K.K, totera 氏に捧ぐ (??)

GNU Make でも使ってみようか

北海道大学 理学院 宇宙理学専攻
博士課程 2 年　森川 靖大

発表の前に

- 結構細かい時が多いので前列でご拝聴下さい
- 本日は一応，「make って使ったことあるけど，ちょっと難しそうで書くのは...」というmake初級者向けなつもりです．
- make って聞いたこともない皆様
  - はじめからどんどん質問してね
- make 初級者の皆様
  - 期待しないで質問してください．
- 中・上級者のみなさま
  - コメントお待ちしてます．

目次

- 最低限 make
- make のすごいとこ
- 現役 make さん見学
- make 応用編
- make 落ち穂拾い

最低限 make

Make って何??

- 「メイク」と呼びます．(和訳：作る)
- プログラムのコンパイル，リンク，インストール作業を自動化してくれるツール
  - 見たこと無いかな??

```
$ make
# make install
```

- 上記作業の時間を劇的に短縮
  ⇒ ソフトウェア開発の大きな助けに
  - その他にもいろいろ応用できます (後述)

種類いろいろ

- Make いろいろ
  - System V Make
    - おそらく一番原始的な make. UNIX 上で動作.
  - mk, nmake, BSD Make, SunOS Make, SGI Make, ...
  - GNU Make (gmake)
    - GNU プロジェクトの一環として標準版 make を模倣
      - 「あきれるほど徹底的に書き直された (Andrew 他, 1997)」らしい
    - 豊富な拡張機能
    - GNU 系の OS だけでなく, UNIX, Windows, Mac 上で使える.
    - もちろん GNU ライセンス
- 今回は GNU Make を念頭にお話します.

御託はいいから使ってみよう

- 作業の流れ
  - GNU make のコマンド名の確認
  - C や Fortran のソースコードを準備
  - Makefile を書いてみる
  - いざ, make 実行!!

御託はいいから使ってみよう

- GNU make のコマンド名の確認

```
$ make -v

GNU Make 3.80
Copyright (C) 2002  Free Software ...
This is free software; see the source ..
There is NO warranty; not even for ...
PARTICULAR PURPOSE.
```

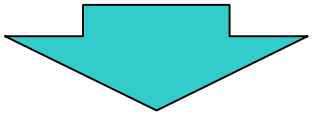
make コマンドが GNU make であることを確認

- もし make が GNU make で無いときは...
  - “gmake” を代わりに使ってみる
  - chkgmake.sh (自作スクリプト) を使って調べる.

御託はいいから使ってみよう

- Fortran のソースコードを準備

```
$ vi main.f90
```



main.f90

```
program main
  print *, "Hello, World!"
end program main
```

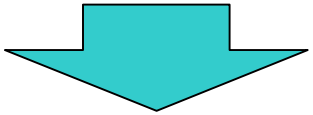
- ちなみに，手動でコンパイルする場合は…

```
$ g95 main.f90 -o main
$ ./main
 Hello, World!
```

御託はいいから使ってみよう

- Makefile を書いてみる

```
$ vi Makefile
```



Makefile

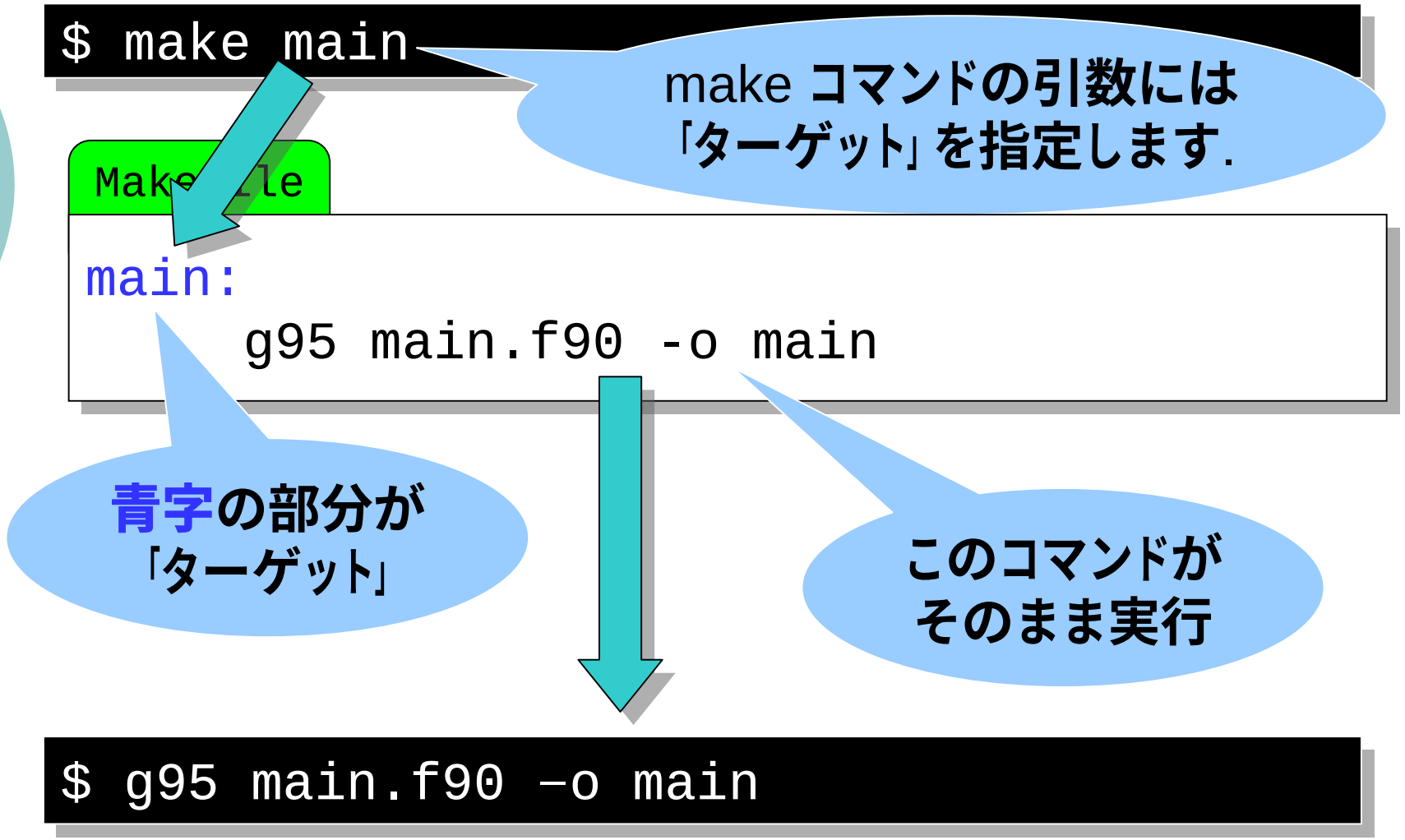
```
main:
	g95 main.f90 -o main
```

御託はいいから使ってみよう

- いざ，make 実行!!

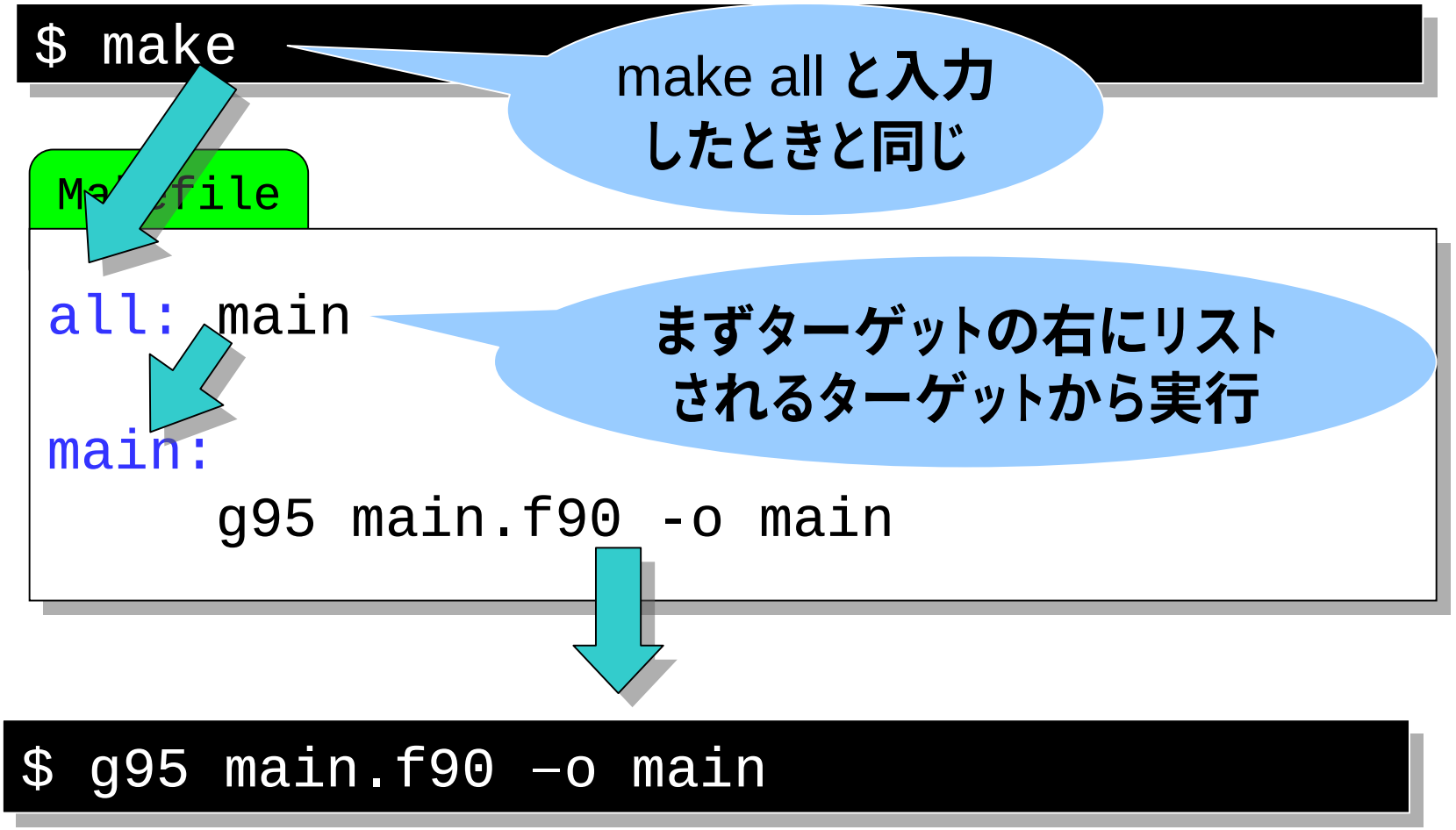
```
$ make main
g95 main.f90 -o main

$ ls –l
-rw-r--r--  ...  2006-06-29 22:45 Makefile
-rwxr-xr-x  ...  2006-06-29 22:51 main*
-rw-r--r--  ...  2006-06-29 22:32 main.f90
-rw-r--r--  ...  2006-06-29 22:51 main.o

$ ./main
 Hello, World!
```

仕組みはどうなってるの??



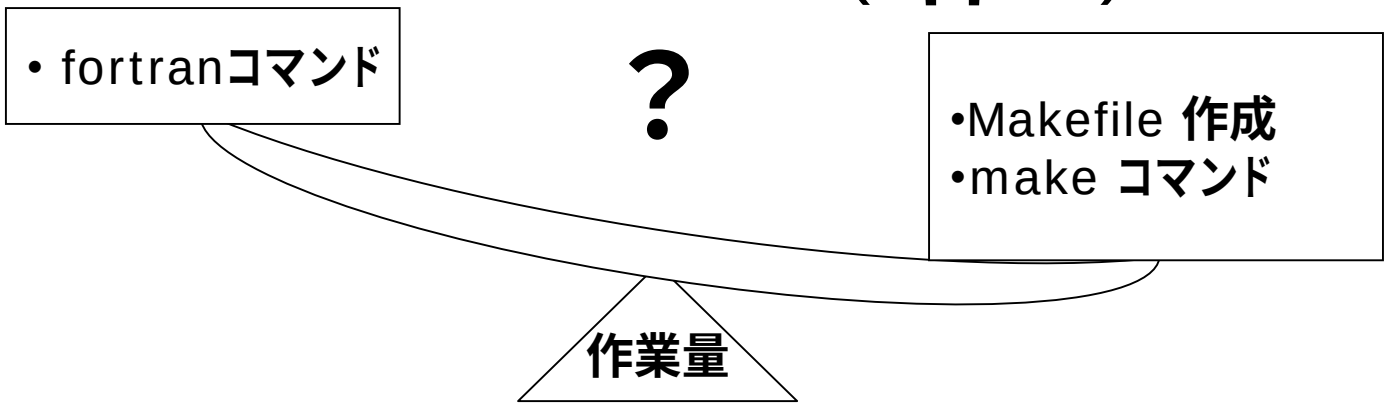
“all” というターゲット

- make に引数を付けない場合は...?
  - “all” というターゲットが「引数なし」に対応します.



ここまで聞いての素朴な疑問??

- Make って便利でつか?? (゜Д゜;)



- ここまでは「最低限 Make」

- ここからが Make のすごいとこ

make のすごいとこ

更新のチェック

- ソースコードが実行ファイルやオブジェクトファイルよりも新しい場合にコンパイル

- ターゲットに生成されるファイル名を指定し，その後ろに依存するファイルを追記

Makefile

```
all: main

main: main.f90
	g95 main.f90 -o main
```

ターゲット (生成されるファイルファイル) と，このソースコードファイルのタイムスタンプをチェック

更新のチェック

- 実際に試してみる

```
$ make
g95 main.f90 -o main


$ ls –l
-rwxr-xr-x ... 2006-06-29 22:51 main*
-rw-r--r-- ... 2006-06-29 22:32 main.f90


$ make
make: `all' に対して行うべき事はありません。
```

ソースコードの方が古い

Make は何もしない

更新のチェック

- ソースを更新

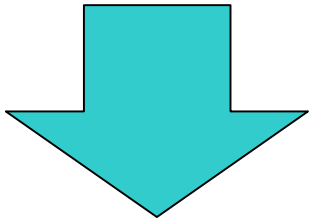
```
$ touch main.f90


$ ls –l
-rwxr-xr-x ... 2006-06-29 22:51 main*
-rw-r--r-- ... 2006-06-29 23:23 main.f90



$ make
g95 main.f90 -o main
```

ソースコードの方が新しい

ちゃんと実行ファイルを生成

更新のチェック

- 更新をチェックする利点は??

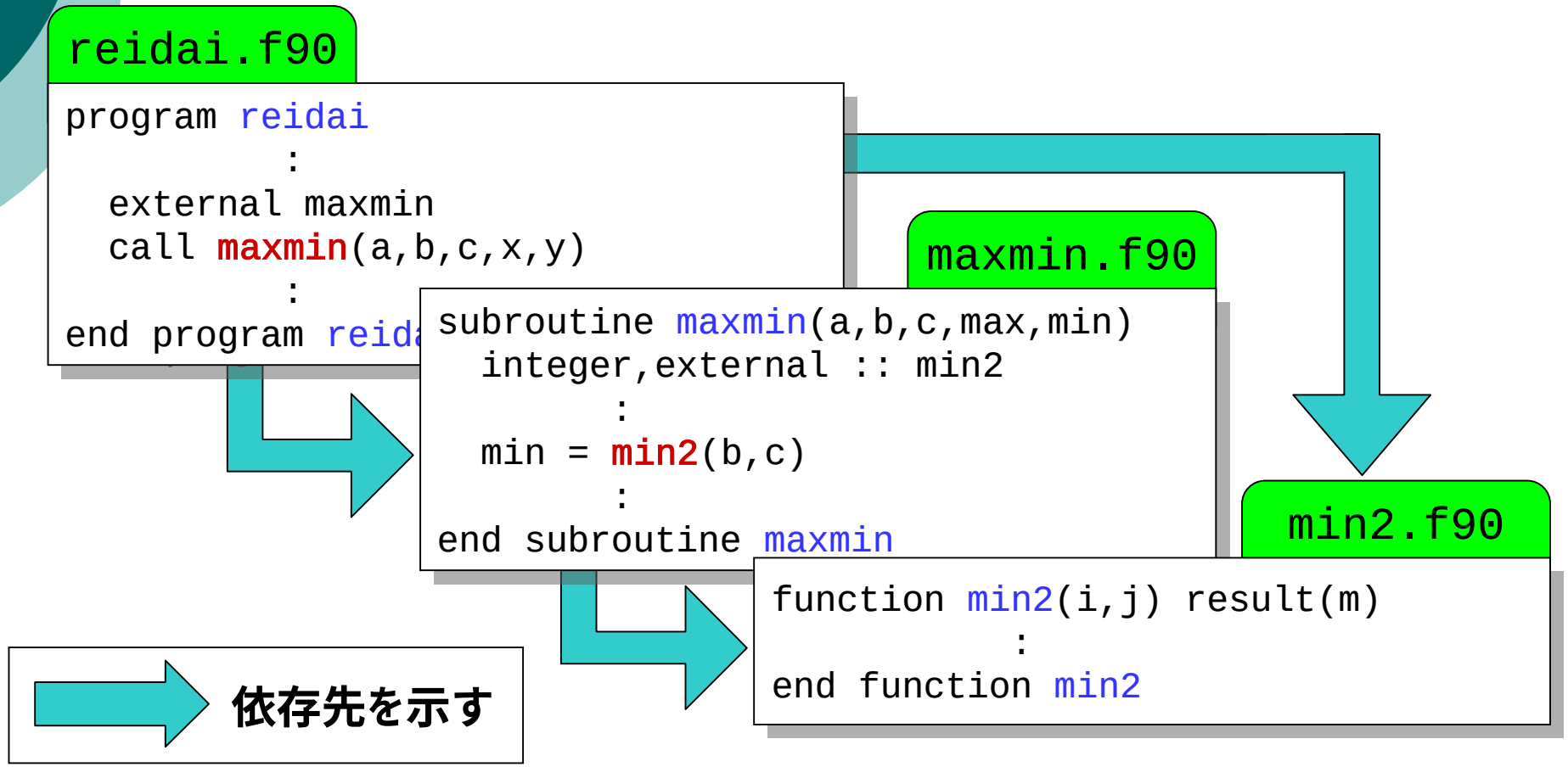


- 無駄なコンパイルを行なわなくて済む

- コンパイルに時間がかかる場合には，作業時間(コンパイルの待ち時間) が大幅に短縮される
  - 特に最適化に優れるコンパイラは一般に個々のファイルのコンパイルに時間がかかったりすることも.

依存関係のチェック

- ファイル同士の依存関係をチェック
- 以下のような場合を考えよう.



依存関係のチェック

- ターゲットに実行ファイル or オブジェクトファイルを指定
- そのターゲットの右に，依存するファイル (オブジェクトファイル or ソースコード) を指定

Makefile

```
all: reidai
reidai: min2.o maxmin.o reidai.f90
	g95 reidai.f90 maxmin.o min2.o -o reidai

maxmin.o: min2.o maxmin.f90
	g95 -c maxmin.f90

min2.o: min2.f90
	g95 -c min2.f90
```

依存関係のチェック

- 動作の仕組み

```
$ make
```

Makefile

```
all: reidai
reidai: min2.o maxmin.o reid
        g95 reidai.f90 maxmi
maxmin.o: min2.o maxmin.f
        g95 -c maxmin.f
min2.o: min2.f90
        g95 -c min2.f90
```

min2.o と min2.f90 のタイムスタンプをチェック

```
$ g95 -c min2.f90
```

依存関係のチェック

- 動作の仕組み

```
$ make
```

Makefile

```
all: reidai
reidai: min2.o maxmin.o rei
        g95 -o reidai.f90 max.in.o min2.o -o reidai
maxmin.o: min2.o maxmin.f90
        g95 -c maxmin.f90
min2.o: min2.f90
        g95 -c min2.f90
```

maxmin.o と maxmin.f90 の タイムスタンプを チェック

```
$ g95 -c min2.f90
$ g95 -c maxmin.f90
```

依存関係のチェック

- 動作の仕組み

```
$ make
```

Makefile

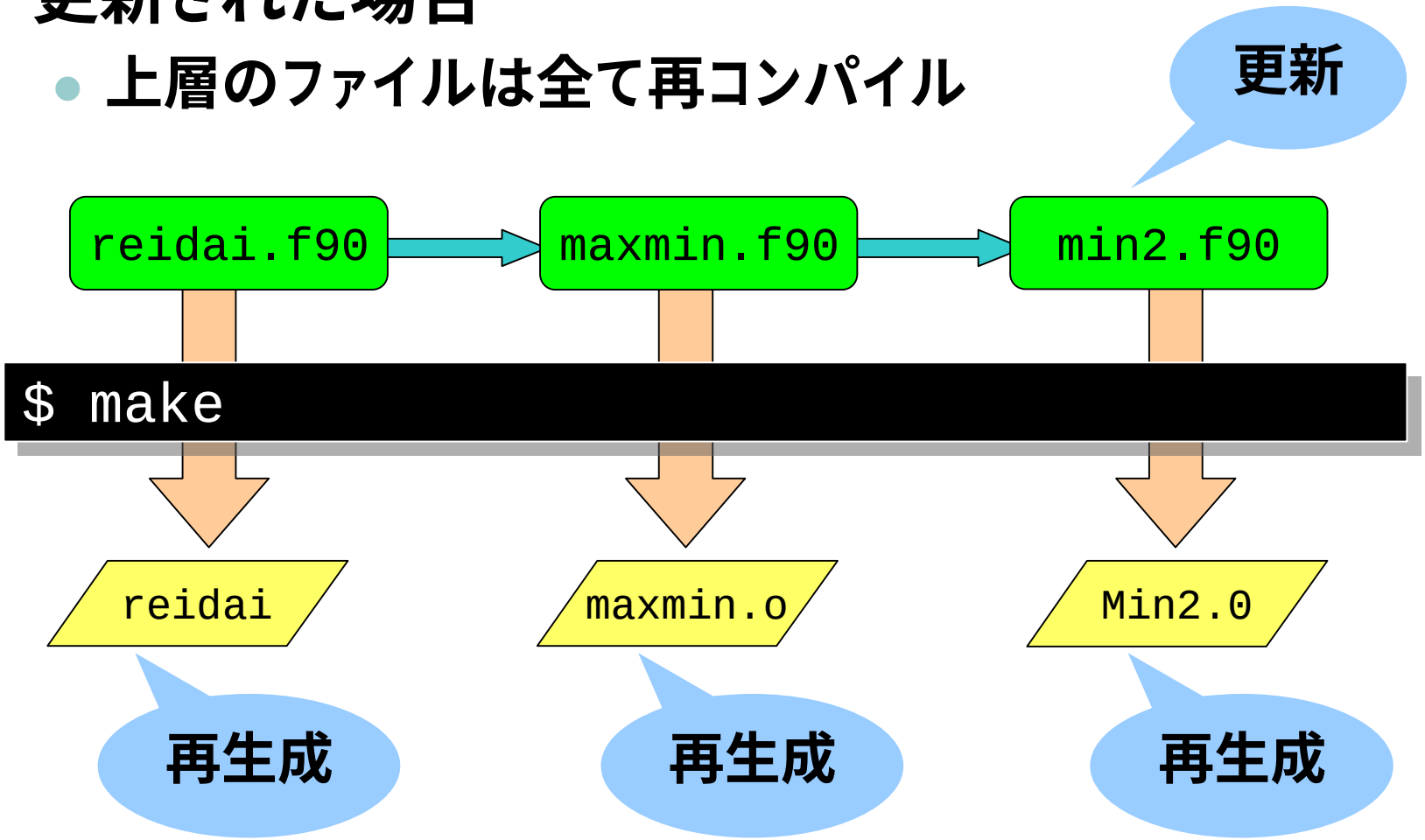
```
all: reidai
reidai: min2.o maxmin.o reidai.f90
        g95 reidai.f90 maxmin.o min2.o -o reidai
maxmin.o: min2.o maxmin.f90
        g95 -c maxmin.f90
min2.o: min2.f90
        g95 -c min2.f90
```

reidai と reidai.f90 のタイムスタンプをチェック

```
$ g95 -c min2.f90
$ g95 -c maxmin.f90
$ g95 reidai.f90 maxmin.o min2.o -o reidai
```

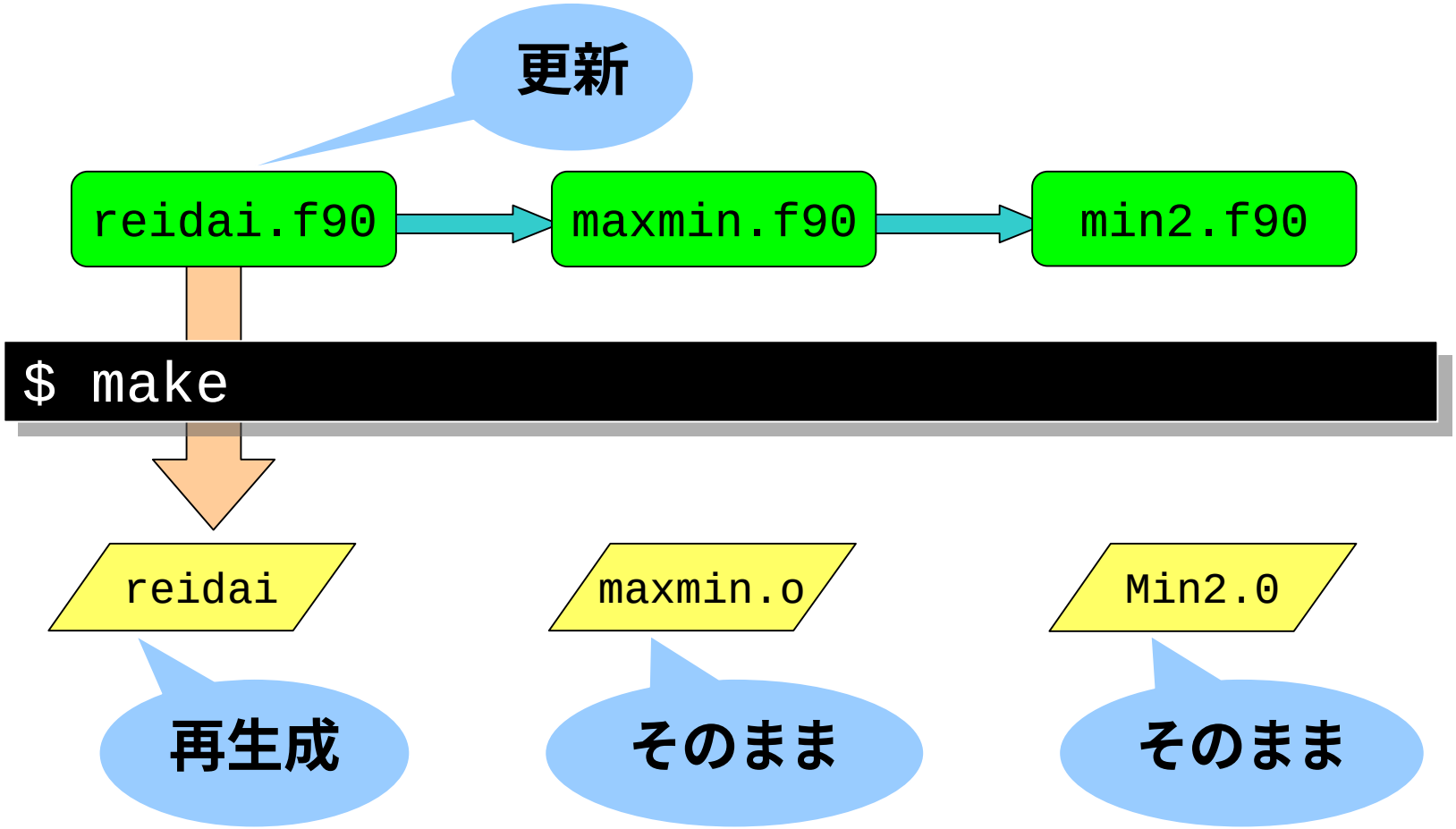
依存関係のチェック

- 依存関係の下層（依存される側）のファイルが更新された場合
  - 上層のファイルは全て再コンパイル



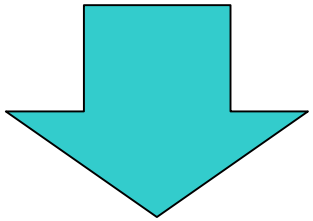
依存関係のチェック

- 依存関係の上層ファイルが更新された場合
  - 上層のファイルのみ再コンパイル



依存関係のチェック

- 依存関係をチェックする利点



- ファイルの更新に対して，必要最低限のコンパイルを行なってくれる．

- ファイルの数が多い場合，コンパイルの待ち時間が大幅に短縮される
  - プログラムが巨大になってくるとこの時間短縮はかなり有効

変数 (マクロ)

- 同じような表記が繰り返されると，改変が面倒...

Makefile

```
all: reidai
reidai: min2.o maxmin.o reidai.f90
	g95 reidai.f90 maxmin.o min2.o -o reidai
maxmin.o: min2.o maxmin.f90
	g95 -c maxmin.f90
min2.o: min2.f90
	g95 -c min2.f90
```

コンパイラを変更するには，3 箇所全部を書きかえる必要あり．
(ターゲットの数が増えると手に負えない...)

変数 (マクロ)

- 変数の利用によって管理が楽に

Makefile

```
FC = g95

all: reidai
reidai: min2.o maxmin.o reidai.f90
	$(FC) reidai.f90 maxmin.o min2.o -o reidai
maxmin.o: min2.o maxmin.f90
	$(FC) -c maxmin.f90
min2.o: min2.f90
	$(FC) -c min2.f90
```

自動変数

- 二度書くのは面倒じゃありません...?

Makefile

```
FC = g95

all: reidai

reidai: min2.o maxmin.o reidai.f90
	$(FC) reidai.f90 maxmin.o min2.o -o reidai

maxmin.o: maxmin.f90
	$(FC) -c maxmin.f90

min2.o: min2.f90
	$(FC) -c min2.f90
```

自動変数

- 主な「自動変数」
  - $@
    - ルールのターゲットのファイル名。
  - $<
    - 最初の依存関係 (ターゲットの右に書かれるファイル名) の名前。
  - $^, $+
    - 全ての依存関係の名前のそれぞれの間にスペースを挟んで並べたもの。 ($+ はちょっと機能が異なる. 詳しくは GNU Make リファレンスマニュアル参照)

自動変数

- 自動変数を使うと…？

Makefile

```
FC = g95

all: reidai
reidai: min2.o maxmin.o reidai.f90
	$(FC) $^ -o $@

maxmin.o: maxmin.f90
	$(FC) -c $<

min2.o: min2.f90
	$(FC) -c $<
```

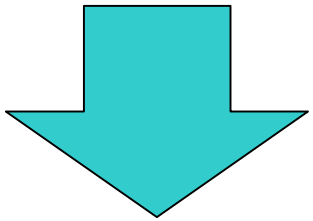
min2.o maxmin.o reidai.f90 が代入される

reidai が代入される

maxmin.f90 が代入される

自動変数

- 自動変数のメリット



- 同じファイル名を2度書きせずに済む
- Makefile をすっきり書くことが出来る
- 次に紹介する「サフィックスルール」との合わせ技ですばらしいことに

サフィックスルール

- まだ「似たようなもの」がありませんか?

Makefile

```
FC = g95

all: reidai
reidai: min2.o maxmin.o reidai.f90
	$(FC) $^ -o $@

maxmin.o: maxmin.f90
	$(FC) -c $<

min2.o: min2.f90
	$(FC) -c $<
```

***.f90 から ***.o を生成するコマンド，一緒ですね?

サフィックスルール

- 「サフィックスルール」を使ってみます
  - サフィックス(suffix) = 接尾辞　　拡張子名

Makefile

```
FC = g95

all: reidai
reidai: min2.o maxmin.o reidai.f90
	$(FC) $^ -o $@

maxmin.o: maxmin.f90

min2.o: min2.f90

%.o: %.f90
	$(FC) -c $<
```

(1) 一つ一つの依存関係にルールを書く必要が無くなる

***.f90 から ***.o を生成する際の一般的なルール

サフィックスルール

- 「サフィックスルール」を使ってみます
  - サフィックス(suffix) = 接尾辞　　拡張子名

Makefile

```
FC = g95

all: reidai
reidai: min2.o maxmin.o reidai.f90
	$(FC) $^ -o $@

maxmin.o:

min2.o:

%.o: %.f90
	$(FC) -c $<
```

(2) ***.f90 を書かなくても サフィックスルールから 自動的に補完

***.f90 から ***.o を生成する際の一般的なルール

サフィックスルール

- 「サフィックスルール」を使ってみます
  - サフィックス(suffix) = 接尾辞　　拡張子名

Makefile

```
FC = g95

all: reidai
reidai: min2.o maxmin.o reidai.f90
	$(FC) $^ -o $@

%.o: %.f90
	$(FC) -c $<
```

(4) ここに書いてあるだけで自動的に ***.f90 から ***.o を生成する.

(3) なので個々のターゲットを作る必要すらない

***.f90 から ***.o を生成する際の一般的なルール

サフィックスルール

- [例題] “reidai” (実行ファイル) 用のサフィックスルールも作ってみましょう

Makefile

```
FC = g95

all: reidai
reidai: min2.o maxmin.o

%.o: %.f90
	$(FC) -c $<
%: %.o
	$(FC) $^ -o $@
```

依存関係を書くだけ

実際にコンパイルを行なうコマンドはサフィックスルールにお任せ

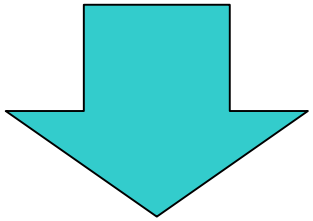
実践 Makefile

Makefile

```
FC       = g95     # Fortran コンパイラ
FFLAGS   = -O -I/usr/include
                   # コンパイル用のフラグ（オプションなど）
LDFLAGS = -L/usr/lib -lnetcdf-g95
                   # リンク用のフラグ（ライブラリなど）

all: reidai
reidai: min2.o maxmin.o


%.o: %.f90
       $(FC) -c $(FFLAGS) $<
%: %.o
       $(FC) $(FFLAGS) $^ $(LDLAGS) -o $@
```

Make のすごいとこ, まとめ

- 更新のチェック
- 依存関係のチェック
- 変数 (マクロ)
- サフィックスルール



- これだけ覚えとけばとりあえず OK !!
  - たぶんね… (￣▽￣;)

現役 make さん見学

Make の仕事を見てみよう

- Apache で働く make を見てみる
  - (まずは準備)

```
$ cd /usr/local/src

$ wget http://www.apache.jp/dist/
  httpd/apache_1.3.36.tar.gz

$ tar xvfz apache_1.3.3

$ cd apache_1.3.36

$ ./configure
```

ソースコード置き場へ移動

Apache の tar ボールをダウンロード

Make の仕事を見てみよう

- Apache で働く make を見てみる
  - (まずは準備)

```
$ cd /usr/local/src

$ wget http://www.apache.jp/dist/
  httpd/apache_1.3.36.tar.gz

$ tar xvfz apache_1.3.36.tar.gz

$ cd apache_1.3.36

$ ./configure
```

Tar ボール を展開

コンパイルを行なう環境が どのようなものであるかを調査

Make の仕事を見てみよう

- make によるコンパイル

```
$ make

===> src/main
make[1]: Entering directory `/usr/local/src/
apache_1.3.34/src/main'

gcc -c  -I../os/unix -I../include
   -DLINUX=22 ... http_config.c
                      :
ar cr libmain.a .. http_config.o http_core.o ..

make[1]: Leaving directory `/usr/local/src/
apache_1.3.34'
<=== src
```

Make 実行

src/main ディレクトリへ移動

Make の仕事を見てみよう

- make によるコンパイル

```
$ make

===> src/main
make[1]: Entering direc
apache_1.3.34/src/main'

gcc -c  -I../os/unix
    -DLINUX=22 ... http_
                       :
ar cr libmain.a .. http_config.o http_core.o ..

make[1]: Leaving directory `
apache_1.3.34'
<=== src
```

C コンパイラ gcc によって
http_config.c をコンパイル

アーカイブコマンド ar によって
ライブラリ libmain.a を作成

仕事を終えてもとの
ディレクトリへ戻る

Make の仕事を見てみよう

- make によるインストール

```
$ make install
make[1]: Entering directory ...

===> [mktree: Creating Apache installation tree]
./src/helpers/mkdir.sh /usr/local/apache/bin
mkdir /usr/local/apache/bin
                    :
===> [programs: Installing Apache ht
./src/helpers/install.sh -c -m 755 -
  /usr/local/apache/bin/httpd
                    :
===> [config: Installing Apache config files]
./src/helpers/install.sh -c -m 644 conf/httpd.conf
  /usr/local/apache/conf/httpd.conf
                    :
make[1]: Leaving directory ...
```

Make 実行

ディレクトリツリーの作成

Make の仕事を見てみよう

- make によるインストール

```
$ make install
make[1]: Entering directory ...

===> [mktree: Creating Apa
./src/helpers/mkdir.sh /usr
mkdir /usr/local/apache/bin
                    :
===> [programs: Instal
./src/helpers/install.sh
  /usr/local/apache/bin/htt
                    :
===> [config: Installing Apache conf
./src/helpers/install.sh -c -m 6
  /usr/local/apache/conf/httpd
                    :
make[1]: Leaving directory ...
```

Apache の実行プログラム httpd のインストール

設定ファイル httpd.conf のインストール

仕事を終えてもとのディレクトリへ戻る

Make の仕事を見てみよう

- おさらい (Apache の場合)
  - make によって...
    - ソースコード ****.c をコンパイル
    - ライブラリ lib*****.a の生成
    - 実行コマンド (例: httpd) の生成
  - make install によって...
    - インストール先のディレクトリツリーの作成
    - ライブラリや実行コマンドをインストール
      - 実際には単にコピーしてパーミッションの設定をするだけ
- 大抵の make は上記のような働きをします

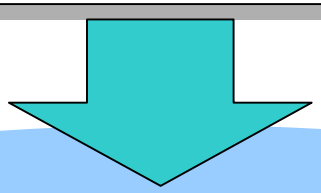
make 応用編

いろいろ応用 (1)

- TeX から DVI, PDF へ

Makefile

```
PTEX    = platex    # LaTeX コマンド
DVIPDF  = dvipdfmx  # DVI -> PDF コンバータ
all: ronbun.pdf
%.dvi: %.tex
	$(PTEX) $<
	$(PTEX) $<
%.pdf: %.dvi
	$(DVIPDF) $<
```



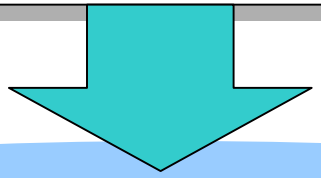
ronbun.tex から ronbun.dvi を経て
ronbun.pdf を生成

いろいろ応用 (2)

- RD から HTML

Makefile

```
RD       = rd2          # rdtool
RDFLAGS = -r rd/rd2html-lib --with-css=$(CSS)
                        # オプション
CSS      = zagaku.css # スタイルシート
all: zagaku.htm
%.htm: %.rd
        $(RD) $(RDFLAGS) $< > $@
```



zagaku.rd から zagaku.htm を生成

http://www.gfd-dennou.org/library/dcmodel/doc/sample_Makefile/Makefile.rd2html
にもう少し凝った Makefile 置いてあります.

make 落ち穂拾い

よくある「ターゲット」

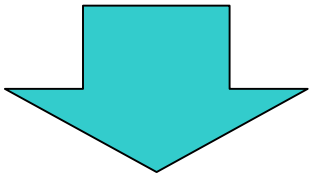
- all
  - プログラムを完全にコンパイルする.
- install
  - コンパイルされたライブラリなどのシステムにインストールする
- clean
  - コンパイルされたオブジェクトファイルや，実行ファイルをカレントディレクトリから削除
- check
  - 作成されたライブラリの動作チェックなど

コマンドエコー

- makeは実行するコマンドを画面に表示する．この機能の事をエコー(echoing＝反響)と呼ぶ．
  - `@‘ではじまる行はエコーしない．

Makefile

```
all:
	echo hogehoge
```



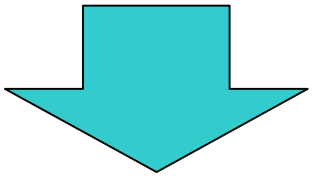
```
echo hogehoge
hogehoge
```

コマンドエコー

- makeは実行するコマンドを画面に表示する．この機能の事をエコー(echoing＝反響)と呼ぶ．
  - `@‘ではじまる行はエコーしない．

Makefile

```
all:
	@echo hogehoge
```



```
hogehoge
```

コマンド内エラー

- makeは実行したコマンドからエラーが返るとその後の処理を残したまま終了する
  - `ー‘ではじまる行はエラーが返っても続く

Makefile

```
all:
	rm hogehoge.txt
	rm foo.txt
```

```
rm hogehoge.txt
rm: cannot remove `hogehoge.txt‘: そのようなファイルやディレクトリはありません
make: *** [all] エラー 1
```

コマンド内エラー

- makeは実行したコマンドからエラーが返るとその後の処理を残したまま終了する
  - `-'ではじまる行はエラーが返っても続く

Makefile

```
all:
	-rm hogehoge.txt
	-rm foo.txt
```

```
rm hogehoge.txt
rm: cannot remove `hogehoge.txt': そのようなファイルや...
make: [all] エラー 1 (無視されました)
rm foo.txt
rm: cannot remove `foo.txt': そのようなファイルやディレクト...
make: [all] エラー 1 (無視されました)
```

関数だってありますよ

- $(*関数 引数*) みたいに使う.
- $(subst *from*,*to*,*text*)
  - *text*という文章の本文の置換動作をしてくれます。つまり文章中にある全部の*from*を*to*に置換します。
- $(dir *names*...)
  - *names*の中のそれぞれのファイル名のディレクトリ部分を抽出します。
- とかとか... (詳しくは GNU Make マニュアル「テキスト変形関数」参照のこと)

シェルスクリプト

- **基本的に** sh (Bourne Shell) **が書けます**.

Makefile

```
FC      = g95    # Fortran
FFLAGS  = -O -I/usr/includ
LDFLAGS = -L/usr/lib -lnet

all: reidai
reidai: min2.o maxmin.o
        if [ -f $@ ] ; then ¥
            chmod g+w $@ ;¥
        fi

%.o: %.f90
        $(FC) -c $(FFLAGS) $<
%: %.o
        $(FC) $(FFLAGS) $^ $(LDLAGS) -o $@
```

IF 文を書くとき FI までの部分の行末に “¥” を付けるよう注意

おわりに

- Make って...
  - ソフトウェア開発，特にコンパイラ言語を用いている場合，コンパイル作業を助けてくれる大変便利なツール
  - 少しだけ覚えることあるけど，その絶大な効果に比べれば大したことない (と思う)
  - 他にも ファイルA ⇒ ファイルB を生成する場合には応用が利く

- 結論
  - 始めは簡単なもので良いので，是非 Make に触ってみよう!!

参考資料

- GNU Make マニュアル (日本語訳)
  - http://www.ecoop.net/coop/translated/GNUMake3.77/
- Andrew Oram, Steve Talbott 著, 矢吹 道朗 監訳, 菊池 彰 訳, 1997: make 改訂版. オライリージャパン, 150pp.
- “mk” に関して
  - http://plan9.aichi-u.ac.jp/mk/
- Windows 上で動作する GNU make を同梱するソフトウェア MinGW
  - http://www.mingw.org/