グループコマンドを使う

||リストは、その左右にある、それぞれ1つのパイプラインにのみにかかります。これは、if文が、パイプラインではなくリストにかかるのとは異なります。したがって、||リストの右側で複数のパイプライン(たとえば複数のコマンド)を実行したい時は、**リストB**のようにグループコマンドの{ }を使って、パイプラインを1つのコマンドにまとめる必要があります。

リストB 右側のパイプラインをグループコマンドにした例

```
test -f file1 || {  ……file1が存在すれば真になる
  echo 'file1が存在しません'  ……偽の場合、エラーメッセージを表示
  exit 1  ……エラーで終了
}  ……グループコマンドの終了
```

参照

if文(p.43)　　グループコマンド(p.74)

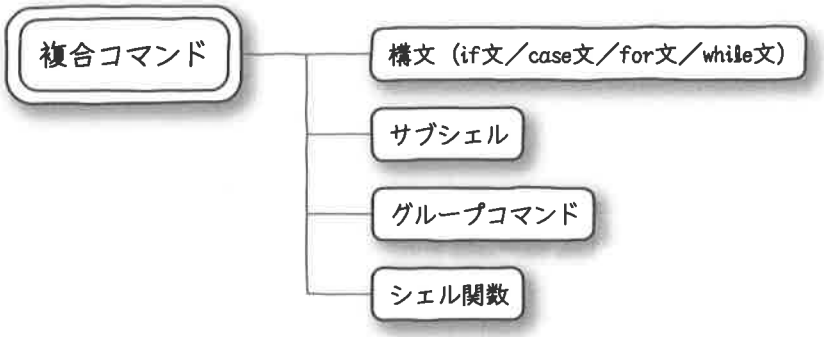
>第4章
複合コマンド

シェル文法における複合コマンド

シェル文法では、if文やfor文などの**構文**や**サブシェル**、**グループコマンド**、**シェル関数**などが**複合コマンド**と解釈されます(**図A**)。

条件分岐を行うif文やcase文、ループを行うfor文やwhile文を使えば基本的なプログラム構造を記述できるでしょう。また、一定の処理をシェル関数としてまとめ、これを適宜呼び出して使用することもできます。

図A 複合コマンド



- 構文➡本書ではif文／case文／for文／while文のことをまとめて構文と呼んでいます
- サブシェル➡リストを( )で囲んだものです
- グループコマンド➡リストを{ }で囲んだものです
- シェル関数➡リストを関数にまとめたものです

if文

Linux (bash)
FreeBSD (sh)
Solaris (sh)

条件判断によってプログラムを分岐する

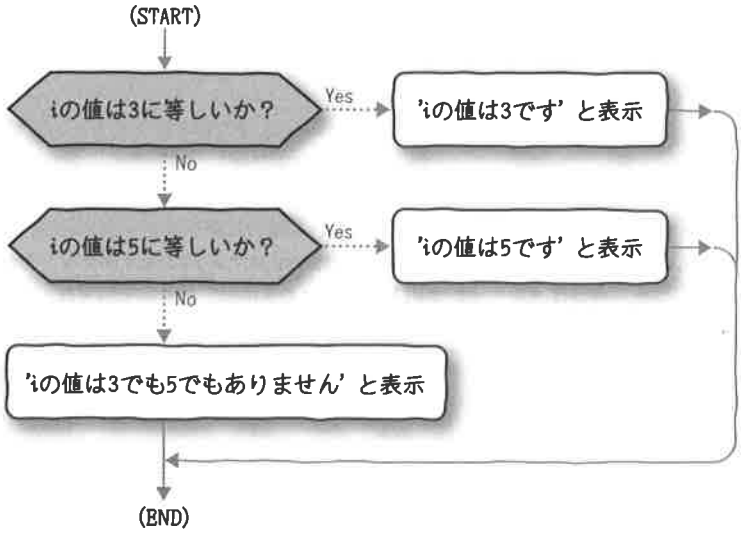
書式 **if** リスト; **then** リスト; [**elif** リスト; **then** リスト;]...
[**else** リスト;] **fi**

リストの右端がすでに改行などで終端されている場合、その右の ; は不要

例

```
if [ "$i" -eq 3 ] ……"$i"と3を数値として比較
then
  echo 'iの値は3です' ……echoコマンドでメッセージを表示
elif [ "$i" -eq 5 ] ……"$i"と5を数値として比較
then
  echo 'iの値は5です' ……echoコマンドでメッセージを表示
else ……ifやelifの条件が満たされなかった場合
  echo 'iの値は3でも5でもありません' ……echoコマンドでメッセージを表示
fi ……if文の終了
```

図



基本事項

if文は、リストの実行結果が**真**か**偽**かによって分岐する構文です。まずifの直後のリストが実行され、その結果が真(終了ステータスが「0」)である場合には次に続くthenの直後のリストが実行され、これでif文が終了します。

ifの直後のリストが偽(終了ステータスが「0」以外)の場合には、次のelifの直後のリストが実行され、これが真である場合はelif内のthenの直後のリストが実行され、これでif文が終了します。ifとすべてのelifの直後のリストが偽であった場合は、elseの直後のリストが実行されます。

elifやelseの部分は、必要なければ省略することができます。また、必要ならば複数のelifをつけることができます。

(if文自体の)終了ステータス

if文全体が1つの複合コマンドを構成しますが、その終了ステータスは次のようになります。thenまたはelseの直後のリストが実行された場合は、そのリストの終了ステータスがif文自体の終了ステータスになります。thenまたはelseの直後のリストが実行されなかった場合は、if文自体の終了ステータスは「0」になります[注1]。

解説

多くの場合、ifの直後のリストには[が記述されます。この[はif文の文法とは直接関係なく、[という名前の独立したコマンドです。[コマンドはtestコマンドの別名であり、testコマンドによって実際の条件判断が行われます。

このように、シェルスクリプトでは、数値や文字列の比較やファイルの存在チェックといった条件判断を、シェル自身では行わず、testコマンドに任せています。つまりif文は、単にtestコマンドの終了ステータスが真か偽かによって条件分岐しているにすぎないのです。

冒頭の例では、シェル変数"$i"の値を-eqによって「3」や「5」といった値と比較していますが、これらの詳細についてはtestコマンドの範疇になります。

testコマンド以外のコマンドを使う方法

ifの直後のリストとしてtestコマンド以外のコマンドを使うこともちろんできます。たとえばcmpコマンドを使って2つのファイルの内容を比較し、それらが一致しているかどうかで分岐するには**リストA**のようにします。

cmpコマンドは、2つのファイルが一致している場合に終了ステータス「0」を返すため、これをif文の条件判断に利用できるのです。なお、cmpコマンドに-sオプションを付け、cmpコマンド自体から余分なメッセージが出力されないようにしていることにも注意してください。

リストA ifの直後のリストとしてcmpコマンドを使った例

```
if cmp -s file1 file2 ……file1とfile2の内容を比較(-sでメッセージ抑制)
then
  echo 'file1とfile2の内容は同じです' ……一致していた場合のメッセージを表示
else
  echo 'file1とfile2の内容は異なります' ……相違していた場合のメッセージを表示
fi
```

注1　if文自体の終了ステータスを実際に使用する場面はあまりありません。

if [...]のスタイルに統一したい場合

testコマンド以外のコマンドで条件判断する場合でも、コマンド実行直後に特殊パラメータ$?を参照することにより、if [...]のスタイルで書くことができます。**リストB**にその例を示します。

ifの直後に複数のコマンドを記述してもよい

一般にリストは複数のコマンドを含んでもかまわないため、**リストC**のようにifの直後のリストにcmpと[の両方を含めて書くこともできます[注2]。

ifの条件判断を逆にするには❶

ifの条件判断を逆にしたい場合は、いったん$?の値をtestコマンドで受けて、testコマンド上で条件判断を逆にするようにします(**リストD**)。

ifの条件判断を逆にするには❷

elseを使ってifの条件判断を逆にする方法もあります(**リストE**)。この場合、thenの直後のリストは:コマンドにして、何も実行されないようにします。

リストB if [...]のスタイルで統一した例

```
cmp -s file1 file2 ……file1とfile2の内容を比較(-sでメッセージ抑制)
if [ $? -eq 0 ] ……cmpコマンドの終了ステータスが0かどうかチェック
then
  echo 'file1とfile2の内容は同じです' ……一致していた場合のメッセージを表示
fi
```

リストC ifの直後のリストにcmpと[の両方を記述

```
if
  cmp -s file1 file2 ……file1とfile2の内容を比較(-sでメッセージ抑制)
  [ $? -eq 0 ] ……cmpコマンドの終了ステータスが0かどうかチェック
then
  echo 'file1とfile2の内容は同じです' ……一致していた場合のメッセージを表示
fi
```

リストD ifの条件判断を逆にした例

```
if cmp -s file1 file2; [ $? -ne 0 ] ……$?の値が0でない場合に真になる
then
  echo 'file1とfile2の内容は異なります' ……相違していた場合のメッセージを表示
fi
```

注2　リストの項(p.35)も合わせて参照してください。

リストE elseを使って条件判断を逆にした例

```
if cmp -s file1 file2 ……file1とfile2の内容を比較（-sでメッセージ抑制）
then
  : ……一致していた場合は何もしない:コマンドを実行
else
  echo 'file1とfile2の内容は異なります' ……相違していた場合のメッセージを表示
fi
```

パイプラインの否定演算を使う方法

Warning

Linux (bash) ○　FreeBSD (sh) ○　Solaris (sh) ×

この方法には制限があります。

bash(またはFreeBSDのsh)では、パイプラインの先頭に！を書いて、終了ステータスを反転することができるため、**リストF**の例のように記述することもできます。なお、！とコマンドとの間にはスペースが必要です。

リストF パイプラインの否定演算を使った例

```
if ! cmp -s file1 file2 ……cmpコマンドの終了ステータスを!で反転して条件判断
then
  echo 'file1とfile2の内容は異なります' ……相違していた場合のメッセージを表示
fi
```

パイプを使った条件判断

ifの直後のリストに、パイプを含めることももちろんできます。**リストG**の例は、whoコマンドの出力の中に、ユーザ名「guest」が含まれているかどうかによって条件分岐しています。ここではgrepによる判定を正確にするため、その引数は「'^guest\>'」としています。また、grepコマンド自体の出力が表示されないように、その標準出力を**/dev/null**にリダイレクトしていますが、代わりにgrepに-qオプションを付けてもかまいません。

if文のネスティング

thenまたはelse(またはelif)の直後のリストの中に、別のif文を記述することにより、if文をネスティングする(入れ子にする)ことができます(**リストH**)。ネスティングの深さがわかりやすいようにインデントを行うとよいでしょう。

elifを使ったほうがよい場合

リストIの例のような内容の場合、if文のネスティングよりも、**リストJ**のようにelifを使って記述したほうが簡潔になります。

リストG ifの直後のリストにパイプを使った例

```
if who | grep '^guest\>' > /dev/null ……whoの出力にguestの行があると真になる
then
  echo 'guestがログイン中です' ……guestがいた場合のメッセージを表示
fi
```

リストH if文のネスティング

```
if [ "$i" -eq 3 ] ……元のif文の開始
then
  if [ "$j" -eq 5 ] ……if文のthen中にネスティングされたif文の開始
  then
    echo 'i＝3かつj＝5です'
  else
    echo 'i＝3かつj≠5です'
  fi ……ネスティングされたif文の終了
else ……元のif文のelse
  if [ "$j" -eq 5 ] ……if文のelse中にネスティングされたif文の開始
  then
    echo 'i≠3かつj＝5です'
  else
    echo 'i≠3かつj≠5です'
  fi ……ネスティングされたif文の終了
fi ……元のif文の終了
```

リストI if文のネスティングが無駄な例

```
if [ "$i" -eq 3 ] ……元のif文の開始
then
  echo 'i＝3です'
else ……元のif文のelse
  if [ "$i" -eq 5 ] ……if文のelse中にネスティングされたif文の開始
  then
    echo 'i＝5です'
  fi ……ネスティングされたif文の終了
fi ……元のif文の終了
```

リストJ elifを使って記述した例

```
if [ "$i" -eq 3 ] ……if文の開始
then
  echo 'i＝3です'
elif [ "$i" -eq 5 ] ……elifを使って別の条件を記述
then
  echo 'i＝5です'
fi ……if文の終了
```

注意事項

ifと[の間にはスペースが必要

ifと[の間には**スペース**または**改行**などの区切り文字が必要です。「if[」のようにスペースを入れないで書くと、「if[」という名前のコマンドを実行するものとみなされ、コマンドが見つからないというエラーになります。

if～thenを1行に書く場合はthenの前に;が必要

if～thenを1行に書く場合は、testコマンドなどの最後に;を付けてリストを終端する必要があります。;がないと、thenという文字列がtestコマンドの最後の引数であると解釈されて、if文全体が正しく認識されません。

○正しい例

```
if [ "$i" -eq 3 ]; then
  echo 'iの値は3です'
fi
```

×誤った例

```
if [ "$i" -eq 3 ] then
  echo 'iの値は3です'
fi
```

thenやelseの直後に;を付けてはいけない

thenやelseの直後の改行は単なる区切り文字としての改行であり、リストの終端ではありません。したがって、ここに;を入れると文法エラーになります。とくに、if文全体を1行で書く場合に注意してください。

○正しい例

```
if [ "$i" -eq 3 ]; then echo 'iの値は3です'; fi
```

×誤った例

```
if [ "$i" -eq 3 ]; then; echo 'iの値は3です'; fi
```

リストがない場合は:が必要

thenやelseの直後には必ずリストを記述する必要があります。何も実行したくない場合(デバッグなどで一時的にコメントアウトする場合も含む)には、リストとして:コマンドを記述するようにします。

Memo

- if文の最後のfiは、ifのスペルを逆にしたものです。
- if文の代わりに&&リストや||リストを使って条件分岐を行うこともできます。
- bashでは、testコマンドの[ ]の代わりに、[[ ]]を使った条件式の評価や(( ))を使った算術式の評価を用いることもできます。

参照

test(p.147)　リスト(p.35)　パイプライン(p.32)　特殊パラメータ$?(p.173)
:コマンド(p.87)　&&リスト(p.37)　||リスト(p.39)　条件式の評価[[ ]](p.83)
算術式の評価(( ))(p.80)

case文

Linux (bash)　FreeBSD (sh)　Solaris (sh)

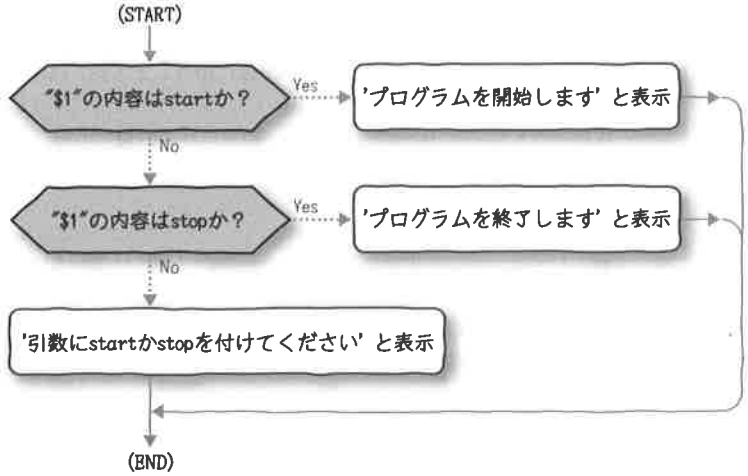
文字列をパターンごとに場合分けしてプログラムを分岐する

書式 **case** 文字列 **in** [(パターン [| パターン]...) リスト;;]... **esac**

例

```
case $1 in ……シェルスクリプトの第1引数によって条件分岐
  start) ……引数がstartだった場合
    echo 'プログラムを開始します' ……プログラム開始のメッセージを表示
    ;; ……リストの終了
  stop) ……引数がstopだった場合
    echo 'プログラムを停止します' ……プログラム停止のメッセージを表示
    ;; ……リストの終了
  *) ……引数がstartでもstopでもなかった場合
    echo '引数にstartかstopを付けてください'…エラーメッセージの表示
    ;; ……リストの終了
esac ……case文の終了
```

図



基本事項

case文は、与えられた文字列をパターンと比較して分岐する構文です。まずcaseの直後に書かれた文字列が各パターンと順に比較され、一致したパターンがあった場合はそのパターンの直後のリストを実行し、これでcase文を終了します。

パターンとリストのセットは、必要なだけいくつでも記述することができます。各リストは;;で終了する必要がありますが、最後のリストだけは;;を省略することもできます。

パターンには、**パス名展開**の特殊文字を使って、複数の文字列に一致するようにすることもできます。ただし、ファイル名のパス名展開とは一部異なり、文字列の先頭の「.」や文字列中の「/」を特別扱いしません[注3]。

また、パターンを|で区切ってOR条件のパターンを記述することもできます。

なお、文字列またはパターンにパラメータ展開やコマンド置換が行われた場合でも、その結果に対して単語分割は行われません[注4]。このため、文字列としてシェル変数等を用いる場合、全体をダブルクォートで囲む必要はありません(囲んでも構いません)。シェル変数が未設定の場合は空文字列として扱われます。

(case文自体の)終了ステータス

case文全体が1つの複合コマンドを構成し、その終了ステータスは、パターンが一致して実行されたリストの終了ステータスになります。ただし、どのパターンにも一致せず、リストが実行されなかった場合は終了ステータスは「0」になります[注5]。

解説

case文はC言語のswitch文に相当し、シェル変数や位置パラメータなどの内容や、コマンド置換によって得られた文字列の内容を元に複数に分岐する場合に便利です。

パターンにはパス名展開の特殊文字が使えるため、*)というパターンは、すべてのパターンに一致します。したがって、*)をパターンの一番最後に書いておくと、デフォルトのパターンとしてC言語のdefault:ラベルのように使うことができます。

リストの終わりの;;は、C言語のswitch文でのbreakに相当しますが、case文では、C言語とは違って;;を省略して、case文を抜けずに次のパターンのリストの処理を継続することはできません。複数のパターンのOR条件を用いたい場合は、「start|begin)」のように、|で区切って並べたパターンを使うようにします。

コマンド置換を使う方法

case文の文字列としては、シェル変数などのパラメータのほか、**リストA**のようにコマンド置換を使うこともできます[注6]。ここでは、uname -sコマンドが出力する文字列によって、OSの種類を判断して分岐しています。なお、unameはシステムの情報を表示するコマンドで、デフォルトで-sオプションつきの動作になるため、省略して単にunameとしてもかまいません。

OR条件のパターンを使う方法

パターンを、|で区切って並べることにより、OR条件のパターンを記述することができます。**リストB**にその例を示します。

注3　パス名展開の10.2節も合わせて参照してください。
注4　単語分割の10.6節も合わせて参照してください。
注5　case文自体の終了ステータスを実際に使用する場合はあまりありません。
注6　コマンド置換については9.3節を参照してください。

if文で記述することもできる

冒頭のcase文の例を、あえてif文で記述すると**リストC**のようになります。case文とは異なり、ifの直後のtestコマンドの引数として$1を記述する場合は、単語分割を避けるためにダブルクォートで囲んで"$1"とする必要があります。if文の場合はifやelifの直後で毎回testコマンドが実行されたり、毎回パラメータの参照が行われたりするため、文字列によって複数に分岐する場合はcase文のほうが簡潔でしょう。

また、条件判断のための文字列がパラメータではなく、前述の例の`uname`のようにコマンド置換によって得られたものの場合、if文で記述すると条件判断のたびにunameコマンドが実行されてしまうため、効率が悪くなります。

リストA　コマンド置換の文字列によって分岐

```
case `uname -s` in ……uname -sコマンドの出力文字列で分岐
  Linux) ……文字列がLinuxだった場合
    echo 'OSはLinuxです' ……OSはLinuxですと表示
    ;; ……リストの終了
  FreeBSD) ……文字列がFreeBSDだった場合
    echo 'OSはFreeBSDです' ……OSはFreeBSDですと表示
    ;; ……リストの終了
  Solaris) ……文字列がSolarisだった場合
    echo 'OSはSolarisです' ……OSはSolarisですと表示
    ;; ……リストの終了
  *) ……それ以外の文字列の場合
    echo 'その他のOSです' ……その他のOSですと表示
    ;; ……リストの終了
esac ……case文の終了
```

リストB　OR条件のパターンを使った例

```
case `uname -s` in ……uname -sコマンドの出力文字列で分岐
  Linux|FreeBSD) ……文字列がLinuxまたはFreeBSDだった場合
    echo 'OSはLinuxまたはFreeBSDです' ……該当メッセージを表示
    ;; ……リストの終了
  *) ……それ以外の場合
    echo 'そのほかのOSです' ……該当メッセージを表示
    ;; ……リストの終了
esac ……case文の終了
```

リストC　case文の代わりにif文を使用した例

```
if [ "$1" = start ] ……"$1"を文字列としてstartと比較
then
    echo 'プログラムを開始します' ……プログラム開始のメッセージを表示
elif [ "$1" = stop ] ……"$1"を文字列としてstopと比較
then
    echo 'プログラムを停止します' ……プログラム停止のメッセージを表示
else
    echo '引数にstartかstopを付けてください' …エラーメッセージの表示
fi
```

パス名展開の利用

case文のパターンの部分には、**リストD**のように各種パス名展開を使用することができます。

ファイル名でのパス名展開[注7]では，や/が特別扱いされ、*のパターンは**/home/username**や**.profile**といったファイル名には一致しませんでしたが、case文の場合は．や/が特別扱いされず、*はすべての文字列に一致します。同様に?のパターンが．や/などのすべての1文字に一致します。

case文のネスティング

case文の中のリストに、他の構文としてif／for／while文などを記述したり、case文自身をネスティングすることもできます。この場合も、パターンの直後のリストの最後には忘れずに;;を記述するようにします[注8]。

リストEは、元のcase文のuname -sでOSの種類によって分岐したあと、さらにネスティングされたcase文で、uname -mによってCPUの種類によって分岐するようにした例です。

リストD パターンとして、パス名展開を使用

```
case $string in ……シェル変数$stringの内容で分岐
  [a-z]) ……a～zまでの1文字に一致するパターン
    echo 'stringは英小文字1文字です' ……該当メッセージを表示
    ;; ……リストの終了
  ?) ……任意の1文字に一致するパターン
    echo 'stringは1文字です' ……該当メッセージを表示
    ;; ……リストの終了
  file*) ……fileで始まる文字列に一致するパターン
    echo 'stringはfileで始まる文字列です' ……該当メッセージを表示
    ;; ……リストの終了
  *) ……すべての文字列に一致するパターン
    echo 'stringはそれ以外です' ……該当メッセージを表示
    ;; ……リストの終了
esac ……case文の終了
```

注7　パス名展開については10.2節を参照してください。
注8　詳しくはリストの項(p.35)を参照してください。

リストE case文をネスティングした例

```
case `uname -s` in ……元のcase文の開始
  Linux) ……文字列がLinuxだった場合
    case `uname -m` in ……ネスティングされたcase文の開始
      i?86) ……文字列がi686／i586／i486などの場合
        echo 'OSはi386版Linuxです' ……該当メッセージを表示
        ;; ……リストの終了
      sparc*) ……文字列がsparcで始まっている場合
        echo 'OSはSPARC版Linuxです' ……該当メッセージを表示
        ;; ……リストの終了
      *) ……それ以外の文字列の場合
        echo 'OSはそのほかのLinuxです' ……該当メッセージを表示
        ;; ……リストの終了
    esac ……ネスティングされたcase文の終了
    ;; ……リストの終了
  *) ……それ以外の文字列の場合
    echo 'OSはLinux以外です' ……該当メッセージを表示
    ;; ……リストの終了
esac ……元のcase文の終了
```

パターンを()で囲むこともできる

Warning
Linux (bash) ○　FreeBSD (sh) ○　Solaris (sh) ×
この方法には制限があります。

bashまたはFreeBSDのshの場合は、case文のパターンの両側を( )で囲んで、**リストF**のように記述することもできます。ただし、このように記述してしまうと従来のshとの互換性がなくなり、またサブシェルの( )とも紛らわしくなるため、この記述法は用いないほうがよいでしょう。

リストF パターンを()で囲んだ例

```
case `uname -s` in
  (Linux) ……パターンの両側を( )で囲む
    echo 'OSはLinuxです'
    ;;
esac
```

注意事項

inやパターン)の直後に;をつけてはいけない

inやパターン)の直後の改行は単なる区切り文字としての改行であり、リストの終端ではありません。したがって、ここに;を入れると文法エラーになります。とくにcase文全体を1行で書く場合に注意してください。

○正しい例

```
case `uname` in Linux) echo 'OSはLinuxです';; esac
```

×誤った例

```
case `uname` in; Linux); echo 'OSはLinuxです';; esac
```

***)は最後のパターンとして書く**

パターンは、記述された順に比較が行われるため、デフォルトのパターンの*)は、最後のパターンとして記述する必要があります。*)を途中に記述してしまうと、それ以降のパターンには一致しなくなり、期待通り動作しません。

○正しい例

```
case `uname` in Linux) echo 'OSはLinux';; *) echo 'OSはLinux以外';; esac
```

×誤った例

```
case `uname` in *) echo 'OSはLinux以外';; Linux) echo 'OSはLinux';; esac
```

リストやパターンがなくてもよい

case文は、if/for/while文とは異なり、;;があればリストがなくてもかまいません。さらに、最後のパターンについては;;すら省略できます。また、パターン自体が1つもないcase文を記述することも可能です。よって、次の例は動作としてはほとんど意味がありませんが、文法的にはすべて正しいものになります。

```
case string in string);; esac
case string in string) esac
case string in esac
```

Memo

●case文の最後のesacは、caseのスペルを逆にしたものです。

参照

if文(p.43)　　リスト(p.35)　　サブシェル(p.72)

for文

Linux (bash)
FreeBSD (sh)
Solaris (sh)

変数に、指定の値をそれぞれ代入しながらループする

リストの右端がすでに改行などで終端されている場合、その右の;は不要

書式 `for 変数名 [in 値1 [値2…]]; do リスト; done`

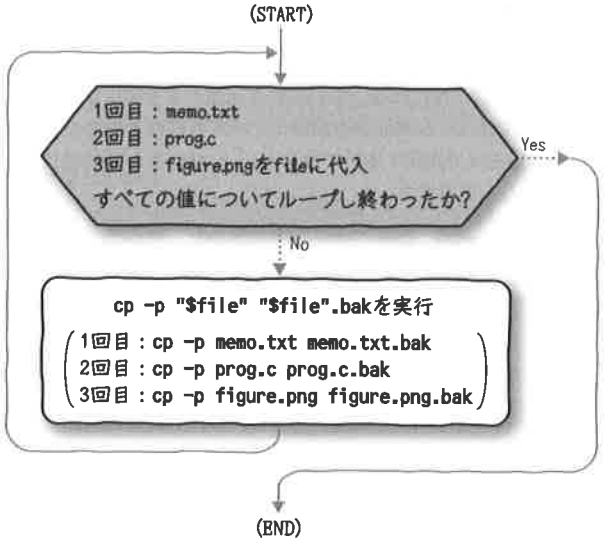
in 値1[値2…]が省略された場合はin "$@"と同じ！

最後の値の右端で改行されている場合、その右の;は不要

例

```
for file in memo.txt prog.c figure.png ……シェル変数fileにmemo.txt、prog.c、figure.pngを順に代入
do …… ループの開始
  cp -p "$file" "$file".bak …… 各ファイル名の末尾に.bakを付けてコピー
done …… ループの終了
```

図



基本事項

for文は、forの直後に[変数名]で指定したシェル変数に、inの直後に羅列した値([値1][値2]…)を順番に代入しながら、それぞれの値ごとにdoとdoneによって囲まれた[リスト]を実行してループする構文です。シェル変数の値は、ループごとに変化しますが、for文を終了しても、最後にシェル変数に代入された値(inの直後で最後に指定した値)が代入されたまま残ります。

inとその直後の値の羅列は省略することもでき、省略すると`in "$@"`が指定されたのと同じで、すべての位置パラメータの値が順にシェル変数に代入されることになります[注9]。

(for文自体の)終了ステータス

for文全体が1つの複合コマンドを構成しますが、その終了ステータスは、最後のループで実行されたリストの終了ステータスになります。ただし、ループが1回も実行されなかった場合(inの直後に空のシェル変数をダブルクォート(`""`)なしで指定した場合など)は、終了ステータスは「0」になります[注10]。

解説

for文は、異なる値の引数を使って同様の処理を繰り返すのに便利です。

例では「memo.txt」「prog.c」「figure.png」といった関連性のないファイル名を値として、ループ中のcpコマンドを実行しています。このように、シェルスクリプトのfor文は、C言語などのfor文とは違って、変数に代入される値に連続値などの規則性がなくても使用できるのが特長です。

パス名展開を使う方法

例ではinの直後に値を直接羅列していますが、代わりに**リストA**のように*などのパス名展開を使うこともできます。カレントディレクトリにあるすべてのファイルについて一定の処理を行うような場合、「`for file in *`」という記述がよく用いられます。このように記述すると、for文の実行時点で*が実際の複数のファイル名に置き換えられます[注11]。

すべてのファイルを*で指定するのではなく、「`*.txt`」とか「`image[0-9].png`」といったパターンの指定ももちろん可能です。

リストA パス名展開を使ったfor文

```
for file in * ……カレントディレクトリにあるファイル名に置き換わる
do ……ループの開始
  cp -p "$file" "$file".bak ……ファイル名の末尾に.bakを付けてコピー
done ……ループの終了
```

注9　特殊パラメータ"$@"についてはp.167を参照してください。
注10　for文自体の終了ステータスを実際に使用する場面はあまりありません。
注11　ただし.で始まるファイル名のファイルを除きます。

コマンド置換を使う方法

あらかじめ1行に1ファイルずつファイル名を記述した**リストB**のような「filelist」というファイルを作成しておけば、**リストC**のようにコマンド置換(バッククォート)でファイルリストを読み込み、これらのファイルに対してfor文を実行できます[注12]。

bashの場合は「`` `cat filelist` ``」の代わりに「`` `< filelist` ``」または「`$(< filelist)`」と記述することもできます。

なお、コマンド置換でファイルリストを読み込む方式では、ファイル名の中に、スペース、タブ、改行といった区切り文字や、*や?などのパス名展開の文字が含まれていると正常に動作しません[注13]。かといって、ダブルクォートを使って「`"`cat filelist`"`」とすることも、この場合はできません[注14]。

シェルスクリプトの引数で指定する方法

変数に代入する値をfor文に直接記述せずに、そのシェルスクリプトの引数(位置パラメータ)を使って値を指定することもできます。そのためには、**リストD**のように値として`"$@"`を指定します。あるいは`in "$@"`を省略して、単に`for file`と書いてもかまいません[注15]。

この内容を記述した「backup_file」というシェルスクリプトがカレントディレクトリにある場合、次の❶や❷のようにシェルスクリプトの引数を指定して実行できます。

```
❶ $ ./backup_file memo.txt prog.c figure.png
❷ $ ./backup_file *.txt *.png
```

リストB filelist

```
memo.txt ……1行に1ファイル名ずつ記述したリスト
prog.c
figure.png
  ⋮
```

リストC コマンド置換を使ったfor文

```
for file in `cat filelist` ……filelistをcatコマンドのコマンド置換で読み込む
do ……ループの開始
  cp -p "$file" "$file".bak ……ファイル名の末尾に.bakを付けてコピー
done ……ループの終了
```

リストD シェルスクリプトの引数の分だけループ

```
for file in "$@" ……すべての引数についてループする
do ……ループの開始
  cp -p "$file" "$file".bak ……ファイル名の末尾に.bakを付けてコピー
done ……ループの終了
```

注12　コマンド置換については9.3節を参照してください。
注13　パス名展開については10.2節を参照してください。
注14　ダブルクォートでは「filelist」の内容の文字列全体が、途中の改行なども含めて1つのファイル名とみなされてしまうためです。
注15　詳しくは「すべての引数についてループする」(p.285)を参照してください。

for文の中にほかの構文を記述

doとdoneの間のリストの中に、別の構文(for／while／if／case文)を記述することももちろんできます。**リストE**の例では、for文の中にcase文を記述し、ファイル名がすでに「*.bak」の形をしていれば、これ以上「*.bak.bak」というファイルにコピーされないようにしています。

continueを使う方法

for文の中で組み込みコマンドのcontinueを実行すると、その回のループの残りの部分を実行せずに、次の回のループに進みます。これを利用して、先述のリストEを**リストF**のように記述することもできます。

なお、ループ中でbreakを実行した場合は、その時点でfor文が終了します。

一定回数ループとfor文のネスティング

for文を使って一定回数ループするには、**リストG**のように、必要な回数分だけ適当な値を並べるようにします。一見、原始的な方法に見えますが、ループ中で値をインクリメントしたりする必要がないため、while文などを使った一定回数ループよりも簡便です。

さらに、**リストH**のようにfor文をネスティングしてループ回数を増やすこともできます。

なお、bashでは、算術式のfor文を使って一定回数のループを記述することもできます。

リストE for文の中にcase文を記述

```
for file in * ……カレントディレクトリにあるファイル名に置き換わる
do ……ループの開始
  case $file in ……ファイル名をcase文で条件判断
    *.bak) ……すでに.bakが付いているファイル名の場合
      ;; ……何もしないでリストの終了
    *) ……それ以外のファイル名の場合
      cp -p "$file" "$file".bak ……ファイル名の末尾に.bakを付けてコピー
      ;; ……リストの終了
  esac ……case文の終了
done ……ループの終了
```

リストF continueで次回のループに進む

```
for file in * ……カレントディレクトリにあるファイル名に置き換わる
do ……ループの開始
  case $file in ……ファイル名をcase文で条件判断
    *.bak) ……すでに.bakが付いているファイル名の場合
      continue ……これ以上何もしないで次のループに進む
      ;; ……何もしないでリストの終了
  esac ……case文の終了
  cp -p "$file" "$file".bak ……ファイル名の末尾に.bakを付けてコピー
done ……ループの終了
```


リストG for文を使った10回ループ

```
for i in 0 1 2 3 4 5 6 7 8 9 ……適当な値を10個並べる
do ……ループの開始
  echo "$i" ……試しに"$i"の値を表示
done ……ループの終了
```

リストH for文を使った100回ループ

```
for j in '' 1 2 3 4 5 6 7 8 9 ……10の位の値を10個並べる
do ……ループの開始
  for i in 0 1 2 3 4 5 6 7 8 9 ……1の位の値を10個並べる
  do ……ループの開始
    echo "$j$i" ……10の位と1の位の値を合わせて表示
  done ……ループの終了
done ……ループの終了
```

注意事項

for～doを1行に書く場合はdoの前に;が必要

for～doを1行に書く場合は、最後の値の後ろに;を付ける必要があります。;がないと、doという値も変数に代入するべき値の1つであると解釈されて、for文全体が正しく認識されません。

○正しい例

```
for i in 1 2 3; do
  echo "$i"
done
```

×誤った例

```
for i in 1 2 3 do
  echo "$i"
done
```

値として特殊な意味を持つ記号を使う場合はクォートが必要

値として、;、\、( )、その他特殊な意味を持つ記号を使う場合は、シングルクォート(' ')などでクォートする必要があります。

```
for i in 1 2 3 ';' '\' '(' ')'; do
  echo "$i"
done
```

doの直後に;を付けてはいけない

doの直後の改行は単なる区切り文字としての改行であり、リストの終端ではありません。したがって、ここに;を入れると文法エラーになります。とくにfor文全体を1行で書く場合に注意してください。

○正しい例

```
for i in 1 2 3; do echo "$i"; done
```

×誤った例

```
for i in 1 2 3; do; echo "$i"; done
```

リストがない場合は:が必要

doとdoneの間には、必ずリストを記述する必要があります。何も実行したくない場合(デバッグなどで一時的にコメントアウトする場合も含む)には、リストとして:コマンドを記述するようにします。

```
for i in 1 2 3
do
  :
done
```

Memo

- doとdoneを、それぞれ{と}に置き換え、C言語風に記述することもできます。
- for文は、csh系でのforeach文に相当します。
- bashでは、数値を使ったループには算術式のfor文が使えます。

参照

特殊パラメータ"$@"(p.167)　case文(p.49)　continue(p.94)　break(p.93)
while文(p.63)　算術式のfor文(p.61)　:コマンド(p.87)

算術式のfor文

○ Linux (bash)
× FreeBSD (sh)
× Solaris (sh)

ループ変数を使い算術式を評価しながらループを繰り返す

書式

```
for ((算術式1; 算術式2; 算術式3)) do リスト; done
for ((算術式1; 算術式2; 算術式3)) { リスト;}
```

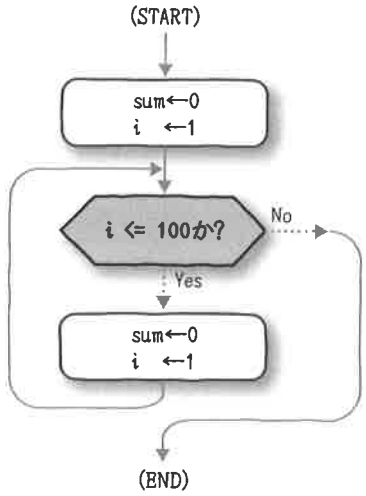
リストの右端がすでに改行などで終端されている場合、その右の;は不要

例

1から100までの整数の和を求める例

```
sum=0 ……合計値のシェル変数sumを0に初期化
for ((i = 1; i <= 100; i++)) { ……シェル変数iを使って1から100までループする
  ((sum += i)) ……sumの値にiの値を加える
} ……ループの終了
echo "$sum" ……最後にsumの値を表示
```

図



基本事項

算術式のfor文では、最初に[算術式1]が評価され、次に[算術式2]を評価し、この値が真であるかぎり繰り返しループを実行します。各ループでは、まずdo～done(または{～})の間の[リスト]を実行し、ループの終わりに[算術式3]を評価します。3つの[算術式]は任意に省略することができ、省略すると評価結果は常に真になります。

(算術式のfor文自体の)終了ステータス

算術式のfor文全体が1つの複合コマンドを構成しますが、その終了ステータスは、最後のループで実行されたdoとdoneの間のリストの終了ステータスになります(最後に実行された算術式2の終了ステータスではありません)。ただし、ループが1回も実行されなかった場合は、終了ステータスは「0」になります[注16]。

解説

bashでは、C言語のfor文と、ほぼ同じ形式の「算術式のfor文」が使えます。forの右側の((; ;))の中では、算術式の評価の(( ))と同じく、シェル変数の参照に$記号は必要なく、代入の=の前後にはスペースを入れることができます。また、演算には++(インクリメント)や+=(代入演算子)などのC言語風の演算子も使えます。

さらに、ループ部分のdoとdoneの代わりに{ }を使えば、記述スタイルがC言語とかなり近くなります。ただし、算術式のfor文を使うとbash依存のシェルスクリプトになってしまうため、一般的にはwhile文を使って記述したほうがよいでしょう。

なお、冒頭の例でのシェル変数sumの初期化を算術式のfor文の中に記述し、さらに最後のsumの値の表示を算術式展開を使って行うと**リストA**のようになります。

リストA シェル変数sumの初期化と結果表示も算術式(算術式展開)を利用した例

```
for ((sum = 0, i = 1; i <= 100; i++)) { ……シェル変数sumの初期化も算術式のfor文に記述
  ((sum += i)) ……sumの値にiの値を加える
} ……ループの終了
echo $((sum)) ……最後のsumの値の表示には算術式展開を利用
```

参照

算術式の評価(( ))(p.80)　算術式展開$(( ))(p.232)　while文(p.63)

注16 算術式のfor文自体の終了ステータスを実際に使用する場面はあまりありません。

while(until)文

Linux (bash)　FreeBSD (sh)　Solaris (sh)

条件が真であるかぎり(偽になるまで)ループを繰り返すにはwhile文を使う

書式

```
while [リスト]; do [リスト]; done
until [リスト]; do [リスト]; done
```

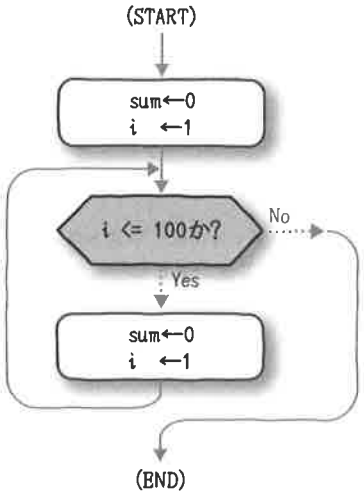
リストの右端がすでに改行などで終端されている場合、その右の;は不要

例

1から100までの整数の和を求める例

```
sum=0 ……合計値のシェル変数sumを0に初期化
i=1 ……ループに用いるシェル変数iを1に初期化
while [ "$i" -le 100 ] ……"$i"の値が100以下であるかぎりループ
do ……ループの開始
  sum=`expr "$sum" + "$i"` ……sumの値にiの値を加える
  i=`expr "$i" + 1` ……iの値に1を加える
done ……ループの終了
echo "$sum" ……最後にsumの値を表示
```

図



基本事項

while文は、リストの実行結果が**真**であるかぎり、ループ中のリストを繰り返し実行する構文です。

まずwhileの直後のリストが実行され、その結果が真(終了ステータスが「0」)である場合には次のdoとdoneの間のリストが実行されます。その後、再びwhileの直後のリストが実行され、その結果が真であれば再びdoとdoneの間のリストが実行されるという動作が繰り返されます。

whileの直後のリストが偽になれば、while文は終了します。

until文は、while文とは真偽が逆で、untilの直後のリストが**偽**(終了ステータスが「0」以外)である間、doとdoneの間のリストが繰り返し実行されます[注17]。

(while文自体の)終了ステータス

while文全体が1つの複合コマンドを構成しますが、その終了ステータスは、最後のループで実行されたdoとdoneの間のリストの終了ステータスになります(最後に実行されたwhileの直後のリストの終了ステータスではありません)。ただし、ループが1回も実行されなかった場合は、終了ステータスは「0」になります[注18]。

解説

whileの直後のリストには、if文の場合と同じく、testコマンドの[が記述されることが多いでしょう。testコマンドを使ってシェル変数の値の比較を行うことにより、さまざまなループの終了条件を設定できます。

例では、シェル変数iの値をまず「1」に初期化してからwhile文を実行し、while文ではtestコマンドによって"$i"の値が「100」以下である間、ループを実行するようになっています。ループ中では、sumにiの値をexprコマンドで加算しているほか、iの値自体もexprでインクリメント(+1)しています。

このように、while文のループ本体には、ループに使用しているシェル変数の値を更新するためのexprコマンドがよく使用されます。そのほかには、シェルの位置パラメータをシフトするためのshiftコマンドが使用される場合もあります。

一定回数ループする方法

while文を使って一定回数ループするには、**リストA**のように、適当なシェル変数を、所定のループ回数に達するまでインクリメントしながらループすることによって行います。

シェル変数のインクリメントには、一般的にはexprコマンドを用い、exprの出力をコマンド置換で取り込んで、新たにシェル変数に代入するようにします[注19]。

ただし、bashの場合は、i=`expr "$i" + 1`の代わりに((i++))と、算術式の評価を使って記述することもできます。

注17 until文は、while文とは真偽が逆になっていることを除き、while文と同じ動作です。until文は実際にはあまり使用されません。

注18 while文自体の終了ステータスを実際に使用する場面はあまりありません。

注19 コマンド置換については9.3節を参照してください。

シェルスクリプトの引数についてループする方法

シェルスクリプトの引数(位置パラメータ)で指定されたファイルすべてについて、そのファイル名に「.bak」を付加したバックアップファイルにコピーするには**リストB**のようにします。なお、同様の処理をfor文を使って行うこともできます[注20]。

ここでは、シェルスクリプトの引数の個数が入っている特殊パラメータ$#の値が「0」になるまでループを繰り返しています。ループ中では、"$1"を使ってファイル名を参照したあと、次のループのためにshiftコマンドで引数をずらします。$#の値は、shiftのたびにデクリメント(-1)されることに注意してください。

リストBの内容を記述した「backup_file」というシェルスクリプトがカレントディレクトリにある場合、次のようにシェルスクリプトの引数を指定して実行できます。

```
$ ./backup_file *.txt *.png
```

while文の中にほかの構文を記述

doとdoneの間のリストの中に、別の構文(for/while/if/case文)を記述することももちろんできます。**リストC**は、while文の中にcase文を記述し、シェルスクリプトの引数として、バックアップファイルのサフィックス(デフォルトは.bak)を変更できる-sオプションと、コピー先のディレクトリ(デフォルトは.)を変更できる-dオプションに対応した例です。これらのオプション以外の引数はファイル名とみなし、breakによって1番目のwhile文を抜けたあと、2番目のwhile文によってファイルのコピーが行われます。

リストCの内容を記述した「backup_file」というシェルスクリプトがカレントディレクトリにある場合、次のように実行できます。

```
$ ./backup_file -s .orig -d /tmp *.txt
```

リストA while文を使った10回ループ

```
i=1 ……シェル変数iを1に初期化
while [ "$i" -le 10 ]; do ……"$i"が10以下である間、ループする
  echo "$i" ……"$i"の値を表示
  i=`expr "$i" + 1` ……iをインクリメント (+1) する
done ……ループの終了
```

リストB シェルスクリプトの引数の分だけループ

```
while [ $# -gt 0 ] ……残りの引数があるかぎりループ
do ……ループの開始
  cp -p "$1" "$1".bak ……ファイル名の末尾に.bakを付けてコピー
  shift ……引数をシフトする
done ……ループの終了
```

注20 詳しくは「すべての引数についてループする」(p.287)を参照してください。

無限ループにする方法

whileの直後のリストに:コマンドを記述すると、:は常に真の値を返すため、whileのループが終了しない、無限ループになります。

リストDでは、無限ループ中でechoコマンドを実行し、「y」という文字を無限に出力しています。これは、yesコマンドの動作に相当します。このwhile文はそのままでは終了しませんので、終了するには割り込みキー(通常はCtrl+Cに設定されている)を押します。

なお、:コマンドの代わりにtrueコマンドを使用して、while trueと記述しても同じです。ただし、シェルによってはtrueコマンドが外部コマンドとして実装されている場合があるため、組み込みコマンドの:を使用したほうが効率がよいでしょう。

リストC while文の中にcase文を記述

```
suffix=.bak ……デフォルトのサフィックスを.bakに設定
dir=. ……デフォルトのコピー先ディレクトリを.に設定

while [ $# -gt 0 ] ……残りの引数(オプション)があるかぎりループ
do ……ループの開始
  case $1 in ……引数の内容によって分岐
    -s) ……-sオプションが指定された場合
      suffix="$2" ……-sの次の引数をサフィックスにする
      shift ……引数をシフト
      ;;
    -d) ……-dオプションが指定された場合
      dir="$2" ……-dの次の引数をコピー先ディレクトリにする
      shift ……引数をシフト
      ;;
    *) ……オプション以外の引数の場合
      break ……while文を中断する
      ;;
  esac ……case文の終了
  shift ……引数をシフト
done ……ループの終了

while [ $# -gt 0 ] ……残りの引数(ファイル名)があるかぎりループ
do ……ループの開始
  cp -p "$1" "$dir"/"$1""$suffix" ……対称ファイルを所定のファイル名でコピー
  shift ……引数をシフト
done ……ループの終了
```

リストD while文を使った無限ループ

```
while : ……無限ループにする書き方
do ……ループの開始
  echo y ……yを出力する
done ……ループの終了
```


注意事項

whileと[の間にはスペースが必要

whileと[の間にはスペースまたは改行などの区切り文字が必要です。「while[」のようにスペースを入れないで書くと、「while[」という名前のコマンドを実行するものとみなされ、コマンドが見つからないというエラーになります。

while～doを1行に書く場合はdoの前に;が必要

while～doを1行に書く場合は、testコマンドなどの最後に;を付けてリストを終端する必要があります。;がないと、doという文字列がtestコマンドの最後の引数であると解釈されて、while文全体が正しく認識されません。

○正しい例

```
while [ "$i" -le 10 ]; do
  echo "$i"
  i=`expr "$i" + 1`
done
```

×誤った例

```
while [ "$i" -le 10 ] do
  echo "$i"
  i=`expr "$i" + 1`
done
```

doの直後に;を付けてはいけない

doの直後の改行は単なる区切り文字としての改行であり、リストの終端ではありません。したがって、ここに;を入れると文法エラーになります。とくにwhile文全体を1行で書く場合に注意してください。

○正しい例

```
while [ "$i" -le 10 ]; do echo "$i"; i=`expr "$i" + 1`; done
```

×誤った例

```
while [ "$i" -le 10 ]; do; echo "$i"; i=`expr "$i" + 1`; done
```

リストがない場合は:が必要

doとdoneの間には、必ずリストを記述する必要があります。while文の場合、ループ中でshiftやexprなど、ループを進めるために必要なコマンドが必ず実行されることが多いため、問題になることはあまりありませんが、何も実行したくない場合には、リストとして:コマンドを記述するようにします。

```
while read line
do
  : ……:コマンド
done
```

Memo

- bashでは、exprコマンドを使わずに算術式の評価を利用して数値計算することができます。
- bashでは、testコマンドの[ ]を使う代わりに、[[ ]]を使った条件式の評価や、(( ))を使った算術式の評価を用いることもできます。
- bashでは、数値を使ったループには算術式のfor文を使うことができます。

参照

expr(p.261)	算術式の評価(( ))(p.80)	for文(p.55)
特殊パラメータ$#(p.171)	shift(p.123)	case文(p.49)
break(p.93)	:コマンド(p.87)	true(p.150)

select文

Linux (bash) ○ FreeBSD (sh) × Solaris (sh) ×

選択メニューを表示しユーザ入力を受け付ける

リストの右端がすでに改行などで終端されている場合、その右の;は不要

書式 select 変数名 [in 文字列1 [文字列2…]]; do リスト; done

in 文字列1[文字列2…]が省略された場合はin "$@"と同じ

最後の文字列の右端で改行されている場合、その右の;は不要

例 ユーザの入力を受け付け、何らかのメッセージを表示する例(select_test)

```
PS3='コマンド? '  ……プロンプトの文字列を設定
select cmd in up down left right look quit …選択肢の文字列を並べてselect文の開始
do  ……selectループの開始
  case $cmd in  ……case文を使って選択肢によって分岐
    up)  ……upが選択された場合
      echo '上に移動しました';;  ……対応するメッセージを表示
    down)  ……downが選択された場合
      echo '下に移動しました';;  ……対応するメッセージを表示
    left)  ……leftが選択された場合
      echo '左に移動しました';;  ……対応するメッセージを表示
    right)  ……rightが選択された場合
      echo '右に移動しました';;  ……対応するメッセージを表示
    look)  ……lookが選択された場合
      echo 'アイテムが落ちています';;  ……対応するメッセージを表示
    quit)  ……quitが選択された場合
      echo '終了します'  ……終了メッセージを表示
      break;;  ……select文を終了
    *)  ……それ以外の入力だった場合
      echo "$REPLY"'というコマンドはありません';;
                  ……入力文字列を含めてエラーメッセージを表示
  esac  ……case文の終了
  echo  ……1行改行
done  ……selectループの終了
```

基本事項

select文は、inの後の各文字列(文字列1 文字列2…)に通し番号を付けたメニューと、シェル変数**PS3**の値を内容とするプロンプトを標準エラー出力に出力します。このあと標準入力から番号を入力すると、その番号に対応する文字列を変数名で指定したシェル変数に代入し、do～doneの間のリストを実行します。ここで、番号に対応する文字列が存在しない場合や、番号以外の文字が入力された場合は、指定のシェル変数には空文字列が代入されます。いずれの場合も、読み込んだ入力そのものはシェル変数**REPLY**に代入されます。リストの実行が終わると再びメニューとプロンプトの表示に戻ります。

select文は、リスト中でbreakコマンドが実行されるか、または標準入力がEOF(*End Of File*)になると終了します[注21]。

なお、inとその後の文字列を省略した場合はin "$@"を指定したのと同じになります。

(select文自体の)終了ステータス

select文全体が1つの複合コマンドを構成し、その終了ステータスは、最後に実行されたリストの終了ステータスになります。ただし、リストが一度も実行されなかった場合は終了ステータスは「0」になります[注22]。

解説

画面に選択肢のメニューを表示し、ユーザにその選択肢の番号を入力させて、入力内容に応じて処理を行うには**select文**が便利です。select文で指定した**シェル変数**には、番号ではなく、その番号に対応する選択肢の**文字列**が代入されるため、これをcase文などを使って場合分けして目的の処理を行えばいいでしょう。また、ユーザの入力そのものがシェル変数**REPLY**に代入されるため、これを使って判断を行うこともできます。

ただ、select文がFreeBSDやSolarisのshでは使えないことと、select文のメニューの出し方が仕様によって固定化されていることなどから、一般的にはwhile文とreadコマンドを使ってユーザ入力を読み込んだほうがよいでしょう。

select文の実行例

冒頭の例を「select_test」というファイルに保存し、それを実行している例を**図A**に示します。ここでは単にメッセージが表示されるだけですが、ユーザの番号入力に反応してselect文が動作している様子がわかります。

注21 標準入力がキーボードの場合、通常はCtrl+Dを入力すると標準入力がEOFになります。

注22 select文自体の終了ステータスを実際に使用する場面はあまりありません。

図A select文の実行例

```
$ ./select_test ……select文が記述されたシェルスクリプトを実行
1) up ……選択メニューが表示される
2) down
3) left
4) right
5) look
6) quit
コマンド? 3 ……プロンプトに対して3を入力
左に移動しました ……メッセージが表示される

コマンド? ……プロンプトが表示されたところに改行を入力
1) up ……再び選択メニューが表示される
2) down
3) left
4) right
5) look
6) quit
コマンド? 5 ……次はプロンプトに対して5を入力
アイテムが落ちています ……別のメッセージが表示される

コマンド? ……プロンプトが表示されたところに改行を入力
1) up ……再び選択メニューが表示される
2) down
3) left
4) right
5) look
6) quit
コマンド? 6 ……quitのため6を入力
終了します ……終了メッセージが表示され、シェルスクリプトが終了する
```

Memo

- doとdoneを、それぞれ{と}で記述することもできます。
- bash 2.05a以前のバージョンでは、2回目以降の選択入力時に改行を入力しなくてもメニューが表示されます。

参照

read(p.111)

サブシェル

Linux (bash)
FreeBSD (sh)
Solaris (sh)

リストをまとめて別のシェルで実行する

書式 (リスト)

リストの右端は、改行や;で終端されていなくてもよい

例

```
( ……サブシェルの開始
  cd /some/dir ……/some/dirに移動
  cp -p "$file" backup-"$file" ……ファイル名の頭にbackup-を付けてコピー
) ……サブシェルの終了
```

基本事項

サブシェルの記述を使うと、リストがサブシェル上で実行されます。文法的には、サブシェルの( )全体が1つの複合コマンドとなります。

終了ステータス

リストの終了ステータスが、そのままサブシェルの終了ステータスになります。

解説

カレントディレクトリの一時変更や、シェル変数の局所的な使用など、元のシェルの状態には影響を及ぼさずに一定の処理を行いたい場合、その部分のリストを( )で囲んで、リストを**サブシェル**で実行させるようにします。サブシェルは、元のシェルとは別の子プロセスになるため、子プロセス上のカレントディレクトリやシェル変数などが変化しても、元のシェルには影響しません。

サブシェル内では、シェル変数への代入のほか、シェル変数やその他のパラメータの操作に関する、export／read／readonly／set／shift／unsetなどのコマンドの効果がサブシェル内のみになります[注23]。

そのほか、cd／umaskコマンドについても、影響がサブシェル内だけになり、exec／exitコマンドでは元のシェル上での動作とは異なる動作になります。

別シェルで実行されるグループコマンド

サブシェルの( )は、別シェルで実行される点を除いてグループコマンドの{ }と似ており、( )で囲まれた全体が1つの複合コマンドになる点も同じです。したがって、**リストA**のように、サブシェル全体の標準出力をファイルにリダイレクトすることも可能です。

注23 「シェル変数の代入と参照」(p.159)や「位置パラメータ」(p.162)も合わせて参照してください。

リストA サブシェルの標準出力をリダイレクト

```
( ……サブシェルの開始
  cd /some/dir ……/some/dirに移動
  pwd ……カレントディレクトリを表示
  ls -l ……ファイルのリストを表示
) > logfile ……サブシェルの標準出力をlogfileにリダイレクト
```

注意事項

グループコマンドの文法とは一部異なる

サブシェルの( )の文法はグループコマンドの{ }とは異なり、(の右側にスペースなどの区切り文字は必要なく、また、リストが;や改行で終端されていなくても)を閉じることができます。サブシェルおよびグループコマンドを、必要以外のスペースを取り除いて1行で記述すると次のようになります。

●**サブシェルの場合**

```
(echo Hello)
```

●**グループコマンドの場合**

```
{ echo Hello;}
```

Memo

●シェル関数の定義での関数本体をサブシェルの( )で記述することにより、**ローカル変数**を実現することができます。シェル関数の項(p.76)も参照してください。

参照

リスト(p.35) 複合コマンド(p.30) export(p.105) read(p.111)
readonly(p.115) set(p.120) shift(p.123) unset(p.131)
cd(p.95) umask(p.128) exec(p.100) exit(p.103)
グループコマンド(p.74)

グループコマンド

Linux (bash)
FreeBSD (sh)
Solaris (sh)

リストを1つのコマンドとしてまとめる

書式

```
{ リスト; }
```

{の直後にはスペースまたは改行が必要

リストの右端がすでに改行などで終端されている場合、その右の;は不要

例

```
{ ……グループコマンドの開始
  uname -a ……OS名やホスト名などの情報を表示
  date ……現在の日時を表示
  who ……ログイン中のユーザを表示
} > logfile ……以上すべての標準出力をlogfileにリダイレクト
```

基本事項

グループコマンドを記述すると、リストが現在のシェルでそのまま実行されます。文法的には、グループコマンド全体が1つの複合コマンドとなります。

終了ステータス

リストの終了ステータスが、そのままグループコマンドの終了ステータスになります。

解説

複数のコマンド(パイプライン)を改行や;などでつなげばリストになり、リストはそのままif文/for文などの構文の要素になれます。しかし、冒頭の例のように、リスト全体をまとめてファイルにリダイレクトしたり、パイプに通したりしたい場合、リスト全体をいったん1つのコマンドとしてまとめる必要があります。そこで使用するのが**グループコマンド**の{ }です。

リストは、そのままではあくまでリストですが、これをグループコマンドの{ }で囲むことにより、全体が1つの複合コマンドになります。シェルスクリプト上でコマンドとして記述できる個所には、単純コマンドなどの代わりにグループコマンドを記述することが可能です。

グループコマンドがないと……

仮に冒頭の例をグループコマンドを使わずに記述すると、**リストA**のようになります。このようにまず1つ目のuname -aコマンドの出力を>で「logfile」にリダイレクトし、2つ目のdateコマンド以降は>>でアペンドモードで同じ「logfile」にリダイレクトすることになります。この方法でも悪くはありませんが、同じ「logfile」を何度も指定しなければならない点が不便です[注24]。

そこで**リストB**のようにグループコマンドを使えば記述が簡潔になります。なお、リストBではグループコマンドを1行で記述しています。

リストA グループコマンドを使わずに記述した例

```
uname -a  > logfile ……uname -aの出力をlogfileにリダイレクト
date     >> logfile ……dateの出力をlogfileにアペンドモードでリダイレクト
who      >> logfile ……whoの出力をlogfileにアペンドモードでリダイレクト
```

リストB グループコマンドを1行で記述

```
{ uname -a; date; who; } > logfile ……3つのコマンドをまとめてlogfileにリダイレクト
```

注意事項

{の直後にはスペースまたは改行が必要

サブシェルの( )の場合とは異なり、{の直後には、区切り文字としてのスペースまたは改行が必要です。次の誤った例のように{の直後にスペースを入れなかった場合、「{echo」という名前のコマンドを実行するものとみなされ、エラーになります。

○正しい例

```
{ echo Hello;}
```

×誤った例

```
{echo Hello;}
```

変数の操作はすべて影響する

サブシェルの場合と異なり、{ }の中で変数に値を代入したり、exportしたり、unsetしたりといった操作を行った場合、それらは{ }を抜けてもすべて影響を及ぼしたままになります。これが不都合な場合は、グループコマンドではなくサブシェルを使う必要があります。

Memo

- グループコマンドは、シェル関数の定義での関数本体の記述に利用されます。シェル関数の項(p.76)も参照してください。

参照

リスト(p.35)　複合コマンド(p.30)　サブシェル(p.72)

注24 標準出力のリダイレクトの項(p.243)や標準出力のアペンドモードでのリダイレクトの項(p.245)を参照してください。

シェル関数

Linux (bash)
FreeBSD (sh)
Solaris (sh)

一定の処理を関数としてまとめる

書式
定義　関数名() { リスト ;}
実行　関数名 [引数 ...]

リストの右端がすでに改行などで終端されている場合、その右の ; は不要

例

```
func() ……シェル関数funcの定義開始
{
  echo 'シェル関数が実行されました' ……試しにメッセージを出力
} ……シェル関数funcの定義終了

func ……シェル関数funcを実行する
```

基本事項

冒頭の書式のようにシェル関数を**定義**すると、以降、関数名で指定した、そのシェル関数名を「コマンド名」として使用することができるようになります。冒頭の書式のようにシェル関数を**実行**すると定義されたリストが実行されます。

シェル関数の実行には引数を付けることができ、シェル関数内の位置パラメータは、一時的にシェル関数の引数で置き換えられます。これにともない、特殊パラメータ"$@"、$*、$#も変化します。シェル関数内の位置パラメータと特殊パラメータ"$@"、$*、$#は、シェル関数内でのみ有効です。

終了ステータス

シェル関数の終了ステータスは、シェル関数内で最後に実行されたリスト(returnコマンドの場合を含む)の終了ステータスになります。シェル関数の定義の終了ステータスは、文法エラーがないかぎり「0」になります。

解説

シェル関数を使えば、一定の処理を**サブルーチン**としてまとめておくことができます。シェル関数には引数を渡せるため、シェル関数の呼び出しは、外部コマンドや組み込みコマンドの実行とほとんど同じ感覚で行えます。

また、シェルスクリプト内だけでなく、コマンドライン上でよく実行するコマンドやオプションの組み合わせをシェル関数として定義しておけば、シェル関数をaliasコマンドの感覚で使うことができます。

なお、いったん定義されたシェル変数は、unsetコマンドによって削除できます。

シェル関数の使用例

シェル関数を使って、ls -lを実行するll(エルエル)を定義している例を**リストA**に示します。シェル関数内では、引数のすべてを"$@"で受け取ってlsコマンドに渡しています。このように定義すれば、以降、単に「ll」または「ll ディレクトリ名」というコマンドを実行すると、それぞれ「ls -l」「ls -l ディレクトリ名」というコマンドが実行されることになります。

リストAが記述されたファイルをコマンドラインのシェル上に反映するには、.コマンドでファイルを読み込む必要があります。

なお、リストAにおいてシェル関数名自体をlsとすると、シェル関数内から自分自身の関数が呼び出されてしまい、再帰呼び出しが無限に発生してシェル関数が終了しなくなるため、注意してください。ただし、シェル関数の再帰呼び出し自体は可能です(詳しくは後述)。

再帰呼び出し

シェル関数内で、自分自身の関数を呼び出す**再帰呼び出し**を行うことは可能です。**リストB**は、再帰呼び出しによって階乗を求める例です。ここでは、「n」の階乗(n!)を求めるために、まず(n-1)!を再帰呼び出しによって求め、その値に「n」をかけて答を出しています。

このシェルスクリプトを「factorial」という名前でカレントディレクトリに保存すれば、**図A**のように階乗を求めることができます。ただし、あまり大きい値を指定するとexprがオーバーフローを起こすため、計算結果が正しくなくなります。

リストA ls -lを実行するシェル関数llを定義

```
ll() ……シェル関数llの定義開始
{
  ls -l "$@" ……シェル関数の引数をそのまま引き継いでls -lを実行
} ……シェル関数llの定義終了
```

リストB 再帰呼び出しを使って階乗を求めるシェルスクリプト

```
#!/bin/sh

factorial() ……シェル関数factorialの定義開始
{
  if [ "$1" -le 1 ]; then ……もし引数が1以下の数値なら
    echo 1 ……答として1を出力
    return ……シェル関数からリターンする
  fi ……if文の終了
  n=`expr "$1" - 1` ……引数から1を引く
  n=`factorial "$n"` ……(引数-1)を引数としてfactorialを再帰呼び出し
  expr "$n" \* "$1" ……その結果に元の引数をかけて答を出力
} ……シェル関数factorialの定義終了

factorial "$1" ……シェル関数factorialを実行する
```

図A factorialを実行して階乗を求める

```
$ ./factorial 1                    1の階乗は
1                                  たしかに1
$ ./factorial 2                    2の階乗は
2                                  たしかに2
$ ./factorial 3                    3の階乗は
6                                  たしかに6
$ ./factorial 10                   10の階乗は
3628800                            たしかに3628800
```

シェル関数内の変数について

C言語とは異なり、シェル関数内の変数は**グローバル変数**のように扱われます。たとえば、シェル関数内で変数に値を代入すると、シェル関数からリターンした後も、その変数には値が代入されたままになります。ただし、位置パラメータについてはシェル関数内のみ有効となり、位置パラメータをsetやshiftコマンドなどで操作しても、シェル関数からリターンすると、シェル関数の呼び出し前の位置パラメータの状態に戻ります。

前述のリストBの再帰呼び出しの例では、使用しているシェル変数nの値が再帰呼び出しによって変化してもかまわない使い方だったため、問題が発生していなかったことに注意してください。

シェル関数内部で、シェル関数の呼び出し元からは独立した変数、つまり**ローカル変数**を使用したい場合は、**リストC**のようにシェル関数の本体を、{ }の代わりにサブシェルの( )を使って記述します。一般に、シェル関数の本体には{ }や( )だけでなく、if文やwhile文などの(シェル関数を除く)複合コマンドを直接1つだけ記述することができます。

リストC サブシェルを使ったローカル変数の実現

```
func() ……シェル関数funcの定義開始
( ……サブシェルの開始
  i=3 ……ローカル変数扱いのシェル変数iに値を代入
  echo "iの値は$iです" ……試しにiの値を表示
) ……サブシェルの終了とともにシェル関数funcの定義終了
```

localコマンドを使ったローカル変数の実現

bashとFreeBSDのshにはlocalという組み込みコマンドがあり、**リストD**のようにlocalコマンドで**ローカル変数**を宣言して使うこともできます。

Warning
Linux (bash) ○ FreeBSD (sh) × Solaris
この方法には制限があります。

リストD localコマンドを使ったローカル変数の実現

```
func() ……シェル関数funcの定義開始
{
  local i ……シェル変数iをローカル変数として宣言する
  i=3 ……ローカル変数扱いのシェル変数iに値を代入
  echo "iの値は$iです" ……試しにiの値を表示
} ……シェル関数funcの定義終了
```


functionをシェル関数の頭に付けて定義

bashでは、**リストE**のように、シェル関数の定義で「function」というキーワードを頭に付けることができます。functionを付けた場合は関数名の直後の()を省略し、単に「function func」と記述することもできます。

Warning
Linux (bash) × FreeBSD × Solaris
この方法には制限があります。

リストE functionを使った例

```
function func() ……シェル関数funcの定義開始
{
  echo 'シェル関数が実行されました' ……試しにメッセージを出力
} ……シェル関数funcの定義終了
```

注意事項

実行する前に定義が必要

シェル関数の定義部分は、シェル関数を実行している部分よりも前になければなりません。シェル関数の定義よりも前に実行しようとすると、シェル関数名に該当するコマンドが見つからないというエラーになります。

{の直後にはスペースまたは改行が必要

シェル関数の定義では、{の記号が正しく認識されるように、{の直後には、区切り文字としてのスペースまたは改行が必要です。シェル関数を1行で定義し、かつ必要以外のスペースを取り除くと、次のようになります。

```
func(){ echo Hello;}
```

リストがない場合は:が必要

関数内のリストは必ず必要です。何も実行しない空の関数を定義する場合は、次のように、リストとして:コマンドを記述するか、またはreturn 0を記述するようにします。

```
func(){ :;}
```

シェル関数の本体部分にグループコマンド以外の複合コマンドを使う場合

シェル関数の定義の本体部分には、シェル関数を除く任意の複合コマンドが使えるため、次のような記述も文法的に可能です。

```
func() if :; then echo 'シェル関数が実行されました'; fi
```

Memo

- シェル関数は、複合コマンドの一つには含めないで考える場合があります。
- 古いbash 1.xではシェル関数の本体に{ }を使った記述しか行えません。

参照

位置パラメータ(p.162) 特殊パラメータ"$@"(p.167) 特殊パラメータ$*(p.169)
特殊パラメータ$#(p.171) return(p.118) unset(p.131) .コマンド(p.90)
サブシェル(p.72) local(p.154) :コマンド(p.87) グループコマンド(p.74)

算術式の評価 (( ))

○ Linux (bash)
× FreeBSD (sh)
× Solaris (sh)

算術演算を行いその結果によって終了ステータスを返す

書式 ((算術式))

例

```
((i = 1)) ……シェル変数iを1に初期化
while ((i <= 10)) ……iの値が10以下であるかぎりループ
do ……ループの開始
  echo "$i" ……iの値を表示
  ((i++)) ……iの値をインクリメント（1を加える）
done ……ループの終了
```

基本事項

(( ))で囲まれた部分は算術式とみなされ、**表A**(次ページを参照)の演算子を使った評価が行われます。算術式の中では、シェル変数は頭に$記号を付けずに参照でき、シェル変数の値が文字の場合はその文字がシェル変数名とみなされ、再度参照が行われます。数値は符号付きの整数として扱われます。算術式では、終了ステータスとは逆に、「0」以外の値を真とみなします。

終了ステータス

算術式の評価結果が真(「0」以外)なら、算術式の評価の終了ステータスは真(0)に、算術式の評価結果が偽(0)なら、算術式の評価の終了ステータスは偽(1)になります。

解説

算術式の評価では、表Aのような、ほぼC言語と同じ演算子が使え、これによってシェル自身で数値演算を行えます。この中には、累乗(**)という、C言語にはない演算子もあります。

算術式の評価は、その終了ステータスを利用してif文やwhile文の条件判断に使うことができるほか、シェル変数の代入やインクリメントなどのために用いることもできます。算術式の評価でのシェル変数への代入の場合、通常のシェル変数への代入とは異なり、=の前後にスペースを入れてもかまいません。

算術式中では、シェル変数の頭に$を付けずに変数名だけで参照できるほか、>、<、*、&その他のシェル上で特殊な意味を持つ記号がクォートなしで使えます。

表A 算術式の評価で使用できる演算子 ……上から優先順位の高い順

演算子	内容
変数 ++	変数の値を評価したあとで変数に1を加える
変数 --	変数の値を評価したあとで変数から1を引く
++ 変数	変数に1を加えたあとで変数の値を評価する
-- 変数	変数から1を引いたあとで変数の値を評価する
-(符号)	負の数(2の補数)を表す
+(符号)	正の数を表す
!	論理的否定
~	ビットごとの否定(1の補数)
**	累乗
//	乗算
/	除算
%	剰余
+	加算
-	減算
<<	左ビットシフト
>>	右ビットシフト
<=	より小さいか等しければ真
>=	より大きいか等しければ真
<	より小さければ真
>	より大きければ真
==	等しければ真
!=	等しくなければ真
&	ビットごとのAND(論理積)
^	ビットごとのXOR(排他的論理和)
|	ビットごとのOR(論理和)
&&	論理的AND(論理積)
||	論理的OR(論理和)
式1 ? 式2 : 式3	式1が真なら式2を、式1が偽なら式3を評価する
=	代入
*=	乗算して代入
/=	除算して代入
%=	剰余をとって代入
+=	加算して代入
-=	減算して代入
<<=	左ビットシフトして代入
>>=	右ビットシフトして代入
&=	ビットごとのAND(論理積)をとって代入
^=	ビットごとのXOR(排他的論理和)をとって代入
|=	ビットごとのOR(論理和)をとって代入
式1 , 式2	式1、式2の順に評価し、式2の値を評価結果とする

算術式の評価を使わない記述

算術式の評価は便利ですが、FreeBSDやSolarisのshでは使えないため、移植性のためには通常の記述方法も知っておく必要があります。

冒頭の例を算術式の評価を使わないで記述すると**リストA**のようになります。このように、シェル変数の代入は=の前後にスペースを入れず、条件判断はtestコマンドを使用し、数値演算はexprコマンドを使用することになります。

リストA 算術式の評価を使わない記述

```
i=1 ……シェル変数iを1に初期化
while [ "$i" -le 10 ] ……iの値が10以下であるかぎりループ
do ……ループの開始
  echo "$i" ……iの値を表示
  i=`expr "$i" + 1` ……iの値に1を加える
done ……ループの終了
```

Memo

- 算術式の評価である((算術式))は、letコマンドを使って「let '算術式'」と記述したのと同じです。
- 算術式の評価の結果を、終了ステータスで判断するのではなく、演算結果の数値を受け取って利用したい場合は、$((算術式))の形の算術式展開を用いります。

参照

test(p.147)　expr(p.261)　let(p.153)　算術式展開$(( ))(p.232)

条件式の評価 [[ ]]

Linux (bash)　FreeBSD (sh)　Solaris (sh)

条件式を評価し、その結果によって終了ステータスを返す

書式 [[条件式]]

例

```
if [[ "$i" -le 3 ]] ……[[ ]]を使って、iの値が3以下かどうかを判断
then ……thenのリストの開始
  echo 'iの値は3以下です' ……メッセージを表示
fi ……if文の終了
```

表A 算術式の評価で使用できる演算子

testコマンドと異なる点のみ掲示。その他の演算子についてはtestコマンドの項の表Aと同じである

条件式	内容
条件式1 && 条件式2	条件式1と条件式2の両方が真ならば真
条件式1 || 条件式2	条件式1と条件式2のどちらか真ならば真
条件式1 -a 条件式2	([[ ]]では使用不可)
条件式1 -o 条件式2	([[ ]]では使用不可)
文字列 == パターン	文字列がパターンに合致すれば真
文字列 != パターン	文字列がパターンに合致しなければ真

基本事項

[[ ]]で囲まれた部分は条件式とみなされ、testコマンドの項の表Aとほぼ同じ演算子を使った条件判断が行われ、その結果を終了ステータスとして返します。

testコマンドとは異なる演算子を**表A**(上記)に示します。testコマンドとは異なり、条件式の演算子として解釈される< >、( )、&&、||をクォートする必要はありません。

終了ステータス

条件式の評価結果が真なら終了ステータスは「0」に、条件式の評価結果が偽なら終了ステータスは「1」になります。

解説

条件式の評価の[[ ]]は、testコマンドの[ ]と似ていますが、testコマンドとは異なり、シェルの文法上で直接条件式を解釈します。

[[ ]]では、AND条件やOR条件を表す-aや-oの演算子がそれぞれ&&と||に改められました。シェル上では同じ記号が&&リストや||リストで使用されますが、[[ ]]の中では条件

式の演算子であると解釈されます。同様に、< >、( )の演算子についても、リダイレクトやサブシェルとは解釈されないため、クォートする必要はありません。

testコマンドのほうにはない機能として、==と!=の演算子の右側の文字列に、パス名展開と同様のパターンが使え、*、?、[a-z]などのパターンで文字列の判定を行えます[注25]。

このように、[[ ]]を使った条件式の評価では、[[ ]]の中身をシェルが特別に解釈する必要があるため、[[という名前の外部コマンドを実装することは原理的にできません。

なお、シェルスクリプトの移植性のためには[[ ]]ではなく、testコマンドの[ ]を使って記述したほうがよいでしょう。

参照

test(p.147)

注25 パス名展開については10.2節を参照してください。

>第5章 組み込みコマンド1