シェルを使おう
- 導入からプログラミングまで -

lilo-bk
友國哲男
(tomokuni@netfort.gr.jp)

1.1 「シェル」の説明

コマンドインタプリタ(ユーザが端末から入力した文字列を解釈し、
その指示に従って動作するプログラム)のこと。

その歴史は UNIX のそれと同じと言っても過言ではない。

1.2 なぜ「殻 (Shell)」と呼ばれるか

オペレーティングシステム(決して「基本システム」とは言わない(笑))
の核 (Kernel) を貝殻のように包んでいるという喩えに由来する。

Kernel はプロセス、メモリ、ファイル等の管理を行う部分で、何か
仕事をする(させる)ときにはこれを使わないといけないが、直接
働きかけることは難しいので、それとユーザとの間を対話的に
取り持つものが「シェル」という訳である。

対話的 (interactive) なインターフェースには

- CUI (Character User's Interface)
- GUI (Graphical User's Interface)

という二つがあるが、シェルは(当然^^;) CUI である。
UNIX を操作する(できる) CUI は事実上シェルのみである。

1.3 シェルの種類(系統)とその変種

1.3.1 系統

大きく分けて以下の二つの系統がある。

- Bourne Shell (sh)
  最も基本的で且つ標準なもの。
  その昔はこれしかなかったので単に "Shell" と言っていたが
  (今でも場合によってはこれを指すが)、次の C Shell が
  登場してから区別するために作者の名前が付けられる
  ようになった。別名 B Shell 。

- C Shell (csh)
  BSD UNIX で Bill Joy さんによって開発されたシェル。
  C like な構文を持ち、 C 言語が扱えるなら用意に入門できる
  ようになっている。
  (今となっては、そんなことは特徴でなくなってしまっているが)
  ヒストリー機能、エイリアス機能、ジョブコントロール機能等が
  最初に実現されたシェルでもある。
  BSD 系の OS はデフォルトのログインシェルになっている。

1.3 シェルの種類(系統)とその変種

1.3.2 主な変種
Bourne Shell 系

- Bash (bash, Bourne Agian Shell)
  GNU プロダクトの一つ。 POSIX 1003.2 準拠を目指している。
  Linux の多くのディストリビューションでデフォルトの
  ログインシェルになっている。

- Ksh (ksh, Korn Shell)
  AT&T の Korn さんが作った大幅な機能追加したシェル。
  昔の System V の WS では標準だった。(知らないけど(笑))
  ちなみにライセンスが AT&T のもの(オリジナル)と PDS の
  ものと二つある。
  SysV 系商用 UNIX ではデフォルトシェルのことが多い。

- Zsh (zsh, Super Korn Shell?)
  Ksh の進化版。一部 C Shell の構文も実装しており、
  対話的、非対話的にも究極のシェル。
  ただ究極過ぎて使いこなせない(私は)。(笑)

1.3 シェルの種類(系統)とその変種

1.3.2 主な変種
C Shell 系

- Tcsh (tcsh, TC Shell)
  C Shell からライン編集機能、ファイル名の補完等を実現した
  シェルで、対話的な部分が強化されている。

2.1 役割

- (1) ユーザーインターフェースとしてのシェル
  前節で説明したとおり。

- (2) ソフトウエアツールとしてのシェル
  単純な機能を持つソフトウエア群からいくつかを組み合わせて
  目的を達成する→ソフトウエアツールの概念 (by Kernighan)

  標準入出力をその入力元/出力先にしたコマンド(「フィルタ」と
  呼ばれたりする)を用いて、この入出力先を切り替えることに
  よって組み合わせを実現する。

2.1 役割

- (3) プログラミング言語としてのシェル
  同じ手順の繰り返しや条件分岐を実現するために、
  インタプリタ型のプログラミング言語としても使える
  ようになっている。

  このプログラミング機能を用いて実現されたコマンドを
  「シェルスクリプト (Shell Script)」と呼ぶ。

  システム管理、起動スクリプト、プロトタイピング
  (主に C Shell) が主な利用する場面である。

2.2 機能

2.2.1 コマンドの実行

コマンド command を実行するときは

$ command [arg1 arg2 ...]

arg1, arg2,... はコマンドの引数(ひきすう)と呼ばれる。
引数はシェルによって空白 (white space) 文字と呼ばれる
「スペース(0x20)」「タブ(0x09)」(実際にはシェル変数 IFS)
で区切られて、コマンドに渡される。

2.2 機能

2.2.1 コマンドの実行

ex.
$ echo LILO Monthly Seminar [ret]
LILO Monthly Seminar

$ echo LILO Monthly Seminar [ret]
LILO Monthly Seminar

連続した区切り文字は一つの区切り文字の意味と同じになる。
後者はコマンドの引数がシェルで処理されてからコマンドに
渡っていることが確認できるものである。

2.2 機能

2.2.2 リダイレクト (redirect) とパイプ (pipe)

リダイレクト (転送) とは、標準入出力とファイルを結合させる
機能をいう。

ex.
ファイル fuga の内容をコマンド hoge の標準入力に読み込む。
$ hoge < fuga [ret]

コマンド hoge の標準出力をファイル fuga に書き込む。
$ hoge > fuga [ret]

コマンド hoge の標エラー準出力をファイル fuga に書き込む。
$ hoge 2> fuga [ret]

2.2 機能

2.2.2 リダイレクト (redirect) とパイプ (pipe)

パイプとは、パイプライン (pipeline, 逐次処理) のことで
コマンドの標準出力を別のコマンドの標準入力と結合させる
機能をいう。

ex.
$ hoge | fuga [ret]
コマンド hoge の標準出力を fuga の標準入力に送る。

2.2 機能

2.2.2 リダイレクト (redirect) とパイプ (pipe)

シェルはコマンドを実行する前にリダイレクトの処理をし
(システムコール dup2(2) を使う)、起動されたコマンドからは
単に標準入出力を使っているように見える、ということ。

リダイレクトとパイプは 2.1 の (2) を実現するために
必要不可欠である。(3 章で詳しく説明する)

2.2 機能

2.2.3 標準入出力

標準入出力とは、コマンドがデフォルトで使用する入出力のことで、
1 つの入力モデルと 2 つの出力モデルがある。

- 標準入力 (stdin) : デフォルトの入力 → キーボード
- 標準出力 (stdout) : デフォルトの出力 → 端末
- 標準エラー出力 (stderr) : デフォルトのエラー出力 → 端末

出力側が 2 つある理由は、コマンドの入力を処理をした結果の
出力とコマンドからのエラーやワーニング等のメッセージのための
出力を分けておくと、それ毎にリダイレクトやパイプが使えて
便利だからである。

2.2 機能

2.2.4 ファイル名とカレントディレクトリ

基本的にファイル名はルートディレクトリ / からの位置で表現される。
ディレクトリはツリー構造になっているので一意に示される。
これを絶対パス名と呼ぶ。

ex.
ルートディレクトリの下の home ディレクトリの下の
hoge ユーザディレクトリ下の bin ディレクトリの下の
fuga というファイルは

/home/hoge/bin/fuga

で表される。

2.2 機能

2.2.4 ファイル名とカレントディレクトリ

しかしこの方法だとパスが長くなると不便である。
そこでシェルは現在そのシェルが実行されているディレクトリを
記憶し、そこを基準にしたファイル名を表す方法を用意している。
これが相対パスと呼ばれるものである。

"." カレントディレクトリ
".." 親ディレクトリ

を表す。

ex.
上の例でいうと、

/home/hoge/bin にいるなら ./fuga
/home/hoge/work にいるなら ../bin/fuga

という具合である。

2.2 機能

2.2.4 ファイル名とカレントディレクトリ

絶対パスも相対パスも何もないとき、

先頭なら　command 名 ($PATH をサーチする、後で説明)
それ以外なら カレントディレクトリのファイル

と解釈されるので、先頭以外でファイルを示すときは
単にそのファイル名だけで指し示すことができる。

2.2 機能

2.2.4 ファイル名とカレントディレクトリ

[Tips 1]
先頭が "-" で始まるファイルを作ってしまって消したい場合、
絶対パスで指定すればコマンドのオプションと解析されずに
できるが、一番簡単なのは "." や ".." を使って相対パスで
表すことである。

$ rm ./-piyo

$ rm ../hoge/-piyo

等とすればよい。

2.2 機能

2.2.5 メタキャラクタ

複数のファイルを指定するときに、いちいち全てを入力していると
大変なので、シェルはそれらをまとめて指定する記号を用意している。
これをメタキャラクタといい、次のものがある。

□* (アスタリスク)
0 個以上の任意の文字とマッチする。
ただし先頭の "." にはマッチしない。

□? (クエスチョンマーク)
任意の 1 文字とマッチする。
ただし先頭の "." にはマッチしない。

□[...] (ブラケット)
ブラケット内に列挙された文字のいずれかとマッチする。

2.2 機能

2.2.5 メタキャラクタ

ex.
$ ls *.c
→ カレントディレクトリの .c で終わる全てのファイル
(. で始まるファイルを除く) を表示する。

$ ls ???
→ カレントディレクトリの 3 文字のファイル全て
(. で始まるファイルを除く) を表示する。

$ ls hoge.[a-z]
→ カレントディレクトリの hoge. で始まり
a から z の小文字で終わるファイルを表示する。

* 以外は、置換できないとそのままの文字を表す。

2.2 機能

2.2.5 メタキャラクタ

現在は以下も使えるものもある。 (sh 以外は普通使える)

□~ (チルダ、「にょろ」という人も)
ユーザのホームディレクトリ (環境変数(次項参照) HOME の
ディレクトリ)を表す。

□[^...] (ブラケット、ハット)
ブラケット内に列挙された文字以外とマッチする。

2.2 機能

2.2.5 メタキャラクタ

ex.
$ ls ~/hoge
→ カレントディレクトリの hoge を表示する。
hoge がディレクトリならそのファイルリストを表示。

$ ls [^ab]*
→ カレントディレクトリの a,b 以外で始まる
ファイルを表示する。

2.2 機能

2.2.5 メタキャラクタ

[Tips 2(1)]
ls を使わずにカレントディレクトリのファイルを表す方法

$ echo *

たったこれだけであるが、 echo と "*" の意味が分かっていても
案外気づきにくいものである。

2.2 機能

2.2.6 シェル変数と環境変数

シェル変数と呼ばれるもの(後に説明)のうち、
文字通り「環境」を表す特殊な変数のこと。

□HOME
ユーザのホームディレクトリ (~) を表す。

□USER (BSD)/ LOGNAME (SysV)
ユーザの名前(ログイン名)を示す。

□PS1,PS2
Prompt String のこと。
コマンドラインから入力を受け付ける状態のときには PS1,
コマンド行が複数に跨ってコマンド入力状態が継続しているときに PS2,
と区別している。

2.2 機能

2.2.6 シェル変数と環境変数

□TERM
使用している端末(仮想端末も含む)を示す。

□PATH
コマンド検索パスを示す。 ":" 区切りのリストになっている。

□SHELL
ログインシェルを表す。

2.2 機能

2.2.6 シェル変数と環境変数

参照するときは変数名を $ (ダラー) に続ける。

ex.
```
 $ echo $PATH
 /usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games

 $ echo $HOME
 /home/hoge
```


2.2 機能

2.2.7 バックグラウンド、フォアグラウンド

コマンドを起動すると、通常はそのコマンドの処理に移る。
しかし実行に時間がかかったりすると、別のコマンドを同時に
実行したいことがある。
マルチタスク OS ならそれが可能で、コマンドの終了を待たずに
バックグラウンドで実行することで、その目的が達成される。
これを「バックグラウンド実行」と呼び、

```
 $ command &
```

とコマンドラインの末尾に "&" (アンパーサンド) を付けることで
実現できる。

対して、通常の実行を「フォアグラウンド実行」と呼ぶ。

2.2 機能

2.2.7 バックグラウンド、フォアグラウンド

バックグラウンドで実行しているコマンドをフォアグラウンドで
実行するには、シェルの内部コマンドである fg を使い

```
 $ fg
```

とすることでフォアグラウンドに実行が移る。

反対のことは内部コマンド bg を使い

```
 $ bg
```

とすれば実現できる。
ただし、端末を支配するようなアプリケーション(スクリーン
エディタ等)は、コマンドラインに一度戻るために ^Z (Ctrl-Z)
で中断した後でないとこれはできない。

2.2 機能

2.2.8 コマンドのグルーピング

グルーピングには次の二種類ある。

□(1) (command)

□(2) { command; }

前者はサブシェル、後者はカレントシェルで実行される。
サブシェルとは子シェルとも呼ばれるもので、親シェル
(この場合はカレントシェル)が一旦シェルを起動し、
そこで実行されるシェル、という意味である。

違いは以下の例を参照。

2.2 機能

2.2.8 コマンドのグルーピング

ex.
```
 $ pwd
 /home/hoge

 $ (cd bin; ls)
 fuga

 $ pwd
 /home/hoge

 $ { cd bin; ls; }
 fuga

$ pwd
/home/hoge/bin
```


2.2 機能

2.2.8 コマンドのグルーピング

また (2) は、中括弧 "{" の直後と "}" の直前には必ず 1 個以上の
スペースが必要で、グルーピングの最後のコマンド末尾にはコマンド
の区切りを表す ";" (セミコロン)が必須である。
これは中括弧がシェルの予約語であるための制限である。

2.2 機能

2.2.9 引用符と展開

□(1) ' (シングルクォート)
シングルクォートを見つけると次のシングルクォートまで
特殊文字 ($ で参照された環境変数や IFS にある文字)を
無視する。

ex.
ファイル名にスペースがある 'foo bar' というファイルの
中身を表示。

```
 $ cat 'foo bar'
```

2.2 機能

2.2.9 引用符と展開

□(2) " (ダブルクォート)
シングルクォートとほぼ同じだが、$ (ダラー)、‘ (バック
クォート)、\ (バックスラッシュ)を特殊文字として認識する。

$ はシェル(環境)変数の展開、\ は特殊文字の意味をエスケープ
という意味である。‘ は次を参照。

ex.
$ echo $HOME
/home/hoge

$ echo "$HOME"
/home/hoge

$ echo "\$HOME"
$HOME

2.2 機能

2.2.9 引用符と展開

□(3) ‘ (バッククォート)
バッククォートで囲まれた部分をコマンドとして実行し、結果を
文字列として引用する。
POSIX には "$(command)" という書き方も用意されているが、
これは古いシェルでは実装されていない。

ex.
$ echo Date is ‘date‘.
Date is Wed Jun 13 02:26:02 JST 2001.

$ echo Date is $(date).
Date is Wed Jun 13 02:26:02 JST 2001.

2.2 機能

2.2.9 引用符と展開

[Tips 2(2)]
2(1) に関連して、ここで

$ echo ’*’
$ echo "*"

とすると、違う結果になることに注意。

2.2 機能

2.2.10 初期化スクリプト

Bourne Shell では

~/.profile

が起動時に読み込むファイルである。

C Shell では

~/.cshrc
起動時に読み込む

~/.login
ログイン時に .cshrc の後に読み込む

となる。

2.2 機能

2.2.10 初期化スクリプト

ちなみに bash では

~/.bash_profile
ログインシェルのときに最初に読み込む

~/.bashrc
ログインシェル「以外の」ときに読み込む

である。

3.1 シェルスクリプトと実行

コマンドや制御文で作業の手順を記述したファイルを
シェルスクリプト (Shell Script) と呼ぶ。

hoge というシェルスクリプトを実行するには

□(1) シェルに hoge の内容をリダイレクトして読み込ませる。

$ sh < hoge

□(2) シェルは引数を入力ファイルとして扱うので、単に

$ sh hoge

でもよい。

3.1 シェルスクリプトと実行

□(3) また、ファイルに実行属性をつけて直接起動することもできる。

$ chmod +x hoge

$ ./hoge

□(4) 上記はサブシェルで実行していたが、"." コマンドで
カレントシェル上で実行もできる。
これは環境変数の再設定など、現在の端末のシェルで
実行しないといけないもののためにある。

$. hoge

3.1 シェルスクリプトと実行

(3) 場合、ファイルの先頭行に #! につづけて、実行したい
コマンド(この場合は sh)の絶対パスを書いておく。
(ただし sh の場合はこの行を書かなくても良い)

ex.
Bourne Shell で実行
#!/bin/sh

C Shell で実行
#!/bin/csh

Perl で実行
#!/usr/bin/perl

3.2 シェル変数と export

3.2.1 代入

シェル変数は先頭に英文字およびアンダースコアで始まり、以降
0 文字以上の英数字およびアンダースコアのならびで表される。
(正規表現を使って表すと [A-Za-z_][0-9A-Za-z_]*)

シェル変数に値を格納したいとき、 "=" を使って代入を行う。

シェル変数=値

ただし、 "=" の両側にスペース文字を入れてはいけない。
スペースがあると、シェルはコマンドの実行と解釈してしまう
からである。

3.2 シェル変数と export

3.2.1 代入

ex.
$ foo=bar
これはオッケー。

$ foo= bar
これはダメ。

$ foo =bar
これもダメ。

$ foo = bar
当然これもダメ。

3.2 シェル変数と export

3.2.2 特徴

C 等のプログラミング言語と違いは

□シェル変数にはデータ型という概念はなく、全て文字列型として扱われる
□新しい変数を見つけると、それを登録しヌル値 (\0 に相当) で初期化する

等である。

Perl 等のスクリプト言語ははシェルスクリプトを
(かなり)意識しているため、その性質は似ている。

3.2 シェル変数と export

3.2.3 参照

参照する場合、変数名の先頭に "$" をつけると、シェルが
変数を値に置き換える。
何も格納されていない、または初めて出てきた変数は、初期値の
ヌルで置き換えられる。

ただしシェルは "$" を付けてシェル変数を表す場合、それに続く
英数字の最も長くなるごくを変数名として認識することに注意。
(文を IFS (次項参照)で区切って走査するので)

$ foo=shell
$ echo $fooscript (何も出力されない)

この場合は "{ }" で囲むことでシェル変数部分を明示できるので、
これを使えばよい。(実はこちらの方が正式)

$ echo ${foo}script
shellscript

3.2 シェル変数と export

3.2.3 参照

また以下のような特殊な参照方法もある。

□${val?message}
val が未定義なら message を出力して終了。
□${val-string}
val が未定義なら val に string を代入。
□${val=string}
${val-string} と同じだが、位置パラメータには適用不可。

3.2 シェル変数と export

3.2.3 参照

[Tips 3]
実はシェル変数には、 ' (シングルクォート)を用いて
メタキャラクタをも格納できる。

$ foo='*'

$ echo $foo
<ディレクトリにあるファイルのリスト>

ただし、 2(2) でやったように

$ echo '$foo'
$ echo "$foo"

とすれば、当然違う結果になる。

3.2 シェル変数と export

3.2.4 export

シェル変数は宣言しただけでは、そのシェル(カレントシェル)でのみ
参照が行われるのみである。
コマンドの起動やシェルスクリプトの呼び出しにおいても参照を
させたい場合、 export (輸出)をして外から参照できるように
してやる必要がある。

$ export val1 val2 val3 ...

サブシェルが起動したとき、これらの値がコピーされてそのサブシェル
で参照できるようになる。

コピーされるので、 export された変数が起動されたサブシェル側で
変更されたとしても、そこで export された場合にそのサブシェル
以降でのみ影響はあるが、親シェルの内容は書き換えられない。

3.3 特殊なシェル変数

特殊なシェル変数として、先に紹介した環境変数もあるが、
それ以外に特にシェルスクリプトにおいて重要なシェル変数も
沢山あるので、それを紹介する。

3.3 特殊なシェル変数

3.3.1 IFS

Internal Field Separator の略。シェルがコマンドラインを
走査するときの区切り文字のリストが格納されている。
デフォルトではホワイトスペース文字と呼ばれるスペース、
タブ、改行文字が含まれている。

3.3 特殊なシェル変数

3.3.1 IFS

[Tips 4]
普通に

```
$ echo $IFS
```

では何も見えない。

```
$ echo -n "$IFS" | od -b
0000000 040 011 012
0000003
```

として文字コード等に置き換えて見ないと分からない。
(echo の -n オプションは改行するのを抑制している。)

3.3 特殊なシェル変数

3.3.2 位置パラメータ

シェルスクリプトも他のコマンド同様に引数を受け取ることができる。
これらを参照するために、シェルに用意された特殊な変数を使う。
これを位置パラメータといい、 "$" につづく一文字の数字で表現する。
$1, ..., $9 はそれぞれ一番目、...、九番目を表す。
十番目以降の参照には、内部コマンド shift を用いれば

$1 に $2 の内容をコピー
$2 に $3 の内容をコピー
$8 に $9 の内容をコピー
$9 に十番目の内容をコピー

となり、参照できるようになる。

3.3 特殊なシェル変数

3.3.2 位置パラメータ

ただし、一度 shift してしまうと、古い内容は失われてしまうので、
後で必要な場合は shift をする前に別のシェル変数にコピーして
退避しておかないといけない。

ちなみに $10 は ${1}0 と解釈されてしまう。

3.3 特殊なシェル変数

3.3.2 位置パラメータ

$0 は特殊で、そのシェルスクリプトのファイル名が格納されている。
これを用いると、そのプログラムの呼び出された名前によって動作が
異なるようなスクリプトも記述できる。

位置パラメータの格納は IFS で区切られて格納されることに注意。
IFS を含む文字列を一つの引数(一つの位置パラメータ変数に格納させる)
と判断させるためには ' や " でクォーテーションしないといけない。

3.3 特殊なシェル変数

3.3.3 引数関連 (#, *, @)

□$#
引数の個数で shift されれば、 shift された分だけ減る。

□$*
$0 以外の全ての引数を示す。 $1 $2 $3 ... と展開される。

□$@
$* と同じだが、 "$1" "$2" "$3" ... とそれぞれを
" (ダブルクォート)囲みで格納ている。

$* と $@ の違いは、 "$@" とすると ' や " を解釈して展開する
前の形で格納されること。

3.3 特殊なシェル変数

3.3.3 引数関連 (#, *, @)

ex.

```
$ cat test1.sh
echo \# of Arg is $#.
for i in $*
do
  echo $1
  shift
done

$ cat test2.sh
echo \# of Arg is $#.
for i in "$@"
do
  echo $1
  shift
done
```


3.3 特殊なシェル変数

3.3.3 引数関連 (#, *, @)

これを以下の引数で比べるとよい。

```
$ sh test1.sh 1 2 3 '4 5'
# of Arg is 4
1
2
3
4 5

$ sh test2.sh 1 2 3 '4 5'
1
2
3
4
5
```

3.3 特殊なシェル変数

3.3.3 引数関連 (#, *, @)

ちなみに test2.sh は

```
$ cat test3.sh
for i
do
  echo $1
  shift
done
```

と同じ意味である。
このことから普通 "$@" が用いられるのが好ましいことが多い。

3.3 特殊なシェル変数

3.3.4 ステータス、プロセス、その他 (?, $, -, !)

□$?
シェルが最後に実行したコマンドの終了ステータスを
保持している変数。

□$$
現在のプロセス番号を保持している変数。
一時作業ファイルを作る場合、この変数を使ったファイル名に
することによって、プロセスは重複しない番号で管理されている
ことから一意性が確保されるので有効な手段となる。

□$-
シェルにセットされているオプションを保持している変数。

□$!
バックグラウンドで実行された直前のプロセスのプロセス番号
を保持している変数。

3.4 主な内部コマンド

3.4.1 set

□(1) シェル変数を表示
□(2) シェルのオプションの設定、解除
□(3) 一つのレコードから IFS で区切られたフィールドを取り出して
位置パラメータに代入することの三つの機能がある。

3.4 主な内部コマンド

3.4.1 set

ex.

```
$ date
Sat Jun 16 23:41:59 JST 2001

$ set ‘date‘
$ echo $# $1 $2 $3
6 Sat Jun 16

$ shift
$ echo $1 $2 $3
5 Jun 16 23:42:34
```


3.4 主な内部コマンド

3.4.2 shift

位置パラメータの内容をシフトする。

```
$ shift n
```

で n 個左にずらすことができる。

3.4 主な内部コマンド

3.4.3 echo

そのまま与えられた文字列を返す。
特殊変数やメタキャラクタがあればシェルがそれを解釈した
ものが echo に渡される。

デフォルトで改行を付けるが、 -n オプションで抑制できる。

外部コマンド (/bin/echo) もあるが、単に echo だけだと
内部コマンドの echo が呼ばれる。

3.4 主な内部コマンド

3.4.3 echo

[Tips 5]
改行を付けることで仮想ファイルのように見えるので、
パイプやリダイレクトを使うといろいろできる。

```
$ echo $hoge | grep string > /dev/null
```

これで変数 hoge に string が入っているかどうかが
$? で判断できる。 (0 ならば入っている)

3.4 主な内部コマンド

3.4.4 exit

シェルは EOF (End Of File, コマンドライン上では ^D) に
達すると終了するが、 exit は任意の位置で終了させる。

```
$ exit n
```

とすることで終了ステータス n を返して終了する。
n を指定しないと exit の直前に実行されたコマンドの
終了ステータスが返される。

ログインシェルで exit とするとログアウトと同じ結果になる。

3.4 主な内部コマンド

3.4.5 exec

シェルに変わって引数で指定されたコマンドを新しいプロセスを作らずに実行する。

exec の実行にあたり、現在のファイルをクローズし、新しいファイルをオープンするので、それ以降の入出力を引数で指定したものに切り替えることもできる。

3.4 主な内部コマンド

3.4.5 exec

ex.
標準入力を hoge に切り替える。
```
$ exec < hoge
```

標準出力を fuga に切り替える。
```
$ exec > fuga
```

標準エラー出力を file に切り替える。
```
$ exec 2> piyo
```


3.4 主な内部コマンド

3.4.6 eval

引数をシェルの入力として解析してから実行する。
eval を実行する前に引数がシェルによって走査されるのでコマンドが実行されると結果としてコマンドラインを 2 回評価することになる。

これを利用すれば、連想配列もどきなことも可能である。

ex.
```
$ eval os_$hostname="$os"
```


3.4 主な内部コマンド

3.4.7 wait

現在実行している子プロセスの終了を待ち、終了状態を保存する。

```
$ wait n
```

n は待つ子プロセスのプロセス番号を指定する。省略すると全ての子プロセスの終了を待つ。

3.4 主な内部コマンド

3.4.8 trap

シェルスクリプトを実行中になんらかの原因で停止することが考えられるが、そのまま終了させると困る場合がある。
そこでこの trap を使えば、後始末をしてから終了させることができる。

```
$ trap command signal ...
```

command を省略すると、デフォルトの動作に戻すことを示す。

3.4 主な内部コマンド

3.4.8 trap

主なシグナルは次のようなものである。

- 0
  正常終了
- 1 HUP
  hangup (デーモン等の再設定によく使われる)
- 2 INT
  interrupt (端末割り込み)
- 3 QUIT
  quit (コアダンプ)
- 9 KILL
  kill (trap できない)
- 15 TERM
  terminate (kill コマンドのデフォルト)

詳しくは signal(7) を参照のこと。

3.4 主な内部コマンド

3.4.8 trap

ex.
シグナル 1, 15 が来たときにテンポラリファイルを消して終了。
```
$ trap 'rm -f $tempfile; exit 1' 1 15
```

シグナル 2 を無視。 ("" は何もしない意味)
```
$ trap '' 2
```

シグナル 3 をデフォルトの動作 (signal(7) の動作) に戻す。
```
$ trap 3
```


3.4 主な内部コマンド

3.4.9 read

標準入力から一行読み込み引数に指定された変数にその行をIFS で区切って代入する。

```
$ read val1 val2 val3 ...
```

変数の数が IFS 区切りの語句数より少ないときは、一番最後の変数に残り全ての部分が入る。

read 文の終了ステータスは EOF を検出しない限り 0 (正常) なので、これをループ制御文(次節参照)と組み合わせて使うことは非常に有用である。

3.4 主な内部コマンド

3.4.10 :

何もしないコマンドだが、引数の評価をする。

ex.
```
$ : ${hoge?hoge is not set.}
```

こうすると、シェル変数 hoge が未定義のときに

```
hoge is not set.
```

と出力して終了する。

3.5 制御文

3.5.1 test コマンド ("[" コマンド)

条件を評価し、結果が真ならば 0、そうでなければ 0 以外の値を返す、重要な外部コマンドである。
(これが条件分岐によく使われるからである。)

"[" で実行された場合、最後に "]" が必要である。

条件評価には次のものがある。

3.5 制御文

3.5.1 test コマンド ("[" コマンド)

(1) 文字列評価

- str1 = str2
  str1 と str2 が一致なら真
- str1 != str2
  str1 と str2 が一致しないなら真
- -n str1
  str1 がヌルでなければ真
- -z str1
  str1 がヌルであれば真

3.5 制御文

3.5.1 test コマンド ("[" コマンド)

(2) 数値評価

- num1 -eq num2
  num1 と num2 が等しいなら真 (=)
- num1 -ne num2
  num1 と num2 が等しくないなら真 (!=)
- num1 -ge num2
  num1 が num2 以上なら真 (<=)
- num1 -gt num2
  num1 が num2 より大きいなら真 (<)
- num1 -le num2
  num1 が num2 以下なら真 (>=)
- num1 -lt num2
  num1 が num2 より小さいなら真 (>)

3.5 制御文

3.5.1 test コマンド ("[" コマンド)

(3) ファイル評価

- -e file
  file が存在するなら真 (exist)
- -f file
  file が通常ファイルなら真 (file)
- -d file
  file がディレクトリなら真 (directory)
- -r file
  file が読み出し可能なら真 (readable)
- -w file
  file が書き込み可能なら真 (writeable)
- -x file
  file が実行可能なら真 (executable)
- -s file
  file のサイズが 0 でないなら真 (size)

3.5 制御文

3.5.1 test コマンド ("[" コマンド)

(4) 論理演算子

- ! cond
  条件 cond が偽のとき真 (not)
- cond1 -a cond2
  cond1 と cond2 が真のとき真 (and)
- cond1 -o cond2
  cond1 か cond2 が真のとき真 (or)

論理演算子を使った場合、評価の短絡は行わない。
つまり最後まで評価する。

3.5 制御文

3.5.2 条件分岐 (if)

if 文は引数のコマンドの終了ステータスをみて
- 正常なら then 以下を
- そうでなければ次の elif または else 以下を

実行する。
書式は以下のとおり。

```
if command1
then
  (処理 1)
elif command2
  (処理 2)
else
  (処理 n)
fi
```

elif は省略可能。

3.5 制御文

3.5.3 短絡実行 (&&, ||)

if を使わないでももっと簡単にする方法がある。
それは && や || を使った短絡実行をすることである。

- $ command1 && command2
  command1 が正常終了した(終了ステータスが真の)とき、command2 を実行しその終了ステータスを返す

- $ command1 || comannd2
  command1 が正常終了しなかった(終了ステータスが偽の)とき、 command2 を実行しその終了ステータスを返す。

3.5 制御文

3.5.3 短絡実行 (&&, ||)

ex.
```
$ if [ -d hoge ] ; then echo hoge is directory ; fi

$ [ -d hoge ] && echo hoge is directory
```

は同じ動作をする。

3.5 制御文

3.5.4 条件分岐 (case)

case 文はシェル変数を評価し、同じパターンが見つかったときに、１つあるいはそれ以上の処理を実行する。
書式は以下のとおり。

```
case $val in
  pattern1)
    (処理 1)
    ;;
  pattern2)
    (処理 2)
    ;;
  patternn)
    (処理 n)
    ;;
esac
```

パターンには * (任意の文字)や | (論理和)も使うことができる。

3.5 制御文

3.5.5 ループ制御 (for)

一組のコマンドを指定された回数だけ実行する。
書式は以下のとおり。

```
for val in arg1 arg2 ... argn
do
  (処理)
done
```

val はループが進む毎に arg1, arg2, ... が次々に「代入」される
シェル変数なので、 3.2.1 のように $ を付けない。

in につづく引数に、 *, ?, [] 等のメタキャラクタも使用できる。
(当然シェルによって展開される)

3.5 制御文

3.5.6 ループ制御 (while)

ある条件が満たされている間だけループを実行する。
書式は以下のとおり。

```
while condition
do
  (処理)
done
```

condition がコマンドの場合は終了ステータスが用いられる。

3.5 制御文

3.5.7 ループ制御 (until)

ある条件が満たさるまでループを実行する。
while と同様、書式は以下のとおり。

```
until condition
do
  (処理)
done
```


3.5 制御文

3.5.8 ループ制御 (break, continue)

break コマンドはループを直ちに脱出する。
continue コマンドはそれ以降の処理をスキップしてループを再度評価から行う。

ともに引数 n を取り、内側から数えたループの数を示している。デフォルトでは 1 になっている。

Acknowledgement

参考にしたもの

□Bourne Shell 自習テキスト (http://www.tsden.org/takamiti/shText/)
□プロフェッショナルシェルプログラミング (アスキー, ISBN4-7561-1632-9)

最後に

ご静聴、ありがとうございました。 m(_o_)m

おしまい。