第 1 回 プログラミング基礎ゼミ

ゼミ担当者　：　山本啓二，狩野浩一，下村大輔
指導院生　　：　實田健，澤田淳二，金美和
開催日　　：　2003 年 5 月 20 日

ゼミ内容: 本ゼミでは，研究を快適に進めるためのプログラミングについて紹介します．研究室では，Cambria や Gregor といった PC クラスタを用いて多くの計算を行うため，計算結果をまとめ，編集する方法を習得することはとても有益です．詳しい内容としては，Makefile の作り方，シェルスクリプトの書き方，awk，Perl によるデータの整理についての実習を行います．

1　make

make はプログラムのコンパイルを自動化するツールです．複数のソースファイルから目的のファイルを作る際，変更のあったファイルだけを再コンパイルすることができます．

例えば file1.c，file2.c，　file3.c という 3 つのソースファイルから実行ファイルを作るとします．　file1.c を変更して再コンパイルする場合，　file2.c および file3.c は再コンパイルする必要はありません．make はこのような処理を自動化してくれます．

make は，ファイルの依存関係を示した Makefile というファイルをもとに処理を行ないます．Makefile をもとに依存ファイルの更新日付を調べ，ある依存ファイルを修正したとき，最小限のコンパイルをおこないます．

make で生成するのは普通，C のプログラムですが，特に C のプログラムに限らず Makefile に書く生成コマンドの書きようによっては，TeX のファイルや，Java のプログラムなど make を利用して作業を軽減することができます．

1.1　Makefile の作り方

make を使うには，前述したように “Makefile” を記述する必要があります．Makefile は，

```
ターゲット:依存ファイル群…
    作成方法
```

のように，作りたいファイル（ターゲットと呼びます）とその構成ファイル（依存ファイルと呼びます）を “:” で区切って記述します．ターゲットの指定は必ず行頭から書いてください．次の行には，ターゲットの作成方法を記述します．この行は，必ずタブで先頭に空白を空けて記述します．作成方法は複数行になっても構いません．次のターゲット定義まではすべて作成方法として扱われます．

依存ファイルのどれかのタイムスタンプがターゲットよりも新しい場合に，依存ファイルが更新されたと判断され，ターゲットが作り直されます．

では，次のような C のソースファイルがあるとして，これらから実行可能ファイル exec.out を生成する Makefile を書くとします．

```
src1.c
src2.c
header.h

header.h は src1.c, src2.c にそれぞれ include されているものとする
exec.out は src1.o と src2.o (それぞれ src1.c, src2.c をコンパイルしたオブジェクトファイル) をリンクしてできる実行可能ファイルとする．
```

まず exec.out は src1.c と src2.c に依存しているので，src1.c，src2.c のどちらかが更新された場合には，exec.out を再リンクする必要があります．さらに src1.c，src2.c は両方へッダファイル header.h に依存しています．したがって，header.h が更新された場合には，src1.c，src2.c ともに再コンパイルする必要があります．当然，それらの影響を受けて exec.out も再度リンクしなければなりません．

以上のような状況で，Makefile を書くと Fig. 1 のようになります．

```
exec.out: src1.o src2.o
        cc -o exec.out src1.o src2.o
src1.o: src1.c header.h
        cc -c src1.c
src2.o: src2.c header.h
        cc -c src2.c
```

Fig. 1　Makefile の内容

Fig. 1 で，src1.o, src2.o は header.h に依存している

ので，src1.o と src2.o のための規則は header.h が更新されると実行され，src1.c と src2.c を自動的に再コンパイルします．

1.2 make の実行

make を実行するには

```
$make
```

とだけタイプすることで Makefile を読み込み，ターゲットの生成を開始します．このように何も指定せずに make を実行すると，make はその最終目標として Makefile の一番最初に書かれているターゲットを生成しようとします．例えば Fig. 1 の Makefile では，make は exec.out というターゲットを生成しようとします．生成するターゲットを指定して make を実行するには

```
$make src1.o
```

のように実行します．この場合，make は Makefile の一番最初に書いてあるターゲットのかわりに，“src1.o” というターゲットを生成します．

2 シェルスクリプト

一連のコマンドを何度も使う場合，それらをファイルに記述しシェルに解析・実行させることができます．コマンドが記述されているファイルのことをシェルスクリプトといいます．シェルスクリプトは通常のファイルと同様に，任意のディレクトリ下に作成することができます．また，シェルスクリプトはテキストファイルであるためエディタ等で作成できます．シェルスクリプトの実行には，次の 2 つの方法があります．

```
$sh シェルスクリプト名
$シェルスクリプト名 （実行許可にしておく）
```

シェルには Table 1 に示すように様々な種類がありますが，移植性を考慮して通常 sh を用います．

2.1 シェルスクリプトの記述

簡単な例として，現在の日時とカレントディレクトリを表示するシェルスクリプトを Fig. 2 に示します．

```
#!/bin/sh
# date command run.
date
# pwd command run.
pwd
```

Fig. 2 シェルスクリプトの例

Table 1 シェルの種類

sh	最初に使われはじめたので，どの UNIX にも必ず乗っている
csh	プログラミン機能が C 言語に似ているため csh と呼ばれる．インターフェイス機能が評価されている．
tcsh	csh の全ての機能を持ちさらにコマンドライン編集機能やスペル訂正機能，コマンド補完機能などを持つ．
ksh	ベル研の David Korn により開発されたシェル．フリーではない．
bash	sh と互換性を持たせ，機能を強化したシェル．Linux の標準シェルになっている．
zsh	bash 互換で色々なシェルの機能を取り入れさらに独自機能追加をしたシェル．最も新しいシェルである．

1 行目は，このスクリプトを実行するのに使用するプログラムを指定します．ここでは “/bin/sh” を指定していますが，システムによって違う場合もあります．

“#” から行末まではコメントと見なされ，このスクリプトは “date” と “pwd” を続けて実行します．

このシェルスクリプトを実行するには，ファイルに実行許可を与える必要があります．通常以下のコマンドで実行許可を与えることができます．

```
$chmod u+x シェルスクリプト名
```

シェルスクリプトでは制御構文や引数渡し等を組み合わせると，簡単に多彩な作業を自動的に行えます．

2.2 制御構文

シェルの制御構文には if，for，case，select，while，until 等がありますが，ここでは if，for，while 文について説明します．

まず，if 文の構造を以下に示します．

```
if 条件; then
     コマンド
     .....
elif 条件; then
     コマンド
     .....
else
     コマンド
     .....
fi
```

条件の書き方ですが，“[” と “]” で Table 2 に示す評価式を囲みます．

Table 2　評価式の種類

A = B	A と B が同じ文字列ならば 0 を返す
A != B	A と B が違う文字列ならば 0 を返す
A -eq B	A ＝ B ならば 0 を返す
A -ne B	A ≠ B ならば 0 を返す
A -lt B	A ＜ B ならば 0 を返す
A -le B	A ≦ B ならば 0 を返す
A -gt B	A ＞ B ならば 0 を返す
A -ge B	A ≧ B ならば 0 を返す
-f A	ファイル A が存在するならば 0 を返す

Fig. 3 に if 文に関するサンプルを示します．

```
#!/bin/sh
if [ $1 = "aiueo" ]; then
     echo "引数は aiueo です．"
elif [ $1 = "argv" ]; then
     echo "引数は argv です．"
else
     echo "引数が渡されました．"
fi
```

Fig. 3　if 文の例

引数とは，実行するスクリプトに外部から与えられたパラメータのようなもので，

```
$./sample.sh abc 123 ggg
```

のとき，シェルスクリプト内の$1 には abc が，$2 には 123 が，$3 には ggg が代入されています．

次に for 文の構造を示します．

```
for 変数 in リスト
do
     コマンド
done
```

Fig. 4 に for 文のサンプルを示します．

次に while 文の構造を示します．

```
#!/bin/sh
for var in a b c d
do
    echo $var
done
```

Fig. 4　for 文の例

```
while 条件;
do
     コマンド
done
```

Fig. 5 に while 文のサンプルを示します．

```
#!/bin/sh

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

Fig. 5　while 文の例

Bourne シェルは，算術演算の機能を自分では持っていません．ここでは，それを実現するために，test と expr という外部コマンドを呼び出しています．test の方は “[” と “]” です．“[” と “]” は test コマンドの省略形になっています．ループ変数が 10 以下かどうかの判定は test コマンドに，ループ変数のインクリメントは expr コマンドに肩代わりしてもらっています．

3　awk

3.1　awk とは

awk とはフィルタの一種で，パターンを検索し，それに対して対応する処理を行なうものです．よく似たフィルタとして grep が挙げられますが， awk はより強力になっています．例えば，awk は計算機能をもっていて簡単な表計算に似たようなこともできます．awk は A.V.Aho, P.J.Weinberger, B.W.Kernighan の三人によって作られたので頭文字をとって名前がつけられました．

awk は基本的に，処理するファイルを 1 行ずつ読み込み，それに対して，スクリプトによって記述された処理

を行なう，という動作をします．処理単位を読み込んだときに，指定されたフィールドセパレータによって複数のフィールドに自動的に分割されるのが awk の特徴で，これにより，たとえば，

```
名前      住所
名前      住所
 ・        ・
```

といったようなある程度構造化されたデータに対してデータベース的な処理を容易に適用することが可能となっています．

3.2 awk の書式

awk の基本書式は次の 2 つです．

- awk ’手続き’ 対象ファイル
- awk -f 手続きファイル 対象ファイル

’手続き’ とは実行されるべきパターン検索と処理で，対象ファイルとはその検索と処理の対象となるファイルです．また，’手続き’ には，シェルのメタキャラクタが含まれることがあるので，全体を ’（シングルクォート）で囲むほうがよいです．

基本書式の後者の場合には，エディタで “手続きファイル” をあらかじめ別に作っておき，それを指定して使います． -f は “手続きファイル” を使うことを意味します．この場合には “手続きファイル” 内にシェルのメタキャラクタが含まれていたとしても ’ で囲む必要はありません．

awk の ’手続き’ 部分は基本的には

```
パターン１   {アクション 1}
パターン２   {アクション 2}
パターン３   {アクション 3}
    ・           ・
    ・           ・
```

の形式をしています．

先ほど述べたように対象ファイルを一行づつ読み込んで，それぞれのパターンに一致しているかを調べ，一致していれば対応するアクションを実行します．ここでいう行とは，通常は改行コードによって区切られたものであり，レコードとも呼ばれます．

パターンとアクションのどちらかは省略してもかまいません．もしアクションを省略すれば，一致した行を表示するようになります．逆にパターンを省略すれば，アクションはすべての行に対して実行されるようになります．

たとえば

```
awk ’/^[A-Z]/’ file1
```

は，ファイル file1 の内容のうち大文字で始まる行を表示することになります．

これは

```
awk ’/^[A-Z]/ {print}’ file1
```

としても同様の実行結果となります．

また

```
awk ’{print}’ file1
```

はすべての行を表示しますので，

```
cat file1
```

と同じ事になります．

3.3 フィールド

awk は，レコード（行）を処理単位としますが，この行は空白などで区切られたより小さなフィールドという単位に分割されます．フィールドとはブランクやタブで区切られたブランクを含まない文字列を意味します．たとえば，

I think therefore I am . は，6 つのフィールドを含むことになります．”I” は第 1 フィールド，”think” は第 2 フィールド，”therefore” は第 3 フィールド，”I” は第 4 フィールド，”am” は第 5 フィールド，”.” は第 6 フィールドとなり，それぞれ $1, $2, $3, $4, $5, $6 と表わします．また現レコードの内容全体を $0 として表現します．

そのため

```
admin    tty1 Jul 28 10:23
user     tty2 Jul 29 13:22
<--$1--> <$2><$3><$4><$5>
<-----------$0----------->
```

というように構成されます．

$1, $2 等はシェルのメタキャラクタとして引数の指定に使用されましたが， awk の ’手続き’ 部分で使用されると， awk 独特の表現としてフィールド指定に使用されるので混同しないでください．

awk の使用例を以下に示します．

```
$ps ax
PID TTY STAT TIME COMMAND
360 ?   S    0:00 /usr/sbin/inetd
364 ?   S    0:00 qmail-send
383 ?   S    0:04 /usr/sbin/sshd
421 ?   S    0:00 /usr/sbin/apache

$ps ax | awk '{print $1,$5}'
PID COMMAND
360 /usr/sbin/inetd
364 qmail-send
383 /usr/sbin/sshd
421 /usr/sbin/apache

$ls -l /etc | awk '$5<1024 {print $3,$5,$9}'
root 47 adjtime
root 672 aliases.old
root 144 at.deny
root 211 bash.bashrc
root 640 crontab
```

3.4 特別なパターン BEGIN と END

BEGIN と END は特別なパターンです．これらは入力レコードの照合には使われず，スタートアップ情報またはクリーンナップ情報を awk スクリプトに提供するために使われます．BEGIN ルールは最初の入力レコードが読み込まれる前に一度実行されます．END ルールはすべての入力が読み込まれたあとに一度実行されます．

次に例をあげます．

```
awk 'BEGIN { print "begin awk" }
           { i++ }
     END   { print "end awk";print i }'
```

このスクリプトは、標準入力の行数をカウントして表示します．ここでは BEGIN ルールを用いて “begin awk” という文字列を表示しています．2 番目のルールは、レコードが読み込まれるたびに、変数 i をインクリメントします．変数の初期化は awk によって自動的に行われるので，BEGIN ルールを用いて初期化する必要はありません．END ルールでは，“end awk” という文字列と i の値を表示します．

```
$ls -l /etc | \
>awk 'BEGIN {print "start"} \
>{i++} \
>END {print "end";print i}'
start
end
122
```

4 Perl

Perl は awk と同じようにテキスト処理に適したツールといえます．Perl は非常に高機能なので，awk ではできない複雑な操作も行うことができます．

Perl とは Larry Wall によって開発されたインタープリタ型のプログラミング言語の名称です．

Perl はもともと，テキスト処理に重点をおいた言語として作成されましたが，その後，次々と言語仕様が拡張され，ファイル操作を始め，システムプログラミングやネットワークプログラミングなども可能な，多くの機能を備えるに至っています．

プログラミング言語としては，非常に強力なテキスト処理，ファイル処理機能を備えていること，awk でできることのほとんどが Perl で記述可能であること，基本的にインタープリタであることなどから，awk の延長線上にある言語として捉えることができます．

最近では，Web において，ユーザーからの入力を受け取りダイナミックなページを作成するようなシステムが，Perl スクリプトを利用して構築されたり，Perl で記述されたスクリプトそのものが，インターネットなどで公開されさたりするなど，汎用プログラミング言語として広く利用されています．

参考として，awk で行った処理を perl で行うスクリプトを示します．

`awk '{print $1,$5}'` に対応する perl スクリプト

```
#!/usr/local/bin/perl
while(<>)
{
    @ary = split ' ';
    print "$ary[0] $ary[4]\n";
}
```

`awk '$5<1024 {print $3,$5,$9}'` に対応する perl スクリプト

```
#!/usr/local/bin/perl
while(<>)
{
    @ary = split ' ';
    if($ary[4] < 1024){
        print "$ary[2] $ary[4] $ary[8]\n";
    }
}
```