OMake Japanese Documentation

リリース *1.3.0*

Jason Hickey 著, 齋藤 真樹 訳

2010 年 10 月 23 日

Contents

CHAPTER 1

ガイド

1.1 OMakeの世界へようこそ！

OMake の世界へようこそ！このドキュメントでは、ビルドツールである OMake についての簡単な使い方や文法、さらには関数のリファレンスや詳細なコマンドラインオプションなどについて掲載しております。

GNU make 自身が持つ様々な文法上、機能上の欠点を克服するために、様々なビルドツールがこれまでに提案されてきました。しかしながらその殆どが Make とは程遠い独自の文法を採用していたり、あるいは機能的に不完全であったりするため、大きな学習コストとリスクが実際に業務として使用する際に必要とされました。OMake は GNU make の文法を自然に拡張した言語であるので学習コストが低く、かつ make が持つ弱点を殆ど解消しております。OMake 言語は関数型であり、オブジェクト指向です。もしあなたがこれらの概念について親しみをもっているのでしたら、そこまで苦労することなく習得できると思います。また、OMake は OCaml を利用して制作されていますが、OCaml の知識を別途習得する必要はありません。ご安心ください。

このドキュメントがあなた (方) の開発サイクルを加速する一助となりますように！

1.2 注意事項

OMake **マニュアル 日本語訳** *v1.3.0*

このドキュメントは OMake バージョン 0.9.8.5 の マニュアル を日本語訳したものです。 **この翻訳に関して原作者の許可は取っておりませんので、正当な権利者からの申し立てにより予告なく削除する場合があります。**

原文のニュアンスを忠実に翻訳するよう心がけていますが、意図しない翻訳ミスやドキュメントの不備があるかもしれません。よって、 **内容の正確さについて保証することはできません。**

1.3 翻訳に関して

- 忠実に翻訳するというよりは読みやすさを重視して、多少崩した形の翻訳となっています。
- この章を除いて、特に指定されていない括弧内の意見は全て原作者の意見です。訳者の意見はすべて『**訳注**:』 をつけています。
- 字句解析については翻訳者の専門外ですので、意図しない翻訳ミスが含まれている可能性が高いです。

- OMake 独自の専門用語も分かりやすさのため翻訳していますが、後に原文と比較できるようにするため、括弧書きで原文を載せています。
- 原文と翻訳文を比較したい方はウェブサイト左の『ソースコードを表示』で rst ファイルを参照してください。コメントに原文を載せています。
- このドキュメントは Sphinx を用いて生成しました。元のコードを参照したい方は、Web サイト上の Git リポジトリにアクセスしてください。

1.3.1 翻訳者について

このドキュメントは **齋藤 真樹** *(rezoo)* が翻訳しました。翻訳に関して不備がある場合は連絡をいただけると嬉しいです。

- mailaddr: rezoolab@gmail.com
- blog_url: http://mglab.blogspot.com/

CHAPTER 2

OMakeクイックスタートガイド

2.1 概要

OMake は複数のディレクトリにまたがって存在するソースファイルをビルドするために製作されたツールです。OMake を使用したプロジェクトは通常、`OMakefile` を各々のプロジェクトディレクトリに置き、`OMakeroot` をプロジェクトのルートディレクトリに置きます。`OMakeroot` には一般的なビルドルールを指定し、`OMakefiles` にはそれぞれのサブディレクトリに固有のビルドパラメータを指定します。いったん OMake を起動すると、OMake はまず設定ファイルのあるディレクトリをスキャンし、すべての `OMakefile` を評価します。そしてプロジェクトは全体のビルドルールの集合としてビルドされます。

2.1.1 自動的な依存関係の解析

従来の make(1) プログラムでは以前から依存関係の解析が問題となっていました。OMake ではこの問題を、依存関係を生成するコマンドを指定した `.SCANNER` ターゲットを追加することで解決しました。たとえば、以下のルール

```
.SCANNER: %.o: %.c
    $(CC) $(INCLUDE) -MM $<
```

は `.c` ファイルの依存関係を生成する常套手段です。OMake はソースファイルの依存関係を知る必要がある時点で、自動的にソースファイルをスキャンし、依存関係を特定します。

2.1.2 ファイル内容から依存関係を解析

どのファイルの依存関係が変更されたかを知るために、OMake では MD5 による要約を用いてチェックを行います。各々のディレクトリで OMake が実行された後に、 OMake は依存関係の情報をプロジェクトルートディレクトリの `.omakedb` ファイルに保存します。いったん OMake が依存関係のルールファイルを保存したのであれば、OMake が最後に実行された時点で、ターゲット、依存関係、ソースコードに関して変更がない場合、ビルドは実行されません。また最適化として OMake は MD5 の再計算を、修正日時、サイズ、ノード番号に変更のないファイルに関しては実行しません。

2.2 既に make に慣れている人向けの注意事項

既に make コマンドに慣れているユーザが omake を使う際には、以下の make と omake の違いを列挙したリストについて留意しておきましょう。

- omake を用いると、あなたがビルドルールを自力で定義するよりもはるかに省力化できます。このシステムは多くのビルドイン関数を提供しています (`StaticCLibrary` や `CProgram` など)。第 13 章 (**ビルド関数とユーティリティ**) で説明した事項を用いると、これらのビルドはより簡単になります。
- `.SUFFIXES` や `.suf1.suf2:` などを用いた暗黙のルール (implicit rule) はサポートされていません。その代わりにあなたはワイルドカード `%.suf2: %.suf1` を用いるべきです。
- スコーピングは重要です。あなたは変数や `.PHONY` ターゲットを、使用する前に定義すべきです。(詳細は "*.PHONY*" を参照してください。)
- サブディレクトリをプロジェクトに組み入れる際には、 `.SUBDIRS:` ターゲットを用います。(詳細は "*.SUBDIRS*" を参照してください。)

2.3 小さな C プログラムのビルド

新しいプロジェクトを作る際にもっとも簡単な方法は、カレントディレクトリをプロジェクトのルートディレクトリに変え、 `omake --install` とコマンドを入力してデフォルトの `OMakefile` と `OMakeroot` をインストールすることです。

```
$ cd ~/newproject
$ omake --install
*** omake: creating OMakeroot
*** omake: creating OMakefile
*** omake: project files OMakefile and OMakeroot have been installed
*** omake: you should edit these files before continuing
```

デフォルトの `OMakefile` は C と OCaml のプログラムをビルドするためのセクションを含んでいます。さて、それでは OMake を用いて簡単な C プロジェクトをビルドしてみましょう。

私たちは `hello_code.c` という C ファイルを持っており、そのコードは以下であったとします。

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello world\n");
    return 0;
}
```

このファイルから `hello` というプログラムをビルドする場合、 `CProgram` という関数を使うことができます。 `OMakefile` に `hello_code.c` のソースコードから `hello` プログラムをビルドすることを指定するため、以下の一行を加えます (ファイルの拡張子はこれらの関数に渡していないことに注意してください)。

```
CProgram(hello, hello_code)
```

これで私たちはこのプロジェクトをビルドするために、OMake を実行することができます。最初に私たちが OMake を実行した時点で、OMake は依存関係の解析に `hello_code.c` を解析し、 `cc` コンパイラを用いてコンパイルします。一番最後の行にはどれだけ多くのファイルが解析されて、どれだけ多くビルドされて、そしてどれだけ多くの MD5 が計算されたのかが表示されます。

```
$ omake hello
*** omake: reading OMakefiles
*** omake: finished reading OMakefiles (0.0 sec)
- scan . hello_code.o
+ cc -I. -MM hello_code.c
- build . hello_code.o
+ cc -I. -c -o hello_code.o hello_code.c
- build . hello
+ cc -o hello hello_code.o
*** omake: done (0.5 sec, 1/6 scans, 2/6 rules, 5/22 digests)
$ omake
*** omake: reading OMakefiles
*** omake: finished reading OMakefiles (0.1 sec)
*** omake: done (0.1 sec, 0/4 scans, 0/4 rules, 0/9 digests)
```

私たちがコンパイルオプションを変更したいと思った場合、CProgram と書いてある行の前に CC と CFLAGS 変数を再定義することで可能となります。たとえば、現在私たちは -g オプションで gcc コンパイラを用いたいとします。加えて、デフォルトでビルドするために .DEFAULT ターゲットを指定したい、さらに Win32 システムで .exe と定義された EXE 変数を用いたいとします。これらの要望は以下のコードで実現できます。

```
CC = gcc
CFLAGS += -g
CProgram(hello, hello_code)
.DEFAULT: hello$(EXE)
```

以下は omake を実行させた場合の全体のコンソールです。

```
$ omake
*** omake: reading OMakefiles
*** omake: finished reading OMakefiles (0.0 sec)
- scan . hello_code.o
+ gcc -g -I. -MM hello_code.c
- build . hello_code.o
+ gcc -g -I. -c -o hello_code.o hello_code.c
- build . hello
+ gcc -g -o hello hello_code.o
*** omake: done (0.4 sec, 1/7 scans, 2/7 rules, 3/22 digests)
```

もちろん、プログラム中に複数のファイルをインクルードすることもできます。hello_helper.c という新しいファイルを追加したとしましょう。以下のように OMakefile を変更することで対応できます。

```
CC = gcc
CFLAGS += -g
CProgram(hello, hello_code hello_helper)
.DEFAULT: hello$(EXE)
```

2.4 巨大なプロジェクト

プロジェクトが成長するにつれて、コードからライブラリをビルドしたいと思うかもしれません。ライブラリは StaticCLibrary 関数を用いることでビルドすることができます。以下は二つのライブラリをビルドする際の OMakefile のサンプルです。

```
CC = gcc
CFLAGS += -g

FOO_FILES = foo_a foo_b
```

```
BAR_FILES = bar_a bar_b bar_c

StaticCLibrary(libfoo, $(FOO_FILES))
StaticCLibrary(libbar, $(BAR_FILES))

# hello プログラムは両方のライブラリを用いてリンクされます
LIBS = libfoo libbar
CProgram(hello, hello_code hello_helper)

.DEFAULT: hello$(EXE)
```

2.5 サブディレクトリ

プロジェクトがさらに成長していく時点で、いくつかのディレクトリにコードファイルを分割することは良いアイデアです。現在私たちは libfoo と libbar をサブディレクトリに置いてあるものとしましょう。

まず、 OMakefile を各々のサブディレクトリ内に置きます。例えば、以下は foo/ サブディレクトリ内の OMakefile のサンプルです。

```
INCLUDES += .. ../bar

FOO_FILES = foo_a foo_b
StaticCLibrary(libfoo, $(FOO_FILES))
```

INCLUDES 変数はプロジェクトの他のディレクトリをインクルードするために定義されています。

さて、次のステップはメインプロジェクトの中にサブディレクトリをリンクさせます。プロジェクトの OMakefile を .SUBDIRS プロジェクトを含めるように修正します。

```
# プロジェクトの設定
CC = gcc
CFLAGS += -g

# サブディレクトリ
.SUBDIRS: foo bar

# ライブラリはサブディレクトリの中に存在します
LIBS = foo/libfoo bar/libbar

CProgram(hello, hello_code hello_helper)

.DEFAULT: hello$(EXE)
```

変数 CC と CFLAGS は .SUBDIRS ターゲットの **前に** 定義されてあることに注目してください。これらの変数はサブディレクトリでも保持されていますので、 libfoo と libbar では gcc -g が使われます。

二つのディレクトリに異なる設定を用いる必要がある場合、二つの方法があります。一つ目は各々のサブディレクトリの OMakefile にそれぞれの設定を書く方法です (普通はこのようにして問題を解決します)。二つ目は、ルートの OMakefile を以下のコードに書き換える方法です。

```
# 通常のプロジェクト設定
CC = gcc
CFLAGS += -g

# libfoo は通常の設定を用います
.SUBDIRS: foo
```

```
# libbar は加えて最適化を行います
CFLAGS += -O3
.SUBDIRS: bar

# メインプログラム
LIBS = foo/libfoo bar/libbar
CProgram(hello, hello_code hello_helper)

.DEFAULT: hello$(EXE)
```

CFLAGS 変数にさらに -O3 オプションを加えることで、hello_code.c と hello_helper.c の両方は -O3 オプションでコンパイルされます。libbar のみにオプションを変更させたい場合、 section 文を使用することで bar/ サブディレクトリ内のみに変更を適用することができます。

```
# 通常のプロジェクト設定
CC = gcc
CFLAGS += -g

# libfoo は通常の設定を用います
.SUBDIRS: foo

# libbar は加えて最適化を行います
section
    CFLAGS += -O3
    .SUBDIRS: bar

# メインプログラムでは最適化を使用しません
LIBS = foo/libfoo bar/libbar
CProgram(hello, hello_code hello_helper)

.DEFAULT: hello$(EXE)
```

後で、私たちがこのプロジェクトを Win32 に移し、そして異なるコンパイラフラグや追加ライブラリが必要になることがわかったとしましょう。

```
# 通常のプロジェクト設定
if $(equal $(OSTYPE), Win32)
    CC = cl /nologo
    CFLAGS += /DWIN32 /MT
    export
else
    CC = gcc
    CFLAGS += -g
    export

# libfoo は通常の設定を用います
.SUBDIRS: foo

# libbar は加えて最適化を行います
section
    CFLAGS += $(if $(equal $(OSTYPE), Win32), $(EMPTY), -O3)
    .SUBDIRS: bar

# 通常のライブラリ
LIBS = foo/libfoo bar/libbar

# Win32 上のみ libwin32 を必要とします
if $(equal $(OSTYPE), Win32)
```

```
   LIBS += win32/libwin32

   .SUBDIRS: win32
   export

# メインプログラムでは最適化を使用しません
CProgram(hello, hello_code hello_helper)

.DEFAULT: hello$(EXE)
```

export によって if 文の中にある変数が外部にエクスポートされます。 *OMake* **での変数はスコープ化** されており、ネストされたブロック内の変数は通常、外部のブロックから使用することはできません。 export はネストされたブロック内の変数を、親のブロックに移す命令です。

ノート: 訳注: 今回の例では $(OSTYPE) 変数を用いて場合分けを行っていますが、 $(CC) 変数に関しては省略することができます。なぜなら、 Unix 上では $(CC) は gcc , Win32 上では cl /nologo が自動的に束縛されるからです。

大抵の場合において、このような設定変数は異なるプラットフォーム上においても正常に動作するよう設計されています (詳細は "*C/C++*用の設定変数" を参照してください)。もちろん、その他になにか追加オプションを指定したい場合には、このような場合分けが有効となるでしょう。

最後に、私たちはすべてのライブラリを共通の lib/ ディレクトリにコピーしたいものとします。私たちはまずはじめにディレクトリの変数を定め、そして lib の文字列を変数で置き換えます。

```
# 共用の lib ディレクトリ
LIB = $(dir lib)

# phony ターゲットはライブラリのみビルドを行います
.PHONY: makelibs

# 通常のプロジェクト設定
if $(equal $(OSTYPE), Win32)
    CC = cl /nologo
    CFLAGS += /DWIN32 /MT
    export
else
    CC = gcc
    CFLAGS += -g
    export

# libfoo は通常の設定を用います
.SUBDIRS: foo

# libbar は加えて最適化を行います
section
    CFLAGS += $(if $(equal $(OSTYPE), Win32), $(EMPTY), -O3)
    .SUBDIRS: bar

# 通常のライブラリ
LIBS = $(LIB)/libfoo $(LIB)/libbar

# Win32 上のみ libwin32 を必要とします
if $(equal $(OSTYPE), Win32)
   LIBS += $(LIB)/libwin32

   .SUBDIRS: win32
   export
```

```
# メインプログラムでは最適化を使用しません
CProgram(hello, hello_code hello_helper)

.DEFAULT: hello$(EXE)
```

$(LIB) ディレクトリの中にライブラリをインストールするため、ライブラリディレクトリ内の OMakefile を修正します。以下は新しく変更された foo/OMakefile です。

```
INCLUDES += .. ../bar

FOO_FILES = foo_a foo_b
StaticCLibraryInstall(makelib, $(LIB), libfoo, $(FOO_FILES))
```

ディレクトリ (そしてファイル名) は現在のパス名で評価されます。 foo/ ディレクトリ内では、 $(LIB) 変数は ../lib として評価されます。

各々のサブディレクトリ内で INCLUDES 変数を個別に定義する代わりに、以下のようにトップレベルで定義することもできます。

```
INCLUDES = $(ROOT) $(dir foo bar win32)
```

foo/ ディレクトリ内において INCLUDES 変数は文字列 .. . ../bar として評価されます。また bar/ ディレクトリ内において、 INCLUDES 変数は文字列 .. ../foo . ../win32 として評価されます。ルートディレクトリでは、 INCLUDES 変数は文字列 . foo bar win32 として評価されます。

ノート: 訳注: 今までの議論の中で「 *Win32* **プラットフォーム上でディレクトリのセパレータに** */(***スラッシュ***)** **を指定していいのだろうか？** *\(***バックスラッシュ***)* **を使わないといけないのでは？** 」と疑問に思った方も多いと思います。実は、この点に関しては心配いりません。omake はたとえ Win32 上で / を使っていたとしても、実際の評価ではー非常に奇妙に思われるかもしれませんがー \ が用いられるのです。同様に、セパレータに \ を用いたとしても Unix プラットフォーム上では / と変換されます。

このように、ディレクトリのセパレータは / , \ どちらを使っても、異なるプラットフォーム上で正常に動作します。ただし、 \ はエスケープ文字として使用されることが多いため、思わぬ誤動作を引き起こすことがあるかもしれません。大抵の場合、ディレクトリのセパレータは / を使うことをおすすめします (詳細は “**ファイルの検索とリスト**” を参照してください)。

2.6 その他の考慮事項

OMake はまたリソースのあるサブディレクトリも扱うことができます。例えば、先ほどの foo/ ディレクトリの中に、さらにいくつかのサブディレクトリを保持しているような場合を考えてみましょう。 foo/OMakefile は foo/ ディレクトリ自身の .SUBDIRS ターゲットを保持しており、また各々のサブディレクトリもまたサブディレクトリ自身の OMakefile を保持しています。

2.7 OCaml プログラムのビルド

通常、OMake は OCaml のプログラムをビルドするための関数群も持っています。OCaml プログラムのための関数群は接頭語として OCaml がつきます。例えば、前回のサンプルを OCaml で置き換えて、 hello_code.ml という以下のコードを含んだファイルを持っている場合を考えましょう。

```
open Printf

let () = printf "Hello world\n"
```

この簡単なプロジェクトの OMakefile のサンプルは以下になります。

```
# バイトコードコンパイラを使用
BYTE_ENABLED = true
NATIVE_ENABLED = false
OCAMLCFLAGS += -g

# プログラムをビルド
OCamlProgram(hello, hello_code)
.DEFAULT: hello.run
```

次に、二つのライブラリのサブディレクトリを持っている場合を考えましょう。 `foo/` ディレクトリは C で書かれており、 `bar/` ディレクトリは OCaml で書かれています。そして私たちは標準的な OCaml の UNIX モジュールを使用したいといった場合です。

```
# 通常のプロジェクト設定
if $(equal $(OSTYPE), Win32)
    CC = cl /nologo
    CFLAGS += /DWIN32 /MT
    export
else
    CC = gcc
    CFLAGS += -g
    export

# バイトコードコンパイラを使用
BYTE_ENABLED = true
NATIVE_ENABLED = false
OCAMLCFLAGS += -g

# ライブラリのサブディレクトリ
INCLUDES += $(dir foo bar)
OCAMLINCLUDES += $(dir foo bar)
.SUBDIRS: foo bar

# C ライブラリ
LIBS = foo/libfoo

# OCaml ライブラリ
OCAML_LIBS = bar/libbar

# Unix モジュールも用います
OCAML_OTHER_LIBS = unix

# メインプログラム
OCamlProgram(hello, hello_code hello_helper)

.DEFAULT: hello
```

`foo/OMakefile` は C ライブラリとして設定します。

```
FOO_FILES = foo_a foo_b
StaticCLibrary(libfoo, $(FOO_FILES))
```

また、 `bar/OMakefile` は ML ライブラリとして設定します。

```
BAR_FILES = bar_a bar_b bar_c
OCamlLibrary(libbar, $(BAR_FILES))
```

2.8 OMakefile と OMakeroot

プロジェクトを設定する際、OMake は OMakefile と OMakeroot の 2 つのファイルを使用します。これらのファイルの構文は同じですが、役割は全く異なります。さらに付け加えると、すべてのプロジェクトは OMakeroot ファイルをプロジェクトのルートディレクトリに必ず置かなければなりません。このファイルはこのディレクトリがプロジェクトのルートディレクトリであることを決め、さらにプロジェクトをセットアップするためのコードを含んでいます。対照的に、複数のディレクトリが存在するプロジェクトは、どのようにサブディレクトリ内のファイルをビルドするのかを設定した OMakefile を、各々のプロジェクトのサブディレクトリに置くことになります。

通常、OMakeroot ファイルはほとんど変更する必要のない決まり文句です。以下のリストは OMakeroot ファイルの一部です。

```
include $(STDLIB)/build/Common
include $(STDLIB)/build/C
include $(STDLIB)/build/OCaml
include $(STDLIB)/build/LaTeX

# コマンドライン変数を再定義
DefineCommandVars(.)

# 現在のディレクトリをプロジェクトの一部として設定
.SUBDIRS: .
```

include が書かれている行では、プロジェクトに必要な、標準的な設定ファイルをインクルードしています。$(STDLIB) 変数は OMake ライブラリのディレクトリを表します。OCaml が動作するために必須となる設定ファイルは Common だけで、その他の設定ファイルはなくても構いません。$(STDLIB)/build/OCaml ファイルはプロジェクトが OCaml で書かれたプログラムを含んでいる場合のみ必要となります。

DefineCommandVars 関数は (VAR=<value> のような形で) コマンドライン上から指定された、いくつかの変数を定義します。 .SUBDIRS が書かれている行では現在のディレクトリがプロジェクトの一部であることを指定しています (よって同ディレクトリの OMakefile が読み込まれます)。

通常、 OMakeroot ファイルはサイズが小さく、かつプロジェクトから独立しています。プロジェクト固有の設定はすべてプロジェクト上の OMakefile に設定すべきです。

2.9 複数のバージョンのサポート

OMake バージョン 0.9.6 では、複数の同じバージョンのプロジェクトや、予備として用いる複数のプロジェクトに関するサポートを導入しました。 vmount(dir1, dir2) 関数を用いることで、 dir1/ ディレクトリを dir2/ ディレクトリに『仮想的にマウント』することができます。『仮想的なマウント』は Unix にて、 dir1/ ディレクトリ内のファイルを dir2/ ディレクトリにマウントするが、新しいファイルは dir2/ ディレクトリに作られるようなものです。さらに具体的には、ファイル dir2/foo は、 dir1/foo が存在している場合は dir1/foo に置き換わりますが、存在していない場合は dir2/foo が用いられます。

vmount 関数によってプロジェクト内の複数のバージョンをビルドすることが容易になりました。 src/ ディレクトリ内にいくつかのソースファイルが入っており、これをデバッグサポートがついているバージョンと、最適化されたバージョンの 2 つにコンパイルする場合を考えてみましょう。まず debug/ と opt/ の 2 つのディレクトリを作成して、 src/ ディレクトリをこれらのディレクトリにマウントします。

```
section
    CFLAGS += -g
    vmount(-l, src, debug)
    .SUBDIRS: debug

section
```

```
CFLAGS += -O3
vmount(-l, src, opt)
.SUBDIRS: opt
```

ここで、 vmount 関数を必要としていないプロジェクトに適用させないために、私たちは section ブロックを用いて vmount のスコープ範囲を定義しました。

-l オプションはなくても構いません。それを指定することで src/ ディレクトリのファイルは、ターゲットとなるディレクトリにリンクされます (システムが Win32 の場合、ファイルはコピーされます)。ファイルが関連付けられるようにファイルへのリンクが追加されます。何もオプションが与えられていない場合、ファイル名は直接 src/ ディレクトリのファイル名に変換されます。

debug/ ディレクトリ内のファイルが参照された時点で、もし src/ ディレクトリにそのファイルが存在するならば、 debug/ ディレクトリのファイルは src/ のファイルにリンクされます。例えば、 debug/OMakefile が参照された場合、 src/OMakefile が debug/OMakefile としてリンクされます。

vmount 関数の動作はだいぶ透過的です。あなたは OMakefile をまるで src/ ディレクトリ内のファイルが関連付けられているかのように書くことができ、マウントについて気にする必要はありません。しかしながら、以下に示すいくつかの注意点を頭に入れておきましょう。

2.10 注意点

バージョン管理の目的で vmount 関数を用いた場合、コンパイルされたファイルをソースファイルから分離することをお勧めします。例えば、ソースディレクトリに src/foo.o ファイルが含まれている場合を考えましょう。もし src/foo.o がマウントされてしまったとすると、 foo.o ファイルはすべてのバージョンにおいて共通のファイルになってしまい、おそらくあなたが求めていた動作とは違うものになるでしょう。 src/ ディレクトリにはソースコード以外何もない状態にして、コンパイルされたコードは含めないようにしましょう。

vmount -l オプションを使用したときは、ソースファイルがプロジェクトから参照された場合のみ、バージョンディレクトリ内にリンクされます。ファイルシステムを用いる関数 ($(ls ...) など) はあなたが期待していない動作を引き起こすことになります。

CHAPTER 3

OMakeビルドサンプル

この章では OMake のビルド体系をさらにもう少し解説します。この議論を決定するただ一つの結論としては、OMake は全体のプロジェクト解析を元にしているということです。これはあなたがプロジェクト全体の設定を決めて、そして OMake のインスタンスを実行することを意味しています。

一つのディレクトリで構成されたプロジェクトではあまり意味がないかもしれませんが、多数のディレクトリからなるプロジェクトでは大きな意味を持ちます。 GNU make では、あなたは通常、プロジェクトの各々のディレクトリにおいて make プログラムを再帰的に呼び出すでしょう。例えば、あなたが現在サブディレクトリ lib と main が入っているソースディレクトリ src を含んでいるプロジェクトをいくつか持っているものとしましょう。具体的には、あなたのプロジェクト構成は以下のアスキーアートのようになります。

```
my_project/
|--> Makefile
`--> src/
     |---> Makefile
     |---> lib/
     |     |---> Makefile
     |     `---> source files...
     `---> main/
           |---> Makefile
           `---> source files...
```

一般的に GNU make では、初めに my_project/ の make インスタンスを呼び出します。この make インスタンスは src/ ディレクトリ内の make インスタンスを呼び出し、そして lib/ と main/ の新しいインスタンスを呼び出します。つまり、 GNU make では単純に、プロジェクト内の Makefile の数だけ make インスタンスが生成されることになります。

大量のプロセスを処理することは今日のコンピュータにとってさほど大きな問題ではありません (ときどき著者の意見に反対する人もいるようですが、私たちはもはや 1970 年代に住んでいるわけではないのです)。この構造に関する問題としては、各々の make プロセスの設定が分離されており、そしてそのすべてが調和のとれたものにするには、非常に多くの負担となってしまう点です。さらには、例えばプログラマが main/ ディレクトリで make プログラムを実行するが、 lib/ はもう時代遅れの代物であった場合を考えてみましょう。この場合、 make は楽しそうにあちこち曲がりくねった挙句、恐らく lib/ 内のファイルをリビルドしようと奮起して、恐らく諦めることとなるでしょう。

OMake ではこの構造を抜本的に変更します。とは言っても実際の変更点はそれほどありません。ソース構造は非常に似通っています。私たちは単純にいくつかの”O” を以下のアスキーアートのように加えただけです。

```
my_project/
|--> OMakeroot   (or Root.om)
|--> OMakefile
```

```
`--> src/
     |---> OMakefile
     |---> lib/
     |      |---> OMakefile
     |      `---> source files...
     `---> main/
            |---> OMakefile
            `---> source files...
```

各々の <dir>/OMakefile の役割は各々の <dir>/Makefile の役割と同様で、どのように <dir> のソースファイルをビルドするのかについて設定します。 OMakefile は Makefile の構造や構文を保持しておりますが、ほとんどの場合 make よりも簡単に記述することができます。

一つ小さな違いがあるとすれば、プロジェクトのルートディレクトリに OMakeroot が存在している点です。このファイルの主な目的は、第一にプロジェクトのルートディレクトリがどこにあるか示すことです (omake がサブディレクトリから呼び出されたときのためです)。 OMakeroot はブートストラップとして働きます。omake はこのファイルを最初に読み込んで実行されます。それ以外では、 OMakeroot の構文と機能は他の OMakefile と全く同様です。

大きな 違いは、OMake は **グローバルな** 解析を行うという点です。以下はどのように omake が動作するのかについて示します。

1. omake は OMakeroot ファイルがあるディレクトリに移動し、読んでいきます。
2. 各々の OMakefile は .SUBDIRS ターゲットを使用して OMakefile があるサブディレクトリを指し示します。例えば、 my_project/OMakefile は以下のルールを持っていたとします。

   ```
   .SUBDIRS: src
   ```

   omake はこれらのルールを読んで、プロジェクト内のすべての OMakefile を評価します。読み込みや評価は高速に行われるので、このプロセスは早期に終わります。
3. 全体の設定が読まれたら、omake はどのファイルが使われていないのかを決定し (グローバルな解析を使用します)、ビルド作業を開始します。これはどのファイルのビルドが実際に必要なのかに依存した、ビルド時間がかかります。

このモデルではいくつかの利点があります。初めに、解析をグローバルにしたことで、単一のビルド設定が用いられることとなり、ビルド設定を一定にするよう保証することがより簡単になるという点です。別の利点はビルドに関する設定が継承されて、再利用可能となり、さらに階層構造となる点です。概して、ルートの OMakefile はいくつかの標準的な決まり文句と設定を定義しており、これはサブディレクトリによって継承されて、調整したり、変更することができます (全体を書き換える必要はありません)。この方法の欠点は容量が増大することで、これは結局グローバルな解析を行っているためです。が、実際にはめったに考慮する必要があるようには見えません。OMake は大きなプロジェクトにおいてでさえ、あなたが使っているウェブブラウザよりもはるかに小さい容量しか使いません。

GNU/BSD の make ユーザは以下の点に留意してください。

- OMake は Makefile と同じくらい多くのファイルを作ります。構文は似ており、make と同様に多くのビルドイン関数が用意されています。しかしながら、この 2 つのビルドシステムは同じではありません。OMake ではいくつかの酷い機能 (これは著者の意見です) が外されており、それに代わって新しい機能が追加されています。
- OMake は Win32 を含んだ、複数のプラットフォーム上で同様に動きます。あなたは複数のプラットフォーム上で動かすためにコードを変更したり、いくつかのトリッキーなテクを使ったり、 $(OSTYPE) 変数を使ってビルド設定を調節する必要はありません。
- OMake の依存関係の解析は MD5 によるファイルの要約 (digest) を元にしています。これはつまり、依存関係の解析はファイルの『修正日時』ではなくファイルの『内容』を元にしていることを表しています。さあ、ローカル日時とファイルサーバの日時から生じるミスマッチや、間違ったタイムスタンプの変更によるビルドミスからおさらばしましょう。

3.1 OMakeroot vs. OMakefile

さて、例を見せる前に、一つ質問をしてみましょう。それは「プロジェクトルートの OMakeroot と OMakefile の違いは何か？」というものです。その質問に関する端的な答えは「違いはないが、あなたは必ず OMakeroot ファイル (あるいは Root.om ファイル) を作らなければならない」です。

しかしながら、通常の範囲で用いるならば OMakeroot は変更する必要のない決まり文句が並んでいるファイルであり、すべてのプロジェクトの OMakeroot は多かれ少なかれ同じような内容となるでしょう。

OMake を始めるために、あなたがこのような決まり文句を入力する必要はありません。ほとんどの場合において、あなたは以下の手順をプロジェクトのルートディレクトリで実行するだけです。

- omake --install をプロジェクトのルートで実行する。

これで最初の OMakeroot と OMakefile が生成されて、編集できるようになりました。

3.2 C プロジェクトのサンプル

OMake を始めるために、まずは簡単なサンプルから始めてみましょう。いま私たちは以下のファイルを含んだディレクトリツリーを持っているものとします。

```
my_project/
|--> OMakeroot
|--> OMakefile
`--> src/
     |---> OMakefile
     |---> lib/
     |     |---> OMakefile
     |     |---> ouch.c
     |     |---> ouch.h
     |     `---> bandaid.c
     `---> main/
           |---> OMakefile
           |---> horsefly.c
           |---> horsefly.h
           `---> main.c
```

以下は OMakeroot , OMakefile リストのサンプルです。

```
my_project/OMakeroot:
    # C アプリケーションの標準的な設定をインクルード
    open build/C

    # コマンドライン上の変数を処理
    DefineCommandVars()

    # このディレクトリの OMakefile をインクルード
    .SUBDIRS: .

my_project/OMakefile:
    # 標準的なコンパイルオプションを設定
    CFLAGS += -g

    # src ディレクトリをインクルード
    .SUBDIRS: src

my_project/src/OMakefile:
```

```
    # あなたの好きなようにオプションを付け加えます
    CFLAGS += -O2

    # サブディレクトリをインクルード
    .SUBDIRS: lib main

my_project/src/lib/OMakefile:
    # 静的ライブラリとしてライブラリをビルドします。
    # これは Unix/OSX 上では libbug.a として、
    # Win32 上では libbug.lib としてビルドされます。
    # 引数にはソースファイルの拡張子を入れていないことに注意してください。
    StaticCLibrary(libbug, ouch bandaid)

my_project/src/main/OMakefile:
    # いくつかのファイルは../lib ディレクトリ上の
    # .h ファイルをインクルードしています。
    INCLUDES += ../lib

    # どのライブラリに対してリンクしたいのかを指定
    LIBS[] +=
        ../lib/libbug

    # プログラムをビルドします。
    # Win32 上では horsefly.exe、
    # Unix 上では horsefly としてビルドされます。
    # 最初の引数はアプリケーション名を指定します。
    # 二番目の引数はソースファイルの配列を指定します (拡張子は抜いてください)。
    # これらの配列はビルドするプログラムの一部となります。
    CProgram(horsefly, horsefly main)

    # デフォルトでこのプログラムをビルドします
    # (他の引数を指定しないで omake が実行される場合です)。
    # EXE 変数は Win32 上では.exe として定義されていますが、
    # 他のプラットフォーム上では空の変数です。
    .DEFAULT: horsefly$(EXE)
```

ほとんどの設定は build/C.om (これは OMake がもつ全機能の一部です) ファイルに定義されています。このファイルはほとんどの作業の面倒を見てくれます。具体的には、

- C ライブラリやプログラムを正当な方法でビルドするための、 StaticCLibrary と CProgram 関数を定義しています。
- 依存関係を特定するために各々のソースコードを調べていくメカニズムを定義しています。これはつまり、C ソースファイルのための .SCANNER ルールを定義していることを意味しています。変数はサブディレクトリにも継承されていき、例えば、 src/main/OMakefile の CFLAGS 変数の値は "-g -O2" となります。

3.3 OCaml プロジェクトのサンプル

前回の C の代わりに OCaml を使った状態で、簡単なサンプルを作ってみましょう。今回では、ディレクトリツリーは以下のようになります。

```
my_project/
|--> OMakeroot
|--> OMakefile
`--> src/
```

```
 |---> OMakefile
 |---> lib/
 |      |---> OMakefile
 |      |---> ouch.ml
 |      |---> ouch.mli
 |      `---> bandaid.ml
 `---> main/
        |---> OMakefile
        |---> horsefly.ml
        |---> horsefly.mli
        `---> main.ml
```

OMakeroot , OMakefile のリストは前回と少し異なります。

```
my_project/OMakeroot:
    # OCaml アプリケーションの標準的な設定をインクルード
    open build/OCaml

    # コマンドライン上の変数を処理
    DefineCommandVars()

    # このディレクトリの OMakefile をインクルード
    .SUBDIRS: .

my_project/OMakefile:
    # 標準的なコンパイルオプションを設定
    OCAMLFLAGS += -Wa

    # バイトコードのコンパイラを使いたいですか？
    # それともネイティブコードのコンパイラを使いたいですか？
    # 今回は両方とも使ってみましょう。
    NATIVE_ENABLED = true
    BYTE_ENABLED = true

    # src ディレクトリをインクルード
    .SUBDIRS: src

my_project/src/OMakefile:
    # サブディレクトリをインクルード
    .SUBDIRS: lib main

my_project/src/lib/OMakefile:
    # ネイティブコードにおいて、積極的にインライン化を行う
    OCAMLOPTFLAGS += -inline 10

    # 静的ライブラリとしてライブラリをビルドします。
    # これは Unix/OSX 上では libbug.a として、
    # Win32 上では libbug.lib としてビルドされます。
    # 引数にはソースファイルの拡張子を入れていないことに注意してください。
    OCamlLibrary(libbug, ouch bandaid)

my_project/src/main/OMakefile:
    # いくつかのファイルは../lib ディレクトリ上の
    # インターフェースに依存しています。
    OCAMLINCLUDES += ../lib

    # どのライブラリに対してリンクしたいのかを指定
    OCAML_LIBS[] +=
        ../lib/libbug
```

```
# プログラムをビルドします。
# Win32 上では horsefly.exe、
# Unix 上では horsefly としてビルドされます。
# 最初の引数はアプリケーション名を指定します。
# 二番目の引数はソースファイルの配列を指定します (拡張子は抜いてください)。
# これらの配列はビルドするプログラムの一部となります。
OCamlProgram(horsefly, horsefly main)

# デフォルトでこのプログラムをビルドします
# (他の引数を指定しないで omake が実行される場合です)。
# EXE 変数は Win32 上では.exe として定義されていますが、
# 他のプラットフォーム上では空の変数です。
.DEFAULT: horsefly$(EXE)
```

この場合、ほとんどの設定は build/OCaml.om ファイルで定義されています。今回は特に、my_project/src/lib ファイルを -inline 10 オプションを用いて積極的にコンパイルするが、my_project/src/lib は普通にコンパイルする設定となっています。

3.4 新しい言語を扱う

前回の二つのサンプルは十分簡単なように見えますが、これは OMake の標準ライブラリ (build/C と /build/OCaml ファイル) がすべての仕事を行ってしまったためです。もし私たちが OMake の標準ライブラリでサポートされていないような言語のビルド設定を書こうとしたら、一体どのようにすれば良いのでしょうか？

例えば、私たちは OMake に新しい言語を適用させているものとしましょう。この言語は標準的なコンパイル/リンクモデルを用いていますが、OMake の標準ライブラリには存在していません。今回は問題をはっきりさせるため、以下のように動作する手順について考えてみましょう。

- .cat 拡張子 (Categorical Abstract Terminology) では複数あるファイルの中のソースファイルを定義します。
- catc コンパイラを用いて .cat ファイルは .woof (Wicked Object-Oriented Format) ファイルにコンパイルされます。
- .woof ファイルは実行可能な .dog (Digital Object Group) ファイルを生成するために、 -c オプションを用いた catc コンパイラによってリンクされます。 catc はまた -a オプションで、いくつかの .woof ファイルをライブラリに結合させることができます。
- 各々の .cat ファイルは他のソースファイルに関連付けることができます。もしソースファイル a.cat が行 open b を含んでいたのなら、 a.cat は b.woof ファイルに依存しており、 a.cat は b.woof が変更されたときに再コンパイルしなければなりません。 catc 関数は -I オプションで依存関係を示した行を探索することができます。

ノート: 訳注: これは cat, woof, dog にもじって作られた仮のソースコードであり、実際に存在しているわけではありません

ビルド設定を定義するために、私たちは以下の 3 つの作業を行う必要があります。

1. 依存関係の情報をソースファイルから探索するための .SCANNER ルールを定義する。
2. .cat ファイルを .woof ファイルにコンパイルするための普遍的なビルドルールを定義する。
3. 実行可能な .dog ファイルを生成するために、 .woof ファイルをリンクするためのルールを一つの関数として定義する。

初めに、これらの定義はプロジェクトルートの OMakefile に置くことになります。

3.4.1 通常の編集ルールの定義

さて、パート 2 に移って、通常の編集ルールを定義していきましょう。今回私たちはビルドルールについて、ソースコードに直接ルールを定義することにします。まずはインクルードパスを扱うために、インクルードパスを指定した変数 CAT_INCLUDES を定義します。これはディレクトリが格納されている配列です。そしてオプションを定義するために、私たちは『遅延評価変数 (lazy variable)(“**遅延評価式**“を参照)』を使用します。この場合は他にも標準的なフラグが存在していますので、 CAT_FLAGS 変数も定義することにしましょう。

```
# 私たちは今回 CATC 変数ををオーバーライドしたいので、catc コマンドを定義します
CATC = catc

# 通常のフラグは空にします
CAT_FLAGS =

# インクルードパスの辞書 (通常は空です)
INCLUDES[] =

# インクルードパスによるインクルードオプションを計算します
PREFIXED_INCLUDES[] = $`(mapprefix -I, $(INCLUDES))

# 通常の方法で.woof ファイルをビルドします
%.woof: %.cat
    $(CATC) $(PREFIXED_INCLUDES) $(CAT_FLAGS) -c $<
```

ノート: 訳注: 今回の場合、 $`(mapprefix -I, $(INCLUDES)) という表記法がまさに遅延評価に相当しています。 $`(v) は v を遅延評価するための表記法です。

最後の部分では、インクルードパスと前に定義されている CAT_FLAGS 変数を含んだ、 catc コンパイラを呼び出すというビルドルールを定義しています。 $< 変数はソースファイル名に置き換わります。

3.4.2 リンクするためのルールを定義

.woof ファイルをリンクするために、どのようにビルド作業を行うのかについて記述した、別のルールを記述します。ここでソースに直接ルールを定義するかわりに、リンク作業を記述した関数を定義することにします。この関数は二つの引数をとり、最初の引数は実行ファイル名 (拡張子なし) で、二つ目の引数はリンクするためのファイル名 (これもまた拡張子はなし) を指定します。以下はコードの断片です。

```
# 副次的なリンクオプション
CAT_LINK_FLAGS =

# どのように.dog プログラムをビルドするのかを定義した関数
CatProgram(program, files) =
    # 拡張子を追加
    file_names = $(addsuffix .woof, $(files))
    prog_name = $(addsuffix .dog, $(files))

    # ビルドルール
    $(prog_name): $(file_names)
        $(CATC) $(PREFIXED_INCLUDES) $(CAT_FLAGS) $(CAT_LINK_FLAGS) -o $@ $+

    # プログラム名を返す
    value $(prog_name)
```

CAT_LINK_FLAGS 変数はちょうど私たちがリンク作業において、追加フラグを渡したいような場合に定義される変数です。さて、新しく関数が定義されましたので、私たちがプログラムをビルドするためのルールを定義したいと思った場合はいつでも、単純にこの関数を呼ぶだけで完了します。前回のような暗黙のルールを記

述する場合ですと、どのように各々のソースファイルがコンパイルされるのかについていちいち指定する必要がありましたが、 CatProgram 関数はどのように実行ファイルをビルドするのか指定するだけで完了します。

ノート: 訳注: $@ , $+ はそれぞれ『ターゲットの名前』と『依存ファイルのリスト』を表しています。詳細は "**ビルドルール**" を参照してください。

```
# rover.dog プログラムをソースファイル neko.cat と chat.cat からビルドします。
# rover.dog は普通にコンパイルされます。
.DEFAULT: $(CatProgram rover, neko chat)
```

3.4.3 依存関係の解析

これでほとんどの作業が終わりましたが、まだ依存関係の解析を自動的に行わせる部分が残っています。これは OMake の利点の一つであり、さらにロバストで書きやすいビルド設定を作る手助けとなっています。厳密に言うと、この箇所は必要というわけではありませんが、あなたは切実にこの機能を欲しがっていると思います。

このメカニズムは通常のルールのように、 .SCANNER ルールを定義することで得られます。しかし .SCANNER ルールはどのように依存関係を解析するのかについて指定するのであって、ターゲット自身を指定しているわけではありません。私たちは以下のような形で .SCANNER ルールを定義したいものとします。

```
.SCANNER: %.woof: %.cat
    <commands>
```

このルールでは、「 .woof ファイルは収集した .cat ファイルから、 <commands> を実行することで展開できる」という、新しい依存関係を追加することを指定しています。 <commands> の実行結果は、通常の端末で出力できる、OMake 形式の依存関係の配列である必要があります。

すでに述べた通り、各々の .cat ファイルは open 構文を用いて、 .woof ファイルに依存していることを指定していたとします。例えば、もし neko.cat ファイルが行 open chat コマンドを含んでいたとするならば、 neko.woof ファイルは chat.woof ファイルに依存しています。この場合、 <commands> は以下の行を出力しなければなりません。

```
neko.woof: chat.woof
```

この類推は、 .o ファイルが .c ファイルをコンパイルすることで生成される C 言語について考えるとより明瞭になります。もしファイル foo.c が #include "fum.h" のような行を含んでいたとすると、 foo.c は fum.c が変更されたときはいつでも再コンパイルを行う必要があります。これはつまり、ファイル foo.o がファイル fum.h に依存していることを表しています。OMake の用語では、このことを『暗黙的な依存関係 (implicit dependency)』と呼んでおり、 .SCANNER <commands> は以下のような行を出力する必要があるでしょう。

```
foo.o: fum.h
```

それでは動物の世界へと戻ってみましょう。 neko.woof の依存関係を解析するために、私たちは一行一行 neko.cat ファイルをスキャンして、 open <name> のような形の構文を含んだ行を探す必要があります。私たちはこのようなプログラムを書かなければなりませんが、OMake はこのような作業を簡略化することができます。この例ですと、ソースファイルをスキャンする awk 関数がビルドインで用意されているので、これを使ってみましょう。一つ難しいことがあるとするならば、それは依存関係が INCLUDE パスに依存していることです。そのために OMake では探索するための find-in-path 関数を用意しています。それでは以下のように書いてみます。

```
.SCANNER: %.woof: %.cat
    section
        # ファイルをスキャン
        deps[] =
        awk($<)
        case $'^open'
            deps[] += $2
```

```
    export

# 重複を削除し、インクルードパスのファイルを探索する
deps = $(find-in-path $(INCLUDES), $(set $(deps)))

# 依存関係を出力
println($"$@: $(deps)")
```

それでは上のソースコードを見てみましょう。初めに、全体の文はシェルコマンドのシーケンスとして扱われずに内部で計算されるよう、 section 文の中で定義されています。

今回私たちはすべての依存関係を集めるために、 deps 変数を用いました。 awk 関数はソースファイル ($<) を一行一行スキャンしていきます。正規表現 ^open (これはこの行が単語 open で始まることを表しています) が見つかった場合、 deps 変数に二番目の単語を追加します。具体的には、入力された行が open chat であった場合、 deps 配列に chat 文字列を追加することになります。ソースファイル中のその他すべての行は無視されます。

次に、 $(set $(deps)) 文によって deps 配列の重複された文字列は削除されます (このとき、アルファベット順に配列をソートします)。 find-in-path 関数はインクルードパス中の各々のファイルの絶対パスを探索します。

最後に、文字列 $"$@: $(deps)" を結果として出力します。クオーテーションには deps 配列を単純な文字列に変換した状態で追加されます。

3.4.4 まとめ

例がすべて終わったので、この成果を一つのプロジェクトにまとめてみましょう。前回の例は以下のような構成とします。

```
my_project/
|--> OMakeroot
|--> OMakefile
`--> src/
     |---> OMakefile
     |---> lib/
     |     |---> OMakefile
     |     |---> neko.cat
     |     `---> chat.cat
     `---> main/
           |---> OMakefile
           `---> main.cat
```

この全体のプロジェクトのリストは以下のようになります。私たちはまたライブラリにいくつかの .woof ファイルをリンクさせるために、 CatLibrary 関数を定義していることに注意してください。

```
my_project/OMakeroot:
    # コマンドライン上の変数を処理
    DefineCommandVars()

    # このディレクトリの OMakefile をインクルード
    .SUBDIRS: .

my_project/OMakefile:
   ##########################################################################
   # .cat ファイルをコンパイルするための標準設定
   #

   # 私たちは今回 CATC 変数ををオーバーライドしたいので、catc コマンドを定義します
```

```
CATC = catc

# 通常のフラグは空にします
CAT_FLAGS =

# インクルードパスの辞書（通常は空です）
INCLUDES[] =

#  インクルードパスによるインクルードオプションを計算します
PREFIXED_INCLUDES[] = $`(mapprefix -I, $(INCLUDES))

# .cat ファイルの依存関係を解析するスキャナ
.SCANNER: %.woof: %.cat
     section
         # ファイルをスキャン
         deps[] =
         awk($<)
         case $'^open'
             deps[] += $2
             export

         # 重複を削除し、インクルードパスのファイルを探索する
         deps = $(find-in-path $(INCLUDES), $(set $(deps)))

         # 依存関係を出力
         println($"$@: $(deps)")

# 通常の方法で.cat ファイルをコンパイルする
%.woof: %.cat
    $(CATC) $(PREFIXED_INCLUDES) $(CAT_FLAGS) -c $<

# 副次的なリンクオプション
CAT_LINK_FLAGS =

# いくつかの.woof ファイルを用いてライブラリをビルド
CatLibrary(lib, files) =
    # 拡張子を追加
    file_names = $(addsuffix .woof, $(files))
    lib_name = $(addsuffix .woof, $(lib))

    # ビルドルール
    $(lib_name): $(file_names)
        $(CATC) $(PREFIXED_INCLUDES) $(CAT_FLAGS) $(CAT_LINK_FLAGS) -a $@ $+

    # プログラム名を返す
    value $(lib_name)

# どのように.dog プログラムをビルドするのかを定義した関数
CatProgram(program, files) =
    # 拡張子を追加
    file_names = $(addsuffix .woof, $(files))
    prog_name = $(addsuffix .dog, $(program))

    # ビルドルール
    $(prog_name): $(file_names)
        $(CATC) $(PREFIXED_INCLUDES) $(CAT_FLAGS) $(CAT_LINK_FLAGS) -o $@ $+

    # プログラム名を返す
```

```
        value $(prog_name)

    ########################################################################
    # これで正しくプログラムが動きます
    #

    # src サブディレクトリをインクルード
    .SUBDIRS: src

my_project/src/OMakefile:
    .SUBDIRS: lib main

my_project/src/lib/OMakefile:
    CatLibrary(cats, neko chat)

my_project/src/main/OMakefile:
    # ../lib ディレクトリからのインクルードを許可
    INCLUDES[] += ../lib

    # プログラムをビルド
    .DEFAULT: $(CatProgram main, main ../cats)
```

注意点としては、 OMakeroot では依存関係の解析や、ソースファイルをコンパイルするための通常のルール、ライブラリやプログラムをビルドするいくつかの関数を含んだ、標準的な設定を定義しています。

これらのルールや関数はサブディレクトリに継承されていますので、 .SCANNER とビルドルールは自動的に各々のサブディレクトリに使われます。よってあなたはこれらを繰り返し記述する必要はありません。

3.4.5 終わりに

これで一通りの作業は終わりましたが、まだ考慮すべき点はいくつか残っています。

まず、 cat プログラムをビルドするためのルールはプロジェクトの OMakefile に定義しました。もしあなたがどこか別の cat プロジェクトを持っていたとすると、 OMakeroot をコピー (そしてもし必要ならば修正も) するかもしれません。その代わりに、あなたは設定ファイルを Cat.om のように名称を変更して、ライブラリの共有ディレクトリに移すべきです。これで、コードをコピーする代わりに、OMake コマンド open Cat を用いてインクルードできるようになります。そのためには、あなたは共有ディレクトリを OMAKEPATH 環境変数に追加することで、omake がどこを探せば良いのか分かるようにすべきです。

もしあなたが満足する仕事をしたのなら、標準の設定となるようにあなたの設定ファイルを送ることを考えてみてください (omake@metaprl.org 宛にリクエストを送ることで)。他の人の作業を省力化することになります。

3.5 階層構造、.SUBDIRS の内容を並列化させる

いくつかのプロジェクトは同一の設定を有した、数多くのディレクトリで構成されているものです。例えば、あなたは現在サブディレクトリが多数あり、その各々がウェブページの画像の集合であるというプロジェクトを持っているものとしましょう。ある特定の画像を除いて、各々のファイルの設定は同一です。

この設定をより強固に構築するため、以下のような場合を考えます。まず、このプロジェクトは 4 つのサブディレクトリ page1, page2, page3, page4 を含んでいるものとします。また、各々のサブディレクトリは二つのファイル image1.jpg, image2.jpg を含んでおり、それらはプログラム genhtml によって生成されるウェブページの一部であるとします。

各々のディレクトリ中に OMakefile を定義する代わりに、OMake では .SUBDIRS コマンドの内容として定義することができます。

```
.SUBDIRS: page1 page2 page3 page4
    index.html: image1.jpg image2jpg
        genhtml $+ > $@
```

.SUBDIRS の内容は、まるで OMakefile が内部にあるかのように正確にふるまい、通常の命令を任意の数だけ実行することができます。 .SUBDIRS の内容は各々のサブディレクトリの内部で評価されます。実際に何が行われているのかについては、現在のディレクトリ名を出力する命令 ($(CWD)) を追加することでより分かりやすくなるでしょう。

```
  .SUBDIRS: page1 page2 page3 page4
      println($(absname $(CWD)))
      index.html: image1.jpg image2jpg
          genhtml $+ > $@
# 出力
  /home/jyh/.../page1
  /home/jyh/.../page2
  /home/jyh/.../page3
  /home/jyh/.../page4
```

3.5.1 glob パターンを扱う

もちろん、上述した指定は非常に強固なものとなっています。実際に、各々のサブディレクトリが異なった画像の集合であり、そのすべてがウェブページに含まれているような場合でも、記述方法は似ています。この問題に対するより簡単な解法の一つは、 glob や ls のようなディレクトリのリストを出力する関数を用いることです。 glob 関数はシェルのパターンを引数に持ち、現在のディレクトリ上でマッチしているファイル名の配列を返す関数です。

```
.SUBDIRS: page1 page2 page3 page4
    IMAGES = $(glob *.jpg)
    index.html: $(IMAGES)
        genhtml $+ > $@
```

3.5.2 簡略化されたサブディレクトリの設定

別の方法は、各々のサブディレクトリ固有の情報を定義した設定ファイルを、それぞれのディレクトリに追加することです。例えば、私たちは現在、各々のサブディレクトリ中に、ディレクトリ内部にある画像のリストを定義した BuildInfo.om ファイルを設置しているものとします。 .SUBDIRS の行は似ていますが、 BuildInfo ファイルをインクルードしている点が異なっています。

```
.SUBDIRS: page1 page2 page3 page4
    include BuildInfo   # IMAGES 変数を定義

    index.html: $(IMAGES)
        genhtml $+ > $@
```

それぞれの BuildInfo.om の内容は以下のようになっています。

```
page1/BuildInfo.om:
    IMAGES[] = image.jpg
page2/BuildInfo.om:
    IMAGES[] = ../common/header.jpg winlogo.jpg
page3/BuildInfo.om:
    IMAGES[] = ../common/header.jpg unixlogo.jpg daemon.jpg
```

```
page4/BuildInfo.om:
    IMAGES[] = fee.jpg fi.jpg foo.jpg fum.jpg
```

3.5.3 サブディレクトリのリストを計算

現在、サブディレクトリのリスト page1, ... , page4 は直接指定しています。他のディレクトリが追加される度に OMakefile を編集するよりも、(glob を用いて) 計算させたほうがはるかに合理的です。

```
.SUBDIRS: $(glob page*)
    index.html: $(glob *.jpg)
        genhtml $+ > $@
```

ディレクトリ構造が階層的である場合を考えてみましょう。その場合 glob 関数を用いる代わりに、階層的に各々のディレクトリを返す subdirs 関数を使います。例えば、以下は OMake プロジェクトのルート上で subdirs 関数を評価した結果です。最初の引数として渡した P オプションでは、OMake のディレクトリ自身を含んでいない、『適切な』リストを返すことを指定しています。

```
osh> subdirs(P, .)
- : <array
        /home/jyh/.../omake/mk : Dir
        /home/jyh/.../omake/RPM : Dir
        ...
        /home/jyh/.../omake/osx_resources : Dir>
```

subdirs を使用することで、上の例は以下のように表現できます。

```
.SUBDIRS: $(subdirs P, .)
    index.html: $(glob *.jpg)
        genhtml $+ > $@
```

この場合ですと、プロジェクト中の **すべての** サブディレクトリが含まれることとなります。

私たちが BuildInfo.om オプションを使用する場合、すべてのサブディレクトリをインクルードする代わりに、 BuildInfo.om ファイルが含んであるディレクトリのみインクルードしたいと思うでしょう。これを実現するために、私たちはディレクトリを階層的に全走査し、特定の表現にマッチしたファイルを返す find 関数を使用します。この場合ですと、 BuildInfo.om という名前のファイルを探したいことになります。以下は find 関数を呼び出したサンプルです。

```
osh> FILES = $(find . -name BuildInfo.om)
- : <array
        /home/jyh/.../omake/doc/html/BuildInfo.om : File
        /home/jyh/.../omake/src/BuildInfo.om : File
        /home/jyh/.../omake/tests/simple/BuildInfo.om : File>
osh> DIRS = $(dirof $(FILES))
- : <array
        /home/jyh/.../omake/doc/html : Dir
        /home/jyh/.../omake/src : Dir
        /home/jyh/.../omake/tests/simple : Dir>
```

この例では、プロジェクト中に 3 つの BuildInfo.om ファイルが doc/html, src, tests/simple ディレクトリに存在しています。また、 dirof 関数は各々のファイルのディレクトリを返します。

先の例に戻って、私たちは以下のように修正することにしました。

```
.SUBDIRS: $(dirof $(find . -name BuildInfo.om))
    include BuildInfo   # IMAGES 変数を定義
```

```
    index.html: $(IMAGES)
        genhtml $+ > $@
```

3.5.4 一時的なディレクトリ

時々、プロジェクトでは中間ファイルを置いておくための一時的なディレクトリが必要となる場合があります。これらの一時ディレクトリはプロジェクトがクリーンアップされたときはいつでも消去されます。これは特に、ディレクトリが消去されたら同ディレクトリの OMakefile も消去されるために、 OMakefile を一時的なディレクトリに置くべきではないことを意味しています。

もしあなたがこれらのディレクトリに関する設定を行いたいのなら、あなたは OMakefile を設置する代わりに、 .SUBDIRS の内容について記述する必要があります。

```
section
    CREATE_SUBDIRS = true

    .SUBDIRS: tmp
        # MD5 ハッシュを計算
        %.digest: %.comments
           echo $(digest $<) > $@

        # ソースファイルからコメントを展開
        %.comments: ../src/%.src
           grep '^#' $< > $@

        .DEFAULT: foo.digest

.PHONY: clean

clean:
    rm -rf tmp
```

今回の例では、私たちは tmp ディレクトリが存在しない場合に新しくディレクトリを生成するため、CREATE_SUBDIRS 変数を true に設定しました。 .SUBDIRS の内容は少々工夫してありますが、だいたいあなたが期待している通りに動作するはずです。 clean phony ターゲットでは、プロジェクトがクリーンアップされた場合は tmp ディレクトリが消去されるように指示しています。

CHAPTER 4

OMake言語の概要と構文

プロジェクトは OMakefile を用いて omake にどのようにビルドするのか指定しており、構文は Makefile と似ています。 OMakefile は 3 つの構文規則『変数の定義』『関数の定義』『ルールの定義』を持ち合わせています。

4.1 変数

変数は以下のような構文で定義されます。変数名は任意のアルファベットとアンダースコア _、ハイフン - を用いることができます。

```
<name> = <value>
```

値 (value) にはリテラル文字と展開された変数が定義できます。変数の展開は $(name) のような形で表されて、 <name> 変数は現在の環境下において <value> に置き換わります。いくつかの例を以下に示します。

```
CC = gcc
CFLAGS = -Wall -g
COMMAND = $(CC) $(CFLAGS) -O2
```

この例では、 COMMAND 変数の値は文字列 gcc -Wall -g -O2 となります。

make(1) とは違い、変数の展開は **先行して** *(eager)* 行われ、 **純粋な** *(pure)* メカニズムとなっています (詳細は "**スコーピング、セクション**","**動的なスコーピング**" を参照してください)。これはつまり、変数の値は即座に展開されることによって、新しい変数への束縛が古い値に影響されないことを意味しています。例えば、前回の例を以下のような変数の束縛に拡張した場合について考えてみましょう。

```
X = $(COMMAND)
COMMAND = $(COMMAND) -O3
Y = $(COMMAND)
```

この例では、変数 X の値は前回のように文字列 gcc -Wall -g -O2 が定義されて、変数 Y の値は gcc -Wall -g -O2 -O3 となります。

4.2 変数に値を追加

変数はまた、既存の変数に新しい文字列を追加する演算子 += を用いることができます。例えば、以下の 2 つの文は等価です。

```
# CLAGS 変数にオプションを追加
CFLAGS = $(CFLAGS) -Wall -g

# 上と下の束縛は等価です
CFLAGS += -Wall -g
```

4.3 配列

配列は変数名の後ろに [] を追加し、初期値を改行を用いて要素を指定することで定義できます。各々の行でのスペースは OMake において重要な役割を担っています。例えば、以下のコードは文字列 c d e が出力されます。

```
X[] =
    a b
    c d e
    f

println($(nth 2, $(X)))
```

4.4 特殊文字とクオート

文字 $ () : , =#\ は OMake の特殊文字に指定されています。これらの文字を通常の文字として OMake で扱うためには、バックスラッシュ文字 \ でエスケープする必要があります。

```
DOLLAR = \$
```

文字列を連結させるために、改行もまたエスケープする必要があります。

```
FILES = a.c\
        b.c\
        c.c
```

バックスラッシュは他の文字でエスケープする必要が **ない** ことに注意してください。よって以下のような例は正常に動作します (これはつまり、文字列中のバックスラッシュが正常に保たれていることを表しています)。

```
DOSTARGET = C:\WINDOWS\control.ini
```

ある文章をクオーティングしたい場合は "#..." エスケープを使用します。ダブルクオーテーションの数は任意で、最も外側のクオーテーションは文字列に含まれません。

```
A = $""String containing "quoted text" ""
B = $"""Multi-line
    text.
    The # character is not special"""
```

4.5 関数定義

関数は以下のような構文を用いて定義されます。

```
<name>(<params>) =
   <indented-body>
```

パラメータは識別のためにカンマを用いて分割し、コードは関数定義からインデントした状態で、別の行に設置する必要があります。例えば、以下のコードは引数 a と b をコロンを用いて結びつける関数について定義しています。

```
ColonFun(a, b) =
    return($(a):$(b))
```

return は関数から値を返す命令文です。 return 文は必須ではありません。この文が除外された場合、最後に関数が評価した命令文の返り値が返されます。

警告: バージョン 0.9.6 から return 文は関数を制御する命令文となりましたので、return 文が呼ばれた場合、関数はただちに値を返して終了します。以下の例では引数 a が true であったのなら、関数 f はただちに print 文を評価することなく値 1 を返します。

```
f(a) =
   if $(a)
      return 1
   println(The argument is false)
   return 0
```

多くの場合、あなたは関数から直接値を返さずに、セクションやネストされたコードブロックから値を返したいと思うことがあるでしょう。このような場合に、あなたは value 演算子を使用できます。実際、 value 演算子は関数だけに限らず、値が必要となった場合はどこでも使用することができます。以下の定義では、変数 X は a の値に依存して 1 か 2 が束縛されて、結果を出力し、関数から値を返します。

```
f_value(a) =
   X =
      if $(a)
         value 1
      else
         value 2
   println(The value of X is $(X))
   value $(X)
```

関数は GNU-make の構文 $(<name> <args>) を用いて呼び出します。 <args> はカンマで分割された値のリストです。例えば、以下のプログラムでは、変数 X は値 foo:bar を含みます。

```
X = $(ColonFun foo, bar)
```

関数の戻り値を必要としない場合には、通常の関数表記を用いて関数を呼び出すこともできます。例えば、以下のプログラムでは文字列“ She says: Hello world ”を出力します。

```
Printer(name) =
    println($(name) says: Hello world)

Printer(She)
```

4.6 コメント

コメントは # 文字から始まり、行の末尾まで続きます。

4.7 ファイルのインクルード

ファイルのインクルードには include か open 文を使います。インクルードされたファイルは OMakefile として、同じ構文で使用できます。

```
include $(Config_file)
```

open 文は include と似ていますが、一回しかインクルードされないのが特徴です。

```
open Config

# 2 回目の open は無視されますので、
# この行はなんの影響も与えません。
open Config
```

ファイル名が絶対パスで指定されていない場合、 include と open 文の両方は OMAKEPATH 変数上のパスを探します。 open 文の場合、この検索は **パースする際に実行される** ので、 open の引数に他の式を含める必要はありません。

4.8 スコーピング、セクション

omake のスコープはインデントのレベルで定義されます。インデントレベルが上がると、omake では新しいスコープが導入されます。

section 文は新しいスコープを追加したい場合に有効です。例えば、以下のコードは X = 2 を出力した後で、 X = 1 を出力します。

```
X = 1
section
    X = 2
    println(X = $(X))

println(X = $(X))
```

この結果について驚くかもしれませんが- section 内での変数の束縛は外部のスコープには影響を及ぼしていないのです。

"**環境のエクスポート**" で説明する export 文を使えば、内部スコープの変数をエクスポートすることでこの制限から抜け出すことができます。例えば、私たちが前回のサンプルに export 文を追加した場合、変数 X の新しい値が返されて、 X = 2 が 2 回出力されます。

```
X = 1
section
    X = 2
    println(X = $(X))
    export

println(X = $(X))
```

分離されたスコープが非常に重要な結果を及ぼす場合があります。例えば、各々の OMakefile はそれ自身のスコープで評価されます。つまり各々のプロジェクトの一部は独立した設定となっているので、一つの OMakefile で変数を定義しても、これは他の OMakefile の定義に影響を及ぼしません。

別の例を見てみましょう。異なったビルドターゲットを指定するために、変数を分割するほうが便利である場合を考えます。この場合の頻繁に使う慣用句として、分割されたスコープを定義する section 文を使用することが挙げられます。

```
section
   CFLAGS += -g
   %.c: %.y
       $(YACC) $<
   .SUBDIRS: foo

.SUBDIRS: bar baz
```

この例では、 foo サブディレクトリには CFLAGS 変数に -g オプションが追加されていますが、 bar と baz ディレクトリには追加されていません。この例の場合ですとスコープのルールは非常によく働いており、 foo サブディレクトリには新しい yacc ルールが追加されていますが、 bar と baz は追加されていません。さらにいうと、この追加されたルールは現在のディレクトリに影響を及ぼしていません。

4.9 条件分岐

トップレベルでの条件分岐は以下のような形となります。

```
if <test>
   <true-clause>
elseif <text>
   <elseif-clause>
else
   <else-clause>
```

まず <test> が評価されて、もしそれが *true* の値 (真偽値についての詳細は "**論理式、真偽関数、コマンドのコントロール**" を参照してください) であるならば <true-clause> のコードが評価されます。そうでなければ、残りの節が評価されます。また、 if 文は複数の elseif 宣言句を持たせることができます。 elseif と else 宣言句はなくても構いません。ただし、新しいスコープを導入するため、それぞれの宣言句はインデントされている必要があります。

if 文では、評価する文字列が空であったり、内容が false , no , nil , undefined , 0 であった場合、真偽値は *false* として評価されます。それ以外はすべて *true* になります。

以下の例では典型的な条件分岐の使い方を示しています。 OSTYPE 変数は現在使っているマシンのアーキテクチャを表しています。

```
# アーキテクチャ上での主要な拡張子
if $(equal $(OSTYPE), Win32)
   EXT_LIB = .lib
   EXT_OBJ = .obj
   EXT_ASM = .asm
   EXE = .exe
   export
elseif $(mem $(OSTYPE), Unix Cygwin)
   EXT_LIB = .a
   EXT_OBJ = .o
   EXT_ASM = .s
   EXE =
   export
else
   # 他のアーキテクチャの場合は強制終了する
   eprintln(OS type $(OSTYPE) is not recognized)
   exit(1)
```

4.10 マッチング

パターンマッチングは switch と match 文を使って実現できます。

```
switch <string>
case <pattern1>
    <clause1>
case <pattern2>
    <clause2>
...
default
   <default-clause>
```

case の数は任意です。 default 宣言句はなくても構いませんが、使う場合は一番最後の宣言句で用いるべきです。

switch の場合、文字列は <patterni> と『文字通りに』比較されます。

```
switch $(HOST)
case mymachine
    println(Building on mymachine)
default
    println(Building on some other machine)
```

<patternN> は定数である必要はありません。以下の関数は pattern1 のマッチ、そして ## デリミタを用いた pattern2 のマッチを表しています。

```
Switch2(s, pattern1, pattern2) =
   switch $(s)
   case $(pattern1)
       println(Pattern1)
   case $"##$(pattern2)##"
       println(Pattern2)
   default
       println(Neither pattern matched)
```

match の場合、パターンとして egrep(1)-正規表現-が使用できます。数値変数 $1, $2, ... は \(...\) 表現を使って値を取得できます。

```
match $(NODENAME)@$(SYSNAME)@$(RELEASE)
case $"mymachine.*@\(.*\)@\(.*\)"
    println(Compiling on mymachine; sysname $1 and release $2 are ignored)

case $".*@Linux@.*2\.4\.\(.*\)"
    println(Compiling on a Linux 2.4 system; subrelease is $1)

default
    eprintln(Machine configuration not implemented)
    exit(1)
```

4.11 オブジェクト

OMake はオブジェクト指向言語です。一般的に、オブジェクトはフィールド (訳注: プロパティもしくはメンバ変数と置き換えても良いです) とメソッドを持っています。オブジェクトは変数名の最後に . を加えることで定義できます。例えば、以下のオブジェクトは 2 次元平面上での点 (1, 5) を表しています。

```
Coord. =
    x = 1
    y = 5
    print(message) =
       println($"$(message): the point is ($(x), $(y)")

# X に 5 を束縛
X = $(Coord.x)

# これは "Hi: the point is (1, 5)" と出力されます。
Coord.print(Hi)
```

オブジェクトのフィールド x と y は点の座標を表しています。print メソッドは点の現在位置を出力します。

4.12 クラス

オブジェクトと同様にして **クラス** も定義できます。例えば、私たちは現在、オブジェクトを生成したり、移動したり、位置を出力するメソッドを持った Point クラスを作りたいものとしましょう。クラスはオブジェクトの作り方と似ていますが、 class 宣言句を用いて名前を定義付ける点が異なります。

```
Point. =
    class Point

    # フィールドの通常の値
    x = 0
    y = 0

    # 座標から新しいクラスを生成する
    new(x, y) =
       this.x = $(x)
       this.y = $(y)
       return $(this)

    # 点を右に移動する
    move-right() =
       x = $(add $(x), 1)
       return $(this)

    # 点を出力する
    print() =
       println($"The point is ($(x), $(y)")

p1 = $(Point.new 1, 5)
p2 = $(p1.move-right)

# "The point is (1, 5)" と出力
p1.print()

# "The point is (2, 5)" と出力
p2.print()
```

変数 $(this) は現在のオブジェクトを参照していることに注目してください。また、クラスとオブジェクトは新しいオブジェクトを返す new と move-right メソッドを持っています。これは、オブジェクト p2 とオブジェクト p1 が別物であり、 p1 はオリジナルの座標 (1, 5) を保持していることを表しています。

4.13 継承

クラスとオブジェクトは継承 (多重継承を含む) を extends 文によってサポートしています。以下の Point3D では、 x , y , z フィールドを持ったクラスを定義しています。新しいオブジェクトは、親クラスやオブジェクトが持つすべてのフィールドやメソッドを継承します。

```
Z. =
   z = 0

Point3D. =
   extends $(Point)
   extends $(Z)
   class Point3D

   print() =
      println($"The 3D point is ($(x), $(y), $(z))")

# "new"メソッドはオーバーライドされていませんので、
# 下のメソッドは新しく点 (1, 5, 0) を返します。
p = $(Point3D.new 1, 5)
```

4.14 static.

static. オブジェクトは OMake が動作している間、ずっと一定の値を保持していたい場合に使うオブジェクトです。このオブジェクトはプロジェクトを設定する際に頻繁に用いられます。プロジェクトを設定する変数が何回も書き換えられるのはリスクが高いので、 static. オブジェクトは設定がちょうど一回だけ行われることを保証してくれます。以下の (どこか冗長な) サンプルでは、 static. 節が LaTeX コマンドが使用可能であるかどうか調べるために使われています。 $(where latex) 関数は latex の絶対パスか、latex コマンドが存在しない場合は false を返します。

```
static. =
   LATEX_ENABLED = false
   print(--- Determining if LaTeX is installed )
   if $(where latex)
       LATEX_ENABLED = true
       export

   if $(LATEX_ENABLED)
      println($'(enabled)')
   else
      println($'(disabled)')
```

OMake の標準ライブラリを用いると第 14 章 (**自動設定用の変数と関数**) にあるような static. をプログラミングするための、多くの有用な関数を試すことができます。標準ライブラリを用いると、上のコードは以下のように書き直せます。

```
open configure/Configure
static. =
   LATEX_ENABLED = $(CheckProg latex)
```

プロジェクトの設定として使われている static. 節は、 ConfMsgChecking や ConfMsgResult 関数 (*ConfMsgChecking, ConfMsgResult*) を使って、 static. 節でどういう動作をしているのかについて出力すべきです (もちろん、標準ライブラリにある多くの関数が、この作業を自動的に行ってくれます)。

4.14.1 .STATIC

この機能はバージョン *0.9.8.5* で搭載されました。

.STATIC 節の書き方は static. 節の書き方と似ています。構文は以下の 3 つのどれを選んでも書くことができます。

```
# body で定義されたすべての変数をエクスポート
.STATIC:
    <body>

# 特にファイル依存を指定したい場合
.STATIC: <dependencies>
    <body>

# ファイル依存と同様に、どの変数をエクスポートしたいのか指定する場合
.STATIC: <vars>: <dependencies>
    <body>
```

<vars> は定義する変数名、 <dependencies> はファイル依存-依存先のファイルのある一つが変更された場合、対象のルールは再評価されます-を指定します。 <vars> と <dependencies> はもし必要ならば除外することができ、 <body> 中で定義されたすべての変数はエクスポートされます。

たとえば、前回のセクションで示した最後のサンプルは以下のように改良できます。

```
open configure/Configure
.STATIC:
    LATEX_ENABLED = $(CheckProg latex)
```

効果は (.STATIC を使用する代わりに) static. を使用した場合とほとんど似ています。しかしながら、殆どの場合において .STATIC のほうが優位です。理由は 2 つあります。

まず、 .STATIC 節は遅延評価されます。これはつまり、 .STATIC 内の変数が一つでも解決されないのならば、評価されることはないということを意味しています。例えばもし $(LATEX_ENABLED) が決して評価されない変数だとすると、 .STATIC 節は決して評価されることはありません。これは少なくとも一回はいつでも評価される static. 節とは対照的です。

次に、 .STATIC 節はファイル依存を指定できます。これは、 .STATIC 節がメモ化として用いられる場合に有効です。例えば、キーと値のペアを持ったテーブルから辞書を作りたい場合を考えてみましょう。 .STATIC 節を使うことによって、omake はこの計算を (omake が毎回動くときに計算するのではなく) 入力されたファイルが変更された場合のみ計算するようにふるまいます。以下の例では、 awk 関数がファイル table-file をパースするために用いられています。 awk 関数は key = value の形をした行を発見する度に、そのキーと値のペアを TABLE 変数に追加します。

```
.STATIC: table-file
    TABLE = $(Map)
    awk(table-file)
    case $'^\([[:alnum:]]+\) *= *\(.*\)'
        TABLE = $(TABLE.add $1, $2)
        export
```

ルールの依存関係が変わった場合はいつでも .STATIC 節は再計算されます。このルール内での対象は、エクスポートする変数となります (この場合ですと TABLE 変数が相当します)。

.MEMO

.MEMO ルールは、その結果が独立して動いている omake インスタンス間で保存されない点を除いて、 .STATIC ルールと等価です。

:key:

.STATIC と .MEMO ルールはまた、計算された値とリンクしている『キー』を表す :key: を使うことができます。 .STATIC ルールを、キーと値がリンクした辞書として考えることは有用です。 .STATIC ルールが評価された場合、結果は指定されたルールによって定義された :key: がテーブル内に保存されます (:key: が指定されていない場合、デフォルトのキーが代わりに用いられます)。言い換えると、ルールは関数のようなものです。 :key: は関数の『引数』を表しており、ルール部分で結果を計算します。

これを確かめるために、 .MEMO ルールをフィボナッチ関数に改良してみましょう。

```
fib(i) =
    i = $(int $i)
    .MEMO: :key: $i
        println($"Computing fib($i)...")
        result =
            if $(or $(eq $i, 0), $(eq $i, 1))
                value $i
            else
                add($(fib $(sub $i, 1)), $(fib $(sub $i, 2)))
    value $(result)

println($"fib(10) = $(fib 10)")
println($"fib(12) = $(fib 12)")
```

このスクリプトを実行した場合、以下のような結果となります。

```
Computing fib(10)...
Computing fib(9)...
Computing fib(8)...
Computing fib(7)...
Computing fib(6)...
Computing fib(5)...
Computing fib(4)...
Computing fib(3)...
Computing fib(2)...
Computing fib(1)...
Computing fib(0)...
fib(10) = 55
Computing fib(12)...
Computing fib(11)...
fib(12) = 144
```

フィボナッチ関数は各々の引数の場合において、一回だけしか計算されていないことに注目してください。これは普通にプログラムした場合ですと、指数関数的に計算時間が増えてしまいます。言い換えると、 .MEMO ルールは計算結果をメモ化 (memoization) しているからこそ、この名前なのです。 .STATIC ルールを代わりに使った場合、すべての omake インスタンスにおいて値が保存されていることに注意してください。

一般的には、あなたは .STATIC か .MEMO ルールを関数内で用いる場合はいつでも、ふつう :key: を使いたくなるでしょう。しかしながら、これは必須ではありません。以下の例では、 .STATIC ルールが、何か計算時間のかかる作業を一回だけ行う場合を表しています。

```
f(x) =
    .STATIC:
        y = $(expensive-computation)
    add($x, $y)
```

あなたがフィボナッチ関数のような再帰的な関数を定義する場合、さらに以下の点に注意すべきです。 :key: を除外してしまった場合、ルールは関数自体に対して定義されてしまい、循環された依存関係で評価されてしまいます。以下は :key: を除いたフィボナッチ関数の出力結果です。

```
Computing fib(10)...
Computing fib(8)...
Computing fib(6)...
Computing fib(4)...
Computing fib(2)...
Computing fib(0)...
fib(10) = 0
fib(12) = 0
```

この動作は `i = 0 || i = 1` の場合に達するまで `result` の値が保存されていないので、 `fib` は自身を `fib(0)` に達するまで再帰的に呼び出し、そして `result` の値は 0 に修正されてしまうために生じます。

再帰的な定義が無難に動作する場合もありますが、あなたは普通 `:key:` 引数をつけることで、各々の再帰的な呼び出しが異なった `:key:` を持つようにするでしょう。これは多くの場合において、 `:key:` が関数の引数すべてに含めるべきであることを示しています。

4.15 定数

OMake ではいろんな方法でそれぞれの値を表すことができます。私たちはこれを以下のリストにしました。

- **int** - 整数型
  - コンストラクタ: `$(int <i>)` (*int*)
  - オブジェクト: `Int` (*Int*)
  - 有限の値をもった整数型で、精度はプラットフォーム上の OCaml に依存します (32 ビットのプラットフォーム上では 31 ビット、64 ビットのプラットフォーム上では 63 ビット)(訳注: 1 ビットは正負の判定に使われます)。
  - 詳細は "**基本的な演算**" を参照してください。
- **float** - 浮動小数点型
  - コンストラクタ: `$(float <x>)` (*float*)
  - オブジェクト: `Float` (*Float*)
  - 浮動小数点型で、精度は 64 ビットです。
- **array** - 配列
  - コンストラクタ: `$(array <v1>, ..., <vn>)` (*array*)
  - オブジェクト: `Array` (*Array*)
  - 配列は有限の数の値をもったリストを表します。配列はまた以下のように定義することもできます。

    ```
    X[] =
        <v1>
        ...
        <vn>
    ```

  - 詳細は "*nth*", "*nth-tl*", "*length*" を参照してください。
- **string** - 文字列
  - オブジェクト: `String` (*String*)
  - 通常、すべての文字からなるシーケンスは配列として表現されるため、単純にソース中に書き表すことで初期化できます。内部で文字列はいくつかの断片としてパースされます。文字列はしばしば、

ホワイトスペース (訳注: ホワイトスペースはスペース、タブを含んだ空白文字のことです) によって分割された値のシーケンスとして定義されます。

```
osh>S = This is a string
- : <sequence
   "This" : Sequence
   ' ' : White
   "is" : Sequence
   ' ' : White
   "a" : Sequence
   ' ' : White
   "string" : Sequence>
   : Sequence
osh>length($S)
- : 4 : Int
```

- **データ** 文字列は、ホワイトスペースが重要な意味を持つ場合に用いられます。これは単純な一つの値として定義されるので、配列にはなりません。コンストラクタはクオーテーション `$"..."` と `$'...'` で表現できます。

```
osh>S = $'''This is a string'''
- : <data "This is a string"> : String
```

- 詳細は "**クオート文字列**" を参照してください。

- **file** - ファイル
  - コンストラクタ: `$(file <names>)` (*file, dir*)
  - オブジェクト: `File` (*File*)
  - ファイルオブジェクトはファイルの絶対パスを表すオブジェクトです。ファイルオブジェクトは絶対パスとして見ることができます。文字列への変換はカレントディレクトリに依存しています。

```
osh>name = $(file foo)
- : /Users/jyh/projects/omake/0.9.8.x/foo : File
osh>echo $(name)
foo
osh>cd ..
- : /Users/jyh/projects/omake : Dir
osh>echo $(name)
0.9.8.x/foo
```

  - 詳細は "*file, dir*" を参照してください。

  ノート: 訳注: 原文では "*vmount*" となっていますが、これはおそらく "*file, dir*" の間違いであると思われますので、置き換えました。

- **directory** - ディレクトリ
  - コンストラクタ: `$(dir <names>)`
  - オブジェクト: `Dir` (*Dir*)
  - ディレクトリオブジェクトはファイルオブジェクトと似ていますが、ディレクトリとしてふるまいます。

- **map (dictionary)** - マップ (辞書)
  - オブジェクト: `Map` (*Map*)
  - マップ/辞書オブジェクトは値と値を結びつけるテーブルです。`Map` オブジェクトは空のマップです。データ構造は永続的に保持されて、すべての演算は分かりやすく関数的です。特別な構文 `$|key|` によって文字列のキーを表現することができます。

```
osh>table = $(Map)
osh>table = $(table.add x, int)
osh>table. +=
        $|y| = int
osh>table.find(y)
- : "int" : Sequence
```

- **channel** - チャネル
  - コンストラクタ: `$(fopen <filename>, <mode>)` (*fopen*)
  - オブジェクト: `InChannel` (*InChannel*), `OutChannnel` (*OutChannel*)
  - チャネルオブジェクトは入力や出力のバッファとして使います。
- **function** - 関数
  - コンストラクタ: `$(fun <params>, <body>)` (*fun*)
  - オブジェクト: `Fun` (*Fun*)
  - 関数オブジェクトはいろんな方法で定義できます。
    - 無名関数

      ```
      $(fun i, j, $(add $i, $j))
      ```

    - 名前をつけた関数

      ```
      f(i, j) =
          add($i, $j)
      ```

    - **この機能はバージョン *0.9.9.0* で導入されました。** 無名関数の引数

      ```
      osh>foreach(i => $(add $i, 1), 1 2 3)
      - : <array 2 3 4> : Array
      ```

- **lexer** - 字句解析
  - オブジェクト: `Lexer` (*Lexer*)
  - このオブジェクトは字句解析器 (レキサ) として表現します。
- **parser** - パーサ
  - オブジェクト: `Parser` (*Parser*)
  - このオブジェクトはパーサとして表現します。

CHAPTER 5

変数と名前空間

コードを評価する際、OMake の変数には 3 つの異なる種類の名前空間があります。変数は **プライベート** なものにしたり、 **現在の** オブジェクトのフィールドを参照したり、 **グローバル** な名前空間の一部として用いることができます。名前空間を指定するには、変数名の前に修飾子を直接明示する必要があります。この 3 つの名前空間は分割されており、変数は一つ、あるいはさらに多くの名前空間に束縛することができます。

```
# プライベートな名前空間
private.X = 1
# 現在のオブジェクト
this.X = 2
# パブリック、グローバルに定義された名前空間
global.X = 3
```

5.1 private.

`private.` 修飾子は変数が現在のファイルやスコープ上でプライベートなものであると定義したい場合に用います。値は外部のスコープから参照することができません。プライベートな変数は静的にスコープされています。

```
Obj. =
   private.X = 1

   print() =
      println(The value of X is: $X)

# 出力:
#    The private value of X is: 1
Obj.print()

# X は Obj 内のプライベート変数なので、エラーとなります
y = $(Obj.X)
```

加えて、プライベート変数はグローバルな変数の値に影響を及ぼしません。

```
# パブリックな x の値は 1
x = 1

# このオブジェクトは x のプライベートな値を使用しています
```

```
Obj. =
    private.x = 2

    print() =
       x = 3
       println(The private value of x is: $x)
       println(The public value of x is: $(public.x))
       f()

# 出力:
#    The private value of x is: 3
#    The public value of x is: 1
Obj.print()
```

プライベート変数は 2 つの性質を持っています。

1. プライベート変数は定義されたファイルしか参照できません。
2. プライベート変数はたとえ明示的に `export` 文を使用したとしてもエクスポートできません。

```
private. =
   FLAG = true

section
   FLAG = false
   export

# FLAG はまだ true です
section
   FLAG = false
   export FLAG

# FLAG は現在 false です
```

5.2 this.

`this.` 修飾子はオブジェクトのローカルなフィールドについて定義したい場合に用います。オブジェクト変数は動的にスコープされます。

```
X = 1
f() =
   println(The public value of X is: $(X))

# 出力:
#    The public value of X is: 2
section
   X = 2
   f()

# X はオブジェクト中の保護されたフィールドを表しています。
Obj. =
   this.X = 3

   print() =
      println(The value of this.X is: $(X))
      f()
```

```
# 出力:
#    The value of this.X is: 3
#    The public value of X is: 1
Obj.print()

# この文は正しく、Y には 3 が束縛されます。
Y = $(Obj.X)
```

一般的に、保護されたオブジェクト変数を定義することは良いこととされています。プロジェクトの他の部分で定義された変数と被ってしまったために起こる、意図しないバグを生み出さないためです。結果として、コードはよりモジュール化されます。

5.3 global.

`global.` 修飾子はグローバルに動的なスコープを持つ変数を定義したい場合に用います。以下の例では、`global.` 宣言は束縛文 `X = 4` が動的にスコープされた変数であることを指定しています。グローバル変数はオブジェクトのフィールドとして定義することが **できません** 。

```
X = 1
f() =
   println(The global value of X is: $(X))

# 出力:
#    The global value of X is: 2
section
   X = 2
   f()

Obj. =
   protected.X = 3

   print() =
      println(The protected value of X is: $(X))
      global.X = 4
      f()

# 出力:
#    The protected value of X is: 3
#    The global value of X is: 4
Obj.print()
```

5.4 protected.

OMake 0.9.8 では、 `protected` は `this` 修飾子と同義語でした。

```
osh>protected.x = 1
- : "1" : Sequence
osh>value $(this.x)
- : "1" : Sequence
```

0.9.9 ではこの仕様は変更されて、 `protected` 修飾子は変数が現在のオブジェクトまたはファイルについてのローカル変数とし、外部からアクセスできないようにしたい場合に用います。

5.5 public.

OMake 0.9.8 では、 public は global 修飾子と同義語でした。

```
osh>public.x = 1
- : "1" : Sequence
osh>value $(global.x)
- : "1" : Sequence
```

0.9.9 ではこの仕様は変更されて、 public 修飾子は変数が現在のファイルまたはオブジェクトの外部からアクセスできるようにしたい場合に用います。

5.6 修飾されたブロック

いくつかの修飾された変数が同時に定義された場合、修飾子のブロックが優先的に定義されます。構文はオブジェクトの定義と似ていますが、それはオブジェクト名それ自体が修飾子だからです。例えば、以下のプログラムはプライベート変数 X と Y を定義しています。

```
private. =
    X = 1
    Y = 2
```

修飾子はブロック内で新しく定義された変数の、デフォルトの名前空間を指定しています。その違いを除いて、ブロックの内容は完全に通常のコードとしてふるまいます。

```
private. =
    X = 1
    Y = 2
    public.Z = $(add $X, $Y)
    # "The value of Z is 3" と出力される
    echo The value of Z is $Z
```

5.7 変数宣言

変数名が修飾されていない場合、その名前空間は最も近くで定義された名前空間か、この変数が定義されているスコープの名前空間が用いられます。私たちは以前すでにこの現象を例を通して見ています。その例では、変数の定義が修飾されていても、その後に来る変数は明示的に修飾されていなかったはずです。以下の例では、最初に宣言された $X は **プライベート** な変数 $(private.X) が関連付けられています。なぜならこれは最も近くで定義されているからです。パブリックな変数 X は未だに 0 であり、この変数を指定するためには明示的に修飾しなければなりません。

```
public.X = 0
private.X = 1

public.print() =
    println(The value of private.X is: $X)
    println(The value of public.X is: $(public.X))
```

時々、修飾子を定義する事なしに変数宣言することが有効である場合があります。例えば、私たちはプログラムの後ろで定義された変数 X を用いる関数を持っていたとしましょう。 declare 文はこのような場合に使うことができます。

```
declare public.X

public.print() =
    println(The value of X is $X)

# "The value of X is 2" と出力される
X = 2
print()
```

最後に、明示的に修飾されていない変数についてはどうなるのでしょうか？このような場合、以下のルールが用いられます。

- もし変数が関数の引数ならば、その変数はプライベートとなります。
- もし変数がオブジェクト中で定義されているならば、 `this.` 修飾子で定義されます。
- それ以外はすべてパブリックとなります。

CHAPTER 6

式と値

omake は多くのシステムと IO 関数を含んでいる、フル機能のプログラミング言語です。この言語はオブジェクト指向言語であり、数値や文字列のような基本の型もすべてオブジェクトで表現されます。しかしながら、この omake 言語は 3 つの点において、他のスクリプト言語とは異なっています。

- 動的なスコーピングを行います。
- IO を除いて、この言語は全体が関数的です。この言語は代入という操作が存在しません。
- 値の評価は通常の場合その場で行われます。これは、式が表れた瞬間に評価されるということを意味しています。

これらの機能を確かめるため、今回私たちは `osh(1)` omake プログラムを使いました。 `osh(1)` プログラムは式を入力したら、すぐに結果が出力されるインタープリターとなっています。 `osh(1)` は通常シェル上で実行するために、入力された文章をコマンド文として解釈しますので、式を直接評価するためには多くの場合 `value` 文を使用します。

```
osh> 1
*** omake error: File -: line 1, characters 0-1 command not found: 1
osh> value 1
- : "1" : Sequence
osh> ls -l omake
-rwxrwxr-x  1 jyh jyh 1662189 Aug 25 10:24 omake*
```

6.1 動的なスコーピング

動的なスコーピングは言い換えると、変数の値が実行されているスコープ中で、もっとも近くで束縛されている変数によって決定されることを意味しています。以下のプログラムについて考えてみましょう。

```
OPTIONS = a b c
f() =
   println(OPTIONS = $(OPTIONS))
g() =
   OPTIONS = d e f
   f()
```

`f()` が `OPTIONS` 変数を再定義することなく呼び出した場合、この関数は文字列 `OPTIONS = a b c` が出力されます。

対照的に、関数 g() は OPTIONS 変数を再定義し、 f() をそのスコープ中で評価しますので、 この関数は文字列 OPTIONS = d e f が出力されます。

g の内容はローカルスコープを定義しています。 OPTIONS 変数の再定義は g についてローカルであり、この関数が終了した場合、この定義も終了します。

```
osh> g()
OPTIONS = d e f
osh> f()
OPTIONS = a b c
```

動的なスコーピングはプロジェクトでのコードを簡略化するための非常に有用なツールです。例えば、OMakeroot ファイルでは関数の集合、 CC や CFLAGS などの変数を使ったプロジェクトのビルドルールについて定義しています。しかしながら、プロジェクト中の異なったパートでは、これらの変数がそれぞれ異なった値であってほしいと思うことがあるでしょう。例えば、サブディレクトリ opt では、私たちは -O3 オプションを、サブディレクトリ debug では -g オプションを用いてビルドしたいものとします。この問題は動的なスコーピングを用いて、関数を再定義することなく、プロジェクト中の一部の変数を置き換えることができます。

```
section
   CFLAGS = -O3
   .SUBDIRS: opt
section
   CFLAGS = -g
   .SUBDIRS: debug
```

しかしながら、動的なスコーピングは欠点も持っています。はじめに、この機能はバグの箇所が分かり辛くなることがあります。具体的にいうと、あなたはプライベートにしたい変数があるとします。しかしこれはどこか別の場所で再定義される恐れがあります。例えば、あなたは以下の検索パスを組み立てるコードを持っていたとします。

```
PATHSEP = :
make-path(dirs) =
   return $(concat $(PATHSEP), $(dirs))

make-path(/bin /usr/bin /usr/X11R6/bin)
- : "/bin:/usr/bin:/usr/X11R6/bin" : String
```

しかしながら、プロジェクトのどこかで PATHSEP 変数がディレクトリのセパレータ / で再定義された場合、この関数は突如文字列 /bin//usr/bin//usr/X11R6/bin を返すようになります。あなたは明らかにそれを望んでいないのにです。

private ブロックはこの問題を解決するために用いられます。 private ブロック内で定義された変数は静的なスコーピングを用います。これは、変数の値がソーステキストのスコープ中で、もっとも最近の定義によって決定されることを示しています。

```
private
   PATHSEP = :
make-path(dirs) =
   return $(concat $(PATHSEP), $(dirs))

PATHSEP = /
make-path(/bin /usr/bin /usr/X11R6/bin)
- : "/bin:/usr/bin:/usr/X11R6/bin" : String
```

6.2 関数評価

IO を除いて、omake のプログラムは全体が関数的です。これは二つの意味を持っています。

- 代入という操作が存在しません。
- 関数は引数を渡して、別の値を返す『値 (value)』です。

二番目についてはそのままの説明です。例えば、以下のプログラムでは関数の値を返すことによって加算する関数を定義しています。

```
incby(n) =
   g(i) =
      return $(add $(i), $(n))
   return $(g)

f = $(incby 5)

value $(f 3)
- : 8 : Int
```

一番目については恐らく最初は困惑することでしょう。代入なしで、いったいどのようにして、サブプロジェクトにプロジェクトのグローバルな振る舞いを修正することができるのでしょうか？実際、この省略された説明は意図的にされています。サブプロジェクトが他のプロジェクトの邪魔をしないことを保障されているとき、ビルドスクリプトはより書きやすくなります。

しかしながら、サブプロジェクトが親のオブジェクトに情報を伝える必要がある場合や、内部のスコープが外部のスコープに情報を伝える必要がある場合が存在することも確かです。

6.3 環境のエクスポート

`export` 文によってすべて、あるいは一部の内部スコープの情報を親スコープに伝えることができます。もし引数が存在しない場合、全体のスコープの情報が親に伝えられます。さもなければ引数の変数のみが伝えられます。もっともよく使うやり方は、条件分岐中のいくつか、あるいはすべての定義をエクスポートする場合です。以下の例では、変数 `B` は評価された後に、2 に束縛されます。変数 `A` は再定義されません。

```
if $(test)
   A = 1
   B = $(add $(A), 1)
   export B
else
   B = 2
   export
```

`export` 文が引数なしに用いられた場合は、以下のすべてが出力されます。

- 動的にスコープされたすべての値 (“*public.*“で説明しました)
- 現在のワーキングディレクトリ
- 現在の UNIX 環境
- 現在の暗黙のルールと暗黙の依存関係 (詳細は “**暗黙のルールのスコーピング**” を参照してください)
- 現在の”phony” ターゲット宣言の集合 (詳細は “*.PHONY*”,”*.PHONY* **ターゲットのスコーピング**” を参照してください)

`export` 文が引数ありで用いられた場合は引数の式が評価されて、返される値は以下のようになります。

- もし値が空であるなら、上で説明したすべてがエクスポートされます。
- もし値が環境 (environment)、あるいは部分的な環境を表現しているのなら (詳細は “*export*” を参照してください)、対象の環境または部分的な環境がエクスポートされます。

- そうでなければ、値は出力したい項目を指定している、文字列のシーケンスでなければなりません。また、以下の文字列は特別な意味を持ちます。
  - .RULE ー 暗黙のルールと、暗黙的な依存関係
  - .PHONY ー "phony" ターゲット宣言の集合

  すべての他の文字列は、出力する必要のある変数名として解釈されます。

例えば以下の (わざとらしい) 例では変数 A と B がエクスポートされて、さらに暗黙のルールはこのセクションが終わった後も、環境の中で保持されます。しかし、変数 TMP とターゲット tmp_phony は変更されずに、このセクションの中にとどまります。

```
section
   A = 1
   B = 2
   TMP = $(add $(A), $(B))

   .PHONY: tmp_phony

   tmp_phony:
      prepare_foo

   %.foo: %.bar tmp_phony
      compute_foo $(TMP) $< $@
   export A B .RULE
```

6.3.1 区域のエクスポート

この機能はバージョン *0.9.8.5* で導入されました。

export 文はブロックの最後で実行する必要はありません。エクスポートはブロック中のブロックも、ブロックの終わりでエクスポートされます。言い換えると、 export はその文の次にくるプログラムでも用いられます。これはコード量を減らすという点で特に有用です。以下の例では、変数 CFLAGS は両方の条件分岐文からエクスポートされます。

```
export CFLAGS
if $(equal $(OSTYPE), Win32)
    CFLAGS += /DWIN32
else
    CFLAGS += -UWIN32
```

6.3.2 エクスポートされた区域から値を返す

この機能はバージョン *0.9.8.5* で導入されました。

ブロックによって返された値は export を使用してもエクスポートされません。この値は普通に計算された場合、ブロック最後の状態の値として解釈されるので、エクスポートを無視します。例えば、私たちはマップの文字列をユニークな整数値に改良したテーブルを作りたいと思い、以下のプログラムを考えたとします。

```
# 空のマップ
table = $(Map)

# テーブルにエントリーを追加
intern(s) =
    export
    if $(table.mem $s)
        table.find($s)
```

```
    else
        private.i = $(table.length)
        table = $(table.add $s, $i)
        value $i

intern(foo)
intern(boo)
intern(moo)
# "boo = 1" と出力
println($"boo = $(intern boo)")
```

文字列 s が与えられると、関数 intern は s に既に関連付けられている値を返し、そうでない場合は新しい値を関連付けます。この場合、このテーブルは新しい値にアップデートされます。関数の初めに export をつけることによって、 table 変数はエクスポートされます。一方、 s や i に束縛されている値はプライベートなので、エクスポートされません。

omake での評価は先行して行われます。これは評価文に遭遇した場合、即座に式の評価が行われることを意味しています。この効果の一つとして、変数が定義されたときに、右側の変数定義が展開されることが挙げられます。

```
osh> A = 1
- : "1"
osh> A = $(A)$(A)
- : "11"
```

二番目の定義文の右側 A = (A)(A) がまず初めに評価されて、シーケンス 11 が生成されました。変数 A は新しい値として再定義されます。動的なスコーピングを用いて束縛した場合、これは従来の命令型プログラミングと同じ多くの特性を持ちます。

```
osh> A = 1
- : "1"
osh> printA() =
    println($"A = $A")
osh> A = $(A)$(A)
- : "11"
osh> printA()
11
```

この例では、出力関数は A のスコープ中で定義されます。最後の行でこの関数が呼び出されたとき、 A の動的な値は 11 であるので、この値が出力されます。

しかしながら、動的なスコーピングと命令型のプログラミングは混同すべきではありません。以下の例では違いについて示しています。二番目の printA は A = x(A)(A)x が定義されているスコープには存在していませんので、この関数は元の値 1 を出力します。

```
osh> A = 1
- : "1"
osh> printA() =
    println($"A = $A")
osh> section
         A = x$(A)$(A)x
         printA()
x11x
osh> printA()
1
```

遅延評価式の使用で評価順序を制御する詳細については、”**遅延評価式**“を参照してください。

6.4 オブジェクト

omake はオブジェクト指向型言語です。数や文字列のような基本的な値を含むすべてがオブジェクトで表現されます。多くのプロジェクトの場合、通常のトップレベルのオブジェクト中でほとんどの式が評価されるため、これを見ることはあまりありませんが、 Pervasives オブジェクトと、2,3 個の他のオブジェクトが最初から定義されています。

しかしながら、オブジェクトはデータを構築するための追加手段を提供し、さらにオブジェクトを慎重に使用することで、あなたのプロジェクトをより簡単にしてくれるでしょう。

オブジェクトは以下の構文で定義されます。これは name をいくつかのメソッドと値を持ったオブジェクトとして定義しています。

```
name. =                      # += も同じくらいよく使います
   extends parent-object     # なくても構いません
   class class-name          # なくても構いません

   # フィールド
   X = value
   Y = value

   # メソッド
   f(args) =
      body
   g(arg) =
      body
```

extends 文はこのオブジェクトが指定された parent-object に継承されていることを指定します。オブジェクトは任意の数の extends 文を含めることができます。もし多くの extends 文が存在した場合、すべての親オブジェクトのメソッドとフィールドは継承されます。もし名前が衝突していた場合、前の定義は後の定義でオーバーライドされます。

class 文はなくても構いません。指定されていた場合、 instanceof 演算子を使うことでオブジェクト名を新たに定義することができるようになります。これは下で議論する :: スコープ文と同様です。

オブジェクトには任意の内容のプログラムを記述できます。オブジェクトの中に定義された変数はフィールドと定義され、関数はメソッドと定義されます。

6.5 フィールドとメソッドの呼び出し

オブジェクトのフィールドとメソッドは object.name 表記を用いて命名されます。例えば、一次元の点の値について定義してみましょう。

```
Point. =
   class Point

   # 通常の値
   x = $(int 0)

   # 新しい点を生成
   new(x) =
      x = $(int $(x))
      return $(this)

   # ひとつ進める
   move() =
```

```
      x = $(add $(x), 1)
      return $(this)

osh> p1 = $(Point.new 15)
osh> value $(p1.x)
- : 15 : Int

osh> p2 = $(p1.move)
osh> value $(p2.x)
- : 16 : Int
```

`$(this)` は常に現在のオブジェクトに置き換える変数です。式 `$(p1.x)` はオブジェクト `p1` の `x` の値を呼び出します。式 `$(Point.new 1)` は `Point` オブジェクトの `new` メソッドを呼び出す式で、初期値として 15 が保持された新しいオブジェクトを返します。 `$(p1.move)` もメソッドの呼び出しで、16 が保持された新しいオブジェクトを返します。

オブジェクトは関数的であり、その場で存在しているオブジェクトのフィールドやメソッドを修正することは不可能であることに注意してください。よって、 `new` と `move` メソッドは新しいオブジェクトを返します。

6.6 メソッドのオーバーライド

1 つ移動させる代わりに、2 つ移動させるメソッドを持った新しいオブジェクトを作ることについて考えてみましょう。 `move` メソッドをオーバーライドすることでこれを実現することができます。

```
Point2. =
   extends $(Point)

   # move メソッドをオーバーライド
   move() =
      x = $(add $(x), 2)
      return $(this)

osh> p2 = $(Point2.new 15)
osh> p3 = $(p2.move)
osh> value $(p3.x)
- : 17 : Int
```

しかし、これを行うと古い `move` メソッドは完全に置き換わります。

6.7 親の呼び出し

新しい `move` メソッドを、古い `old` メソッドを二回呼び出すことで定義したい場合について考えてみましょう。これは表記 `$(classname::name <args>)` を用いることで親を呼び出すことができます。 `classname` は親クラスの名前で、 `name` のメソッドやフィールドが関連付けられている必要があります。それでは、 `Point2` オブジェクトを別の方法で定義してみましょう。

```
Point2. =
   extends $(Point)

   # 古いメソッドを 2 回呼び出す
   move() =
      this = $(Point::move)
      return $(Point::move)
```

最初の `$(Point::move)` の呼び出しは現在のオブジェクト (`this` 変数) を再定義していることに注意してください。なぜならこのメソッドは新しいオブジェクトを返し、二回目の呼び出しで再利用されるからです。

CHAPTER 7

さらなる言語例

この章では、一連の例を通して言語の核心へと迫ります (ビルドシステムのサンプルについては *OMake* **ビルドサンプル** を参照してください)。

私たちはこれらの例のほとんどに osh インタープリターを用いています。また、簡単にするため、 osh によって出力された値は省略されています。

7.1 文字列と配列

OMake の基本となる型は文字列とシーケンス、そして値からなる配列です。シーケンスはホワイトスペースによって分割された配列のようなもので、関数の要求に応じて分割されます。

```
osh> X = 1 2
- : "1 2" : Sequence
osh> addsuffix(.c, $X)
- : <array 1.c 2.c> : Array
```

時々あなたは明示的に配列を定義したいと思うでしょう。この場合、 [] を変数名の後に追加し、配列の成分をそれぞれインデントされた行に書くことで実現できます。

```
osh> A[] =
        Hello world
        $(getenv HOME)
- : <array "Hello world" "/home/jyh"> : Array
```

配列は成分にホワイトスペースを含めることができます。これは配列の主要な特徴の一つであり、重要な役割を担います。また、これはホワイトスペースを含むファイル名にとって特に役立ちます。

```
# ディレクトリ中の現在のファイルを並べる
 osh> ls -Q
 "fee"  "fi"  "foo"  "fum"
 osh> NAME[] =
         Hello world
 - : <array "Hello world"> : Array
 osh> touch $(NAME)
 osh> ls -Q
 "fee"  "fi"  "foo"  "fum"  "Hello world"
```

7.2 クオート文字列

String は単一の値です。文字列の中でホワイトスペースは重要な意味を持っています。文字列をクオートするには 4 通りの方法があります。まず一番目にクオートをつけることが挙げられるでしょう。シンボル ' (シングルクオート) と " (ダブルクオート) を用いることで、シェルを扱うときのように成分をクオートすることができます。クオーテーションシンボルは結果の文字列に **含まれます** 。変数は常に内部にクオートを含んだ状態で展開されます。 osh(1) (15 章で説明) は文字列にダブルクオートをつけて出力しますが、これは出力時だけであり、文字列の中には含まれていないことに注意してください。

```
osh> A = 'Hello "world"'
- : "'Hello \"world\"'" : String
osh> B = "$(A)"
- : "\"'Hello \"world\"'\"" : String
osh> C = 'Hello \'world\''
- : "'Hello 'world''" : String
```

二番目の方法は $' と $" クオートを導入することです。始まりと終わりのクオートシンボルの数は任意です。また、これらのクオーテーションはいくつかの性質をもっています。

- クオートデリミタは文字列の一部では **ありません** 。
- 文字列中のバックスラッシュ \ 文字は通常の文字として扱われます。
- 文字列は何行にわたって書くこともできます。
- 変数は $" を用いて展開することができます。ただし、 $' では展開できません。

```
osh> A = $'''Here $(IS) an '''' \(example\) string['''
- : "Here $(IS) an '''' \\(example\\) string[" : String
osh> B = $""""A is "$(A)" """"
- : "A is \"Here $(IS) an '''' \\(example\\) string[\" " : String
osh> value $(A.length)
- : 38 : Int
osh> value $(A.nth 5)
- : "$" : String
osh> value $(A.rev)
- : "[gnirts )\\elpmaxe(\\ '''' na )SI($ ereH" : String
```

文字列とシーケンスの両方はホワイトスペースが含まれていない文字列とくっつけることができます。

```
osh> A = a b c
- : "a b c" : Sequence
osh> B = $(A).c
- : <sequence "a b c" : Sequence ".c" : Sequence> : Sequence
osh> value $(nth 2, $(B))
- : "c.c" : String
osh> value $(length $(B))
- : 3 : Int
```

配列は異なります。配列の成分はどのような方法を用いても文字列とくっつけることができません。配列は括弧 [] を変数名につけ、インデントした内容を記述することで成分を定義できます。各成分にはホワイトスペースを含めることもできます。

```
osh> A[] =
        a b
        foo bar
- : <array
       "a b" : Sequence
       "foo bar" : Sequence>
       : Array
```

```
osh> echo $(A).c
a b foo bar .c
osh> value $(A.length)
- : 2 : Int
osh> value $(A.nth 1)
- : "foo bar" : Sequence
```

配列はしばしばシステム上にホワイトスペースを含んだファイル名を使う場合において、非常に有用なツールとなります。

```
osh> FILES[] =
         c:\Documents and Settings\jyh\one file
         c:\Program Files\omake\second file

osh> CFILES = $(addsuffix .c, $(FILES))
osh> echo $(CFILES)
c:\Documents and Settings\jyh\one file.c c:\Program Files\omake\second file.c
```

7.3 ファイルとディレクトリ

複数のディレクトリにまたがっており、かつ異なったパートに分かれている OMake のプロジェクト上では、まったく違うディレクトリ上でコマンドが実行されます。これはファイル、あるいはディレクトリの名前が位置的に独立して定義されている必要があることを表しています。

この問題は `$(file <names>)` や `$(dir <names>)` 関数を用いて解決できます。

```
osh> mkdir tmp
osh> F = $(file fee)
osh> section:
         cd tmp
         echo $F
../fee
osh> echo $F
fee
```

`section:` を用いて `cd` コマンドのスコープに制限を加えていることに注意してください。このセクションでは一時的に `tmp` ディレクトリに移動しているので、ファイルの名前には `../fee` が用いられます。このセクションが終了してカレントディレクトリに戻ってきたとき、ファイルの名前には `fee` が用いられます。

ファイル関数を使う主な目的は、あなたのプロジェクトの `OMakefile` で、ファイル名が正しく定義されるようにするためです。これを用いればプロジェクト上の様々なパートに移動したとしても、変数は同一のファイルを指し示します。

```
osh> cat OMakefile
ROOT = $(dir .)
TMP  = $(dir tmp)
BIN  = $(dir bin)
...
```

ノート: 訳注: file, dir について詳しく知りたい方は "*file, dir*" を参照してください。

7.4 イテレーション、マップ、foreach

ほとんどのビルドイン関数では配列を何も考慮することなく処理できます。

```
osh> addprefix(-D, DEBUG WIN32)
- : -DDEBUG -DWIN32 : Array
osh> mapprefix(-I, /etc /tmp)
- : -I /etc -I /tmp : Array
osh> uppercase(fee fi foo fum)
- : FEE FI FOO FUM : Array
```

mapprefix と addprefix 関数は全く異なります (addsuffix と mapsuffix 関数も同様です)。addprefix 関数は接頭辞を各々の成分にくっつけます。 mapprefix 関数は元の成分の前に新しい接頭辞を成分として加えるので、配列の長さは 2 倍になります。

ほとんどの関数は配列でも動きますが、あなた自身が配列にも対応した関数を作りたいと思うこともあるでしょう。 foreach 関数はその要望を実現します。 foreach 関数は二つの表記を使いますが、このコードを伴った表記法はとても便利です。この関数は二つの引数とコードが必要です。まず、一つ目の引数は変数で、二つ目のは配列を指定します。そして foreach のコードは各々の成分を対象に、引数で指定された変数がその成分に束縛された状態で、配列の長さぶんだけ実行されます。さあ、それでは値を持った配列の各々の成分に 1 を加える関数を定義してみましょう。

```
osh> add1(l) =
         foreach(i, $l):
             add($i, 1)
osh> add1(7 21 75)
- : 8 22 76 : Array
```

あなたはファイル名を持った配列を持ち、そのそれぞれにビルドルールを定義したいと思うこともあるでしょう。ビルドルールは特別なものではなく、あなたはどの箇所でもビルドルールを定義することができます。さて、私たちは配列にある、各々のファイルの処理について記述し、さらに結果を tmp/ ディレクトリの中に置く関数を書きたいものとします。

```
TMP = $(dir tmp)

my-special-rule(files) =
   foreach(name, $(files))
      $(TMP)/$(name): $(name)
         process $< > $@
```

後に、プロジェクトの他の部分で、私たちはこの関数を用いていくつかのファイルを処理することを決めたとしましょう。

```
# src/lib に処理するためのファイルが入っています
MY_SPECIAL_FILES[] =
    fee.src
    fi.src
    file with spaces in its name.src
my-special-rule($(MY_SPECIAL_FILES))
```

my-special-rule を呼んだ結果は、以下の 3 つのルールを明示的に書いた場合と全く同じになります。

```
$(TMP)/fee.src: fee.src
    process fee > $@
$(TMP)/fi.src: fi.src
    process fi.src > $@
$(TMP)/$"file with spaces in its name.src": $"file with spaces in its name.src"
    process $< > $@
```

もちろん、これらのルールを記述することは関数を呼ぶことより好ましいものではありません。関数を抽象化することによる普通の特性は、普通の利点となります。ビルドルールを定義するためのコードが一つだけで済み、コードはさらに短くなります。後でもし私たちがルールを修正したりアップデートしたいと思ったときも、単純に一つのルールを修正するだけでよいのです。

7.5 遅延評価式

omake での評価は通常の場合先行して行われます。これは、omake が式に遭遇した場合、即座に評価が行われることを意味しています。この効果の一つとして、変数定義式の右側は、変数が定義される時点で展開されることが挙げられます。

この振る舞いをコントロールするための 2 つの方法があります。 `$'(v)` は遅延評価を行うための、 `$,(v)` は先行評価に戻すための表記法です。以下のシーケンスについて考えてみましょう。

```
osh> A = 1
- : "1" : Sequence
osh> B = 2
- : "2" : Sequence
osh> C = $'(add $(A), $,(B))
- : $(apply add $(apply A) "2" : Sequence)
osh> println(C = $(C))
C = 3
osh> A = 5
- : "5" : Sequence
osh> B = 6
- : "6" : Sequence
osh> println(C = $(C))
C = 7
```

`C = $'(add $(A), $,(B))` では遅延評価を定義しています。 `add` 関数はこの場合、実際に値が必要となるときまで評価しません。上の式を見てみると、 `$,(B)` は `B` が即座に評価する変数であることを指定します。これが遅延評価式の中で定義されているにもかかわらずです。

最初私たちが `C` の値を出力したとき、 `A` は 1 で `B` が 2 であったので、結果は 3 と評価されました。次に私たちが同様に `C` を出力したとき、 `A` は 5 に再定義されていたので、結果は 7 と評価されました。二回目の `B` は `C` の定義時に評価されているため、なんの影響も与えていません。

7.5.1 遅延評価式についての追加例

遅延評価式は実際に結果が必要とされるときまで評価されません。筆者を含む結構な人数のプログラマは、遅延評価を多用しているコードを見ると眉をひそめます。なぜならこれは実際にどこで評価が行われているのか分かりにくくなるからです。しかしながら、これらの難点をペイオフできるケースも確かに存在します。

一つの例としてオプションの処理が挙げられます。C コンパイラに”include” ディレクトリの指定をコマンドライン上から指定する場合を考えましょう。もし私たちが”/home/jyh/include” と”../foo” 上のファイルをインクルードしたい場合、コマンドラインにはオプション `-I/home/jyh/include -I../foo` を指定する必要があります。

C ファイルをビルドするための通常のルールを定義する場合について考えましょう。私たちはインクルードされるディレクトリを指定するため、 `INCLUDES` 配列を定義し、ルートの `OMakefile` に通常用いる暗黙のルールを定義しました。

```
# C ファイルをコンパイルする通常の設定
CFLAGS = -g
INCLUDES[] =
%.o: %.c
   $(CC) $(CFLAGS) $(INCLUDES) -c $<

# src ディレクトリは 4 つのソースファイルから my_widget をビルドします。
# これはインクルードディレクトリからインクルードファイルを読み込みます。
.SUBDIRS: src
    FILES = fee fi foo fum
```

```
    OFILES = $(addsuffix .o, $(FILES))
    INCLUDES[] += -I../include
    my_widget: $(OFILES)
       $(CC) $(CFLAGS) -o $@ $(OFILES)
```

しかしこれは全く正しいというわけではありません。問題としては、 INCLUDES はオプションが格納されてある配列であり、ディレクトリではないという点です。もし私たちが後にディレクトリ名を変更したい場合は、まず -I 接頭辞を配列から分割する必要があり、これは混乱の元となります。さらに、私たちはディレクトリの絶対パスを使用していません。この問題を解決する方法は、遅延評価式を使うことです。まず私たちは INCLUDES をディレクトリの配列として定義し、さらに新しい変数 PREFIXED_INCLUDES を定義することによって -I 接頭辞を追加します。 PREFIXED_INCLUDES は遅延評価を行うことで、最新の INCLUDES 変数値が使われることを保証してくれます。

```
# C ファイルをコンパイルする通常の設定
CFLAGS = -g
INCLUDES[] =
PREFIXED_INCLUDES[] = $`(addprefix -I, $(INCLUDES))
%.o: %.c
   $(CC) $(CFLAGS) $(PREFIXED_INCLUDES) -c $<

# 今回の例では、私たちはインクルードディレクトリの絶対パスを定義しました。
STDINCLUDE = $(dir include)

# src ディレクトリは 4 つのソースファイルから my_widget をビルドします。
# これはインクルードディレクトリからインクルードファイルを読み込みます。
.SUBDIRS: src
    FILES = fee fi foo fum
    OFILES = $(addsuffix .o, $(FILES))
    INCLUDES[] += $(STDINCLUDE)
    my_widget: $(OFILES)
       $(CC) $(CFLAGS) -o $@ $(OFILES)
```

遅延する値と関数が密接に繋がっていることに注目してください。上の例では、私たちは PREFIXED_INCLUDES を、引数を持たない関数として定義しているのと同じことを行っています。

```
PREFIXED_INCLUDES() =
    addprefix(-I, $(INCLUDES))
```

7.6 スコープとエクスポート

OMake の言語は (IO とシェルコマンドは除きますが) 関数型言語となっています。これは、まず関数が最上級のものであることと、変数が不変なものである (代入という操作が存在しない) という 2 つから、そうであると言えるでしょう。後者に関しては、おそらく従来の GNU make を使っていたユーザからすると奇妙なものに思えるかもしれません。しかしこれは実際に OMake を使う上で非常に重要な点となります。変数は修正できませんので、プロジェクトの一部分が他の部分に干渉することは不可能 (あるいは非常に困難) です。

これを従来の純粋な関数型言語のように実装してしまうと、非常に使いにくいものになるかもしれません。OMake では、インデントすることでレベルを一つ挙げた場合、新しいスコープが導入されます。そしてスコープが終わると、そのスコープで新しく定義された変数は消去されます。もし OMake が馬鹿真面目にスコープに関して厳格な仕様であったなら、おそらくコードはもっと複雑なものになったでしょう。

```
osh> X = 1
osh> setenv(BOO, 12)
osh> if $(equal $(OSTYPE), Win32)
         setenv(BOO, 17)
```

```
         X = 2
osh> println($X $(getenv BOO))
1 12
```

export コマンドはこの制限を外に出します。このコマンドは内部のスコープの値 (あるいは全体の変数環境) を外部に『エクスポート』するお世話をします。

```
osh> X = 1
osh> setenv(BOO, 12)
osh> if $(equal $(OSTYPE), Win32)
         setenv(BOO, 17)
         X = 2
         export
osh> println($X $(getenv BOO))
2 17
```

エクスポートは、ループ中のイテレーションから次のイテレーションへ値をエクスポートするのに特に役立ちます。

```
# オーケー、それでは配列の各成分を足し合わせてみよう
osh>sum(l) =
        total = 0
        foreach(i, $l)
            total = $(add $(total), $i)
        value $(total)
osh>sum(1 2 3)
- : 0 : Int

# おっと、正常に動いていないじゃないか！
osh>sum(l) =
        total = 0
        foreach(i, $l)
            total = $(add $(total), $i)
            export
        value $(total)
osh>sum(1 2 3)
- : 6 : Int
```

while ループは自動的にエクスポートしてくれる、別の形のループ文です。

```
osh>i = 0
osh>total = 0
osh>while $(lt $i, 10)
        total = $(add $(total), $i)
        i = $(add $i, 1)
osh>println($(total))
45
```

7.7 シェルエイリアス

ときどきあなたは **エイリアス** を定義したいと思うことがあるかもしれません。そのために OMake では、実際にシェルコマンドが存在しているかのようにふるまうコマンドが存在します。あなたはこれを、 Shell オブジェクトに対象の関数を追加することで実現できます。

例えば、 awk 関数を用いて、あるファイル中のすべてのコメントを出力する場合について考えてみましょう。

```
osh>cat comment.om
# Comment function
comments(filename) =
    awk($(filename))
    case $'^#'
        println($0)
# File finished
osh>include comment
osh>comments(comment.om)
# Comment function
# File finished
```

これをエイリアスとして追加するには、 Shell オブジェクトにメソッドを追加します。 += を用いて、シェルの既存の内容を保存している点に注意してください。

```
osh>Shell. +=
        printcom(argv) =
            comments($(nth 0, $(argv)))
osh>printcom comment.om > output.txt
osh>cat output.txt
# Comment function
# File finished
```

シェルコマンドは引数として配列 argv が渡されます。これはエイリアスの名前には **含まれていません** 。

7.8 簡単に入出力のリダイレクションを行う

結果的に、スコーピングによってリダイレクションの実行に関しての良い代替案も表れることとなりました。それでは、既に標準の出力先に出力するコードが大量にあるが、この出力先のリダイレクションを行いたいというような場合について考えてみましょう。まず一つ目の方法としては、前回のテクニックを用いることが挙げられます。具体的には、関数をエイリアスとして定義し、あなたが望む出力先にすることでシェルのリダイレクションを行うといった方法です。

別の方法については、前者の方法よりも簡単な場合があります。変数 stdin stdout stderr は標準 I/O の出力先について定義しています。出力先をリダイレクトするには、これらの変数をあなたが望むように再定義します。もちろん、あなたはこれを普通にネストされたスコープ上で行うことができるので、外部の出力先に影響を与えることはありません。

```
osh>f() =
        println(Hello world)
osh>f()
Hello world
osh>section:
        stdout = $(fopen output.txt, w)
        f()
        close($(stdout))
osh>cat output.txt
Hello world
```

これはシェルコマンドに対しても同様です。もしあなたがギャンブル好きであるならば、以下の例を試してみるのもいいでしょう。

```
osh>f() =
        println(Hello world)
osh>f()
Hello world
osh>section:
```

```
        stdout = $(fopen output.txt, w)
        f()
        cat output.txt
        close($(stdout))
osh>cat output.txt
Hello world
Hello world
```

CHAPTER 8

ビルドルール

OMake で使われているルールは、どのようにしてファイルをビルドするのかについて指定しています。最も簡単な例は以下のような形です。

```
<target>: <dependencies>
    <commands>
```

<target> ではビルドするファイル名を記述します。 <dependencies> では <target> をビルドする前に必要なファイルのリストを記述します。 <commands> はターゲットをビルドするためのコマンドを、インデントした状態で記述します。例えば、以下のルールではどのようにファイル hello.c をコンパイルするのかについて指定しています。

```
hello.o: hello.c
    $(CC) $(CFLAGS) -c -o hello.o hello.c
```

このルールでは、 hello.o ファイルは hello.c ファイルに依存しています。 hello.c ファイルが変更された場合、コマンド $(CC) $(CFLAGS) -c -o hello.o hello.c が実行されて、ターゲットファイル hello.o は更新されます。

ルールの中には任意の数のコマンドを含めることができます。各々の独立したコマンドラインはコマンドシェルによって、独立した状態で実行されます。コマンドの最初にタブを含めることはできません。しかし、依存関係を指定している行からはインデントされなければなりません。

通常の変数に加えて、以下の特殊変数をルールの内容で使うことができます。

- $* : 拡張子を除いたターゲットの名前
- $@ : ターゲットの名前
- $^ : 依存ファイルのリストをアルファベット順に並べ、かつ重複した内容を削除したもの
- $+ : 元の順番で並んでいる依存ファイルのリスト
- $< : 最初の依存ファイル

例えば、上の hello.c ルールは以下のように簡略化されます。

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

通常の値とは異なり、ルール中の変数は遅延評価されて、かつ動的なスコーピングが行われます。以下の関数定義はこの性質のいくつかを端的に表しています。

```
CLibrary(name, files) =
    OFILES = $(addsuffix .o, $(files))

    $(name).a: $(OFILES)
        $(AR) cq $@ $(OFILES)
```

この関数は .o ファイルのリストから $(name) という名前のプログラムをビルドしています。引数のファイルは拡張子なしで指定されているので、まず最初の定義では各々のファイル名に拡張子 .o を加えた配列を、変数 OFILES に束縛しています。次のステップでは $(OFILES) からターゲットライブラリ $(name).a をビルドするためのルールを定義しています。 $(AR) は関数が呼び出されて、呼び出してるスコープから変数 AR が与えられた時点で評価されます (詳細はスコーピングのセクションを参照してください)。

8.1 暗黙のルール

ルールはまた暗黙的にすることもできます。これは、ファイルがワイルドカードパターンによって指定されることを示しています。ワイルドカードとして OMake では % を使用します。例えば、以下のルールでは .o ファイルをビルドするための通常のルールを指定しています。

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

このルールは任意の .c ファイルから .o ファイルをビルドするためのテンプレートとなっています。

通常、暗黙のルールはカレントディレクトリ中のターゲットのみに用いられます。しかしながら、 .SUBDIRS ルールを経由したものも含むサブディレクトリには、スコープ中にある暗黙のルールがすべて継承されます (詳細はスコーピングのセクションを参照してください)。

8.2 束縛された暗黙のルール

暗黙のルールのターゲットにはいくつかのファイルを指定することもできます。そのためには以下のような構文を用います。

```
<targets>: <pattern>: <dependencies>
    <commands>
```

例えば、以下のルールではファイル a.o と b.o のみにルールを適用しています。

```
a.o b.o: %.o: %.c
     $(CC) $(CFLAGS) -DSPECIAL -c $*.c
```

8.3 section

ルールの内容に書かれているコマンドはシェルによって頻繁に評価されます。omake はまた、omake 自身も評価の対象に加えることもできます。

これらの『ファイルを作るためのルール』を表現するためには section 表現を使います。以下のルールではターゲットの hello.c を生成するために omake の IO 関数を用いています。

```
hello.c:
    section
        FP = fopen(hello.c, w)
```

```
        fprintln($(FP), $""#include <stdio.h> int main() { printf("Hello world\n"); }"")
        close($(FP))
```

この例では出力するテキストをクオートするために、クオーテーション $""..."" を用いています (詳細は B.1.6 のセクションを参照してください)。これらのクオートは出力されたファイルには含まれていません。 fopen , fprintln , close 関数は IO セクションで評価するようにファイルの IO を実行します。

加えて、関数の呼び出しや他の特殊なコマンドはその場で正しく評価されます。また、 fprintln 関数はファイルを直接出力することもできるので、上のルールは下のような形に省略できます。

```
hello.c:
   fprintln($@, $""#include <stdio.h> int main() { printf("Hello world\n"); }"")
```

8.4 section rule

また、ターゲットの依存関係がルールの内容で記述されるような場合、 section rule を使うことで計算することができます。以下のルールでは、ファイル a.c が存在していた場合は内容を hello.c にコピーし、そうでなければ hello.c は default.c から生成されます。

```
hello.c:
    section rule
       if $(target-exists a.c)
          hello.c: a.c
             cat a.c > hello.c
       else
          hello.c: default.c
             cp default.c hello.c
```

8.5 特別な依存関係

8.5.1 :exists:

時々、依存しているファイルの内容ではなく、ファイルが存在しているのかどうかが重要となる場合もあるでしょう。 :exists: 修飾子はこのような依存関係を指定したい場合に用いられます。

```
foo.c: a.c :exists: .flag
   if $(test -e .flag)
       $(CP) a.c $@
```

8.5.2 :effects:

コマンドの中には副産物として別のファイルを生み出すものもあるでしょう。例えば、 latex(1) コマンドは .dvi ファイルを生成する際、副産物として .aux ファイルを生成します。このような場合、 :effects: 修飾子はこのような副産物のファイルを明示的に指定したい場合に用いられます。omake はこのファイルを上書きしないように、平行してプログラムを実行させないようにします。

```
paper.dvi: paper.tex :effects: paper.aux
    latex paper
```

8.5.3 :value:

`:value:` 依存関係は、ルールの実行がある式の値に依存していることを指定するために用いられます。例えば、以下のルールを見てください。

```
a: b c :value: $(X)
    ...
```

上のルールでは、"a" が `$(X)` の値が変わった場合に再コンパイルされる必要があることを指定しています (X がファイル名である必要はありません)。これはファイルや変数の依存関係をより精密にコントロールすることを意図しています。

加えて、これは他の依存関係の代わりに用いることができます。例えば、以下のルールは、その次のルールと等価です。

```
a: b :exists: c
    commands

a: b :value: $(target-exists c)
    commands
```

ノート:

- 任意の値をとることができます（指定する変数に制限はありません)。
- 値はルールが展開されるときに評価されます。なので式には `$@` や `#^` のような変数も含めることができます。

8.6 .SCANNER ルール

スキャナルールでは、自動的に依存関係の解析を行う方法について定義します。 `.SCANNER` ルールは以下のような形となります。

```
.SCANNER: target: dependencies
    commands
```

ルールは、指定されたターゲットのソースファイルに定義されている、追加の依存関係を計算するのに用いられます。 `.SCANNER` コマンドの実行結果は、OMake のフォーマットで依存関係のシーケンスが標準の出力先に出力される必要があります。例えば GNU システムでは、 `gcc -MM foo.c` はファイル `foo.c` の依存関係を生成するコマンドです (これは `#include` の情報を元にしています)。

私たちはこれを用いて、 `.c` ファイルから `.o` ファイルを生成するため、スキャンされた依存関係を既存の依存関係に追加することができます。以下のスキャナではファイルの依存関係を指定しており、仮に `foo.o` が指定されたとすると、 `gcc -MM foo.c` を実行することによって依存関係の解析が行われます。さらには、`foo.c` は依存されているので、 `foo.c` が変更された場合はいつでもスキャナは再計算されます。

```
.SCANNER: %.o: %.c
    gcc -MM $<
```

コマンド `gcc -MM foo.c` は以下の行を出力するものとします。

```
foo.o: foo.h /usr/include/stdio.h
```

以上から、ファイル `foo.h` と `/usr/include/stdio.h` は `foo.o` に依存しているものと考えられます。これは、もしこれらのファイルのどれか一つが変更された場合、 `foo.o` はリビルドされるべきであることを示しています。

これはある程度ですがうまく動きます。この機能の利点の一つとしては、 `foo.c` が変更された場合、いつでもスキャナが再解析を行ってくれるというのが挙げられます。しかしながらこれには問題があります。C の依

存関係は **再帰的** なのです。すなわち、もしファイル `foo.h` が修正されたとしたら、そのファイルは他のファイルを含んでいることで、さらなる依存関係が生じているのかもしれません。必要なのは `foo.h` の変更に応じて、再びスキャナを実行することです。

私たちはこの問題を『依存関係を示す値 (value dependency)』を用いて解決しました。変数 `$&` は任意の前回のスキャンから、依存関係の結果を返す変数として定義されています。私たちはこれらの依存関係を、ファイルの MD5 による要約を計算して返す `digest` 関数を用いて追加しました。

```
.SCANNER: %.o: %.c :value: $(digest $&)
    gcc -MM $<
```

これで、ファイル `foo.h` が変更されたときには要約も変更されているのでスキャナは再計算されます。これは依存関係を示す値 (`$&` には `foo.h` がインクルードされています) によるものです。

ですが、これはまだ完全に正しいというわけではありません。C コンパイラはインクルードファイルのために **検索パス** *(search-path)* を使用しているのです。ファイル `foo.h` には何通りものバージョンが存在しており、そのうちの選んでいる一つがインクルードパスに依存しているのかもしれません。必要なのは検索パスの依存関係も含めることです。

`$(digest-in-path-optional ...)` 関数は検索パスを元にした要約を計算し、この問題に解決案を提示します。

```
.SCANNER: %.o: %.c :value: $(digest-in-path-optional $(INCLUDES), $&)
   gcc -MM $(addprefix -I, $(INCLUDES)) $<
```

スキャナルールの標準出力は OMake によって捉えられるので、OMake が依存関係を解析できない内容を含むことは許されません。まだ関連付けられていない依存関係について出力することは許されますが、このような依存関係は無視されます。スキャナルールでは標準エラーの出力先に任意の内容を出力することが許されています。このような出力は通常のルールの出力と同様に扱われます。(言い換えれば、これは –output-… オプションを有効にすることで、ユーザーに見せることのできる出力です。)

`.SCANNER` ルールについての追加例は "**依存関係の解析**" で見つけることができます。

8.6.1 スキャナの命名と :scanner: 依存関係

スキャナがルール中で使われていることを明示的に指定するほうが有用である場合があります。例えば、私たちは `.c` ファイルを異なったオプションでコンパイルしたり、あるいは (天よ我らを助けて!) `gcc` と Microsoft Visual C++ コンパイラ `cl` の両方を使いたいとしましょう。通常、 `.SCANNER` のターゲットは特定のターゲットに結びついていないのですが、私たちはこれを好きなように命名することができます。

```
.SCANNER: scan-gcc-%.c: %.c :value: $(digest-in-path-optional $(INCLUDES), $&)
    gcc -MM $(addprefix -I, $(INCLUDES)) $<

.SCANNER: scan-cl-%.c: %.c :value: $(digest-in-path-optional $(INCLUDES), $&)
    cl --scan-dependencies-or-something $(addprefix /I, $(INCLUDES)) $<
```

次のステップはスキャナの依存関係を明示的に定義することです。 `:scanner:` 依存関係はこのために用いられます。この場合、スキャナの依存関係は明示的に指定されます。

```
$(GCC_FILES): %.o: %.c :scanner: scan-gcc-%c
    gcc ...

$(CL_FILES): %.obj: %.c :scanner: scan-cl-%c
    cl ...
```

明示的な `:scanner:` スキャナの指定は、単体の `.SCANNER` ルールを複数のターゲットに向けて依存関係を生成することにも用いられます。例えば以下の例を見てみましょう。

```
.SCANNER: scan-all-c: $(GCC_FILES) :value: $(digest-in-path-optional $(INCLUDES), $&)
    gcc -MM $(addprefix -I, $(INCLUDES)) $(GCC_FILES)

$(GCC_FILES): %.o: %.c :scanner: scan-all-c
    ...
```

上の例は gcc が一回だけ呼び出されるという利点と、一つのソースファイルが変更された場合、すべてのファイルが再スキャンされてしまうという欠点の両方を持ち合わせています。

8.6.2 ノート

多くの場合において、あなたは自分の手でスキャナを定義する必要はありません。OMake には C, OCaml, LaTeX ファイルのためのスキャナ (しかも暗黙的、明示的に命名されたスキャナの両方) が標準で用意されています。

SCANNER_MODE 変数は、暗黙のスキャナの依存関係の使用についてコントロールする変数です。

明示的に :scanner: 依存関係を指定することは、スキャナが間違って指定してしまう危険を減らします。巨大で複雑なプロジェクトでは SCANNER_MODE を error に設定することで、名前がある .SCANNER ルールと明示的な :scanner: 指定を使うようにしましょう。

8.7 .DEFAULT

.DEFAULT ターゲットは、omake が明示的にターゲットを指定していないようなデフォルトの状態でビルドされるターゲットを指定するために用いられます。以下のルールでは hello プログラムがデフォルトの状態でビルドすることを指定しています。

```
.DEFAULT: hello
```

8.8 .SUBDIRS

.SUBDIRS ターゲットはプロジェクトの一部であるサブディレクトリの集合を指定するために用いられます。各々のサブディレクトリには、サブディレクトリの内容をその環境下で評価するための OMakefile をそれぞれ有しているべきです。

```
.SUBDIRS: src doc tests
```

このルールでは src , doc , test ディレクトリの中の OMakefile をそれぞれ読み込むことを指定しています。

いくつかの場合-特に OMakefile の内容が似ている大量のサブディレクトリを持っているような場合-各々のディレクトリに分割して OMakefile を持たせるのは不便でしょう。もし .SUBDIRS ルールの中に内容を記述したとすると、その内容が OMakefile の代わりとして使われます。

```
.SUBDIRS: src1 src2 src3
   println(Subdirectory $(CWD))
   .DEFAULT: lib.a
```

この場合、サブディレクトリ src1 , src2 , src3 には OMakefile が必要ありません。さらには、たとえ OMakefile が存在していた場合でも、OMake はそれを無視します。以下の例では OMakefile が含まれていた場合、そのファイルをインクルードするよう指定しています。

```
.SUBDIRS: src1 src2 src3
    if $(file-exists OMakefile)
       include OMakefile
    .DEFAULT: lib.a
```

8.9 .INCLUDE

.INCLUDE ターゲットは include 文と似ていますが、 .INCLUDE ではたとえ指定されたファイルが存在していなくても、ビルドするためのルールを指定できます。

```
.INCLUDE: config
    echo "CONFIG_READ = true" > config

 echo CONFIG_READ is $(CONFIG_READ)
```

あなたはまた .INCLUDE ルールの依存関係を指定できます。

```
.INCLUDE: config: config.defaults
   cp config.defaults config
```

順番どおりにターゲットと依存関係が記述されています。通常の場合ですと、この記法はルールがいつ期限切れになったのかどうかを決定するために用いられます。 .INCLUDE 内のルールは、以下の場合のどれかに当てはまったときに実行されます。

- ターゲットが存在しない場合
- 以前からずっとこのルールが実行されていなかった場合
- ルールが実行された最後の時間から現在までの間に、以下のうちどれかが変更されていた場合
  - ターゲット
  - 依存先
  - コマンド文

これは、たとえ既にターゲットファイルが存在していたとしても、ルールが実行される場合もあることを示しています。もしターゲットが、エディターなどで変更するような (それゆえ上書きされたくない) ファイルを指定する場合、あなたはルールの評価を、ターゲットが既に存在しているかどうかで条件分岐させるべきです。

```
.INCLUDE: config: config.defaults
    # わたしが注意深く手作業で変更したファイルを上書きさせません！
    if $(not $(file-exists config))
       cp config.defaults config
```

8.10 .PHONY

“phony” ターゲットは実際には存在しませんが、複数の依存関係を持っているようなターゲットです。Phony ターゲットは .PHONY ルールを用いて指定します。以下の例では、 install ターゲットは実際のファイルではありませんが、(omake install が実行されることによって) install ターゲットが生起された場合はいつでも、いくつかのコマンドが実行されます。

```
.PHONY: install

install: myprogram.exe
   cp myprogram.exe /usr/bin
```

8.11 スコープ規則

以前私たちが注意したように、omake は **スコープ化された** 言語です。これは非常に融通のきく-プロジェクトの異なるパートは別のパートを気にすることなく、異なった設定を定義することができる-ものとなっていま

す。例えば、プロジェクトのあるパートには CFLAGS=-O3 を持たせてコンパイルさせるが、別のパートには CFLAGS=-g を持たせるといった具合です。

しかし、どのようにしてターゲットファイルのスコープが選択されるのでしょうか？そこで、現在私たちはファイル dir/foo.o をビルドしているものとしましょう。omake ではスコープを決定するために、以下のルールを用います。

- まず、もし fir/foo.o をビルドするための明示的なルールが指定されていた (そしてそのルールはワイルドカードを用いていない) 場合、そのルールに従って、ターゲットをビルドするためのスコープが決定されます。
- さもなければ、ディレクトリ dir/ はプロジェクトの一部であるので、普通に考えて設定ファイル dir/OMakefile が存在するはずです (あるいは、 OMakefile を .SUBDIRS セクションを用いるなどのなにか別の方法で指定しているかもしれません)。この場合、ターゲットのスコープは dir/OMakefile の終わりのスコープです。

このスコープ規則を確かめるために、2 つのファイルから “Hello world” プログラムを作る例へと戻ってみましょう。これは OMakefile のサンプルです。(CFLAGS の 2 つの定義がスコープ規則の確認に用いられます)

```
# 実行ファイルはデバッグオプションを用いてコンパイルされます
CFLAGS = -g
hello: hello_code.o hello_lib.o
   $(CC) $(CFLAGS) -o $@ $+

# CFLAGS の再定義
CFLAGS += -O3
```

このプロジェクトでは、ターゲット hello は **明示的** です。 hello ターゲットのスコープは hello: から始まる行なので、 CFLAGS の値は -g となります。別の 2 つのターゲット hello_code.o と hello_lib.o は明示的にターゲットが指定されていないため、これらのスコープは OMakefile の終わりで決定しますので、 CFLAGS の値は -g -O3 となります。これは、 hello が CFLAGS=-g でリンクされて、 .o ファイルが CFLAGS=-g -O3 でコンパイルされることを表しています。

私たちは明示的にターゲットを指定することで、任意のターゲットのふるまいを変えることができます。例えば、私たちは hello_lib.o を、プリプロセッサ変数 LIBRARY を用いてコンパイルしたいものとしましょう。

```
# 実行ファイルはデバッグオプションを用いてコンパイルされます
CFLAGS = -g
hello: hello_code.o hello_lib.o
   $(CC) $(CFLAGS) -o $@ $+

# hello_lib.o を CFLAGS = -g -DLIBRARY オプションでコンパイル
section
    CFLAGS += -DLIBRARY
    hello_lib.o:

# CFLAGS の再定義
CFLAGS += -O3
```

この場合、 hello_lib.o の暗黙のターゲットが、 CFLAGS=-g -DLIBRARY のスコープ中で指定されています。これはルールの内容が指定されていないため、 .o ファイルは、通常の暗黙のルールを使って (CFLAGS=-g -DLIBRARY のオプションで) コンパイルされます。

8.11.1 暗黙のルールのスコーピング

暗黙のルール (ワイルドパターンを含む) は **グローバルではなく** 、通常のスコープ規則に従っています。これによって、異なるプロジェクトのパートは、異なる暗黙のルールの集合を持つことができるようになりました。

もしやってみたいのであれば、私たちは上のサンプルを、以下のような新しい暗黙のルールに修正することができます。

```
# 実行ファイルはデバッグオプションを用いてコンパイルされます
CFLAGS = -g
hello: hello_code.o hello_lib.o
   $(CC) $(CFLAGS) -o $@ $+

# hello_lib.o を CFLAGS = -g -DLIBRARY オプションでコンパイル
section
    %.o: %.c
        $(CC) $(CFLAGS) -DLIBRARY -c $<
    hello_lib.o:

# CFLAGS の再定義
CFLAGS += -O3
```

この場合、ターゲット hello_lib.o は、%.o をビルドするための新しい暗黙のルールがあるスコープ中でビルドされます。この暗黙のルールでは -DLIBRARY オプションを加えています。この暗黙のルールはターゲット hello_lib.o だけに定義されるため、ターゲット hello_code.o は普通にビルドされます。

8.11.2 .SCANNER ルールのスコーピング

スキャナルールは通常のスコープ規則と同様にスコープされます。 .SCANNER ルールが明示的に指定されていた (ワイルドカードパターンを含まない) 場合、スキャンターゲットのスコープはルールと同様に扱われます。.SCANNER ルールが暗黙的に指定されていた場合、その環境は :scanner: 依存関係によって決定されます。

```
# 実行ファイルはデバッグオプションを用いてコンパイルされます
CFLAGS = -g
hello: hello_code.o hello_lib.o
   $(CC) $(CFLAGS) -o $@ $+

# .c ファイルのスキャナ
.SCANNER: scan-c-%.c: %.c
   $(CC) $(CFLAGS) -MM $<

# hello_lib.o を CFLAGS = -g -DLIBRARY オプションでコンパイル
section
    CFLAGS += -DLIBRARY
    hello_lib.o: hello_lib.c :scanner: scan-c-hello_lib.c
       $(CC) $(CFLAGS) -c $<

# hello_code.c を CFLAGS = -g -O3 オプションでコンパイル
section
    CFLAGS += -O3
    hello_code.o: hello_code.c :scanner: scan-c-hello_code.c
       $(CC) $(CFLAGS) -c $<
```

再びこの例が登場しましたーあなたは恐らく、このような複雑怪奇な設定を書きたいとは思っていないでしょう！この場合、 .SCANNER ルールでは、C コンパイラが依存関係を計算するために -MM フラグを用いて呼び出すことを指定しています。ターゲット hello_lib.o には、スキャナは CFLAGS=-g -DLIBRARY が呼ばれ、 hello_code.o には、 CFLAGS=-g -O3 が呼ばれます。

8.11.3 .PHONY ターゲットのスコーピング

Phony ターゲット (実際のファイルに相当しないターゲット) は .PHONY ルールを用いて定義されます。Phony ターゲットは普通にスコープされます。以下の例はよくある間違いで、 .PHONY ターゲットはそれが使われている **後で** 宣言されています。

```
# !!この例は正常に動きません!!
all: hello

hello: hello_code.o hello_lib.o
    $(CC) $(CFLAGS) -o $@ $+

.PHONY: all
```

.PHONY 宣言が非常に遅れてしまっているために、この例は期待している通りには動きません。正しくは .PHONY 宣言を最初に持っていきます。

```
# Phony ターゲットはそれが使われる前に宣言されなければなりません
.PHONY: all

all: hello

hello: hello_code.o hello_lib.o
    $(CC) $(CFLAGS) -o $@ $+
```

Phony ターゲットはサブディレクトリに渡されます。実用性から、すべての .PHONY ターゲットを .SUBDIRS が表れる前に、ルートの OMakefile に宣言することは賢い判断と言えるでしょう。これは以下の点を保証してくれます。

1. 各々のサブディレクトリに Phony ターゲットが渡される
2. プロジェクトのルートディレクトリからビルドすることができる

```
.PHONY: all install clean

.SUBDIRS: src lib clib
```

ノート: .SUBDIRS を経由したサブディレクトリによって .PHONY ターゲットが継承されたときに、全体の .PHONY ターゲット (これはグローバルの一部分です) の階層構造が作られます。詳細は下のセクション "*.PHONY* **ターゲットの階層構造**" を参照してください。

8.12 サブディレクトリから OMake を実行

omake foo を実行した場合、 **全体** のプロジェクトの文脈から foo ファイルに関連する部分だけをビルドします。たとえそれがプロジェクトのサブディレクトリから実行したとしてもです。それゆえ、もし bar/baz が通常のターゲット (.PHONY ターゲットではない) であるとすると、 omake bar/baz を実行することは (cd bar; omake baz) を実行することと等価です。

上のルールには、2 つの注目に値する例外が存在します。

- サブディレクトリがプロジェクトの一部でなかった (.SUBDIRS に存在しない) 場合、あなたがそのディレクトリから実行させようとすると、OMake は文句を言うでしょう。
- サブディレクトリ自身に OMakeroot が含まれている場合、OMake はサブディレクトリを別のプロジェクトとして解釈します。これはあまり良い考えではないので推奨しません。

8.12.1 サブディレクトリの Phony ターゲット

.PHONY: clean がルートの OMakefile で宣言されており、さらにルートの OMakefile と、サブディレクトリのいくつかの OMakefile が clean: ルールを含んでいるものとします。この場合、

- omake clean をルートのディレクトリで実行したときには、(適切なディレクトリの中にある各々の) すべての clean: ルールが実行されます。
- omake clean をサブディレクトリで実行したときには、カレントサブディレクトリの clean: ルールだけが実行されます。

上のルールは .DEFAULT を含んだ、ビルドインの .PHONY ターゲットに等しく適用されます。すなわち、もし OMake がプロジェクトのルートディレクトリで (引数なしで) 実行された場合、プロジェクト中のすべての .DEFAULT がビルドされます。一方で、サブディレクトリで OMake が (引数なしで) 実行された場合、サブディレクトリ中で定義された .DEFAULT ターゲットだけがビルドされます。

以下のセクションでは上のようなふるまいをもたらしている、基本的な概念について説明します。

8.12.2 .PHONY ターゲットの階層構造

ルートの OMakefile が .PHONY:clean を含んでいた場合、OMake は以下を生成します。

- “グローバル (global)” な Phony ターゲット /.PHONY/clean (先頭の “/” に注目してください)
- カレントディレクトリに所属している、” 関連している (relative)”Phony ターゲット .PHONY/clean (先頭の “/” が欠けていることに注目してください)
- 依存関係 /.PHONY/clean: .PHONY/clean

ルートの OMakefile では、 .PHONY: clean 宣言の後にくるすべての clean: ... ルールは、.PHONY/clean ターゲットのルールとして解釈されます。

それでは、次に OMake は (上の .PHONY: clean 宣言のスコープ上で) .SUBDIRS: foo に遭遇したとしましょう。この場合、OMake は以下の処理を行います。

- 新しい .PHONY/foo/clean Phony ターゲットを生成します。このターゲットは” 関連している”Phony ターゲットです。
- 依存関係 .PHONY/clean: .PHONY/foo/clean を生成します。
- .SUBDIRS: foo の内容を処理するか、もし内容が空であった場合は foo/OMakefile を読み込みます。処理している間、これらの指示は foo ディレクトリに関連しているものと解釈します。特に、すべての clean: ... ルールは .PHONY/foo/clean に適用されます。

あなたが omake clean をプロジェクトのルートディレクトリで実行した場合、これは omake .PHONY/clean として解釈され (通常のターゲットに関しても同様です)、 .PHONY/clean のルールと、その依存関係 .PHONY/foo/clean のルールの両方が実行されます。また、 (cd foo; omake clean) を実行することは、 omake .PHONY/foo/clean を実行することと等価であり、 .PHONY/foo/clean のルールだけが実行されます。これもまた、通常のターゲットに関して同様です。

8.13 ルール中でのパス名

ルール中では、ターゲットと依存先は最初に” ファイル” として変換されます (詳細は “*file, dir*” を参照してください)。これらはコマンドライン上で文字列として変換される値です。また、これによっていくつかの期待していないふるまいを起こすことがあります。以下の例では、 absname 関数はファイル a の絶対パスを返す関数ですが、それにも関わらずこのルールでは相対パスとして出力されています。

```
.PHONY: demo
demo: $(absname a)
    echo $<

# omake のデモ
a
```

この振る舞いについては、議論の余地がある良い理由があります。Win32 のシステム上では、 / という文字は『オプションの指定子』として判定されます。そして、パス名のセパレータには\ が用いられています。OMake はファイル名を自動的に変換することで、両方のシステムで期待通りの動きをするようにしてくれます。

```
demo: a/b
    echo $<

# omake のデモ (Unix のシステム上)
a/b
# omake のデモ (Win32 のシステム上)
a\b
```

ときどき、あなたはターゲット名を、ルール中のコマンドに文字通り渡したいと思うことがあるかもしれません。これを解決する一つの方法は、変数を指定してあげることです。

```
SRC = a/b $(absname c/d)
demo: $(SRC)
    echo $(SRC)

# omake のデモ (Win32 のシステム上)
a/b c:\...\c\d
```

ついでに、あなたはファイル名を自動的に絶対パスに展開してほしいと思うこともあるかもしれません。例えば、エラーを見るために、OMake の出力を解析するような場合には有効でしょう。これを実現するために、あなたは --absname オプションを用いることができます (詳細は *-absname* を参照してください)。もしあなたが omake を --absname オプションで呼び出した場合、すべてのファイル名は絶対パスとして展開されます。

```
# omake --absname のデモ (Unix のシステム上)
/home/.../a/b /home/.../c/d
```

ついでに、 --absname オプションはスコープ化されています。もしあなたがこれを限られたルールでのみ用いたい場合は、 OMakeFlags 関数を使うことで、 --absname オプションを適用するかどうかをコントロールすることができます。

```
section
   OMakeFlags(--absname)
   demo: a
       echo $<

# omake デモ
/home/.../a
```

警告: --absname オプションは現在、実験的な機能として搭載しています。

CHAPTER 9

基本ライブラリ

9.1 ビルドイン変数

- **OMAKE_VERSION**

  OMake のバージョンを表します。

- **STDLIB**

  OMake の基本ライブラリのファイルがあるディレクトリを表します。起動時に、この変数のデフォルトの値は以下のようにして決定されます。

  – `OMAKELIB` 環境変数の値が存在している場合はその値が用いられます。ただし、値は絶対パスでなければなりません。

  – Windows 上では、レジストリのキー `HKEY_CURRENT_USER\SOFTWARE\MetaPRL\OMake\OMAKELIB` と `HKEY_LOCAL_MACHINE\SOFTWARE\MetaPRL\OMake\OMAKELIB` が調べられ、存在している場合にはその値が用いられます。

  – さもなければコンパイルされた時の値が用いられます。

  現在のデフォルトの値は `omake --version` を実行することによって参照できます。

- **OMAKEPATH**

  `include` と `open` 文における検索パスを指定した、ディレクトリの配列です (詳細は "**ファイルのインクルード**" を参照してください)。デフォルトの値は `.` と `$(STDLIB)` の 2 つの成分を持った配列です。

- **OSTYPE**

  omake を実行しているマシンのアーキテクチャの集合です。考えられる値は `Unix` (Linux や Mac OS X を含む、すべての Unix のバージョンを表します), `Win32` (MS-Windows では、OMake は MSVC++か Mingw を用いてコンパイルします), `Cygwin` (MS-Windows では、OMake は Cygwin を用いてコンパイルします) があります。

- **SYSNAME**

  現在のマシンの OS の名前を表します。

- **NODENAME**

  現在のマシンのホスト名を表します。

- **OS_VERSION**

  OS のバージョンを表します。

- **MACHINE**

  マシンのアーキテクチャを表します (例: `i386` , `sparc` , etc...)。

- **HOST**

  `NODENAME` と等価です。

- **USER**

  処理を実行しているユーザのログイン名を表します。

- **HOME**

  処理を実行しているユーザのホームディレクトリを表します。

- **PID**

  OMake のプロセス ID を表します。

- **TARGETS**

  コマンドラインのターゲットを表す文字列です。例えば、OMake が以下のコマンドラインで実行されたとしましょう。

  ```
  omake CFLAGS=1 foo bar.c
  ```

  この場合、 `TARGETS` は `foo bar.c` が定義されます。

- **BUILD_SUMMARY**

  `BUILD_SUMMARY` 変数は `omake` がビルド状況を要約したファイルが定義されています (メッセージはビルドの最後で出力されます)。ビルドが開始されたとき、このファイルは空です。あなたはビルド中にこのファイルを編集したり追加することで、ビルドの要約に何らかのメッセージを追加できます。

  例えば、もしあなたがいくつかのアクションが発生した場所を把握しておきたいとしますと、ビルドの要約に以下を追加することで実現できます。

  ```
  foo: boo
      echo "The file foo was built" >> $(BUILD_SUMMARY)
      ...build foo...
  ```

- **VERBOSE**

  いくつかのコマンドのメッセージが冗長に出力されます。デフォルトの値は `false` で、 `--verbose` オプションを用いて OMake が実行された場合、変数の値は `true` となります。

9.2 論理式、真偽関数、コマンドのコントロール

omake のブーリアン型は状況に無反応な (case-insensitive) 文字列によって表現されます。『偽』は文字列 `false` , `no` , `nil` , `undefined` , `0` のいづれかによって表現されます。それ以外はすべて『真』となります。

9.2.1 not

```
$(not e) : String
   e : String
```

not 関数は真偽値を反転します。

例えば、 $(not false) は true が返されて、 $(not hello world) は false が返されます。

9.2.2 equal

equal 関数は 2 つの値が等しいかどうか比較します。

例えば、$(equal a, b) は false が返されて、$(equal hello world, hello world) は true が返されます。

9.2.3 and

```
$(and e1, ..., en) : String
   e1, ..., en: Sequence
```

and 関数は引数の論理積を評価します。

例えば、以下のコードでは X は真で、 Y は偽となります。

```
A = a
B = b
X = $(and $(equal $(A), a) true $(equal $(B), b))
Y = $(and $(equal $(A), a) true $(equal $(A), $(B)))
```

9.2.4 or

```
$(or e1, ..., en) : String
   e1, ..., en: String Sequence
```

or 関数は引数の選言を評価します。

例えば、以下のコードでは X は真で、 Y は偽となります。

```
A = a
B = b
X = $(or $(equal $(A), a) false $(equal $(A), $(B)))
Y = $(or $(equal $(A), $(B)) $(equal $(A), b))
```

9.2.5 if

```
$(if e1, e2[, e3]) : value
   e1 : String
   e2, e3 : value
```

if 関数は真偽値を基にした条件分岐を行います。例えば、 $(if $(equal a, b), c, d) は d と評価されます。

条件分岐は以下のような文を用いても宣言できます。

```
if e1
   body1
elseif e2
   body2
...
```

```
else
   bodyn
```

もし式 e1 が偽でなかったら body1 が評価されて、結果は条件分岐の値として返されます。もし e1 が偽であるなら、条件分岐は移り変わり e2 の式が用いられます。もしどの条件式も真でなかった場合、 bodyn が評価されて、結果は条件分岐の値として返されます。

if 文は任意の数の elseif 文を加えることができます。また、 else 文はなくても構いません。

ノート: 各々の条件分岐文はそれぞれのスコープを持っているので、条件文中で定義された変数は通常外から見ることができません。 export コマンドはスコープ中で定義された変数をエクスポートするために用いられます。たとえば、以下の式は C コンパイラの設定を定義するために良く用いられる方法です。

```
if $(equal $(OSTYPE), Win32)
   CC = cl
   CFLAGS += /DWIN32
   export
else
   CC = gcc
   CFLAGS += -g -O2
   export
```

9.2.6 switch, match

switch , match 関数はパターンのマッチングに用いられます。

```
$(switch <arg>, <pattern_1>, <value_1>, ..., <pattern_n>, <value_n>) $(match
<arg>, <pattern_1>, <value_1>, ..., <pattern_n>, <value_n>)
```

<pattern>/<value> の数は任意です。ただし、引数の数は必ず奇数でなければなりません。

<arg> は文字列として評価されて、 <pattern_1> を用いて比較されます。もしマッチしている場合、結果の式は <value_1> が返されます。そうでない場合、マッチする文が見つかるまで、残りのパターンを用いて評価が行われます。

switch 関数はパターンと引数を比較するために用いられます。例えば、以下の表現式では、 FILE 変数は OSTYPE 変数の値に依存して、 foo , bar , あるいは空の文字列が定義されます。

```
FILE = $(switch $(OSTYPE), Win32, foo, Unix, bar)
```

match 関数は正規表現を用います (“**正規表現**” を参照してください)。もしマッチしているパターンが見つかった場合、変数 $1, $2, ... は \(と \) デリミタの間にある文字列が束縛されます。 \0 変数は全体のマッチ文が定義されており、 $* はマッチした文字列の配列が定義されます。

```
FILE = $(match foo_xyz/bar.a, foo_\\\(.*\\\)/\\\(.*\\\)\.a, foo_$2/$1.o)
```

switch と match 関数は代わりに (もっと便利な) 以下のような形に書くことができます。

```
match e
case pattern1
   body1
case pattern2
   body2
...
default
   bodyd
```

式 e が前回のパターンでマッチせずに pattern_i でマッチした場合、 body_i が評価されて、 match の結果として返されます。 switch 関数は文字列の比較を行います。 match 関数は正規表現でのマッチングを行います。

```
match $(FILE)
case $".*\(\.[^\/.]*\)"
   println(The string $(FILE) has suffix $1)
default
   println(The string $(FILE) has no suffix)
```

9.2.7 try

```
try
   try-body
catch class1(v1)
   catch-body
when expr
   when-body
...
finally
   finally-body
```

try 文は例外を扱うために用いられます。はじめに、 try-body の式が評価されます。

値 v が例外を出さなかった場合、 finally-body の式が評価され値 v が結果として返されます。

try-body の評価がオブジェクト obj の例外を送出した場合、 catch 文が代わりに評価されます。 catch 文の catch class(v) を実行している最中、もし例外のオブジェクト obj がクラス名 class のインスタンスであったならば、変数 v が例外のオブジェクトとして束縛されて、 catch-body の式が評価されます。

catch 文が評価されている間中 when 文に遭遇した場合、評価式 expr が評価されます。もし結果が真であったならば、 when-body の式が続けて評価されます。さもなければ、次の catch 文が評価されます。

catch-body か when-body の評価が完全に終わった場合、別の when 文を評価することなく finally-body の式が評価されて、値 v が返されます。

try 文には任意の数の catch 文を含めることができます。また、 finally 文はなくても構いません。

9.2.8 raise

raise 関数は例外を送出します。 exn は任意のオブジェクトです。しかしながら、通常は Exception オブジェクトを送出します。

例外が捕らえられなかった場合、全体のオブジェクトはエラーメッセージとして詳細に出力されます。しかしながら、もしオブジェクトが Exception で message フィールドを含んでいるのなら、エラーメッセージは message のみが出力されます。

9.2.9 exit

```
exit(code)
   code : Int
```

exit 関数は omake を異常終了させます。

```
$(exit <code>)
```

exit 関数は終了コードである整数を引数に指定します。0 でない値は異常終了を表します。

9.2.10 defined

```
$(defined sequence) : String
   sequence : Sequence
```

defined 関数はシーケンス中のすべての変数が現在定義されているか試します。例えば、以下のコードでは変数 X が既に定義されていないかどうかを定義しています。

```
if $(not $(defined X))
   X = a b c
   export
```

これは修飾された変数にも用いることができます。

```
$(defined X.a.b)
$(defined public.X)
```

9.2.11 defined-env

```
$(defined-env sequence) : String
   sequence : String
```

defined-env 関数は処理している環境で、指定された変数が定義されているかどうか試します。

例えば、以下のコードでは、環境変数 DEBUG が定義されている場合は -g コンパイルオプションを追加します。

```
if $(defined-env DEBUG)
    CFLAGS += -g
    export
```

9.2.12 getenv

```
$(getenv name) : String
$(getenv name, default) : String
```

getenv 関数は現在処理している環境での変数の値を取得します。この関数は一つか二つの引数を指定する必要があります。

一つの引数を指定した場合、もし環境中で変数が定義されていなかったならば例外を送出します。二つの引数を指定した場合、もし定義されていなかったならば二番目の引数が返されます。

例えば、以下のコードでは、もし環境変数 PATH が定義されていた場合は、その値を空白で分割したリストとして X を定義します。さもなければ /bin /usr/bin が代わりに使われます。

```
X = $(split $(PATHSEP), $(getenv PATH, /bin:/usr/bin))
```

以下のような形でも定義することができます。

```
getenv(NAME)
    default
```

9.2.13 setenv

```
setenv(name, value)
   name : String
   value : String
```

setenv 関数は現在処理している環境での変数を定義します。環境変数は通常の変数のようにスコープされます。

9.2.14 unsetenv

```
unsetenv(names)
   names : String Array
```

unsetenv 関数は現在処理している環境からいくつかの変数を削除します。環境変数は通常の変数のようにスコープされます。

9.2.15 get-registry

```
get-registry(hkey, key, field) : String
get-registry(hkey, key, field, default) : String
    hkey : String
    key : String
    field : String
```

get-registry 関数は Win32 上のシステムレジストリから文字列を取得します。他のアーキテクチャ上では、レジストリの値は返されません。

hive (私はこの呼び方が正しいと思っています) では、使用するレジストリの区分を指定します。これは以下の値である必要があります。(訳注: よく意味がわかりませんが hkey と five をもじったもの？)

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_CONFIG
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS

もしこれらの意味について知りたい場合はマイクロソフトのドキュメントを参照してください。

key はレジストリから取得したいフィールドを指定します。これは A\B\C のような形である必要があります (通常のスラッシュを用いた場合はバックスラッシュに変換されます)。 field は key のサブフィールドを指定します。

4 つの引数を取る場合、 default の値が失敗したときに返されます。あなたはこれを別の形で用いることもできます。

```
get-registry(hkey, key, field)
   default
```

9.2.16 getvar

```
$(getvar name) : String
```

getvar 関数は変数の値を取得します。

変数が定義されていない場合は例外が送出されます。

例えば、以下のコードでは X を文字列 abc で定義します。

```
NAME = foo
foo_1 = abc
X = $(getvar $(NAME)_1)
```

これは修飾された変数にも使うことができます。

```
$(getvar X.a.b)
```

9.2.17 setvar

```
setvar(name, value)
   name : String
   value : String
```

setvar 関数は新しい変数を定義します。例えば、以下のコードでは X は文字列 abc で定義されます。

```
NAME = X
setvar($(NAME), abc)
```

これは修飾された変数にも使うことができます。

```
setvar(public.X, abc)
```

9.3 配列とシーケンス

9.3.1 array

```
$(array elements) : Array
   elements : Sequence
```

array 関数はシーケンスから配列を生成します。もし <arg> が文字列だった場合、配列の成分はホワイトスペースによって区切られた文字列となります。また、クオートによっても区切られます。

加えて、配列の変数は以下のように宣言することもできます。

```
A[] =
   <val1>
   ...
   <valn>
```

この場合、配列の成分は <val1> , ... , <valn> であり、ホワイトスペースは文字通りに取り扱われます。

9.3.2 split

```
$(split sep, elements) : Array
   sep : String
   elements : Sequence
```

split 関数は二つの引数を必要とし、一つめには文字列のデリミタ、二つめには区切りたい文字列を指定します。結果は elements シーケンスをセパレータによって区切った配列が返されます。

例えば、以下のコードでは、変数 X は配列 /bin /usr/bin /usr/local/bin に定義されます。

```
PATH = /bin:/usr/bin:/usr/local/bin
X = $(split :, $(PATH))
```

sep は除外することもできます。この場合 split はホワイトスペースで区切ります。クオーテーションは区切りません。

9.3.3 concat

```
$(concat sep, elements) : String
   sep : String
   elements : Sequence
```

concat 関数は二つの引数を必要とし、一つめには文字列のセパレータ、二つめにはシーケンスを指定します。結果は隣接した成分の間にセパレータを配置した、結合された文字列が返されます。

例えば、以下のコードでは、変数 X は文字列 foo_x_bar_x_baz に定義されます。

```
X = foo  bar     baz
Y = $(concat _x_, $(X))
```

9.3.4 length

```
$(length sequence) : Int
   sequence : Sequence
```

length 関数は引数の成分の数を返します。

例えば、式 $(length a b "c d") は 3 と評価されます。

9.3.5 nth

```
$(nth i, sequence) : value
   i : Int
   sequence : Sequence
raises RuntimeException
```

nth 関数は引数の i 番目の成分を返します。配列のインデックスは 0 から始まります。もしインデックスが存在しない成分を指定していた場合、例外が送出されます。

例えば、式 $(nth 1, a "b c" d) は "b c" と評価されます。

9.3.6 replace-nth

```
$(replace-nth i, sequence, x) : value
   i : Int
   sequence : Sequence
   x : value
raises RuntimeException
```

replace-nth 関数は i 番目の成分を新しい値 x に置き換えます。インデックスは 0 から始まります。もしインデックスが存在しない成分を指定していた場合、例外が送出されます。

例えば、式 $(replace-nth 1, a "b c" d, x) は a x d と評価されます。

9.3.7 nth-hd

```
$(nth-hd i, sequence) : value
   i : Int
   sequence : Sequence
raises RuntimeException
```

nth-hd 関数は最初から i 個までの成分をもった配列を返します。もしシーケンスが i 個より少ない場合は例外が送出されます。

例えば、式 $(nth-hd 2, a "b c" d) は a "b c" と評価されます。

9.3.8 nth-tl

```
$(nth-tl i, sequence) : value
   i : Int
   sequence : Sequence
raises RuntimeException
```

nth-tl 関数は最初から i 個までの成分を除いた、残りの配列を返します。もしシーケンスが i 個より少ない場合には例外が送出されます。

例えば、式 $(nth-tl 1, a "b c" d) は "b c" d と評価されます。

9.3.9 subrange

```
$(subrange off, len, sequent) : value
   off : Int
   len : Int
   sequence : Sequence
raises RuntimeException
```

subrange 関数はシーケンスの一部分を返します。インデックスは 0 から始まります。もし指定された範囲に成分が存在しない場合には例外が送出されます。

例えば、式 $(subrange 1, 2, a "b c" d e) は "b c" d と評価されます。

9.3.10 rev

```
$(rev sequence) : Sequence
   sequence : Sequence
```

rev 関数は指定された配列の順番を逆にした配列を返します。例えば、式 $(rev a "b c" d) は d "b c" a と評価されます。

9.3.11 join

```
$(join sequence1, sequence2) : Sequence
   sequence1 : Sequence
   sequence2 : Sequence
```

join 関数は二つのシーケンスの成分を互いに結合させます。例えば、 $(join a b c, .c .cpp .h) は a.c b.cpp c.h と評価されます。もし二つの入力シーケンスが異なる長さであった場合は、長いシーケンスの残りの成分は、出力先の配列の終わりに修正されない状態でコピーされます。

9.3.12 string

```
$(string sequence) : String
   sequence : Sequence
```

string 関数はシーケンスを一つの文字列にまとめます。これは concat 関数と似ていますが、この成分はホワイトスペースによって分割されています。結果は一つのユニットとして扱われます。ホワイトスペースは重要です。

9.3.13 string-length

```
$(string-length sequence) : Int
   sequence : Sequence
```

string-length 関数は引数の文字列の長さを返します。もし引数がシーケンスであった場合、まとめられた状態で評価されます。よって、 $(string-length sequence) は $(string-length $(string sequence)) と等価です。

9.3.14 string-escaped, ocaml-escaped, html-escaped, html-pre-escaped, c-escaped, id-escaped

```
$(string-escaped sequence) : String Array
$(ocaml-escaped sequence) : String Array
$(html-escaped sequence) : String Array
$(html-pre-escaped sequence) : String Array
$(c-escaped sequence) : String Array
$(hex-escaped sequence) : StringArray
   sequence : Array
```

string-escaped 関数は引数の各々の成分を文字列に変換し、もしその成分が OMake の特殊文字を含んでいた場合は、その文字をエスケープした状態で返します。特殊文字は :()\,$'"# とホワイトスペースを含みます。この関数はスキャナルール中で、 stdout に出力する前にファイル名をエスケープするために使われます。

ocaml-escaped 関数は OCaml の特殊文字をエスケープした状態で返します。

c-escaped 関数は C の文字定数に使われるような形の文字列に変換します。

id-escaped 関数は OMake で使われるような識別子に変換します。

html-escaped 関数は HTML で文字通りに読み込まれるような形の文字列に変換します。 html-pre-escaped 関数と似ていますが、この関数では改行は
 に変換されません。

```
println($(string $(string-escaped $"a b" $"y:z")))
a\ b y\:z
```

9.3.15 decode-uri, encode-uri

```
$(decode-uri sequence) : sequence
    sequence : Sequence
```

これら二つの関数は URI のエンコーディングに用いられ、特殊文字を 16 進数の文字に置き換えます。

```
osh> s = $(encode-uri $'a b~c')
"a+b%7ec"
osh> decode-uri($s)
"a b~c"
```

9.3.16 quote

```
$(quote sequence) : String
   sequence : Sequence
```

quote 関数はシーケンスを一つの文字列にまとめ、さらに文字列にクオートを付与します。内部のクオーテーションはエスケープされます。

例えば、式 $(quote a "b c" d) は "a \"b c\" d" に、 $(quote abc) は "abc" に評価されます。

9.3.17 quote-argv

```
$(quote-argv sequence) : String
   sequence : Sequence
```

quote-argv 関数はシーケンスを一つの文字列にまとめ、さらに文字列にクオートを付与します。クオーテーションは変換されるため、コマンドラインのパーサは、正常に文字列をその構成要素へと戻すことができます。

9.3.18 html-string

```
$(html-string sequence) : String
   sequence : Sequence
```

html-string 関数はシーケンスを一つの文字列にまとめ、さらに特殊な HTML 文字にエスケープします。concat 関数と似ていますが、この関数はホワイトスペースで分割を行います。結果は一つの文字列として返されます。

9.3.19 addsuffix

```
$(addsuffix suffix, sequence) : Array
   suffix : String
   sequence : Sequence
```

addsuffix 関数はシーケンスの各々の成分に接尾辞を付与します。返される配列の長さは、指定されたシーケンスの配列の長さと全く同じです。

例えば、 $(addsuffix .c, a b "c d") は a.c b.c "c d".c と評価されます。

9.3.20 mapsuffix

```
$(mapsuffix suffix, sequence) : Array
   suffix : value
   sequence : Sequence
```

mapsuffix 関数はシーケンスの各々の成分に接尾辞を付与します。これは addsuffix 関数と似ていますが、この関数は文字列をくっ付ける代わりに新しく成分を追加します。よって、返される配列の長さは、指定されたシーケンスの配列の長さの 2 倍です。

例えば、 $(mapsuffix .c, a b "c d") は a .c b .c "c d" .c と評価されます。

9.3.21 addsuffixes

```
$(addsuffixes suffixes, sequence) : Array
   suffixes : Sequence
   sequence : Sequence
```

addsuffixes 関数は最初の引数に指定されたすべての接尾辞をシーケンスの各々の成分に付与します。もし suffixes が n 個の成分を、 sequence が m 個の成分を持っていた場合、結果は n * m 個の成分を持ったシーケンスが返されます。

例えば、 $(addsuffixes .c .o, a b c) は a.c a.o b.c b.o c.c c.o と評価されます。

9.3.22 removeprefix

```
$(removeprefix prefix, sequence) : Array
   prefix : String
   sequence : Array
```

removeprefix 関数はシーケンスの各々の成分から接頭辞を取り除きます。

9.3.23 removesuffix

```
$(removesuffix sequence) : Array
   sequence : String
```

removesuffix 関数はシーケンスの各々の成分から接尾辞 (拡張子) を取り除きます。

例えば、 $(removesuffix a.c b.foo "c d") の結果は a b "c d" となります。

9.3.24 replacesuffixes

```
$(replacesuffixes old-suffixes, new-suffixes, sequence) : Array
   old-suffixes : Sequence
   new-suffixes : Sequence
   sequence : Sequence
```

replacesuffixes 関数はシーケンスの各々の成分の接尾辞 (拡張子) を置き換えます。 old-suffixes と new-suffixes シーケンスは同じ長さである必要があります。

例えば、 $(replacesuffixes .h .c, .o .o, a.c b.h c.z) の結果は a.o b.o c.z となります。

9.3.25 addprefix

```
$(addprefix prefix, sequence) : Array
   prefix : String
   sequence : Sequence
```

addprefix 関数はシーケンスの各々の成分に接頭辞を付与します。返される配列の長さは、指定されたシーケンスの配列の長さと全く同じです。

例えば、$(addprefix foo/, a b "c d") は foo/a foo/b foo/"c d" と評価されます。

9.3.26 mapprefix

```
$(mapprefix prefix, sequence) : Array
   prefix : String
   sequence : Sequence
```

mapprefix 関数はシーケンスの各々の成分に接頭辞を付与します。これは addprefix 関数と似ていますが、この関数は文字列をくっ付ける代わりに新しく成分を追加します。よって、返される配列の長さは、指定されたシーケンスの配列の長さの 2 倍です。

例えば、$(mapprefix foo, a b "c d") の結果は foo a foo b foo "c d" となります。

9.3.27 add-wrapper

```
$(add-wrapper prefix, suffix, sequence) : Array
   prefix : String
   suffix : String
   sequence : Sequence
```

add-wrapper 関数はシーケンスの各々の成分に接頭辞と接尾辞の両方を付与します。例えば、$(add-wrapper dir/, .c, a b) は dir/a.c dir/b.c と評価されます。文字列は結合されるため、返される配列の長さは、指定されたシーケンスの長さと全く同じです。

9.3.28 set

```
$(set sequence) : Array
   sequence : Sequence
```

set 関数は文字列の集合をソートします。さらに、重複した成分を除去します。

例えば、$(set z y z "m n" w a) の結果は "m n" a w y z となります。

9.3.29 mem

```
$(mem elem, sequence) : Boolean
   elem : String
   sequence : Sequence
```

mem 関数はシーケンス中に指定した成分が含まれているかどうか調べます。

例えば、$(mem "m n", y z "m n" w a) は true と評価されて、一方で $(mem m n, y z "m n" w a) は false と評価されます。

9.3.30 intersection

```
$(intersection sequence1, sequence2) : Array
   sequence1 : Sequence
   sequence2 : Sequence
```

intersection 関数は指定された二つの集合の和をとります。返される配列の長さは不定であり、重複があればそれを含みます。もし結果をソートし、さらに重複を除きたい場合は set 関数を使ってください。

例えば、$(intersection c a b a, b a) は a b a と評価されます。

9.3.31 intersects

```
$(intersects sequence1, sequence2) : Boolean
   sequence1 : Sequence
   sequence2 : Sequence
```

intersects 関数は二つの集合の和が空集合でないかどうか調べます。これは集合の和を計算し、空であるかどうか調べるよりも少しだけ効率的です。

例えば、$(intersects a b c, d c e) は true と評価されて、$(intersects a b c a, d e f) は false と評価されます。

9.3.32 set-diff

```
$(set-diff sequence1, sequence2) : Array
   sequence1 : Sequence
   sequence2 : Sequence
```

set-diff 関数は二つの集合の差異を計算します。結果は sequence1 には含まれるが sequence2 には含まれていない成分からなる配列です。返される配列の長さは不定であり、重複があればそれを含みます。もし結果をソートし、さらに重複を除きたい場合は set 関数を使ってください。

ノート: 訳注: この関数は集合論における f(A,B) = A - B と等価です。

例えば、$(set-diff c a b a e, b a) は c e と評価されます。

9.3.33 filter

```
$(set-diff sequence1, sequence2) : Array
   sequence1 : Sequence
   sequence2 : Sequence
```

filter 関数はシーケンスから特定の成分を抜き出します。 patterns にはパターンを定義した、空でないシーケンスを指定します。また、パターンにはワイルドカード % を含めることができます。

例えば、$(filter %.h %.o, a.c x.o b.h y.o "hello world".c) は x.o b.h y.o と評価されます。

9.3.34 filter-out

```
$(filter-out patterns, sequence) : Array
   patterns : Sequence
   sequence : Sequence
```

filter-out 関数はシーケンスから特定の成分を除去します。 patterns にはパターンを定義した、空でないシーケンスを指定します。また、パターンにはワイルドカード % を含めることができます。

例えば、$(filter-out %.c %.h, a.c x.o b.h y.o "hello world".c) は x.o y.o と評価されます。

9.3.35 capitalize

```
$(capitalize sequence) : Array
   sequence : Sequence
```

capitalize 関数はシーケンスの各々の成分の単語を大文字化します。例えば、$(capitalize through the looking Glass) は Through The Looking Glass と評価されます。

9.3.36 uncapitalize

```
$(uncapitalize sequence) : Array
   sequence : Sequence
```

uncapitalize 関数は引数に指定された各々の単語を小文字化します。

例えば、$(uncapitalize through the looking Glass) は through the looking glass と評価されます。

9.3.37 uppercase

```
$(uppercase sequence) : Array
   sequence : Sequence
```

uppercase 関数はシーケンス中の文字すべてを大文字化します。例えば、$(uppercase through the looking Glass) は THROUGH THE LOOKING GLASS と評価されます。

9.3.38 lowercase

```
$(lowercase sequence) : Array
   sequence : Sequence
```

lowercase 関数はシーケンス中の文字すべてを小文字化します。

例えば、$(lowercase through tHe looking Glass) は through the looking glass と評価されます。

9.3.39 system

```
system(s)
   s : Sequence
```

system 関数はシェル上のコマンドを評価するために用いられます。シェルコマンドを評価するため、omake は内部でこの関数を使用しています。

例えば、以下のプログラムは式 system(ls foo) と等価です。

```
ls foo
```

9.3.40 shell

```
$(shell command) : Array
$(shella command) : Array
$(shell-code command) : Int
   command : Sequence
```

shell 関数はシェルコマンドを用いてコマンドを評価し、さらに標準出力先に出力された、ホワイトスペースで区切ってある出力結果を返します。

shella 関数は同様に振る舞いますが、この関数では改行をそのまま出力するのではなく、分割された配列として返します。

shell-code は結果として終了コードを返します。出力は返されません。

例えば、もしカレントディレクトリがファイル OMakeroot , OMakefile , hello.c を含んでいる場合、$(shell ls) は Unix システム上では hello.c OMakefile OMakeroot と評価されます。

9.3.41 export

export 関数は現在の環境中の変数の値を保存します。

例えば、以下のコードは 1 1 2 と出力されます。

```
A = 1
B = 1
C = 1
SAVE_ENV = $(export A B)
A = 2
B = 2
C = 2
export($(SAVE_ENV))
println($A $B $C)
```

この関数に引数を指定することは、 export 文を用いて引数を指定するのと全く等価なものとして解釈されます (詳細は "**環境のエクスポート**" を参照してください)。

9.3.42 while

```
while <test>
   <body>
```

あるいは

```
while <test>
case <test1>
   <body1>
...
case <testn>
   <bodyn>
default
   <bodyd>
```

<test> が真である間はずっとループの式が実行されます。最初の形では、 <body> はすべてのループにおいて実行されます。二番目の形では、もし <testI> が真であった場合は <bodyI> が実行されます。もしなにも当てはまらない場合には <bodyd> が実行されます。もしすべての場合において真でなかったならば、ループは終了します。なお、ループ中の環境は自動的にエクスポートされます。

例えば、 i を 0 から 9 まで繰り返します。

```
i = 0
while $(lt $i, 10)
   echo $i
   i = $(add $i, 1)
```

上の例は以下の例と等価です。

```
i = 0
while true
case $(lt $i, 10)
   echo $i
   i = $(add $i, 1)
```

以下の例は似ていますが、いくつかの特殊な場合においてある文字が出力されます。その他は値が出力されます。

```
i = 0
while $(lt $i, 10)
case $(equal $i, 0)
   echo zero
case $(equal $i, 1)
   echo one
default
   echo $i
```

`break` 関数は `while` ループを早期に抜けたい場合に用いられます。

9.3.43 break

`break`

最も近いループから抜け出し、現在の状態を返します。

9.3.44 random, random-init

```
random-init(i)
    i : Int
random() : Int
```

乱数を生成します。値は疑似乱数で、暗号として用いられるほどセキュアではありません。

乱数生成器はシステムの乱数器を用いて初期化します。よって、次にプログラムを実行したときの乱数の値は前回と異なります。 `ramdom-init` 関数は特定の値を用いて乱数生成器を初期化します。

9.4 演算

9.4.1 int

`int` 関数は整数値を作るために用いられ、 `Int` オブジェクトを返します。

```
$(int 17)
```

9.4.2 float

`float` 関数は浮動小数点値を作るために用いられ、 `Float` オブジェクトを返します。

```
$(float 3.1415926)
```

9.4.3 基本的な演算

以下の関数は基本的な数学の演算を行います。

- `$(neg <numbers>)` : 数学的な反転
- `$(add <numbers>)` : 加算
- `$(sub <numbers>)` : 減算
- `$(mul <numbers>)` : 乗算
- `$(div <numbers>)` : 除算
- `$(mod <numbers>)` : 余り
- `$(lnot <numbers>)` : ビット単位 NOT
- `$(land <numbers>)` : ビット単位 AND
- `$(lor <numbers>)` : ビット単位 OR
- `$(lxor <numbers>)` : ビット単位 XOR
- `$(lsl <numbers>)` : 論理左シフト
- `$(lsr <numbers>)` : 論理右シフト
- `$(asr <numbers>)` : 算術右シフト
- `$(min <numbers>)` : 最も小さい成分
- `$(max <numbers>)` : 最も大きい成分

9.4.4 評価

以下の関数は数学的な評価を行います。

- `$(lt <numbers>)` : 〜より少ない (A < B)
- `$(le <numbers>)` : 〜以下 (A <= B)
- `$(eq <numbers>)` : 等しい (A == B)
- `$(ge <numbers>)` : 〜以上 (A >= B)
- `$(gt <numbers>)` : 〜より多い (A > B)
- `$(ult <numbers>)` : 〜より少ない (符号なし)
- `$(ule <numbers>)` : 〜以下 (符号なし)
- `$(uge <numbers>)` : 〜以上 (符号なし)
- `$(ugt <numbers>)` : 〜より多い (符号なし)

ノート: 訳注: ここでいう (符号なし) とは符号ビットを考慮しないで評価を行うことを表しています。例えば、int 型と unsigned int 型では同じビット数に対してそれぞれ表している数値が異なります。(符号なし) の演算子はこのような場合に用いられます。

9.5 基本的な関数群

9.5.1 fun

fun 関数は匿名関数を生成します。

```
$(fun <v1>, ..., <vn>, <body>)
```

最後の引数には関数の内容を記述します。他の引数はパラメータ名を指定します。

例えば、以下の 3 つの関数定義は等価です。

```
F(X, Y) =
   return($(addsuffix $(Y), $(X)))

F = $(fun X, Y, $(addsuffix $(Y), $(X)))

F =
   fun(X, Y)
      value $(addsuffix $(Y), $(X))
```

9.5.2 apply

apply 関数は関数に値を適用します。

```
$(apply <fun>, <args>)
```

以下の関数定義を行った場合について考えてみましょう。

```
F(X, Y) =
   return($(addsuffix $(Y), $(X)))
```

以下の 2 つの式は等価です。

```
X = F(a b c, .c)
X = $(apply $(F), a b c, .c)
```

9.5.3 applya

applya 関数は引数の配列を関数に適用します。

```
$(applya <fun>, <args>)
```

例えば、以下のプログラムでは Z の値は file.c となります。

```
F(X, Y) =
   return($(addsuffix $(Y), $(X)))
args[] =
   file
   .c
Z = $(applya $(F), $(args))
```

9.5.4 create-map, create-lazy-map

create-map 関数は簡単に Map オブジェクトを作る関数です。 create-map 関数はキー/値のペアを引数によって指定するので、引数の数は等しくなければなりません。例えば、以下の 2 つの式は等価です。

```
X = $(create-map name1, xxx, name2, yyy)

X. =
    extends $(Map)
    $|name1| = xxx
    $|name2| = yyy
```

create-lazy-map 関数は create-map と似ていますが、この関数は値が遅延評価されます。例えば、以下の 2 つの式は等価です。

```
Y = $(create-lazy-map name1, $(xxx), name2, $(yyy))

Y. =
    extends $(Map)
    $|name1| = $`(xxx)
    $|name2| = $`(yyy)
```

create-lazy-map 関数はルールを生成する際に用いられます。

9.6 イテレーションとマッピング

9.6.1 foreach

foreach 関数はシーケンスすべての成分にわたって関数を適用します。

```
$(foreach <fun>, <args>)

foreach(<var>, <args>)
   <body>
```

例えば、以下のプログラムでは変数 X は配列 a.c b.c c.c と定義されます。

```
X =
   foreach(x, a b c)
      value $(x).c

# 等価な式
X = $(foreach $(fun x, $(x).c), abc)
```

これらの表現を省略することもできます。

export 文は foreach の内容に使うこともできます。例えば、以下の式の X は最終的に a.c b.c c.c となります。

```
X =
foreach(x, a b c)
   X += $(x).c
   export
```

break 関数はこのようなループを早期に抜けたい場合に用いられます。

9.7 ブーリアン関数群

9.7.1 sequence-forall

forall 関数は <body> がシーケンスのすべての成分に当てはまっているかどうか調べます。

```
$(sequence-forall <fun>, <args>)

sequence-forall(<var> => ..., <args>)
   <body>
```

9.7.2 sequence-exists

exists 関数は <body> がシーケンスのいくつかの成分に当てはまっているかどうか調べます。

```
$(sequence-exists <fun>, <args>)

sequence-exists(<var> => ..., <args>)
   <body>
```

9.7.3 sequence-sort

sort 関数は配列の成分を与えられた評価関数を元にソートします。評価関数は二つの引数 (x, y) を取ります。もし x < y であった場合、評価関数は負の値を返す必要があります。同様に、x > y の場合は正の値、x = y の場合は 0 を返します。

```
$(sequence-sort <fun>, <args>)

sort(<var>, <var> => ..., <args>)
   <body>
```

9.7.4 compare

compare 関数は二つの値 (x, y) を比較します。もし x < y であった場合、この関数は負の値を返します。同様に、x > y の場合は正の値、x = y の場合は 0 を返します。

```
$(compare x, y) : Int
```

CHAPTER 10

システム関数

10.1 ファイル名

10.1.1 file, dir

```
$(file sequence) : File Sequence
   sequence : Sequence
$(dir sequence) : Dir Sequence
   sequence : Sequence
```

file と dir 関数は omake が動く場所に依存しない、ファイルとディレクトリの位置を定義します。omake では、ターゲットをビルドするコマンドはターゲット上のディレクトリにて実行されます。プロジェクト中には多くのディレクトリが存在するため、omake ではあるディレクトリ上のファイルを確実に指定する方法が存在します。これにより、たとえ別のディレクトリ上に移動したとしても、明示的にパスを修正せずに特定のファイルを指定することができます。 file , dir 関数は以下のように用いることで、ディレクトリやファイルのパスが関連付けられます。

例えば、カレントディレクトリ中でファイル foo を示す変数を作ったとしましょう。

```
FOO = $(file foo)
.SUBDIRS: bar
```

FOO 変数が bar サブディレクトリ中で展開された場合、これは ../foo と展開されます。

これらのコマンドはトップレベルのディレクトリを確実に指定するために、トップレベルの OMakefile にてしばしば用いられます。よって、ビルドコマンドはこれらのディレクトリを、まるで絶対パスで表しているのように扱うことができます。

```
ROOT = $(dir .)
LIB  = $(dir lib)
BIN  = $(dir bin)
```

これらの変数はいったん定義されたらサブディレクトリ中でもビルドコマンドに使うことができるので、下の $(BIN) はコマンドが実行されたディレクトリに関連付けられた bin ディレクトリで展開されます。

```
install: hello
  cp hello $(BIN)
```

10.1.2 tmpfile

```
$(tmpfile prefix) : File
$(tmpfile prefix, suffix) : File
    prefix : String
    suffix : String
```

tmpfile 関数は一時的なディレクトリ中にある、一時的な空のファイル名を返します。

10.1.3 in

```
$(in dir, exp) : String Array
   dir : Dir
   exp : expression
```

in 関数は dir や file 関数と密接に関係しています。ディレクトリとファイル名を指定することで、ディレクトリ上からのファイルの位置を返します。例えば、ファイルをインストールするためによく使われる方法の一つとしてシンボリックリンクを定義することで、その値は特定のディレクトリに関連付けられているものとします。

以下のコマンドでは $(LIB) ディレクトリ中でリンクを生成しています。

```
FOO = $(file foo)
install:
   ln -s $(in $(LIB), $(FOO)) $(LIB)/foo
```

ノート: in 関数は Node (File と Dir) が与えられた場合のみに評価を行います。

10.1.4 basename

```
$(basename files) : String Sequence
   files : String Sequence
```

basename 関数は引数に指定されたファイルのリストから元のファイル名のみを返します。任意のディレクトリパスは除去されます。

例えば、式 $(basename dir1/dir2/a.out /etc/modules.conf /foo.ml) は a.out modules.conf foo.ml と評価されます。

10.1.5 dirname

```
$(dirname files) : String Sequence
   files : String Sequence
```

dirname 関数は引数に指定されたファイルのリストからディレクトリ名を返します。任意のファイル名は除去されます。ディレクトリ部分が指定されていない場合は、”.” が返されます。

例えば、式 $(dirname dir1\dir2\a.out /etc/modules.conf /foo.ml bar.ml) は dir1/dir2 /etc / . と評価されます。

ノート: この関数は dirof 関数と異なります。 dirname 関数は単純に文字列のシーケンスを解析して返すのに対し、 dirof は File オブジェクトを解析する関数です。

10.1.6 rootname

```
$(rootname files) : String Sequence
   files : String Sequence
```

rootname 関数は引数に指定されたファイルのリストから基底の名前を返します。ここでの『基底の名前』とは拡張子が除去されたファイル名のことです。

例えば、式 $(rootname dir1/dir2/a.out /etc/a.b.c /foo.ml) は dir1/dir2/a /etc/a.b /foo と評価されます。

10.1.7 dirof

```
$(dirof files) : Dir Sequence
   files : File Sequence
```

dirof 関数は引数に指定されたファイルのリストからディレクトリを返します。

例えば、式 $(dirof dir/dir2/a.out /etc/modules.conf /foo.ml) は dir1/dir2 /etc / と評価されます。

10.1.8 fullname

```
$(fullname files) : String Sequence
   files : File Sequence
```

fullname 関数は指定した各々のファイルやディレクトリの、プロジェクトルートを元にしたパスを返します。

ノート: 訳注：このような動きをします。

```
% a = foo/bar
- : "foo/bar" : Sequence
% b = $(fullname $a)
- : <data "/your-project/foo/bar"> : String
```

10.1.9 absname

```
$(absname files) : String Sequence
   files : File Sequence
```

absname 関数は指定した各々のファイルやディレクトリの絶対パスを返します。

10.1.10 homename

```
$(homename files) : String Sequence
   files : File Sequence
```

homename 関数は可能であればチルダ ~ を用いてパス名を返します。展開できない形のパスは遅延的に計算されます。具体的にいうと、 homename 関数は通常、始めてチルダ型のパスとして展開できるようになるまでは、絶対パスとして評価します。

10.1.11 suffix

```
$(suffix files) : String Sequence
   files : StringSequence
```

suffix 関数はファイルのリストから拡張子を返します。もし拡張子が存在しなかった場合、空の文字列が返されます。

例えば、式 $(suffix dir1/dir2/a.out /etc/a /foo.ml) は .out .ml と評価されます。

10.2 パスによる検索

10.2.1 which

```
$(which files) : File Sequence
   files : String Sequence
```

which 関数は現在のコマンド検索パスから実行可能なものを検索し、引数に指定されたコマンドのファイルパスを返します。コマンドが見つからない場合にはエラーが発生します。

10.2.2 where

where 関数は which 関数と似ていますが、この関数は与えられた実行コマンドのすべての位置をリストとして返します。この場合ですと echo コマンドは Shell オブジェクトによって内的に扱われるので、出力先の最初の文字列ではビルドイン関数として扱われていることを表しています。

```
% where echo
echo is a Shell object method (a built-in function)
/bin/echo
```

10.2.3 rehash

```
rehash()
```

rehash 関数はすべての検索パスをリセットします。

10.2.4 exists-in-path

```
$(exists-in-path files) : String
   files : String Sequence
```

exists-in-path 関数は引数に指定されたすべてのコマンドが、現在の検索パス上に存在しているかどうか試します。

10.2.5 digest, digest-optional

```
$(digest files) : String Array
   file : File Array
raises RuntimeException
```

```
$(digest-optional files) : String Array
   file : File Array
```

digest と digest-optional 関数はファイルの MD5 を計算します。 digest 関数はファイルが存在しない場合には例外を送出します。一方で、 digest-optional はこの様な場合 false を返します。また、MD5 はキャッシュされます。

10.2.6 find-in-path, find-in-path-optional

```
$(find-in-path path, files) : File Array
   path : Dir Array
   files : String Array
raises RuntimeException

$(find-in-path-optional path, files) : File Array
```

file-in-path 関数は指定されたパスから指定されたファイルを検索します。これらの関数はファイル名が一致した場合のみ処理の対象となります (訳注: 一部ではない)。 find-in-path 関数はファイルが見つからない場合には例外を送出します。一方で、 find-in-path-optional 関数はこの様な場合、例外を送出せずに結果から取り除かれます。

10.2.7 digest-in-path, digest-in-path-optional

```
$(digest-in-path path, files) : String/File Array
   path : Dir Array
   files : String Array
raises RuntimeException

$(digest-in-path-optional path, files) : String/File Array
```

diget-in-path 関数は指定されたパスからファイルを検索し、各々のファイルのファイルパスと MD5 を返します。これらの関数はファイル名が一致した場合のみ処理の対象となります。 digest-in-path 関数はファイルが見つからない場合には例外を送出します。一方で、 digest-in-path-optional 関数はこの様な場合、例外を送出せずに結果から取り除かれます。

10.3 ファイル検査

10.3.1 file-exists, target-exists, target-is-proper

```
$(file-exists files) : String
$(target-exists files) : String
$(target-is-proper files) : String
    files : File Sequence
```

file-exists 関数はリストされているファイルが存在しているかどうか調べます。 target-exists 関数は file-exists と似ていますが、この関数はファイルが存在するか、現在のプロジェクトでターゲットとなっているような場合に true を返します。 target-is-proper 関数は指定されたファイルが現在のプロジェクトで生成できる場合のみ true を返します。

10.3.2 stat-reset

```
$(stat-reset files) : String
    files : File Sequence
```

OMake では Stat によるキャッシュを行っています。 stat-reset 関数は stat から指定されたファイルの情報をリセットし、次回にこれらの情報が要求された場合には stat 情報が強制的に再計算されます。

10.3.3 filter-exists, filter-targets, filter-proper-targets

```
$(filter-exists files) : File Sequence
$(filter-targets files) : File Sequence
$(filter-proper-targets) : File Sequence
   files : File Sequence
```

filter-exists , filter-targets , filter-proper-targets 関数は指定されたファイルのリストをフィルタリングして、特定のファイルのみを返します。

- filter-exists : 存在しているファイルのリストのみが結果として返されます。
- filter-targets : ファイルが存在しているか、現在のプロジェクト上でビルドできるような場合は結果として返されます。
- filter-proper-targets : 現在のプロジェクトでビルドできるファイルのリストのみが結果として返されます。

“distclean” ターゲットを作る

プロジェクトによって生成されたファイルを消去する distclean ルールを作るような場合に、一つの簡単な方法が存在します。それは、現在のプロジェクトでビルドすることのできるすべてのファイルを消去することです。

> **警告:** この方法を何も考えずに実行するのではなく、慎重に考えてください。このルールはプロジェクトで作られるであろう **すべてのファイル** を消去してしまいます。言いかえると、この方法は消去されたファイルのリビルドが成功するかどうかについては、まったく考慮していません。また、この方法では現在のプロジェクトの外にあるファイルは消去されません。

```
.PHONY: distclean

distclean:
    rm $(filter-proper-targets $(ls R, .))
```

もしあなたが CVS を用いているのであれば、CVS に知らせずにすべてのファイルを消去する distclean ルールを作るのではなく、OMake によって作られた cvs_realclean プログラムを作りたいと思うことでしょう。例えば、もしあなたが既によく使われている clean ターゲットを作っていたのなら、そしてあなたが distclean ルールをデフォルトでインタラクティブなものにしたいのであれば、以下のように記述することができます。

```
if $(not $(defined FORCE_REALCLEAN))
    FORCE_REALCLEAN = false
    export

distclean: clean
    cvs_realclean $(if $(FORCE_REALCLEAN), -f) -i .omakedb -i .omakedb.lock
```

あなたは -i オプションを用いることで、いつでも保持していたいより多くのファイル (設定ファイルなど) を追加することができます。

同様に、Subversion を使っていた場合、OMake では build/svn_realclean.om スクリプトを用いて以下のように記述できます。

```
if $(not $(defined FORCE_REALCLEAN))
    FORCE_REALCLEAN = false
    export

open build/svn_realclean

distclean: clean
    svn_realclean $(if $(FORCE_REALCLEAN), -f) -i .omakedb -i .omakedb.lock
```

中間ファイルを除去する別の方法についての詳細は、 dependencies-proper 関数を参照してください。

10.3.4 find-targets-in-path, find-targets-in-path-optional

```
$(find-targets-in-path path files) : File Array
$(find-targets-in-path-optional path, files) : File Array
    path : Dir Array
    files : File Sequence
```

find-target-in-path 関数はパス上のターゲットを検索します。指定された各々の file はディレクトリ dir を順番に検索していき、ターゲット dir/file が存在していた場合はファイル dir/file が返されます。

例えば、あなたは C のプロジェクトをビルドしようとしており、そのプロジェクトはサブディレクトリ src/ を含んでおり、その中には fee.c と foo.c が含まれているものとしましょう。そのような場合、以下の式はたとえそのファイルが存在していなくても、 src/fee.o src/foo.o と評価されます。

```
$(find-targets-in-path lib src, fee.o foo.o)

# このように評価されます。
src/fee.o src/foo.o
```

find-targets-in-path 関数はファイルが見つからない場合には例外を送出します。一方で、find-targets-in-path-optional 関数はこの様な場合、例外を送出せずに結果から取り除かれます。

```
$(find-targets-in-path-optional lib src, fee.o foo.o fum.o)

# このように評価されます。
src/fee.o src/foo.o
```

10.3.5 find-ocaml-targets-in-path-optional

find-ocaml-targets-in-path-optional 関数は find-targets-in-path-optional と似ていますが、この関数はパス上のすべての成分と名前を検索し、最初に小文字のバージョンがビルド可能であるか調べた後で、大文字のバージョンがビルド可能であるか調べる、OCaml スタイルの検索が使われます。

10.3.6 file-sort

```
$(file-sort order, files) : File Sequence
   order : String
   files : File Sequence
```

file-sort 関数はソートルールの集合を用いてビルドのソート順を指定することで、ファイル名のリストをソートします。ソートルールは .ORDER ターゲットを用いて宣言できます。 .BUILDORDER は通常のソート順が定義されています。

```
$(file-sort <order>, <files>)
```

例えば、以下のようなルールの集合を考えてみましょう。

```
a: b c
b: d
c: d

.DEFAULT: a b c d
   echo $(file-sort .BUILDORDER, a b c d)
```

このような場合、 file-sort 関数の結果は d b c a となります。これは依存関係が生成した **後で** ターゲットがソートされることを表しています。この関数は依存関係がリンクされてあるファイルをソートする際 (依存関係が問題となってくる言語を扱う場合)、頻繁に用いられます。

この関数は 3 つの重要な制約があります。

- この関数はルールの内容だけを使ってソートします。理由としては、ソートを実行する前に **すべての** 依存関係を知っておかなければならないからです。
- この関数は現在のプロジェクトでビルド可能なファイルのみをソートします。
- この関数は依存関係が輪状 (cyclic) になっている場合、失敗します。

ソートルール

ソートルールを使用することで、 file-sort 関数にさらなる制限を加えることができます。ソートルールは 2 つの手順を用いて宣言します。まず、ターゲットは .ORDER ターゲットに加えなければなりません。次に、ソートルールの集合が与えられていなければなりません。ソートルールには制限 (pattern constraint) を定義します。

```
.ORDER: .MYORDER

.MYORDER: %.foo: %.bar
.MYORDER: %.bar: %.baz

.DEFAULT: a.foo b.bar c.baz d.baz
   echo $(sort .MYORDER, a.foo b.bar c.baz d.baz)
```

この例では、 .MYORDER ソートルールは、拡張子 .foo の任意のファイルは拡張子 .bar の任意のファイルの後に置く必要があり、さらに拡張子 .bar の任意のファイルは拡張子 .baz の任意のファイルの後に置く必要があることを指定しています。

この例では、ソート結果は d.baz c.baz b.bar a.foo となります。

10.3.7 file-check-sort

```
file-check-sort(files)
   files : File Sequence
raises RuntimeException
```

file-check-sort 関数はファイルのリストがソート順になっているかどうか調べます。もしそうならば、リストは変更されずに返されます。そうでない場合は、この関数は例外を送出します。

```
$(file-check-sort <order>, <files>)
```

10.4 ファイルの検索とリスト

OMake のコマンドは実行される前に『glob 展開』が行われます。これは、ファイル名にはディレクトリやファイル名のシーケンスに展開された **パターン** を含めることができることを意味しています。構文は標準的な bash(1), csh(1) や以下のルールに従っています。

- パス名は / か \ の文字によって分割された、ディレクトリやファイル名から成るシーケンスです。例えば、2 つのパス名 /home/jyh/OMakefile , /home\jyh/OMakefile は同一のファイルを表しています。
- Glob 展開はパスの文字を元にして実行されます。もしパスが下記の特殊文字を含んでいた場合、パスはシステム上の実際にあるファイルに対してマッチしているものだけがリストされます。この展開によって、パターン文字はマッチしたすべてのファイルやディレクトリのシーケンスに展開されます。

以下、ディレクトリ /dir にはファイル a , -a , a.b , b.c が含まれているものとします。

- * : 0 以上の文字を持つ任意のシーケンスにマッチします。例えば、パターン /dir/a* は /dir/a /dir/aa /dir/a.b に展開されます。
- ? : 任意の一つの文字にマッチします。パターン /dir/?a は /dir/-a に展開されます。
- [...] : 大括弧の中には文字の集合や、ASCII 文字の範囲を指定します。パターンには独立な文字 c や文字の範囲 c1-c2 を含めます。パターンのマッチには任意の文字や任意の範囲を指定することができます。 ^ をつけることで与えられたパターンを反転 (指定した文字を含んでいないとマッチする) できます。また、パターンの中に - を含めたい場合、パターンの最初の文字に - と指定しなければなりません。

パターン	展開
/dir/[a-b]*	/dir/a /dir/a.b /dir/b.c
dir/[-a-b]*	/dir/a /dir/-a /dir/a.b /dir/b.c
/dir/[-a]*	/dir/a /dir/-a /dir/a.b

- {s1,...,sN} : 中括弧の中にはコンマによって分割された文字列のシーケンスを指定します。N 個の文字列が与えられていた場合、結果は N 個のパターンのコピーが生成されて、各々の文字列は si となります。

パターン	展開
a{b,c,d}	ab ac ad
a{b{c,d},e}	abc abd ae
a{?{[A-Z],d},*}	a?[A-Z] a?d a*

チルダ (~) はホームディレクトリを指定する際に用いられます。この値はあなたのシステムに依存して展開されます。

パターン	展開
~jyh	/home/jyh
~bob/*.c	c:\Documents and Settings\users\bob

\ 文字はパス名のセパレータとしても、文字をエスケープするのにも使われます。もし特殊 blob 文字の前に用いられていた場合、 \ は次の文字を一種の特殊でない文字に変形します。さもなければ、 \ はパス名のセパレータとして解釈されます。

パターン	展開
~jyh/*	~jyh/* (* は文字通り扱われる)
/dir/\[a-z?	/dir/[a-z? ([は文字通り、 ? はパターンとして扱われる)
c:\Program Files\[A-z]	c:\Program Files[A-z]*

ノート: 最後の \ 文字に関するケースは非常に曖昧です。 \ はパス名のセパレータとして扱うべきで、[をエスケープするために用いるのではありません。もしあなたが Win32 上でこのような解釈を避けたいとするならば、たとえ Win32 のパス名であっても / を用いるべきです (/ は \ に変換されて出力されます)。

パターン	展開
c:/Program Files/[A-z]*	c:\Program Files\WindowsUpdate ...

10.4.1 glob

```
$(glob strings) : Node Array
   strings : String Sequence
$(glob options, strings) : Node Array
   options : String
   strings : String Sequence
```

glob 関数は glob 展開を行います。

. と .. エントリは常に無視されます。

オプションは以下です。

- **b**

  csh(1) スタイルの大括弧展開を行いません。

- **e**

  \ 文字を特殊文字にエスケープしません。

- **n**

  展開が失敗した場合、無視する代わりに展開を文字通り返します。

- **i**

  展開が失敗した場合、何も行いません。

- **.**

  . から始まるファイルにマッチする、ワイルドカードパターンを許可します。

- **A**

  . から始まるファイルを含んだ、すべてのファイルを返します。

- **F**

  通常のファイルのみ (ディレクトリでない任意のファイル) にマッチします。

- **D**

  ディレクトリファイルのみにマッチします。

- **C**

  cvs(1) ルールに関するファイルを無視します。

- **P**

  適切なサブディレクトリのみを含みます。

加えて、以下の変数が glob の動作に影響を与えるように定義されています。

- **GLOB_OPTIONS**

  デフォルトのオプションを定義している文字列

- **GLOB_IGNORE**

  glob が無視すべき、ファイル名のシェルパターンのリスト

- **GLOB_ALLOW**

  シェルパターンのリスト。ファイルが GLOB_ALLOW のパターンにマッチしなかった場合、そのファイルは無視されます。

また、返されるファイルは名前順でソートされます。

10.4.2 ls

```
$(ls files) : Node Array
   files : String Sequence
$(ls options, files) : Node Array
   files : String Sequence
```

ls 関数はディレクトリからファイル名を返します。

. と .. エントリは常に無視されます。パターンにはシェルライクなパターンを指定することで、 ls 関数は glob 展開を行います。

オプションは glob 関数のオプションをすべて含んでいるほかに、以下のオプションが含まれています。

- **R** : ディレクトリを再帰的に走査し、リストします。

GLOB_ALLOW と GLOB_IGNORE 変数は glob 検索の動作を制御するために定義されています。また、返り値は名前によってソートされます。

10.4.3 subdirs

```
$(subdirs dirs) : Dir Array
   dirs : String Sequence
$(subdirs options, dirs) : Dir Array
   options : String
   dirs : String Sequence
```

subdirs 関数はディレクトリのリストに存在している、すべてのサブディレクトリを再帰的に返します。

とりうることのできるオプションは以下に定義されています。

- **A** : a から始まるディレクトリを返します。
- **C** : .cvsignore ルールに基づいて、特定のファイルを無視します。
- **P** : 適切なサブディレクトリのみを含めます。

10.5 ファイル操作

10.5.1 mkdir

```
mkdir(mode, node...)
   mode : Int
   node : Node
raises RuntimeException

mkdir(node...)
   node : Node
raises RuntimeException
```

mkdir 関数はディレクトリかディレクトリの集合を生成します。以下のオプションがサポートされています。

- **-m** mode : 作られるディレクトリのパーミッションを指定します。
- **-p** : もし親のディレクトリが存在しない場合は、新しく生成します。
- : この文字の後にくる文字列を、たとえ特殊文字が含まれていても文字通りに解釈します。

10.5.2 Stat

stat や lstat 関数によって返される Stat オブジェクトはファイルシステムノードについての情報を提供します。このオブジェクトは以下のフィールドを含んでいます。

- **dev** : デバイス番号
- **ino** : inode 番号
- **king** : ファイルの種類を表し、以下の内の一つが指定されています: REG (通常のファイル), DIR (ディレクトリ), CHR (キャラクタデバイス), BLK (ブロックデバイス), LNK (シンボリックリンク), FIFO (名前付きパイプ), SOCK (ソケット)
- **perm** : アクセス権。数値で表現されます。
- **nlink** : リンクの数
- **uid** : オーナーのユーザー ID
- **gid** : ファイルのグループのグループ ID
- **rdev** : デバイスのマイナー番号
- **size** : ファイルのバイト数
- **atime** : アクセス日時。浮動小数点で表現されます。
- **mtime** : 修正日時。浮動小数点で表現されます。
- **ctime** : ステータス変更日時。浮動小数点で表現されます。

すべてのフィールドがすべての OS 上で意味をもつわけではない点に注意してください。

10.5.3 stat, lstat

```
$(stat node...) : Stat
   node : Node or Channel
$(lstat node...) : Stat
```

```
   node : Node or Channel
raises RuntimeException
```

stat 関数はファイル情報を返します。もしファイルがシンボリックリンクであった場合は、 stat 関数はリンク先を参照します。一方で、 lstat 関数はリンク自身を参照します。

10.5.4 unlink

```
$(unlink file...)
   file : File
#(rm file...)
   file : File
$(rmdir dir...)
   dir : Dir
raises RuntimeException
```

unlink と rm 関数はファイルを削除します。 rmdir 関数はディレクトリを削除します。

rm および rmdir 関数は以下のオプションをサポートしています。

- **-f** : 存在しないファイルを確認なしで無視します。
- **-i** : 消去する前に確認します。
- **-r** : ディレクトリの内容を再帰的に削除します。
- **-v** : どのような処理を行っているのかを出力します。
- **-** : この文字の後に来る値は文字通りに解釈されます。

10.5.5 rename

```
rename(old, new)
   old : Node
   new : Node
mv(nodes... dir)
   nodes : Node Sequence
   dir   : Dir
cp(nodes... dir)
   nodes : Node Sequence
   dir   : Dir
raises RuntimeException
```

rename 関数はファイルかディレクトリの名前を old から new に変更します。

mv 関数は似ていますが、もし new がディレクトリでかつ存在している場合、シーケンスによって指定されたファイルはディレクトリの中に移動されます。そうでない場合、 mv の動作は rename と全く同じです。 cp 関数は似ていますが、この関数はオリジナルのファイルを消去しません。

mv と cp 関数は以下のオプションをとります。

- **-f** : 上書きする前の確認を行いません。
- **-i** : 上書きする前に確認します。
- **-v** : どのような処理を行っているのか出力します。
- **-r** : ディレクトリの内容を再帰的にコピーします。
- **-** : この文字の後に来る引数は文字通りに解釈されます。

10.5.6 link

```
link(src, dst)
   src : Node
   dst : Node
raises RuntimeException
```

link 関数はファイルやディレクトリ src へのハードリンクを dst に生成します。

ハードリンクは NTFS を用いていた場合は、Win32 システム上でも正常に動作します。

通常、スーパーユーザだけがディレクトリへのハードリンクを生成できます。

10.5.7 symlink

```
symlink(src, dst)
   src : Node
   dst : Node
raises RuntimeException
```

symlink 関数は src ファイルへのシンボリックリンクを dst に生成します。

リンク名はターゲットのディレクトリに合わせて計算されます。例えば、式 $(symlink a/b, c/d) は c/d -> ../a/b というリンクが生成されます。

シンボリックリンクは Win32 上ではサポートされていません。もしクロスプラットフォームでリンク/コピーを行いたい場合は ln-or-cp か Shell エイリアスを使用してください。

10.5.8 readlink

```
$(readlink node...) : Node
   node : Node
```

readlink 関数はシンボリックリンクの値を読み取ります。

10.5.9 chmod

```
chmod(mode, dst...)
   mode : Int
   dst : Node or Channel
chmod(mode dst...)
   mode : String
   dst : Node Sequence
raises RuntimeException
```

chmod 関数は対象のパーミッションを変更します。

オプション:

- **-v** : どのような処理を行っているのか説明します。
- **-r** : ファイルやディレクトリを再帰的に変更します。
- **-f** : エラーが生じても続けて実行します。
- **-** : 残りの引数を文字通りに解釈します。

10.5.10 chown

```
chown(uid, gid, node...)
   uid : Int
   gid : Int
   node : Node or Channel
chown(uid, node...)
   uid : Int
   node : Node or Channel
raises RuntimeException
```

chown 関数はファイルのユーザやグループ ID を変更します。もし gid が指定されていない場合は、値は変更されません。ID が-1 であった場合でも、その ID は変更されません。

10.5.11 truncate

```
truncate(length, node...)
    length : Int
    node : Node or Channel
raises RuntimeException
```

truncate 関数はファイルを与えられた長さに切り詰めます。

10.5.12 umask

```
$(umask mode) : Int
   mode : Int
raises RuntimeException
```

ファイル生成マスクを設定します。この関数は前回のマスクの値が返されます。この値はスコープ化されていませんので、この変更はグローバルに影響を及ぼします。

10.6 vmount

10.6.1 vmount

```
vmount(src, dst)
   src, dst : Dir
vmount(flags, src, dst)
   flags : String
   src, dst : Dir
```

src ディレクトリを dst ディレクトリに『マウント』します。これは仮想的なマウントで、 $(file ...) 関数のふるまいを変更します。 $(file str) が使われた場合、返される値はもしファイルが存在していたら src ディレクトリに関連付けられます。さもなければファイルは現在のディレクトリに関連付けられます。

vmount 関数の主な目的は、分割された設定やアーキテクチャを用いて、複数のビルドを行うことをサポートするためです。

オプションは以下の通りです。

- **l** : src ディレクトリ中のファイルへのシンボリックリンクを生成します。
- **c** : src ディレクトリからファイルをコピーします。

なお、マウント操作はスコープ化されています。

10.6.2 add-project-directories

```
add-project-directories(dirs)
   dirs : Dir Array
```

omake がプロジェクトの一部とみなしているディレクトリの集合に、ディレクトリを新しく追加します。これは主に、現在のディレクトリがプロジェクトの一部でないと omake がエラーを出すことを避けるために用いられます。

10.6.3 remove-project-directories

```
remove-project-directories(dirs)
   dirs : Dir Array
```

omake がプロジェクトの一部とみなしているディレクトリの集合から、ディレクトリを削除します。これは主に、特定のディレクトリがコンパイルされる必要がないことが分かっているので、インクルードしているディレクトリから .SUBDIRS を取り消すような場合に用いられます。

10.7 ファイルの内容を元にした検索

10.7.1 test

```
test(exp) : Bool
   exp : String Sequence
```

評価式の文法は以下の通りです。

- ！exp : exp は真でない
- exp1 -a exp2 : 両方の exp が真である
- exp1 -o exp2 : 1 以上の exp が真である
- (exp) : exp は真である

基本となる評価式は以下の通りです。

- -n str : **文字列** は 0 でない長さである
- -z str : **文字列** は 0 の長さである
- str = str : 両方の文字列は等しい
- str != str : 両方の文字列は等しくない
- int1 -eq int2 : 両方の整数は等しい
- int1 -ne int2 : 両方の整数は等しくない
- int1 -gt int2 : int1 は int2 よりも大きい (int1 > int2)
- int1 -ge int2 : int1 は int2 以上である (int1 >= int2)
- int1 -gt int2 : int1 は int2 よりも小さい (int1 < int2)
- int1 -le int2 : int1 は int2 以下である (int1 <= int2)

- file1 -ef file2 : Unix 上では、file1 と file2 は同一のデバイスと inode 番号である。Win32 上では、file1 と file2 は同一の名前である。
- file1 -nt file2 : file1 は file2 よりも新しい
- file1 -ot file2 : file1 は file2 よりも古い
- -b file : file はブロック型特殊ファイルである
- -c file : file はキャラクタ型特殊ファイルである
- -d file : file はディレクトリである
- -e file : file が実在している
- -f file : file は通常のファイルである
- -g file : set-group-id bit が file に設定されている
- -G file : file のグループ ID が現在影響を受けているグループである
- -h file : file がシンボリックリンクである (-L でもよい)
- -k file : file にスティッキー・ビットが設定されている
- -L file : file がシンボリックリンクである (-h でもよい)
- -O file : file のオーナーが現在影響を受けているグループである
- -p file : file が名前付きパイプである
- -r file : file が読み込み可能である
- -s file : file は空である
- -S file : file はソケットである
- -u file : set-user-id bit が file に設定されている
- -w file : file は書き込み可能である
- -x file : file は実行可能である

str は任意の文字列で、 - 文字をつけることができます。

int は整数として解釈できる文字列です。従来の test プログラムのバージョンとは異なり、先頭の文字にはアリティを指定することもできます。接頭辞 0b は数値をバイナリで扱います。同様に、接頭辞 0o は数値を 8 進数で扱い、接頭辞 0x は数値を 16 進数で扱います。 *int* は -l を用いることで数値を文字列として扱うことができます。これによって、数値ではなく文字列の長さが評価されます。

file はファイル名を表す文字列です。

構文は test(1) プログラムと似ています。もしあなたが Unix のシステム上にいるのならば、man を参照すればさらに詳細を説明してくれるでしょう。以下にいくつかのサンプルを挙げておきます。

```
# 空のファイルを作成
osh> touch foo
# ファイルは空ですか？
osh> test(-e foo)
- : true
osh> test(! -e foo)
- : false
# 別のファイルを作成
osh> touch boo
# 新しく作ったファイルは前よりも新しいですか？
osh> test(boo -nt foo)
- : true
```

```
# さらに複雑な式を示します。
# boo は foo よりも新しく、さらに foo は空である。
osh> test(\( boo -nt foo \) -a -e foo)
- : true
```

10.7.2 find

```
find(exp) : Node Array
   exp : String Sequence
```

find 関数はディレクトリを再帰的に検索し、 exp の評価が真であるファイルを返します。

引数の exp は以下の例外を除いて、 test 関数と同じ構文を用いています。

1. exp はディレクトリから始まります。もし特に指定していない場合はカレントディレクトリから検索します。
2. {} 文字は現在のファイルを表す文字に拡張されています。

exp の構文は test 関数のそれと同一ですが、以下の点が加わっています。

- -name str : 現在のファイルは glob 表現にマッチしている (詳細は "**ファイルの検索とリスト**" を参照してください)。
- -regex str : 現在のファイルは正規表現にマッチしている。

find 関数はすべてのサブディレクトリを再帰的にスキャンします。以下の関数呼び出しは omake のソースディレクトリのルートから実行させています。

```
osh> find(. -name fo* )
- : <array
        /home/jyh/.../omake/mk/.svn/format
        /home/jyh/.../omake/RPM/.svn/format
        ...
        /home/jyh/.../omake/osx_resources/installer_files/.svn/format>
```

別の例では、通常のファイルかシンボリックリンクのファイルのみを並べています。

```
osh> find(. -name fo* -a \( -f {} -o -L {} \))
- : <array
        /home/jyh/.../omake/mk/.svn/format
        /home/jyh/.../omake/RPM/.svn/format
        ...
        /home/jyh/.../omake/osx_resources/installer_files/.svn/format>
```

10.8 I/O 関数

10.8.1 標準出力先

以下の変数が標準出力先として定義されています。

- **stdin**

  ```
  stdin : InChannel
  ```

  標準入力チャネルで、読み込みを担当します。

- **stdout**

  ```
  stdout : OutChannel
  ```

  標準出力チャネルで、書き込みを担当します。

- **stderr**

  ```
  stderr : OutChannel
  ```

  標準エラーチャネルで、書き込みを担当します。

10.8.2 open-in-string

open-in-string 関数は文字列をまるでファイルのように扱い、読み込むためのチャネルを返します。

```
$(open-in-string s) : Channel
    s : String
```

10.8.3 open-out-string, out-contents

open-out-string 関数はファイルの代わりに文字列を書き込むチャネルを作成します。文字列は out-contents 関数を用いて取得することができます。

```
$(open-out-string) : Channel
$(out-contents chan) : String
    chan : OutChannel
```

10.8.4 fopen

fopen 関数はファイルを読み書きするために、新しくファイルを開きます。

```
$(fopen file, mode) : Channel
   file : File
   mode : String
```

file は読み書きするファイル名を指定します。 mode は以下の文字を組み合わせた文字列です。

- **r** : 読み込むためにファイルを開きます。もしファイルが存在していない場合はエラーが発生します。
- **w** : 書き込むためにファイルを開きます。もしファイルが存在していない場合は新しくファイルを作ります。
- **a** : 追記モードでファイルを開きます。もしファイルが存在していない場合は新しくファイルを作ります。
- **+** : 読み書き両方するためにファイルを開きます。
- **t** : ファイルをテキストモードで開きます (デフォルト)。
- **b** : ファイルをバイナリモードで開きます。
- **n** : ファイルをノンブロッキングモードで開きます。
- **x** : ファイルが既に存在している場合は失敗します。

Unix システム上ではバイナリモードは意味を持たず、テキストモードとバイナリモードは等価に扱われます。

10.8.5 close

```
$(close channel...)
   channel : Channel
```

close 関数は以前に fopen によって開かれたファイルを閉じる関数です。

10.8.6 read, input-line

```
$(read channel, amount) : String
$(input-line channel) : String
   channel : InChannel
   amount  : Int
raises RuntimeException
```

read 関数は amount バイトを入力チャネルから読み込み、読み込んだデータを返します。 input-line 関数はファイルから一行を読み込み、読み込んだ一行を改行コード抜きで返します。もしファイルの終わりまでたどり着いた場合、両方の関数は例外 RuntimeException を送出します。

10.8.7 write

```
$(write channel, buffer, offset, amount) : String
   channel : OutChannel
   buffer  : String
   offset  : Int
   amount  : Int
$(write channel, buffer) : String
   channel : OutChannel
   buffer  : String
raises RuntimeException
```

4 つ引数をとる場合、 write 関数は出力チャネル channel に位置 offset から、バイト buffer を書き込みます。なお、上限は amount バイトです。この関数は何バイトが書き込まれたのかというバイト数を返します。

3 つ引数をとる場合も同様ですが、 offset は 0 となります。

2 つ引数をとる場合、 offset は 0 で、かつ amount は buffer の長さになります。

ファイルの終わりまでたどり着いた場合、この関数は例外 RuntimeException を送出します。

10.8.8 lseek

```
$(lseek channel, offset, whence) : Int
   channel : Channel
   offset  : Int
   whence  : String
raises RuntimeException
```

lseek 関数はチャネル channel のオフセット位置を whence に基づいて変更します。 whence は以下の通りです。

- **SEEK_SET** : オフセットは offset に設定されます。
- **SEEK_CUR** : オフセットは現在の位置から offset バイト分移動します。

- **SEEK_END** : オフセットはファイルサイズ + offset バイトの位置に設定されます。

lseek 関数はファイルの新しいオフセット位置を返します。

10.8.9 rewind

```
rewind(channel...)
   channel : Channel
```

rewind 関数は現在のファイル位置をファイルが始まる位置に設定します。

10.8.10 tell

```
$(tell channel...) : Int...
   channel : Channel
raises RuntimeException
```

tell 関数は channel の現在位置を返します。

10.8.11 flush

```
$(flush channel...)
   channel : OutChannel
```

flush 関数は書き込み用途にファイルが開かれている場合のみに使われます。この関数はファイルにまだ書き込まれていないすべてのデータを消去します。

10.8.12 channel-name

```
$(channel-name channel...) : String
   channel : Channel
```

channel-name 関数はチャネルに関係している名前を返します。

10.8.13 dup

```
$(dup channel) : Channel
   channel : Channel
raises RuntimeException
```

dup 関数は引数に指定されたチャネルと同一のファイルを示している、新しいチャネルを返します。

10.8.14 dup2

```
dup2(channel1, channel2)
   channel1 : Channel
   channel2 : Channel
raises RuntimeException
```

dup2 関数は channel2 を channel1 と同一のファイルを示すようにします。

10.8.15 set-nonblock

```
set-nonblock-mode(mode, channel...)
   channel : Channel
   mode : String
```

set-nonblock-mode 関数は与えられたチャネルのノンブロッキングフラグを設定します。もしチャネルが入出力中で、この操作が即座に完了しないような場合、 RuntimeException が送出されます。

10.8.16 set-close-on-exec-mode

set-close-on-exec-mode 関数は与えられたチャネルの close-on-exec フラグを設定します。もし close-on-exec フラグが設定されている場合、このチャネルは子プロセスから継承されません。そうでない場合は、このチャネルは子プロセスから継承されます。

10.8.17 pipe

```
$(pipe) : Pipe
raises RuntimeException
```

pipe 関数は 2 つのフィールドを持つ Pipe オブジェクトを生成します。 read フィールドは読み込むためのチャネルで、 write フィールドは書き込むためのチャネルです。

10.8.18 mkfifo

```
mkfifo(mode, node...)
   mode : Int
   node : Node
```

mkfifo 関数は名前付きパイプを生成します。

10.8.19 select

```
$(select rfd..., wfd..., efd..., timeout) : Select
   rfd : InChannel
   wfd : OutChannel
   efd : Channel
   timeout : float
raises RuntimeException
```

select 関数は与えられたチャネルの集合が入出力可能であるか監視します。 rfd には読み込み可能なチャネルのシーケンスを指定します。 wfd には書き込み可能なチャネルのシーケンスを指定します。 efd にはエラー状態を監視するチャネルのシーケンスを指定します。 timeout にはイベントを待つ最大時間を指定します。

正常な戻り値の場合、 select 関数は以下のフィールドを持つ Select オブジェクトを返します。

- **read** : 読み込み可能なチャネルの配列
- **write** : 書き込み可能なチャネルの配列
- **error** : エラーが生じたチャネルの配列

10.8.20 lockf

```
lockf(channel, command, len)
   channel : Channel
   command : String
   len : Int
raises RuntimeException
```

lockf 関数は与えられたチャネルの POSIX ロックの範囲を設定します。範囲は現在の位置から len バイトまでです。

command のとりうる値は以下の通りです。

- **F_ULOCK** : 指定された範囲をアンロックします。
- **F_LOCK** : 指定された範囲を書き込みロックします。既にロックされている場合はブロックされます。
- **F_TLOCK** : 指定された範囲を書き込みロックします。既にロックされている場合は失敗します。
- **F_TEST** : 指定された範囲に他のロックがないか試します。
- **F_RLOCK** : 指定された範囲を読み込みロックします。既にロックされている場合はブロックされます。
- **F_TRLOCK** : 指定された範囲を読み込みロックします。既にロックされている場合は失敗します。

10.8.21 InetAddr

InetAddr オブジェクトはインターネットアドレスを記述しています。このオブジェクトは以下のフィールドを含んでいます。

- **addr** -> String : インターネットアドレス
- **port** -> Int : ポート番号

10.8.22 Host

Host オブジェクトは以下のフィールドを含んでいます。

- **name** -> String : ホスト名
- **aliases** -> String Array : ホストの別名の配列
- **addrtype** -> String : より好ましいドメイン・ソケット
- **addrs** -> InetAddr Array : ホストに所属しているインターネットアドレスの配列

10.8.23 gethostbyname

```
$(gethostbyname host...) : Host...
   host : String
raises RuntimeException
```

gethostbyname 関数は指定されたホストの Host オブジェクトを返します。 host にはドメイン名かインターネットアドレスを指定します。

10.8.24 Protocol

Protocol オブジェクトはプロトコルエントリーを表現します。このオブジェクトは以下のフィールドを含んでいます。

- **name** -> String : 正規のプロトコル名
- **aliases** -> String Array : プロトコルのエイリアスの配列
- **proto** -> Int : プロトコル番号

10.8.25 getprotobyname

```
$(getprotobyname name...) : Protocol...
   name : Int or String
raises RuntimeException
```

getprotobyname 関数は指定されたプロトコルから Protocol オブジェクトを返します。 name にはプロトコル名かプロトコル番号を指定します。

10.8.26 Service

Service オブジェクトはネットワークサービスを表現します。このオブジェクトは以下のフィールドを含んでいます。

- **name** -> String : サービス名
- **aliases** -> String Array : サービスのエイリアスの配列
- **port** -> Int : サービスのポート番号
- **proto** -> Protocol : サービスのプロトコル

10.8.27 getservbyname

```
$(getservbyname service...) : Service...
   service : String or Int
raises RuntimeException
```

getservbyname 関数はネットワークサービスの情報を取得します。 service にはサービス名か番号を指定します。

10.8.28 socket

```
$(socket domain, type, protocol) : Channel
   domain : String
   type : String
   protocol : String
raises RuntimeException
```

socket 関数は束縛されていないソケットを生成します。

引数のとりうる値は以下の通りです。

domain は以下の値をとります。

- **PF_UNIX** あるいは **unix** : Unix システムでのみ利用可能な Unix ドメイン
- **PF_INET** あるいは **inet** : IPv4 インターネットドメイン
- **PF_INET6** あるいは **inet6** : IPv6 インターネットドメイン

`type` は以下の値をとります。

- **SOCK_STREAM** あるいは **stream** : ストリームソケット
- **SOCK_DGRAM** あるいは **dgram** : データグラムソケット
- **SOCK_RAW** あるいは **raw** : 生 (raw) ソケット
- **SOCK_SEQPACKET** あるいは **seqpacket** : パケットシーケンスソケット

`protocol` はプロトコルのデータベースにあるプロトコルを指定した `Int` か `String` 型の引数です。

10.8.29 bind

```
bind(socket, host, port)
   socket : InOutChannel
   host : String
   port : Int
bind(socket, file)
   socket : InOutChannel
   file : File
raise RuntimeException
```

`bind` 関数はソケットをアドレスに束縛します。

3 つ引数をとる場合、 `bind` 関数はインターネットの接続方法について指定し、 `host` にはホスト名か IP アドレスを、 `port` にはポート番号を指定します。

2 つ引数をとる形は `Unix` ソケットのために用意されています。 `file` にはファイル名のアドレスを指定します。

10.8.30 listen

```
listen(socket, requests)
   socket : InOutChannel
   requests : Int
raises RuntimeException
```

`listen` 関数は `requests` 個のまだ取得されていないリクエストを取得するように、ソケットを設定します。

10.8.31 accept

```
$(accept socket) : InOutChannel
   socket : InOutChannel
raises RuntimeException
```

`accept` 関数はソケットの接続を受け入れます。

10.8.32 connect

```
connect(socket, addr, port)
   socket : InOutChannel
   addr : String
   port : int
connect(socket, name)
   socket : InOutChannel
   name : File
raise RuntimeException
```

connect 関数はソケットを対象のアドレスに接続します。

3 つ引数をとる場合、 connect 関数はインターネットの接続方法について指定します。 addr にはリモートホストのインターネットアドレスをドメイン名か IP アドレスの形で指定します。 port にはポート番号を指定します。

2 つ引数をとる形は Unix ソケットのために用意されています。 name にはソケットのファイル名を指定します。

10.8.33 getchar

```
$(getc) : String
$(getc file) : String
   file : InChannel or File
raises RuntimeException
```

getc 関数はファイルの次の文字を返します。もし引数が指定されなかった場合、 stdin が入力として用いられます。もしファイルの終わりまでたどりついた場合、この関数は false を返します。

10.8.34 gets

```
$(gets) : String
$(gets channel) : String
   channel : InChannel or File
raises RuntimeException
```

gets 関数はファイルから次の行を返します。この関数はファイルの終わりまでたどり着いていた場合、空の文字列を返します。改行コードは取り除かれます。

10.8.35 fgets

```
$(fgets) : String
$(fgets channel) : String
   channel : InChannel or File
raises RuntimeException
```

fgets 関数は fopen によって読み込みが開かれているファイルから、次の行を返します。この関数はファイルの終わりまでたどり着いていた場合、空の文字列を返します。返される文字列は文字通りのデータ (literal data) として返されます。改行コードは取り除かれません。

10.9 出力関数

print と println 関数を用いて出力を表示します。 println 関数は表示する値に新しく改行コードを加えます。 print 関数は加えません。

```
fprint(<file>, <string>)
print(<string>)
eprint(<string>)
fprintln(<file>, <string>)
println(<string>)
eprintln(<string>)
```

fprint 関数は fopen で開かれたファイルに対して出力します。 print 関数は標準出力チャネルに対して出力しますが、 eprint 関数は標準エラーチャネルに対して出力します。

10.10 値を出力する関数

値は printv と printvln 関数を用いて表示できます。 printvln 関数は表示する値に新しく改行コードを加えます。 printv 関数は加えません。

```
fprintv(<file>, <string>)
printv(<string>)
eprintv(<string>)
fprintvln(<file>, <string>)
printvln(<string>)
eprintvln(<string>)
```

fprinv 関数は fopen で開かれたファイルに対して出力します。 printv 関数は標準出力チャネルに対して出力しますが、 eprintv 関数は標準エラーチャネルに対して出力します。

10.10.1 その他の関数

set-channel-line

```
set-channel-line(channel, filename, line)
    channel : Channel
    filename : File
    line : int
```

指定されたチャネルの行番号情報を設定します。

10.11 高レベルな I/O 関数

10.11.1 正規表現

多くの高レベルな関数では正規表現を使用しています。正規表現は awk(1) の構文とだいたい似ている文字列から成り立っています。

文字列には以下の文字定数を含めることができます。

- \\ : バックスラッシュ文字

- \a : アラート文字 ^G
- \b : バックスペース文字 ^H
- \f : 改ページ文字 ^L
- \n : 改行文字 ^J
- \r : キャリッジリターン (復帰,CR) 文字 ^M
- \t : タブ文字 ^I
- \v : 垂直タブ文字
- \xhh... : 16 進数として表現される文字列 h 。すべての正しい 16 進数文字列はシーケンス文字列の一部として扱われます。
- \ddd : 1〜3 桁の 8 進数として扱われる文字列

正規表現は特殊文字 .\^$[(){}*?+ を使うことができます。

- c : c が特殊文字でない場合は文字通りに扱います。
- \c : c が特殊文字であっても、 c を文字通りに扱います。
- . : 改行を含む、任意の文字にマッチします。
- ^ : 行の始めにマッチします。
- $: 行の終わりにマッチします。
- [abc...] : abc... の任意の文字にマッチします。
- [^abc...] : abc... を除く、任意の文字にマッチします。
- r1|r2 : r1 と r2 のいづれかであった場合はマッチします。
- r1r2 : r1 の次に r2 が来ていた場合はマッチします。
- r+ : 1 以上の r の文字列にマッチします。
- r* : 0 以上の r の文字列にマッチします。
- r? : 0 もしくは 1 つの r にマッチします。
- (r) : r にマッチします。括弧はグループ化のために用いられます。
- \(r\) : グループとして扱われますが、括弧の中でマッチした式は変数 $1 , $2 , ... を通して参照することができます。
- r{n} : r が正確に n 回続いている場合はマッチします。
- r{n,} : r が n 回以上続いている場合はマッチします。
- r{n,m} : r が n 回以上 m 回以下続いている場合はマッチします。
- \y : 単語の始まりあるいは単語が終わった後の空文字にマッチします。
- \B : 単語の中にある空文字にマッチします。
- \< : 単語の始まりの空文字にマッチします。
- \> : 単語が終わった後のから文字にマッチします。
- \w : 単語の任意の文字にマッチします。
- \W : 単語の中に入っていない、任意の文字にマッチします。
- \~ : ファイルの始まりの空文字にマッチします。
- \' : ファイルの終わりの空文字にマッチします。

上の文字クラスは文字のシーケンスを絶対的に指定するのに用いられます。これらのシーケンスはあなたの言語環境次第で変更することもできます。

- [:alnum:] : 英数字文字
- [:alpha:] : アルファベット文字
- [:lower:] : アルファベット小文字
- [:upper:] : アルファベット大文字
- [:cntrl:] : 制御文字
- [:digit:] : 数字
- [:xdigit:] : 数字あるいは 16 進数文字
- [:graph:] : 出力可能でかつ表示可能な文字
- [:print:] : 出力可能で、表示可能あるいは不可能な文字
- [:punct:] : 句読点文字
- [:blank:] : スペースかタブ文字
- [:space:] : ホワイトスペース文字

10.11.2 cat

```
cat(files) : Sequence
   files : File or InChannel Sequence
```

cat 関数は複数のファイルを連結し、文字列として返します。

10.11.3 grep

```
grep(pattern) : String  # stdin から入力されて、デフォルトのオプションが用いられる
   pattern : String
grep(pattern, files) : String  # デフォルトのオプションが用いられる
   pattern : String
   files   : File Sequence
grep(options, pattern, files) : String
  options : String
  pattern : String
  files   : File Sequence
```

grep 関数はファイルの集合から正規表現 pattern に適合しているものを検索し、マッチした行を表示します。これは grep(1) の高度に簡素化されたバージョンです。

オプションは以下の通りです。

- **q** : 指定した場合、 grep からの出力は表示されません。
- **h** : 指定した場合、出力された行はファイル名を含めません (一つの入力ファイルだけが与えられた場合のデフォルト)。
- **n** : 指定した場合、出力された行はファイル名を含めます (複数の入力ファイルが与えられた場合のデフォルト)。
- **v** : 指定した場合、マッチした行の代わりにマッチしなかった行を出力します。

pattern は正規表現です。

もし成功した (grep がマッチした行を見つけた) 場合、この関数は true を返します。そうでない場合は false を返します。

10.11.4 scan

```
scan(input-files)
case string1
   body1
case string2
   body2
...
default
   bodyd
```

scan 関数はコマンドライン上からの入力機能を提供します。この関数はファイル、あるいはファイル名を引数にとります。もし何も引数が呼ばれなかった場合、 stdin から入力されます。もし引数が指定された場合、各々の引数には InChannel が指定されるか、入力としてファイル名が用いられます。なお、出力は常に stdout です。

scan 関数は入力から一行を読み込み、以下のアルゴリズムに従って処理を行います。

各々の行で、レコードはまずいくつかのフィールドに分割されて、それらのフィールドは変数 $1, $2, ... に束縛されます。変数 $0 は行全体として定義されており、 $* はすべてのフィールドの値が定義されている配列です。 $(NF) 変数はフィールドの数が定義されています。

次に case 文が実行されます。もし string_i がトークン $i にマッチした場合、 body_i が評価されます。もし case の内容が export で終わっていたのなら、現在の状態は次の宣言句へ受け継がれます。そうでない場合、この値は捨てられます。

例えば、以下の scan 関数は単純なコマンドプロセッサのように振る舞います。

```
calc() =
   i = 0
   scan(script.in)
   case print
      println($i)
   case inc
      i = $(add $i, 1)
      export
   case dec
      i = $(sub $i, 1)
      export
   case addconst
      i = $(add $i, $2)
      export
   default
      eprintln($"Unknown command: $1")
```

scan 関数はまたいくつかのオプションをサポートしています。

```
scan(options, files)
...
```

- **A** : 各々の行を引数のリストとして、クオート化された状態でパースします。例えば、以下の行は 3 つのワード “ls” , “-l” , “Program Files” を持つことになります: ls -l "Program Files"
- **O** : 各々の行を、ホワイトスペースをセパレータとしてパースします。これは通常 OMake が文字列をパースする際のアルゴリズムを使用しています。この動作はデフォルトです。

- **x** : 各々の行を分割し、さらに 16 進数表現を用いてワードの個数を減らします。これは URL を指定する際には通常 16 進数表現が用いられるためです。例えば、文字列 `Program Files` は代わりに以下の形 `Program+Files` に置き換わります。

ノート: もしあなたが出力をファイルにリダイレクトしたい場合、最も簡単な方法は `stdout` 変数を再定義することです。 `stdout` 変数は他の変数と同様スコープ化されているので、この再定義は `calc` 関数の外にある `stdout` には影響を及ぼしていません。

```
calc() =
    stdout = $(fopen script.out, w)
    scan(script.in)
       ...
    close(stdout)
```

10.11.5 awk

```
awk(input-files)
case pattern1:
   body1
case pattern2:
   body2
...
default:
   bodyd
```

あるいは

```
awk(options, input-files)
case pattern1:
   body1
case pattern2:
   body2
...
default:
   bodyd
```

`awk` 関数は awk(1) と似た入力機能を提供します。が、この関数は制限されています。引数 `input-files` には値のシーケンスを指定することで、各々の値には `InChannel` が指定されるか、入力としてファイル名が用いられます。もしオプションとファイルの引数が何も指定されなかったら、入力は `stdin` が用いられます。なお、出力は常に `stdout` です。

変数 `RS` と `FS` にはレコードとフィールドを分割するセパレータが正規表現の形で定義されています。なお、デフォルトの `RS` の値は正規表現 `\r|\n|\r\n` で、 `FS` は `[ \t]+` です。

`awk` 関数は入力から一つのレコードを読み込み、以下のアルゴリズムに従って処理を行います。

各々の行で、レコードはまずフィールドセパレータ `FS` を用いていくつかのフィールドに分割されて、それらのフィールドは変数 `$1, $2, ...` に束縛されます。変数 `$0` は行全体として定義されており、 `$*` はすべてのフィールドの値が定義されている配列です。変数 `$(NF)` はフィールドの数が定義されています。

次に、 `case` が順番どおりに評価されていきます。各々の `case` において、もし正規表現 `pattern_i` がレコード `$0` にマッチしていた場合は、 `body_i` が評価されます。もし `body_i` が `export` で終わっていたのなら、現在の状態は次の宣言句へ受け継がれます。そうでない場合、この値は捨てられます。もし正規表現が `\(r\)` を含んでいたのなら、フィールド `$1, $2, ...` はこれらの表現で書き換えられます。

例えば、以下のコードはテキストが二つのデリミタ `\begin{<name>}` と `\end{<name>}` の間にあり、さらに `filter` 関数の引数として渡された配列の中に `<name>` が入っているときだけ、その間のテキストを出力しています。

```
filter(names) =
   print = false

   awk(Awk.in)
   case $"^\\end\{\([:alpha:]+\)\}"
      if $(mem $1, $(names))
         print = false
         export
      export
   default
      if $(print)
         println($0)
   case $"^\\begin\{\([:alpha:]+\)\}"
      print = $(mem $1, $(names))
      export
```

ノート: もしあなたが出力をファイルにリダイレクトしたい場合、最も簡単な方法は stdout 変数を再定義することです。 stdout 変数は他の変数と同様スコープ化されているので、この再定義は filter 関数の外にある stdout には影響を及ぼしていません。

```
filter(names) =
    stdout = $(fopen file.out, w)
    awk(Awk.in)
       ...
    close(stdout)
```

オプション :

- **b** : case を評価する際にループを『中断 (Break)』します。ただし、複数のマッチ文が選択されるような場合のみです。

break 関数はループを停止する際に用いられます。これを用いると awk 関数は即座に中断します。

10.11.6 fsubst

```
fsubst(files)
case pattern1 [options]
   body1
case pattern2 [options]
   body2
...
default
   bodyd
```

fsubst 関数は sed(1) のような置換機能を提供します。 awk と似ていて、もし fsubst が何の引数も指定されずに呼び出された場合、入力は stdin が用いられます。もし引数が与えられていた場合、各々の引数は InChannnel が指定されるか、入力としてファイル名が用いられます。

RS 変数はレコードのセパレータを指定する正規表現が定義されており、 RS のデフォルトの値は \r|\n|\r\n です。

fsubst 関数は 1 回につき 1 つのレコードを読み込みます。

各々のレコードで、 case 文は順番どおりに評価されます。各々の case ではマッチした pattern を、定義された文字列に置換する機構について定義しています。

現在のところ、omake では g オプションだけがサポートされています。指定した場合、各々の宣言句は全体の置換を行い、すべての pattern のインスタンスによって置換が行われます。そうでない場合、置換は 1 回だけ行われます。

出力は stdout 変数を再定義することによってリダイレクトできます。

例えば、以下のプログラムは word に適合した文字列すべてを大文字化し、置換します。

```
section
   stdout = $(fopen Subst.out, w)
   fsubst(Subst.in)
   case $"\<\([[:alnum:]]+\)\." g
      value $(capitalize $1).
   close(stdout)
```

10.11.7 lex

```
lex(files)
case pattern1
   body1
case pattern2
   body2
...
default
   bodyd
```

lex 関数はシンプルな文法解析器を提供します。入力はファイルやチャネルのシーケンスです。 case には正規表現を指定します。この関数は入力を読み込むたび、 **最も長い接頭辞** にマッチした正規表現を選択し、その内容を評価します。

同じ長さで 2 つの case 文がマッチしてしまった場合、 **後ろの** case 文が実行されます。 default 文は正規表現 . にマッチするので、パターンリストの最初に設置するのが恐らく望ましいでしょう。

もし case の内容が export で終わっていたのなら、現在の状態は次のループへ受け継がれます。

例えば、以下のプログラムは入力されたファイルからすべての英数字を集めます。

```
collect-words($(files)) =
   words[] =
   lex($(files))
   default
      # empty
   case $"[[:alnum:]]+" g
      words[] += $0
      export
```

default 文が存在する場合、この文は任意の 1 つの文字のみにマッチします。また、もし入力がどの正規表現にもマッチしなかった場合、この関数はエラーとなります。

break 関数はループを停止する際に用いられます。

10.11.8 lex-search

```
lex-search(files)
case pattern1
   body1
case pattern2
   body2
...
default
   bodyd
```

lex-search 関数は lex 関数と似ていますが、この関数はどの正規表現にもマッチしなかった入力をスキップします。 default 文を含んでいた場合、 default はすべてのスキップしたテキストにマッチします。

例えば、以下のプログラムは入力されたファイルからすべての英数字文字を集め、含まれていない他のテキストはスキップします。

```
collect-words($(files)) =
   words[] =
   lex-search($(files))
   default
      eprintln(Skipped $0)
   case $"[[:alnum:]]+" g
      words[] += $0
      export
```

default 文が存在する場合、この文は任意の 1 つの文字のみにマッチします。また、もし入力がどの正規表現にもマッチしなかった場合、この関数はエラーとなります。

break 関数はループを停止する際に用いられます。

10.11.9 Lexer

Lexer オブジェクトは容易に字句解析を行えるようにするオブジェクトで、 lex(1) や flex(1) プログラムと似ています。

omake では、字句の解析は Lexer クラスを継承することによって動的に構成することができます。字句解析器 (以後レキサと呼ぶ) の定義はメソッドを呼び出すことで指示文 (directive) を指定しているものの集合と、ルールとして宣言句 (clause) を指定しているものの集合によって成り立っています。

例えば、以下のシンプルなデスクトップ電卓の演算の字句解析を行う、レキサの定義について考えてみましょう。

```
lexer1. =
   extends $(Lexer)

   other: .
      eprintln(Illegal character: $* )
      lex()

   white: $"[[:space:]]+"
      lex()

   op: $"[-+*/()]"
      switch $*
      case +
         Token.unit($(loc), plus)
      case -
         Token.unit($(loc), minus)
      case *
         Token.unit($(loc), mul)
      case /
         Token.unit($(loc), div)
      case $"("
         Token.unit($(loc), lparen)
      case $")"
         Token.unit($(loc), rparen)

   number: $"[[:digit:]]+"
      Token.pair($(loc), exp, $(int $* ))
```

```
   eof: $"\'"
      Token.unit($(loc), eof)
```

このプログラムは Lexer オブジェクトから字句解析の環境を定義している lexer1 を継承しています。

残りの定義では宣言句の集合の定義を行っています。コロン (:) の前にはメソッド名を指定し、コロンの後には正規表現を指定します。この場合は内容も指定しています。内容はなくても構いません。指定されなかった場合、レキサの定義で既に存在している、与えられたメソッド名が用いられます。

警告: 最も長い 接頭辞にマッチした宣言句が選択されます。もし 2 つの宣言句が同じ長さの場合、**後ろの** 宣言句が選択されます。これはほとんどの標準的なレキサとな異なっていますが、拡張性から見ればこの仕様は大きな意味を持ちます。

最初の宣言句は他の宣言句にマッチしなかった任意の入力文字列がマッチします。この場合、未知の文字のエラーメッセージが出力されます。この宣言句は他の宣言句にマッチしなかった場合のみ選択されることに注意してください。

2 番目の宣言句ではホワイトスペースを無視する役割を持っています。ホワイトスペースが見つかった場合、これを無視し、再帰的にレキサを呼び出します。

3 番目の宣言句では演算子の役割を持っています。ここでは Token オブジェクトを利用しています。なお、この Token オブジェクトは 3 つのフィールド (ソース位置を表現している loc , name , value) を定義しています。

レキサは各々のメソッドの body 部で、現在の語彙素 (lexeme) の位置を表す loc 変数が定義されているので、私たちはトークンを生成するためにこの値を用いています。

Token.unit($(loc), name) メソッドは与えられた名前とデフォルトの値を用いて新しい Token オブジェクトを構成します。

number 宣言句は正の整数の定数にマッチします。 Token.pair($(loc), name, value) は与えられた名前と値でトークンを構成します。

Lexer オブジェクトは InChannel オブジェクトを操作します。lexer1.lex-channel(channel) メソッドは与えられたチャネルから次のトークンを読み込みます。

10.11.10 レキサのマッチング

字句解析においては、最も長くマッチした宣言句が選択されます。これは、最も長い入力文字のシーケンスにマッチした宣言句が評価対象になるということです。もしどの宣言句にもマッチしなかった場合、レキサは RuntimeException を送出します。もし 1 つ以上の宣言句が同じ量の入力にマッチした場合、最初の 1 つが評価に用いられます。

10.11.11 拡張したレキサの定義

それでは前回のレキサのサンプルを、コメントを無視するように拡張してみましょう。ここで、コメントを (* から *) で終わる任意のテキストとして定義します。なお、コメントはネスト化されているものとします。

これを実現する一つの簡単な方法としては、コメントをスキップする別のレキサを定義することが挙げられます。

```
lex-comment. =
   extends $(Lexer)

   level = 0
```

```
   other: .
      lex()

   term: $"[*][)]"
      if $(not $(eq $(level), 0))
         level = $(sub $(level), 1)
         lex()

   next: $"[(][*]"
      level = $(add $(level), 1)
      lex()

   eof: $"\'"
      eprintln(Unterminated comment)
```

このレキサには、ネストレベルを記録し続けている lebel フィールドを含んでいます。(* に遭遇すると、この変数はレベルを 1 増やし、 *) が来たら、0 でない場合はレベルを 1 減らし、続けます。

次に、前回のレキサを、コメントをスキップするような形に修正してみましょう。これはちょうど前に作った lexer1 オブジェクトを拡張することで実現できます。

```
lexer1. +=
   comment: $"[(][*]"
      lex-comment.lex-channel($(channel))
      lex()
```

comment 宣言句の内容にはコメントに遭遇した場合、 lex-comment レキサを呼び出し、このレキサが返されたときに解析し続けることを指定しています。

10.11.12 lexer オブジェクトを扱いやすくする

宣言句の内容はまた指示文のエクスポートで終了します。この場合、レキサオブジェクト自身がトークンを返すものとして使われます。もしレキサオブジェクトの上で Parser オブジェクトを使うのであれば、レキサは loc , name , value フィールドを、各々の export 宣言句の中で定義すべきです。毎回 Parser オブジェクトはレキサを呼び出し、さらに前回のレキサを起動することで返されるレキサを呼び出します。

10.11.13 Parser

Parser オブジェクトは『文脈自由な文法 (context-free grammars)』をベースとした、文法解析機能を提供しています。

Parser オブジェクトは指示文のシーケンスとして指定されます。また、 Parser オブジェクトはメソッドの呼び出し、生成、ルールの指定も担当しています。

例えば、前回の Lexer のサンプルを使って、デスクトップ計算機を作ってみましょう。

```
parser1. =
   extends $(Parser)

   #
   # 主に使うレキサを定義
   #
   lexer = $(lexer1)

   #
   # 昇順に優先順位を定義
   #
```

```
left(plus minus)
left(mul div)
right(uminus)

#
# プログラム
#
start(prog)

prog: exp eof
   return $1

#
# 単純な算術式定義
#
exp: minus exp :prec: uminus
   neg($2)

exp: exp plus exp
   add($1, $3)

exp: exp minus exp
   sub($1, $3)

exp: exp mul exp
   mul($1, $3)

exp: exp div exp
   div($1, $3)

exp: lparen exp rparen
   return $2
```

パーサは Parser クラスの式として定義されています。パーサオブジェクトは lexer フィールドを持たなければなりません。 lexer は Lexer オブジェクトでなければならないというわけではありませんが、トークンオブジェクトを返す lexer.lex() メソッドと name , value フィールドを提供していなければなりません。例えば、今回私たちは前回の項で定義した lexer1 オブジェクトを使用しました。

次のステップでは演算子記号の優先順位を定義します。優先順位は left , right , nonassoc メソッドの順に上がっていきます (left が一番下で nonassoc が一番上)。

文法 (grammar) は最低でも一つの開始記号を持たなければなりません。これは start メソッドで宣言できます。

次に、文法の中の生成部 (productions) はルールに従って並べられます。生成部の名前はコロンの前に指定し、変数のシーケンスはコロンの後に指定します。内容部には、生成部が入力の一部として解釈された場合における、意味論的な行動 (semantic action) を指定します。

今回の例では、デスクトップの計算機によって解析された算術式を生成しています。今回の場合、『意味論的な行動』は数値計算としてふるまいます。変数 $1, $2, ... は生成部の右から順に、各々の変数に関連付けられた値として扱われます。

10.11.14 パーサの呼び出し

パーサは $(parser1.parse-channel start, channel) や $(parser1.parse-file start, file) で呼び出されます。 start 引数には開始記号を指定し、 channel あるいは file にはパーサへの入力を指定します。

10.11.15 パースの制御

パーサの生成器はLALR(1) テーブルをベースにした、『先読み後出しのオートメーション (pushdown automation)』を生成します。通常、文法が曖昧である場合には、このオートメーションは『移動してから減らすのか』あるいは『減らしてからさらに減らすのか』で衝突することになります。これらの衝突はオートメーションが生成された時点で、標準出力に表示されます。

通常は、オートメーションはパーサが始めて使われることになるまで構築されません。

`build(debug)` メソッドはオートメーションの構築を強制的に行います。 `build(debug)` メソッドを呼び出すことによって、各々のパーサが要求されなくなるまで完全に構築させるというのは賢い方法です。 `debug` 変数が設定されている場合、このメソッドはパーサテーブルの任意の衝突をそれぞれ出力します。

`loc` 変数は行動部 (action bodies) の内部で定義されます。また、生成部の右側にある、すべてのトークンの入力範囲を表現します。

10.11.16 拡張したパーサ

パーサはまた継承することで拡張できます。例えば、文法を拡張することでシフト演算 << と >> を理解できるようにしてみましょう。

初めに、私たちはレキサを拡張することで、これらのトークンを理解できるようにします。今回、私たちは += 演算子を使う代わりに、既存の `lexer1` を完全なものにすることを選びました。

```
lexer2. =
   extends $(lexer1)

   lsl: $"<<"
      Token.unit($(loc), lsl)

   asr: $">>"
      Token.unit($(loc), asr)
```

次に、私たちはこれらの新しい演算子を扱うように、既存のパーサを拡張しました。ビット演算子は既存の算術演算子よりも低い優先順位にするつもりです。今回は二つの引数をとる `left` メソッドを使うことで実現しました。

```
parser2. =
   extends $(parser1)

   left(plus, lsl lsr asr)

   lexer = $(lexer2)

   exp: exp lsl exp
      lsl($1, $3)

   exp: exp asr exp
      asr($1, $3)
```

今回の場合、私たちは新しいレキサ `lexer2` を使用して、さらに新しいシフト演算子の生成部を追加しました。

10.11.17 Passwd

`Passwd` オブジェクトはシステムユーザのデータベース上にあるエントリを表現します。このオブジェクトは以下のフィールドを持っています。

- pw_name : ログインネーム
- pw_passwd : 暗号化されたパスワード
- pw_uid : ユーザのユーザ ID
- pw_gid : ユーザのグループ ID
- pw_gecos : ユーザ名かコメント欄
- pw_dir : ユーザのホームディレクトリ
- pw_shell : ユーザが通常使うシェル

すべてのフィールドがすべての OS 上で意味をもつわけではないことに注意してください。

10.11.18 getpwnam, getpwuid

```
$(getpwnam name...) : Passwd
   name : String
$(getpwuid uid...) : Passwd
   uid : Int
raises RuntimeException
```

getpwnam 関数はユーザのログイン名からエントリを探しだします。 getpwuid 関数はユーザ ID(numerical id, uid) からエントリを探し出します。もしエントリが見つからなかった場合、例外が送出されます。

10.11.19 getpwents

```
$(getpwents) : Array
```

getpwents 関数は Passwd オブジェクトの配列を返します。すべてのユーザは、システムユーザのデータベースによって用意されます。この関数は OS やユーザデータベースの状況に依存し、返される配列は完全でなかったり、空である可能性もある点に注意してください。

10.11.20 Group

Group オブジェクトはシステムのユーザグループに関するデータベースのエントリを表現します。このオブジェクトは以下のフィールドを含んでいます。

- gr_name : グループ名
- gr_group : 暗号化されたパスワード
- gr_gid : グループのグループ ID
- gr_mem : グループメンバのユーザ名

すべてのフィールドがすべての OS で意味を持つわけでは無い点に注意してください。

10.11.21 getgrnam, getgrgid

```
$(getgrnam name...) : Group
   name : String
$(getgrgid gid...) : Group
   gid : Int
raises RuntimeException
```

getgrnam 関数はグループ名からグループエントリを探しだし、 getgrgid 関数はグループ ID(gid) からエントリを探し出します。何も見つからなかった場合は例外が送出されます。

10.11.22 tgetstr

```
$(tgetstr id) : String
   id : String
```

tgetstr 関数は指定された id を用いて『端末の能力 (terminal capability, termcap)』を調べます。これは、”terminfo” の調査が TERM 環境変数によって与えられることを保証しています。もし与えられた”terminal capability” が定義されていなかった場合、この関数は空の値を返します。

ノート: シェルプロンプト内部の tgetstr によって返された値を使用したい場合、あなたは prompt-invisible 関数を用いてラップする必要があります。

10.11.23 xterm-escape-begin, xterm-escape-end

```
$(xterm-escape-begin) : String
$(xterm-escape-end) : String
```

xterm-escape-begin と xterm-escape-end 関数は XTerm のウィンドウタイトルを設定したい場合に用いることができる、エスケープのシーケンスを返します。この機能が使えない場合、この関数は空の値を返します。

ノート: シェルプロンプト内部の値を用いるようにしたい場合、あなたは $(prompt_invisible_begin)$(xterm-escape-begin) と $(xterm-escape-end)$(prompt_invisible_end) を使う必要があります。

10.11.24 xterm-escape

```
$(xterm-escape s) : Sequence
```

TERM 環境変数が『XTerm のタイトルを設定する機能が利用できる』ことを表していた場合、$(xterm-escape s) は $(xterm-escape-begin)s$(xterm-escape-end) と等価です。そうでない場合、この関数は空の値を返します。

ノート: シェルプロンプト内部の xterm-escape によって返された値を用いるようにしたい場合、あなたは prompt-invisible 関数を使ってラップする必要があります。

10.11.25 prompt-invisible-begin, prompt-invisible-end

```
$(prompt-invisible-begin) : String
$(prompt-invisible-end) : String
```

prompt-invisible-begin と prompt-invisible-end 関数は、シェルプロンプトに『見えない』セクション (様々なエスケープシーケンスのような) を設定するために用いる必要のある、エスケープのシーケンスを返します。

10.11.26 prompt-invisible

```
$(prompt-invisible s) : Sequence
```

prompt-invisible は指定された引数を $(prompt-invisible-begin) と $(prompt-invisible-end) でラップします。シェルプロンプトに『見えない』すべてのセクション (様々なエスケープシーケンスのような) はこの方法でラップしなければなりません。

10.11.27 gettimeofday

```
$(gettimeofday) : Float
```

gettimeofday 関数は 1970 年 1 月 1 日から始まる、現在までの秒数を返します。

CHAPTER 11

シェルコマンド

シェルコマンド (OS によって実行されるコマンド) は自由に他のコードとミックスすることができます。

ノート: 構文とシェルの使い方はすべてのプラットフォーム (Win32 を含む) において同一です。Win32 上に移植した場合の問題を避けるために、ネイティブのインタープリター cmd を使わないことをおすすめします。

```
LIB = $(dir lib)
println(The contents of the $(LIB) directory is:)
ls $(LIB)
```

11.1 簡単なコマンド

シェルコマンドの構文は Unix シェル bash を使うときの構文と似ています。通常、コマンドは **パイプライン** となっています。通常のコマンドはパイプラインの一部です。コマンドを実行するためには、実行可能なコマンド名といくつかの引数を指定する必要があります。以下はいくつかの例です。

```
ls
ls -AF .
echo Hello world
```

コマンドは実行可能コマンドが格納されているディレクトリの配列 PATH[] 変数の値を用いて探します。

コマンドは環境変数の定義によって修正されることがあります。

```
# "Hello world" と出力
env X="Hello world" Y=2 printenv X
# Visual C++ のインクルードパスを検索
env include="c:\Program Files\Microsoft SDK\include" cl foo.cpp
```

11.2 検索

コマンドにはワイルドカードパターンを含めることができます。パターンには制限された正規表現を用いたファイルの集合を指定します。パターンは関数が実行される前に展開されます。

```
# .c 拡張子のファイルをすべてリスト
ls *.c
```

```
# 1 文字の接頭辞と.c の拡張子を持ったすべてのファイルをリスト
ls ?.c

# hello.ml ファイルを foo.ml にリネーム
mv {hello,foo}.ml
```

OMake の glob パターンのさらなる説明は “**ファイルの検索とリスト**” で与えられます。

11.3 バックグラウンドでのジョブ

コマンドはまたアンパサンド & をコマンドの後に付与することで、バックグラウンド上で実行されます。ユーザへの制御は、ジョブが完了するまで待つことなく返されます。ジョブはバックグラウンド上で走り続けます。

```
gcc -o hugeprogram *.c &
```

11.4 ファイルのリダイレクション

入力と出力は <, >, >& をコマンドの後に付与することによって、ファイルにリダイレクトすることができます。

```
# "foo" ファイルに書き込み
echo Hello world > foo

# foo ファイルからの入力をリダイレクト
cat < foo

# 標準出力、標準エラーを foo ファイルにリダイレクト
gcc -o boo *.c >& foo
```

11.5 パイプライン

パイプラインはコマンドのシーケンスで、各々のコマンドの出力は次のコマンドへ送られます。パイプは | と |& で定義されます。 | は出力はリダイレクトされますが、エラーはされません。 |& は出力とエラーの両方がリダイレクトされます。

```
# ls コマンドの出力をプリンターに送る
ls *.c | lpr

# 出力とエラーを E メールを使って jyh に送る
gcc -o hugefile *.c |& mail jyh
```

11.6 条件分岐の実行

コマンドは || と && の条件分岐を使うことで複雑に組み合わせることができます。すべてのコマンドは 0 か他の整数の終了コードを返します。コマンドは終了コードが 0 であった場合、成功したと宣言します。式 command1 && command2 は、 command1 が成功した場合のみ command2 が実行されます。式 command1 || command2 は、 command1 が失敗した場合のみ command2 が実行されます。

```
# 可能な場合のみ x/y ファイルを表示する
cd x && cat y

# foo.exe を実行するか、エラーメッセージを表示する
(test -x foo.exe && foo.exe) || echo "foo.exe is not executable"
```

11.7 グループ化

パイプラインをグループ化したり、条件分岐をさせる場合には括弧を使います。以下の式では、 `test` 関数は `foo.exe` ファイルが実行可能であるかどうか試し、もしそうであったのなら、 `foo.exe` ファイルが実行されます。もし実行可能でなかったのなら (あるいは `foo.exe` コマンドが失敗したのなら)、メッセージ `"foo.exe is not executable"` が表示されます。

```
# foo.exe を実行するか、エラーメッセージを表示する
(test -x foo.exe && foo.exe) || echo "foo.exe is not executable"
```

11.8 シェルコマンドとは何か？

構文的には、シェルコマンドは以下のうち一つも該当していない任意の行を指します。

- `VAR=string` の形の変数定義
- 関数の呼び出し `f(...)` かメソッドの呼び出し `o.f(...)`
- コロン:を含んだルールの定義 `string: ...`
- **以下のリストを含む特殊コマンド**
  - `if ...`
  - `switch ...`
  - `match ...`
  - `section ...`
  - `return ...`

コマンドはまたビルドイン (エイリアス) でもあります。さらなる情報は "*Shell*" を参照してください。

11.9 基本的なビルドイン関数

11.9.1 echo

`echo` 関数は文字列を表示します。

```
$(echo <args>)
echo <args>
```

11.9.2 cd

cd 関数はカレントディレクトリを変更します。

```
cd(dir)
   dir : Dir
```

cd 関数は 2 つの引数を取ることもできます。

```
$(cd dir, e)
   dir : Dir
   e : expression
```

2 つ引数を取る形では、式 e がディレクトリ dir 上で評価されます。カレントディレクトリは変更されません。

cd 関数のふるまいはディレクトリの検索パスを指定している CDPATH 変数を用いて変更できます。これは通常、osh コマンドインタープリタ上でのみ有効です。

```
CDPATH : Dir Sequence
```

例えば、以下の式ではディレクトリを最初のディレクトリ ./foo, ~/dir1/foo, ~/dir2/foo に変更します。

```
CDPATH[] =
   .
   $(HOME)/dir1
   $(HOME)/dir2
cd foo
```

11.10 ジョブを制御するビルドイン関数

11.10.1 jobs

jobs 関数はジョブのリストを表示します。

```
jobs
```

11.10.2 bg

bg 関数はバックグラウンド上にジョブを配置します。

```
bg <pid...>
```

11.10.3 fg

fg 関数はジョブをフォアグラウンドに持っていきます。

```
fg <pid...>
```

11.10.4 stop

stop 関数はジョブを停止 (suspends) します。

```
stop <pid...>
```

11.10.5 wait

wait 関数はジョブが完了するまで待機します。プロセス固有の値が指定されなかった場合、シェルはすべてのジョブが完了するまで待機します。

```
wait <pid...>
```

11.10.6 kill

kill 関数はジョブにシグナルを送ります。

```
kill [signal] <pid...>
```

11.11 コマンド履歴

11.11.1 history

```
$(history-index) : Int
$(history) : String Sequence
history-file : File
history-length : Int
```

ヒストリ変数は osh 上のコマンドライン履歴を管理します。これらの変数は omake に影響を及ぼしません。

history-index 変数はコマンドライン履歴における、現在のインデックスを表します。 history 変数は現在のコマンドライン履歴を表します。

コマンドライン履歴を保存したいと思った場合、あなたは history-file 変数を再定義することができます。デフォルトの値は ~/.omake/osh_history です。

コマンドライン履歴の行数の最大値を指定したい場合、あなたは history-length 変数を再定義することができます。デフォルトの値は 100 です。

標準的なオブジェクト群

ここでの『広く使われている』は、すべてのプログラムで使われているオブジェクトを意味しています。以下のオブジェクトが定義されています。

12.1 広く使われているオブジェクト

12.1.1 Object

親オブジェクト : 無し

`Object` オブジェクトはルートとなるオブジェクトです。すべてのクラスは `Object` のサブクラスです。

以下のフィールドやメソッドを提供します。

- `$(o.object-length)` : オブジェクトのメソッドとフィールドの総数
- `$(o.object-mem <var>)` : <var>がオブジェクトのフィールドやメソッドであった場合は `true` を返します。
- `$(o.object-add <var, <value>)` : オブジェクトにフィールドを追加した新しいオブジェクトを返します。
- `$(o.object-find <var>)` : オブジェクトのメソッドやフィールドを取得します。これは `$(o.<var>)` と等価ですが、変数は定数でなくても構いません。
- `$(o.object-map <fun>)` : オブジェクト全体に対して関数を適用します。この関数は二つの引数を取り、一つ目はフィールドの名前で、二番目にはフィールドの値を指定します。結果は関数によって返される値が適用された、新しいオブジェクトを返します。
- `$(o.object-foreach)` : `object-foreach` は `object-map` と等価ですが、別の構文を適用できます。

  ```
  o.object-foreach(<var1>, <var2>)
     <body>
  ```

  例えば、以下の関数はオブジェクト `o` のすべてのフィールドを表示します。

  ```
  PrintObject(o) =
   o.object-foreach(v, x)
      println($(v) = $(x))
  ```

export 文は object-foreach の内部に適用できます。以下の関数ではオブジェクトのフィールド名を収集します。

```
FieldNames(o) =
   names[] =
   o.object-foreach(v, x)
      names[] += $(v)
      export
   return $(names)
```

12.1.2 Map

親オブジェクト : Object

Map オブジェクトは値から値を返す辞書です。 <key> の値は単純な型のみが使用できます。具体的には、整数型、浮動小数点型、文字列型、ファイル型、辞書型、そして単純な型で構成された配列です。

Map オブジェクトは以下のメソッドを提供します。

- $(o.length) : 辞書に格納されているアイテムの総数
- $(o.mem <key>) : <key> が辞書に定義されている場合は true を返します。
- $(o.add <key>, <value>) : 辞書にフィールドを追加した、新しい辞書を返します。
- $(o.find <key>) : 辞書から対象のフィールドを取得します。
- $(o.keys) : 辞書の中にあるすべてのキーの配列をアルファベット順で取得します。
- $(o.values) : 辞書の中にあるすべての値の配列を、キーのアルファベット順で取得します。
- $(o.map <fun>) : 一つ目はフィールドの名前で、二番目にはフィールドの値を指定します。結果は関数によって返される値が適用された、新しいオブジェクトを返します。
- $(o.foreach) : foreach 文は map と等価ですが、別の構文を適用できます。

  ```
  o.foreach(<var1>, <var2>)
     <body>
  ```

  例えば、以下の関数はオブジェクト o のすべてのフィールドを表示します。

  ```
  PrintObject(o) =
     o.foreach(v, x)
        println($(v) = $(x))
  ```

  export 文は object-foreach の内部に適用できます。以下の関数では辞書のフィールド名を収集します。

  ```
  FieldNames(o) =
     names =
     o.foreach(v, x)
        names += $(v)
        export
     return $(names)
  ```

キーが文字列である場合には単純な記法が用意されています。キー-値のテーブルは $|key| の文を用いて定義することができます (パイプシンボル | の数は、多様性のためにいくらでも使うことができます)。

```
$|key 1| = value1
$||key1|key2|| = value2     # キーは key1|key2
X = $|key 1|                # X にフィールドの値 $|key 1| を定義
```

通常用いる修飾子も適用できます。式 $`|key| はキーの遅延評価として解釈されて、また式 $,|key| は通常の評価を行います。

12.1.3 Number

親オブジェクト：Object

Number オブジェクトは整数や浮動小数点の親オブジェクトです。

12.1.4 Int

親オブジェクト：Number

Int オブジェクトは整数を表現します。

12.1.5 Float

親オブジェクト：Number

Float オブジェクトは浮動小数点を表現します。

12.1.6 Sequence

親オブジェクト：Object

Sequence オブジェクトは一次元的に成分が格納されている、一般的なオブジェクトを表現します。このオブジェクトは以下のメソッドを提供しています。

- $(s.length)：シーケンスの長さを返します。
- $(s.map <fun>)：シーケンスのフィールド全体に対して関数を適用します。この関数は一つの引数を取ります。結果は関数によって返される値からなる、新しいシーケンスを返します。
- $(s.foreach)：foreach 文は map と等価ですが、別の構文を適用できます。

  ```
  s.foreach(<var>)
     <body>
  ```

  例えば、以下の関数ではシーケンスのすべての成分を表示します。

  ```
  PrintSequence(s) =
     s.foreach(x)
        println(Elem = $(x))
  ```

  export 文は foreach の内部に適用できます。以下の関数はシーケンスの 0 の数を数えます。

  ```
  Zeros(s) =
     count = $(int 0)
     s.foreach(v)
        if $(equal $(v), 0)
           count = $(add $(count), 1)
           export
        export
     return $(count)
  ```

- $(s.forall <fun>)：シーケンスの各々の成分が、与えられた関数の評価を満足しているかどうか調べます。

- $(s.exists <fun>) : シーケンスに与えられた関数の評価を満足するような成分があるかどうか調べます。
- $(s.sort <fun>) : シーケンスをソートします。 <fun> は比較用の関数です。この関数は二つの成分 (x, y) を持つシーケンスを引数に取り、x,y を比較して、もし x < y であったのなら負の値を返し、x > y であったなら正の値を返し、二つの成分が等価であったなら 0 を返します。

```
osh> items = $(int 0 3 -2)
osh> items.forall(x => $(gt $x, 0))
- : bool = false
osh> items.exists(x => $(gt $x, 0))
- : bool = true
osh> items.sort($(compare))
- : Array = -2 3 0
```

12.1.7 Array

親オブジェクト : Sequence

Array は自由にアクセスできるシーケンスです。このオブジェクトは以下の追加メソッドを提供しています。

- $(s.nth <i>) : シーケンスの i 番めの成分を返します。
- $(s.rev <i>) : 逆転させたシーケンスを返します。

12.1.8 String

親オブジェクト : Array

12.1.9 Fun

親オブジェクト : Object

Fun オブジェクトは以下のメソッドを提供しています。

- $(f.arity) : 関数の場合はアリティ(関数が取る引数の個数) を返します。

12.1.10 Rule

親オブジェクト : Object

Rule オブジェクトはビルドルールを表現します。これは現在なんのメソッドも持っていません。

12.1.11 Target

親オブジェクト : Object

Target オブジェクトは指定したターゲットファイルの情報を収集しているオブジェクトです。

- target : ターゲットファイル
- effects : ターゲットがビルドされたときに、副次的に修正されるであろうファイル群
- scanner_deps : ターゲットがスキャンできるようになる前にビルドしなければならない静的な依存関係
- static-deps : 静的に定義されている、ターゲットの依存関係

- build-deps : 静的、あるいはスキャンされた依存関係を含む、ターゲットに対するすべての依存関係
- build-values : ビルドに関係している、すべての依存関係の値
- build-commands : ターゲットをビルドするためのコマンド
- output-file : --output-* オプション (*-output-normal* 以降を参照) を用いて、出力が適当なファイルに書き込まれるような場合、このフィールドはファイル名を表します。そうでない場合は false を示します。

このオブジェクトは以下のメソッドをサポートしています。

- find(file) : 与えられたファイルのターゲットオブジェクトを返します。指定したターゲットがプロジェクトの一部でない場合は RuntimeException を送出します。
- find-optional(file) : 与えられたファイルのターゲットオブジェクトを返します。指定したターゲットがプロジェクトの一部でない場合は false を返します。

ノート: ターゲットの情報は動的に構築されるので、あるノードの Target オブジェクトは異なる箇所で異なった値を含む場合があります。 Target の情報が完全にするためのもっとも簡単な方法は、対象のターゲットファイルに依存しているルール中、もしくは対象のターゲットファイルの依存関係の中で Target オブジェクトを計算することです。

12.1.12 Node

親オブジェクト : Object

Node オブジェクトはファイルやディレクトリの親オブジェクトです。このオブジェクトは以下の操作をサポートしています。

- $(node.stat) : ファイルの stat オブジェクトを返します。もしファイルがシンボリックリンクであったなら、 stat の情報はリンク先になります。リンク自身ではありません。
- $(node.lstat) : ファイルか、シンボリックリンクの stat オブジェクトを返します。
- $(node.unlink) : ファイルを消去します。
- $(node.rename <file>) : ファイルをリネームします。
- $(node.link <dst>) : このファイルのハードリンクを <dst> に生成します。
- $(node.symlink <file>) : このファイルのシンボリックリンクを <dst> に生成します。
- $(node.chmod <perm>) : このファイルのパーミッションを変更します。
- $(node.chown <uid>, <gid>) : このファイルの所有者とグループ ID を変更します。

12.1.13 File

親オブジェクト : Node

File オブジェクトはファイル名を表現します。

12.1.14 Dir

親オブジェクト : Node

Dir オブジェクトはディレクトリ名を表現します。

12.1.15 Channel

親オブジェクト : Object

Channel オブジェクトは一般的な IO チャネルを表現します。このオブジェクトは以下のメソッドを提供しています。

- $(o.close) : チャネルを閉じます。
- $(o.name) : チャネルに関係しているファイル名を返します。

12.1.16 InChannel

親オブジェクト : Channel

InChannel は入力チャネルです。変数 stdin は標準入力チャネルです。

このオブジェクトは以下のメソッドを提供しています。

- $(InChannel.fopen <file>) : 新しい入力チャネルを開きます。
- $(InChannel.of-string <string>) : 文字列を入力として、新しい入力チャネルを開きます。
- $(o.read <number>) : チャネルから、与えられた数だけ文字を読み込みます。
- $(o.readln) : チャネルから一行を読み込みます。

12.1.17 OutChannel

親オブジェクト : Channel

OutChannel は出力チャネルです。変数 stdout と stderr は標準出力とエラーチャネルです。

このオブジェクトは以下のメソッドを提供しています。

- $(OutChannel.fopen <file>) : 新しい出力チャネルを開きます。
- $(OutChannel.string) : 文字列を書き込む、新しい出力チャネルを開きます。
- $(OutChannel.to-string) : 現在の出力チャネルの文字列を取得します。これは OutChannel.open-string を使って作られた、出力チャネル用のメソッドです。
- $(OutChannel.append) : ファイルを追加した、新しい出力チャネルを開きます。
- $(c.flush) : 出力チャネルをクリアします。
- $(c.print <string>) : チャネルに文字列を出力します。
- $(c.print <string>) : チャネルに、改行コードを付与した文字列を出力します。

12.1.18 Location

親オブジェクト : Location

Location オブジェクトはファイルの位置を表現します。

12.1.19 Exception

親オブジェクト : Object

Exception オブジェクトは例外の基底となるオブジェクトとして用いられます。このオブジェクトはなんのフィールドやメソッドを持ちません。

12.1.20 RuntimeException

親オブジェクト : Exception

RuntimeException オブジェクトはランタイムシステムからの例外を表現します。このオブジェクトは以下のフィールドを持ちます。

- position : 例外が送出された位置を表現している文字列
- message : 例外のメッセージを保持している文字列

12.1.21 UnbuildableException

親オブジェクト : Exception

UnbuildableException オブジェクトはターゲットがビルドできないことを知らせるために用いられます。このオブジェクトは target-exists() のような関数を使った場合に捕えられます。この例外は以下のフィールドを持ちます。

- target : どのターゲットがビルドできないのかを示します。
- message : 例外のメッセージを含んでいる文字列

12.1.22 Shell

親オブジェクト : Object

Shell オブジェクトはシェルコマンドとして利用できるビルドイン関数のコレクションを含んでいます。

あなたはこのオブジェクトにメソッドを追加し拡張することで、エイリアスを定義することができます。このクラスの中にあるすべてのメソッドは一つの引数をとる必要があります。引数には引数のリストが含まれた、単純な配列が与えられます。

- echo

  echo 関数は引数を標準出力チャネルに表示します。

- jobs

  jobs メソッドは現在実行しているコマンドの状態を表示します。

- cd

  cd 関数はカレントディレクトリを変更します。カレントディレクトリは現在のスコーピングルールに従うことに注意してください。例えば、以下のプログラムでは foo ディレクトリのファイルをリストしていますが、カレントディレクトリは変更されません。

  ```
  section
     echo Listing files in the foo directory...
     cd foo
     ls
  ```

```
echo Listing files in the current directory...
ls
```

- `bg`

  `bg` メソッドはバックグラウンドにジョブを移します。ジョブが中断 (suspended) されている場合、そのジョブは再開されます。

- `fg`

  `fg` メソッドはジョブをフォアグラウンドに持っていきます。ジョブが中断されている場合、そのジョブは再開されます。

- `stop`

  `stop` 関数は実行しているジョブを中断します。

- `wait`

  `wait` 関数は実行しているジョブが終了するまで待機します。この関数は、中断しているジョブを待機することができません。

  ジョブをフォアグラウンドに持っていくことはしません。また、 `wait` が割り込まれた場合、ジョブはバックグラウンド上で実行し続けます。

- `kill`

  `kill` 関数はジョブにシグナルを送ります。

  ```
  kill [signal] <pid...>.
  ```

  シグナルは数値かシンボリックのどちらでも構いません。シンボリックシグナルは以下のように命名されています。

  ABRT, ALRM, HUP, ILL, KILL, QUIT, SEGV, TERM, USR1, USR2, CHLD, STOP, TSTP, TTIN, TTOU, VTALRM, PROF

- `exit`

  `exit` 関数は現在のセッションを終了します。

- `which` , `where`

  相当する関数のドキュメントを参照してください。

- `rehash`

  検索パスをリセットします。

- `ln-or-cp` *src dst*

  src を *dst* にリンクもしくはコピーします。 *dst* は上書きされます。通常、 `ln-or-cp` は初めに出力先のファイルを (存在している場合は) 消去します。次にこの関数は入力先のファイルを指し示しているシンボリックリンクを出力先に生成します (パスはできる限り相対パスとして設定されます)。シンボリックリンクが生成できない場合 (例. OS やファイルシステムがシンボリックリンクをサポートしていない場合)、この関数はハードリンクを生成しようと試みます。もしこの試みも失敗したのなら、強制的に入力先のファイルを出力先にコピーしようと試みます。

- `history`

  現在のコマンドライン履歴を表示します。

- `digest`

  与えられたファイルの要約を出力します。

- Win32 関数群

  Win32 ではスクリプトのために使える、多くのプログラム (DOS `cmd.exe` に含まれている関数を除く) を提供しています。以下の関数は Win32 で定義されており、Win32 上でしか使えません。他のシステム上では、この関数はすでに存在している他のプログラムに置き換わることになるでしょう。

  - `grep`

    ```
    grep [-q] [-n] pattern files...
    ```

    `grep` 関数は omake の `grep` 関数を呼び出します。

- omake 内部の標準システムコマンド

  通常、以下のコマンドは omake が内部に保有している関数を使います: `cp`, `mv`, `cat`, `rm`, `mkdir`, `chmod`, `test`, `find`。もしあなたがどうしてもこれらのコマンドを、システムが標準で使っているコマンドで呼び出したいのなら、 `USE_SYSTEM_COMMANDS` を `OMakeroot` ファイルの先頭に設定してください。

  - `pwd`

    ```
    pwd
    ```

    `pwd` エイリアスはカレントディレクトリの絶対パスを表示します。

  - `mkdir`

    ```
    mkdir [-m <mode>] [-p] files
    ```

    `mkdir` 関数はディレクトリを生成します。 `-m` オプションには生成されるディレクトリのパーミッションを指定します。 `-p` オプションが指定された場合、存在しない親ディレクトリもまとめて生成します。

  - `cp`

  - `mv`

    ```
    cp [-f] [-i] [-v] src dst
    cp [-f] [-i] [-v] files dst
    mv [-f] [-i] [-v] src dst
    mv [-f] [-i] [-v] files dst
    ```

    `cp` 関数は `src` ファイルを `dst` ファイルにコピーします。出力先のファイルが存在している場合は上書きします。一つ以上の入力ファイルが指定された場合、最後のファイルはディレクトリでなければならず、入力ファイルはそのディレクトリの内部にコピーされます。

    * **-f**

      確認をしないで、強制的にファイルをコピーします。

    * **-i**

      出力先のファイルを消去する前に確認します。

    * **-v**

      実際になにが行われているのか表示します。

  - `rm`

    ```
    rm [-f] [-i] [-v] [-r] files
    rmdir [-f] [-i] [-v] [-r] dirs
    ```

    `rm` 関数はファイルの集合を消去します。ファイルが存在しなかった場合、もしくはファイルが消去できなかった場合、なんの警告も表示しません。

    オプション:

* **-f**

  確認をしないで、強制的にファイルを消去します。

* **-i**

  ファイルを消去する前に確認します。

* **-v**

  実際になにが行われているのか表示します。

* **-r**

  再帰的にディレクトリの内容を消去します。

– `chmod`

```
chmod [-r] [-v] [-f] mode files
```

`chmod` 関数はファイルの集合やディレクトリのパーミッションを変更します。この関数は Win32 上ではなにも行いません。`mode` には 8 進数か、シンボリックフォーム `[ugoa]*[-=][rwxXstugo]+` を指定します。詳細は `chmod` の man を参照してください。

オプション:

* **-r**

  ディレクトリ内のすべてのファイルのパーミッションを再帰的に変更します。

* **-v**

  実際になにが行われているのか表示します。

* **-f**

  エラーが生じても実行し続けます。

– `cat`

```
cat files...
```

`cat` 関数はファイルの内容を stdout に出力します。

– `test`

```
test expression
[ expression +]+
[ --help
[ --version
```

詳細は “*test*” を参照してください。

– `find`

```
find expression
```

詳細は “*find*” を参照してください。

CHAPTER 13

ビルド関数とユーティリティ

13.1 ビルドイン .PHONY ターゲット

以下にビルドイン .PHONY ターゲットの完全なリストを示します。

- **.PHONY**

  新しい phony ターゲットを宣言します。(*.PHONY*)

- **.DEFAULT**

  デフォルトのビルドターゲットを宣言します。(*.DEFAULT*)

- **.SUBDIRS**

  プロジェクトの一部としてディレクトリをインクルードします。(*.SUBDIRS*)

- **.SCANNER**

  依存関係のスキャナを定義します。(*.SCANNER* **ルール**)

- **.INCLUDE**

  ファイルをインクルードします。(*.INCLUDE*)

- **.ORDER**

  ファイルの依存関係ルールの順番を定義します。(*file-sort*)

- **.BUILD_BEGIN**

  ビルド開始時に実行されるコマンド

- **.BUILD_SUCCESS**

  ビルドが成功したときに実行されるコマンド

- **.BUILD_FAILURE**

  ビルドが失敗したときに実行されるコマンド

.BUILD ターゲットはビルドの開始と終了時に実行されるコマンドを指定するために用いられます。.BUILD_BEGIN ターゲットはプロジェクトをビルドする前に実行されて、 .BUILD_FAILURE または .BUILD_SUCCESS はビルドが終了したときに実行されます。

例えば、以下のルールの集合はビルドの状況について、簡単なメッセージを表示します。

```
.BUILD_BEGIN:
    echo Build starting

.BUILD_SUCCESS:
    echo The build was successful

.BUILD_FAILURE:
    println($"The build failed: $(length $(find-build-targets Failed)) targets could not be built")
```

通常用いる別の使い方としては、ビルドが完了したときに通知するように定義するという方法があります。例えば、以下のルールでは (BUILD_SUMMARY 変数を使うことで) ビルドの要約を新しい X ターミナル上に表示します。

```
.BUILD_FAILURE:
    xterm -e vi $(BUILD_SUMMARY)
```

あなたがプロジェクトに直接これらのルールを追加したくない場合 (もしあなたが他の場所で実行するなら、これはよいアイデアです)、あなたはこれらのルールを .omakerc に定義することができます (詳細は *.omakerc* を参照してください)。

find-build-targets 関数はさらに具体的なビルドの状況を取得したいときに有用です。 --output-* オプションを使って出力先を変更している場合、コマンドによって生成された出力はファイルにコピーされることに注意してください。ファイル名は Target オブジェクトの output-file フィールドによって指定されます。あなたはビルドの状況をカスタマイズしたいときに、この方法を試してみてください。

13.2 オプションとバージョン管理

13.2.1 OMakeFlags

```
OMakeFlags(options)
   options : String
```

OMakeFlags 関数は OMakefile の内部で omake のオプションを設定したいときに用いられます。オプションはコマンドライン上でのオプション指定と全く同じフォーマットで指定します。

例えば、以下のコードは VERBOSE 環境変数が定義されるまで、プログレスバーを表示し続けます。

```
if $(not $(defined-env VERBOSE))
    OMakeFlags(-S --progress)
    export
```

13.2.2 OMakeVersion

```
OMakeVersion(version1)
OMakeVersion(version1, version2)
   version1, version2 : String
```

OMakeVersion 関数は OMakefile のバージョンをチェックするときに用いられます。この関数は 1 つ、または 2 つの引数を取ります。

1 つの引数をとる形では、omake のバージョン番号が <version1> よりも低いときに例外を送出します。2 つ引数をとる形では、omake のバージョン番号は version1 と version2 の間になければなりません。

13.2.3 cmp-versions

```
$(cmp-versions version1, version2)
   version1, version2 : String
```

cmp-versions 関数は任意のバージョン文字列を比較することができます。2 つのバージョン文字列が等しいときには、この関数は 0 を返します。一方で、最初の文字列が 2 番めよりも若いバージョンであるときには負の値を返し、そのいづれでもないときには正の値を返します。

13.2.4 DefineCommandVars

```
DefineCommandVars()
```

DefineCommandVars 関数はコマンドライン上から受け渡した変数を再定義します。変数定義はコマンドラインから name=value の形で受け渡されます。この関数は、これらの変数を最初の時点で定義するため、omake が内部で使用します。

13.3 依存関係グラフの調査

13.3.1 dependencies, dependencies-all, dependencies-proper

```
$(dependencies targets) : File Array
$(dependencies-all targets) : File Array
$(dependencies-proper targets) : File Array
   targets : File Array
raises RuntimeException
```

dependencies 関数は与えられたターゲットの明示的な依存関係の集合を返します。この関数はルールの本体だけに使うことができます。さらに、 dependency 関数のすべての引数は、このルールの依存先でなければいけません。この制約は、関数が実行されたときにすべての依存関係が解析されていることを保証してくれます。

dependencies-all 関数は似ていますが、この関数は依存関係を再帰的に展開し、明示的な依存関係だけでなくターゲットの依存関係すべてを返してくれます。

dependencies-proper 関数は『末端』となる依存先を除く、すべての依存関係を再帰的に返します。『末端』のターゲットは依存先やビルドコマンドが存在しないターゲット、つまり、現在のプロジェクト上にあるソースファイルの集合を指しています。

この 3 つの関数では、現在のプロジェクトの一部でないファイルは無視されます。また、3 つの関数はすべて『実際のファイル』に関連している phony や scanner ターゲットも返します。

dependencies-proper 関数を使う一つの目的としては”clean” ターゲットを実装することが挙げられます。例えば、ビルドしたすべての明示的なファイルを削除する一つの方法としては、 dependencies-proper をルール部分に使うことです。しかしながら、このルールでは消去する前にプロジェクトをビルドすることが必要となってきます。

```
.PHONY: clean

APP = ...      # 対象となるアプリケーション名
clean: $(APP)
   rm -f $(dependencies-proper $(APP))
```

dependencies-proper 関数はまた、実際のファイルに加えて phony や scanner ターゲットも返してしまうことに注意してください。

(さらにより良く実装している) 別のサンプルについては、 *filter-exists, filter-targets, filter-proper-targets* か filter-proper-targets 関数を参照してください。

13.3.2 target

```
$(target targets) : Target Array
   targets : File Sequence
raises RuntimeException
```

target 関数はそれぞれのターゲットに関連している Target オブジェクトを返します。詳細な情報は Target オブジェクト (*Target*) を参照してください。

13.3.3 find-build-targets

```
$(find-build-targets tag) : Target Array
   tag : Succeeded | Failed
```

find-build-targets はビルドの実行結果を返します。 tag では、どのターゲットを返すべきなのかについて指定します。結果は tag の場合によって異なります。

- **Succeeded**

  ビルドが成功したターゲットのリスト

- **Failed**

  ビルドできなかったターゲットのリスト

これらは主に .BUILD_SUCCESS や .BUILD_FAILURE phony ターゲットと一緒に使われます。例えば、あなたのプロジェクトの OMakefile に以下を加えることで、(ビルドが失敗したときに) 失敗したターゲットの数を表示します。

```
.BUILD_FAILURE:
    echo "Failed target count: $(length $(find-build-targets Failed))"
```

13.3.4 project-directories

```
$(project-directories) : Dir Array
```

project-directories 関数はプロジェクトの一部であると考えられる、すべてのディレクトリのリストを返します。

完全なディレクトリのリストを取得するためには、この関数をルールの本体で呼び出すべきです。

13.3.5 rule

rule 関数はビルドルールが定義されたときにいつでも呼び出されます。非常に特別なケースを除いて、あなたはこの関数を再定義すべきではありません。

```
rule(multiple, target, pattern, sources, options, body) : Rule
   multiple : String
   target   : Sequence
   pattern  : Sequence
   sources  : Sequence
```

```
   options  : Array
   body     : Body
```

rule 関数はルールが評価されたときに呼び出されます。

- **multiple**

  ルールがダブルコロン :: を用いて定義されているかどうかを示しています。

- **target**

  ターゲット名のシーケンスです。

- **pattern**

  パターンのシーケンスです。このシーケンスは通常の 2 パートのルール定義のときには空となります。

- **sources**

  依存関係のシーケンスです。

- **options**

  オプションの配列です。各々のオプションは、オプション名と値が関連付けられている Map (辞書型の) オブジェクトとして表現されます。

- **body**

  ルールの内容を表しています。

以下のルールを考えてみましょう。

```
target: pattern: sources :name1: option1 :name2: option2
   expr1
   expr2
```

上の式はまず以下の関数の呼び出しに置き換えられ、[] は配列を指定するために用いられ、さらに{}は Map オブジェクトに置き換わります。

```
rule(false, target, pattern, sources,
     { $|:name1:| = option1; $|:name2:| = option2 }
     [expr1; expr2])
```

13.3.6 build

```
build(targets : File Array) : bool
```

与えられたターゲットをビルドします。ビルドが成功した場合、この関数は真を返します。この関数は osh 上でのみ使うことができます。

13.4 OMakeroot ファイル

標準の OMakeroot ファイルでは、普通のプロジェクトをビルドするためのルール関数を定義しています。

13.4.1 変数

- **ROOT**

  現在のプロジェクトのルートディレクトリ

- **CWD**

  カレント作業ディレクトリ (このディレクトリは、プロジェクトにある各々の OMakefile の集合です)

- **EMPTY**

  空文字列

- **STDROOT**

  標準でインストールされている OMakeroot のファイル名

- **ABORT_ON_COMMAND_ERROR**

  true に設定した場合、たとえコマンドの一つがビルドに失敗したとしても、ターゲットはそれを無視します。デフォルトの値は true で、通常はそのままにしておくべきです。

- **SCANNER_MODE**

  この変数は以下の 4 つの値のうちの 1 つを選んで定義する必要があります (デフォルトは `enabled` です)。

  - **enabled**

    デフォルトの `.SCANNER` ルールを使うことを許可します。たとえルールが明示的に `:scanner:` 依存関係を指定していなくても、omake は同じ名前のターゲットから `.SCANNER` を検索します。

  - **disabled**

    デフォルトの `.SCANNER` ルールを使いません。

  - **warning**

    デフォルトの `.SCANNER` ルールを使うことを許可しますが、そのうちの一つを使うことになったとき、omake は常に警告を表示します。

  - **error**

    デフォルトの `.SCANNER` ルールを使うことを許可しません。もしルールが明示的に `:scanner:` 依存関係をしておらず、さらにデフォルトの `.SCANNER` を使うことになったとき、このルールのビルドは異常終了します。

13.4.2 システム変数

- **INSTALL** : プログラムをインストールするためのコマンド (`Unix` は `install` 、 `Win32` は `cp`)
- **PATHSEP** : 通常用いるパスのセパレータ (`Unix` は `:` 、 `Win32` は `;`)
- **DIRSEP** : 通常用いるディレクトリのセパレータ (`Unix` は `/` 、 `Win32` は `\`)
- **EXT_OBJ** : オブジェクトファイルの拡張子 (`Unix` は `.o` 、 `Win32` は `.obj`)
- **EXT_LIB** : 静的ライブラリの拡張子 (`Unix` は `.a` 、 `Win32` は `.lib`)
- **EXT_DLL** : 共有ライブラリの拡張子 (`Unix` は `.so` 、 `Win32` は `.dll`)
- **EXT_ASM** : アセンブリファイルの拡張子 (`Unix` は `.s` 、 `Win32` は `.asm`)
- **EXE** : 実行可能形式の拡張子 (`Unix` は空文字列 、 `Win32` や `Cygwin` は `.exe`)

13.5 C/C++コードのビルド

OMake では C や C++のプログラムのビルドを広範囲にわたってサポートする機能を提供しています。このセクションで列挙している関数を使うためには、まずあなたの OMakeroot ファイルに、以下の一行を追加してください。

```
open build/C
```

13.5.1 自動設定変数 (Autoconfiguration variables)

これらの変数はまず最初に OMake を実行した時点で autoconf スタイルの static. が実行されて、その結果を元に決定されます。あなたのプロジェクトを調和の取れた状態にするためには、これらの変数を使用してください。また、あなたはこれらの変数を再定義すべきではありません。

あなたは強制的にすべてのテストを実行させるために、 --configure コマンドラインオプション (*-configure* を参照) を使うことができます。

これらの自動設定変数の値はビルドする環境に依存します。これらの変数の一部は Win32 の環境下と、(Linux, OS X, Cygwin を含む)Unix ライクな環境下では異なるふるまいをします。

Unix ライクなシステム

- **GCC_FOUND** : gcc バイナリを発見したかどうかを示す真偽値
- **GXX_FOUND** : g++ バイナリを発見したかどうかを示す真偽値

Win32

- **CL_FOUND** : cl バイナリを発見したかどうかを示す真偽値
- **LIB_FOUND** : lib バイナリを発見したかどうかを示す真偽値

13.5.2 C/C++用の設定変数

以下の変数があなたのプロジェクト上で定義されています。

- **CC**

  C コンパイラの名前 (Unix では gcc が発見されたときはデフォルトの値として gcc が使われます。そうでない場合は cc が使われます。 Win32 ではデフォルトの値は cl /nologo です。)

- **CXX**

  C++コンパイラの名前 (Unix では gcc が発見されたときはデフォルトの値として gcc が使われます。そうでない場合は c++ が使われます。 Win32 ではデフォルトの値は cl /nologo です。)

- **CPP**

  C プリプロセッサの名前 (Unix ではデフォルトの値は cpp 、 Win32 の場合は cl /E)

- **CFLAGS**

  C コンパイラに渡すコンパイルフラグ (Unix ではデフォルトの値は空文字列、 Win32 の場合は /DWIN32)

- **CXXFLAGS**

  C++コンパイラに渡すコンパイルフラグ (Unix ではデフォルトの値は空文字列、 Win32 の場合は /DWIN32)

- **INCLUDES**

  C/C++コンパイラに渡す、追加する検索パスを指定しているディレクトリの値 (デフォルトの値は .)。これらのディレクトリは -I オプションを追加して C/C++コンパイラに受け渡されます。また、 -I 接頭辞を追加したインクルードパスは PREFIXED_INCLUDES 変数で定義されます。

- **LIBS**

  プログラムをビルドするときに必要となる追加ライブラリ (デフォルトの値は空文字列)

- **CCOUT**

  C/C++コンパイラの出力ファイルの場所を指定するオプション (Unix ではデフォルトの値は -o 、 Win32 の場合は /Fo)

- **AS**

  アセンブラの名前 (Unix ではデフォルトの値は as 、 Win32 の場合は ml)

- **ASFLAGS**

  アセンブラに渡すフラグ (Unix ではデフォルトの値は空文字列、 Win32 の場合は /c /coff)

- **ASOUT**

  AS の出力ファイルの場所を指定するオプション文字列 (Unix ではデフォルトの値は -o 、 Win32 の場合は /Fo)

- **AR**

  静的ライブラリを生成するためのプログラム名 (Unix ではデフォルトの値は ar cq 、 Win32 の場合は lib)

- **LD**

  リンカの名前 (Unix ではデフォルトの値は ld 、 Win32 の場合は cl)

- **LDFLAGS**

  リンカに渡すオプション (デフォルトの値は空文字列)

- **LDFLAGS_DLL**

  共有ライブラリをコンパイルするときにリンカに受け渡すオプション (Unix ではデフォルトの値は -shared 、 Win32 の場合は /DLL)

- **LDOUT**

  C/C++リンカが生成する出力ファイルの場所を指定するオプション (Unix ではデフォルトの値は -o 、 Win32 の場合は /Fe)

- **YACC**

  yacc パーサジェネレータの名前 (Unix ではデフォルトの値は yacc 、 Win32 の場合は空文字列)

- **LEX**

  lex レキサジェネレータの名前 (Unix ではデフォルトの値は lex 、 Win32 の場合は空文字列)

13.5.3 C ファイルの生成

C のスキャナは (生成されたヘッダファイルのような) 生成されたソースファイルについては何も知りません。よって、これらのファイルはスキャナが実行される前に生成される必要があります。

CGeneratedFiles, LocalCGeneratedFiles

```
CGeneratedFiles(files)
LocalCGeneratedFiles(files)
```

CGeneratedFiles と LocalCGeneratedFiles 関数は、C ファイルについての依存関係をスキャンする前に生成される必要があるファイルを指定します。例えば、 config.h と inputs.h が両方とも生成されるファイルである場合、以下のように指定します。

```
CGeneratedFiles(config.h inputs.h)
```

CGeneratedFiles 関数は **グローバル** です。この関数で指定したファイルは、任意の場所にある任意の C ファイルの依存関係をスキャンする前に生成されます。 LocalCGeneratedFiles 関数は OMake の通常のスコープルールに従います。

13.5.4 C プログラムとライブラリをビルド

StaticCLibrary, DynamicCLibrary

StaticCLibrary は静的ライブラリをビルドし、 DynamicCLibrary 関数は共有ライブラリ (DLL) をビルドします。

```
StaticCLibrary(<target>, <files>)
DynamicCLibrary(<target>, <files>)
```

<target> にライブラリの拡張子は **含めません** 。また、同様にして <files> のリストにもオブジェクトの拡張子は含めません。これらの拡張子は EXT_LIB (EXT_DLL) や EXT_OBJ 変数によって決定されます。

この関数はライブラリのファイル名を返します。

以下のコマンドは Unix 上のファイル a.o b.o c.o からライブラリ libfoo.a をビルドします。あるいは、Win32 上のファイル a.obj b.obj c.obj からライブラリ libfoo.lib をビルドします。

```
StaticCLibrary(libfoo, a b c)
.DEFAULT: $(StaticCLibrary libbar, a b c d)
```

CDLL_IMPLIES_STATIC

もし CDLL_IMPLIES_STATIC 変数が真である (Win32 上でこれはデフォルトの値です) ならば、すべての DynamicC 関数は共有ライブラリを自動的に生成し、さらに静的ライブラリを生成することを保障してくれます。

StaticCLibraryCopy, DynamicCLibraryCopy

StaticCLibraryCopy と DynamicCLibraryCopy 関数はインストール先にライブラリをコピーします。

```
StaticCLibraryCopy(<tag>, <dir>, <lib>)
DynamicCLibraryCopy(<tag>, <dir>, <lib>)
```

<tag> はターゲットの名前を指定します (大抵は .PHONY ターゲットです)。 <dir> はインストール先のディレクトリで、 <lib> はコピーするライブラリ (拡張子は除く) を指定します。

この関数はターゲットディレクトリ内でのライブラリのファイル名を返します。

例えば、以下のコードではライブラリ libfoo.a を /usr/lib ディレクトリにコピーします。

```
.PHONY: install

StaticCLibraryCopy(install, /usr/lib, libfoo)
```

StaticCLibraryInstall, DynamicCLibraryInstall

StaticCLibraryInstall と DynamicCLibraryInstall 関数はライブラリをビルドし、さらに出力先にインストール先を指定します。これらの関数はターゲットディレクトリ内でのライブラリのファイル名を返します。

```
StaticCLibraryInstall(<tag>, <dir>, <libname>, <files>)
DynamicCLibraryInstall(<tag>, <dir>, <libname>, <files>)

StaticCLibraryInstall(install, /usr/lib, libfoo, a b c)
```

StaticCObject, StaticCObjectCopy, StaticCObjectInstall

これらの関数は StaticCLibrary , StaticCLibraryCopy , StaticCLibraryInstall 関数と似ていますが、これらは **オブジェクト** ファイルをビルドします (Unix では .o ファイル、 Win32 では .obj ファイル)。

CProgram

CProgram 関数はオブジェクトファイルやライブラリの集合から C プログラムをビルドします。

```
CProgram(<name>, <files>)
```

<name> にはビルドするプログラムの名前を指定します。 <files> にはリンクするファイル名を指定します。この関数は実行可能なファイル名を返します。

オプションは以下の変数を受け渡すことで指定できます。

- **CFLAGS** : リンクする際に C コンパイラに受け渡すフラグ
- **LDFLAGS** : ローダー (loader) に受け渡すフラグ
- **LIBS** : リンクするための追加ライブラリ

例えば、以下のコードはプログラム foo がファイル bar.o と baz.o 、そしてライブラリ libfoo.a をリンクすることによって生成します。

```
section
   LIBS = libfoo
   LDFLAGS += -lbar
   CProgram(foo, bar baz)
```

CProgramCopy

CProgramCopy 関数はインストール先にファイルをコピーします。

```
CProgramCopy(<tag>, <dir>, <program>)

CProgramCopy(install, /usr/bin, foo)
```

CProgramInstall

CProgramInstall 関数は同様にプログラムをビルドし、さらに出力先にインストール先を指定します。

```
CProgramInstall(<tag>, <dir>, <name>, <files>)

section
   LIBS = libfoo
   LDFLAGS += -lbar
   CProgramInstall(install, /usr/bin, foo, bar baz)
```

CXXProgram, CXXProgramInstall

CXXProgram と CXXProgramInstall 関数は $(CC) や $(CFLAGS) の代わりに $(CXX) と $(CXXFLAGS) を使う点を除いて、対応する C の関数と等価です。

StaticCXXLibrary, StaticCXXLibraryCopy, StaticCXXLibraryInstall, DynamicCXXLibrary, DynamicCXXLibraryCopy, DynamicCXXLibararyInstall

同様にして、6 つの CXXLibrary 関数は関連する CLibrary 関数と等価です。

13.6 OCaml コードのビルド

OMake では OCaml コードをビルドするための、広範的なサポートを提供しています。このサポートには ocamlfind 、 ocamlyacc 、 menhir のようなツールへのサポートも含まれています。このセクションで列挙している関数を使うためには、まずあなたの OMakeroot ファイルに、以下の一行を追加してください。

```
open build/OCaml
```

13.6.1 OCaml コンパイルに用いる自動設定用の変数

これらの変数はまず最初に OMake を実行した時点で『自動設定スタイル (autoconf-style)』が実行されて、その結果を元に決定されます。あなたのプロジェクトを調和の取れた状態にするためには、これらの変数を使用してください。また、あなたはこれらの変数を再定義すべきではありません。

あなたは強制的にすべてのテストを実行させるために、 --configure コマンドラインオプション (*-configure* を参照) を使うことができます。

- **OCAMLOPT_EXISTS**

  あなたのマシン上で ocamlopt (あるいは ocamlopt.opt) が利用可能である場合は真となります。

- **OCAMLFIND_EXISTS**

  あなたのマシン上で ocamlfind が利用可能である場合は真となります。

- **OCAMLDEP_MODULES_AVAILABLE**

  あなたのマシン上で `-modules` オプションが理解できるバージョンの `ocamldep` が利用可能である場合は真となります。

- **MENHIR_AVAILABLE**

  あなたのマシン上で Menhir パーサジェネレータが利用可能である場合は真となります。

13.6.2 OCaml コンパイルに用いる設定用の変数

以下の変数はあなたのプロジェクト上で再定義可能な変数です。

- **USE_OCAMLFIND**

  `ocamlfind` ユーティリティを利用するかどうか (デフォルトは `false`)

- **OCAMLC**

  OCaml バイトコードコンパイラ (`ocamlc.opt` が存在していてかつ `USE_OCAMLFIND` が設定されていない場合は `ocamlc.opt` 。そうでない場合は `ocamlc`)

- **OCAMLOPT**

  OCaml ネイティブコードコンパイラ (`ocamlc.opt` が存在していてかつ `USE_OCAMLFIND` が設定されていない場合は `ocamlc.opt` 。そうでない場合は `ocamlopt`)

- **CAMLP4**

  `camlp4` プリプロセッサ (デフォルトは `camlp4`)

- **OCAMLLEX**

  OCaml レキサジェネレータ (デフォルトは `ocamllex`)

- **OCAMLLEXFLAGS**

  `ocamllex` に渡すフラグ (デフォルトは `-q`)

- **OCAMLYACC**

  OCaml パーサジェネレータ (デフォルトは `ocamlyacc`)

- **OCAMLYACCFLAGS**

  `$(OCAMLYACC)` に渡す追加オプション

- **OCAMLDEP**

  OCaml 依存関係解析器 (dependency analyzer)(デフォルトは `ocamldep`)

- **OCAMLDEP_MODULES**

  `-module` オプションが理解できる OCaml 依存解析器 (デフォルトの値は `ocamldep` 、ただし `ocamldep` が `-modules` で動いた場合のみ。`ocamlrun ocamldep-omake -module` が動く場合は `ocamlrun ocamldep-omake` が使われる。どちらでもない場合は空となる)

- **OCAMLDEP_MODULES_ENABLED**

  従来の伝統的な `make -style` の形で `OCAMLDEP` を使う代わりに `$(OCAMLDEP_MODULES) -modules` を実行させて、 関連しているすべての `.ml` と `.mli` ファイルを探しだし、その出力を内部で後処理します。OMake と、 `OCAMLDEP` や生成されたファイルの相互間の動作に関して、より詳しく知りたい方は "*OCaml* **ファイルを生成**" を参照してください。

- **OCAMLMKTOP**

  OCaml トップループコンパイラ (デフォルトは `ocamlmktop`)

- **OCAMLLINK**

  OCaml バイトコードリンカ (デフォルトは $(OCAMLC))

- **OCAMLOPTLINK**

  OCaml ネイティブコードリンカ (デフォルトは $(OCAMLOPT))

- **OCAMLINCLUDES**

  OCaml コンパイラに受け渡す検索パス (デフォルトは .)。 -I 接頭辞を加えた検索パスについては、PREFIXED_OCAMLINCLUDES 変数で定義されています。

- **OCAMLFIND**

  ocamlfind ユーティリティ(USE_OCAMLFIND が設定されている場合は ocamlfind 。そうでない場合は空文字)

- **OCAMLFINDFLAGS**

  ocamlfind に渡すフラグ (デフォルトは空文字で、USE_OCAMLFIND が設定されていなければならない)

- **OCAMLPACKS**

  ocamlfind に渡すパッケージ名 (USE_OCAMLFIND が設定されていなければならない)

- **BYTE_ENABLED**

  バイトコードコンパイラを使用するかどうかを示すフラグ (ocamlopt が見つからない場合は true 。そうでない場合は false)

- **NATIVE_ENABLED**

  ネイティブコードコンパイラを使用するかどうかを示すフラグ (ocamlopt が見つかる場合は true 。そうでない場合は false)。 BYTE_ENABLED と NATIVE_ENABLED の両方はどちらとも真にすることができます。また、最低でも一つの変数は真であるべきです。

- **MENHIR_ENABLED**

  ocamlyacc の代わりに menhir を使いたい場合は true に設定してください (デフォルトは false)。

13.6.3 OCaml コマンドフラグ

以下の変数は OCaml のツールに渡す **追加** オプションを示しています。

- **OCAMLDEPFLAGS**

  OCAMLDEP と OCAMLDEP_MODULES に渡すフラグ

- **OCAMLPPFLAGS**

  CAMLP4 に渡すフラグ

- **OCAMLCFLAGS**

  バイトコードコンパイラに渡すフラグ (デフォルトは -g)

- **OCAMLOPTFLAGS**

  ネイティブコードコンパイラに渡すフラグ (デフォルトは空)

- **OCAMLFLAGS**

  両方のコンパイラに渡すフラグ (デフォルトは -warn-error A)

- **OCAML_BYTE_LINK_FLAGS**

  バイトコードリンカに渡すフラグ (デフォルトは空)

- **OCAML_NATIVE_LINK_FLAGS**

  ネイティブコードリンカに渡すフラグ (デフォルトは空)

- **OCAML_LINK_FLAGS**

  両方のリンカに渡すフラグ

- **MENHIR_FLAGS**

  `menhir` に渡す追加フラグ

13.6.4 ライブラリ変数

以下の変数がリンクの際に用いられます。

- **OCAML_LIBS**

  リンカに渡すライブラリ。これらのライブラリはリンク作業時に依存先となります。

- **OCAML_OTHER_LIBS**

  リンカに渡す追加ライブラリ。これらのライブラリはリンク作業時に依存先には **含まれません** 。一般的な使用方法としては、OCaml の `unix` や `str` のような標準ライブラリに用いられます。

- **OCAML_CLIBS**

  リンカに渡す C ライブラリ

- **OCAML_LIB_FLAGS**

  ライブラリリンカに渡すその他のフラグ

- **ABORT_ON_DEPENDENCY_ERRORS**

  OCaml のリンカは依存する順番でリストされた OCaml のファイルを必要としています。通常、このセクションにあるすべての関数は `<files>` 引数として渡した OCaml のモジュールリストを自動的にソートします。しかし、この変数が `true` に設定してある場合、これらの関数に受け渡されたファイルの順番はそのままになります。また、OMake はこの順番が正しくなかったとしても、生じたエラーメッセージを無視します。

13.6.5 OCaml ファイルを生成

OCaml バージョン 3.09.2 に関しては、標準の `ocamldep` スキャナは『壊れています』。これに関しての主な問題は、既に存在している依存関係しか検索しないという点です。例えば、 `foo.ml` が `Bar` に依存しているものとしましょう。

```
foo.ml:
   open Bar
```

この場合、ファイル `bar.ml` あるいはインクルードパス上の `bar.ml` が存在しているときには、デフォルトの `ocamldep` はそのファイルの依存関係しか調べません。つまり、もし `ocamldep` を実行した時点では `bar.mly` しか存在しなかったとしたら、たとえ `bar.ml` と `bar.mli` が `bar.mly` から生成されたとしても、これらの依存関係は調べない (あるいは出力しない) のです。

現状の OMake では、この問題を解決するための 2 つの方法を提供しています。一つは生成されるファイルを手動で指定してあげることです。二つめは実験的な方法ですが、自動的に『隠れた』依存関係を調査することです。 `OCAMLDEP_MODULES_ENABLED` 変数はどちらの方法を使うのか、制御する変数です。この変数が偽であるときには、手動で指定するという方法が期待されます。また、真であるときには、自動的な調査を試みます。

OCamlGeneratedFiles, LocalOCamlGeneratedFiles

```
OCamlGeneratedFiles(files)
LocalOCamlGeneratedFiles(files)
```

OCAMLDEP_MODULES_ENABLED 変数が false に設定してある場合、 OCamlGeneratedFiles と LocalOCamlGeneratedFiles 関数は任意の OCaml ファイルが依存関係をスキャンする前に、生成される必要のあるファイルを指定します。例えば、 parser.ml と lexer.ml の両方が生成されるファイルであった場合、以下のように指定します。

```
OCamlGeneratedFiles(parser.ml lexer.ml)
```

OCamlGeneratedFiles 関数は **グローバル** です。この関数の引数は、プロジェクト上の任意の場所にある任意の OCaml ファイルの依存関係をスキャンする前に生成されます。 LocalOCamlGeneratedFiles 関数は OMake の通常のスコープルールに従います。

これらの関数は OCAMLDEP_MODULES_ENABLED 変数が真であるときにはなんの影響も与えません。

依存関係の解析中に生成されるファイルを自動的に調査

OMake が自動的に調査してくれるときに、いちいち手動で生成されるファイルを指定してあげるのは明らかに最適とは言えません。まずこれについて話すために、ファイルの自由なモジュール名 **のみ** を探しだす ocamldep について話し、その後内部でその結果を後処理することを伝えました。

この自動的な機構は OCAMLDEP_MODULES_ENABLED 変数が true に設定されているときに許可されます。通常、 OCAMLDEP_MODULES_ENABLED 変数は $(OCAMLDEP_MODULES_AVAILABLE) に設定されています。

この処理に依存している ocamldep の機構はバージョン 3.10 の OCaml かそれ以降のみに含まれています。一時的に、私たちは ocamldep を似せて修正したバイトコードバージョンの ocamldep-omake を用意しました。この似せて作った ocamldep は自動的に依存関係を調査してくれます。詳細は OCAMLDEP_MODULES_AVAILABLE と OCAMLDEP_MODULES 変数の項を参照し、さらにこれらの変数を正しく設定し、利用してください。

13.6.6 Menhir パーサジェネレータを使用

Menhir は ocamlyacc とほとんど互換性がとれており、かつ様々な改良がなされているパーサジェネレータです。機能の一部を以下にリストします (さらに知りたい方は Menhir のホームページ http://cristal.inria.fr/~fpottier/menhir/ を参照してください)。

- Menhir の解釈 (explanations) は、多くの人間が分かりやすくなるように改良されています。
- Menhir は複数のファイルにわたって文法を記述できます。これはまた、トークンの集合を複数の grammer が共有できることを意味しています。
- Menhir は Objective Caml モジュールによって記述されたパーサを提供することができます。
- jyh によって -infer オプションが追加されました。これによって、Menhir は生成時にあなたの文法における意味動作 (semantic actions) をチェック (typecheck) することができます。

ノート: 訳注: 見ての通り、訳が安定していません。すいませんがあまり信用しないでください

どのようにして ocamlyacc の代わりに Menhir を使用できるのでしょうか？

1. あなたのプロジェクトの適切な位置 (Menhir をどこでも使いたいのならプロジェクトトップの OMakefile) に、以下の定義を追加してください。

   ```
   MENHIR_ENABLED = true
   ```

2. 必要ならば、Menhir に追加させたいオプションを MENHIR_FLAGS に追加してください。

```
MENHIR_FLAGS += --infer
```

このセットアップによって、任意の .mly 拡張子のファイルは Menhir でコンパイルされます。

もしあなたの文法 (grammar) がいくつかのファイルにわたって分割されているのなら、あなたは MenhirMulti 関数を用いて、そのことを明示的に指定してあげる必要があります。

```
MenhirMulti(target, sources)
    target : 拡張子を除いたファイル名
    sources : 拡張子を除いた、文法を定義しているファイル群
```

例えば、文法を指定しているファイル a.mly と b.mly からパーサファイル parse.ml と parse.mli を生成したい場合は、以下のように記述します。

```
MenhirMulti(parse, a b)
```

OCamlLibrary

OCamlLibrary 関数は OCaml ライブラリをビルドします。

```
OCamlLibrary(<libname>, <files>)
```

<libname> と <files> は拡張子を **付けないで** 指定してください。

この関数はルールを定義しているすべてのターゲットのリスト (NATIVE_ENABLED が設定されているときは $(name)$(EXT_LIB) を含む) を返します。

以下のコードは (NATIVE_ENABLED が設定されている場合は) foo.cmx と bar.cmx から libfoo.cmxa ライブラリをビルドし、(BYTE_ENABLED が設定されている場合は) foo.cmo と bar.cmo から libfoo.cma をビルドします。

```
OCamlLibrary(libfoo, foo bar)
```

OCamlPackage

OCamlPackage 関数は OCaml パッケージをビルドします。

```
OCamlPackage(<name>, <files>)
```

<name> と <files> は拡張子を **付けないで** 指定してください。<files> は OCaml コンパイラに -for-pack <ident> フラグを加えた状態でコンパイルされます。

この関数はルールを定義しているすべてのターゲットのリスト (NATIVE_ENABLED が設定されているときは $(name)$(EXT_LIB) を含む) を返します。

以下のコードは (NATIVE_ENABLED が設定されている場合は) foo.cmx と bar.cmx から package.cmx パッケージをビルドし、(BYTE_ENABLED が設定されている場合は) foo.cmo と bar.cmo から package.cmo をビルドします。

```
OCamlPackage(package, foo bar)
```

OCamlLibraryCopy

OCamlLibraryCopy 関数はライブラリをインストール先にコピーします。

```
OCamlLibraryCopy(<tag>, <libdir>, <libname>, <interface-files>)
```

<interface-files> では INSTALL_INTERFACES 変数が真である場合にコピーする、追加のインターフェイスファイルを指定します。

OCamlLibraryInstall

OCamlLibraryInstall 関数はライブラリをビルドし、加えてインストール先にコピーします。

```
OCamlLibraryInstall(<tag>, <libdir>, <libname>, <files>)
```

OCamlProgram

OCamlProgram 関数は OCaml プログラムをビルドします。この関数はルールを定義している、すべてのターゲットの配列を返します ($(name)$(EXE) , $(name).run $(name).opt は NATIVE_ENABLED と BYTE_ENABLED 変数の値に依存しています)。

```
OCamlProgram(<name>, <files>)
```

以下の変数が利用されます。

- **OCAML_LIBS**

  リンカに渡す追加ライブラリ (拡張子を除く)。これらのファイルは対象のプログラムの依存先となります。

- **OCAML_OTHER_LIBS**

  リンカに渡す追加ライブラリ (拡張子を除く)。これらのファイルは対象のプログラムの依存先に **なりません** 。

- **OCAML_CLIBS**

  リンカに渡す C ライブラリ

- **OCAML_BYTE_LINK_FLAGS**

  バイトコードリンカに渡すフラグ

- **OCAML_NATIVE_LINK_FLAGS**

  ネイティブコードリンカに渡すフラグ

- **OCAML_LINK_FLAGS**

  両方のリンカに渡すフラグ

OCamlProgramCopy

OCamlProgramInstall 関数は OCaml プログラムをインストール先にコピーします。

```
OCamlProgramCopy(<tag>, <bindir>, <name>)
```

以下の変数が利用されます。

- **NATIVE_ENABLED**

  NATIVE_ENABLED 変数が設定されている場合、ネイティブコードの実行形式がコピーされます。そうでない場合はバイトコードの実行形式がコピーされます。

OCamlProgramInstall

OCamlProgramInstall 関数はプログラムをビルドし、加えてインストール先にコピーします。

```
OCamlProgramInstall(<tag>, <bindir>, <name>, <files>)
```

13.7 LaTeX ファイルのビルド

OMake では LaTeX ドキュメントのビルドをサポートします。このサポートには自動的に BiBTex を実行させ、PostScript や PDF ファイルを生成するためのサポートも含まれます。このセクションで列挙している関数を使うためには、まずあなたの OMakeroot ファイルに、以下の一行を追加してください。

```
open build/LaTeX
```

13.7.1 設定用の変数

以下の変数があなたのプロジェクト上で編集できます。

- **LATEX**

  LaTeX コマンド (デフォルトは latex)

- **TETEX2_ENABLED**

  TeTeX v.2 にある、発展的な LaTeX オプションを利用するかどうかを示すフラグです (デフォルトの値は最初に omake が LaTeX.src を読み込むことで決定されて、さらにインストールしてある LaTeX のバージョンに依存します)。

- **LATEXFLAGS**

  LaTex のフラグ (デフォルトの値は TETEX2_ENABLED 変数に依存する)

- **BIBTEX**

  BibTeX コマンド (デフォルトは bibtex)

- **MAKEINDEX**

  索引をビルドするコマンド (デフォルトは makeindex)

- **DVIPS**

  .dvi から PostScript に変換するコンバータ (デフォルトは dvips)

- **DVIPSFLAGS**

  dvips に渡すフラグ (デフォルトは -t letter)

- **DVIPDFM**

  .dvi から .pdf に変換するコンバータ (デフォルトは dvipdfm)

- **DVIPDFMFLAGS**

  dvipdfm に渡すフラグ (デフォルトは -p letter)

- **PDFLATEX**

  .latex から .pdf に変換するコンバータ (デフォルトは pdflatex)

- **PDFLATEXFLAGS**

  pdflatex に渡すフラグ (デフォルトは $`(LATEXFLAGS))

- **USEPDFLATEX**

  .pdf ドキュメントを生成するのに dvipdfm ではなく pdflatex を使うかどうかを示すフラグ (デフォルトは false)

13.7.2 LaTeX ドキュメントのビルド

LaTeXDocument

LaTeXDocument 関数は LaTeX ドキュメントを生成します。

```
LaTeXDocument(<name>, <texfiles>)
```

<name> と <texfiles> には拡張子を **付けないで** 指定してください。この関数は生成された .ps と .pdf ファイルのファイル名を返します。

以下の変数が利用されます。

- **TEXINPUTS**

  LaTeX の検索パス (ディレクトリの配列で、デフォルトは TEXINPUTS 環境変数から取得される)

- **TEXDEPS**

  このドキュメントに依存している追加ファイル

- **TEXVARS**

  OMake の TEXINPUTS 変数の値に基づいてアップデートすることになっている、環境変数の名前の配列。デフォルトは TEXINPUTS BIBINPUTS BSTINPUTS 。

TeXGeneratedFiles, LocalTeXGeneratedFiles

```
TeXGeneratedFiles(files)
LocalTeXGeneratedFiles(files)
```

TeXGeneratedFiles と LocalTeXGeneratedFiles 関数は任意の LaTeX ファイルの依存関係をスキャンする前に生成される必要のあるファイルを指定します。例えば、 config.tex と inputs.tex が両方とも生成されるファイルであった場合、以下のように指定します。

```
TeXGeneratedFiles(config.tex inputs.tex)
```

TeXGeneratedFiles 関数は **グローバル** です。この関数で指定したファイルは、任意の場所にある任意の TeX ファイルの依存関係をスキャンする前に生成されます。 LocalTeXGeneratedFiles 関数は OMake の通常のスコープルールに従います。

LaTeXDocumentCopy

LaTeXDocumentCopy 関数はインストール先にドキュメントをコピーします。

```
LaTeXDocumentCopy(<tag>, <libdir>, <installname>, <docname>)
```

この関数は .pdf と .ps ファイルだけコピーします。

LaTeXDocumentInstall

LaTeXDocumentInstall 関数はドキュメントをビルドし、加えてインストール先にドキュメントをコピーします。

```
LaTeXDocumentInstall(<tag>, <libdir>, <installname>, <docname>, <files>)
```

CHAPTER 14

自動設定用の変数と関数

OMake の標準ライブラリでは、異なったビルド環境下でのビルドを調整するための、自動設定 (autoconfiguring) 用のコードを書くことを手助けする、数多くの関数や変数を提供しています。

14.1 汎用的な自動設定関数

以下にリストしてある汎用的な関数では、あなたの『ビルド環境 (the properties of your build environment)』について調べることができます。これらの関数の中には、恐らくあなたがよく知っているであろう GNU autoconf ツールによく似ているものも含まれています。また、これらの関数は適切な `static.` ブロックで使うことを推奨しています (詳細は "*static.*" を参照してください)。

以下の汎用的な関数を使うためには、まずあなたの `OMakeroot` あるいは `OMakefile` ファイルに、以下の一行を追加してください。

```
open configure/Configure
```

14.1.1 ConfMsgChecking, ConfMsgResult

```
ConfMsgChecking(<msg>)
...
ConfMsgResult(<msg>)
```

`ConfMsgChecking` 関数は `--- Checking <msg>...` のような形のメッセージを **改行なしで** 出力します。`ConfMsgChecking` が実行されて、後にテストが終わったときには、`ConfMsgResult` 関数を使って結果を出力すべきです。

これらの関数を再定義したいという場合があるかもしれません。例えば、出力の形式を異なったものにしたいという場合や、メッセージをログファイルにコピーしたいというような場合です。

例:

```
static. =
   ConfMsgChecking(which foo to use)
   foo = ...
   ConfMsgResult($(foo))
```

14.1.2 ConfMsgWarn, ConfMsgError

```
ConfMsgWarn(<msg>)
ConfMsgError(<msg>)
```

警告やエラーメッセージをそれぞれ表示します。 ConfMsgError は OMake を中断させます。

14.1.3 ConfMsgYesNo, ConfMsgFound

```
flag = $(ConfMsgYesNo <bool expr>)
flag = $(ConfMsgFound <bool expr>)
```

ConfMsgFound 関数を使う前にまず ConfMsgChecking 関数を用いて「コマンドが使用できるかどうか調べます」ということをユーザに伝えます。その後、その結果を真偽値で ConfMsgFound 関数に渡します。ConfMsgFound は結果を適切な形 (“found” あるいは”NOT found”) に直し、 ConfMsgResult 関数を使って出力します。また引数として渡された結果を、加工しないでそのまま返します。

ConfMsgYesNo 関数は ConfMsgFound と似ていますが、出力がシンプルです (“yes” あるいは”NO”)。

14.1.4 TryCompileC, TryLinkC, TryRunC

```
success = $(TryCompileC <prog_text>)
success = $(TryLinkC <prog_text>)
success = $(TryRunC <prog_text>)
```

C プログラムの **コード** を引数として渡します。 TryCompileC , TryLinkC , TryRunC 関数は与えられた引数を『コンパイル/コンパイルとリンク/コンパイルとリンクと実行』できるかどうか試します。テストが成功した場合、この関数は真を返します。一方で、失敗した場合は偽を返します。

TryCompileC は C コンパイラを実行するときに CC , CFLAGS , INCLUDES 変数を使用します。 TryLinkC と TryRunC はそれに加えて LDFLAGS 変数を、C コンパイラとリンカを実行するときに使用します。しかし、/WX , -Werror , -warn-error のようなフラグは、たとえ CFLAGS 変数に含まれていたとしても、コンパイラに渡されることはありません。

これらの関数は結果を表示しません。また、通常の場合は適切な ConfMsgChecking 〜 ConfMsgResult 間で使用すべきです。

14.1.5 RunCProg

```
output = $(RunCProg <prog>)
```

RunCProg 関数は TryRunC 関数と似ていますが、この関数はプログラムの出力を返します (プログラムのコンパイルや実行に失敗した場合は false が返されます)。

ノート: 訳注: 原文では “RunCProg is similar to the RunCProg function, ...” となっていましたが、恐らくこれは TryRunC の間違いではないかと思われます

14.1.6 CheckCHeader, VerboseCheckCHeader

```
success = $(CheckCHeader <files>)
success = $(VerboseCheckCHeader <files>)
```

TryCompileC 関数を用いることで、あなたの C コンパイラが指定したヘッダファイルの位置をつきとめ、さらに処理できるかどうか調べることができます。 TryCompileC 関数はヘッダファイルをインクルードする前に <stdio.h> をインクルードします。

両方の関数は真偽値を返します。 CheckCHeader 関数は結果を表示しません。 VerboseCheckCHeader 関数はテストと結果を表示するのに ConfMsgChecking と ConfMsgResult 関数を使います。

例:

```
static. =
   NCURSES_H_AVAILABLE = $(VerboseCheckCHeader ncurses.h)
```

14.1.7 CheckCLib, VerboseCheckCLib

```
success = $(CheckCLib <libs>, <functions>)
success = $(VerboseCheckCLib <libs>, <functions>)
```

TryLinkC 関数を用いることで、与えられたライブラリのリンク時に、あなたの C コンパイラとリンカが、与えられた関数を見つけられるかどうか調べることができます。 TryLinkC 関数は -l フラグを使ってコンパイラに <libs> を渡します。

両方の関数は真偽値を返します。 CheckCLib 関数は結果を表示しません。 VerboseCheckCLib 関数はテストと結果を表示するのに ConfMsgChecking と ConfMsgResult 関数を使います。

例:

```
static. =
    NCURSES_LIB_AVAILABLE = $(VerboseCheckCLib ncurses, initscr setupterm tigetstr)
```

14.1.8 CheckProg

```
success = $(CheckProg <prog>)
```

あなたのパス中に <prog> が存在しているかどうかを調べます。この関数はテストと結果を表示するのに ConfMsgChecking と ConfMsgResult 関数を使います。

14.2 autoconf スクリプトを OMake 用に翻訳する

上の関数のいくつかは、 autoconf に存在しているものと非常に似ています。以下はそのような関数のための簡単な翻訳テーブルです。

- AC_MSG_CHECKING は ConfMsgChecking 関数と非常に似ています。
- AC_MSG_RESULT は ConfMsgResult 関数と非常に似ています。
- AC_MSG_WARN は ConfMsgWarn 関数と非常に似ています。
- AC_MSG_ERROR は COnfMsgError 関数と非常に似ています。
- AC_TRY_COMPILE は TryCompileC 関数といくらか似ています。ただし、 TryCompileC 関数は真偽値を返し、 C 言語のみに機能するという点が異なります。
- AC_TRY_LINK は TryLinkC 関数と似ています。
- AC_TRY_RUN は TryRunC 関数と似ています。

14.3 事前に用意された設定テスト

OMake では数多くの設定テスト (configuration tests) が標準ライブラリに含まれています。あなたのプロジェクトでこれらを使うためには、単純に対象の `OMakefile` に、設定テストを記述したビルドファイルを『`open` (詳細は "**ファイルのインクルード**" を参照してください)』するだけです。そうすれば、OMake を初めて実行した時点で設定テストが実行されます。これは、あなたのプロジェクト上で二回以上設定テストが `open` されるわけではない点に注意してください。たとえ二回以上呼び出したとしても、このテストは一回だけしか実行されません。

14.3.1 NCurses ライブラリの設定

あなたの `OMakefile` に `open configure/ncurses` を加えることで、以下の自動設定変数を利用できます。

- **NCURSES_AVAILABLE**

  `curses.h`, `term.h` ヘッダファイルと `ncurses` ライブラリが見つかった場合に真となる真偽値

- **NCURSES_TERMH_IN_NCURSES**

  `term.h` が `<term.h>` の代わりに `<ncurses/term.h>` としてインクルードされていた場合に真となる真偽値

- **NCURSES_CFLAGS**

  ncurses コードをコンパイルする際に用いる `CFLAGS` の値。`NCURSES_AVAILABLE` と `NCURSES_TERMH_IN_NCURSES` がそれぞれ真である場合には `-DNCURSES` と `-DTERMH_IN_NCURSES` が含まれます。

- **NCURSES_CLIBS**

  ncurses コードをリンクする際に用いる `LDFLAGS` の値。ncurses が見つかって、それ以外が空のままである場合には通常 `-lncurses` が含まれます。

14.3.2 ReadLine ライブラリの設定

あなたの `OMakefile` に `open configure/readline` を加えることで、以下の自動設定変数を利用できます。

- **READLINE_AVAILABLE**

  `readline/readline.h`, `readline/history.h` ヘッダと `readline` ライブラリが見つかった場合に真となる真偽値

- **READLINE_GNU**

  readline ライブラリの GNU バージョン (BSD バージョンではなく) が見つかった場合に真となる真偽値

- **READLINE_CFLAGS**

  readline コードをコンパイルする際に用いる `CFLAGS` の値。`READLINE_AVAILABLE` と `READLINE_GNU` がそれぞれ真である場合には `-DREADLINE_ENABLED` と `-DREADLINE_GNU` が含まれます。

- **READLINE_CLIBS**

  readline コードをリンクする際に用いる `LDFLAGS` の値。readline が見つかって、それ以外が空である場合には通常 `-lncurses -lreadline` が含まれます。

14.3.3 Snprintf の設定

あなたの `OMakefile` に `open configure/snprintf` を加えることで、以下の自動設定変数を利用できます。

- **SNPRINTF_AVAILABLE**

  標準 C ライブラリで snprintf 関数が利用可能であるかどうかを示す真偽値

CHAPTER 15

OSHシェル

OMake はまた、独立して動くコマンドラインインタープリタ `osh` を含んでいます。これはインタラクティブなシェルとして使うことができます。このシェルは omake と同様の構文で利用することができ、さらに Win32 を含む、omake がサポートしているすべてのプラットフォーム上で同様の機能を提供します。

15.1 起動時

起動時に、osh は存在しているのであれば `~/.oshrc` を読み込みます。このファイルの構文は `OMakefile` と同様です。また、以下の追加された変数は、osh 上で重要な意味を持ちます。

- **prompt**

  `prompt` 変数はコマンドラインプロンプトを指定します。この変数は単純な文字列を指定できます。

  ```
  prompt = osh>
  ```

  あるいは、引数をもたない関数として定義することもできます。

  ```
  prompt() =
      return $"<$(USER):$(HOST) $(homename $(CWD))>"
  ```

  後者のプロンプトの例は以下のようになります。

  ```
  <jyh:kenai.yapper.org ~>cd links/omake
  <jyh:kenai.yapper.org ~/links/omake>
  ```

  プロンプト上にターミナルのエスケープ文字のような『見えない』文字を含ませたい場合は、`prompt-invisible` 関数 (*prompt-invisible*) を使ってラップさせなければなりません。例えば、サポートしているターミナル上でプロンプトを太字にしたい場合、以下のように書くことができます。

  ```
  prompt =
     bold-begin = $(prompt-invisible $(tgetstr bold))
     bold-end = $(prompt-invisible $(tgetstr sgr0))
     value $(bold-begin)$"osh>"$(bold-end)
  ```

- **ignoreeof**

  `ignoreeof` が `true` の場合、 `osh` はターミナルの”end-of-file(EOF)” で終了しません (Unix システム上では通常 `^D` となります)。

15.2 エイリアス

コマンドのエイリアスは Shell. オブジェクトに関数を追加することによって定義できます。以下のエイリアスは -AF オプションを ls コマンドに追加します。

```
Shell. +=
   ls(argv) =
      "ls" -AF $(argv)
```

クオートされたコマンドはエイリアス展開の影響を受けません。 "ls" のようにクオーテーションをつけることで、再帰的にエイリアスとなることを防ぎます。

15.3 インタラクティブな構文

osh でのインタラクティブな構文はインデントという一つの例外を除いて、 OMakefile の構文と同様です。まず、インデントされたブロックの前にある行は、必ず行の終わりにコロン:をつける必要があります。次に、. を行の終わりにつけるか ^D を使うことで、対象のブロックを終了させます。以下の例では、最初の行の if true はコロンをつけていないため、内容のブロックを持つことはできません。

```
# 以下の if は内容を持ちません
osh>if true
# 以下の if は内容を持ちます
osh>if true:
if>       if true:
if>          println(Hello world)
if>          .
Hello world
```

インデントされたブロックの中にいる際には、プロンプトをいくらか修正する必要があり、かつ osh は自動的にテキストをインデントすることに注意してください。

また、コロン修飾子はファイルにも適用できますが、必須ではありません。

OMake コマンドラインオプション

```
omake [-j <count>] [-k] [-p] [-P] [-n] [-s] [-S] [-w] [-t] [-u] [-U]
[-R] [--verbose] [--project] [--depend] [--progress] [--print-status]
[--print-exit] [--print-dependencies] [--show-dependencies <target>]
[--all-dependencies] [--verbose-dependencies] [--force-dotomake] [--dotomake
<dir>] [--flush-includes] [--configure] [--save-interval <seconds>]
[--install] [--install-all] [--install-force] [--version] [--absname]
[--output-normal] [--output-postpone] [--output-only-errors] [--output-at-end]
filename...  [var-definition...]
```

A.1 一般的な使い方

ブーリアンオプション (例えば `-s` , `--progress` など…) は意味を逆転させる接頭辞 `--no` を付与することができます。例えば、オプション `--progress` は『プログレスバーを表示する』ことを示していますが、オプション `--no--progress` は『プログレスバーを表示しない』ことを示しています。

複数の重複するオプションが指定されていた場合、最後のオプションが OMake でのふるまいを決定します。以下のコマンドラインでは、最後の `--no-S` が前の `-S` オプションを打ち消します。

```
% omake -S --progress --no-S
```

A.2 出力のコントロール

A.2.1 -s

`-s`

コマンドが実行された場合でも、決してコマンドの内容を表示しません (“silent”)。

A.2.2 -S

`-S`

コマンドが出力を生成したり、失敗したりするまでは、コマンドの内容を表示しません。このオプションはデフォルトで有効です。

A.2.3 -w

`-w`

コマンドが実行される時点でのディレクトリの情報を `make` フォーマットで表示します。これはエラー場所を特定するため、make スタイルのディレクトリの情報を切望しているエディターにとって、とても有用なオプションです。

A.2.4 -progress

`--progress`

進捗状況 (プログレスバー) を表示します。このオプションは、OMake の出力先 (`stdout`) がターミナルとなっている場合はデフォルトで有効となり、OMake の出力先がリダイレクトされている場合はデフォルトで無効となります (Windows を除く)。

A.2.5 -print-status

`--print-status`

ステータスライン (+ と - ライン) を表示します。

A.2.6 -print-exit

`--print-exit`

コマンドが成功した場合には終了コードを表示します。

A.2.7 -verbose

`--verbose`

OMake の出力を詳細に表示します。このオプションは `--no-S --print-status --print-exit VERBOSE=true` と等価です。

A.2.8 -output-normal

`--output-normal`

ルールコマンドを実行した時点で、コマンドの出力を OMake の出力へと即座に受け渡します。これは `--output-postpone` か `--output-only-errors` が有効になっていない限りデフォルトで有効です。

A.2.9 -output-postpone

`--output-postpone`

ルールが完了した際に、出力を一つのブロックとして一気に表示します。これは複数のサブプロセスの出力をまとめてくれる (garbled) `-j` オプション (*-j* を参照) と一緒に使うと有用です。コマンドの出力は一つにまとめられて表示します。

ノート: `--output-postpone` を有効にした際、デフォルトで `--output-normal` オプションが無効になります。この仕様は、インタラクティブな入力を要請するコマンドを使いたいような場合に問題となるでしょう。 `--output-postpone` が有効になって、さらに `--output-normal` が無効になっているような場合、

このようなコマンドのプロンプトを表示することはなくなるので、ビルドがなぜ『固まって』いるのか理解することは殆ど困難となります。このような状況に直面した際、あなたは --process フラグ (*-progress* を参照) を用いて、ビルドが実行されているのかどうか確認すべきです。

A.2.10 -output-only-errors

--output-only-errors

--output-postpone と似ていますが、このオプションは成功したコマンドの出力を表示しません。つまりユーザが望んでいない出力を減らすことができるので、あなたは任意のエラーが出た箇所のみに集中できます。

A.2.11 -output-at-end

--output-at-end

任意のルールやコマンドが失敗した場合、失敗したコマンドの出力を OMake のビルドが終了した際に表示します。これは -k , -p , -P オプションのうち一つでも有効としている場合に、特に有用となります。

このオプションはデフォルトで無効となっています。しかしながら、 -k オプションが有効となっているような場合ー明示的、あるいは -p/-P オプションを経由した場合ー --output-at-end オプションはデフォルトで有効となります。

A.2.12 -o

-o [01jwWpPxXsS]

上記の出力オプションを簡単に指定するために、 -o オプションが出力オプションの代替手段として提供されています。 -o オプションは文字のシーケンスで構成された引数が必要となります。文字は左から右へ読み込まれ、各々の文字は出力オプションの集合を指定しています。一般的に、大文字はオプションを有効にして、小文字はオプションを無効化します。

- **0**

  -s --output-only-errors --no-progress と等価です。

  このオプションは omake の出力ができるだけサイレントとなるように指定します。任意のエラーがビルド中に生じた場合、出力はビルドが終了するまで延期されます。成功したコマンドの出力は表示されません。

- **1**

  -S --progress --output-only-errors と等価です。

  このオプションは上の『サイレント』なバージョンよりも、もう少し寛容な形になっています。成功したコマンドの出力は表示しません。失敗したコマンドの出力はコマンドが完了した後で、即座に表示します。失敗したコマンドの出力は 2 回表示します。まず 1 回目はコマンドが完了した際に表示し、2 回目はビルドが完了した際に表示します。また、プログレスバーも表示するので、あなたはビルドが実行されているのかどうか確認することができます。もしプログレスバーを表示したくないのであれば、 p オプションを含めてください (例: omake -o 1p)。

- **2**

  --progress --output-postpone と等価です。

  このオプションはさらに寛容な形となっており、成功したコマンドの出力も表示します。これは -j オプションを用いて出力を重ね合わせたくない (deinterleaving) ような場合にしばしば有用となります。

- **W**

  -w と等価です。

- **w**

  --no-w と等価です。

- **P**

  -progress と等価です。

- **p**

  --no--progress と等価です。

- **X**

  --print-exit と等価です。

- **x**

  --no-print-exit と等価です。

- **S**

  -S と等価です。

- **s**

  --no-S と等価です。

A.3 ビルドオプション

A.3.1 -k

-k

ビルドコマンドが失敗したとしても中断せず、可能な限りプロジェクトのビルドを続けます。このオプションは -p あるいは -P オプションを指定した場合、暗黙的に有効となります。さらに、このオプションは暗黙的に --output-at-end オプションを有効にします。

A.3.2 -n

-n

このオプションによって、プロジェクトがビルドされる場合にどのような手順が踏まれるのか、実際に確認することができます。

A.3.3 -p

-p

ファイルシステムの変化を監視し、ビルドが成功するまで実行し続けます。このオプションを指定した場合、omake はソースファイルが変更された際にはいつでもビルドを開始します。また、暗黙的に -k オプションを有効にします。

A.3.4 -P

-P

ファイルシステムの変更点を永久に監視します。このオプションを指定した場合、 omake はソースファイルが変更された際にはいつでもビルドを開始します。また、暗黙的に -k オプションを有効にします。

A.3.5 -R

-R

カレントディレクトリを無視し、ルートディレクトリからプロジェクトをビルドします。 omake をプロジェクトのサブディレクトリから実行し、さらに明示的なターゲットをコマンドライン上から与えていないような場合、OMake は通常カレントディレクトリとそのサブディレクトリ内にあるファイルのみをビルドします (正確には、すべてのカレントディレクトリ内の .DEFAULT ターゲットをビルドします)。 -R オプションを指定した場合、まるで omake がプロジェクトのルート上から実行しているかのようにビルドを行います。

言い換えると、 -R オプションを用いることで、コマンドライン上から指定したすべての関連するターゲットを、(カレントディレクトリに関連付ける代わりに) プロジェクトのルートに関連付けます。なんのターゲットもコマンドライン上に指定していない場合には、プロジェクト上のすべての .DEFAULT ターゲットが (カレントディレクトリに関係なく) ビルドされます。

A.3.6 -t

-t

強制的に omake のデータベースを更新します。

A.3.7 -U

-U

キャッシュされたビルド情報を用いません。このオプションを指定した場合、強制的にプロジェクト全体をリビルドします。

A.3.8 -depend

--depend

キャッシュされた依存関係の情報を用いません。このオプションを指定した場合、強制的にファイルの依存関係を再スキャンします。

A.3.9 -configure

--configure

OMake ファイルに含まれている static. セクションを、キャッシュされた結果を用いる代わりに再実行します。

A.3.10 -force-dotomake

```
--force-dotomake
```

常に $HOME/.omake の .omc キャッシュファイルを用います。

A.3.11 -dotomake

```
--dotomake <dir>
```

$HOME/.omake を用いる代わりに、指定したディレクトリに置かれた .omc キャッシュファイルを用います。

A.3.12 -j

```
-j <count>
```

平行して複数のビルドコマンドを実行します。 count では同時に実行するコマンドの限界数を指定します。加えて、 server=count の形でリモートサーバのコマンドを実行することもできます。例えば、オプション -j 2:small.host.org=1:large.host.org=4 は 2 つのジョブをローカルで実行し、さらに 1 つをサーバ small.host.org 上、4 つを large.host.org 上で実行します。それぞれのリモートサーバはプロジェクト上の同一の場所で実行しなければなりません。

リモートサーバでの実行は現在実験的な機能として搭載しています。NFS のようなリモートファイルシステムでは、十分なファイルの整合性を保つことはできません。

A.3.13 -print-dependencies

```
--print-dependencies
```

コマンドライン上でのターゲットの、依存関係に関する情報を表示します。

A.3.14 -show-dependencies

```
--show-dependencies <target>
```

target をビルドする場合、依存関係に関する情報を表示します。

A.3.15 -all-dependencies

```
--all-dependencies
```

オプション --print-dependencies あるいは --show-dependencies が指定されている場合、動的な依存関係も表示します。これは、すべての依存関係を再帰的に表示することを表しています。オプション --print-dependencies , --show-dependencies のどちらも指定していない場合、このオプションはなんの影響も与えません。

A.3.16 -verbose-dependencies

```
--verbose-dependencies
```

オプション --print-dependencies あるいは --show-dependencies が指定されている場合、各々の依存関係のリストを表示します。出力はとても冗長となり、ファイルへのリダイレクトも考慮されるようにな

ります。オプション --print-dependencies , --show-dependencies のどちらも指定していない場合、このオプションはなんの影響も与えません。

A.3.17 -install

--install

デフォルトのファイル OMakefile と OMakeroot をカレントディレクトリにインストールします。典型的な使い方としては、カレントディレクトリ内で OMake のプロジェクトを開始しようとする際に、このオプションを用います。

A.3.18 -install-all

--install-all

OMakefile と OMakeroot をインストールし、さらにデフォルトの OMakefile をカレントディレクトリ内のサブディレクトリにインストールします。その際に、サブディレクトリのリストをフィルタリングするため cvs(1) のルールが用いられます。例えば、 OMakefile は CVS , RCCS などのディレクトリ内にはコピーされません。

A.3.19 -install-force

--install-force

通常、 omake は既存の OMakefile を上書きする前に警告を行います。このオプションが与えられている場合、すべてのファイルは強制的に警告を発することなく上書きされます。

A.3.20 -absname

--absname

ファイル名は絶対パスとして展開されるようになります。

ノート: これは実験的なオプションで、廃止される恐れがあります。

A.3.21 変数の定義

name=[value]

omake の変数は name=value の形で、コマンドライン上から指定することもできます。例えば、 CFLAGS 変数はコマンドライン上から、引数に CFLAGS="-Wall -g" を指定することで定義することもできます。

A.4 さらなるオプション

これらのオプションに加えて、 omake ではコマンドライン上で利用できる、数多くのデバッグフラグをサポートしています。これらのフラグの概要について知りたい方は、 omake --help を実行してください。

A.5 環境変数

A.5.1 OMAKEFLAGS

OMAKEFLAGS 変数が定義されている場合、 OMAKEFLAGS で指定したオプションの集合が、コマンドライン上から指定したものと同様に扱われます。

A.5.2 OMAKELIB

OMAKELIB が定義されている場合、 OMAKELIB の環境変数は OMake での標準ライブラリの場所を表しています。これは Pervasives.om などを含んだディレクトリとなっています。Unix システム上では、これは大まかに /usr/lib/omake か /usr/local/lib/omake へ関連付けられ、Win32 システム上では c:\Program Files\OMake\lib へ関連付けられます。

この環境変数が定義されていない場合、 omake は設定されたデフォルトのパスを用います。通常はこの環境変数を未定義のままにしておいて構いません。

A.6 関数

A.6.1 OMakeFlags

OMakeFlags 関数はオプションの集合を OMakefile の内部で修正するために用いられます。このオプションはコマンドライン上で指定するのと同様にして指定する必要があります。例えば、ある特定のプロジェクト内で出力をサイレント、かつプログレスバーを表示したい場合、あなたは OMakefile 上に以下の行を追加します。

```
OMakeFlags(-S --progress)
```

この関数によって指定したオプションは、まるで変数のようにスコープ化されます。例えば、OMake での出力を (プロジェクト全体で有効にする代わりに) 一つのルール上でサイレントにしたい場合は、オプションを適用させる範囲をそのままスコープ化します。

```
section
    # foo が生成される際に、コマンドラインの出力を表示しません
    OMakeFlags(-S)

    foo: fee
       echo "This is a generated file" > foo
       cat fee >> foo
       chmod 555 foo
```

A.7 オプションの処理過程

omake が実行された際、オプションは以下の順に処理されます。

1. OMAKEFLAGS 環境変数によって指定したすべてのオプションが、プロジェクト全体に作用します。
2. コマンドラインによって指定したすべてのオプションが、プロジェクト全体に作用します。
3. OMakeFlags 関数によって呼び出された任意のオプションが、プロジェクトの一部分に作用します。

A.8 .omakerc

$(HOME)/.omakerc が存在する場合、任意の OMakefiles が読み込まれる前にまず $(HOME)/.omakerc が読み込まれます。このファイルは、ユーザが自由に OMake をカスタマイズする用途として頻繁に用いられます。例えば、 OMAKEFLAGS 環境変数を定義する代わりに、以下の行を .omakerc に加えることもできます。

```
$(HOME)/.omakerc:
    # プライベートなオプションを記述
    OMakeFlags(-S --progress)
```