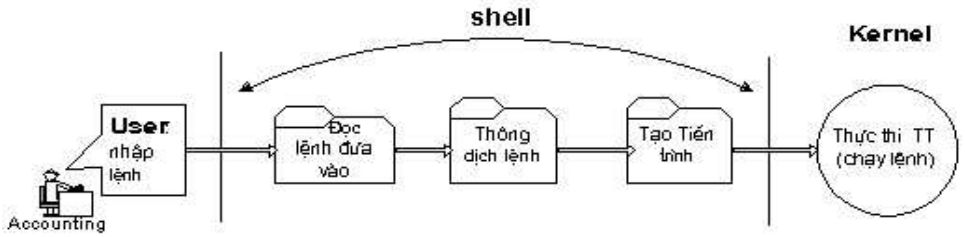
UNIX/Linux
LẬP TRÌNH TRONG MÔI TRƯỜNG SHELL

Trước khi bắt tay vào viết những ứng dụng không cần tới các ngôn ngữ lập trình phức tạp khác, chương này sẽ đề cập cách tiếp cận với ngôn ngữ kịch bản (script) của hệ vỏ (shell, từ đây sẽ gọi là shell script), dùng điều khiển và tương tác với Linux. Khi tiếp cận với DOS, DOS cung cấp một shell để tạo các xử lí theo lô trên những tập tin *.bat, tương đối rõ ràng, đơn giản. Tuy nhiên shell của DOS không mạnh và hữu dụng bằng shell script trên Linux. Tài liệu này sẽ cung cấp những kiến thức vừa đủ để người dùng UNIX/LINUX có thể dùng shell tạo ra các chương trình thực thi hữu hiệu, thậm chí còn có thể dùng shell để thực hiện được mọi thao tác kiểm soát hệ điều hành (như các nhà chuyên nghiệp vẫn nói). Những đích chính cần đạt được như sau:

1.Shell và mục đích sử dụng
2.Cú pháp và cách điều khiển các lệnh của ngôn ngữ shell
3.Hiển thị và thể hiện màu sắc
4.các ví dụ thực hành

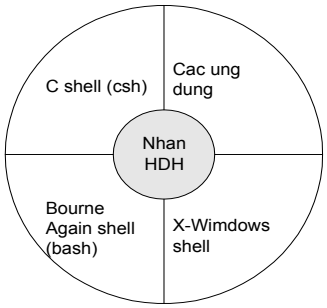
1. Shell của UNIX/Linux

Mọi thứ được thực hiện trên Unix đều bởi tiến trình. Vậy tạo ra tiến trình như thế nào ? Cách thứ nhất là viết ra các chương trình mà các chương trình này biết cách tạo ra tiến trình (C/C++). Tuy nhiên cách này đòi hỏi nhiều hiểu biết và nỗ lực. Cũng như các hệ điều hành làm việc kiểu ảo khác, Unix hổ trợ một phương tiện *xử lí lệnh* làm giao diện giữa lệnh máy (mà người dùng đưa vào) và việc thực thi của lệnh đó (bởi Unix). Phương tiện đó gọi là ***shell***. Từ khi ra đời Unix đã có vài kiểu shell, đó là Bourne, C, Korn shell. Thực ra shell làm gì ? Tòan bộ mục đích của shell là để khởi động các tiến trình xử lí lệnh đưa vào: yêu cầu đưa (dòng) lệnh vào, đọc đầu vào, thông dịch dòng lệnh đó, và tạo ra tiến trình để thực hiện lệnh đó. Nói cách khác shell quét dòng lệnh đưa vào máy tính, cấu hình môi trường thực thi và tạo tiến trình để thực hiện lệnh.



Hình 2: Vị trí của shell khi "thực hiện" lệnh của người dùng Shell dịch các lệnh nhập vào thành lời gọi hệ thống, chuyển các ký hiệu dẫn hướng >, >> hay | thành dữ liệu di chuyển giữa các lệnh. Đọc các biến môi trường để tìm ra thông tin thực thi lệnh.

Như vậy tìm hiểu shell thực tế là học một ngôn ngữ lập trình, cho dù không phức tạp như C, hay các ngôn ngữ khác, nhưng cũng phải qua những đòi hỏi cần thiết. Trong Unix/Linux có các lọai shell khác nhau và có thể lựa chọn để dùng theo nhu cầu mà người dùng thấy phù hợp. Hình 2 là mô hình tương tác giữa các shell, chương trình ứng dụng, hệ X-Window và hạt nhân.



Hình 2

Linux/Unix tách biệt các ứng dụng, lệnh gọi các hàm chức năng của nhân thành những đơn thể rất nhỏ (tiến trình). Tuy nhiên, nhiều lệnh của Linux có thể kết hợp lại với nhau để tạo nên chức năng tổng hợp rất mạnh mẽ. Ví dụ:

$ ls -al | more

lệnh trên được kết hợp bằng hai lệnh, ls liệt kê toàn bộ danh sách tệp và thư mục trên đĩa ra màn hình, nếu danh sách quá dài, ls chuyển dữ liệu kết xuất cho lệnh more xử lý hiển thị kết quả thành từng trang màn hình. Linux có cách kết hợp dữ liệu kết xuất của các lệnh với nhau thông qua cơ chế chuyển tiếp (redirect), ống dẫn (pipe).

Kết hợp các lệnh với nhau chỉ bằng dòng lệnh không chưa đủ. Nếu muốn tổ hợp nhiều lệnh đồng thời với nhau và tùy vào từng điều kiện, kết xuất của lệnh, mà có những ứng xử thích hợp thì sao? Lúc đó sẽ dùng đến các cấu trúc lập trình rẽ nhánh như *if, case*. Trường hợp bạn muốn thực hiện các thao tác vòng lặp, phải dùng các lệnh như *for, while* ... Shell chính là trình diễn dịch cung cấp cho người dùng khả năng này. Hầu hết các Shell trong Unix/Linux sử dụng một ngôn ngữ gần giống với C (điều này cũng dễ hiểu bởi trong thế giới Unix/Linux, C là ngôn ngữ lập trình thống trị). Ngôn ngữ Shell càng giống C thì lập trình viên hay người điều khiển Linux càng cảm thấy thân thiện với HĐH.

Hệ thống cung cấp cho người dùng rất nhiều chương trình shell. Mỗi shell có một số tiện ích như hỗ trợ chế độ gõ phím, ghi nhớ lệnh. Kết hợp các tiện ích của shell để tạo ra một chương trình chạy được, thì một chương trình như vậy được lưu dưới dạng một tệp, gọi là tệp kịch bản (script, hãy thử mở một tệp như vậy và quan sát cấu trúc của tệp). Viết được một tệp script, thực chất là đã lập trình theo shell. Một khi đã quen thuộc với một shell và cách hoạt động của shell đó, người dùng có thể làm chủ được các shell khác một cách để dàng.

Các shell trên Unix/Linux:

sh (Bourne)	shell nguyên thủy áp dụng cho Unix
Csh, tcsh và zsh	dòng shell sử dụng cấu trúc lệnh của C làm ngôn ngữ kịch bản. Được tạo ra đầu tiên bởi Bia Joy. Là shell thông dụng thứ hai sau bash shell.

bash	shell chủ yếu của Linux. Ra đời từ dự án GNU. *bash* (Viết tắt của Bourne Again Shell có lợi điểm là mã nguồn được công bố rộng rãi. Nếu bash chưa có sẵn trong hệ thống Unix hay Linux, hãy tải về, biên dịch và sử dụng miễn phí tại địa chỉ *www.gnu.org*
rc	shell mô rộng của csh với nhiều tương thích với ngôn ngữ C hơn. *rc* cũng ra đời từ dự án GNU.

Shell chuẩn thường được các nhà phân phối Linux sử dụng hiện nay là bash shell. Khi cài đặt Linux, trình cài đặt thường mặc định bash là shell khởi động. Có thể tìm thấy chương trình shell này trong thư mục /bin với tên chương trình là bash. *bash* đôi khi là một chương trình nhị phân đôi khi là một script gọi đến liên kết nhị phân khác. Có thể dùng lệnh *file* để xem bash là một tập tin nhị phân hay script như sau:

```
$ file /bin/bash
/bin/bash: ELF 32-bit LSB executable. Intel 80386
```

nếu kết quả kết xuất là dạng ELF thì có nghĩa là bash là chương trình nhị phân.

Tuy *bash* là shell sử dụng phổ biến trong Linux, nhưng các ví dụ về lập trình sẽ sử dụng ngôn ngữ và lệnh của shell ***sh*** bởi vì ***sh*** là shell nguyên thủy, có thể chạy trên cả Unix. Bằng lệnh *file* ta sẽ thấy trong hầu hết các bản Linux hiện nay *sh* chỉ là liên kết đến *bash* mà thôi. Ví dụ:

```
$ file /bin/sh
/bin/sh: symbolic link to bash
```

điều này có nghĩa là *bash* hoàn toàn có thể diễn dịch và điều khiển các lệnh của shell *sh*.

2. SỬ DỤNG SHELL NHƯ NGÔN NGỮ LẬP TRÌNH

Có hai cách để viết chương trình điều khiển shell: gõ chương trình ngay từ dòng lệnh 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), điều này hiệu quả và tận dụng triệt để tính năng tự động hóa của shell.

2.1. Điều khiển shell từ dòng lệnh

Chúng ta hãy bắt đầu, giả sử trên đĩa cứng có rất nhiều file 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 trạng 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 myfolđer; 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 cho 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).

2.2. 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) để soạn nội dung tập tin first.sh như sau:

```
# ! /bin/ sh
# first.sh
# Script nay se 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. Đẩy thực sự không phải là chú thích. Cặp ký tự #! là chỉ thị yêu cấu 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 kì 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, 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 tệp 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.

2.3. Thực thi script

Chúng ta vừa tạo ra tập tin script first.sh, nó có thể được gọi thực thi theo 2 cách. Cách đơn giản nhất là triệu 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 của bạn thường không chứa đường dẫn hay vị trí thư mục hiện hành. Để khắc phục, ta có thể thêm vào biến môi trường PATH chỉ định 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 đang nhập - tương tự autoexec.bat của DOS). Tuy nhiên cách gọn và đơn giản nhất mà ta vẫn thường làm là định rõ dấu thư mục hiên 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 qui 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. 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 qui 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 của 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 quyến 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, UNTX 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

3. CÚ PHÁP NGÔN NGỮ SHELL

Chúng ta đã thấy cách viết lệnh và gọi thực thi tập tin scirpt. Phần tiếp theo nay dành cho bạn khám phá sức mạnh của ngôn ngữ lập trình shell. Trái với lập trình bằng trình biên dịch khó kiểm 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 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. 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
- Danh shell
- Hàm
- Các hình nội tại của shell
- Lấy về kết quả của một lệnh
- Tài liệu Here

3.1. 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 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:

```
$ xinahao=hello
$ echo $xinchao
Hello
$ xin chao= "I am here"
$echo $xin chao
I am here
$ xinchao=12+l
$echo $xin chao
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 nhập liệu do người dùng đưa vào và giữ lại trong biến để sử dụng. Ví dụ:

```
$ read yourname
XYZ
$echo "Hello " $yourname
Hello XYZ
```

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

3.1.1. Các kí tự đặc biệt (Metalcharacters của Shell)

3.1.1.1 Chuyển hướng vào/ra

Một tiến trình Unix/Linux bao giờ cũng gắn liền với các đầu xử lí 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:

Kí hiệu Ý nghĩa (… tượng trưng cho đích đổi hướbg)

>	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 lí 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 lí 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 cho thấy ta có thể thay đổi đầu và đầu ra cho lệnh như thế nào. Những lệnh cho phép đổi đầu ra/vào gọi chung là qúa 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 là sư 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 lí 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 mahn 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 tác động trên tệp vào/ra. Ta cũng có cách xử lí ngay trong một dòng của tệp, cái đó gọi là *đổi hướng trong dòng (in-line Redirection)*. Loại này bao gồm hai phần: đổi hướng (<<) và *dấu hiệu đánh dấu* (là bất kì kí tự gì) 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, phai 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 hòai vọng nhận nhiều (thông tin) ở đầu vào.

Hãy thử với ví dụ sau:

```
$ cat << EOF
> Logged in
> EOF > login.time
$ date >> login.time
$cat login.time
Login in
Fri May 19 12:40:15 PDT 2004
```

3.1.1.2 Các kí 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 &
$
```

Bằng lệnh ps sẽ thấy lệnh sort đang chạy kèm với só ID của tiến tình đó.

Bằng lệnh

```
$ jobs
[1]
```

sẽ thấy số hiệu của lệnh đang chạy ngầm.

Để kết thúc thực thi, dùng

```
$ kill 1234          #1234 là só 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 ~)

Hay còn gọi là dấu thay thế. Bất kì 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 `**date**` > login.time

sẽ nói 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

1.Hãy thử với lệnh:
$echo Logged in Fri May 12:52:25 UTC 2004
Kết quả ?
2. Kết hợp:
$cat << EOF
> Logged in `date`
> EOF > Login.time (ENTER)
Sau đó thực hiện:
$ cat login.time
Kết quả ?

4. Ống dẫn (Pipelines)

Shêll 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 lí 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), sẽlà đầi vào (stdin) của *ls –l*.

Ví dụ:

$ (date ; who) | ls -

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 đàu ra của lệnh trong dòng lệnh*
cmd1 | cmd2	*nối đầu ra của cmd1 vào đầu vào của cmd2*

3.1.1.2 Dấu bọc chuỗi (quoting)

Shell có một tập các kí tự đặc biệt mà hiệu lực của chúng là để vô hiệu hóa ý nghĩa của các kí tự đặc biệt khác. Khi một kí tự đặc biệt bị giải trừ hiệu lực, ta gọi kí 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 đặt biệt $ trong chuỗi.

1. 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:

$cat file1\&2 sẽ cho kết quae như momg 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 3-1: ***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 chuỗi Hi there. Nội dung của biến sau đó được hiển thị bằng lệnh echo trong các trường hợp bọc chuôi bằng nháy kép, nháy đơn và dấu hiển thị ký tự đặc biệt \. Tóm lại '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.

3.1.2. Biên 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 shell 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 shell các đường dẫn (phần cách bằng dấn hai chấm :). Linux thường tìm các 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 shell chuỗi. Biến này chứa danh shell 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 uer đăng nhập chứa một số danh shell biến môi trường dùng cho mục đích riêng. Có thể xem đanh shell 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**).

3.1.3. 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 shell của tất cả các tham số trên dòng lệnh. Chúng được lưu trong một chuỗi duy nhất phản cách bằng ký tự đầu tiên quy định trong biến $IFS
$@	Danh shell các tham số được chuyển thành chuỗi. Không sử dụng dấu phân cách của biến IFS.

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

```
$IFS= "A”
$set foo bar bam
$echo “$@”
foo bar bam
$echo "$*”
foo^ bar^bam
$unset IFS
$echo "$*"
foo bar bam
```

Ta nhận thấy, lệnh set tiếp nhậnn 3 tham số trên dòng lệnh là foo bar bam. Chúng ảnh hưởng đến biến môi trường $* và $@. Khi IFS được qui đinh là ký tự ^ , $* chứa danh shell các tham số phân cách bằng ký tự ^ . Khi đặt IFS vế 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 " $ # "
3
```

Khi lệnh không có tham số thì $0 chính là tên lệnh còn $# trả về giá trị 0.

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

Ví dụ3-2: *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 qủa 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 scnpt is now complete

3.2. 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 điề kiện.

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

3.2.1. 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.*

Dước 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
```

Mà 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 thể 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
```

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 bàng 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 f ile	*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 nhiễu 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.

3.3. Cấu trúc điều khiển

Shell cung cấp cấu trúc lệnh điều khiển rất giống với các ngôn ngữ lặp 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 mẻ 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.*

1 3.3.1. 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 đụng if tùy vào cấu trả lời của bạn mà đưa ra lời chào thích hợp

Ví du 3-3 ***if_control.sh***

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

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
$

Ờ 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 morningg”, nếu không (mệnh đề else) in ra chuỗi “Goođ afternoon".

Shell không đòi hỏi phải canh lề hay thụt đầu dòng cho từng lệnh. Chúng ta canh lề để có pháp được rõ ràng. 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.*

3.3..2. Lệnh *elif*

Thật không may, có rất nhiễu vấn đề phát sinh với đoạn trình script trên. Tất cả 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 điếu kiện lần thứ hai bên trong else. Script dưới đãy của có thể 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 du 3-4: ***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 gián như ví dụ 3-3, 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.

3.3.3. 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 rổng, 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 du: 3-5: e*lif_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 chi 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 đùng lệnh **printf** (sẽ đề cập ở phần sau) 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.*

3.3.4. 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 qủa 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 du 3-6: *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 barr 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 shell 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 du 3-7: ***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)* (sẽ được chúng ta tìm hiểu chi tiết hơn trong phần sau). Danh shell của các phần tử trong lệnh **for** được cung cấp bởi kết quả trả vệ 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.

3.3.5. 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 du: 3-8: ***password.sh***

```
#!/bin/sh

echo "Enter password"
read trythis

while [ "$trythis" != "secret" ]; do
  echo "Sorry, try again"
  read trythis
done

exit 0
```

Kết xuất của scnpt

$./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 tòan có thể thay thế **for** trong trường hợp tập dữ liệu lớn. H theo dõi ví dụ sau:

Ví dụ: 3-9 ***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 đã 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.

3.3.6. Lệnh *intil*

Cú pháp của lệnh **until** như sau:

until *condition*
do
    statements
done

Lệnh **until** tương tự lệnh **while** nhưng điều kiện kiểm tra bi 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:1

Ví dụ 3-10: *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 shell các user đăng nhập vào hệ thống, chuyển danh shell 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.

3.3.7. 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 chầm nhảy **;;** 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 3-4 với phiên bản viết bằng **case** như sau:

Ví dụ 3-11 ***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 trên đầu, 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 biếntrong 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 du 3-12 ***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 du 3-13 ***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 đùng 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 mẩu 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 thạnh 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]*.

3.4. Danh shell thực thi lệnh (Lists)

Đôi lúc cần kết nối các lệnh lại với nhau thực hiện theo thứ tự kiểm tra trước khi ra một quyết định nào đó, ví dụ, cần kiểm tra hàng loạt điều kiện phải đúng bằng **if** trước khi in ra thông báo như sau:

```
if [-f this_file] ; then
        if [- f that_file ] ; then
                if [-f other_file ] ; then
                        echo "All files present, and correct"
                fi
        fi
fi
```

Hoặc giả muốn thực hiện lệnh khi một trong các điều kiện là đúng

```
if [-f this_filel]; then
        foo="true"
elif [ -f that_file 1 ; then
        foo="true"
elif [-f other_file ] ; then
        foo="true"
        echo “some condition are checked"
else
        foo="false"
fi
if [ $foo=”true” ]  ; then
        echo “One of the files exists"
fi
```

Hoàn toàn có thể dùng **if** để thực hiện các yêu cầu trên, nhưng chúng không thuận tiện lắm. Shell cung cấp một cú pháp danh shell AND và OR gọn hơn. Chúng thường sử dụng chung với nhau, nhưng ta hãy tạm thời xét chúng tách biệt để dễ hình dung.

3.4.1. Danh sách AND (&&)

Danh shell AND cho phép thực thi một chuỗi lạnh kề nhau, lệnh sau chỉ thực hiện khi lệnh trước đã thực thi và trả về mã lỗi thành công. Cú pháp sử dụng như sau:

Statement1 && statement2 && statement3 && . . .

Bắt đầu từ bên trái *statement1* sẽ thực hiện trước, nếu trả về *true* thì *statement2* tiếp tục được gọi. Nếu *statement2* trả về *false* thì shell chấm dứt danh shell AND ngược lại *statement3* sẽ được gọi ... Toán tự **&&** dùng để kiểm tra kết qủa trả về của statement trước đó.

Kết quả trả về của AND sẽ là *true* nếu tất cả các lệnh *statement* đều được gọi thực thi. Ngược lại là *false*.

Hãy xét ví dụ sau, dùng lệnh touch file_one (để kiểm tra file_one tồn tại hay chưa, nếu chưa thì tạo mới) tiếp đến rm file_two. Sau cùng danh shell AND sẽ kiểm tra xem các file có đồng thời tồn tại hay không để đưa ra thông báo thích hợp.

Ví dụ 3-14 ***and_list.sh***

```
#!/bin/sh

touch file_one
rm -f file_two

if [ -f file_one ] && echo "hello" && [ -f file_two ] && echo
"there"
then
    echo -e "in if"
else
    echo -e "in else"
fi

exit 0
```

Clhạy thử script trên bạn sẽ nhận được kết qủa kết xuất như sau:

```
$./and_list.sh
hello
in else
```

Cách chương trình làm việc: Lệnh **touch** và **rm** đảm bảo rằng file_one tồn tại và file_two không có. Trong danh shell biểu thức if, && sẽ gọi lệnh [-f file_one] trước. Lệnh này thành công vì touch đã tạo sẵn file_one. Lệnh echo tiếp tục được gọi echo luôn trả về trị true nên lệnh tiếp theo
[-f file_two] thi hành. Do file_two không tồn tại nên echo "there" không được gọi. Toàn bộ biểu thức trả vế trị false (vì các lệnh trong danh shell không được thực thi hết). Do if nhận trị false nên echo trong mệnh đề else của lệnh if được gọi.

3.4.2 Danh sáchl OR (||)

Danh shell OR cũng tương tự với AND là thực thi một dãy các lệnh, nhưng nếu có một lệnh trả về true thì việc thực thi ngừng lại. Cú pháp như sau:

statementl || statement2 || statement3 && . . .

Bắt đầu từ bên trái, *statementl* được gọi thực hiện, nếu *statement1* trả về *false* thì *statement2* được gọi, nếu *statement2* trả về *true* thi biểu thức lệnh chấm đứt, ngược lại *statement3* được gọi. Kết qủa sau cùng của danh shell OR chỉ đúng (true) khi có một trong các *statement* trả về *true*. Nếu **&&** gọi lệnh tiếp theo khi các lệnh trước đó *true*, thì ngược lại || gọi lệnh tiếp theo khi lệnh trước đó *false*.

Ví dụ 3-14 của danh shell AND có thể sửa lại thành OR như sau:

Ví du 3-15 *or_list.sh*

```
#!/bin/sh

rm -f file_one

if [ -f file_one ] || echo "hello" || echo "there"
then
    echo "in if"
else
    echo "in else"
fi

exit 0
```

Kết qủa kết xuất sẻ là.

```
$./and_list.sh
hello
in if
```

Cách chương trình làm việc: file_one đầu tiên được loại bỏ để báo đảm lệnh if tiếp theo không tìm thấy nó. Lệnh [-f file_one] trả về false vì file_one không tồn tại. Lệnh echo tiếp theo trong chuỗi danh shell OR sẽ được gọi in ra hello. Do echo luôn trả về true nên echo tiếp theo không được gọi. Bởi vì trong danh shell OR có một lệnh trả về true nên toàn bộ biểu thức sẽ là true. Kết quả cuối cùng là echo trong if được gọi để in ra chuỗi 'in if'.

> *Lưu ý, danh shell AND và OR sử dụng thuật toán thẩm định tắt 1 biểu thức, có nghĩa là chỉ cần một lệnh sai hoặc đúng thì coi như toàn bộ biểu thức sẽ có cùng chân trị Điều này cho thấy không phải mọi biểu thức hay lệnh của trong danh shell AND / OR đều được ước lượng. Hãy đặt các biểu thức hay lệnh có độ ưu tiên cao về bên trái. Xác suất ước lượng chúng sẽ cao hơn các biểu thức hay lệnh nằm bên phải.*

Kết hợp cả AND và OR sẽ xử lý được hầu như mọi trường hợp logic trong lập trình. Ví dụ:

[-f flle_one] && command_for_true || command_for_false

Cú pháp trên bảo đảm rằng nếu [-f file_one] trả vế *true* thì command_for_true sẽ được gọi. Ngược lại command_for_false sẽ thực thi (một cách viết ngắn gọn khác của *if else*)

3.4.3. Khối lệnh

Trường hợp bạn muốn thực thi một khối hình tại nơi chỉ cho phép đặt một lệnh (như trong danh shell AND hay OR chẳng hạn) bạn có thể sử dụng cặp **{ }** để bọc khối lệnh như sau:

```
if [ -f file_one ] && {
                        ls -l
                        echo “complex block execute"
                        }
then
        echo "command completed"
fi
```

3.5. Hàm (function)

Tương tự các ngữ trình khác, shell cho phép bạn tự tạo hàm hay thủ tục để triệu gọi bên trong script. Mặc dù bạn có thể gọi các script con khác bên trong script chính, chúng tương tự như việc gọi hàm. Tuy nhiên triệu gọi các script con thường tiêu tốn tài nguyên và không hiệu quả bằng triệu gọi hàm.

Để định nghĩa hàm, bạn khai báo tên hàm tiếp theo là cặp ngoặc đơn **( )** , lệnh của hàm nằm trong ngoặc nhọn **{ }** . Cú pháp như sau:

```
function_name ( ) {
                        Statements
                        }
```

Hãy làm quen với cách sứ dụng hàm bằng ví dụ đơn giản sau:

Ví dụ 3-16 *my_function.sh*

```
#!/bin/sh

foo() {
    echo "Function foo is executing"
}

echo "script starting"
foo
echo "script ended"

exit 0
```

Kết quả kết xuất khi bạn chạy script hiển thị như sau

```
$./my_function.sh
script starting
Function foo is executing
script ending
```

Cách chương trình làm việc: Shell sẽ bắt đầu thực thi lệnh trong script từ đầu đến cuối, khi gặp foo() lần đầu, shell sẽ hình dung foo là một hàm. Shellghi nhớ nội dung hàm và không thực thi hàm. Shell tiếp tục bỏ qua nội dung hàm cho đến cuối ký tự } và thực hiện lệnh echo "script starting". Khi gặp lại foo lần thứ hai, shell biết là ta muốn triệu gọi hàm, shell quay lại thực hiện nội dung của hàm foo(). Một khi chấm dứt lời gọi hàm, dòng lệnh tiếp theo sau hàm sẽ được thực thi. Như ta thấy, cần phải khai báo và định nghĩa hàm trước khi sử dụng và gọi nó bên trong script. Điều này tương tự cách qui định của Pascal và C, tuy nhiên shell không cho phép bạn khai báo hàm kiểu chỉ nêu nguyên mẫu của hàm (forward), mà chưa cần đinh nghĩa nội dung chi tiết hàm.

3.5.1 Biến cục bộ và bên toàn cục

Để khai báo biến cục bộ chỉ có hiệu lực bên trong hàm, hãy dùng từ khóa *local*. Nếu không có từ khóa *local*, các biến sẽ được xem là toàn cục (*global*), chúng có thể tồn tại và lưu giữ kết quả ngay sau khi hàm đã chấm dứt. Biến toàn cục được nhìn thấy và có thể thay đổi bởi tất cả các hàm trong cùng script. Trường hợp đã có biến toàn cục nhưng lại khai báo biến cục bộ cùng tên, biến cục bộ sẽ có giá trị ưu tiên và hiệu lực cho đến khi hàm chấm dứt.

Ví dụ 3-17 *function2.sh*

```
#!/bin/sh

sample_text="global variable"

foo() {
    local sample_text="local variable"

    echo "Function foo is executing"
    echo $sample_text
}

echo "script starting"
echo $sample_text

foo

echo "script ended"
echo $sample_text

exit 0
```

Kết qủa kết xuất

```
$./function2.sh
script starting
global variable
Function foo is executing
local variable          #sample_text is local in function
script ended
global variable         #sample_text is global outside the
                        function
```

Hàm có thể trả về một giá trị. Để trả vế giá trị số, bạn có thể dùng lệnh return. Ví dụ:

```
foo( ) {
        ...
        return 0
        }
```

Để trả vế giá trị chuỗi, bạn có thể dùng lệnh *echo* và chuyển hường nội đung kết xuất của hàm khi gọi như sau:

```
foo() {
        echo "string value"
        }
. . .

x= $ ( foo )
```

Biến x sẽ nhận trị trả về của hàm foo() là "string value". $() là cách lấy về nội dung của một lệnh. Có một cách khác để lấy trị trả về của hàm, đó là sử đụng biến toàn cục (do biến toàn cục vẫn lưu lại trị ngay cả khi hậm chấm dứt). Các script trong chương trình ứng dụng ở cuối chương sẽ sử dụng đến kỹ thuật này.

3.6.2. Hàm và cách truyền tham số

Shell không có cách khai báo tham số cho hàm như cách của C, Pascal hay các ngôn ngữ lập trình thông thường khác. Việc truyền tham số cho hàm tương tự như truyền tham số trên dùng lệnh. Ví dụ để truyền tham số cho foo(), ta gọi hàm như sau

foo "paraml", "param2", param3 . . .

Vậy làm cách nào hàm nhận và lấy về được nội dung đối số truyền cho nó? Bên trong hàm, ta gọi các biến môi trường **$*, $1, $2** ... chúng chính là các đối số truyền vào khi hàm được gọi. Lưu ý, nội dung của **$*, $1, $2** do biến môi trường nắm giữ sẽ được shell tạm thời cất đi. Một khi hàm chấm dứt, các giá trị cũ sẽ được khôi phục lại.

Mặc dù vậy, có một số shell cũ trên UNIX không phục hồi tham số môi trường về giá trị ban đầu khi hàm kết thúc. Nếu muốn bảo đảm, hãy nên tự lưu trữ các biến tham số

môi trường trước triệu gọi hàm. Tuy nhiên các shell mới và nhất là nếu chỉ muốn hướng về Linux, thì không cần lo lắng điều này.

Dưới đây là một ví dụ cho thấy cách gọi và nhận trị trả về của hàm đồng thời xử lý đối số truyền cho hàm được gọi.

Ví dụ 3-18 ***get_name.sh***

```
#!/bin/sh

yes_or_no() {
  echo "In function parameters are $*"
  echo "Param 1  $1 and Param2  $2"
  while true
  do
    echo -n "Enter yes or no"
    read x
    case "$x" in
      y | yes ) return 0;;
      n | no )  return 1;;
      * )       echo "Answer yes or no"
    esac
  done
}

echo "Original parameters are $*"

if yes_or_no "Is your name” ‟ $1?"
then
  echo "Hi $1"
elif
  echo "Never mind"
fi

exit 0
```

Kết quả kết xuất khi gọi lệnh như sau:

$,/get_name.sh HoaBinh SV
Original parameters are HoaBinh SV
In function parameters are Is your name HoaBinh
Param 1 Is your name param 2 HoaBinh
Is your name HoaBinh ?
Enter yes or no : yes
Hi HoaBinh, nice name

Cách làm làm việc: Hàm yes_or_no( ) được định nghĩa khi script thực thi nhưng chưa được gọi. Trong mệnh đề if, hàm yes_or_no được sử dụng với tham số truyền cho ham là nội dung của biến môi trường thứ nhất (ở ví dụ trên $1 được thay thế bằng HoaBinh) và chuỗi “Is your name”. Bên trong hàm nội dung của $1 và $2 được in ra (Hãy để ý là chúng khác với giá trị $1, $2 của môi trường shell ban đầu). Hàm yes_or_no xây dựng cấu trúc case tùy theo lựa chọn của người dùng mà trả vế trị 0 hay 1. Khi người dùng chọn yes, hàm trá về giá trị 0 (true). Lệnh bên trong if được gọi để in ra chuỗi "nice

name".

3.6. Các lệnh nội tại của shell

Ngoài các lệnh điều khiển, shell còn cung cấp cho các lệnh nội tại (build-in) khác rất hữu ích. Chúng được gọi là lệnh nội tại bởi vì không thể thấy chúng hiện hữu như những tập tin thực thi trong một thư mục nào đó trên hệ thống tệp. (Có thể xem những lệnh này tương tự khái niệm lệnh nội trú của DOS). Trong quá trình lập trình shell, chúng sẽ thường xuyên được sử dụng.

3.6.1. break

Tương tự ngôn ngữ C, shell cung cấp lệnh **break** để thoát khỏi vòng lập **for, while** hoặc **until** bất kể điều kiện thoát của các lệnh này có diễn ra hay không.

Ví dụ 3-19: ***break.sh***

```
#!/bin/sh

rm -rf fred*
echo > fred1
echo > fred2
mkdir fred3
echo > fred4

for file in fred*
do
  if [ -d "$file" ]; then
    break;
  fi
done

echo first directory fred was $file

exit 0
```

Đoạn script trên dùng lệnh for để duyệt toàn bạ tên của tập tin và thư mục hiện hành bất đầu bằng chuỗi *fred*. Khi phát hiện thư mục đầu tiên trong danh shell các tập tin, sẽ in ra tên thư mục và dùng **break** rể thoát khỏi vòng lặp (không cần duyệt tiếp các tập tin khác).

Lệnh **break** thường ngắt ngang logic của chương trình , vì vậy nên hạn chế dùng **break.** Lệnh break không có tham số cho phép thoát khỏi vòng lặp hiện hành. Nếu đặt tham số cho lệnh ví dụ, break 2, break 3 chẳng hạn, có thể thoát khỏi nhiều vòng lặp cùng một lúc. Tuy nhiên chúng sẽ làm cho chương trình khó theo dõi. Tốt nhất ta.nên dùng**.break** không tham số.

3.6.2 continue

Lệnh **continue** thường được dùng bên trong vòng lặp, continue yêu cầu quay lại thực hiện bước lặp kế tiếp mà không cần thực thi các khối lệnh còn lại.

Ví dụ 3-20 ***continue.sh***

```
#!/bin/sh

rm -rf fred*
echo > fred1
echo > fred2
mkdir fred3
echo > fred4

for file in fred*
do
  if [ -d "$file" ]; then
    continue
  fi
  echo file is $file
done

exit 0
```

Đoạn script trên dùng lệnh **for** để duyệt toàn bộ tên của tập tin và thư mục hiện hành bắt đầu bằng chuỗi *fred*. Nếu kiểm tra tên tập tin là một thư mục, thì **continue** yêu cầu quay lại duyệt tiếp file khác. Ngược lại lệnh **echo** sẽ in ra tên tệp.

continue còn cho phép truyền tham số để xác định số lần lặp cần quay lại.

Ví dụ:

```
for x in 1 2 3 4 5
do
 echo before $x
        if [ $x = =2 ] ; then
                continue 2
        fi
 echo after $x
done
```

Kết quả

```
before 1
after 1
before 2
before 5
after 5
```

3.6.3. Lệnh *:* (lệnh rỗng)

Lệnh **:** được gọi là lệnh rỗng (null command). Đôi lúc lệnh này được đùng với ý nghĩa logic là *true*. Khi dùng lệnh **:** thực thi nhanh hơn việc so sánh *true*.

Một số shell cũ còn sử dụng lệnh **:** với ý nghĩa chú thích một dòng lệnh. Tuy nhiên bất kỳ khi nào có thể, hãy nên dùng # thay cho chú thích bằng **:**.

Ví dụ: 3-21 ***colon.sh***

```
#!/bin/sh
```

```
rm -f fred
if [ -f fred ]; then
   :
else
  echo file fred does not exist
fi

exit 0
```

Trong đoạn script trên, nếu kiểm tra *fred* tồn tại thì không làm gì cả, nếu không ta sẽ in ra thông báo lỗi.

3.6.4. Lệnh . (thực thi)

Lệnh **.** dùng để gọi thực thi một script trong shell hiện hành. Điều này có vẻ hơi lạ, vì chỉ cần gõ tên script là script có thể tự thực thi mà không cần tới **.** , tuy vậy nó có một ý nghĩa đặc biệt: thi hành và giữ nguyên những thay đổi về môi trường mà script tác động (xem lại fork() và exec()).

Thông thường, khi thực thi một script, shell sẽ bảo lưu lại toàn bộ biến môi trường hiện hành và tạo ra một môi trường mới (hay shell phụ - sub shel1) để script hoạt động. Một khi script chấm dứt bằng lệnh **exit**, thì toàn bộ thông số môi trường của shell hiện hành sẽ được khôi phục lại.

Cú pháp sử dụng như sau: **. ./shell-script**

Ví dụ sau sẽ cho thấy cách tác động vào biến môi trường hiện hành bằng lệnh **.** .

Ví du 3-22: *dot_command.sh*

```
#!/bin/sh
echo "Inside script"
PATH=/mypath/bin: /usr/local
echo $PATH
echo "Script end"
```

Trước khi chạy, hãy in ra nội dung của biến PATH trong shell hiện hành. Tiếp đến chạy *do_command.sh* bằng lệnh **.** và in lại kết quả của PATH như sau:

```
$echo $PATH
/usr/bin: usr/lib

$. /dot_command.sh   #Không dùng .
Inside script
/mypath/bin : /usr/local
Script end

$echo $PATH
/usr/bin: usr/11b              #shell khôi phục lại môi trường gốc
```

Bây giờ chạy lệnh với **.**

```
$ . ./dot_command.sh
Inside script
/mypath/bin: /usr/local
 Script end

$echo $PATH
/mypath/bin: /usr/1ocal          #Bảo lưu thay đổi do script thực hiện
```

3.6.5. eval

Lệnh **eval** cho phép ước lượng một biểu thức chứa biến. Cách dễ hiểu nhất là xem **eval** làm việc trong ví dụ sau:

```
foo=10
x=foo
y= '$' $x
echo $y
```

Đoạn lệnh trên in ra kết quả là chuỗi $foo.
Bây giờ nếu bạn sử dụng eval

```
foo=10
x=foo
eval y= '$' $x
echo $y
```

Kết quả in ra sẽ là 10. Lý do y = '$' $x sẽ được diễn dịch thành chuỗi y=$x Lệnh **eval** tiếp đến sẽ ước lượng y=$x như là biểu thức gán. Kết quả là y mang giá trị của nội dung biến x (10). **eval** rất hữu dụng, cho phép sinh ra các đoạn lệnh thực thi động ngay trong quá trình script thi hành.

3.6.6. exec

Lệnh **exec** dùng để gọi một lệnh bên ngoài khác. Thường **exec** gọi một shell phụ khác với shell mà script đang thực thi.

Ví dụ 3-23: ***exec_demo.sh***

```
#! /bin/sh
echo "Try to execute mc program"
exec mc
echo "you can not see this message !"
```

Đoạn script in ra chuỗi thông báo sau đó triệu gọi **mc**. exec sẽ chờ cho chương trình gọi thực thi xong mới chấm dứt script hiện hành.
Mặc đinh **exec** sẽ triệu gọi exit khi kết thúc lệnh. Chính vì vậy, nếu gọi **exec** ngay từ dòng lệnh, sau khi lệnh thực hiện xong, điều khiển sẽ thoát ra khỏi shell phụ, quay trở về shell gốc, là màn hình đăng nhập.

3.6.7. exit n

Lệnh **exit** sẽ thoát khỏi shell nào gọi nó và trả về mã trạng thái lỗi n. Trước giờ ta vẫn gọi **exit 0** bên trong shell phụ. Nếu gọi **exit** ngay từ dòng lệnh, nó sẽ khiến thoát khỏi shell chính trở về màn hình đăng nhập luôn (đây cũng là cách thường dùng để thoát khỏi user hiện hành, đăng nhập làm việc dưới tên user khác).

exit rất hữu dụng trong các script, nó trả về mã lỗi cho biết script được thực thi thành công hay không. Mã lỗi 0 có nghĩa là thành công. Các giá trị từ 1-125 script tùy nghi sử dụng cho nhiều mục đích khác nhau. Các giá trị còn lại được dành cho mục đích riêng. Cụ thể là:

126	file không thể thực thi
127	Lệnh không tìm thấy
Lớn hơn 128	Đã nhận được tín hiệu (signal) phát sinh

Sử dụng 0 là giá trị thành công có thể gây lầm lẫn cho một số lập trình viên C (ở đó 0 được coi là *false* còn khác 0 là *true*). Tuy nhiên bằng cách này, ưu điểm là có thể tận dụng các giá trị khác 0 làm mã qui định lỗi, không cần phải dùng thêm biến toàn cục để lưu giữ mã lỗi trả về.

Ví dụ đơn giản về **exit** dưới đay kiểm tra xem tệp *.profile* có tồn tại hay không, nếu có trả lại 0, còn không trả về 1.

Ví dụ 3-24 ***test_exits.sh***

```
#!/bin/sh
if  [ -f .profile ] ; then
    exit 0
fi
exit 1
```

Nếu muốn, có thể đổi lệnh **if** sang cấu trúc danh sách lệnh && hay || như sau:

```
[ -f  .profile ]  && exit 0 || exit 1
```

3.6.8. export

Khi bắt đầu thực thi một shell, các biến môi trường đều được lưu lại. Khi có khai báo và sử dụng biến trong một script, nó chỉ có giá trị đối với shell phụ gọi script đó. Để biến có thể thấy được ở tất cả các script trong shelll phụ hay các script gọi từ shell khác, hãy dùng lệnh **export**. Lệnh **export** có tác dụng như khai báo biến toàn cục. Ví dụ sau sẽ cho thấy cách sử dụng **export**.

Ví dụ 3-25 ***export2.sh***

```
#! /bin/sh
echo "Value : $foo”
echo "Value : $bar"
```

Ví dụ 3-26 ***export1.sh*** #xuất biến ra toàn cục

```
#! /bin/sh
foo="This is foo"
export bar = “This is bar"
```

Chạy lệnh, xuất biến *bar* ra

```
$./export2.sh
```

Kết quả khi gọi export1.sh là

```
$./export1.sh
Value :
Value: This is bar
```

Dòng đầu cho kết quả biến $foo rỗng vì foo không được khai báo toàn cục từ export1.sh, nên export2.sh không thấy được biến. Dòng thứ 2 cho kết quả là nội dung của biến $bar do bắt được khai báo bằng **export.** Biến bar trở nên toàn cục và các script khác nhìn thấy *bar*.

Nếu muốn tất cả các biến mặc định là toàn cục trong tất cả các script, có thể gọi lệnh set –a hay set-allexport trước khi thiết lập nội dung cho biến.

3.6.9 Lệnh expr

Lệnh **expr** tính các đối đầu vào như một biểu thức. Thường **expr** được dùng trọng việc tính toán các kết qủa toán học khi đổi giá trị từ chuỗi sang số. Ví dụ :

```
x= “12"
x= `expr $x  +  1`
```

Kết quả x=13. Lưu ý, cặp dấu ‘ ‘ bọc biểu thức expr không phải là dấu nháy đơn (Ký tự này là phím nằm dưới phím ESC và bên trái phím 1, chung với phím ~. Các toán hạng và toán tử phải cách nhau bằng khoảng trắng. Ở đây $x và 1 cách ký tự + khoảng trắng. Nếu để chúng sát nhau, khi diễn dịch shell sẽ báo lỗi biểu thức.

Dưới đây là một số biểu thức ước lượng mà expr cho phép:

Biểu thức	Ý nghĩa
axprl | expr2	Kết quả là expr1 nếu expr1 khác 0 ngược lại là expr2
axprl & expr2	0 nếu một trong hai biểu thức là zero ngược lại kết quả là expr1
exprl = expr2	Bằng
exprl >expr2	Lớn hơn
exprl >= expr2	Lớn hơn hay bằng
exprl < expr2	Bé hơn
exprl <= expr2	Bé hơn hay bằng
exprl != expr2	không bằng
exprl + expr2	Cộng

exprl - expr2	Trừ
exprl * expr2	Nhân
exprl / expr2	Chia
exprl % expr2	Modulo (lấy số dư)

Trong các shell mới sau này lệnh **expr** được thay thế bằng cú pháp $ ((. . .)) hiệu quả hơn, và sẽ đề cập cú pháp này ở phần sau.

3.6.10. printf

Lệnh **printf** của shell tương tự **printf** của thư viện C. Mặc dù vậy, cơ bản **printf** của shell có một số hạn chế sau: không hỗ trợ định dạng số có dấu chấm động (float) bởi vì tất cả các tính toán của shell đấu dựa trên số nguyên. Dầu sổ \ dùng chỉ định các hiển thị đặc biệt trong chuỗi (xem bảng dưới). Dấu **%** dùng định dạng số và kết xuất chuỗi. Dưới đây là danh sách các ký tự đặc biệt có thể dùng với dấu \, chúng được là chuỗi thoát.

Chuỗi thoát (escape sequence)	**ý nghĩa**
\\	Cho phép hiển thị ký tự \ trong chuỗi
\a	Phát liếng chuông (beep)
\b	Xóa backspace
\f	Đẩy dòng
\n	Sang dòng mới
\r	Về đầu dòng
\t	Căn tab ngang
\v	căn tab dọc
\ooo	Kí tự đơn với mã là ooo

Định dạng số và chuỗi bằng ký tự % bao gồm

Kí tự định dạng	**Ý nghĩa**
d	Số nguyên
c	Ký tự
s	chuổi
%	hiển thị ký hiệu %

Dưới đây là một số ví dụ về printf

```
$ printf  "Your name %s. It is nice to meet you \n" NV An
Your name NV An. It's nice to meet you .

$ printf “%s  %d\t  %s” “Hi There" “15” "people"
Hi There 15 people
```

Các tham số của lệnh printf cách nhau bằng khoảng trắng. Chính vì vậy nên dùng dấu “ “ để bọc các tham số chuỗi. Lệnh printf thường được dùng thay thế **echo**, mục đích để in chuỗi không sang dòng mới. **printf** chỉ sang dòng mới khi thêm vào chuỗi thoát "\n".

3.6.11 return

Lệnh **return** dùng để trả về giá trị của hàm. Lệnh **return** không tham số sẽ trả về mã lỗi của lệnh vừa thực hiện sau cùng.

3.6.12 set

Trước hết, lệnh **set** dùng để áp đặt giá trị cho các tham số môi trường như **$1, $2, $3 ...** Lệnh **set** sẽ loại bỏ những khoảng trắng không cần thiết và đặt nội dung của chuỗi truyền cho nó vào các biến tham số. Ví dụ:

```
$ set This is parameter
$echo $ 1
This
$echo $3
parameter
```

Thoạt nhìn lệnh **set** không mấy ích lợi nhưng nó sẽ vô cùng mạnh mẽ nếu bạn biết cách sử dụng.

Ví dụ lệnh **date** của linux phát sinh chuỗi sau:

```
$da te
Fri March 13  16:06: 6 EST 2001
```

Nếu chỉ muốn lấy về ngày, tháng hoặc năm thì sao? **set** sẽ thực hiện điều này theo ví dụ sau:

Ví du 3-27 ***set_use.sh***

```
#!bin/sh
echo Current date is  $(date)
set $(date)
echo The month is $2
echo The year is $6
exit 0
```

Kết quả kết xuất

```
$./set_use.sh
Current date Fri March 13      16:06:16 EST 2001
The month is March
The year is 2001
```

Lệnh **set** còn được dùng để đặt các thông số thiết lập cho shell. Ví dụ như khi nói về lệnh **export**, ta đã làm quen với **set -a** hoặc **set –allexport,** cho phép shell khai báo biến toàn cục cho một biến trong script. Khi tìm hiểu và cách dò lỗi của shell ở phần sau, sẽ có thông số đầy đủ hơn về các thiết lập của **set**.

Trong ví dụ trên hãy làm quen với cú pháp lệnh **$( )** . Lệnh này nhận kết quả trả

về của một lệnh dùng làm danh sách biến. Chúng sẽ được nêu chi tiết hơn ở mục sau.

3.6.13. shift

Lệnh **shift** di chuyển nội dung tất cả các tham số môi trường **$1, $2 ...** xuống một vị trí. Bởi vì hầu như ta chỉ có tối đa 9 tham số từ **$1. $2. … $9**, cho nên nếu các shell cần nhận từ 10 tham số trở lên thì sao? Lệnh **shift** dùng để giải quyết vấn đề này.

Nếu gọi $10 shell thường hiểu là $1 và '0'.

Ví dụ dưới đây sẽ in ra nội dung của tất cả các tham số truyền cho script. Số tham số có thể lớn hơn 10.

Ví dụ 3-28 *using_shift.sh*

```
#!/bin/sh

while [ "$1" != "" ]; do
  echo "$1"
  shift
done
exit 0
```

Kết quả kết xuất

$./using_shift.sh here is a long parameter with 1 2 3 4 5

here
is
a
long
parameter
. . .
5
6

Cách chương trình làm việc:.

Chương trình tiếp nhận và in ra tham số dòng lệnh chỉ bằng biến **$1**. Mỗi lần nhận được nội dung của biến, ta dịch chuyển các tham số về trái một vị trí, bằng cách này biến $2 chuyển cho $1, $3 chuyển cho $2 ... vòng lặp **while** kiểm tra cho đến khi nào $1 bằng rỗng, có nghĩa là không còn tham số nào để nhận nữa thì dừng lại.

3.6.14. trap

Lệnh **trap** dùng để bẫy một tín hiệu (signal) do hệ thống gửi đến cho shell trong quá trình thực thi script. Tín hiệu thường là một thông điệp của hệ thống gửi đến chương trình yêu cầu hay thông báo về công việc nào đó mà hệ thống sẽ thực hiện. Ví dụ INT thường được gửi khi người dùng nhấn Ctrl-C để ngắt chương trình. TERM là tín hiệu khi hệ thống shutdown ... Chúng ta sẽ đi sâu vào cơ chế của việc gởi/nhận và xử lý tín hiệu

giữa chương trình và hệ thống trong phần sau.
Với trap trong script, ta đón và xử lí một số tín hiệu thường xảy ra sau đây:

Tín hiệu (signal)	**Ý nghĩa**
HUP (1)	Hang-up. nhận được khi người dùng logout.
INT (2)	Interrupt. Tín hiệu ngắt được gửi khi người dùng nhấn Ctrl-C.
QUIT (3)	Tín hiệu thoát, nhận khi người dùng nhấn Ctrl-C
ABRT (6)	Abort - Tín hiệu chấm dứt, nhận khi đạt thời gian quá hạn (Timeout)
ALRM (14)	Alarm -Tín hiệu thông báo được đùng xử lý cho tình huống timeout
TERM (15)	Terminate - Tín hiệu nhận được khi hệ thông yêu cầu shutdown.

Tín hiệu là các hằng được định nghĩa trong tập tin signal.h. Để sử dụng các hằng này trong các chương trình C, bạn có thể dùng #include <signal.h>. Muốn biết chi tiết hơn, bạn có thể dùng ngay lệnh trap -l như sau

```
$ trap -l
1) SIGHUP     2) SIGIN      3) SIGQUIT          4) SIGILL      5) SIGTRAP
6) SIGABRT    7) SIGBUS     8) SIGFPE
```

Bên trong script trap được sử đụng theo cú pháp sau:

trap command signal

Khi tín hiệu signal xảy ra thì command sẽ được gọi thực thi bất kể chương trình đang ở dòng lệnh nào. Do script thường được diễn dịch lệnh theo thứ tự từ trên xuống dưới nên bẫy trap thường đặt ngay đầu script.

Để giải trừ hoặc vô hiệu hoá lệnh **trap** trước đó, hãy thay command bằng **-** . Muốn đặt **trap** nhưng không cần xử lý tín hiệu nhận được, đặt command bằng chuỗi **“**. Lệnh trap không tham số sẽ hiển thị danh sách các tín hiệu do script đặt bẫy nếu có.
Dưới đây là một ví dụ minh họa về cách sử dụng trap trong script

Ví dụ 3-29 ***use_trap.sh***

```
#!/bin/sh

trap 'rm -f /tmp/my_tmp_file_$$' INT        #đặt bẫy
echo creating file /tmp/my_tmp_file_$$
date > /tmp/my_tmp_file_$$

echo "Press interrupt (Ctrl-C) to interrupt...."
while [ -f /tmp/my_tmp_file_$$ ]; do
  echo File exists
  sleep 1
done
echo The file no longer exists

# Bỏ trap đối với INT
trap -INT                                   #giải trừ bẫy
echo creating file /tmp/my_tmp_file_$$
```

```
date > /tmp/my_tmp_file_$$

echo "Press interrupt (Ctrl-C) to interrupt...."
while [ -f /tmp/my_tmp_file_$$ ]; do
  echo File exists
  sleep 1
done

echo We never get here
exit 0
```

Kết xuất của chương trình :

```
$./use_trap.sh
Creating file /tmp/my_tmp_file_131
Press interrupt (Ctrl-c) to interrupt. . . .
File exists
File exists
File exists
The file no longer exists
creating file /tmp/my_tmp_file_131
File exists
File exists
File exists
```

Cách chương trình làm việc

Vào đầu script ta đặt bẫy **trap**, yêu cầu khi nhận được tín hiệu INT (người dùng nhấn Ctrl-c) thì thực hiện: xóa tệp tạm bằng lệnh **'rm –f /mp/my_tmp_file_$$'** , lưu ý lệnh yêu cầu **trap** thực hiện lệnh, nên đặt lệnh đo trong dấu nháy đơn. $$ chính là biến môi trường trả về số **id** của tiến trình shell. Số này là số duy nhất được dùng để ghép làm tên tệp tạm. Lệnh **date > /tmp/my_tmp_file_$$** sẽ đưa nội dung ngày giờ hiện hành vào tệp tạm. Tiếp theo, vòng lặp **while** được gọi và lặp liên tục để kiểm tra sự tồn tại của tệp tạm. Mỗi lần lặp, chương trình in ra chuỗi thông báo File exists. Bây giờ nếu ta nhấn Ctrl-c, điều gì sẽ xảy ra ? Thông thường chương trình sẽ chấm dứt, tuy nhiên do lệnh bẫy **trap** "…" **INT** đã được đặt, cho nên khi Ctrl-C được nhấn, tín hiệu INT được gửi đến shell đang chạy script. Lệnh **rm** được gọi để xóa file tạm. Chương trình trong trường hợp này vẫn tiếp tục hoạt động. Tuy nhiên khi vòng lặp **while** không tìm thấy file tạm nữa, nó sẽ thoát ra ngoài. Lệnh kế tiếp là **trap -INT** được dùng để xóa bẫy trước đó. Chúng ta tạo lại flle tạm và bước vào vòng lặp kiểm tra sự tồn tại của tệp này. Trong lúc này nếu nhấn Ctrl-C, chương trình sẽ ngắt ngang, vì bẫy INT không còn hiệu lực nữa, ý nghĩa cuả Ctrl-C sẽ là xử lí mặc định của Hệ điều hành, tức chấm dứt shell khi người dùng nhấn Ctrl-C, do vậy lệnh **echo** và **exit** sau cùng sẽ không bao giờ được gọi đến.

3.6.15. unset

Lệnh **unset** dùng để loại bỏ biến khỏi môi trường shell. Ví dụ:

```
#!/bin/sh
foo="Hello World"
```

echo $foo
unset foo
echo $foo

Đầu tiên e cho sẽ in ra chuỗi Hello World, sau lệnh **unset** echo sẽ in ra chuỗi rỗng. Lý do, biến foo không còn tồn tại nữa.
Có thể gán chuỗi rỗng cho foo theo cách foo = . Tuy nhiên foo bằng rỗng và foo bị loại khỏi môi trường là khác nhau. Đối với DOS, biến môi trường bằng rỗng cũng đồng nghĩa với việc loại biến đó ra khỏi môi trường, còn Linux thì không. (unset không thường được sử dụng lắm).

3.7. Lấy về kết quả của một lệnh

Khi viết lệnh cho script chúng ta thường có nhu cầu lầy vế kết xuất hay kết quả của một lệnh để dùng cho lệnh tiếp theo. Ví dụ, ta gọi thực thi một lệnh và muốn lấy kết quả trả về của lệnh làm nội đung lưu trữ vào biến. Ta có thể làm điều này dựa vào lệnh có cú pháp **$(command)** (chúng ta đã gặp ở ví dụ khi minh họa lệnh **set**). Cú pháp này còn có thể dùng ở dạng khác là **`command** (lưu ý về dầu nháy ` là phím nằm chung với ký tự ~ chứ không phái là nháy đơn thông thường, dấu này được gọi là dấu bao ngược - backquote).

Kết qủa của **$(command)** đơn giản chỉ là kết xuất của **command**. Nó không phải là mã lỗi trả về của lệnh.

Ví dụ 3-30 use_command.sh

```
#!/bin/sh
echo Current directory is $PWD
echo It contents $(ls –a) files
exit 0
```

Nếu đang ở thư mục /root và thư mục này có các the .bash_profile, use_command.sh, thì kết quả kết xuất sẽ như sau:

$./use_command.sh
Current directory is /root
It contents $ (ls -a) .bash_profile use_command.sh files.

Cách chương trình làm việc

Lệnh ls -a dùng liệt kê nội dung thư mục /root. Kết quả trả về được đặt trong $ ( ) sẽ được diễn dịch thành nội dung:

$ (ls -a), ch kết quả: $ (.bash_profile use_conunand.sh)

Nội dung bên trong $ ( ) được xem là một chuỗi biến thông thường sau khi lệnh ls -a thực thi. Thực tế lệnh $ ( ) rất mạnh và sử dụng khá phổ biến trong lập trình shel1 cho Linux. Ta có thể lấy kết quả của $ ( ) làm đối số truyền tiếp cho các lệnh khác. Hãy xem lại ví dụ về **set** ở phần trên.

3.7.1. Ước lượng toán học

Trước đây đã thấy cách ước lượng biểu thức bằng lệnh **expr** (xem mục 3-6.9) tuy

nhiên expr thường thực thi chậm và không hiệu quả. Các shell mới cung cấp cú pháp **$((...))** dùng ước lượng biểu thức bên trong (...) thay cho lệnh **expr**. Ví dụ đơn giản cho thấy cách sử dụng cú pháp này như sau:

Ví dụ: 3-31 ***evaluate.sh***

```
#!/bin/sh
x=0
while [ "$x" -ne 10 ] ; do
   echo $x
   x=$(($x + 1))
done
exit 0
```

Đoạn lệnh trên sẽ liên tục in ra 10 số nguyên. Chúng ta tăng giá trị x bằng phép tính cộng thông qua lệnh ước lượng $ ((. . .)) .

3.7.2. Mở rộng tham số

Hình như shel1 không cung cấp cấu trúc mảng ? Đúng vậy, nhưng có một cách khác cũng tương tự có thể sử dụng, đó là việc thay thế tên biến, còn gọi là mở rộng tham số.

Hãy xét ví dụ sau:

```
#!/bin/sh
l_tmp ="Hello"
2_tmp="There"
3_tmp=:"World"
for i in 1 2 3
do
        echo $i_tmp
done
```

Ta có 3 biến 1_tmp, 2_tmp, 3_tmp, vòng lặp for trong script dự định chỉ dùng một lệnh echo để in ra nội dung của cả 3 biến một cách tuần tự. Thế nhưng kết quả kết xuất trái lại, sẽ nhận được 3 chuỗi rỗng. Hiển nhiên ! bởi vì lệnh echo $i_tmp in ra nội dung của biến mang tên i_tmp chứ không phải 1_tmp hay 2_tmp như ta mong muốn. Do i_tmp chưa có, nên kết qủa là những dòng trắng được in ra.

Shell cung cấp cú phập **{ }** để bọc các phần của biến mà ta muốn thay thế. Lệnh **for** giờ đây có thể hiệu chỉnh lại để chạy như mong muốn theo cách sau:

```
for i in 1 2 3
do
        echo ${i}_tmp
done
```

Kết qủa khi duyệt for lệnh echo sẽ thay thế ${i} bằng 1, 2 và 3 , kết qủa echo cho in ra đủ nội dung của 3 biến 1_tmp, 2_tmp và 3_tmp.

Shell cho rất nhiều cách thay thế tham số khác nhau. Thực sự thì thay thế tham số là công cụ rất mạnh và dùng rất nhiều trong các script chuyên nghiệp. Dưới đây là bảng

trình bầy một số phương pháp mở rộng và thay thế tham số rất hữu ích.

Mở rộng	Ý nghĩa
${param:-default}	Nếu giá trị của param=null thì gán trị mới là default
${#param}	Trả về chiều đài của param.
${param%word}	Bắt đầu từ cuối chuỗi param loại bỏ chuỗi con ngắn nhất của param so khớp với chuỗi word. Trả về kết quả là chuỗi còn lại .
${param%%wọrd}	Bắt đầu từ cuối chuỗi param loại bỏ chuỗi con dài nhất của param so khớp với chuỗi word. Trả về kết quả là chuỗi còn lại.
$({param#word}	Bắt đầu từ đầu chuỗi param loại bỏ chuỗi con nắần nhất của param so khớp với chuỗi word. Trả về kết quả là chuỗi còn lại.
${param##word}	Bắt đầu từ đầu chuỗi param loại bỏ chuỗi con dài nhất của param so khớp với chuỗi word. Trả về kết quả là chuỗi còn lại.

Việc thay thế chuỗi trong tham số đặc biệt hữu dụng trong trường hợp xử lý tên đường dẫn, tập tin, thư mục như trong ví dụ sau:

Ví du 3-32 ***param_expansion.sh***

```
#!/bin/sh

unset foo
echo ${foo:-bar}

foo=fud
echo ${foo:-bar}

foo=/usr/bin/X11/startx
echo ${foo#*/}
echo ${foo##*/}

bar=/usr/local/etc/local/networks
echo ${bar%local*}
echo ${bar%%local*}

exit 0
```

Kết xuất sẽ là

```
$./param_expansion.sh
bar
fud
usr/bin/X11/startx
startx
/usr/local/etc
```

/usr

Cách chương trình làm việc:

Lệnh đầu tiên ${foo:-bar} gán trị bar cho biến foo bởi hiện tại foo đang là rỗng vào lúc thực thi lệnh. Tuy nhiên sau khi gán foo=fud thì lệnh ${foo:-bar} không làm thay đổi trị của foo.

Ta có thể có một vài cách mở rộng sau:
${foo:=bar} sẽ trả về trị bar nếu foo tồn tại trước đó và foo đang chứa trị khác rỗng. Ngược lại trị trả về sẽ là bar và nội dung bar cũng sẽ được gán cho foo.
${foo:?bar} sẽ in ra foo: bar và chấm dứt lệnh thực thi nếu biến foo chưa tồn tại hoặc foo đang là rỗng.
Cuối cùng, ${foo:+barr} trả về trị bar nếu foo tồn tại và khác rỗng.

Lệnh **{foo#*/ }** so khớp và chỉ loại bỏ ký tự / bên trái của biến chuỗi (* mang ý nghĩa đại diện, nó có thể dùng để so khớp một hoặc nhiều ký tự).

Lệnh **{foo##* /}** so khớp và loại bỏ càng nhiều ký tự càng tốt, việc loại bỏ bắt đầu từ bên phải đối với tất cả ký tự đứng trước dấu / sau cùng.

Lệnh echo **${bar%local*}** so khớp và loại bỏ các ký tự bắt đầu từ bên phải cho đến khi gặp chuỗi local đầu tiên xuất hiện.

Ngược lại **echo ${bar%%local*}** sẽ cố gắng loại bỏ càng nhiều ký tự càng tốt cho đến khi gặp chuỗi local sau cùng.

Một ví dụ khác cho thấy cách sử dụng thay thế chuỗi trong tham số rất hữu dụng như sau:

cjpeg là chương trình chuyển ảnh ,gif thành ánh .jpg. cpeg được sủ dụng như sau:

$cjpeg image.gif > image.jpg

Nếu muốn chuyển đồng loạt tất cá các flle *.gif trong thư mục thành *.jpg thì sao. Hãy dùng script sau:

Ví du 3-33 *giftojpg.sh*

```
#!/bin/sh

for image in *.gif
do
   cjpeg $image > ${image%%gif}jpg
done
```

3.8. Tài liệu Here

UNIX và Linux cung cấp cơ chế tự động hóa mô phỏng việc nhập liệu gõ vào từ bàn phím bằng tài liệu here (Here Document). Ta để sẵn các phím hay chuỗi cần gõ trong một tập tin và chuyển hướng tập tin này cho lệnh cần thực thi. Nó sẽ tiếp nhận và đọc nội dung tập tin như những gì đã gõ vào từ bàn phím.

Ví dụ, khi gõ lệnh **cat**, nó sẽ chờ nhập dữ liệu gõ vào từ bàn phím. Nếu khi script thực thi không có mặt người dùng ở đó thì sao? Quá trình tự động của script sẽ dừng lại chờ đến khi ta xuất hiện để gõ dữ liệu vào. Cơ chế tài liệu here giúp thực hiện tự động nhập liệu như sau:

Ví dụ 3-34 ***cat_here.sh***

```
#! /bin/sh

cat > test.txt <<!YOURLABEL!
Hello
This is
here document
!YOURLABEL!
```

Kết qủa khi thực thi cat_here.sh, tệp test.txt được tạo ra. Với nội dung là chuỗi Hello This is ... ta không cần phải dùng tay nhập liệu cho lệnh **cat.**

Tài liệu **here** yêu cầu đặt cú pháp ở giữa nhãn bắt đầu và nhãn kết thúc. Trong ví dụ trên, nhãn bắt đầu là !YOURLABEL! (lưu ý đến ký tự **<<** ở đầu dùng để cho biết nơi bắt đầu của tài liệu Here), nhãn kết thúc là !YOURLABEL!. Dấu **!** hai bên nhãn YOURLABEL chỉ để dễ dàng nhận ra nhãn mà thôi, trong trường hợp nội dung dữ liệu của có chuỗi YOURLABEL thì cặp **! !** cũng dùng để phân biệt riêng tên nhãn của người dùng.

Có thể dùng hàng loạt lệnh echo đề chuyển hướng kết xuất ra file. Tuy nhiên tài liệu here hữu dụng và tiện lợi hơn. Here không đơn thuần chỉ chuyển nhập liệu ra file, nó cung cấp khả năng tương tác với chính chương trình ứng dụng. Ví dụ, khi cần soạn thảo văn bản bằng lệnh **ed**, các thao tác cần làm là gõ **ed** từ dòng lệnh để hiện cửa sổ soạn thảo. Nhập vào dữ liệu văn bản thực hiện chỉnh sửa, xóa, sau đó chấm dứt, đóng màn hình soạn thảo trở về dấu nhắc. Thực hiện với tài liệu here ta sẽ làm như sau:

Hãy tạo tập tin a_text_flle.txt có nội dung
This is line 1
This is line 2
This is line 3
This is line 4

script auto_edit.sh dưới đây sẽ mở trình **ed** và loại bỏ dòng 3 trong tệp văn bản vừa tạo, thay đổi và chỉnh sửa nội dung dòng 4.

Ví dụ 3-35 ***auto_edit.sh***

```
#!/bin/sh .

ed a_text_file << !AutoEdit!
3
d
.,\$s/is/was
w
q
!AutoEdit!
```

Kết qủa kết xuất là nội dung a_text_flle.txt sẽ bị đổi thành
This ịs line 1

This is line 2
This was line 4

Cách chương trình làm việc

Sau khi chuyển nội dung tệp cho **ed** bằng lệnh **ed a_text_file.txt** nội dung nằm giữa cặp nhãn !AutoEdit! đại diện cho các ký tự gõ vào từ bàn phím. Hãy lưu ý **\$s** được dùng để yêu cầu shell diễn dịch đây là chuỗi $s (một lệnh tìm kiếm của **ed**) chứ không phải biến mang tên s.

4. DÒ LỖI (DEBUG) CỦA SCRIPT

Vì scipt chỉ là lệnh văn bản được shell thông địch, cho nên việc dò lỗi không khó như các chương trình biên dịch nhị phân. Mặc dù vậy không có công cụ hay trình trợ giúp nào đặc biệt giúp thực hiện công việc này. Dưới đây là tổng hợp một số phương thức dò lỗi của script thường dùng.

Khi một lỗi xuất hiện, shell thường in ra số thứ tự của dòng gây lỗi. Ta có thể thêm vào lệnh echo để in ra nội dung của các biến có khá năng gãy lỗi cho chương trình, cũng có thể kiểm tra ngay các đoạn mã trực tiếp trên dòng lệnh để xem cách thức lệnh hoạt động thực tế có được shell chấp nhận hay không.

Cách chủ yếu và hay nhất là hãy để cho shell tự thực hiện công việc bắt lỗi bằng cách dùng lệnh **set** đặt một số tùy chọn cho shell hoặc đặt thêm tham số khi gọi shell thực thi script như sau:

Tham số dòng lệnh cho shell	**Tùy chọn**	**Ý nghĩa**
sh -n <script>	set –o noexec set -n	Chỉ kiểm tra cú pháp không thực thi lệnh
sh -v <script>	set -o verbose sự -v	Hiển thị lệnh trước khi thực hiện
sh –x	set -o xtrace set -x	Hiển thị lệnh sau khi đã thực thi lệnh
	set -o nounset set –u	Hiển thị thông báo lỗi khi một biến sử dụng nhưng chưa được định nghĩa.

Lệnh **set** cho phép dùng khóa chuyển **-o** và **+o** để bật tắt cờ tùy chọn.
Cũng có thể dùng lệnh **trap** để bẫy tín hiệu thoát EXIT và in ra nội dung của một biến nào đó. Ví dụ:

trap 'echo exiting : error variable = $problem_var' EXIT.

5. HIỂN THỊ MÀU SẮC (COLOR)

Khi đã bắt đầu quen với lập trình trên Linux, phần tiếp theo sẽ là vấn đề về màu sắc. Đơn gián ngôn ngữ lập trình script chỉ cung cấp lệnh *echo* hay *printf* để in một chuỗi

ra màn hình console trắng đen mà thôi. Lập trình liên quan đến màu sắc phải sử dụng đến ngôn ngữ biên dịch như C/c++ chăng ? Không hẳn thế, với script, người lập trình có thể hiển thị đủ mọi sắc màu mà card màn hình và máy tính hỗ trợ.

5.1. Màu chữ

Hãy để ý đến lệnh *ls* khi sử dụng Linux. *ls* có thể liệt kê tên thư mục với rất nhiều màu sắc bắt mắt, ví dụ các tập tin thực thi được hiển thị bằng màu xanh lá cây, tập tin nén là màu đỏ, tệp thông thường là màu trắng, tên tệp ảnh như *.gif hay *.jpg là màu hồng ... Không chỉ có lệnh *ls*, lệnh *echo* cũng có thể thực hiện được điều này. Đơn giản chỉ cần thêm vào chuỗi kết xuất của lệnh *echo* ký hiệu điều khiển escape cộng với số hiệu của màu muốn thể hiện cho chuỗi trên màn hình. Hãy thử gõ chuỗi sau từ dòng lệnh:

```
$echo -e " \033 [35m Hello Color ~033 [0m"
```

Kết quả ? Chuỗi Hello Color xuất hiện với màu hồng. Điều này là do mã điều khiển escape\033 thực hiện. Mã này tương đương với chuỗi ^[hay số hexa 0x1B. Khi Linux xử lý lệnh và nhận được chuỗi điều khiển này, nó sẽ xem các ký tự chuỗi theo sau là một dây lệnh điều khiển. Những lệnh này có thể làm được rất nhiều chuyện. Ở đây ta có thể lợi dụng, yêu cầu Linux xử lý màu sắc của chuỗi văn bản kết xuất bằng lệnh [m tiếp theo. Số 32 trước m thể hiện màu chữ. Các ký tự theo sau m là văn bản sẽ kết xuất. Nếu muốn khôi phục về trạng thái màu ban đầu, dùng chuỗi [0m. Do vậy trong lệnh:

```
$echo -e "\033[35mHello Color \033 [0m"
```

Cách diễn giải như sau: hãy thực hiện lệnh (\033) in chuỗi màu hồng ([35m) nội dung chuỗi là Hello Color, sau đó hãy khôi phục trở lại màu ban đầu [0m. Nếu không khôi phục về trạng thái màu trước đó thì có thể một số lệnh chuẩn sau đó sẽ kết xuất với cùng màu chữ mà đã định.

Ví dụ:

```
$echo -e "\033 [32m This is green text"
$echo -e "And this"
$echo -e " \033[0m Now we are back to normal."
```

Nếu muốn, có thể in các màu phối hợp với nhau trong cùng một chuỗi của dòng như ví dụ sau:

```
$echo - e "\033[032m Green text \033[34m and Blue "
```

Chuỗi điều khiển escape không chỉ giới hạn dùng trong script, hàm printf của C cũng có thể thực hiện được điều này. Nếu muốn hiển thị màu sắc ra màn hình console đơn giản không cần dùng thêm thư viện nào cả, trong C có thể viết:

```
printf ("l\033[34m This is blue . \033 [0m\n" );
```

Hay trong Perl:

```
Printf "\033[34m This is blue. \033 [0m\n" ;
```

Một số màu chữ chuẩn có thể sử dụng được liệt kê trong bản sau:

Mã điều khiển	Màu
30	Đen
31	Đỏ
32	Xanh lá cây
33	Nâu
34	Xanh nước biển

35	Hồng
36	Xanh da trời
37	Xám

Ví dụ sau sẽ in ra một loạt các màu để tham khảo

```
for color in 30 31 32 33 34 35 36 37
echo -e "\033 [$Colorm This is color text"
done
echo -e "\033 [0m"
```

5.2. Thuộc tính văn bản

Còn có thể kết hợp các thuộc tính cho phép chữ đậm (bold) hay nhấp nháy (blink) với nhau. Các thuộc tính kết hợp được đặt cách nhau bằng đầu chấm phẩy (;).
Ví dụ lệnh sau sẽ in ra chuỗi đậm màu nâu:

```
$echo -e "\033[33;1m This is bold ana red text \033[0m"
```

Đối với màn hình EGA thường thuộc tính bold làm cho chữ chuyển sang màu sáng. Ví dụ như màu nâu sẽ chuyển sang màu vàng, màu xám sẽ chuyển sang sáng trắng. Một vài thuộc tính khác khá thông dụng như: 0 đặt về thuộc tính bình thường, 5 đặt thuộc tính nhấp nháy, 7 đảo màu, 25 tất màu nhấp nháy ...

5.3. Màu nền

Có thể đất màu nền cho chuỗi kết xuất thay cho nền đen của màn hình console. Ví dụ, đặt chữ đỏ trên nền trắng như sau:

```
$echo -e “\033 [47 ; 31m Red on white. \033 [Om"
```

Ở đây đặt màu nến và màu chữ cách nhau bằng dấu chấm phẩy (;). Dưới đây là danh sách các màu nền sử dụng hầu hết trên các màn hình console

Mã điều khiển	Màu
40	Đen
41	Đỏ
42	Xanh lá cây
43	Nâu
44	Xanh nước biển
45	Hồng
46	Xanh da trời
47	Trắng

Ví dụ

```
$echo -e “\033 [46;37 Gray blue \033 [47;30 Black and white"
```

6. KẾT CHƯƠNG

Chương giới thiệu các kỉ thuật cơ bản cách lập trình shell, một thế mạnh truyền thống của UNIX/Linux. Làm quen với lập trình shell là làm quen với công cụ quản trị hệ thống, mà các nhà chuyên nghiệp hay sử dụng, trong khi kết hợp với các tiện ích tạo ra từ C/C++ hay Perl, PHP …Có thể nói lập trình shell là không thể thiếu được khi sử dụng UNIX/Linux. Tài liệu này chỉ như phần dạo đầu, có thể tìm hiểu sâu ở các sách chuyên cho shell trong môi trường UNIX.

7. MỘT SỐ TÓM TẮT và VÍ DỤ

Để tiện thực hành, dưới đây ta quy ước sẽ sử dụng shell mặc định của Linux là bash (Bourne Again Shell). Như tên của nó đã nói rõ bash rất giống Bourne shell của UNIX, dấu nhắc cũng là $ nên không cần viết lại ở đây nữa.

7.1 Tạo và chạy các chương trình shell

Nói một cách đơn giản nhất, các chương trình shell chỉ là các tệp chứa một hoặc nhiều câu lệnh của shell hoặc của hệ thống, kể cả trình ứng dụng. Các tệp này còn được gọi là các tệp kịch bản (*script*) và việc viết các chương trình shell còn được gọi là viết kịch bản (*scripting*). Các chương trình shell thường được dùng để:

-làm đơn giản hoá các tác vụ lặp đi lặp lại

-thay thế một hoặc nhiều câu lệnh luôn luôn được thực hiện cùng nhau bằng một câu lệnh duy nhất

-tự động hoá quá trình cài đặt

-viết một ứng dụng tương tác đơn giản

7.1.1 Tạo một chương trình shell

Ví dụ, có một ổ CD-ROM được mount trên hệ Linux của ta và giả sử ta đang đọc dữ liệu từ một đĩa CD nằm trong ổ đó. Nếu muốn đổi đĩa CD khác, ta phải làm cho Linux đọc nội dung thư mục của đĩa CD mới bằng cách: đầu tiên unmount ổ CD-ROM bằng lệnh `umount` để lấy đĩa CD cũ ra; sau đó đưa đĩa mới vào và mount đĩa đó bằng lệnh `mount`. Chuỗi lệnh đó như sau:

```
umount /dev/cdrom
mount –t iso1960 /dev/cdrom /cdrom
```

Thay vì phải gõ cả hai câu lệnh này mỗi khi ta muốn thay đổi đĩa CD, ta có thể tạo ra một chương trình shell như sau:

-tạo ra một tệp text bằng một trình soạn thảo văn bản (`vi`, `emacs`...)

-gõ hai dòng lệnh trên vào tệp đó

-ghi (save) và đặt tên tệp là `remount` (hoặc bất cứ tên nào mà ta muốn)

7.1.2 Chạy chương trình shell

Có một vài cách thực hiện các lệnh liên quan đến tệp ở ví dụ trên (`remount`).

a. Cách thứ nhất

Làm cho tệp có thuộc tính `eXecutable` (khả thi) bằng câu lệnh sau:

```
chmod +x remount
```

Lệnh này đổi thuộc tính của tệp sao cho nó trở nên khả thi. Vì vậy, sau đó chỉ cần gõ vào dòng lệnh tên tệp:

```
remount
```

để chạy chương trình shell mới này của ta.

Lưu ý 1:

> Chương trình shell **`remount`** phải nằm trong thư mục có trong đường dẫn tìm kiếm (search path), nếu không thì shell không thể tìm ra chương trình này để thực thi nó.

Lưu ý 2:

> Dòng đầu tiên của một tệp kịch bản thường bắt đầu bởi hai ký tự **`#!`** và theo sau là tên của chương trình thông dịch (interpret) nội dung của tệp. Ví dụ nếu dòng đầu tiên là **`#!/bin/bash`** thì nội dung của tệp kịch bản đó được thực hiện như một chương trình shell của **`bash`**.

b. Cách thứ hai

Nếu chương trình của ta được viết cho shell nào thì hãy chạy shell đó với tham số là tên tệp chương trình.

Ví dụ, chương trình `remount` được viết cho shell `tcsh`, vì vậy để chạy nó hãy gõ :

```
tcsh remount
```

Câu lệnh trên khởi động một shell mới (`tcsh`) và bắt nó thực hiện các câu lệnh có trong tệp `remount`.

c. Cách thứ ba

Đối với các shell `pdksh` và `bash`, hãy thực hiện câu lệnh `.` với tham số là tên của chương trình shell:

```
. remount
```

Câu lệnh `.` bắt shell hiện hành (`pdksh` hoặc `bash`) thực hiện các câu lệnh có trong `remount`.

Tương tự, đối với shell `tcsh`, câu lệnh `.` được thay thế bởi câu lệnh `source` :

```
source remount
```

7.2 Sử dụng biến

Giống như các ngôn ngữ máy tính khác, việc sử dụng các biến (variable) trong lập trình shell rất quan trọng. Trong các bài trước chúng ta đã làm quen với một số biến môi trường như `PATH` và `PS1`.

7.2.1 Gán một giá trị cho biến

Đối với shell `pdksh` và `bash`, để gán giá trị cho một biến, ta gõ tên của biến theo sau là một dấu bằng (`=`) và giá trị mà ta muốn gán cho biến. Ví dụ:

```
bien=5
```

Với shell `tcsh`, để gán giá trị cho biến ta thêm từ set như sau:

```
set bien = 5
```

Lưu ý:

> Đối với `bash` và `pdksh`, cú pháp gán không chấp nhận các ký tự trống (space) phía trước và sau dấu bằng, còn với `tcsh` thì trước và sau dấu bằng có thể có ký tự trống hoặc không.

Không giống các ngôn ngữ lập trình như C hoặc Pascal, ta không phải khai báo các biến trong shell. Vì biến shell không có kiểu (*type*) xác định, ta có thể dùng một biến để gán một giá trị nguyên (*integer*) sau đó lại gán cho biến đó một giá trị chuỗi (*string*). Ví dụ, sau khi gán biến `bien` bằng một giá trị số (`5`) như trong ví dụ trên ta có thể tiếp tục gán như sau:

`bien=Linux` (`pdksh` hoặc `bash`)

`set bien = Linux` (`tcsh`)

7.2.2 Truy nhập giá trị của một biến

Sau khi đã gán giá trị cho biến, để truy nhập giá trị của biến đó trong chương trình shell, hãy thêm dấu đôla (`$`) vào phía trước tên của biến.

Tên của biến trong các ví dụ trên là `bien`, còn giá trị mà biến đó mang là `$bien` (là chuỗi `Linux`). Để in giá trị đó ra màn hình ta có thể sử dụng lệnh `echo` như sau:

```
echo $bien
```

Lưu ý:

> Nếu bỏ qua dấu đôla trong câu lệnh trên (thành **`echo bien`**), shell hiểu **`bien`** là một chuỗi và sẽ in chuỗi đó ra màn hình (chứ không phải chuỗi **`Linux`**).

7.2.3 Tham số vị trí và biến xây dựng sẵn trong shell

Ta có thể truyền các tham số cho chương trình shell qua dòng lệnh. Ví dụ, dòng lệnh sau:

```
remount thamso1 thamso2
```

cũng thực hiện chương trình shell `remount` nhưng có thêm hai tham số dòng lệnh (còn gọi là tuỳ chọn dòng lệnh) là `thamso1` và `thamso2`.

Khi ta chạy một chương trình shell có hỗ trợ các tuỳ chọn dòng lệnh như trên thì mỗi tuỳ chọn được lưu vào trong một tham số vị trí (*positional parameter*). Tham số đầu tiên được lưu vào một biến có tên là `1`, tham số thứ hai được lưu vào biến có tên là `2`... Các shell hiện thời có thể hỗ trợ đến 9 biến như vậy. Để truy nhập vào các biến này ta cũng thêm ký tự đôla vào trước tên biến (ví dụ `$1`, `$2`,..).

Chương trình shell sau nhận vào hai tuỳ chọn dòng lệnh và in ra màn hình tuỳ chọn thứ hai trước rồi mới in tham số thứ nhất sau:

```
# Chuong trinh in dao nguoc
echo "$2" "$1"
```

Nếu ta thực hiện chương trình này (đã đặt tên là `daonguoc`) trên dòng lệnh như sau:

```
daonguoc CHAO BAN
```

thì trên màn hình sẽ xuất hiện :

```
BAN CHAO
```

Các biến đặc biệt `$1`, `$2`,... còn gọi là các biến shell xây dựng sẵn (*built-in shell variable*). Còn một số biến shell xây dựng sẵn khác rất quan trọng trong lập trình shell. Bảng sau đây liệt kê các biến này.

Bảng 1. Các biến shell xây dựng sẵn

Biến	*Cách dùng*
`$#`	Lưu giữ số lượng các tham số dòng lệnh được truyền cho chương trình shell
`$?`	Lưu giữ giá trị trả về (exit code) của câu lệnh thực hiện sau cùng
`$0`	Lưu giữ từ (word) đầu tiên của dòng lệnh shell (chính là tên của chương trình shell)
`$*`	Lưu giữ toàn bộ các tham số trên dòng lệnh (`$1 $2`...)
`"$@"`	Lưu giữ toàn bộ các tham số trên dòng lệnh nhưng được bao bọc trong hai dấu nháy kép (`"$1" "$2"` ...)

7.2.4 Ký tự đặc biệt và cách thoát khỏi ký tự đặc biệt

Nếu ta muốn gán chuỗi `Xin chao` vào biến `loichao` mà làm như sau:

```
loichao=Xin chao
```

hoặc: `set loichao = Xin chao`

Kết quả sẽ không được như ý muốn. `bash` và `pdksh` sẽ không hiểu dòng lệnh và sẽ báo lỗi. `tcsh` chỉ gán chuỗi `Xin` cho biến `loichao`. Nguyên nhân là do ký tự dấu trống nằm giữa `Xin` và `chao` là ký tự đặc biệt đối với shell. Trong shell, các ký tự đặc biệt này không còn giữ nguyên nghĩa "đen" của chúng nữa. Vì vậy, dấu trống kể trên không còn giữ nguyên ý nghĩa dấu cách trắng giữa hai chuỗi `Xin` và `chao`, mà nó trở thành dấu hiệu phân cách các phần của dòng lệnh.

Để trở về nghĩa "đen" của các ký tự đặc biệt, ta phải thoát khỏi (*escape*) ý nghĩa hiện thời của chúng bằng các cách sau:

-Dùng cặp dấu nháy đơn (' ')

-Dùng cặp dấu nháy kép (" ")

-Dùng ký tự thoát: ký tự chéo ngược (\)

a. Cặp dấu nháy đơn

Chuỗi ký tự nằm giữa hai dấu nháy đơn sẽ được hiểu theo đúng nghĩa "đen" của nó. Trở về ví dụ trên, ta có thể thực hiện như sau:

```
loichao='Xin chao'
```

hoặc :

```
set loichao = 'Xin chao'
```

Lúc này biến `loichao` có giá trị đúng là chuỗi `Xin chao`.

b. Cặp dấu nháy kép

Cách sử dụng của dấu nháy kép cũng giống như dấu nháy đơn ngoại trừ một điểm là dấu nháy kép không thoát khỏi ý nghĩa đặc biệt của ký tự đôla. Điều đó có nghĩa là ta có thể đưa giá trị một biến vào trong một chuỗi. Ví dụ:

```
loichao="Xin chao $LOGNAME"
```

hoặc :

```
set loichao = "Xin chao $LOGNAME"
```

Nhắc lại, `LOGNAME` là biến môi trường lưu giữ tên người sử dụng đã đăng nhập vào hệ thống. Vì vậy nếu người sử dụng là `root` thì biến `loichao` sẽ có giá trị là `Xin chao root`. Nếu trong câu lệnh trên ta thay thế dấu nháy kép bằng dấu nháy đơn thì biến lại có giá trị là `Xin chao $LOGNAME`.

c. Ký tự chéo ngược

Ký tự chéo ngược (*backslash*) đi trước một ký tự đặc biệt sẽ làm thoát khỏi ý nghĩa đặc biệt của ký tự đó. Ví dụ:

```
loichao=Xin\ chao
```

hoặc:`set loichao = Xin\ chao`

sẽ gán `Xin chao` cho `loichao`, còn câu lệnh :

```
giatien=\$5
```

hoặc :

```
set giatien = \$5
```

sẽ gán biến `giatien` giá trị `$5` (5 đô la).

d. Ký tự nháy ngược

Ký tự nháy ngược hay dấu huyền (`) là một ký tự đặc biệt trong shell. Chuỗi ký tự nằm giữa hai dấu nháy ngược sẽ được xem là một câu lệnh, được thực hiện và giá trị trả về sẽ được lưu vào biến. Ví dụ:

```
noidung=`ls`
```

hoặc: `set noidung = `ls``

sẽ thực hiện lệnh `ls` và kết quả trả về sẽ lưu trong biến `noidung`.

7.2.5 Lệnh `test`

Trong `bash` và `pkdsh`, lệnh `test` dùng để đánh giá một biểu thức điều kiện. Người ta thường sử dụng lệnh này để đánh giá một điều kiện trong một mệnh đề điều kiện (`if`) hoặc trong một mệnh đề vòng lặp (`while`). Cú pháp của lệnh `test` như sau:

```
test bieu_thuc
```

hoặc: `[bieu_thuc]`

trong đó `bieu_thuc` là biểu thức điều kiện cần được đánh giá.

Lệnh `test` thường được dùng với một số toán tử đã được xây dựng sẵn trong shell. Các toán tử này có thể phân thành 4 nhóm sau:

-toán tử số nguyên

-toán tử chuỗi

-toán tử tệp

-toán tử logic

Sau khi thực hiện, lệnh `test` sẽ trả về giá trị logic là Đúng (True) hoặc Sai (False).

a. Toán tử số nguyên

Giả sử `int1` và `int2` là hai số nguyên. Các toán tử số nguyên và ý nghĩa của chúng sẽ được liệt kê trong bảng dưới đây.

Bảng 2. Các toán tử số nguyên của lệnh `test`

Toán tử	Ý nghĩa
`int1 -eq int2`	Trả về Đúng nếu `int1` bằng `int2`
`int1 -ge int2`	Trả về Đúng nếu `int1` lớn hơn hoặc bằng `int2`
`int1 -gt int2`	Trả về Đúng nếu `int1` lớn hơn `int2`
`int1 -le int2`	Trả về Đúng nếu `int1` nhỏ hơn hoặc bằng `int2`
`int1 -lt int2`	Trả về Đúng nếu `int1` nhỏ hơn `int2`
`int1 -ne int2`	Trả về Đúng nếu `int1` không bằng (khác) `int2`

b. Toán tử chuỗi

Toán tử chuỗi dùng để so sánh hai chuỗi ký tự. Giả sử có hai chuỗi ký tự `str1` và `str2`. Các toán tử chuỗi được liệt kê trong bảng sau:

Bảng 3. Các toán tử chuỗi của lệnh `test`

Toán tử	Ý nghĩa
`str1 = str2`	Trả về Đúng nếu `str1` giống (tương đồng) với `str2`

`str1 != str2`	Trả về Đúng nếu `str1` khác (không tương đồng) với `str2`
`str1`	Trả về Đúng nếu `str1` không rỗng
`-n str1`	Trả về Đúng nếu độ dài của `str1` lớn hơn 0
`-z str1`	Trả về Đúng nếu độ dài của `str1` bằng 0

c. Toán tử tệp

Toán tử tệp dùng để kiểm tra các thuộc tính của tệp. Giả sử ta có một tệp có tên là `tệpname` thì các toán tử tệp có thể dùng sẽ được liệt kê trong bảng sau:

Bảng 4. Các toán tử tệp của lệnh `test`

Toán tử	Ý nghĩa
`-d tệpname`	Trả về Đúng nếu `tệpname` là một thư mục
`-f tệpname`	Trả về Đúng nếu `tệpname` là một tệp thông thường
`-r tệpname`	Trả về Đúng nếu `tệpname` có thể đọc được
`-s tệpname`	Trả về Đúng nếu `tệpname` có độ dài lớn hơn 0
`-w tệpname`	Trả về Đúng nếu `tệpname` có thể ghi được
`-e tệpname`	Trả về Đúng nếu `tệpname` có thể thực thi được

d. Toán tử logic

Toán tử logic dùng để kết hợp một hoặc nhiều toán tử số nguyên, toán tử chuỗi và toán tử tệp hoặc đảo ngược kết quả của các toán tử trên. Giả sử `expr1` và `expr2` là các biểu thức logic (lấy được bằng cách sử dụng các toán tử). Các toán tử logic được liệt kê trong bảng sau:

Bảng 5. Các toán tử logic của lệnh `test`

Toán tử	Ý nghĩa
`! expr1`	Trả về Đúng nếu `expr1` là không đúng (Sai)
`expr1 -a expr2`	Trả về Đúng nếu cả `expr1` và `expr2` đều là Đúng
`expr1 -o expr2`	Trả về Đúng nếu hoặc `expr1` hoặc `expr2` là Đúng

7.3 Các hàm shell

Ngôn ngữ shell cho phép người sử dụng tự định nghĩa các hàm (*function*). Các hàm này sử dụng gần giống như các hàm trong C và các ngôn ngữ lập trình khác. Lưu ý Shell `tcsh` không hỗ trợ các hàm.

7.3.1 Cú pháp tạo hàm

Cú pháp tạo hàm của `bash` và `pdksh` như sau:

```
ten_ham () {
  cau_lenh_shell
```

```
   ...
  ;
 }
```

Ngoài ra `pdksh` còn cho phép một cú pháp tương đương như sau:

```
function ten_ham {
  cau_lenh_shell
   ...
  ;
 }
```

Sau khi đã định nghĩa hàm, ta có thể gọi nó bằng cách gõ dòng lệnh sau:

```
ten_ham [thamso1 thamso2 ...]
```

Lưu ý rằng ta có thể truyền bao nhiêu tham số cho hàm cũng được. Khi truyền các tham số cho một hàm, hàm sẽ coi các tham số này như các tham số vị trí (`$1=thamso1`, `$2=thamso2`, ...) giống như khi truyền các tham số dòng lệnh cho chương trình shell.

7.3.2 Các ví dụ tạo hàm

Ví dụ sau bao gồm nhiều hàm khác nhau, mỗi hàm thực hiện một nhiệm vụ tương ứng với một tuỳ chọn dòng lệnh. Chương trình này sẽ nhận các tuỳ chọn dòng lệnh:

-Tuỳ chọn thứ nhất chỉ ra thao tác thực hiện

-Tuỳ chọn thứ 2, ... là tên (các) tệp nhập vào

Dựa vào tuỳ chọn thứ nhất, chương trình thực hiện các thao tác sau:

-Tuỳ chọn `-u`: đọc (các) tệp vào, biến đổi nội dung của chúng thành chữ hoa, và ghi ra (các) tệp ra. Thao tác này do hàm `chu_hoa()` đảm nhiệm.

-Tuỳ chọn `-l`: đọc (các) tệp vào, biến đổi nội dung của chúng thành chữ thường, và ghi ra (các) tệp ra. Thao tác này do hàm `chu_thuong()` đảm nhiệm.

-Tuỳ chọn `-p`: đọc (các) tệp vào, và in nội dung của chúng ra. Thao tác này do hàm `in_ra()` đảm nhiệm.

Nếu không phải các tuỳ chọn trên: in ra cách sử dụng chương trình. Thao tác này do hàm `in_cachsd()` đảm nhiệm.

Các tệp ra có tên giống với các tệp vào nhưng có thêm phần mở rộng `.out`.

```
chu_hoa () {
  shift
   for i
   do
     tr a-z A-Z <$1 >$1.out
    rm $1
     mv $1.out $1
    shift
   done; }
```

```
 chu_thuong () {
  shift
   for i
   do
     tr A-Z a-z <$1 >$1.out
    rm $1
     mv $1.out $1
    shift
   done; }

 print () {
  shift
   for i
   do
     lpr $1
    shift
   done; }

 in_cachsd () {
  echo "Cu phap cua $1 là $1 [-u|-l|-p] <tệp_vao>"
   echo " "
  echo "u ... luu thanh cac tệp chu hoa"
   echo "l ... luu thanh cac tệp chu thuong"
  echo "p ... in cac tệp ra may in"; }

case $1
 in
p|-p)
   in_ra $@
  ;;
 u|-u)
  chu_hoa $@
   ;;
l|-l)
   chu_thuong $@
  ;;
 *)
  in_cachsd $0
   ;;
esac
```

7.4 Các mệnh đề điều kiện

Các mệnh đề điều kiện được dùng để thi hành các phần khác nhau của chương trình shell tuỳ thuộc vào từng điều kiện cụ thể. Cả `bash`, `pdksh` và `tcsh` đều có hai dạng mệnh đề điều kiện là mệnh đề `if` và mệnh đề `case`. Cú pháp của các mệnh đề này có khác biệt chút ít đối với các shell khác nhau.

7.4.1 Mệnh đề `if`

Cả 3 loại shell nói trên đều hỗ trợ mệnh đề điều kiện dạng `if-then-else`. Cú pháp của mệnh đề này có các dạng như sau :

a. Dạng đơn giản

`bash` và `pdksh`	`tcsh`

`if [bieu_thuc]`
`then`
`cau_lenh`
`...`
`fi`
	`if (bieu_thuc) then`
	`cau_lenh`
	`...`
	`endif`

Nếu biểu thức `bieu_thuc` được đánh giá là Đúng thì (các) câu lệnh `cau_lenh` sẽ được thực hiện, còn không thì chương trình sẽ bỏ qua và thực hiện ngay câu lệnh phía sau `fi` hoặc `endif`.

Nếu chỉ có một câu lệnh được thực hiện trong `if` thì `tcsh` còn có một dạng đơn giản hơn là :

```
if (bieu_thuc) cau_lenh
```

b. Dạng `if-else`

`bash` và `pdksh`	`tcsh`

`if [bieu_thuc]`
`then`
`cau_lenh`
`...`
`else`
`cau_lenh`
`...`
`fi`
	`if (bieu_thuc) then`
	`cau_lenh`
	`...`
	`else`
	`cau_lenh`
	`...`
	`endif`

Dạng này mở rộng dạng đơn giản nói trên ở chỗ: nếu `bieu_thuc` là Sai thì (các) câu lệnh `cau_lenh` sau `else` sẽ được thực hiện.

c. Dạng `else-if`

Nếu sau `else` còn tiến hành kiểm tra một điều kiện `bieu_thuc2` nữa thì người ta phải đưa thêm một mệnh đề `if` nữa vào trong khối mệnh đề `else`.

`bash` và `pdksh`	`tcsh`

```
if [bieu_thuc]
 then
  cau_lenh
   ...
elsif [bieu_thuc2]
 then
  cau_lenh
   ...
else
   cau_lenh
  ...
 fi
```

```
if (bieu_thuc) then
   cau_lenh
  ...
 else if (bieu_thuc2) then
  cau_lenh
   ...
else
   caulenh
  ...
 endif
```

d. Ví dụ

Ví dụ sau sẽ thực hiện kiểm tra tệp `tai_lieu` có nằm trong thư mục hiện tại không và in kết quả ra màn hình.

Đối với `bash` và `pdksh`:

```
if [ -f tai_lieu]
 then
  echo "Co tệp tai_lieu trong thu mục hien thoi"
 else
  echo "Khong tim thay tệp tai_lieu trong thu muc hien thoi"
 fi
```

Đối với `tcsh` (lưu ý phải có ký tự `#` ở đầu chương trình) :

```
 #
if ( { -f tai_lieu } ) then
   echo "Co tệp tai_lieu trong thu muc hien thoi"
else
   echo "Khong tim thay tệp tai_lieu trong thu muc hien thoi"
endif
```

7.4.2 Mệnh đề case

a. Cú pháp case

Mệnh đề `case` cho phép so một mẫu (chuỗi ký tự) với nhiều mẫu khác nhau và thực hiện đoạn mã tương ứng với mẫu trùng khớp. Cú pháp của nó như sau:

`bash` và `pdksh`	`tcsh`

`case mau in`
` mau1)`
`  cau_lenh`
`   ...`
`  ;;`
	`switch (mau)`
	` case mau1:`
	`  cau_lenh`
	`   ...`
	`  breaksw`

mau2)
  cau_lenh
  ...
  ;;
 	
...
*)
  cau_lenh
  ...
  ;;
esac
	case mau2:
	  cau_lenh
	  ...
	  breaksw
	
	...
	default:
	  cau_lenh
	  ...
	  breaksw
	endsw

Trong đó, `mau` được so sánh lần lượt với các mẫu `mau1`, `mau2`... Nếu có một mẫu trùng khớp thì (các) câu lệnh tương ứng sẽ được thực hiện cho đến khi gặp hai dấu chấm phảy (`;;`) (`bash` và `pdksh`) hoặc `breaksw` (`tcsh`). Nếu không có mẫu nào trùng khớp thì (các) câu lệnh trong khối `*` (`bash` và `pdksh`) hoặc `default` (`tcsh`) được thực hiện.

b. Ví dụ

Đoạn chương trình sau được viết cho `bash` hoặc `pdksh`. Nó kiểm tra xem tuỳ chọn dòng lệnh đầu tiên (lưu trong biến `$1`) có phải là `-i` hoặc `-e` không. Nếu là `-i` thì sẽ in ra số dòng trong tệp xác định bởi tuỳ chọn dòng lệnh thứ hai (biến `$2`) bắt đầu bằng chữ cái i. Còn nếu là `-e` thì sẽ in ra số dòng trong tệp xác định bởi tuỳ chọn dòng lệnh thứ hai bắt đầu bằng chữ cái e. Nếu tuỳ chọn đầu tiên không phải là `-i` hoặc `-e` thì in ra màn hình thông báo.

```
 case $1 in
-i)
   count=`grep ^i $2 | wc -l`
  echo "So dong trong $2 bat dau bang chu cai i la $count"
   ;;
-e)
   count=`grep ^e $2 | wc -l`
  echo "So dong trong $2 bat dau bang chu cai e la $count"
   ;;
*)
   echo "Tuy chon khong hop le"
  echo "Cach dung: $0 [-i|-e] tệpname"
   ;;
esac
```

Và sau đây là đoạn mã tương tự cho `tcsh` :

```
#
 switch ($i)
case -i | i:
   set count=`grep ^i $2 | wc -l`
```

```
  echo "So dong trong $2 bat dau bang chu cai i la $count"
   breaksw
case –e | e:
   set count=`grep ^e $2 | wc –l`
  echo "So dong trong $2 bat dau bang chu cai e la $count"
   breaksw
default:
   echo "Tuy chon khong hop le"
  echo "Cach dung: $0 [-i|-e] tệpname"
   breaksw
endsw
```

7.5 Các mệnh đề vòng lặp

Ngôn ngữ shell cũng cung cấp các mệnh đề vòng lặp. Vòng lặp hay được sử dụng nhất là vòng lặp `for`. Ngoài ra còn có các loại vòng lặp `while`, `until`.

7.5.1 Mệnh đề `for`

Mệnh đề `for` thực hiện các câu lệnh trong vòng lặp với một số lần nhất định. Nó có các dạng sau:

a. Dạng thứ nhất

`bash` và `pdksh`	`tcsh`

`for bien in danh_sach`
`do`
`  cau_lenh`
`  ...`
`done`
	`foreach bien (danh_sach)`
	`  cau_lenh`
	`  ...`
	`end`

Trong dạng này, mệnh đề `for` thực hiện mỗi vòng lặp cho mỗi mục trong danh sách `danh_sach`. Danh sách này có thể là một biến chứa các từ ngăn cách nhau bởi một dấu cách hoặc cũng có thể được gõ trực tiếp các từ đó vào dòng lệnh. Mỗi vòng lặp, biến `bien` được gán lần lượt một mục (từ) trong danh sách cho đến hết danh sách.

b. Dạng thứ hai

Đối với `bash` và `pdksh`, mệnh đề `for` còn có một dạng như sau:

```
for bien
do
  menh_de
  ...
done
```

Trong dạng này, mệnh đề `for` thực hiện mỗi vòng lặp cho mỗi mục trong biến `bien`. Khi cú pháp này được sử dụng, chương trình shell giả sử rằng biến `bien`

chứa mọi tham số vị trí đã được truyền cho chương trình thông qua dòng lệnh. Thông thường, dạng mệnh đề này tương đương với mệnh đề sau:

```
for bien in "$@"
do
  menh_de
  ...
done
```

c. Ví dụ mệnh đề `for`

Ví dụ sau (`bash` và `pdksh`) sẽ lấy các tuỳ chọn dòng lệnh là các tệp text. Đối với mỗi tệp, chương trình sẽ đọc và chuyển đổi các chữ thường thành chữ hoa và lưu vào một tệp mới có tên giống tệp cũ nhưng có thêm phần mở rộng `.caps`.

```
for tệp
do
  tr a-z A-Z < $tệp > $tệp.caps
done
```

Còn đây là ví dụ tương đương viết cho `tcsh`

```
#
foreach tệp ($*)
  tr a-z A-Z < $tệp > $tệp.caps
end
```

7.5.2 Mệnh đề `while`

a. Cú pháp của mệnh đề `while`

Mệnh đề `while` thực hiện đoạn chương trình bên trong chừng nào mà biểu thức đã cho còn là Đúng. Cú pháp của nó như sau:

`bash` và `pdksh`	`tcsh`

`while bieu_thuc`
`do`
`  menh_de`
`  ...`
`done`
	`while (bieu_thuc)`
	`  menh_de`
	`  ...`
	`end`

b. Ví dụ mệnh đề `while`

Ví dụ sau (`bash`, `pdksh`) liệt kê các tham số truyền cùng với số lượng tham số :

```
count=1
while [ -n "$*" ]
do
  echo "Day la tham so thu $count: $1"
  shift
  count=`expr $count + 1`
done
```

Ghi chú: Lệnh shift (xem thêm ở mục nhỏ cuối chương) sẽ dịch chuyển các tham số dòng lệnh sang một vị trí phía bên trái.

Còn đây là đoạn mã tương đương dành cho `tcsh` :

```
#
set count = 1
while ("$*" != "")
  echo "Day la tham so thu $count: $1"
  shift
  set count = `expr $count + 1`
end
```

7.5.3 Mệnh đề `until`

a. Cú pháp của mệnh đề `until`

Cú pháp của mệnh đề `until` giống với mệnh đề `while`. Điểm khác biệt là ở chỗ, mệnh đề `while` thực hiện vòng lặp chừng nào biểu thức điều kiện còn Đúng, còn mệnh đề `until` thực hiện vòng lặp chừng nào biểu thức điều kiện còn Sai. Cú pháp của nó trong `bash` và `pdksh` như sau :

```
until bieu_thuc
do
  cau_lenh
  ...
done
```

b. Ví dụ mệnh đề `until`

Ta viết lại ví dụ trên bằng vòng lặp `until` như sau:

```
count=1
until [ -z "$*" ]
  echo "Day la tham so thu $count: $1"
  shift
  count=`expr $count + 1`
done
```

Điểm khác biệt giữa 2 ví dụ chỉ là biểu thức điều kiện `-n "$*"` được thay bằng `-z "$*"` (ý nghĩa ngược lại). Đó là do mệnh đề `until` hoàn toàn giống với mệnh đề `while` khi đảo ngược điều kiện. Vì vậy `tcsh` không có mệnh đề này.

7.5.4 Câu lệnh `shift`

a. Giới thiệu câu lệnh `shift`

Cả `bash`, `pdksh` và `tcsh` đều hỗ trợ câu lệnh `shift`. Câu lệnh này dịch chuyển giá trị hiện thời lưu trong các tham số vị trí sang một vị trí về phía bến trái. Ví dụ nếu ta có :

```
$1=Xin $2=chao $3=ban
```

thì sau khi thực hiện lệnh `shift` ta có :

```
$1=chao $2=ban
```

Ta cũng có thể chuyển sang trái hơn một vị trí bằng cách thêm số bước dịch chuyển vào câu lệnh:

```
shift 2
```

Bằng cách dùng `shift` ta có thể lần lượt duyệt qua các tuỳ chọn dòng lệnh một cách dễ dàng. Vì vậy câu lệnh này rất hữu dụng trong việc phân tích các tuỳ chọn dòng lệnh.

b. Ví dụ câu lệnh `shift`

Ví dụ sau nhận các tệp: một tệp đầu vào (`-i tệp_vao`) và một tệp đầu ra (`-o tệp_ra`). Chương trình sẽ đọc tệp đầu vào, chuyển các ký tự thành chữ hoa và ghi ra tệp đầu ra.

```
while [ "$1" ]
do
  if [ "$1" = "-i" ]
  then
    tệp_vao=$2
    shift 2
  elif [ "$1" = "-o" ]
  then
    tệp_ra="$2"
    shift 2
  else
    echo "Chuong trinh $0 khong nhan ra tham so $1"
    echo "Cach su dung: $0 -i tệp_vao -o tệp_ra"
  fi
done
tr a-z A-Z $tệp_vao $tệp_ra
```

8.VÍ DỤ XÂY DỰNG ỨNG DỤNG BẰNG NGÔN NGỮ SCRIPT

Ngôn ngữ script là một ngôn ngữ mạnh, có thể dùng để viết một chương trình hoàn chỉnh. Để kết thúc chương này nhằm giúp bạn hiểu sâu hơn về các lệnh và cú pháp đã học, chúng ta hãy cùng nhau xây dựng một ứng dụng rất thực tế, ứng dụng quản lý đĩa CD nhạc, cũng có thể mở rộng thành chương trình quản lý đĩa CD software, album sách ... Chúng ta hãy bắt đầu.

8.1. Phần tích yêu cầu

Chúng ta dự định thiết kế và cài đặt chương trình quản lý đĩa CD. Giả sử ta có một sưu tập về rất nhiều CD nhạc. Điều trước là muốn chương trình làm, là phải tìm cách nào đó

lưu trữ được thông tin về CD như tên CD, số bài hát, tên từng bài hát (track), sau đó là chế truy tìm tên của các bài hát có trong bộ sưu tập. Để tạo thành một ứng dụng hoàn hảo, chương trình phải có khả năng chèn một tuyển tập CD mới, tạo mới bài hát, sửa đổi cập nhật tên bài hát, xóa các bài hát cũ, liệt kê danh sách bài hát có trong bộ sưu tập.

8.2. Thiết kế ứng dụng

Với 3 yêu cầu - cặp nhật, tìm kiếm và hiển thị bài hát - ta nên xây dựng một trình đơn với thực đơn (memu), tạo sự lựa chọn theo ý thích. Ta sẽ lưu dữ liệu ở dạng văn bản (text file). Giả sử tệp dữ liệu không lớn đến hàng triệu bài hát, chúng ta có thể xây dựng và dùng tệp văn bản làm cơ sở dữ liệu. Lưu dữ liệu trong tệp văn bản sẽ dễ đọc và dễ chỉnh sửa bằng các chương trình soạn thảo thông thường.

> *Trong thế giới UNIX và Linux, hầu như các tệp chứa thông tin về dữ liệu đều được lưu Ở dạng văn bản. Chính vì vậy shell cũng như các ngôn ngữ khác có trên Linux (như Perl, Python, awk ...) cung cấp khả năng xử lý chuỗi rất mạnh.*

Hãy xem xét 3 cách lưu dữ liệu sau:

1. Lưu tất cả thông tin trong 1 tệp văn bản, trong đó dùng một dòng để lưu tên CD và **n** dòng cố định để lưu tên bài hát.
2. Thông tin về bài hát và tên CD được lưu chung một dòng.
3. Thông tin về tên CD lưu trong một tập tin và quan hệ với thông tin về tên bài hát lưu trong tệp tin khác.

Ta chọn cách lưu trữ thứ 3 vì nó tuân theo mô hình quan hệ của cơ sở dữ liệu.
Một tệp tin lưu tên phân loại của từng đĩa CD, một tập tin lưu tên các bài hát tương ứng với đĩa CD đó. Số bài hát chứa trên một CD là không giới hạn. Cụ thể mối quan hệ giữa tệp tin phần loại CD và tập tin chứa tên bài hát được thể hiện như sau:

Cataloe	Trực	Type	Composer
CD123	Love Music	Romantic	Mozard
CD234	classic vioin	Classic	Batch
CD345	Pop	Jackson	Varios

catalog	TrackNo	Title
CD123	1	Some song
Cd123	2	Other song
CD345	1	Rose
CD234	1	Sonate

Bảng phân loại CD bao gồm các trường

Tên trường	**Ý nghĩa**
Catalog	Số CD (dùng làm khóa chính)
Title	Tên CD
Type	Thể loại
Composer	Tác giả

Bảng chứa tên bài hát tương ứng với CD bao gồm:

Tên trường	**Ý nghĩa**
Catalog	Số CD (khoá ngoại liên kết với bảng phân loại)
TrackNo	Số track (vị trí) của bài hát
Title	Tên bài hát

Vấn đề đặt ra là tệp dữ liệu không phải là một cơ sở dữ liệu chứa các bảng và cột, chúng ta cần quy đinh ký tự dùng làm phân cách các cột dữ liệu văn bản với nhau. Ký tự thường dùng là ký tự *tab*, khoảng trắng hay dấu phẩy. Trường hợp này ta sử dụng dấu nhảy (,) làm ký tự ngăn cách các cột văn bản với nhau (tệp tin văn bản dạng này còn được gọi là CSV - Common seperataad variable).
Trước khi bắt tay vào cài đặt chương trình, về tổng thể cần dựng những hàm sau đây:

get_return( )
get-confirm()
set-menu-choice()
insert_title( )
insert_track()
add_record_track()
add_records()
find_cd( )
update_cd( )
count_cds( )
remove_records( )
list_track( )

Dưới đây là mã nguồn được thực hiện qua các bước cài đặt chi tiết. Để dễ hiểu chúng ta tạo từng phần của mã lệnh, sau đó có thể kết hợp chúng lại như một tổng thể duy nhất.

Ví dụ 3-36 cd_apps.sh

Bước 1: Như thường lệ phần đầu của script thường là lời triệu gọi shell thực thi sh. Bạn có thể đặt một số thông tin hường dẫn và nêu bản quyền copyright theo qui định mã nguồn mở GPL.

```
#!/bin/sh .
# Ví dụ đơn giản về sữ dụng shell script xây dựng ứng dụng album CD
# Dưới đây là các điều khoản thỏa thuận về bản quyền cuả mã nguồn miễn phí.
#
# This program is free software; you can redistribute it and/or modify it
# unđer the terms of the GNU General Public License as publíhed by the
# Free Software Foundation; either version 2 of the License, or (at your option)
```

any older version.
This program (s distributed in the hopes that it will be useful, but
WITHOUT ANY WARRANTY, without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
Public Licenae for more details.
You should have received a copy of the GNU Generl Public Licene along with this program;
if not, write to the Free Software Foundtion, Inc.
675 Mass Ave, Cambridge, MA 02139, USA.
A few global variables we need. Put them here for clarity.

Bước 2: Khai báo một số biến toàn cục mà ta sẽ dùng xuyên suốt script. Chúng ta khai báo tên các tệp tin lưu trữ dữ liệu, tên tệp tin tạm, đồng thời sử dụng lệnh trap bẫy phím Ctrl-C. Khi người dùng nhấn phím này thì ta xóa tệp tạm và chấm dứt chương trình.

```
menu_choice=""
current_cd=""
title_file="title.cdb"
tracks_file="tracks.cdb"
temp_file=/tmp/cdb.$$
trap 'rm -f $temp_file' EXIT
```

Bước 3: Tất cả các hàm phaỉ khai báo trước khi sử dụng. Hãy cài đặt 2 hàm đơn giản nhất get_return() yêu cầu người dùng ấn ENTER và get_confirm() yêu cầu xác nhận “yes” hay “no”.

```
get_return() {
  echo -e "Press return \c"
  read x
  return 0
}

get_confirm() {
  echo -e "Are you sure? \c"
  while true
  do
    read x
    case "$x" in
      y | yes | Y | Yes | YES )
        return 0;;
      n | no  | N | No  | NO )
        echo
        echo "Cancelled"
        return 1;;
      *) echo "Please enter yes or no" ;;
    esac
  done
}
```

Bước 4: cài đặt menu hiển thị danh sách các tuỳ chọn. Menu xây dựng động dựa vào biến $cdcatnum. Lênh echo –e sẽ hiện thị chuỗi nhưng không xuống dòng. Nếu shell không chấp nhận echo –e, có thể thay bằng printf.

```
set_menu_choice() {
  clear
  echo "Options :-"
  echo
  echo "    a) Add new CD"
  echo "    f) Find CD"
  echo "    c) Count the CDs and tracks in the catalog"
  if [ "$cdcatnum" != "" ]; then
    echo "  l) List tracks on $cdtitle"
    echo "  r) Remove $cdtitle"
    echo "  u) Update track information for $cdtitle"
  fi
  echo "    q) Quit"
  echo
  echo -e "Please enter choice then press return \c"
  read menu_choice
  return
}
```

Bước 5: Tại đây tạo 3 hàm khá quan trọng,
insert_title(): chèn tên một CD mới vào danh mục;
insert_track(): chèn một bài hát mới;
add_record_track(): thực hiện nhập liệu và goị hàm chèn dữ liệu vào tệp dữ liệu. add_record_track()goị các hàm insert_title(),insert_track() bằng cách truyền tham số cho chúng. Trong các hàm, ta sẽ lấy về các tham số bằng các biến $*.

```
insert_title() {
  echo $* >> $title_file
  return
}

insert_track() {
  echo $* >> $tracks_file
  return
}

add_record_tracks() {
  echo "Enter track information for this CD"
  echo "When no more tracks enter q"
  cdtrack=1
  cdttitle=""
  while [ "$cdttitle" != "q" ]
  do
      echo -e "Track $cdtrack, track title? \c"
      read tmp
#      case "$tmp" in
#      "")     continue
#              ;;
#      *,*)    echo "Sorry, no commas allowed"
#              continue
```

```
#               ;;
#      esac
       cdttitle=${tmp%%,*}
       if [ "$tmp" != "$cdttitle" ]; then
         echo "Sorry, no commas allowed"
         continue
       fi
       if [ -n "$cdttitle" ]; then
         if [ "$cdttitle" != "q" ]; then
           insert_track $cdcatnum,$cdtrack,$cdttitle
         fi
       else
         cdtrack=$((cdtrack-1))
       fi
     cdtrack=$((cdtrack+1))
  done
}
```

Bước 6: add_track(0 sẽ tạo một thư mục cho CD mới.

```
# This allows adding of a new CD

add_records() {
  # Prompt for the initial information

  echo -e "Enter catalog number \c"
  read tmp
  cdcatnum=${tmp%%,*}

  echo -e "Enter title \c"
  read tmp
  cdtitle=${tmp%%,*}

  echo -e "Enter type \c"
  read tmp
  cdtype=${tmp%%,*}

  echo -e "Enter artist/composer \c"
  read tmp
  cdac=${tmp%%,*}

  # Check that they want to enter the information

  echo About to add new entry
  echo "$cdcatnum $cdtitle $cdtype $cdac"

  # If confirmed then append it to the titles file

#  get_confirm && insert_title $cdcatnum,$cdtitle,$cdtype,$cdac
  if get_confirm ; then
    insert_title $cdcatnum,$cdtitle,$cdtype,$cdac
    add_record_tracks
  else
    remove_records
  fi
```

```
  return
}
```

Bước 7: find_cd() chủ yếu sử dụng lệng grep để tìm tên một CD mới trong tệp phân loại. grep sẽ chuyển tất cả các dòng tìm thấy ra tệp tạm, dùng wc để đếm số dòng có trong tệp tạm (cũng là kết quả tìm được). wc trả về số từ, tiếp đến dùng set để láy kết quả cuả wc ở vị trí thứ nhất (số dòng) và lưu và biến listfound.

Nếu tìm thấy nhiều CD cùng một lúc, hiển thị nôị dung tìm được, không liêt kê danhsách các bài hát. Nếu chỉ có một CD, đọc nôị dung tệp tạm vào các biến toàn cục bằng read. Lưu ý IFS được đặt gí trị “,” để read đọc các trường phân cách cách nhau bằng dấu phẩy (,). Tiếp theo hiển thị nôị dung CD và toàn bộ các bài hát bằng ham list_track().

```
find_cd() {
  if [ "$1" = "n" ]; then
    asklist=n
  else
    asklist=y
  fi
  cdcatnum=""
  echo -e "Enter a string to search for in the CD titles \c"
  read searchstr
  if [ "$searchstr" = "" ]; then
    return 0
  fi

  grep "$searchstr" $title_file > $temp_file

#  set $(wc -l $temp_file)
#  linesfound=$1
  linesfound=$(wc -l $temp_file)
  case "$linesfound" in
  0)    echo "Sorry, nothing found"
        get_return
        return 0
        ;;
  1)    ;;
  2)    echo "Sorry, not unique."
        echo "Found the following"
        cat $temp_file
        get_return
        return 0
  esac

# Có thể đổi case theo cách sau:
#  if [ "$linesfound" = "0" ]; then
#    echo "Sorry, nothing found"
#    get_return && return 0
#  fi
#  if [ "$linesfound" != "1" ]; then
#    echo "Sorry, not unique."
#    echo "Found the following ..."
#    cat $temp_file
#    get_return && return 0
#  fi
```

```
#  cdcatnum=$(cut -f 1 -d , $temp_file)

  IFS=","
  read cdcatnum cdtitle cdtype cdac < $temp_file
  IFS=" "

#  if [ "$cdcatnum" = "" ]; then
  if [ -z "$cdcatnum" ]; then
    echo "Sorry, could not extract catalog field from $temp_file"
#    cat $temp_file
    get_return
    return 0
  fi

#  cdtitle=$(cut -f 2 -d , $temp_file)
#  cdtype=$(cut -f 3 -d , $temp_file)
#  cdac=$(cut -f 4 -d , $temp_file)

  echo

  echo Catalog number $cdcatnum
  echo $cdtitle
  echo $cdtype
  echo $cdac
  echo
  get_return

  if [ "$asklist" = "y" ]; then
    echo -e "View tracks for this CD? \c"
      read x
    if [ "$x" = "y" ]; then
      echo
      list_tracks
      echo
    fi
  fi
  return 1
}
```

Bước 8: update)cd() dùng để nhập lại thông tin cuả CD. Lưu ý cách tìm cuả grep: tìm xâu $cdcatnum kèm theo (,) cho chính xác. Kết quả chỗi tìm được có dạng "${cdcatnum}},". Kí hiệu ^ yêu cầu grep tìm bắt đầu từ chuỗi được chỉ định. Khoá –v dùng xác định tìm nghịc đảo, chỉ lấy các dòng không thiả mãn điều kiện. Kết quả đưa vào tệp tạm. Mục đích lad để xoá tệp CD trước đó, bằng cách đưa các dòng không xoá và một tệp khác, xoá tệp hiện hành sau đó chuyển nội dung từ tệp tạm vào đó. Chú ý ở đây dùng && để thực hiện liên tục các lệnh liền nhau (xem 3.4)

```
update_cd() {
  if [ -z "$cdcatnum" ]; then
    echo "You must select a CD first"
    find_cd n
  fi
  if [ -n "$cdcatnum" ]; then
```

```
    echo "Current tracks are :-"
    list_tracks
    echo
    echo "This will re-enter the tracks for $cdtitle"
    get_confirm && {
      grep -v "^$cdcatnum" $tracks_file > $temp_file
      mv $temp_file $tracks_file
#      cat $temp_file > $tracks_file
      echo
      add_record_tracks
    }
  fi
  return
}
```

Bước 9: count_cds() lại dung đến wc, và set để có được tổng số CD và bài hát có trong hai tệp dữ liệu.

```
count_cds() {
  num_titles=$(wc -l $title_file)
  num_tracks=$(wc -l $tracks_file)
  echo found $num_titles CDs, with a total of $num_tracks tracks
  get_return
  return
}
```

Bước 10: remove_records() loại khỏi tệp dữ liệu tên CD và các bài hát bằng grep –v. Cách làm tương tự như uapdat_cd()

```
remove_records() {
  if [ -z "$cdcatnum" ]; then
    echo You must select a CD first
    find_cd n
  fi
  if [ -n "$cdcatnum" ]; then
    echo "You are about to delete $cdtitle"
    get_confirm && {
      grep -v "^$cdcatnum" $title_file > $temp_file
      mv $temp_file $title_file
      grep -v "^$cdcatnum" $tracks_file > $temp_file
      mv $temp_file $tracks_file
      cdcatnum=""
      echo Entry removed
    }
    get_return
  fi
  return
}
```

Bước 11: list_track() hiển thị danh sách các bài hát cuả một CD.cut được dùng để lấy một trường trong tệp văn bản, dựa vào dầu phân cách (,) để định vị các trường dữ liệu. Kết quả gởi vào more đưa ra màn hình.

```
list_tracks() {
```

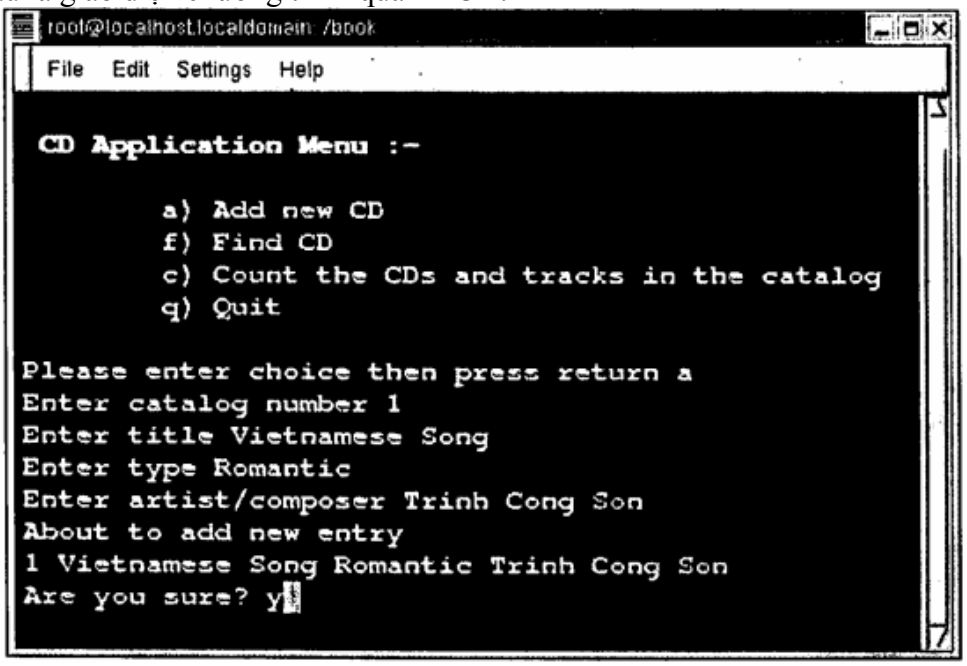
```
  if [ "$cdcatnum" = "" ]; then
    echo no CD selected yet
    return
  else
    grep "$cdcatnum" $tracks_file > $temp_file
    set $(wc -l $temp_file)
    num_tracks=$1
    if [ "$num_tracks" = "0" ]; then
      echo no tracks found for $cdtitle
    else {
      echo
      echo "$cdtitle :-"
      echo
      cut -f 2- -d , $temp_file
      echo }| more
    fi
  fi
  get_return
  return
}
```

Bước 12: Chương trình chính bắt đầu tại đây, sau khi đã cài đặt hoàn tất. Khi chọn mục thoát, script sẽ cho thông báo và kết thúc vòng while, trả về 0 cho biết script hoàn tất.

```
# Main routine starts here
#
# get the files in a known state

rm -f $temp_file

if [ ! -f $title_file ]; then
  touch $title_file
fi
if [ ! -f $tracks_file ]; then
  touch $tracks_file
fi

# Now the application proper

clear
echo
echo
echo "Mini CD manager"
sleep 3

quit=n
while [ "$quit" != "y" ];
do
  set_menu_choice
  case "$menu_choice" in
    a) add_records;;
    r) remove_records;;
    f) find_cd y;;
    u) update_cd;;
    c) count_cds;;
```

```
    l) list_tracks;;
    b)
      echo
      more $title_file
      echo
      get_return;;
    q | Q ) quit=y;;
    *) echo "Sorry, choice not recognized";;
  esac
done

#Tidy up and leave

rm -f $temp_file
echo "Finished"
exit 0
```

Hình sau là giao diện chương trình quản lí CD:



Hình 3-2 Màn hình kết xuất của chương trình script cd-app

TÀI LIỆU THAM KHẢO

Ray Swat — UNIX Applications Programming: Mastering the Shell. 1990

Lập trình Linux. Tập 1 Nguyễn Phương Lan, Hòang Đức Hải. NXB GD, 2001
Mendel Cooper Advanced Bash-Scripting Guide, 2.8 11 July 2004

Và một số tài liệu khác.