2006 年 4 月 5 日
2006 年 4 月 27 日修正

AWK による簡単なタイプ練習ソフト

新潟工科大学 情報電子工学科 竹野茂治

1 はじめに

本稿では、AWK の行単位フィルタの機能を利用して、簡単なタイプ練習ソフトを作ってみることにします。

同じことを C 言語でやることもそう難しくありませんし、AWK で作ることにそれほど意義があるわけではありませんが、AWK プログラミングの 1 例として紹介したいと思います。

また、このプログラムでは「ユーザ定義関数」を積極的に使います。ユーザ定義関数 (サブルーチン) の考え方は、AWK に限らず C でも、他のプログラム言語でも重要なものですので、プログラミング全般の勉強にもなるのでは、と思っています。

2 タイプ練習ソフトの仕様

本稿では、以下のような機能を持つタイプ練習ソフトを作成することを目標とします。

- 40 文字程度のアルファベットなどの文字列を画面に表示し、それを課題として、ユーザにそれと同じ文字列を入力させる。
- 入力後に課題と入力の違いの数や、入力にかかった時間などを表示する。
- 課題と入力に違いがある場合は、再び同じ課題を入力してもらい、違いがない場合はそこで終了する。
- 途中でやめることもできるようにする。

もちろん、他にも色々な機能を追加することも可能ですが、とりあえずはこれを目標とします。

色んな要求が含まれていて難しいと感じるかもしれませんが、おおまかな設計図を書いて複数の関数 (サブルーチン) を作るように問題を分けて考えていけば、そう難しくはありません。

3 おおまかな設計図

[1] に書いたように、AWK を行単位のフィルタとして使う場合、実行部分は大きく 3 つのブロックに分かれます (そのうちいくつかは省略可)。

1. **BEGIN{} ブロック**: 最初 (入力の前) に 1 回だけ実行される
2. **{} ブロック**: 各入力行に対して 1 回ずつ実行される
3. **END{} ブロック**: 入力が終わった後に 1 回だけ実行される

{} ブロックは、その前に条件 (パターン) をつけて、そのブロックを適用する行を選択することも可能ですし、{} ブロック自体を複数置くことも可能ですが、それはパターンのない一つの {} ブロックでも同じことが実現できますので、ここでは簡単のため、パターンのない一つの {} ブロックで作ることにします。

課題に対するキーボードからの入力 (標準入力) を {} ブロックに対する各入力行と見れば、このプログラムのおおまかな設計図として、次のようなものが考えられます。

```
BEGIN{
  #（1）課題文を作成して画面に表示
  #（2）現在時刻を記憶（入力開始時刻）
}
{ ## ここはキーボードからの 1 行の入力が済んだ後に実行される
  #（3）現在時刻を記憶（入力終了時刻）
  #（4）入力行と課題文とを比較
  #（5）違いがなければ終了し、
  #       違いがあればその結果を表示し、現在時刻を記憶（入力開始時刻）
}
END{
  #（6）途中でやめた場合はそのような表示をする
  #（7）正常終了した場合はそのような表示をする
}
```

ここでは、AWK のコメント (# から行末までが AWK のコメント) の形で書いてみました。

ちなみに、私がプログラムを実際に書くときは、だいたいいつもこのようなおおまかな設計図から書き始めます。後はこの (1)~(7) の部分を実際に作っていけばいいわけです。

4 課題文の作成

4.1 乱数生成関数

課題文は、毎回同じ文字列ではおもしろくありませんから、毎回違った文が得られるように「乱数」を利用して作ることにします。

なお、実際のタイプ練習では、ランダムな文字列よりも意味のある文字列の方が入力しやすいので、課題文は意味のある文章例などを利用する場合もありますが、ここでは簡単のためランダムな文字列を使うことにします。

課題文の文字は、簡単のため a∼z (小文字) と空白からなる文字列とします。空白は適当に入れてやる方が入力しやすいと思いますので、入れることにします。

AWK で乱数を使う場合は、rand(), srand() という関数を使用します。

- `rand()`: 呼ばれる度に 0 以上 1 未満のランダムな数値を返す (引数はなし)
- `srand()`: rand() の種を設定する

これらを例を用いながら説明します。まず、実際に rand() がどのような値を返すのか見てみることにします。

```
BEGIN{ for(j=1;j<=10;j++) print rand() }
```

というスクリプト test1.awk を

```
awk -f test1.awk
```

と実行させます。このスクリプトは `BEGIN{}` ブロックしかないので、AWK は `BEGIN{}` ブロック内を実行するとすぐに終了します。for 文は C 言語と同じ構文で、このスクリプトでは "`print rand()`" を 10 回実行しています。print rand() は、rand() の返す実数値を 1 行ずつ表示するだけです。rand() は引数はありません。上のスクリプトの実行結果は以下のようになります (FreeBSD 4.7 上の gawk–3.0.3+mb1.09 の場合)。

```
% awk -f test1.awk
0.487477
0.0715607
0.00395315
0.832913
0.765252
```

```
0.599829
0.131783
0.88603
0.0732417
0.34195
```

これでランダムな数字の列が作られるのですが、実はこれはランダムに見えるだけで、ある規則 (数式) で生成されている疑似乱数列にすぎません。よって、もう 1 回上のスクリプトを実行するとこれと同じ乱数列が生成されてしまいます。これでは、課題文をランダムにできても、毎回実行の度に課題を変えることができません。

そこで乱数列を変化させるのに使われるのが srand() で、srand() は、疑似乱数列をどこから始めるのかを決める関数です。しかし、引数として一定の数字を与えてしまうと、それによってまた一定のところから始まる同じ乱数列が生成されてしまいます。実は、srand() は引数を省略すると、「srand() を実行した時刻 (単位は秒)」を使って rand() の疑似乱数列の初期値を設定してくれますので、これによって実行の度に違った乱数列を得ることができます。

今度は、

```
BEGIN{ srand(); for(j=1;j<=10;j++) print rand() }
```

というスクリプト test2.awk を実行してみます。

```
% awk -f test2.awk
0.270585
0.666318
0.724231
0.813703
0.810067
0.0925101
0.00122359
0.722831
0.084683
0.953172
```

このスクリプトは、実行時刻が少なくとも 1 秒以上違っていれば、実行する度に違った乱数列を生成します。この例のように、srand() は、rand() を実行する度に呼びだす必要はなく[1]、そのプログラムの中で最初に 1 回だけ呼びだします。

[1]むしろそうしてはいけません。なぜか考えてみましょう。

なお、AWK の「関数」は、C 言語同様、値を返してくれる関数と、値を返さず何らかの作業を行うだけの関数の 2 種類があります[2]。AWK では C 言語同様両方とも関数と呼びますが、プログラム言語によっては、これらをそれぞれ「関数」「手続き」のように区別することもあります。その意味では、rand() は関数、srand() は手続き、となります。

また、C 言語でも、通常この rand(), srand() と同様の関数が用意されていますが、C 言語では普通 srand() (に相当するもの) は引数を省略することはできず、引数として現在の時刻を実際に与えます。

4.2 ランダムな整数列

さて、0 以上 1 未満の乱数から、ランダムな文字列を生成するにはどうすればいいでしょうか。これは、例えば次のようにすればできます。

1. アルファベットの文字に $1 \sim N$ までの番号をつける
2. $1 \sim N$ までの整数の乱数を 40 回作る
3. その数字に対応する文字をひとつずつ並べた文字列を作る

rand() は 0 以上 1 未満の数を返すので、例えばそれを 3 倍すれば 0 以上 3 未満の数になり、その整数部分をとれば 0,1,2 の数がランダムに得られることになります。整数部分を求める関数は int() で、

`int(x)`: x の整数部分を返す

となっています。よって、

```
x = int(rand()*3)
```

により、x に 0,1,2 のランダムな数が得られます。

なお、rand() を 2.5 倍したものは 0 以上 2.5 未満の数なので、

```
x = int(rand()*2.5)
```

[2]数学では普通これよりも狭く、引数があるもののみを関数といいます

でも x には 0,1,2 のランダムな数が得られますが、これでは 0,1,2 の現れる確率が違ってしまいます。

rand() は 0 以上 1 未満の「一様乱数」を生成するように設計されていますので、rand() の 3 倍の値が

- 0 以上 1 未満 (整数部分は 0)
- 1 以上 2 未満 (整数部分は 1)
- 2 以上 3 未満 (整数部分は 2)

になる確率は等しく、それぞれ 1/3 となります。ところが、rand() の 2.5 倍だと、それは 0 以上 2.5 未満の一様乱数になりますので、それが 2 以上 2.5 未満の値になる確率は、0 以上 1 未満の値になる確率や 1 以上 2 未満の値になる確率の半分しかありません。

一般に、$0 \leq s < t \leq 1$ に対して、rand() の値が s 以上 t 未満になる確率は $(t-s)$ となります[3]。これを使って、`x = int(rand()*2.5)` が 2 となる確率を計算してみましょう。

$$
\begin{aligned}
&\text{「}\texttt{int(rand()*2.5)} = 2 \text{ となる確率」} \\
&= \text{「}2 \leq \texttt{rand()*2.5} < 3 \text{ となる確率」} = \text{「}\frac{2}{2.5} \leq \texttt{rand()} < \frac{3}{2.5} \text{ となる確率」} \\
&= \text{「}\frac{2}{2.5} \leq \texttt{rand()} < 1 \text{ となる確率」} = 1 - \frac{2}{2.5} = \frac{1}{5}
\end{aligned}
$$

となります。

4.3 文字列値配列の生成

文字に番号をつけるには、配列を利用すればいいでしょう。簡単に言えば、

```
a[1]="a", a[2]="b", a[3]="c",..., a[26]="z", a[27]="␣"
```
（␣ はスペースを意味します）

のようにするわけですが、これを 27 個も書くのは面倒なので、ここでは split() を利用する方法を紹介します。

[3]正確には、rand() の返す値が本当に一様乱数ならばそうなりますが、実際には rand() の値は近似的な一様乱数に過ぎないので、厳密にはそうとは言えません。

split(s, h, r): 文字列 s を、正規表現 r を区切りとして区切って配列 h に保存し、その個数を返す

正規表現 r を省略すると、split() はスペース、タブ区切りとなります。例えば、

```
x=split("A␣23␣␣c",h)
```

の結果は、

$x = 3$, h[1]="A", h[2]="23", h[3]="c"

となります。一方、r として "" (空文字列) を指定すると文字列を 1 文字ずつ切り離してくれますので、

```
x=split("A␣23␣␣c",h,"")
```

の結果は、

$x = 7$, h[1]="A", h[2]="␣", h[3]="2", h[4]="3", h[5]="␣", h[6]="␣", h[7]="c"

となります。よって、a∼z の配列は、

```
split("abcdefghijklmnopqrstuvwxyz",a,"")
```

で作れることになります。

しかし、スペースも入れて、しかもスペースが適当に出るようにしたいので、5 回に 1 回くらいスペースが出るようにスペースを 6 個追加して $(6/(26+6)=6/32\approx 1/5)$、

```
N=split("abcdefghijklmnopqrstuvwxyz␣␣␣␣␣␣",a,"")
```

とします。

あとは、$1 \sim N$ までの乱数を

```
x=int(rand()*N)+1
```

と作って、これを 40 回やればランダムな 40 字の文字列が作れることになります。

4.4 ユーザ定義関数化

ここまでの内容を一つの関数にしてみましょう。

```
function mkkadai(M,  j,x,N,a,s)
{
    s=""
    srand()
    N=split("abcdefghijklmnopqrstuvwxyz      ",a,"")
    for(j=1;j<=M;j++){
        x=int(rand()*N)+1
        s = s a[x]
    }
    return s
}
```

これを少し説明します。

`function mkkadai(M,...)` は、これが mkkadai という名前の関数の定義ブロックであることを意味します。スクリプトのこの部分はすぐに実行されるわけではなく、実行ブロック内でその名前の関数が使われたときに、この定義部分が実行されます。

`M,j,x,N,a,s` は引数を意味しますが、この関数の場合実際に引数として与えられるのは M だけで、後は引数というよりもむしろ関数内部でのみ通用する「局所変数」です。ここは、AWK 独特のテクニックで、詳しいことは付録 A をご覧ください。とりあえずは、AWK の関数定義では、

```
function 関数名（引数,...,引数,   局所変数,...,局所変数）
{
    関数本体
}
```

と書くのだと覚えておけばいいでしょう。引数と局所変数の間はタブやスペースを入れて区別して書くと見やすいと思います。ここに書かれていない変数を関数内部で使用すると、それは局所変数ではなく AWK スクリプト全体で通用する外部変数 (大域変数) と見なされます。

srand() がこの関数の中に入っていますが、この関数 mkkadai() はプログラムで 1 回しか呼ばないということでここに入れてあります。しかし、この関数が複数回呼びだされる可能性がある場合は srand() ここには入れずに、`BEGIN{}` ブロックの先頭で呼ぶようにした方がいいでしょう。

return s は、s を関数の値として返す命令で、よって s に 40 文字 (ここでは M 文字) の文字列を作っています。s の作成には文字列の連結を使っていますが、AWK で文字列を連結する方法は 2 種類あって、

- 単にスペースをあけて並べる $\Rightarrow$ そのまま連結される
- , (コンマ) を入れて並べる $\Rightarrow$ スペースが一つ入って連結される

ということになっています。例えば、

s="a"␣"b","c" $\Rightarrow$ s="ab␣c"

となります。

この関数では最初に s = "" と空文字列として、s = s a[x] で、s の最後に a[x] を連結しているので、a[x] を M 回 s の後ろに連結していくことで M 文字分の文字列ができることになります。

文字列の長さは 40 と固定せずに引数として M としましたが、これは後でこの数字を変更したい場合のことを考えてこのようにしています。

もし、関数内で M でなくて 40 と書いた場合、その数値を後で変更しようと思ったら関数の中の 40 をすべて変更していかなければいけませんが、このように変える可能性のある数字を引数とすれば、この関数を呼びだすときに引数を一つ変えるだけで済みます。これは、独立性や汎用性の高いサブルーチンを作るときに標準的に行われる方法ですので、覚えておくといいでしょう。

そういう点からすると、実は "abc..." の文字列もこの中に書かずに、引数として与えたり外部変数としたりする方が汎用性は高くなります。例えば、アルファベットの大文字を追加したり、記号を追加したり、または課題のレベルに従って文字を増やしたりしたい場合は、この文字列を変える必要がありますが、その場合この文字列を関数の外に出してしまってそれを引数として与える仕組みにすれば変更は容易になります。

5 入力チェック関数

次は、おおまかな設計図の (4) の課題と入力の違いの比較を行う関数を作成します。引数として、$s_1 =$ 文字列 1 (課題文字列)、$s_2 =$ 文字列 2 (入力文字列) を与え、その違いの数を返す関数を作ることにします。

簡単に言えば、4.3 節でやったように文字列を 1 文字ずつの配列に展開し、それを先頭から比較していけばいいのですが、その前に、関数の仕様について、もう少し細かく検討しておきます。

s_1 の長さ $(= N_1$ とします$)$ と s_2 の長さ $(= N_2$ とします$)$ が等しいときはそれでいいのですが、違うときも考えておく必要があります。

例えば、単純に 1 番目から N_1 番目までの比較しかしないのだとすると $N_1 < N_2$ で、1 番目から N_1 番目までは全く同じだけど、その後に余計な文字列がついている、という場合もありえます。それはタイプ練習ソフトとしては正解とはいい難いと思います。よって、

- $N_1 = N_2$ のときは、N_1 文字目までの違いの数を返す
- $N_1 < N_2$ のときは、$(N_1$ 文字目までの違いの数$) + (N_2 - N_1)$ を返す
- $N_1 > N_2$ のときは、$(N_2$ 文字目までの違いの数$) + (N_1 - N_2)$ を返す

とするのが自然だと思います。

すなわち、文字列の短い方に合わせて比較し、そこまでの違い $(= \mathrm{diff})$ と、その余りの分、つまり N_1 と N_2 の差 $(= L)$ とを足したものを返せばいいことになります。

よって、おおまかには次のような関数を作ればいいことになります。

```
function check(s1,s2,  h1,h2,N1,N2,N,L,j,diff)
{
    # s1 を配列 h1 に展開（N1 がその長さ）
    # s2 を配列 h2 に展開（N2 がその長さ）
    # N = N1 と N2 の小さい方
    # L = N1 と N2 の差 (=|N1-N2|)
    for(j=1;j<=N;j++) if(h1[j]!=h2[j]) diff++
    return diff+L
}
```

C 言語同様、!= は「等しくない」ことを表します。diff は局所変数で、明示的に初期化しなければ 0 と初期化されますから、この for 文で、短い方に合わせての比較が行えることになります。

最初の配列に展開する部分は、4.3 節で説明したように、

```
N1=split(s1,h1,"")
N2=split(s2,h2,"")
```

とすれば済みますし、N と L も、

```
    if(N1<=N2) N=N1; L=N2-N1
    else N=N2; L=N1-N2
```

で設定できます。

6 その他の部分

残りの部分はそれほど難しくはありません。

現在の時刻は、秒単位ですが systime() を使えば取得できます。

- `systime()`: 1970 年 1 月 1 日 0 時 0 分 0 秒 (UTC)[4]から現在の時刻までの秒数を返す

(5) の、違いがなければ終了、は exit でできます。exit は入力の読み込みをやめて、`END{}` ブロックに進みます (`END{}` ブロックがなければそこで終了)。

途中で入力をやめる場合は、いわゆる **EOF** と呼ばれる、入力の終わりを示すコードを送ってあげればいいです。MS–DOS/MS–Windows の場合は Ctrl–Z, Unix の場合は通常 Ctrl–D で EOF が送られます。

これで自動的に入力が終了するので `END{}` ブロックに進みます。

各行毎の実行ブロックでは、入力行は `$0` と取得できます。

7 ソースコード全体

以上をまとめたソースコード全体を紹介します。

```
BEGIN{
    len=40
    # (1) 課題文を作成して画面に表示
    kadai=mkkadai(len)

    print "課題は [ ] 内の文字列です。それと同じ文字列を入力してください。"
```

[4]この時刻のことを Unix Epoch といいます。基本的にすべての Unix の時刻はこれを基点として秒数で数えていますので、秒数を符号付き 32 ビット整数で保持している Unix の場合、Unix Epoch から 2^{31} 秒後となる 2038 年 1 月 19 日 03:14:08 が、1901 年 12 月 13 日 20:45:52 になってしまう 2038 年問題が指摘されています。

```
    print "[ ] は入力しないでください。入力が終わったら改行キーを打ちます。"
    print ""
    printf "課題: [%s]\n",kadai
    printf "入力:  "

    # (2) 現在時刻を記憶（入力開始時刻）
    time1=systime()
    time2=0
    Loops=1 # 何回目の入力か
}
{   ## ここはキーボードからの 1 行の入力が済んだ後に実行される
    # (3) 現在時刻を記憶（入力終了時刻）
    time2=systime()
    # (4) 入力行と課題文とを比較
    N=check(kadai,$0)
    # (5) 違いがなければ終了し、
    #     違いがあればその結果を表示し、現在時刻を記憶（入力開始時刻）
    if(N==0) exit
    else{
        printf "[%d] ミス = %d 個、時間 = %d 秒\n",Loops,N,time2-time1
        printf "課題: [%s]\n",kadai
        printf "入力:  "
        time1=systime()
        time2=0
        Loops++
    }
}
END{
    # (6) 途中でやめた場合はそのような表示をする
    # (7) 正常終了した場合はそのような表示をする
    if(time2==0) print "途中であきらめましたね。次回また頑張ってください。"
    else{
        printf "おめでとうございます。正解です。\n"
        printf "回数 = %d 回、時間 = %d 秒\n",Loops,time2-time1
    }
}

function mkkadai(M,  j,x,N,a,s)
{
    s=""
    srand()
    N=split("abcdefghijklmnopqrstuvwxyz      ",a,"")
```

```
    for(j=1;j<=M;j++){
        x=int(rand()*N)+1
        s = s a[x]
    }
    return s
}

function check(s1,s2,  h1,h2,N1,N2,N,L,j,diff)
{
    N1=split(s1,h1,"")
    N2=split(s2,h2,"")
    if(N1<=N2){ N=N1; L=N2-N1 }
    else{ N=N2; L=N1-N2 }
    for(j=1;j<=N;j++) if(h1[j]!=h2[j]) diff++
    return diff+L
}
```

上記のファイルを type.awk とでもして、

```
awk -f type.awk
```

とすれば実行されます。

8 最後に

今回は単純な CUI の対話型ゲームとして、簡単なタイプ練習ソフトを取りあげてみました。

必要最小限の機能しか入れていませんからよりゲーム性を高めたり、タイプ練習ソフトとしての機能を高める改良が色々考えられると思います。例えば、

- 正しく打った数も数えて、正答率を表示する
- ミスや時間を点数化する
- 得点の履歴をファイルに残せるようにする
- 成功までの正答率や一分間の打鍵率をグラフ化する
- 課題をレベルに応じて変化させる

- 制限回数や制限時間を設けて、それをクリアできないとレベルを落とす
- アルファベットの大文字や記号も対象にする
- 課題を具体的な文章や単語にする
- 日本語の課題、入力にも対応させる

など、色々な発展が考えられるでしょう。AWK の範囲でも、上記のいくつかは容易に行えます。是非ユニークでおもしろいものを作ってもらいたいと思います。

A ユーザ定義関数の引数

ユーザ定義関数の引数 (本当の引数、局所変数) および外部変数 (大域変数) について、少し説明をします。

実は AWK 自体は、ユーザ定義関数の引数部に書かれたものはすべて引数 (仮引数) と認識していますが、AWK の関数は実行時に引数の省略が可能なので、実際に値が与えられたものだけに代入、すなわち関数実行時の引数の初期化が行なわれています。

例で説明します。

```
function f1(a,b,c,d,e,f)
{
  ...
}
```

というユーザ定義関数を、`f1(1,2)` と使えば、`a=1, b=2` が代入されますが、`c,d,e,f` は初期化されませんから、暗黙の初期化、すなわち、`c="", d="", e="", f=""` (数字として使う場合はいずれも 0) が行われているものと見なされます。

しかし、あくまで仮引数なので、これらの変数はこの中だけで通用する変数です (これは C 言語と同じです)。つまり、この関数の外 (例えば実行ブロック) で `a,b,c` などの変数が使われていても、それはこの関数の引数の `a,b,c` とは無関係なものと見なされます。例えば、

```
BEGIN{
  a=1; b=3;
  f1()
  printf "a=%d, b=%d\n",a,b
}
```

```
function f1(a,b){ a=5; b=10 }
```

を実行すると、表示されるのは “a=1, b=3” です。

AWK では、変数宣言がないので局所変数の定義ができないのですが、その代わりに仮引数を局所変数として使っているわけです。

なお、本当に引数を省略する関数 (デフォルトで値を設定する関数) を作りたい場合は、このような状況を利用して、次のようにします。

```
function f1(a,b,c)
{
   if(a=="") a=1 # a のデフォルトの値
   if(b=="") b=2 # b のデフォルトの値
   if(c=="") c=3 # c のデフォルトの値
   return a+b+c
}
```

このようにしておけば、

```
   f1()=6 (=1+2+3), f1(5)=10 (=5+2+3), f1(5,6)=14 (=5+6+3)
```

のようになります。

一方、仮引数に書かれていない変数が使われたときは、それは外部変数 (大域変数) と見なされます。例えば

```
BEGIN{
  a=1; b=3;
  f1()
  printf "a=%d, b=%d\n",a,b
}

function f1(a){ a=5; b=10 }
```

を実行すると、表示されるのは “a=1, b=10” となります。f1() では b は仮引数としては書かれていないので、BEGIN{} ブロックで使われている変数 b だと思われて、その値が変更されます。

外部変数 (大域変数)、局所変数は C 言語でもでてくる重要な考え方です。しっかり把握しておきましょう。

参考文献

[1] 竹野茂治、「AWK に関する基礎知識」(2006)

[2] A.V. エイホ、B.W. カーニハン、P.J. ワインバーガー (足立高徳訳)、「プログラミング言語 AWK」、新紀元社 (2004) (元版は 1989)

[3] D.Dougherty、A.Robbins (福崎俊博訳)、「sed & awk プログラミング 改訂版」、オライリー・ジャパン (1997)

[4] 志村拓、鷲北賢、西村克信、「AWK を 256 倍使うための本」、アスキー出版 (1993)