１．Shell（シェル）

1-0．シェルの役割

シェルの機能は大きく分けると，以下の２つになる．

・コマンドインタープリタ

・プログラミング環境

1-1．　コマンドインタープリタとしてのシェル

シェルは入力されたコマンドを解釈し，実行のためにそれを（OSの）カーネル*に渡す．カーネルは渡されたコマンドをユーザプロセスとして実行し，その出力結果を（もし存在すれば）シェルに返す．シェルはカーネルからの出力結果をユーザに示す(図1)．このようにシェルは，ユーザから見るとカーネル(OS)の周りを取り巻いている貝殻（シェル）のように見えるため，この名で呼ばれている．

シェルには，コマンドの入力とコマンドからの出力扱うためのさまざまな機能が備わっており，これらの機能を使いこなすことによって，効率よくコンピュータを操作することが可能となる．

一般に UNIX系のシェルには　B-Shell（Born Shell）と C-Shell（Corn Shell)の２系統がある．従来，「B-Shell系 はコマンドインタープリタとしての機能は低いが，プログラミング環境ではスピードが速い」とされ，一方「C-Shell 系はコマンドインタープリタとしての機能は高いが，プログラミング環境では高機能な分だけスピードが遅い」とされてきた．しかし最近では，コマンドインタープリタとしての機能も高い B-Shell系の bash なども現れ，Linuxなどでは B-Shell が標準のシェルとなっている．B-Shell系とC-Shell系のどちらがコマンドインタープリタとして使いやすいかという問題は，最近では純粋に好みの問題であると言える．

＊カーネル： OSの中核．さまざまなサービスを提供する．

1-2．コマンドとコマンド入力

コマンドには内部コマンド（ビルトインコマンド：シェルそれ自身がもっているコマンド)と外部コマンド（トランジェントコマンド：実行形式のファイルとして存在するコマンド）がある． 内部コマンドには cd， source，setなどがある．他の大部分のコマンドは外部コマンドである（例： /bin/ls，/bin/cp，/bin/mv)．外部コマンドを実行する場合は，パスを指定して実行するか，予めコマンドのサーチパス（コマンドを探すためのパス）を設定しなければならない．

通常，コマンドは，カーソルが明滅しているコマンドプロンプト（C-Shell系の場合は**%**，B-Shell系の場合は**$**，管理者の場合は両シェルとも**#**）から入力され，Enterキーを押すことによってシェルに渡される． コマンドを入力する行をコマンドラインとも呼ぶ．コマンドの入力の仕方には以下のような方法がある．

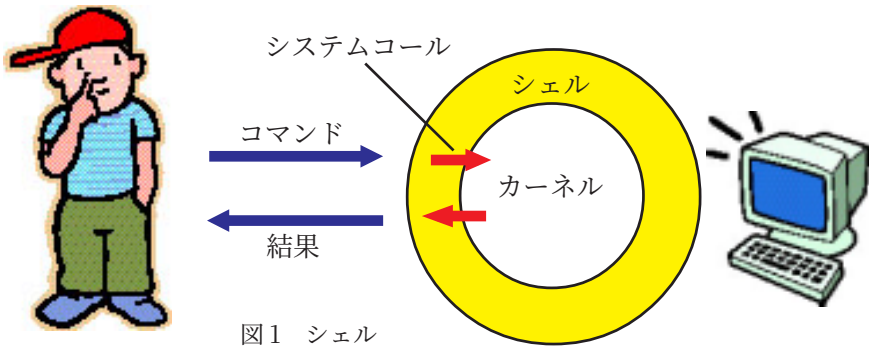

図１　シェル

０．コマンドを一つ入力し，実行する．（通常）
１．複数のコマンドを　；　で区切って入力すると，その順にコマンドを実行する．
２．コマンドを　&&　で区切ると，直前のコマンドが成功した場合のみ次のコマンドを実行する．
３．コマンドを　||　で区切ると，直前のコマンドが失敗した場合のみ次のコマンドを実行する．

演習）

上記のコマンドの入力例を試してみよう．

1. の例　**$ cd /tmp; pwd; ls; cd**
2. の例　**$ true && ls /tmp && ls**
3. の例　**$ false || ls /temp || ls**

注）true ：　必ず成功するコマンド．　　false ：　必ず失敗するコマンド．

次のコマンドの ls は実行されるか？

$ false || true && false || ls

コマンドが「成功する」，「失敗する」というのは具体的にどういうことだろう？true はなぜ何時も成功し，false は失敗するのだろう？

1-3．コマンドのバックグラウンド起動

通常コマンドはフォアグラウンドで実行されるが，コマンドの最後に**&**を付けると，そのコマンドはバックグラウンドで起動される．バックグラウンドとフォアグラウンドの違いはなんだろうか？

シェルは（外部）コマンドを渡されたとき，それを自分では処理することはない．シェルは fork というシステムコール*を使い，子プロセスと呼ばれる分身を作り出し，自分は親プロセスとなる．子プロセスは親プロセスの環境を全て受け継ぎ，親プロセスに代わってコマンドを実行する（exec システムコールによりそのコマンドプロセスに変化する）．

フォアグラウンドでコマンドを起動した場合，親プロセス（シェル）は子プロセスが終了するまで待つ（wait）ことになるが（図2-1），コマンドをバックグラウンド起動した場合，親プロセスは子プロセスを突き放し，子プロセスの終了を待たずそのまま動き続ける．子プロセスも親プロセスに頼らず，勝手にコマンドを実行する（図2-2）．つまり，フォアグラウンドで（外部）コマンドを実行した場合，コマンドが終

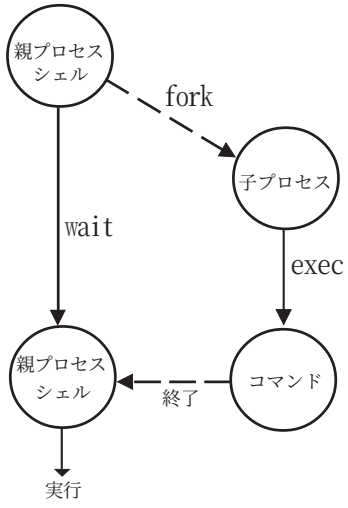

図2-1　フォアグラウンド起動

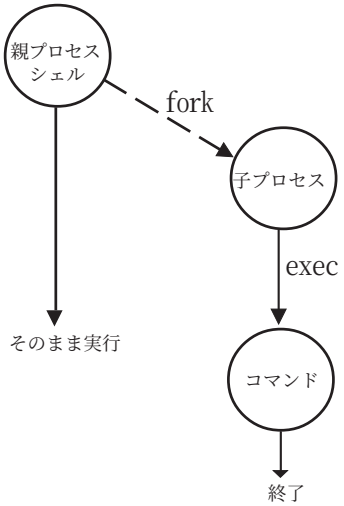

図2-2　バックグラウンド起動

了するまでコマンドプロンプトは返ってこないが, バックグラウンド起動の場合はコマンドプロンプトはすぐに返ってくる.

外部コマンドと違い, 内部コマンドの実行では, シェルは子プロセスを生成せず, シェル自身がコマンドを実行する.

*システムコール： カーネルの提供するサービスを呼び出す手段のこと. 魔法の呪文.

演習）

$ emacs test と **$ emacs test &** の違いを把握しよう.

なぜ, 内部コマンドは子プロセスで実行されないのだろう？ cd コマンドに**&**を付けてバックグラウンドで起動してみよう. 旨く動くだろうか？

1-3-1. バックグラウンド起動してはいけないコマンド

cat コマンドはファイルの内容を表示するコマンドだが, 引数を指定せずに単独で使用すると, キーボードから入力されたキーをそのまま画面に表示する.

$ cat と入力して試してやってみよう. **Ctrl+C(^C)**で終了.

ここで cat コマンドをバックグラウンド起動してみよう(**$ cat &**). 何が起こるだろうか？ 何も起こらない？

cat コマンドをバックグラウンド起動した後 jobs コマンドを入力してみよう.

演習）

```
$ cat &
[1] #####
$ jobs
[1]+  Stopped    cat
```

cat コマンドは停止していることが分る (jobsはそのシェル内で起動したプロセスの状態を表示する内部コマンド). なぜ cat コマンドは停止したのだろうか？

バックグラウンド起動すると 図2-2のように二つのプロセスが同時に動く. 親プロセスはそのまま動きつづけキーボードから次のコマンドが入力されるのを待つ. 子プロセスも cat コマンドに変化し, そのまま入力を待つ. ここでおかしな事に気づくだろうか？ キーボードからの入力は親プロセスに行く. 子プロセスの cat にはどこから入力がやってくるのだろう. 当然どこからも入力はやってこない. しかたなく cat コマンドはキーボードが自分に割り当てられるまで, 実行を一時停止するのである.

$ fg %1 （**1** は **jobs** で表示させた cat のシェル内のプロセス番号）

と入力してコマンドをフォアグラウンドに持ってくると, catコマンドはキーボードの入力を獲得してまた動き始める. このように, **シェル内でキーボード（標準入力）からデータを入力するコマンドは, バックグラウンド起動してはいけない**と言える**.**

1-3-2. フォアグラウンド・バックグラウンドの制御

コマンドを普通に起動してからバックグラウンド起動にすれば良かったと思うことは無いだろうか？ そのような場合は一旦 **Ctrl+Z(^Z)** でコマンドを一時停止し, **jobs** コマンドでプロセス番号を調べて, **bg** コマンドでバックグラウンド起動すればよい. 一時停止状態から再びフォアグラウンドに持ってくるには, 前節で紹介した通り **fg** コマンドを使う.

演習）

```
  $ emacs test
 ^Z
  $ jobs
[1] ......
[2]+ Stopped emas
  $ bg %2
```

1-4．シェルの設定

一般に，C-Shell 系のシェルは起動されると，ホームディレクトリにある ~/.chsrc を実行する．もしシェルがログインシェルの場合は ~/.cshrc に続いて ~/.login が実行される．ログインシェルは，システムにログインした時に一番最初に起動されるシェルである（手動でログインシェルを起動することも可能である）．従って ~/.login には端末の設定など，最初に一度だけ実行すればよいものを記述する．

B-Shell 系の場合は具体的なシェルによって微妙に設定が違う．例えば Linux の標準シェルの bash の場合は，ホームディレクトリにある ~/.bash_profile を実行する． ただし通常は ~/.bash_profile の先頭で ~/.bashrc を呼んでいるので，実質的には ~/.bashrc が最初に実行される．

Linux では場合によってはユーザの設定ファイルが実行される前に，いくつかのシステム設定ファイルが実行される場合があるが，それらを全て説明すると複雑になるのでここでは省略する（シェルのマニュアルを見よ）．一般ユーザが気にすべき設定ファイルは

C-Shell : ~/.cshrc，~/.login （リスト１，２）

B-Shell : ~/.bash_profile，~/.bashrc （ただし，bash の場合． リスト３，４）

である．シェルの設定は，コンピュータ操作の効率に直接影響する．自分の設定を他人任せにせず，必ず自分自身で確認することが大事である．

シェル変数

シェル変数はそのシェル内でのみで有効な変数である．プログラミング環境で使用したり，シェルの動きを設定するために使用する．ただし，C-Shell と B-Shell ではシェル変数の考え方が違うように思われる．

C-Shell ： シェル変数には習慣的に小文字を使用する（path，noclobber，prompt，history，ignoreeof など）．

B-Shell ： 環境変数と同様に習慣的に大文字で表す．自由に変数を設定可能．export コマンドを使用することによって，環境変数にすることが可能．

環境変数

シェル変数はシェルだけが使用するが，環境変数はシステムや色々なアプリケーションが（シェル外から）使用する変数である．習慣的に大文字で表わす．環境変数を表示するには env コマンドを使用する．set コマンドはシェル変数と環境変数の両方を表示する．

演習）

リスト 1-4 に現れるシェル変数，環境変数の意味を調べてみよう．

リスト１　~/.login の例

```
# @(#).login
#######################################################################
#
#         .login file
#
#######################################################################
#
if ($TERM == "kterm" ) then
        stty dec erase '^H' susp '^Z' pass8 cs8
        setenv LANG ja_JP.ujis
        setenv JLESSCHARSET japanese
        setenv GDK_CONV $HOME/.gtk
        setenv EDITOR emacs
endif
#
if ($TERM == "vt100" || $TERM == "vt220" || $TERM == "kon") then
        stty dec erase '^H' susp '^Z' pass8 cs8
        setenv LANG ja_JP.ujis
        setenv JLESSCHARSET japanese
        setenv GDK_CONV $HOME/.gtk
        setenv EDITOR "emacs -nw"
endif
#
if ($TERM == "linux" || $TERM == "xterm") then
        stty dec erase '^H' susp '^Z' pass8 cs8 -istrip
        setenv LANG C
        setenv EDITOR "emacs -nw"
endif
#
#
#  環境変数の設定
setenv NNTPSERVER newshost.edu.tuis.ac.jp
setenv POP3SERVER popsvr.edu.tuis.ac.jp
setenv XMODIFIERS "@im=kinput2"
setenv PAGER /usr/bin/jless
#
```

リスト２　~/.cshrc の例

```
# @(#).cshrc
#####################################################################
#
#         .cshrc file
#
#####################################################################
#
#  サーチパスと CD パスの設定
set lpath = ( /usr/java/j2sdk1.4.0/bin/ ~/bin)
set path = (/usr/bin /usr/local/bin /usr/X11R6/bin /bin /sbin /usr/sbin $lpath .)
set cdpath = (.. ~)
#
#  シェルの設定
set noclobber
limit coredumpsize  0
umask 027
#
#  コマンドの別名１
alias cd            'cd \!*;echo $cwd'
alias cp            'cp -i'
alias mv            'mv -i'
alias rm            'rm -i'
#
#
#  非インターラクティブシェルの場合はここで終了
if ($?USER == 0 || $?prompt == 0) exit
#
#
#  シェル変数の設定
set myname=`whoami`
set history=500
set savehist=500
set ignoreeof
set notify
set filec
set mail=/var/spool/mail/$USER
set dspkanji=euc
set autologout=0
#
#
#  コマンドプロンプトの設定
if ($myname == "root") then
        set prompt="#"
        set cdpath =
else
        set prompt="%"
endif
set prompt="`hostname`[${myname}]\!:%~${prompt} "
#
#
echo ''
echo ''
#
#
#  コマンドの別名２
alias hist          'history \!* | tail -20'
alias histx         'history| grep \!* | tail -30'
alias cls           clear
alias ls            'ls -F'
alias nkf           'nkf -e'
alias bd            popd
alias pd            pushd
alias rl            rlogin
alias run           source
alias su            'su -'
#alias dir           'ls -xAF'
#alias help          man
#alias bye           logout
#alias ds            dirs
#
```

リスト３　~/.bash_profile の例

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs
stty dec erase '^H'
if [ "$TERM" = "linux" -o "$TERM" = "xterm" ]; then
    export LANG=C
elif [ "$TERM" = "kon" -o "$TERM" = "kterm" ]; then
    export LANG=ja_JP.eucJP
    stty pass8 cs8
    alias man=jman
elif [ "$TERM" = "vt100" -o "$TERM" = "vt220" ]; then
    export LANG=ja_JP.eucJP
    stty pass8 cs8
    alias man=jman
fi


# addpath $HOME/bin
BASH_ENV=$HOME/.bashrc
USERNAME=""

# 環境変数へ
export USERNAME BASH_ENV PATH LESSOPEN
```

リスト４　~/.bashrc の例

```
# .bashrc
#####################################################################
#
#         .bashrc file
#
#####################################################################
#
# User specific aliases and functions

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi


#
#  サーチパスの設定
LPATH=/usr/java/j2sdk1.4.2_04/bin/:~/bin
PATH=/usr/bin:/bin:/usr/X11R6/bin:/sbin:/usr/sbin:$LPATH

#
#  シェルの設定
ulimit -c 0 -s unlimited
set -o noclobber
umask 027
#stty -ixon

#
#  コマンドの別名１
alias cp='cp -i'
alias mv='mv -i'
alias rm='rm -i'

#
#
#  シェル変数の設定
IGNOREEOF=100
HISTSIZE=500
HISTFILESIZE=500

#
#  コマンドプロンプトの設定
if [ "$USER" = "root" ]; then
    PS1="#"
else
    PS1="$"
fi
PS1=$"[\\u@\\h \\W]:\!$PS1 "

#
#
#
echo

#
#  コマンドの別名２
alias ls='ls -F --color=auto'
alias eng='LANG=C LANGUAGE=C LC_ALL=C'
alias hist='history | tail -20'
alias cls=clear
alias nkf='nkf -e'
alias bd=popd
alias pd=pushd
alias su='su -'

#alias help         man
#alias bye          logout
#alias ds           dirs
#
```

1-5．コマンドラインの補完とコマンドヒストリ

最新のシェルでは例外なく，コマンドラインでの補完機能とコマンドのヒストリ機能を備えている．コマンドラインの補完機能は，コマンド入力の途中で ESC キー（2回押す）かまたは TAB キーを押すことによって，コマンド名またはファイル名を補完する機能である． 例えば，カレントディレクトリ（現在のディレクトリ）に name，morning，hello のファイルがある場合， **$ ls n** を入力し，補完キー（ESC 2回 or TAB）を押すと， 自動的に **$ ls name** と補完される．

ヒストリ機能は，入力したコマンドを覚えている機能である．通常コマンドプロンプトに数字が現れる場合，それはヒストリ番号を表わしている場合が多い(シェル変数の prompt や PS1 に **\!** がある場合，**\!** がヒストリ番号を表わす)．ヒストリ番号は入力したコマンドに付けられる通し番号であり，コマンドを入力するたびに +1 される．

シェルがヒストリ機能を備えている場合，上下のカーソル移動キーで以前入力したコマンドをコマンドライン上に表示することができる．コマンドライン上に表示されたコマンドは左右のカーソル移動キーを使って修正することが可能なので，コマンドの入力を効率よく行える．また，カーソルキーによるコマンドの修正の他にも以下のようなヒストリ関連のコマンドを使用することが可能である．

$ history	以前入力されたコマンドで記憶しているものを全て表示する．
$!!	直前に入力したコマンドをもう一度実行する．
$!a	a で始まる最も最近入力したコマンドを実行する．
$!220	ヒストリ番号 220 番のコマンドをもう一度実行する．
$ ^abc^efg	直前に実行したコマンドの abc の部分を efg に替えて実行する．

これら機能は，十分考えて使用しないと，予期しないコマンドを実行してしまう可能性がある． 特に root（管理者）の場合，間違ったコマンドの実行は致命傷にもなりかねないので十分注意する必要がある．

２． テキスト処理

2-1．標準入出力

UNIX（Linux）ではデータの入出力は全てファイル（と呼ばれるもの）を通して行われるようになっている．例えば，あるファイルに文字を書き込むと，そのファイルは実は裏でディスプレイに繋がっており，書き込まれた文字がディスプレイに表示されると言った具合である．

UNIX（Linux）では，ユーザが何もしなくとも最初から**「標準入力」(0)，「標準出力」(1)，「標準エラー出力」(2)**というファイルがオープンされており，それぞれデフォルトでキーボード，ディスプレイ，ディスプレイに繋がっている．

課題) 「標準出力」と「標準エラー出力」はどちらもデフォルトでディスプレイに繋がっている．それでは，「標準出力」と「標準エラー出力」の違いは何であろうか？ 本当に２つもいるのか？ どちらか１つでも十分ではないだろうか？ ここではすぐに答えはでないかもしれない．この章が終わるまでに考えておこう．

2-2. リダイレクション

「標準入力」,「標準出力」,「標準エラー出力」の接続先を簡単にキーボードやディスプレイ以外のもの（主に通常のファイル）に切り替えることができる.

C-Shell

<	「標準入力」をファイルに切り替える.	例） cat < ~/.cshrc
>	「標準出力」をファイルに切り替える. ファイルの上書き.	例） ls -l > list
>>	「標準出力」をファイルに切り替える. ファイルへ追加.	例） ls -l >> list
>&	「標準出力」と「標準エラー出力」を同じファイルに切り替える. ファイルの上書き.	
>>&	「標準出力」と「標準エラー出力」を同じファイルに切り替える. ファイルへ追加.	
>!	「標準出力」をファイルへ強制上書き	
>&!	「標準出力」と「標準エラー出力」をファイルへ強制上書き	

注） ファイル保護のため, シェルがリダイレクションの上書き禁止モードになっている場合, > や >& でファイルを上書きしようとするとエラーとなる. 強制的に上書きするにはそれぞれ >! や >&! を使用する.

C-Shell 系では単独で「標準エラー出力」をリダイレクトすることができない.「標準出力」と「標準エラー出力」を別々のファイルに分けるには

% （コマンド > out）>& err

のようにする. out に「標準出力」が, err に「標準エラー出力」がそれぞれリダイレクトされる. （ ）で括った部分はシェルによって, 一個のコマンドとみなされる.

B-Shell

<	「標準入力」をファイルに切り替える.	例） cat < ~/.cshrc
>	「標準出力」をファイルに切り替える. ファイルの上書き.	例） ls -l > list
>>	「標準出力」をファイルに切り替える. ファイルへ追加.	例） ls -l >> list
&>	「標準出力」と「標準エラー出力」を同じファイルに切り替える. ファイルの上書き.	
>ST 2>ERR	「標準出力」をファイル ST へ.「標準エラー出力」をファイル ERR へ.	
>FL 2>&1	「標準出力」と「標準エラー出力」を同じファイル FL に切り替える. ファイルの上書き.	
>|	「標準出力」をファイルへ強制上書き	

2-3. パイプ

ＵＮＩＸのコマンドの考え方は, 小さな部品（コマンド）をいくつも組み合わせて一つの仕事をさせようと言うものである（統合ソフトとは対照的な考え方だ）. そのためＵＮＩＸにはパイプと呼ばれる機能が用意されている. コマンドがパイプ（|）で繋がれた場合, 前のコマンドの標準出力（標準エラー出力ではないことに注意！）は次のコマンドの標準入力に渡される.

$ ls -l | less この例では ls -l の結果（標準出力）は less に渡され1ページづつ表示される. 次のような例もある.

$ ls -l |grep ^d |awk -F”“ ‘{print $9}’ この例では ls -l の標準出力は grep, awk に渡され（grep, awk についてはマニュアル参照）, ディレクト名だけが表示される.

注意しなければならないことは, 標準出力はパイプに渡されるが, 標準エラー出力はパイプには渡されないということである. これは何を意味するのだろうか？

回答編

C-Shell のシェル変数

path ： コマンドの検索パス．これ以外の場所にあるコマンドはパスを指定しないと起動することができない．
cdpath ： cd する時の候補パス．
noclobber ： リダイレクションの出力先のファイルが既に存在する場合はエラーにする．
limit coredumpsize ： core ダンプのサイズを指定する（通常は 0 にする）．
mask : ファイルやディレクトリを作るときのパーミッションのマスクを設定する．
history ： コマンドヒストリ（過去のコマンドの記憶）の数．
savehistory ： シェルから抜けるとき ~/.history に保存するヒストリの数（通常 history と同じ）．
ignoreeof ： データの終わり（EOF, 通常 ^D)でシェルが終了しないように無視する．
notify ： バックグラウンド起動した子プロセスが終了した場合，ユーザに通知する．
filec: ファイル名やユーザ名の補完を行う．最近のシェルはこれを設定しなくともデフォルトで補完する．
mail ： メールスプールファイルの指定．
dspkanji : 取り扱う漢字コード．
autologout ： 入力がない場合の自動ログアウトまでの時間．0を指定するとログアウトしない．

B-Shell のシェル変数

（省略）

必ず成功するコマンドと失敗するコマンド

コマンドが成功すると言うことと，そのコマンドが何を行ったかにはまったく関係ない．コマンドの成功とは，そのコマンドが終了ステータス 0で終了したと言うことである．C言語の場合は exit(0) で終了した場合，JAVA の場合は System.exit(0) で終了した場合にそのコマンドは成功したことになる．ちなみに失敗の場合は ，0 以外の終了ステータスで終した場合である．

必ず成功するコマンドのソース（C）

```
#include <stdio.h>
main(){
  exit(0);
}
```

標準出力と標準エラー出力

標準出力はパイプに渡されるが，標準エラー出力はパイプには渡されない．これは標準出力が次のコマンドのために出力であるのに対して，標準エラー出力はユーザ（コマンド入力者）やオペレータ（もしくはそれに相当するプログラム）に対する出力である．従ってＵＮＩＸのプログラミングでは，入力を促すメッセージや，ユーザに注意を喚起するメッセージは標準出力に書いてはいけない．標準出力にはパイプで繋がれた場合に次のコマンドに渡すべきデータのみを記述するべきである．