本資料は、2014 年度までの奈良女子大学理学部 旧·情報科学科の授業「情報科学実験 1」のシェルスクリプトの回 (2 コマ ×4 週) で使用していたものです。従って、当時の同学科の計算機環境に依存する部分や、同環境からのみアクセスできるファイル·URL への言及などがあります。
また、執筆は基本的に同年度で終了していますが、以後も細部の訂補は行われることがあります。

情報科学実験 1・新出 担当分
—シェルとシェルスクリプト—

2014 年 4 月 8 日　　nide@ics.nara-wu.ac.jp

UNIX の**シェル**とは、ユーザからマシンへの指令の窓口役となるプログラム。例えばユーザが端末から「`ls`↵」と打ち込んでファイルの一覧を見ようとしているとき、ユーザからのキー入力「`ls`↵」を受け付けて、それが ls コマンドの起動を命じるものであることを理解し、実際に ls コマンドを起動してくれるプログラムが動いている (図 0.0)。それがシェルである。

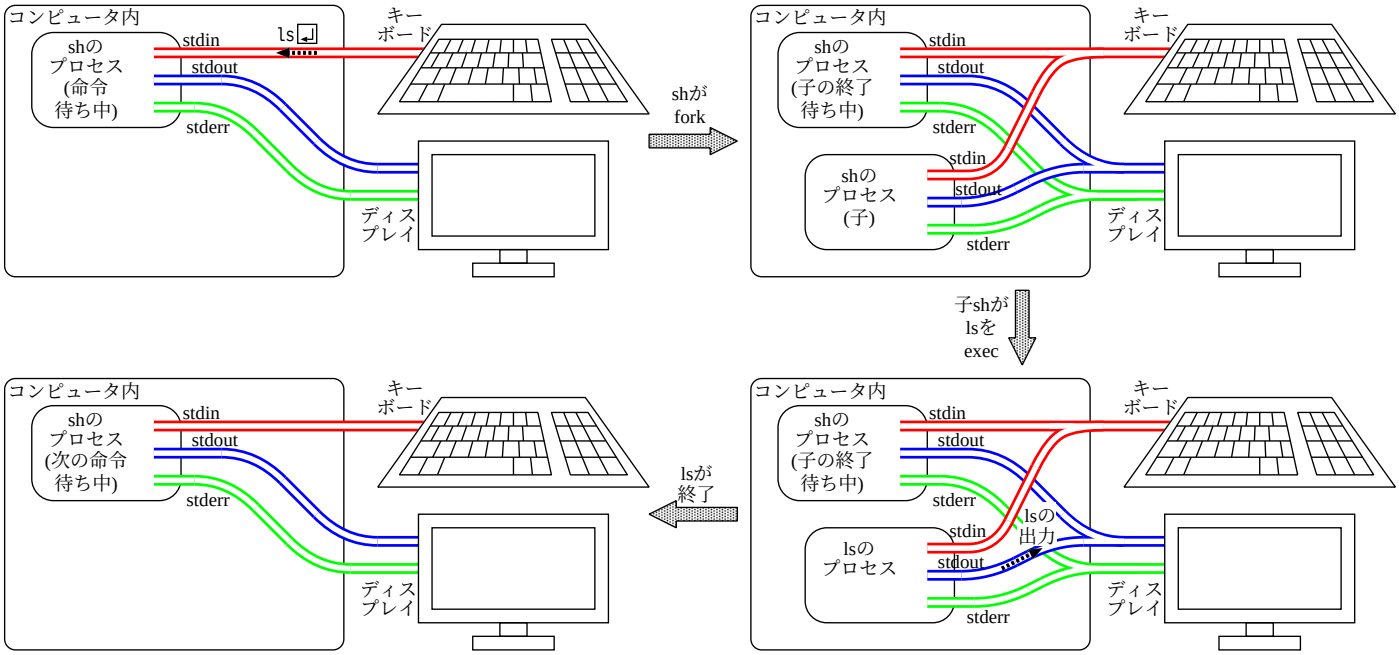


図 0.0: シェルの役割

シェルは**プログラミング言語**としても使える。シェルへのコマンドの連鎖をプログラムとして形成し、それをシェルに与えて実行させるわけである。いくつかのコマンドを連携して使うような作業を、**自動化**する手段として非常に有用である。

シェル言語によるプログラムは「**シェルスクリプト**」(1 章以降) と呼ばれる (「スクリプト」と略する[1]こともある)。本授業ではシェルスクリプトについて学ぶ。

> コンピュータの大きな利点の一つは、作業を**自動化**できることである。しかし、GUI (グラフィカル・ユーザ・インタフェース、マウスやウィンドウでソフトを操作する手法) ではそれが行いにくく (例えば「○○という操作を行い、次いで××という操作を行う。以上を 100 回繰り返す」というような作業を自動化するのは GUI では極めて困難である)、せっかくのコンピュータの利点を削いでしまう。これに対し、シェルスクリプトに代表される CUI (コマンドライン・ユーザ・インタフェース、キーボードからコマンドでソフトを操作する方法) ではそうしたことが容易にできる。コンピュータを学ぶにあたって、GUI だけでなく CUI も学ぶべきであるのは、そうした理由による。
>
> ただしそのためには、CUI の能力を生かすように設計された、優れた道具が必要である。Windows にはそれがあるとは言い難い[2]。こうしたことも、UNIX を学ぶべきである理由の 1 つである。

シェルは、B シェル系と C シェル系に大別される。B シェル系のシェルには sh, ksh, zsh, bash など、C シェル系のシェルには csh, tcsh などがある。両者には共通点も多いが、違いもある。また、使われ方の傾向に

[1]「**スクリプト言語**」と呼ばれる一群の言語があり (2 章)、その言語によるプログラムを一般に「スクリプト」と呼ぶ。例えばシェル言語のプログラムを「シェルスクリプト」、AWK 言語のプログラムを「AWK スクリプト」と呼ぶなど。しかし、文脈からシェルスクリプトのことを指すと明らかにわかる場合は、シェルスクリプトを単に「スクリプト」と呼ぶこともある。

[2]UNIX のシェルコマンド群を Windows に移植したものもなくはないのだが、主流とはなり得ていない。

も、下記のように若干の差がある。

- 対話的に (i.e. コマンドを端末から打ち込みながら) 使う場合のシェル
  B シェル系/C シェル系どちらも広く使われている。
- シェルスクリプトを利用する場合のシェル
  C シェル系はこの目的で使うにはかなり大きな欠点があるので[3]、B シェル系を使うのが普通。

この両者は統一できる方が好ましいという理由で、対話的に使うシェルも B シェル系にしている人も多い。本学 G 棟計算機システム (以下「G 棟システム」) のマシンでも、対話的に使うシェルとしては**B シェル系**の**bash** を標準としている[4]。

本資料 0 章は B シェル系/C シェル系共通。1 章以降は**B シェル系**を前提に書かれている。本資料は計 88 ページである。

目 次

[3]参考:「Csh Programming Considered Harmful」という有名な文書がネットで出回っている。日本語訳もあるがそれは古いので、既に解決済みの項目が問題点として取り上げられている箇所もある。

[4]2007 年度より。2006 年度までの旧システムでは C シェル系の tcsh を使っていた。

第 0 章 シェルとコマンドの基本的な利用法

本章はシェルスクリプトの話というわけではなく、UNIX でコマンドとシェルを使うために知っておきたい基本的知識を述べる。

0.1 基本コマンドの説明

UNIX の基本コマンド群は、シェルのコマンドラインから単独で、あるいは 0.2 節で述べるリダイレクトやパイプと組み合わせて使うだけでなく、シェルスクリプトにおいても基本となるものばかりである。ここでは概略しか述べないので、詳細は man コマンドなどで確認のこと。

0.1.1 ファイル操作関係のコマンド

表 0.1 は、ファイル操作に関する代表的なコマンド。ファイルの中身を見たり、ファイルのコピー・削除・改名を行ったりする[5]。

0.1.2 フィルタ類

表 0.2 は、**フィルタ** (→0.2.3 節) と呼ばれるコマンドの代表的なもの。これらは、標準入力 (ただし、引数にファイル名が指定されていればそのファイルの内容) を読んで所定の処理をし、標準出力に出す。ただし tr は例外で、ファイル名を引数に指定できず、常に標準入力を読む。(注意: 表中の「␣」は**空白文字**を表す。そこに空白文字があることを明示するために、このように表すことがある。**以後の節でも同様**。)

[5]表中の tar+gzip については `http://www.ics.nara-wu.ac.jp/ics-only/pdf/file-reduction.pdf` (本学学内からのみ閲覧可) も参照。

表 0.1: ファイル操作関係の基本コマンド

コマンド	説明
cat	引数で指定したファイルの中身を表示 (標準出力へ出力)。
less	引数で指定したファイルの中身を 1 画面ずつ表示。f キー (またはスペース) と b キーで画面を上下、q キーで終了。
cp	ファイルを他のファイルまたはディレクトリへコピー。
mv	ファイルを改名、または他のディレクトリへ移動。
rm	ファイルを削除。`rm -r` でディレクトリ以下の全削除可能。
ls	ファイルのリストアップ。`ls -a` で「ドットファイル」もリスト、`ls -l` で詳しいリスト。
ln	ファイルのリンク (別名付け)。シンボリックリンク (`ln -s` で可能) と通常のリンク (「ハードリンク」と呼ぶ) の違いに注意 (本資料では略)。
chmod	ファイルのモード変更。実行可能にしたり、他人から読めなくしたりする。
touch	ファイルのタイムスタンプ (更新日時など) を変更。デフォルトでは、ファイルの更新時刻を現在時刻にする。touch は、指定したファイルがなければ作成するので、空ファイルの作成用途にも使える。
tar, gzip	ファイルのアーカイブと圧縮を行う。詳細略。
od	ファイルのバイナリダンプ。バイナリファイルの中身を見るのに多用。`od -c` が便利。
lpr	指定したファイルをプリンタに送る。プリンタが PostScript プリンタの場合、プリンタには原則として PostScript データを送らねばならない。
mkdir	ディレクトリを作成。
rmdir	ディレクトリを削除。空のディレクトリでないと削除できない。空でないディレクトリとそこ以下のファイル全てを削除するには、`rm -r` を使う。
cd	カレントディレクトリを変更。
pwd	カレントディレクトリを表示。

表 0.2: フィルタとして用いられる基本コマンド

コマンド	説明
sort	ファイルの中身を行毎にソートして出力。比較は文字列として行われる。数値として比較するなら `sort -n` を使用。
uniq	ファイル内の隣接行が同じなら 1 つにまとめ、出力。`uniq -c` だと、何行を 1 つにまとめたかも併せて出力。
wc	ファイルの行数、単語数、文字数を数える。`wc -l` だと行数だけ。
fgrep	ファイル内の、指定した文字列を含む行を探し出して出力。例えば「`fgrep␣hello␣`ファイル名↵」で、hello という文字列を含む行だけ出力。
egrep	fgrep に似ているが、文字列を「正規表現」(詳しくは A.1 節参照) で指定可。例えば「`egrep␣'ab|c.*d'␣`ファイル名↵」で、ab を含むか、または c で始まり d で終わる文字列を含む行を出力。grep というコマンドもあり、egrep とほぼ同じだが正規表現の書き方が微妙に違う。
head	ファイルの先頭の指定した行数ぶん (指定がなければ 10 行) を出力。例えば「`head␣-8␣`ファイル名↵」で先頭 8 行を出力。
tail	ファイルの末尾の指定した行数ぶん (指定がなければ 10 行) を出力。例えば「`tail␣-8␣`ファイル名↵」で末尾 8 行を出力。「`tail␣-n␣+8␣`ファイル名↵」だと先頭から 8 行目以降を出力
tr	文字の置換。例えば「`tr apx bqy`↵」で標準入力の a を全て b に、p を全て q に、x を全て y に置き換えて出力。
sed, awk	入力に対し、文字列の置換などさまざまな加工を施して出力。詳しくは参考書籍などを。awk の簡単な例は 2 章で扱う。
nkf	文字コードの変換 (本資料では略)。システムによっては別途インストールが必要。

表 0.1 のうちの cat, less, lpr, od, gzip もフィルタの仲間で、ファイル名の指定がなければ標準入力を読む (ただし lpr は標準出力への出力はしない)。

0.1.3 その他の基本コマンド群

その他によく使うコマンド群を表 0.3 に挙げる。

表 0.3: その他の基本コマンド

コマンド	説明
echo	引数に与えたメッセージを単に出力。シェルスクリプトで多用。
sleep	引数に与えた秒数だけ停止。シェルスクリプトで多用。
find	指定したディレクトリ下を再帰的に探し、その下にあるファイル名を出力。条件を指定し、その条件に合うファイル名だけ出力させることもできる。詳しくは A.4 節参照。
xargs	ファイル名 (空白あるいは改行で区切られているものとする) を標準入力から読み、その各ファイルに対し、指定したコマンドを実行。ただし、できるだけコマンドの起動回数 (プロセス数) を少なくするよう、可能な限りファイル名をまとめてコマンドに渡す。
expr	整数の演算や、引数に与えた文字列から正規表現 (→A.1) による切り出しを行うなどの処理をする。例えば「`expr␣1␣+␣3`↵」で 4 を出力、「`expr␣文字列␣:␣'\(..\)'`↵」で文字列の先頭 2 文字を出力。詳しくは A.3 節参照。
test	条件の判定。数値や文字列の比較、ファイルの存在の判定などが行える。詳しくは A.2 節参照。簡単な例は本資料で扱う。条件が真かどうかをコマンドの「戻り値」(→1.3.1) に返すだけで、画面 (標準出力) には何も出力しないので注意が必要。
printf	C 言語の printf 関数に似たことを行えるコマンド。数や文字列を桁揃えして出力する場合などに使う。
file	引数に指定したファイルの内容が何であるかを推測して出力。

0.2 シェルの便利な使い方

コマンド入力が効率よく行えれば、コマンドラインインタフェースはぐっと快適になる。そのために知っておくべきことをここに挙げる。

0.2.1 ファイル名補完・コマンドヒストリ・コマンド行編集

古いシェルにはこれができないものもある[6]。また、シェルの種類によっては (あるいは設定によっても) 操作のキーが異なることもある。

• ***ファイル名補完*** コマンドの引数に長いファイル名を打ち込みたい場合、途中まで打ち込んで Tab キーを押すと補完してくれる。

候補が複数の場合は、一意に決まるところまでしか補完できない。その場合、自動的に候補一覧が表示されるので、さらに何文字か打ち込んで再度補完させる。(候補一覧を自動表示するには所定の設定が必要だが、G 棟システムではこの設定は既に済んでいる)

• ***コマンドヒストリ*** 上向き矢印キー (あるいは Ctrl-P) で、直前に実行したコマンド行を呼び出せる。戻り過ぎたら下向き矢印キー (あるいは Ctrl-N)。

[6]新しいシェルであっても、人間との対話よりシェルスクリプトの実行効率を重視して軽量化してあるものでは、これができないことがある。

• ***コマンド行編集***　リターン (Enter) キーを押す前なら、コマンド行の編集ができる。左/右向き矢印キー (あるいは Ctrl-B/F) でカーソルを動かし、Delete キーで字を削除し、普通の文字キーで字を挿入。リターンキーで実行 (カーソルを右端に戻す必要はない)。コマンドヒストリと併用でさらに便利。

0.2.2　ワイルドカード

コマンドの引数に、いくつものファイル名を一度に指定する方法[7]。以下のものがある (抜粋)。

記法	意味
*	任意の文字列を表す
?	任意の 1 文字を表す

記法	意味
[abc]	a, b, c のうちどれか 1 文字を表す
[!abc]	a, b, c **以外**のどれか 1 文字を表す[8]
[a-f]	a, b, ···, f のうちどれか 1 文字を表す

コマンドの引数にワイルドカードがあると、まずワイルドカードが、それにマッチする一群のファイル名に展開され、**それから**コマンドが実行される。このとき、ワイルドカードの展開結果は**文字コード順** (アルファベット順) に並ぶ。

例えば、「abcd.c」「abcde.c」「abce.c」「abcf.c」という名の 4 つのファイルがあるとき

```
$ cat abc*.c↵
```
(先頭の「$」はプロンプトを表すので、実際には打ち込まないこと。以下も同様)

とすると、まず「abc*.c」の部分が「abcd.c」「abcde.c」「abce.c」「abcf.c」の 4 つに展開される (**アルファベット順**なので、必ずこの順番になる) ため、「cat abcd.c abcde.c abce.c abcf.c↵」としたのと同じことになり、その結果、この 4 つのファイル全ての内容が出力される (図 0.1)。また、

```
$ cat abc?.c↵
```

で abcd.c, abce.c, abcf.c の内容が出力され、

```
$ cat abc[df].c↵
```

では、abcd.c と abcf.c の内容が出力される。

ワイルドカードの実験には echo コマンドが便利。例えば

```
$ echo abc*.c↵
```

で、「abc」で始まり「.c」で終わる名前のファイルのファイル名を全て表示するので、ワイルドカードがどう展開されるかわかりやすい。

> ただし、B シェル系の場合、ワイルドカードにマッチするファイル名が 1 つもなければ、ワイルドカードは展開されない。例えば「echo abc*.c↵」は、「abc」で始まり「.c」で終わる名前のファイルが 1 つもなければ単に「abc*.c」を表示する。
>
> C シェル系だと、ワイルドカードにマッチするファイル名が 1 つもない場合はエラーになる。

なお、ファイル名の**先頭**の「.」はワイルドカードの展開結果に含まれない、という特例がある。例えば「.abc」および「a.bc」という名前の 2 つのファイルがあるとき、

```
$ echo *bc↵
```

は「a.bc」のみ表示し、「.abc」は表示されない。しかし

```
$ echo .a*↵
```

なら「.abc」が表示される (先頭の「.」がワイルドカードではなく直接指定されているからである)。

---

[7] ワイルドカードは A.1 節の「正規表現」と見かけが少し似てはいるが、**全く違う**ものなので注意。例えば「任意の文字列」をワイルドカードでは「*」、正規表現では「.*」と表すし、「数字ばかりからなる文字列」は正規表現では書けるがワイルドカードでは書けない。

[8] 正規表現では同じ意味のことを「[^abc]」と書くが、ワイルドカードでは「[!abc]」と書くことに注意。ただし、ワイルドカードでも「[^abc]」とも書けるシステムもある (というより、そう書けるシステムがほとんど)。

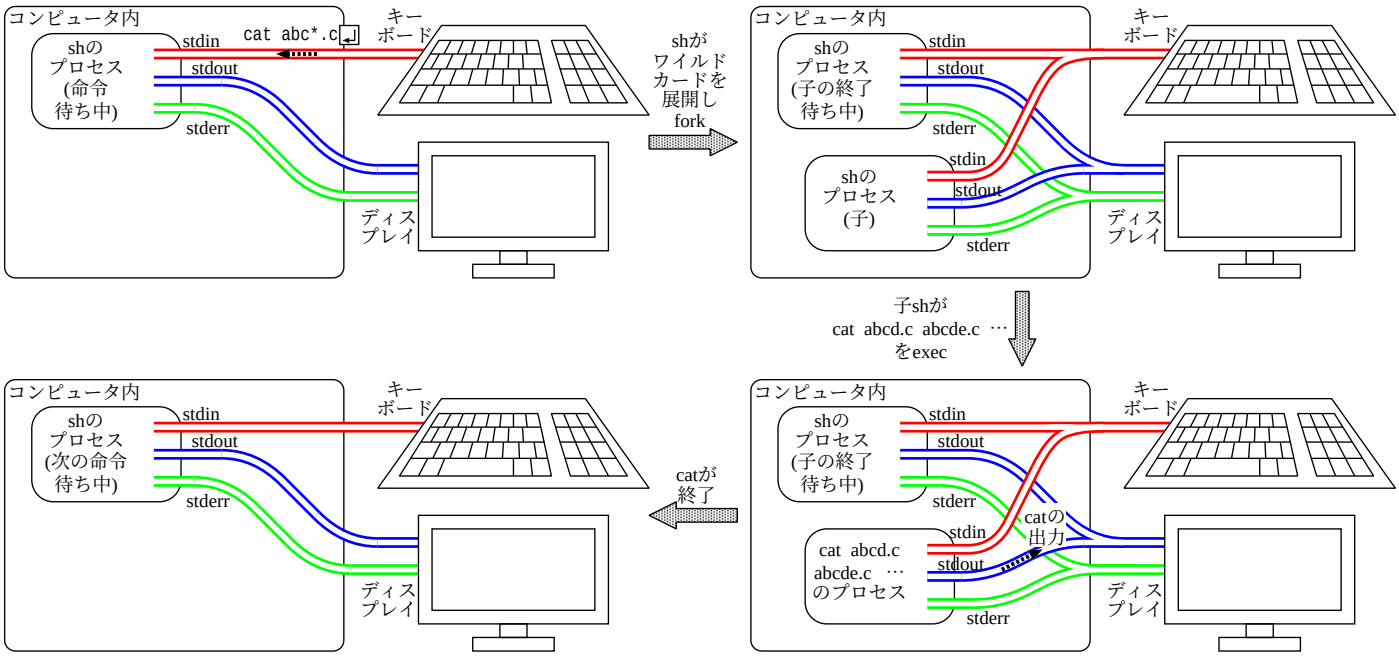


図 0.1: ワイルドカードの展開のようす

[注意点 1] ワイルドカードで指定できるのは**既存の**ファイルの名前だけ。例えば、ufj, ufo, usj の 3 つの空ディレクトリがあって、それぞれの中にさらに new というディレクトリを新規に作りたいとき、

```
$ mkdir u??/new
```

とはできない (ufj/new, ufo/new, usj/new という名前のファイルがまだないので、これらのファイル名には展開できない)。

[注意点 2] ワイルドカードはシェルによって展開され、コマンドには展開済みのファイル名が渡される。従って、例えば

a.c　a.c.old　b.c　b.c.old　c.c　c.c.old

という 6 つのファイルがあったとして、a.c を a.c.old に、b.c を b.c.old に、c.c を c.c.old に mv しようとして

```
$ mv ?.c ?.c.old
```

とはできない。なぜなら、引数「`?.c`」「`?.c.old`」は mv コマンドに渡される前に展開され、mv コマンドには「`a.c`」「`b.c`」「`c.c`」「`a.c.old`」「`b.c.old`」「`c.c.old`」の 6 つの引数が渡される。従って「`mv a.c b.c c.c a.c.old b.c.old c.c.old`」と操作したのと同じことになり、mv コマンドの使い方に合わないからである。

ワイルドカードを展開しているのはシェルである、ということを実感してみよう。以下の C プログラム argvecho.c を作り、

```
#include <stdio.h>
int main(int argc, char **argv)
{
        int i;

        for(i = 0; i < argc; i++) printf("argv[%d] = %s\n", i, argv[i]);
        return 0;
}
```

そして

```
$ cc argvecho.c -o argvecho↵
$ ./argvecho *↵
```

のようにコンパイル·実行すると、

```
argv[0] = ./argvecho
argv[1] = argvecho
argv[2] = argvecho.c
  ⋮
```

のような出力が得られる。プログラム中ではワイルドカードの処理を一切していないのに、argvecho コマンドにはワイルドカードを展開した結果が引数として渡されていることがわかるだろう。これは、シェルがワイルドカードを展開してからそれを引数として argvecho コマンドを起動しているからである。

一方、argvecho をコンパイルしたのと同じディレクトリに下記左の C プログラム execit.c を作り、

```
#include <unistd.h>
#include <stdio.h>
int main(void)
{
        char    *cmd = "argvecho";
        execl(cmd, "wahaha", "*", NULL);
        perror(cmd);
        return 0;
}
```

そして

```
$ cc execit.c -o execit↵
$ ./execit↵
```

とすると

```
argv[0] = wahaha
argv[1] = *
```

という出力を得る。

この例では execit コマンドが argvecho コマンドを起動しているのだが、引数のワイルドカードは展開されずに argvecho コマンドに渡されている。argvecho コマンド自体にはワイルドカードの展開機能はないこと、シェルから実行しなければワイルドカードは展開されないことがこれでわかる。

ワイルドカードがシェルによって展開されるということは、(シェルから起動する限り) 自分で作ったコマンドも含め、**どんな**コマンドでもワイルドカードは利用可能、ということである。0.2.3 に出てくるリダイレクトやパイプが任意のコマンドで利用可能なのも、やはりシェルが行ってくれることだからである。

0.2.3　リダイレクト・パイプ

標準出力の出力先 (普段は端末画面) や標準入力の入力元 (普段はキーボード) をファイルに切り替えたり、あるコマンドの標準出力を他のコマンドの標準入力に直接渡したりする。

コマンド実行時の指定	機能
$ コマンド > 出力先ファイル↵	標準出力のリダイレクト (上書き)
$ コマンド >> 出力先ファイル↵	標準出力のリダイレクト (追加書き)
$ コマンド < 入力元ファイル↵	標準入力のリダイレクト
$ コマンド | 別なコマンド↵	パイプ。前段のコマンドの標準出力を別な (後段の) コマンドに渡す

例えば

```
$ date > outfile↵
```

としたとき、date コマンドの出力は端末画面ではなくファイル outfile に書き込まれる (図 0.2)。ファイル outfile が既存だった場合、「>」だと上書きだが、「>>」にすると追加書きになる。

「tr a b」というコマンドは、標準入力からの入力のうち文字 a を b に変換して (それ以外はそのまま) 標準出力に出すものだが、

```
$ tr a b < outfile↵
```

とすると、outfile の内容を読みながら文字 a を b に変換して標準出力に出す。

また、cal 2014 というコマンドの出力は (端末エミュレータを縦にでかくしない限り) 1 画面に収まり切らないが、

```
$ cal 2014 | head -6↵
```

で、cal 2014 というコマンドの出力の先頭 6 行だけを見たり、

```
$ cal 2014 | less↵
```

で、cal 2014 というコマンドの出力を 1 画面毎に停止させて見たりできる (less コマンドを抜けるには q キー! お忘れなく)。

さらに、上記の組み合わせ使用も可。

```
$ コマンド < 入力元ファイル > 出力先ファイル↵
$ コマンド | 別なコマンド > 出力先ファイル↵
```

など。

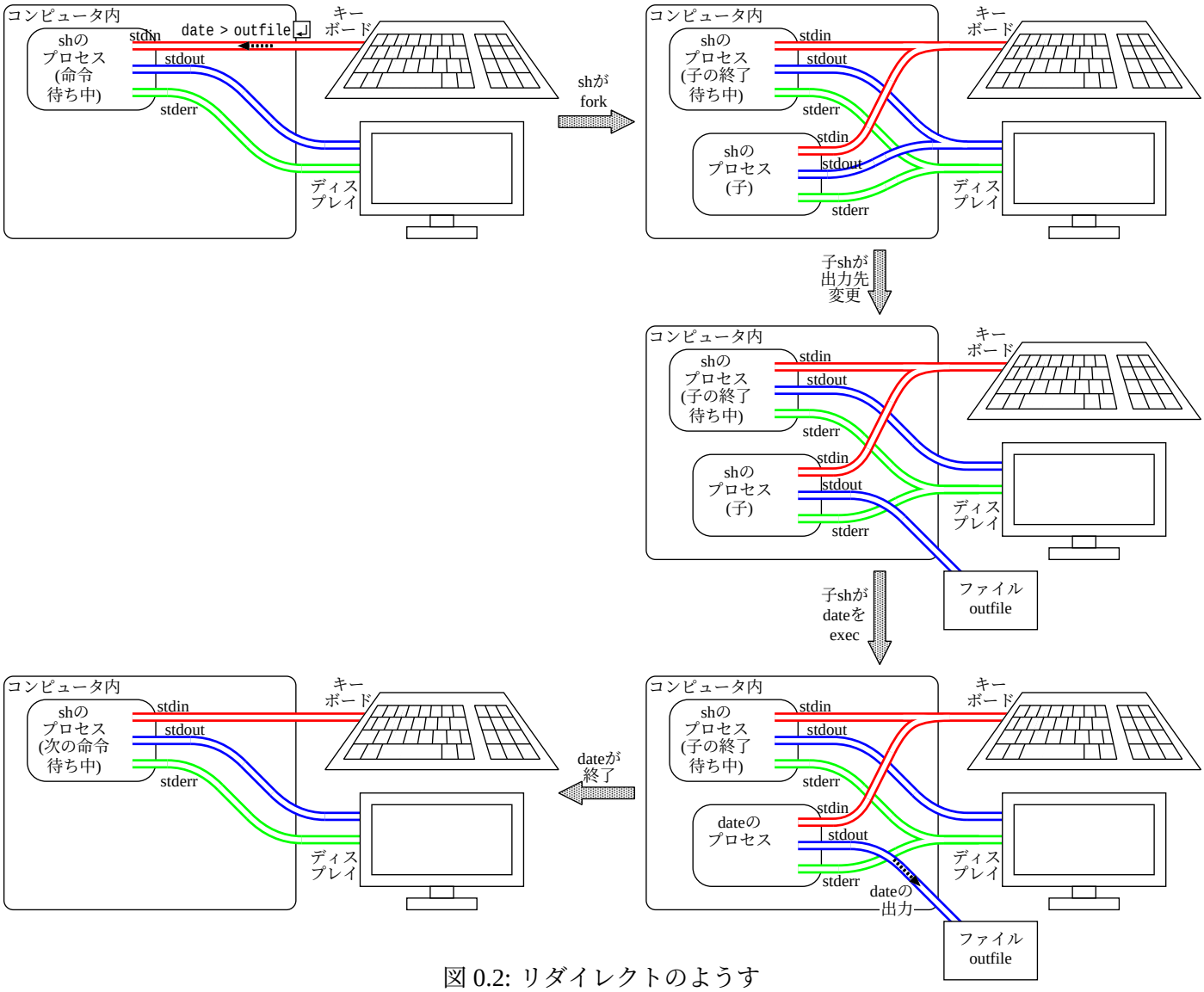


図 0.2: リダイレクトのようす

リダイレクトやパイプの処理はシェルが行ってくれる。従って、(シェルから起動する限り) 自分で作ったコマンドも含め、**どんな**コマンドでもリダイレクトやパイプは利用可能。特に、パイプで他のコマンドとつないで使われることを意識して作られているコマンドを「**フィルタ**」と呼ぶことがある。

0.2.3.1 標準エラー出力のリダイレクト

この節 (0.2.3.1) は B シェル系だけの話。

B シェル系では、標準エラー出力 (C 言語でいう stderr) をリダイレクトするには

```
$ コマンド 2> 出力先ファイル↵
```

とする。例えば「`$ コマンド 2> /dev/null↵`」で標準エラー出力を捨てられる (/dev/null とは特殊なファイルで、そこへ書き込んだ内容は黙って捨てられる)。

```
$ コマンド > 出力先ファイルA 2> 出力先ファイルB↵
```

のように取り分けることもでき、また「`2>&1`」で標準エラー出力を標準出力と同じところにリダイレクトできるので

```
$ コマンド > 出力先ファイル 2>&1↵
```

のようにすると、まず標準出力をファイルにリダイレクトしてから、標準エラー出力を標準出力と同じところにリダイレクトするため、結局両者を同じファイルへリダイレクトできる。

> この書き方 (「2>」) と区別するため、リダイレクトやパイプしたいコマンドの最後の引数が数字だけからなる場合、リダイレクト記号やパイプ記号の直前に空白を開けねばならない。例えば
>
> ```
> $ echo a b c > aaa↵
> ```
>
> の場合、> の前に空白なしで「`echo a b c> aaa↵`」でも OK だが、
>
> ```
> $ echo a b 34 > aaa↵
> ```
>
> の場合、> の前に空白なしで「`echo a b 34> aaa↵`」とすることはできない。なお、いずれの場合もリダイレクト記号の直後の空白はあってもなくても可。

さらに、

```
$ コマンド >&2↵
```

で、コマンドの標準出力を標準エラー出力へリダイレクトすることができる。シェルスクリプトで、メッセージを標準エラー出力に出すためによく使う。

> C シェル系では、「コマンド `>&` 出力先ファイル↵」で標準出力も標準エラー出力も同じファイルへリダイレクトすることはできるが、両者を別々にリダイレクトしたり、標準エラー出力だけリダイレクトしたりはできない。また、標準出力を標準エラー出力へリダイレクトすることもできない。これが、シェルスクリプトに C シェル系を使わない理由の 1 つ。特に、後者ができないのは痛い。

0.2.3.2　原則と原則破り

例えば

```
$ cal 2014↵
```

とするのと

```
$ cal 2014 > outfile↵
$ cat outfile↵
```

とするのとでは、結果的に画面に出てくるものは全く同じである。「`cal 2014`」の動作は、標準出力に 2014 年のカレンダーを出すことであり、標準出力が実際にどこにつながっているかは cal コマンド側では全く関知しない。従って、標準出力がリダイレクトされていてもいなくても、出力されるものは全く**変わらない**。出力が実際に出て行く先が違うだけ。

このように、標準出力のリダイレクトは、コマンドの出力自体には**全く影響を与えない**のが大原則である。標準入力についても同様。

しかし、**例外**もまれにある。顕著な例外は ls コマンド。

ls コマンドは (オプション引数を特に指定しなければ)、標準出力が端末の場合は 1 行につきできるだけ多くのファイル名を詰め込んで出力し[9]、標準出力が端末でない場合は 1 行につき 1 つだけのファイル名を出力する。従って

[9] 大昔の UNIX ではそうでないことがある。

```
$ ls↵
```

と

```
$ ls > outfile↵
$ cat outfile↵
```

では出力が異なる。

> ls は、まず標準出力が端末につながっているかどうかを調べ、それによって出力の形式を切り替えているのである (ただし `ls -la` のような場合は、もともと常に 1 行 1 ファイルで出力するので、標準出力が端末であってもなくても出力は変わらない)。
> ls がこのように作られている理由は、出力が端末の場合には出力をできるだけコンパクトにした方が見やすく、一方、出力がパイプなどの場合には出力が 1 行 1 ファイルになっている方が後続の処理がしやすいからである。例えば「`ls | wc -l`↵」でファイルの個数を数えることができる。

でも、このようなものは**あくまで例外**。標準出力をリダイレクトしてもしなくても、標準出力に出ていく出力自体は全く変わらないのが**原則**、と認識しておこう。

0.2.4 コマンドの逐次実行

「;」で区切って、1 つの行にコマンドをいくつも書ける。次はその例。

```
$ date; who↵
```

0.2.5 バックグラウンド実行

```
$ コマンド &↵
```

で、コマンドを**バックグラウンドで実行**できる。例えば、「`emacs` *filename* `&`↵」で Emacs を起動すると、Emacs の終了を待たずにシェルのプロンプトが出て、次のコマンドを入力できる。

古いシェルを除き、既に動いているコマンドを

Ctrl-Z

で一旦停止し、

```
$ bg↵
```

で**バックグラウンドに回す**こともできる。例えば、Emacs を起動する際に「`&`」を忘れてしまって「`emacs` *filename*↵」で起動した場合、そのままではシェルは Emacs の終了までプロンプトを出さず、次のコマンドが入力できない。ここで、端末に (Emacs にではない) Ctrl-Z を入力し「`bg`↵」とすると、Emacs を終了させずにバックグラウンドに回せる。その結果、シェルは Emacs の終了を待たずにプロンプトを出し、次のコマンドの入力を受け付ける。

> (注!) Ctrl-Z はコマンドの一時停止であって**終了ではない**! Emacs を Ctrl-Z で止めて、終了させたつもりになって放置し、新たな Emacs を起こす人をたまに見るが、これは Emacs を多重起動することになりシステムに不要な負荷をかけるので、やってはいけない。Emacs 以外のコマンドについても同じで、Ctrl-Z は終了ではない。

コマンドをバックグラウンドに回したままログアウトすることもできる。この場合、ログアウト後もそのコマンドは**動き続ける** (ただしシステムによっては、nohup というコマンドを使わないと、バックグラウンドのコマンドがログアウト時に自動的に終了させられてしまうものもある)。

ただし、マシンを**シャットダウン**してしまうとバックグラウンドのコマンドも当然終了してしまう。

> (注) 他の人と共用で使うシステム (端末室のマシンや remote01 など) で、バックグラウンドでプログラムを動かしっぱなしにすると、CPU パワーを消費して他のユーザに迷惑をかけうることに注意。特に、CPU パワーを多量に消費するプログラムをバックグラウンドで動かす場合は、他の人に迷惑をなるべくかけないよう、nice コマンド (本資料では説明していない) で優先度を下げること。

0.3　ファイルの属性について

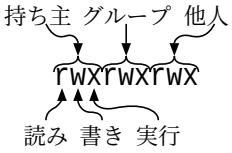
UNIX では、各ファイルにはそのファイルの**持ち主**が誰か、および**所属グループ**がどれか、という情報がつく。そして、(1) 持ち主 (2) 持ち主以外で所属グループに属する人 (3) その他の人、の 3 通りについて、そのファイルを (i) 読み出せるか (ii) 書き換えられるか (iii) プログラムとして実行 (あるいは、ディレクトリの場合はそこへ移動) できるか、という 3 種類 (合計 $3 \times 3 = 9$ 種類) の属性がつく。

0.3.1　ファイルの属性の表示

ファイルの属性は ls コマンドの `-l` オプションでわかる。例えば「`ls -l abc.c`↵」とすると

```
-rw-r--r--    1 nide     ics          10843 Jun 19 15:06 abc.c
```

のように表示されるが、一番左の欄の 10 文字のうち一番左の 1 文字を除いた残り 9 文字が 9 種類の属性を表し、



のように対応している (該当する許可がないものは「`-`」が表示される)。上の例では、abc.c は

- ファイルの持ち主 (nide) は読み出すことも書き換えることもできるが実行は不可 (`rw-`)
- 所属グループ (ics) に属する人は読み出すことはできるが書き換えも実行も不可 (`r--`)
- その他の人は読み出すことはできるが書き換えも実行も不可 (`r--`)

ということになる。

なお、一番左の欄の 10 文字のうちの一番左の 1 文字はそのファイルの種別 (普通のファイルなら「`-`」、ディレクトリなら「`d`」、シンボリックリンクなら「`l`」、その他の種別も若干ある) である。

> 「abc.c は C 言語のプログラムのファイルのはずなのに、それに実行可能属性が付いていないんじゃ、実行できないのでは?」と心配する必要はない。システムが実行するのは、コンパイラが生成した機械語プログラムの方であり、C プログラムのファイルそのものを実行するのではないので、C プログラムのファイルに実行可能属性が付いている必要はないのである。試しに、コンパイラが生成した実行可能プログラムの属性を `ls -l` で見てみれば、ちゃんと実行可能属性が付いているだろう。

「`ls -l *.c`↵」で「.c」で終わる名のファイル全部に関する情報を表示させたり、「`ls -la`↵」でカレントディレクトリ (現在いるディレクトリ) にある全てのファイル (カレントディレクトリ自体も含む) に関する情報を表示させたりすることもできる。

ファイルではなくディレクトリに関する情報を表示したい場合、単に「`ls -l` ディレクトリ名↵」では、そのディレクトリの中にあるファイルも含めて表示されてしまう。そのディレクトリに関する情報だけ表示させたい場合は、「`ls -ld` ディレクトリ名↵」としよう。例えば、カレントディレクトリに bin というディレクトリがある場合、「`ls -ld bin`↵」で

```
drwxr-xr-x    3 nide     ics          8192 Mar 28 12:57 bin
```

のような情報が得られる。ここでのファイル属性の「`x`」は、対象がディレクトリなので「そのディレクトリに (cd コマンドで) 移動できる」という権限を表す。よってこの例では、bin ディレクトリは

- 持ち主 (nide) は読み出すことも書き換えることもそこへ移ることもできる (`rwx`)
- 所属グループ (ics) に属する人は読み出すこととそこへ移ることができるが書き換えは不可 (`r-x`)

- その他の人は読み出すこととそこへ移ることができるが書き換えは不可 (r-x)

ということになる。ここで、ディレクトリを「読み出すことができる」とは、そのディレクトリの中にあるファイルを ls でリストアップできること、「書き換えることができる」とは、そのディレクトリの中にファイルを新たに作ったり、そのディレクトリの中のファイルを消したりすることができることを意味する。

0.3.2 ファイルの属性の変更

ファイルの属性変更は chmod コマンドで行える。使い方の例を挙げる。

```
chmod go-r abc.c          ···abc.c が自分以外から読めないようにする
chmod u-w abc.c           ···abc.c を自分が書き換えられないようにする
chmod a+x def             ···def を誰でも実行可能にする
```

このように chmod は、第 1 引数に属性の変更の指定、第 2 引数 (以降) にファイル名を与える。もちろん、持ち主が自分であるファイル (またはシステムの管理者) しか属性の変更はできない。

属性の変更の指定は

- 第 1 パート: 誰に対する指定かを u (持ち主—つまり自分), g (ファイルの所属グループに属するユーザ), o (その他の人), a (全員) またはその組み合わせで指定
- 第 2 パート: - (不許可にする), + (許可にする) のいずれかで指定 (「=」というのもあるが略)
- 第 3 パート: r (読み出しに対する許可/不許可を変更), w (書き換えに対する許可/不許可を変更), x (実行に対する許可/不許可を変更) またはその組み合わせで指定 (他にも若干あるが略)

をこの順に、空白を置かずにつなげて行う。よって、「go-r」は自分以外の読み出しを不許可にすること、「u-w」は自分の書き換えを不許可にすること (そのファイルが自分で書き換えられなくなる。ただし、chmod で再度自分による書き換えを許可にすれば書き換えられるようになる)、「a+x」は全員に対して実行を許可することを表す (この他に、変更後の属性を **8 進数**で直接指定する方法もあるが略)。

変更後は、念のため ls -l で変更後の属性を確認しておくとよいかもしれない。下記は実行例。

```
$ ls -l abc.c↵
-rw-r--r--    1 nide     ics         10843 Jun 19 15:06 abc.c
$ chmod go-r abc.c↵
$ ls -l abc.c↵
-rw-------    1 nide     ics         10843 Jun 19 15:06 abc.c
$ chmod u-w abc.c↵
$ ls -l abc.c↵
-r--------    1 nide     ics         10843 Jun 19 15:06 abc.c

$ ls -l def↵
-rw-r--r--    1 nide     ics          1474 Aug  2  2009 def
$ chmod a+x def↵
$ ls -l def↵
-rwxr-xr-x    1 nide     ics          1474 Aug  2  2009 def

$ ls -ld diary↵
drwxr-xr-x    3 nide     ics          8192 Mar 21 10:51 diary
$ chmod go-rx diary↵      (··· 他のユーザはそのディレクトリの中へ入れなくなる)
$ ls -ld diary↵
drwx------    3 nide     ics          8192 Mar 21 10:51 diary
```

ファイルの属性は、システムやソフトウェアが自動的に調整してくれることもある。例えば Mail ディレクトリは、メールソフトが自動的に、他の人からは読み・書き・そこへの移動ともできない属性で作成してくれている。

```
$ ls -ld Mail↵     (カレントディレクトリはホームディレクトリであるとする)
drwx------   12 nide     ics          4096 Jun 19 01:38 Mail
```

また、C などの言語のコンパイラは、最終的に作られる実行可能プログラムに、自動的に実行可能属性をつけてくれる。よって、先の例のように chmod コマンドでファイルに明示的に実行可能属性をつけることが必要になるのは、主に、シェルスクリプトなど、コンパイラが使われない開発形態でのプログラムを作るときである (1.1.2 節に登場する)。

第1章　シェルスクリプト

1.1　シェルスクリプトの基本

1.1.1　シェルスクリプトの作り方

コマンドをファイルに書いておき、

```
$ sh ファイル名↵
```

で実行できる。複数のコマンドも書ける。例えば、ldt.sh というファイルに

```
ls -l
date
```

と書いておいて「`sh ldt.sh`↵」とすれば、2 つのコマンドが順次実行される。

このとき、我々は sh に与えるプログラムをファイルに書き、それを sh に実行させていることになる。そこで、このファイルを「**シェルスクリプト**」という。

> (注) シェルスクリプトを実行するシェルは、対話的に使っているシェルとは別のプロセス。従って、対話的に使っているシェルがもし C シェル系であっても、それとは関係なく、シェルスクリプトは B シェル系のシェルの使用を前提に書くことができ、B シェル系のシェルで実行できる。

1.1.2　シェルスクリプトをコマンド化

以下の手順を踏めば、シェルスクリプトのファイル名を直接コマンドにできる。つまり、「`sh` ファイル名↵」でなく「`./`ファイル名↵」で実行できるようになる。

1. ファイルの先頭行に「`#!/bin/sh`」と書く
2. 「`chmod a+x` そのファイル名↵」を実行する (**一度だけ**でよい。ファイルを修正するたびに行う必要はない)

例えば、ldt というファイル (ldt.sh ではなく) に

```
#!/bin/sh
ls -l
date
```

と書き (「#!」は先頭行に書く。また「#!」の前には空白を開けない)、「chmod a+x ldt↵」とすれば ldt というコマンドができ、

```
$ ./ldt↵
```

で実行可能。あるファイルが既に実行可能にされているかどうかは、ls -l (0.3.1 節) でわかる。

> なお、1.1.1 節のように「sh ファイル名↵」として実行する場合は、ファイル名の末尾は「.sh」で終えることが多いが、本節のようにシェルスクリプトをコマンドにしてしまう場合は、ファイル名の終わりに拡張子 (「.」以降) は通常付けない。理由は、シェルスクリプトのファイル名が直接コマンド名になるので、ファイル名の末尾を「.sh」にすると、コマンド名にも「.sh」が付いてしまい、うっとうしいため。例えば、上記の例でファイル名を ldt.sh にすると、実行する際にも「./ldt.sh↵」とせねばならなくなる (大して困らないといえばその通りだが)。

シェルスクリプトの中でも、ワイルドカードやリダイレクト・パイプなど (→0.2) は使用可能。また、シェルスクリプトとして作ったコマンド自体に対しても、ワイルドカードやリダイレクト・パイプは使用可能。

1.1.3 コマンドサーチパス

実行したいコマンドのファイルのあるディレクトリが、環境変数 PATH で[10]指定されているディレクトリ群 (「**コマンドサーチパス**」という) のどこかに入れてあれば、「./ls↵」とか「/bin/ls↵」のようにディレクトリ名を指定することなく、「ls↵」のようにコマンド名の部分だけで実行できる (コマンドファイルがシェルスクリプトである場合だけでなく、C 言語などで作成した機械語ファイルであってもそう)。

コマンドサーチパス (つまり環境変数 PATH の値) は

```
$ echo $PATH↵
```

で確認できる。G 棟システムのマシンでは、コマンドサーチパスの先頭に「自分のホームディレクトリ/bin」が入っている。

従って、(G 棟システムでは) ホームディレクトリの下に bin というディレクトリを作り、自分で作ったコマンドファイルをそこに置けば[11]、(「./」を付けることなく) コマンド名の部分だけで実行できるようになる。

しかも、そのようにすれば、そのコマンドファイルのあるディレクトリだけではなく、他のディレクトリにいても、そのコマンドを (ディレクトリ名を指定することなく) 実行できるのが便利な点。

例えば、1.1.2 で作った ldt シェルスクリプト (「chmod a+x ldt↵」を 1 回しておくのをお忘れなく) を、ホームディレクトリ下の bin ディレクトリに置けば、次回からは「./ldt↵」でなく単に

```
$ ldt↵
```

でこのシェルスクリプトを実行できるようになる。しかも、ldt ファイルの置いてあるディレクトリに限らずどこのディレクトリにいても、単に「ldt↵」で実行できる。つまり、自作のコマンドが、システムに最初から用意されているコマンドと同等に使えるようになるのである。

> G 棟システムのマシンでは、環境変数 PATH の値の先頭に「自分のホームディレクトリ/bin」が入っているが、世の中のどのマシンでもそうだというわけではない。ただし、PATH の値は自分で変更できる。方法はシステムによっても異なるが、例えば多くの Linux システムの場合、PATH の先頭に「自分のホームディレクトリ/bin」を追加するには、ホームディレクトリの下の .bashrc というファイルの末尾に以下のような 1 行を書き足し、
>
> ```
> PATH="$HOME"/bin:"$PATH"
> ```
>
> そしてログインし直せばよい。(G 棟システムではシステム全体の標準設定が既にそうなっているので、.bashrc ファイルにはこれが書かれておらず、また、改めてこれを書き足すこともいらない。念のため。)

[10] 環境変数については 4.6 節で述べる。

[11] ただし、対話に使っているシェルが C シェル系の場合は、コマンドファイルを bin ディレクトリに置いた直後の 1 回だけは、事前に「rehash」というコマンドを実行せねば (またはログインし直さねば) ならない。

[ここで重要な注意!] 以下では、コマンドを実行する例を説明する場合、自作のコマンドであっても、資料では原則としていちいち「./」は**付けずに**記述する。従って、本節の操作をして、「./」を付けずに実行できるようにしておくか、あるいは、そうしない場合は「./」を**補って**読む (操作する) こと。例えば、1.2 節の例で「`calen 2014 5`↵」と操作するように書かれている場合、実際には「`./calen 2014 5`↵」と操作するか、あるいは calen ファイルをホームディレクトリ下の bin ディレクトリに置いてから「`calen 2014 5`↵」とする。

1.2　引数を取るシェルスクリプト

シェルスクリプトを実行するとき、引数を渡すことができる。シェルスクリプト内では、第 1 引数は「`$1`」、第 2 引数は「`$2`」、··· のように書く。

例えば

```
#!/bin/sh
cal $2 $1
date +'%m/%d %H:%M'
```

というシェルスクリプト calen があるとき、

```
$ calen 2014 5↵    ···「cal 5 2014」と「date +'%m/%d %H:%M'」が実行される
$ calen 2014 6↵    ···「cal 6 2014」と「date +'%m/%d %H:%M'」が実行される
```

のようになる (date の引数は日時の出力フォーマットの指定である[12])。

> このくらいの短いスクリプトでも、間違える人は間違えるものである。過去の例としては、上のスクリプトの「`$`」を「`&`」と打ち間違えた例が複数回あった。当然、それでは正しく動かない。「どうやったら `$` と `&` を打ち間違えるの?」と思うかもしれないが、人間というのは思いもしないところで間違えるものなのだ。

また、

```
#!/bin/sh
cp $1 $1.bak
echo $1 を $1.bak にコピーしました >&2
```

というシェルスクリプト cpbak があるとき、「`cpbak a.c`↵」とすると、a.c というファイルを a.c.bak にコピーし、「`a.c を a.c.bak にコピーしました`」と (標準エラー出力に) 出力する。

> ここでの「～コピーしますか」のメッセージは、端末でのユーザとのやりとりを意図したものであるので、標準エラー出力へのリダイレクト記法「`>&2`」(→0.2.3.1 節) を用いて、標準エラー出力に出すのが筋である[13]。

しかし、このシェルスクリプトでは、仮に cp コマンドがコピーに失敗しても (例えばそもそも a.c というファイルがなければコピーに失敗する)、その次の echo コマンドが実行され、「～コピーしました」のメッセージを出力してしまう。

1.3　制御構造 (if 文・while 文・for 文)

1.3.1　if 文

if 文は以下の形をしている。

[12] 古いシステムの date コマンドでは出力フォーマットを指定できないことがある。

[13] ちなみに、エラーメッセージも、そうした「端末でのユーザとのやりとりを意図したものであるため、標準エラー出力に出すべき」であるメッセージの 1 つである。

```
if コマンド1; then
        コマンド1 が真の場合に実行するコマンド列
elif コマンド2; then
        コマンド2 が真の場合に実行するコマンド列
elif コマンド3; then
        コマンド3 が真の場合に実行するコマンド列
  ⋮
elif コマンドn; then
        コマンドn が真の場合に実行するコマンド列
else
        どれも真でなかった場合に実行するコマンド列
fi
```

「コマンド$_1$」「コマンド$_2$」$\cdots$「コマンド$_n$」のそれぞれの後ろに「;」(または改行) が必要な点に注意。

「コマンド$_i$が**真の場合**」とは、コマンド$_i$を実行した結果の「**戻り値**」が 0 であった場合 (C 言語とは逆で、0 が真)。UNIX ではどんなコマンドも、終了したときには何らかの整数値を「戻り値」として返す。そして多くのコマンドは、正常終了したときに「戻り値」0、何らかのエラーで終了したときに非 0 の「戻り値」を返すように作られている。従って、多くのコマンドでは、「コマンド$_i$が真の場合」とは「コマンド$_i$が**正常終了**した場合」である。(ただし、コマンドの戻り値に、「正常終了したかどうか」以外の意味を持たせているコマンドもある。従って、コマンドの戻り値の正確な意味は、コマンド毎にマニュアルで調べなければならない。例えば fgrep コマンドは、探索文字列が見つかったかどうかを「戻り値」で返す。)

なお、「戻り値」は、コマンドが画面に表示するものとは全く**無関係**なので注意。「戻り値」は通常、画面には見えない。

コマンドの戻り値を目で確認するには、コマンドの実行直後に「`echo $?`↵」とすればよい[14]。例えば、fofofo というファイルが存在しないとする。

```
$ cat fofofo↵
cat: fofofo: No such file or directory
$ echo $?↵
1
$ echo $?↵
0
```

cat コマンドがエラー終了で非 0 の戻り値を返していることがわかる。2 度目の「`echo $?`」が 0 を表示するのは、1 度目の「`echo $?`」コマンドが正常終了して戻り値 0 を返したからである。

ちなみに C で作ったプログラムの場合、`exit(`n`)` すれば (あるいは main 関数から `return` n で終了すれば) 戻り値は n。試してみよう。

```
int main(void)
{
        return 7;
}
```

というプログラム a.c があるとする。

```
$ cc a.c↵
$ ./a.out↵
$ echo $?↵
7
```

実行するとこのようになる。

C 言語の main 関数の型が void でなく int と定められているのは、こういう事情による。

elif は必要に応じて何個も書けるし、elif や else が必要ない場合は書かなくてもいい。例えば

```
if コマンド1; then
        コマンド1 が真の場合に実行するコマンド列
else
        コマンド1 が真でなかった場合に実行するコマンド列
fi
```

あるいは

[14] C シェルでは `echo $status` とする必要がある。ただし tcsh では `echo $?` でもよい。

```
if コマンド1; then
        コマンド1 が真の場合に実行するコマンド列
fi
```

だけでも OK。むしろ先にこちらを理解し、後で elif 込みの if 文を理解する方が理解が早まるかも。

1.3.1.1　例

例えば

```
#!/bin/sh
if cp $1 $1.bak; then
        echo $1 を $1.bak にコピーしました >&2
else
        echo $1 を $1.bak にコピーできませんでした >&2
fi
```

というシェルスクリプトは、コピーに成功したかどうかで異なるメッセージを出力する。「`cp $1 $1.bak`」の部分が、1.3.1 節冒頭 (p. 17) の if 文の形の中の「コマンド$_1$」の箇所に当たる。

ただしこのシェルスクリプトでは、コピーに失敗した場合、cp コマンド自体が出すエラーメッセージも表示される。これに対し、

```
#!/bin/sh
if cp $1 $1.bak 2>/dev/null; then
        echo $1 を $1.bak にコピーしました >&2
else
        echo $1 を $1.bak にコピーできませんでした >&2
fi
```

というシェルスクリプトにすれば、cp コマンド自体の出すエラーメッセージは抑止される (/dev/null については 0.2.3.1 節参照)。下記は実行例 (スクリプト名が cpwithmsg だとする)。

```
$ cpwithmsg abc↵                    (abc というファイルは存在するとする)
abc を abc.bak にコピーしました
$ cpwithmsg def↵                    (def というファイルは存在しないとする)
def を def.bak にコピーできませんでした   (cp コマンド自体のエラーメッセージは出てこない)
```

cp コマンドが失敗した場合だけメッセージを出し、成功した場合は何もしないようにするには

```
#!/bin/sh
if cp $1 $1.bak 2>/dev/null; then
        true
else
        echo $1 を $1.bak にコピーできませんでした >&2
fi
```

とする。「true」は何もしないコマンド (true コマンドには「:」という別名もある。1.3.2 節参照)。

> システムによっては、「!」で条件を反転することができるので、同じ意味のことを
>
> ```
> #!/bin/sh
> if ! cp $1 $1.bak 2>/dev/null; then
>         echo $1 を $1.bak にコピーできませんでした >&2
> fi
> ```
>
> とも書ける。こうすれば else はこの場合不要である。
>
> しかし、シェルスクリプトでのこの「!」による条件反転の書き方は、全てのシステムで使えるわけではない (ほとんどのシステムでは使えるが)。本資料でも、以後この書き方は使わない。

1.3.1.2 Emacs による自動インデント

他のプログラミング言語と同じく、シェルスクリプトでも if 文や後出の while 文、for 文などでは適切な**インデント** (字下げ) を行うべきである。これによってプログラム (スクリプト) の**構造**がわかりやすく、読みやすいものになる。バグの入りにくい、入っても修正しやすいプログラムを書くためには、インデントは必須条件の 1 つである。

Emacs には、多くのプログラミング言語 (C, Java, ···) に対して、その言語専用の「**モード**」というものがあり、その言語の文法に応じて、インデント (字下げ) を支援したり、文字を構文別に色分けしてプログラムの構造を把握しやすくしたり (システムによってはこの機能はないこともある) してくれる機能を持っている。その一環として、Emacs には**シェルスクリプト用モード** (sh-mode) もある。

Emacs がシェルスクリプトモードになっているかどうかは、Emacs の画面の下の方にある「**モード行**」で分かる。そこに「`(Shell-script[sh])`」と表示されていればシェルスクリプトモードである (図 1.1)。

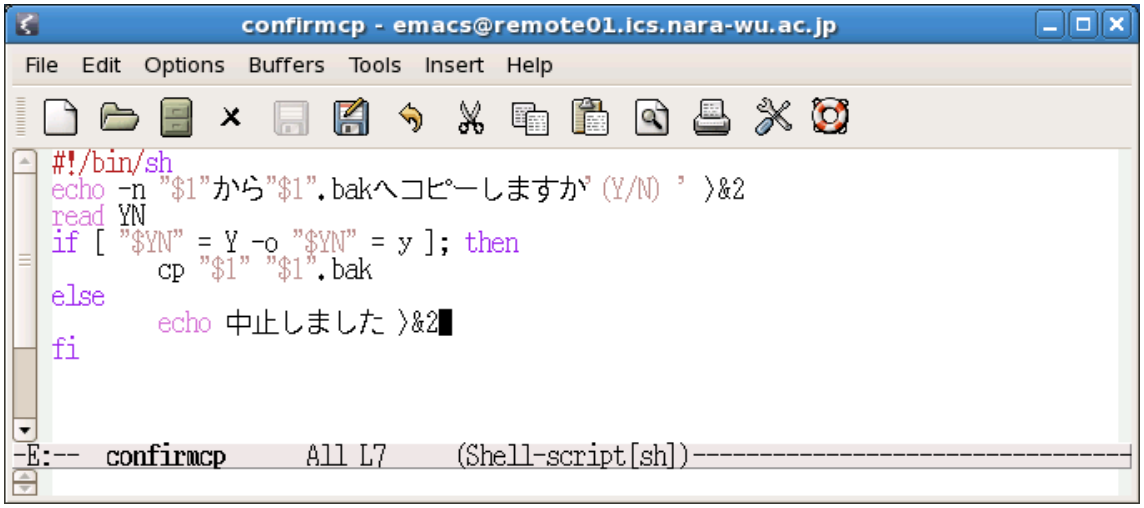


図 1.1: Emacs のシェルスクリプトモード (編集中のスクリプトは 1.8 節に出てくるもの)

Emacs は、既に先頭に「`#!/bin/sh`」と書かれているファイルを編集開始すると、自動的にシェルスクリプトモードになる[15]。また、**G 棟システム**の Linux の Emacs に限り、Emacs の現在の状態が「Fundamental モード」であるとき (モードラインに「`(Fundamental)`」と表示されているとき) に ESC ESC s とすると、シェルスクリプトモードに変わり、同時にファイルの先頭に「`#!/bin/sh`」が挿入される (ただし既に先頭が「`#!`」で始まっている場合を除く)。これは、シェルスクリプトのファイルを最初に編集開始する際に便利である。なお、他の環境の Emacs を含め、ESC x sh-mode ↵ とすることでもシェルスクリプトモードになる (どのモードからでも。ただし先頭に「`#!/bin/sh`」を入れてくれるようなおまけ機能はない)。

シェルスクリプトモードになっている場合、`if` や `then` などのキーワードや、クォート (1.7 節) されている部分などが色分けされ、構文の把握がしやすくなる。また、**G 棟システム**の Linux の Emacs に限り、Ctrl-x \ (Ctrl キーを押しながら x、続いて Ctrl を離してからバックスラッシュ) によって、シェルスクリプトの文法に基づいた**インデントを自動的に**行ってくれる[16](他の環境の Emacs にも自動インデントの機能自体はあるが、標準では短いキー操作で呼び出せるようにはなっていない)。例えば図 1.2 のように、現在編集中のシェルスクリプトが全くインデントされていない、もしくはインデントがぐちゃぐちゃの状態のとき、Ctrl-x \ と操作すると自動的に図 1.1 のような整然としたインデントになる (Fundamental モードでこれをやらないよう注意のこと)。

Emacs のインデント機能を積極的に活用して、読みやすいスクリプトを書くよう心がけよう。

---

[15] また、ファイル名末尾が「`.sh`」で終わっているファイルを Emacs で編集開始した場合も、自動的にシェルスクリプトモードになる。しかし、1.1.2 節でも述べたように、シェルスクリプトの場合、ファイル名を「`.sh`」で終えることにはあまり必然性はない。

[16] G 棟システムの Linux の Emacs では、シェルスクリプトモードに限らず他のプログラミング言語のモードのときでも、Ctrl-x \ による自動インデントは働くようになっている。

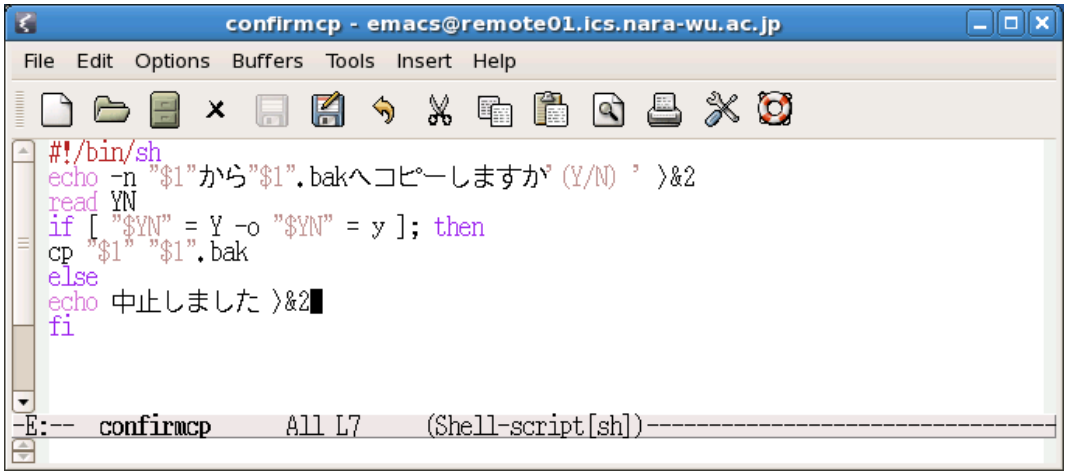


図 1.2: インデントのされていないシェルスクリプト

1.3.1.3　test コマンドの利用

if 文の条件判断用によく使われるコマンドとして、「test」というコマンドがある。このコマンドで、ファイルの存在検査や整数の比較、文字列の比較などいろいろな条件判断が行える。

例えば「`test -e` ファイル名」というコマンド[17]は、そのファイルが存在すれば戻り値 0 (真) を、でなければ戻り値 1 (偽) を返す (戻り値が返されるだけで、画面には何も表示されない)。

確かめてみよう。xyz というファイルが存在しないとする。端末にて、

```
$ test -e xyz↵          (何も表示されない)
$ echo $?↵              (直前のコマンドの戻り値を出力。1.3.1 節参照)
1                       (戻り値が 1 であることがわかる)
```

test コマンドを使うと、例えば次のようなシェルスクリプトが書ける。

```
#!/bin/sh
if test -e $1; then
        echo ファイル $1 は存在します
else
        echo ファイル $1 は存在しません
fi
```

「`test -e $1`」の部分が、if 文の形 (1.3.1 節) の「コマンド$_1$」にあたる。

test コマンドは他にもいろいろなオペレータを持つ (詳しくは A.2 節参照)。例えば「`-eq`」「`-le`」などのオペレータで**整数の比較**、「`=`」「`!=`」オペレータで**文字列比較** (「`==`」ではなく**「`=`」である**ことに注意[18]) が行える。また、条件の **and · or · not** を表すオペレータ「`-a`」「`-o`」「`!`」もある (特に、1.3.1.1 節で述べた **if 文の**条件反転の「`!`」がないシステムであっても、**test コマンドの**条件反転オペレータ「`!`」は必ず**使える**)。

以下は、文字列比較を行うシェルスクリプトの例。第 1 引数が「abc」ならば「OK」と出力している。

```
#!/bin/sh
if test $1 = abc; then
        echo OK
fi
```

ここで「`=`」の前後には**空白が必要** (例えば「`if test $1=abc; then`」では**だめ**)。なぜなら、test コマンドには「`$1`」や「`=`」や「`abc`」を**別々の引数**として与えねばならない約束だからである (A.2 節参照)。

[17] 非常に古いシステムの test コマンドには `-e` がないことがあり、その場合、状況に応じて `-f` や `-r` など (A.2 節参照) で代用することになる。

[18] シェルの種類によっては「`==`」でも OK だが、全てのシェルでそうであるわけではない。

また、次は整数の比較を行うシェルスクリプトの例。「-ge」が ≥ の意味であることに注意。「$1 >= 100」とは書かない (「>」がリダイレクトの意味になってしまうからである)。

```
#!/bin/sh
if test $1 -ge 100; then
        echo $1は100以上です
else
        echo $1は100未満です
fi
```

さらにおまけで、and を表す「-a」オペレータとの併用例。

```
#!/bin/sh
if test $1 -ge 100 -a $1 -lt 200; then
        echo $1は100以上200未満です
else
        echo $1は100未満か200以上です
fi
```

test コマンドは、if 文の中だけでなく、後述の while 文などの中でも多用される。

• ***test コマンドの別名*** test コマンドには、「[」という別名がある。ただし、「[」というコマンド名を使う場合は、最後に「]」という引数を 1 つ余分につける。すなわち、「test 何々」の代わりに「[␣何々␣]」と書く (両者は**同じ意味**)。

例えば「[␣-e␣xyz␣]」は「test -e xyz」と同じ意味になる。

確かめてみよう。今度も、xyz というファイルが存在しないとする。

```
$ [ -e xyz ]↵          (何も表示されない)
$ echo $?↵
1
```

このことを使うと、例えば、先の文字列比較のシェルスクリプトを次のようにも書ける (注: 下の例で、2 行目の「#」から右は**注釈**である。1.5 節参照)。「[␣$1␣=␣abc␣]」の部分が、if 文の形 (1.3.1 節) の「コマンド$_1$」にあたる。

```
#!/bin/sh
if [ $1 = abc ]; then    #「if test $1 = abc; then」と書くのと同じ
        echo OK
fi
```

C 言語での条件判断では、「if(x==3){···」や「if (x == 3) {···」のように、空白があってもなくても文法的に正しいが、上記シェルスクリプトでは、「if」の右、「[」の右や「]」の左、「=」の左右には**空白が必要**である。例えば

```
if[ $1 = abc ]; then  とか  if [$1 = abc]; then  とか  if [ $1=abc ]; then  とか
```

では**だめ**。その理由は test を使う場合と同じで、「[」は**コマンド名**なので if とは離さねばならず、しかもこのコマンドには「$1」「=」「abc」「]」をそれぞれ**別々の引数**として与えねばならないからである。**理由とセットで**理解しよう。

「if 文の条件部は『[ ]』で囲む」という約束**ではない**ことに注意。毎年 1～2 人位は間違える人が必ずいる。

- if 文の条件部には**コマンドを書く**
- 「[」はコマンド名の 1 つであり、test コマンドの別名である
- 「if コマンド$_1$; then···」の中の「コマンド$_1$」の部分に「[$1 = abc]」が来たのが上記の例である

ということを**正確に**理解しよう。1.3.1.1 節の例で「`if [ cp $1 $1.bak ]; then`」のように「`[ ]`」で囲んで**いない**のは、

- 「`cp $1 $1.bak`」の部分がコマンドであり、このコマンドの真偽によって分岐しようとしている
- 「`[`」コマンドに「`cp`」「`$1`」… という引数を与えようとしているのではない

からである。

- ***コマンドの戻り値の判断***　例えば、下の 2 つは同じ働きをする。

```
if cp $1 $1.bak; then
        … # cpコマンドの
          # 戻り値が真の場合
else
        … # 同・偽の場合
fi
```

```
cp $1 $1.bak
if [ $? -eq 0 ]; then
        … # cpコマンドの
          # 戻り値が真の場合
else
        … # 同・偽の場合
fi
```

どちらも、cp コマンドの戻り値が 0 (真) の場合とそうでない場合で処理を分けているからである。ただし、この例の場合は左の方が書き方として素直である。

1.3.2　while 文

while 文は以下の形をしている。

```
while コマンド1; do
        コマンド1 が真であるあいだ実行するコマンド列
done
```

コマンド$_1$ が真かどうかの判定は、if 文の場合と同じ。例えば、1.3.1.1 節に登場した、何もしないコマンド「true」は、戻り値が常に 0、つまり常に真を返すので、

```
#!/bin/sh
while true; do
        echo hahaha
        sleep 1
done
```

というシェルスクリプトで無限ループを実現できる (Ctrl-C で止められる)。

> true コマンドには「`:`」という別名があるので、この例は
>
> ```
> #!/bin/sh
> while :; do
>         echo hahaha
>         sleep 1
> done
> ```
>
> と書いても同じ意味。しかしわかりにくくなるので、本資料ではきちんと「true」と書くことにする。

以下のシェルスクリプトは、waaiwaai というファイルが存在しない限りループするが、実行中に他の端末から waaiwaai というファイルを作成してやるとループを脱出する。「`!`」が test コマンドの not オペレータ (1.3.1.3 節) であることに注意。

```
#!/bin/sh
while [ ! -e waaiwaai ]; do
        echo hahaha
        sleep 1
done
echo finished
```

while ループや後述の for ループの中では、C 言語と同様の continue や break も使える。例えば次のスクリプトは、上記のスクリプトと同じ動作をする。

```
#!/bin/sh
while true; do
        if [ -e waaiwaai ]; then
                break   # whileループを脱出
        fi
        echo hahaha; sleep 1
done
echo finished
```

1.3.3 for 文

for 文は以下の形をしている。

```
for 変数名 in 文字列1 ··· 文字列n; do
        繰り返し実行するコマンド列
done
```

変数名は、C 言語で使えるものと同様のものを使えばいい (変数名の大文字小文字は区別される)。

for 文の使い方は C 言語とずいぶん違う。例えば

```
#!/bin/sh
for i in a.c b.c c.c; do
        cp $i $i.bak
done
```

というシェルスクリプトを実行すると、その過程は

- まず最初の「文字列$_1$」のところ (つまり「a.c」) が変数 i に代入され、コマンド「cp $i $i.bak」を実行。その際、「$i」のところは変数 i の値に置き換えられるので、結局「cp a.c a.c.bak」が実行される。
- 次に「文字列$_2$」のところ (つまり「b.c」) が変数 i に代入され、コマンド「cp $i $i.bak」を実行。「$i」は変数 i の値に置き換えられるので、「cp b.c b.c.bak」が実行される。
- 最後に「文字列$_3$」(つまり「c.c」) が変数 i に代入され、コマンド「cp $i $i.bak」を実行。「$i」は変数 i の値に置き換えられるので、「cp c.c c.c.bak」が実行される。

となるので、結局

```
cp a.c a.c.bak
cp b.c b.c.bak
cp c.c c.c.bak
```

を実行したのと同じになる。

```
#!/bin/sh
for i in $1 $2 $3; do
        cp $i $i.bak
done
```

とすると、シェルスクリプトに与える 3 つの引数に対して同様のことができる。でも、このシェルスクリプトは、引数がちょうど 3 つの時しかうまく動かない。引数がいくつであってもうまく動くようにするには? それには 1.7.1.3 節に出てくる知識が必要になる。

1.4　複数のコマンドの結合

シェルスクリプト内では、複数のコマンドを柔軟に結合したり、結合したコマンド群を単一のコマンドと同様に扱ったりできる。多少高度な構文ではあるが、処理を簡潔に書くために有用であるため、本資料でも後の方で時々出てくるので、理解しておきたい。

1.4.1　if 文・while 文・for 文

if 文・while 文・for 文は、それ全体で 1 つのコマンドと見なされる。従って例えば

```
for i in 2012 2013 2014; do
        cal $i
done | less
```

のように、for 文全体をパイプにつないだり、リダイレクトしたりもできる。

C シェル系ではこれができない。シェルスクリプトに C シェル系を使わない理由のもう 1 つがこれ。

1.4.2　コマンドのグループ化

複数のコマンドを単に「{」「}」でまとめて、それ全体を 1 つのコマンドと同様に扱える。例えば

```
{
        cal 2012
        cal 2013
        cal 2014
} | less
```

のようなことが可能である。

なお、「{」の次には空白か改行、「}」の前には「;」か改行が必要。上の例のように、「{」の次と「}」の前で改行し、その間でインデントしておけば、書き間違えずに済む。
「{」「}」の代わりに「(」「)」で囲むこともできるが、両者には書き方にも意味にも違いがある。ここでは深入りしない。

1.4.3　論理オペレータ

「コマンド$_1$ `&&` コマンド$_2$」や「コマンド$_1$ `||` コマンド$_2$」のように、コマンドを「`&&`」や「`||`」で結合することができる。この「`&&`」や「`||`」は、C 言語のそれと同様の意味を持っており、例えば「コマンド$_1$ `||` コマンド$_2$」は、コマンド$_1$ が**真を返さなかった場合のみ**コマンド$_2$ を実行するし、また「コマンド$_1$ `&&` コマンド$_2$」は、コマンド$_1$ が**真を返した場合のみ**コマンド$_2$ を実行する[19]。

この書き方は、if 文や while 文の条件部に使われることも多いが、それ以外の箇所で使うことも多い。まずそちらを先に紹介しよう。例えば、1.3.2 節に出た while と break を使った例の中の

```
if [ -e waaiwaai ]; then
        break   # whileループを脱出
fi
```

の部分は、同じ意味のことを if 文を使わずに

```
[ -e waaiwaai ] && break
```

と書くことができる。また、1.4.2 節で述べた「コマンドのグループ化」と併用すれば、

[19] ちなみに、C 言語などとは違って、シェルスクリプトの `&&` と `||` は優先順位は同じである。

```
if [ -e waaiwaai ]; then
        echo waaiwaaiファイルの存在が確認されました >&2
        break
fi
```

の代わりに

```
[ -e waaiwaai ] && {
        echo waaiwaaiファイルの存在が確認されました >&2
        break
}
```

と書くようなこともできる。ただし、この例では if 文の方が、条件判断であることがわかりやすい。

もちろん、if 文や while 文の条件部でもこの書き方は有用。例えば「第 1 引数が整数として 2 以上 5 以下なら OK を出力」という処理は

```
if [ 2 -le $1 ] && [ $1 -le 5 ]; then
        echo OK
fi
```

と書ける。ただし、この例のように test コマンド同士の and や or の場合は、test コマンド自体のオペレータに and や or の意味のものがある (「-a」や「-o」。1.3.1.3 節参照) ので、そちらを使って以下のように書く方が簡潔。(「&&」はコマンド同士を結ぶもの、「-a」は test コマンドの引数 (オペレータ) である。区別して理解すること。)

```
if [ 2 -le $1 -a $1 -le 5 ]; then
        echo OK
fi
```

1.5 注釈

シェルスクリプトでは、行頭、あるいは空白の次に「#」が現れれば、そこから行末までは注釈。ただし、先頭行の行頭の「#」は「#!/bin/sh」の始まりの意味に解釈されるので、そこには注釈は書けない。

注釈を表す「#」は、**行頭**または**空白の次**になければならないことには注意が必要。例えば

```
if [ $1 = abc ]; then# 注釈と見なされない例
        echo OK
fi
```

は、「#」の前に空白がないので注釈としては扱われず「then#」という語と解釈され、「if 文の then が現れる前に fi が現れた」ということになってエラーになる。

1.6 シェル変数

シェルスクリプトの中で変数を使うことができる。これをシェル変数という。1.3.3 節で出た for 文では、ループ変数 i への暗黙の代入が行われていたが、この i もシェル変数である。

for 文の実行に伴うシェル変数への暗黙の代入ではなく、明示的にシェル変数への代入を行うには、「変数名=値」とする。代入する値は任意の文字列でよい。例えば

```
#!/bin/sh
A=a.c
cat $A
```

というスクリプトは「cat a.c」と同じ動作をする。代入の際に「A␣=␣a.c」のように「=」の前後に空白を入れては**いけない**。なぜなら、「=」の両側に空白があると、『「A」というコマンドに「=」「a.c」という引数を与える』という意味になり、「A」というコマンドが存在しないためエラーになるからである。これも、**理由とセットで**理解しよう。(注意! **test コマンド**のときには「=」の左右に**空白が必要**だったが (1.3.1.3 節)、**シェル変数への代入**のときは「=」の左右に空白を**入れてはいけない**。両者で逆なので、混乱せぬよう。理由とセットで理解すれば、混乱を起こすことはない。)

実は、変数の値を参照するには「$A」の代わりに「${A}」と書いてもよく、両者は**同じ意味**。特に、変数名の後ろに英数字や「_」が続く場合、変数名と区別するために「{ }」が**必要となる**。下記はその例。

```
#!/bin/sh
HA=abc; HAHAHA=def
echo $HA                # 「abc」と表示
echo ${HA}              # これも「abc」と表示

echo $HAHAHA            # 「def」と表示
echo ${HA}HAHA          # 変数HAの値と「HAHA」、つまり「abcHAHA」を表示
```

シェル変数が有用な場合の 1 つは、同じ値をスクリプト中で何度も使う場合である。以下は、1.3.1.1 節最初の例のスクリプトを、シェル変数を使って書き直した例。1.3.1.1 節のスクリプトのままだと、例えばコピー先を $1.bak から $1.old に変更したい場合、echo によるメッセージ出力の部分も含めると 3ヶ所「bak」を「old」に変えねばならない (が、どこかを変え忘れるというミスを犯しやすい)。これに対し下記のスクリプトなら、1ヶ所変更するだけで済む。

```
#!/bin/sh
BAK=$1.bak
if cp $1 $BAK; then
        echo $1 を $BAK にコピーしました >&2
else
        echo $1 を $BAK にコピーできませんでした >&2
fi
```

なお、シェル変数はあくまでそのシェルの中だけでの存在で、シェルスクリプト内から起動した他のコマンドには影響を及ぼさない。

ただし、そのシェル変数が「**環境変数**」でもある場合は別である。環境変数については 4.6 節で述べる。

1.6.1　特殊なシェル変数

あらかじめ設定されていて、特別な意味に使われるシェル変数もある。例えばシェル変数 HOME には通常、そのユーザのホームディレクトリのフルパスが入っている (この変数は通常、環境変数でもある)。1.1.3 節に出た PATH も、特別な意味に使われるシェル変数 (であり、環境変数) の 1 つである。

$1 や $2、さらには 1.3.1 節に出てきた $? なども、特殊なシェル変数の一種と考えることもできる。ただし、これらの変数には直接代入はできない。また、これらは (シェル変数ではあるが) 環境変数ではない。

1.7　クォート

シェルのコマンド行では、空白や「$」「<」「>」「*」「#」などが特別な意味を持つが、「’ ’」(**シングルクォート**) で囲むことにより、特別な意味を消すことができる。これをクォートと呼ぶ。

```
#!/bin/sh
echo a > b
echo ’a > b’
```

2 回使われている echo コマンドのうち、上は「a」を出力しそれがファイル b にリダイレクトされるが、下は「a > b」を出力する。

これは (もちろん) シェルスクリプトの中だけではなく、シェルを対話的に使っているときにもそうである。

```
$ echo a > b↵
(何も画面に出ず、b というファイルに a と書き込まれる)
$ echo 'a > b'↵
a > b     (と画面に出る)
$ fgrep a b* fuhyo↵
(「a」が含まれる行を、b で始まる名前のファイル、および fuhyo というファイルの中から探す)
$ fgrep 'a b*' fuhyo↵
(「a b*」が含まれる行を、fuhyo というファイルの中から探す)
$ cp a 'b c'↵
(a というファイルを「b c」というファイルにコピーする)
$ rm 'b c'↵
(「b c」というファイルを消す)
```

「\」によって、直後の 1 文字の特別な意味を消すこともできる[20]。例えば、以下の 2 つは同じ意味である。

```
$ echo 'a > b'↵
$ echo a\␣\>\␣b↵
```

「' '」でなく「" "」(**ダブルクォート**) で囲むことによってもクォートができる[21]。ただし「" "」の中では、①「$」と後述 (1.9 節) の「`」は特別な意味を失わない。また ②「\」(上述) も、直後に「$」「`」「"」「\」または改行文字のいずれかがくる場合に限り、特別な意味を失わない。

```
#!/bin/sh
A=hello
echo $A > b
echo '$A > b'
echo "$A > b"    # echo "${A} > b" でも同じ意味
echo "\$A > b"
```

この例では、最初の echo は「hello」を出力してファイル b へリダイレクト、2 番目は「$A > b」を出力、3 番目は「hello > b」を出力する。最後の echo は、「" "」内の「\」が特別な意味を失っていないため、その「\」の働きによって後続の「$」が特別な意味を失い、「$A > b」と出力する。

1.7.1 シェル変数の利用とクォート

1.7.1.1 シェル変数への代入時のクォート

シェル変数への代入を行う場合も、代入する文字列が空白など特別な意味を持つ文字を含む場合は、クォートが必要である[22]。例えば

```
#!/bin/sh
A='a b'
fgrep "$A" fuhyo       #「fgrep "a b" fuhyo」と同じ意味になる
```

上の例において、変数 A への代入のところを

```
#!/bin/sh
A=a b                  # (誤り)
fgrep "$A" fuhyo
```

と書いてはならない。これは違う意味になる (4.6.1 節参照)。

[20] ただし、「\」の直後が改行である場合だけは、その改行が無視される。

[21] C 言語では「' '」と「" "」の意味が異なっており、前者は文字、後者は文字列を表すものであった。これに対しシェルスクリプトでは「' '」「" "」のどちらも文字列のクォートであり、クォートのされ方が両者で異なるだけである。

[22] ただし、代入する文字列の中に「*」や「?」などがあっても、ワイルドカードの展開は行われない。

1.7.1.2　シェル変数の値の利用時のクォート

「" "」の中でも外でも「$」は変数置換を受けるが、「" "」の中の「$」が変数置換を受けた場合、その値がさらに展開されることはないのに対し、「" "」の外の「$」が変数置換を受けた場合、その値はさらに展開 (空白で切られたり、ワイルドカードが展開されたり) される。

```
#!/bin/sh
A='a b'
fgrep $A fuhyo          #「fgrep a b fuhyo」と同じ意味になる
fgrep "$A" fuhyo        #「fgrep "a b" fuhyo」と同じ意味になる

A='*'
fgrep $A fuhyo          #「fgrep * fuhyo」と同じ意味になるので
                        # ワイルドカードが展開されてしまう
fgrep "$A" fuhyo        #「fgrep "*" fuhyo」と同じ意味になる
```

このため、シェルスクリプト内でシェル変数の値を使う場合 (for ループに使う変数やコマンド行引数も含む) は、その変数の値に特殊文字が含まれる可能性がないとはっきりわかる場合以外は、「" "」で囲む方が**安全**。

例えば、以下の内容のシェルスクリプト (名前を「grepfuhyo」とする) があるとする。

```
#!/bin/sh
fgrep $1 fuhyo
```

これは、第 1 引数に与えた文字列を含む行をファイル fuhyo の中から検索しようというものである。しかしこれを、「a␣b」という文字列を検索しようとして

```
$ grepfuhyo 'a b'
```

のように使うと「fgrep a b fuhyo」という、意図と違うことが起きる。このシェルスクリプトは

```
#!/bin/sh
fgrep "$1" fuhyo
```

と書き直すべきである。これなら「$ grepfuhyo 'a b'↵」で意図通りの検索をする。ただし、こう直しても、「$ grepfuhyo a b↵」では (「"$1"」が「a」だけになるので) やはりだめであることに注意。

また、第 1 引数が abc であるかどうかを if 文と「[」コマンドで判定する場合も、下記の左より、右のように「$1」をダブルクォートする書き方の方がよい。

```
if [ $1 = abc ]; then
        …
fi
```

```
if [ "$1" = abc ]; then
        …
fi
```

左の書き方だと、もし第 1 引数がなかった (空だった) 場合「if␣[␣=␣abc␣]」と書いたのと同じことになり、「[」の引数にいきなり演算子「=」が現れたことになって、文法エラーになるが、右の書き方だと、第 1 引数がなかった場合も、空文字列と abc を比較するという意味になってちゃんと動作する。

[ここでちょっと練習問題] 1.3.1.1 節の 2 番目のスクリプト (名前が cpwithmsg だとする) も、クォートがないゆえ引数に特殊文字が混じった場合にうまく動かない例である。これを、引数に特殊文字が混じってもうまく動くようにクォートを加えてみよう。正しくできていれば、例えば下記のように動作するはずである。

```
$ touch 'a b'↵                (「a b」という空ファイルを作った)
$ cpwithmsg 'a b'↵
a b を a b.bak にコピーしました
$ rm 'a b' 'a b.bak'↵         (実験が終わったので「a b」「a b.bak」というファイルを消す)
$ cpwithmsg '*'↵              (「*」というファイルは存在しないとする)
* を *.bak にコピーできませんでした
```

以上のようにならない場合は、どこかのクォートが足りない。

(注意) 上の例では「*」というファイルは作っていないが、もし作ったとしてそれを消そうとする場合は、くれぐれも「rm *↵」とはしないこと。もしそうすると、ワイルドカードの働きでカレントディレクトリの全てのファイルが消えてしまう。「*」というファイルを消したければ「rm '*'↵」(あるいは「rm *↵」) としなければならない。

1.7.1.3 「引数全体」を表す記法

特例として「"$@"」は (シェルスクリプトの)「引数全体」を表す。つまり「"$@"」は、「"$1" "$2" "$3" ···」を、存在するだけ書き並べたのと同じ意味になる。

「" "」で囲まない「$@」や、「$*」も似た意味を持つが、「" "」で囲まれていないことによって、1.7.1.2 節で述べたのと同じ問題が発生するので、特に理由がない限り「"$@"」を使うべきである。

例として、以下の内容のシェルスクリプト (名前を「hahahagrep」とする) があるとする。

```
#!/bin/sh
fgrep hahaha "$@"
```

これは、引数 (任意個) に与えたファイルから、「hahaha」を含む行を検索するものである。例えばこれを

```
$ hahahagrep a b↵
```

として使うと、「fgrep hahaha a b」が実行される。(ここで、このスクリプトの「"$@"」のところを「$@」あるいは「$*」に変えたとしても、「$ hahahagrep a b↵」と実行すると元と同じ動作をする。では、「$@」あるいは「$*」に変えてしまうとまずいのは、どのようなコマンド行を実行した場合だろうか? 考えてみよ。) また、

```
$ hahahagrep a*.c↵
```

だとワイルドカードが展開され、「a」で始まり「.c」で終わる名のファイル全てから「hahaha」を含む行を探索することになる。

さらに、「"$@"」を使うと次のような「全ての引数に対して繰り返すシェルスクリプト」も書ける。

```
#!/bin/sh
for i in "$@"; do
        cp "$i" "$i".bak
done
```

1.8 端末入力

シェルの組み込みコマンド read を「read 変数名」のように使うと、標準入力から 1 行読んで、その内容をその変数に代入してくれる (ただし、行頭行末の空白は取り除かれる)。

これを使ってみよう。下記は、引数を 1 つ取り、引数に指定されたファイルを「その名前.bak」という名前のファイルにコピーするが、その前にユーザに確認を取り、返事が Y か y のときだけコピーを行うようなシェルスクリプトである。

```
#!/bin/sh
echo -n "$1"から"$1".bakへコピーしますか'(Y/N) ' >&2
read YN
if [ "$YN" = Y -o "$YN" = y ]; then
        cp "$1" "$1".bak
else
        echo 中止しました >&2
fi
```

先頭で echo -n としているのは、改行なしの出力を行うため[23]。また、test コマンドのオペレータ -o は or を意味する (A.2 節)。

cp コマンドの部分は「cp "$1" "$1.bak"」でも構わない。また、先頭の echo コマンドは、シェル変数や特殊文字をまとめてクォートして、

```
echo -n "$1から$1.bakへコピーしますか(Y/N) " >&2
```

のように書く方が多少扱いやすいだろう。

1.8.1　case 文—新たな制御構造

返事が Y でも y でも Yes でも yeah でも、Y か y で始まっていればコピーを行うように、1.8 節のシェルスクリプトを改造してみよう。その場合、「if ["$YN" = Y -o "$YN" = y -o "$YN" = Yes -o …」というように書き並べるわけにはいかない (Y か y で始まる文字列は無限個あるから)。そこで、代わりに case 文というものを使う。これは次のように書く。

```
case 文字列 in
パターン1)
        コマンド列1
        ;;
パターン2)
        コマンド列2
        ;;
  ⋮
パターンn)
        コマンド列n
        ;;
esac
```

「case」の次に書かれた文字列がパターン$_1$と一致すればコマンド列$_1$を、そうでなくてパターン$_2$と一致すればコマンド列$_2$を、… というように実行する。各「コマンド列$_i$」の後ろの「;;」は、コマンド列の終わりの目印として必要なものである (これがないと、シェルはどこまでが「コマンド列$_i$」なのかわからないため) ので、忘れずにつけること (ただし、case 文以外では「;;」は**必要ない**)。「;;」の直前の改行はなくても (あっても) よい。

「パターン$_i$」にはワイルドカード (0.2.2 節) と同じものが使える[24]。従って

```
#!/bin/sh
echo -n "$1から$1.bakへコピーしますか(Y/N) " >&2
read YN
case "$YN" in
Y*)
        cp "$1" "$1.bak";;
y*)
        cp "$1" "$1.bak";;
*)
        echo 中止しました >&2;;
esac
```

のような書き方ができる。「それ以外の場合」が「*)」と書かれていることに注意。また、パターンをいくつか「|」でつなげて書くこともできる (これはワイルドカードにはない記法) ので、

[23]実は、改行なしの出力を行う方法はシステムによって違い、echo -n ではないこともある。4.3 節で別途考慮する。

[24]ただし、ワイルドカードと違って「指定できるのは既存のファイルの名前だけ」という制約はなく、ファイルが存在するかどうかとは関係ないマッチングが可能。

```
#!/bin/sh
echo -n "$1から$1.bakへコピーしますか(Y/N) " >&2
read YN
case "$YN" in
Y*|y*)
        cp "$1" "$1.bak";;
*)
        echo 中止しました >&2;;
esac
```

でもよい。ただし**この例に関する限り**、パターンを「`Y*|y*)`」でなく「`[Yy]*)`」と書いても同じことができ、わざわざパターンに「`|`」を使う必要はない。

さて、もう 1 つの改良として、リターンキーだけが (つまり空文字列が) 入力されたら、そうでないものが入力されるまで聞き直すようにしてみよう。それには次のようにする。

```
#!/bin/sh
while true; do
        echo -n "$1から$1.bakへコピーしますか(Y/N) " >&2
        read YN
        case "$YN" in
        '')
                ;;
        *)
                break;;
        esac
done
··· 以下、さっきのスクリプトと同様に変数 YN の値が Y か y で始まるかどうかで分岐
```

ここでは「`''`」が (長さ 0 の文字列をクォートしているので) 空文字列を表している。なお、この例では case 文の代わりに「`if [ "$YN" = '' ]; then ···`」で条件判断してもよい。

1.9 コマンド置換

コマンド行の中の「‘ ‘」(**バッククォート**) とそれで囲まれた部分は、「‘ ‘」の中のコマンドを実行した際の**標準出力** (ただし最後の改行は削除される) に置き換えられる (シングルクォート「' '」と見かけは似ているが、意味は**全く違う**ので注意。毎年数名は間違える人がいる)。例えば、以下のスクリプト

```
#!/bin/sh
A=‘expr 3 + 5‘                  # A=‘expr 3+5‘ ではだめなので注意
echo 3と5の和は $A です
```

は「3と5の和は 8 です」と出力する。なぜなら、「‘ ‘」の中のコマンド「expr␣3␣+␣5」が標準出力へ「8」と出力する (試してみよ) ため、2 行目のうち「‘expr 3 + 5‘」の部分全体が「8」に置き換えられ、2 行目は「A=8」と書いてあるのと同じことになるので、シェル変数 A には「8」が代入されるからである。

ここで使った expr (表 0.3) は初登場のコマンドで、A.3 節にも示すように、数値の演算 (整数に限る) や正規表現 (→A.1) による文字列切り出しなどが行える (この資料では整数の演算に用いる機会が多い[25])。

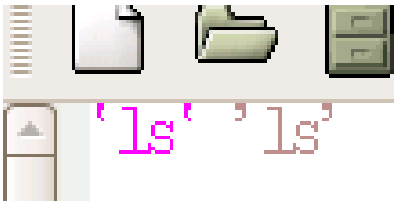
なお、先のスクリプトは、下 2 行の代わりに「`echo 3と5の和は ‘expr 3 + 5‘ です`」とも書ける。

> バッククォート「‘ ‘」で取り出せるものは、コマンドの「戻り値」(1.3.1 節) とは**関係ない**。「‘ ‘」はコマンドの**標準出力**を取り出すものであるが、戻り値は標準出力には出てこないからである。ここを混同する人が毎年 1～2 人はいるので、要注意。

[25] シェルの種類によっては、expr を使わずに整数演算を行える機能を持つものもある (4.4.2 節参照)。ただし、その機能は全てのシェルに共通ではない (非常に古いシステムにはない場合がある) ので、その機能を使ったシェルスクリプトは若干汎用性が落ちる。

では、バッククォートを使った下記左のようなスクリプト[26]は、どういう動作をするだろうか? (実際に試した上で、なぜその動作をするのか考えよ)

```
#!/bin/sh
A=1
while [ $A -le 100 ]; do
        echo $A
        A=‘expr $A + 1‘
done
```



参考: G 棟システムの Emacs 画面でのバッククォート (左) とシングルクォートの見え方の違い。シェルスクリプトモード (1.3.1.2 節) の場合は色でも見分けられる

1.9.1　ダブルクォート中のバッククォート

「‘ ‘」が「" "」の中で特別な意味を失わない (1.7 節) ことを利用して、実際に「" "」の中で使う場合がある。

```
#!/bin/sh
# 第1引数のファイルがJPEGファイルかどうかで場合分け
case "‘file -b "$1"‘" in
*JPEG*)
        echo "$1"はJPEGファイル;;
*)
        echo "$1"はJPEGファイルにあらず;;
esac
```

file コマンドもここで初登場で、引数に指定したファイルが何であるかを推測するコマンド。例えば

```
$ file -b aaa↵
JPEG image data, JFIF standard 1.01
```

この例では、ファイル aaa は JPEG ファイル (画像ファイルの一種) と推測されている[27]。

そこで、例えば $1 が aaa とすると、‘file -b "$1"‘ は上記 file コマンドの出力「JPEG image ··· 1.01」に置き換えられる。それを、空白などの特殊文字の影響なしに 1 つの文字列として case 文で使うために、「" "」で囲んでいる[28]。これによって、上記の case 文は「case "JPEG image ··· 1.01" in ···」とするのと同じことになる。この「" "」内の文字列が「*JPEG*」という形をしていれば (すなわち「JPEG」を含んでいれば) JPEG ファイルと判定するのが、上記のスクリプトである。

逆に、下記の例のような場合は「‘ ‘」を「" "」内に入れてはならない。

```
#!/bin/sh
cp ‘fgrep -l "$1" *.c‘ sub
```

この例は、.c で終わる名のファイルのうち第 1 引数の文字列を含むようなものを、sub というディレクトリにコピーするように作ったものである (fgrep の -l オプションは、指定した文字列をファイル中に発見した場合、ファイル名のみを出力する効果がある)[29]。もしこれを「cp "‘fgrep -l "$1" *.c‘" sub」のようにすると、例えば fgrep コマンドが abc.c と def.c を見つけた場合、cp "abc.c def.c" sub にあたる操作をすることになるが、これは望む動作ではない。

---

[26] この例では変数 A の値は決して特殊文字を含まないため、「$A」を「" "」で囲む必要はないので、囲んでいない。

[27] ここで、file コマンドに -b オプションを付けなければ、出力の先頭にファイル名が付く。1.9.1 節冒頭のシェルスクリプトでは、そうするとファイル名にたまたま「JPEG」という文字列が含まれていた場合に誤判定を起こすため、file コマンドに -b オプションを付けてそれを防いでいる。

[28] 「‘ ‘」がコマンドの引数内で使われる場合と、case 文の「case」の次で使われる場合とでは、特殊文字がある場合の影響が若干違うのだが、いずれにせよその影響による不測の事態を防ぐため、「" "」で囲んでおく方がよい。

[29] このスクリプトは、ファイル名に特殊文字を含むファイルがあると誤動作することがある。例えば、fgrep コマンドが a␣bc.c と def.c というファイルを見つけた場合、「cp a bc.c def.c sub」を実行してしまう。なので、そのような場合にはこのスクリプトの手法は適さない。

1.10　途中での終了

exit コマンドを使えば、シェルスクリプトをその時点で終了することができる。例えば、1.3.2 に出てきたシェルスクリプトの break を exit に置き換えて

```
#!/bin/sh
while true; do
        if [ -e waaiwaai ]; then
                exit
        fi
        echo hahaha; sleep 1
done
echo finished
```

とすると、waaiwaai ファイルの存在を確認した段階で、while ループから抜けるのではなくシェルスクリプトの実行そのものを終了する。従って、最後の「`echo finished`」の文は決して実行されない。

また、例えばシェルスクリプトの先頭部が

```
#!/bin/sh
if [ $# -gt 2 ]; then
        echo '引数が多すぎます' >&2
        exit 1
fi
  ⋮  (以下続く)
```

のように始まっていると、`$#` (これは特殊なシェル変数で、シェルスクリプトへ渡された引数の個数を表す) が 2 より大きければ、「引数が多すぎます」というメッセージを標準エラー出力に出して、戻り値 1 で終了し、そうでなければ続きを実行する。このようにして、シェルスクリプト内でコマンド引数のエラーチェックなどを行うことができる。しかもその場合は、戻り値が 1 (すなわち非 0) になるので、このシェルスクリプトを実行した側では、このシェルスクリプトがエラー終了したものとして扱うことができる。

第 2 章　AWK の活用

AWK は、プログラミング言語のうちでも「スクリプト言語」と呼ばれるものの 1 つ。「スクリプト言語」には他に、Perl, Python, Ruby, Tcl/Tk, sh などもある。AWK 言語は特に、フィルタ (0.2.3 節) としての利用を意識して設計されている言語で、標準入力あるいはファイルからの入力に対し、文字列の置換などさまざまな加工を施して出力する。対話的にも、シェルスクリプトの中でも多用するので、本資料でも取り上げる。

2.1　使い方の基本

AWK の基本的な使い方は

`$ awk␣'プログラム'␣ファイル名1␣ファイル名2␣···` ↵

である[30]。指定された名前の各ファイルが「プログラム」に従って順次読み込まれて処理される。

特に、「ファイル名」を **1 つも指定しない** 場合は、標準入力が「プログラム」に従って順次読み込まれて処理される。従って

[30] ちなみに、言語の名前は大文字の「AWK」だが、その処理系であるコマンドの名前は小文字の「awk」である。

```
$ 他のコマンド | awk␣'プログラム'↵
```

のようにすれば、他のコマンドの出力をパイプから読んで処理することができる。

ここで、プログラムは

パターン$_1$ {アクション$_1$}
パターン$_2$ {アクション$_2$}
⋮
パターン$_n$ {アクション$_n$}

の形。AWK は、指定されたファイルまたは標準入力から**1 行読んでは**、その行が各「パターン$_i$」($1 \leq i \leq n$) に合っているかチェックし、合っていれば「アクション$_i$」を行う、ということを、読み終えるまで繰り返す。

もし、ファイルまたは標準入力から読んだある行が、**複数のパターン**に合っていれば、それぞれのパターンに対応するアクションが実行される。例えば、ある行を読んで、その行がパターン$_2$ とパターン$_3$ の 2 つに合っていたら、アクション$_2$ とアクション$_3$ の両方が実行され[31]、それが終わってから次の行が読まれる。

ちなみに、プログラムには (シェルにとっての) 特殊文字 (「$」「"」「>」など) が頻出するので、プログラム全体を「' '」で囲んで (シェルにとっての) 特別な意味を失わせる。

2.1.1 例

まず、準備として適当なディレクトリにて

```
$ ln -s ~nide/jugyo/jjikken1-14/data/access_log .↵
```

としておこう。これにより、そのディレクトリに access_log というファイルが作られる。このファイルは「シンボリックリンク」[32]というもので、「~nide/jugyo/jjikken1-14/data/access_log というファイルを、カレントディレクトリの access_log というファイル名でも参照可能にする」という働き (言い換えれば「ファイルに別名を付ける) がある。よって、例えば「less access_log↵」により、~nide/jugyo/jjikken1-14/data/access_log ファイルを less で閲覧できるようになる。

> ちなみに、~nide/jugyo/jjikken1-14/data/access_log というファイルは、とある期間の、情報科学科の Web サイトのアクセスログを取ってきたもの。かなり大きいファイルなので、コピーはせずに、上記の操作で「シンボリックリンク」を作ること! ディスク領域の節約にご協力下さい。

下記は、そのファイルの (長い) 内容のうち、中程の 2 行を参考のため例示したものである (長い行を折り曲げているので、4 行に見えるが実際は 2 行)。

```
gpc1w10.e.ics.nara-wu.ac.jp - - [18/Jul/2013:11:17:25 +0900] "GET /ics-only/
keisanki/etiquette.html HTTP/1.1" 200 5216
gpc1w10.e.ics.nara-wu.ac.jp - - [18/Jul/2013:11:17:33 +0900] "GET /ics-only/
keisanki/faq.html HTTP/1.1" 200 10104
```

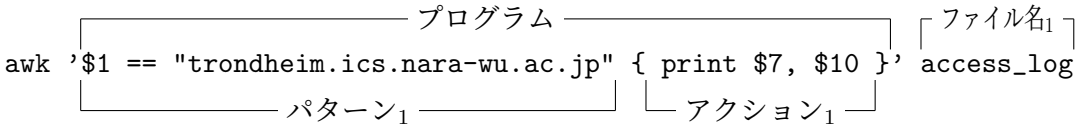
以上が済んだら、次を試してみよう。

```
$ awk␣'$1 == "trondheim.ics.nara-wu.ac.jp" { print $7, $10 }'␣access_log↵
```

この例では「$1 == "trondheim.ics.nara-wu.ac.jp" { print $7, $10 }」が AWK のプログラムにあたる。またその中で、「$1 == "trondheim.ics.nara-wu.ac.jp"」がパターン$_1$ に、「print $7, $10」がアクション$_1$ にあたる。そこで、access_log ファイルの各行のうちでパターン$_1$ に合うものに対し、アクション$_1$ を行うことになる。

[31]ただし、アクション内に next (→3.7.1 節) がある場合はこの限りではない。

[32]Windows でのショートカットにおおむね相当するものである。なお、http://www.ics.nara-wu.ac.jp/ics-only/pdf/file-reduction.pdf (本学学内からのみ閲覧可) の 3 章に、参考となる資料がある (09/05/20 現在)。



ここで、パターン$_1$やアクション$_1$の中の「$1」「$7」「$10」は、第 1·第 7·第 10 フィールドを表す。第 k **フィールド**とは、(ファイルや標準入力から読んだ) 1 行を空白で区切ったときの先頭から k 番目。

従って、上の例は、access_log の行のうち、第 1 フィールドが文字列「trondheim.ics.nara-wu.ac.jp」と一致するものについてだけ、その行の第 7·第 10 フィールドを出力する。なお、AWK プログラム内では「" "」で囲まれた部分が**文字列**を表す。

> フィールドを表す「$1」「$2」··· は、シェルスクリプトの引数を表す「$1」「$2」··· とは、書き方は同じでも**違う意味**なので注意。「' '」の中で使われているので、これらの「$」はシェルにとっての特別な意味を失っており (→1.7)、従って「シェルスクリプトの引数」と解釈されることはない。
> なお、上の例の「print $7, $10」を「,」なしの「print $7 $10」にすると、違う意味 (文字列連接) になる。本節では触れない。

さて、access_log の各行の第 1 フィールドは、Web サイトにアクセスしてきたホストの名前かアドレス、第 7 フィールドはアクセスされたファイルの名前[33]で、第 10 フィールドがそのサイズ。従って、上の例は結局、「trondheim.ics.nara-wu.ac.jp というホストが情報科学科の Web サイトにアクセスしてきて閲覧したファイルの名前とサイズをリストアップ」していることになる。なお、出力が長いので

```
$ awk '$1 == "trondheim.ics.nara-wu.ac.jp" { print $7, $10 }' access_log | less↵
```

のように less を併用すると見やすい。

ちなみに、AWK を (コマンド行で) 使う場合、コマンド行が長くなりやすいので、0.2 節のコマンド行編集やコマンドヒストリをフル活用するのがコツ。

[要注意] `$ awk'$1 == "trondheim.ics.nara-wu.ac.jp" { print $7, $10 }'␣access_log↵` のように、「awk」の**直後の空白**がないのは**だめ**。これでは、「awk$1 == ···$10 }」までがコマンド名と見なされ、「そんなコマンドはない」というエラーになるのである。「そんなミスするわけない」と思うかもしれないが、毎年 1～2 人位はこれで当授業の課題を不正解にしてしまうのだ。

また、ファイル名 (この場合は「access_log」) の**直前の空白**がないのも**だめ**。ないと、「access_log」はその前の引数「···$10 }」の続きと見なされ、AWK プログラムの一部と解釈されてしまうからである。

2.1.2 シェルスクリプト内での利用

もちろん、シェルスクリプト内で AWK を利用することも可能。例えば

```
#!/bin/sh
awk '$1 == "trondheim.ics.nara-wu.ac.jp" { print $7, $10 }' access_log | less
```

というシェルスクリプトは、先の例と同じことをする。さらに

```
#!/bin/sh
awk '$1 == "trondheim.ics.nara-wu.ac.jp" { print $7, $10 }' "$1" | less
```

のようなシェルスクリプトにすると、探索すべきファイル名を access_log に固定せず、シェルスクリプトへの第 1 引数で与えることもできるようになる。

> 「awk '···'」の中の「$1」「$7」「$10」は先述のように、AWK プログラム中での「第 1·第 7·第 10 フィールド」の意味である。一方、「awk '···'」の中にない「$1」(「" "」で囲まれたもの) は、「$」がシェルにとっての特別な意味を失っていないので「シェルスクリプトへの引数」である。**混同しないこと**。

[33] アクセスされたファイルが CGI ファイルである場合は、第 7 フィールドがファイル名そのものではなく、その後ろに「?」をはさんで CGI への引数をつけたものであることもある。細かい話だし本筋とも無関係なので、本資料では気にしない。

2.2　いろいろな使い方

2.2.1　正規表現

パターン$_1$ に「`$1 ~ /nara-wu/`」のように書くと、「`$1` が文字列 nara-wu を**含む**」の意味になる。例えば

```
$ awk '$1 ~ /nara-wu/ { print $1, $7 }' access_log
```

のようにする (と、何を出力していることになるだろうか?)。

この「`~`」の右辺の「`/ /`」で囲まれた部分は「**正規表現**」と呼ばれ、これを使うと、固定の文字列を含むかどうかだけでなく、文字列の任意回の繰り返しや選択などいろいろなパターンが書ける。例えば

```
$ awk '$1 ~ /\.(ac|go)\.jp$/ { print $1, $7 }' access_log
```

で、「`$1` が .ac.jp または .go.jp で終わる場合は $\cdots$」の意になる。「`ac|go`」は「ac または go」、「`$`」は文字列の末尾を意味する。「`.`」は正規表現では「任意の 1 文字」の意味になるので、そうさせずに「ピリオド 1 文字」の意味に解釈させるためには「`\`」でエスケープする。正規表現についての詳細は A.1 節参照。

ちなみに以下のように「`!~`」を使うと、「`$1` が文字列 nara-wu を**含まない**」という、逆の判断になる。

```
$ awk '$1 !~ /nara-wu/ { print $1, $7 }' access_log
```

正規表現が使えるコマンドやソフトや言語は、AWK 以外にも多いので、知っておくと得。ただし、コマンド類によって正規表現が微妙に違うことにも注意が必要。

2.2.2　複数のパターン

```
$ awk '$10 >= 20000000 { print $1, $4, $5, $10 } $7 ~ /\.mp3$/  (→ 同じ行に続けて)
    { print $1, $4, $5, $7 }' access_log
```

これは、パターン$_i$ と {アクション$_i$} の組が **2 つ**ある例である。この場合、access_log ファイルを 1 行ずつ読みながら、「第 10 フィールドが 2 千万以上なら第 1·4·5·10 フィールドを出力」「第 7 フィールドが .mp3 で終わっていれば第 1·4·5·7 フィールドを出力」という処理を行う。2.1 節でも述べたように、複数のパターンにマッチする行に対しては、複数のアクションが実行される。

また、この例は**数値**に対する演算 (ここでは比較) も使っている (整数だけでなく実数も使用可)。数の比較は、test コマンドのような「`-ge`」などではなく、通常のプログラミング言語のように「`>=`」などを使う。

2.2.3　パターン$_i$ やアクション$_i$ の省略

パターン$_i$ と {アクション$_i$} は**ペアで記述**するのが基本だが、**どちらか一方を略する**こともできる。

パターン$_i$ が略された場合、「無条件」と解釈され、アクション$_i$ は全ての行に対して実施される。例えば

```
$ awk '{ print $1, $7 }' access_log
```

は、パターン$_1$ が略された {アクション$_1$} だけがある例であり、アクション$_1$ が全ての行に対して実行されるので、全ての行の第 1, 7 フィールドを出力する。

また、{アクション$_i$} が略された場合、デフォルトのアクションとして、「その行をそのまま出力」というものが実行される。つまり、パターン$_i$ を満たす行が単にそのまま出力される。例えば

```
$ awk '$1 == "trondheim.ics.nara-wu.ac.jp"' access_log
```

は、{アクション$_1$} が略されたパターン$_1$ だけがある例であり、よって、パターン$_1$ を満たす行、つまり第 1 フィールドが trondheim.ics.nara-wu.ac.jp であるような行を全てそのまま出力する。

上の例は

```
$ awk '$1 == "trondheim.ics.nara-wu.ac.jp" { print $0 }' access_log
```

とも書ける。「`$0`」は「第 0 フィールド」ではなく「行全体」の意味になるため。また、アクション$_i$ の中の「`print $0`」は「`print`」と略することもできるため、上の例はさらに次のようにも書ける。

```
$ awk '$1 == "trondheim.ics.nara-wu.ac.jp" { print }' access_log
```

2.2.4 複雑な条件式の利用

パターン$_i$の中には、C 言語と同様に「&&」(かつ) や「||」(または)、「!」(否定) などが使える[34]。次はその例。(何をするものか分かるだろうか?)

```
$ awk '$1 ~ /nara-wu/ && ($7 ~ /\.pdf$/ || $10 >= 500000) { print $1, $7 }' access_log↵
```

2.2.5 フィールド区切り文字の変更

フィールドが空白でない文字で区切られているような入力を処理したい場合がある。その場合は「awk␣**-F区切り文字**␣'プログラム'␣···」のようにする。以下はその例。

```
$ getent passwd | awk -F: '$1 == "wd" { print $1, $6 }'↵
```

getent passwd コマンドは、そのシステムに登録されているユーザの一覧を表示するもの[35]で、各行は「:」で区切られ、その第 1 フィールドがユーザ名、第 2 フィールドが暗号化されたパスワード、第 6 フィールドがホームディレクトリ、などとなっている (「getent passwd↵」を単独で実行して試してみよう)。そこで上の例は、wd というユーザのユーザ名とホームディレクトリを出力することになる。

また、ここでは awk コマンドの引数としてファイル名を指定していないので、AWK は標準入力を読んで処理 (→2.1 節) する、すなわち getent コマンドの出力をパイプから受け取って処理していることにも注意。

2.2.6 計算

AWK で合計の計算などもできる。例えば

```
$ awk '$1 ~ /nara-wu/ { total += $10 } END { print total }' access_log↵
```

で、第 1 フィールドに nara-wu を含むような全ての行の第 10 フィールドの合計を出力。これもパターン$_i$と{アクション$_i$} の組が 2 つある例だが、パターン$_2$の「END」は特別なパターンで、どの行にもマッチしない代わり、全ての行の処理が終わった後に、END パターンのアクションが 1 回だけ実行される。

また、total は (AWK の) 変数 (変数名は「total」でなくてもよい)。AWK では、**未代入の変数**は値として 0 (数値として) あるいは空文字列 (文字列として) を持つため、変数 total を明示的に 0 で初期化しなくても正しく合計が計算できる。

> AWK の変数は、**AWK プログラム内だけ**のものであり、**シェル変数とは全く別**の存在。混同せぬよう。

各行の第 10 フィールドを出力しつつ、その合計も出すには、**アクション$_i$ の中に複数の文が書ける**ことを使う。その場合、文同士は**「;」で区切る**。例えば

```
$ awk '$1 ~ /nara-wu/{print $10; total += $10} END{print "total = ", total}' access_log↵
```

のようにする。同じことをシェルスクリプト内で行う場合は、以下のように AWK プログラム内で改行してインデントすると見やすくなる。**アクション$_i$ の中**の文同士を区切るのは「;」でなく**改行**でもよいので、シェルスクリプト中で AWK を使う場合はそうすることが多く、下記のプログラムでもそうしている。

```
#!/bin/sh
awk '
        $1 ~ /nara-wu/ { # 第1フィールドが「nara-wu」を含む
                print $10
                total += $10
```

[34]細かい話だが、対話に使っているシェルが C シェル系の場合、状況によっては「!」が特殊な解釈をされてしまうことがあるので、それを避けるために「\!」と打ち込むことが必要な場合がある。G 棟システムでは B シェル系を使っているので、このことは気にしなくてよい。

[35]システムのユーザ管理機構の種類によっては、getent ではなく ypcat というコマンドが使われる場合もある。

```
        }
        END{
                print "total = ", total
        }
' access_log
```

この例は、「' '」の中では改行文字も特別な意味を失うことを利用している。これにより、「awk」の次の「'」から「access_log」の前の「'」までが、(awk コマンドにとって) **改行を含む 1 つの長い引数**となり、ここが (2.1 節で述べた) awk の使い方の「'プログラム'」の部分に相当しているわけである。

ただし、パターンと、それと組になるアクションを始める「{」の間では**改行してはならない** (間違える人が例年多い!)。上の例では、「$1 ~ /nara-wu/」とその次の「{」の間、および「END」とその次の「{」の間がそれに該当する。もし「$1 ~ /nara-wu/」とその次の「{」の間で改行すると、『アクションが省略された「$1 ~ /nara-wu/」というパターン』と、『パターンが省略された「print $10; total += $10」というアクション』(2.2.3 節) が別々にあるものと見なされ、違う意味の AWK プログラムになるからである。

また、AWK プログラムを閉じる「'」とその次のファイル名 access_log の間も、**改行してはならない**。改行するとそこでコマンド行が途切れ、その次の access_log は別なコマンド名と解釈されるからである。

なお、AWK プログラム中でも「#」から行末までは**注釈**である。

> このような書き方をした場合、「' '」の中、つまり AWK プログラムの部分には、残念ながら Emacs の自動インデント (1.3.1.2 節) は**効かない**。シェルスクリプトにとっては「' '」の中は単なるクォートされた文字列であるため、そこにある (AWK プログラムにとっての) 構造は、シェルスクリプトモードにとっては「知らないもの」だからである。頑張って手動でインデントしよう。

ちなみに、END の反対の役目を持つ「BEGIN」という特別なパターンもある。BEGIN パターンのアクションは、入力を読み始めるよりも前に (つまり最初に)、1 回だけ実行される。以下は BEGIN パターンを用いた例。printf も用いている (AWK の printf は、C 言語のそれとほぼ同じ)。

```
#!/bin/sh
awk '
        BEGIN{
                print "  転送量  |  ホスト"
                print "----------|--------------------"
        }
        $1 ~ /nara-wu/ { # 第1フィールドが「nara-wu」を含む
                printf("%10d|%s\n", $10, $1)
                total += $10
        }
        END{
                print "----------|--------------------"
                printf("%10d|総和\n", total)
        }
' access_log | less
```

2.2.7　if 文、for 文などの制御構文

AWK のプログラムのうち、**アクションの中**では、C 言語と同様の if 文や for 文、while 文も使える[36]。例えば、2.2.6 節の最後のシェルスクリプトは、以下のようにも書ける。

```
#!/bin/sh
awk '
        BEGIN{
                print "  転送量  |  ホスト"
```

[36] さらに、C 言語と書き方がまったく違うもう 1 種類の for 文もある。3.7.1 節参照。

```
                print "----------|---------------------"
        }
        { # あえてパターンを省略したアクションとし、アクション内でif文を使った例
                if($1 ~ /nara-wu/){ # 第1フィールドが「nara-wu」を含む
                        printf("%10d|%s\n", $10, $1)
                        total += $10
                }
        }
        END{
                print "----------|---------------------"
                printf("%10d|総和\n", total)
        }
' access_log | less
```

ただしこの例に限って言えば、2.2.6 節最後の例のように**パターンを使って書く方が自然**である。

2.2.8 AWK と他のコマンドの組み合わせ使用

sort コマンドは、入力行をアルファベット順 (文字コード順) に並べ替えて出力する (`sort -n` の場合は数値の順に並べ替える) ものであった。また uniq コマンドは、隣り合う行で同じものがあればそれを 1 つにまとめる (`uniq -c` の場合は隣り合う同じ行が何行あったか数えて出力する) ものであった。これらと AWK を併用した次の例は、上から順に何をするものだろうか。

```
$ awk '{ print $1 }' access_log↵
$ awk '{ print $1 }' access_log | sort↵
$ awk '{ print $1 }' access_log | sort | uniq -c↵
$ awk '{ print $1 }' access_log | sort | uniq -c | sort -n↵
```

これによって、Web サイトにアクセスしてきたサイト (access_log の第 1 フィールド) を、回数が少ない順に並べ替えて出力できる。

第3章　実例

本章では、もう少し進んだシェルスクリプトの作成例を、実例中心に取り上げる。

3.1 ディレクトリ下を再帰的に探しながらファイル中の文字列検索

fgrep コマンド (表 0.2) は

```
$ fgrep 文字列 ファイル名 ファイル名 ··· ↵
```

のように使い、ファイル内での文字列の検索ができる。これまでにも何度か登場した。ファイル名はいくつでも指定できるので、例えばワイルドカードを使って「`fgrep 文字列 ディレクトリ名/*`↵」のようにすると、そのディレクトリの中の全てのファイル (名前が「.」で始まるものは除く。ワイルドカードにマッチしないため) 内から、指定した文字列を検索できる。

> (注) バイナリファイルの中も検索してしまうため、バイナリファイルの中にたまたま「文字列」と一致する部分があったらその周辺を表示して端末の文字化けを起こす可能性があるので、その可能性を減らすために、試すときは (2～3 文字ではなく) 少し長めの文字列を使おう。以下でも同じ。

しかし、そのディレクトリの中にさらにディレクトリがあった場合、その中のファイルも検索したい、さらにそこにもディレクトリがあった場合はその中のファイルも ··· というように、指定したディレクトリ内を**再帰的に**探って、そこ以下の全ファイルに文字列検索をかけたいときはどうすればいいだろう。(最近の grep コマンド類には、単体でそれを行える `-r` というオプションを持つものがあるが、練習のためここではあえてそれを使わないものとする[37]。)

そのためにはまず、find と xargs という 2 つのコマンドについて知ろう。

3.1.1　find コマンド

find コマンド (表 0.3) は、指定したディレクトリ以下にあるファイル名 (ディレクトリも含む) を再帰的にリストアップし、1 行につき 1 ファイル名の形で出力するコマンド。例えば

```
$ find ディレクトリ名↵
```

で、指定したディレクトリの中にあるファイル名 (引数に指定したディレクトリ自身を含む) を全て表示する (ただし、ソートはされない)。指定したディレクトリの中にさらにディレクトリがあればそれも、さらにその中にあるファイルも ··· というように、再帰的に出力に含まれる。「`find ~`↵」などとして試してみよう (「~」がホームディレクトリを表すことに注意)。

ディレクトリ名の指定より後に、検索するファイルの条件を記述することもでき、その場合は、それに当てはまるファイルの名前しか表示されない。詳しくは A.4 節参照。例えば

```
$ find ディレクトリ名 -type f↵
```

とすると、指定したディレクトリの中を再帰的に探すが、見つかったもののうち (ディレクトリやシンボリックリンクなどの特殊ファイルでない) 通常ファイルについてのみ、その名前を出力する。「`-type f`」の部分が「通常ファイル」という条件の記述になっている。

3.1.2　xargs コマンド

xargs (表 0.3) は、引数に指定したコマンドに、標準入力から読んだものを (空白や改行で区切って) 引数として与えて実行する。例えば

```
$ echo abc def | xargs cat↵
```

とすると、xargs の標準入力には「`abc def`」が入ってくるので、これを空白で区切った「`abc`」「`def`」を cat コマンドに引数として与えて実行する。つまり

```
$ cat abc def↵
```

としたのと同じことになる (適切なファイル名を与えて試してみよ)。同様に、

```
$ echo abc def | xargs fgrep fuhyo↵
```

とすると、`$ fgrep fuhyo abc def`↵ としたのと同じことになる。

> 本節の範囲では気にする必要はないが、xargs コマンドにはもう 1 つ役割がある。
>
> UNIX では、コマンド行の長さには上限がある (システムによって異なるが、典型的には数万バイト程度か、あるいはそれ以上)。そのため、xargs の標準入力に非常に大量の入力が渡された場合、xargs は、指定されたコマンドに、標準入力から読んだもの全てを引数として与えて実行しようとしても、この上限に引っかかって実行できない場合がある。そのような場合、xargs は自動的に、渡された入力をいくつかのパートに区切って、
>
> - 最初は、指定されたコマンドに、標準入力のうちパート 1 を引数として与えて実行
> - 次には、同じコマンドに、標準入力のうちパート 2 を引数として与えて実行
>
> というように、指定されたコマンドを複数回に分けて実行してくれるようになっている。

[37] また、後に 3.1.5 節で言及するように、本節で述べる方法には、`-r` オプションにない利点もある。

3.1.3 find と xargs の併用

そこで、探索したいディレクトリの中にあるファイルの名前を find でリストアップさせ、それを xargs にパイプで渡して fgrep コマンドの引数としてやれば、目的のことができるわけである。つまり

```
$ find ディレクトリ名 -type f | xargs fgrep -H 文字列↵
```

とすればそれができるだろう。

ただしここで、先程の説明にはなかった、fgrep コマンドの「-H」というオプションを指定した[38]。その理由を説明する。

「-H」オプションを**指定しない場合**、fgrep コマンドは、引数にファイル名を**複数指定**したときとそうでないときで、ちょっと動作が異なる。

```
$ fgrep include a.c b.c↵        (ファイル名を複数指定した)
a.c:#include <stdio.h>          (行頭にファイル名が表示される)
b.c:#include <stdlib.h>         (同上)
$ fgrep include a.c↵            (ファイル名を 1 つだけ指定した)
#include <stdio.h>              (行頭にファイル名が表示されない)
```

複数のファイルを指定した場合、探し当てた行がどのファイルから見つかったのものかを各行頭に表示している。しかし、ファイルを 1 つ以下しか指定しない場合は、出力の行頭にファイル名が表示されない。

これに対し、「-H」オプションを指定すると、

```
$ fgrep -H include a.c b.c↵
a.c:#include <stdio.h>
b.c:#include <stdlib.h>
$ fgrep -H include a.c↵
a.c:#include <stdio.h>          (ファイル名が 1 つでも、行頭にファイル名が表示される)
```

指定したファイルが 1 つでも複数でも、必ず出力の行頭にファイル名が表示される。

このため、

```
$ find ディレクトリ名 -type f | xargs fgrep 文字列↵
```

としたのでは、もし、find が**たまたま 1 つ**しかファイル名を出力しなかった場合、fgrep にもファイル名が 1 つしか渡らないので、fgrep は行頭にファイル名を表示しないことになる。しかしこれでは、人間にはどのファイルから見つかったものか分からない (人間は直接にはファイル名を指定していないので)。これに対し

```
$ find ディレクトリ名 -type f | xargs fgrep -H 文字列↵
```

とすると、find が見つけたファイルがたまたま 1 つであっても複数であっても、出力の行頭には必ずファイル名が表示されるので、どのファイルから見つかった行か人間にわからないということは起きない。

> ただし、非常に古い fgrep コマンドにはオプション -H がないことがある。その場合、ファイル名として 1 つ余分に「/dev/null」を指定するという代替手段により、同じ効果を実現できる。なぜなら、/dev/null は、そこから読み出す際には空のファイルとして扱われる (読み出しが直ちに EOF になる) 特殊ファイルだからである[39]。従って、このファイルを余分に指定しても余計な出力は増えず、また、本来のファイル名引数に加えて /dev/null を指定することで、常に fgrep の引数にファイルを複数個指定することになり、必ず出力の先頭にファイル名が出力される。G 棟システムではこの代替手段は必要ない。

では、これをシェルスクリプトにしてみよう。rfgrep というシェルスクリプトを以下の内容で作る。

```
#!/bin/sh
find "$2" -type f | xargs fgrep -H "$1"
```

このシェルスクリプトを

```
$ rfgrep 文字列 ディレクトリ↵
```

として使うと、与えたディレクトリ以下の全ファイルから、指定した文字列を検索する。

---

[38] あくまで **fgrep コマンドの** -H オプションであることに注意。xargs と組み合わせるなら何でも -H を指定するわけではない。

[39] 0.2.3.1 節では、/dev/null のもう 1 つの性質「そこへ**書き込んだ**内容は黙って捨てられる」を紹介した。ここではそれとは違い、**読み出すとき**の性質を使っていることに注意。

3.1.4　空白を含むファイル名への対処

しかし、実は以下のようなスクリプトの方がもっと良い[40]。

```
#!/bin/sh
find "$2" -type f -print0 | xargs -0 fgrep -H "$1"
```

それはなぜだろうか。実は

```
#!/bin/sh
find "$2" -type f | xargs fgrep -H "$1"
```

というスクリプトだと、find が見つけたファイル名の中に、**空白文字**を含むファイル名がある場合に困るのである[41]。その場合、xargs はそれを空白文字で切って、fgrep には別々な引数として渡してしまう。

find の検索条件の後に「`-print0`」を書いておくと、それは条件とは見なされず (常に真となり)、副作用として、find は見つけたファイル名を改行で区切るのではなく、ヌル文字 (C 言語でいう「`'\0'`」) で区切るようになる。例えば、find が見つけたファイル名が「`a␣b.c`」と「`c.tex`」の場合、「`-print0`」なしでは

'a'	'␣'	'b'	'.'	'c'	'\n'	'c'	'.'	't'	'e'	'x'	'\n'

という出力になるが、「`-print0`」ありだと

'a'	'␣'	'b'	'.'	'c'	'\0'	'c'	'.'	't'	'e'	'x'	'\0'

という出力になる (正確な扱いについては A.4 節参照)。

一方、xargs に `-0` オプションを付けると、標準入力を空白や改行ではなくヌル文字で区切ってコマンドに渡す。従って

```
#!/bin/sh
find "$2" -type f -print0 | xargs -0 fgrep -H "$1"
```

というスクリプトなら、find が見つけたファイル名にたとえ空白文字が混じっていても、間違ってそこで切られることはない。また、ヌル文字は、ファイル名の中には含まれないことが保証されるので、ファイル名が意図しない場所で切られることはなくて済む。

3.1.5　複数のディレクトリを探させる

さて、先の rfgrep シェルスクリプトは、ディレクトリを 1 つしか指定できない。そこで、もう一工夫して

`$ rfgrep␣`文字列`␣`ディレクトリ$_1$`␣`$\cdots$`␣`ディレクトリ$_n$↵

とすると、それら全てのディレクトリ内のファイルから、指定した文字列を検索するようにできるだろうか。

find は複数の探索ディレクトリを指定できるので、シェルスクリプトの引数に複数のディレクトリが指定されても、それらをまとめて find に渡せばよい。しかし

```
#!/bin/sh
find "$@" -type f -print0 | xargs -0 fgrep -H "$1"
```

というスクリプトではだめである。例えば

`$ rfgrep` *string directory1 directory2*↵

[40] ただし、システムによっては find で `-print0` が使えなかったり、xargs に `-0` オプションがなかったりする。

[41] 特に、現行の G 棟システムのマシンでは、Windows のファイルも Linux のホームディレクトリの下のどこか (世代によって違うが、例えば windows10 ディレクトリ以下) に置かれているので、ほとんどの利用者は、ホームディレクトリの下のどこかに、ファイル名に空白文字を持つようなファイルがたくさんある。「`find "$HOME" -name '* *'`↵」として試してみよう。

とした場合、「"$@"」はシェルスクリプトの引数全てであるから「*string directory1 directory2*」と同じことになる。しかし、1 つめの「*string*」は探したい文字列であってディレクトリ名の指定ではないので、これを find に渡してはいけない。

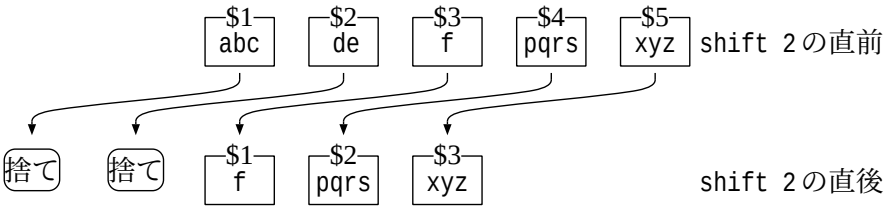
そこで、shift というシェル組み込みコマンドを使おう。このコマンドは、シェルスクリプトの引数を「ずらす」働きをする。以下は shift コマンドを使った、説明用のあまり意味のないシェルスクリプトの例。

```
#!/bin/sh
echo before shift: "$@"
shift 2
echo after shift: "$@"
```

このシェルスクリプト (名前を shifttest とする) の実行は次の例のようになる。

```
$ shifttest abc de f pqrs xyz↵
before shift: abc de f pqrs xyz
after shift: f pqrs xyz
```

この場合、`shift 2` は以下のような働きをする。



すなわち、shift コマンドはその引数の個数だけシェルスクリプトの引数をずらす[42]。また、引数を与えず単に `shift` とすると、`shift 1` と同じ動作、すなわちシェルスクリプトの引数を 1 つずらす。

そこで、rfgrep シェルスクリプトに与えられた引数を shift で 1 つずらせば、残りの全引数はディレクトリ名だから、それを find に与えればよい。しかし、

```
#!/bin/sh
shift
find "$@" -type f -print0 | xargs -0 fgrep -H "$1"
```

というスクリプトではやはりだめである。shift した時点で、スクリプトへの元々の第 1 引数、つまり探させたい文字列は捨てられてしまい、その後での "$1" は元々の第 2 引数、つまり引数に与えた探索ディレクトリのうちの最初のものになってしまっている。

ではどうすればよいだろうか? 考えて、rfgrep シェルスクリプトを完成させてみよう。

また、こうして完成させた rfgrep コマンドを

```
$ rfgrep␣文字列␣ディレクトリ1␣···␣ディレクトリn␣-name␣'*.c'↵
```

のように起動すると何が起こるだろうか、考えてみよ。(このようなことは、3.1 節で述べた grep コマンド類の `-r` オプションだけではできないので、find と xargs を組み合わせて実現することの利点の 1 つである。)

3.1.6 別な実現法

探索対象とするファイル (find コマンドによって見つけられるファイル) があまり多くなく、かつそのファイル名に空白や「*」など特殊文字が混じらないことがわかっている場合は、rfgrep コマンドは 1.9 節の**バッククォート**を用いた次のシェルスクリプトでも実現できる (ここではディレクトリ名は第 2 引数に 1 つだけ指定するものとする)。

[42] このとき、シェル変数 `$#` (1.10 節) の値もその分だけ減る。

```
#!/bin/sh
fgrep -H "$1" ‘find "$2" -type f‘
```

しかし、find コマンドで見つかるファイルが非常に多い場合、このスクリプトではうまく動かない。バッククォートの働きによって、find が見つけたファイル名全てを fgrep の引数にしようとするが、それが極端に多いと UNIX でのコマンド行の長さの上限 (3.1.2 節) に引っかかるためである。

また、find コマンドが見つけるファイル名の中に**特殊文字**がある場合もうまくいかない。見つかったファイル名がバッククォートの働きによって fgrep の引数に渡される際に、空白や「*」など特殊文字が特別な扱いを受けてしまうためである。

それらの場合には、これまでに述べたように find と xargs を組み合わせて実現すべきである。

3.2　他のコマンドから受け取った PostScript ファイルを gv で表示

3.2.1　PostScript とは

PostScript とは「ページ記述言語」と呼ばれるプログラミング言語の 1 種で、紙面に文字や画像をどのように配置するかを記述する用途に特化された言語である。PostScript プログラムの形で記述された文字・画像データを、PostScript 対応のプリンタに送ると、その画像がプリントされる (G 棟システムのプリンタは PostScript 対応である)。また、PostScript プログラムの描画結果をディスプレイ上で見せてくれるプレビューアである、gv というコマンド[43]もある (ちなみに gv は内部で、フリーソフトの PostScript 処理系である ghostscript を動かしている)。

例えば、a.ps というファイルに図 3.1 の左の内容を書き込み (これが PostScript プログラムである)、そして「`gv a.ps`↵」としてプレビューすると、同図右の画像が得られる (外枠や点線、および数字は説明用のもので、実際の出力の一部ではない)。また、「`lpr a.ps`↵」とすると同じものが印刷される (紙がもったいないのでお薦めはしない)。ちなみに、座標の長さの単位は 1 ポイント (=1/72 インチ ≒ 0.35mm) である。

```
%!
% PostScriptのプログラムは「%!」で始める約束
200 200 moveto  % ペンを座標(200,200)の点へ
400 400 lineto  % そこから座標(400,400)の点まで
                % 直線を引く
stroke          % 以上を実際に描画せよ
showpage        % ページ全体を出力
```

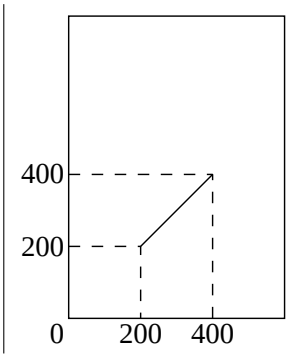


図 3.1: PostScript のプログラム (左) とその実行結果の例

人間が PostScript 言語で直接プログラムを記述することはあまりない。大抵は、文書作成ソフトや描画・画像編集ソフトなどが出力の 1 形式として PostScript プログラムを出力し、これをプリンタに送るという形態をとる。(本資料でも、人間が PostScript プログラムを書くことは目指さない。)

3.2.2　一時ファイルを使った処理

さて、画像を PostScript 形式で標準出力に出力できるコマンドは多数ある。例えば、文書作成ソフト LaTeX の清書結果 (DVI ファイル) を PostScript に変換するコマンド dvips、グラフの描画結果を PostScript でも出力できるソフト gnuplot (3.4 節に登場)、画像ファイルの形式変換用コマンドで PostScript への変換も可能な convert などがある。しかしここでは、よりお手軽な実例として、`~nide/jugyo/jjikken1-14/sample/`

[43] PostScript のプレビューアのコマンド名はシステムによって違い、ggv などのコマンド名の場合もある。

enmoyou というコマンドを用意しておいたのでこれを使うとしよう。このコマンドは 3 つの引数を取り、第 1・2 引数は正整数、第 3 引数は正実数である。このシェルスクリプトをコピーしてきて、例えば

```
$ enmoyou 16 19 0.9↵
```

としてみよう。PostScript プログラムが標準出力に出るだけなので、人間の目には一見わからない出力が延々と出る。そこで、

```
$ enmoyou 16 19 0.9 > out.ps↵
$ gv out.ps↵
```

としてプレビューア gv で見てみよう。きれいな模様が描かれているのが見えただろうか[44]。

しかし、ここで若干不便な点がある。gv は、標準入力から PostScript プログラムを受け取ってプレビューしてみるということができない[45]。従って、ここでは出力をいちいち一時ファイルに入れてから gv でプレビューせざるを得なくなっている。

そこで、標準入力から PostScript ファイルを受け取ってプレビューできるようなシェルスクリプトを作ってみよう、というのがこの節の本題である。

そのようなシェルスクリプトの作り方としては

1. 標準入力から受け取った内容をいったん cat で一時ファイルに溜める
2. それを gv で表示
3. 終わったら一時ファイルを消去

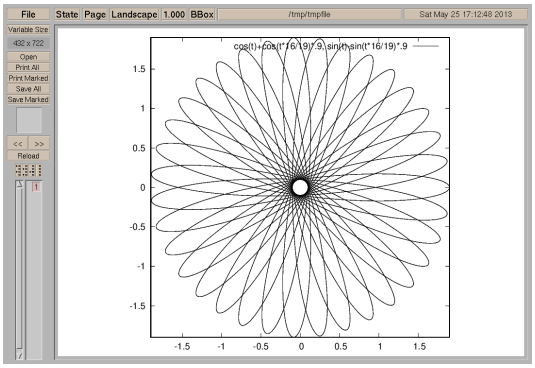
とすればよい。UNIX には一時ファイル用に /tmp というディレクトリがあるので、そこに一時ファイル /tmp/tmpfile を作ることにしてみよう。一時ファイルの名前は何度も使うので、シェル変数に入れて使うのがよい。

```
#!/bin/sh
TMPFILE=/tmp/tmpfile
cat > $TMPFILE
gv $TMPFILE
rm $TMPFILE
```

このシェルスクリプトの名前を gvfilt とすると、例えば

```
$ enmoyou 16 19 0.9 | gvfilt↵
```

のようなことが可能となる (右図)。



3.2.3 問題点

実は、一時ファイルの名前がいつも同じでは問題がある。例えば、自分が同じスクリプトを別々の端末から **2 つ並行に**使うと、それぞれが /tmp/tmpfile に違う内容を書こうとして干渉しあうことがあるし、また、自分と他人が同じシステムで**同時に**そのスクリプトを使おうとすると、他人が先に /tmp/tmpfile を作成しているため自分が同ファイルに書き込めず、正常動作できないこともある。

そこで、できるだけ毎回違う一時ファイルを作る工夫として、かつてよく用いられたのが、$LOGNAME や $$ という特殊シェル変数 (前者は自分のログイン名、後者はシェルのプロセス番号が入っている) を使って一時ファイルの名前を作る方法である。例えば、先のスクリプトの「TMPFILE=/tmp/tmpfile」の箇所を

[44] enmoyou に与える第 1・2 引数は互いに素とし、その 2 つの比が 1 からあまり離れないこと、第 3 引数は 1 からあまり離れない実数を与えることが、きれいな模様にするコツである。

[45] 実は、最近の gv では (引数に「-」を指定すれば) できてしまう (この資料を執筆していた当時はできなかったのである)。ここでは、できないものと仮定して読んでほしい。

```
TMPFILE=/tmp/tmpfile_${LOGNAME}_$$
```

のように変える (「$LOGNAME_」の部分が「『LOGNAME_』という名前のシェル変数の値」と解釈されないよう、「${LOGNAME}_」と書いている。1.6 節参照)。こうすると、プロセス番号は実行毎に異なるし、ログイン名も他人とは違うので、一時ファイル名がたまたまかち合って問題を起こすことは防げる。

しかし、これでも問題は残る。悪意のあるユーザが、他人が作成する一時ファイルの名前を予測し、

1. その内容を覗き見する
2. それと同じ名前のファイルを先回りして作成し、スクリプトの実行を妨害したり、意図しない処理を実行させたりする

などの攻撃は依然として可能だからである。

3.2.4　mktemp コマンドの利用

近年のシステムは、3.2.3 節で述べた問題が起きないよう安全に一時ファイルを作るための、mktemp というコマンドを備えることが多い。これを使って、先の gvfilt スクリプトをより安全に書き直してみよう。

mktemp コマンドは以下のように使う。引数の最後に X が 6 つ以上付いている必要がある。

```
$ mktemp /tmp/適当な名前XXXXXX↵
```

mktemp は、この X の部分を (既存のファイルと重ならないように) 適当な文字に置き換え、一時ファイルを (他人が読み書きできないようなモードで) 新たに作った上で、そのファイル名を出力してくれる。この一時ファイル名を後で使いたいので、シェルスクリプト内で

```
TMPFILE=‘mktemp /tmp/適当な名前XXXXXX‘
```

のようにしよう (1.9 節の「バッククォート」を利用している)。ただし、mktemp が何らかの理由で失敗した場合についても考慮して、実際には

```
TMPFILE=‘mktemp /tmp/適当な名前XXXXXX‘ || {
        エラーメッセージを標準エラー出力に出す
        戻り値 1 で終了
}
```

のようにする。それ以後の部分は元のスクリプトと同様でいい。gvfilt をそのように書き直してみよう。

> ここでは 1.4.3 節で述べた「{ }」と「||」の併用を使っている。ただし、「||」の左の部分が真か偽かは、この部分のうち最後に実行されたコマンドの真偽で決まる。最後に実行されるコマンドは「‘ ‘」の中身なので、ここが真かどうか (つまり、mktemp が成功したかどうか) によって「||」の左の部分全体が真と扱われるかどうかが決まる。そして、それが偽である場合に「||」の右の部分が実行される。よって、mktemp コマンドが偽を返した場合のみ、「{ }」の内部が実行されることになる。

3.2.5　シェルスクリプトが中断した場合の一時ファイルの後片付け

さて、もう 1 つ別な問題がある。一時ファイルを作ってから削除するまでの間に、このスクリプト自体が様々な原因で中断する場合もありうる。

1. 端末のハングアップ (端末エミュレータの窓を [×] ボタンで消したとか、ネットワークなどで外から入ってきてこのスクリプトを実行している最中に、回線が切れてしまったとか) による中断
2. キーボードからの割り込み (典型的には Ctrl-C) による中断
3. キーボードからの中止命令 (典型的には Ctrl-\) による中断

4. 終了シグナルによる中断 (kill コマンドなどによる)

このとき、一時ファイルは残ったままになってしまう。

例えば、gvfilt スクリプト (3.2.2 節で作ったものでも、3.2.4 節で書き直したものでも) を実行中に、Ctrl-C で強制終了してみよう。一時ファイルは残ったままになる。また、ある端末エミュレータの窓にて同スクリプトを実行中に、その端末エミュレータの窓を [×] ボタンで消してから、gv のウィンドウを消してみよう。これでも同じ問題が起こる。

一時ファイルは後片付け (つまり消去) してから終了するのが望ましいので、上記のような場合にも一時ファイルが消せるように直そう。それにはシェルスクリプト中で、一時ファイルの**生成直後**に、シェルの組み込みコマンド trap を以下のように使う。

```
trap 'コマンド列' HUP INT QUIT TERM
```

ここで、HUP, INT, QUIT, TERM とはそれぞれ上記の 4 つの中断要因を表す (一部だけ書くことも可能)。UNIX 系 OS では、上記の 4 つの中断要因は「シグナル」という形でプロセスに伝えられ、デフォルトではこれらのシグナルを受け取ったときのプロセスの振る舞いは「終了する」である。trap コマンドは、これらのシグナルを受け取ったときの振る舞いを変える[46]。「コマンド列」のところには、これらのシグナルを受け取ったときにして欲しいことを書く。例えば、これらのシグナルを受け取ったら一時ファイルを消して終了して欲しいのであれば、

```
trap 'rm -f $TMPFILE; exit 1' HUP INT QUIT TERM
```

のように[47]書く (rm に -f オプションが付いているのは、何らかの原因で一時ファイルが既に消されていた場合に、エラーメッセージを出さないためにである)。もちろん、**正常終了**した場合に一時ファイルを消す処理も、これとは別に依然必要である。

一時ファイルが生成されてから、trap コマンドが実行されるまでの一瞬の間にシグナルで中断されたら、これでもまだ一時ファイルが残ってしまうが、今回それには目をつぶるとしよう。

3.2.6 似た例

文書清書システム TEX が作成する DVI ファイルを引数に取り、これを dvips で PS ファイルに変換したものを一時ファイルに入れて gv で見るシェルスクリプト dvigv を、ここまでと同様の方法で作ってみよう。xdvi と似た使い勝手になるだろうか。(xdvi より遅いはずだが、その代わり、xdvi では「虫めがね」機能による拡大が EPS ファイルの取り込み部分には効かないのに対し、こちらはできるという長所も生まれる。)

3.3 プログラムの実行時間計測

引数を 2 つ受け取って、多少時間のかかる何らかの計算をし、結果を標準出力に 1 つ出力する、dummy というプログラムがあるとする (とりあえず、~nide/jugyo/jjikken1-14/sample/dummy にあるものを使おう[48])。例えば「dummy␣0.8␣0.5↵」とすると、何か計算して、結果を「0.40000」のように出力する。

このプログラムに特定の引数を与えたときの、実行に要する**時間の計測**を行うシェルスクリプトを作ってみよう。dummy に与える引数は、シェルスクリプトの引数として指定するものとする。また、dummy を 1 回だけ実行するのでなく、**何回か実行**してその実行時間の平均を出すこととしよう。

実行時間の計測は、time コマンドで行える。例えば

```
$ time -p dummy 0.8 0.5↵
```

[46] UNIX 系 OS には、これら 4 つ以外にもシグナルがあり、その中には、受け取ったときの振る舞いを trap コマンドで変更することができないシグナルもある。例えば KILL (強制終了) シグナルがそうであり、このシグナルを受け取ったプロセスは必ず終了させられる。他のシグナルを無視していて、それらでは終了させられないプロセスを、強制終了させる最後の手段である。

[47] 古い UNIX では、シグナルの種類を HUP, INT などの名前ではなく、1 や 2 などの番号で指定しなければならないことがある。

[48] 自分のディレクトリにコピーするか、またはフルパスで ~nide/jugyo/jjikken1-14/sample/dummy と指定して実行しよう。

とすると、「dummy 0.8 0.5」を実行した上で、その**実行時間**を**標準エラー出力に**出す。実行時間の出力は、例えば以下のようになる (この他に、**dummy コマンドそのものの出力**が通常通り出てくる)。

```
real 0.96       (← 実経過時間 (入力待ち時間などがあればそれも含む)。単位は秒)
user 0.57       (← ユーザモードでの CPU 消費時間。単位は秒)
sys 0.38        (← カーネルモードでの CPU 消費時間。単位は秒)
```

ここで、time に -p オプションを付けている理由は、-p なしだと出力の形式がシステムによって異なるため、time コマンドの出力をプログラムで自動処理しにくいからである[49]。

実行を 3 回繰り返して計測するなら、シェルスクリプト内で

```
for i in a a a; do      # 「in」の後ろの文字列は3つあれば何でもいい
        time -p dummy $1 $2
done
```

とすることになる (シェルスクリプトの引数を dummy コマンドの引数に与えていることに注意)。しかし、これだけでは 3 回計測はできるが、その平均が出せない。そこで、平均を出すために、時間の計測結果をパイプで他のプログラムに渡したい。

だが、time は時間の計測結果を**標準エラー出力**に出すので、そのままではパイプで他のコマンドに渡せない。そこで、「2>&1」(0.2.3.1 節) によって time の標準エラー出力を**標準出力**にリダイレクトしよう。その際、本来の標準出力は**捨てる**必要がある (なぜなら、dummy コマンドの標準出力がそのまま time コマンドの標準出力に出てくるからである。これが一緒にパイプに渡されると、パイプの後ろのコマンドは、どれが時間の計測結果か分別できなくて困る)。そこで、標準出力を /dev/null (0.2.3.1 節) にリダイレクトし

```
for i in a a a; do
        time -p dummy $1 $2
done 2>&1 >/dev/null
```

とする[50]。これで、標準エラー出力を元々の標準出力にリダイレクトしてから標準出力だけをリダイレクトして捨てることになる。逆の「>/dev/null 2>&1」にすると、標準出力を /dev/null にリダイレクトしてから標準エラー出力をそれと同じところにリダイレクトするため、標準エラー出力も捨ててしまうので注意。

これで時間の計測結果が標準出力に出ているので、これをパイプで他のプログラムに渡す。今回は、計測結果のうち**「real」の欄だけ** (先の出力例でいうと「0.96」のところ) の平均を出すことにし、平均を出すプログラムは AWK で作ろう。第 1 フィールドが「real」である行の第 2 フィールドの平均を出せばよいので

```
#!/bin/sh
for i in a a a; do
        time -p dummy $1 $2
done 2>&1 >/dev/null | awk '
        $1 == "real"{total += $2; count++}      # 「real」の欄の数を加算
        END{print total/count}                  # 平均を出力
'
```

とする。これで、「$ スクリプト名 0.8 0.5↵」とすると、「dummy 0.8 0.5」を 3 回実行してその実行時間の平均を出すスクリプトになる。

3.3.1　バリエーションその 1

次に、dummy に与える引数を、シェルスクリプトの引数から与えるのではなく、例えば、第 1 引数に 0.2,

[49] -p なしだと出力が秒単位でなく「分:秒」のような形式になるため、平均するなどの処理が少しだけやりにくくなるからでもある。

[50] 細かい話だが、time は正しくはコマンドではなく、シェルが特別扱いする予約語であり (bash の場合)、コマンドでないため本来はリダイレクトが効かない。しかし time を for ループや「{ }」などで囲むと、1 つの大きなコマンドと見なされる (1.4.1 · 1.4.2 節) ためリダイレクトが効く。ここではたまたま for ループで囲んでいるため、time の出力をリダイレクトできるのである。

0.4, 0.6, 0.8, 1.0、第 2 引数に 0.1, 0.3, 0.5, 0.7, 0.9 のいずれかを与えるとして、計 25 通りの引数の組み合わせのそれぞれについて時間を計測してみよう (シェルスクリプトは**引数をとらなくなる**)。

まず、とりあえず第 2 引数は 0.1 に固定し、第 1 引数を 5 通りに変えてみよう。それには、先程のスクリプトの内容全体を

```
#!/bin/sh
for x in 0.2 0.4 0.6 0.8 1.0; do
        …ここに先程のスクリプトの内容を丸ごと入れ込む…
done
```

という **for ループの中**に入れ込み、さらに **dummy コマンドの引数**を `$x` と 0.1 に変えれば OK。ここで、ループの中でも for 文を使うので、中と外の for 文の変数が別でなければならないことに注意。

しかし、それだけだと、時間の計測結果の平均が 5 個出てくるだけなので、そのうちどれが、dummy コマンドにどの引数を渡したときのものかがわからない。そこで、計測結果の平均の後ろに、dummy コマンドに渡した引数も出力されるようにしよう。

```
#!/bin/sh
for x in 0.2 0.4 0.6 0.8 1.0; do
        for i in a a a; do
                time -p dummy $x 0.1
        done 2>&1 >/dev/null | awk '
                $1 == "real"{total += $2; count++}
                END{printf "real:%.2f ", total/count}   # 平均の出力後改行しない
        '
        echo 第1引数:$x 第2引数:0.1     # dummyコマンドへの引数を後ろに出力する
done
```

これで出力は、「`real:×.××` 第1引数:×.× 第2引数:×.×」の形で、計測結果の平均の後ろに、dummy コマンドに渡した第 1・2 引数が表示される。書式に `%.2f` を使ったのは、time コマンドが時間の計測結果を小数第 2 位までしか表示しないため、それ以上の精度が得られないからである。

第 2 引数も変えるなら、さらにもう 1 重の for 文を使えばよい。

3.3.2 バリエーションその 2

これまでは、dummy コマンドの 1 通りの引数について、3 回繰り返しを行って平均を出していた。今度は繰り返しを 3 回ではなく、5 回や 10 回に変えてみよう。

これは、上記のスクリプトの「`for i in`」の後ろの「`a`」の個数を変えればできる。しかしそれでは、例えば 50 回繰り返そうとすると、「`a`」を 50 回書かねばならない。回数を「`50`」のような数字で与え、しかもそれを 1 箇所だけ書き換えれば回数を変更可能なようにできるだろうか (1.9 節を参考にしよう)[51]。

本節に述べたような方法を採ると、time コマンドの出力をいちいち画面から LibreOffice Calc あるいは Excel のような表計算ソフトに転記して平均を出すやり方に比べ、処理を**全自動化**できる点、データの量が増えても人間の手間は一定で済む点、また転記ミスや誤操作など人間のミスによる誤りもなくせる点が有利なのである。本資料先頭で触れた、GUI と比べての CUI の利点の 1 例である。

3.3.3 終了を通知

話が少し変わるが、本節の例は dummy の (何回もの) 実行に長時間かかることをストーリーとして想定している。そこで、全ての作業が終わったら自分に**電子メール**で通知することを考えてみよう。それができ

[51] うまくいかない場合は、一旦パイプ以後の awk の部分を取り除き、その前の部分が正しく動いているか調べてみるのがコツ。

れば、作業開始後自分は端末を離れていても、自分へのメールがスマートフォンなどに転送されていれば、作業が終わり次第、終わったことを知ることができる。

これはそんなに難しくない。コマンドラインでメールを送る、その名も mail[52] というコマンドがあるので[53]、それをスクリプトの末尾で使えばよい。mail コマンドの使い方は

```
$ mail␣-s␣'メールの題目'␣宛て先メールアドレス···↵
```

である (宛て先アドレスは複数指定可)。メールの本文は標準入力から読まれるので、実際には

(方法 1) `$ echo␣'メッセージ'␣|␣mail␣-s␣'メールの題目'␣宛て先メールアドレス···↵`

(方法 2) `$ mail␣-s␣'メールの題目'␣宛て先メールアドレス··· < ファイル名↵`

などとすることになる。ただし、以下の点に注意。

- mail コマンドだと、メールの題目や本文に**日本語**を使うには特別な工夫がいる[54]。その工夫については、本資料ではとりあえず考えないことにしよう (つまり、題目や本文に日本語が事実上使えない)。作業の終了を自分に短いメールで知らせる程度なら、これでも別に困らないであろう。
- どんなシステムでも mail コマンドでメールが送れるわけではなく、システムによっては管理者がこの機能を提供していない (mail コマンドがそもそもない、あるいはあっても外部にメールが送れない) 場合もある[55]。ただし、G 棟システムのマシンではちゃんと mail コマンドで外部にもメールが送れる。

そんなわけで、最後の工夫として、作業が終わったら自分に「operation finished at 月/日 時:分」という内容のメールを送るように、本節のスクリプトを改造してみよう (「月/日 時:分」の部分を生成するには、date コマンドを 1.2 節の引数で使う。バッククォート (1.9 節) の利用も考えよう)。

3.4　2 次関数のグラフを gnuplot で描画

gnuplot は X ウィンドウシステム上でのグラフ描画ソフト。例えば、

```
$ gnuplot↵
```

で起動して

```
gnuplot> plot sin(x)↵
```

とすれば、ウィンドウが現れて $y = \sin(x)$ のグラフを描画する。端末に入力終わりのサインを送れば (つまり Ctrl-D を入力すれば) そのウィンドウが消えて gnuplot は終了する。また、data.txt というファイルに

```
1.3     7.9
2.7     4.2
3.8     2.6
  ⋮
```

のように、データが 1 行あたり 1 ペアの形で延々と書いてあるとき、

```
$ gnuplot↵
gnuplot> plot "data.txt" with lines↵
```

のようにすれば、折れ線グラフを描画する。この場合も、端末に Ctrl-D を入力すればウィンドウが消えて gnuplot は終了する。

gnuplot は対話的に使うことが多いが、工夫すればシェルスクリプトの中からも使える。やってみよう。

---

[52] システムによっては、mailx というコマンド名になっている場合がある (その場合、mail という名前の別な機能のコマンドもあるので、要注意)。

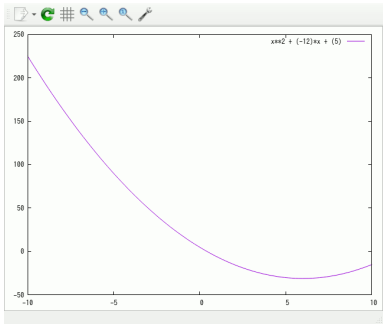
[53] そもそも、mail (ないしは mailx) コマンドこそがかつてはメールを送受信する基本手段であった。Mew や Sylpheed など、マウスやメニューでの操作でメールを読み書きする、いわゆるメールソフトが登場するのは、後になってからである。

[54] いわゆるメールソフトは、メールを発送するにあたって、その “工夫” を内部で自動的に行ってくれているのである。

[55] よくあるのが「ユーザが使えるメールソフトは (例えば Sylpheed のような) GUI のものだけ」、あるいは「ユーザが使えるメール機能は Web メールだけ」などのような形態である。これでは、mail コマンドのようにユーザがメールを自動で発送することは難しい。「作業の自動化」というコンピュータのメリットの 1 つをわざわざ削いでいるわけである。

3.4.1 シェルスクリプトからの gnuplot の利用

そのような例として、「引数を 2 つ受け取り、それを a と b とすると、$y = x^2 + ax + b$ のグラフを描く」ようなシェルスクリプトを作ってみよう。これは、



```
#!/bin/sh
{
        echo "plot x**2 + ($1)*x + ($2)"
        read dummy
} | gnuplot
```

でできる。このシェルスクリプトの名前が niji だとし、試しに

```
$ niji -12 5↵
```

としてみよう。ウィンドウが現れて $y = x^2 - 12x + 5$ のグラフが描かれ (上図)、端末に**リターンキー**を入力するとそのウィンドウが消えて終了する。(引数に負数を与えても、式の解釈が混乱を起こさないように、念のため「$1」や「$2」を「( )」で囲んである。もっとも実は括弧がなくても大丈夫なのだが…)

ここで、「read dummy」は何の役割を果たしているのだろうか。それについては 3.4.2 節で考えよう。

> 1.7.1.2 節で「変数の利用時には『" "』で囲む方がよい」と述べたが、ここでは $1 や $2 を個別に「" "」で囲まずに、echo の引数全体を「" "」で囲んでいることに注意。なおこの場合、 $1 や $2 を改めて「" "」で囲んで「"plot x**2 + ("$1")*x + ("$2")"」のようにする必要はない。というより、そのようにすると、「"plot x**2 + ("」と「$1」と「")*x + ("」と「$2」と「")"」をくっつけたことになり、肝心の $1 や $2 が「" "」で囲まれていないことになって、かえって悪い。

3.4.2 「read dummy」の役割?

「read dummy」の役割について考えるため、試しにこれを省いた

```
#!/bin/sh
{
        echo "plot x**2 + ($1)*x + ($2)"
} | gnuplot
```

あるいは

```
#!/bin/sh
echo "plot x**2 + ($1)*x + ($2)" | gnuplot
```

というシェルスクリプトで同じことをしてみよう。ウィンドウは、一瞬だけ現れてすぐに消えてしまう。

これは、gnuplot は標準入力から描画命令を受け取ってグラフ描画を行い、**標準入力が尽きたら終了**するようになっているためである。上記のスクリプトだと、(例えば第 1 引数が -12、第 2 引数が 5 だとして) 標準入力から「plot x**2 + (-12)*x + (5)」(と改行) が来たところで標準入力が尽きる。そこで、gnuplot はその時点で終了し、グラフ描画ウィンドウは消えてしまう。

これに対し、3.4.1 節のスクリプトでは、「echo "plot x**2 + (-12)*x + (5)"」と「read dummy」の標準出力をまとめて、パイプで gnuplot に渡している (1.4.2 節に出てきた、コマンドをグループ化してまとめてパイプに渡す手法が使われていることに注意)。

「read dummy」は自分の標準入力 (つまりキーボード) から 1 行読んで、それを dummy というシェル変数に入れて終了 (→1.8) する (ちなみにこの変数の値は使われていない)。よって、キーボードからリターンキーが押されると「read dummy」は (特に何も出力せずに) 終了する。

これで「{」「}」内のコマンドが全て終了し、従ってパイプからの gnuplot への標準入力が尽きるので、ここで初めて gnuplot が終了する。というわけで、gnuplot はキーボードからリターンキーが押されるまで**待ってから**終了してくれるので、リターンキーが押されるまではウィンドウが消えずに済む[56]。

[56] gnuplot の `-persist` **オプション**を使ってもウィンドウの消失を防げるが、それだとプログラム終了後もウィンドウが残ってしまうので、プログラムの終了時に自動的にウィンドウを消したい応用を考慮して、ここではこのオプションの使用は避けた。

3.4.3　改良第 1 弾: 範囲指定オプションの追加

さて、これでは x の範囲が -10〜10 に固定されてしまう。x の範囲を与えることができるようにするにはどうすればいいだろう? つまり、(シェルスクリプトの名前は引き続き niji として) 例えば

```
$ niji -12 5↵
```

で今までと同じ動作、

```
$ niji -r 0:15 -12 5↵
```

のように -r オプションを指定すれば x を 0〜15 の範囲に指定して描画するようにしたい (ちなみになぜ -r なのかというと、range の略で r なのである)。

gnuplot を対話的に使っているときは、plot に対して

```
gnuplot> plot [0:15] sin(x)↵
```

のように x の範囲を与えることができるので、シェルスクリプトからもそれを行うようにしよう。

この場合、最初は以下のような書き方を試みる人が例年多い。でも、これは**良くない**。

```
#!/bin/sh
if [ "$1" = -r ]; then  # 「niji -r 0:15 -12 5」のような場合(-rオプションあり)
        {
                echo "plot [$2] x**2 + ($3)*x + ($4)"
                read dummy
        } | gnuplot
else                    # 「niji -12 5」のような場合(-rオプションなし)
        {
                echo "plot x**2 + ($1)*x + ($2)"
                read dummy
        } | gnuplot
fi
```

これだと、グラフを描く処理を**2ヶ所**で書かねばならず、しかもその 2 つが微妙に異なる (一方で $3, $4 となっている箇所がもう一方では $1, $2 であるなど)。グラフを描く部分の処理を変えたい場合は、微妙に違う 2ヶ所を、整合性を崩さないように書き換えねばならず、書き誤りが起きやすくなる。

さらに (こちらの方がもっと重要なのだが)、3.4.4 節で出てくるように、例えばオプション引数が -r と -p の 2 種類に増えると、このやり方では「-r オプションが指定されていて -p オプションが指定されていない場合 (あるいはその逆)」「両方とも指定されているが -r オプションの方が先に指定されている場合 (あるいはその逆)」などのように、場合の数が**急激に増える**。オプションが 3 種類くらいになるともう破綻する。

よいやり方は、オプション引数の解析を行う部分とグラフ描画の部分を**分離**して書くことである。それが以下のスクリプトで、~nide/jugyo/jjikken1-14/sample/niji に置かれている。

```
#!/bin/sh
# 先にオプション引数の解析を済ませてオプション引数を取り除く。
# この部分にはオプション引数の解析だけ書き、描画命令の出力は書かない
unset RANGE
if [ "$1" = -r ]; then
        RANGE="[$2]"
        shift 2
fi
# すると、もと-rオプションがあったかなかったかに関わらず、この時点では
# -rオプションは取り除かれていて、$1は1次の係数、$2は定数項になっている。
# オプション引数の解析はここまでで完了し、ここ以降ではやらない
{ # これ以降ではグラフ描画処理だけを行う
```

```
        echo "plot $RANGE x**2 + ($1)*x + ($2)"
        #  「echo "plot …"」はこの1回しか現れない
        read dummy
} | gnuplot
```

まず、unset コマンドで変数 RANGE を**クリアしておく**。続いて、オプション引数のチェックを行う。すなわち、第 1 引数が「-r」だったら、第 2 引数を「[ ]」で囲んだものを変数 RANGE に入れて shift 2 を行う (shift コマンドについては 3.1.5 節)。

こうすると、オプション -r が**指定されていてもいなくても**、オプション引数のチェックが済んだ時点 (if 文の終了時点) では、$1 は 1 次の係数、$2 は定数項になっている。また、変数 RANGE はオプション -r が指定されていればその値を「[ ]」で囲んだもの、そうでなければ (クリアされているため) 空文字列になっている。そこで、このスクリプトを例えば

```
$ niji -r 0:15 -12 5↵
```

のように実行した場合は gnuplot に「plot [0:15] x*x + (-12)*x + (5)」が渡されて $0 \leq x \leq 15$ の範囲でグラフを描画し、

```
$ niji -12 5↵
```

の場合は gnuplot に「plot x*x + (-12)*x + (5)」が渡されて今までと同じ動作になる。

> **未代入のシェル変数**を参照した場合、その値は**空文字列**ということになっている。**unset コマンド**はシェル変数を未代入の状態に戻す働きがあるので、変数 RANGE に改めて代入しない限り、その値は空文字列になるのである。
>
> ちなみに、スクリプトの先頭で変数 RANGE を**明示的にクリア**しておかないと、-r オプションを指定しなかった場合の変数 RANGE の値が空とは限らなくなってしまう。なぜなら、**環境変数** (4.6 節) でもあるシェル変数の場合、スクリプト中では未代入に見えても、スクリプトの**実行開始時**に既に何らかの値が代入されていることがありうるからである。どの名前の変数が環境変数であるかということはあらかじめ決まってはいないので、「スクリプト中でまだ値を代入されていないシェル変数の値は空文字列だ」と勝手に前提してスクリプトを書くと、後で困る可能性がある。シェル変数の値が空文字列であることを**保証したい**場合、明示的にクリアする (か、空文字列を代入する) べきである。

これにより、オプションの有無に関わらず、gnuplot にグラフを描かせる命令 (plot 命令) を echo で出力する過程はまとめて **1 箇所だけに**書けばよい (上の例での「echo "plot …"」のところがそれ)。まとめて書けるので見通しがよく、記述や修正がしやすくなる。また、オプションが 2 種類、さらには 3 種類以上に増えても、同様のやり方をとれば、やはりこの過程は 1 箇所だけに書けば済む。

3.4.4 改良第 2 弾: PostScript 出力オプションの追加

さて、gnuplot にはグラフを X 画面に出すだけでなく、PostScript, EPS, Tgif, PNG, アスキーアートなどさまざまな形式で出力する機能がある。

そこで、今度は「-p」オプションを追加し、これが指定されたら、(グラフを画面に出すのでなく) PostScript によるグラフ描画結果 (PostScript については 3.2.1 節冒頭参照) を標準出力に出すようにしてみよう。例えば

```
$ niji -p -12 5↵
```

とした場合、$y = x^2 - 12x + 5$ のグラフを PostScript で描画したものが標準出力に出るようにしたい。なおこの場合、今までと違って、グラフ描画ウィンドウが出ないので、端末からのリターンキー入力を待つ必要はない。従って、「-p」オプションで PostScript 出力を行う場合にはリターンキーを**待たない**ようにスクリプトを作ろう。

これができれば、lpr コマンドと併用して

```
$ niji -p -12 5 | lpr↵
```

のようにすると描画結果の印刷ができ[57]、また 3.2.2 節で実現した gvfilt を併用すると

```
$ niji -p -12 5 | gvfilt↵
```

で描画結果のプレビューもできるはずである。さらに、例えば次のような使い方も可能になるだろう。

```
for i in 1 2 3; do
        niji -p $i 0 | gvfilt
done
```

もちろん、-r と -p の 2 種類のオプションを任意に組み合わせた指定もできるようにしよう。例えば

```
$ niji -12 5↵                    (両方とも指定せず。通常の描画)
$ niji -r 0:15 -12 5↵            (-r だけ指定。0 ≤ x ≤ 15 の範囲で描画)
$ niji -p -12 5↵                 (-p だけ指定。PostScript で出力し、リターンキーは待たない)
$ niji -r 0:15 -p -12 5↵         (両方指定。0 ≤ x ≤ 15 の範囲で描画、PostScript で出力し、
                                  リターンキーは待たない)
$ niji -p -r 0:15 -12 5↵         (両方指定。同上)
```

のいずれもが正しく動くように。(ただし、オプションは常に引数の先頭部に指定するものとしよう。例えば「`niji -12 5 -p`↵」のように、本来の引数より後ろにオプションが来るような使い方は考えない。)

これをどのように実現すればいいだろうか。gnuplot の機能としては

```
gnuplot> set term post
```

で PostScript を出力するようになるので、これを使おう。例えば、-p オプションを指定して「`niji␣-p␣-12␣5`↵」とされた場合は gnuplot の標準入力に

```
set term post
plot  x*x + (-12)*x + (5)
```

が渡るようにすればよい。

また、PostScript 出力のときは、リターンキーを待つ必要がないので、-p オプションが指定された場合は「`read dummy`」は不要ということになる (しかし、-p オプションを指定しない場合は依然として「`read dummy`」が必要なことにも注意しよう)。

3.4.3 節と同様、オプション引数の解析部と実際にグラフを描く部分は**分離**して記述しよう。まずオプション引数の解析は、「シェルスクリプトへの引数を先頭から順に調べ、それがオプションだったら対応する処理を行うとともにその引数を取り除き、オプションでない引数が見つかればその処理をやめる」というように作ればよい。従って、シェルスクリプトの先頭で while 文を使って

```
#!/bin/sh
…まず必要な変数の値をunsetで初期化しておく…

# オプション引数解析部。オプション引数を先頭から順に調べて取り除いていく
while true; do
        if [ "$1" = -r ]; then
                …オプションで指定されたxの範囲を変数に記録しておく…
                shift 2
        elif [ "$1" = -p ]; then
                …オプション-pが指定されたということを記録しておく…
                shift
        else # それ以外の引数が見つかったのでオプション解析を終了
                break
        fi
done
# すると、もともとのオプション引数の有無や順番にかかわらず、この時点では
# オプション引数は全て取り除かれていて、$1は1次の係数、$2は定数項になっている
```

[57] 紙がもったいないので、lpr を使ったテストはみだりにやらないようにしよう。

のように行うことになる。そして、オプション引数解析部 (上記の while 文の部分) ではオプション引数の解析だけに専念し、それ以後はグラフの描画だけに専念する。特に、plot 命令の出力は

```
{ # この部分ではグラフ描画処理だけを行う
        …-pが指定されていれば「set term post」を出力…
        echo "plot …"       # plot命令をechoで出力するのはここ1箇所だけ
        …-pが指定されていなければread dummyを実行…
} | gnuplot
```

のように**まとめて 1 箇所でだけ**行う。この考え方で、スクリプトを完成させてみよう。

ちなみに、オプション引数の解析にもっと適した手段として、getopts コマンドというものがある。ここでは触れないが、後に 4.1 節で登場する。

3.4.5 改良第 3 弾: 高次式のグラフ描き

3.4.4 節で完成させたスクリプトをさらに、「引数を $a_1, \cdots, a_{n-1}, a_n$ の n 個与えれば、n 次関数 $y = x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n$ のグラフを描く」ように改良できるだろうか? 例えば引数を 3 つ与えて

```
$ niji -12 5 7↵
```

とすれば、3 次関数 $y = x^3 - 12x^2 + 5x + 7$ のグラフを描くようにしたい。むろん、オプション引数もそのまま使えるようにするのである。

この場合、gnuplot にプロットさせる式を、$(((1)x - 12)x + 5)x + 7$ のような「x を掛けては定数を足す」を繰り返す形に変形すると考えやすい。オプション引数を取り除き終わった後で

```
FORMULA=1
for i in "$@"; do
        FORMULA="($FORMULA)*x+($i)"
done
```

とすると、例えばオプション引数以外の引数が「-12」「5」「7」の 3 つの場合、for ループを抜けた時点でのシェル変数 FORMULA の値は「(((1)*x+(-12))*x+(5))*x+(7)」になる。そこで、それを gnuplot の plot 命令に与えればよい。また、オプション解析部には変更の必要はない。やってみよう。

3.4.4 節までで、plot 命令を echo する処理を 1 箇所だけで行うように書いたことが、ここでも利点として生きる。plot 命令の書き換えは、その 1 箇所だけに対して行えばよいのである。

3.4.6 データファイルの自動プロット

話は変わって、今度は数式ではなくデータファイルの自動プロットを行ってみよう。いま、

```
1.3     7.9     -2.3
2.7     4.2     -0.4
3.8     2.6     0.8
4.4     3.3     1.2
6.1     1.7     4.7
```

のように各行あたり 3 つのデータが与えられているファイルがあり、「第 1 列と第 2 列のペア」「第 1 列と (第 2 列 + 第 3 列) のペア」でプロットしたいものとする。(本節の例を試したい場合、データファイルは自分で用意すること。ただしその場合、ファイル末尾に**余分な空行**が付かないように注意。でないと、予想外のエラーに悩む可能性がある。)

方法はいくつかあるが、まず考えられるのは gnuplot で

```
$ gnuplot↵
gnuplot> plot "データファイル名" using ($1):($2) with lines, \↵
>             "データファイル名" using ($1):($2+$3) with lines↵
```

とする方法である (入力行が長いので、「\」によって継続行とした)。この操作をシェルスクリプト化し、さらにデータファイル名を**スクリプトの引数**として与えられるようにしてみよう。

```
#!/bin/sh
DATAFILE="$1"   # データファイル名を第1引数として与える
{
        # プロット命令の出力
        echo 'plot "'"$DATAFILE"'" using ($1):($2) with lines, \'
        echo '     "'"$DATAFILE"'" using ($1):($2+$3) with lines'
        # リターンキーが押されるまで待たせる
        read dummy
} | gnuplot
```

最初の echo の引数は「'plot "'」と「"$DATAFILE"」と「'" using ($1):($2) with lines, \'」を、空白を置かずにつなげていることに注意 (1 番目と 3 番目は「' '」で、2 番目は「" "」でクォートされている)。第 2 の echo の引数も同様。ここでの $1 や $2 などは gnuplot の記法であって、シェルスクリプトの引数を表す $1 や $2 とは**関係ない**ことにも注意しよう (**「' '」で**クォートしているのはそのため)。

以上によって、例えばデータファイル名が data.txt である場合、2 つの echo が

```
plot "data.txt" using ($1):($2) with lines, \
     "data.txt" using ($1):($2+$3) with lines
```

を出力する。これがパイプで gnuplot に渡されてプロットされる。なお、「read dummy」がある理由は 3.4.2 節と同じである。

ちなみに、このスクリプトはデータファイル名に「"」が混じっていると正しく動かない (理由を考えよう)。

計算式が複雑である場合などは、gnuplot に計算もプロットもさせるより、何らかのプログラミング言語に計算をやらせ、結果のデータを gnuplot に渡して、gnuplot にはプロットだけやらせる方が簡明に書ける場合もある。そこで、そのようなやり方を考えよう。まず、準備として以下のスクリプト

```
#!/bin/sh
DATAFILE="$1"   # データファイル名を第1引数として与える
# プロット命令の出力
echo 'plot "-" with lines, "-" with lines'
# データ部その1の出力
awk '{print $1, $2}' "$DATAFILE"
echo e
# データ部その2の出力
awk '{print $1, $2 + $3}' "$DATAFILE"
echo e
```

に、引数として上記 (3.4.6 節冒頭) のデータファイルのファイル名を渡して実行してみよう。出力として

```
plot "-" with lines, "-" with lines
1.3 7.9
2.7 4.2
3.8 2.6
4.4 3.3
6.1 1.7
e
1.3 5.6
```

```
2.7 3.8
3.8 3.4
4.4 4.5
6.1 6.4
e
```

が得られる。これをパイプで gnuplot に渡せばよい。gnuplot の plot 命令には、プロットすべきファイル名として「-」を渡すと標準入力からデータを読む (その際、データの終わりは「e」で表される) という特例があるので、これでうまくいく。

完成版のスクリプトは以下のようになる (~nide/jugyo/jjikken1-14/sample/plottwo に置かれている)。「title "plot No.～"」は、プロット画面での折れ線のタイトル表示の指定で、これらがないと 2 本の折れ線のどちらもタイトルが「"-"」になってしまう。

```
#!/bin/sh
DATAFILE="$1"   # データファイル名を第1引数として与える
{
        # プロット命令の出力
        echo 'plot "-" with lines title "plot No.1", \'
        echo '     "-" with lines title "plot No.2"'
        # データ部その1の出力
        awk '{print $1, $2}' "$DATAFILE"
        echo e
        # データ部その2の出力
        awk '{print $1, $2 + $3}' "$DATAFILE"
        echo e
        # リターンキーが押されるまで待たせる
        read dummy
} | gnuplot
```

なお、このスクリプトは先のものと異なり、データファイル名に「"」が混じっていても (あるいは、他のどのような特殊文字が混じっていても) うまく動く。

もう 1 つのやり方としては、一時ファイルを使う手もある。こちらの方が考えやすいかも知れない。なお、下記のスクリプトにはもう 1 つ変更を施した。それは、『スクリプトに**第 2 引数**が指定されていれば、第 1 のプロットは「第 1 列と第 2 列のペア」ではなく、「第 1 列と (第 2 列 × スクリプトの第 2 引数に指定した数) のペア」とする』というものである。一時ファイルの作成には mktemp (3.2.4 節) を使った。

```
#!/bin/sh
DATAFILE="$1"   # データファイル名を第1引数として与える
if [ $# -gt 1 ]; then
        SCALE="$2"      # 第1プロットに使う倍率を第2引数に与える
else
        SCALE=1         # 与えられなければ1としておく
fi

# 一時ファイルを2つ作成
TMPDATA1=`mktemp /tmp/dataXXXXXX` || exit 1
TMPDATA2=`mktemp /tmp/dataXXXXXX` || exit 1
# データ部その1を第1の一時ファイルに出力
awk '{print $1, $2 * '"$SCALE"'}' "$DATAFILE" > $TMPDATA1
# データ部その2を第2の一時ファイルに出力
awk '{print $1, $2 + $3}' "$DATAFILE" > $TMPDATA2

{
        # 2つの一時ファイルの内容をプロットさせる
```

```
        echo 'plot "'$TMPDATA1'" with lines title "plot No.1", \'
        echo '     "'$TMPDATA2'" with lines title "plot No.2"'
        # リターンキーが押されるまで待たせる
        read dummy
} | gnuplot

# 一時ファイルの後始末
rm -f $TMPDATA1 $TMPDATA2
```

mktemp が生成する一時ファイル名は、(引数として与えるファイル名部分にもともと特殊文字が混じっていない限り) 特殊文字を含まない。そのため、上の例では $TMPDATA1 や $TMPDATA2 を「" "」でクォートする必要はないし、また、データファイル名に「"」が混じっていることに起因する混乱を gnuplot が起こすこともなくて済む。

3.5　複数の画像ファイルの自動コンバートおよび連番化

~nide/jugyo/jjikken1-14/img ディレクトリの下に、JPEG 形式の画像ファイル (ファイル名は「*.jpg」の形) がいくつか用意してある。まずこれを自分のディレクトリに ln コマンドでシンボリックリンクしよう (シンボリックリンクについては 2.1.1 節参照)。

これらの JPEG ファイルを全て、BMP 形式に変換するにはどうすればいいだろうか。ファイルが 1 つだけなら、画像ファイルの形式変換用の convert というコマンドを使って

```
$ convert mantova.jpg mantova.bmp↵
```

で足りるが、多数あると面倒。そこで、引数に与えた画像ファイルの形式をまとめて BMP に変換する、convbmp というコマンドをシェルスクリプトとして作ろう。つまり

```
$ convbmp *.jpg↵
```

で、「*.jpg」にマッチする名の全ての画像ファイルを BMP 形式に変換する。生成するファイル名は、元のファイル名の拡張子 (最後の「.」以後) を「.bmp」に変えたものにしよう。つまり、もともと「aaa.jpg」「bbb.jpg」の 2 ファイルがあったとすると、「convbmp *.jpg↵」で「aaa.bmp」「bbb.bmp」の 2 ファイルが作られるようにする。

まず、与えた (変換前の) ファイル名から、変換後のファイル名を生成しなければならない。そのためには、与えたファイル名の拡張子を取り除くことができればよい (そうすれば、その後に「.bmp」を付けることによって、変換後のファイル名を生成できる)。これはちょっと難しいので結果だけ書くと、(シェル変数 i に変換前のファイル名が入っているとして)

```
echo "$i" | sed 's/\.[^./]*$//'
```

でできる[58](これは、$i の値の末尾の「.」+『「/」「.」以外の文字からなる文字列』という部分を取り除いて出力する、ということをやっている)。そこで、これをバッククォート「‘ ‘」(1.9 節) 内に入れて

```
convert "$i" "‘echo "$i" | sed 's/\.[^./]*$//'‘".bmp
```

で[59]、(例えばシェル変数 i に「aaa.jpg」が入っていたら)「convert aaa.jpg aaa.bmp」が実行される、つまり、aaa.jpg を BMP に変換した結果が aaa.bmp に入る。

そこで、後は for ループによって引数の 1 つ 1 つにこれを適用すればいい。

[58] この部分は、sed の代わりに expr を用いて「expr "$i" : '\(.*\)\.[^/]*$' '|' "$i"」でもできる。なお、シェルの種類によっては、sed や expr に頼らずに同等のことができる機能を持つものもある。しかしその機能は、全てのシェルに共通ではないので、その機能を使ったシェルスクリプトは、若干汎用性が落ちる。

[59] これは、1.9.1 節で述べた「ダブルクォート中でバッククォートを使う」典型例である。この例のバッククォートをダブルクォートで囲んでいないと、$i の値に空白が入っていた場合などに、バッククォート部を置き換える文字列も空白を含むので、それが複数の引数に分離されて困ってしまう。

ただし、一部のバージョンの Emacs のシェルスクリプトモードにはバグがあり、上記の「"‘echo "$i" | sed ’s/\.[^./]*$//’‘"」の部分をファイル中に書き込むと、構文解析が混乱して自動インデントが正しくできなくなってしまう。その場合は、シェルスクリプトの「‘ ‘」の代替記法として「$( )」と書ける (両者は同じ意味) ということを使って、この部分を

```
"$(echo "$i" | sed ’s/\.[^./]*$//’)"
```

と書こう。これで、シェルスクリプトモードのバグは回避できる。なお、現在の G 棟システムではこの回避法は必要ない。

3.5.1 上書きのチェック

さて、このままでは、「aaa.jpg」「bbb.jpg」「ccc.jpg」「bbb.bmp」の 4 ファイルがあったとき、「convbmp aaa.jpg bbb.jpg ccc.jpg↵」で元の bbb.bmp は上書きされてしまう。bbb.bmp がもし存在したら、bbb.jpg の変換は行わない (そしてその旨エラーメッセージを出す) ようにできるだろうか? (ただし、それで処理全体を中止するのではなく、後続の ccc.jpg の変換は行われてほしい。)

ここで、一般にエラーメッセージとは、端末でのユーザとのやりとりを意図したものであるので、1.2 節でも述べたように、**標準エラー出力**に出すべきであり、標準出力に出すべきではない。よって、エラーメッセージの出力には、標準エラー出力へのリダイレクト記法「>&2」(0.2.3.1 節) を使って

```
echo ’メッセージ’ >&2
```

としよう。

また、シェル変数を活用して、「echo "$i" | sed ’s/\.[^./]*$//’」を複数回行わないよう工夫しよう。これを複数回行うのは効率が悪い。

なお、変換先ファイルが既存の場合、上書きするかどうかをユーザに問い合わせるというバリエーションも考えられよう。

3.5.2 連番ファイル名の生成

上記ができたら今度は、「aaa.jpg」「bbb.jpg」··· を「aaa.bmp」「bbb.bmp」··· に変換するのではなく、「1.bmp」「2.bmp」··· のような連番のファイル名にする版も考えてみよう。(3.5 節のような、拡張子を取り除く難しい操作がいらなくなるので、こっちの方がかえって簡単かも。)

しかし、そのままだと、元のファイルが 10 個以上あった場合、できるファイルは「9.bmp」の次が「10.bmp」になって、ls したときに数の順番に並ばない。「00001.bmp」,「00002.bmp」, ···,「00009.bmp」,「00010.bmp」,「00011.bmp」, ··· のようになるようにする (5 桁位で十分だろう) ことはできるだろうか (ヒント: 表 0.3 の printf コマンドを使おう)。

3.6 リモート処理

3.6.1 ssh

ssh によって、ネットワークでつながっている他のマシンにリモートログインできる。それには

```
$ ssh remotehost↵
```

とする。*remotehost* には、リモートマシンのホスト名あるいは IP アドレスを指定する。例えば

```
$ ssh gpc1x61↵
```
あるいは
```
$ ssh 172.16.12.141↵
```

のようにする (どちらも G 棟システム G401 室の教員用卓のマシンの Linux に入るときである[60])。ここでパスワードを聞かれたり、(*remotehost* に入るのが初めてのときは) fingerprint の確認を求めてきたりする。詳しくは `http://www.ics.nara-wu.ac.jp/ics-only/pdf/ssh.pdf` 参照 (本学学内からのみ閲覧可)。

remotehost にログイン後、(通常は Ctrl-D あるいは exit コマンドにより) シェルを終了させると、手元のマシンのシェルに戻ってくる。下記は、gpc1x02 の端末を使っている人が、gpc1x01 に入ってみた例 (下線部がユーザの入力)。プロンプトの違いに注目。

```
gpc1x02[7]:~$ ssh gpc1x01
  ··· パスワードを要求 ··· (ここでパスワードを入れる)
gpc1x01[1]:~$ w      (← gpc1x01 で w コマンドを実行)
  ··· gpc1x01 に現在ログインしている人のリストが表示される[61]···
gpc1x01[2]:~$ (Ctrl-D)
gpc1x02[8]:~$       (gpc1x02 に戻った)
```

「`ssh` *remotehost command*」とすることによって、ssh で他のマシンのコマンドを直接動かすことも可能。

```
gpc1x02[8]:~$ ssh gpc1x01 w
  ··· パスワードを要求 ··· (ここでパスワードを入れる)
  ··· gpc1x01 に現在ログインしている人のリストが表示される ···
gpc1x02[9]:~$
```

ssh-agent を使うと、ssh コマンドのたびにパスワードを入力しなくてもすむようにできる (これについても詳しくは前掲資料参照)。そこで、実際に ssh-agent を使って、上記が**パスワードの入力なし**でできるようにしておこう。

3.6.2　シェルスクリプトによるリモート処理の自動化

ここで、例えば下記のようなシェルスクリプトを作ってみる。

```
#!/bin/sh
for i in "$@"; do
        echo --"$i"--
        ssh "$i" w
done
```

このシェルスクリプト loopw を「`loopw gpc1x01 gpc1x02`↵」のように動かすと、(gpc1x01 と gpc1x02 の Linux が共に起動していれば) 両マシンにログイン中のユーザが表示されるだろう。ただし、**ssh-agent** を使ってパスワードなしで ssh できるようにしてからでないと、指定したマシンの数だけ毎回パスワードを打ち込むはめになる。

さて、上のスクリプトでは、対象マシンのホスト名を引数で個別に指定せねばならない。これを、G401 のマシン (その時点で Linux が動いているもの) 全てを自動的に対象とするよう作り変えてみよう。もちろん、全マシンのホスト名をいちいち引数としてスクリプトに渡すことなしに、である。(注: ネットワークに**負荷をかける**ので、やみくもにやらないように。また、大勢でいっぺんにやるのは避けること。)

G401 の Linux マシンのホスト名は、gpc1x01 ～ gpc1x61 である[62](教卓のものを含む)。そこで、まず

```
N=1
while [ $N -le 61 ]; do
        ···
        N=‘expr $N + 1‘
done
```

[60] 2014 年現在。その後はシステム更新により、ホスト名も IP アドレスもこれと異なる場合がある。
[61] 正確には、ログインして X 画面を利用しているだけでは表示されず、ターミナル (端末エミュレータ) が起きていないと表示されない。
[62] これも注釈 60 同様、2014 年現在。以後は異なることがある。

というループを考えよう。

このループの中で、変数 N の値から「gpc1x○○」というホスト名を作って変数 RHOST に代入し (N が 1 桁の場合に、ホスト名は「gpc1x1」でなく「gpc1x01」のようにならなければいけないことに注意。そうすることは if 文を使ってもできるが、if 文を使わずに printf コマンド (表 0.3) を使うことによってもできる)、「`ssh $RHOST w`」を実行すれば、G401 の全てのマシンを対象に w コマンドを動かすシェルスクリプトができる。つまり、以下のような形になる。

```
N=1
while [ $N -le 61 ]; do
        ··· 変数 RHOST にホスト名「gpc1x○○」を代入 ···
        echo --$RHOST--
        ssh $RHOST w
        N=‘expr $N + 1‘
done
```

これで基本的には OK なのだが、しかし、各マシンに対して ssh で w コマンドを実行しようとしているとき、そのマシンの Linux が立ち上がっていないと、「`ssh $RHOST w`」に応答が返ってこないため、タイムアウトまで長時間待たされてしまう (どのくらい長いかは設定によっても違うが)。

そこで、それを避けるため、ping コマンドであらかじめそのマシンが立ち上がっているかどうか検査し、立ち上がっているマシンに対して**だけ** ssh で w コマンドを実行する (そして、立ち上がっていないマシンに対しては「`echo --$RHOST--`」の部分も**実行しない**)、というようにしよう。

ping とは、相手のマシンまでネットワークがつながっているかどうかの確認に用いられるコマンド。G401 内では「`ping -c1 -w1 ホスト名`」とすると、そのマシンへの通信を試み、1 秒以内にそのマシンから応答があれば戻り値として 0 (真) を、なければ戻り値として非 0 (偽) を返す[63](このとき、標準出力にもメッセージを表示するが、それは今回の目的には不要なので、/dev/null にリダイレクトして捨ててしまおう)。

そこで、この ping コマンドの戻り値が真のときにだけ ssh で w を実行すれば、1 マシンにつき最大 1 秒待たされるだけで済む。

完成したスクリプトを (引数なしで) 実行したときの出力は、例えば以下のようになるだろう。

```
--gpc1x14--
 09:47PM  up 1 day,  5:13,  2 users,  load average: 0.69, 0.43, 0.23
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
takasu   :0       -                29May09   ?     0.00s   ?     -
takasu   pts/0    :0.0             29May09 29:01m  2.03s  0.02s  bash
takasu   pts/1    :0.0             29May09 28:38m  4.25s  1.40s  mathematica
--gpc1x21--
 09:47PM  up  8:29,  2 users,  load average: 1.38, 1.40, 1.51
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
kako     :0       -                01:19PM   ?     0.00s   ?     -
kako     pts/0    :0.0             01:20PM  2:28m  2:16m  2:14m  ./a.out
--gpc1x37--
 09:47PM  up  2:14,  2 users,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
wd       :0       -                07:34PM   ?     0.00s   ?     -
wd       pts/0    :0.0             07:35PM 13:18  21.33s  0.18s  bash
--gpc1x61--
 09:47PM  up  3:58,  1 user,  load average: 0.00, 0.00, 0.00
```

[63]ただし一般には、ping コマンドに `-w` オプションがない場合もあるし、管理の方針上、ping に応答しない設定になっているマシンもある。また、ping からの応答が返ってきても、次に ssh で入ろうとするまでの一瞬の間に相手が落ちるということもあり得るなど、さまざまな理由から、この方法は完全ではない。だが、今回はそこまでは考えないことにしよう。(ちなみに、相手のマシンが ssh での接続を受け付ける状態かどうかを知るには、ssh-keyscan コマンドを使う方法もある。というより、こちらの方が適切かも。)

```
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
muramatu :0       -                 08:52PM   ?     0.00s   ?     -
muramatu pts/0    :0.0              09:10PM 36:44   5.65s  5.63s  xemacs -nw
```

3.7　AWK による文字列処理—プロセスの階層型表示

ips というシェルスクリプトが既に作ってある (~nide/jugyo/jjikken1-14/sample/ips に置いてある。名前は indented ps の略のつもり)。これは、「ps␣xwwl」というコマンド (自分が実行しているプロセスの一覧を表示) の出力を、(ちょっと長めの) AWK プログラムで処理し、プロセス間の**親子関係**をインデント (段付け) で表して出力するもの。例えば ps xwwl の出力が

```
  F   UID   PID  PPID PRI  NI    VSZ  RSS WCHAN  STAT TTY        TIME COMMAND
140  1317 26648 26646   9   0   6196 2084 do_sel S    ?          0:00 sshd -i -4
000  1317 26649 26648   9   0   5224 2540 do_sel S    ?          0:00 kterm -km
000  1317 26663 26649   8   0   3728 1628 read_c S    pts/1      0:00 -tcsh
140  1317 27630 27628   9   0   5580 1616 do_sel S    ?          0:00 sshd -i -4
000  1317 27631 27630   9   0   5104 2324 do_sel S    ?          0:00 kterm -km
000  1317 27645 27631   8   0   3696 1560 read_c S    pts/0      0:00 -tcsh
100  1317 28103 28099   9   0   8748 4456 do_pol S    ?          0:00 /usr/bin/gnome-session
000  1317 28149 28103   9   0   5108 2804 do_sel S    ?          0:00 kinput2
000  1317 28156     1   9   0   6464 2308 do_sel S    ?          0:00 gnome-smproxy
000  1317 28158     1   9   0   7704 3192 do_pol S    ?          0:00 magicdev
000  1317 28171     1   9   0   3432 1520 do_pol S    ?          0:00 gnome-name-service
000  1317 28174     1   9   0   7056 4116 do_sel S    ?          0:00 sawfish
000  1317 28187     1   9   0   3928 2028 do_sel S    ?          0:00 xscreensaver
000  1317 28195     1   9   0   9960 5636 do_pol S    ?          0:02 panel
000  1317 28197     1   9   0  10024 5372 do_pol S    ?          0:00 gmc
000  1317 28202     1   9   0   8880 4160 do_pol S    ?          0:00 tasklist_applet
140  1317 28519 28517  10   0   5888 1888 do_sel S    ?          0:00 sshd -i -4
000  1317 28520 28519   9   0   5204 2520 do_sel S    ?          0:00 kterm -km
000  1317 28534 28520  12   0   3896 1772 rt_sig S    pts/2      0:00 -tcsh
000  1317 29329 28187  17  10   4024 2236 do_sel SN   ?          0:00 ccurve
000  1317 29385 28534  14   0   2908  948 wait4  S    pts/2      0:00 /bin/sh
000  1317 29386 29385  14   0   3208 1220 -      R    pts/2      0:00 ps xwl
040  1317 29387 29385  14   0   2908  960 wait_o D    pts/2      0:00 /bin/sh
000  1317 29388 29385  15   0   2600  700 pipe_w S    pts/2      0:00 awk
```

のようなものだったとする (ps コマンドはシステムによってだいぶ仕様が違うのだが、とりあえず G 棟システムのマシンのものを前提に考える)。第 3 フィールドはプロセス番号、第 4 フィールドは親プロセスの番号、第 13 フィールドがコマンド名になっている。AWK 側では、これをもとにプロセスの親子関係を調べ、

```
28158 magicdev
27630 sshd
27631     kterm
27645         -tcsh
28195 panel
28187 xscreensaver
29329     ccurve
28197 gmc
28103 /usr/bin/gnome-session
28149     kinput2
28519 sshd
28520     kterm
28534         -tcsh
29385             /bin/sh
29386                 ps
```

```
29387                 /bin/sh
29388                 awk
28202 tasklist_applet
26648 sshd
26649     kterm
26663         -tcsh
28171 gnome-name-service
28174 sawfish
28156 gnome-smproxy
```

のように、インデントで親子関係を表現して出力する。この例では、プロセス番号 28519 の sshd の子プロセスとして、プロセス番号 28520 の kterm があり、そのさらに子がプロセス番号 28534 の -tcsh、その子としてプロセス番号 29385 の /bin/sh がある。そしてその子プロセスは、プロセス番号 29386 の ps と同 29387 の /bin/sh と同 29388 の awk の 3 つがある、といったことがわかる。

3.7.1 処理内容—AWK の配列と for 文

この ips シェルスクリプトの処理内容を調べてみよう。スクリプトは以下の通り。

```
#!/bin/sh
# 自分のプロセス一覧を(親プロセス番号込みで)出力させ、
# AWKで加工させインデント付けを行う
ps xwwl | awk '
        NR == 1{ # psの出力中1行目(ヘッダ行)は読み飛ばす
                next
        }
        { # 各プロセスのプロセス番号からコマンド名および親プロセスへの対応を
          # 連想配列に保存
                cmd[$3] = $13
                parent[$3] = $4
        }
        END{
                for(pid in cmd){ # 各プロセス番号pidについて…
                        if(!(parent[pid] in cmd)){
                                # pidの親プロセスが配列cmdの添字として記録
                                # されてないなら、pidは自分のプロセスの中で
                                # 一番上の親にあたるプロセスなので、それを
                                # 出発点としたプロセスの親子構造を出力
                                print_cmd_and_children(0, pid)
                        }
                }
        }

        function print_cmd_and_children(indentlevel, pid,       otherpid){
                # pidを一番上の親プロセスとしたプロセスの親子関係を出力。
                # pid自身のインデントレベルをindentlevelとし、1つ下の子を
                # 1段分のインデントで表示する
                # 実際のインデント量は、インデントレベル×4個分の空白とする
                printf "%5d %*s%s\n", pid, indentlevel*4, "", cmd[pid]

                # 配列cmdの添字として記録されている他の各プロセスotherpidに
                # ついて…
                for(otherpid in cmd){
                        # otherpidの親プロセスがpidであれば、otherpidは
```

```
                        # pidの子なので、otherpid以下の親子関係を、
                        # print_cmd_and_childrenの再帰呼び出しによって出力
                        if(parent[otherpid] == pid)
                                print_cmd_and_children(indentlevel+1, otherpid)
                }
        }
'
```

AWK プログラムの部分 (「awk '…'」の「' '」の中) が二十数行にわたっている。パターン$_1$ にあたる「NR == 1」の部分は「その行が入力の 1 行目ならば」の意味である。それに対応するアクション$_1$ に書かれている「next」は、「これが実行されると、現在の行に対する処理を終了する」(それ以降のパターンは検査されず、対応するアクションも実行されない) という働きを持つ。つまり、これによって、ps コマンドの出力の 1 行目の「F UID PID PPID…」の行を読み飛ばしている[64]。

パターン$_2$ は略されているので、アクション$_2$ (注釈「…連想配列に保存」のところ) は入力の各行 (ただし、入力の 1 行目はアクション$_1$ によって読み飛ばされるので除く) に対して行われる。処理の内容は配列への代入で、あるプロセス番号のコマンド名は何か、親プロセスはどれか、といった情報を配列変数に記録する。この結果、配列変数 cmd と parent は以下のようになっていく。

	cmd[26648]	cmd[26649]	cmd[26663]	
cmd	"sshd"	"kterm"	"-tcsh"	…

	parent[26648]	parent[26649]	parent[26663]	
parent	26646	26648	26649	…

AWK の配列は宣言はいらないし、添字をとびとびに使っても構わない (例えば a[1000] と a[2000] が存在するのに、a[1001] ～ a[1999] は存在しない、ということもよくある[65])。さらに、ここでは使っていないが、AWK の配列は、添字が文字列であっても構わない[66](例えば a["hahaha"] という配列要素を作っても構わない)。

最後に END パターンのアクションで、存在する各プロセスについて (「for(pid in cmd)」のところ)、もしそのプロセスに親がいなければ、print_cmd_and_children という関数でそのプロセスおよび子孫のプロセスの情報を出力する。AWK の for 文には、以下の 2 通りの互いに全く違う形式があり、

i) 「for(式$_1$; 式$_2$; 式$_3$) 文」の形。C 言語の for 文とほぼ同じ

ii) 「for(変数名 in 配列) 文」の形。配列の各要素について、その添字を in の左辺の「変数名」の変数に代入しながら、本体の文を実行する。シェルスクリプトの for 文に似ている。ただし、配列の要素の選ばれる順序は**順不同**

そしてここの for 文は形式 ii) の方である。配列 cmd の要素の添字を 1 つずつ (順不同に) 選びながら、その添字を変数 pid に代入して、for 文の本体を実行する。

for 文の本体内の if 文の条件式にも「式 in 配列」が使われているが、これは上記の for 文 ii) とは別物。これは、その配列に、in の左辺の式の値を添字に持つ要素があれば真となる条件式である。ここでは、cmd[parent[pid]] という配列要素が存在すれば print_cmd_and_children 関数を呼んでいる。

「function …」の部分以降は AWK の関数定義で[67]、ここで print_cmd_and_children という関数を定義

[64] 実はこんなことをしなくても、ps コマンドに h オプションを付けておけばいいのだが (さらに、そもそも最近の ps コマンドには、それ自身でインデントつき出力の機能を持つものもあるのだが)、AWK によるテキスト処理の例を兼ねてこのままにしてある。

[65] もちろん、存在しない要素がメモリを余分に消費したりはしない。

[66] そのような配列のことを「連想配列」あるいは「ハッシュ」「辞書」などと呼ぶ。言語によっては、連想配列と普通の配列の両方を持つものもあるが、AWK の配列は連想配列だけである。

[67] 非常に古い AWK には関数定義がない。が、そのような古い AWK は現在はほとんど使われない。ただ、システムによっては、awk という名のコマンドは古い AWK で、関数定義の使える新しい AWK は nawk というコマンド名になっているものもある。

している。引数にはインデントのレベルとプロセス番号を与えておき、プロセスの番号に続けてコマンド名を所定のインデント (インデントのレベル × 4 個分のスペース) で出力してから、そのプロセスの子孫を、print_cmd_and_children 関数の再帰呼び出しで出力する。ちなみに、この関数の定義の本体の中には for 文が 2 つあるが、1 つ目が形式 i) で、2 つ目が形式 ii)。

なお、AWK では、関数内**ローカル変数**の宣言は、引数リストに余分な引数を加えることで行う。この例では print_cmd_and_children() は 2 引数で呼び出されるため、otherpid がローカル変数。これを他の引数よりちょっと離して書くのは、ローカル変数であることがわかりやすいようにするための、AWK 独特の慣習である。

3.7.2 拡張問題

さて、このスクリプトを、コマンドの引数まで出力するように改良できるだろうか。つまり、例えば

```
28158 magicdev
27630 sshd -i -4
27631     kterm -km
27645         -tcsh
28195 panel
   ⋮
```

のような出力をするようにできるだろうか。

ps xwwl の出力にはコマンドの引数が含まれているのだから、それがこのシェルスクリプトの出力にも出てくるように改造することになる。`cmd[$3]` に現在はコマンド名 (`$13`) しか入れていないが、`$14` 以降存在するところまで全て (空白をはさんで) くっつけて `cmd[$3]` に入れるようにすればよい (ヒント: 2.1.1 節では触れなかった、AWK の「**文字列連接**」を使う)。入力行にフィールドがいくつあるかは、NF という変数でわかる。

第 4 章　応用編

4.1 オプション引数の解析

シェルスクリプトに限らず何かコマンドを作る際、**オプション**によって振る舞いを微妙に変更できるように作ることは多い。そのようにしたい場合、C 言語だと getopt 関数を用いるのが手軽だが、シェルスクリプトの場合、組み込みコマンド getopts でほぼ同様のことができる[68]。

下記は getopts を用いた、説明用のあまり意味のないシェルスクリプトの例 (`~nide/jugyo/jjikken1-14/sample/gosmp` にあり)。

```
#!/bin/sh
HAHAHA=f; KEKEKE=f; FOFOFO=f; EHERA=f # 変数名はわざと変なのにしてある
while getopts xza:c: FUHYO; do
        if [ "$FUHYO" = x ]; then
                HAHAHA=t
        elif [ "$FUHYO" = z ]; then
                KEKEKE=t
        elif [ "$FUHYO" = a ]; then
```

[68] 古いシェルには getopts がないものもある。

```
                FOFOFO=t
                FOFOFO_ARG="$OPTARG"
        elif [ "$FUHYO" = c ]; then
                EHERA=t
                EHERA_ARG="$OPTARG"
        else # 誤ったオプションが指定された。偽の戻り値で終了する
                exit 1
        fi
done
shift ‘expr $OPTIND - 1‘

if [ "$HAHAHA" = t ]; then
        echo オプションxは指定されています
else
        echo オプションxは指定されていません
fi
if [ "$KEKEKE" = t ]; then
        echo オプションzは指定されています
else
        echo オプションzは指定されていません
fi
if [ "$FOFOFO" = t ]; then
        echo オプションaのオプション引数は"$FOFOFO_ARG"
else
        echo オプションaは指定されていません
fi
if [ "$EHERA" = t ]; then
        echo オプションcのオプション引数は"$EHERA_ARG"
else
        echo オプションcは指定されていません
fi

if [ $# -eq 0 ]; then
        echo オプション以降の引数はありません
else
        echo オプション以降の引数は:
        for i in "$@"; do
                echo "$i"
        done
fi
```

このシェルスクリプトには、サブ引数を取らないオプション引数 x, z と、サブ引数を取るオプション引数 a, c がある。このシェルスクリプトの名前が gosmp (GetOpts SaMPle の略のつもり) だとして、以下を試してみよう。

```
$ gosmp↵                                (引数なし)
$ gosmp -x -z↵
$ gosmp -x -a pon -z pen kan↵
$ gosmp -x -a pon pen kan↵
$ gosmp -x -apon pen kan↵
$ gosmp -xz pen kan↵
$ gosmp -xzapon pen kan↵
$ gosmp -xza pon pen kan↵
$ gosmp -y↵                             (エラーになるはず)
$ gosmp -x -- -z pon pen kan↵
```

getopts の第 1 引数には、そのコマンドのオプション引数 (サブ引数を取るオプション引数には「:」を付ける) を渡し (上の例では「xza:c:」)、第 2 引数には変数名を指定する。

getopts は、シェルスクリプトに渡された引数を先頭から解釈し (次に何個目の引数を解釈するのかをシェル変数 OPTIND に保持する)、オプションが見つかればオプション引数を第 2 引数に指定された変数 (この場合は FUHYO) に代入して、戻り値 0 (真) を返す (そのオプション引数がサブ引数を取る場合は、それをシェル変数 OPTARG に入れる)。オプションでないものが見つかれば、戻り値として非 0 (偽) を返す。(getopts の第 2 引数に指定する変数名は「FUHYO」でなくてもよい。もちろん、もっとわかりやすい変数名の方がいい。これに対し、OPTIND と OPTARG は getopts で特別な役割をするので、これ以外の変数名を使うことはできない。)

よって、while ループで getopts が偽を返すまで繰り返せば、全てのオプションを解析できる。このとき OPTIND には、解析したオプション引数の数 + 1 が入っているので、`shift ‘expr $OPTIND - 1‘` により、オプション引数の部分を捨てることができる。

なお、x, z, a, c 以外のオプションが見つかった場合は、オプションの指定誤りであるから、exit を用いて、偽の戻り値で終了している。このときエラーメッセージは getopts が出力してくれるので、上のシェルスクリプトではあえて自前のエラーメッセージは出力していない。

getopts の振る舞いは C 言語の getopt() とほぼ同じである。特に、「-」で始まるがオプション引数でない引数を与えたい場合、「--」で区切ることによってそれ以降を強制的に「オプション引数ではない」と解釈させることができる (上の最後の例を実際に試してみよう)。

実は出来合いのコマンドにも、C の getopt() を使っているためそれと同じ挙動をするものが多い。例えば cat, cp, rm など基本コマンドも多くはそうで[69]、「-r」というファイルを消すには「`rm -- -r`↵」とする。

4.1.1 応用

3.4 節で作った niji シェルスクリプトを、オプション解析に getopts を使うように書き直してみよう。

また、getopts を使って書き直したシェルスクリプトでは、$y = x^2 - 12x + 5$ のグラフを描くには「`./niji -12 5`↵」ではいけなくなる。それはなぜか、またどうすれば $y = x^2 - 12x + 5$ のグラフを描けるだろうか (答は 4.1 節の中に書いてあることなので探そう)。

4.2 ヒアドキュメントと shar

4.2.1 ヒアドキュメント

シェルスクリプト内で

```
command << 'EOF'
  ...
EOF
```

のように書くと、「...」の部分がそっくりそのままコマンド *command* の標準入力に渡されて実行される。「...」の部分のデータのことを「ヒアドキュメント」と呼ぶ。また、「<<」を使ったこの機能自身のことも「**ヒアドキュメント**」と呼ぶ。

上の「EOF」を「**デリミタ文字列**」といい、「デリミタ文字列」に**ちょうど**一致する行が現れるまでの間がヒアドキュメントになる。デリミタ文字列は「EOF」である必要はなく、例えば

```
command << 'owari'
  ...
owari
```

[69] 非常に古いシステムなどではそうでないこともある。

でもよい。ただし、デリミタ文字列とヒアドキュメントの終わりはぴったり一致せねばならないので、例えば

```
command << 'EOF'
  ...
EOF␣
```

ではだめ (「EOF」の後ろに空白が余分にあるのでデリミタ文字列「EOF」と一致しない)。

ヒアドキュメントは、定型の、比較的長いデータをコマンドに渡すときに用いられる。例えば

```
#!/bin/sh
cat << 'EOF'
こんにちは。
只今の時刻をお知らせします。
EOF
date
```

というシェルスクリプトは

```
こんにちは。
只今の時刻をお知らせします。
2008年 10月 12日 日曜日 20:07:35 JST
```

のような出力をする[70]。

定型メッセージの長さがこの程度であれば

```
#!/bin/sh
echo 'こんにちは。'
echo '只今の時刻をお知らせします。'
date
```

あるいは

```
#!/bin/sh
echo 'こんにちは。
只今の時刻をお知らせします。'
date
```

のようなスクリプトでも同じことができるが、メッセージが長かったり、メッセージ中に「'」が混じったりする場合は、ヒアドキュメントを用いたスクリプトの方がわかりやすいと言えよう。

4.2.1.1　ヒアドキュメントのバリエーション

• ***クォートしないデリミタ文字列***　デリミタ文字列を「' '」で囲まないこともできる。ただしその場合、ヒアドキュメント内の「$」「\」やバッククォート「` `」は特別な扱いを受ける (1.7 節に出てきた「" "」の中と同じ扱いになる)。例えば

```
#!/bin/sh
cat << EOF
こんにちは。
只今の時刻は`date`です。
EOF
```

というスクリプトの出力は、「`date`」の部分が date コマンドの出力に置き換えられて以下のようになる。

```
こんにちは。
只今の時刻は2008年 10月 12日 日曜日 20:07:35 JSTです。
```

この記法は便利ではあるが、ヒアドキュメント内に特殊文字を気にせずに任意の文字を入れたい場合には向かない。本資料ではこの記法についてはこれ以上扱わない。

[70] ここでは、date コマンドが日本語の出力を行うものと仮定している。

• ***ヒアドキュメントのインデント*** ヒアドキュメントの「<<」の代わりに「<<-」を使うと、ヒアドキュメントのデータ部 (データ部を終えるデリミタ文字列の部分を含む) の**各行頭のタブ文字**が取り除かれる。

「タブ文字」とは ASCII 文字コード 9 の特殊文字、C 言語では「\t」と書かれる文字で、この文字が端末などに出力されると、カーソルが「タブストップ」と呼ばれる特定の位置 (標準的には行左端から 8, 16, 24, ··· 文字目の位置) へ動く。従って、タブ文字はインデントに多用される。Emacs でテキストファイルを編集している場合には、Tab キーでタブ文字を入力できる[71]。例えば

```
$ emacs fofofo↵
```

として fofofo ファイルを編集開始し、a b Tab c ↵ の順にキー入力すると、Tab のところでカーソルが左から 8 文字目に飛ぶ。これは「タブ文字」という 1 文字が入力されたのであって、スペース (空白文字) が複数個入力されたのではないことに注意。

例えば

```
#!/bin/sh
cat <<- 'EOF'
        こんにちは。
        只今の時刻をお知らせします。
EOF
date
```

というスクリプト (ただし、ヒアドキュメントのデータ部の**各行頭**には空白文字ではなくタブ文字があるものとする) の実行結果は、タブが取り除かれるため、先のスクリプトと同様、

```
こんにちは。
只今の時刻をお知らせします。
2008年 10月 12日 日曜日 20:07:35 JST
```

のようになる。

ただし、取り除かれるのは行頭の**タブ文字だけ**であり、空白文字は取り除かれない。従って、画面上では同じように見えても、データ部の行頭が空白文字である場合、すなわち

```
#!/bin/sh
cat <<- 'EOF'
␣␣␣␣␣␣␣␣こんにちは。
␣␣␣␣␣␣␣␣只今の時刻をお知らせします。
EOF
date
```

のようなスクリプトの実行結果は、行頭の空白が取り除かれず

```
␣␣␣␣␣␣␣␣こんにちは。
␣␣␣␣␣␣␣␣只今の時刻をお知らせします。
2008年 10月 12日 日曜日 20:07:35 JST
```

のような出力になってしまう。

また、行頭のタブ文字はいくつあっても全て取り除かれるので

```
#!/bin/sh
if [ "$1" = w ]; then
        cat <<- 'EOF'
                こんにちは。
                只今のログイン状況をお知らせします。
```

[71] とはいうものの、Fundamental モード (1.3.1.2 節) 以外の場合はそうとは限らない。特に、Emacs が各種プログラミング言語用のモードになっている場合、Tab キーを押した場合の動作はタブ文字の挿入ではなく、インデントの実施であることが多い。シェルスクリプトモード (1.3.1.2 節) でもそうで、標準では Tab キーを押した場合の動作はインデントの実施であり、タブ文字の挿入ではない。ただし、G 棟システムの Linux の Emacs は、シェルスクリプトモードで Tab キーを押した場合は、**タブ文字の挿入**が行われるように設定されている。なお、Tab キーの動作がタブ文字の挿入でない場合であっても、Ctrl-Q Tab とすれば必ずタブ文字の挿入ができる。

```
        EOF
        w
else
        cat <<- 'EOF'
                こんにちは。
                只今の時刻をお知らせします。
        EOF
        date
fi
```

のようなスクリプトも書ける (これも、各行頭がタブ文字でなく空白文字だと意図通りに動作しない)。このスクリプトは、もし「<<-」でなく「<<」を使って書くと

```
#!/bin/sh
if [ "$1" = w ]; then
        cat << 'EOF'
こんにちは。
只今のログイン状況をお知らせします。
EOF
        w
else
        cat << 'EOF'
こんにちは。
只今の時刻をお知らせします。
EOF
        date
fi
```

のようにヒアドキュメント部だけ**インデントなし**にせねばならないため、スクリプトの構造がわかりにくくなってしまう。

以上のように、「<<-」を使うとヒアドキュメントを自然にインデントすることができる。本資料では以後この記法は出てこないが、知っておくと便利である。

4.2.2 shar

複数のファイルを 1 まとめにしておき、あとで元々のファイルを復元できるようにしたファイルをアーカイブといい、アーカイブを作るソフトをアーカイバという。tar (表 0.1) や zip、それに Windows でかつてよく用いられていた LHa などはアーカイバの一種。

さて、shar という、シェルの機能を使った (テキストファイル用の[72]) アーカイバがある。下記はその原始的な版で、`~nide/jugyo/jjikken1-14/sample/primshar` に置いてある (名称は primitive shar に由来)。

```
#!/bin/sh
for i in "$@"; do
        echo "cat << 'SHAR_EOF' > '$i'"
        cat "$i"
        echo 'SHAR_EOF'
done
```

いま、ファイル abc と def (いずれもテキストファイルとする) があるとき、この primshar を使って

```
$ primshar abc def > xyz↵
```

とすると、xyz が、abc と def をまとめたアーカイブファイルになる。そして、xyz はそれ自体がシェルスクリプトでもあり、

[72] バイナリファイルも扱えるように改良された shar も存在する。

```
$ sh xyz↵
```

とすると、元の abc と def を取り出すことができる。

この primshar は、ヒアドキュメントを用いている。実際に上記のように動かしてみて、その動作を調べてみよう。また、上記でできる xyz ファイルの中身を見て、その動作原理を理解しよう。

4.2.3 簡易版 shar の問題点と改良

4.2.2 節の primshar の問題点の 1 つ、しかも最大の問題点は、たまたま「SHAR_EOF」という行が含まれているファイルに対しては正しく動かない点である。例えば、ファイル abc か def のどちらかに「SHAR_EOF」という行が含まれている場合、「`primshar abc def > xyz`↵」として xyz を生成しても、「`sh xyz`↵」はファイル abc と def を正しく復元しない。まずは、それがなぜであるか考えよう。

では、primshar 中の「SHAR_EOF」を別の文字列、例えば「HAHAHA」に変えれば解決するだろうか? そうすると今度は、たまたま「HAHAHA」という行を含むファイルに対して正しく動かなくなる。どのような文字列をデリミタ文字列として使っても、それとたまたま一致する行を含むファイルはありうるのだから、任意のファイルに対して正しく動作するようにはできない。

そこで、別な方法を使って、どのような内容のテキストファイルに対しても primshar が正しく動くようにしよう。それには、アーカイブを作る際に、各ファイルの各行の先頭に「X」という文字を付加し、アーカイブからファイルを復元する際には各行の先頭の「X」という文字を除去すればよい。前者は「`sed␣-e␣'s/^/X/'␣`ファイル名」でき、後者は「`sed -e 's/^X//'`」でできる。こうすれば、アーカイブするファイル中にたまたまデリミタ文字列と同じ「SHAR_EOF」という文字列があっても、それは「XSHAR_EOF」に変換されるので、デリミタ文字列と間違われることはない。

このようにして実際に primshar を改良してみよう。

4.3 シェル関数

シェルには関数定義もあって、

```
関数名 (){
        関数の本体となるコマンド列
}
```

のように書く。関数はコマンドと同様に使える。関数の定義は、それが使われるより前でなければならない。関数の本体内では $1, $2, ··· は、シェルスクリプトへの第 1, 2, ··· 引数ではなく、**関数に渡された**第 1, 2, ··· 引数を参照する。

例えば、以下のようなシェルスクリプト threetimes があるとする。

```
#!/bin/sh
echo3(){
        echo "$1 $1 $1"
}

for i in "$@"; do
        echo3 "$i"
done
```

これを「`threetimes fuhyohyo sayonara`↵」と実行すると、以下の出力を得る。

```
fuhyohyo fuhyohyo fuhyohyo
sayonara sayonara sayonara
```

もう 1 つの例。1.8 節では改行なしのエコーをやるのに echo -n を使ったが、実は**改行なしのエコー**をやる方法はシステムによって違っていて、2 種類ある。あるシステムでは、改行なしのエコーをやるには

```
echo 'メッセージ\c'
```

とせねばならず、「echo -n 'メッセージ'」では「-n メッセージ」と (改行つきで) 表示されてしまう[73]。

そこで、どちらの流儀のシステムでも改行なしのエコーを行えるシェル関数を作り、これを使って 1.8 節のシェルスクリプトを改造しよう。

```
#!/bin/sh
if [ "`echo -n`" = -n ]; then
        # echo -nが「-n」を出力した。従って改行なしのechoは
        # -nオプションでは行えず、引数末尾に\cをつけて行う
        echon(){
                echo "$@\c"
        }
else
        # 改行なしのechoは-nオプションで行える
        echon(){
                echo -n "$@"
        }
fi

echon "$1から$1.bakへコピーしますか(Y/N) " >&2
read YN
case "$YN" in
[Yy]*)
        cp "$1" "$1.bak";;
*)
        echo 中止しました >&2;;
esac
```

関数の本体内では、$@ も $1 や $2 と同様、「シェルスクリプトの引数全て」ではなく「関数に渡された引数全て」を表すことに注意しよう。

このスクリプトだと echon が 1 回しか使われてないのであまりうまみはないが、echon が何度も使われるようなスクリプトだと、これをシェル関数にすることでスクリプトが簡潔になり、保守もしやすくなる。

4.3.1　シェル関数の戻り値

シェル関数の本体の中で「return n」とすると、そのシェル関数から戻り値 n で戻る。

以下のシェルスクリプトはこのことを使い、1.8 節のスクリプトのうち、ユーザに問い合わせをする部分をシェル関数として分離したものである (なお、話を簡単にするためここでは、4.3 節で考慮した改行なしのエコーの方式の違いについては考えず、echo -n で改行なしのエコーができるものとした)。

```
#!/bin/sh
confirm(){
        # 第1引数と「(Y/N)」を出力してユーザに問い合わせ、
        # ユーザがYで答えれば真(0)、そうでなければ偽(1)を返すシェル関数
        echo -n "$1(Y/N) " >&2
        read YN
```

[73] 例えば macOS (10.4 以降) の /bin/sh がそうである。従って、echo -n を用いているシェルスクリプトを macOS で動かすと、改行なしのエコーにはならない。ただし、ユーザとの対話に用いられているシェルは bash や zsh なので、端末から直接コマンドを打ち込む場合は echo -n で改行なしのエコーができる。

```
        case "$YN" in
        [Yy]*)
                return 0;;
        *)
                return 1;;
        esac
}

if confirm "$1から$1.bakへコピーしますか"; then
        cp "$1" "$1.bak"
else
        echo 中止しました >&2
fi
```

このスクリプト名がconfirmcpだとして、「`confirmcp abc`↵」とすると、まずシェル関数confirmに「`abc`から`abc.bak`へコピーしますか」が第1引数として渡されて呼び出される。confirm関数は「`abc`から`abc.bak`へコピーしますか`(Y/N)`␣」と改行なしで表示した後、キーボードから1行読んでシェル変数YNに代入し、その値がYかyで始まれば戻り値0(真)、でなければ1(偽)で戻る。この値を使ってif文で分岐している。

なお、4.4.2節で述べる**ローカル変数**の使えるシェルの場合は、confirm関数の本体の先頭行に「`local YN`」と書いて、変数YNをローカル変数にする方がよい。

シェル関数内で、戻り値なしで単に`return`とした場合、あるいは`return`せずに関数の最後まで行き着いた場合は、関数内で最後に実行されたコマンドの戻り値が、関数からの戻り値となる。

4.4 再帰処理

4.4.1 シェル関数の再帰呼び出し

シェルスクリプトと言えど、ときには再帰処理が必要になる。下記は「ハノイの塔」パズルを解くシェルスクリプト(ハノイの塔パズルについては本資料では略する)で、シェル関数の再帰呼び出しによって、再帰処理を実現している。このスクリプトは、`~nide/jugyo/jjikken1-14/sample/hanoi0`に用意してある。

```
#!/bin/sh
hanoi(){
        # 第1引数:円盤枚数 第2引数:動かし元 第3引数:動かし先 第4引数:第3の棒
        if [ "$1" -gt 0 ]; then
                hanoi `expr "$1" - 1` "$2" "$4" "$3"
                echo "$1: $2 -> $3"
                hanoi `expr "$1" - 1` "$4" "$3" "$2"
        fi
}

hanoi "$1" a b c
```

このスクリプトは、第1引数に円盤の枚数nを与えて起動する。3本の棒a, b, cがあって、aにn枚の円盤がはまっているとき、それをbに移し変える手順の列を出力する。

```
$ hanoi 3↵
1: a -> b
2: a -> c
1: b -> c
3: a -> b
1: c -> a
2: c -> b
1: a -> b
```

4.4.2　ローカル変数の利用

4.4.1 節のスクリプトは、円盤の移動のたびに 2 回ずつ、expr コマンドのプロセスを起こすことになるので、効率が悪い。しかし、expr の呼び出し回数を減らそうとして

```
#!/bin/sh
hanoi(){ # 正しく動かない版
        # 第1引数:円盤枚数 第2引数:動かし元 第3引数:動かし先 第4引数:第3の棒
        if [ "$1" -gt 0 ]; then
                N=‘expr "$1" - 1‘
                hanoi "$N" "$2" "$4" "$3"
                echo "$1: $2 -> $3"
                hanoi "$N" "$4" "$3" "$2"
        fi
}

hanoi "$1" a b c
```

としては正しく動かなくなる。hanoi が再帰呼び出しされる過程で、変数 N の値が変わってしまうからである。

シェルによっては、シェル関数内で**ローカル変数**が使えるので[74]、次のように書ける (~nide/jugyo/jjikken1-14/sample/hanoi1 にあり)。これなら正しく動き、しかも、expr の呼び出し回数を減らせるので少し効率が良くなる。

```
#!/bin/sh
hanoi(){
        # 第1引数:円盤枚数 第2引数:動かし元 第3引数:動かし先 第4引数:第3の棒
        if [ "$1" -gt 0 ]; then
                local N=‘expr "$1" - 1‘   # ローカル変数の利用
                hanoi "$N" "$2" "$4" "$3"
                echo "$1: $2 -> $3"
                hanoi "$N" "$4" "$3" "$2"
        fi
}

hanoi "$1" a b c
```

さらに、シェルによっては、整数の演算も expr の助けを借りずに行える。そのためには「$((計算式))」と書く。expr コマンドを使うのと違い、余計なプロセスを起こさなくて済むので効率がよい。なお、この計算機能は、expr コマンドと違って、数や演算子の前後に空白がいらない。例えば「echo $((␣3␣+␣2␣))↵」も「echo $((3+2))↵」もどちらも 5 を出力する。

下記のスクリプトはこの機能を使ったもの。expr を使わなくてよくなったため、上のスクリプトよりさらに効率がよい。~nide/jugyo/jjikken1-14/sample/hanoi2 に用意してある。

```
#!/bin/sh
hanoi(){
        # 第1引数:円盤枚数 第2引数:動かし元 第3引数:動かし先 第4引数:第3の棒
        if [ "$1" -gt 0 ]; then
                local N=$(($1-1))
                # ローカル変数、およびシェル組み込みの整数演算の利用
                hanoi "$N" "$2" "$4" "$3"
                echo "$1: $2 -> $3"
                hanoi "$N" "$4" "$3" "$2"
```

[74]実はシェルのローカル変数のスコープは、動的スコープである。ちょっとややこしい話なのでここでは詳しくは触れない。

```
        fi
}

hanoi "$1" a b c
```

しかし、ローカル変数や「$((···))」を使ったシェルスクリプトはどのシェルでも動くわけではない (Linux, FreeBSD, macOS などでは問題なく動くが) ので、シェルスクリプトの**汎用性をやや損なう**欠点もある。

4.4.3 シェルスクリプト自体の再帰呼び出し

再帰では大抵ローカル変数の必要性が出てくるので、ローカル変数の使えないシェルでは、シェル関数による再帰の実現はしにくい。

シェルスクリプトでの再帰処理には、シェル関数を用いるのではなく、自分自身のスクリプトを呼び出す手法もある。シェルスクリプト自身のファイル名が特殊シェル変数 $0 に入っていることを利用する。この方法なら、ローカル変数の使えないシェルでも動くので、汎用性が上がる利点がある。ただし、シェル関数による再帰と違って、再帰のたびに別プロセス (シェルの) を起こすことになるので効率はかなり悪く、実用性は落ちる。このスクリプトは、~nide/jugyo/jjikken1-14/sample/hanoi3 に用意してある。

```
#!/bin/sh
# ユーザがこのスクリプトを実行する際は引数を1つ与える 引数は円盤の枚数
# このスクリプトは引数として動かし元・動かし先・第3の棒の3引数を加えて
# 自分自身を4引数で起動し 以下第1引数を1ずつ減らしながら4引数で再帰する
if [ $# -eq 1 ]; then   # 引数が1つの場合
        # 引数にa b cを加えた4引数で自分自身を起動し直す
        "$0" "$1" a b c
else                    # 引数が4つの場合、第1引数を1ずつ減らしながら再帰
        if [ "$1" -gt 0 ]; then
                N=‘expr "$1" - 1‘
                "$0" "$N" "$2" "$4" "$3"        # 自分自身を呼ぶ
                echo "$1: $2 -> $3"
                "$0" "$N" "$4" "$3" "$2"        # 自分自身を呼ぶ
        fi
fi
```

このスクリプトの場合、「sh␣スクリプト名␣引数↵」ではなく、スクリプト名自身をコマンドとして起動できるようにしてある必要がある (つまり、chmod などによって実行可能属性を付けておかねばならない)。

4.5 データ処理

データ処理も、簡単なものならシェルスクリプトで可能である。

データ処理ではしばしば配列変数が必要になるが、もともとの UNIX の B シェル (sh) には配列変数はなかった。C シェルにはもとから配列変数がある。この点、スクリプトプログラミングにおける、C シェルの B シェルに比べてのほぼ唯一のアドバンテージである。

が、スクリプトプログラミングの道具としての C シェルの欠点は、配列が扱える利点に比してもあまりに大きい。また、シェルスクリプトで配列を使う場合、シェルそのもので配列を扱うことは実はあまりなく、AWK などのスクリプト言語でその能力を補うのが普通である。さらに、B シェル系でも最近のもの (bash など) では配列が扱えるものもある[75]。従って、配列が使えることだけを理由に、スクリプトプログラミングの道具として C シェルを選ぶ意味はほとんどない。

ここでは bash によるデータ処理の例、および AWK によるデータ処理の例を取り上げる。

[75] bash にも以前は配列はなかったが。

4.5.1　bash スクリプト

bash を使ったシェルスクリプト (**bash スクリプト**) を書くには、スクリプトの先頭に「`#!/bin/bash`」と書くことになる。··· と言いたいところだが、システムによっては bash が /usr/local/bin/bash や他の場所にあったりするので、スクリプトの先頭が「`#!/bin/bash`」では動かない可能性もある。

これに対処する手としてよく使われるのは、スクリプトの先頭に

```
#!/usr/bin/env bash
```

と書く方法である。これなら、bash がコマンドサーチパス (環境変数 PATH で指定されているディレクトリ群。1.1.3 節) 中のどこにあっても動いてくれる。

もちろん、そもそも bash がそのシステムにインストールされていなかったり、あるいはコマンドサーチパス内になかったりすると、こう書いても動かない。

ところで、Linux では /bin/sh の正体は bash であることが多い。その場合は、シェルスクリプトの先頭を

```
#!/bin/sh
```

と書いても、bash の機能を使ったスクリプトが動く。しかし、そのスクリプトが**汎用性を失う** (どのシステムに持っていっても動くというわけではなくなる) ことは承知しておく必要がある。Linux でも /bin/sh の正体が bash でないこともあるし、Linux 以外のシステムでは、むしろ /bin/sh の正体が bash でないことの方が普通なので。

この資料ではその点を承知の上で、bash スクリプトの先頭も「`#!/bin/sh`」と書くことにする。

4.5.2　bash での配列の利用

bash での配列は 1 次元。「変数名`[`添字`]=`値」で自動的に配列変数が作られ、値が代入される。添字は 0 以上の整数。代入する値は任意の文字列でよい。普通のシェル変数と同様、代入時の「`=`」の左右に空白を入れてはならない。

配列の宣言は必要ないし、使う添字は飛び飛びでもよい。例えば `a[2]` と `a[4]` は存在するが、`a[0]`, `a[1]`, `a[3]` はまだ存在しない、といったことも起こりうる。

配列の要素の値を取り出すには「`${`変数名`[`添字`]}`」とする。普通のシェル変数の値を取り出すには「`$`変数名」でも「`${`変数名`}`」でもいいが、配列の要素の場合は、値を取り出すのに「`$`変数名`[`添字`]`」ではだめで、「`${`変数名`[`添字`]}`」のように「`{ }`」が必要なことに注意。

添字のところに「`@`」と書くと (つまり「`${`変数名`[@]}`」とすると)、配列の全ての要素の値を一括して取り出せる。シェルスクリプトの全ての引数を取り出す「`"$@"`」のときと同じ理由で、「`${`変数名`[@]}`」も、要素に決してシェルにとっての特殊文字が混じらないとわかっている場合を除き、特に理由がない限り「`" "`」で囲んで「`"${`変数名`[@]}"`」のように使うのが望ましい。また、「`${#`変数名`[@]}`」で要素の個数がわかる。

下記の bash スクリプトは配列変数の使用例。

```
#!/bin/sh
WAHAHA[2]=hi
WAHAHA[4]=hello
WAHAHA[7]=goodbye

echo ${WAHAHA[2]}       #  「hi」と表示
echo ${WAHAHA[@]}       #  「hi hello goodbye」と表示
echo ${#WAHAHA[@]}      #  「3」と表示
```

read コマンドで、普通のシェル変数でなく配列に値を読み込むには、「`read -a` 変数名」とする。1 行入力を待ち、それを空白で区切って、配列の添字 0, 1, 2, ··· の要素に順に代入する。

例えば、以下のような bash スクリプトがあって、arraysample という名だとする (`~nide/jugyo/jjikken1-14/sample/arraysample` に用意してある)。

```
#!/bin/sh
read -a KERAKERA
echo 添字0と1の要素は "${KERAKERA[0]}" "${KERAKERA[1]}"
echo 全ての要素は "${KERAKERA[@]}"
echo 要素の個数は ${#KERAKERA[@]}

i=0
while [ $i -lt ${#KERAKERA[@]} ]; do
        echo 添字$iの要素は "${KERAKERA[$i]}"
        i=$(($i+1))
done

dispall(){
        local i
        for i in "$@"; do
                echo 各要素 "$i"
        done
}
dispall "${KERAKERA[@]}"
```

これを「`echo ab c defg hij | arraysample`↵」として実行すると、下記のような結果になる。

```
添字0と1の要素は ab c
全ての要素は ab c defg hij
要素の個数は 4
添字0の要素は ab
添字1の要素は c
添字2の要素は defg
添字3の要素は hij
各要素 ab
各要素 c
各要素 defg
各要素 hij
```

添字が 0, 1, 2, $\cdots$ と使われるため、while ループで i の値が要素の個数より小さい間繰り返すことによって、全ての要素を出力できることに注意。

bash では、4.4.2 節で述べた**ローカル変数**や、「`$((`$\cdots$`))`」による**整数の演算**なども使える。上の例でも使った。今回は、bash を前提とするわけなので、4.4.2 節でも述べたように、整数の演算は expr を使わずに「`$((`$\cdots$`))`」を用いる方が効率がよい。

> 4.4.2 節でも述べたが、bash の整数演算、つまり「`$(( ))`」による演算機能は、expr と違って、数式内に空白を入れても入れなくてもよいことに注意。つまり、例えば「`echo $((3+2))`↵」も「`echo $(( 3 + 2 ))`↵」もちゃんと 5 を出力する。

4.5.3 データ処理

4.5.3.1 データ集計

以下のように、テキストファイルで表形式の数値データのファイルがあるとする。各行のデータの個数は同じであるものと仮定する。

```
164 107 130 187  71
 78 170 133 148  77
 55 184  97  68  90
```

このデータを標準入力から読み、行毎に横方向に合計して、結果として

```
659
606
494
```

を出力する bash スクリプトは、下記のように作れる。

```
#!/bin/sh
while read -a a; do  # 標準入力から1行読むたびに下記を実行
        line_t=0        # 合計用変数。最初は0を代入
        i=0             # 添字。0, 1, 2…と増えていく
        while [ $i -lt ${#a[@]} ]; do
                line_t=$(($line_t + ${a[$i]}))  # line_tに加算
                i=$(($i+1))
        done
        echo $line_t  # 合計を出力
done
```

このスクリプトでは、「read は標準入力から **EOF** が来ると戻り値として非 0 (偽) を返す」という性質を使っている[76]。

このスクリプト名を yokototal、データファイル名が data だとすると

```
$ yokototal < data↵
```

で上述のような結果が得られる (上のスクリプトは、`~nide/jugyo/jjikken1-14/sample/yokototal` に用意してある。データファイルは用意してないので、このスクリプトを試したい場合は自分でデータファイルを用意せよ)。ただし、bash の `$(( ))` は整数演算しかできないので、このスクリプトも整数データしか扱えない。

今度は、同じデータを標準入力から読み、列毎に縦方向に合計して、結果として

```
297 461 360 403 238
```

を得る bash スクリプトを作ろう。

```
#!/bin/sh
unset col_t     # 変数col_tをクリアしておく
while read -a a; do
        i=0  # 添字
        while [ $i -lt ${#a[@]} ]; do
                if [ "${col_t[$i]}" = '' ]; then
                        # 配列要素col_t[$i]が不存在なら、加算でなく単なる代入
                        col_t[$i]=${a[$i]}
                else
                        col_t[$i]=$((${col_t[$i]} + ${a[$i]}))
                fi
                # col_t[0]に1列目の合計、col_t[1]に2列目の合計、…が入る
                # a[0], a[1], …は今読んだ行の1, 2, …列目
                i=$(($i+1))
        done
done
# 読み終えた時点でcol_t[0], col_t[1], …に1, 2, …列目の総和が入っている
# これからそれを出力
i=0
```

[76] ちなみにこの性質は、配列でなく普通の変数に読み込む場合も (その場合、bash でない普通の B シェルでも) 成り立つ。

```
while [ $i -lt ${#col_t[@]} ]; do
        echo -n "${col_t[$i]} "   # (i+1)列目の総和を出力
        i=$(($i+1))
done
echo ''
```

bash では echo -n で改行なしのエコーができるので、bash を使うとわかっているのなら、改行なしのエコーは単に echo -n でよく、4.3 節に述べた (改行なしのエコーに関する) 工夫は不要である。

このスクリプトでは、最初に unset コマンド (3.4.3 節) で変数 col_t をクリアしている (unset コマンドは、配列変数/普通の変数の区別なく、シェル変数をクリアできる)。

その後、標準入力から 1 行を配列 a に読み込むたびに、各列の数を、配列 col_t に入っている今までの各列の合計に加算する。ただし、データの 1 行目を処理しているときは、配列 col_t の要素がまだ存在せず、値として参照すると空文字列になる[77]。このときだけは、加算でなく配列 col_t の各要素へ直接代入を行う。

このスクリプト名を tatetotal とすると

```
$ tatetotal < data↵
```

で上述のような結果が得られる。最後の「echo ''」は、これがないと出力の最後の改行が行われないためである。(上のスクリプトは ~nide/jugyo/jjikken1-14/sample/tatetotal に用意してある)

データの個数で割れば、平均の算出もできる。ただし、先と同じ理由 (bash の「$(( ))」は整数演算である) により、平均も整数部までしか出せない。

4.5.3.2 ソート

下記は、標準入力からいくつかの整数を 1 行で読み込み、クイックソート法でソートする bash スクリプトである。シェル関数の再帰呼び出しを行っている。

```
#!/bin/sh
qsort(){ # 配列aの、第1引数の添字から{第2引数-1}の添字までをソート。
        local min=$1; local max=$2

        # 要素が1個以下なら何もしない
        if [ $(($max - $min)) -le 1 ]; then
                return
        fi

        local i=$min; local j=$max

        # $minより小さい要素を左(添字の小さい方)に、大きい要素を右に移動
        while [ $(($j - $i)) -gt 1 ]; do
                if [ ${a[$(($i+1))]} -lt ${a[$i]} ]; then
                        swap $i $(($i+1))
                        i=$(($i+1))
                else
                        j=$(($j-1))
                        swap $(($i+1)) $j
                fi
        done

        # 分けられたそれぞれを再帰的にソート
```

[77] 3.4.3 節で述べたように、未代入の変数の値を参照すると空文字列となるが、このことは配列の要素にも当てはまる。しかも、最初に変数 col_t をクリアしたので、最初は配列 col_t の各要素が未代入であることは保証されている。

```
        qsort $min $i
        qsort $(($i+1)) $max
}
swap(){ # 配列aの、第1引数の添字と第2引数の添字の要素を入れ換え
        local e
        e=${a[$1]}; a[$1]=${a[$2]}; a[$2]=$e
}

# シェル変数aを配列として1行読み込み
read -a a
# 配列aの添字0から添字{aの要素の個数-1}までの間をソート
qsort 0 ${#a[@]}
# 配列a全体を出力
echo ${a[@]}
```

このスクリプトでは、qsort() や swap() が扱う配列の名前は a に固定している。シェル関数に配列とそうでない変数の両方を渡すすっきりした方法がないため[78]、配列を引数で与えてそれをソートさせるという設計にしにくいからである。

このスクリプトの名前を mysort とする (~nide/jugyo/jjikken1-14/sample/mysort に用意してある) と、このスクリプトは

```
$ echo '7 -2 5 9 3' | mysort↵
-2 3 5 7 9
```

のように動作する。

もっとも、今回の場合、sort コマンドを使って次のようにやる方が、同じことをずっと簡単にできるし、手でソートアルゴリズムを組むことに起因するバグも発生しないのだが。(でもそれじゃ配列を使う演習にならないしね)

```
#!/bin/sh
read A
# 空白を改行に変換して、sortで数値としてソートさせ、再び改行を空白に戻す
echo "$A" | tr -s ' ' '\n' | sort -n | tr '\n' ' '
echo ''
```

4.5.4　AWK によるデータ処理

データ処理 (特に配列処理) は、一般にシェルの能力だけでむりやりやるより、AWK などの力を借りる方がずっとやりやすい。下記は AWK を使って、4.5.3.1 節の yokototal と同じ機能を実現したスクリプト。これは bash の機能は使っていないから、bash のないシステム、あるいは /bin/sh の正体が bash でないシステムでも動く。また、bash 版では整数演算しかできないため整数のデータしか処理できないが、AWK 版なら**実数データ**の処理もできる。

```
#!/bin/sh
awk '
        {
                # 入力の各行について、まずline_tをクリアしてから
                line_t = 0
                # 各フィールドをline_tに加算し
                for(i = 1; i <= NF; i++) line_t += $i
                # 合計を出力
                print line_t
        }
'
```

[78] 方法がなくはないのだが、結構ややこしい。

2 つの「'」の間全体が AWK のプログラム。ファイル名が指定されていないので、標準入力が読まれて処理される。パターンが略されたアクションが 1 つあるだけなので、このアクションが標準入力からの各入力行に対して実行される。

このスクリプトは AWK しか使っていない。そういう場合、スクリプトを次のようにも書ける。先頭が「`#!/bin/sh`」ではないことに注意。

```
#!/usr/bin/awk -f
{
        # 入力の各行について、まずline_tをクリアしてから
        line_t = 0
        # 各フィールドをline_tに加算し
        for(i = 1; i <= NF; i++) line_t += $i
        # 合計を出力
        print line_t
}
```

ただしこのスクリプトは、引数を渡した場合の動作が先のスクリプトと異なる。この資料ではこの書き方についてはこれ以上詳しくは扱わない。

AWK の for 文の 2 つの形については 3.7.1 節で述べた。ここでの for 文はそのうち i) の方で、C 言語の for 文と同じ働きのもの。NF は特殊な変数で、フィールドの個数が自動的にこの変数に格納される (3.7.2 節にも出てきた)。`$i` は第 i フィールド。よって、全てのフィールドを変数 `line_t` に合計し、それを出力していることになる。

その次の tatetotal と同じ機能を実現したスクリプトは、次のようになる。

```
#!/bin/sh
awk '
        {
                # 各行の第iフィールドをcol_t[i]に加算
                for(i = 1; i <= NF; i++) col_t[i] += $i
        }
        END{
                # col_t[i]の内容を(存在するだけ)出力
                for(i = 1; i in col_t; i++) printf "%d ", col_t[i]
                # 最後に改行
                print ""
        }
'
```

2.2.6 節で述べたように、AWK の変数は最初は自動的に 0 (数値としては) に初期化されるので、配列 `col_t` の各要素を明示的に 0 に初期化せずに済んでいることに注意。

END パターン (→2.2.6 節) のアクションの中の for 文は、(「`in`」があるので間違いやすいが) 3.7.1 節に出た for 文の 2 つの形のうちの i) の方であり、ii) の方ではない (「`;`」が中に 2 つあるのでわかる)。そして、その中の「式$_2$」にあたる部分が「`i in col_t`」という条件式になっている。これは、`col_t[i]` という配列要素が存在すれば真になる式 (この形の条件式は 3.7.1 節にも登場した)。従って、これによって、存在した列の全ての合計が出力できる。

AWK では、print で出力すると自動的に改行を行ってしまうので、END パターンのアクション中で

```
for(i = 1; i in col_t; i++) print col_t[i]
```

としたのでは、1 つの列の合計を出力するたびに、毎回改行してしまう。これに対し、printf は自動的には改行しないので、こちらを使うことによって、全ての出力を同じ行に出している (なお、実数データを扱う場合は、printf の書式を「`"%f␣"`」などに変える必要がある)。それだけだと出力の最後の改行が起きないので、最後に 1 回、空文字列を print することによって改行している (print は自動改行を行うので)。

4.5.3.2 節のクイックソートと同じ処理は、AWK を使えば次のように書ける (`~nide/jugyo/jjikken1-14/sample/awksort` に用意してある)。

```
#!/bin/sh
awk '
        {
                n = split($0, a) # 行を空白で区切って配列a[1], a[2], …に入れる
                qsort(a, 1, n+1)
                for(i = 1; i <= n; i++) printf "%d ", a[i]
                print ""
        }
        function qsort(a, min, max,     i, j){
                # 配列aの添字minから(max-1)までをソート
                # qsort()は3引数で呼び出されるため、iとjがローカル変数

                if(max - min <= 1) return       # 要素が1個以下なら何もしない

                i = min; j = max

                # a[min]より小さい要素を左(フィールド番号の小さい方)に、
                # 大きい要素を右に移動
                while(j - i > 1){
                        if(a[i+1] < a[i]){
                                swap(a, i, i+1); i++
                        } else {
                                j--; swap(a, i+1, j)
                        }
                }

                # 分けられたそれぞれを再帰的にソート
                qsort(a, min, i)
                qsort(a, i+1, max)
        }
        function swap(a, i, j,          e){
                e = a[i]; a[i] = a[j]; a[j] = e
        }
'
```

シェル関数には、配列とそうでない変数を両方渡す簡単な手段がないため、シェル関数の qsort() にはソートすべき配列そのものを渡せず、qsort() がソートする配列名は固定にせざるを得なかった。AWK は関数に配列とそうでない変数を自然に渡せるので、qsort() 関数を自然に、汎用的に書くことができる。

なお、AWK の関数定義内でのローカル変数の使い方の約束については、3.7.1 節参照。

4.6　環境変数

UNIX では各プロセスが「環境変数」というものを持つ。環境変数は、シェル変数と同様の、名前 (変数名) と値の組である。シェル変数は、シェルだけにある概念で、子プロセスにも伝搬しないのに対し、環境変数は、どのプロセスにもあり、子プロセスにも伝搬する。従って、シェルの環境変数は、シェルから起動したコマンドにも伝搬する。

B シェルでは、組み込みコマンド export を用いて、シェル変数を環境変数に変更することができる (その変数はシェル変数でもあり続ける)。例えばシェル変数 A を環境変数に変えるには

```
export A
```

とする (「export $A」ではないので注意)。このようにして環境変数に変更した、あるいはもともと環境変数であった変数に、シェルスクリプト内で代入を行うと、それ以降にシェルスクリプト内から起動した他

のコマンドにも影響を及ぼす (子プロセス起動時に、環境変数が子プロセスに伝搬されるからである)。ただし、それ以前にシェルスクリプト内から起動されていたコマンドにはもちろん影響を与えない。

また、env コマンドで環境変数の一覧が見られる (ちなみにシェル変数の一覧は set コマンド)。

4.6.1 環境変数による振る舞いの変化

UNIX のコマンドには、環境変数によって振る舞いの変わるものがある。例えば、時刻関係のコマンドは、環境変数 TZ にタイムゾーンを設定しておくことによって、そのタイムゾーンでの時刻表示を行う。下記のシェルスクリプトを試してみよう。

```
#!/bin/sh
date
TZ=UTC; export TZ     # シェル変数TZに値UTCを代入し、TZを環境変数に変更
date                  # 環境変数TZがdateコマンドに伝搬され、dateはその影響を受ける
```

1 度目の date はおそらく日本標準時での表示、2 度目の date は世界標準時での表示である。

このスクリプトでは、2 度目の date の前にシェルの環境変数 TZ も変わっているため、もし 2 度目の date の後にこのシェルスクリプトの続きがまだあるとすると、それ以後に起きる全てのコマンドが環境変数 TZ の変更の影響を受けてしまう。これに対し、「TZ=UTC␣date」のようにすると、シェルのシェル変数や環境変数には全く影響を与えずに、そのときに起動される date コマンドにだけ環境変数 TZ の変更を施すことができる。例えば

```
#!/bin/sh
date
TZ=UTC date      # ここで起動したdateでたけ環境変数TZが変更
date
··· その先延々と続く ···
```

のようなスクリプトにすると、2 度目の date は TZ の変更の影響を受けるが、3 度目の date や、その後に起動されるコマンドは TZ の変更の影響を受けない。スクリプトの中で、ある 1 つのコマンドだけ環境変数を変更して起動したい場合は、この方法を採るべきである。

最近は、環境変数 LANG や LC_ALL で、エラーメッセージなど諸メッセージの言語を変更できるコマンドも多い。例えば

```
#!/bin/sh
cat hahaha
LANG=C cat hahaha
cat hahaha
```

というスクリプトは (ファイル hahaha が存在しないものとすると)

```
cat: hahaha: そのようなファイルやディレクトリはありません
cat: hahaha: No such file or directory
cat: hahaha: そのようなファイルやディレクトリはありません
```

と表示するだろう (ちなみに、LANG より LC_ALL の方が優先される)。

また、

```
#!/bin/sh
TZ=UTC LANG=C date
```

は、環境変数 TZ と LANG の両方をセットして date コマンドを起動する。このように、複数の環境変数をセットしてコマンドを起動することもできる。

なお、C シェル系では、シェル変数や環境変数の設定のやり方は B シェルとずいぶん異なるので注意[79]。

[79] シェル変数や環境変数の設定を、コマンドラインから行う機会は多い。そのため、対話的に使うシェルとして C シェル系を用いる人は、シェル変数や環境変数の設定方法が、(B シェルによる) シェルスクリプトの場合とは異なることに混乱しがちである。

4.7　シェルスクリプトからの GUI の利用

シェルスクリプト内で、「スクリプト言語」と呼ばれる言語を用いて、シェルに足りない能力を補うことがよくある。2 節で扱った AWK 言語がその典型例である。

特に、スクリプト言語の中には、GUI を扱えるものもある。Tcl/Tk や、本節で取り上げる Python (外部モジュール tkinter[80]を取り込むことによって GUI が使える) などがその例である。それらをシェルスクリプト内で用いることによって、シェルスクリプトで GUI を扱うことも可能になる[81]。

ここでは、Python (と tkinter) をシェルスクリプト内で用いて GUI を実現する簡単な例を示す (Python 言語および tkinter の詳細には立ち入らず、既に作成済みの Python プログラムを説明なしに用いる)。以下は、4.3.1 節のシェルスクリプトを変更して、ユーザへの確認をキーボードでなく図 4.1 のようなボタンで行うようにしたものである (`~nide/jugyo/jjikken1-14/sample/confirmcp` にある)。

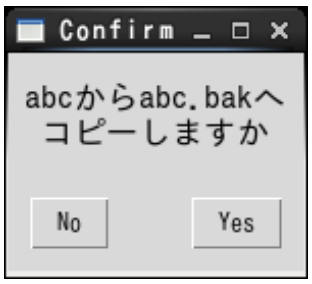


図 4.1: Python+tkinter で作ったボタンの例

```
#!/bin/sh
confirm(){ # Python言語のスクリプトを使い、画面上でユーザに確認を取るシェル関数
        # このシェル関数への第1引数をメッセージとしてウィンドウに表示し、
        # ユーザがYesボタンを押せばこのシェル関数は真を、Noボタンなら偽を返す
        # プログラム中の注釈は日本語EUCで書かれているものと仮定する(そのため
        # 環境変数LANGをja_JP.eucJPに設定の上でpython3コマンドを呼ぶ)
        LANG=ja_JP.eucJP python3 -c 'if True:
                # Tk用モジュールtkinter(とsysモジュール)のインポート
                import tkinter, sys

                # ルートフレーム(一番外側の枠)を生成。タイトルは「Confirm」に固定
                root = tkinter.Tk()
                root.title("Confirm")
                # メッセージ用ウィジェットを生成
                msg = tkinter.Message(root,
                        text = sys.argv[1], # Pythonへの第1引数がメッセージ
                        justify = "center",
                        width = 160,
                        font = "-*-fixed-medium-r-normal--16-*")
                msg.pack(pady = 10)
                # YesとNoのボタンウィジェットを生成
                yes = tkinter.Button(root,
                        text = "Yes",
                        command = lambda: sys.exit(0))
                no = tkinter.Button(root,
```

[80]tkinter は Python 言語の Tk ライブラリであり、すなわち Tcl/Tk の GUI 機能 (Tk) と同じものを Python 言語から利用できるようにしたものである。参考のため、本節の例の Python の部分を Tcl/Tk で書き直したものを `~nide/jugyo/jjikken1-14/sample/confirmcp2` に置いてあるが、記法が違うだけで全く同じことをしているのがわかるであろう。

[81]もっとも、どうせそれらの言語を用いるのなら、シェルスクリプトと併用せず、最初からそれらの言語だけで GUI を実現することの方が多いが。

```
                        text = "No",
                        command = lambda: sys.exit(1))
                yes.pack(pady = 10, padx = 10, side = "right")
                no.pack(padx = 10, side = "left")

                root.mainloop() # 実行開始
        ' "$1"
}

if confirm "$1から$1.bakへコピーしますか"; then
        cp "$1" "$1.bak"
else
        echo 中止しました >&2
fi
```

シェル関数 confirm の中身は、「`python3 -c 'Pythonのプログラム' "$1"`」というコマンドを呼ぶ形になっており、これが、confirm 関数への第 1 引数をメッセージとして表示する図 4.1 のようなウィンドウを作成する。この python3 コマンドは、ユーザが Yes のボタンを押せば 0 (真)、No ボタンなら 1 (偽) を返す[82]。confirm 関数には return がないので、4.3.1 節で述べたように、関数内で最後に実行されたコマンド、つまりこの python3 コマンドの戻り値が、confirm 関数からの戻り値である。この戻り値を用いて if 文で分岐している。

なおこのスクリプトは 4.3.1 節のものと比べ、confirm 関数の定義**だけ**が変わっていることに注意。このスクリプトはユーザ·インタフェースと処理本体を分離し、独立に変更できるようにした例でもあるのである。

付録 A　別表

ここでは、正規表現 (egrep コマンドで使えるもの)、test・expr コマンドのオペレータ、find コマンドの評価式をまとめる。

A.1　正規表現

egrep コマンドでは表 A.1 のような正規表現が使える (抜粋)。AWK の正規表現もほぼ同じ。その他の (正規表現が使える) コマンドや言語などでの正規表現は、この表とちょっとずつ異なる場合がある。

egrep コマンドは、(オプションを除いた) 第 1 引数の正規表現にマッチする文字列を含む行を、第 2 引数以降のファイル (第 2 引数以降がなければ標準入力) の中から探すコマンドである。例えば

`$ egrep 'ab*c'` *filename*↵

で、ファイル *filename* の中から、正規表現 `ab*c` にマッチする文字列を含む行 (つまり、文字列 ac, abc, abbc, abbbc, ··· のうちいずれかがある行) を全て表示してくれる。

表中の**連接** (▲) は、結合優先順位が選択 (「$\alpha|\beta$」の「`|`」) よりも高く、`+` や `*` や `?` よりも低い。

A.2　test コマンドのオペレータ

test コマンドでは表 A.2 のようなオペレータが使える (抜粋)[83]。なお、`!`, `-a`, `-o` は、この順に結合優先順

[82] 実は、このようなことをもっと手軽にできるようにした、zenity というコマンドも作られている。
[83] 非常に古い test コマンドには `-x` あるいは `-e` がないことがある。

表 A.1: egrep コマンドでの正規表現 (抜粋)

正規表現	それにマッチする文字列	例	それにマッチする文字列
特殊文字 (「*」「|」など) 以外の 1 文字	その文字自身	a	a
例えば英字や数字などがこの「特殊文字以外」に含まれる。			
[文字並び]	[ ] 内の文字のいずれか	[abc]	a, b, c
[文字$_1$-文字$_2$]	文字コード順で文字$_1$ と文字$_2$ の間にある 1 文字	[a-f]	a, b, c, d, e, f
[^···]	「···」の部分に当てはまらない文字	[^a-fx]	a, b, c, d, e, f, x 以外の 1 文字
. (ピリオド 1 つ)	任意の 1 文字	.	任意の 1 文字
\特殊文字	その特殊文字自身	\.	.
$\alpha\beta$ (α, β は正規表現とする。以下も同様)	α にマッチするものと β にマッチするものとを**連接**した文字列	ab	ab
α|β	α か β にマッチするもの	ab|cde	ab, cde
α*	α にマッチするものを 0 回以上つなげたもの	ab*c	ac, abc, abbc, abbbc, ···
α+	α にマッチするものを 1 回以上つなげたもの	ab+c	abc, abbc, abbbc, ···
α?	α にマッチするもの 1 回または 0 回	ab?c	ac, abc
( )	優先順位の変更	a(n|pq)*	a, an, apq, ann, anpq, apqn, apqpq, annn, annpq, ···
^	行頭	^a	行頭にある a
$	行末	a$	行末にある a

位が高い。

数や文字列などは、オペレータ (「=」「!=」「-eq」「-a」など) とは別の**独立した引数**として、test コマンドに与えねばならない。例えば「test a = b↵」は文字列「a」と「b」を比較するという意味になって偽となるが、「test a=b↵」はその意味には**ならない**。

A.3 expr コマンドで使えるオペレータ

expr コマンドでは表 A.3 のようなオペレータが使える (抜粋)。

test コマンドと同様、expr コマンドにとっても数やオペレータはそれぞれ**独立した引数**として与えねばならない。例えば「expr 1 + 3↵」は 4 を出力するが、「expr 1+3↵」はその意味には**ならない**。

A.4 find コマンドで使える評価式

find コマンドの使い方は

$ find *directoryname*$_1$ *directoryname*$_2$ ··· *directoryname*$_n$ 評価式↵

である (*directoryname*$_{1\sim n}$ のところには実はディレクトリでないファイル名も書けるのだが、普通はディレクトリ名を書く)。find は、*directoryname*$_{1\sim n}$ の下 (そのさらに下の下の··· も含む) にあるファイル (ディレクトリも含む) の 1 つ 1 つに対し、「評価式」を評価する (ディレクトリ名が 1 つも与えられない場合は、カ

表 A.2: test コマンドでのオペレータ (抜粋)

オペレータ	真になる条件
`-e` ファイル名	そのファイルが存在すれば真
`-f` ファイル名	そのファイルが存在し、普通のファイル (ディレクトリ、シンボリックリンク、デバイスファイルなどでないもの) であれば真
`-d` ファイル名	そのファイルが存在し、ディレクトリであれば真
`-r` ファイル名	そのファイルが存在し、読み取り可能であれば真
`-w` ファイル名	そのファイルが存在し、書き込み可能であれば真
`-x` ファイル名	そのファイルが存在し、実行可能であれば真
ファイル名$_1$ `-nt` ファイル名$_2$	ファイル名$_1$ がファイル名$_2$ より新しいファイルであれば真
ファイル名$_1$ `-ot` ファイル名$_2$	ファイル名$_1$ がファイル名$_2$ より古いファイルであれば真
数$_1$ `-eq` 数$_2$	整数の比較。数$_1$ と数$_2$ が等しければ真
他に `-ne`, `-lt`, `-le`, `-gt`, `-ge` の各オペレータあり (それぞれ $\neq, <, \leq, >, \geq$ を表す)	
文字列$_1$ `=` 文字列$_2$	文字列$_1$ と文字列$_2$ が等しければ真 (一部のシェルでは「`==`」でも可)
文字列$_1$ `!=` 文字列$_2$	文字列$_1$ と文字列$_2$ が等しくなければ真
`!` 式$_1$	式$_1$ が偽ならば真
式$_1$ `-a` 式$_1$	式$_1$ と式$_2$ がともに真ならば真
式$_1$ `-o` 式$_2$	式$_1$ と式$_2$ の少なくとも一方が真ならば真
`\(` 式$_1$ `\)`	式$_1$ が真ならば真。優先順位の変更に使う。「`(`」「`)`」がシェルにとっての特殊文字のため「`\`」でエスケープが必要 (「`'('`」「`')'`」でもよい)

表 A.3: expr コマンドでのオペレータ (抜粋)

オペレータ	意味
式$_1$ `+` 式$_2$	整数の加算。他に `-`, `\*`, `/`, `%` もある。乗算だけ「`*`」でなく「`\*`」である理由は、「`*`」がシェルにとっての特殊文字 (ワイルドカード、0.2.2 節) であるので、その特殊な意味を失わせるため「`\`」でエスケープしなければならないから (「`'*'`」でもよい)。例えば「`expr 3 * 3`」ではだめで、「`expr 3 \* 3`」とする
`\(` 式$_1$ `\)`	式$_1$ を評価。優先順位の変更に使う。上と同様、「`(`」「`)`」がシェルにとっての特殊文字 (コマンドのグループ化、1.4.2 節) のため、「`\`」によるエスケープが必要 (「`'('`」「`')'`」でもよい)。例えば「`expr \( 1 + 2 \) \* 3`」のようにする
文字列 `:` 正規表現	文字列と正規表現とのマッチング。詳細はマニュアル参照のこと。正規表現は egrep のものと細部が異なる

レントディレクトリ以下にあるファイルを対象にする[84])。評価式には表 A.4 のようなもの (抜粋) がある[85]。なお、表中の **AND 結合** (▲) は、結合優先順位が `-o` より高く、`!` より低い。

あるファイルに対する「評価式」の評価は、C での `&&` や `||` を含む式と同じように、全体が真か偽かが定まった時点で終了する (例えば「式$_1$ 式$_2$」の場合、式$_1$ が偽であれば式$_2$ は評価されない)。そこで、例えば

```
$ find . -name '*.c' -print↵
```

の場合、カレントディレクトリ以下にある、「.c」で終わる名前のファイル名を全て表示する。名前が「`*.c`」に当てはまらないファイルに対しては「`-print`」は評価されないためである。

なお、評価式のうち、対象のファイルに対する検査ではなく何らかの副作用を伴う動作を行うもの (表 A.4

[84] システムによってはこの扱いがなく、find にディレクトリ名を明示的に 1 つ以上与えねばならないものもある。

[85] 対話に使っているシェルが C シェル系の場合、状況によっては「`!`」が特別な解釈をされてしまうことがあるので、それを避けるために「`\!`」と打ち込むことが必要な場合がある。

表 A.4: find コマンドでの評価式 (抜粋)

評価式	真になる条件
`-mtime` n	n は (符号なしの) 数。そのファイルが最後に更新されたのが n 日前なら真。n の代わりに「+n」と書くと「n 日前よりも前」、「-n」と書くと「n 日前よりも後」の意味になる
他に `-atime`, `-ctime` もあり	
`-name` *filename*	そのファイルの名前 (ディレクトリ部を除く) が *filename* なら真。*filename* にはワイルドカードが使えるが、その場合は *filename* を「' '」で囲むこと
`-newer` *filename*	そのファイルの最終更新がファイル *filename* より新しければ真
`-perm` *mode*	そのファイルの許可属性が *mode* と一致すれば真。例えば `-perm 0644` のようにする。*mode* の前に「-」を付けると「*mode* で指定された属性が全て立っていれば真」の意に、また「+」を付けると「*mode* で指定された属性が 1 つでも立っていれば真」の意になる
`-size` n	n は (符号なしの) 数。そのファイルのサイズが n ブロック (1 ブロックは 512 バイト) なら真。n の代わりに「+n」と書くと「サイズが n ブロックより大」、「-n」と書くと「n ブロックより小」の意になる
`-type f`	そのファイルが普通のファイル (ディレクトリなどでない) なら真
`-type d`	そのファイルがディレクトリなら真
`-user` *username*	そのファイルの持ち主が *username* なら真
`-exec` *command* `\;`	指定されたコマンドを実行し、戻り値が 0 なら真。*command* の中に「`{}`」があれば、そのファイルの名前に置き換えられる。最後に「`\;`」が必要 (でないとどこまでが *command* でどこからが評価式の続きかわからなくなるので)。これは本来は単なる「;」なのだが、それがシェルにとっての特殊文字でもあるので、エスケープして「`\;`」と書かねばならない (「`';'`」でもよい)
`-print`	常に真。副作用としてそのファイルの名前を出力
`-print0`	常に真 (古い find にはない)。副作用としてそのファイルの名前を出力するが、`-print` が改行で区切るのに対しこちらはヌル文字 '`\0`' で区切る。`xargs -0` との組み合わせでよく使う (→3.1.4 節)
`-ls`	常に真。副作用としてそのファイルに関する諸情報を出力
`!` 式$_1$	式$_1$ が偽ならば真
式$_1$ `-o` 式$_2$	式$_1$ と式$_2$ の少なくとも一方が真ならば真
式$_1$ 式$_2$	式$_1$ と式$_2$ がともに真ならば真 (AND 結合⏏)。新しい find では明示的に「式$_1$ `-a` 式$_2$」とも書ける
`\(` 式$_1$ `\)`	式$_1$ が真ならば真。優先順位の変更に使う。「`(`」「`)`」がシェルにとっての特殊文字のため「`\`」でエスケープが必要 (「`'('`」「`')'`」でもよい)

の中では `-exec`, `-print`, `-print0`, `-ls` が該当) を「アクション」と呼ぶ。そして、評価式全体の中にアクションが 1 つもない場合は、評価式を真にした各ファイルに対し `-print` アクションが実行される (すなわち、そのファイルの名前が出力される)。従って例えば

```
$ find . -name '*.c'↵
```

とすると、評価式中にアクションがないので、「.c」で終わる名前のファイル名を全て表示することになり、すなわち「`find . -name '*.c' -print`↵」とするのと同じことになる。