[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4. ルールの記述

makefileに現われる*ルール*はいつどのようにファイルを再構成するかを示し、ルールの*ターゲット*といわれます。多くの場合、1つのルールに1つの ターゲットです。そして、ターゲットの*必要条件*となるほかのファイルを リストし、ターゲットを生成したりアップデートするための命令をリストします。

ルールの順序は*default goal*の決定を除けば重要ではありません。異なる形で指定しない限りmakeが判断するターゲットが デフォルトゴールです。また、デフォルトゴールは最初のmakefileの最初の ルールで、仮に最初のルールが複数のターゲットを持つ場合にはその1番目が デフォルトゴールとなります。ただし、ここで2つの例外があります。ピリオドで始まるターゲットで、1つあるいは複数のスラッシュ'/'を 含まない限り、それはデフォルトのターゲットではありません。同様に、パターンルールを定義するターゲットはデフォルトゴールに対して 何の影響も与えません（see section パターンルールの定義と再定義.）。

したがって、通常は、最初のルールがすべてのプログラムをコンパイルするように、あるいは'all'と呼ばれるターゲットをしばしば用いるようにmakefileを 記述します。See section ゴールを指定する引数.。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.1 ルールのシンタックス

一般的にルールは以下のようになります。

```
targets : prerequisites
        command
        …
```

もしくは、以下のようになります。

```
targets : prerequisites ; command
        command
        …
```

*targets*はスペースで区切られたファイル名で、ワイルドカードが使われる こともあります（see section ファイル名におけるワイルドカードの使用.）。‘*a*(*m*)’というファイル名では、アーカイブファイル*a*の メンバーである*m*を表わしていることがあります（see section ターゲットとしてのアーカイブメンバー.）。通常は、ルールごとに1つのターゲットがありますが、まれに、複数の場合もあります（see section ルール中の複数のターゲット.）。

*command*行の先頭はタブ文字で始まります。最初の命令は必要条件のあとに タブ文字とともに現われるか、セミコロンで同じ行に現われます。どちらの方法でも 効果は同じです。See section ルールのなかでのコマンドの書き方.。

変数の参照を開始するためにドル記号が使用されるため、ルールのなかでドル記号を 用いたい場合には、‘$$’（see section 変数の使用法.）のように2つ続けて書かなければなりません。また、長い行はバックスラッシュで 分割することができますが、makeはmakefileのなかでの行の長さに 制限を持たないため、必ずしもそうする必要はありません。

ルールはmakeに対して、ターゲットがいつ期限切れであるか、必要な場合にどのようにアップデートするかの2つを指示します。

期限切れであるということの尺度は、（スペースによってファイル名が 区切られている）必要条件で指定されています。（ワイルドカードとその アーカイブメンバー（see section アーカイブファイルのアップデートにmakeを使用する.）も同様に置くことができます）もし、それが存在しないか あるいは必要条件のいずれよりも古ければ（最終更新日付との比較によって）、そのターゲットは期限切れとされます。このアイデアは、ターゲットファイルの 内容が必要条件の情報に基づいて計算されるということです。したがって、いずれかの必要条件に変更があった場合には、既存のターゲットファイルの内容は 有効ではなくなるのです。

どのようにアップデートするかは命令によって指定され、それらはシェル（通常は ‘sh’）といくつかの特別な機能を用いて実行されます（see section ルールのなかでのコマンドの書き方.）。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.2 ファイル名におけるワイルドカードの使用

*ワイルドカード*を用いると1つのファイル名で多くのファイルを指定することが できます。makeにおけるワイルドカード文字は、‘*’、‘?’と ‘[…]’でBourneシェルと同じです。たとえば、‘*.c’は ワーキングディレクトリの‘.c’という名前のファイルすべてを指定します。

ファイル名の先頭が‘~’で始まるファイルもまた特別な意味を持ちます。単独もしくは次にスラッシュがある場合は、ホームディレクトリを表わし、たとえば‘~/bin’は‘/home/you/bin’と展開されます。また、‘~’の 次に単語がある場合、その単語が示すユーザーのホームディレクトリを表わします。たとえば、‘~john/bin’は‘/home/john/bin’と展開されます。MS-DOSや MS-Windowsのようにそれぞれのユーザーのホームディレクトリを持たないシステムの 場合には、この機能は環境変数*HOME*の設定によってシミュレートされます。

ワイルドカードの展開は、ターゲット、必要条件、シェルが実行するコマンドに よって自動的に行なわれます。ほかの文脈においても、この機能を明示的に 指示することでワイルドカード展開は行なわれます。

ワイルドカード文字の特別な意味は、バックスラッシュを前置することで 無効にできます。したがって、‘foo¥*bar’は‘foo’、アスタリスクと ‘bar’からなる具体的なファイルを参照することになります。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.2.1 ワイルドカードの例

ワイルドカードはシェルによって展開されるルール中の命令で用いられます。たとえば、すべてのオブジェクトファイルを削除するルールの例は 以下のようになります。

```
clean:
        rm -f *.o
```

また、ワイルドカードはルールの必要条件においても有益です。makefileの なかで以下のようなルール、すなわち‘make print’は、前回印刷した以降に 更新されたすべての‘.c’ファイルを印刷します。

```
print: *.c
        lpr -p $?
        touch print
```

このルールでは、‘print’を空のターゲットファイルとして扱います。レコードイベントに対する空のターゲットファイル.を 参照してください。(また、自動変数‘$?’は更新されたファイルだけを 対象としています。自動変数.参照。)

しかし、変数を定義した場合にはワイルドカードによる展開は起こりません。したがって、

```
objects = *.o
```

のように書いた場合には、変数objectsの値は、実際の‘*.o’となります。しかしながら、ターゲット、必要条件や命令でobjectsの値を用いる場合には、ワイルドカードによる展開が行なわれます。objectsが展開されるように 設定するには、かわりに以下のようにしてください。

```
objects := $(wildcard *.o)
```

See section ワイルドカードの機能.。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.2.2 ワイルドカードの落とし穴

ここにワイルドカードを使用する際に意図しない動作をする微妙な例があります。たとえば、あるディレクトリのすべてのオブジェクトファイルから実行可能なファイル ‘foo’が作られる場合に、次のように記述することでしょう。

```
objects = *.o

foo : $(objects)
        cc -o foo $(CFLAGS) $(objects)
```

objectsの値は実際の‘*.o’の文字列です。そして、ワイルドカードに よる展開は、‘foo’に対するルールのなかで行なわれます。そして、それぞれの *実在する*‘*.o’ファイルが‘foo’の必要条件となり、必要に応じて 再コンパイルされます。

しかし、‘*.o’ファイルをすべて削除したらどうなるでしょうか。ワイルドカードがマッチするものがなくなり、‘*.o’という妙な名前のついた ファイルに‘foo’が依存することになります。したがって、そのようなファイルが 存在しそうもないため、makeは‘*.o’を作ることができないという エラーを表示します。これは望んだ結果ではないはずです。

実際には、ワイルドカードによる展開によって、望む結果が得られるかも しれませんが、ワイルドカードによる機能と文字列の代用による洗練された テクニックを必要とするでしょう。

Microsoftのオペレーティングシステム（MS-DOSとMS-Windows）では、パス名の ディレクトリの区切りはバックスラッシュを用い以下のように書かれます。

```
c:¥foo¥bar¥baz.c
```

これは、Unixスタイルの‘c:/foo/bar/baz.c’（ここで‘c:’は ドライブレターと呼ばれます）に等しく、makeがこのようなシステム上で 動作するときには、パス名においてはバックスラッシュがUnixでのスラッシュに 相当します。しかし、バックスラッシュがクォート文字となる場合には、ワイルドカードによる展開はサポート*されません*。したがって、そのような場合にはUnixスタイルのスラッシュを使用*しなければ*なりません。

[＜] [＞] [＜＜] [上] [＞＞] [冒頭] [目次] [見出し] [？]

4.2.3 ワイルドカードの機能

ワイルドカードによる展開はルール中で自動的に行なわれますが、変数が 設定されている場合、あるいはファンクションの引数のなかでは通常は 行なわれません。そのような場合でもワイルドカードによる展開が必要な場合は、以下のようなワイルドカードの機能を用います。

```
$(wildcard pattern…)
```

makefileのどこの位置で用いられてもこの文字列は、与えられたファイル名の パターンの1つに一致する実在のファイルの名前をスペースで区切ったリストに 置き換えられます。実在のファイルがパターンに一致しない場合には、ワイルドカードの機能による出力からは省略されます。注意しなければならないのは、無視されるというよりもむしろまったく同じとして扱われる場合に、ルールのなかで 一致しないワイルドカードの挙動が異なることです（see section ワイルドカードの落とし穴.）。

ワイルドカードの機能の1つは、以下のようにあるディレクトリのCの ソースファイルのリストを得ることです。

```
$(wildcard *.c)
```

そして、拡張子‘.c’を‘.o’に置き換えることによってCの ソースファイルのリストをオブジェクトのリストに変更することができます。

```
$(patsubst %.c,%.o,$(wildcard *.c))
```

（ここでは、patsubstという別の機能を用いています。See section 文字列の代用と分析のファンクション.。）

そして、makefileはすべてのCのソースファイルをそのディレクトリでコンパイルし、以下のように書かれているように互いにリンクします。

```
objects := $(patsubst %.c,%.o,$(wildcard *.c))

foo : $(objects)
        cc -o foo $(objects)
```

（これは、Cのプログラムをコンパイルするための暗黙のルールを利用し、そこでは コンパイルに必要な明示的なルールを記述する必要はありません。See section 変数の2つのフレーバー.）

[＜] [＞] [＜＜] [上] [＞＞] [冒頭] [目次] [見出し] [？]

4.3 必要条件のためのディレクトリサーチ

大規模なシステムの場合には、バイナリとは異なるディレクトリにソースを 置くことが望ましいことがあります。makeの*ディレクトリサーチ*機能は 必要条件をみつけるためにいくつかのディレクトリをサーチします。ファイルをいくつかのディレクトリに再配置する場合でも個々のルールを 変更する必要はなく、サーチパスの変更だけですみます。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.3.1 VPATH:すべての必要条件のサーチパス

makeにおける可変VPATHの値は、makeがサーチすべき ディレクトリのリストを指定します。たいていの場合は、カレントディレクトリに ない必要条件のファイルを含むことが期待されますが、VPATHは、ルールのターゲットを含み、しかもmakeがすべてのファイルに適用する サーチリストを指定します。

したがって、仮にターゲットや必要条件としてリストされたファイルが カレントディレクトリに存在しない場合、makeはVPATHで リストされたディレクトリをサーチすることになります。そのどこかの ディレクトリでファイルがみつかった場合、そのファイルが必要条件と なります（下記参照）。そして、カレントディレクトリにすべてのファイルが あるかのごとく、ルールは必要条件のリストのなかでファイル名を 特定できるようになります。See section ディレクトリサーチにおけるシェルコマンドの記述.。

VPATH変数においては、ディレクトリ名はコロンあるいは空白で 区切られます。ディレクトリのリストの順序は、makeのサーチの 順序に従います。（MS-DOSとMS-Windowsにおいては、パス名自体にコロンが 使用されるため、ドライブ名のあとにVPATHのディレクトリ名のセパレータ としてセミコロンが使われます。）

たとえば、

```
VPATH = src:../headers
```

は、‘src’と‘../headers’の2つのディレクトリを指定し、makeはその順序でサーチをします。

VPATHの値によって、以下のルールは、

```
foo.o : foo.c
```

以下のように書かれたかのごとくに解釈されます。

```
foo.o : src/foo.c
```

ファイル‘foo.c’を仮定し、カレントディレクトリに存在しなくとも ‘src’ディレクトリでそれを発見します。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.3.2 vpathディレクティブ

VPATH変数に似ていますが、より選択的なディレクティブが vpath（小文字であることに注意してください）で、あるパターンにマッチする ファイル名の特定のクラスのためのサーチパスを指定できます。したがって、一定の サーチディレクトリをファイル名の1つのクラス、およびほかのファイル名に対する ほかのディレクトリに与えることが可能です。

vpathディレクティブには3つの形式があります。

`vpath` *`pattern directories`*

*pattern*にマッチするファイル名に対してサーチパスの *directories*を指定する。

サーチパスの*directories*は、サーチされるディレクトリのリストで、コロン（MS-DOSとMS-Windowsではセミコロン）で区切られ、VPATH変数に おけるサーチパスと同様です。

`vpath` *`pattern`*

*pattern*で関連付けられたサーチパスをクリアします。

`vpath`

vpathディレクティブで以前に指定されたサーチパスをクリアします。

vpathパターンは‘%’文字を含む文字列です。その文字列は、サーチされる必要条件のファイル名にマッチしなければならず、‘%’文字は ゼロ以上長さの文字列とマッチします（see section パターンルール の定義と再定義.）。たとえば、%.hは.hで終わる ファイルとマッチします。（‘%’がない場合には、必要条件に正確に マッチしなければならず、ほとんどの場合には有益ではありません。）

vpathディレクティブパターンにおける‘%’文字は、そのまえに書かれたバックスラッシュによって引用されます。‘%’文字を 引用するバックスラッシュはさらに多くのバックスラッシュで 引用されることになりますが、それがファイル名と比較されるまえに、‘%’文字またはほかのバックスラッシュで引用されるバックスラッシュはパターンから削除されます。‘%’を引用する危険のない バックスラッシュは悩みの種にはなりません。

仮に、必要条件がカレントディレクトリに存在せず、vpath ディレクティブにおける*pattern*が必要条件のファイル名にマッチする場合、そのディレクティブの*directories*はvpath変数のディレクトリと 同様にサーチされます。

たとえば、

```
vpath %.h ../headers
```

という記述はmakeに対して、カレントディレクトリに存在しない場合は、‘.h’で終わるファイル名の必要条件を‘../headers’ディレクトリで 探すように指示します。

そして、いくつかのvpathパターンが必要条件のファイル名にマッチする ならば、makeはvpathディレクティブにおけるすべてのディレクトリを 1つずつそれぞれマッチさせます。さらにmakeはmakefileに現われる順序で 複数のvpathディレクティブを扱います。ここで、同じパターンを持つ複数の ディレクティブは互いに独立したものです。

したがって、

```
vpath %.c foo
vpath %   blish
vpath %.c bar
```

は、‘.c’で終わるファイルを‘foo’でサーチし、それから ‘blish’、‘bar’でサーチします。

```
vpath %.c foo:bar
vpath %   blish
```

の場合は、‘.c’で終わるファイルを‘foo’でサーチし、それから ‘bar’、‘blish’でサーチします。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.3.3 ディレクトリサーチはどのように行なわれるのか

一般的あるいは選択的にかかわらず、ディレクトリサーチをして必要条件が みつかるとき、そのパス名は、makeが実際に提供する必要条件のリストの うちの1つではないかもしれません。したがって、ときおりディレクトリサーチを 通じて発見されたパス名は破棄されることがあります。

makeがディレクトリサーチで発見したパスを保持するか破棄するかを 決定するためのアルゴリズムは以下のようなものです。

1. makefile中で指定されたパスでターゲットファイルが存在しない場合には、ディレクトリサーチが行なわれます。
2. ディレクトリサーチに成功した場合は、そのパスが保持され、そのファイルは 一時的にストアされます。
3. ターゲットのすべての必要条件が同じ方法を用いて試されます。
4. 必要条件の処理のあと、必要に応じてターゲットが再構成されることもあります。
    1. もし、ターゲットが再構成される必要のない場合は、ディレクトリサーチの あいだに発見されたファイルへのパスは、そのターゲットが含まれるすべての 必要条件のリストに使用されます。簡単にいえば、makeがターゲットを 再構成する必要のない場合は、ディレクトリサーチによるパスを使用するのです。
    2. 一方、期限切れによって、ターゲットの再構成が必要な場合は、ディレクトリサーチのあいだに発見されたファイルへのパスは*破棄*され、makefile中で指定されたファイル名を使用してそのターゲットは再構成されます。いいかえれば、makeがターゲットを再構成する際は、ディレクトリサーチによるディレクトリではなくローカルに再構成します。

このアルゴリズムは複雑そうに見えますが、実際には意図したとおり、正しく 動作します。

makeのほかのバージョンはより簡単なアルゴリズムを使用します。ファイルが存在せず、ディレクトリサーチで発見された場合はターゲットの 再構成の必要性に関係なくそのパス名が使用されます。したがって、ディレクトリサーチで発見されたパス名でターゲットが再構成されます。

実際、そのような動作をさせたい場合には、make中でGPATH変数を 使用することができます。

GPATHはVPATHと同じくスペースあるいはコロンでパス名を 区切る書式とシンタックスを持ちます。ディレクトリサーチで、あるディレクトリに期限切れのターゲットが発見された場合、しかもそれがGPATHにある場合、そのパス名は破棄されず、ターゲットは展開されたパスを用いて再構成されます。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.3.4 ディレクトリサーチにおけるシェルコマンドの記述

ディレクトリサーチのあいだに別のディレクトリで必要条件が発見された場合でも、ルールの命令を変更することはできず、記述されたとおりに実行されます。したがって、makeがその必要条件を発見したディレクトリのなかを探すように 注意深く命令を記述しなければなりません。

この場合は、'$^'のような*自動変数*を用います(see section 自動変数.)。たとえば、'$^'の値は ルールのすべての必要条件のリストで、それらが発見されたディレクトリの名前を 含み、'$@'の値はターゲットとなります。

```
foo.o : foo.c
        cc -c $(CFLAGS) $^ -o $@
```

(変数CFLAGSは暗黙のルールにおけるC言語の互換性のために指定が 可能なようになっていて、すべてのC言語のコンパイルに同じように影響を与え、整合性を取るために使用されます。see section 暗黙のルールで使用される変数..)

必要条件は同じようにヘッダファイルを含むことが多く、そのヘッダファイルに ついては言及したくない場合があります。自動変数の'$<'はまさに最初の 必要条件です。

```
VPATH = src:../headers
foo.o : foo.c defs.h hack.h
        cc -c $(CFLAGS) $< -o $@
```

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.3.5 ディレクトリサーチと暗黙のルール

複数のディレクトリにわたる、VPATHの中あるいはvpathで指定される サーチは暗黙のルールの解釈のあいだに行なわれます（see section 暗黙のルールの使用.）。

たとえば、ファイル‘foo.o’が明示的なルールをまったく持たない場合、makeが暗黙のルールを考慮し、それはファイルが存在する場合には ‘foo.c’をコンパイルするというあらかじめ用意されているルールです。カレントディレクトリにそのファイルがない場合には適切なディレクトリが サーチされます。どこかのディレクトリに‘foo.c’が存在する、あるいはmakefileで指示されている場合には、C言語のコンパイルに 関するルールが暗黙のルールとなります。

暗黙のルールの命令は、通常必ず自動変数を使用します。その結果、特別なことを せずにディレクトリサーチによって発見されたファイル名を使用します。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.3.6 リンクライブラリに対するディレクトリサーチ

リンカによって用いられるライブラリに対してはディレクトリサーチは特別な 方法を用います。この仕組みは、必要条件の名前が‘-l*name*’の 書式のときに働きます。（通常は必要条件がファイル名であるため、ライブラリの 名前の*file name*が、一般的に‘-l*name*’ではなく ‘lib*name*.a’のように見えるため、何か奇妙な感じを受けるかもしれません。）

必要条件の名前が‘-l*name*’の書式である場合、makeは その名前を‘lib*name*.so’としてカレントディレクトリやvpathのサーチパスとVPATHのサーチパスにマッチした ディレクトリでサーチし、その後、‘/lib’、‘/usr/lib’、と‘*prefix*/lib’のディレクトリでサーチします（ここで、通常は ‘/usr/local/lib’ですが、MS-DOS/MS-Windowsのバージョンの makeでは*prefix*がDJGRPインストレーションツリーの ルートディレクトリであるかのようにふるまいます）。

もし、ファイルがみつからない場合にはファイル‘lib*name*.a’が 上記と同様のディレクトリでサーチされます。

たとえば、‘/usr/lib/libcurses.a’ライブラリがシステムにある場合（かつ‘/usr/lib/libcurses.so’はない場合）、

```
foo : foo.c -lcurses
        cc $^ -o $@
```

は、‘foo’が‘foo.c’あるいは‘/usr/lib/libcurses.a’よりも 古い場合、‘cc foo.c /usr/lib/libcurses.a -o foo’というコマンドを 実行します。

デフォルトの設定でサーチされるのは‘lib*name*.so’と ‘lib*name*.a’ですが、.LIBPATTERNS変数によってカスタマイズ することができます。この変数の値における個々の語はパターン文字列です。‘-l*name*’のような必要条件が現われる場合には、make は*name*でリスト中の個々のパターンの%を置き換え、ライブラリの ファイル名を用いて上記のディレクトリのサーチを行います。ライブラリがみつからない場合には、リストの次の語が使用されます。

.LIBPATTERNS変数のデフォルトの値は、‘lib%.so lib%.a’で、前記の ふるまいを行なうようになっています。

しかし、この変数の値を空にすることで、リンクライブラリへの 展開をしないように設定することもできます。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.4 偽のターゲット

偽のターゲットは本当はファイルの名前ではありません。明示的な要求を作る際に実行されるいくつかの命令の名前（通称）です。偽のターゲットを使うための、2つの理由があります。1つは同一のファイル名による コンフリクトを避けるためで、もう1つは、パフォーマンスの向上のためです。

ターゲットファイルを作成しない命令によるルールを記述する場合、その命令は ターゲットが現われるたびに実行されます。例をあげます。

```
clean:
        rm *.o temp
```

rmコマンドは、‘clean’というファイルを作成はしないため、そのようなファイルは存在しません。したがって、rmコマンドは ‘make clean’を実行するたびに実行されることになります。

もし何かが‘clean’という名前のファイルをこのディレクトリに 作成するならば、偽のターゲットは動作を停止するでしょう。なぜなら ‘clean’は必要条件を持たないからで、必然的に最新であると考えられ、命令の実行は行なわれません。この問題を避けるには、特別なターゲット .PHONY（see section 特別な組み込み済みのターゲット名.）を使用して、明示的にターゲットが偽であるように宣言しておきます。

```
.PHONY : clean
```

いったん、このようにした場合、‘clean’の有無にかかわらず、‘make clean’が実行されます。

偽のターゲットはほかのファイルから作られる実際のファイル名をつけないことが わかっているため、makeは偽のターゲットのサーチを行なう暗黙のルールを スキップします（see section 暗黙のルールの使用.）。このことから、実際のファイルが存在している場合でもそれを気にすることがないため、偽のターゲットの宣言が効果的なのです。

したがって、cleanを最初の行に置き、これを偽のターゲットにします。この場合のルールの書き方は以下のようになります。

```
.PHONY: clean
clean:
        rm *.o temp
```

偽のターゲットの利用のもう1つの有益な例は、makeの再帰的な呼び出しの 場合にあります。この場合、makefileはビルドされる多くのサブディレクトリの リストの変数を持つことが多く、これを処理する1つの方法は、サブディレクトリ間を 越えてシェルのコマンドが実行されるルールを扱うようにすることです。たとえば、以下のようにします。

```
SUBDIRS = foo bar baz

subdirs:
        for dir in $(SUBDIRS); do ¥
          $(MAKE) -C $$dir; ¥
        done
```

しかしながら、この方法にはいくつか問題があります。1番目の問題は、このルールでは、submakeのなかでなんらかのエラーが起きた場合にそれが 記録されないことです。したがって、エラーが起きてもほかのディレクトリでの ビルドは続行されます。エラーとそれによる終了はシェルコマンドの追加によって そうさせないことはできますが、その場合はmakeに-kオプションが ついていても実行されてしまいます。2番目の問題は、より重要ですが、ルールが1つであるため、makeの持つパラレルビルドの機能を利用することが できません。

しかし、サブディレクトリが偽のターゲット（サブディレクトリはつねに存在する ようにしておかなければビルドされません）であるように宣言することによって、これらの問題を取り除くことが可能です。

```
SUBDIRS = foo bar baz

.PHONY: subdirs $(SUBDIRS)

subdirs: $(SUBDIRS)

$(SUBDIRS):
        $(MAKE) -C $
foo: baz
```

この例では、'baz'サブディレクトリがビルドされたあとでさえも'foo' サブディレクトリがビルドされないように宣言しています。このような関係の 宣言はパラレルビルドを行なう際にはとくに重要です。

偽のターゲットは実際のターゲットファイルの必要条件である必要はなく、もしそうであっても、makeがそのファイルをアップデートするたびに 命令が実行されます。偽のターゲットが実際のターゲットの必要条件でない限り、偽のターゲットの命令は偽のターゲットが特別なゴールであるときにだけ 実行されます(see section ゴールを指定する引数.)。

また、偽のターゲットは必要条件を持つことができます。たとえば、1つのディレクトリに複数のプログラムがある場合、1つの'.Makefile'で すべてのプログラムを記述できるほうが便利です。デフォルトで再構成されたターゲットがmakefileの最初のものであるため、'all'と名付けられた 偽のターゲットおよび必要条件をすべての個々のプログラムに与えるのが 共通になります。たとえば、

```
all : prog1 prog2 prog3
.PHONY : all

prog1 : prog1.o utils.o
        cc -o prog1 prog1.o utils.o

prog2 : prog2.o
        cc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o
        cc -o prog3 prog3.o sort.o utils.o
```

とすることによって、'make'によって3つのプログラムを 再構築することができ、その際に引数として扱うこともできます ('make prog1 prog3'のように)。

偽のターゲットが別のターゲットの必要条件である場合にはサブルーチンとして 働きます。たとえば、この'make cleanall'はオブジェクトファイル、差分ファイル、'program'ファイルを削除します。

```
.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff
        rm program

cleanobj :
        rm *.o

cleandiff :
        rm *.diff
```

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.5 命令または必要条件なしのルール

ルールが必要条件や命令を何も持たない場合、またルールのターゲットが 存在しないファイルである場合、makeはルールが実行されるたびに そのターゲットを仮定することになります。これは、依存しているすべての ターゲットがそれらの命令の実行をつねに行なうことを暗示しています。

以下の例を見てください。

```
clean: FORCE
        rm $(objects)
FORCE:
```

ここで、ターゲット‘FORCE’は特別な条件を満たしています。つまりターゲット‘clean’は強制的にその命令を実行させられます。名前の‘FORCE’には特別な意味はありませんが、このような場合に 一般的に使用される名前の1つです。

見てわかるように、‘FORCE’を使用することは、‘.PHONY: clean’を 使用した場合とまったく同じ結果になります。

‘.PHONY’を使用することはより明確で効率的な方法です。しかしながら、makeのほかのバージョンでは‘.PHONY’をサポートしていないため、‘FORCE’がmakefile中に出てきます。See section 偽のターゲット.。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.6 レコードイベントに対する空のターゲットファイル

*empty target*は偽のターゲットのバリエーションで、ときどき明示的に要求する動作のために命令を保持するために使用されます。しかし、偽のターゲットと異なり、ファイルの実態を持つことができますが、内容が重要ではなく通常は空です。

空のターゲットファイルの目的は、ルールの命令が実行されたときの 最終更新時刻を記録することです。つまり、命令の1つがターゲットファイルの アップデートのためのtouchコマンドだからです。

また、空のターゲットファイルはいくつかの必要条件を持たなければなりません（そうでないと無意味だからです）。空のターゲットを再構成する要求をする場合、いずれかの必要条件がターゲットよりも新しい場合に命令が実行されます。すなわち、ターゲットを再構成した最終の時刻のあとに必要条件が変更された場合、命令が実行されます。例をあげます。

```
print: foo.c bar.c
        lpr -p $?
        touch print
```

この‘make print’ルールは、最後に‘make print’を実行されてから ソースファイルが変更されたかどうかでlprコマンドを実行します。自動変数‘$?’は変更のあったファイルだけを出力する際に使用されます（see section 自動変数.）。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.7 特別な組み込み済みのターゲット名

ターゲットとして使用される場合、いくつかの名前は特別な意味を持ちます。

.PHONY

> 特別なターゲット.PHONYの必要条件は偽のターゲットであると みなされます。それがターゲットであるとみなされる場合には、makeはその名前のファイルが存在するかどうかや更新時刻によらず 無条件に命令を実行します。See section 偽のターゲット.。

.SUFFIXES

> 特別なターゲット.SUFFIXESの必要条件は、サフィックスの一覧で、サフィックスルールのチェックに用いられます。See section 古いスタイルのサフィックスルール.。

.DEFAULT

> .DEFAULTを指定する命令はルール（明示的でも暗黙のルールでも）が みつからない場合にすべてのターゲットに対して適用されます。See section 最後の手段のルールの定義.。.DEFAULTが指定される

とき、ルールのターゲットではなく必要条件として 言及されるすべてのファイルはそれらの命令を実行されることになります。See section 暗黙のルールのサーチアルゴリズム.。

.PRECIOUS

.PRECIOUSが依存するターゲットは次のような特殊な扱いをされます。もしmakeがkillされたり命令の実行が中断された場合、ターゲットは 削除されません。See section makeの中断あるいはkill.。また、もしターゲットが中間ファイルである場合には、たとえ不要になったあとでも 通常は削除されることはありません。See section 暗黙のルールの連鎖.。この後者の点については、.SECONDARYの特別なターゲットと重なる部分があります。

また、ターゲットパターンがファイル名にマッチするルールによって生成された 中間ファイルを保持するために特別なターゲット.PRECIOUSの必要条件として 暗黙のルール（たとえば‘%.o’）のターゲットパターンのリストをすることが 可能です。

.INTERMEDIATE

.INTERMEDIATEが依存するターゲットは中間ファイルとして扱われますが、必要条件のない.INTERMEDIATEは何の効果も持ちません。See section 暗黙のルールの連鎖.。

.SECONDARY

.SECONDARYが依存するターゲットは自動的に削除されない限り、中間ファイルとして扱われます。See section 暗黙のルールの連鎖.。

また、必要条件のない.SECONDARYはすべてのターゲットをセカンダリ（二次的な）として扱います（すなわち、中間ファイルであるとみなされるため ターゲットは削除されないのです）。

.DELETE_ON_ERROR

makefileにおいて.DELETE_ON_ERRORがターゲットとして記述されている 場合、ルールが変更されて、シグナルを受け取って終了ステータスが ゼロ以外で終了した場合には、makeはルールのターゲットを削除します。See section コマンドのエラー.。

.IGNORE

.IGNOREに対して必要条件を指定する場合、makeは特定の ファイルに対して命令の実行時にエラーが起きてもそれを無視します。.IGNOREのための命令はあまり意味がありません。

必要条件なしのターゲットとして言及する場合、.IGNOREはすべての ファイルに対する命令の実行におけるエラーを無視するようにします。この‘.IGNORE’の使用は歴史的な互換性のためだけにサポートされています。makefileにおけるすべての命令に対する影響はあまり有益ではないため、特定の命令のエラーに対してはより選択的な方法がすすめられます。See section コマンドのエラー.。

.SILENT

.SILENTに対する必要条件を指定する場合、makeは実行前に それらの特定のファイルを再構成する際に命令を出力しません。したがって .SILENTに対する命令はあまり有益ではありません。

必要条件なしでターゲットについて言及する場合は、.SILENTは命令の 実行前にいかなる命令も出力しないようにします。‘.SILENT’のこの用法は、歴史的な互換性のためにサポートされています。See section コマンドエコー.。特定の命令を出力させないようにするにはもっと選択的な方法がすすめられます。この場合はmakeの実行時に‘-s’あるいは‘--silent’ オプションを用います（see section オプションのサマリー.）。

.EXPORT_ALL_VARIABLES

たんにターゲットとして言及されることによって、makeに対して デフォルトで子プロセスにすべての変数をエクスポートします。See section サブのmakeへの変数の伝達.。

.NOTPARALLEL

.NOTPARALLELがターゲットとして言及される場合は、‘-j’ オプションがつけられてもmakeはシリアルに実行されます。しかし、makefileがこのターゲットを含んでいない限り、再帰的なmakeの呼び出しは パラレルに行なわれます。そして、このターゲットの必要条件はすべて無視されます。

ターゲットとして現われる場合には、暗黙のルールのサフィックスもまた特別な ターゲットとしてカウントされ、'.c.o'のように2つのサフィックスの連結が 行なわれます。これらのターゲットはサフィックスのルールであり、暗黙のルールの 定義としては古い方法です（広く使われてはいますが）。原則的に、2つに分解し 両方のサフィックスをリストに追加した場合は、どんなターゲットの名前でも この方法では特別になるでしょう。実際には、サフィックスは通常は'.'で 始まるため、特別なターゲットの名前もまた、'.'で始まることになります。See section 古いスタイルのサフィックスルール.。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.8 ルール中の複数のターゲット

複数のターゲットを持つルールはたくさんのルールを記述することに等しく、それぞれが1つのターゲットを別々に記述することと同じです。すべてのターゲットに同じ命令を適用する場合には、'$@'を使用する ことによって実際のターゲットを代用できるため、効果は変わるかもしれません。

この2つの場合では有益です。

- たとえば、必要条件だけで命令は不要な場合、

  ```
  kbd.o command.o files.o: command.h
  ```

  は、3つのオブジェクトファイルのそれぞれに対して必要条件を与えます。

- 同様な命令がすべてのターゲットに対して働きます。自動変数'$@'が特定の ターゲットの代用となり命令を再構成するので、命令は完全に同一である必要は ありません（see section 自動変数.）。たとえば、

  ```
  bigoutput littleoutput : text.g
          generate text.g -$(subst output,,$@) > $@
  ```

  は、以下と同じです。

  ```
  bigoutput : text.g
          generate text.g -big > bigoutput
  littleoutput : text.g
          generate text.g -little > littleoutput
  ```

  ここで、仮想のプログラムgenerateは'-big'と'-little'に よって2つのタイプの出力をします。subst機能の説明については、See section 文字列の代用と分析のファンクション.。

ターゲットに従って必要条件を変更したい場合には、変数'$@'によって 命令を変更することができます。ここで、普通のルールでは複数のターゲットに ついてこの方法は用いることができませんが、*static pattern rule*（静的なパターンルール）の場合は可能です。See section 静的なパターンルール.。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.9 1つのターゲットに対する複数のルール

1つのファイルがいくつかのルールのターゲットになる場合があります。この場合、すべてのルールのなかで言及されるすべての必要条件が、ターゲットに対する 必要条件のリストにマージされます。そして、ターゲットがルールの必要条件よりも 古い場合には命令が実行されます。

あるファイルに対して実行される命令のセットは1つだけかもしれませんが、同じファイルに対して複数のルールが命令を与える場合には、makeは 最後のセットを与えてエラーを出力します。（ただし、ファイル名がドットで 始まる場合にはメッセージは出力されません。この奇妙な動作はほかの makeのインプリメンテーションとの互換性のためです。）したがって、makeの出力するメッセージがあるため、この方法でmakefileを 記述する必要はありません。

必要条件だけを持つ特別なルールは、同時に多くのファイルにいくつかの特別な 必要条件を与えるのに使用されます。たとえば、objectsという名前を持つ 変数は通常、コンパイラが出力するファイルのリストを含んでいますが、もし、‘config.h’の変化が以下のように書かれるならば、すべて再コンパイルされなければならず、これを簡単にする方法は、以下のようになります。

```
objects = foo.o bar.o
foo.o : defs.h
bar.o : defs.h test.h
$(objects) : config.h
```

これによって、どのようにしてオブジェクトファイルを作るかというルールを 変更せずに挿入あるいは取り出しが行なわれ、付加的な必要条件を間欠的に 追加したい場合には便利な書式となります。

ただし、makeのコマンドの引数によるセットを変数に与えて付加的な 必要条件が特定されるという別の影響もあります（see section 変数のオーバーライド.）。たとえば、

```
extradeps=
$(objects) : $(extradeps)
```

は、コマンド‘make extradeps=foo.h’は‘foo.h’をそれぞれの オブジェクトファイルの必要条件としてみなしますが、makeは文句を いいません。

ターゲットに対する明示的なルールのどれもが命令を持たない場合には、makeはいくつかの命令を発見する暗黙のルールのサーチを行ないます（see section 暗黙のルールの使用.）。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.10 静的なパターンルール

*静的なパターンルール*は複数のターゲットを指定し、ターゲットの名前に 基づいてそれぞれのターゲットに対して必要条件の名前を作製します。ターゲットは同一の必要条件を持つ必要がないため、複数のターゲットを持つ 通常のルールよりもより一般的です。このとき必要条件は*類似*していなければ なりませんが、*同一*である必要はありません。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.10.1 静的なパターンルールのシンタックス

ここに静的なパターンルールのシンタックスの例があります。

```
targets …: target-pattern: dep-patterns …
        commands
        …
```

*targets*のリストはルールの適用されるターゲットを指定します。そして そのターゲットは通常のルールとまったく同様にワイルドカードを含むことが 可能です（see section ファイル名におけるワイルドカードの使用.）。

*target-pattern*と*dep-patterns*は、それぞれのターゲットの 必要条件の計算法について示します。それぞれのターゲットは*target-pattern*に マッチし名前の部分展開を行ないます。これを*stem*と呼びます。このstemは *dep-patterns*のそれぞれに代用され、必要条件の名前の作成に用いられます。

個々のパターンは通常、‘%’を1個含んでいます。*target-pattern*が ターゲットにマッチした場合、‘%’はターゲット名のどの部分にも マッチすることができます。この部分のことを*stem*といいます。この部分以外は正確

にマッチする必要があり、たとえばターゲット‘foo.o’が パターン‘%.o’にマッチするとき‘foo’がstemとなります。そして、ターゲット‘foo.c’と‘foo.out’はそのパターンとはマッチしません。

個々のターゲットに対する必要条件の名前は、それぞれの必要条件のパターンの なかの‘%’に対するstemの置き換えによって作成されます。たとえば、ある必要条件のパターンが‘%.c’である場合、stemの代用である ‘foo’は必要条件の名前である‘foo.c’を与えます。また、‘%’を 含まない必要条件のパターンを記述することもまったく問題はありません。この場合はすべてのターゲットに対して同じ必要条件となります。

パターンルールにおける‘%’文字は先行するバックスラッシュによって 引用されます。‘%’を違った形で引用するバックスラッシュは さらに多くのバックスラッシュで引用できます。‘%’文字あるいは ほかのバックスラッシュを引用するバックスラッシュは、ファイル名と比較されたり stemを代用させるまえにパターンから削除されます。‘%’文字を引用する 可能性のないバックスラッシュは悩みの種ではありません。たとえば、パターン‘the¥%weird¥¥%pattern¥¥’は‘%’文字に先行する ‘the%weird¥’を持ち、‘pattern¥¥’があとに続きます。最後の2つのバックスラッシュはどの‘%’にも影響を持たないため 取り残されます。

ここに、‘.c’に対応した‘foo.o’と‘bar.o’のそれぞれを コンパイルする例があります。

```
objects = foo.o bar.o

all: $(objects)

$(objects): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
```

ここで、‘$<’は必要条件の名前を保持する自動変数で、‘$@’は ターゲットの名前を保持する自動変数です。詳しくは、自動変数.を参照してください。

個々のターゲットはターゲットパターンにマッチしているはずで、そうでない場合は警告が出ます。パターンにマッチするわずかなファイルの リストを持つ場合、マッチしないファイル名を削除するためにfilter 機能を使用することができます（see section 文字列の代用と分析のファンクション.）。

```
files = foo.elc bar.o lose.o

$(filter %.o,$(files)): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
$(filter %.elc,$(files)): %.elc: %.el
        emacs -f batch-byte-compile $<
```

この、‘$(filter %.o,$(files))’の結果の例では‘bar.o lose.o’を示し、最初の静的なパターンルールは、対応するCのソースファイルのコンパイルによって アップデートされるオブジェクトファイルを作り出します。‘$(filter %.elc,$(files))’の結果は、‘foo.elc’となり、‘foo.el’からファイルが作り出される結果になります。

別の例では、静的なルールにおける$*の使用法を示しています。

```
bigoutput littleoutput : %output : text.g
        generate text.g -$* > $@
```

generateコマンドが実行されるとき、$*は‘big’あるいは ‘little’のstemに展開されます。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.10.2 静的なパターンルール対暗黙のルール

静的なパターンルールは、パターンルールとして定義された暗黙のルールを 一般的に多く持っています（see section パターンルールの定義と再定義.）。それらの両方ともにターゲットに対するパターンと必要条件の 名前を構成するパターンを持ちますが、その違いはmakeがそのルールを いつ適用するかです。

パターンとマッチするどんなターゲットに対しても暗黙のルールは適用することが できます。しかし、それは、ターゲットが異なる形で指定された命令を 持たないときと必要条件が発見されたときだけに限られます。も

し、複数の暗黙のルールが適用可能であるならば、そのなかの1つだけが適用されます。その適用の選択はルールの順序に依存します。

静的なパターンルールは、ルール中で指定したターゲットの正確なリストに 適用されますが、ほかのターゲットには適用できず、指定されたターゲットの それぞれに対して適用されます。仮に2つの矛盾したルールが適用された場合で 両方が命令を持つ場合にはエラーとなります。

静的なパターンルールは以下のような理由で暗黙のルールよりも優れています。

- シンタックスで分類できない名前を持ち、明示的なリストを与えられる いくつかのファイルに対する通常の暗黙のルールをオーバーライドすることを 望む場合もあります。
- 使用しているディレクトリの正確な内容について確信が持てない場合、なんらかの不適切なファイルがmakeに誤った暗黙のルールを 使用させてしまうかについて確信が持てないでしょう。その場合の選択は 暗黙のルールがサーチした順序に依存します。静的なパターンルールによって、不確実性はなくなり、指定されたターゲットに対して正確に個々のルールが 適用されます。

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.11 ダブルコロンルール

*ダブルコロン*ルールはターゲットの名前のあとに‘:’のかわりに‘::’の 書かれたルールです。

複数のルールに1つのターゲットが現われる場合、すべてのルールは同じタイプである はずで、すべて普通のものか、あるいはすべてダブルコロンルールです。ダブルコロンルールの場合にはそれぞれが独立したものです。個々の ダブルコロンルールの命令は、そのルールのいかなる必要条件よりもターゲットが 古い場合に実行されます。これは、場合によってはダブルコロンルールのすべて、あるいは何も結果として生じないかもしれません。

同じターゲットについてのダブルコロンルールは完全に互いに別のものです。それぞれのダブルコロンルールはあたかも異なるターゲットが処理されるように 個別に処理されます。

あるターゲットについてのダブルコロンルールは、makefileでの出現順序で 実行されます。しかしながら、ダブルコロンルールが本当に意味があるのは 命令を実行する順序が重要ではない場合です。

ダブルコロンルールは多少わかりにくく、いつでも有益だというわけではありません。ターゲットをアップデートするために用いられる方法において異なり、アップデートを必要とする必要条件に依存する場合の仕組みを提供しますが、そのようなケースは稀です。

個々のダブルコロンルールは命令を指定しなければならず、さもないと 暗黙のルールが適用されてしまいます。See section 暗黙のルールの使用..

[<] [>] [<<] [上] [>>] [冒頭] [目次] [見出し] [?]

4.12 必要条件の自動生成

あるプログラムについてのmakefileにおいて、記述しなければならない多くのルールは いくつかのヘッダファイルに依存するオブジェクトファイルだけである場合が ほとんどです。たとえば、#includeによって‘main.c’が‘defs.h’を使用する場合、以下のように記述します。

```
main.o: defs.h
```

‘defs.h’が変更された場合にいつでも‘main.o’は 再構成されなければならないことをmakeが知っている必要があるので、このようなルールが必要ですが、規模の大きなプログラムの場合に、このようなルールをmakefileにたくさん記述しなければならないことが すぐにわかるはずです。そして、#includeを加えたり削除したりする際に つねにmakefileをアップデートすることに注意しなければならないのです。

そのようなことを避けるために、最近のCコンパイラはソースファイルの #include行を見てこれらのルールを書けるようになっています。通常は、‘-M’オプションをつけることによって可能です。たとえば、

```
cc -M main.c
```

は、以下のような出力を生成します。

```
main.o : main.c defs.h
```

したがって、もはや、そのようなルールを自分で書く必要はなく、コンパイラがやってくれます。

ここで、makefileのなかで‘main.o’に言及している必要条件に注意して ください。それは、暗黙のルールのサーチによる中間ファイルとして みなされていません。このことは、makeがそのファイルを使用したあとに 削除しないことを意味しています（see section 暗黙のルールの連鎖.）。

古いmakeプログラムの場合、‘make depend’というコマンドによって 要求に応じて必要条件を生成するコンパイラの機能を使用することが伝統的に 有効でしたが、そのような命令は必要条件を自動生成する‘depend’という ファイルを作ってしまいます。そしてmakefileはincludeを使用して それを読み込むことになります（see section ほかのMakefileをインクルードする.）。

GNUのmakeにおいては、makefileの再構成をする機能は時代遅れとなって います。つまり、makeに対して明示的に必要条件の再生成を命じる 必要はないのです。なぜなら期限切れのいかなるmakefileも つねに再生成するからです。See section Makefileの作られ方.。

必要条件の自動生成についての実践的な方法は、それぞれのソースファイルに 対して1つのmakefileを持つことです。‘*name*.c’の個々の ソースファイルについて‘*name*.d’という1つのmakefileを持ち、オブジェクトファイル‘*name*.o’がどれに依存しているかのリストを 持つことです。この方法によって、新しい必要条件を生み出すために変更のあった ソースファイルだけがスキャンされることになります。

ここに、‘*name*.c’というCのソースファイルから ‘*name*.d’という必要条件のファイルを生成するパターンルールが あります。

```
%.d: %.c
        set -e; $(CC) -M $(CPPFLAGS) $< ¥
                  | sed 's/¥($*¥)¥.o[ :]*/¥1.o $@ : /g' > $@; ¥
                [ -s $@ ] || rm -f $@
```

パターンルールを定義する情報については、See section パターンルールの定義と再定義.。シェルに対する‘-e’フラグは、$(CC)コマンドの失敗（ゼロ以外のステータスで終了する場合）の際にただちに終了させます。通常は、パイプラインの最後のコマンドのステータスで終了するため、makeはコンパイラからのゼロ以外のステータスに気づきません。

GNUのCコンパイラの場合、‘-M’のかわりに‘-MM’フラグを 使用するでしょう。これはシステムヘッダファイルの必要条件を省略します。詳細は、See (gcc.info)Preprocessor Options section ‘Options Controlling the Preprocessor’ in *Using GNU CC*.。

sedコマンドの目的は（たとえば）翻訳することです。

```
main.o : main.c defs.h
```

という内容を

```
main.o main.d : main.c defs.h
```

と翻訳します。この例では、個々の‘.d’ファイルが、対応する‘.o’ ファイルが依存するすべてのソースとヘッダファイルに依存しています。そして、makeはソースファイルやヘッダファイルに変更があるときにはつねに 必要条件の再生成が必要であることを知っています。

一度‘.d’ファイルの再構成をするルールを定義すると、include ディレクティブを使用してすべてを読み込むことが可能です。See section ほかのMakefileをインクルードする.。たとえば、

```
sources = foo.c bar.c

include $(sources:.c=.d)
```

(この例ではソースファイル‘foo.c bar.c’のリストを必要条件のmakefileの‘foo.d bar.d’に翻訳するために代替変数による参照を使用しています。See section 代用の参照.。)‘.d’ファイルはほかと似ているmakefileのため、makeはほかに何もせずに必要に応じて再構成してくれるのです。See section Makefileの作られ方.。

[<<] [>>] [冒頭] [目次] [見出し] [?]

この文書は*新堂 安孝*によって*2009年9月22日*に*texi2html 1.82*を用いて生成されました。