2010 年 07 月 02 日
(2011 年 05 月 02 日改訂)
(2013 年 06 月 20 日改訂)
(2013 年 07 月 12 日改訂)
(2013 年 07 月 15 日改訂)
(2014 年 05 月 02 日改訂)

バッチファイルの概要

新潟工科大学 情報電子工学科 竹野茂治

1 MS–Windows のスクリプト環境

Unix には「小さいプログラムを使い回したり、組み合わせて使う」ための環境として、対話型シェル環境やシェルスクリプトがあります。MS–Windows にもそれに近いものとして、以下のものが用意されています。

- コマンドプロンプト + バッチファイル (以下、バッチ環境)
- WSH (Windows Scripting Host)
- Windows PowerShell

	バッチ環境	WSH	PowerShell
対話環境	コマンドプロンプト	なし	ある (専用のシェルウィンドウ)
スクリプト言語	バッチファイル (.bat)	VBScript (.vbs), JScript (.js) 等	専用のスクリプト言語 (.ps1)
構文や変数	一応使える	使える	使える
外部コマンドやモジュール	一応ある	ある (オートメーションオブジェクト)	一応ある
パイプ、リダイレクション	ある (ファイルベース)	少なくとも簡単なものはなさそう	ある (オブジェクトベース)
実行ファイル	cmd.exe	wscript.exe (cscript.exe)	PowerShell.exe
XP では	そのまま使える	そのまま使える	要インストール
歴史	MS–DOS から[1]	Windows 98 から[2]	2007 年リリース[3]

表 1: バッチ環境、WSH、PowerShell の特徴

本稿ではこれらのうち、手軽で、非常に古くから使われているバッチファイルによるスクリプトプログラミングについて紹介します。これにより、小さいプログラムしか作れない人、あるいはプログラムを作ったことがない人でも、定型作業の自動化や、複数のソフトの連携、小さなソフトの繰り返しの使用などが行えるようになるのでは、と思います。

なおコマンドプロンプトでは、通常

```
C:¥Documents and Settings¥hoge¥My Documents> dir
```

のように、現在自分のいるディレクトリが記号 > の左に表示されていて、ユーザが実際に入力するのは > より右側の部分 (上の例では dir) なのですが、これだと無駄な表示部分が長いので、以後コマンドプロンプトの例示では、特別な場合を除いて、カレントディレクトリ部分は省略して

```
> dir
```

のように表記することとします。そして、このように > から始まる例示はコマンドプロンプトでの実行例を意味することにしますので、実際に試す場合は先頭の > は入力しないでください[4]。なお、この > も含めてその左側の部分に表示されるものについては、prompt コマンドで変更することができます (詳しくは help prompt 参照)。

また、本稿は MS–Windows XP のコマンドプロンプトを仮定して書いているため、XP 以前の MS–Windows (特に「MS–DOS プロンプト」と呼ばれていたソフト) には当てはまらない記述も多少ありますので注意してください。

2 バッチファイルとは

バッチファイルとは、拡張子 .bat のテキストファイルで[5]、基本的には、コマンドプロンプト上で実行できるコマンドを並べて書いたもので、cmd.exe というソフト (コマンドインタープリタ) が、それを一行ずつ解釈しながら実行してくれます。C 言語とは違いバッチファイルをコンパイルする必要はなく、そのファイルを直接単独で実行できます。

[1]正確には MS–DOS の頃は command.com が使われていました。その拡張版である cmd.exe は Windows NT 以降のようですが、現在は PowerShell への移行が推奨されているようです。

[2]WSH も PowerShell への移行が推奨されていて、今後はメジャーバージョンアップもないそうです。

[3]PowerShell はまだ新しいので、安定性や仕様の変更も含めて今後どうなるかわかりません。

[4]ちなみに、このユーザに入力をうながしている > をプロンプトと呼びます。「コマンドプロンプト」とは、元々「コマンドの入力をうながす」という意味です。

[5]テキストファイルとは、「メモ帳」等のエディタで編集する可読なファイルを意味し、逆に .doc, .xls, .pdf 等の特定のソフト専用の形式でメモ帳では見られないものをバイナリファイルといいます。

バッチファイルは、通常

- エクスプローラ上でそのバッチファイルを示すアイコンをクリックする
- コマンドプロンプト上でそのバッチファイル名を入力する

のいずれかの方法で実行します。例えば

```
dir
date /t
chkdsk c:
```

と書いたバッチファイル (ファイル名を test1.bat とします) を、コマンドプロンプトで

```
> test1.bat
```

あるいは拡張子をつけずに

```
> test1
```

と実行すれば、現在のディレクトリ内のファイルの一覧を表示し (dir)、現在の日付を表示し (date /t)、C ドライブのチェックを実行して表示します (chkdsk c:)。なお、コマンドプロンプトやバッチファイル内では、コマンド名やパラメータは大文字、小文字の区別はされませんので、本稿では主に小文字で表記することにします[6]。

バッチファイルの編集は「メモ帳」(notepad) でもできますが、キーワードの色づけ機能や高度の編集機能を持つ、よりプログラマ向けのエディタ (MkEditor, TeraPad, Notepad++ 等) を用いる方が効率的でしょう[7]。

バッチファイルで利用できる構文や、MS–Windows に用意されている基本コマンドにはそれらのヘルプドキュメントも用意されていて、例えば

```
> if /?
> dir /?
```

のようにコマンド名やキーワードの後ろに /? をつけるか、help コマンドを用いて

[6] 実は、コマンドプロンプトやバッチファイルに関する本のほとんどがコマンドやパラメータはなぜか大文字で書いてあるのですが...

[7] ただし、これらも含め最近の MS–Windows 向けの多くのエディタの編集機能は、昔の MS–DOS の時代のエディタよりも編集効率は悪いような気がしています。

```
> help if
> help dir
```

とすれば、その説明が表示されます。その出力をファイルに保存したければ、

```
> if /?  > if.txt
```

のようにすれば、if の説明文が if.txt というファイルに保存されます (この > の使い方については 5 節参照)。単に

```
> help
```

とすると、コマンドプロンプト用に用意されているコマンドの一覧が表示されます。

3 バッチファイルの変数

バッチファイル内では変数が使用できます。局所変数も使えますが、本稿では主に大域変数である 環境変数 を使用します。ただし、環境変数はすでに設定されているものもありますので、一時的な変数としてはそれらと同じ名前は使用しないでください。環境変数の一覧は set コマンドを用いて

```
> set
```

とすると表示されます。環境変数の設定、削除はそれぞれ

```
set␣[変数名]=[値]
set␣[変数名]=
```

のようにします[8]。なお、スペースをそれとわかるように ␣ で表しましたが、上にあるように = の前後にはスペースを入れてはいけません。値は基本的には文字列で、値文字列にスペースが含まれていても文字列全体を引用符で囲む必要はありません[9]。

環境変数の値を利用するときは、変数名を % で囲んで %[変数名]% とします。例えば、

```
> set␣a=竹の
> set␣b=%a%␣茂治
> echo␣%b%
```

[8]実際の設定ではかっこ [ ] は不要です。
[9]逆に文字列を引用符で囲むと、引用符自体も値に含まれてしまいます。

とすると「竹の 茂治」と表示されます。ここで echo は、その後ろに指定した文字列を画面に表示するコマンドです[10]。

現在のコマンドインタプリタ (の拡張機能) では、変数の整数値、およびその計算も set コマンドでできるようになっていて、

```
set␣/a␣[変数名]=[式]
```

で式の整数演算の結果を変数に代入できます。式には、

- ( ) (カッコ)
- ! (論理反転)、~ (ビット反転)、- (符号反転) の単項演算子
- *, /, +, -, % 等の四則演算子 (% は剰余)
- C 言語と同等の <<, >>, &, ^, | 等のビット演算子

が使えますし、代入を意味する = も、C 言語と同様の演算付きの代入記号

```
*=, /=, +=, -=, %=, <<=, >>=, &=, ^=, |=
```

で置きかえることが可能です。

なお、剰余演算を意味する % は、コマンドプロンプトの対話的な実行時は % でいいのですが、バッチファイル中では % は特別な意味を持つため、2 つ重ねて %% とする必要があります。また、上記の演算子のうちいくつかは他の意味にも使われるので、式を二重引用符で囲む必要がある場合 (例えばビット演算子) もあります。

整数値は、デフォルトでは 10 進数ですが、頭に 0x をつければ 16 進数、0 をつければ 8 進数として扱われます。

また、「set /a」の右辺では変数の値は % で囲まずに単に変数名だけで参照できることになっています。よって、例えばバッチファイルで

```
set x=10
set y=4
set /a y+=3*x%%7
echo %y%
```

[10] echo には、特にバッチファイルでよく用いられる別の用法もあります。詳しくは 4.4 節参照。

とすると、結果として 4 に 3×10 を 7 で割った余りを加えた 6 が表示されることになります。3 行目の /a を書かないと、= の後の式「3*x%7」が文字列として「y+」という変数に代入されてしまうことになります。

さらに、カンマ演算子 (,) があり、1 行の式の中で以下のような逐次評価が行えます。

```
set x=1
set /a x+=(a=2, b=2*a, c=2*b, 2*c)
echo %a% %b% %c% %x%
```

これは「2 4 8 17」と表示されます。

また、コマンドライン上でバッチファイルに与えたオプションパラメータは、バッチファイル内では順に %1,%2,...,%9 として参照できます。%0 はバッチファイル自身の名前を指し[11]、さらに %* は %1 以降のオプション文字列全体を表します。

例えば、test.bat というバッチファイルを

```
> test.bat␣123␣竹の␣茂治
```

として実行すると、test.bat 内では

```
%0=test.bat, %1=123, %2=竹の, %3=茂治, %*=123 竹の 茂治
```

となります。%4 以降はこの場合空文字列となります。

なお、バッチファイルのオプションの区切りは、スペースの他にカンマ (,) も利用できるので、

```
> test.bat␣123,竹の␣茂治
```

としても %* 以外は上と同じ結果になりますが、%* は

```
%*=123,竹の 茂治
```

となります。

さらに、動的に変化する以下の変数があります。なお、これらは環境変数の一覧表示の set コマンドでは表示されません[12]。

[11] これは C 言語の main() の引数と似た仕組みです。
[12] 通常の静的な環境変数とは扱われ方が別なようです

- %cd% : ドライブ名付きのカレントディレクトリ
- %date% : 現在の日付 (例: 2010/06/26)
- %time% : 現在の時刻 (例: 16:48:54.26)
- %random% : 0 から 32767 の間の乱数
- %errorlevel% : 直前のコマンドの終了コード

4 バッチファイルの構文

バッチファイルには、if 文、for 文、goto 命令等の簡単な構文が用意されていて、これにより一応ループ処理、条件分岐等が行えます。本節ではそれらを順に説明していきます。

なお、バッチファイルには、C 言語での while 文や switch 文に相当するものはありません。

4.1 if 文

if 文は、以下の 4 種類の形式が使用できます。

1. 1 行で if のみ

   ```
   if␣[条件]␣[コマンド]
   ```

2. 1 行で if と else

   ```
   if␣[条件]␣[コマンド 1]␣else␣[コマンド 2]
   ```

3. 複数行で if のみ

   ```
   if␣[条件]␣(
       [コマンド 1]
       [コマンド 2]
       ..........
   )
   ```

4. 複数行で if と else

```
if␣[条件]␣(
    [コマンド A1]
    [コマンド A2]
    ............
)␣else␣(
    [コマンド B1]
    [コマンド B2]
    ............
)
```

if 文の後ろのコマンドは条件部分が真のときに実行され、else の後ろは条件部分が偽のときに実行されます。実行するコマンドが複数ある場合は 3. または 4. の形式を使いますが、3., 4. の形式はコマンドが 1 つしかない場合でも使えます。

if 文の条件部分には、以下の形式が使えます (「 」や [ ] は不要です)。

- 「[値 1]==[値 2]」 (等しければ真)
- 「[値 1]␣[比較演算子]␣[値 2]」 (文字列、数値の比較)
- 「exist␣[ファイル名]」 (ファイルやディレクトリが存在すれば真)
- 「errorlevel␣[番号]」(直前のコマンドの終了コードが番号以上なら真)
- 「defined␣[変数名]」 (その環境変数が定義されていれば真)

さらに、以上の条件の前に not をつけるといずれの場合も真偽が反転されますが、C 言語のように複数の条件を AND (&&) や OR (||) で結ぶことはできません。

この条件の形式の 1 番目と 2 番目の形式の値の部分には文字列が使えますが、文字列の代わりに %[変数名]% を置くこともできますし、それを文字列中の部分文字列とすることもできます。また、2 番目の形式では、両辺の値がどちらも整数値を表す文字列であれば整数値として比較が行われます。比較演算子はいずれも 3 文字の文字列で、

- leq : 以下 ($\leq$ の意。**l**ess than or **eq**ual の略か)
- geq : 以上 ($\geq$ の意。**g**reater than or **eq**ual の略か)
- equ : 等しい (= の意。**equ**al の略か)
- neq : 等しくない ($\neq$ の意。**n**ot **eq**ual の略か)
- lss : より小さい ($<$ の意。**l**e**ss** than の略か)

- gtr : より大きい (> の意。greater than の略か)

のいずれかが指定できます。

値が文字列ならば辞書式順序で比較され、デフォルトではアルファベットの大文字と小文字は区別されますが[13]、2 番目の比較演算子を用いた形式では if と条件の間に /i を置くと、大文字と小文字を区別せずに比較します。

4.2 for 文

for 文は、主に以下の形式で使用します。

1. 指定リスト分だけ実行

```
for␣%[変数名]␣in␣([リスト])␣do␣(
    [コマンド]
    ..........
)
```

2. 単調な整数リストに対して実行 (C 言語の for 文似)

```
for␣/l␣%[変数名]␣in␣([初期値],[増分],[終了値])␣do␣(
    [コマンド]
    ..........
)
```

3. テキストファイルの各行に対して実行

```
for␣/f␣"[オプション]"␣%[変数名]␣in␣([ファイルリスト])␣do␣(
    [コマンド]
    ..........
)
```

if 文の場合と同様、実行するコマンドが 1 つしない場合は、1 行で

```
for␣%[変数名]␣in␣([リスト])␣do␣[コマンド]
for␣%[変数名]␣in␣([リスト])␣do␣([コマンド])
```

[13]文字列比較は「0 < 9 < a < A < z < Z」のような順になっていて、ASCII コード順ではありません。また、大文字、小文字が混じると厳密には辞書式順序ではなく、「a < A < aa < aA」のようになっているようです。

のように書くことも可能です。

for 文の局所変数 %[変数名] の変数名はアルファベット 1 文字であること、およびバッチファイル内で使うときは、%[変数名] は %%[変数名] と % を重ねる必要があることに注意してください。この局所変数は、値を参照するときも、set /a の右辺に書く場合でも %[変数名] (バッチファイル内では %%[変数名]) として使用します。

上の 1. の形式の場合は、リストには複数の文字列を書くことができ、それが各変数に代入されてその分だけコマンドが実行されます。リストの各要素の区切りは、スペースかカンマ (,) が使用できます。リストとしてワイルドカードを含む文字列を指定した場合は、ワイルドカードにマッチするファイル名を探してそれをリストの要素とします (ワイルドカードについては 6 節参照)。例えば

```
> for %a in (dog cat mouse) do echo %a
```

とすると、%a には順番に dog, cat, mouse という文字列が代入されて、そのそれぞれに対してコマンド「echo %a」が実行されるので、結果として

```
dog
cat
mouse
```

と表示されることになります。

2. の形式は、C 言語の for 文とは増分指定 (C では 3 つ目)、終了条件 (C では 2 つ目) の指定の順番が違いますが、ほぼ似た使い方ができます。増分や整数値には負の値も使用できます。例えば、

```
> for /l %a in (9,-2,-4) do echo %a
```

とすると、%a には順に 9,7,5,3,1,−1,−3 が代入されて do の後ろのコマンドが実行されます。

3. の形式では、「"[オプション]"」は省略することもできますが、その場合は指定したファイルの各行の、タブや空白で区切られた第 1 フィールド (最初の単語) を %[変数名] に代入してコマンドを実行します。第 2 フィールド以降を利用したり、フィールドの区切り文字を指定したりもできますが[14]、それらに対するオプションの指定方法、および 1., 2., 3. 以外の for 文の形式については、for /? によるヘルプを参照してください。

[14] ファイルの各行をフィールド (単語) に分割して利用する方法は、AWK 等のスクリプト言語を少し意識したような仕様です。

4.3 goto 命令

goto 命令は

```
goto␣[ラベル名]
```

の形式で用いて、そのバッチファイル内の任意の箇所へのジャンプを行います。ラベルは、

```
:[ラベル名]
```

の形の行で、任意の位置に置けます。例えば、バッチファイルで

```
set j=1
:start1
if %j% gtr 10 goto last1
[コマンド1]
...........
set /a j+=2
goto start1
:last1
```

と書くと、これは

```
for /l %%j in (1,2,10) do (
    [コマンド1]
    ...........
)
```

と書いたのとほぼ同じことになります[15]。

4.4 コメント行と画面表示の消し方

バッチファイルでは rem␣ で始まる行をコメント行とみなし、その行の実行を無視します。バッチファイルに説明を書いたり、一時的に実行させない行の先頭に rem を書いて無視させたりするのに使えます。

[15] これは、後者の for を用いる方がわかりやすいようですが、デフォルトでは遅延展開が無効になっているという問題もあります。前者ならそれを考慮する必要はありません。詳しくは、8 節 の 22. 参照。

バッチファイルをコマンドプロンプトで実行すると、デフォルトではコメント行も含め、その中に書かれている行も 1 行 1 行画面に表示しながら実行します。バッチファイルに書かれたもの自体は表示しないようにするには、バッチファイルの先頭に

```
@echo off
```

と書きます。「echo off」は以降のバッチファイルの内容を表示しない、という設定のためのコマンドで、デフォルトでは「echo on」の状態になっています。@ をコマンドの前につけると、そのコマンド行を表示しません。よって、これでバッチファイル内の記述自体はすべて表示しなくなります[16]。

echo を off にしても、バッチファイルに書かれている文字列を表示しなくなるだけで、コマンドの出力や文字列を表示する echo コマンドの出力は画面に表示します。

コマンドの出力自体も画面に出したくない場合は、そのコマンドの最後に「> nul」をつけて

```
[コマンド] > nul
```

のようにします[17] (> nul」の意味については 5 節参照)。

4.5 強制終了

バッチファイルを強制終了するには exit コマンドを使用します。ただし、単に「exit」とすると、そのバッチファイルを実行しているコマンドプロンプトまで終了してしまうので、バッチファイルでは

```
exit␣/b
```

とします。そのバッチファイルが別なバッチファイル (親) から call コマンドで呼ばれている場合は、そのバッチファイルの処理を終了した後で親のバッチファイルの呼ばれた箇所に処理が戻ります。またその際、

```
exit␣/b␣[終了コード]
```

のように後ろに終了コードを書けば、親のバッチファイルに戻った側でその終了コードを if errorlevel での分岐に利用できます (詳しくは 4.1 節、4.7 節参照)。

[16]逆に先頭の「echo off」に @ をつけないと、その行の時点ではまだ「echo on」の状態なので、その行だけは表示されてしまうことになります。

[17]このようにしても、コマンドのエラーメッセージなどは表示されてしまうことがあります。詳しくは 5 節参照。

4.6 一時停止

バッチファイルの処理を一時停止するには pause コマンドか、「set /p」が使えます。

pause コマンドは「続行するには何かキーを押してください」のようなメッセージを表示し、ユーザが何かキー入力を行うまでバッチ処理を中断します。そのメッセージを変更したければ、4.4 節で紹介した「> nul」を用いて

```
echo [中断時に表示させる任意のメッセージ]
pause > nul
```

のようにすればいいでしょう。一方、「set /p」はキー入力を受け取ってそれを環境変数に代入する命令で、

```
set␣/p␣[変数名]=[表示メッセージ]
```

のように使用します。この表示メッセージの部分が表示されて入力待ちとなり、そこでキー入力したものが指定した環境変数に代入されます。これは、バッチファイルの処理の流れを対話的に分岐するのに利用できます。例えば、

```
:check1
set /p isy=これでいいですか？ (Y/N)
if "%isy%"=="y" goto yes1
if "%isy%"=="Y" goto yes1
if "%isy%"=="n" goto no1
if "%isy%"=="N" goto no1
goto check1
:yes1
.....
:no1
.....
```

といった具合です。

4.7 別バッチファイルの呼び出し

call コマンドを用いて

```
call␣[バッチファイル]
```

とすれば、バッチファイル内から別なバッチファイル (子) を呼び出すことができます。この場合、バッチファイル名の後ろに子のバッチファイル用のオプションパラメータを指定することもできます。

呼び出した子のバッチファイルが終了したら、処理は元 (親) のバッチファイルに戻り、その call 行の次の行に処理が移ります。

子のバッチファイル内で終了時に

```
exit /b [終了コード]
```

の形式で終了コードが指定してあれば、通常のコマンドの終了コードと同様に親のバッチファイル側でそれを if errorlevel の分岐に利用できます。

5 パイプとリダイレクション

コマンドプロンプトには、Unix にもある「パイプ」と「リダイレクション」という仕組みが用意されていて、コマンドとファイルとのデータのやりとり、データファイルの加工、コマンド間のデータの受け渡しなどが行えるようになっています。本節では、そのパイプとリダイレクションの概要を説明します。

コマンドプロンプト上で利用できる dir や date のような非ウィンドウソフトは、コマンドプロンプトのウィンドウ画面上に出力を表示します。また、`set /p` のようなコマンドはキーボードから入力をもらいます。これらの出力、入力は、

- 標準出力 (stdout: 通常はコマンドプロンプト画面)
- 標準入力 (stdin: 通常はキーボードからの入力)

と呼ばれています。C 言語で言えば、printf() や putchar() 等の関数は標準出力への出力を行い、scanf() や getchar() 等は標準入力から入力をもらう関数です。

この標準出力、標準入力は、コマンドプロンプトの リダイレクションの機能を利用すれば、それらを (コマンド毎に) ファイルに切り替えることができます。コマンドへの標準入力をファイルへ切り替えるには、コマンド (やオプションパラメータ) の後ろに入力リダイレクション

```
< [ファイル名]
```

を追加します。これは、対話的に入力が必要なコマンドへの固定入力を自動化するのに利用できます。

コマンドの標準出力をファイルに切り替えるには、コマンド (やオプションパラメータ) の後ろに出力リダイレクション

> [ファイル名]

を追加します。これはコマンドの出力を保存するのに使えます。

入力も出力もファイルに切り替えるには、コマンドの後ろに

< [ファイル 1] > [ファイル 2]

を追加します。この場合、ファイル 1 とファイル 2 は別な名前でないと問題が起こるでしょう。なお、この指定の順は逆でもいいようです。

例えば、

```
> echo abc > f1.dat
```

(先頭の > はプロンプト) とすると画面には何も表示されず、「abc」(と最後に改行) だけが書かれたテキストファイル f1.dat が作られます。さらに、

```
> set /p s=リダイレクトテスト < f1.dat
```

とすれば、入力待ちにはならず直ちに set コマンドが終了し、環境変数 s に abc という文字列が代入されます。

出力リダイレクションには、もう 1 つ、追加出力の

>> [ファイル名]

という形式もあります。これは、指定したファイルが存在しない場合は > と同じでそのファイルが新規に作られるのですが、指定したファイルが既に存在する場合は、

- > の場合は、指定したファイルが消される (上書きされる)
- >> の場合は、指定したファイルの最後にコマンド出力が追加される

となります。

一方、パイプ は、コマンドの標準出力を、そのまま別のコマンドの標準入力として渡すときに使います。記号は | です。例えば、

```
> comA | comB
```

とすると、コマンド comA と comB の両方が起動され、comA の標準出力が、そのまま comB の標準入力に流されます。これと同じことはリダイレクションを用いて、

```
> comA > f1.dat
> comB < f1.dat
```

としても行えるのですが、パイプの方が 1 行で済みますし、中間ファイル f1.dat のようなものを必要としません。しかもパイプは多段階につなぐことも可能で、例えば

```
> comA | comB | comC
```

とすれば、コマンド comA,comB,comC が起動され、comA の標準出力が comB の標準入力として渡され、その comB の標準出力が comC の標準入力として渡されることになります。これとリダイレクションを組み合せることも可能で、例えば、

```
> comA < f1.dat | comB | comC > f2.dat
```

とすると、f1.dat に対する comA の処理の結果が comB に、その処理の結果が comC に渡され、その結果が f2.dat として保存されることになります。

標準入力からデータをもらって、標準出力にその処理結果を書き出すソフトを フィルタ と呼びますが、この例からもわかるように、パイプはフィルタを組み合わせてデータを加工するのに向いている仕組みです。一つ一つのフィルタが単一の機能しかなくても、複数のフィルタを組み合わせることで、流れ作業的にデータの複雑な加工処理が行えます。

MS–Windows に用意されている使えそうなフィルタには、

- sort: テキストファイルの行の並べかえ (ソーティング) を行う
- find: テキストファイルから指定したキーワードを含む行を抜き出す
- findstr: find の拡張版 (検索キーワードとして正規表現が使える)

などがあります。他にも、リダイレクトやパイプに関連の深いコマンドとして、

```
more, type, copy, echo
```

などがあり、これらについては 7 節で説明しますが、MS–Windows に標準的に用意されているフィルタはそれほど多くはありません。C 言語が使えるなら、必要なものを自作するのもいいでしょう。

次に、パイプやリダイレクションと関係が深い 2 つの特別な疑似ファイル (デバイスファイル) **con** と **nul** について説明します。これらは入力、出力の両方に使えますが、いずれも定義済みのファイルで、よって逆にこの名前のファイルを作ることはできません。これらの入力、出力は表 2 のようになっています。例えば、copy コマンドは

	入力	出力
con	標準入力	標準出力
nul	なし	なし

表 2: con と nul

```
> copy␣fileA␣fileB
```

でファイル fileA と同じ内容の fileB を作るコマンドですが、

```
> copy con fileB
```

とすると、コピー元のファイルが標準入力なのでキーボードからの入力待ちとなり、その後キーボードで入力したものが直接 fileB に保存されます。この場合入力を終了するには、`Ctrl-Z` を入力します。

一方、nul は入力も出力も何もしない疑似ファイルで、例えば 4.4 節で見たように、

```
> comA > nul
```

として、コマンド comA の標準出力を消すことができますし、

```
> copy nul fileA
```

とすれば、サイズ 0 のファイル fileA を作ることができます。

なお、コマンドの出力先は標準出力以外にもうひとつ、標準エラー出力 (stderr) と呼ばれるものがあり、ここへの出力は > ではリダイレクトすることができず、普通は常に画面に表示されます。コマンドのエラーメッセージなどはここに出力されることが多いので、それは `> nul` で消すことはできません。標準エラー出力をリダイレクトする場合は、2> を用います。例えば、type コマンドで fileA と fileB を連結して fileC を作る場合、

```
> type fileA fileB > fileC
```

とすると、type コマンドは fileA, fileB の名前を標準エラー出力に表示しますが、

```
> type fileA fileB > fileC 2> nul
```

とすれば、それも出なくなります。ただし、2 と > の間にスペースを開けてはいけません。

6 ワイルドカード

バッチファイルでは、ファイルやディレクトリを扱うことが多いと思いますが、そこで使用できるワイルドカードについて説明しておきます。

コマンドプロンプトでは、ファイル名やディレクトリの指定に ワイルドカード として次の 2 種類の特殊文字 (半角文字) を使うことができます[18]。

- *：0 文字以上の任意の文字列にマッチ (適合) する
- ?：空文字または 1 文字の任意の文字にマッチ (適合) する

例えば、カレントディレクトリに、

```
test.bat, test2.c, test2.exe, test3.bat, test4
```

というファイルがあったとすると、

- test?.bat：test.bat と test3.bat がマッチ
- test2.*：test2.c と test2.exe がマッチ
- test?.*：上の 5 つのファイルすべてがマッチ

することになります。なお、MS–Windows では「.」で始まる名前のファイルは作れるのですが、「.」で終わる名前のファイルを作ることはできず、例えば test4. という名前のファイルを作ってもそれは test4 という名前になるので、ファイル名としては

[18] よって、これらの文字を含む名前のファイルを作ることはできません。

test4 と test4. とは実質同じことになり、よって test4 は「test4.」として上の test?.* というパターンにマッチすることになります。

ワイルドカードは、主に複数のファイルを指定できるコマンド、例えば copy コマンド (詳しくは 7 節参照) で

```
> copy test?.* ..¥trash
```

のように使いますが、この場合は上の 5 つのファイルが ..¥trash というディレクトリにコピーされることになります[19]。

他のワイルドカードの使用例を示すと、

- *abc* : 名前に abc という文字列が含まれるファイル名にマッチ
- *.bat : すべてのバッチファイル名にマッチ
- test* : test で始まるすべてのファイルにマッチ
- * : すべてのファイル名にマッチ
- ?????.??? : [名前].[拡張子] の名前の部分が 5 文字以下、[拡張子] の部分が 3 文字以下のファイル名等にマッチ

のようになります。

7 バッチファイル向けのコマンド

本節では、MS–Windows に標準的に用意されている、コマンドプロンプト、バッチファイル向けのコマンドをざっと紹介します。なお、コマンド名の後ろに省略可能なオプションを持つコマンドも多いのですが、ここではよく使いそうなものだけを適宜紹介します。オプションは、原則的にコマンド名のすぐ後ろに指定し、排他的でないオプションは複数を並べて指定することもたいては可能です。詳しい説明は、

```
> help [コマンド名]
```

を参照してください。以下のコマンドのうちの多くは MS–DOS の頃からあったものですが、機能はかなり拡張されているものも多いです。

[19] .. は一つ上のディレクトリ、. はカレントディレクトリを意味します。

- echo – 画面への文字列出力
  - echo␣on：実行するコマンド自体も画面表示する (デフォルト)
  - echo␣off：コマンド自体は画面表示しない
  - echo␣[文字列]：文字列を標準出力に表示する
  - echo：現在の echo の状態 (ON か OFF か) を表示する

  この最後の形式は、他のコマンドの標準入力にパイプ経由で文字列を直接渡すのにも利用できますし、簡単なテキストファイルを作るのにも使えます。例えば、バッチファイル内で、

  ```
  echo @echo off> test1.bat
  echo echo これは子バッチです (%%0)>> test1.bat
  echo exit /b>> test1.bat
  call test1.bat
  ```

  のようにして >,>> で別のバッチファイルなどを作って実行することもできます。

  なお、引数を与えないか、あるいは空白だけを引数にして echo を実行すると、現在の echo の状態を表示しますが、空行を表示する (つまり改行だけ行う) のに echo を利用する場合は、「echo. 」のように echo の後ろに空白を入れずに . をつけて実行します。

- type – 画面にファイルの内容を表示
  - type␣[ファイル名]：ファイルの内容を標準出力に表示する
  - type␣[ファイル 1]␣[ファイル 2]␣...：複数の内容を連続して標準出力へ

  最後の形式は、複数のファイルを連結して他のコマンドにパイプで渡したり、簡単にファイルを連結するのに利用できます。例えば、

  ```
  > type file1 file2 > file3
  > copy file1+file2 file3
  ```

  はほぼ同じことになります (詳しくは copy の項参照) が、複数のファイルの連結結果をそのまま他のコマンドにパイプで渡すには type の方を使います[20]。

- more – 簡易ページャ
  - more␣[ファイル名]：ファイルの内容を 1 画面ずつ表示
  - more [ファイル 1] [ファイル 2] ...：複数のファイルを 1 画面ずつ表示
  - [コマンド]␣|␣more：コマンドの出力を 1 画面ずつ表示

[20] 5 節で説明したように con を使えば copy でも可能です。

more は、ページャと呼ばれるテキストファイル参照ソフトの一つで、space キーで次の画面を、enter キーで 1 行スクロールします。複数のファイルを指定しているときは f で次のファイルの表示に移ります。なお、前の画面、前の行、前のファイルに戻ることはできません。オプションには、例えば

- /s : 複数の空白行を 1 行の空白行にまとめて表示

があります。

- cls – 画面のクリア
  cls にはオプションはなく、コマンドプロンプト画面をクリアします。

- color – コマンドプロンプト画面の色指定

  - color␣[色指定] : コマンドプロンプト画面の色を変更する
  - color : コマンドプロンプト画面の色をデフォルトに戻す

  色指定は、2 桁の 16 進数で指定し、上の桁が背景色、下の桁が文字色 (前景色) を表します。色は、以下のように割り当てられています。

  - 0 ~ 7: 暗色の黒、青、緑、水色、赤、紫、黄、白
  - 8 ~ f: 明色の黒、青、緑、水色、赤、紫、黄、白

  ただし、7 と 8 は実際にはどちらも灰色ですが、7 (暗色の白) の方が 8 (明色の黒) よりも明るい灰色で、デフォルトは 07 (背景は黒、文字は暗色の白) のようです。例えば

  ```
  > color f2
  ```

  とすると、背景が白で文字が緑になります。

  なお、コマンドプロンプトの色の変更は、タイトルバーをクリックして現れるプロパティウィンドウの方が細かく設定できます。

- copy – ファイルのコピー

  - copy␣[パス指定]␣[ファイル B] :
    パス指定で指定したファイルをファイル B という名前でコピーする
  - copy␣[パス指定]␣[ディレクトリ] :
    パス指定で指定したファイルをディレクトリ内にそのままの名前でコピー
  - copy␣[パス指定] :
    上のディレクトリとしてカレントディレクトリを指定したことと同じ

  基本的には、1 つ目に指定したファイルを 2 つ目に指定したファイルやディレクトリにコピーするコマンドです。1 つ目のオプションのパス指定には、次の 3 つの形式があって、それぞれ意味が異なります。

- [ファイル A] : ファイル A をコピーする。
- [ワイルドカードを含むファイル指定] :
  コピー先がファイル (ファイル B) の場合は、ワイルドカードにマッチしたファイルを連結してファイル B とする。コピー先がディレクトリの場合は、そのディレクトリ内にそれらのファイルをそのままの名前でコピーする。
- [ファイル 1]+[ファイル 2]+...+[ファイル n] :
  ファイル 1 からファイル n までを連結してコピーするが、コピー先がディレクトリの場合は、そのディレクトリ内にファイル 1 の名前のファイルを作成する。

ワイルドカードを使うことで複数のファイルも一度にコピーできます。

- ren – ファイル名の変更 (rename も同じ)

  ren␣[ファイル A]␣[ファイル B]

  でファイル A の名前をファイル B に変更します。ファイル A にはパス指定もできますが、ファイル B には名前だけを指定します。ディレクトリの名前の変更にも使えますし、制限はありますが、ワイルドカードを使って複数の名前を一度に変更することも可能です。例えば、

  ```
  > ren test*.bat *.t
  ```

  とすると、test*.bat にマッチする複数のファイル名の拡張子を一度に .t に変更できます。

- del – ファイルの削除

  del␣[パス]

  [パス] 部分はファイル、またはディレクトリで、複数指定することもできます。ファイルの場合はそのファイルを、ディレクトリの場合は、そのディレクトリ内のファイルをすべて消去します。ただしディレクトリの削除はできません。

  指定できるオプションには

  - /s : 指定パスにあるサブディレクトリ内のファイルもすべて削除
  - /p : 削除前に確認メッセージを出す
  - /q : ディレクトリ指定の際の確認メッセージなしに実行

  などがあります。

- move – ファイルの移動
  - move␣[ファイル A]␣[ファイル B] :
    ファイル A をファイル B に移動 (ファイル A は削除される)

- move␣[ファイル A]␣[ディレクトリ B] :
  ファイル A (ワイルドカード指定も可) をディレクトリ B 内にそのままの名前で移動
- move␣[ディレクトリ A]␣[ディレクトリ B] :
  ディレクトリ A をディレクトリ B 内に移動

move は copy と del を合わせたようなもので、コピーした後で元のファイルを削除します。ただし copy とは違い、ファイルの連結はできません。

- dir – ディレクトリの一覧の表示
  - dir␣[パス] : パスの内容を表示

  パスがファイル (ワイルドカードがついてもよい) ならば指定ファイルを表示し、ディレクトリならばディレクトリ内の一覧を表示します。パスは複数指定することもできます。

  指定できるオプションには、

  - /w : 名前のみをワイド形式で表示
  - /l : 大文字を小文字にして表示
  - /s : サブディレクトリ内も表示
  - /o:[キー] : 指定キーでソートして表示

  などがあります。最後のソートのキー指定は、

  n=名前順、s=サイズ順、e=拡張子名順、d=日付順、
  g=ディレクトリ、ファイルの順

  となっていて、それぞれデフォルトでは昇順ですが、- を頭につけることでいずれも降順になります。

- md/rd/cd – ディレクトリ操作
  - md␣[ディレクトリ] : ディレクトリの作成 (mkdir も可)
  - rd␣[ディレクトリ] : ディレクトリの削除 (rmdir も可)
  - cd␣[ディレクトリ] : カレントディレクトリの移動 (chdir も可)

  rd は、デフォルトでは指定ディレクトリ内にファイルやディレクトリが残っていると削除できませんが、以下のオプションをつければ削除できます。

  - /s : ディレクトリ内のファイルやディレクトリも削除
  - /q : /s 時の確認メッセージを出さない

また、cd はデフォルトではカレントドライブの移動はしませんが、以下のオプションでそれも可能になります。

- /d：ドライブ名も指定した場合カレントドライブも変更

- pushd/popd – ディレクトリスタック操作
  - pushd␣[パス]：
    カレントディレクトリをディレクトリスタックに積んで指定パスに移動
  - popd：最後に積まれたディレクトリスタックのパスに移動

これは、一時的に作業ディレクトリを変えて、その後元のディレクトリに戻りたい場合に利用できます。なお、これは cd とは違い別ドライブにもオプション無しで移動できます。

- date/time – 日付/時刻の表示/設定
  - date：日付の対話的な設定 (管理者権限が必要)
  - time：時刻の対話的な設定 (管理者権限が必要)
  - date␣[日付]：日付の設定 (管理者権限が必要)
  - time␣[時刻]：時刻の設定 (管理者権限が必要)
  - date␣/t：現在の日付の表示
  - time␣/t：現在の時刻の表示

日付や時刻の設定を行う場合は通常、日付の指定は YY-MM-DD、時刻の指定は HH:MM:SS という形式で指定します。

- sort – ファイルの整列 (ソート)
  - sort␣[ファイル]：
    テキストファイルの内容を整列 (ソート) して標準出力に出力

ファイルを指定しなかった場合は、標準入力をソートしますが、ファイルを指定する場合は、1 つだけしか指定できません。ソートは、文字列の辞書式順で行われます。オプションは、以下のものが利用できます。

- /r：降順にソート (デフォルトは昇順)
- /+[数値]：各行の指定番目の文字以降をキーとしてソート
- /o[ファイル名]：出力結果をファイルに保存

- find – 指定文字列を含む行の抽出
  - find␣"[文字列]"␣[ファイル]：
    ファイルから指定文字列を含む行を標準出力へ出力

検索する文字列は二重引用符で囲む必要があります。ファイルは複数指定でき、またオプションには以下のものが指定できます。

- /v：指定した文字列を含まない行を出力
- /c：マッチした行数のみを出力
- /n：行番号も表示
- /i：大文字、小文字の区別をせずに検索

- findstr – find の拡張版 (正規表現が使える)
  詳しくは help findstr 参照。

- fc – テキストファイルの比較
  - fc␣[ファイル 1]␣[ファイル 2]：
    ファイル 1 とファイル 2 の違いを標準出力に出力

  オプションは以下のものが指定できます。

  - /c：大文字、小文字の区別をしない
  - /n：行番号も表示
  - /a：違う範囲の先頭と最後の行だけをリポートする
  - /w：複数の空白 (タブとスペース) を一つのスペースとして比較
  - /t：タブはタブのまま比較

- start – 別ウィンドウでのプログラムの起動
  - start␣[コマンド]：別ウィンドウでコマンドを実行
  - start␣[ファイル]：ファイルに関連づけられたアプリケーションを起動
  - start␣[ディレクトリ]：エクスプローラを起動してディレクトリを開く
  - start␣[URL]：既定のブラウザで URL を開く
  - start␣mailto:[アドレス]：既定のメーラを起動

  start は、コマンドプロンプト用 (非ウィンドウ系) のコマンドに対しては、別のコマンドプロンプトウィンドウを開いてそこで指定コマンドを実行し、ウィンドウ用のプログラムを指定した場合は、そのウィンドウプログラムを起動します。

  この方法の場合は、デフォルトでは別のウィンドウのコマンドが立ち上がった時点で (そのコマンドが終了しなくても) 元のコマンドプロンプトは次の処理の入力待ちになり、そちらで次の作業に移ることができます。

  ファイルを指定すると、それに関連づけられたアプリケーションを開くことができますので、バッチファイル内から MS–Word や MS–Excel を起動することも

できます。さらに、URL を指定してブラウザやメーラを立ち上げることもできます。

オプションは例えば以下のものが指定できます。

- "[タイトル]" : ウィンドウタイトル (オプションの最初に指定)
- /min : ウィンドウを最小化して起動
- /max : ウィンドウを最大化して起動
- /wait : アプリケーションが終了するまで待機する

- shift – バッチファイルオプションのシフト
  - shift : %[n] をそれぞれ %[n-1] にシフトする
  - shift␣/[番号] : 指定番号以降のみシフトする

例えば、

%0=test.bat, %1=123, %2=竹の, %3=茂治

のときに shift コマンドをバッチファイル内で実行すると

%0=123, %1=竹の, %2=茂治

となりますが、shift /2 とすると %2 以降のみシフトされ、

%0=test.bat, %1=123, %2=茂治

となります。なお、%* の値は shift では変化しません。

8 注意

最後に、バッチファイルに関する注意を上げていきます。ただし、以下の記述は私が個人的に調べた上での結果なので、間違っている可能性もあります。そのような箇所に気がついたら、是非教えてください。

1. コマンドプロンプト上で日本語を入力したいときは、[Alt]-[半角/全角] で行います。日本語入力を終了するときも [Alt]-[半角/全角] です。
2. echo コマンドに off や on (大文字でも) を指定した場合は、それは表示する文字列ではなく特別なオプションと見なされるわけですが、「off」「on」のみを表示させるのに echo を利用する場合は、7 節の echo の項で説明した空行の表示のやり方のように、「echo.off」や「echo.on」のように . を echo の直後につけるといいようです。

3. バッチファイル中で echo と出力リダイレクト (>, >>) で簡単なファイルの書き出しを行うとき (例えば 7 節の echo の項の例参照)、最後に改行を入れないようにもできるといいのですが、MS–Windows の echo にはそのようなオプションはないようで[21]、とりあえず、MS-Windows に用意されている標準的なコマンドでは、それはできないようです。

   よって、そういうことを行えるフリーソフトを利用するか、またはそういうコマンドを自作するとかしないと、とりあえずは無理なようです。

   また、[7] で見つけましたが、if 文や for 文で使われる ( ) は複数のコマンドをブロック化するために単独で使うこともできるようです。これによりバッチファイルから簡単なファイルの書き出しを行う 7 節の echo の例は、以下のように >> なしで書くこともできます。

```
(
    echo @echo off
    echo echo これは子バッチです [%%0]
    echo exit /b
) > test1.bat
call test1.bat
```

   こちらの方が多少見やすいですが、この方法では ( ) 内部の echo の文中に ( ) を使うことができません (よって 7 節の echo の例の ( ) をここでは [ ] に変えてあります)。

4. set コマンドによる環境変数の設定で、= の前後にスペースを入れると、それが変数名や値にも入ってしまいます。例えば、

   - set␣a=3 ⇒ %a% が「3」
   - set␣a=␣3 ⇒ %a% が「␣3」
   - set␣a␣=3 ⇒ %a␣% が「3」
   - set␣a␣=␣3 ⇒ %a␣% が「␣3」
   - set␣a␣=␣3␣ ⇒ %a␣% が「␣3␣」

   のようになってしまいます[22]。

5. %[変数]% は、その環境変数が定義されていればもちろんその値になりますが、定義されていないときは、コマンドプロンプトとバッチファイルではその扱われ方が違うようで、例えば環境変数 a が定義されていない場合、%a% は以下のようになります。

   - コマンドプロンプト: %a% は「%a%」という 3 文字の文字列

[21] UNIX の echo コマンドでは -n オプションで改行を入れないようにできます。
[22] よって、バッチファイルを編集する場合は、行末が表示されるエディタを利用するといいでしょう。

- バッチファイル: %a% は空文字列

なお、%1, %2 等のバッチファイルの引数として使われるものは、未定義のものは常に空文字列となります。

6. set コマンドによって設定した環境変数は、バッチファイルの終了後もそのバッチファイルを実行したコマンドプロンプト上に残ります。よって、そのバッチファイル内だけで使用する一時的な変数は、最後に

```
set␣[変数名]=
```

で削除するのを忘れないでください[23]。

逆に、バッチファイル実行後も環境変数の設定が残るという性質を利用して、そのコマンドプロンプトの環境変数 (特に PATH 環境変数) を設定するためのバッチファイルを使うこともできます。

7. コマンドやパラメータは大文字、小文字の区別はされませんが、ラベル名や環境変数名も大文字、小文字の区別はありません。ただし、for 文の局所変数だけは、大文字と小文字が区別されますので注意してください。よって、バッチファイルで例えば

```
for %%a in (1 2 3) do echo [%%a][%%A]
```

とすると、

```
[1][%A]
[2][%A]
[3][%A]
```

のように表示されます。

8. if で条件を書く場合、例えば

```
if %1==2 [コマンド]
```

のような条件を書くと、バッチファイルのコマンドラインオプションの %1 が与えられなかった場合は

```
if ==2 [コマンド]
```

と展開されてしまうので、構文エラーとなってしまいます。よって、if 文の条件の両辺が空文字列となりうる場合は、

```
if "%1"=="2" [コマンド]
```

のように引用符で囲むといいでしょう。

[23] 変数を局所化するための **setlocal/endlocal** という命令も用意されています。詳しくは `help setlocal` を参照してください。

9. if 文に else 部分をつけるときは、4.1 節の 2., 4. のように書く必要があり、例えば C 言語のように次のように書いてはいけません。

   - コマンドが 1 行の場合
     ```
     if [条件] [コマンド]
     else [コマンド]
     ```
   - コマンドが複数行の場合
     ```
     if [条件] (
         [コマンド]
         ..........
     )
     else (
         [コマンド]
         ..........
     )
     ```

   else は、if 文の終わりの行につながっている必要があり、else で始まる行があるとエラーになります。

   ただし、if の実行部が 1 行で、else の実行部が複数行 (あるいはその逆) であるときは、以下のように書くことはできます。

   ```
   if [条件] [コマンド] else (
       [コマンド]
       ..........
   )
   ```

10. if の後ろに else も続けて 1 行で書く場合、if 部分のコマンドが 1 単語でない場合は、それを ( ) で囲む必要があります。そうしないとその後ろの else が認識されません。例えば、

    ```
    if "%a%"=="3" echo 「3 です。」 else echo 「3 以外です。」
    ```

    とすると、最初の echo から else も含んで行末までが if 部分の実行コマンドと認識されてしまいますので、

    ```
    if "%a%"=="3" (echo 「3 です。」) else echo 「3 以外です。」
    ```

    のように書く必要があります。

11. C 言語の else if

    ```
    if ([条件 1]) {
        [ブロック 1]
    } else if ([条件 2]) {
        [ブロック 2]
    ```

```
} else {
    [ブロック 3]
}
```

は、バッチファイルでも同様に

```
if [条件 1] (
    [ブロック 1]
) else if [条件 2] (
    [ブロック 2]
) else (
    [ブロック 3]
)
```

のように書くことができますし、入れ子にして

```
if [条件 1] (
    [ブロック 1]
) else (
    if [条件 2] (
        [ブロック 2]
    ) else (
        [ブロック 3]
    )
)
```

のようにすることも可能です。

12. バッチファイルには C 言語の switch 文はありませんが、それは if 文と goto 命令を組み合わせれば容易に実現できます。 同様に、C 言語の while 文もありませんが、これも if 文と goto 命令で代用できます。ただし、goto の使い過ぎは読みにくいプログラムの元となりますので注意が必要です[24]。

13. バッチファイル内では、バッチファイルのコマンドライン引数は %1 から %9 で利用できますが、%10 と書くとそれは「%1 に文字 0 を追加した文字列」となってしまいますので、10 番目のオプションを直接は利用できません。

    しかし、オプションは 9 つまでしか認識できないわけではなく、shift コマンドを利用すればそれより多くのオプションも使うことはできます。例えば、1 度の shift コマンドにより 10 番目の引数が %9 に、2 度目の shift コマンドにより 11 番目の引数が %9 になります。

    なお、バッチファイルのオプションの区切りにはスペースだけでなく、カンマも使えることに注意してください。逆に、スペースやカンマを含む文字列をオプ

[24] goto によってあっちこっちに飛んでいて読みにくいプログラムを スパゲッティプログラム と呼ぶことがあります。

ションとして指定したい場合は、その部分を二重引用符 " " で囲む必要があります。ただし、その場合二重引用符付きの文字列となりますが、例えば %1 からその引用符を取り除いた文字列を使いたい場合は、%~1 とします。詳しくは、help call 参照。

14. バッチファイルに与えられたオプションの個数を、C 言語の argc のように最初に取得したければ、例えば次のように for 文を利用すればいいでしょう。

```
set argc=0
for %%a in (%0 %*) do set /a argc+=1
```

ちなみに上の方法だと、C 言語同様、argc にはバッチファイル名も入れたコマンドラインオプションの個数が入ります。

15. %0 はバッチファイル名を表しますが、これはバッチファイルのヘルプメッセージを作るときに便利です。例えば、オプションとしてファイル名を 1 つ取るバッチファイルを書いたときに、

```
@echo off
if not "%1"=="" goto run1
rem ##### 以下はヘルプメッセージ #####
:help
echo 使用法:  %0 [ファイル名]
echo      [ファイル名] のファイルの行をランダムに 1 行表示
exit /b 1
rem ##### 以下からが main #####
:run1
.....
```

のように、説明部分のバッチファイル名に当たるところを %0 を使って書けば、後でバッチファイルの名前を変更してもこの help 部分を変更する必要はなくなります[25]。

16. copy コマンドは、ワイルドカードを使えば複数のファイルを一度にコピーすることもできますが、例えば、カレントディレクトリに

```
test1, test2, test3, test4
```

のような 4 つのファイルがあって、このうち test1 と test3 だけをワイルドカードで選ぶことはできませんので、その 2 つだけを別なディレクトリにコピーする場合は、

```
> copy test1 dir
> copy test3 dir
```

[25] C 言語の main() の引数の argv[0] も同じような目的で使われることが多いです。

のようにするしかありません。また copy コマンドは、例えば

```
> copy test1 .
```

のように自分自身のファイルにコピーしようとするとエラーメッセージを出して止まるのですが、+ の形式を使った場合はそうではなく、

```
> copy test1+test3 .
```

あるいは

```
> copy test1+test3
```

とすると、test1 と test3 を連結した結果を test1 として上書きします。

17. MS–DOS の頃、つまりファイル名が

    [名前 (8 文字以下)].[拡張子 (3 文字以下)]

    と制限されていたころは、すべてのファイル名にマッチさせるワイルドカードは *.* と書いてしましたが、今は * だけですべてのファイル名にマッチします。ちなみに MS–DOS の頃は、* は「拡張子がないすべてのファイル」を意味していました。

    なお、今でも *.* はすべてのファイル名にマッチするので、現在のコマンドプロンプトに関する本でもすべてのファイル名にマッチするパターンとして *.* を紹介しているものは少なくないようです。

    逆に、MS–DOS の頃の「拡張子がないすべてのファイル」を意味する * に相当するワイルドカードパターンは、現在は存在しません。

18. コマンドの出力を環境変数に代入するには、次のような方法があります。

    ```
    > date /t > tmpf
    > set /p s= < tmpf
    ```

    こうすると date /t の出力が tmpf というファイルに保存され、それが set /p にリダイレクトされて環境変数 s に代入されます。

    しかし、これを中間ファイルなしにパイプで以下のようにやってもうまくいきません。

    ```
    > date /t | set /p s=
    ```

    理由は正確にはわかりませんが、set が cmd.exe の内部コマンドなので、パイプ渡しができないのではないかと想像しています。

    なお、日付はこのようにしなくても、動的な環境変数 %date% で参照できます (3 節参照)。

19. set /a の右辺では通常の環境変数の値は % なしで参照できますが、3 節の最後に紹介した %random% 等の動的な環境変数は、set /a の右辺でも % が必要です。例えば、

```
> set /a a=random
> set /a b=%random%
```

とすると、a には 0 が、b には乱数値が入ります。前者は、多分「静的」な環境変数 random を探してそれがないために 0 が代入されているのではないかと思います。

20. バッチファイルのオプションの %[n] がファイルやディレクトリ名を指している場合、そこからドライブ名、パス部分、拡張子部分などを取りだすことができます。例えば %3 がファイル名 (例えば D:¥hoge というディレクトリにある "file1.txt") である場合、

- %~3 = 引用符を削除 (file1.txt)
- %~f3 = フルパス名 (D:¥hoge¥file1.txt)
- %~d3 = ドライブ名のみ (D:)
- %~p3 = パス部分のみ (D:¥hoge¥)
- %~n3 = ファイル名部分のみ (file1)
- %~x3 = 拡張子部分のみ (.txt)
- %~t3 = ファイル時刻 (2010/07/01 18:25)
- %~z3 = ファイルサイズ (1625)

これらは複数組み合わせて %~dx3 (上の例では D:.txt) のように使用することもできますし、for 文の局所変数でも同じ仕組みが利用できます。詳しくは、help call, help for 参照。

21. help set を見るとわかりますが、環境変数から部分文字列を取得したり、文字列の簡単な置換も行えます。

- %var% = 環境変数 var の文字列全体
- %var:~[m]% = $[m]$ 文字目から最後までの部分文字列
- %var:~[m],[n]% = $[m]$ 文字目から $[n]$ 文字
- %var:[s1]=[s2]% = 文字列 [s1] をすべて [s2] に置換した文字列

$[m]$ は、先頭の文字列を「0 番目」と数えます。$[m]$ として負の値を指定すると、文字列の先頭からではなく後ろから数えます (一番後ろの文字が (-1) 番目)。また、$[n]$ として負の値を指定すると、最後の $[n]$ 文字を削除したもの、となります。例えば、環境変数 var が「1234567890123」という文字列の場合、

- %var:~5% =「67890123」
- %var:~5,2% =「67」
- %var:~5,-2% =「678901」
- %var:~-5% =「90123」
- %var:~-5,2% =「90」
- %var:~-5,-2% =「901」
- %var:123=aBc% =「aBc4567890aBc」
- %var:123=% =「4567890」

となります。詳しくは、help set 参照。

22. for 文のブロック内で局所変数以外の環境変数を使用する場合、それはデフォルトでは「for 文が実行される前」に展開されてしまうことになっているので、期待通りの結果が得られない場合があります。例えばコマンドプロンプト上で

```
> set j=0
> for %a in (1 2 3) do set /a j+=%a
> echo %j%
```

とすると、期待通り 1+2+3 の結果の 6 が表示されるのですが、

```
> set s=
> for %a in (竹 の 茂治) do set s=%s%%a
> echo %s%
```

は、「竹の茂治」とは表示されず、「%s%茂治」と表示されてしまいます。%a を %%a としてバッチファイルで行えば「茂治」だけが表示されます。

これは、for 文の局所変数である %a はリストの要素に順に展開されるのに対し、環境変数の値である %s% は、この for 文が実行される前に展開されてしまうからで、その時点では環境変数 s は最初の set s= によって未定義となっていますから、上の 5. で説明した通り、コマンドプロンプトでは「%s%」という 3 文字の文字列に展開され、よって for 文が実行されるときは既に展開されてしまった文字列 %s% に対して順に

```
set s=%s%竹
set s=%s%の
set s=%s%茂治
```

が実行されてしまうことになり、よって最後の set コマンドにより s の値は「%s%茂治」となるわけです。

この現象は、上のうまくいく例の方の for 文の set コマンドを %j% を使って

```
> set j=0
> for %a in (1 2 3) do set /a j=%j%+%a
> echo %j%
```

と変えても起こり、これも for 文内の %j% が for 文が実行される前に for 文の直前の値である 0 に展開されるので、for 文では実際には

```
set /a j=0+1
set /a j=0+2
set /a j=0+3
```

が実行されてしまい、結果として 6 ではなく 3 と表示されてしまいます。

これを防いで、for 文内でも環境変数の値を逐次展開される変数として利用するには、遅延展開 という機能を利用します。

デフォルトのコマンドプロンプト (cmd.exe) では、遅延展開機能が使えるようにはなっていません (off) から、それを on にする必要があります。遅延展開機能を on にするには、

- 現在のコマンドプロンプト上で
  ```
  > cmd␣/v:on
  ```
  として、遅延展開機能が有効なコマンドインタプリタを二重に起動する
- start コマンドを使って、遅延展開機能が有効なコマンドインタプリタの立ち上がった別ウィンドウのコマンドプロンプトを利用する
  ```
  > start␣"遅延展開可"␣cmd␣/v:on
  ```
  (最初のオプション文字列はウィンドウのタイトル)
- 「cmd /v:on」を実行するようなショートカットを作成して、コマンドプロンプトを開くときにそれを利用する
- 遅延展開機能を必要とするバッチファイル (例えば test.bat とする) を起動するときに、
  ```
  > cmd␣/v:on␣/c␣test.bat
  ```
  のようにして、そのバッチファイルの実行のみ遅延展開機能を on にする
- 「setlocal enabledelayedexpansion」を実行しておく (特別な局所変数の設定)

などの方法があります。

そして、for 文で逐次展開したい環境変数の値は、%[変数]% ではなく、![変数]! のように ! で囲みます。例えば

```
> set j=0
> set s=
> for %a in (1 2 3) do (
```

```
>     set /a j=!j!+%a
>     if defined s (set s=!s!%a) else (set s=%a)
>     rem バッチファイルなら set s=!s!%%a でよい
> )
> echo [j=%j%][s=%s%]
```

のようにすれば正しく [j=6][s=123] と表示されます。なお、バッチファイル中では、上の 5. で説明した通り s が未定義の場合は !s! は空文字になるので、if defined による場合分けは不要です。

9 最後に

MS–DOS の頃と違って、現在のコマンドプロンプトのバッチファイルでは整数変数が使え、その演算も行えるようになり、さらに乱数も使えるようになっているので、可能性がだいぶ広がりました。

しかし、まだ Unix のシェルスクリプトにはかなり劣る点があります。

- 「引用符」がちゃんと使える形にはなっていないので、文字列変数、コマンドのパラメータ等に関する制限がある。
- while 文、switch 文がないので、if, goto に頼りがちになり、スパゲッティ化しやすい。
- フィルタやバッチファイル用に用意されているコマンドが少ない。
- ヒアドキュメントが書けない。
- 長いコマンドを複数行に分けて書けない。

よって、本当に実用的なものを作るには、バッチファイルと AWK やフリーソフトなどを組み合わせる必要があると思います。AWK 1 つで、Unix にあるような基本的なフィルタやコマンド (wc,sed,uniq,grep,printf 等) はほぼ代行ができます。

ただ、バッチファイルで AWK を使う場合、AWK スクリプトをバッチファイルと別に用意すると保守が面倒になりますが、直接 AWK のコードをバッチファイルに書くには、「引用符」と「複数行の記述」の問題があるためうまくいかない場合もありますし、「ヒアドキュメント」が使えないのでバッチファイルからスクリプトを吐きだすのも簡単ではないですが、echo で 1 行ずつ吐きださせることならできなくはないです。

次回はそのような話でも書いてみようかと思います。

参考文献

[1] 五十嵐貴之「Windows 自動処理のための WSH プログラミングガイド」ソシム (2009)

[2] 飯島弘文「Windows コマンドプロンプト スパテク 242 Vista/XP/2000 対応」翔泳社 (2008)

[3] 村上俊一「ウィンドウズの仕組みがわかると『MS–DOS/コマンドプロンプト』に強くなる」メディア・テック出版 (2004)

[4] 藤田英時「コマンドまたはファイル名が違います」ナツメ社 (1992)

[5] 田中一郎「早わかり MS–DOS Ver.3.3 実用マニュアル」新星出版社 (1991)

[6] ウィキペディア http://ja.wikipedia.org/wiki/

[7] xyzzy_user「BAT 上でのヒアドキュメントの 代替 (楽天ブログ)」 http://plaza.rakuten.co.jp/u703331/diary/200907230000/ (2009 07/23)