プログラミング環境 第7回

久野 靖 [*]

1991.11.12

9 テキストエディタと行エディタ ed

9.1 テキストエディタの歴史

9.1.1 テキスト編集の必要性

さて、今回は久しぶりに「プログラミング環境」らしく、テキストエディタの歴史の話から始めよう。計算機ができたばかりの頃はコンソールパネルのボタンやスイッチで直接プログラムを主記憶に書き込んでいた、という話はずっと前に出てきた。しかしアセンブラ語とか、Fortran のようにもう少し高級な言語が出てくるとそれは「文字の集まり」で表現されているから、それを扱う必要がある。ここで「扱う」というのは、

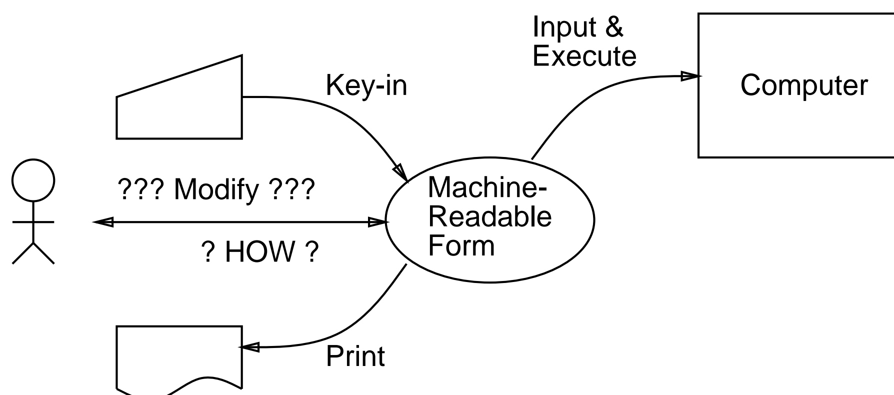


図 1: テキスト編集の概念

1. 文字の集まりであるプログラムを、何らかの計算機に読める (Machine-Readable、機械可読なんて訳すこともある) 形にすることがまず必要である。そうしたら機械に読み込ませて実行できるから。
2. プログラムには間違いはつきものである。だから、一度機械可読形式にしてしまったプログラムを、意図した通りに変更できなければいけない。
3. 普通、機械可読なプログラムはそのままでは人間には読めない。だから、人間に読めるように紙に印刷したり画面に表示したりする仕事も必要である。

などがあろう。打ち込んだり打ち出したりするのはまだしも、一度打ち込んだものを全部打ち直さずに修正する、というのはなかなか難しいものがある。これがこの節のテーマなわけである。

* 筑波大学経営システム科学専攻

9.1.2 オフライン媒体の時代

さて、初期の計算機システムでは、機械可読形式にする、というのは要するに紙テープやパンチカードに打つことを意味する、というのが一般的であった。[1]カードの場合には1行が1枚のカードに対応するので、プログラムの行を入れ換えるのはカードを入れ換えるだけで済む。行の挿入や削除も簡単である。[2]行の中の一部を直したい場合にはそのカードを打ち直すことになるが、これもパンチ機にそのための機能が備わっているのでそれほど大変ではない。紙テープの場合には行単位で分かれてもいないし内容の印字もないのでだいぶ大変だが、切ったり貼ったりするための道具とか、全部の場所に穴を空けたりする道具などが用意されていてそれで編集をした。

9.1.3 バッチエディタの時代

しかし、プログラムが大きくなってくると、それを毎回読ませるのは大変であるし、カードケースを何箱も抱えて計算センターまで歩いて行く、というのも重労働である。そこで必然的にプログラム等は一度読ませたらあとはディスクにいれておき、修正したければディスク上のファイルを修正する、という (今ではごく当たり前の) やり方に移行することになる。しかし計算機システムについているのは依然としてカードリーダとプリンタであるから、現在のように端末から対話的に編集する、というのができない。そこで予めどこをどう編集する、というデータをカードに打ち、それを食べさせるとそれに従ってディスク上のファイルを変更するプログラム、つまりバッチエディタが動く、という方式になったわけである。

当時よく見られたバッチエディタの方式は次のようなものである。まず、プログラム等のファイルは各行に5桁くらいの一連番号を、ただし10とびくらいにつけておく。そして、ある行を修正したい場合には置き換えるべき内容のカードを打って、それに置き換えたい行と同じ番号をつけておくと置き換わる、ということにする。挿入の場合には番号がとびとびなのを利用して、間の番号をつけたカードを用意すればそれが然るべき場所に入る。削除はしかたないから「どの行番号からどの行番号まで削除」と指定した制御カードを用意する。これをやってゆくときれいにつけてあった番号が不揃いになるので、時々番号を振り直す。

これを聞いて、どこかで見たことがあると思った人はいませんか。そう、実はマイコンBasicの処理系では今でも (いちいちカードを打つわけではないが) こういう行番号方式の編集が使われていることがある。あと、IBMカードでは一連番号を80桁のうち右端8桁に打つのが普通だったわけだが、現在でもなぜか各行が80桁きっちりの長さで、右端に一連番号が打ってあるというファイルを (特にFortranのソースに) 見かけることがある。これによって浪費されているディスク容量の合計は、全世界で合計したらどのくらいになるだろう。実に、「歴史的事情」というのは恐ろしいものである。

ただしすべてのバッチエディタがこういう行番号方式であるという必然性はなく、もっとスマートなものも存在する。例えばsedなどもバッチエディタとして使うことができるが、これは上のやり方とは全然異なる。ともあれ、次に述べるような対話的編集が可能になった現在はバッチエディタが活躍する場はあまり広くないが、それでもそれなりの存在意義はあるものと思う。

9.1.4 対話型エディタの時代

バッチエディタの最大の欠点は、「このように直そう」と思って編集データを用意したのに、動かしてみたら些細なまちがいから全く意図したのと違うことになっていた、ということが起きが

[1]パンチカードの時代の終り頃、「キーツーフロッピー」というのがはやったことがあった。これはなんと、カードパンチ機のカードをフロッピーに置き換えたもので、キーボードをたたくとその内容がフロッピーに入るだけの巨大な机がそのために売られていた。

[2]ただし、これらはカードの上部にそのカードに打たれている内容が印字してあれば、の話であり、それがないとカードの穴のパターンを読む才能が必要になる。

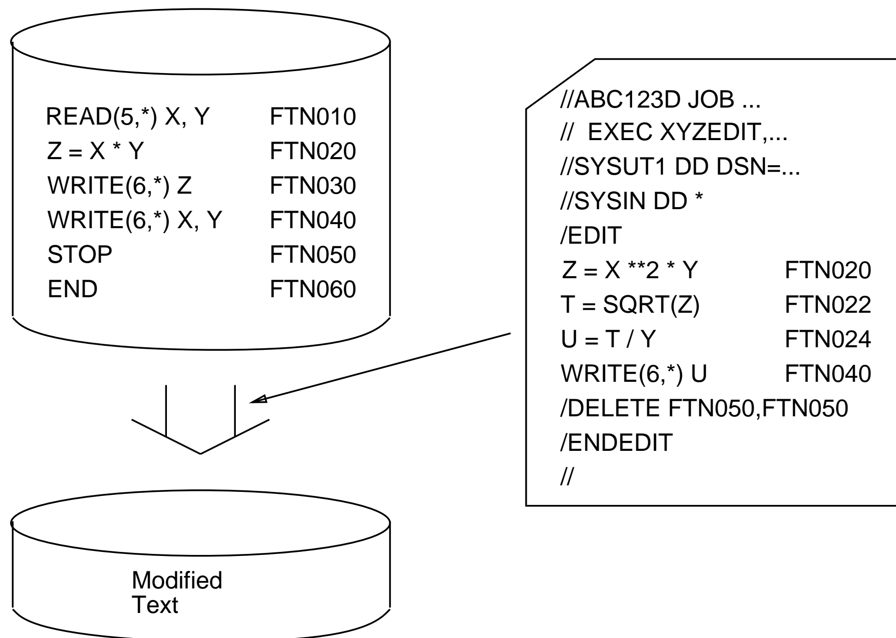


図 2: 古典的なバッチエディタ

ちなことである。そこで、計算機に端末がつながって対話的に使えるとなると直ちに、「こう直す」と言ったらすぐその通りの修正が行え、その結果が端末に打ち出されて確認できる、というエディタが使われるようになった。実は Unix の標準エディタである ed はそのような対話型エディタの代表的なものであった QED の子孫であり、あとで ed を体験していただけばそれがどのようなものかは十分おわかりいただけるはずである。他にもこの手のエディタの有名なものとしては teco、SOS などがある。

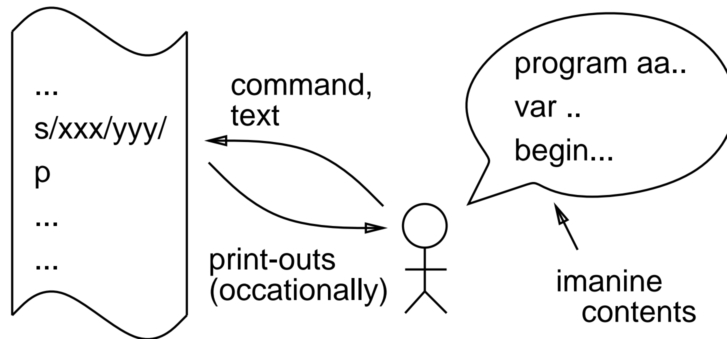
さて、当時のシステムでは端末というのはテレタイプが当たり前であり、印字速度も毎秒 10 文字といったところだった。そこでこれらのエディタはどれも「打ち出せ」という指令をもらうまでは編集された結果なども端末に打ち出さないようにできていた。10 行打ち出してみるのに 1 分もかかるのではそれは当然のことであるが、従って編集する人もファイルの中身を思い浮かべながらめくらで編集を進め、ひと区切りついたらちょっと打ち出してみる、というやり方をした。

やがて画面端末が普及し、毎秒 1000 文字くらい平気で画面に表示できるようになると、いちいち「ここを表示しろ」というかわりに編集している付近を絶えず自動的に画面に表示して置く方がよい、ということになった。これが現在の画面エディタということになる。ただし一口に画面エディタと言ってもハードの制約や設計思想から様々なバリエーションがある。とても全部を列挙はできないが、代表的なカテゴリをあげておく。

大型機文明の画面エディタ IBM を代表とする大型機文明では、端末装置というのはいくらキーボードを叩いてもそれらは端末内に蓄えられ、最後に「送信」キーを押すとはじめてそれらがまとめて計算機に送信される、という方式になっている。したがってそのような機種での画面エディタは端末画面上で端末の機能を用いて表示されているものの修正を行ったり、指令を指令用の場所に打ち込んだりして、最後に「送信」キーを押すとそれがまとめて画面に反映される、というやりかたである。[3]

パソコン文明のエディタ パソコン文明のエディタはともかく「売りもの」であるから、初心者にも分かりやすく工夫されているものが多い。例えば、絶えず画面上にどのキーはどの指令か、

[3]それで、計算機が落ちても送信を押すまではわからないので一生懸命編集を続けていたりするそうな。



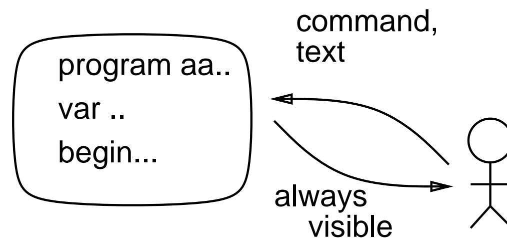


図 3: テレタイプエディタと画面エディタ

というガイドが出ていたり、指令が木構造のメニューになっているなどがある。また、パソコンについているファンクションキーなどを駆使しているものが多いが、これは普通のキー→文字入力、ファンクションキー→メニューや指令、という使い分けを物理的に明示することで分かりやすくするための工夫といえる。これらの特性は以下に出てくる Unix/プログラマ文明のエディタとは大幅に異なっている。すなわち、プログラマ文明ではホームポジションから指をそらすファンクションキーを使ったり、せっかく編集のために利用できる画面の一部をメニューなどの表示に割くのは「罪悪」だからである。

行エディタの画面エディタ化 UCB で Unix 文明が画面端末に移行しようとした時行なわれたことは、ed のようなエディタ (正確には UCB で作られたその強化版である ex) をもとに、それを画面エディタに改造する、というものであった。これはこれで一つの見識である。その結果できたのが vi である。vi の特徴は、テキストを打ち込むモードと指令を打ち込むモードがはっきり区別されている点である。これは指令として使える文字の数が多いと言う利点をもたらすが、一方でモードを気にしていないと使えないという弱点にもつながる。おまけに vi ではもとの ex の指令も基本的にすべて使えるので、機能は豊富だがやや複雑で初心者には使いづらい、という評価につながっている。

Emacs Emacs はもともと Lisp 文明から来たエディタといえるが、現在では GNU Emacs のおかげで Unix でも広く使われている。その特徴は全ての図形文字は「その字をバッファに挿入する」という指令であり、したがって vi のようなモード切替えが不要という点である。一方、その結果指令はすべて Control キーや Meta-キーを使って指定するので、指 (特に小指) が疲れるし、vi ほど速く打てないという弱点もある。その他の Emacs の特徴としては、複数のバッファを持って多数のファイルを並行して扱えること、表示もそれに呼応して多数の窓を同時に表示できること、内部では Emacs-Lisp という Lisp 処理系が動き、これによって指令の実行が行なわれるため Lisp のプログラムを書けばほとんどどんな風にでも拡張でき

る、などが挙げられる。

マウス方式のエディタ Macintosh をはじめとする最近のマウスを装備したシステムでは、編集したい付近を表示したり、編集箇所を指示するのにマウスを使用し、また編集指令そのものもキーボードではなくメニューやボタンで指示するようなエディタを備えているものが多い。これらのエディタの機能は決して強力という訳ではないが、直観的に分かりやすく操作のしかたも簡単なので初心者にはとっつきやすいと言える。

なお、この中に自分のひいきのエディタがないといって文句を言わないように。この他エディタの話題はいくらでもあるが、きりがないのでこの辺でおしまいにして、ed の話に戻ろう。

9.2 Ed – Unix の標準エディタ

9.2.1 Ed の基本的考え方

Ed でファイルを編集するには

```
ed ファイル名
```

のようにファイル名を指定して ed を起動する。ed ではプロンプトは出ないので一見何も起きないように見えるが、ちゃんと指令を受け付ける状態になっている。Ed は他の多くのエディタと同様、編集が始まる時に内部バッファにファイルの内容を読み込み、その上で変更を施す。また、編集が終ったらその結果を指令によりファイルに書き出さなければ、変更の結果はファイルに反映されない。

9.2.2 指令の形式、アドレス

Ed の指令の形式は sed とそっくりで (実は sed が ed に似せてあるのだが)、

[アドレス [, アドレス]] 指令 [引数]

のようなものである。ここで「アドレス」は要するに「どの行か」を表す情報である。一つの指令についてアドレスは最大 2 つまで指定できる。例えば「d」(行消去) 指令では 2 つ指定すると「この行から、この行までを消せ」の意味になるし、1 つだと「この行を消せ」、0 だと「今いる行を消せ」という意味になる。アドレスとしては次のようなものが指定できる。

1. `.` – これは、「現在行」つまり最後に操作したり打ち出したりした行を表す。
2. 番号 – これは「何行目」を表す。ところで、バッファがからっぽの時は行が 1 つもなくて `.` もない – と不便なので、「0 行目」というのが仮想的に常に存在することになっている。
3. `$` – これは「最後の行」を表す。
4. `/パターン/`および`?パターン?` – これは、下 (上) 向きに指定したパターンを探していって、見つかった最初の行を表す。[4]
5. アドレス`+n` およびアドレス`-n` – これは、上の 4 種類のアドレスで指定した行の n 行先/前の行を意味する。[5]

以下の説明では「`(.,.)d`」のような記法を使うが、これは「アドレスは最大 2 つまで持てるが、省略した場合は `.` を指定したものと見なす」という意味である。

---

[4] 例えば Pascal であるブロックの中にいて、そのブロックの中をそっくり消したい場合には`?begin?,/end/d` でできるのだけど、こういうのは画面エディタでやると以外と時間掛かったりしませんか?

[5] 例えば端末画面の行数が 24 位だった場合、`.,.+20p` などはよく使う指令である。

9.2.3　行を打ち込む指令

まず、プログラムを組むのには行を打ち込まないといけない。そのための指令が 3 つある。

(.)a – Append. 指定した行の下に新しい行を打ち込んで行く。

(.)i – Insert. 指定した行の上に新しい行を打ち込んで行く。

(.)c – Change. 指定した行のかわりに新しい行を打ち込んで行く。

いずれも、指令を打ち込むと入力モードに切り替わり、続いて何行でも行が打ち込める。最後に . のみからなる行を打ち込むと入力モードが終ってまた指令が打ち込める。

9.2.4　打ち出し、削除、移動

編集した内容が正しいかどうかを確認するための打ち出しは p 指令で指定しないと自動的には起こらない。p 指令には「どこから、どこまで」で 2 つまでアドレスがつく。削除も同様。移動の場合は「どこへ」で 3 つ目のアドレスがいるがこれは引数の場所に書く。

(.,.)p – Print. 指定した範囲を端末に打ち出す。

(.,.)d – Delete. 指定した範囲を削除。

(.,.)m アドレス – Move. 指定した範囲を 3 番目のアドレスの次に移動。

9.2.5　置き換え

行の中を一部だけ変更する指令は s 指令しかない。これは sed の s と同様。

(.,.)s/パターン/置換/[g [p]] – Substitute. 行内の置き換え。

アドレスを範囲で指定した時はその各行について置換えを行なう。最後に g がつくと 1 行のなかで見つかる限り何回でも置き換える、というのは sed と同じ。p がつくと結果を打ち出す。これがないとどう置き換わったかも p 指令を使わないと見れないので不安である。

9.2.6　広域指令

あるパターンが見つかるすべての行について、これこれを実行、という言い方ができる。これはかなり強力である。

(1,$)g/パターン/指令 – Global. パターンが見つかった各行について指令を実行。

(1,$)v/パターン/指令 – ???. パターンが**見つからなかった**各行について指令を実行。

9.2.7　その他の指令

あと、使いやすさのために次のような機能がある。まず、ただの `[ret]` は「`.+1p`」と同じ。だから `[ret]` を繰り返し打ってずっとバッファの内容を見ていくことができる。また「アドレス `[ret]`」は「アドレス `p[ret]`」と同じ。そして、パターンが空だと最後に指定したパターンが再度利用される。

加えて次のものくらい知っていればとりあえず ed で遊ぶには十分であろう。

w [ファイル名] – バッファから元のファイルまたは指定したファイルに書き出す。

q – Ed を終了する。

9.2.8 Ed による編集の実行例

それでは簡単な編集の例を見てもらおう。

```
% cat sam.c
int main() {
  int x, y;
  scanf("%d %d", &x, &y);
  printf("x + y = %d\n", x + y); ← +でなく*にし、同様に/も入れたい。
}
% cp sam.c sam1.c
% ed sam1.c
86            ← ed はファイルを読むとそのバイト数を表示する。
/+/                                   ← +のある行へ行き、
  printf("x + y = %d\n", x + y);  ← アドレスのみだと p だから表示。
s//*/gp                               ← それを*に。
  printf("x * y = %d\n", x * y);  ← p がついてるので表示。
a                                     ← 入力モードに切替え、
  printf("x / y = %d\n", x / y);  ← /の行を打ち込む。
.                                     ← 入力モードおわり。
w             ← 書き出す。
119           ← 確かにバイト数が増えている。
q             ← おしまい。
%
```

だいたい、どんな感じか分かりますね?ちなみに自信があれば移動と置換えを一緒にして打ち出しもせず/+/s//*/g でもいいが、思わぬ所が見つかって悲惨な目に会うことも多いので。

9.2.9 バッチエディタとしての ed

さて、実は!ed は指令を標準入力から読むので、上の指令を端末から打ち込む代わりにファイルに入れておいてもいい。つまり、ed はバッチエディタとして使える!のである。例えば、上のをバッチでやるとする。

```
% cp sam.c sam2.c
% cat t.ed          ← ed script ファイル。さっき打ったのと同じ。
/+/
s//*/gp
a
  printf("x / y = %d\n", x / y);
.
w
q
% ed sam2.c <t.ed    ← これを標準入力から食べさせると。
86
  printf("x + y = %d\n", x + y);
  printf("x * y = %d\n", x * y);
119                                   ... 表示はさっきと同じ。
% cat sam2.c
int main() {         ← ちゃんと編集されてる。
  int x, y;
  scanf("%d %d", &x, &y);
  printf("x * y = %d\n", x * y);
  printf("x / y = %d\n", x / y);
}
%
```

もちろん、ed script を自動生成したりしてもいいわけだ!というわけで、ed の出番はシステムが壊れて画面エディタが使えなくなった時だけ、というわけではないのである。

10 Awk

10.1 Awk の位置付け

前回、シェルスクリプトと指令の組合せで仕事をするのは普通の言語でプログラムを書くのにくらべてどうか、という話がでた。さて、実はその両極端の中間として、C などの言語ほど汎用でない代わりに文字列処理などの機能が充実していてスクリプトよりはもう少し効率良く普通のプログラミングっぽく書ける、という選択肢がある。実は awk というのはそういう「言語処理系」である。その他にも perl、Icon など様々な文字列処理言語があるが、とりあえず awk が一番シェルスクリプトの線に近くてお手軽で簡単なので取り上げる。他の言語に興味がある方はそれなりに探求して見られたい。

さて、awk はとても簡単な言語なので多くの役に立つプログラムがたった 1 行で書けたりする。そういう場合には次のように指令行に直接プログラムを書いてしまえば良い。

```
awk 'プログラム' ファイル ...
```

しかし結構長いプログラムを書く場合もあるので、そういう時はプログラムをファイルに入れておき、

```
awk -f プログラムファイル ファイル ...
```

のようにして動かす。この辺は sed と同じである (もちろん`#!awk` を使ってもいい)。そしてファイルを指定しなければ標準入力から読むので、パイプラインの途中に使ってももちろんいい。

10.2 Awk の基本

Awk のプログラムは基本的に

```
  パターン { 文 ... }
  パターン { 文 ... }
...
```

という形をしている。その意味は、入力の各行がパターンにマッチした場合に対応する文の列を実行する、というものである。個々の文については後述するが、基本的に普通のプログラム言語のようなもので、例えば変数とか代入などはすべて使える。一方パターンには次のようなものがある。

- 空 – 空なパターンは、すべての入力行にマッチする。
- `/パターン/` – これまでにおなじみの、パターン。
- `BEGIN` – 「はじまりの前」。
- `END` – 「おわりの後」。
- その他の条件 – マニュアル参照のこと。

この `BEGIN` と `END` がなんとも面白いのだが、その意味は次のようなものである。そもそもファイル処理ではまず最初に何らかの初期設定があり、続いてファイルの各行について繰り返し処理があり、最後に後始末がある、というパターンを取るのが普通である。そこで `BEGIN` と `END` に対応してこれら初期設定と後始末を記述できるようになっているわけである。例えばファイルの各行に入っている数値を合計する、という例題は次のように書ける。

```
% cat t1
1
2
3
% awk 'BEGIN { n = 0; } { n += $0; } END { print n; }' t1
6
%
```

ちなみに変数$0 には入力の行が入る。あと、実は初期設定してない変数は空文字列かつ 0 と見なすので、BEGIN の方は不要である。

10.3　フィールドと特別な変数

Awk は入力の各行を自動的にフィールドに分解する。フィールドは普通は空白で区切られた空白でない文字の集まりだが、空白の替わりに別の文字フィールドの区切りに設定することもできる。分解された各フィールドはシェルの引き数のように、$1、$2、... のように書くことで参照できる。入力の行全体が$0 なのは既にのべた。その他、現在処理している行のフィールドの数は NF という変数に、そして現在の行が何行目かという情報は NR という変数に入っている。例えばこれを使えば前にやった「ls -l からファイル名とバイト数だけ持ってくる」というのは次のように素直に書ける。

```
% ls -l
total 4
drwxr-xr-x  2 kuno          512 Oct  3 16:05 bin
-rw-r--r--  1 kuno           52 Nov 10 12:52 hello.p
-rw-r--r--  1 kuno          115 Nov 10 12:52 t1.awk
drwxr-xr-x  2 kuno          512 Oct 29 22:18 work
% ls -l | awk 'NF == 8 { print $8, $4; }'
bin 512
hello.p 52
t1.awk 115
work 512
%
```

10.4　Awk の文

Awk の文には次のようなものがある。

```
  if ( 条件 ) 文 [ else 文 ]  ← 普通の if 文
  while ( 条件 ) 文          ← 普通の while
  for ( 式 ; 条件 ; 式 ) 文    ← C の for
  break                        ← while や for から抜ける
  continue                     ← while や for の次の周回へ進む
{ 文 ... }                    ← 複合文
変数 = 式                     ← 単一変数と配列とがある
print 式の並び             ← 適当な形式で出力
printf 書式文字列 , 式の並び  ← 書式文字列に従って出力
```

ただし awk の基本構造そのものが while と if を兼ねたようなものだから、あまり制御構造の出番は多くない。

printf に出てくる書式文字列というのは、要するに出力されてほしい文字列だが、ただしその中に%X というものが含まれていてもよく、もしあればその部分に対応する式の値が埋め込まれて出力される、というものである。X のところには出力したいものに対応する英字がくる。例えば

```
  d -- 整数
e -- 実数、指数形式
f -- 実数、小数点形式
s -- 文字列
```

のような具合である。さらに%の直後に数値を指定することで欄の幅を指定することもできる。その他詳しいことは man printf 参照。例えば先のを少し飾りをつけて欄の幅を揃えると次のようになる。

```
% ls -l | awk 'NF == 8 { printf "< %8s : %5d >\n", $8, $4; }'
<      bin :   512 >
<  hello.p :    52 >
<   t1.awk :   115 >
<     work :   512 >
% ls -l | awk 'NF == 8 { printf "< %-8s : %5d >\n", $8, $4; }'
< bin      :   512 >
< hello.p  :    52 >
< t1.awk   :   115 >
< work     :   512 >
%
```

ちなみに、"
n"というのは改行文字を表し、これがないと printf では改行は起きない。

10.5 Awk の条件

条件としてはつぎのものが指定できる。

```
式 関係演算子 式   ← 演算子は<,<=,==,!=,>,>=
  式 ~ /.../          ← パターンにマッチすれば成立
  式 !~ /.../         ← パターンにマッチしなければ成立
式 && 式      ← and
  式 || 式      ← or
  !式         ← not
```

この他にかっこが使用できる。パターンマッチの条件を利用するとシェルスクリプトの case の代わりに使えるのはお分かりだと思う。例えば前にやった Yes-No の例題だが:

```
% cat t1.awk
{ if( $0 ~ /Y|y|[Yy]es|[Oo]k/)
    print "You said yes.";
  else
    print "You didn't say yes."; }
% awk -f t1.awk
Yes
You said yes.
ok
You said yes.
oui
You didn't say yes.
^D
%
```

10.6 Awk の式

式は数値や文字列の値を計算するもので、次のようなものがある。

```
  定数          ←  整数、実数、文字列
変数名
変数名 [式]        ←  配列
関数名 (式,...)   ←  関数呼び出し
式 演算子 式   ←  +,-,*,/,%,空白,++,--,+=,-=,*=,/=,%=
```

ただし空白の演算子というのは文字列の連結を意味する。

関数を定義する機能が (この版では) ないので、関数は組み込み関数だけに決まっている。例えば次のようなものがある。

```
length(文字列)            ←  文字列の長さを返す
substr(文字列,位置,長さ)     ←  部分文字列を取り出す
sprintf(書式,式,...)   ←  printf と同様だが出力する代わり文字列を返す
```

例えば、前にやった継続行をつなげる、というのも awk では次のように書ける。

```
% cat t2.awk
/\\$/  { printf "%s", substr($0, 1, length($0)-1); }
!/\\$/ { print $0; }
% cat t
aaaaa bbbbbbb \
ccc \
ddd eee
fff ggg \
hhh iii
% awk -f t2.awk t
aaaaa bbbbbbb ccc ddd eee
fff ggg hhh iii
%
```

最後に、配列についてはその添え字は整数でなくてもなんでもよい。こういうのを連想配列といい、なかなか便利であるが、どんな添え字のところに値を入れたか分からないと意味がないのでそのため特別な for 文がある:

```
for(変数名 in 配列名) 文
```

これは指定した配列の添え字の値を順に変数に入れながら繰り返し文を実行するものである。これを使うと前にやった単語帳も次のようにできる。

```
% cat t3.awk
{ for(i = 1; i <= NF; ++i) a[$i] += 1; }
END { for(w in a) printf "%4d : %s\n", a[w], w; }
% cat t
this is a pen
that is a dog
that isn't a dog
% awk -f t3.awk t
   2 : dog
   2 : is
   1 : pen
   2 : that
   1 : isn't
   1 : this
   3 : a
%
```

10.7 Awk のまとめ

このように、awk は結構プログラミング言語らしい構造を持っているのだが、実際に使う時はそんなに大きいプログラムを書くというよりは、せいぜい数行のプログラムを書いたり、本当に 1 行のプログラムで他のフィルタと組み合わせたパイプラインの途中に書いたりする方がずっと多い。そういう意味でシェルと普通の言語の中間、という気分が分かっていただけたかと思う。

なお、ここで述べた awk は旧版の簡単なやつである (だから 1 回で説明してしまえるのだが)。「Unix プログラミング環境」で説明されているのもこれである。一方、最近の版は手続き呼び出しなども可能でずっとプログラム言語らしい仕様である。「プログラミング言語 awk」という本にはそちらの話が載っている。だからそっちの本を買われた方は旧版の awk で試して動かないといって悩まないように (gawk というのが使えれば、それは新版の方である)。

A 演習、課題

A.1 9 節の演習

9-1. ed を使って、20 行程度のプログラムを 1 から作ってみよ。ただし作業を始めるまえに、打ち込む指令の数、打ち込む行数、間違いの数など自分で決めたいくつかの項目をあらかじめ予想し、作業は script を取りながら行なう。最後に script を見てこれらの数値を実際に集計し、予想との違いについて考察する。集計は最初の打ち込み時、1 回目の修正、2 回目の修正、... のように分けるとよい。

9-2. ed を使って、次のような作業を行なってみよ。

- あるファイルの中に数回現れるある単語について、その単語を含む行をその前後の行とともにすべて打ち出す。[6][7]
- あるファイルの内容を上下さかさまにする。つまり、もとの 1 行目が最終行、2 行目が最終行の 1 つ前の行、... のようにする。

9-3. ed script を使って、次のような作業を自動的に行なわせてみよ。

- あるディレクトリにある Pascal プログラム全部について、そのプログラム本体の begin と最後の end を大文字になおす。
- 適当な Pascal プログラムを取り上げ、それに現れる begin のうち最初の 5 個に `begin {1}` のように順番に番号を振る。

9-4. ed script を何らかの方法で自動生成することにより、次のような作業を自動的に行なわせてみよ。

- あるディレクトリにある Pascal プログラム全部について、各ファイルの 1 行目にファイル名とプログラム名が入ったコメントを挿入する。
- 先に出た、`begin` に番号を振る、というのを最初の 5 個でなく全部に対して行なってみよ。

A.2 10 節の演習

この節の演習は当然ながら、「awk でこんなことをやってみよ」というのばかりです。

10-1. ファイルに含まれている文字の数、行の数、単語の数を数える。wc とおんなじ。結果が wc と同じになるかどうかチェックすること。

10-2. egrpe と同様に、ファイルに指定したパターンが含まれていればその行を打ち出す。awk を呼び出すシェルスクリプトにして、引数でパターンを指定できるようにする。できたら-n というオプションがあれば行を打ち出す時、それが何行目かも表示するようにする。

10-3. 継続行をくっつけるのと反対に、長い行があったら折り畳む。決まった長さでぶったぎるのでなく、見やすいように工夫すること。何が「見やすい」かは自分なりに考える。

10-4. 上の、「ed script を自動生成する」というのを awk でやる。

---

[6] この辺のものは大体広域指令の応用問題である。

[7] 適当なファイルが欲しければ`/usr/man/cat1/`の下にあるファイル (マニュアルページのファイル) などを利用するとよい。

A.3　7 回目の課題

この回の報告を出される場合には 9-1〜9-4 から 1 個以上、10-1〜10-4 から 1 個以上、なおかつ**合計して 3 個以上**選択して下さい。