情報通信コース実験 II
スクリプトプログラミング(担当 竹野、2020 年度)

(http://takeno.iee.niit.ac.jp/%7Eshige/math/lecture/jikken1/jikken1.html)

第 2 回 バッチファイル、AWK

1 環境変数値の整数計算

環境変数値は基本的には文字列だが、set コマンドの /a オプションにより、整数値の簡単な計算を行うことができる。

```
> set /a [変数名]=[数式文字列]
```

これは、右辺の数式の計算結果の数値 (整数値) を文字列としたものを左辺の環境変数に代入する。右辺の数式には、整数値、環境変数名 (この場合は % で囲む必要はない)、かっこ (丸かっこのみ) 以外に、C 言語と同様の表 1 の演算記号が使える (この他にビット演算なども行えるが省略する)。

単項演算子		
記号	意味	例など
!	!a = a が 0 以外なら 0、a が 0 なら 1	!11=0
-	-a = $-a$ (符号の反転)	
四則演算		
記号	意味	例など
*	a*b = a と b の積	14*5 (=70)
/	a/b = a を b で割った商	14/5 (=2)
%	a%b = a を b で割った余り	14%5 (=4)
+	a+b = a と b の和	14+5 (=19)
-	a-b = a と b の差	14-5 (=9)

表 1: set /a の右辺で使える演算記号

整数値は通常は 10 進数であるが、先頭が 0x の場合は 16 進数、先頭が 0 の場合は 8 進数とみなされる。

また、表 1 の他に、set /a の後ろの代入を意味する等号 (=) の代わりに、以下の演算つき代入記号も利用できる:

*=, /=, +=, -=, %=

これらの意味も C 言語と同様であり、例えば「set /a x/=3」は「set /a x=x/3」と同じ意味になる。例:

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

というバッチファイルは、4 に、$3 \times x$ (=$3 \times 10 = 30$) を 7 で割った余り (=2) を加えた 6 を表示する。

コマンドプロンプト上で set /a を使用すると、代入だけでなく右辺の値も表示されるので、例えば

```
> set /a x=0x3f * 2
```

と実行すると、右辺の 16 進数 (0x3f×2 $= 63 \times 2 = 126$) を 10 進数に変換した「126」が表示される。同様に、

```
> set /a x=0342 * 2
```

のようにして 8 進数から 10 進数への変換もできる。

注意:

- 上の例のように、set /a の右辺の数式には適宜スペースを入れても構わない。
- 「set /a」の右辺で環境変数値を使う場合は、その変数名を % で囲む必要はない (囲んでも構わない)。ただし、%random% などの動的環境変数は set /a の右辺でも % で囲む必要がある。
- /a をつけない set では、右辺の数式がそのまま文字列として変数に設定される。逆に、右辺が数式ではない文字列に対して set /a とすると、計算に失敗し、変数には 0 などの適当な整数値が代入されてしまう。

%random% と set /a を組み合わせると、色々な範囲の乱数を利用することができる。例えば、

```
> set /a r=%random% % 4 + 9
```

とすると、r にはほぼ均等の割合で 9 以上 12 以下のランダムな整数が代入される。それは、

- 整数を 4 で割った余り = 0 以上 3 以下の整数
- 「0,1,2,3,4,5,6,7,8,9,. . .」を 4 で割った余り=「0,1,2,3,0,1,2,3,0,1,. . .」
  (0,1,2,3 の繰り返しがほぼ均等の割合で現れる)

なので、

- %random% % 4 = ほぼ均等な確率の 0 以上 3 以下のランダムな整数
- %random% % 4 + 9 = ほぼ均等な確率の 9 以上 12 以下のランダムな整数

が得られることになる。

注意:

- 余り計算の % はバッチファイル内では %% と重ねて書く必要がある (上の例はバッチファイルではなくコマンドプロンプト)。しかし、変数を囲む % の方は重ねてはいけない。例:

  ```
  @echo off
  set /a x=%random% %% 6 + 1
  echo %x%
  ```

  これは 1 から 6 までの整数をランダムに (ほぼ均等に) 出力する。

なお、if での ( ) 内の変数値は実行前にすべて置換されてから実行されるので、( ) 内で set /a で計算したり、表示させたりする場合は注意が必要。例えばバッチファイル内で、変数 n が偶数ならその旨と $n/2$ の値を表示させ、n が奇数なら何も表示しない、という処理は、

```
set /a m=n %% 2
if %m%==0 ( set /a x=n / 2
    echo n は偶数、%x% )
```

とすると、x の計算前に %x% が置換されてしまうのでうまくいかない。よって、例えば %x% を ( ) の外に出すために if の実行文を 2 つに分けて、

```
set /a m=n %% 2
if %m%==0 set /a x=n / 2
if %m%==0 echo n は偶数、%x%
```

とする方法がある。 または、次節で説明する goto を利用する方法もある。

2 goto 命令とループ

goto 命令は、実行の流れをバッチファイル内の任意の箇所へジャンプさせる (表 2)。ラベル行は、「:[ラベル名]」の形の行で、バッチファイル内の任意の位置に置くことができる。

命令形式	意味
goto [ラベル名]	指定ラベル行へ処理をジャンプ

表 2: goto 命令

例えば、C 言語の「if (x==1 || y==2)」に相当するものは、バッチファイルでは goto で以下のように書くことができる。

```
if %x%==1 goto runTrue
if %y%==2 goto runTrue
echo 偽
goto nextf
:runTrue
echo 真
:nextf
```

goto 命令と if 文を組み合わせることでループも作れる。例えば

```
set j=1
:start1
if %j% gtr 9 goto last1
set /a x+=j
set /a y+=j*j
set /a j+=2
goto start1
:last1
```

は、「set /a x+=j」と「set /a y+=j*j」までの部分を 1,3,5,7,9 の j の値に対して行うループになっていて、C 言語での「for(j=1; j<=9; j+=2)」のループに相当し、x には $1+3+5+7+9$, y には $1^2+3^2+5^2+7^2+9^2$ が追加される。

なお、この goto 命令を使うと簡単に無限ループが作れるが、無限ループとなったバッチファイルの実行は、コマンドプロンプトでは Ctrl-C で終了できる。

3 AWK

AWK (おーく) は、スクリプト言語と呼ばれるインタプリタ型プログラミング言語で、次のような特徴を持っている。

- プログラムの文法、書式は C 言語に非常に近い。
- テキストデータに対する行単位のフィルタとして動作するが、`BEGIN` ブロックを使えば C ライクな簡易インタプリタとしても使える。
- 正規表現や連想配列、文字列のフィールド分割、文字列処理関数など、C 言語にはない機能もある。
- 関数の自作も可能で、サブルーチンのライブラリ化もある意味で可能。

本実習では、以下で公開されている MS–Windows 用の GNU 版 AWK (コマンド名 `gawk`) を使用する (USB メモリにはインストール済み)。なお、必要なのはそこに含まれる `gawk.exe` という小さい実行ファイル一つだけであり、それをコピーすればどこでも利用できる。

- マルチバイト対応版の GNU AWK 3.1.5 (Win32 バイナリ)
  `http://www.vector.co.jp/soft/dl/win95/util/se376460.html`

AWK プログラム (**スクリプト** と呼ぶ) は、「`test1.awk`」のような「`.awk`」という拡張子を持つ名前のファイルに保存する。otbedit で AWK スクリプトを編集する場合は、「`.awk`」の拡張子のファイルを読み込むか、または画面左上にある AWK というアイコンで AWK モードにする。

4 AWK の BEGIN ブロックのみでの動作

AWK スクリプトの行フィルタとしての使い方は次回紹介することとし、しばらくは簡易インタプリタとしての以下のような構造のスクリプトを使用する。

```
BEGIN {
    [プログラム本体]
}
```

`gawk` にこの形式のスクリプトを処理させると、[プログラム本体] 部分を実行し、それが終了すると `gawk` も終了する。

このスクリプトを実行させるには、そのスクリプトの名前が例えば `test1.awk` であれば、コマンドプロンプトで

```
> gawk -f test1.awk
```

とする。`gawk` の `-f` オプションは、その後ろに `gawk` に処理させる AWK スクリプトを指定する。(C 言語とは違い、コンパイルは必要ない。)

また、gawk のバージョンや copyright は

```
> gawk --version
```

で表示できるが、`gawk` は MS–Windows の標準コマンドではないため、`help gawk` ではヘルプは表示できない。

インタプリタ形式の AWK スクリプトの簡単な例:

```
BEGIN {
    x = 3; y += 2
    printf "%d¥n", x + y
    for (j=1; j<=5; j+=2) z += j
    printf "%3d¥n", z
    if (x+y < z) s = "x + y < z"
    else s = "x + y >= z"
    printf "%s¥n", s
}
```

このスクリプトからわかるように、AWK スクリプトの文法は C 言語と非常に近いが違うところもある。基本的な文法について説明する。

1. 変数
    - 変数宣言は不要。
    - 変数値は数値か文字列で、数値はすべて実数 (整数型はない) で、文字列は `" "` で囲む。’ ’ で囲む文字型はない。
    - 数値を表わす文字列は、文字列としても、数値としても使える。例えば「`s = "12345"; x = s + 5`」とすると `x` には数値の 12350 が代入される。
    - 明示的に初期化されていない変数は、数値としては 0 が、文字列としては空文字列が初期値となる。
    - 配列は、C 言語と同様の形式で使えるが、添字は 0 から始める必要はない。配列の値も実数でも文字列でもよい。

2. 書式

   - # から行末までは無視されるのでコメントとして使える。逆に、/* */ や // はコメントにはならない。
   - 1 行に複数の文を書く場合は、それらを ; で区切る必要があるが、行末に ; をつける必要はない (つけても問題はない)。
   - 複数の文のグループ化は C 言語同様 { } で囲む。
   - C 言語同様、空白、タブ、改行はほぼ任意に書けるが、文の途中で改行する場合は、その改行の直前 (行末) に ¥ が必要になる場合がある。例えば、

```
BEGIN { for (j=1; j<=5; j++) printf "%d %d¥n", j, 2*j }
```

     を、改行を入れて以下のように書いた場合、

```
BEGIN {
    for (j=1; j<=5; j++)
        printf ¥
        "%d %d¥n" ¥
        , j,
        2*j
}
```

     3 行目、4 行目の行末の ¥ は、それを取ってしまうと次の行まで文がつながっているとみなされずにエラーになるので省略できない。逆に 2 行目、5 行目の行末には ¥ は必要ない (つけても問題はない)。

     なお、行末に ¥ を置く場合は、¥ と行末の間に空白を入れてはいけない。

3. 演算子、構文

   - 四則演算 (+, -, *, /, %, ++, --)、比較演算 (<, >, <=, >=, ==, !=)、論理演算子 (&&, ||)、代入演算子、(+=, -=, *=, /=, %= 等)、3 項演算子 (A?B:C) などの演算子は、ほぼ C 言語と同じものが使える。
   - 整数も実数とみなされるので、C 言語とは違い「5/3」は 1 でなく 1.6666... になるが、% (剰余計算) は例外で、「5%3」は 2 になる。
   - 整数の定数には、先頭に 0x をつけた 16 進数整数表記、先頭に 0 をつけた 8 進整数表記も使える。
   - 累乗 x^y は「x^y」と書けるが、$x < 0$ の場合は整数乗 (y が整数) しか許されない ($x \geq 0$ なら y は非整数も可)。
   - if 文、while 文、do while 文、for 文も C 言語と同じものが使えるが、switch 文は AWK にはない。
   - 条件文での真偽は、数値としては 0、文字列としては空文字が偽で、それ以外はすべて真となる。

- printf は、上の例のようにほぼ C 言語と同じ書式であるが、書式文字列や変数をまとめて ( ) で囲む必要はない。

簡単なスクリプトの例をいくつか紹介する。

```
BEGIN { for (j=1; j<=150; j++)
    printf "ren file%d.txt file%04d.txt¥n", j, j }
```

これは、「ren file1.txt file0001.txt」のような行を 150 行出力する。次節のリダイレクションを用いれば、その出力をファイルに保存して、バッチファイルとして実行することもできる (ren はファイル名の変更コマンド)。

```
BEGIN { x0 = 1; x1 = 20
    printf " x、2 乗、平方根 (%d <= x <= %d)¥n", x0, x1
    for (x=x0; x<=x1; x++)
        printf "%2d、%4d、%5.2f¥n", x, x^2, x^(1/2)
}
```

このスクリプトは、平方と平方根の簡単な数表を出力する。

```
BEGIN { a[0] = "日"; a[1] = "月"; a[2] = "火"
    a[3] = "水"; a[4] = "木"; a[5] = "金"; a[6] = "土"
    for (j=1; j<=30; j++)
        printf "2018 年 11 月 %d 日 (%s)¥n", j, a[(j + 3)%7]
}
```

これは、11 月の日付一覧を表示する。配列で曜日を定義しているが、その出力で使われている添字の「(j + 3)%7」は、7 での割り算の余りにより 0 から 6 までを繰り返す数列になり、3 を足すことでそれが 4 から始まるようにしている (1 日が木曜)。

```
BEGIN { N = 27; maxloop = 10000; printf "0: %d¥n", N
    for (j=1; j<=maxloop && N>1; j++) {
        N = (N%2 == 0) ? N/2: 3*N + 1
        printf "%d: %d¥n", j, N }
}
```

これは、ある数列を計算するもので、N の更新に 3 項演算子を使っているが (3 行目)、N が偶数なら N には $N/2$ が、N が奇数なら $3N+1$ が代入される。簡単な条件分岐や、条件で値を変えるものは、if 文でなくても 3 項演算子で行える。

ここまでいくつかの例で見たように、AWK での出力 (表示) には、C 言語と同様の書

式付き出力命令 printf が使えるが、AWK には書式無し出力命令 print も用意されている。

命令	意味
printf "[書式]", [値], [値], ...	書式に従って値を表示
print [値], [値], ...	値を順に表示して改行

表 3: printf, print

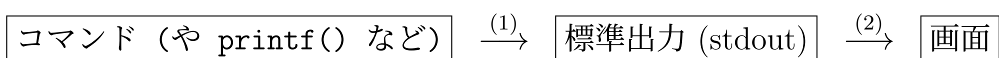
注意:

- print には書式指定はなく、指定した値を (複数指定した場合は) 空白区切りで表示し、最後に改行する (printf は ¥n を指定しないと改行しない)。
- 両者とも命令の後ろの部分を（ ）で囲む必要はないが、「printf("Hello.¥n")」のようにかっこをつけても問題はない。
- 空行を出力するには「printf "¥n"」か「print ""」とする。
- printf の書式文字列は、C 言語とほぼ共通 (%s, %d, %f など)。

AWK で利用できる関数については、次回紹介する。

5 出力リダイレクション

コマンドプロンプト内でのコマンドの画面出力は、**標準出力** (C 言語の stdout) と呼ばれる仮想デバイスを介して行われている。C 言語で言えば、printf() や putchar() 等の関数は標準出力へ出力をする関数である:



コマンドプロンプトやバッチファイルでは、この標準出力と画面とのつながり (2) を、表 4 の 2 種類の **出力リダイレクション** (記号 >,>>) を使うことで画面からファイルにつなぎかえることができる。

種類	名前
[コマンド] > [ファイル]	出力リダイレクション (上書き形式)
[コマンド] >> [ファイル]	出力リダイレクション (追加出力形式)

表 4: 出力リダイレクション

例えば (左端の > はリダイレクション記号ではなくプロンプト記号)

```
> time /t > time1.dat
```

とすると、本来画面に表示される「time/t」の出力 (現在の時刻) が、time1.dat という名前のファイルに保存される。

出力リダイレクションの「>」と「>>」の違いは以下の通り。

- 「[コマンド] > [ファイル]」は [ファイル] がなければ新規に作成し、[ファイル] があれば上書きする (削除して新規に作るのと同等)。
- 「[コマンド] >> [ファイル]」は [ファイル] がなければ新規に作成し、[ファイル] があればその末尾に追加出力する。

例えば、上の時間の保存を

```
> echo %time% >> time1.dat
```

のような追加出力にすれば、実行する度に time1.dat に結果が 1 行ずつ追加される。

数行程度の簡単なテキストファイルであれば、エディタを使わなくても echo コマンドと出力リダイレクション「>, >>」を使うだけで作ることができる。例:

```
> echo ファイルの 1 行目 > test.dat
> echo ファイルの 2 行目 >> test.dat
> echo ファイルの 3 行目 >> test.dat
```

上の 1 行目で「>」を使うのは、test.dat というファイルが既にあったときでもその中身を消して新たに書き始めるためで、2,3 行目で「>>」を使うのは、「>」では最後に実行した echo の行 1 行しかファイルに残らないためである。