オペレーティングシステム 第 1 回レポート課題

21T2166D 渡辺大樹

2025 年 5 月 31 日

演習資料 2

問 17 - ディレクトリの削除

指定した名前のディレクトリを削除するとき子ディレクトリやファイルもすべて削除するには、以下のようにする。

ソースコード 1　問 17 の解答

```
$ rm -r <ディレクトリ名>
```

問 20 - cp と ln の違い

問 20 では a.txt を cp,ln,ln -s でコピーした場合の違いを可視化する課題となっている。cp と ln はどちらもファイルをコピーするコマンドである。

ソースコード 2　問 20 の解答

```
$ echo aaaa > a.txt
$ cp a.txt b.txt
$ ln a.txt c.txt
$ ln -s a.txt d.txt
```

以上のコードを実行すると、a.txt,b.txt,c.txt,d.txt の内容はすべて aaaa となる。続いて、

ソースコード 3　問 20 の解答

```
$ echo bbbb > a.txt
```

を実行すると、a.txt の内容が bbbb に変更される。このとき、b.txt の内容は aaaa のままであるが、c.txt,d.txt の内容は bbbb に変更される。これは、cp はファイルの内容をコピーするのに対して、ln はファイルのリンクを作成するためである。また、ln -s はシンボリックリンクを作成するため、リンク先のファイルの内容が変更されるとリンク元のファイルの内容も変更される。

また続いて

ソースコード 4　問 20 の解答

```
$ rm a.txt
```

を実行すると、a.txt が削除される。このとき、b.txt は aaaa、c.txt は bbbb のままであるが、d.txt はリンク先のファイルが削除されたため、cat: d.txt: No such file or directory と表示される。また ls d.txt を実行すると、d.txt が赤文字で表示されるようになる。

問 22 - grep コマンド

ソースコード 5　問 22 の解答

```
$ cat *.txt | grep tanaka > tanaka.data
```

上記のコマンドは、全ての.txt ファイルの内容を結合し、その中から tanaka という文字列を含む行を抽出し、tanaka.data というファイルに出力するコマンドである。

問 24 - grep のオプション

1. メールアドレスの抽出

ソースコード 6　問 24-1 の解答

```
$ grep -E '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}' test >
    result1
```

メールアドレスは、ユーザ名@ドメイン名の形式であるため、ユーザ名とドメイン名を正規表現で指定することで抽出することができる。そのため、ユーザ名を [a-zA-Z0-9._%+-]+、ドメイン名を [a-zA-Z0-9.-]+\.[a-zA-Z]{2,} として抽出している。

2. 電話番号の抽出

ソースコード 7　問 24-2 の解答

```
$ grep -E '(\+81-)?0\d{1,4}-\d{1,4}-\d{4}' test > result2
```

電話番号は、0 から始まる 11 桁の数字、もしくは +81-から始まる 10 桁の数字列であるため、このパターンで抽出することができる。

3. grep のパターン

■(a). 電話番号のパターン

ソースコード 8　問 24-3(a) の解答

```
(\+81-)?0\d{1,4}-\d{1,4}-\d{4}
```

電話番号のパターンは以上の正規表現で表される。

■(b). 郵便番号のパターン

ソースコード 9　問 24-3(b) の解答

```
\d{3}-\d{4}
```

郵便番号のパターンは以上の正規表現で表される。

問 25 - ファイルパーミッション

chmod コマンドで適当なファイルの権限を操作してみる。

ソースコード 10　問 25 の解答

```
$ touch test.txt
$ ls -l test.txt
-rw-r--r-- 1 user user 0 May 29 21:00 test.txt
$ chmod u-r test.txt
$ ls -l test.txt
--w-r--r-- 1 user user 0 May 29 21:00 test.txt
$ chmod g+w test.txt
$ ls -l test.txt
--w--rw-r-- 1 user user 0 May 29 21:00 test.txt
$ chmod 777 test.txt
$ ls -l test.txt
-rwxrwxrwx 1 user user 0 May 29 21:00 test.txt
```

またディレクトリに対しても行い、変化を見てみる。

ソースコード 11　問 25 の解答

```
$ mkdir test
$ ls -ld test
drwxr-xr-x 2 user user 4096 May 29 21:00 test
$ chmod u-r test
$ ls -ld test
d-wxr-xr-x 2 user user 4096 May 29 21:00 test
$ chmod g+w test
$ ls -ld test
d-wxrw-r-x 2 user user 4096 May 29 21:00 test
$ chmod 777 test
$ ls -ld test
drwxrwxrwx 2 user user 4096 May 29 21:00 test
```

chmod コマンドではこのようにファイルやディレクトリの権限を変更できる。試しに実行パーミッションをなくしたディレクトリに対して cd コマンドを実行してみる。

ソースコード 12　問 25 の解答

```
1    $ cd test
2    bash: cd: test: Permission denied
```

このように、実行パーミッションがないディレクトリに対して cd コマンドを実行すると Permission denied と表示される。

問 26 - chgrp コマンド

chgrp コマンドはファイルやディレクトリのグループを変更するコマンドである。

ソースコード 13　問 26 の解答

```
1    $ chgrp <グループ名> <ファイル名>
```

このようにして、ファイル名のグループをグループ名に変更することができる。

問 27 - グループの追加

あるユーザ foo をグループ wheel に追加するには、以下のコマンドを実行する。

ソースコード 14　問 27 の解答

```
1    $ usermod -aG wheel foo
```

問 28 - kill コマンド

kill コマンドがデフォルトで送るシグナルは TERM である。

演習資料 3

問 4 - シェル変数

ユーザが一時的に価設定をしたシェル変数を無効化するには、以下のコマンドを実行する。

ソースコード 15　問 4 の解答

```
1    $ unset <変数名>
```

問 5 - 文字列のスクリプト

以下のスクリプトは、実行すると「Hello, world!」と現在の日付が表示される。

ソースコード 16　問 4 の解答

```
1    #!/bin/bash
2    # Greeting Script
3    greeting='Hello, world!'
4    dateInf="today is $(date +%m/%d)"
5    echo "${greeting}, ${dateInf}"
```

このスクリプトでは、まず変数「greeting」に「Hello, world!」という文字列を代入している。次に、変数「dateInf」に「today is」と現在の日付を取得するコマンド「date +%m/%d」を組み合わせた文字列を代入している。最後に、変数「greeting」と「dateInf」を結合して表示するために、echo コマンドを使用している。

実行結果は、例えば「Hello, world!, today is 07/01」といった形式で表示される。

問 6 - シェルの関数

以下のスクリプトについて動作を説明していく。

ソースコード 17　問 6 の解答

```
1    #!/bin/bash
2    mawaru(){
3        local i = 1
4        while [ $i -le 10 ]
5        do
6            echo $i
7            i = 'expr $i + 1'
8        done
9    }
10   mawaru
```

このスクリプトは、‘mawaru‘ という名前のシェル関数を定義し、それを呼び出しています。関数 ‘mawaru‘ は、内部でローカル変数 ‘i‘ を 1 で初期化し、‘while‘ ループを使用して ‘i‘ が 10 以下である間、‘i‘ の値を表示し、‘i‘ を 1 ずつインクリメントすることを意図しています。正しく動作すれば、1 から 10 までの数字が 1 行ずつ表示されます。

問 7 - export コマンド

シェルスクリプトやコマンドラインで設定したシェル変数は、そのシェル内でのみ有効である。‘export‘コマンドは、シェル変数を環境変数としてエクスポートし、そのシェルから起動されるサブシェル（子プロセス）でもその変数が利用できるようにする。例えば、‘export FOO‘ とすると、変数 ‘FOO‘ が環境変数として設定され、このシェルから実行される他のコマンドやスクリプトからも参照可能になる。よく使われる例として、‘ /.profile‘ や ‘ /.bashrc‘ で ‘PATH‘ 環境変数を設

定する際に ‘export PATH‘ と記述される。

問 8 - シェル変数への代入

‘FOO=abcde‘ は、シェル変数 ‘FOO‘ に文字列 ‘abcde‘ を代入するコマンドである。このとき、‘=‘ の前後にスペースを入れてはならない。スペースを入れると、シェルは ‘FOO‘ をコマンド名として解釈しようとするためエラーとなる。代入された変数は、同じシェルセッション内であれば ‘$FOO‘ や ‘$FOO‘ のようにしてその値を参照できる。

問 9 - 変数展開

シェルスクリプトにおいて、変数名の前に ‘$‘ を付けることで変数の値を参照できる。これを変数展開と呼ぶ。

- ‘$‘単独: 通常、特別な意味を持たないか、文脈によって意味が変わる（例: 正規表現の行末)。シェルプロンプトとして表示されることが多い。
- ‘$FOO‘: シェル変数 ‘FOO‘ の値を展開する。
- ‘$FOO‘: シェル変数 ‘FOO‘ の値を展開する。波括弧 ‘‘ で囲むことで、変数名の区切りを明確にすることができる。例えば、‘$FOObar‘ と書けば ‘FOO‘ の値と文字列 ‘bar‘ が連結されるが、‘$FOObar‘ と書くとシェルは ‘FOObar‘ という名前の変数を探そうとしてしまう。

問 10 - クォーテーションの違い

シェルスクリプトでは、シングルクォート (‘’‘)、ダブルクォート (‘”‘)、バッククォート (“ ‘ “) はそれぞれ異なる意味を持つ。

- シングルクォート (‘’‘): 囲まれた文字列を完全にリテラルとして扱う。変数展開やコマンド置換は行われない。例えば ‘’Hello, $USER’‘ はそのまま ‘Hello, $USER‘ という文字列になる。
- ダブルクォート (‘”‘): 囲まれた文字列内で変数展開 (‘$VAR‘)、コマンド置換 (“ ‘command‘ “ または ‘$(command)‘)、特定のバックスラッシュシーケンス (‘\n‘, ‘ \t‘ など) が解釈される。例えば ‘”Hello, $USER”‘ は ‘Hello, (現在のユーザー名)‘ のように展開される。
- バッククォート (“ ‘ “): 囲まれた文字列をコマンドとして実行し、その標準出力を文字列として展開する（コマンド置換)。例えば “ ‘date‘ “ は現在の日時を表す文字列に置き換えられる。現代の bash では、可読性やネストの容易さから ‘$(command)‘ の形式が推奨されることが多い。

問 11 - スクリプトの引数

シェルスクリプト実行時に渡される引数は、スクリプト内で特殊な変数を使って参照できる。

- ‘$#‘: スクリプトに渡された引数の総数を表す。
- ‘$n‘: n 番目の引数を表す。例えば ‘$1‘ は最初の引数、‘$2‘ は 2 番目の引数。n が 9 を超える場合は ‘$10‘ のように波括弧で囲む必要がある。
- ‘$0‘: 実行されているスクリプト自体の名前を表す。
- ‘$*‘: 全ての引数を一つの文字列として展開する。IFS 環境変数の最初の文字で区切られる。
- ‘$@‘: 全ての引数を個別の文字列として展開する。ダブルクォートで囲んだ場合 (‘”$@”‘)、各引数が個別の単語として扱われるため、引数にスペースが含まれる場合に有用。
- ‘shift‘: 引数リストを左にシフトする。例えば ‘shift‘ を実行すると、元の ‘$2‘ が新しい ‘$1‘ になり、元の ‘$1‘ は失われる。‘$#‘ の値も 1 減少する。

これらの変数を利用することで、スクリプトは与えられた引数に応じて動作を変えることができる。

問 12 - read コマンド

‘read‘コマンドは、標準入力から 1 行読み込み、その内容を変数に格納するシェル組み込みコマンドである。提示されたスクリプトの動作は以下の通り。

ソースコード 18　問 12 のスクリプト例

```
1 #!/bin/bash
2 echo -n "Please input something:"
3 read a
4 echo "Your input is ${a}"
```

1. ‘echo -n ”Please input something:”‘: ‘-n‘ オプションにより、末尾の改行なしで ”Please input something:” というプロンプトを表示する。
2. ‘read a‘: ユーザーからの入力を待ち、入力された行を変数 ‘a‘ に格納する。
3. ‘echo ”Your input is $a”‘: 変数 ‘a‘ の内容を含めて ”Your input is (入力された内容)” という文字列を表示する。

このスクリプトは、ユーザーに何らかの入力を促し、入力された文字列をそのまま表示する簡単な対話処理を行う。

問 13 - ヒアドキュメント

‘cat ¡¡END ... END‘ の形式はヒアドキュメントと呼ばれる。これは、スクリプト内に複数行のテキストを直接記述し、それを標準入力としてコマンドに渡すための機能である。提示された例の動作は以下の通り。

ソースコード 19　問 13 のスクリプト例

```
1 cat <<END
2 This sentence will be
3 input through
4 standard input stream
5 END
```

‘cat‘ コマンドに対して、‘END‘ という区切り文字が現れるまでの間の全ての行 (‘This sentence will be‘, ‘input through‘, ‘standard input stream‘) が標準入力として渡される。その結果、‘cat‘ コマンドはこれらの行をそのまま標準出力に出力する。区切り文字 (‘END‘ の部分) は任意の文字列を指定でき、開始の区切り文字の行に他の文字があってはならず、終了の区切り文字は行頭になければならない（インデントされている場合は ‘¡¡-END‘ を使うことで先頭のタブを除去できる)。

問 14 - 算術演算 (expr, bc)

シェルスクリプトで算術演算を行うには、‘expr‘ コマンドや ‘bc‘ コマンドなどが利用される。

expr を使った例

ソースコード 20　問 14 expr のスクリプト例

```
1 #!/bin/bash
2 a=5
3 b=‘expr ${a} + 10‘ ; echo ${b}
4 c=‘expr ${b} \* 2‘ ; echo ${c}
```

- ‘a=5‘: 変数 ‘a‘ に ‘5‘ を代入。
- “b=‘expr $a + 10‘ “: ‘expr‘ コマンドを使って ‘$a‘ (5) と ‘10‘ を加算し、結果 (15) をコマンド置換で変数 ‘b‘ に代入。
- ‘echo $b‘: 変数 ‘b‘ の値 (15) を表示。
- “c=‘expr $b 2‘ “: ‘expr‘ コマンドを使って ‘$b‘ (15) を ‘2‘ 倍し、結果 (30) を変数 ‘c‘ に代入。乗算記号 ‘*‘ はシェルによってワイルドカードとして解釈されるのを防ぐため、‘‘ とエスケープする必要がある。
- ‘echo $c‘: 変数 ‘c‘ の値 (30) を表示。

‘expr‘ は整数演算のみをサポートし、演算子や数値はスペースで区切る必要がある。

bc を使った例

ソースコード 21　問 14 bc のスクリプト例

```
#!/bin/bash
a=‘echo "4*a(1.0)" | bc -l‘ ; echo ${a}
b=‘echo "c($a)" | bc -l‘ ; echo ${b}
```

‘bc‘ は高精度計算が可能な計算言語であり、浮動小数点演算や数学関数（‘-l‘ オプションで数学ライブラリを読み込む）も扱える。

- “a=‘echo ”4*a(1.0)” — bc -l‘ “: ‘echo‘ で文字列 ‘”4*a(1.0)”‘ を ‘bc‘ コマンドの標準入力に渡し、計算を実行。‘a(1.0)‘ は逆正接関数 arctan(1.0) を意味し、その値は $\pi/4$ である。したがって、$4 \times (\pi/4) = \pi$ が計算され、その結果 (例: 3.14159...) が変数 ‘a‘ に代入される。
- ‘echo $a‘: 変数 ‘a‘ の値 (円周率 π) を表示。
- “b=‘echo ”c($a)” — bc -l‘ “: ‘echo‘ で文字列 ‘”c($a)”‘ を ‘bc‘ コマンドの標準入力に渡し、計算を実行。‘c($a)‘ は余弦関数 $\cos(a)$ を意味し、a には前のステップで計算された π が入るので $\cos(\pi) = -1$ が計算され、変数 ‘b‘ に代入される。
- ‘echo $b‘: 変数 ‘b‘ の値 (-1) を表示。

考察: ‘expr‘ は簡単な整数演算に、‘bc‘ は浮動小数点数を含む複雑な計算や数学関数を利用する場合に適している。‘bc‘ を利用する際は ‘-l‘ オプションで数学ライブラリを読み込むと便利である。

問 15 - シェル関数定義

シェルスクリプト内で一連の処理をまとめて名前を付け、再利用可能にする仕組みがシェル関数である。定義の基本的な形式は以下の通り。

ソースコード 22　問 15 シェル関数定義の形式

```
functionName () {
    commands
}
```

または

```
function functionName {
    commands
}
```

- ‘functionName‘: 関数の名前。

- ‘()‘: 引数を取ることを示す括弧（省略可能な場合もあるが、POSIX 準拠のためには付けることが推奨される）。
- ‘‘: コマンド群を囲む波括弧。波括弧とコマンドの間にはスペースや改行が必要。
- ‘commands‘: 関数内で実行される一連のコマンド。

関数内で引数を扱う場合は、スクリプト全体への引数と同様に ‘1‘, ‘2‘, ... を使用する。関数内で ‘local‘ キーワードを使って変数を宣言すると、その変数は関数内でのみ有効なローカル変数となる。定義された関数は、スクリプト内でその名前を記述することで呼び出すことができる。

提示されたスクリプトは ‘mawaru‘ という関数を定義し、それを呼び出している。

ソースコード 23　問 15 のスクリプト例

```
#!/bin/bash
mawaru () {
    local i=1
    while [ $i -le 10 ]
    do
        echo $i
        i=‘expr $i + 1‘
    done
}
mawaru
```

このスクリプトは、関数 ‘mawaru‘ を定義し、それを呼び出している。関数 ‘mawaru‘ は、ローカル変数 ‘i‘ を 1 から 10 までインクリメントしながらその値を表示する。実行結果は 1 から 10 までの数字が 1 行ずつ表示される。考察: 想定通りの結果であった。‘expr‘ の代わりに ‘i=$((i + 1))‘ を使うとより簡潔に書ける。

問 16 - if 条件分岐

‘if‘ 文は、条件に応じて処理を分岐させるための制御構造である。基本的な構文は以下の通り。

ソースコード 24　問 16 if 文の構文

```
if cond1
then
    com1
elif cond2
then
    com2
else
    comN
fi
```

- ‘if cond1‘: ‘cond1‘ (コマンド) を実行し、その終了ステータスが 0（成功）であれば ‘then‘ 以下の ‘com1‘ を実行する。
- ‘elif cond2‘: ‘cond1‘ が偽の場合に ‘cond2‘ (コマンド) を実行し、その終了ステータスが 0 であれば ‘then‘ 以下の ‘com2‘ を実行する。‘elif‘ は複数記述可能。
- ‘else‘: いずれの ‘if‘ または ‘elif‘ の条件も満たされなかった場合に ‘comN‘ を実行する。‘else‘ は省略可能。
- ‘fi‘: ‘if‘ 文の終わりを示す。

条件式 ‘cond‘ には ‘test‘ コマンド（または ‘[‘ コマンド）がよく用いられる。‘test‘ コマンドはファイルの種類、パーミッション、文字列比較、数値比較など様々な条件を評価できる。提示されたスクリプトの動作は以下の通り。

ソースコード 25　問 16 のスクリプト例

```
1 #!/bin/bash
2 fn=~/.bashrc
3 if [ -e ${fn} ]
4 then
5     cat ${fn}
6 else
7     echo "You has no ${fn}."
8 fi
```

1. ‘fn= /.bashrc‘: 変数 ‘fn‘ に ‘ /.bashrc‘ (ホームディレクトリの ‘.bashrc‘ ファイル) のパスを代入。
2. ‘if [-e $fn]‘: ‘test -e $fn‘ (または ‘[-e $fn]‘) で、ファイル ‘$fn‘ が存在するかどうかをチェックする。
3. ‘then cat $fn‘: ファイルが存在する場合（条件が真の場合)、‘cat $fn‘ コマンドを実行し、ファイルの内容を表示する。
4. ‘else echo ”You has no $fn.”‘: ファイルが存在しない場合（条件が偽の場合)、”You has no (ファイルパス).” というメッセージを表示する。
5. ‘fi‘: ‘if‘ 文の終了。

このスクリプトは、‘ /.bashrc‘ ファイルが存在すればその内容を表示し、存在しなければその旨を伝えるメッセージを表示する。考察: 想定通りの結果であった。‘test‘ コマンドの ‘-e‘ はファイルが存在するかどうかを調べるオプションとしてよく使われる。

演習 4 - 算術演算スクリプト

問 14 で扱った ‘expr‘ や ‘bc‘ を用いた算術演算のスクリプトを作成・実行する演習。

expr を用いたスクリプト

ソースコード 26　演習 4 expr のスクリプト例

```
#!/bin/bash
# Exercise 4: expr
val1=20
val2=5
sum_val=$(expr $val1 + $val2)
diff_val=$(expr $val1 - $val2)
prod_val=$(expr $val1 \* $val2)
quot_val=$(expr $val1 / $val2)

echo "Using expr:"
echo "$val1 + $val2 = $sum_val"
echo "$val1 - $val2 = $diff_val"
echo "$val1 * $val2 = $prod_val"
echo "$val1 / $val2 = $quot_val"
```

実行結果の例:

```
Using expr:
20 + 5 = 25
20 - 5 = 15
20 * 5 = 100
20 / 5 = 4
```

考察: ‘expr‘ を用いた基本的な四則演算が正しく実行された。乗算記号 ‘*‘ のエスケープが必要な点に注意する。

bc を用いたスクリプト (浮動小数点とべき乗)

ソースコード 27　演習 4 bc のスクリプト例

```
#!/bin/bash
# Exercise 4: bc
val_float1=3.14
val_float2=2.5
power_base=2
power_exp=10

sum_float=$(echo "$val_float1 + $val_float2" | bc)
prod_float=$(echo "$val_float1 * $val_float2" | bc)
```

```
power_val=$(echo "$power_base^$power_exp" | bc)
sqrt_val=$(echo "sqrt(16)" | bc -l) # -l for math library

echo "Using bc:"
echo "$val_float1 + $val_float2 = $sum_float"
echo "$val_float1 * $val_float2 = $prod_float"
echo "$power_base^$power_exp = $power_val"
echo "sqrt(16) = $sqrt_val"
```

実行結果の例:

```
Using bc:
3.14 + 2.5 = 5.64
3.14 * 2.5 = 7.850
2^10 = 1024
sqrt(16) = 4.00000000000000000000
```

考察: ‘bc‘ を用いることで浮動小数点演算やべき乗、平方根の計算が容易に行えることを確認した。数学関数を利用する際は ‘-l‘ オプションが必要である。

演習 5 - シェル関数スクリプト

問 15 で扱ったシェル関数を作成し、引数を渡して実行する演習。

ソースコード 28　演習 5 シェル関数のスクリプト例

```
#!/bin/bash
# Exercise 5: Shell Function with Arguments

greet_user() {
    if [ $# -eq 0 ]; then
        echo "Usage: $0 <name>"
        return 1
    fi
    local name=$1
    echo "Hello, ${name}! Welcome."
}

greet_user "Alice"
greet_user "Bob"
greet_user
```

実行結果の例:

```
Hello, Alice! Welcome.
Hello, Bob! Welcome.
Usage: ./script_name <name>
```

考察: シェル関数に引数を渡し、関数内で ‘$1‘ を使って引数を参照できることを確認した。また、引数の数を ‘$#‘ でチェックし、使用法を表示する簡単なエラーハンドリングも実装できた。

演習 6 - if 文スクリプト

問 16 で扱った ‘if‘ 文を用いて、数値比較を行うスクリプトを作成する演習。

ソースコード 29　演習 6 if 文のスクリプト例

```
1  #!/bin/bash
2  # Exercise 6: if statement for number comparison
3
4  read -p "Enter a number: " num
5
6  if [ -z "$num" ]; then
7      echo "No input provided."
8  elif ! [[ "$num" =~ ^[0-9]+$ ]]; then
9      echo "Invalid input: Not a number."
10 elif [ "$num" -lt 10 ]; then
11     echo "${num} is less than 10."
12 elif [ "$num" -gt 10 ]; then
13     echo "${num} is greater than 10."
14 else
15     echo "${num} is equal to 10."
16 fi
```

実行例 1 (入力: 5):

```
Enter a number: 5
5 is less than 10.
```

実行例 2 (入力: 15):

```
Enter a number: 15
15 is greater than 10.
```

実行例 3 (入力: abc):

```
Enter a number: abc
Invalid input: Not a number.
```

考察: ‘if-elif-else‘ 構造と数値比較演算子 (‘-lt‘, ‘-gt‘) を用いて、入力された数値に応じた処理の分岐ができた。入力値の検証 (空でないか、数値であるか) も追加した。

演習 7 - case 文スクリプト

ファイル拡張子に応じてメッセージを表示する ‘case‘ 文のスクリプト。

ソースコード 30　演習 7 case 文のスクリプト例 (PDF 記載例)

```
#!/bin/bash
echo -n "Please input filename:"
read fn
case ${fn} in
    *.txt)
        echo "${fn} is text file." ;;
    *.c)
        echo "${fn} is C program file." ;;
    *)
        echo "Hey, what is ${fn}? Do you know it?" ;;
esac
```

実行例 1 (入力: report.txt):

```
Please input filename:report.txt
report.txt is text file.
```

実行例 2 (入力: main.c):

```
Please input filename:main.c
main.c is C program file.
```

実行例 3 (入力: image.jpg):

```
Please input filename:image.jpg
Hey, what is image.jpg? Do you know it?
```

考察: ‘case‘ 文を用いることで、入力されたファイル名が特定のパターンに一致する場合に応じた処理を実行できた。‘*‘ はワイルドカードとして機能し、任意の文字列にマッチする。

演習 8 - for ループスクリプト

0 から 10 までの数値に対して処理を行う ‘for‘ ループのスクリプト。

ソースコード 31　演習 8 for ループのスクリプト例 (PDF 記載例)

```
#!/bin/bash
```

```
for x in 0 1 2 3 4 5 6 7 8 9 10
do
    v=‘echo "${x} * 0.31416" | bc -l‘
    w=‘echo "s(${v})" | bc -l‘
    echo "sin(${v}) = ${w}."
done
```

実行結果の例 (一部):

```
sin(0.00000) = .00000000000000000000.
sin(0.31416) = .30902182152311109099.
...
sin(3.14160) = -.00000732421875000000.
```

考察: ‘for‘ ループで指定したリストの各要素を変数 ‘x‘ に順次代入し、ループ内の処理を実行できた。‘bc -l‘ を用いて三角関数計算を行っている。

演習 9 - while ループスクリプト

条件が満たされる間、処理を繰り返す ‘while‘ ループのスクリプト。

ソースコード 32　演習 9 while ループのスクリプト例 (PDF 記載例)

```
#!/bin/bash
x=0
while [ ${x} -le 10 ]
do
    v=‘echo "${x} * 0.31416" | bc -l‘
    echo "sin(${v}) = " ‘echo "s(${v})" | bc -l‘.
    x=‘expr ${x} + 1‘
done
```

実行結果の例 (一部、演習 8 と同様):

```
sin(0.00000) = .00000000000000000000.
sin(0.31416) = .30902182152311109099.
...
sin(3.14160) = -.00000732421875000000.
```

考察: ‘while‘ ループで変数 ‘x‘ が 10 以下である間、ループ内の処理を繰り返し実行できた。カウンタ変数 ‘x‘ のインクリメントには ‘expr‘ を使用している。‘x=$((x + 1))‘ の方が現代的である。

演習 10 - break 文

ループ処理を途中で抜ける ‘break‘ 文の動作を確認する。

ソースコード 33　演習 10 break 文のスクリプト例

```
#!/bin/bash
# Exercise 10: break statement
echo "Counting from 1 to 10, but will break at 5."
count=1
while [ $count -le 10 ]; do
    echo "Count is: $count"
    if [ $count -eq 5 ]; then
        echo "Breaking loop at count = 5."
        break
    fi
    count=$((count + 1))
done
echo "Loop finished."
```

実行結果:

```
Counting from 1 to 10, but will break at 5.
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Breaking loop at count = 5.
Loop finished.
```

考察: ‘while‘ ループ内で ‘count‘ が 5 になった時点で ‘if‘ 条件が真となり ‘break‘ 文が実行され、ループが途中で終了することを確認した。

演習 11 - trap コマンド (SIGINT, SIGTERM)

シグナルを捕捉して特定の処理を行う ‘trap‘ コマンドの基本的な使い方。PDF 記載のスクリプト (e13) は以下の通り。

ソースコード 34　演習 11 trap コマンドのスクリプト例 (e13)

```
#!/bin/bash
trap 'echo =======I am trapped.======= ' SIGINT
```

```
x=1
while true # infinite loop!
do
    echo "${x}-th loop!!"
    x=‘expr ${x} + 1‘
done
```

このスクリプトを実行し、Ctrl+C を押すと ‘SIGINT‘ が送られ、‘trap‘ で設定したメッセージが表示される。ターミナルを 2 つ開き、片方でスクリプトを実行し、もう片方から ‘kill ¡PID¿‘ (SIGTERM) や ‘kill -INT ¡PID¿‘ (SIGINT) を送ることで動作を確認できる。考察: ‘trap‘ コマンドにより、‘SIGINT‘ シグナルを補足し、指定したコマンド (‘echo ...‘) が実行されることを確認した。‘SIGTERM‘ はデフォルトでは捕捉されないため、別途 ‘trap‘ で指定する必要がある。無限ループは ‘Ctrl+C‘ (SIGINT) では終了せず、‘kill -9 ¡PID¿‘ (SIGKILL) などで強制終了する必要がある場合がある。

演習 12 - expr と算術展開

‘expr‘ コマンドの代わりに bash の算術展開 ‘$(())‘ を使用する。

ソースコード 35　演習 12 算術展開のスクリプト例

```
#!/bin/bash
# Exercise 12: Arithmetic Expansion
a=10
b=20

# Using expr
sum_expr=$(expr $a + $b)
echo "Sum using expr: $sum_expr"

# Using arithmetic expansion
sum_arith=$((a + b))
echo "Sum using arithmetic expansion: $sum_arith"

product_arith=$((a * b))
echo "Product using arithmetic expansion: $a * $b = $product_arith"

# No need to escape * in arithmetic expansion
# Spaces around operators are optional but good for readability
diff_arith=$((b - a))
echo "Difference: $b - $a = $diff_arith"
```

実行結果:

```
Sum using expr: 30
Sum using arithmetic expansion: 30
Product using arithmetic expansion: 10 * 20 = 200
Difference: 20 - 10 = 10
```

考察: bash の算術展開 ‘$(())‘ は ‘expr‘ よりも簡潔に記述でき、‘*‘ のエスケープも不要であるため、可読性が向上する。整数演算に推奨される方法である。

演習 13 - trap のリセット

一度設定した ‘trap‘ をリセット（デフォルトの動作に戻す）する。PDF 記載のスクリプトは以下の通り。

ソースコード 36　演習 13 trap リセットのスクリプト例

```
1 #!/bin/bash
2 trap 'echo =======I am trapped.=======; trap SIGINT' SIGINT
3 x=1
4 while true # infinite loop!
5 do
6     echo "${x}-th loop!!"
7     x=‘expr ${x} + 1‘
8 done
```

このスクリプトでは、最初の ‘SIGINT‘ でメッセージが表示され、その後 ‘trap SIGINT‘ (引数なし) が実行される。これにより ‘SIGINT‘ に対するトラップがリセットされ、2 回目の ‘Ctrl+C‘ (SIGINT) ではスクリプトがデフォルト通り終了する。考察: ‘trap ¡シグナル名¿‘ のようにコマンドを指定せずに ‘trap‘ を実行すると、そのシグナルに対するトラップがリセットされることを確認した。

演習 14 - 複数シグナルの trap

複数のシグナルを一度に ‘trap‘ する。PDF 記載のスクリプトは以下の通り。

ソースコード 37　演習 14 複数シグナル trap のスクリプト例

```
1 #!/bin/bash
2 trap '/bin/rm tmp.$$; exit 1' SIGINT SIGTERM
3 x=1
4 while true # infinite loop!
5 do
6     echo "${x}-th loop!!"
7     echo "${x}-th loop!!" > tmp.$$
8     x=‘expr ${x} + 1‘
```

```
9 done
```

このスクリプトは ‘SIGINT‘ または ‘SIGTERM‘ を受信すると、‘tmp.(プロセス ID)‘ というファイルを削除し、終了コード 1 で終了する。‘$$‘ は現在のシェルのプロセス ID に展開される。考察: ‘trap ’ コマンド’ シグナル 1 シグナル 2 ...‘ のように記述することで、複数のシグナルに対して同じ処理を割り当てることができる。終了前に一時ファイルをクリーンアップするなどの処理に利用できる。

演習 15 - SIGKILL の trap 不可

‘SIGKILL‘ (シグナル番号 9) は ‘trap‘ できないことを確認する。

ソースコード 38　演習 15 SIGKILL trap 試行スクリプト例

```
1  #!/bin/bash
2  trap 'echo "SIGKILL trapped?"' SIGKILL
3  trap 'echo "SIGTERM trapped, exiting gracefully."; exit 0' SIGTERM
4  echo "Script running with PID: $$"
5  echo "Try killing with SIGTERM (kill $$) and SIGKILL (kill -9 $$)"
6  count=0
7  while true; do
8      echo "Looping... $count"
9      count=$((count + 1))
10     sleep 1
11 done
```

このスクリプトを実行し、別のターミナルから ‘kill ¡PID¿‘ (SIGTERM) を送ると ”SIGTERM trapped...” が表示されて終了する。しかし ‘kill -9 ¡PID¿‘ (SIGKILL) を送ると、”SIGKILL trapped?” は表示されず、スクリプトは即座に強制終了する。考察: ‘SIGKILL‘ と ‘SIGSTOP‘ は捕捉 (trap) したり無視したりすることができない特別なシグナルである。これらはプロセスを確実に終了させる（または停止させる）ために OS によって予約されている。

演習 16 - while true ループの終了

‘while true‘ で作成した無限ループをどのように終了させるかの考察。PDF のスクリプト例:

ソースコード 39　演習 16 while true スクリプト例

```
1  #!/bin/bash
2  trap 'echo ====I am trapped====' SIGINT SIGTERM
3  x=1
4  while true # infinite loop!
5  do
6      echo "${x}-th loop!!"
```

```
    x=‘expr ${x} + 1‘
done
```

このループは、‘SIGINT‘ (Ctrl+C) や ‘SIGTERM‘ (kill) を送っても、トラップメッセージが表示されるだけで終了しない。終了させる方法:

1. トラップ内で ‘exit‘ コマンドを実行するようにスクリプトを修正する。

```
trap ’echo ====I am trapped, exiting...====; exit 0’ SIGINT SIGTERM
```

2. スクリプトを修正せず外部から終了させる場合は、‘kill -9 ¡PID¿‘ (SIGKILL) を使用する。SIGKILL はトラップできないため、プロセスを強制終了できる。
3. ループ内に終了条件を設ける（例: 特定の回数実行したら ‘break‘ する、特定のファイルが作成されたら終了するなど）。

考察: ‘while true‘ のような無限ループは、意図的に終了処理を組み込むか、外部から強制終了シグナルを送らない限り終了しない。‘trap‘ でシグナルを捕捉している場合は、そのトラップ処理内で ‘exit‘ を呼ばない限り、捕捉したシグナルでは終了しない点に注意が必要。

演習 17 - 画像管理スクリプト

指定された画像フォルダ内の画像ファイルに対して、一覧表示、タグ付け、タグによる検索を行うシェルスクリプトを作成する。タグ情報は、画像ファイル名に追従する形で、例えば ‘画像ファイル名.tag‘ という別ファイルに保存する形式を想定する。

スクリプト (imgtool.sh)

ソースコード 40　演習 17 画像管理スクリプト ‘imgtool.sh‘

```
#!/bin/bash

IMAGE_DIR="${HOME}/images" # 画像が保存されているディレクトリ
TAG_EXT=".tag" # タグファイルの拡張子

# --- 関数定義 ---

# 画像一覧表示
list_images() {
    echo "--- Images in ${IMAGE_DIR} ---"
    ls -1 "${IMAGE_DIR}" | grep -E ’\.(jpg|jpeg|png|gif)$’
    echo "--------------------------"
}

```

```
15 # タグ付け（既存タグは上書き）
16 add_tag() {
17     if [ -z "$1" ] || [ -z "$2" ]; then
18         echo "Usage: $0 tag <image_file> <tag_string>"
19         return 1
20     fi
21     local image_file="$1"
22     local tag_string="$2"
23     local image_path="${IMAGE_DIR}/${image_file}"
24     local tag_file_path="${image_path}${TAG_EXT}"
25 
26     if [ ! -f "${image_path}" ]; then
27         echo "Error: Image file '${image_file}' not found in ${IMAGE_DIR}."
28         return 1
29     fi
30 
31     echo "${tag_string}" > "${tag_file_path}"
32     echo "Tagged '${image_file}' with '${tag_string}'."
33 }
34 
35 # タグ表示
36 show_tag() {
37     if [ -z "$1" ]; then
38         echo "Usage: $0 showtag <image_file>"
39         return 1
40     fi
41     local image_file="$1"
42     local image_path="${IMAGE_DIR}/${image_file}"
43     local tag_file_path="${image_path}${TAG_EXT}"
44 
45     if [ ! -f "${image_path}" ]; then
46         echo "Error: Image file '${image_file}' not found in ${IMAGE_DIR}."
47         return 1
48     fi
49 
50     if [ -f "${tag_file_path}" ]; then
51         local tag_content=$(cat "${tag_file_path}")
52         echo "Tags for '${image_file}': ${tag_content}"
53     else
54         echo "No tags found for '${image_file}'."
55     fi
56 }
```

```
57
58 # タグで検索
59 search_by_tag() {
60     if [ -z "$1" ]; then
61         echo "Usage: $0 search <search_tag>"
62         return 1
63     fi
64     local search_tag="$1"
65     echo "--- Images tagged with '${search_tag}' ---"
66     local found_count=0
67     for tag_file in "${IMAGE_DIR}"/*"${TAG_EXT}"; do
68         if [ -f "${tag_file}" ]; then
69             if grep -q -i "${search_tag}" "${tag_file}"; then
70                 local image_file=$(basename "${tag_file}" "${TAG_EXT}")
71                 echo "${image_file}"
72                 found_count=$((found_count + 1))
73             fi
74         fi
75     done
76     if [ "${found_count}" -eq 0 ]; then
77         echo "No images found with tag '${search_tag}'."
78     fi
79     echo "-------------------------------------"
80 }
81
82 # ヘルプ表示
83 show_help() {
84     echo "Image Management Tool"
85     echo "Usage: $0 <command> [options]"
86     echo ""
87     echo "Commands:"
88     echo " list List all image files."
89     echo " tag <image_file> <tags> Add/update tags for an image file."
90     echo " (e.g., $0 tag photo.jpg \"holiday sunset beach\")"
91     echo " showtag <image_file> Show tags for an image file."
92     echo " search <tag_keyword> Search images by a tag keyword."
93     echo " help Show this help message."
94     echo ""
95     echo "Note: Image directory is set to ${IMAGE_DIR}"
96 }
97
98 # --- メイン処理 ---
```

```
if [ -z "$1" ]; then
    show_help
    exit 1
fi

# IMAGE_DIR が存在しなければ作成
mkdir -p "${IMAGE_DIR}"

COMMAND="$1"
shift # コマンド名を除去して残りを引数とする

case "${COMMAND}" in
    list)
        list_images
        ;;
    tag)
        add_tag "$1" "$2"
        ;;
    showtag)
        show_tag "$1"
        ;;
    search)
        search_by_tag "$1"
        ;;
    help)
        show_help
        ;;
    *)
        echo "Error: Unknown command '${COMMAND}'."
        show_help
        exit 1
        ;;
esac

exit 0
```

スクリプトの解説

このスクリプト ‘imgtool.sh‘ は、指定された画像ディレクトリ (デフォルトは ‘ /images‘) 内の画像ファイルに対して、以下の操作を行うツールである。

- **環境変数・定数**:

  - ‘IMAGE_DIR‘: 画像が保存されているディレクトリを指定。デフォルトはユーザのホームディレクトリ下の ‘images‘。スクリプト実行時にこのディレクトリがなければ作成される。
  - ‘TAG_EXT‘: タグ情報を保存するファイルの拡張子。デフォルトは ‘.tag‘。
- **関数**:
  - ‘list_images‘: ‘IMAGE_DIR‘ 内の画像ファイル (jpg, jpeg, png, gif) を一覧表示する。
  - ‘add_tag ¡image_file¿ ¡tag_string¿‘: 指定された画像ファイルにタグを付ける。タグはスペース区切りで複数指定可能（ダブルクォートで囲む）。タグ情報は ‘画像ファイル名.tag‘ というファイルに保存される。既存のタグは上書きされる。
  - ‘show_tag ¡image_file¿‘: 指定された画像ファイルのタグ情報を表示する。
  - ‘search_by_tag ¡search_tag¿‘: 指定されたタグ（キーワード）を含む画像ファイルを検索し、一覧表示する。検索はタグファイルの内容に対して行われ、大文字・小文字を区別しない。
  - ‘show_help‘: スクリプトの使用方法を表示する。
- **メイン処理**:
  - スクリプト実行時の最初の引数をコマンドとして解釈する。
  - ‘case‘ 文を用いて、指定されたコマンドに応じた関数を呼び出す。
  - コマンドが指定されない場合や未知のコマンドの場合はヘルプメッセージを表示して終了する。

スクリプトは実行権限 (‘chmod +x imgtool.sh‘) を与えた後、‘./imgtool.sh ¡command¿ [options]‘ のようにして使用する。

実行結果の例

まず、画像ディレクトリとサンプルファイルを作成する。

```
$ mkdir ~/images
$ touch ~/images/photo1.jpg
$ touch ~/images/photo2.png
$ touch ~/images/document.txt # 画像ではないファイル
```

スクリプトを実行する。

```
$ ./imgtool.sh list
--- Images in /home/user/images ---
photo1.jpg
photo2.png
-------------------------
```

```
$ ./imgtool.sh tag photo1.jpg "cat cute animal"
Tagged 'photo1.jpg' with 'cat cute animal'.

$ ./imgtool.sh tag photo2.png "dog park sunny"
Tagged 'photo2.png' with 'dog park sunny'.

$ ./imgtool.sh showtag photo1.jpg
Tags for 'photo1.jpg': cat cute animal

$ ./imgtool.sh showtag photo2.png
Tags for 'photo2.png': dog park sunny

$ ./imgtool.sh search cat
--- Images tagged with 'cat' ---
photo1.jpg
-----------------------------------

$ ./imgtool.sh search sunny
--- Images tagged with 'sunny' ---
photo2.png
-----------------------------------

$ ./imgtool.sh search holiday
--- Images tagged with 'holiday' ---
No images found with tag 'holiday'.
-----------------------------------

$ ./imgtool.sh tag photo1.jpg "cat fluffy indoor" # タグの上書き
Tagged 'photo1.jpg' with 'cat fluffy indoor'.

$ ./imgtool.sh showtag photo1.jpg
Tags for 'photo1.jpg': cat fluffy indoor

$ ./imgtool.sh help
Image Management Tool
Usage: ./imgtool.sh <command> [options]
...（ヘルプメッセージが表示される）...
```

考察

このスクリプトは、基本的なファイル操作、文字列操作、条件分岐、ループ、関数といったシェルスクリプトの要素を組み合わせて作成されている。タグ情報を画像ファイルとは別の ‘.tag‘ ファイルに保存する方式はシンプルで実装が容易である。改善点としては、以下のようなものが考えられる。

- タグの追加・削除を個別に行えるようにする（現在は上書きのみ）。
- 複数のタグで AND/OR 検索できるようにする。
- タグ情報を SQLite などの軽量データベースに保存することで、より高度な検索や管理を可能にする。
- 画像ファイルの存在確認をより厳密に行う（‘grep‘ での拡張子フィルタリングだけでなく、‘file‘ コマンドで MIME タイプを確認するなど）。

演習の範囲内では、指定された機能をシェルスクリプトで実現できた。

演習 18 - SIGTERM による終了処理 (ボーナス)

‘trap‘ を使用して ‘SIGTERM‘ シグナルを捕捉し、終了前に特定の処理（例えば一時ファイルの削除）を実行するスクリプトを作成する。PDF の最後のスクリプト例 (‘trap ’/bin/rm tmp.$$; exit 1’ SIGINT SIGTERM‘) がこれに該当するが、ここでは ‘SIGTERM‘ に特化し、より明確なクリーンアップ処理とメッセージを加えてみる。

スクリプト例

ソースコード 41　演習 18 SIGTERM による終了処理スクリプト

```
1  #!/bin/bash
2
3  TEMP_FILE="/tmp/myscript_temp_$$"
4
5  # SIGTERM シグナルを捕捉し、クリーンアップ関数を呼び出す
6  trap 'cleanup_and_exit' SIGTERM
7  # SIGINT (Ctrl+C) も同様に処理する
8  trap 'cleanup_and_exit' SIGINT
9
10 cleanup_and_exit() {
11     echo "" # 改行
12     echo "Signal received. Cleaning up temporary file: ${TEMP_FILE}"
13     if [ -f "${TEMP_FILE}" ]; then
14         rm -f "${TEMP_FILE}"
15         echo "Temporary file removed."
```

```
16      else
17          echo "No temporary file to remove."
18      fi
19      echo "Exiting gracefully."
20      exit 0 # 正常終了
21  }
22
23  echo "Script started. PID: $$"
24  echo "Creating temporary file: ${TEMP_FILE}"
25  date > "${TEMP_FILE}"
26  echo "Temporary file created. Content: $(cat ${TEMP_FILE})"
27  echo "Script is running. Send SIGTERM (kill $$) or press Ctrl+C to test
        cleanup."
28
29  E スクリプトが何らかの処理を続けることを模倣
30  count=0
31  while true; do
32      echo "Working... ($((count+=1))) - Temp file exists: $(if [ -f "
            $TEMP_FILE" ]; then echo "Yes"; else echo "No"; fi)"
33      sleep 2
34  done
```

スクリプトの解説

- ‘TEMP_FILE‘: スクリプトが使用する一時ファイルのパスを定義。‘$$‘ は現在のシェルのプロセス ID に展開され、ユニークなファイル名を生成するのに役立つ。
- ‘trap ’cleanup_and_exit’ SIGTERM SIGINT‘: ‘SIGTERM‘ または ‘SIGINT‘ シグナルを受信した際に ‘cleanup_and_exit‘ 関数を呼び出すように設定する。
- ‘cleanup_and_exit()‘:
  - 終了シグナル受信のメッセージを表示する。
  - ‘TEMP_FILE‘ が存在すれば削除し、その旨を通知する。
  - 正常終了のメッセージを表示し、‘exit 0‘ でスクリプトを終了する。
- スクリプトのメイン部分:
  - スクリプトの開始と PID を表示する。
  - 一時ファイルを作成し、その内容を表示する。
  - ユーザーにテスト方法を案内する。
  - ‘while true‘ ループで、スクリプトが何らかの作業を継続していることを模倣する。ループ内では一時ファイルの存在状況も表示する。

実行結果の例

1. スクリプトを実行する (‘./script_name.sh‘)。

```
$ ./sigterm_cleanup.sh
Script started. PID: 12345
Creating temporary file: /tmp/myscript_temp_12345
Temporary file created. Content: (現在の日時)
Script is running. Send SIGTERM (kill 12345) or press Ctrl+C to test cleanup.
Working... (1) - Temp file exists: Yes
Working... (2) - Temp file exists: Yes
...
```

2. 別のターミナルから ‘kill 12345‘ (PID は実際の値に置き換える) を実行するか、スクリプト実行中のターミナルで ‘Ctrl+C‘ を押す。スクリプト実行中のターミナルに以下のような出力が表示される。

```
Working... (N) - Temp file exists: Yes

Signal received. Cleaning up temporary file: /tmp/myscript_temp_12345
Temporary file removed.
Exiting gracefully.
$ # プロンプトに戻る
```

この後、‘/tmp/myscript_temp_12345‘ ファイルが削除されていることを確認できる。

考察

‘trap‘ を用いることで、‘SIGTERM‘ や ‘SIGINT‘ といった終了シグナルを捕捉し、スクリプトが中断される前に指定したクリーンアップ処理（一時ファイルの削除など）を実行できることを確認した。これにより、予期せぬ中断が発生した場合でも、システムに不要なファイルを残さず、より安全にスクリプトを終了させることができる。‘exit‘ コマンドをトラップ処理の最後に含めることで、クリーンアップ後にスクリプトを確実に終了させることが重要である。この演習は、堅牢なシェルスクリプトを作成する上で非常に有用なテクニックを示している。