Bourne Shell 自習テキスト

木村 孝道
平林 浩一 監修

1993 年 6 月 21 日

目 次

pLaTeX2e 化，および２段組とするためにかなり手を加えました．by 松田七美男 [May/06/2004]

1　シェルとは

シェル (shell) とは unix のコマンドインタプリタで、ユーザ端末から入力された文字列を解釈し、その指示に従って仕事をするプログラムです。しかし、シェルは決して特殊プログラムではありません。シェルも他のツールと同様に unix 上の１つのコマンドに過ぎません。シェルが他の多くのプログラムと違う点は自分自身がある特定の仕事をするのではなく「他のコマンド類のまとめ役」として機能することです。

なぜコマンドインタプリタが「殻」を意味するシェル (shell) なのでしょうか。ユーザから見るとオペレーティングシステムの核 (kernel) を貝殻のように包んでいることに由来しているようです。Rod Maris, Marc H. Meyer 著「The UNIX Shell Programing Language」には名前の由来について次のような記述があります。

> It is called the shell, beacuse, like the shell of a nut or an egg, it is the part that we see from the outside. The inside part is called the kernel. The shell takes to us and to the kernel.
>
> それは、胡桃や卵の殻のように、外から眺める部分なので、シェル（殻:shell）と呼ばれている。内部は核（カーネル:kernel）と呼ばれる。シェルはカーネルと我々との仲立ちをする。

kernel とはオペレーティングシステムの中枢で、プロセスやメモリ、ファイル等の管理を行う部分です。unix 上で何か仕事をする時には必ずお世話にならなければならないものですが、ユーザ側から kernel に働きかける手段はシステムコールのみです。プログラムを書くのでしたらシステムコールを操れますが、対話形式ではシェルを通す以外触れる方法がありません。シェルを通して備え付けのコマンドなり、自分の作ったプログラムを起動して初めてユーザは kernel へなんらかの指示を出すことができるわけです。プログラムを書かないユーザにとってはシェルこそがオペレーティングシステム、あるいはコンピュータそのものに見えますから、シェルを使いこなすことがそのままコンピュータを使いこなすことになるわけです。

unix の世界標準、POSIX のドラフトでは唯一のシェルとされる「Bシェル (Bourne Shell)」を minix 上で体験しながら unix の心髄に触れてみましょう。なお、文中で minix と表現している部分はそのまま unix と読み換えても不都合は起こらないように配慮したつもりです。

2　シェルの機能

シェルのもっとも簡単に実感できる機能はユーザとの対話形式で入力された文字列（コマンド）を解析し、指示された仕事を行うものでしょう。「対話形式」だけを考えるとユーザから入力された文字列を修正する機能や、入力を記憶しておき必要に応じて呼び出す履歴機能などに注目されます。しかし、対話形式での操作はシェルの提供する機能の一部分に過ぎません。シェル本来の機能は「他のコマンド類のまとめ役」にあります。この機能を上手に組み合せるとわずかな手間でかなり複雑な仕事をこなすことができます。ひとつの仕事を能率良く処理できるよう、独立した単純なコマンド類をまとめるため、シェルは次のような機能を持っています。

1. プログラムの実行
2. ファイル名の置換
3. 入出力の切り換え
4. パイプ機能
5. ユーザの環境設定
6. インタプリタ型のプログラミング言語

2.1　プログラムの実行

シェルはユーザが端末から入力したコマンドを実行する機能を持っています。コマンドの入力が完了するとそれを解析し、何をすべきかを決定します。シェルに対して入力された文字列はコマンドラインと呼ばれ、一般的に次の様な形をしています。

$ コマンド名　[引数 *1*　引数 *2*　$\cdots$　引数 *n*]

行頭の **$** はシェルがユーザからのコマンドを受け付ける状態にあることを示すプロンプトで、ユーザがタイプするものではありません。引数を必要としないコマンドもたくさんありますから省略することもできます。コマンド名や各引数間は一般にホワイトスペース文字と呼ばれる「スペース」あるいは「水平タブ」で区切ります。コマンドラインでホワイトスペース文字が重複した場合は区切りとして認識されるだけで、重複した個数については無視されます。いま、

```
$ echo I drink coffee
```

と入力したとします。するとシェルはこのコマンドラインを走査し、実行すべきコマンド名として行頭から最初の区切り文字までの文字列 **echo** を取り出します。続いて次の区切り文字までの文字列 **I** を **echo** への最初の引数として取り出します。同様な操作をコマンドラインの最後まで繰り返し **drink** と **coffee** を第２、第３引数として取り出します。コマンドラインの解析が終了するとシェルはここ

で取り出した３つの引数を echo コマンドに渡しその終了を待ちます。echo コマンドが終了するとプロンプトを表示してユーザから次のコマンドラインの入力を待ちます。それではもう一つ、

```
$ echo Do you       enjoy            minix?
```

を試してみてください。echo の出力は次のようになったはずです。

```
Do you enjoy minix?
```

これは先に説明したように区切り文字が重複していても区切りとして認識するだけ で重複個数は無視されるために起こります。echo コマンドは受け取った引数を１個のスペースで分けてユーザの端末に表示するだけなのです。

この２つの例からコマンドラインを解釈するのは echo などのシェルによって起動されるコマンドではなく、シェル自身だということを理解してください。シェルにより起動された echo はシェルから渡された引数を見て仕事をしているだけで、実際にコマンドラインとして入力された文字列がどのようなものか、あるいはどのような指示がされたかなどは知らないのです。

ちょっとせっかちな話しなのですが、後の例ではシェルはコマンドと引数の分離、実行だけではなく、もう少し複雑な仕事をしています。それは次項で。

2.2 ファイル名の置換（展開）

シェルはコマンドラインを解析し、実行すべきコマンドや引数を決定する前に特殊文字 *, ?, [ ] を見つけるとその特殊文字部分をファイル名に置き換えます。いま、ユーザのカレントディレクトリに次の様なファイルがあると仮定します。

```
$ ls
beer
carrot
ell
rabbit
```

ここで echo コマンドを使ってファイル名の置換を試してみましょう。次のようにタイプしてください。

```
$ echo *
beer carrot ell rabbit
```

シェルは与えられたコマンドラインの解析を始めます。そして特殊文字 * を見つけると、カレントディレクトリ内にあるすべてのファイル名で * 部分置き換えます。そのあとにシェルは起動すべきコマンド名と渡すべき引数を決定します。

echo は特殊文字 * を解釈することも、存在していいることも分かりません。この場合、echo は４個の引数がシェルから与えられたことを知っているだけなのです。このようにシェルは grep や ed などに比べると制限されているものの、次のような正規表現を解釈することができます。これらの文字をメタ文字と呼びます。なお、シェルによるメタ文字の展開はファイル名のみが対象となります。

特殊文字	意　味
*	０個以上の任意文字と一致
?	任意の１文字と一致
[ ]	[と] で囲まれた文字のいずれかと一致

さて、ここで前項で例とした

```
$ echo Do you       enjoy            minix?
Do you enjoy minix?
```

を考えてみてください。このコマンドラインにはメタ文字 ? があるのに気が付いたでしょうか。ところが echo の出力を見るとシェルにより置換が行なわれずにそのまま表示されています。これはシェルが手抜きをしたわけではありません。シェルはコマンドラインの解析で ? を見つけています。そしてカレントディレクトリにあるファイル名との置換を試みています。しかし、置換条件を満たすファイルが見つからないので置換に失敗してし、入力された文字列をそのままを引数として echo に渡してしまうので minix? がそのまま表示されたのです。

しかし、場合によってはシェルによる置換が思いもよらない弊害を引き起こすことがあります。このためにシェルによってコマンドラインにあるメタ文字の解釈を禁止する方法が用意されています。［「3.3 引用符」(p.9) 参照］

シェルがコマンドラインを解析する手順をまとめると概ね次のようになります。

1. コマンドラインから特殊文字 *, ?, [,] を探す
2. メタ文字が見つかればファイル名で置換を試みる
3. コマンドラインから起動すべきコマンドと引数を取り出しコマンドを起動する

2.3 入出力の切り換え (I/O redirection)

シェルは起動したコマンドの入出力先を切り換える機能（Ｉ/Ｏリダイレクト）を持っています。シェルはコマンドラインを解釈し、リダイレクトを表す特殊な文字 >、<、>> が見つかるとそれに従った処理を行います。仮に

```
$ echo It will rain tomorrow >memo
```

それに続く語で指定されたファイルに（この場合は memo）出力先を変更します。この例ではカレントディレクトリに

memo というファイルを作り、そこに echo の出力を書き出します。結果としてファイル ./memo に "It will rain tomorrow" が書き込まれることになります。もし、この時にファイル ./memo がなければ新たに作られますし、既にある時には上書きされて古い内容は失われます。

ここで大切なことは、シェルはコマンドラインで指定されたコマンドの 実行を開始する前 にその標準出力を > に続いて指定されたファイルに切り換えていることです。起動されたコマンドはシェルにより標準出力が切り換えられているのことを知らずに、標準出力に結果を出力しているだけです。Ｉ/Ｏリダイレクトを表す文字とその意味は次のようなものです。

特殊文字	意　　味
[*n*]< *file*	ディスクリプタ *n* (省略時 0) で *file* を読み込み用にオープン
[*n*]> *file*	ディスクリプタ *n* (省略時 1) で *file* を書き込み用にオープン
[*n*]>> *file*	ディスクリプタ *n* (省略時 1) で *file* を追加書き込み用オープン
<< '*eof*'	標準入力を次行から *eof* の直前行までとする (here document)
n>&*m*	ディスクリプタ *n* の出力をディスクリプタ *m* に変更
n<&*m*	ディスクリプタ *m* の入力をディスクリプタ *n* に変更
[*n*]<&-	入力ディスクリプタ *n* (省略時 0) をクローズ
[*n*]>&-	出力ディスクリプタ *n* (省略時 1) をクローズ

[ ] 内は省略可。ディスクリプタはオープンしたファイル番号で 0〜9 の数字で表す

2.3.1 ヒアドキュメント (here document)

<< はヒアドキュメントと呼ばれる興味深いメカニズムを提供してくれます。通常の仕事でコマンドとそれが参照するデータが対になることがしばしば起こります。例えば電話番号簿などでは検索プログラムとデータファイルの２つが必要になります。ヒアドキュメント機能を利用してシェルスクリプトを書けばこれらのファイルを１つにまとめることができ、保守性やディスクの使用効率が良くなります。[「6.3 ハノイの塔」(p.20) 参照]

```
grep $1 <<'END-OF-FILE'
JAL:(03)5489-1111
JAS:(03)3438-1155
ANA:(03)3272-1212
   ...
KLM:(03)3216-0771
END-OF-FILE
```

この例を理解するにはシェルスクリプトの基礎知識が必要になりますが、簡単にメカニズムを説明します。grep は文字列検索プログラムで、$1 はシェルによりコマンドラインで指定されたキーワードと置き換えられます。シェルは<< を見つけると、それに続く文字列 *END-OF-FILE* を EOF マークとして覚えます。そして、その直後の行から次の EOF マークの直前の行までをコマンド grep への入力に結合します。ヒアドキュメントの EOF マークは１行単位で評価されますから EOF マークには途中のデータ行に出現しない文字列を選ばなければなりません。

2.3.2 エラー出力を標準出力にマージする

標準出力の記録とエラー履歴を一緒に取るためにコマンドラインで次のような記述をすると思います。

```
$ command >foo 2>&1
```

この動作を追いかけてみます。リダイレクト機能表でも分かるように >& の働きは、左側に書かれたファイルディスクリプタの出力を右側に書かれたファイルディスクリプタに変更することです。シェルは標準入力・標準出力・エラー出力の３つのファイルをオープンしています。そして、それらをファイルディスクリプタ０、１、２として管理しています。ですから 2>&1 はファイルディスクリプタ２を１に、つまりエラー出力を標準出力に変更することになります。さらに、>foo として標準出力が foo にリダイレクトされていますから、foo には標準出力とエラー出力がマージされたものが書き出されることになります。

リダイレクトのメカニズムを理解する上で大切なことは、コマンドラインで記述されたものが 右から左の順 に実行されるということです。エラー出力を標準出力にマージするこの例では、まず最初に 2>&1 が実行されてから >foo が実行されます。もし、コマンドラインで

```
$ command 2>&1 >foo
```

と書いたならば違った結果になってしまいます。どうなるかはご自分でお試しください。

2.4 パイプ機能

シェルはリダイレクト文字や正規表現をコマンドラインから解釈すると同様にパイプ記号 | も識別します。シェルはパイプ記号 | をコマンドラインに見つけるとその前

にあるコマンドの標準出力をその後ろにあるコマンドの標準入力に結合させます。そしてシェルは両方のコマンドを同時に実行させます。次のコマンドラインを例にして考えてみましょう。

```
$ who | wc -l
```

まず、シェルはコマンドラインを解析して who と wc の間にあるパイプ記号| を見つけます。次にシェルは最初のコマンド who の標準出力をそれに続くコマンド wc の標準入力と結合させて２つのコマンドの実行を開始します。コマンド who はログインしているユーザのリストを標準出力に書き出します。who の標準出力は wc の標準入力につながれていますから端末に who の出力は表示されません。who と同時に起動された wc はファイル名の指定がないので標準入力からの行数を数えますので、who の処理結果の行数を数えることになります。リダイレクトの項目でも触れたので繰り返しになりますが、シェルにより起動された各々のコマンドは自分の標準入出力が何に割り当てられているかは知りません。

2.5 コマンドの区切り文字

シェルのコマンドラインではセミコロン ; で区切ることにより複数のコマンドを書くことができます。

2.6 コマンドのグルーピング

シェルは幾つかのコマンドをコマンド群をまとめ、あたかも１つのコマンドのように実行させることができ、これをコマンドのグルーピングと言います。グルーピングを行なうにはまとめたいコマンド類を小括弧で囲んで (···) とするか、あるいは中括弧を使って { ··· ; } のようにします。この２つはいずれもコマンドのグルーピングを行なう点は同じですが、(···) はサブシェル（Page: 8 参照）で実行され、{ ··· ; } はカレントシェルで実行されます。この違いを次の例で確認してください。

```
$ pwd
/usr/try
$ (cd /bin; ls -C)
$ pwd
/usr/try
$ { cd /bin; ls -C; }
$ pwd
/bin
```

(···) はサブシェルで実行されるために終了後のカレントディレクトリは変わりません。一方、{ ··· ; } はカレントシェルで実行されますので終了後のカレントディレクトリが変わってしまうのが確認できたと思います。

中括弧 { ··· ; } を使ってグルーピングを１行に書くときに、{ の直後と、 } の直前には１個のスペースが必要です。さらに中括弧でグルーピングされたコマンドの末尾にはコマンドの区切りを表すセミコロン ; がなければいけません。これは中括弧がシェルの予約語であるための制限です。［「4.1 構文規則」(p.12) 参照］

2.7 バックグランド処理

minix はマルチタスク OS ですから複数のプログラムを同時に走らせることができます。しかし、特に指定をしない限りシェルは入力されたコマンドの実行終了を待って次のコマンドを受け付けます。プリンタへの出力やテープの巻き戻しなどは数分、あるいはもっと時間がかかるかもしれません。これを待っていては何のためのマルチタスクなのか分かりませんので、シェルにコマンドの終了を待たずに次のコマンドの受付ができるようにバックグランドで実行するように指示を出すことができます。

入力したコマンドをバックグランドで実行させるためにはコマンドの末尾にアンパーサンド & を付けます。こうするとシェルはプログラムをバックグランドで実行し、そのプロセス ID を表示し、すぐにプロンプトを表示してユーザからの次の入力を受け付け状態になります。ここで表示されるプロセス ID はバックグランドで実行されているコマンドを識別する唯一のものです。

バックグランドで実行したプログラムは DEL キーによって発生する端末割り込みで中断させることができません。途中で停止させるにはここで表示されたプロセス ID（ps コマンドで調べることもできます）を使って SIGNAL9 を送ります。

2.8 ユーザの環境設定

シェルはユーザの希望する環境を設定できるいくつかの環境変数と呼ばれるものを持っています。これらはユーザのホームディレクトリやコマンド入力を促すためのプロンプト文字列、ユーザが実行させたいコマンドを探すためのディレクトリ・リストなどを記録しています。主な環境変数として次のようなものがあります。

2.8.1 HOME

ユーザがシステムにログインした時に自動的に決定されるユーザの家 (HOME) とされるディレクトリです。この環境変数 **HOME** はユーザのホームディレクトリを識別

するためにプログラムから参照することができます。例えば、引数なしで cd コマンドを実行した時にこの **HOME** が参照され自分の家に迷わず帰ることができます。もちろんこの **HOME** はユーザが好きなものに書き換えることができます。しかし、不用意に変更してしまうと **HOME** を参照するコマンドの動作に影響があるばかりか、他のユーザに迷惑をかけることになりますから、注意してください。

HOME はログインした時に/etc/passwd 中のホームディレクトリ・フィールドに従って定義されます。

2.8.2 Prompt String(PS1,PS2)

シェルがユーザにコマンドラインの受け付け状態にあることを示すために表示する文字列は変数 **PS1** に格納されます。ログイン後に表示されている$ がそうです。コマンド行が複数にまたがり、コマンド入力状態継続していることを表す二次的なプロンプト文字列は **PS2** が示しており、通常は > が格納されています。この変数はユーザが自由に書き換えて構いません。

2.8.3 PATH

シェルはユーザからコマンドラインを解析して起動するプログラムを決定したならばそのコマンドを環境変数 **PATH** が示すディレクトリ・リストから探します。この **PATH** はログイン時に自動的に設定されます。**PATH** を始めとする環境変数は echo コマンドで次のようにすれば、その内容を見ることができます。

```
$ echo $PATH
/bin:/usr/bin::
```

環境変数 **PATH** の前にドル記号 $ を置いてやるとシェルはその部分を環境変数 **PATH** に格納されている内容で置換してから echo に引数として与えます。この場合は ”/bin:/usr/bin::” がコマンド探索用のディレクトリ・リストになります。

PATH の示すディレクトリ・リストはそれぞれのディレクトリをコロン**:** で区切って表されます。コマンドの実行が指示されたならばシェルは”/bin” ⇒ ”/usr/bin” ⇒ ” ” の順にディレクトリから指定されたコマンドを探します。このディレクトリ・リストの最初の 2 つは文字の通りの場所です。 3 番目の ” ” はカレントディレクトリ ”./” を意味しています。今まで頻繁に使ってきた echo コマンドは/usr/bin にありますから、コマンドが入力されるたびにシェルはこれらのディレクトリ・リストを参照して探していたわけです。

コマンドラインでコマンド名にパスを含めた指定をするとシェルは **PATH** の示すディレクトリ・リストを無視して直接指定されたコマンドをだけを探します。例えば、

```
$ /bin/date
```

とすると、シェルは **PATH** の内容を無視して/bin/date を実行させます。このような絶対パスだけでなく、カレントディレクトリからの相対パスを指定しても同様に環境変数 **PATH** の内容は無視されます。

2.8.4 TERM

環境変数 **TERM** はユーザの端末属性を記憶しています。どのような端末でも 10 進数の 65 というコードを受け取ると ”A” という文字を表示します。しかし、1 というコードを受け取ると端末によっては強調モードになるかも知れませんし、別の端末では反転モードになるかも知れず、動作の保証がありません。スクリーンエディタなどの端末を制御するプログラムは適切な制御コードを端末に送らないと正しい結果が得られません。世間には無数の端末がありますからそれぞれに合わせたプログラムを一個一個作るのは途方もない時間と労力が必要になり不経済です。これを避けるために、minix ではたくさんある端末の制御コードをデータベース化して/etc/termcap というファイルに持っています。この中から自分の使っている端末のエントリを環境変数 **TERM** に格納しておきます。

termcap については本テキストで触れていません。興味を持たれた方は次章以降でシェルプログラミング教材としている Cabinet から参考文献を探すことができますので読まれてはいかがでしょう。［「7.7 Cabinet の文献リスト」(p.30］

2.8.5 .profile によるログイン環境の設定

ここで触れた環境変数を始めとする色々な設定は **.profile** というファイルに記述してホームディレクトリに置けばログイン時に自動的に設定されます。この **.profile** はログインと同時に一度だけ実行されるシェルスクリプトですから環境設定以外にもログインした時に実行させたいものを書いておくことができます。

2.9 インタプリタ型のプログラミング言語

シェルはその中にインタプリタ型のプログラミング言語を持っています。このプログラミング言語は多くのプログラミング言語同様に条件判断やループ処理などの機能を持っており、複数のコマンドのまとめ役としてシェル本来の力を出してきます。このプログラミング言語の理解を深

め、minix の一番おいしい部分を楽しむことがこのテキストの最大の目的です。

3 シェルプログラミングの基礎

今まで見てきたように、シェルには１つのコマンドを実行させたり、また複数のコマンドを組み合わせて実行する機能ばかりか、あるコマンドの行った仕事結果を見てさらに次のアクションを起こさせたり、他の処理に分岐させたりすることができます。この機能を実現するために使われるのがシェル・プログラミング言語です。

端末に向って仕事をする時間が長いせいかもしれませんが、シェルというと一般的にヒストリ機能等、対話型の操作性に目を奪われるかも知れません。しかし、対話性を重視する環境はともすれば人間が機械の忠実な子守役を強いられる可能性があります。１つのコマンドの結果をみてからそのつど人間が次の仕事を機械に指示するのは本末転倒で何ともバカバカしいかぎりです。シェルの持つプログラミング機能はこの人間が行うべき仕事、すなわち１つのコマンドの結果を見て次に起こすアクションをあらかじめシェルに教えておき、実行時にそれらを代行させることにあります。

シェル・プログラミング言語（シェル言語）そのものはいたって単純なものですが、これを使いこなすには minix 上の各コマンドの使い方を理解しなければなりません。シェルは各々のコマンドを起動し、その結果を文字列として引用しますが、実際にデータを処理するのは minix 上の小さなコマンドです。データを処理しようと考えた時には数あるコマンドの中からどれを、どのようなオプションで、どういう組み合わせで使えば目的を達することができるかを考えなければなりません。

3.1 シェルスクリプトと実行方法

人間の仕事をシェルに代行させるための手順を書いたテキストファイルのことをシェルスクリプトといます。シエルスクリプトに記述できる内容はコマンドラインで受け付けられるものならば何でも書くことができます。シェルにはパイプに代表されるような複数のコマンドを組み合せる機能や、あるコマンドの実行結果を見てそれを次のコマンドに引用するためのロジックが組み込まれています。

シェルスクリプトは純粋なテキストファイルですからテキストエディタで書き起こしたり、修正することができます。そして、そのスクリプトを実行させるには端末からのシェルへの入力を代行させる意味で

```
$ sh < script
```

とシェルの標準入力に流し込む方法が使えます。また、シェルは引数があるとそれを入力ファイルとして扱いますから

```
$ sh script
```

としても構いません。しかし、このようにスクリプトを起動する度にシェルの引数とするのは面倒ですから、スクリプトに「実行権」を与えて使います。「実行権」とは minix のファイルシステムに用意されている許可属性の１つで、この権利が与えられているファイルはコマンドとして直接起動することができます。

テキストファイルとして書かれたスクリプトに実行権を与えるには chmod コマンドを使用して、

```
$ chmod +x script
```

とします。このあとはスクリプトを修正してもファイルの実行権が失われることはありません。

このようにしてコマンドラインからシェルスクリプトを起動すると、カレントシェルは子プロセスとしてもう一つシェルを走らせ、そこでシェルスクリプトを実行します。この実際にスクリプトを実行するために走るシェルをサブシェル といいます。子プロセスのサブシェルは親プロセス（カレントシェル）の環境を受け継ぐことはできますが、その逆はできません。例えば、実行属性を与えたシェルスクリプトを使って環境変数を再設定しようとしても、再設定されるのはサブシェル側の環境変数であってカレントシェルのものではありません。

現在の環境変数を再設定するにはスクリプトをカレントシェルで実行しなければなりません。このためシェルにはドットコマンドと呼ばれるものが用意されおり、コマンドラインでドット . の後にスペースを置いて実行したいスクリプト名を書きます。

```
$ . script
```

ドット . を先頭に置くことによりカレントシェルは *script* を自分自身で実行しますので、現在の環境変数を再設定することができます。なお、ドット . に続く *script* には実行権の必要はありません。

3.2 シェル変数

すべてのプログラミング言語と同様にシェル言語でも変数に値を格納したり参照することができます。シェル変数は英文字あるいはアンダースコア _ で始まり、その後ろに０個以上の英数字あるいはアンダースコアのならび で表します。

シェル変数に値を格納するにはC言語などと同様にシェル変数と格納したい値を等号 = で連結し、

シェル変数＝値

とします。この時に等号 = の両側にスペース文字を入れてはいけません。CやPASCALなどのソースを読みやすくするため演算子の両側にスペースを置く習慣のある人は要注意です。

3.2.1 シェル変数の初期化と参照

シェル言語ではCやPASCAL等のプログラミング言語と異なりデータ型の概念がなく、すべての変数は文字列として扱われます。使用する変数は前もって宣言する必要がなく、シェルは新しい変数を見つけるとそれを登録し、ヌル値で初期化します。

シェル変数に格納されている値はシェル変数名の前にドル記号 **$** を付けることで参照できます。シェルは **$** の後ろに続く文字列が正しいシェル変数名ならばその変数に格納されている値と置き換えます。何も格納されていないシェル変数をいきなり参照した場合には初期値のヌルで置き換えられます。1つの例を見てみましょう。シェル変数 val に ”nora” という値（文字列）を格納し、echo コマンドでこれを表示させるには次のようにします。

```
$ val=nora
$ echo $val
nora
```

最後の nora は echo コマンドの出力です。このようにシェルは $val をその変数に格納された内容と置き換えてから echo に引数として渡します。いま例に引いた val を使い、”nora_neko”と表示させようとする時には注意が必要です。

```
$ echo $val_neko
```

とするとシェルは ”val” ではなく ”val_neko ”を変数名として処理します。これは、シェル変数を表す **$** に英数文字が続いているならば最も長い語句を変数名として切り出すために起こります。ここでシェル変数 ”val_neko” は未定義ですからヌル値で置き換えられ何も表示されません。この不都合を回避するには中括弧 {, } でシェル変数部分の範囲を明示してやります。

```
$ echo ${val}_neko
nora_neko
```

シェル変数は一般の文字列以外にもメタ文字と呼ばれるシェルにとっては特殊な意味をもつ文字を格納することもできます。次の例を見てください。

```
$ val=*
$ echo $val
```

普通の文字列が格納されている時にはシェル変数 $val の値を展開するだけでしたが、今度はもう少し手の込んだ仕事をしてきます。シェルはコマンドラインに echo $val を受け取るとこれを走査して変数 val を見つけ、その内容を変数 val に格納されている * で置き換えます。この文字 * はシェルにとっては特別な意味を持つ文字ですからカレントディレクトリ内の全ファイル名を引数として echo を起動します。この3つの例からシェルはどのように変数を評価しているかを理解してください。

ここまでの説明で気が付いたと思いますが、環境変数はシェル変数そのものです。シェルスクリプト内で使われる一般のシェル変数とは違い、ユーザの使用環境を整えるために使われるので環境変数と呼ばれています。

3.2.2 エクスポート変数

あるスクリプトから別のスクリプトにシェル変数値を渡したり、ログイン中に実行するすべてのコマンドから参照させたいシェル変数は **export** 文で宣言することによりその内容を引き渡すことができます。**export** 文の書式は以下の通りです。

export *shell_variables*

shell_variables は他のスクリプトに引き渡したいシェル変数名のリストで、スペースで区切ってならべます。

ユーザがログイン完了後に走る最初のシェルをログインシェル と言い、すべての親シェルとなります。ここでユーザが何かスクリプトを実行させるとログインシェルはこのために新しいシェルを走らせます。シェルスクリプトの実行方法を思い出してください。

```
$ sh script
```

これはシェルのコマンドラインで *script* を引数としてもう一つのシェルを走らせていることをにほかなりません。シェルスクリプトに実行属性を持たせて起動しても内部ではこれとまったく同じ手順で実行されています。親シェルはこの時に自分のシェル変数の中で **export** されているものだけをサブシェル側でも参照できるようにコピーします。このシェル変数はエクスポート (export) 変数とも呼ばれ、その場所から実行されるサブシェルに対して伝えられていきます。**export** 宣言されなかったものはローカル変数として子孫には伝えられません。

もし、エクスポート変数がサブシェル側で書き換えられたならば、その影響はコピーされていく子孫側にだけ及び、親となったシェル側のシェル変数内容を変更することは 決してありません。親シェル側からすれば出て行く export であって入ってくる import ではありません。

3.2.3 readonly 変数

間違って書き換えては困るシェル変数には **readonly** 文で読み出し専用属性を与え、保護することができます。**readonly** 文は引数として与えられたシェル変数を読み出し専用属性にします。一般書式は次のようになります。

readonly *shell_variables*

読み出し専用属性を設定されたシェル変数を書き換えようとするとエラーメッセージが表示されます。シェル変数をいったん read only としてしまうと再び元の状態に戻す方法はありません。

なお、現在のシェルでエクスポート変数に read only 属性を与えてもサブシェルに渡されるのはその変数値だけであり、read only 属性は渡されません。

3.3 引用符

シェルはコマンドラインにメタ記号などの特殊文字を見つけると展開や置換を行います。これはシェルの持つ便利な機能ですが、時としてユーザが期待しないことをやってしまいます。次の例をみてください。

$ echo * means all files in the directory.

echo コマンドで「* means all files in the directory.」とメッセージを出力たいだけなのですが、シェルは自分の仕事を忠実に実行し * をカレントディレクトリのファイル名で置換してしまいます。さらに * と「means」の間にある２つのスペースも区切りとしては認識されますが、個数は無視されています。

期待通りの表示をさせるためにはコマンドラインをシェルの置換機能から保護しなければなりません。そのために引用符を用います。シェルにはそれぞれ異なった働きをする単一引用符 ’、二重引用符 ”、逆引用符 ‘ の３つが用意されており必ず対で使われます。

3.3.1 単一引用符 ’

シェルは最初の単一引用符 ’ を見つけるとそれを閉じる単一引用符 ’ に出会うまですべての特殊文字を無視します。先の例では

$ echo ’ * means all files in the directory.’

とすれば期待通りのメッセージが表示されます。

3.3.2 二重引用符 ”

単一引用符と同様にほとんどの特殊文字を無視します。しかし二重引用符の中であってもドル記号 **$**、逆引用符 ‘、バックスラッシュ \ の３つについては認識されます。ドル記号 **$**が解釈されますので二重引用符内ではシェル変数の置換が行われます。次の２つの例は引用符で囲まれた文字列はまったく同じものですが、囲んでいる引用符が一方は単一引用符で他方は二重引用符です。実際に試して２つ違いを確かめてください。

$ echo ’ $val means all files in the directory.’
$ echo ” $val means all files in the directory.”

バックスラッシュ \ はその直後に続く一文字の特殊な意味を取り除く時に使われます。二重引用符内ではシェル変数の置換が行われことを説明しましたが、

$ echo ” \$val means all files in the directory.”

とするとしてしまうと結果は大きく変わってしまいます。シェルはバックスラッシュに続く **$** を無視しますので $val はシェル変数ではなく単なる文字列としてなってしまい、変数の置換は行われません。

3.3.3 逆引用符 ‘

単一引用符、二重引用符はコマンドラインの文字列をシェルから保護する働きがありますが、逆引用符 ‘ はこれで囲まれたコマンドを実行し、その結果を文字列として引用する機能を持っています。言い替えるならば逆引用符に囲まれた文字列をみつけるとそれをコマンドとして実行し、そのコマンドからの標準出力で逆引用符部分を置き換えます。

$ echo ”The date & time is: ‘date‘ ”

はその一例です。シェルはコマンドラインを走査しその中に ‘date‘ を見つけて date コマンドを実行します。そして、その出力でコマンドラインの ‘date‘ 部分を置き換えます。あとは今まで説明したようにして echo コマンドを走らせます。

逆引用符内で実行できるコマンドは１つとは限りません。セミコロン ; で区切れば複数のコマンドを書くことができますし、( ) を使ってのグルーピングやパイプを使って、

$ echo ” ‘ls | wc -l‘ files in your directory.”

などということもできます。

3.4 特殊なシェル変数

3.4.1 IFS

この変数にはシェルがコマンドラインを走査するときの区切り文字のリストが格納されています。通常はホワイト

スペース文字と呼ばれるスペース、タブ、改行文字がこれにあたります。

この **IFS** の内容はユーザの任意の文字に変更することができますから、ホワイトスペース文字以外で区切られた１行から文字列を切り出す時に有効に働きます。このため、**IFS** には１つのレコードを構成するそれぞれのフィールドを区切る文字が格納されていると考えた方がより現実的です。ちなみに **IFS** とは **I**nternal **F**ield **S**eparator の略です。

ここでちょっと IFS の内容を見てみましょう。IFS はシェル変数ですから

```
$ echo $IFS
```

とするだけで見れそうですが、次の１行が空くだけで何も表示されません。これは IFS の内容が空白文字としてスペース、タブ、改行から成っているためです。ちょっと工夫をして、次のようにすれば内容を見ることができます。

```
$ echo -n "$IFS" | od -b
0000000 040 011 012
0000003
```

これは echo の出力を od コマンドにパイプでつなぎ IFS の内容の８進数ダンプしたものです。先頭の数字は入力ファイルの先頭からのオフセット番地で、それに続く 040 011 012 が IFS の内容となります。

3.4.2 $#

シェルスクリプトが実行されるとシェル変数 `$#` にはコマンドラインに与えられた引数の個数が格納されます。ユーザが入力した引数の数が正しいかを調べたり、引数の数を見て処理を分岐させるときなどに利用します。

3.4.3 位置パラメータ ($1〜$9, $0) と shift

シェルスクリプトも他のコマンド同様に引数を受け取るとができます。シェルはコマンドラインを処理した後でこのために用意された特殊なシェル変数に与えられた引数値を格納してからスクリプトを実行します。これらの特殊なシェル変数は位置パラメータと呼ばれドル記号 **$** に続く１文字の数字で表現します。$1, $2 ··· $9 はそれぞれ第１、第２、··· 第９引数に対応します。

位置パラメータはドル記号 **$** に続く１個の数字で表されますので９個を超える引数を直接参照することはできません。もし 10 個目、あるいはそれ以降の位置パラメータを参照するには **shift** コマンドを使います。**shift** コマンドを実行すると位置パラメータの内容が左に１つシフトし、$1 に $2 の内容が、$2 には $3 の内容が、··· と位置パラメータの内容が順次左に送られ、$9 にいままで隠れていた第 10 引数の内容が入ります。この時に引数の数を表すシェル変数 `$#` の内容も１つ減少します。

shift コマンドで位置パラメータをシフトさせると古い $1 の内容は失われてしまいます。もしその後の処理に古い $1 が必要ならば **shift** を使う前に退避しておかなければなりません。

また、**shift** コマンドに引数 n を与えることにより一度に n 回のシフトさせることもできます。一般書式は次のようになります。

shift n

引数の数（`$#`）がゼロになりこれ以上シフトできなくなった場合にはエラーメッセージ "nothing to shift" が返されます。

$0 にはプログラム名（スクリプト名）が格納されています。これを使えば/usr/bin/compress のように zcat にリンク張り、コマンド名によって処理内容を分けることもできます。

3.4.4 $*

シェル変数 `$*` はシェルスクリプトが受け取った $0 以外のすべての引数に対応します。不特定数の引数を処理する時に利用します。

3.4.5 $@

`$*` と同じくスクリプトが受け取ったすべての引数に対応します。`$*` との違いは二重引用符で囲んで `"$@"` としたときに位置パラメータの評価をせずにコマンドに渡すことです。もし、二重引用符で囲まなければ `$*` と同じ意味になります。

3.4.6 $?

シェル変数 `$?` はシェルが最後に実行したコマンドの終了状態を保持していいます。直前に実行したプログラムの終了状態を知りたいときに使います。

3.4.7 $$

シェル変数 `$$` は現在のシェルのプロセス番号を保持しています。シェルスクリプト内で一時作業ファイルを作る時に利用します。すべてのプロセスは重複しない固有の番号で管理されていますから一時作業ファイル（テンポラリファイル）にプロセス番号を利用すると他のプロセスが

使ってかも知れない作業ファイルとの衝突を避けることができます。

3.4.8 $-

シェルにセットされているオプションを保持しています。

3.4.9 $!

バックグラウンドで実行された直前のプロセスのプロセス番号を保持しています。

3.5 set コマンド

set はシェルの内部コマンドです。シェルのオプションを設定／解除する機能だけでなく、１つのレコードからシェル変数 **IFS** で区切られたフィールド取り出し、それを位置パラメータに代入する機能も持っています。

set によるシェルオプションの設定／解除機能は起動時にオプションをコマンドラインで指定したのと等価です。さらにスクリプト内の任意の位置でオプションの設定や解除ができますのでスクリプトをデバッグするときに必要な部分だけの実行状態を監視することもできます。

位置パラメータへの代入機能は **set** の引数として与えられた文字列（レコード）から **IFS** を区切としてそれぞれのフィールドを切り出し、$1〜$9 ··· に格納します。この機能はユーザが位置パラメータに値を代入できる唯一の方法で、シェルスクリプト内で多用されます。

```
$ set ‘date‘ ; echo $1
```

を実行してみると[1] メカニズムが良く分かると思います。date コマンドの出力が **set** に渡されています。シェルの **IFS** はホワイトスペース文字ですから date コマンドの出力をスペースで区切って位置パラメータに代入します。date コマンドの出力の第１フィールドは曜日を表していますので、それが位置パラメータ**$1** に抜き出されているのが echo の出力で分かります。また、この時に位置パラメータの数を表すシェル変数 $# には **set** コマンドで分解されたフィールド数が格納されています。これも echo で確認してみてください。この代入が行われると古い位置パラメータの値は永久に失われてしまいます。もし、あとで必要なものがあればユーザの責任で事前に退避しておかなければなりません。

[1] date コマンドの出力は、以下のようになる。

```
$date
 Mon Jun 21 17:32:39 JST 1993
```

ここで「JST」は、地域の時間であり、日本では JST であるが、例えばグリニッチ標準時刻を採用しているマシンでは、GMT となる。minix では、JST も GMT も表示されない。

set コマンドに引数を与えずに使うと、ユーザの環境内に存在するすべての変数が表示されます。

4 シェルの構文

4.1 構文規則

シェルプログラム言語は minix の単純なコマンドを組み合せるための制御機構を備えており、対話形式のコマンドラインで可能なことはすべて書くことができます。

１つのコマンドは改行文字 (0x0a)、あるいはセミコロン **;** で終結します。そして、**if**, **while**, **do**, **done**, ··· などのシェルの予約語は必ず行の先頭になければなりません。行の先頭とは改行文字の直後のことを言いますが、セミコロンやパイプ文字の直後も行の先頭に含まれます。

4.2 注釈文

どんなプログラミング言語にも必ず用意されているのが注釈文（コメント文）です。プログラムの保守を容易にし可読性を高めるために注意点やメモを挿入するのに使われ、シェルスクリプトの実行には影響を及ぼしません。シェル言語では # で始まる文字以降から行末までが注釈文とみなされます。行頭から始まる場合はその行全体がコメント行として実行時に無視されます。

また、シェルスクリプト内には何も書かない空白行も許されます。コメント文と組み合わせて適切に使用すると後日のデバッグや保守が容易になります。

4.3 入力文

4.3.1 read 文

read 文は標準入力から１行を読み込み、引数として与えられたシェル変数リストに順次代入します。

read *val-list*

この時に、読み込む行の先頭にある **IFS** で指定された空白文字は無視されます。標準入力が端末のキーボードならば、スクリプト内でユーザからの１行入力に使うことができます。リダイレクトやパイプなどにより標準入力が切り換えられているならばそこから１行を読み込むことになります。

read の引数がシェル変数リストならば **IFS** で区切られた語句がそれぞれに代入されます。もし、読み込んだ１行から分解した語句の数よりもシェル変数名リスト *val-list* に列記された数が少なければあふれた語句は変数名リストの最後に書かれたものにまとめて代入されます。

read 文の終了状態は EOF を検出しない限りゼロ（真）です。

4.4 条件判断と分岐

4.4.1 if 文

シェル言語もほとんどのプログラム言語と同様に条件判断のための **if** 文を持っています。**if** 文は１つ以上の条件をテストし、その結果に基づいてプログラムの流れを分岐させます。**if** 文の一般書式は次の通りです。

```
if cond                    if cond
then                       then
    commands                   commands
    ...                        ...
else                       elif cond
    commands               then
    ...                        commands
fi                             ...
                           else
                               commands
                               ...
                           fi
```

この文では *cond* の位置に書かれたコマンドの終了状態（実行結果）を調べて終了状態がゼロ（真）の場合には **then** 節が、そうでない場合には **else** 節が実行されます。もし必要がなければ **else** 節は省略することができますが、**if** 文の終了を表す予約語 **fi** は省くことができません。なお、後者の書式では **elif** 節のネストができます。

終了状態がゼロ（真）の時に **then** 節が実行されることは他のプログラミング言語からすれば逆の印象を受けるかもしれません。minix では、あるコマンドが終了するとその終了状態 (exit status) を表す数値がシステムに返されます。これはそのプログラムが正常に実行／終了できたかどうかを示すもので、正常に終了した場合はゼロ（真）が返されます。もし、終了状態が非ゼロ（偽）ならば何らかの原因で、例えば引数の数が適切でなかったとか、プログラムがエラーを検出したとか ··· 等々、異常が起きたと考えられます。ですからコマンドが正常に終了したならば **then** 節に分岐、と考えれば受け入れやすいと思います。

4.4.2 test コマンド

初めて minix に接した時、使用目的にとまどうコマンドの１つです。これはシェル内部に組み込まれたものではなく、 minix の一般コマンドの１つですが、シェルスクリプトで条件判断を行なう上で避けて通れませんから少し説明しておきます。

test コマンドは与えられた引数を条件式として評価し、その結果が真の時は終了状態にゼロ（真）を、偽の場合はゼロ以外の値を返します。**test** コマンドは下記のような文字列、整数、ファイル状態等について多くの条件式を評価をすることができます。さらにそれらの論理演算を組み合わせてより複雑な評価もできます。ただし、評価のショートカット[2] は行われません。

文字列評価式:

str1 = *str2*	文字列 *str1* と *str2* は一致する
str1!=*str2*	文字列 *str1* と *str2* は一致しない
-n *str*	文字列 *str* は空 (null) でない
-z *str*	文字列 *str* は空 (null)

数値評価式:

int1 **-eq** *int2*	整数 *int1* と *int2* は等しい (*int1* == *int2*)
int1 **-ge** *int2*	整数 *int1* は *int2* 以上である (*int1* >= *int2*)
int1 **-gt** *int2*	整数 *int1* は *int2* よりも大きい (*int1* > *int2*)
int1 **-le** *int2*	整数 *int1* は *int2* 以下である (*int1* <= *int2*)
int1 **-lt** *int2*	整数 *int1* は *int2* よりも小さい (*int1* < *int2*)
int1 **-ne** *int2*	整数 *int1* と *int2* は等しくない (*int1* != *int2*)

ファイル評価式:

-d *file*	*file* はディレクトリである
-f *file*	*file* は通常ファイルである
-r *file*	*file* は読み出し可能である
-s *file*	*file* の長さは０バイトではない
-w *file*	*file* は書き込み可能である
-x *file*	*file* は実行可能である

論理演算子:

!	直後に続く条件評価式の結果を否定する
-a	２つの条件評価式の論理積 (and) をとる
-o	２つの条件評価式の論理和 (or) をとる

4.4.3 : コマンド

コロン **:** で表されるこのコマンドは引数を評価するだけで終了状態にゼロ（真）を返します。何もしないコマン

[2] AND 条件や OR 条件のついた判定をする場合、例えば AND にあっては前の項が False であれば、後ろの項が True であっても False であっても、結果は False になるので、後ろの項の判定をしないことがある。これらをショートカットと言う。

ドは存在価値も無いと思われそうですが、シェル言語にとっては「何もしない」コマンドの必要性がしばしばあります。次の例にある **if** 文や **while** 文と組み合せた使い方はその典型です。

```
if cond ; then          while    :
    :                   do
else                        commands
    commands            done
fi
```

また、**:** コマンドでは引数の評価も行なわれますのでシェル変数を検査させるためにも使われます。「6.3 ハノイの塔」(p.20) に具体例を上げておきます。

4.4.4 && と ||

`&&` と `||` は左側に置かれたコマンドの実行結果を見て右側に置かれたコマンドを実行させるもので、`&&` は *command1* の終了状態がゼロ（真）の時に *command2* を実行します。一方、`||` は *command1* の終了状態が非ゼロ（偽）の時に *command2* を実行します。終了状態は最後に実行されたコマンドのものになります。それぞれの一般書式と **if** 文で書いた等価式は次のようになります。

command1 `&&` *command2*

⇒ if *command1*; then *command2*; fi

command1 `||` *command2*

⇒ if ! *command1*; then *command2*; fi

4.4.5 case 文

case 文は１つのシェル変数値を評価し、同じパターンが見つかったならば１つあるいはそれ以上のコマンドを実行させます。一般書式は次のようになります。

```
case $val in
    pat1 )  commands
              ;;
    pat2 )  commands
              ;;
            ...
    patn )  commands
              ;;
esac
```

ここでの処理はシェル変数 *val* の値と *pat1*, *pat2*, ···, *patn* と連続して比較し、一致したものが見つかるとそこから２個の連続したセミコロン **;;** まで間に書かれたコマンド群を実行します。もし、一致するものが見つからなかった場合は何もしません。最後の **easc** は **case** 文の終りを意味し省略することはできません。

シェルでファイル名の置き換えに用いられるメタ文字も **case** 文の比較対象 (*patn*) として使うことができます。また、シェルではパイプ記号として用いられる垂直バー | を使うと複数パターンの論理和 (or) を取ることができます。

4.5 ループ制御

4.5.1 for 文

一組のコマンドを指定された回数だけ実行するのに使われ、一般書式は次のようになっています。

```
for val in arg1 arg2 ··· argn
do
    commands
      ...
done
```

do と **done** に囲まれたコマンド群がループの本体で、これらのコマンド群は **in** の後ろに並べられた引数の数だけ繰返し実行されます。このループが実行されると最初に *arg1* の内容が *val* に、次に *arg2* の内容が *val* に ···、と順次が参照されながら **in** に続くリストの中身が空になるまでループ処理が継続します。つまり、**in** の後ろに *n* 個の引数があったならば *n* 回ループが実行されることになります。**in** に続く引数に *****, **?**, [ ] などのメタ文字が含まれる場合にはシェルにより展開されてから **for** ループが実行されます。

シェル変数 `$*` はスクリプトに渡されたすべての引数に対応するために **for** 文と組み合わせて使うことができます。しかし、実際の使用に当っては少し注意が必要です。次に示すものはコマンドラインに入力された引数を１行に１個づつ表示させるものですが、時として期待を裏切ります。

```
echo "Number of arguments: $#"
for i in $*
do
    echo $i
done
```

このスクリプトを for.sh として実行してみましょう。まずは、

```
$ for.sh A B C
Number of argumets: 3
A
```

B
C

これは正常です。ではつぎに A と B を引用符で囲んでみましょう。

$ for.sh 'A B' C
Number of argumets: 2
A
B
C

さて、不思議なことが起こるものです。単一引用符は文字列をシェルから保護しますから引数は２つ。事実、与えられた引数の数を示すシェル変数`$#` は間違いなく２と表示されています。なのになぜか３行に出力されてしまいました。

種明かしはこうです。'A B' はシェルから保護されて１つの引数としてスクリプトに渡されます。しかし、保護されるのはコマンドラインでのことであり、引数としてスクリプトに渡った時には単一引用符が外れています。ですから、**for** 文で `$*` が展開された時、**in** 以降のリストには A と B を１つにしている単一引用符がありませんので、

for i in A B C

となってしまい、ループは３回実行されることになります。これを回避するために **for** 文には **in** 以降のリストを持たない特別な記述が許されています。for.sh ２行目の **in** 以降を省略し、

```
echo "Number of arguments: $#"
for i
do
    echo $i
done
```

としたものです。これで試してみると

$ for.sh 'A B' C
Number of argumets: 2
A B
C

となるはずです。さらに、特殊なシェル変数 `$@` を使うもの１つの方法です。シェル変数 `$*` はコマンドラインでの引用符が外された状態の引数を $1, $2, $3 として持っています。この `$*` の代わりに"`$@`" を使うと "`$1`", "`$2`", "`$3`" として置き換えられるために `$*` でのような不都合は起こりません。ただし、二重引用符で囲み "`$@`" としなければ `$*` とまったく同じように展開されてしまいます。

4.5.2 while 文

ある条件が満たされている間ループを実行します。一般書式は次の通りです。

```
while  cond
do
    commands
    ...
done
```

まず最初に *cond* コマンドが実行されその終了状態がテストされます。もし終了状態がゼロ（真）ならば **do** と **done** で囲まれたコマンド群を実行します。そして、もう一度 *cond* が実行されて終了状態が検査されます。もしゼロ（真）ならば再び **do**〜**done** のコマンド群が実行され、非ゼロ（偽）ならば **done** の次のステップに進みます。for.sh と同じ働きを **while** 文で実現した例を示します。ループ条件の判断には test コマンドを使って引数の数を検査しています。

```
echo "Number of arguments: $#"
while test $# -gt 0
do
    echo $1
    shift
done
```

4.5.3 until 文

until 文は while 文とは反対に終了状態が非ゼロ（偽）である間ループが実行されます。期待する現象が起こるのを待って処理を行う場合などに使われ、一般書式は次のようになります。

```
until  cond
do
    commands
    ...
done
```

minix のようなマルチタスク OS で複数のプロセスが１つのファイルを共有する時には排他制御をしなければなりません。このためのロックファイルをスクリプトで作るときに次のような使い方をします。

```
until（ロックファイルを作る）
do
    sleep 30
done
```

until の条件文で排他制御のためのロックファイルを作ろうとします。既にロックファイルが存在していたならば、そのファイルは他のプロセスで使用中ですから作成に失敗して非ゼロ（偽）が返されます。ファイルへのアクセス権が得られないならば **do**〜**done** が実行されます。この例では 30 秒間隔で再試行を行ない、ロックファイルが作れたならば次の処理に進みます。

4.5.4 break 文

ループ処理をしている時に、ある状態になったならばすぐにそのループから脱出したい時があります。シェル言語ではこのために **break** 文が用意されています。**break** 文が実行されると制御はただちにそのループの外に移り、**done** の次のステップから実行されます。さらに、ループがネストしており、複数のループを一気に脱出したい時には **break** に引数として脱出したいループの数を指定します。一般書式は次の通りです。

break n

引数 n が省略された場合には 1 として、最も内側のループから脱出します。

4.5.5 continue 文

continue はある条件が満たされているときなどにループ内のコマンド群をスキップするために使われます。**continue** 文はそれ以降のコマンド群をスキップするだけでループは継続条件が満さたれている限り続けられます。break 文と同様に引数をつけることにより n 番目のループから実行させることができます。一般書式は次の通りです。

continue n

4.5.6 ループ文のリダイレクト、パイプとバックグランドでの実行

for, **while**, **until** のループ制御文は **do**〜**done** とで囲まれるコマンド群を 1 つのコマンドのように実行しますのでループ全体の入出力をリダイレクトしたり、パイプに接続することができます。バックグランドでの実行もループ全体が対象となります。

次のリストは **for** 文の標準出力をリダイレクトする例です。

```
1         for i in beer carrot ell
2         do      echo $i
3         done >food
```

ループ全体が 1 つのコマンドとして扱われますからリダイレクト文字 > はループの終了を表す **done** の後書きます。この時のリダイレクト対象はループ内で標準出力に書き出すものすべてが対象となります。しかし、ループ内でリダイレクト先が明示されているものはループ全体のものに優先してされます。例えば、上記のリストで

```
1         for i in beer carrot ell
2         do      echo "I like $i" >/dev/tty
3                 echo $i
4         done >food
```

としたならば、3 行目の出力は **done >food** に優先して **/dev/tty** にリダイレクトされます。（この場合には端末です）

同様に下記のようにパイプ記号 | を **done** に続けて書くことによりループの出力をパイプに流し込むこともできます。

```
1         for i in beer carrot ell
2         do      echo $i
3         done | food
```

ループ処理全体をバックグランドで実行させるにはループの終了を示す **done** の後ろにバックグランドへ送る指示のアンパーサンド & を付けます。例えば複数のソースからなるプログラムリストをバックグランドで連続紙に印刷するには次のようにします。

```
1         for i in *.[hc]
2         do      pr -l66 -w132 $i | lpr
3         done &
```

for や **while** 文などのリダイレクト処理はカレントシェルではなく、サブシェルで実行される ことに注意してください。このことを知らないとバグでもないのにおかしな現象に悩まされることになります。次のスクリプトは自分自身を行番号付きで表示するもので、仮に ”myself” と名付けておきましょう。

```
1         n=0
2         while read line
3         do      n=‘expr $n + 1‘
4                 echo "$n: $line"
5         done < $0
6         echo "total line= $n"
```

これを myself として実行してみてください。各行の先頭にふられた行番号は正しく表示されていますが、最後に total line= 0 と表示され期待した結果が得られません。種明しをすると、ループ文の内側で行番号を表示するために用いられているカウンタ **n** はサブシェル側の変数で、ループの外側にある変数 **n** はカレントシェルのものです。（この myself を起動したシェルから見ると子、孫の関係になります）同じ名称の変数でもループの内と外ではまったく別物ですから、**while** ループの外側にある変数 **n** にはスクリプトの最初で初期化されたままになっています。

ここで混同しないで頂きたいのですが、ループ文がサブシェルで実行されるのは スクリプト内 でリダイレクト処理を指定したときだけです。リダイレクトを指定しないループ文はカレントシェルで実行されます。スクリプト myself を次のように書き直してみてください。

```
1         n=0
2         while read line
3         do      n=‘expr $n + 1‘
4                 echo "$n: $line"
5         done
6         echo "total line: $n"
```

そして、コマンドラインから ”myself < myself ” とすると行数を数える変数 **n** は期待する値を取ることで確認できます。さらに、入力をパイプから読み込むよう、次のように書き直して変数 **n** の値を調べてみてください。

```
1         n=0
2         cat $0 | while read line
3         do      n=‘expr $n + 1‘
4                 echo "$n: $line"
5         done
6         echo "total line: $n"
```

4.6 算術演算

シェルプログラミングで算術演算を行うには次のような書式で **expr** コマンドに算術式を引数として与えます。

expr *val1* 算術演算子 *val2*

この算術演算子には以下のようなものが使えます。

算術演算子	意　　味
(*arguments*)	括弧で囲んで優先順位を明示する
str : *regexp*	文字列 *str* と正規表現 *regexp* を比較する
val1 * *val2*	*val1* と *val2* の積
val1 / *val2*	*val1* ÷ *val2* の商
val1 % *val2*	*val1* ÷ *val2* の余り
val1 + *val2*	*val1* と *val2* の和
val1 - *val2*	*val1* と *val2* の差
val1 op *val2*	op に <, <=, =, != >=, >を用いて比較演算を行なう．条件成立ならば１を、不成立ならば０を返す
val1 & *val2*	*val1, val2* 共に０でなければ *val1* の値を、それ以外は０を返す
val1 | *val2*	*val1* が０でなければ *val1* を、*val1* が０ならば *val2* を返す

一致演算子 *str* : *regexp* は左側の文字列 *str* と右側の正規表現 *regecp* を比較して一致した文字数を返します。（もし一致なかったならばゼロが返ります）この時の正規表現には **ed** と同じ記法が使えますし、**ed** と同じ記憶メカニズム

\(正規表現 \)

を使って *str* の中から正規表現に一致した部分を抜き出すこともできます。

4.7 実行制御

4.7.1 exit 文

シェルはスクリプトの最後（End OF File）に達すると自動的に終了しますが、**exit** を使うことにより任意の位置で実行を終了させることができます。一般書式は次の通りです。

exit *n*

引数 *n* ではシステムに返す終了状態を指定します。通常、コマンドが正常に終了した時はゼロを返す慣わしになっています。もし *n* が省略された場合には **exit** の直前に実行されたコマンドの終了状態が返されます。

なお、ユーザ端末からコマンドラインで **exit** を実行させると現在のシェルを終了させることになります。もし、それがログインシェルならばログオフと同じ結果となります。

4.7.2 exec 文

シェルに代わって引数で指定されたコマンドを実行しますが、新しいプロセスは作られません。書式は次の通りです。

exec *argument*

シェルは **exec** の実行にあたり現在のファイルをクローズし、新しいファイルをオープンしますのでそれ以降の入出力を引数で指定したものに切り換えることができます。この機能を使って **exec** 文以降の標準入力を *file* に切り換えたいならば

exec < *file*

とします。同様に標準出力やエラー出力を *file* に切り換えたいのならば次のようにします。[「6.2 配列」(p.19) 参照]

exec >*file*
exec 2>*file*

exec の引数に普通のコマンドが交じっていたならば、それが実行されるだけです。

4.7.3 eval 文

引数をシェルの入力として解析してからコマンドとして実行します。引数を実行する前にコマンドラインの解析が 1 度行なわれますから、結果としてコマンドラインを 2 回走査させることができます。書式は次の通りです。[「6.2 配列」(p.19)、「6.3 ハノイの塔」(p.20) 参照]

eval *argument*

4.7.4 wait 文

ユーザが実行している子プロセスの終了を待ち、終了状態を保存します。書式は次の通りです。

wait *n*

引数 *n* には待ちたい子プロセスの識別番号（プロセス ID）を指定します。*n* を省略するとその時点で走っているすべての子プロセスの終了を待つことになります。なお、このコマンド自体の終了状態は待っていたプロセスの終了状態そのものです。

バックグランドで実行させた子プロセスの ID を知るにはシェル変数 `$!` を参照します。

4.7.5 trap 文

シェルスクリプトを書くときには実行中になんらかの原因で停止することも念頭に置かなければなりません。停止する要因としてはユーザの DEL キーよる割り込みとか、異常終了、シグナル 9 が送られたとか、さまざまなものが考えられます。

この時にただちにシェルスクリプトを終了させてしまうとまずい場合があります。例えば、一時作業ファイルを作って処理をしていて後始末をせずに終了してしまったのでは作業ファイルがディスクのゴミとして残ってしまいます。また、排他制御のためのロックファイルを削除せずに終了したならば困ったことになります。このような不都合を回避するために **trap** 命令を使います。

trap はあるシグナルを受信した時になすべき仕事を指定するのに使い、書式は次のようになっています。

trap command *signal*

command は *signal* に指定されたシグナルリストのいずれかを受信したときに実行されるコマンドです。*signal* を受信したときに実行するコマンドが 2 つ以上の場合にはそれらを引用符で囲まなければなりません。signal は一連の番号で表され、minix で使われる主なものは次のようなものです。

signal	意　　味
0	シェルから脱出するときに必ず発生する
1	ハングアップ。通常、回線のキャリアが切断すると発生する
2	DEL キーが押されたときに発生する端末割り込み
3	クイットシグナル。プロセスを停止させコアダンプを行なう
9	キルシグナル。すべてのプロセスで無視も受信もできない
15	kill コマンドにより発生するシグナル

次のリストは **trap** 文を使って一時作業ファイルを削除し、スクリプトを終了させる例です。

```
prog=‘basename $0‘
tmpfile="/tmp/$prog$$"
trap ’rm -f $tmpfile;exit 1’  2
any-command > $tmpfile
    ...
```

一時作業ファイル名には他のプロセスとの衝突を避けるためにプロセス ID を含めたものが使われます。ここでもその慣例に従ってシェルスクリプト名とプロセス ID が格納されているシェル変数 `$$` で一時作業ファイルを作っています。そして **trap** 文です。ここではシグナル 2 を受け取ったときに`’rm $tmpfile;exit 1’` が実行されるように設定します。なお、**trap** の設定は一時作業ファイルが作られる前に行なわなければ意味がありません。

シグナルを無視したい時には実行すべきコマンドを書く部分を引用符で囲んで「ヌル」にします。

trap ’’ *signal*

しかし、「コマンドを書かなければ無視される」と早合点し **trap** 文の第一引数を省略して

trap *signal*

としてはいけません。このようにすると *signal* を受け付けたときの処理をデフォルトに再設定してしまいます。例えばシグナル 2 ならばシェルスクリプトを停止させる標準処理を行ないます。

なお、ユーザがあるシグナルを無視するように設定すると、そこから起動されるサブシェルも（シェルスクリプト）そのシグナルを無視します。しかし、あるシグナルを受信したならば特定の処理を行うように設定した場合には、その処理内容はサブシェル側にはいっさい伝わらず、該当シグナルに対して既定の処理を行うだけです。

5 シェルスクリプトのデバッグ

シェルはさまざまな実行環境を作り出すために幾つかのオプションを持っています。スクリプトのデバッグ用としては**-v**、**-x** の２つオプションを使うことができます。

-v はスクリプトからコマンドを１行読むごとに表示させるためのオプションで、構文のチェックに利用できます。オプションを設定するにはシェルスクリプト内に

set **-v**

の１行を追加するか、あるいは次のようにしてシェルスクリプト走らせます。

sh **-v** *script*

-x オプションはコマンドを実行するたびにそのコマンド名と引数を + に続いて表示するものです。引数はシェルにより展開されたものが表示されますので実行状態をトレースすることができます。これらのオプションをまとめて、あるいは設定されているものだけを解除するには

set **-**

とします。もし、個別に解除したければ

set **＋v**

などとします。なお、現時点でシェルに設定されているオプションを調べたい場合にはシェル変数 $- を参照します。

実際に使われることは少ないですが **-n**, **-u** なども役に立つかも知れません。**-n** は読み込んだコマンドの実行を禁止させるためのオプションです。予想もしないバグで大切なファイルを削除してしまうなどの被害を未然に防ぐことができます。カレントシェルで set -n とすると EOF(End Of File) を入力するまでその端末からの操作ができなくなります。

-u オプションを設定すると値が格納されていないシェル変数を参照しようとした時に unset variable のエラーメッセージが表示されスクリプトが停止します。

それといまさらでしょうが、ソースデバッグもお忘れなく。結局、これに勝るデバッグ手法はないようです。

6 シェルスクリプトの実例

短いシェルスクリプトを例に取りながら今までの復習をしてみましょう。

6.1 挨拶

きちんと挨拶されると気持ちがいいものです。そこで minix にも login したとき、ご主人様にちゃんとご挨拶ができるように教えることにしましょう。

```
 1: #!/bin/sh
 2:
 3: set `date`
 4: IFS=:
 5: set $4
 6:
 7: case $1 in
 8:     0[6-9] | 1[0-1])
 9:         echo "Good morning. Sir"
10:         ;;
11:     1[2-7])
12:         echo "Good afternoon. Sir"
13:         ;;
14:     1[89] | 2[01])
15:         echo "Good evning. Sir"
16:         ;;
17:     *)
18:         echo "GO TO Bed!! :-)"
19:         ;;
20: esac
```

解説

1 行目は minix 上ではコメントとしての役割しかありませんが、unix の csh では#! に続いて指定されたシェルでそのスクリプトを実行します。B シェルは .*nix の標準シェルですからそのまま他の環境に持ち込んでも走らせるオマジナイです。

ご挨拶スクリプトは最初に date コマンドの出力を set の引数として与え、スペースで区切られたフィールドを切り出します。続いて **IFS** をコロンに設定して、いま切り出した第４フィールドをさらに分割します。ここまでの操作で date コマンドの出力から時間を表す部分だけを取り出します。

そして、その時間を見て表示する挨拶メッセージを **case** 文で選択します。「おはよう」「こんにちは」「こんばんわ」のどれも言えないような時間帯にはあなたの健康を気づかって「寝た方がいいんじゃない？」となります。

このスクリプトをあなたの .profile に書いておくと login した直後に挨拶メッセージを見ることができます。

演習問題

さて、このスクリプトで挨拶メッセージを表示する時にあなたのログイン名を用いて”Good morning. taroh” などと表示させるにはどのように修正すればよいか考えてみてください。（メッセージ内にログイン名を直接埋め込むのは論外です）

6.2 配列 (表引き)

B シェルには配列操作のコマンドが組み込まれていませんが、シェル変数とコマンドを上手に操ることにより配列に格納されている値の参照と等価な仕事を実現できます。

```
 1: #!/bin/sh
 2:
 3: exec 3<&0 <month.name
 4:
 5: i=0
 6: while read k month
 7: do
 8:     i=‘expr $i + 1‘
 9:     eval M_$k=’"$month"’
10: done
11:
12: exec 0<&3 3<&-
13:
14: while test $i -gt 0
15: do
16:     eval echo \$i -- \$M_$i
17:     i=‘expr $i - 1‘
18: done
```

ファイル month.name の内容

```
3 March
1 January
11 November
2 Feburuary
4 April
12 December
6 June
8 August
9 Sepember
7 July
10 October
5 May
```

解説

このスクリプトは大きく分けて配列として扱うシェル変数へ別ファイルから読み込んだデータをセットする部分 (5〜9 行目) と、それを表示する部分 (11〜15 行目) から成っています。ここでは 12 カ月の各月に対応する名称を書いた month.name というファイル用意して使うことにします。

このスクリプトは最初に **read** 文を使いファイルからデータを読み込みますが、この時に 5 行目から始まる **while** 文の入力を month.name にリダイレクトして

```
while read k month
do
    ......
done < month.name
```

としてはいけません。理由は「4.5.6 ループのリダイレクト」(p.15) で説明したように **while** 文はサブシェルで実行されるためです。このためループ内のシェル変数はループの外側では参照できません。そこで 3 行目の処理となるわけです。「4.7.2 **exec**」(p.17) と 「2.3 リダイレクト」(p.4) とを併せて考えてください。3 行目の `exec 3<&0 <month.name` はまずファイルディスクリプタ 0 の入力をファイルディスクリプタ 3 に変更します。ファイルディスクリプタ 0 はシェルの標準入力ですからこの処理で標準入力がファイルディスクリプタ 3 に保存されることになります。続いてファイルディスクリプタ 0 で month.name を読み込み用にオープンしますので、それ以降の標準入力はファイル month.name となります。これでカレントシェルで month.name の内容を **read** で読むことができるようになりました。

read で読み込んだものを配列状態でシェル変数に格納する部分が 8 行目です。month.name から 1 行目の ”3 March” を読み込んだとします。この内容は **read** 文によりシェル変数 k に ”3”、month に ”March” 取り込まれ、8 行目に来ます。**eval** はシェルが引数を 2 度評価するのと等価な働きをします。1 度目の評価では `$k` の置換と右辺の単一引用符が外されますので

```
M_3="$month"
```

となります。続く 2 度目の評価で右辺の `$month` が置換されます。最終的に 8 行目は次のようになり、シェル変数 `M_3` への代入操作が実現できます。

```
M_3="March"
```

この処理は month.name が EOF になるまで繰り返されます。

while ループから抜けたならば 10 行目の **exec** 文でもう一度標準入力を切り換えます。先ほどファイルディスクリプタ 3 に保存しておいた元の標準入力を復帰させ、不要になったファイルディスクリプタ 3 をクローズします。これで 3 行目で標準入力を切り換える前の状態に戻ったことになります。

最後はシェル変数を配列の添字を移動してアクセスする動作を真似て month.name から読み込んだ内容を表示させます。シェル変数 **$i** は読み込んだ項目数を保持していますので、これがゼロになるまで減算しながら表示を行ないます。13 行目の **eval** でのシェル変数の変遷はご自身でたどってみてください。

6.3 ハノイの塔

C 言語を憶えたてのころに見たことがある方もたくさんいると思います。再帰呼び出しのサンプルとして有名な「ハノイの塔」をシェルスクリプトで書いたものです。速度は期待できませんが再帰処理さえも簡単にこなしてしまうシェル言語には目を見張るものがあります。

再帰による「ハノイの塔」の解を求めるスクリプトそのものは簡単なものですが、このことにこじつけて他のことも併せて説明しようと欲張ったので行数の多いスクリプトになってしまいました。

```
 1: #!/bin/sh
 2: # Tower of HANOI
 3:
 4: #: ${1?parameter unset}
 5: set -u
 6:
 7: sub='./hanoisub'
 8: export sub
 9: trap 'rm -f $sub;exit' 0 1 2 3 15
10:
11: cat > $sub << 'EOF'
12: eval X\=\`expr 6 - `expr $2 + $3`\`
13: if test $1 -gt 1; then
14:     $sub `expr $1 - 1` $2 $X
15: fi
16: echo "Move disk #$1 from $2 to $3."
17: if test $1 -gt 1; then
18:     $sub `expr $1 - 1` $X $3
19: fi
20: EOF
21:
22: chmod +x $sub
23: $sub $1 1 2
```

解説

「ハノイの塔」スクリプトの本質は 23 行目の `$sub $1 1 2` で、`$1` にはスクリプトの第 1 引数としてに与えられた円盤の枚数がセットされています。このあとは `$sub` が再帰呼び出しを重ねて解を表示してきます。

サブルーチン `$sub` の所在は 11〜20 行目のヒアドキュメント部分です。サブルーチンをメインのシェルスクリプトに含めておき、実行するときにだけサブルーチンをディスクに展開して使用し、終了時には削除するようにしてあります。

では 7 行目から見て行きましょう。ここは前準備部分で、サブルーチンとするシェルスクリプト名を定義し、それを **export** 宣言 (8 行目) しサブシェル側に伝わるようにしておきます。続いて **trap** 文を使って終了する時には必ずサブルーチンとしてディスクに書き出したスクリプトを削除するようにしておきます。これをスクリプトの最後に (24 行目あたりに) 削除命令を入れておいたのでは DEL キーによる中断などの時にゴミを残してしまいます。

11 行目の `cat >$sub << 'EOF'` が 12〜19 行目をサブルーチンとしてディスクに書き出します。

```
1: eval X=\`expr 6 - `expr $2 + $3`\`
2: if test $1 -gt 1; then
3:         $sub `expr $1 - 1` $2 $X
4: fi
5: echo "Move disk #$1 from  $2 to  $3."
6: if test $1 -gt 1; then
7:         $sub `expr $1 - 1` $X $3
8: fi
```

`$sub` を **export** 宣言した意味がお分かりでしょうか。このサブルーチンにも同じシェル変数が 3 行目と 7 行目に使われています。もし、**export** されていなければこの変数はサブシェル側では未定義となりますのでシェルは `` `expr $1 - 1` `` をコマンドと解釈して誤動作の原因になります。

サブルーチン `$sub` を切り出したならば実行属性を与えてスクリプトとして実行できるようにします。そして、柱に通してある円盤の数を第 1 引数として `$sub` を起動します。解を求める作業が終了すれば`$sub` から戻ってきます。

ディスクに書き出したサブルーチンの後始末は 9 行目の **trap** が請け負っています。実際には 23 行目の処理が済むスクリプトを終える時点でシグナル 0 が送られてきますから **trap** の第 1 引数で指定した `rm -f $sub` が実行されます。

4 行目の `set -u` は「5 スクリプトのデバッグ」(p.18) で説明したスイッチで、未定義の変数が参照された時にエラーメッセージを表示してスクリプトを停止させるものです。ここでは引数なしで hanoi が呼び出されたときの処理に利用しています。

さて、5 行目に見慣れないものが書かれています。これはシェル変数置換の一種で、第一引数 ($1) が未定義ならばエラーメッセージ "Parameter unset" を表示してスクリプトを停止させます。表示させたいメッセージを書かずに: `${1?}` とすればシェルが持っているエラーメッセージを表示します。この時にヌルコマンドとしてのコロン : を行頭に置くことを忘れないでください。[「4.4.3 ヌルコマンド」(p.13) 参照]

クエスチョンマーク? 以外にも、マイナス記号 - や等号 = も使うことができ、それぞれ次のような意味を持っています。

:${*val*?*msg*}	シェル変数 *val* が未定義ならば *msg* を表示してスクリプトを停止
:${*val*-*str*}	シェル変数 *val* が未定義ならば *str* を代入する
:${*val*=*str*}	": ${*val*-*str*}" に同じ。但し、位置パラメータには適用不可

本題からそれますが、このスクリプトを PS5523-S(386sx, 12MHz) で実行させたところ円盤が 8 枚の時の 255 個の解を求めるのに 7 分 20 秒ほどかかりました。この時に `$sub(./hanoisub)` が 7 つ重なり、さらにそれを呼び出した親シェル hanoi、そしてログインシェルと 9 つのシェルが走ります。1 つのシェルは 50k ほどのメモリを必要としますから、小さい minix といえどもかなりのメモリが必要になります。(BSD あたりに比べると可愛いのですが) リアルモードの minix で円盤が 8 枚の解を求めるのは無理かも知れません。

演習問題

例として取り上げたハノイの塔のスクリプトは起動された本体からサブルーチン用のスクリプトを展開し、それを呼び出しています。解を求めるために 2 つのスクリプトが必要になることに変わりありません。そこで、サブルーチンを用いずに 1 つのスクリプトだけで解を求めるものを作ってみてください。

7 シェルスクリプトによる簡単なデータベース

シェル言語を使って特別なプログラムによらずにシェル言語と実装されているコマンドだけで少し実用的なスクリプト「超簡易文献データベース **Cabinet**」を作ってみましょう。

7.1 概要

キャビネットから物を捜したり、出し入れしたりといったことを真似ていますので名前もそのものズバリ **Cabinet** です。実際のキャビネットに収められている物は文献やビデオなど多種多様です。違った種類のものを詰め込むような設計にもできます。初心者にも分かりやすいようにできるだけ簡素な構成の方が良いでしょうから、文献検索用に目的を限定します。

サンプルとた文献データにはこれから minix あるいは、本格的に unix を勉強する際の手助けになる資料として使えるものを用意しました。

7.2 データ形式

扱うデータはすべて unix や minix の標準的なテキストファイルです。テキストファイルは人間も読めますが、ユーザが扱いやすいデータ形式と、minix のツール類が扱いやすい形式は違います。人間は極めて融通がききますから (考えようによってはいいかげんな)、1 つのデータが数行になろうが、コーヒーをこぼしたシミがあろうが、必要な情報だけを簡単に拾い出すことができます。一方、minix のツールの多くは 1 行を処理単位とする時が最も効率良く動作します。どれほど高度な処理ができる人間でも文字がベタ書きされていたのではうんざりします。1 つの文献を「著者」「標題」「出版社」「出版年」「ISBN」からなるデータの集合で表すとします。さて、あなたは (a) と (b) のどちらが扱い易いでしょうか。

(a)
Andrew S.Tanenbaum
OPERATING SYSTEMS DESIGN AND IMPLEMENTATION
Prentice-Hall
1987
0-13-637331-3
Andrew S.Tanenbaum
COMPUTER NETWORKS
Prentice-Hall
1981
0-13-165183-8

(b)
Alan Deikman:UNIX PROGRAMMING ON THE 80286 80386:M & T Publishing, ···
Alfred V.Aho, Brian W.Kernighan, Peter J.Weinberger:The AWK Progra···
Allen I.Holub:COMPILER DESIGN IN C:Prentice-Hall:1990:0-13-155045-4···
Andrew S.Tanenbaum:COMPUTER NETWORKS:Prentice-Hall:1981:0-13-165183···
Andrew S.Tanenbaum:OPERATING SYSTEMS DESIGN AND IMPLEMENTATION:Pren···

よほどのアマノジャクでない限り人間が扱い易いデータ形式は (a) の方だろうと思います。一方、(b) は minix 上のコマンド群が扱いを得意としているものです。

そこで、1 つの文献データの入力は (a) の形式で著者、表題、出版社、出版年、ISBN をそれぞれ 1 行に書き、5 行で 1 つの文献として入力します。そして、それぞれのデータは 1 行以上の空白行で区切ることとします。これを (b) のデータ形式に変換して検索用のデータとして蓄えることにします。このデータをスクリプトで検索し、目的のデータが見つかったならば再度 (a) のような構成にして人間に提示すれば良いことになります。

7.3 Cabinet の部品

それではこのためにどのような部品を用意すればよいでしょうか。データの入力、検索、出力と分けて考えてみます。

まず入力されるデータは通常のテキストファイルですから minix 備え付けのエディタ ed, mined, elle, vi などから好きなものを使うことにし、(a) から (b) のデータ形式に変換するスクリプトと組み合せることにします。検索には正規表現が使える grep を使いましょう。出力は、データ形式 (b) のフィールドがコロン「:」で区切られているので **IFS** を切り換えて各々のフィールド取り出し echo で表示させます。この方針で次のような５つのスクリプトを用意してみました。

1. add - 文献データ追加用のスクリプト
2. upd - 検索用データファイルの更新スクリプト
3. all - 全文献データの表示のスクリプト
4. se - 文献検索スクリプト
5. cab - メニュー処理

この先しばらく 1 段組にします．なぜなら，スクリプトの行が長くて 2 段組みの幅にはとうてい収まらないからです．ご容赦ください．（松田 [2004/05/06]）

7.4 Cabinet のスクリプト

7.4.1 add - 文献データ追加

引数として検索用データに新しい文献データをデータ形式 (a) で書き込んだファイル名を与えることにより追加できます。もし、引数がない場合はコンソールから文献データを１件だけ取り込みます。

```
 1: #!/bin/sh
 2: #   ADD data to the Cabinet File
 3:
 4: : ${CABINET?}
 5: ORG=$CABINET
 6: REC="Auther Title Publisher Year ISBN"
 7:
 8: if test $# -ne 0; then
 9:     for i ; do
10:         if test -f "$i"; then
11:             continue
12:         else
13:             echo "Error: \'$i\' dose not exist."
14:             exit
15:         fi
16:     done
17:     if test -f "$ORG"; then
18:         cp $ORG ${ORG}.bak
19:     fi
20:     for i ; do
21:         echo >> $ORG
22:         cat $i >> $ORG
23:         echo "\'$i\' has been added to the $CABINET Cabinet"
24:     done
25:     upd
26: else
27:     echo "Type in NEW data"
28:     for i in $REC; do
29:         echo -n "$i --> "
30:         read line
31:         eval $i=\"\$line\"
32:     done
33:     echo '----------------------------'
34:     for i in $REC; do
35:         eval echo \"$i\: \$$i\"
36:     done
37:     echo '----------------------------'
38:     echo -n '       Ok? (y/n) '
39:     read line
40:     if test "$line" = y -o "$line" = Y; then
41:         echo >> $ORG
42:         for i in $REC; do
43:             eval echo \"\$$i\"
44:         done >> $ORG
45:         echo; echo "This data has been added to the $CABINET cabinet."
46:         upd
47:     fi
48: fi
```

7.4.2 upd - 検索データの更新

人間による入力データは作業性を考えて複数行で１レコードを構成していますが、minix のコマンドは１行で１つのレコードを構成するものを扱うように設計されています。そのための変換を行ない、検索用のデータファイルの更新を

行ないます。このスクリプトは入力用のスクリプト **add** から呼び出されます。

```
 1 : #!/bin/sh
 2 : #    update cabinet
 3 :
 4 : : ${CABINET?}
 5 : RACK=${CABINET}.items
 6 : TMP=/tmp/_$$
 7 : trap 'rm -f $TMP' 0 2
 8 :
 9 : echo -n "Now, updating $CABINET cabinet. Just a minute, Please!"
10 : lbuf=""
11 : > $TMP
12 : cat $CABINET | while read line; do
13 :     if test "$line" = ""; then
14 :         if test -n "$lbuf"; then
15 :             echo $lbuf >> $TMP
16 :             lbuf=""
17 :         fi
18 :     else
19 :         if test "$lbuf" = ""; then
20 :             lbuf=$line
21 :         else
22 :             lbuf="$lbuf:$line"
23 :         fi
24 :     fi
25 : done
26 : if test -n "$lbuf"; then
27 :     echo $lbuf >> $TMP
28 : fi
29 : echo
30 : sort -fut':' +0.0 -2.0 $TMP > $RACK
```

7.4.3 all - 全文献の表示

80 桁のコンソールでも読み易いように 3 行で 1 件のデータを表示させています。データの出力先は標準出力ですからパイプで more などのページャにつなげばページ単位で停止させられますし、lpr を指定すればプリンタに出力できます。

```
 1 : #!/bin/sh
 2 : #   LIST all of the entries in the Cabinet File
 3 :
 4 : : ${CABINET?}
 5 : RACK=${CABINET}.items
 6 :
 7 : if test ! -s "$RACK"; then
 8 :     echo 'Empty Cabinet.'
 9 :     exit
10 : fi
11 : echo "  `wc -l < $RACK` item(s) in your Cabinet."
12 :
13 : IFS=:
14 : cat $RACK | while read A B C D E; do
15 :     echo; echo $B
16 :     echo $A
17 :     echo "$C,  $D,  ISBN $E"
18 : done
19 : echo
```

7.4.4 se - 文献の検索

引数として与えられた文字列の条件を満たす文献すべてを探し出します。検索キーワード文字列には正規表現が使えます。

```
 1: #!/bin/sh
 2: #   SEarch data in the Cabinet File
 3:
 4: : ${CABINET?}
 5: RACK=${CABINET}.items
 6: TMP=/tmp/_$$
 7:
 8: trap 'rm $TMP; echo; exit' 0 2
 9:
10: if test $# -eq 0 -o ! -f "$RACK"; then
11:     exit
12: fi
13:
14: echo -n "looking for \'$*\' .."
15: grep "$*" $RACK > $TMP
16:
17: if test -s "$TMP"; then
18:     echo "    `wc -l < $TMP` item(s)"
19:     IFS=:
20:     cat $TMP | while read A B C D E; do
21:         echo; echo $B
22:         echo $A
23:         echo "$C,  $D,  ISBN $E"
24:     done
25:     echo
26: else
27:     echo " Sorry, I can\'t find in the $CABINET Cabinet"
28: fi
```

7.4.5 cab - Cabinet メニュー

add, **upd**, **all**, **se** は部品として作られており、単独で走らせることができます。初心者にとってはメニューの方がなじみやすいでしょうから、メニュー画面で数字で指定することにより 5 つの作業ができるようにしてみました。

```
 1: : ${CABINET=Book}
 2: export CABINET
 3:
 4: OUTPUT=more
 5: OUT=Display
 6: RACK=items
 7:
 8: trap 'continue' 2
 9:
10: if test $# -ne 0; then
11:     if test -d $1;  then
12:         CABINET=$1
13:     else
14:         exit
15:     fi
16: fi
17:
18: while true ;do
19:     echo; echo; echo -n "
20:     $CABINET Cabinet  .. `wc -l < ${CABINET}.items` item(s)
21:       ================================================
22:     Would you like to:
```

```
23 :
24 :         1) Search data in the Cabinet
25 :         2) Add data to the Cabinet
26 :         3) List all of the Cabinet
27 :         4) Change output (current: $OUT)
28 :         5) Quit
29 :       ================================================
30 :                 SELECT (1-5): "
31 :     read command
32 :     echo
33 :     case "$command" in
34 :         1 | s)  echo -n '[SEARCH] Enter Keyword: '
35 :             read line
36 :             if test ! -z "$line"; then
37 :                 se "$line" | "$OUTPUT"
38 :                 if test "$OUT" = Display; then
39 :                 echo '>>> Hit ENTER to continue <<<'
40 :                 read line
41 :                 fi
42 :             fi;;
43 :
44 :         2 | a)  echo -n '[ADD] Enter filename: '
45 :             read line
46 :             if test -z "$line"; then
47 :                 add
48 :             else
49 :                 add "$line"
50 :             fi;;
51 :
52 :         3 | l)  echo '[ALL]'
53 :             all | $OUTPUT;;
54 :
55 :         4 | c)  if test "$OUT" = Display; then
56 :                 OUTPUT=lpr; OUT=Printer
57 :             else
58 :                 OUTPUT=more; OUT=Display
59 :             fi;;
60 :
61 :         5 | q)  exit;;
62 :         *)  echo "??? \'$command\'";;
63 :     esac
64 : done
```

7.5 Cabinet の使い方

Cabinet は初心者の学習用として作った文献検索用のシェルスクリプトです。使用方法についてはスクリプトを読んでいただければ一目瞭然ですし、それがまたシェルスクリプト理解への近道でしょう。さらに新しい機能を追加するなどして、実際に手を加えてみてください。それが思い通りに走ったときには楽しさも倍増すること請け合いです。とはいえ、「とにかく遊んでみたい」というせっかち屋さんのために簡単な説明を用意します。

Cabinet 本体はデータを追加・更新を行なう **add**、**upd**、検索を担当する **se** そして全データをダンプする **all** の４つのスクリプトから構成されています。これらはそれぞれがが独立して走るようになっており、コマンドラインから直接起動させることができます。しかし、コマンドラインからは柔軟な使い方ができる反面、シェル変数の設定などをユーザが直接で行わなければなりません。まったくの初心者には取付きにくいかもしれません。

このために **Cabinet** はメニュー処理のためのスクリプト **cab** が用意されています。シェルについてまったくの初心者はメニューで慣れてから移った方が無難でしょう。

7.5.1 メニューの起動

メニュー処理用のスクリプト **cab** を走らせると **Cabinet** の操作メニューが表示されます。

$ cab

とタイプして起動した場合は親シェル（ほとんどの場合はログインシェルでしょう）からコピーされてきた変数 **CABINET** の内容で指定されたものを操作の対象とします。もし、この時に親シェルが **CABINET** という変数を export していなかったり、export していても中身が空の場合は **Book** を操作対象として立ち上がります。このメニュー画面で操作できる対象は

1. 親シェルが export した変数 CABINET で指定されたもの
2. Book

の順となります。メニュースクリプト **cab** が起動されると次のような画面が表れます。

```
Cabinet = Book  68 items
================================================
Would you like to:

      1) Search data in the Cabinet
      2) Add data to the Cabinet
      3) List all of the Cabinet
      4) Change output (current: Display)
      5) Quit
================================================
                                SELECT (1-5):
```

一番上に表示されている ”Book 68 items” とは現在開かれている（シェル変数 CABINET で指定されている）キャビネットの名称とその中にあるデータ件数です。この Book を対象に検索、追加、表示の操作を行います。

(1) データの検索

現在開かれているキャビネットから指定されたキーワードを含む項目を表示します。”1” を選びリターンキーを押すと

[SEARCH] Enter Keyword:

と探すためのキーワードを聞いてきます目的のものを指定してください。指定するキーワードには一般の語句はもちろん、正規表現を用いることもできます。例えば ”IBM PC” というタイトルのついた文献を探したければ

[SEARCH] Enter Keyword: IBM PC

とタイプし（下線部分）リターンキーを打ちます。しばらくすると見つかった項目数とそれらの内容が表示されます。もし、表示する内容が１画面に納まらない場合は more がポーズをかけますので、スペースで１画面、リターンで１行先に進みます。

(2) データの追加

現在開かれているキャビネットに新しいデータを追加します。追加は事前にテキストエディタなどで作っておいたファイルの内容を追加する方法と１項目だけ手作業で入力する方法があります。どちらの場合も ”2” を選択します。

[ADD] Enter filename:

追加するデータを収めたファイル名を尋ねられますので該当するファイル名を指定してください。ファイル名はパスを含むこともできますし、スペースで区切って複数のファイルを指定することもできます。

もし、指定したファイルが存在しない場合はエラーとなりデータの追加は行われません。この時に追加するデータの内容についてはまったく検査されませんので、間違った指定をすると文献データにビデオやＣＤのデータ、最悪の場合にはプログラムソースなどを追加してしまうこともあります。また、同じ文献のデータでもフィールドの列びが既存のデータと一致していないとトンチンカンなものになってしまいます。

１つの物件のレコード構造についてはすべて使う人間の責任に任されています。

ここで使用している文献データのレコード構造は 「7.2 データ形式」(p.22) で説明したものです。例えば

[ADD] Enter filename: b00 ../b002

と指定するとカレントディレクトリにある b001 というファイルと ../b002 というファイルの内容が文献データに追加されます。

もし、ここでファイル名を入力せずにリターンキーのみを押すと手作業で１件だけデータを追加できます。入力内容は既存の検索用ファイルのレコード構造と一致させてください。手作業の入力は概ねつぎのようになります。

```
[ADD] Enter filename: <- リターンのみ
Type in NEW data
Auther -> Andrew S.Tanenbaum
Title -> OPERATING SYSTEM DESIGN AND IMPLEMENTATION
Publisher -> Prentice-Hall
Year -> 1988
ISBN -> 0-13-637331-3
```

著者、タイトル、出版社、年、ISBN の４項目の入力が完了すると、入力されたデータを表示し、確認を求めてきます。

```
----------------------------
Auther: Andrew S.Tanenbaum
Title: OPERATING SYSTEM DESIGN AND IMPLEMENTATION
Publisher: Prentice-Hall
Year: 1988
ISBN: 0-13-637331-3
----------------------------
                    Ok? (y/n)
```

ここで入力データに間違いがなければ ”y” で答えてください。”y” の入力でデータが追加され、それ以外のキーならば入力されたデータが捨てられます。

(3) 全項目の表示

現在開かれているキャビネットに収められているすべてのデータを表示します。ここでは ”3” を選択するだけです。画面に表示するときは more が１画面ごとにポーズを入れてきます。「4) Change output」で出力をプリンタ (lpr) に切り換えているならばプリンタに連続出力されます。

(4) 出力の切り替え

検索されたデータと全項目を表示するときの出力先を切り換えます。”4” を選択するたびにディスプレイとプリンタをスイッチし、現在の出力先は

4) Change output(current: Display)

として表示されています。

(5) 終了

メニューを終了し、シェルのコマンドラインに戻ります。

7.5.2 コマンドラインからの使用法

すでに説明したメニューからの操作は **cab** がそれぞれスクリプトを呼び出して実現しています。それらをコマンドラインから呼び出して直接使うこともできます。これらのスクリプトは実行されると操作対象とするキャビネットを決めるために必ずシェル変数 **CABINET** の内容を参照します。もし、この変数が export されていなかったり内容が空の場合は正常な処理は期待できません。文献データを処理したいのでしたら最初にログインシェルのプロンプトからシェル変数 **CABINET** に **Book** を設定します。

$ CABINET=Book; export CABINET

(1) 検索

検索をするスクリプトは **se**(SEarch) です。これに検索させたいキーワードを引数として与えます。例えば ”IBM PC” というものをキーワードとして与えたい場合には

$ se 'IBM PC'

とします。この時キーワードとして **se** に渡す文字列をシェルから保護するために必ず単一引用符で囲みます。また、

$ se '.*NIX' | more

などと正規表現を使ったり、検索結果をパイプに流すこともできます。

(2) 追加

add というスクリプトが担当します。追加したいデータが入っているファイル名を引数として与えます。例えば /user/mybooks というファイルに追加したいデータが入っているとすると

$ add /user/mybooks

とします。引数として与えるファイル名は１つに限りませんが、指定したファイルすべてが見つからない場合はエラー中断しますのでデータの追加は行われません。

もし、引数を与えなかった場合は１件のみを手作業で入力するように動作します。手作業入力の具体的な例は「メニューからの操作」部分を参照してください。

(3) 全表示

スクリプト **all** を走らせると登録されているすべての項目を表示します。すべての項目をプリンタに送りたい場合は

```
$ all | lpr
```

とすれば良いでしょう。画面で見たいのでしたら more にパイプでつないでください。

(4) 更新

人間が入力したデータからキャビネットを構成するスクリプト類が操作しやすい形に変換し、検索用のデータを更新します。引数なしで

```
$ upd
```

とするだけです。なお、このスクリプトは最初に1度だけ使うものです。あとは **add** でデータを追加すると自動的に **upd** が呼び出されます。

7.6 Cabinet の拡張

Cabinet には不必要になったデータを削除するためのスクリプトがありません。腕試しにチャレンジしてみてはいかがでしょうか。大まかには

1. 削除したい項目を入力させる（引数として指定する）
2. それに一致する項目を見つけ
3. オペレータの確認を得たのちに
4. 該当項目を削除する
5. できればその際に元のデータのバックアップを取る

という手順行えば良いと思います。そのためにはこの研修用のディスクにある限られたツールをどんなオプションで、どのよう組み合わせれば良いか？ 等など、興味は尽きません。また、この Cabinet は文献ファイルをアクセスする時に排他制御も行なっていませんのでこちらも試してみてください。

参考文献

[1] 平林浩一/平林小枝子
UNIX のバックグラウンド. プロセッサ No.64 Aug 1990. 技術評論社

[2] S.R. Bourne/三好・木下訳
UNIX システム. マイクロソフトウェア

[3] ©1987 Andrew S.Tanenbaum, Prentice Hall
minix 1.6.24B, shell ソース・リスト

7.7 Cabinet の文献リスト

[1] 4.3BSD UNIX Operating System
Samuel J.Leffer/Marshall K.McKustick/Micael J.Karels/John S.Quarterman
Addison-Wesley, 1989 ISBN 0-201-06196-1

[2] ADVANCED UNIX PROGRAMMING
Mark J.Rochkind
Prentice-Hall, 1985 ISBN 0-13-011800-1
「UNIX システムコール・プログラミング」, ASCII, ISBN 4-87148-360-X

[3] BCPL and C
Glyn Emery
Blackwell Science Publications, 1986 ISBN 0-632-01571-3

[4] BCPL the language and its compiler
Martin Richards/Colin Whitby-Strevens
Cambridge University Press, 1980 ISBN 0-521-28681-6

[5] BISON
Charles Donnelly and Richard Stallman
FSF, 1988

[6] COMPILER DESIGN AND CONSTRUCTION Tools and Techniques With C and Pascal
Arthur B.Pyster
Van Nostrand Reinhold, 1988 ISBN 0-442-27536-6

[7] COMPILER DESIGN IN C
Allen I.Holub
Prentice-Hall, 1990 ISBN 0-13-155045-4

[8] Compilers Principles, Techniques, and Tools
Alfred V.Aho/Ravi Sethi/Jefffrey D.Ullman
Addison-Wesley, 1986 ISBN 0-201-10088-6

[9] Computer Games I
David N.L.Levy ed.
Springer-Verlag, 1987 ISBN 0-387-96496-7

[10] Computer Games II
David N.L.Levy ed.
Springer-Verlag, 1987 ISBN 0-387-96609-9

[11] COMPUTER NETWORKS
Andrew S.Tanenbaum
Prentice-Hall, 1981 ISBN 0-13-165183-8

[12] Data Structures and C Programs
Christopher J.Van Wyk
Addison-Wesley, 1988 ISBN 0-201-16116-8

[13] DOCUMENT FORMATTING & TYPESETTING ON THE UNIX SYSTEM Vol.1
Narain Gehani
Silicon Press, 1988 ISBN 0-9615336-2-5

[14] DOCUMENT FORMATTING & TYPESETTING ON THE UNIX SYSTEM Vol.2
Narain Gehani
Silicon Press, 1988 ISBN 0-9615336-3-3

[15] EXPLORING THE UNIX SYSTEM
Stephen G.Kochan/Patrick H.Wood
Hayden Book Company, 1984 ISBN 0-8104-6268-0

[16] GDB Manual
Richard M.Stallman
FSF, 1988

[17] GNU Emacs Manual
Richard Stalllman
FSF, 1987

[18] GNU Make
Richard M.Stallman/Roland McGrath
FSF, 1989

[19] INTERNETWORKING WITH TCP/IP PRINCIPLES, PROTOCOLS, AND ARCHITECTURE
Douglas Comer
Prentice-Hall, 1988 ISBN 0-13-470188-7

[20] INTRODUCTION TO COMPILER CONSTRUCTION WITH UNIX
Axel T.Schreiner/H.George Friedman Jr.
Prentice-Hall, 1985 ISBN 0-13-474396-2

[21] LIFE WITH UNIX
Don libes/Sandy Ressler
Prentice-Hall,
「Life with UNIX」, ASCII, ISBN 4-7561-0783-4

[22] Managing UUCP and Usenet
Tim O'Reilly/Grace Todino
O'Reilly & Associates, Inc, 1989 ISBN 0-937175-09-9
「UUCP システム管理」, ASCII, ISBN 4-7561-0087-2

[23] MINIX FOR THE IBM PC,XT,AND AT REFERENCA MANUAL
Andrew S.Tanenbaum
Prentice-Hall, 1988 ISBN 0-13-584400-2

[24] Modern Operating Systems
Andrew S.Tanenbaum
Prentice-Hall, 1992 ISBN 0-13-595752-4

[25] More Programming Pearls
John Bentley
Addison-Wesley, 1988 ISBN 0-201-11889-0

[26] OPERATING SYSTEM DESIGN THE XINU APPROACH
Douglas Comer

Pretice-Hall, 1984 ISBN 0-13-637339-1

[27] OPERATING SYSTEM DESIGN THE XINU APPROACH, MACINTOSH EDITION
Douglas Comer
Pretice-Hall, 1989 ISBN 0-13-638529-X

[28] OPERATING SYSTEM DESIGN THE XINU APPROACH, P.C.EDITION
Douglas Comer
Pretice-Hall, 1988 ISBN 0-13-638180-4

[29] OPERATING SYSTEM DESIGN-VOLUME II INTERNETWORKING WITH XINU
Douglas Comer
Prentice-Hall, 1987 ISBN 0-13-637646-0

[30] OPERATING SYSTEMS DESIGN AND IMPLEMENTATION
Andrew S.Tanenbaum
Prentice-Hall, 1987, ISBN 0-13-637331-3

[31] Practical C Programming Qualine,S
O'reilly & Associates, 1991, ISBN 0-937175-65-X

[32] Practical UNIX Security
Simson Garfinkel/Gene Spefford
O'reilly & Associates, 1991, ISBN 0-937175-72-2
「UNIX セキュリティ」, ASCII, ISBN 4-7561-0274-3

[33] PREPARING DOCUMENTS WITH UNIX
Constance C.Brown/Jack L.Falk/Richard D.Sperline
Prentice-Hall, 1986 ISBN 0-13-699976-X

[34] PROGAMS AND DATA STRUCTURES IN C
Leendert Ammeraal
John Willey & Sons, 1987 ISBN 0-471-91751-6

[35] Programming Pearls
John Bentley
Addison-Wesley, 1986 ISBN 0-201-10331-1

[36] Programming the UNIX System
M.R.M.Dunsmuir/G.J.Davis
Macmillan, 1985 ISBN 0-333-37156-9

[37] PROGRAMS AND DATA STRUCTURES IN C
Leendert Ammerael
John Weiley, 1987 ISBN 0-471-91751-6
「C-データ構造とプログラム」, オーム社, ISBN 4-274-07552-4

[38] Solutions in C
Rex Jaeschke
Addison-Wesley, 1986 ISBN 0-201-15042-5

[39] STRUCTURED COMPUTER ORGANUZATION
Andrew S.Tanenbaum
Prentice-Hall, 1984 ISBN 0-13-854489-1

[40] Termcap
Richard M.Stallman
FSF, 1988

[41] Termcap & Terminfo
John Strang/Tim O'Reilly/Linda Mui
O'Reilly & Associates, 1989 ISBN 0-93717522-6

[42] Texinfo
Richard M.Stallman/Robert J.Chassel
FSF, 1988

[43] TEXT PROCESSING AND TYPESETTING WITH UNIX
David Barron/Mike Rees
Addison-Wesley, 1987 ISBN 0-201-14219-8

[44] The AWK Programming Language
Alfred V.Aho/Brian W.Kernighan/Peter J.Weinberger
Addison-Wesley, 1988 ISBN 0-201-07981-X
「プログラミング言語 AWK」, トッパン, ISBN 4-8101-8008-5

[45] THE BELL SYSTEM TECHNICAL JOURNAL
Bell System
JULY 1978, 1978

[46] THE C PROGRAMMING LANGUAGE
Brian W.Kernighan/Dennis M.Ritchie
Prentice-Hall, 1978 ISBN 0-13-110163-3

[47] The Design of the UNIX Operating System
Maurice.J.Bach
Prentice-Hall, 1987 ISBN 0-13-201799-7
「UNIX カーネルの設計 bit 別冊」, 共立出版

[48] The GAWK Mannual
Diane Barlow Close/Arnold D.Robbins/Paul H.Rubin/Richard Stallman
FSF, 1989

[49] The Kornshell Command and Programming Language
Morris I.Bolsky/David G.Korn
Prentice-Hall, 1989 ISBN 0-13-516972-0

[50] THE UNIX OPERATING SYSTEM
Kaare Christian
Weily-Interscience, 1983 ISBN 0-471-87542-2

[51] THE UNIX operating system BOOK
Mike Banahan/Andy Rutte
John Wiley & Sons, 1983 ISBN 0-471-89676-4

[52] The UNIX Shell Programming Language
Rod Manis/Marc H.Meyer
Howard W.Sams, 1986 ISBN 0-672-22497-6

[53] THE UNIX System
S.R.Bourne
Addison-Wesley, 1983 ISBN 0-201-13791-7
「UNIX システム」, マイクロソフトウェア

[54] THE UNIX SYSTEM V ENVIRONMENT
Stephen R.Bourne
Addison-Wesley, 1987 ISBN 0-201-18484-2

[55] THE UNIX TEXT PROCESSING SYSTEM
Kaare Christian
John Wiley & Sons, 1987 ISBN 0-471-85581-2

[56] troff Typesetting for UNIX Systems
Sandra L.Emerson/Karen paulsell
Prentice-Hall, 1987 ISBN 0-13-930959-4

[57] UNIX FOR SUPER-USERS
Eric Foxley
Addison-Wesley, 1985 ISBN 0-201-14228-7

[58] UNIX NROFF/TROFF A User's Guide
Kevin P.Roddy
Holt,Rinehart and winston, 1987 ISBN 0-03-000167-6

[59] UNIX Papers for UNIX Developpers and Power Users
Mitchell Waite ed.
Howard W.Sams & Company, 1987 ISBN 0-672-22578-6

[60] UNIX programmer's manual vol 1
Bell Lab.
Holt,Rinehart and Winston, 1983 ISBN 0-03-061742-1

[61] UNIX programmer's manual vol 2
Bell Lab.
Holt,Rinehart and Winston, 1983 ISBN 0-03-061743-X

[62] UNIX PROGRAMMING ON THE 80286 80386
Alan Deikman
M&T Publishing, Inc, 1989 ISBN 1-55851-060-5

[63] UNIX PROGRAMMINIG ENVIRONMENT
Brian W.Kernighan/Rob Pike
Prentice-Hall, 1984 ISBN 0-13-937681-X
「UNIX プログラミング環境」, ASCII, ISBN 4-87148-351-7

[64] UNIX Syatem Readings and Applications Volume 2
AT&T Bell Laboratories
Prentice-Hall, 1987 ISBN 0-13-939845-7

[65] UNIX SYSTEM ADMINISTRATION
David Fiedler/Bruce H.Hunter
Hayden Books, 1986 ISBN 0-8104-6289-3

[66] UNIX SYSTEM PROGRAMMING
Keith Haviland Ben Salama
Addison-Wesley, 1987 ISBN 0-201-12919-1

[67] UNIX System Software Readings
AT&T Unix Pacific Co.,Ltd
Prentice-Hall, 1988 ISBN 0-13-938358-1

[68] UNIX Sytem Readings and Applications Volume 1
AT&T Bell Laboratories
Prentice-Hall, 1987 ISBN 0-13-938532-0

[69] UNIX TEXT PROCESSING
Dale dougherty/Tim O'Reilly
HAYDEN BOOKS, 1987 ISBN 0-672-46291-5

[70] Using UUCP and Usenet
Tim O'Reilly/Grace Todino
O'Reilly & Associates, Inc, 1987 ISBN 0-937175-10-2
「UUCP 入門」, ASCII, ISBN 4-7561-0280-8

[71] Writeing UNIX Device Drivers
George Pajari
Addison-Wesley, 1992 ISBN 0-201-523774-4

[72] WRITING A UNIX DEVICE DRIVER
Janet I.Egan/Thomas J.Teixeita
John Wiley & Sons, 1988 ISBN 0-471-62811-5