Chúng được lưu trong một chuỗi duy nhất phân cách bằng ký tự đầu tiên quy định trong biến $IFS.
$@	Danh sách các tham số được chuyển thành chuỗi. Không sử dụng dấu phân cách của biến IFS.

Để hiểu rõ sự khác biệt của biến $ * và $@, hãy xem ví dụ sau:

$IFS= "A"

$set foo bar bam

$echo “$@”

foo bar bam

$echo "$*"

fooAbarAbam

$unset IFS

$echo "$*"

foo bar bam

Ta nhận thấy, lệnh set tiếp nhận 3 tham số trên dòng lệnh là foo bar bam. Chúng ảnh hưởng đến biến môi trường $* và $@. Khi IFS được quy định là ký tự A , $* chứa danh shell các tham số phân cách bằng ký tự A. Khi đặt IFS về NULL bằng lệnh unset, biến $* trả về danh shell thuần tuý của các tham số tương tự biến $@.

Biến $# sẽ chứa số tham số của lệnh, trong trường hợp trên ta có:

$echo " $# "

sẽ in ra kết quả là 3.

Khi lệnh không có tham số thì $0 chính là tên lệnh còn $# trả

về giá trị 0.

Đoạn chương trình mẫu sau sẽ minh họa một số cách đơn giản xử lý và truy xuất biến môi trường.

Ví dụ 2-2: ***try_variables.sh***


```
#!/bin/sh

salutation="Hello"
echo $salutation
echo "The program $0 is now running"
echo "The second parameter was $2"
echo "The first parameter was $1"
echo "The parameter list was $*"
echo "The user's home directory is $HOME"

echo "Please enter a new greeting"
read salutation

echo $salutation
echo "The script is now complete"

exit 0
```

Lưu tên tập là try-variables.sh, đổi thuộc tính thực thi x cho tập tin bằng lệnh: **$chmod +x try_variablebles.sh** Khi chạy try-variables.sh từ dòng lệnh, bạn sẽ nhận được kết quả kết xuất như sau:

$./try_variables.sh foo bar baz

Hello

The program . /try_vanables.sh is now running

The second parameter was bar

The first parameter was foo

The parameter list was foo bar baz

The user's home directory is /home/xyz #tên người dùng login là xyz

please enter a new greeting

Xin chao!

Xin chao!

The script is now complete

2.4.3 Cấu trúc điều kiện

Nền tảng cơ bản trong tất cả ngôn ngữ lập trình, đó là khả năng kiểm tra điều kiện và đưa ra quyết định rẽ nhánh thích hợp tùy theo điều kiện đúng hay sai. Trước khi tìm hiểu cấu trúc điều

khiển của ngôn ngữ script, ta hãy xem qua cách kiểm tra điều kiện.

Một script của shell có thể kiểm tra mã lỗi trả về của bất kỳ lệnh nào có khả năng triệu gọi từ dòng lệnh, bao gồm cả những tập tin lệnh script khác. Đó là lý do tại sao chúng ta thường sử dụng lệnh exit ở cuối mỗi script khi kết thúc.

2.4.3.1 Lệnh *test* hoặc *[ ]*

Thực tế, các script sử dụng lệnh [] hoặc test để kiểm tra điều kiện boolean rất thường xuyên. Trong hầu hết các hệ thống UNIX và Linux thì [ ] và test có ý nghĩa tương tự nhau, thường lệnh [ ] được dùng nhiều hơn. Lệnh [ ] trông đơn giản, dễ hiểu và rất gần với các ngữ lập trình khác.

Trong một số shell của Unix, lệnh test có khả năng là một lời triệu gọi đến chương trình bên ngoài chứ không phải lệnh nội tại của ngôn ngữ script. Bởi vì test ít khi được dùng và hầu hết các lập trình viên có thói quen thường tạo các chương trình với tên test, cho nên khi thử lệnh test không thành công bên trong script, thì hãy xem lại, đây đó bên trong hệ thống có một chương trình tên là test khác biệt nào đó đang tồn tại. Hãy thử dùng lệnh **which test**, lệnh này sẽ trả về cho bạn đường dẫn đến thư mục test được triệu gọi. Chẳng hạn /bin/test hay /usr/bin/test.

Dưới đây là cách sử dụng lệnh test đơn giản nhất. Dùng lệnh test để kiểm tra xem file mang tên hello.c có tồn tại trong hệ thống hay không. Lệnh test trong trường hợp này có cú pháp như sau: *test -f <mename>* , trong script ta có thể viết lệnh theo cách sau:

```
if test -f hello.c
    then
        …
    fi
```


Cũng có thể sử dụng [ ] để thay thế test

```
if [ -f hello.c ]
then
    …
fi
```

Mã lỗi và giá trị trả về của lệnh mà test kiểm tra sẽ quyết định điều kiện kiểm tra là đúng hay sai.

Lưu ý, phải đặt khoảng trắng giữa lệnh [ ] và biểu thức kiểm tra. Để dễ nhớ, xem [ ] tương đương với lệnh test, và dĩ nhiên giữa một lệnh và tham số truyền cho lệnh phải phân cách nhau bằng khoảng trắng để trình biên dịch có thể hiểu.

Nếu thích đặt từ khóa **then** chung một dòng với lệnh **if**, bạn phải phân cách **then** bằng dấu chấm phẩy (;) như sau: **if [-f hello.c**

] ; then

. . .

fi

2.4.3.2 Các kiểu điều kiện kiểm tra

Điều kiện mà lệnh test cho phép kiểm tra có thể rơi vào một trong 3 kiểu sau:

So sánh chuỗi

So sánh Kết quả

stringl = string2 *true* nếu 2 chuỗi bằng nhau (chính xác từng ký tự)

tring1 != string2 *true* nếu 2 chuỗi không bằng nhau -n string1 *true* nếu string1 không rỗng

-z stringl *true* nếu string1 rỗng (chuỗi null)

So sánh toán học

So sánh	Kết quả
expression1 –eq expression2	*true* nếu hai biểu thức bằng nhau
expression1 -ne expression2	*true* nếu hai biểu thức không bằng nhau
expression1 -gt expression2	*true* nếu biểu thức expression1 lớn hơn expression2

expreesion1 -ge expression2	*true* nếu biểu thức expression1 lớn hơn hay bằng expression2
expression1 -lt expression2	*true* nếu biểu thức expression1 nhỏ hơn expression2
expression1 -le exprebbion2	*true* nếu biểu thức expression1 nhỏ hơn hay bằng expression2
!expression	*true* nếu biểu thức expression là *false* (toán tử *not*)

Kiểm tra điều kiện trên tập tin

-d file *true* nếu file là thư mục

-e file *true* nếu file tồn tại trên đĩa

-f file *true* nếu file là tập tin thông thường

-g file *true* nếu set-group-id được thiết lập trên file -r file *true* nếu file cho phép đọc

-s file *true* nếu kích thước file khác 0

-u file *true* nếu set-ser-id được áp đặt trên file

-w file *true* nếu file cho phép ghi

-x file *true* nếu file được phép thực thi

Lưu ý về mặt lịch sử thì tùy chọn -e không khả chuyển

(portable) và -f thường được sử dụng thay thế.

Câu hỏi có thể đặt ra là set-group-id và set-ser-id (còn được gọi là set-gid và set-uid) mang ý nghĩa gì. Set-uid cho phép chương trình quyền của chủ thể sở hữu (owner) thay vì quyền của user thông thường. Tương tự set-gid cho phép chương trình quyền của nhóm.

Tất cả các điều kiện kiểm tra tập tin đều yêu cầu file phải tồn tại trước đó (có nghĩa là lệnh *test -f filename* phải được gọi trước). Lệnh test hay [ ] còn có thêm nhiều điều kiện kiểm tra khác nữa, nhưng hiện thời ta chưa dùng đến. Có thể tham khảo chi tiết test bằng lệnh *help test* từ dấu nhắc của hệ thống.

2.4.4 Cấu trúc điều khiển

Shell cung cấp cấu trúc lệnh điều khiển rất giống với các ngôn ngữ lập trình khác, đó là *if, elif, for, while, until, case*. Đối với một vài cấu trúc lệnh (ví dụ như case), shell đưa ra cách xử lý uyển chuyển và mạnh mẽ hơn. Những cấu trúc điều khiển khác nếu có thay đổi chỉ là những thay đổi nhỏ không đáng kể.

*Trong các phần sau **statements** được hiểu là biểu thức lệnh (có thể bao gồm một tập hợp các lệnh) sẽ được thực thi khi điều kiện kiểm tra **condition** được thoả mãn.*

2.4.4.1 Lệnh *if*

Lệnh **if** tuy đơn giản nhưng được sử dụng nhiều nhất. **if** kiểm tra điều kiện đúng hoặc sai để thực thi biểu thức thích hợp *if* ***condition***

then

statements

else

statements

Ví dụ, đoạn script sau sử dụng if tùy vào câu trả lời của bạn mà đưa ra lời chào thích hợp.

Ví dụ 2-3: *if_control.sh*

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday
#chú ý khoảng trắng trước sau [ và trước ]
if [ $timeofday = "yes" ]; then
echo "Good morning"
else
echo "Good afternoon"
fi
exit 0
```

Kết quả kết xuất của script

$./ if_control.sh

Is it mornining ? Please answer yes or no

yes

Good morning

$

Ở ví dụ trên chúng ta đã sử dụng cú pháp [ ] để kiểm tra điều kiện thay cho lệnh test. Biểu thức kiểm tra xem nội dung của biến $timeofday có khớp với chuỗi "yes" hay không. Nếu có thì lệnh echo cho in ra chuỗi “Good morning”, nếu không (mệnh đề else) in ra chuỗi “Good afternoon".

Shell không đòi hỏi phải canh lề hay thụt đầu dòng cho từng lệnh. Chúng ta canh lề để cú pháp được rõ ràng, dễ đọc. Mặc dù

*vậy sau này bạn sẽ thấy ngôn ngữ của chương trình **make** sẽ yêu cầu canh lề và xem đó là yêu cầu để nhận dạng lệnh.*

2.4.4.2 Lệnh *elif*

Tuy nhiên, có rất nhiều vấn đề phát sinh với đoạn trình script trên. Tất cả câu trả lời khác với “yes” đều có nghĩa là “no”. Chúng ta có thể khắc phục điều này bằng cách dùng cấu trúc điều khiển

elif. Mệnh đề này cho phép kiểm tra điều kiện lần thứ hai bên trong else. Script dưới đây được sửa đổi hoàn chỉnh hơn, bao gồm cả in ra thông báo lỗi nếu người dùng không nhập đúng câu trả lời "yes" hoặc "no".

Ví dụ 2-4: *elif_control.sh*

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday
if [ $timeofday = "yes" ]; then
echo "Good morning"
elif [ $timeofday = "no" ]; then
        echo "Good afternoon"
else
        echo "Sorry, $timeofday not recognized. Enter yes or  no"
        exit 1
```



```
fi
exit 0
```

Cũng đơn giản như ví dụ 2-3, nhưng chúng ta sử dụng thêm *elif* để kiểm tra trường hợp người dùng không nhập "no". Thông báo lỗi được in ra và mã lỗi trả về bằng lệnh exit là 1. Trường hợp hoặc "yes" hoặc "no" được nhập vào, mã lỗi trả về sẽ là 0.

2.4.4.3 Vấn đề phát sinh với các biến

elif trong ví dụ trên khắc phục được hầu hết các điều kiện nhập liệu và yêu cầu người dùng nhập đúng trước khi ra quyết định thực thi tiếp theo. Mặc dù vậy, có một vấn đề khá tinh tế còn lại, nếu chạy lại *elif_control.sh* nói trên, nhưng thay vì nhập vào một chuỗi nào đó, hãy gõ Enter (tạo chuỗi rỗng cho biến $timeofday), sẽ nhận được thông báo lỗi của shell như sau:

[: = : unary operator expected

Điều gì xảy ra? Lỗi phát sinh ngay mệnh đề *if* đầu tiên. Khi biến *timeofday* được kiểm tra nó cho trị là rỗng và do đó lệnh *if* sẽ được shell diễn dịch thành:

if [= "yes"]

và dĩ nhiên shell không hiểu phải so sánh chuỗi "yes" với cái gì. Để tránh lỗi này cần bọc nội dung biến bằng dấu bao chuỗi như sau:

if ["$timeofdayl" ="yes"]

Trong trường hợp này nếu chuỗi nhập vào là null, shell sẽ diễn dịch biểu thức thành:

if [" "="yes "]

và script sẽ chạy tốt. Elif_control.sh có thể sửa lại hoàn chỉnh hơn như sau:

Ví dụ 2-5: ***elif_control2.sh***

```
#!/bin/sh
echo -n "Is it morning? Please answer yes or no: "
read timeofday
if [ "$timeofday" = "yes" ]; then
 echo "Good morning"
elif [ "$timeofday" = "no" ]; then
 echo "Good afternoon"
else
 echo "Sorry, $timeofday not recognized. Enter yes or no"
exit 1
 fi
 exit 0
```

Hãy kiểm tra lại elif_controi2.sh bằng cách chỉ nhấn Enter khi shell đưa ra câu hỏi. Script giờ đây chạy rất bảo đảm và chuẩn.

Lệnh ***echo*** *thường xuống hàng sau khi đưa ra thông báo. Có thể dùng lệnh* ***printf*** *thay cho* ***echo****. Một vài shell cho phép lệnh* ***echo -e*** *trên một dòng, nhưng chúng không phải là phổ biến để bạn sử dụng.*

2.4.4.4 Lệnh for

Sử dụng **for** để lặp lại một số lần với các giá trị xác định. Phạm vi lặp có thể nằm trong một tập hợp chuỗi chỉ định tường minh bởi chương trình hay là kết quả trả về từ một biến hoặc biểu thức khác.

Cú pháp:

for *variable* **in** *values*

do

statemens

done

Ví dụ sau sẽ in ra các giá trị chuỗi trong tập hợp:

Ví dụ 2-6: ***for_loop.sh***

```
#!/bin/sh
for foo in bar fud 13
do
 echo $foo
done
```



```
exit 0
```

Kết quả kết xuất sẽ là

```
 $./ for_loop.sh
bar
```

fud

13

foo là một biến dùng trong vòng lặp for để duyệt tập hợp gồm 3 phần tử (cách nhau bằng khoảng trắng). Mặc định shell xem tất cả các giá trị gán cho biến là kiểu chuỗi cho nên 13 ở đây được xem là chuỗi tương tự như chuỗi bar và fud.

Điều gì sẽ xảy ra nếu bạn thay thế lệnh for foo in bar fud 13 thành for foo in "bar fud 13". Hãy nhớ lại, dấu nháy kép cho phép coi tất cả nội dung bên trong nháy kép là một biến chuỗi duy nhất. Kết quả kết xuất nếu sử dụng dấu nháy kép, lệnh echo chỉ được gọi một lần để in ra chuỗi "bar fud 13".

for thường dùng để duyệt qua danh sách tên các tập tin. Bằng cách dùng ký tự đại diện * (wildcard) ở ví dụ first.sh, ta đã thấy cách **for** tìm kiếm tập tin kết hợp với lệnh **grep**. Ví dụ sau đây cho thấy việc mở rộng biến thành tập hợp sử dụng trong lệnh **for**. Giả sử bạn muốn in ra tất cả các tệp *.sh có ký tự đầu tiên là f.

Ví dụ 2-7: ***for_loop2.sh***


```
#!/bin/sh
for file in $(ls f*.sh); do
        lpr $file
done
```

Ví dụ trên đây cũng cho thấy cách sử dụng cú pháp *$(command)*. Danh sách các phần tử trong lệnh **for** được cung cấp bởi kết quả trả về của lệnh *ls f** và được bọc trong cặp lệnh mở rộng biến $ ( ) .

Biến mở rộng nằm trong dấu bao **$ (command)** chỉ được xác đinh khi lệnh command thực thi xong.

2.4.4.5 Lệnh while

Mặc dù lệnh **for** cho phép lặp trong một tập hợp giá trị biết trước, nhưng trong trường hợp một tập hợp lớn hoặc số lần lặp không biết trước, thì **for** không thích hợp. Ví dụ:

```
for foo in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
do
    echo $foo
done
```

Lệnh **while** cho phép thực hiện lặp vô hạn khi điều kiện kiểm tra vẫn còn đúng.

Cú pháp của while như sau:

while *condition* **do**

    statements

done

Ví dụ sau sẽ cho thấy cách while liên tục kiểm tra mật khẩu

(password) của người dùng cho đến khi đúng bằng chuỗi secret mới chấp nhận.

Ví dụ 2-8: ***password.sh***

```
#!/bin/sh

echo "Enter password"
read trythis
while [ "$trythis" != "secret" ]; do
echo "Sorry, try again"
read trythis
done
exit 0
```

Kết quả của script

```
$./password.sh
Enter password:
abc
Sorry, try again
secret #gõ đúng
$
```


Mặc dù để password hiển thị khi nhập liệu rõ ràng là không thích hợp, nhưng ở đây ta chủ yếu minh họa lệnh **while**. Lệnh **whil**e liên tục kiểm tra nội dung biến $trythis, yêu cầu nhập lại dữ

liệu bằng lệnh read một khi $trythis vẫn chưa bằng với chuỗi "secret".

Bằng cách sử dụng biến đếm và biểu thức so sánh toán học, **while** hoàn toàn có thể thay thế **for** trong trường hợp tập dữ liệu lớn. Hãy theo dõi ví dụ sau:

Ví dụ 2-9: ***while_for.sh***

```
#!/bin/sh
foo=1

while [ "$foo" -le 16 ]
do
        echo "Here $foo"
        foo=$(($foo+1))
done
exit 0
```

Lưu ý, cú pháp ***$(( ))*** *do shell "ksh" khởi xướng. Cú pháp này dùng để đánh giá và ước lượng một biểu thức. Với các shell cũ khác có thể thay thế cú pháp này bằng lệnh* ***expr****. Tuy nhiên*

expr *không hiệu quả. Bất cứ khi nào, nếu có thể hãy nên dùng* ***$ (( ))*** *thay cho* ***expr****.*

Script *while_for.sh* sử dụng lệnh [ ] để kiểm tra giá trị của biến $foo vẫn còn nhỏ hơn hay bằng 16 hay không. Nếu còn, lệnh lặp **while** sẽ in ra tổng cộng dồn của biến $foo.

2.4.4.6 Lệnh until

Cú pháp của lệnh **until** như sau:

until *condition*

do

statements

done

Lệnh **until** tương tự lệnh **while** nhưng điều kiện kiểm tra bị đảo ngược lại. Vòng lặp sẽ bị dừng nếu điều kiện kiểm tra là đúng. Ví dụ sau sẽ sử dụng lệnh **until** để chờ một user nào đó đăng nhập:

Ví dụ 2-10: *until_user.sh*

```
#!/bin/sh
echo "Locate for user ..."
until who | grep "$1" > /dev/null
do
        sleep 60
done
echo -e \\a
echo "***** $1 has just logged in *****"
```



```
exit 0
```

Để thử lệnh này, nếu chạy ngoài màn hình console, hãy dùng hai màn hình ảo (Alt+F1 và Alt+F2), một màn hình dùng chạy script *until_user.sh*, màn hình kia dùng đăng nhập với tên user muốn kiểm tra. Nếu trong chế độ đồ họa, bạn có thể mở hai cửa sổ terminal và sẽ dễ hình dung hơn. Hãy chạy until_user.sh từ một màn hình như sau:

```
$./until_user.sh xyz
Locate for user . . .
```

Script sẽ rơi vào vòng lặp chờ user tên là xyz dăng nhập. Hãy nhập từ một màn hình khác (với user tên là xyz), ta sẽ thấy màn hình đầu tiên đưa ra thông báo cho thấy vòng lặp until chấm dứt.

```
* * * * *
xyz has just logged in * * * * *
```

Cách chương trình làm việc:

Lệnh **who** lọc danh sách các user đăng nhập vào hệ thống, chuyển danh sách này cho **grep** bằng cơ chế đường ống (|). Lệnh **grep** lọc ra tên user theo biến môi trường $1 hiện có nội dung là chuỗi *xyz*. Một khi lệnh **grep** lọc ra được dữ liệu, nó sẽ chuyển ra vùng tập tin rỗng /dev/null và trả lại giá trị *null*, lệnh **until** kết thúc.

2.4.4.7 Lệnh *case*

Lệnh **case** có cách sử dụng hơi phức tạp hơn các lệnh đã học. Cú pháp của lệnh **case** như sau:

case variable **in**

pattern [| partten] . . .) *statements;;*

pattern [| partten] . . .) *statements;;*

. . .

esac

Mặc dù mới nhìn hơi khó hiểu, nhưng lệnh **case** rất linh động. **case** cho phép thực hiện so khớp nội dung của biến với một chuỗi mẫu *pattern* nào đó. Khi một mẫu được so khớp, thì (lệnh) *statement* tương ứng sẽ được thực hiện. Hãy lưu ý đặt hai dấu chấm phẩy **;;** phía sau mỗi mệnh đề so khớp *pattern*, shell dùng dấu hiệu này để nhận dạng mẫu *pattern* so khớp tiếp theo mà biến cần thực hiện.

Việc cho phép so khớp nhiều mẫu khác nhau làm **case** trở nên thích hợp cho việc kiểm tra nhập liệu của người dùng. Chúng ta hãy xem lại ví dụ 2-4 với phiên bản viết bằng **case** như sau:

Ví dụ 2-11: ***case1.sh***

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
```

```
read timeofday

case "$timeofday" in
        "yes") echo "Good Morning";;
        "no" ) echo "Good Afternoon";;
        "y" ) echo "Good Morning";;
        "n" ) echo "Good Afternoon";;
            * ) echo "Sorry, answer not recognised";;
esac
exit 0
```

Cách thực hiện: Sau khi người dùng nhập vào câu trả lời, lệnh case sẽ lấy nội dung của biến $timeofday so sánh với từng chuỗi. Khi gặp chuỗi thích hợp nó sẽ thực thi lệnh đằng sau dấu **)** và kết thúc (không tiếp tục so khớp với các mẫu khác). Ký tự đại diện * cho phép so khớp với mọi loại chuỗi. * thường được xem như trường hợp so sánh đúng cuối cùng nếu các mẫu so sánh trước đó thất bại. Bạn có thể xem * là mệnh đề *default* trong lệnh **switch** của C hay **case … else** của Pascal.

Việc so sánh thường thực hiện từ mẫu thứ nhất trở xuống cho nên bạn đừng bao giờ đặt * đầu tiên, bởi vì như thế bất kỳ chuỗi nào cũng đều thỏa mãn **case**. Hãy đặt những mẫu dễ xảy ra nhất lên đầu tiên, tiếp theo là các mẫu có tần số xuất hiện thấp. Sau

cùng mới đặt mẫu * để xử lý mọi trường hợp còn lại. Nếu muốn có thể dùng mẫu * đặt xen giữa các mẫu khác để theo dõi (debug) lỗi của chương trình (như in ra nội dung của biến trong lệnh case chẳng hạn).

Lệnh **case** trong ví dụ trên rõ ràng là sáng sủa hơn chương trình sử dụng **if**. Tuy nhiên có thể kết hợp chung các mẫu so khớp với nhau khiến cho **case** ngắn gọn hơn như sau:

Ví dụ 2-12: ***case2.sh***

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
        "yes" | "y" | "Yes" | "YES" ) echo "Good Morning";;
        "n*" | "N*" ) echo "Good Afternoon";;
        * ) echo "Sorry, answer not  recognised";;
esac
exit 0
```

Ở script trên sử dụng nhiều mẫu so khớp trên một dòng so sánh của lệnh case. Các mẫu này có ý nghĩa tương tự nhau và yêu cầu thực thi cùng một lệnh nếu điều kiện đúng xảy ra. Cách viết

này thực tế thường dùng và dễ đọc hơn cách viết thứ nhất. Mặc dù vậy, hãy thử tìm hiểu **case** ở một ví dụ sau cùng này. **case** sử dụng lệnh **exit** để trả về mã lỗi cho từng trường hợp so sánh mẫu đồng thời **case** sử dụng cách so sánh tắt bằng ký tự đại diện.

Ví dụ 2-13 ***case3.sh***

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
        "yes" | "y" | "Yes" | "YES" )
                echo "Good Morning"
                 echo "Up bright and early this morning?"
;;
        [nN]* )
                echo "Good Afternoon"
                ;;
        * )
                echo "Sorry, answer not recognised"
                echo "Please answer yes or no"
                exit 1
                ;;
esac
```

```
exit 0
```

Cánh thực hiện: Trong trường hợp 'no' ta dùng ký tự đại diện * thay thế cho tất cả ký tự sau n và N. Điều này có nghĩa là nx hay Nu ... đều có nghĩa là 'no'. Ở ví dụ trên ta đã thấy cách đặt nhiều lệnh trong cùng một trường hợp so khớp. **exit 1** cho biết người dùng không chọn yes và no. **exit 0** cho biết người dùng đã chọn yes, no theo yêu cầu. :

*Có thể không cần đặt **;;** ở mẫu so khớp cuối cùng trong lệnh **case** (phía trước **esac**), vì không còn mẫu so khớp nào cần thực hiện nữa. Không như C yêu cầu phải đặt lệnh break ở mỗi mệnh đề **case**, shell không đòi hỏi điều này, nó biết tự động chấm dứt khi lệnh case tương ứng đã tìm được mẫu thoả mãn.*

Để làm case trở nên mạnh mẽ và so sánh được nhiều trường hợp hơn, có thể giới hạn các ký tự so sánh theo cách sau: [yy] | [Yy] [Ee] [Ss]. Khi đó y,Y hay YES, YeS, ... đều được xem là yes. Cách này đúng hơn là dùng ký tự thay thế toàn bộ * trong trường hợp [nN]*.

2.5 Bài tập ôn tập

1. Chạy tất cả các đoạn lệnh ví dụ ở phần 2.4. Chụp hình kết

quả chạy các file script và lưu vào báo cáo.

2. Viết chương trình cho phép nhập vào tên và mssv. Kiểm tra nếu mssv đó không trùng với mình thì bắt nhập lại. In ra màn hình kết quả.
3. Viết chương trình cho phép nhập vào một số n. Kiểm tra nếu n < 10 thì bắt nhập lại. Tính tổng các số từ 1 đến n. In kết quả ra màn hình.
4. Viết trình cho phép nhập vào một chuỗi. Kiểm tra chuỗi đó có tồn tại trong một file text (ví dụ test.txt) cùng thư mục hay không.

TÀI LIỆU THAM KHẢO

Tài liệu lập trình shell trên Linux:

http://sinhvienit.net/forum/tai-lieu-lap-trinh-shell-linux

unix.211046.html, truy cập ngày 10/1/2019.