シェルを使おう
- Bourne Shell と C Shell -

lilo-bk
友國哲男
(tomokuni@netfort.gr.jp)

歴史的背景から (2)

C Shell

- Bourne Shell でなかった機能の補完と使いやすさ/理解しやすさを考慮して設計
- 文字通り C 言語に近い文法を持っている
- コマンドライン補完機能、ジョブコントロールを初めて実装した

どちらかというと対話的なことを考えてされている。
また C 言語の文法に近いことから、 C のプロトタイプ(実験)
としてのスクリプト言語としても用いられる。

以下では Bourne Shell のプロンプトを '$' で、 C Shell のプロンプトを
'%' で、それぞれ表記する。

1.2 変数 (1)

C Shell の変数は 0 個以上の句のリストとして扱われている

- Bourne Shell になくて C Shell にあるもの
  - 1) 配列 $var[num]
    - num によって選択された var の値(リストの num 番目)
  - 2) $#var(${#var})
    - var の句の数
  - 3) $?var(${?var})
    - var の set/unset 情報 (boolean 値)
  - 4) $<
    - 一行読み込んでその値を返したもの

1.2 変数 (3)

- Bourne Shell による実現方法(配列の扱いについては次節で紹介)
  - 1) target=‘set -- $var; shift <num-1>; echo $1‘
  - 2) var_num=‘set -- $var; echo $#‘
  - 3) var_def=‘var_def=${var+1}; echo ${var_def:-0}‘
  - 4) read line
  - 5) cut, sed, tr 等と ‘...‘ を用いて実現する。

- Bourne Shell にあって C Shell にないもの
  - 環境変数 IFS
    - 従って C Shell では IFS の操作はできない
  - 変数参照(置換)
    - ${var-word}, ${var:-word}
    - ${var=word}, ${var:=word}
    - ${var+word}, ${var:+word}
    - ${var?word}, ${var:?word}

歴史的背景から (1)

Bourne Shell

- 最初の Shell
- おそらくどの UNIX システムにも搭載されている

オリジナルの Bourne Shell は)対話的には用いられることは
少ないが、移植性や互換性があるため主にスクリプトで用いられる。

1.1 メタキャラクタ

- Bourne Shell になくて C Shell にあるもの
  - ~ (チルダ) -> ホームディレクトリ
  - [^...] -> ... に列挙されている文字以外
  - {a,b,...} -> a,b,... という文字列のリスト

- Bourne Shell と C Shell で同じもの
  - *, ?, [...]

1.2 変数 (2)

  - 5) 修飾子
    - $var:h -> パス名の最後の部分を除いたパス名
    - $var:t -> パス名の最後の部分だけ
    - $var:r -> 拡張子を除いた部分
    - $var:e -> 拡張子だけ
    - $var:s/x/y/ -> x を y に置換する
    - $var:q -> 文字列をクォートする
    - $var:x -> 文字列を空白やタブで区切り、クォートする

1.2 変数 (4)

- 置き換えが必要なもの

Bourne Shell		C Shell
$@	<->	$argv
$1, $2, $3, ...	<->	$argv[1], $argv[2], $argv[3]
$?	<->	$status

- Bourne Shell と C Shell で同じもの
  - $0
  - $$
  - $!
  - $*

1.3 配列

C Shell ではもともと変数がリストとして扱われており
最初から配列アクセスが可能である。
Bourne Shell では配列機能がないが、 eval を使用する
ことで配列(連想配列)のようなものを便宜的に作ることは
可能である。

```
# 定義
hoge_1=hoge
hoge_2=hogehoge
hoge_3=hogehogehoge
...
# アクセス
num=1
while [ $num -lt $max ]
do
  eval echo '$'hoge_$num
done
```


2.1 変数代入

Bourne Shell での代入は

- $ var=hoge

であるが、 C Shell では

- % set var=hoge

 or

- % set var = (hoge fuga)

となる。

Bourne Shell では '=' の前後は空白を入れられなかったが、
C Shell では入れてもよい。その場合は両側にいれないと
いけない。片側だけではダメである。

2.2 変数のスコープ (2)

Bourne Shell の変数には C Shell の意味での
種類はないが、最初から用意されている環境変数と
ユーザーが定義する Shell 変数がある。
環境変数は子シェルにも引き継がれるが、 Shell 変数は
export コマンドで外から見えるようにしてやらないと
子シェルからは参照できない。

```
(親)$ VAR=HOGE
(親)$ var=hoge
(親)$ echo $VAR
HOGE
(親)$ echo $var
hoge
(親)$ export VAR
(親)$ sh
(子)$ echo $VAR
HOGE
(子)$ echo $var
```


2.3 制御文 (2)

ここで注意したいのは、 Bourne Shell では if の
判断にはコマンドの終了ステータスを使用しているが、
C Shell では式の評価をしているところが決定的に
違う部分である。

1.4 演算

C Shell には演算も機能としてある。

- % @ i++ # 変数 i をインクリメント

演算子としては C で使えるものは大抵使える。
(優先順位は、、、 C とほとんど一緒のようだがきちんとは調べていません。)

Bourne Shell では配列と同じく演算機能もないが、
外部コマンドである expr を用いれば実現可能である。

- $ i=‘expr $i + 1‘

2.2 変数のスコープ (1)

C Shell の変数には以下の 2 種類がある。

- 環境変数
  - 子シェルにも引き継がれる変数。 setenv で定義する。
- Shell 変数
  - カレントシェルのみで使用できる変数。 set で定義する。

```
(親)% setenv VAR HOGE
(親)% set var = hoge
(親)% echo $VAR
HOGE
(親)% echo $var
hoge
(親)% csh
(子)% echo $VAR
HOGE
(子)% echo $var
```


2.3 制御文 (1)

if 文

Bourne Shell では

```
if command1
then
  ...
elif command2
  ...
else
  ...
fi
```

となるが、 C Shell では

```
if (exp1) then
  ...
else if (exp2) then
  ...
```


2.3 制御文 (3)

while 文

if 文と同じように Bourne Shell では

```
while command
do
  ...
done
```

となり、 C Shell では

```
while (exp)
  ...
end
```

となる。

Bourne Shell では更に until 文も用意されている。

2.3 制御文 (4)

foreach 文

Bourne Shell では

```
for name in list
do
  ...
done
```

となり、 C Shell では

```
foreach name (list)
  ...
end
```

となる。

特に機能的な違いはない。

2.3 制御文 (5)

文字列分岐

Bourne Shell では case 文を用いて

```
case $var in
  pat1)
    ...
    ;;
  pat2)
    ...
    ;;

    ...
  patn)
    ...
    ;;
esac
```

となる。

2.3 制御文 (6)

C Shell では C 言語と同じく switch-case 文を用いて

```
switch ($var)
case pat1:
   ...
   breaksw
case pat2:
   ...
   breaksw

   ...
case patn:
   ...
   breaksw
endsw
```

と書く。

共に pat にはメタキャラクタが使用できる。

2.3 制御文 (7)

Bourne Shell の case 文は pat 一つにつき ';;' が
必要だが、 C Shell の switch-case 文には breaksw
でターミネートしないことも可能である(breaksw が
あるか endsw まで文を実行する)。

2.3 制御文 (8)

その他の文

- eval, exit 文は双方とも同じである。
- exec 文も同じであるが、 Bourne Shell では更にファイルディスクリプタの切り替えも行える。 (2.6 節参照)
- Bourne Shell での . コマンドは source コマンドが対応する。
- Bourne Shell での trap 文は C Shell での onintr 文が対応するが文法が異なる。また onintr は任意のシグナルが扱えない(SIGINT のみ)。
- shift 文は双方に存在するが、 Bourne Shell での shift 文の引数は数字を取り "$@" に対して shift する数を表すが、 C Shell での引数はシェル変数を取りその名前のシェル変数のリストを 1 つだけシフトすることを表す。
- wait 文は Bourne Shell では引数(子プロセスのプロセス ID)を取ることが出来るが、 C Shell ではできない。
- 繰り返しを表す repeat 文やラベルにジャンプする goto 文は C Shell にしか存在しない。
- 逆に read 文、':' コマンド、 getopts コマンドは Bourne Shell にしか存在しな

2.4 その他の文法 (1)

alias, function

alias 文は基本的に同じ意味だが、 alias 定義をする際の

- sh$ alias ls='ls -F' # '=' がない
- csh% alias ls 'ls -F' # '=' がある

という書き方の相違と、 C Shell では alias に引数が取れる
(Bourne Shell では取れない)ことが異なる。

- % alias search 'find . -name \!* -print'

(カレントディレクトリ以下で引数で指定された名前をもつファイルの
一覧が表示される。)

2.4 その他の文法 (2)

Bourne Shell では代わりに function が用意されており、
上の例であれば

- $ search () { find . -name $* -print; }

と書ける。

2.4 その他の文法 (3)

C Shell と Bourne Shell で同じもの

- クォート '', "", ``
- 短絡演算子 &&, ||
- サブシェル実行 ()
- バックグラウンド実行 &

2.5 リダイレクト/パイプ

C Shell では以下のリダイレクト/パイプが使用可能である。
- cmd > file
  - cmd の標準出力を file に書き込む
- cmd >& file
  - cmd の標準出力/標準エラー出力を file に書き込む
- cmd >> file
  - cmd の標準出力を file に追加する
- cmd >>& file
  - cmd の標準出力/標準エラー出力を file に追加する
- cmd < file
  - file の内容を cmd の標準入力へ読み込ませる
- cmd < word
  - ヒアドキュメント
- cmd1 | cmd2
  - cmd1 の標準出力を cmd2 の標準入力に繋ぐ
- cmd1 |& cmd2
  - cmd1 の標準出力/標準エラー出力を cmd2 の標準入力に繋ぐ

しかし、例えば

2.6 ファイルディスクリプタ

前節で紹介した C Shell の制限は、実は任意のファイルディスクリプタ制御ができないことに起因する。

一方 Bourne Shell では、任意のファイルディスクリプタをユーザーが exec を使用することで処理でき、その結果複雑な処理も可能になる。

```
exec 3<&0 0<infile
exec 4>&1 1>outfile
while read line
do
  case "$line" in
    [!#]*'='*)
      eval $line
      ;;
  esac
done
exec 0<&3 3<&-
exec 1>&4 4>&-
```


3.1 シェルスクリプト (1)

シェルのオプション

- Bourne Shell
  - sh コマンドに直接オプションを与えて制御できる。
  - また set コマンドのオプションでも制御できる。(つまりインタラクティブシェルのときでも途中で状態を変更できる)
- C Shell
  - csh コマンドに直接オプションを与える。

3.1 シェルスクリプト (2)

2) シェルスクリプト起動

- Bourne Shell
  - 起動時に何も初期設定ファイルを読み込まない。
- C Shell
  - 起動時に ~/.cshrc があればそれを読み込む。

.cshrc にいろいろ設定を書いていると、 C Shell スクリプトを実行する度に ~/.cshrc を読み込むので、実行速度が遅くなる。
そこで C Shell スクリプトには、最初に

- #!/bin/csh -f

と -f オプション(初期設定ファイルを読み込まない)を追加して書いていることが多い。

3.2 変数

変数の処理に関しても実装面での相違がある。(1.2 節を参照)

予約されたシェル変数
- Bourne Shell
  - $*, $@, $0, $1, ..., $9, $#, $$ 等特殊文字と数字だけである。
- C Shell
  - $argv, $status 等一般文字の組み合わせたものが予約されている。

変数の処理
- Bourne Shell
  - 置換や代入する機能は存在する
  - が、直接的に文字列操作はできない
- C Shell
  - 限定的ではあるが文字列操作ができる
  - が、汎用的な意味での置換処理をする機能はない
  - (1.2 節参照)

3.3 処理の例外 (1)

C Shell には重大な例外の処理がある。
それは '!' (エクスクラメーションマーク)である。

'!' はヒストリを参照する際に使用されるが、その
プライオリティが高く設定されている(と思う)ため、
いつでもシェルで展開される。

```
% ls
foo  bar  baz
% !!
foo  bar  baz
% '!!'
foo  bar  baz
% "!!"
foo  bar  baz
```


3.3 処理の例外 (2)

このように ' (シングルクォート)や " (ダブルクォート)で囲んだ
場合でもヒストリに展開されてしまう。
ただしこの場合でも ' や " で囲んだ中のヒストリは alias を
サーチしないので、違う結果になることに注意。

```
% alias ll 'ls -alF'
% ll
foo
bar
baz
% 'll'
'll'
ll: Command not found.
```


3.3 処理の例外 (3)

この結果として、例えば引数を取るような alias を定義する場合に
注意が必要になる。

```
% alias search 'find . -name !* -print'
alias search 'find . -name  -print'
% search hoge
find: paths must precede expression
Usage: find [path...] [expression]
% alias search 'find . -name \!* -print'
% search hoge
./tmp/hoge
```

3.4 構文解釈 (1)

Bourne Shell では改行または ';' (セミコロン)で文の解釈をし
前のものから一文ずつ実行する。

C Shell でも区切りは改行または ';' と同じであるが、
構文解析が改行が基本にあるため、常に行全体を解析して
評価する。

そのため途中で ';' があったり、また前の状態に依存した
文を記述すると意図していない動作をすることがある。

3.4 構文解釈 (2)

```
$ alias cmd='if [ $? = 0 ]; then echo Success.; else echo Fail.; fi'
$ cmd
Success.

% alias cmd 'if ($status == 0) then echo Success.; else echo Fail.; endif'
% cmd
if: Improper then.


$ if [ x"$foo" = x ]; then bar=$foo; fi
$ echo $bar
(何も出力されない)

% if ($?foo) setenv bar $foo
foo: Undefined variable.
```


3.4 構文解釈 (3)

C Shell には前節で述べた例外の他にも、構文解釈の仕方による例外解釈がある。
それは " クォート中で " あるいは $ 、改行を使用した場合である。

```
sh$ foo="You said, \"let's go.\""
sh$ echo $foo
You said, "let's go."

csh% set foo = "You said, \"let's go.\""
Unmatched '.
```


3.4 構文解釈 (4)

```
sh$ foo="\$bar $HOME"
sh$ echo $foo
$bar /home/tomokuni

csh% set foo = "\$bar $HOME"
bar: Undefined variable.
```


3.4 構文解釈 (5)

```
sh$ foo="line 1 \
> line 2"
sh$ echo $foo
line 1 line 2
sh$ echo "$foo"
line 1 line 2

csh% set foo = "line 1 \
line 2"
csh% echo $foo
line 1  line 2
csh% echo "$foo"
Unmatched ".
```

Acknowledgement

参考にしたもの

□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)

スクリプト言語として見て

2.6 節や 3.3 節、 3.4 節で挙げたような相違がある所以は、
ひとえに
□「 Bourne Shell の悪いユーザーインターフェースを改善しよう」
という C Shell を作り上げる際の動機(主にヒストリ機能の
充実と文法を C 言語のようにすること)に集約されている
と思われる。(ユーザーインターフェースを重視する姿勢)

互換性の面だけではなく、上記のような事情によって
Bourne Shell が所謂「シェルスクリプト」として使用される
場面で使われるのであろう。

ただし、プロトタイプとしてのスクリプトや配列を使用する
ようなスクリプトに関しては C Shell を使用するメリットは
あると思われる。
(現在では単に配列を使用するだけであれば bash や zsh のような
Bourne Shell 系の高機能シェルを使用すればいいという話もあるが。)

最後に

資料は以下にあります。
□http://www.netfort.gr.jp/~tomokuni/lms/shell/

ご静聴、ありがとうございました。 m(_o_)m

また次回(まだまだやるつもり^^;;)もご期待ください。

おしま