2008 年 01 月 13 日

csh スクリプトに関する基礎知識

新潟工科大学 情報電子工学科 竹野茂治

1 csh スクリプトとは

Unix では、コマンドは基本的にターミナルソフト上でキーボード入力する形で実行します (MS–Windows のコマンドプロンプトでの作業のような使い方です) が、そのコマンド入力を解釈してプログラムの実行を OS に命令するソフト、つまり OS とユーザのインターフェースを取りもつソフトを**シェル** と呼びます。

シェルには、直接キーボードから命令を与えることができるだけでなく、ファイルに命令を書いておいてそれを順番に実行させる (**バッチ処理** と言います) こともできるのですが、そこには単にコマンドを書き並べるだけでなく、条件分岐やループ、変数などを使うこともできて、簡単なプログラミングができるようになっています。それを **シェルスクリプト** と呼びます。

Unix でよく使われるシェルには何種類かあって、それぞれで文法が違います。よく使われるシェルには、

> sh (Bourne shell)、csh (C shell)、ksh (Korn shell)、tcsh (TENEX C shell)、
> bash (Bourne Again shell)、zsh (Z shell)

などがあります。

Unix は、起動時の種々のサービスの実行にもシェルスクリプトが使われていますが、その多くがシステムに標準装備されている sh や ksh, bash などのスクリプトで書かれていることもあり、また csh は「スクリプトプログラムには向かない」という批判もあるため ([1], [2] 参照)、「シェルスクリプトプログラミング」というと sh や bash スクリプトの解説が多いようです。しかし、csh は BSD 系 Unix では伝統的に標準的なシェルであり、csh や tcsh を普段利用している人 (私もそう) にはむしろ csh スクリプトの方が自然で、スクリプトの知識を知っていると対話処理に役立つこともありますし、また csh スクリプトは C 言語に似た構文も持っていて C 言語になじんでいる人にはわかりやすいと思いますので、ここでは csh スクリプトを紹介します。

本稿では、その概要のみを解説します。

2 csh の実行環境

csh は通常の Unix ならデフォルトでインストールされていると思います (/bin/csh, /usr/bin/csh 等)。なお、最近の FreeBSD のように、csh という名前でも実体は tcsh である場合もあると思いますが、tcsh は csh のほぼ上位互換なので、もちろんそれで構いません。

MS–Windows では、もちろん csh はデフォルトではインストールされていませんが、以下のようなフリーソフトを利用できます。

- wcsh – MS–Windows 上で動作する csh ライクなシェル
  `http://www.threeweb.ad.jp/~ishioka/wcsh/wcsh.shtml`
- Cygwin (MS–Windows で Unix 環境を構築)
  `http://www.cygwin.com`
- 泥縄流 Cygwin サポート 2006 (Cygwin 上の tcsh)
  `http://www.math.meiji.ac.jp/~mk/cygwin2006/`

ただし以下の説明は、すべて Unix 環境での話です。また以下の説明は、使用する csh によっては細かい点に違いがあるかもしれません。詳細は、使っている csh のマニュアル (`man csh`) を確認してください。

3 csh スクリプトの構造

シェルスクリプトは、基本的にバッチ処理を実現するものなので、ターミナル上で書いて実行させるコマンドラインを 1 行ずつ並べて書いて、それを csh に処理させれば、自動的に順に実行してくれます。

Unix の場合は、1 行目に

`#!` *command*

という行を書いて、そのスクリプトを実行可能 (`chmod u+x`) にしておけば、このスクリプトを実行することで、ここに書いたコマンド *command* の後にこのスクリプト名を与えて実行してくれます[1]。

よって、csh スクリプトでは、

[1] これはシェルスクリプトだけではなく、すべての種類のファイルに共通する仕様です。

```
#! /bin/csh -f
```

のように書くのが普通です。なお csh スクリプトでは、各行の '#' 以後はコメントと見なされるので、そのスクリプト (例えば script.csh という名前だとします) を普通にターミナル上で

```
% csh -f script.csh
```

と実行しても、その先頭行はコメントとして無視されるだけなので問題はありません。

なお、csh の '-f' オプションは、各ユーザの初期設定ファイル .cshrc を読みこまないためのオプションで、csh スクリプトを実行する場合は、通常このオプションをつけて実行します。csh のスクリプト処理時の他のオプションについては、6 節で紹介します。

4 csh の文法の概略

この節では、csh スクリプトの文法の概略を、多少分類しながら紹介していきます。

4.1 コメントと長い行の折り返し

3 節で述べたように、csh スクリプトでは各行の '#' から行末まではコメントと見なされます。行頭に書けば、その行が無視されます。

csh スクリプトは行毎に処理されますが、行末に '\' を書くとその行は次の行に連結されているとみなされますので、長い行を書く場合に便利です。

4.2 変数、リスト、引用符

csh スクリプトでは、変数を使用することができます。変数には、**環境変数** (setenv で設定) と **シェル変数** (set で設定) の 2 種類がありますが、環境変数は他のコマンドの動作にも影響を与えることがありますので、特別な目的以外にはあまり利用はせず、変数として主に利用するのはシェル変数の方です。

なお、環境変数名は大文字に、シェル変数名は小文字にするのが慣例となっています。

シェル変数の設定と参照は以下のように行います (*name* が変数名、*value* が値)。

- 設定: `set` *name* `=` *value*
- 参照: `$`*name* または `${`*name*`}`

シェル変数名 *name* はアルファベットまたは下線で始まり、アルファベットか数字か下線からなる 20 字以内の文字列です。ただし、以下の名前は特別な意味を持つので、その特別な機能を使用する場合以外には使ってはいけません (csh によっては、以下以外にも予約されている名前がある場合もあります)。argv 以外のそれぞれの詳しい意味については、csh のマニュアルや [3] などを参照してください。

> argv, cdpath, cwd, echo, filec, histchars, history, home, ignoreeof, mail, noclobber, noglob, nonomatch, notify, path, prompt, savehist, shell, status, time, verbose

参照の方の第 2 の形式は、その直後に文字や記号などを付ける場合に、変数名とそれ以外の部分との区別をするために `{ }`で変数名をくくります。

変数の値は、文字列、または **リスト** です。

変数値として数字を設定してもそれは文字列と見なされますが、`@` 計算式内や条件式で数字と解釈される状況では数字として評価されます。ただし、整数しか使用できません。

スペースが含まれる文字列は、引用符 (`" "` または `' '`) で囲むことで一つの文字列にできます。なお、`" "` (二重引用符) 内では、`$`*name* の展開 (その値への置き換え) が行われますが、`' '` (単一引用符) 内ではその展開は行われません。

リストは、文字列を要素とする配列のようなもので、詳しくは以下のように設定、参照できます。

- 設定:
  `set` *name* `=(` *str1 str2 ... strn* `)`
- 参照:
  `$#`*name* = リストの要素の個数
  `$`*name*`[`*n*`]` = リストの n 番目の値 ($1 \leq n \leq$ `$#`*name*)
  `$`*name* = リストの要素をスペースで挟んでつなげた文字列

なお、予約済みのリスト変数 `argv` は、シェルスクリプトの実行時に与えられたコマンドラインオプションをリストとして保存しますので、これを参照することで、コマンドラインオプションを処理するシェルスクリプトを作成できます。ただし、C 言語とは違い、`$argv[0]` はなく、代わりにそのシェルスクリプトファイル名自体は `$0` として参照します。

`$?`*name*、あるいは `${?`*name*`}` は、その変数が定義されているときに 1、定義されていないときには 0 という値になります。変数を削除 (未定義) にするには、`unset` 命令を利用します:

`unset` *name*

なお、数値を設定する場合は、4.3 節の `@` を使って変数を設定することもできます。

4.3 条件式と計算

csh スクリプトの条件分岐 (if) やループ (while) などでは、条件式を利用しますが、そこでは C 言語と同様の数式、条件式が使用できます。実際には数式、条件式は、

`@` の右辺、if や while の条件式部分 (かっこ内)、exit の引数 (かっこ内)

で使用できます。ただし、代入文は `@` でしか使用できません。

式による変数値の設定文 `@` は、基本的には以下のように使用します。

`@` *name* `=` *expr*

name は変数名、*expr* は式を表します。

使用できる数式、条件式は、表 1 の通りです。このうち、和や積などは普通の数式とは違い右から左へ計算するので、適宜かっこが必要なことに注意してください。

`==`, `!=` は、両辺を文字列として等しいか等しくないかの判別を行います。また、`=~`, `!~` は、その左辺の文字列が、右辺のパターン (ファイル名置換パターン) と一致するか (`=~`) 一致しないか (`!~`) を判別します。

csh スクリプトのパターンには、通常の文字列に加えて、表 2 の特殊文字が使用できます。ただし、`[ ]` 内には、`-` を使って範囲を指定することも可能です。例えば、

```
200[0-7].[A-Z][a-z][a-z].[0-9][0-9]
```

というパターンは、“2007.Dec.27” のような文字列に一致 (マッチ) します。

なお、このパターンを含む単語が csh スクリプトのコマンドラインで使用された場合、その単語はそれにマッチするすべてのファイル名をスペースで区切った文字列に展開されます。例えば

式	意味
( )	式のグループ化 (通常の数式のかっこ)
~	1 の補数
!	論理否定
*, /, %	整数演算での積、商、剰余 (右から)
+, -	和、差 (右から)
<<, >>	ビット単位の左シフト、右シフト
<, >, <=, >=	大小の不等号
==, !=, =~, !~	等号、不等号、パターンの一致、不一致
&	ビット単位の AND
^	ビット単位の XOR
|	ビット単位の OR
&&	論理式の AND
||	論理式の OR

表 1: 使用できる数式 (上の方ほど優先順位が高い)

特殊文字	意味
*	0 個以上のすべての文字に一致
?	1 個のすべての文字に一致
[...]	かっこ内のすべての 1 文字に一致

表 2: パターンの特殊文字

```
cp ab*.tex dir
```

というコマンドラインでは、パターン文字列を含む単語が実際に存在するファイル名による

```
cp ab.tex ab12.tex ab23.tex abc.tex dir
```

のようなコマンドラインに展開されてから実行されます。

@ 文では、代入式 = の代わりに C 言語と同様に += や *= のような代入文も使用できます。また、後置のみですが、++, -- も使用できます。例:

```
@ j ++
```

さらに、式には「*-l name*」という形式の表 3 のファイル検査式も使用できます。もちろん、! を *-l* の前につければ意味が逆になります。

式	意味
`-r` *name*	*name* が読み込み可なら真 (1)
`-w` *name*	*name* が書き込み可なら真
`-x` *name*	*name* が実行可なら真
`-e` *name*	*name* が存在していれば真
`-o` *name*	*name* の所有者がユーザなら真
`-z` *name*	*name* のサイズが 0 なら真
`-f` *name*	*name* が通常のファイルなら真
`-d` *name*	*name* がディレクトリなら真

表 3: ファイル検査式

4.4 コマンド置換

csh スクリプトでは、‘ ‘ (バッククォート) で囲んだ文字列はコマンドとみなされ、その実行結果の標準出力の改行をスペースで置き換えた文字列に置きかえられます。これは、" " (二重引用符) 内でも展開されますが、’ ’ (単一引用符) 内では展開されません。例えば、

```
set s = "‘date‘"
```

とすると、変数 s には date コマンドの出力であるその日の日付

```
Thu Dec 27 16:31:32 JST 2007
```

のような文字列がセットされます。

4.5 サポートコマンド

ここでは、csh スクリプト内で特に良く利用するであろうコマンドを簡単に紹介します。表 4 に紹介するのは、csh 自体に組み込まれている内部コマンドです。もちろんこれ以外にも内部コマンドはありますが、その他のコマンド、またここに紹介するコマンドの詳しい説明は、csh のマニュアル (`man csh`) を参照してください。なお、`echo` は外部コマンドとしても利用可能な Unix も多いと思います。

表 5 は、csh スクリプトでの利用が多いと思われる Unix の標準的な外部コマンドです。使用法等は、各コマンドのオンラインマニュアルや Unix の一般書 (例えば [3]) などを参照してください。

コマンド	意味
`cd` *dir*	ディレクトリ *dir* へ移動
`shift` *name*	リスト変数 *name* の先頭を削除 (*name* を省略すると `argv`)
`echo` *string*	文字列 *string* を画面 (標準出力) 表示
`pushd` *dir*	現在のディレクトリをスタックに入れて *dir* へ移動
`popd`	スタックから最後のディレクトリを取り出し、そこへ移動
`eval` *string*	文字列 *string* をコマンドラインと見なして実行

表 4: csh の代表的な内部コマンド

コマンド名	意味
`bc`	数式の結果を出力
`cat`	複数のファイルの内容を連結
`tee`	標準入力を標準出力と指定ファイルに出力
`date`	日付を指定した書式で出力
`head`, `tail`	ファイルの先頭、最後のみを出力
`basename`, `dirname`	パスのファイル部分、ディレクトリ部分を出力
`sort`	ファイルの内容をソートして出力
`grep`	ファイルからパターンにマッチした行を出力
`tr`	単純な文字の変換、削除等を行うフィルタ
`sed`	tr より高機能のパターン処理を行うフィルタ
`awk`	sed より高機能な処理を行うフィルタ
`sleep`	指定秒間作業を中断する

表 5: csh スクリプトでよく使用される外部コマンド

4.6 構文

csh スクリプトでは、作業の流れを制御する、C 言語に似た以下のような構文が利用できます。

- if 文 (条件分岐)
- while 文 (条件式に基づくループ)
- foreach 文 (リストに基づくループ)
- switch 文 (文字列の値による分岐ジャンプ)
- goto ジャンプ (無条件ジャンプ)
- exit 命令 (強制終了)

if 文には、2 種類の形式があり、一つは

```
if ( expr ) command
```

で、*expr* は式が入り、その値が真 (0 以外) ならコマンド *command* が実行されます。この場合、*command* 部分には複数のコマンドは書けません。

もう一つは、

```
if ( expr ) then
    command
    ...
else if ( expr ) then
    command
    ...
else
    command
    ...
endif
```

という形式です。この形式では、`else if` ブロックはいくつあっても構いませんし (なくても構いません)、`else` ブロックもなくても構いません。各ブロックでは、複数のコマンド行を書くこともできますし、もちろんその中にさらに if 文を書くことも可能です。

while 文は、以下のような形式です。

```
while ( expr )
    command
    ...
end
```

C 言語の while 文と同様に、式 *expr* が真 (0 以外) である間、while ブロックの `end` までのコマンドの実行を繰り返します。C 言語でも使用する break 命令や、continue 命令も使えます。

foreach 文は、以下のような形式です。

```
foreach var ( str1 ... strn )
    command
    ...
end
```

これは、まず後ろのかっこ内 (リスト) の先頭の値 *str1* を変数 *var* に代入して `end` までの一連のコマンドを実行し、次に 2 番目の値を *var* に代入して実行し、それをリストの最後の値 *strn* まで繰り返します。これは C 言語には直接対応するものはありませんが (for 文とは少し違います)、ある種のループになっていて、やはり break や continue が使用できます。

switch 文は以下のような形式です。

```
switch ( string )
case label1:
    command
    ...
    breaksw
case label2:
    command
    ...
    breaksw
default:
    command
    ...
    breaksw
endsw
```

これは C 言語の switch 文とほぼ同様ですが、*string* を文字列として *label1*, *label2* 等と比較し、C 言語では各 `case` ブロックから抜け出すのに `break` を使いますが、csh スクリプトの switch 文では `breaksw` を使い、switch 文の終わりは `endsw` と書きます。

goto ジャンプ命令は、

```
goto label
```

のようにジャンプ先ラベル名を指定して書くだけです。ジャンプ先ラベルは、

```
label:
```

のように名前の後ろに : をつけて書いただけの行を任意の位置に置きます。

`exit` は、単にそこでスクリプトを強制終了します。引数をつけて

```
exit ( expr )
```

とすることもできますが、あまり使用することはないと思います。

5 パイプ、リダイレクション等

Unix では、コマンドの標準入力 (stdin, 通常はキーボードからの入力)、標準出力 (stdout, 通常は画面出力) をファイルに切り替えたり、他のコマンドに渡すための仕組みがあります。基本的には以下のように使います。

- *command* < *file* : **入力リダイレクション** (*file* の内容を *command* の標準入力とする)
- *command* > *file* : **出力リダイレクション** (*command* の標準出力を *file* に書き出す)
- *command1* | *command2* : **パイプ** (*command1* の標準出力を *command2* の標準入力とする)

これは、以下のように多段に組み合わせて使うことも可能です。

```
% com1 < file1 | com2 | com3 > file2
```

この場合、

- *file1* の内容を標準入力として *com1* へ渡し、
- その *com1* の標準出力が *com2* の標準入力へ渡され、
- その *com2* の標準出力が *com3* の標準入力へ渡され、
- その *com3* の標準出力が *file2* として保存される

ということになります。上の *com1, com2, com3* のように、標準入力からデータをもらい、それを加工して標準出力に流すプログラムを **フィルタ** と呼びます。Unix ではこの仕組みにより、単一の仕事しかしない複数のフィルタを組み合わせて必要な多段階の加工を行う、という作業がよく行われ、そのための色々なフィルタが用意されています (4.5 節の外部コマンドの多くもその類いです)。

出力リダイレクションには、次のような別の形式もあります。

- *command* >> *file* : *file* の最後へ追加出力 (> の場合は上書き)
- *command* >& *file* : 標準エラー出力 (stderr) も合わせて *file* に出力
- *command* >>& *file* : 標準エラー出力 (stderr) も合わせて *file* に追加出力

また、<< は、**ヒアドキュメント** として使われます。

command << *name*
string
...
name

これは、*name* までの文字列の並びを *command* への標準入力として渡します。一行の文字列なら、コマンド echo を使って

echo "*string*" | *command*

としても渡せるのですが、ヒアドキュメントは複数行の文字列を渡すときに便利です。*name* は渡す文字列内に現れない任意の文字列を使用できます。*name* の代わりに '*name*' とすれば文字列部分の変数展開は行われません。

パイプ以外で、複数のコマンドを 1 行で連続実行することもできます[2]。

- *com1* ; *com2* : *com1* の次に *com2* を実行
- *com1* && *com2* : *com1* が正常終了した場合のみ *com2* を実行
- *com1* || *com2* : *com1* が異常終了した場合のみ *com2* を実行

3 つ以上を並べることも可能です。

これらの ';', '||', '&&' を含んだコマンド列を ( ) でくくると、それ自体が一つのコマンドとみなされ、それをまたコマンド列の要素とすることもできます。例:

com1 | (*com2* > *file1* ; *com3 file1*) | *com4*

6 コマンドラインオプション

csh のスクリプト実行時のコマンドラインオプションには、以下のようなものがあります。

- -f : ユーザのホームディレクトリにある .cshrc などの csh 初期化ファイルを読み込まず、よって動作が少し早くなります。
- -n : シェルスクリプトの構文解析は行いますが、実行はしません。

[2] この場合の正常終了、異常終了は、コマンド終了時にセットされる特別変数 status の値で判別します (0 ならば正常終了、それ以外なら異常終了)。この値は、C 言語の main 関数の返り値、あるいは exit の引数として与える符号なし 1 バイト値です。

- -s：標準入力からシェルスクリプトの内容を受け取ります。
- -v：特殊変数 verbose をセットし、それにより実行される各コマンドが、実行前に画面表示されます。
- -x：特殊変数 echo をセットし、それにより変数置換が行われた後の各コマンドが (コマンド列の場合は各コマンドに分解されたコマンドが)、実行前に画面表示されます。

7 注意

以下に、いくつか注意を上げます。

- スペースや改行
  csh スクリプトでは、先頭行の #! 以外では、見やすさのために行頭にインデント (いくつかのタブやスペース) を入れることは構いません。適宜空行を入れることも可能です。

  ただし、スペースを入れるべきところにスペースを入れないとエラーとなります。例えば if や while とカッコの間は、C 言語とは違いスペースを入れる必要があります。

  set も、= の左右のスペースが合っていないとエラーになるようで、“set␣a=3”, “set␣a␣=␣3” はいいのですが、“set␣a␣=3”, “set␣a=␣3” はだめなようです。

- 変数展開と構文解釈のタイミング
  csh スクリプトでは、各行を読み込んだときに、行全体の変数展開が行われてから構文の解釈が行われるようなので、例えば if 文では以下はエラーになりえます。

```
if ( $#argv > 0 ) goto $argv[1]
if ( $#argv > 0 && $argv[1] == "2" ) then
    ...
```

  C 言語ならば、最初の方は $#argv が 0 のときは goto の部分は無視されるので、$argv[1] が定義されていようがいまいが関係ないのですが、csh スクリプトの場合は、この if 文の評価の前に $#argv と $argv[1] の展開をするので、$#argv が 0 ならば $argv[1] が定義されていないというエラーになってしまいます。後者も、C 言語ならば $#argv が 0 のときは && の後半は無視されるのですが、csh スクリプトでは先に $#argv と $argv[1] の展開をしてから if 文の評価をしようとするのでエラーとなります。

- - から始まる文字列
  Unix では “-h” のように、- から始まる文字列をオプションとして使うことが多いですが、そのような文字列を直接以下のように比較するとエラーとなります:

```
if ( $argv[1] == -h ) then
    ...
```

この場合例えば $argv[1] に “-h” のような文字列が入っていると、if 文のカッコ内が “-h” から始まることになり、それがファイル検査式と判断されようとしてエラーになるので、先頭に余計なアルファベットを付加して、

```
if ( "X$argv[1]" == X-h ) then
    ...
```

のようにしないといけません。

なお、while 文のカッコ内も式の評価が行われますので同じ問題が起こりますが、switch 文のカッコ内は式の評価が行われませんので、このような対処は不要です。

- リスト変数に部分リスト変数を代入
  リスト変数は、それ自体を他のリスト変数に部分リストとして代入することで結合することができます。例:

```
set s1 = ( 3 4 5 )
set s2 = ( 1 2 $s1 6 7 )
```

とすると、s2 には (1 2 3 4 5 6 7) が入ります。csh スクリプトでは、変数の展開が行われてから式の評価が行われるので、以下の s2 も上と同じ結果になります:

```
set s1 = "3 4 5"
set s2 = ( 1 2 $s1 6 7 )
```

しかし、これを

```
set s1 = ( 3 4 5 )
set s2 = ( 1 2 "$s1" 6 7 )
```

としてしまうと、$s2[3] が “3 4 5” という文字列、$#s2 は 5 となります。だから、逆に、

```
set s1 = ( "1 2" 3 )
set s2 = ( $s1 4 5 )
```

とすると、s1 が持っていたスペースの含まれる文字列要素 $s1[1] を s2 では保持することができず、バラバラの要素に分かれてしまいます。特に、‘*’, ‘?’, ‘[ ]’ のような特別な文字が含まれる文字列を要素として持つ場合は、引用符で囲まれなくなった時点でファイル名パターンマッチ機能が働こうとしておかしくなり得ます。

- サブルーチン
  csh スクリプト内ではサブルーチンを書けません。goto ジャンプを使えば多少は実現できますが、goto ジャンプでは while や foreach 文などのループの中に戻ることはできません。

参考文献

[1] Tom Christiansen (Hiroki Mori 訳)、「なぜ csh でプログラムを書くのが良くないのか」
`http://www.klab.ee.utsunomiya-u.ac.jp/~hiroki/csh-whynot.euc`、(1997/09/28)

[2] Hiroki Mori、「あなたはこれで C シェルを捨てられる」
`http://www.klab.ee.utsunomiya-u.ac.jp/~hiroki/sh-perl-tips/`、(1998)

[3] 山口和紀監修、「The UNIX Super Text 上」技術評論社 (1992)