GNU Make
第3版

Robert Mecklenburg 著
矢吹 道郎 監訳
菊池 彰 訳

O'REILLY®
オライリー・ジャパン

THIRD EDITION

Managing Projects with GNU Make

Robert Mecklenburg

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

For Ralph and Buff

監訳者まえがき

makeの長い歴史と同様、オライリーのmakeの本にも長い歴史があります。

makeの初版本の翻訳書が出版されたのが1990年（啓学出版）、makeの改訂版の翻訳書が出版されたのが1992年（啓学出版）、そして、makeの改訂版が復刊されたのが1997年（オライリー・ジャパン）になります。初版から既に15年たとうとしていますが、改訂版はいまだに現役の息の長い出版物となっています。つまり、それだけmakeというユーティリティが開発環境においてなくてはならないものであることの証明となっていると言えるでしょう。

1970年代～1980年代、Unixを使う人だけの便利なユーティリティであったmakeも、そのコマンドの優位性を認められ、今や本書にもあるようにWindowsの開発環境を含め幅広く利用されるようになってきています。これには、GNUプロジェクトによるGNU Makeが大きく貢献しています。

現在もっとも広く使われているmakeのひとつである、本書が扱っているGNU Makeは、makeを元にGPLの許諾条件の下にソースプログラムを配布可能としたばかりでなく、実に多くの機能強化が行われ、基本機能は同じでも、別ものと言ってよいほどで、ほとんどプログラミング言語であるかのようです。ソースプログラムが10個程度の小さなソフトウェア製作では元々のmakeで満足できても、多くの人間がかかわるようなソフトウェアの製作ではGNU Makeを利用せざるを得ません。

本書は原著者が実際の大きなソフトウェア製作プロジェクトでGNU Makeを利用した経験を元に、実に多くのGNU Makeの活用の手法を示し解説しています。特に本書は実際に解決しなくてはならなくなるであろうさまざまなケースをあげ、有効な解決例を示してくれます。もしも何らかの問題に遭遇したら是非本書を開いてみて下さい。きっと解決のヒントを得られると思います。

最後に、本書翻訳にあたりお骨折りいただいたオライリー・ジャパンの赤池氏に感謝致します。

2005年11月
矢吹道郎

まえがき

makeユーティリティは、常に傍らに控え、常に忠実である有能な奉公人のようなものです。多くの小説や映画で欠くべからざる名脇役のように、初めのうちは半端な仕事を少し担うだけで評価されることのない存在です。しかし次第に大事業の多くの部分を取り仕切るようになります。

私が関与するあらゆるプロジェクトにおいてmakeを開発の中心に置くという作業の最終局面にさしかかっていたころ、私の上司であり最初の『Managing Projects with make』の著者であるSteve Talbottは、私の取り組みに気づき第2版を書かないかと勧めてくれました。それは私にとって貴重な成長体験（であると同時にちょっとした波乱の日々）であり、すばらしいO'Reillyの世界への入り口となりましたが、我々はこの成果がどのくらい市場で支持されるのかまったく予想もつきませんでした。1つの版が13年もの間、受け入れられるものでしょうか。

ずっと以前、私がまだプロのテクニカルライタであったころに思いをはせると、私は『Managing Projects with make 第2版』が出版されて以来、makeの進化についてまとめたリストを前にして深く考え込んでいました。

- GNU版のmake。これは第2版を出版したころにはすでに多くの優れたプログラマにより選択され、広く普及し、事実上の業界標準となっていました。
- GNU/Linuxの勃興がGNU版のmakeを含めたGNUコンパイラツール群をより一般的にしたことを考慮しました。その1つの例が、本書の11章でも書かれていますが、Linuxのカーネルの構築がGNU makeの提供する拡張機能に深く依存しているという事実です。
- BSDの1種であるDarwinがMac OS Xのコアとして採用されたことにより、GNUツール群とGNU make優位性の増加傾向に拍車がかかったことも1つの要素です。
- 頑強で誤りがなくポータビリティに富み柔軟な使い方をするための技がいろいろと発見されたことも見逃せません。大きなプロジェクトによくある問題に対する標準的な解が、プログラミングコミュニティの中で醸成されてきました。そういった解法の多くを、伝承の領域から本書が行うような文書化された世界へ持ち込む時期がきたのです。

- makeが開発されたときには存在していなかったC++やJava™に対して、makeを適応させるための新しい手法が必要とされていたことにも注目しなければなりません。時の流れを反映させるために、かつてオリジナルのmakeは次の2つの機能を提供しました。1つはいまだにその痕跡を残していますが、FORTRANに対する2つの改良版をサポートする機能、そしてどちらかといえばあまり効果的ではなかったSCCSとの統合機能です。
- 奇妙な点は多いものの、makeはほぼすべての開発プロジェクトにおける重要なツールとして生き残っています。13年前、makeに対する多くの（そして洞察に富んだ）批判のどれもが今日の状況を予測していませんでした。雨後の筍のごとくこれまで何度もmakeの代替物が登場してきました。新しいツールはどれもmakeのデザインからくる制限を迂回するための機能を提供しており、そのほとんどは実際に巧妙でかつ賞賛に値するものです。それでも単純さによりmakeは最高の地位を保ち続けています。

このような傾向を見るにつけ、この10年もの間、私の心の奥底には『Managing Projects with make』の新版を書くという考えがありました。しかし、だれか私よりもっとプロとしての広い経験を持つ者が必要であるとの考えも持っていました。そしてついに専門家としての知識を豊富に持つRobert Mecklenburgが現れ、我々O'Reillyは諸手をあげてこれを歓迎したのです。私は喜んで彼に執筆の任を譲り、本書のコピーライトのページに登場して横から口を出すだけの役割に退きました（ちなみにGNU makeがGPLを採用していることを反映して本書はGNU Free Documentation Licenseの元に置いてあります）。

Robertは非常に控えめで自分の博士号をひけらかしたりしませんが、彼の持つ深く正確な思考力は本書を通して明確に示されています。おそらく本書にとってさらに重要なことは、彼が実用性に焦点を当てていることでしょう。makeが読者の役に立つように最大限の努力をしただけでなく、その努力は効率的であることに注意を払うことに始まりmakefileを原稿の一部として使うことでタイプミスを排除するところにまで及んでいます。

すばらしい瞬間です。O'Reillyにおける最も初期のそして最も息の長い書籍の新版が完成しました。くつろいで本書を楽しんでください。気取らないちょっとしたツールが、ほとんどあらゆるプロジェクトの背後で想像もつかないような力を発揮していることがわかるでしょう。古ぼけて納得いかないmakefileに甘んじることなく、今日持ちうる最高の機能を活用してください。

——Andy Oram
Editor, O'Reilly Media
August 19, 2004

第3版にむけて

第3版までの道のり

筆者が最初にmakeに触れたのは、1979年のバークレーで大学院生のときでした。当時「最新」の装備で仕事ができることにワクワクしていました。それは128キロバイトのメモリを積んだDEC PDP 11/70にADM 3a「グラスtty」がつながり、20人ものユーザが同時にバークレーUnixを使っていました。課題の締め切りが近くなると、ユーザ名を入力してからログインが完了してプロンプトが出るまで5分もかかったことがあるのを覚えています。

大学院を卒業した後、1984年に筆者はまたUnix上で仕事をすることになりました。そのときはNASAのエイムズ研究センタにプログラマとして雇われていました。我々はUnixが使えるマイクロプロセッサを搭載した最初のコンピュータを購入しましたが、それは（68010や68020ではなく）68000のCPUに1メガバイトのメモリを搭載したもので、同時にたった6人のユーザがバージョン7のUnixを使用していました。研究所での最後の仕事は人工衛星データの対話型分析システムで、Cとyacc/lexと、そしてもちろんmakeを使って作るものです。

1988年までに大学に戻りスプラインベースの幾何学モデリングツールを作るプロジェクトに参加していました。このシステムはC言語で120,000行もの規模を持ち、20以上の実行ファイルに分かれていました。構築システムはmakefileのテンプレートを使い、テンプレートはgenmake（考え方はimakeと似ていました）と呼ばれる自作のツールにより処理されて通常のmakefileとなります。このツールはファイルのインクルードと条件付きコンパイルと、ソースとバイナリツリーを管理するためのいくつかの機能を提供していました。構築システムとして完備するには、このような補助ツールがmakeには必要であると当時は信じられていました。数年後、GNUプロジェクトとGNU makeについて知るにつれ、補助ツールはおそらく必要ないことがわかってきました。テンプレートや生成ツールを使わなくても済むように筆者は構築システムを作り直しました。残念なことに、その構築システムの保守を次の4年間でやめてしまいました（その日までおろかにも同じようなことを繰り返していました）。この構築システムは5つの異なるUnixシステムに移植され、ソースツリーとバイナリツリーの分

離と夜間の自動構築の機能を備え、開発者が必要とするオブジェクトファイルを構築システムから自動的に補充するようにもなっていました。

筆者がmakeと次に興味深い出会いをしたのは1996年のことで、商用のCADシステムの開発を行っていたときです。そこでの仕事は2,000,000行ものC++コード（加えて400,000行のLisp）をMicrosoftのC++コンパイラを使いUnixからWindows NTに移植するというものでした。Cygwinプロジェクトを知ったのはこの時期です。移植作業の副産物として、構築システムがNTをサポートするように作り変えました。この構築システムはソースツリーとバイナリツリーが分離され、Unix的な部分を多く持ち、幾つかのグラフィックスをサポートし、夜間の自動構築機能を備えていました。またリファレンスツリーを活用することで、開発者はソースの一部だけをチェックアウトしていればよかったのです。

2000年にはJavaを使う情報管理システムの研究所で働き始めました。ここはそれまでに長い間働いてきたような環境とはまったく違っていました。プログラマの多くはWindowsをバックグランドとし、そのほとんどはJavaが最初に覚えたプログラミング言語だったようです。構築システムは商用の統合開発環境（Integrated Development Environment：IDE）が生成したファイルでほとんどが構成されていました。プロジェクトのファイルはチェックインされますが、それが「そのままで」動作することは滅多になく、プログラマはそれぞれ自分の席で構築に関する問題を解決していました。

もちろん筆者はmakeを使って構築システムを作り始めましたが、1つ奇妙なことが起きました。開発者の多くはコマンドラインツールの使用に乗り気ではありませんでした。さらに環境変数、コマンドラインオプション、などの考え方に関するしっかりとした理解を欠いており、そしてプログラムを構築するためのツールを知りませんでした。IDEがすべて覆い隠していたのです。こういった問題に対処するために、筆者の作る構築システムはもっと複雑になっていきます。よりよいエラーメッセージを加え、前提条件をチェックし、開発者の設定を管理し、IDEのサポートをするようになりました。

こんなことを続けている間、GNU makeのマニュアルを何十回と読み返しました。その他の資料を探すうちに、本書の第2版に出会ったのです。有益な題材が満載でしたが、悲しいほどにGNU makeについては何も書かれていませんでした。第2版が書かれた時代を考慮すれば、別に驚くことではありません。書かれている内容は時を経ても色あせていませんが、2003年の段階では書き直す必要がありました。第3版はまずGNU makeに焦点を当てています。Paul Smith（GNU makeの保守担当者）はこう書いています。「ポータビリティのあるmakefileを書くことに悩むのはやめて、ポータビリティのあるmakeを使おう。」

本書の内容

1章から5章までは、ある程度の詳しさでGNU makeの機能と使い方を学びます。

「1章　簡単なメイクファイルを書いてみよう」では、シンプルでかつ完結した例を通して、makeを紹介します。ここではターゲットや必須項目といった基本的な概念とmakefileの文法を説明します。最初のmakefileを書くのに必要な知識がここで得られるはずです。

「2章　ルール」では、ルールの文法について論じます。旧来のサフィックスルールに加えて明示的ルールとパターンルールについて詳細に説明します。特殊ターゲットと簡単な依存関係の生成についても、2章で扱います。

「3章　変数とマクロ」では、単純変数と再帰変数を扱います。makefileの構文がどのように解析されるのか、いつ変数が展開されるのかを論じ、makefile内で条件判断を行うための命令を紹介します。

「4章　関数」では、バラエティに富んだGNU makeの組み込み関数を調べます。ユーザ定義関数についても、取るに足らないものから高度な機能を示すためのものまで豊富な例とともに紹介します。

「5章　コマンド」では、コマンドスクリプトの詳細を示し、スクリプトがどのように構文解析され評価されるのか説明します。ここでは、コマンドの修飾子、コマンドの終了ステータスの確認、コマンドの実行環境についても扱います。またコマンドラインの制限と、その対処方法についても言及します。この5章までで、本書で取り上げるGNU makeのすべての機能が登場します。

6章から12章までは、makeを大きなプロジェクトで使う方法、ポータビリティ、デバッグなどの、より大きな話題を取り上げます。

「6章　大きなプロジェクトの管理」では、大きなプロジェクトをmakeで構築する際に出会う問題について論じます。最初に扱うのはmakeの再帰起動ですが、同じことをmakeを1回だけ起動し非再帰的なmakefileで実行する方法にも触れます。加えてファイルシステムの構造、コンポーネント管理、構築とテストの自動実行などの話題も取り上げます。

「7章　ポータブルなmakefile」では、makefileの移植性に関する問題、特にUnixとWindows間に横たわる特色の違いについて論じます。CygwinのUnixエミュレーション環境についても少し詳しく触れるとともに、互換ではないファイルシステムとツールの機能による問題についても扱います。

「8章　CとC++」では、ソースファイルツリーとバイナリツリーを分離する方法とソースツリーを読み出し専用にする方法についての具体的な例を示します。依存関係の分析が再登場し、言語固有の解決策に重点を置いて説明します。8章と9章は多くの課題について1章と関連しています。

「9章　Java」では、makeをJavaの開発環境に適応させる方法を説明します。CLASSPATHの管理、多数のファイルをコンパイルする方法、jarの作成方法を紹介します。

「10章　makeの性能改善」では、効率的なmakefileを書く方法を示すために、makeの行ういくつかの処理の性能を見てみます。性能のボトルネックを見つけ出し、改善するためのテクニックについて議論します。そしてGNU makeの並列実行機能について少し詳しく説明します。

「11章　makefileの実例」では、実際に使われている複雑なmakefileの例を紹介します。最初は本書を作成するためのmakefileです。これは非常に高度に自動化を行っているという点と非伝統

的な領域に対して適用しているという点で興味深いものです。もう1つの例はLinux 2.6のkbuildシステムからの抜粋です。

「12章　makefileのデバッグ」では、問題のあるmakefileを修正するための魔術について掘り下げます。ここではmakeが覆い隠しているものを見る方法と、開発時の苦痛を和らげる方法を紹介します。

「付録A」、「付録B」、「付録C」では補助的な素材を提供します。

「付録A　makeの実行」は、GNU makeのコマンドラインオプションに関するリファレンスガイドです。

「付録B　限界を超えて」では、GNU makeの制限を明らかにし、データ構造体の管理と数値計算という通常では提供していない2つの機能について説明します。

「付録C　GNU Free Documentation License–GNU Project–Free Software Foundation（FSF）」は、本書のテキストを配布する際のライセンスを収録しています。

本書での表記法

本書では、以下の書体を使用します。

太字（Gothic）
　本文中の技術用語や概念など、強調するときに用います。

等幅（Constant width）
　URL、電子メールアドレス、ファイル名、ファイル拡張子、パス名、ディレクトリ、コマンド、オプション、変数、関数などを表します。

等幅の斜体（*Constant width italic*）
　ユーザが指定した値に置き換えられる個所を表します。

等幅の太字（Constant width bold）
　入力するコードに用います。

サンプルコードの利用

本書は、皆さんの仕事を完了させる助けとなるために存在します。通常は、本書のコードを皆さんのプログラムやドキュメント内で使用できます。コードの大部分をそのまま使用するのでなければ、著者の許可を得る必要はありません。例えば、本書のコードの一部を利用してプログラムを作成する場合の許可は必要ありません。サンプルコードのCD–ROMを販売したり、配布したりする場合は許可が必要です。本書に言及して質問に回答したり、サンプルコードを引用したりする場合の許可は必要ありません。本書のサンプルコードの大部分を皆さんのプロダクトのドキュメントに組み入れる場合は許可が必要です。

引用時に属性を明らかにしていただけるとありがたいのですが、必須ではありません。属性には、書名、著者名、出版社、ISBNが含まれます。例えば、「Managing Projects with GNU Make, Third Edition、Robert Mecklenburg著、Copyright 2005 O'Reilly Media, Inc., 0-596-00610-1」です。

サンプルコードの利用時に許可が必要だと感じた場合は、お気軽にpermissions@oreilly.comに電子メール（英文）を送ってください。

ご意見の送付先

本書に関するコメントや質問は出版社に送ってください。

株式会社オライリー・ジャパン
〒160-0002　東京都新宿区坂町26番地27 インテリジェントプラザビル1F
電話　03-3356-5227
FAX　03-3356-5261
電子メール　japan@oreilly.co.jp

例、正誤表、補足情報を掲載した本書のWebページがあります。以下のURLから参照してください。

http://www.oreilly.com/catalog/make3/　（英語）
http://www.oreilly.co.jp/books/4873112699/　（日本語）

本書に関する技術的な質問や意見は、以下の宛先に電子メール（英文）を送ってください。

bookquestions@oreilly.com

オライリーの書籍、カンファレンス、リソースセンタ、オライリーネットワークに関する情報は、以下のオライリーのWebサイトを参照してください。

http://www.oreilly.com

謝辞

実現可能なビジョンと信念を示してくれたRichard Stallmanに感謝します。もちろんPaul Smithがいなければ GNU `make` は現在の形で存在してはいなかったでしょう。

編集者であるAndy Oramの忍耐強いサポートと情熱に感謝します。

Cimarron Software社にはこのプロジェクトを始めるための、そしてRealm Systems社にはプロジェクトを終了させるための環境を提供していただきました。その中でも特にDoug Adamson、Cathy Anderson、Peter Bookmanに感謝します。

レビューアであるSimon Gerraty、John Macdonald、Paul Smithからは洞察に満ちたコメントをいただくとともに、厄介なエラーも修正していただきました。

以下は今回の作業に対して価値のある貢献をしていただいた方々です。Steve Bayer、Richard Bogart、Beth Cobb、Julie Daily、David Johnson、Andrew Morton、Richard Pimentel、Brian Stevens、Linus Torvalds。日々の奮闘の中で安全な隠れ場所を提供してくれた、Christine Delaney、Tony Di Sera、John Major、Daniel Readingに感謝します。

そして最後に、16か月に渡りサポートと励ましと愛を示してくれた妻のMaggie Kastenそして子供達WilliamとJamesに最大の感謝と愛を捧げます。本書を書き上げる苦楽をともにしてくれて本当にありがとう。

目 次

1章 簡単なメイクファイルを書いてみよう

プログラミングという作業は、ソースファイルを編集する、ソースファイルをコンパイルして実行形式に変換する、できたものをデバッグするといった非常に単純な工程をたどるのが普通です。ソースの実行形式への変換は決まりきった作業ではありますが、正しく行われないと問題が生じたときの原因追求にプログラマが多大な時間を費やすということにもなりかねません。コードを修正して実行してみたけれども全然バグが治っていないことにイライラした経験はたいていの開発者なら持っているでしょう。そして実は、ソースを再コンパイルしていなかったとか、再リンクを忘れていただとか、jarを作り直していなかったなど、作業手順のミスにより修正済みのコードを実行していたわけではないことに後になって気づいたりするものです。さらにいうと、他のプラットフォームや別のライブラリに対応するなど異なるバージョンのプログラムを開発するにつれ、プログラム自体の複雑さが増せば増すほど、このようなありふれた作業の間違えやすさも増大していきます。

ソースコードを実行形式に変換する作業のつまらない部分を自動化するためにmakeは存在します。スクリプトといったものに比べてmakeが優れているのは、構築するプログラムの要素間の関連を指定すれば、そこからファイルのタイムスタンプの比較を行うなどしてプログラムを再構築するために何を行えばよいのかをmakeは正確に認識できるという点です。これらの情報を使い、makeは不必要な作業を排除することで構築作業を最適化できます。

GNU make（そしてその他のmake）は、この作業を正確に実行します。ソースコード、中間ファイル、そして実行形式ファイルの関連を記述するための言語をmakeは提供しています。その他にさまざまな構成情報を管理したり、再利用可能なルールの断片をライブラリ化したり、ユーザ定義マクロにより開発プロセスをパラメータ化するなどの機能も提供しています。要するに、プログラムの構成要素にはどのようなものがあるのか、それらをどう組み合わせるのかといったことを示す設計図を備えていることから、makeは開発プロセスの中心であると考えることができます。

makeに指示する情報は通常makefileという名前のファイルに保存されます。次に示すのは伝統的な“Hello, World”プログラムを構築するためのmakefileです。

```
hello: hello.c
        gcc hello.c -o hello
```

プログラムを構築するには、シェルのコマンドプロンプトに次のように入力しmakeを実行します。

```
$ make
```

これによりmakeはmakefileを読み込み、次のように最初のターゲットを構築します。

```
$ make
gcc hello.c -o hello
```

makeのコマンドライン引数にターゲットが指定されていれば、そのターゲットが構築されます。指定されていなければデフォルトターゲット、つまりmakefile中最初のターゲットが構築されます。

普通、makefileのデフォルトターゲットはプログラムを構築するためのものです。その作業はたいていいくつもの工程を含んでいます。ソースコードが存在せずflexやbisonといったユーティリティプログラムによりソースコードを生成しなければならないことはよくあることです。次にソースコードはコンパイルされてバイナリのオブジェクトファイル（CやC++では.oファイル、Javaでは.classファイルが相当します）が作られます。そしてCやC++のオブジェクトファイルはリンカ（普通はgccなどのコンパイラから起動されます）により結合されて実行形式のファイルが作られます。

何らかのソースコードを変更した後、makeを再度実行すると、その変更が実行ファイルに正しく反映されるように前回実行されたコマンドの全部ではなく、いくつかがもう一度実行されます。実行ファイルを再構築するのに必要かつ最小限のコマンドを選ぶことができるようにするために、makefileにはソースファイル、中間ファイル実行ファイルの関係が記述されています。

つまりmakeの持つ第一の価値は、プログラムの構築に必要な一連の複雑な過程を最適化することにより「編集－コンパイル－デバッグ」の繰り返しに費やす時間を節約できるという点にあります。さらにmakeはCやC++から最近ではJavaやTeXそしてデータベース管理など、ある種のファイルが別のファイルに依存しているという環境に対して柔軟に適応させられます。

1.1 ターゲットと必須項目

基本的にmakefileにはプログラムを構築するためのルールが書かれています。最初のルールはデフォルトルールとして使用されます。ルールは**ターゲット**（*target*）、**必須項目**（*prereq*）そして**実行コマンド**（*commands*）の3つの部分から構成されています。

```
target: prereq1 prereq2
        commands
```

ターゲットはファイルもしくは何か作り出されるものでなければなりません。必須項目（または依存項目ともいう）はターゲットが作成される前には存在していなければならないものです。実行コマンドは必須項目からターゲットを作るためのものです。

次はCのファイルfoo.cをコンパイルしてオブジェクトファイルfoo.oを作るためのルールです。

```
foo.o: foo.c foo.h
        gcc -c foo.c
```

ターゲットであるファイルfoo.oはコロン（:）の前に置かれています。必須項目であるfoo.cとfoo.hはコロンの後ろに置かれます。実行コマンドは次の行に書かれ先頭にタブが入ります。

makeがルールを解釈する段階に入ると、まずターゲットと必須項目に列挙されているファイルを探します。必須項目の中で関連するルールを持つものがある場合には、まずそれらを更新することから始めます。その後でターゲットであるファイルについて検討されます。必須項目の中でターゲットよりも新しいものがあるなら、ターゲットはコマンドを実行することで再構築します。コマンド行†はそれぞれシェルに渡され、それぞれ別のシェル（サブシェル）で実行されます。コマンドのうちどれかでエラーが発生した場合には、ターゲットの構築を中止しmakeは終了します。2つのファイルを比較する際には、後で修正された方を新しいファイルであると考えます。

次に示すのは入力されたテキストの中で、“fee”、“fie”、“foe”、“fum”がそれぞれ何回出てくるかを数えるプログラムです。単純なmainからflexのスキャナを呼び出しています。

```
#include <stdio.h>
extern int fee_count, fie_count, foe_count, fum_count;
extern int yylex( void );
int main( int argc, char ** argv )
{
    yylex();
    printf( "%d %d %d %d\n", fee_count, fie_count, foe_count, fum_count );
    exit( 0 );
}
```

スキャナ††はとても簡単です。

```
        int fee_count = 0;
        int fie_count = 0;
        int foe_count = 0;
        int fum_count = 0;
%%
fee     fee_count++;
fie     fie_count++;
foe     foe_count++;
fum     fum_count++;
```

† 訳注：本書では“command line”の訳語として、makeを実行するためのシェルに対するコマンド列に「コマンドライン」、makefileの中に記述されたコマンドに「コマンド行」を使います。

†† 訳注：lexまたはflexと呼ばれるプログラムに字句パターンの解析ルールを入力すると、解析器、すなわちスキャナを生成します。スキャナの関数名はyylex()であり、上記プログラムではmainの最初でそのyylex()を呼び出しています。ここでは解析ルールやflexの詳細については理解する必要はありませんが、flexというプログラムを使えばyylex()という関数の入ったソースファイルが生成されるということを知っておく必要があります。

このプログラム用のmakefileもとても簡単です。

```
count_words: count_words.o lexer.o -lfl
        gcc count_words.o lexer.o -lfl -ocount_words

count_words.o: count_words.c
        gcc -c count_words.c

lexer.o: lexer.c
        gcc -c lexer.c

lexer.c: lexer.l
        flex -t lexer.l > lexer.c
```

最初にこのmakefileが使われたときには、次のように実行されます。

```
$ make
gcc -c count_words.c
flex -t lexer.l > lexer.c
gcc -c lexer.c
gcc count_words.o lexer.o -lfl -ocount_words
```

これで実行可能なプログラムが作成されました。もちろん実際のプログラムはこの例よりももっと多くのモジュールからできているでしょう。このmakefileは後で紹介するmakeが持つ機能の多くを使っていないため、いささか冗長でもあります。とはいうものの、これは実際に動作する便利なmakefileに変わりはありません。

1.2 依存関係の検証

makeはすべきことをどのようにして決めているのでしょうか。前出の例を詳しく調べてみましょう。

まずmakeのコマンドラインにはターゲットが指定されていなかったため、makeはデフォルトターゲットであるcount_wordsを作成しようとします。そこでその必須項目であるcount_words.o、lexer.o、-lflに着目します。ここでmakeはcount_words.oを作成する方法について考え、ルールを再検証します。count_words.oの必須項目であるcount_words.cにはルールがないこと、しかしそのファイルは存在していることからmakeはcount_words.cからcount_words.oを作るためのコマンドを実行しようとします。

```
gcc -c count_words.c
```

このターゲットから必須項目へ、そしてそのまた必須項目へと続く一連の流れはmakeが実行するコマンドを決定するためにmakefileをどうやって解析するかを示しています。

次に着目する必須項目はlexer.oです。同様のルール連鎖によりlexer.cにたどり着くのですが、今度はそのファイルが存在していません。makeはlexer.lからlexer.cを生成するルールを発見し、flexプログラムを実行します。これでlexer.cが存在することになり、gccコマンドを実行す

ることが可能になりました。

最後にmakeは-lflを調べます。gccの-lオプションはアプリケーションプログラムにリンクしなければならないシステムライブラリを指示します。“fl”から導かれる実際のファイル名はlibfl.aです。GNU makeはこの文法を扱う特殊な機能を持っています。必須項目に-l<NAME>という形式の指定が存在している場合にはlibNAME.soという形式のファイルを探します。なければ次にはlibNAME.aというファイルを探します。ここでは/usr/lib/libfl.aが発見され、最後の作業であるリンクに駒を進めることになります。

1.3 再構築作業を最小にする

このプログラムを実行してみると、“fee”、“fie”、“foe"、“fum”以外の入力テキストが出力されることがわかると思います。これは望んだ結果ではありません。字句解析器にいくつかのルールを指定し忘れたため、flexは認識できなかったテキストをそのまま出力したわけです。これを解決するには、その他すべての文字のためのルールと改行のルールを加えればよいのです。

```
          int fee_count = 0;
          int fie_count = 0;
          int foe_count = 0;
          int fum_count = 0;
%%
fee       fee_count++;
fie       fie_count++;
foe       foe_count++;
fum       fum_count++;
.
\n
```

このファイルを修正した後、その修正をテストするためにプログラムを再構築する必要があります。

```
$ make
flex -t lexer.l > lexer.c
gcc -c lexer.c
gcc count_words.o lexer.o -lfl -ocount_words
```

今回はcount_words.cが再コンパイルされませんでした。ルールを解析する際にcount_words.oがすでに存在していることとその必須項目であるcount_words.cよりも新しいことがわかったので、count_words.oを最新の状態にするためにmakeは何もする必要がないことを知ったのです。一方、lexer.cは、ターゲットのlexer.cよりも必須項目であるlexer.lのほうが新しかったため makeはlexer.cを再作成する必要が生じました。そして押し出されるようにlexer.oが作られcount_wordsが再構築されることになったのです。さてこれで単語カウントプログラムは正しく修正されました。

```
$ count_words < lexer.l
3 3 3 3
```

1.4 makeの実行

ここでの例は以下の条件を前提としていました。

- すべてのソースコードとmakeのルールを記述したファイルは1つのディレクトリに入っている
- makeのルールを記述したファイルは、makefile、MakefileまたはGNUMakefileというファイル名である
- makeを実行したディレクトリにmakeのルールを記述したファイルは存在している

これらの条件下で実行されると、makeは最初に見つけたターゲットを自動的に更新します。それとは異なるターゲット（または2つ以上のターゲット）を更新するにはコマンドラインにターゲットの名前を指定します。

```
$ make lexer.c
```

makeは実行されるとルールを記述したファイルを読み込み、更新すべきターゲットを特定します。ターゲットまたはその必須項目のファイルが古かった（もしくは存在していない）場合には、ルールのコマンド欄に書かれているシェルコマンドを1つずつ実行します。コマンドを実行し終えたらmakeはターゲットが更新されたと考え、次のターゲットを処理するか終了します。

指定したターゲットがすでに最新のものであった場合、makeはそのこと（「lexer.cは最新です」の意）を表示して何もせずに終了します。

```
$ make lexer.c
make: `lexer.c' is up to date.
```

指定したターゲットがmakefile内にないか、あるいはターゲットに対して適用すべき暗黙ルール（2章で説明します）がない場合にはmakeは下記（「non-existent-targetを作るためのルールがありません」の意）のように表示します。

```
$ make non-existent-target
make: *** No rule to make target `non-existent-target'. Stop.
```

makeには多くのコマンドラインオプションがあります。最も便利なものの1つに、ターゲットを構築するために実行されるであろうコマンドを、実際には実行せずに表示だけ行うための--just-printオプション（-nオプション）があります。これは特にmakefileの作成時に役立ちます。また、makefileの中で使われている変数のデフォルト値もしくはmakefile内でセットされている値を、コマンドライン上の指定で上書きしてしまうことも可能です。

1.5 makefileの基本文法

これで自分用のmakefileを書くために必要になる基礎的な点を九分どおり押さえました。ここでmakeを使い始めるのに必要な文法と構造について残りの部分を学ぶことにしましょう。

makefileは普通トップダウン構造を持ち、先頭にあるallといった最も一般的なターゲットの構築が既定の動作となります。その他の詳細なターゲットに続いてプログラムを保守管理するための、例えば不要の一時ファイルを削除するcleanのようなターゲットが最後に位置します。これらのターゲット名から想像可能なように、ターゲットは実際のファイルである必要はなく、どのような名前でも付けられます。

これまでの例ではルールの単純な形式を見てきました。より詳細な（しかしながらまだ完全ではない）ルールの形式は以下のようになります。

```
target1 target2 target3 : prerequisite1 prerequisite2
        command1
        command2
        command3
```

コロン（:）の左側には1つかそれ以上のターゲットを、右側には必須項目を0個以上記述できます。必須項目が指定されない場合には、ターゲットの中で存在しないものだけが構築されます。ターゲットを更新するために実行されるコマンドは**コマンドスクリプト**と呼ばれることもありますが、たいていは単にコマンドと呼ばれます。

コマンドはそれぞれタブの後に書かれます。この（わかりにくい）文法により、makeはタブの後に続く文字列を実行するためにサブシェルに渡す必要があることを知ります。もし誤ってタブをコマンドではない行に書いてしまった場合、makeはタブに続く文字列をコマンドとして解釈してしまうのが普通です。もし幸運にも誤ったタブが文法エラーであると認識された場合、次のようなメッセージ（「最初のターゲットの前にコマンドが書かれています」の意）が表示されるでしょう。

```
$ make
Makefile:6: *** commands commence before first target. Stop.
```

タブの複雑さについては2章で説明します。

makeに対するコメント文字は#です。#から行末までのテキストはすべて無視されます。コメントはインデントしてもかまいません。そのための行頭の空白は無視されます。コメント文字である#は、コマンドの中ではコメントを意味しません。#とそれ以降のテキストを含めてすべての文字が実行のためシェルに渡されます。それがどのように扱われるかはシェルによって異なります。

長い行はUnixの標準エスケープ文字であるバックスラッシュ（\）†で継続することができます。しばしばコマンドはこの方法で書かれます。必須項目を次の行に継続する方法としてもよく使われます。後で長い必須項目リストを扱う別の方法を説明します。

これで簡単なmakefileを書くのに十分な知識を得ることができました。2章ではルールについてを、続いてmakeの変数についてを3章で、コマンドについて5章でそれぞれ詳しく扱います。それまでは変数やマクロそして複数行に渡るコマンドなどを使うのはお預けにしておきましょう。

† 訳注：ASCII文字ではバックスラッシュ（\）に当たる文字が、JISでは円記号（¥）となっているため、日本語の環境ではバックスラッシュの代わりに円記号となります。

2章 ルール

前章で単語カウントプログラムをコンパイルしリンクするための簡単なルールを書きました。それぞれのルールにはターゲットつまり更新されるべきファイルが定義され、そして各ターゲットは必須項目であるファイルにそれぞれ依存していました。ターゲットを更新するよう要求されると、必須項目であるファイルの中でターゲットよりも後で変更されたものが存在する場合にmakeはコマンドスクリプトを実行します。あるルールのターゲットは別のルールの必須項目であってもかまわないため、ターゲットと必須項目の集まりは依存関係の連鎖構造またはグラフ構造（省略して「依存関係グラフ」）を形成します。依頼されたターゲットの更新を行うために、この依存関係グラフを構成し処理することがmakeの行うすべてです。

makeにとってルールは非常に重要なので、数多くの異なる種類のルールが存在します。前章で見てきたような明示的なルールは、特定のターゲットが必須項目のどれかと比較して古かった場合には更新されなければならないことを示すものです。これが最もよく見られるタイプのルールです。パターンルールはファイル名を指定する代わりにワイルドカードを用いるものです。これによりパターンに合致したファイルが更新される必要があるときにはいつでもルールを適用することが可能になります。暗黙ルールとはmakeに組み込まれたルールデータベースに存在するパターンルールもしくはサフィックスルールを指します。組み込みのルールデータベースがあれば、ファイルの種類やサフィックスやターゲットを更新するプログラムなどに対してどう扱えばよいのかをmakeはすでに知っていることになるため、makefileを簡単に書けます。静的パターンルールは指定されたターゲットファイルのリストに対してのみ適用されることを除けば、通常のパターンルールと同じようなものです。

GNU makeは他のmakeの代わりとして気軽に使うことができるうえに、互換性のためだけの機能もいくつか提供しています。サフィックスルールはmakeに元から備わっている一般的なルールを書くための手段です。GNU makeはサフィックスルールの機能も提供しますが、それは時代遅れの機能であり、より汎用性の高くて明快なパターンルールで置き換えられるべきものであると考えられています。

2.1 明示的ルール

最もよく使われるルールは特定のファイルをターゲットや必須項目として指定する明示的ルールです。ルールは1つ以上のターゲットを持つことができます。この場合すべてのターゲットが同じ必須項目を持っていることを意味します。もしターゲットのいくつかが古かった場合、それぞれのターゲットに対して同じ処理が行われます。例えば以下のようになります。

```
vpath.o variable.o: make.h config.h getopt.h gettext.h dep.h
```

この指定によりvpath.oとvariable.oはどちらも同じC言語ヘッダファイルに依存していることを示しています。この行は次の2行と同じ意味を持っています。

```
vpath.o: make.h config.h getopt.h gettext.h dep.h
variable.o: make.h config.h getopt.h gettext.h dep.h
```

2つのターゲットはそれぞれ独立して扱われます。オブジェクトファイルのどちらかが、その必須項目に対して古かった（つまりヘッダファイルのどれかが、そのオブジェクトファイルよりも更新時刻が新しかった）場合、makeはルールに付随しているコマンドを実行してオブジェクトファイルを更新します。

ルールを1か所ですべて定義する必要はありません。makeはターゲットを見るたびに、そのターゲットと必須項目を依存関係グラフに登録していきます。すでにそのターゲットがグラフに登録されていた場合、そこに追加の必須項目を加えます。そのため、makefileを読みやすくするためにルールを2か所で定義して、長い行を無理なく分割することができます。

```
vpath.o: vpath.c make.h config.h getopt.h gettext.h dep.h
vpath.o: filedef.h hash.h job.h commands.h variable.h vpath.h
```

もっと複雑な使い方として、まったく異なる方法で扱われる必須項目群をそれぞれまとめることができます。

```
# lexer.cはvpath.cがコンパイルされる前に作成されるようにする
vpath.o: lexer.c
...
# 特別のフラグ付きでvpath.cをコンパイルする
vpath.o: vpath.c
        $(COMPILE.c) $(RULE_FLAGS) $(OUTPUT_OPTION) $<
...
# プログラムにより生成された依存関係を読み込む
include auto-generated-dependencies.d
```

最初のルールはlexer.cが更新された場合にはvpath.oも更新されなければならないことを示しています（おそらくlexer.cの生成には何か別の働きもあるのでしょう）。またこのルールはターゲットが更新される前には必須項目が必ず新しいものであることを保証するために働きます（これはルールが持つ双方向の性質といえます。正方向に見ると、lexer.cが更新されていたならばvpath.oを

更新しなければならないということを示しています。逆方向に見ると、vpath.oを作るか使うかする場合には、まずlexer.cが最新であることを確実にしなければならないということを示しています)。この難解な関係を開発者が忘れないようにするために、lexer.cを管理するためのルールのそばにこのルールを置くことになるでしょう。vpath.oをコンパイルするルールは他のコンパイルルールとともに後方に書かれます。このルールのコマンドは3つのmake変数を使っています。これらについては後で詳しく触れますが、今のところはドル記号の後に1文字かドル記号の後ろに括弧で囲まれた単語を置く形式がmakeの変数であるということを知っていれば十分です（本章の後のほうでもう少し説明し、3章で詳しく説明します)。最後に.oと.hの依存関係は別のプログラムにより管理されたファイルとしてmakefileに取り込まれています。

2.1.1 ワイルドカード

makefileには非常に長い行がしばしば登場します。これを簡単に扱うために、makeではワイルドカード（ファイル名展開としても知られています）の機能を備えています。makeのワイルドカードは、Bourne Shellのものと同じで、~、*、?、[...]、[^...]などが使えます。例えば、*.*は、ファイル名にピリオドを持つすべてのファイルとして扱われます。クエスチョンは何らかの文字1つを表し、[...]は文字集合を表します。逆の（補集合としての）文字集合を使う場合には[^...]とします。

加えて、チルダ（~）は、ユーザのホームディレクトリを表し、チルダの後ろにユーザ名があると、そのユーザのホームディレクトリを表します。

ワイルドカードはターゲットや必須項目、コマンドスクリプトの中に存在すると、いつでもmakeにより自動的に展開されます。その他の個所では呼び出した機能により展開されることになるでしょう。ワイルドカードを使うことで、柔軟なmakefileが書きやすくなります。例えばプログラムに関連するファイルをすべて列挙する代わりに、ワイルドカードの*を使うことができます†。

```
prog: *.c
        $(CC) -o $@ $^
```

ワイルドカードの使用に慎重であることも大切です。そうでないと、次の例のように簡単に間違いを作ってしまいます。

```
*.o: constants.h
```

これの意図するところは明確で、すべてのオブジェクトファイルがヘッダファイルのconstans.hに依存するというものです。ところが、オブジェクトファイルの一切存在しないディレクトリで使われた場合どうなるか考えてみてください。

```
: constants.h
```

† 意図しないファイルを誤ってプログラムにリンクしてしまう可能性があるため、厳密に管理された開発環境では、ファイルを選択する手段としてワイルドカードを使用することは、よくない手法だと考えられています。

これはmakeに対して正当な表現であり、これ自体がエラーとなることはありません。しかし同時にユーザが望んだ依存関係を表現することもありません。正しい方法は、ワイルドカードをソースファイル（それらは常に存在しているので）に対して使い、オブジェクトファイルの名前に変換してからルールに適用するというものです。このテクニックについては、4章でmakeの関数について説明した際に扱います。

ワイルドカードのパターンがターゲットや必須項目に現れた際にはmake自身によりワイルドカードの展開が行われることを知っておくべきです。一方、パターンがコマンドの中に現れた場合にはサブシェルにて展開が行われます。両者の違いは重要な意味を持つことがあります。というのもmakeはmakefileを読み込むとすぐにワイルドカードを展開するのに対し、シェルはコマンド中のワイルドカードをもっと後の、コマンドを実行する際に展開するからです。多数の複雑なファイル操作が行われた後では、この2種類のワイルドカード展開はまったく違った結果をもたらすことになりかねません。

2.1.2 擬似ターゲット

これまでは、ターゲットや必須項目はすべて作成されたり更新されるファイルを表していました。普通はそうなのですが、ターゲットをコマンドスクリプトの単なるラベルとして使うと便利な場合があります。例えば以前、多くのmakefileファイルで標準的に最初のターゲットがallであると説明しました。実際のファイルを示していないターゲットは、**擬似ターゲット**（Phony Target）と呼ばれます。よく使われる擬似ターゲットの1つにcleanがあります。

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

通常、擬似ターゲットはルールに関連づけられたコマンドがターゲット名のファイルを作らないため、必ず実行されます。makeが擬似ターゲットとファイルターゲットを区別できないことは重要です。もし擬似ターゲット名のファイルが存在していたなら、makeはそのファイルと擬似ターゲットを依存関係グラフに関連づけてしまいます。例えばcleanというファイルが作られていたとすると、make cleanを実行した結果は次のような紛らわしいメッセージ（「cleanは最新です」の意）が出ることになります。

```
$ make clean
make: `clean' is up to date.
```

たいていの擬似ターゲットには必須項目がないので、この場合のcleanターゲットは常に最新であることになり、コマンドが実行されることがありません。この問題を回避するために、特殊なターゲット.PHONYがGNU makeには用意されていて、実在のファイルとは関連していないターゲットを指定することができます。.PHONYの必須項目とすることで、いかなるターゲットも擬似ターゲットにすることができます。

```
.PHONY: clean
clean:
    rm -f *.o lexer.c
```

これで、たとえcleanというファイルが存在していたとしても、cleanターゲットに関連づけられたコマンドをmakeは必ず実行することになります。ターゲットが最新ではないとマークを付ける以外に、ターゲットが擬似ターゲットであると指定することで、そのファイルはソースファイルからターゲットを構築するという通常のルールに従う必要がないことをmakeに知らせることになります。これにより、makeは通常のルール検索を最適化しパフォーマンスを改善することができます。

擬似ターゲットは常に最新ではないと判断され再構築対象のターゲットとなるため、擬似ターゲットを実在のファイルの必須項目として指定することは、めったに意味を持ちません。しかし擬似ターゲットに必須項目を与えるのは多くの場合で有用です。例えばallターゲットは通常、構築されるプログラムのリストとなります。

```
.PHONY: all
all: bash bashbug
```

これによりallターゲットでbashシェルと、エラーのレポートツールbushbugを構築します。

擬似ターゲットはmakefileに埋め込まれたシェルスクリプトと考えることもできます。擬似ターゲットを他のターゲットの必須項目とすることで、そのターゲットを構築する前に擬似ターゲットのスクリプトを実行することになります。例えばディスクの空きが厳しい状態で、ディスクを大量に必要とする作業をする前に残りがどれくらいか表示することを考えてみます。次のように書けるでしょう。

```
.PHONY: make-documentation
make-documentation:
        df -k . | awk 'NR == 2 { printf( "%d available\n", $$4 ) }'
        javadoc ...
```

dfとawkを使ったコマンドはその他のいろいろなターゲットで何度も使われることになりますが、dfコマンドの出力フォーマットが異なる別のシステムに移植する際には、それぞれの個所で使われたコマンドをすべて修正しなければならないという保守上の問題が生じてしまいます。代わりに独立した擬似ターゲットにdfコマンドを置くことにします。

```
.PHONY: make-documentation
make-documentation: df
        javadoc ...

.PHONY: df
df:
        df -k . | awk 'NR == 2 { printf( "%d available\n", $$4 ) }'
```

擬似ターゲットdfをmake-documentationの必須項目としているため、javadoc文章を生成する前にdfターゲットが実行されることになります。make-documentationも擬似ターゲットなので、うまく働きます。これでdfを他のターゲットでも簡単に再利用できるようになりました。

擬似ターゲットはmakefileの「ユーザインタフェース」を改善する際にも役立ちます。たいていのターゲットはパス名を含む複雑な文字列や、（バージョン番号など）ファイル名の付加的な構成要素やサフィックスなどを持ちます。こういったターゲットのファイル名をコマンドライン上で指定するのは骨の折れる作業です。これは単純な名前を持つ擬似ターゲットの必須項目に実際のターゲットファイルを指定することにより回避できます。

慣例として、多くのmakefileで使われている標準的な擬似ターゲットがいくつか存在します。表2-1は、標準のターゲットのリストです。

表2-1 標準的な擬似ターゲット

ターゲット	機能
all	アプリケーションを構築するすべての作業を行う
install	コンパイル済みのバイナリをインストールする
clean	ソースから作られたバイナリを削除する
distclean	元の配布物に含まれていなかったすべての生成物を削除する
TAGS	エディタの使用するtagsを作成する
info	Texinfoのソースより GNU infoファイルを作成する
check	このアプリケーションに関連するすべてのテストを実行する

ctagsやetagsといったプログラムはTAGSという名前のファイルを出力するので、TAGSは実際には擬似ターゲットではありません。ですが、知りうるかぎり、これが唯一の標準的に使われている非擬似ターゲットなので、このリストに加えました。

2.1.3 空のターゲット

makeの機能を活用するための道具として使えるという点で**空のターゲット**は擬似ターゲットと似ています。擬似ターゲットは常に最新ではないと判断されるため、そのルールは常に実行されるとともに依存しているもの（必須項目として関連づけられている別のターゲット）も再構築されます。ところで、次のようなコマンドについて考えて見ましょう。そのコマンドはファイルを作成することはないけれども、常に実行したいわけではありません。またそのコマンドで依存しているものも更新したくもありません。こういう場合、空のファイル（**クッキー**と呼ばれることもあります）をターゲットに使ったルールを使うことができます。

```
prog: size prog.o
        $(CC) $(LDFLAGS) -o $@ $^

size: prog.o
        size $^
        touch size
```

sizeというルールは、実行の最後でtouchコマンドを使いsizeという名前の空のファイルを作成します。prog.oが更新された場合にのみこのsizeルールが発動されるようにするため、この空のファイルのファイル日付が利用されます。さらにいうと、progの必須項目としてsizeが指定されていても、prog.oが新しくならないかぎりprogが再構築されることはありません。

空のファイルは自動変数$?と組み合わせた場合、特に役立ちます。自動変数については「2.2.1 自動変数」で取り扱いますが、少しこの変数に対して先取りしても害にはならないでしょう。ルールのコマンドスクリプト内では、makeは変数$?を必須項目の中でターゲットよりも新しいもののリストに展開します。次に示すのはmakeが前回印刷を実行してから変更されたものをすべて印刷するルールです。

```
print: *.[hc]
        lpr $?
        touch $@ †
```

普通、空のファイルは何らかのイベントが起きたことの印として使われます。

2.2 変数

すでに紹介した例の中でも使っていた変数について少し見てみましょう。最も簡単な変数は次のような構造をしています。

```
$(variable-name)
```

これにより、variable-nameという名前の変数を展開するよう指示することになります。変数にはあらゆるテキストを入れることができますし、変数名はピリオドを含めてほとんどの文字が使えます。普通、変数名はmakeが認識できるように$( )か${ }で囲みますが、特殊なケースとして変数名が1文字の場合は囲む必要がありません。

一般的にmakefileには多数の変数が使われますが、makeが自動的に定義する特別な変数も多く出てきます。そのうちのいくつかはmakeのふるまいを調整するためにユーザによって設定されますが、その他はmakeがユーザのmakefileと情報をやり取りするためにmakeにより設定されます。

2.2.1 自動変数

自動変数は実行するルールが定まった後makeにより設定されます。自動変数は、ファイル名を明示的に指定しなくても済むように、ターゲットや必須項目リストの要素へアクセスする手段を提供します。自動変数は同じことを何度も書かずに済むという点で便利ではありますが、汎用のパターンルールで使う際には危険な点があります（これは後で説明します）。

† 訳注：すぐ後で出てきますが、自動変数$@はターゲットの名前（この場合print）に展開されます。

次に示すのが7つの主要な自動変数です。

`$@`
ターゲットのファイル名を表します。

`$%`
ライブラリの構成指定中の要素を表します。

`$<`
最初の必須項目のファイル名を表します。

`$?`
ターゲットよりも後で更新された必須項目のすべてを、スペースで区切ったリストで表します。

`$^`
すべての必須項目をスペースで区切ったリストで表します。コンパイルやファイルのコピーなど項目の重複が望ましくない場合のために、重複したファイルはリストから取り除かれます。

`$+`
`$^`と同様にすべての必須項目をスペースで区切ったリストで表しますが、重複を含みます。これは重複した値に意味があるリンカへの指定などといった特定の用途のために用意されました。

`$*`
ターゲットファイル名の一部を表します。一部とはサフィックスを除いたファイル名を示すのが普通です（どのようにファイル名の一部を作り出すかは「2.4 パターンルール」で扱います）。この変数をパターンルール以外での使用することは推奨されません。

さらに加えると、上記の変数は他のmakeとの互換性維持のためそれぞれ2つの拡張形を持ちます。1つは値のディレクトリ部分を返し、`$(@D)`や`$(<D)`のようにDを加えます。もう1つは値のファイル部分を返し、`$(@F)`や`$(<F)`のようにFを加えます。これらの変数名は1文字以上になるので、括弧で囲むことが必要になります。GNU makeでは、dirとnotdir関数というもっと見やすい代替手段を提供しています。それらは4章で扱います。

自動変数はターゲットや必須項目に対して適用されるルールが決定してから値が設定されるため、使用できるのはコマンド行に限定されます。

以下に示すのは例で使用したmakefileを修正し、明示的に指定していたファイル名を適切な自動変数に置き換えたものです†。

```
count_words: count_words.o counter.o lexer.o -lfl
        gcc $^ -o $@

count_words.o: count_words.c
        gcc -c $<
```

† 訳注：このmakefileでは、「2.3 VPATHとvpathによるファイルの検索」で行うソースコードの分離を少し先取りしてcounter.cというファイルが登場しています。counter.cの内容は「2.3 VPATHとvpathによるファイルの検索」を参照してください。

```
counter.o: counter.c
        gcc -c $<

lexer.o: lexer.c
        gcc -c $<

lexer.c: lexer.l
        flex -t $< > $@
```

2.3 VPATHとvpathによるファイルの検索

これまで使っていた例は、makefileとすべてのソースが1つのディレクトリに入っているという単純なものでした。しかし現実のプログラムはもっと複雑です。例をもっと現実的な構造にしてみましょう。単語カウントプログラムを修正してmainの一部をcounterという関数に変更します。

```
#include <lexer.h>
#include <counter.h>
void counter( int counts[4] )
{
    while ( yylex( ) )
        ;
    counts[0] = fee_count;
    counts[1] = fie_count;
    counts[2] = foe_count;
    counts[3] = fum_count;
}
```

再利用可能なライブラリ関数はヘッダファイルに宣言を置くべきなので、宣言を入れるcounter.hを作りましょう。

```
#ifndef COUNTER_H_
#define COUNTER_H_
extern void
counter( int counts[4] );
#endif
```

lexer.lのための宣言をlexer.hに置きます。

```
#ifndef LEXER_H_
#define LEXER_H_
extern int fee_count, fie_count, foe_count, fum_count;
extern int yylex( void );
#endif
```

昔ながらのソースツリー構造ではヘッダファイルをincludeディレクトリに、ソースファイルをsrcディレクトリに置きます。加えてmakefileをそれらの親ディレクトリに置くことにしましょう。

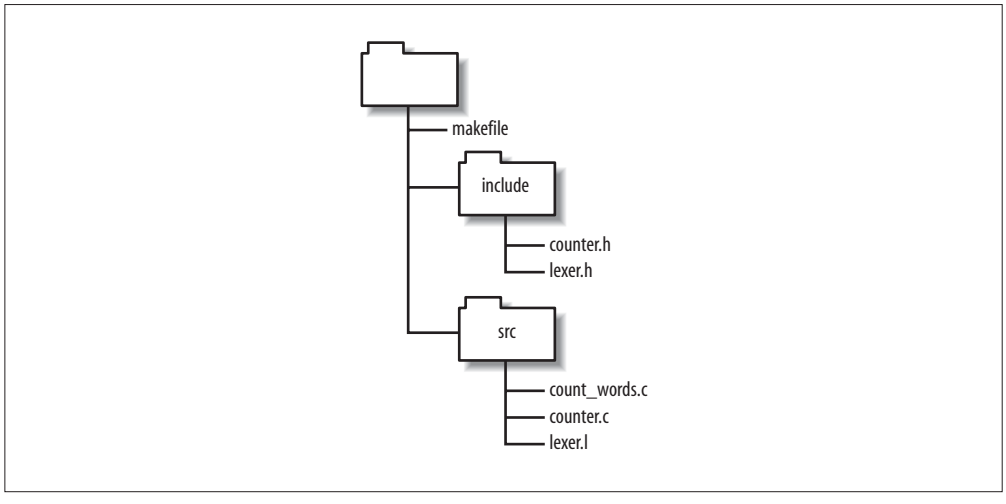

図2-1 例題プログラムのソースツリー構造

これでプログラムは図2-1に示す構造を持つことになりました†。

ソースファイルはヘッダファイルを持つことになったため、ヘッダファイルが修正された際には該当するオブジェクトファイルが更新されるように新しい依存関係をmakefileに記述しなければなりません。

```
count_words: count_words.o counter.o lexer.o -lfl
gcc $^ -o $@

count_words.o: count_words.c include/counter.h
gcc -c $<

counter.o: counter.c include/counter.h include/lexer.h
gcc -c $<

lexer.o: lexer.c include/lexer.h
gcc -c $<

lexer.c: lexer.l
flex -t $< > $@
```

† 訳注：この一連の変更により、main()のあるcount_words.cは次のようになります。

```
#include <stdio.h>
#include <counter.h>

int main( int argc, char ** argv )
{
    int counts[4];
    counter( counts );
    printf( "%d %d %d %d\n", counts[0], counts[1], counts[2], counts[3] );
    exit( 0 );
}
```

これでmakeを実行してみると、「count_word.oで必要となるcount_word.cを作るルールが存在しない」という出力を得ます。

```
$ make
make: *** No rule to make target `count_words.c', needed by `count_words.o'. Stop.
```

何が起こったのでしょう。makefileはcount_word.cを更新しようとしましたが、しかしそれはソースファイルです。理解するためにmakeになったつもりで考えてみましょう。最初の必須項目はcount_word.oです。そのファイルが見当たらないので、作成するためのルールを探します。count_words.oを作るルールではcount_words.cを必要としています。なぜmakeはソースファイルを見つけられないのでしょうか。それはソースファイルがカレントディレクトリではなくsrcディレクトリにあるからです。何も指示がなければmakeはターゲットと必須項目をカレントディレクトリだけから探し出します。ソースファイルがsrcにあるということをどうすればmakeに知らせることができるでしょうか。もしくはもっと一般的に、ソースコードの場所をmakeに指示するにはどうすればよいのでしょうか。

VPATHまたはvpath機能を使えば、ソースファイルを別のディレクトリに探しに行くようmakeに指示することができます。ここでの問題を解決するために、VPATHの指定をmakefileに加えます。

```
VPATH = src
```

これにより目的のファイルがカレントディレクトリに存在しなかった場合にはsrcを見るようにmakeに指示することになります。これでmakeを実行すると、次のような結果になります。

```
$ make
gcc -c src/count_words.c -o count_words.o
src/count_words.c:2:21: counter.h: No such file or directory
make: *** [count_words.o] Error 1
```

ソースファイルに正しく相対パスを補うことにより、最初のファイルのコンパイルは正しく実行されています。これは自動変数を使用すべきもう1つの理由です。つまりmakeはファイル名をハードコードしている場合には、ソースファイルへの正しいパスを扱えないのです。残念なことにgccがヘッダファイルを見つけられなかったのでコンパイルは失敗してしまいました。この問題は-Iオプションを加えてコンパイルルールをカスタマイズすることで解決できます。

```
CPPFLAGS = -I include †
```

加えて、各gccコマンドも、gcc $(CPPFLAGS)に変更します。これで正しく動作するようになりました。

```
$ make
gcc -I include -c src/count_words.c -o count_words.o
```

† 訳注：変数CPPFLAGSの変更については、「2.5.2 ルールの構造」で触れています。

```
gcc -I include -c src/counter.c -o counter.o
flex -t src/lexer.l > lexer.c
gcc -I include -c lexer.c -o lexer.o
gcc count_words.o counter.o lexer.o /lib/libfl.a -o count_words
```

VPATH変数はmakeがファイルを探すディレクトリのリストです。このリストは必須項目とターゲットを探す際に使われますが、コマンドスクリプトに書かれているファイルには無効です。ディレクトリのリストはUnixでは空白またはコロンで、Windowsでは空白またはセミコロンで区切ります。空白のほうが望ましい理由として、すべてのシステムで動作しコロンとセミコロンの起こす混乱を避けることができます。またディレクトリは空白で区切られているほうが読みやすいものです。

VPATH変数はファイル検索の問題を解決できたという点ではよいのですが、どちらかというと扱いにくい点もあります。makeは目的のファイルを各ディレクトリから見つけ出そうとします。もし同じ名前のファイルがVPATHで指定された複数の場所に存在した場合、makeは最初のものを使います。これが問題を引き起こす場合もあります。

vpath命令は目的を達成するための、より適正な方法です。この命令の構文は次のとおりです。

```
vpath pattern directory-list
```

例で使用したVPATHはこのように書き換えることができます。

```
vpath %.c src
vpath %.l src
vpath %.h include
```

これでmakeに.cファイルと.lファイルはsrcディレクトリから探し.hファイルをincludeディレクトリから探すように指示することになります（必須項目のヘッダファイル名からinclude/を取り除くことができます）。もっと複雑な状況では、この手法を使うことで多くの頭痛の種とデバッグ時間を削減することができます。

カレントディレクトリ以外に置かれたソースファイルを見つけるという問題を解決するためにvpathを使いました。ソースファイルはソースツリーに、オブジェクトファイルは別のバイナリツリーに書かれているようなアプリケーションをどのようにして構築するかという問題は、関連してはいますが別の問題です。vpathを適正に使うことにより、この新しい問題にも対処できますが、やるべきことはすぐに複雑化しvpath単独では間に合わなくなってしまいます。この問題については後述します。

2.4 パターンルール

これまで見てきた例題のmakefileはまだ冗長です。ファイルが10以下の小さなプログラムではあまり気になりませんが、ターゲット、必須項目、そしてコマンドスクリプトを指定するファイルが数百や数千になるともう手に負えません。例題のmakefileではコマンド自体も重複しています。もしコマンドにバグがあったり変更が入ったりした場合には、すべてのルールを変更しなければなりません。これは保守に関する重大な問題であり、バグの源でもあります。

特定のタイプのファイルを読み、別のタイプのファイルを出力する多くのプログラムは標準的な慣例に従っています。例えばすべてのCコンパイラは、.cサフィックスを持つファイルはCソースコードであること、そしてオブジェクトファイル名は.cサフィックスを.o（Windowsのコンパイラは.obj）に置き換えたものであるという想定をします。前章でflexの入力ファイルは.lサフィックスを使用し、.cファイルを生成するということを紹介しました。

こういった慣例のおかげで、ファイル名のパターンを認識し処理する組み込みルールを使うことによりルールの作成作業を単純化することができます。例えば、組み込みルールを使えば17行あった例題のmakefileは次のように簡単になります。

```
VPATH = src include
CPPFLAGS = -I include

count_words: counter.o lexer.o -lfl
count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h
```

組み込みルールはパターンルールによって作られています。ファイル名の本体の部分（サフィックスの前の部分）が%で表されている点を除けば、パターンルールはこれまで見てきたルールと同じように見えます。上記のmakefileは3つの組み込みルールにより動作します。最初は.cファイルをコンパイルして.oファイルを作るルールです。

```
%.o: %.c
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

次は.lファイルから.cファイルを作るルールです。

```
%.c: %.l
        @$(RM) $@
        $(LEX.l) $< > $@
```

最後はサフィックスを持たないファイル（通常実行ファイルです）を作り出す特別なルールです。

```
%: %.o
        $(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

文法の詳細を見る前に、makeの出力をよく観察してmakeがどのように組み込みルールを適用しているのかを調べてみましょう。

上記makefileを使ってmakeを実行すると、次のような出力になります。

```
$ make
gcc -I include -c -o count_words.o src/count_words.c
gcc -I include -c -o counter.o src/counter.c
flex -t src/lexer.l > lexer.c
gcc -I include -c -o lexer.o lexer.c
```

```
gcc count_words.o counter.o lexer.o /lib/libfl.a -o count_words
rm lexer.c
```

まずmakeはmakefileを読み、コマンドラインにターゲットが指定されていなかったためcount_wordsをデフォルトターゲットに設定します。デフォルトターゲットを調べることで、4つの必須項目が必要であることを特定します。すなわちcount_words.o（これはmakefileには書かれていませんが、暗黙ルールにより必須項目となります）、counter.o、lexer.oそして-lflです。makeはこの必須項目を順に更新します。

最初の必須項目であるcount_words.oについて調べることで、明示的にルールは書かれていないけれども暗黙ルールが適用できることが判明します。カレントディレクトリを探してもソースファイルが見つからないので、VPATHを探し該当するソースファイルをsrcで発見します。src/count_words.cには必須項目がないため、makeは暗黙ルールを適用してcount_words.oを更新することがこの時点で可能となります。counter.oも同様です。lexer.oに目を移すと、ソースファイルを見つけることが（srcの中からも）できないため、makeはこのファイル（見つからないソースファイル）は中間ファイルであると想定し、lexer.cを探すのはやめて別のソースファイルを探し始めます。やがて.lファイルから.cファイルを作成するルールとlexer.lが見つかります。lexer.lの更新は必要ないので、flexコマンドを適用してlexer.cを更新する作業に取りかかります。続いてCソースファイルからオブジェクトファイルを作成します。このようなターゲットを更新するための一連のルール適用を**ルール連鎖**と呼びます。

次にライブラリ指定である-lflを調べ、標準のライブラリディレクトリを探すことで/lib/libfl.aを見つけ出します†。

これでcount_wordsすべての必須項目がそろったので、最後のgccコマンドを実行します。最後に、makeは取っておく必要のない中間ファイルを作ったことに気づいて、それを削除します。

ここで見てきたように、ルールを使うことでmakefileを詳細に書く必要がなくなります。ルールには非常に強力なふるまいをもたらす複雑な相互作用を持たせることができます。特に共通ルールの組み込みデータベースを持つことにより多くのmakefileにおいて記述を劇的に単純化にすることが可能となります。

組み込みルールはコマンドスクリプトで使われている変数の値を変更することによりカスタマイズが可能です。実行するプログラムの名前に始まり、出力ファイルの指定や最適化、デバッグなど主要なコマンドラインオプションをまとめた変数など、多くの変数が典型的なルールには使われています。make --print-data-baseを実行することによりデフォルトのルール（そして変数も）を見ることができます。

2.4.1 パターン

パターンルール中のパーセント文字（%）は、Unixシェルの*とほぼ同じで、あらゆる文字列を表しています。パーセント文字はパターン中のどこでも置けますが、一度しか使えません。以下はパー

† 訳注：OSによっては/usr/lib/libfl.aです。

セント文字の正しい用例です。

```
%,v
s%.o
wrapper_%
```

パーセント文字以外は書いたとおりにファイル名と照合されます。パターンにはプリフィックスとサフィックスの両方とも含めることができます。makeが適用すべきルールを探す際、最初に着目するのはパターンルールのターゲットです。パターンルールのターゲットは、（もし付いているなら）プリフィックスで始まりサフィックスで終わらなければなりません。パターンに合致したものが見つかると、プリフィックスとサフィックスの間にある文字列をファイル名の本体として取り出します。次にパターンルールの必須項目に着目し、ファイル名の本体を必須項目パターンに適用します。その結果に合致したファイルが見つかるか別のルールを適用することでその必須項目が作成できるならば、ルールは適合できたことになり実行されます。ファイル名の本体部分は少なくとも1文字以上の長さがなければなりません。

パーセント文字1つだけのパターンも使うことができます。これはたいていUnixの実行ファイルを作成する場合に使われます。例えば次のパターンルールはGNU makeがプログラムを作成するために備えているものです。

```
%: %.mod
        $(COMPILE.mod) -o $@ -e $@ $^

%: %.cpp
        $(LINK.cpp) $^ $(LOADLIBES) $(LDLIBS) -o $@

%: %.sh
        cat $< >$@
        chmod a+x $@
```

これらのパターンはModulaのソースとCのソースとBourne shellスクリプトから実行ファイルを作成するためにそれぞれ使われます。GNU makeに用意されているその他多くのルールは、「2.5　暗黙ルールのデータベース」で詳しく見ることにします。

2.4.2　静的パターンルール

静的パターンルールは特定のターゲットリストに対してのみ適用されるルールです。

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

このルールと普通のパターンルールとの違いは、最初に書かれている$(OBJECTS)の指定のみです。これにより$(OBJECTS)変数にリストされているファイルに対してのみルールが適用されます。

これはパターンルールと非常に似ています。$(OBJECTS)に入っているオブジェクトファイルはパターン%.oに適用され、ファイル名の本体部分が取り出されます。続いてファイル名の本体は%.cに

当てはめられ、ターゲットの必須項目となります。ターゲットのパターンに適合するものが存在していなければmakeは警告を出力します。

ターゲットファイルをサフィックスやその他のパターンで指定するよりも直接指定したほうが簡単な場合にはいつでも静的パターンルールを使うことができます。

2.4.3 サフィックスルール

サフィックスルールは間接的にルールを指定するために元から備わっている（そして旧式の）機能です。GNU makeのパターンルールの文法をサポートしていないmakeもあるので、広く配布するためのmakefileではまだサフィックスルールを目にすることがあるでしょう。そのため、この文法を理解できるようになることは重要です。目的のシステムでもGNU makeを使うのはmakefileのポータビリティからも好ましい方法だといえますが、それでも事情によりサフィックスルールを書く必要が生じる場合があり得るのです。

サフィックスルールでは1つまたは2つのサフィックスが連結されてターゲットとして記述されます。

```
.c.o:
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

必須項目のサフィックスが先でターゲットのサフィックスが後に置かれるところが少しややこしい点です。このルールは以下のルールと同じターゲットおよび必須項目と合致します。

```
%.o: %.c
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

サフィックスルールではファイル名からターゲットのサフィックスを取り除いてファイル名の本体とします。次にターゲットサフィックスを必須項目のサフィックスで置き換えることで、必須項目のファイル名を作ります。サフィックスルールは2つのサフィックスがmakeにとって既知のサフィックスである場合にかぎり、makeに正しく認識されます。

上のサフィックスルールは2つのサフィックスでできているため、**ダブルサフィックスルール**と呼ばれます。**シングルサフィックスルール**というものもあります。容易に想像できるように、シングルサフィックスルールは1つのサフィックスでできていて、それはソースファイルのサフィックスです。Unixでは実行ファイルにサフィックスをつけないため、実行ファイルを作成するためのルールとして使われます。

```
.p:
        $(LINK.p) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

このルールはPascalのソースから実行ファイルを生成します。このルールは次のパターンルールとまったく同じ意味を持ちます。

```
%: %.p
        $(LINK.p) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

既知のサフィックスのリストはサフィックスルール文法の最も奇妙な部分です。まず.SUFFIXESに既知のサフィックスリストをセットします。次は.SUFFIXESのデフォルト定義の最初の部分です。

```
.SUFFIXES: .out .a .ln .o .c .cc .C .cpp .p .f .F .r .y .l
```

makefileの中では単純に.SUFFIXESにセットするだけで独自のサフィックスを追加することができます。

```
.SUFFIXES: .pdf .fo .html .xml
```

もし（独自に決めたサフィックスに干渉してしまうなどの理由で）既定のサフィックスをすべて無効にしたいなら、単純に必須項目部分に何も書かない定義をします。

```
.SUFFIXES:
```

同じことはコマンドラインオプション--no-builtin-rules（または-r）を使っても可能です。

GNU makeのパターンルールのほうがより明快で汎用性があるため、以降本書ではこの古い文法は使わないことにします。

2.5 暗黙ルールのデータベース

GNU make 3.80にはおよそ90の組み込み暗黙ルールがあります。暗黙ルールは、パターンルールかサフィックスルールのどちらかです。組み込みのパターンルールではC、C++、Pascal、FORTRAN、ratfor、Modula、Texinfo、TeX（これにはTangleとWeaveも含みます）、Emacs Lisp、RCS、SCCSに対応しています。加えてこれらの言語に対する補助ツール、例えばcpp、yacc、lexx、tangle、weaveやdviツール群などにも対応しています。

もしこれらのツールを使っているなら、やりたいことのほとんどは暗黙ルールに定義されているでしょう。JavaやXMLなど暗黙ルールで対応していないツールを使う場合には、独自にルールを書かなければなりません。しかし、たいていはとても簡単なルールを少しだけ書けばよいのです。

makeに組み込まれているルールを調べるには、コマンドラインオプション--print-data-base（-p）を使います。これにより何百行ものルールが出力されます。バージョン情報とコピーライトが出力された後、変数がどこで定義されているのかを示すコメントとともに変数を1つずつ出力します。変数といっても例えば環境変数の場合もあれば、あらかじめ定義されている変数そして自動変数などさまざまです。変数の後にはルールが出力されます。GNU makeが出力する実際のフォーマットは次のようなものです。

```
%: %.C
# commands to execute (built-in):
        $(LINK.C) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

makefileで定義されているルールには、そのルールがどのファイルの何行目で定義されているの

かがコメントに付加されます。

```
%.html: %.xml
# commands to execute (from `Makefile', line 168):
    $(XMLTO) $(XMLTO_FLAGS) html-nochunks $<
```

2.5.1 暗黙ルールを活用する

ターゲットが認識されており、それに適用すべきルールが明示的に定義されていない場合に組み込みの暗黙ルールが適用されます。暗黙ルールを使うのはとても簡単です。単にmakefileに指定するターゲットに対してコマンドスクリプトを書かなければよいのです。これによりmakeはターゲットに適合するルールを組み込みのデータベースから探します。たいていの場合はこれでよいのですが、開発環境によってはまれに問題が生じる場合があります。例えばLispとCを両方使用した開発を行っているとしましょう。そして同じディレクトリにeditor.1とeditor.c（一方は他方から使われる下位レベルの機能とします）が存在していると仮定します。makeはLispのファイルを見て、「これはflexのファイルでありCのソースはflexコマンドの出力により作られる」と解釈します（flexはソースファイルに.1サフィックスを使うことを思い出しましょう）。editor.oがターゲットで、editor.1がeditor.cよりも新しいと、makeはCのファイルを更新しようとしてflexの出力でeditor.cを上書きしてしまうでしょう。困ったことになりました。

次のように書いてflexに関する2つのルールを暗黙ルールデータベースから取り除くことでこの問題には対処できます。

```
%.o: %.l
%.c: %.l
```

コマンドスクリプトの伴わないパターンは、makeのデータベースからルールを削除する働きを持ちます。実際には、こういった状況は非常にまれです。しかし、組み込みルールデータベースのルールはmakefileに記述した内容に対して予想もしなかった影響を与えるかもしれないと認識しておくのは重要なことです。

これまでにmakeがどのようにルールを連鎖させてターゲットを更新するか見てきました。このことは、ここで検討するある種の複雑さにつながります。makeはどのようにしてターゲットを更新しようか考える際に、そのターゲットに合致するパターンを手持ちの暗黙ルールから探し出します。ターゲットファイルに合致するそれぞれのターゲットパターンに対してmakeは既存のファイルから必須項目に合致するものを探します。つまりターゲットパターンが合致したら、makeは直ちに必須項目となるソースファイルを探すことになります。もし必須項目が見つかれば、そのルールが適用されます。いくつかのターゲットパターンに対しては、ソースファイルの候補が数多く存在してます。例えば、.oファイルは.c、.cc、.cpp、.p、.f、.r、.s、.modなどから作り出すことができます。さて、すべてのルールを調べた後でソースとなるべきファイルが見つからなかった場合はどうなるのでしょうか？ その場合makeはソースファイルとなるべきファイルを、更新すべき新たなターゲットとしてルールの再調査を行います。このように再帰的に調べることにより、makeはターゲットを更新するルール

の連鎖を見つけることができるのです。このことはすでにlexer.oの例で見てきました。中間ファイルである.cファイルは存在していませんでしたが、.lから.cを作るルールと.cから.oを作るルールを適用することでmakeはlexer.oを更新することができました。

makeが自動的に行う作業の中で最も感動的なものの1つを次に示します。まず、実験の準備として空のyaccソースファイルを作りciコマンドでRCSに登録します（つまり、バージョンコントロールされたyaccのソースファイルが必要なのです）。

```
$ touch foo.y
$ ci foo.y
foo.y,v <-- foo.y
.
initial revision: 1.1
done
```

さあ、makeに実行可能なfooをどのように作るのか聞いてみましょう。--just-printオプション（-nオプション）によりmakeは行うべき作業の内容を、実際に実行することなく出力します。ここではmakefileもありませんし、ソースファイルも存在しません。あるのはRCSのファイルだけです。

```
$ make -n foo
co foo.y,v foo.y
foo.y,v --> foo.y
revision 1.1
done
bison -y foo.y
mv -f y.tab.c foo.c
gcc -c -o foo.o foo.c
gcc foo.o -o foo
rm foo.c foo.o foo.y
```

暗黙ルールと必須項目の連鎖に従い、もしオブジェクトファイルfoo.oが存在していれば、makeは実行可能なfooを作成できると判断するでしょう。もしCソースファイルfoo.cが存在していれば、foo.oが作成できます。foo.yが存在していればfoo.cが作成できます。そして実際に存在しているRCSのファイルfoo.y,vからチェックアウトすればfoo.yを作成できることを突き止めます。この作業計画を立案した後は、実際に実行します。つまり、foo.yをcoを使ってチェックアウトし、bisonを使ってfoo.cに変換し、gccでコンパイルしてfoo.oを作成し、もう一度gccでリンクしてfooを作成します。これらはすべてルールデータベースにより可能となっています。なんとすばらしい動作でしょう。

ルール連鎖の中で作成されたファイルは**中間ファイル**と呼ばれ、makeにより特別扱いされます。まず中間ファイルはターゲットとしては現れない（でなければ、中間ファイルではありません）ので、makeは単純に中間ファイルだけを更新しません。次に中間ファイルはターゲットの生成過程における副作用の結果生じるものなので、makeは終了前に中間ファイルを削除します。例の最後の行がそれです。

2.5.2 ルールの構造

組み込みルールは、カスタマイズが簡単にできるように標準的な構造を持っています。まずその構造を簡単に眺めた後、どのようにカスタマイズするか論じることにしましょう。以下のものは（もうすでに見慣れたものですが）Cソースファイルからオブジェクトファイルを作るルールです。

```
%.o: %.c
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

ルールのカスタマイズはすべて変数の設定により行います。ここでは2つの変数を見ることができますが、特にCOMPILE.cはいくつか別の変数から作られています。

```
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
CC = gcc
OUTPUT_OPTION = -o $@
```

Cコンパイラは CC 変数の値を別のものにすることで変えることができます。コンパイルオプションを設定するために(CFLAGS)が、プリプロセッサオプション向けに(CPPFLAGS)が、アーキテクチャ特有のオプションのために(TARGET_ARCH)が使われています。

組み込みルール中の変数は、ルールをできるだけ簡単にカスタマイズできるよう使われています。このため、これらの変数をmakeファイルで設定するときには十分な注意が必要です。もしこれらの変数を不用意に設定すると、makeの持つカスタマイズ能力を殺してしまうことになりまねません。例えばmakefileで次の設定をしたとしましょう。

```
CPPFLAGS = -I project/include
```

もしmakefileの利用者がコマンドラインでCPPの設定を追加しようとしたなら、以下のように行うでしょう。

```
$ make CPPFLAGS=-DDEBUG
```

しかしそうすることで、（おそらく）コンパイルに必要な-Iオプションを図らずも消してしまうことになってしまいます。コマンドラインでの変数設定は、その他の方法で行われた変数の設定を上書きしてしまうからです（コマンドラインでの変数設定についての詳細は「3.6　変数はどこからくるのか」を参照してください）。makefileで不適切にCPPFLAGSを設定したことにより、カスタマイズが期待したとおりに機能しなくなってしまいます。単純に代入するのではなく、独自に追加する値をコンパイルの変数に入れることを検討しましょう。

```
COMPILE.c = $(CC) $(CFLAGS) $(INCLUDES) $(CPPFLAGS) $(TARGET_ARCH) -c
INCLUDES = -I project/include
```

あるいは、「3.2.1　その他の代入」で扱うアペンド形式の代入を使うこともできます。

2.6 特殊ターゲット

特殊ターゲットとは組み込みの擬似ターゲットのことを指し、makeのデフォルト動作を変更するために使われます。例えば特殊ターゲットである.PHONYについてはすでに扱いましたが、これは必須項目に指定されたものが実際のファイルとして存在せず、常に最新ではないことを示すためのものです。.PHONYは頻繁に使われる特殊ターゲットですが、その他のものも同様によく使われます。

特殊ターゲットもまた「ターゲット:必須項目」という通常の文法に従いますが、ここでのターゲットはファイルでも普通の擬似ターゲットでもありません。それは実際にはmakeの内部アルゴリズムを変更するための命令のようなものです。

特殊ターゲットは12種類あります。それらは次の3つに分類されます。1つはすでに書いたようにターゲットを更新する際のmakeの挙動を変更するためのもの。もう1つは単純に大域フラグとして働き、その必須項目を無視するためのもの。最後は.SUFFIXESで、旧式のサフィックスルールで使うものです（これはすでに「2.4.3　サフィックスルール」で取り上げました）。

（.PHONYに加えて）最も役に立つターゲットを以下に示します。

.INTERMEDIATE

このターゲットの必須項目は、中間ファイルとして扱われます。もし何らかのターゲットの更新作業において、ここで指定されたファイルを作成した場合には、makeが終了するときにファイルは消去されます。更新作業時にファイルがすでに存在していた場合には消されません。

この機能は独自のルール連鎖を作る際に役立ちます。例えば多くのJavaツールはWindowsスタイルのリストファイル†を受け付けます。リストファイルを作るためのルールを作り、リストファイルを中間ファイルとすることでmakeがこの一時的に作られたファイルを最後にまとめて削除できるようになります。

.SECONDARY

この特殊ターゲットの必須項目に指定されたファイルは中間ファイルとして扱われますが、自動的に消されることはありません。.SECONDARYはライブラリに入れるオブジェクトファイルに印を付けるために使われるのが一般的です。こういったオブジェクトファイルはアーカイブに追加された時点で削除されるのが普通です。開発の途中では、makeのアーカイブファイルを更新する機能を使いながらも、こうしたファイルを取っておいたほうが便利な場合もあります。

.PRECIOUS

makeは実行を中断された際、更新中のターゲットがもしmakeの実行中に更新されていたならそのファイルを消してしまうでしょう。これは部分的に更新された（壊れているかもしれない）ファイルをmakeは構築ツリーの中に残しておかないということを意味しています。特にファイルが巨大で作成するのに多くの計算が必要である場合など、勝手に消されては困る場合があります。そういったファイルにprecious（高価な）という印を付けることで、makeは実行を中断

† 訳注：ツールに対して処理すべきファイルを大量に指定する際に使用する、1行に1つのファイル名が書かれたファイル。

された場合でもファイルを消さなくなります。

.PRECIOUSを使う機会は多くありませんが、必要な場合には非常に助かります。ルールにより実行されたコマンドがエラーになった場合には、makeはファイルを自動的には消さないことを覚えておきましょう。消されるのはシグナルにより実行が中断された場合だけです。

.DELETE_ON_ERROR

これはある意味.PRECIOUSの反対です。ターゲットに.DELETE_ON_ERRORとして印を付けることで、ルールにより実行されたコマンドがエラーになった場合にはターゲットが削除されます。通常makeはシグナルで実行が中断された場合にのみターゲットを削除します。

その他の特殊ターゲットは、後ほど関連性のある個所で説明します。

2.7 自動的な依存関係の生成

コンピュータは検索したりパターン照合するのがとても得意です。プログラムを使ってファイル間の依存関係を特定し、ついでにそれをmakefileの文法で書き出すようにしましょう。ご賢察のとおり、こういったプログラムは、少なくともCとC++についてはすでに存在しています。gccや他のC/C++コンパイラはソースを読み取りmakefile形式の依存関係を書き出すためのオプションを持っています。例えば下記はstdio.hの依存関係を調べる方法です。

```
$ echo "#include <stdio.h>" > stdio.c
$ gcc -M stdio.c
stdio.o: stdio.c /usr/include/stdio.h /usr/include/_ansi.h   \
  /usr/include/newlib.h /usr/include/sys/config.h            \
  /usr/include/machine/ieeefp.h /usr/include/cygwin/config.h \
  /usr/lib/gcc-lib/i686-pc-cygwin/3.2/include/stddef.h       \
  /usr/lib/gcc-lib/i686-pc-cygwin/3.2/include/stdarg.h       \
  /usr/include/sys/reent.h /usr/include/sys/_types.h         \
  /usr/include/sys/types.h /usr/include/machine/types.h      \
  /usr/include/sys/features.h /usr/include/cygwin/types.h    \
  /usr/include/sys/sysmacros.h /usr/include/stdint.h         \
  /usr/include/sys/stdio.h
```

これで作業は楽になりました。しかし、読者の皆さんは「エディタを開いて、gccの実行結果をmakefileにコピー＆ペーストしなければならないとは、なんて面倒くさいんだろう」と思われることでしょう。方法がこれしかないなら、この不満は正当だといえます。自動的に生成された依存関係を取り込む伝統的な方法が2つあります。古いほうの1つは、

```
# 以下は自動生成された依存関係 - 変更不可
```

のような行をmakefileの最後に置き、生成された部分を書き換えるシェルスクリプトを使うというものです。この方法は明らかに手作業よりも改善されていますが、それでもまだ不細工です。もう1つの方法はmakeのinclude命令を使うというものです。現在ではGNU makeを始めとしてたいていのmakeがinclude命令を備えています。

そこで、すべてのファイルに対してgccを-Mオプション付きで実行するようなターゲットをmakefileに加え、その結果をファイルに保存するようにします。そしてmakeを再度実行しファイルをインクルードすることで必要な更新作業が開始されるようにします。GNU make登場以前は、次のようなルールで上記作業を行っていました。

```
depend: count_words.c lexer.c counter.c
        $(CC) -M $(CPPFLAGS) $^ > $@

include depend
```

プログラムを構築するためのmake実行に先立って、依存関係を生成するためにmake dependを実行する必要があります。これが正しく行われている間はよいのですが、ソースファイルの依存関係を修正してもdependファイルを再生成し忘れるのは、よくあることです。そしてソースが再コンパイルされず、混乱がまた始まるのです。

GNU makeはこのささいな問題を簡単な手順と気の利いた機能で解決します。最初に手順を説明します。もしソースファイル個々の依存関係をそれぞれ別のファイル、例えば.dサフィックスを持つファイルに記述し、それをソースファイルの依存関係ルールにターゲットとして置いたとしたら、makeはソースファイルが変更された際には（オブジェクトファイルとともに）.dファイルも更新しなければならないことを知ることができます。

```
counter.o counter.d: src/counter.c include/counter.h include/lexer.h
```

このルールの生成は、パターンルールと（多少見栄えは悪いのですが）コマンドスクリプト（これはGNU makeのマニュアルからそのまま持ってきました）により達成できます†。

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

次に気の利いた機能です。makeはinclude命令に現れるファイルを更新すべきターゲットとして扱います。そのため、.dファイルをインクルードするべく記述するとmakefileを読み込む際にmakeは自動的に.dファイルを作成します。以下は自動依存関係生成機能を加えたmakefileです。

---

† これは興味深い小さなスクリプトですが、少し解説する必要があると思います。まず、Cコンパイラを-Mオプション付きで実行し、ターゲットに関する依存関係を保持する一時ファイルを作成します。ファイル名は$@によるターゲットのファイル名と$$$$によるユニークな数字列のサフィックスから作られます。シェルでは$$を現在実行しているシェルのプロセス番号として解釈し、プロセス番号はそれぞれ異なるので、この過程で独自のファイル名が作成されます。次にsedを使い.dファイルをルールのターゲットに追加します。sedに指定した式は検索部'\($*\)\.o[:]*と置換部\1.o $@ :からできていて、それらはカンマで分けられています。検索部は最初にターゲットファイル名の本体部を表す$*が正規表現（RE）グループを表す\(\)で囲われていて、ファイルのサフィックスとなる\.oが続いています。ターゲットファイル名に続いて、0個以上の空白またはコロンを表す[:]*が置かれています。置換部では\1.oにより最初のREグループで参照できる元のターゲットを取り出しサフィックスを加えます。そして$@で依存ファイルターゲットを加えます。

```
VPATH = src include
CPPFLAGS = -I include

SOURCES = count_words.c   \
          lexer.c         \
          counter.c

count_words: counter.o lexer.o -lfl
count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h

include $(subst .c,.d,$(SOURCES))

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

デフォルトターゲットが依存関係ファイル内のルールで置き換えられないよう、include命令は手で設定した依存関係の後に置くべきです。include命令にはファイル名のリスト（ワイルドカードを含んでもかまいません）を指定することができます。ソースファイルのリストを依存関係ファイルのリストに変換するため、ここではmakeの関数であるsubstを使っています（substは「4.2.1　文字列関数」で扱います）。今は、$(SOURCES)中の各単語に対して.cの部分を.dに置き換えているということを知っていれば十分です。

--just-printオプション付きでmakeを実行すると、次の出力が得られます。

```
$ make --just-print
Makefile:13: count_words.d: No such file or directory
Makefile:13: lexer.d: No such file or directory
Makefile:13: counter.d: No such file or directory
gcc -M -I include src/counter.c > counter.d.$$;                    \
sed 's,\(counter\)\.o[ :]*,\1.o counter.d : ,g' < counter.d.$$ >
counter.d;                                                         \
rm -f counter.d.$$
flex -t src/lexer.l > lexer.c
（以下省略）
```

最初に出力されているmakeのメッセージは、何やらエラーメッセージのようで驚いてしてしまいます。しかし心配はいりません、これはただの警告です。makeはインクルードファイルを探したけれども見つからなかったので、ファイルを作成するためのルールを探す前に、そのようなファイルやディレクトリは存在しないという警告を出力したのです。この警告はinclude命令の前にマイナス記号（-）を置くことで抑制することができます。この警告に続く行でmakeはgccを-Mオプション付きで実行し、次にsedコマンドを呼び出していることがわかります。makeはgccを実行してlexer.cを作成した後、デフォルトゴールを作成する前に一時ファイルであるlexer.cを削除することを忘れてはいけません。

以上で依存関係の自動生成について雰囲気はつかめたかと思います。その他の言語に対する方法や構築ツリー構造を自動生成する方法など説明することはまだまだあります。この話題は6章以降で再度扱うことにしましょう。

2.8 ライブラリ管理

通常、簡単にライブラリとかアーカイブとか呼ばれる**アーカイブライブラリ**は、メンバと呼ばれる他のファイルを内包している特殊な形式のファイルです。アーカイブは関連するオブジェクトファイルを管理しやすい単位にまとめるために使われます。例えばC標準ライブラリである`libc.a`には基本的なCの関数が入っています。ライブラリは一般的に使われるものなので、ライブラリを作成、保守、参照するための特殊な機能をmakeは提供しています。アーカイブの作成および変更は`ar`プログラムを使って行います。

それでは例を見ていきましょう。単語カウントプログラムを変更して再利用可能な部品をライブラリに入れるようリファクタリングします。ライブラリには`counter.o`と`lexer.o`が入ります。このライブラリを作成するarコマンドは次のとおりです。

```
$ ar rv libcounter.a counter.o lexer.o
a - counter.o
a - lexer.o
```

`rv`オプションはリストされたオブジェクトでアーカイブのメンバを置き換えることと、arの動作を詳細に出力することを指示します。この置き換えオプションはアーカイブが存在していなくても使うことができます。オプションの後ろにはアーカイブの名前が、次にオブジェクトファイルの名前が続きます（アーカイブが存在していない場合には`c`オプションが必要なarも存在しますが、GNUのarには必要ありません）。

arコマンドに続く2行は、オブジェクトファイルが追加されたというarコマンドからの出力です。

置き換えオプションを使うことでアーカイブの作成や更新を順次行うことができます。

```
$ ar rv libcounter.a counter.o
r - counter.o
$ ar rv libcounter.a lexer.o
r - lexer.o
```

ここでは`r`がarの出力に現れました

ライブラリはいくつかの方法で実行形式ファイルとリンクすることができます。最も素直な方法は、ライブラリファイルをコマンドラインで指定することです。コンパイラまたはリンカはコマンドライン上に書かれたファイル名のサフィックスからファイルの種類を判別し、それを適切に扱います。

```
cc count_words.o libcounter.a /lib/libfl.a -o count_words
```

ccは`libcounter.a`と`/lib/libfl.a`をライブラリであると認識し、未解決のシンボルを探すた

めに使います。ライブラリは-lオプションを使っても指定できます。

```
cc count_words.o -lcounter -lfl -o count_words
```

このオプションを使うと、ライブラリファイル名のプリフィックス（lib）とサフィックス（.a）を省略できます。-lオプションにはコマンドラインをコンパクトに読みやすくするだけではなく、もっと便利な機能を提供しています。ccコマンドが-lオプションを見つけるとシステムの標準ライブラリディレクトリでライブラリファイルを探します。これによりプログラマはライブラリファイルの正確な場所を把握する必要がなくなり、コマンド行のポータビリティも高くなります。シェアードライブラリ（Unixシステムでは.soサフィックスが付いたライブラリです）をサポートしているシステムでは、リンカはアーカイブライブラリの前にシェアードライブラリを探します。これにより特に指定しなくてもシェアードライブラリの利点を享受できることになります。これはGNUのリンカおよびコンパイラの既定動作です。古いリンカやコンパイラには、この最適化を行わないものがあります。

検索するディレクトリの順番を-Lオプションに指定することでコンパイラの検索場所を変更できます。指定されたディレクトリはシステムライブラリディレクトリの前に追加され、コマンドラインオプションの-lで指定されたライブラリファイルすべてに対して有効となります。実際に最後に示した例では、カレントディレクトリがccの検索対象ではないため失敗します。この問題は次のようにカレントディレクトリを追加することで解決できます。

```
cc count_words.o -L. -lcounter -lfl -o count_words
```

ライブラリはプログラムの構築作業を少しだけ複雑にします。makeはどのようにしてこの作業を簡単にするのでしょう。GNU makeにはライブラリの作成とリンクの両方に対して特殊な機能を提供しています。それらがどのように働くのかを見ていきましょう。

2.8.1 ライブラリの作成と更新

makefileでは、ライブラリも他のファイルと同じように指定します。次は、例題のライブラリを作成するための簡単なルールです。

```
libcounter.a: counter.o lexer.o
        $(AR) $(ARFLAGS) $@ $^
```

このルールは組み込みの定義ARからarプログラムを、同じくARFLAGSから標準オプションのrvを得ます。自動的にアーカイブの出力ファイルが$@に、必須項目が$^にセットされます。

ここでcount_wordsの必須項目であるlibcounter.aを作ると、makeは実行形式ファイルにリンクする前にライブラリを更新します。しかしながら多少問題もあります。この場合アーカイブ中すべてのメンバは変更の有無にかかわらず置き換えられてしまいます。これは時間の無駄ですし、もう少しよい方法があります。

```
libcounter.a: counter.o lexer.o
        $(AR) $(ARFLAGS) $@ $?
```

$^の代わりに$?を使ったら、makeはターゲットより後で更新されたファイルだけをarに渡すようになるでしょう。

さらに改善できるでしょうか。できるかもしれないし、できないかもしれません。makeはアーカイブ中のファイルそれぞれに対してarコマンドを実行することで個々に更新する機能を提供していますが、この機能について掘り下げる前にライブラリの構築手法に関して注目すべき点がいくつかあります。makeの主たる目的は更新すべきファイルだけを処理することで作業を効率よく行うというものです。あいにく更新すべきメンバそれぞれに対してarコマンドを実行していては、すぐに行き詰ってしまいます。アーカイブに何十ものファイルが入っていたとすると、個々のファイルに対してarを実行するために払うコストは、これから紹介するエレガントな文法の持つ価値を凌駕してしまうでしょう。上で書いたようなarを明示的に実行するルールを使うことで、すべてのファイルに対してarを一度だけ実行すれば、数多くのよけいな実行を行わないで済みます。さらに加えると、多くのシステムにおいてarのrオプションはあまり効率的でない実装になっています。1.9GHzのPentium4プロセッサで大きなアーカイブ（14,216個のメンバを持ちサイズは55MBになります）を最初から作ると4分24秒かかりました。しかし、1つのオブジェクトファイルをar rコマンドで置き換えると28秒かかります。つまり（14,216個のうちで）更新するファイルが10を超えると、最初から作り直したほうが速いということになってしまいます。このような状況では自動変数$?を使うことですべての更新されたオブジェクトファイルに対して1回の置き換えだけを行うことを慎重に検討する必要があるでしょう。小さなライブラリで速いプロセッサを使っているなら、パフォーマンス上の理由からこれから紹介するエレガントな手法よりも上記の単純な方法を選ぶ必要はありません。そのような状況では下記のライブラリ機能を使うのはよい方法です。

GNU makeではアーカイブのメンバは以下の記法で参照することができます。

```
libgraphics.a(bitblt.o): bitblt.o
        $(AR) $(ARFLAGS) $@ $<
```

ここでlibgraphics.aがライブラリ名、bitblt.o（bit block transfer）がメンバ名です。*libname*.a(*module*.o)という記法によりライブラリに入っているモジュールを表します。このターゲットの必須項目は単純にそのオブジェクトファイルであり、コマンドはアーカイブにオブジェクトファイルを追加するというものです。コマンドで使われている自動変数$<は必須項目の最初のものを取り出します。実際に組み込みのパターンルールではこれと同じことを行っています。

これらすべてを詰め込み、makefileは次のようになりました。

```
VPATH = src include
CPPFLAGS = -I include

count_words: libcounter.a /lib/libfl.a

libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)

libcounter.a(lexer.o): lexer.o
        $(AR) $(ARFLAGS) $@ $<
```

```
libcounter.a(counter.o): counter.o
        $(AR) $(ARFLAGS) $@ $<

count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h
```

makeを実行すると、次のように実行過程が表示されます。

```
$ make
gcc -I include -c -o count_words.o src/count_words.c
flex -t src/lexer.l> lexer.c
gcc -I include -c -o lexer.o lexer.c
ar rv libcounter.a lexer.o
ar: creating libcounter.a
a - lexer.o
gcc -I include -c -o counter.o src/counter.c
ar rv libcounter.a counter.o
a - counter.o
gcc count_words.o libcounter.a /lib/libfl.a -o count_words
rm lexer.c
```

アーカイブ更新用のルールについて一言。ターゲットとしてlibcounter.a(lexer.o)と記述されていても自動変数$@はライブラリ名を表します。

最後に、アーカイブライブラリは収容しているシンボルのインデックスを持っていることに触れる必要があります。GNU arのように最近のアーカイブプログラムでは、新しいモジュールがアーカイブに追加された際には自動的にこのインデックスを生成します。しかし古いバージョンのプログラムでは、そうなりません。ranlibという別のプログラムを使ってアーカイブのインデックスを作成したり更新します。そういったシステムではアーカイブを更新するための組み込みの暗黙ルールでは不十分です。その場合には次のようなルールが必要となります。

```
libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)
        $(RANLIB) $@
```

大きなアーカイブに対する別の手段を選択した場合にはこのようになります。

```
libcounter.a: counter.o lexer.o
        $(RM) $@
        $(AR) $(ARFLGS) $@ $^
        $(RANLIB) $@
```

もちろんアーカイブのメンバを扱うための文法は組み込みのルールとともに使うこともできます。GNU makeはアーカイブを更新する組み込みルールを提供しているので、例題のmakefileは次のように書くことができます。

```
VPATH = src include
CPPFLAGS = -I include
```

```
count_words: libcounter.a -lfl
libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)
count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h
```

2.8.2 ライブラリを必須項目として使う

ライブラリが必須項目として指定される場合、普通にファイル名を指定しても-lを使う書き方をしてもかまいません。ファイル名を使う場合は次のように指定します。

```
xpong: $(OBJECTS) /lib/X11/libX11.a /lib/X11/libXaw.a
        $(LINK) $^ -o $@
```

リンカはコマンドラインで指定されたファイルを読み込み、普通に処理します。-l形式が使われた場合には、必須項目には普通のファイルとして記述されません。

```
xpong: $(OBJECTS) -lX11 -lXaw
        $(LINK) $^ -o $@
```

必須項目が-l形式で指定された場合、makeはライブラリ（できればシェアードライブラリ）を探し出し、絶対パス名に置き換えて^と?にセットします。この形式が非常に優れているのは、システムのリンカがライブラリの検索やシェアードライブラリの優先といった機能を提供しない場合でも、makeにそれを行わせることができるという点です。別の利点として、ライブラリを検索する場所を修正してアプリケーションのライブラリをシステムのライブラリと同じように探し出せることがあげられます。上記の例において、最初の形式ではリンク行で指定されているとおりにシェアードライブラリではなくアーカイブライブラリが使われることになります。次の-l形式ではシェアードライブラリのほうが好ましいことをmakeは知っているので、アーカイブライブラリに決定する前にX11のシェアードライブラリ版を探そうとします。-l形式で指定されるライブラリファイルとして認識されるパターンは、.LIBPATTERNSに格納されているので、修正を加えることで別のライブラリファイル名形式に対しても適応させることができます。

残念なことに小さな落とし穴が1つあります。makefileでライブラリをターゲットとして指定する場合、そのファイルを別のターゲットの必須項目に-l形式で指定することができません。例えば、

```
count_words: count_words.o -lcounter -lfl
        $(CC) $^ -o $@

libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)
```

というmakefileでは以下のエラー（「count_wordsで必要な-lcounterを作るルールがありません」の意）になります。

```
No rule to make target `-lcounter', needed by `count_words'
```

このエラーはmakeがターゲットを探す際に-lcounterをlibcounter.aに展開せず、そのまま探そうとするのが原因です。そのため同じmakefileで作り出すライブラリはファイル名を指定する形式を使わなければなりません。

複雑なプログラムをエラーなしでリンクするのは、ちょっとした黒魔術みたいなものです。リンカはコマンドラインで指定された順番でライブラリを探そうとします。そのため、もしライブラリAが未解決のシンボルを含んでいて、それがライブラリBで定義されていた場合、リンカのコマンドラインではBの前にAを置く必要があります（つまりAはBを必要としています）。BをAの前に置いてしまうと、リンカがAを読み込んで未解決のシンボルが出てきたとしても、時すでに遅しです。リンカは後戻りしません。このようにコマンドライン上のライブラリの順番は基本的に重要です。

必須項目が^や?変数に格納される際、その順番は維持されます。前述の例で使った$^には必須項目に書かれたとおりの順番でファイル名が入っています。これは必須項目が複数のルールに分かれていても成り立ちます。その場合、各ルールの必須項目は現れた順に追加されることになります。

2.8.3 二重コロンルール

二重コロンルールは、どの必須項目が新しくなったかでターゲットを更新するためのコマンドを変えたいという要求を満たすための、わかりにくい機能です。通常、同じターゲットが複数回使われると、すべての必須項目は1つの長いリストにまとめられて、更新するための1つのコマンドと関連づけされます。しかし二重コロンルールでは同じターゲットが出てくるたびに、それぞれ別のものとして扱われます。このことは、あるターゲットに対しては必ずすべて二重コロンルールか1つのコロンルールで統一しなければならないことを意味しています。

実際のところ、この機能の実用的な例を手に入れるのは難しいので（そのあたりが、この機能をわかりにくくしている理由なのですが）、次の例はちょっと不自然なものとなります。

```
file-list:: generate-list-script
        chmod +x $<
        generate-list-script $(files) > file-list

file-list:: $(files)
        generate-list-script $(files) > file-list
```

file-listターゲットは2つの方法で生成することができます。生成スクリプトが更新された場合には、まずスクリプトを実行可能にしてからそれを実行します。ソースファイルが更新された場合にはスクリプトを実行するだけです。多少取って付けたような例ですが、この機能をどのように使うかは伝わると思います。

makeの本質である変数とコマンドとともに、makeルールの多くの機能を取り上げました。主として特定の文法と機能のふるまいについて着目しましたが、より複雑な状況でどのように適用するのかを詳しく掘り下げることはしませんでした。それらは6章以降のテーマです。さしあたり変数とコマンドについて議論を続けることにしましょう。

3章
変数とマクロ

これまでもmakefileの変数について見てきましたし、それらが組み込みルールやユーザ定義のルールでどのように使われているのか多数の例も紹介しました。しかしそれらの例はほんのさわりの部分でしかありません。変数とマクロ†はもっと複雑で、GNU makeに驚くべきパワーを与えるものなのです。

先に進む前に再確認しておきます。makeがある意味で2つの言語を1つにしたものだと理解することは重要です。1つはターゲットと必須項目から成る依存関係グラフを表現します（この言語は2章で扱いました）。もう1つはテキストの置換を行うマクロ言語です。他のマクロ言語つまりCのプリプロセッサ、m4、TeX、マクロアセンブラについてはよくご存じかと思います。それらのマクロ言語と同じように、makeのマクロも長い文字列を短い単語で定義し、短い単語を使ってプログラムを書くことができます。マクロプロセッサは短い単語を認識し、それを長い文字列に展開します。makefileの変数を古典的なプログラム言語の変数と同じようにとらえると考えやすいのですが、マクロの「変数」と「古典的な」変数との間には違いが存在します。マクロ変数は「その場」で展開され、後でさらに展開されるかもしれない文字列に置き換わります。この違いは先に進むにつれ明確になります。

変数名にはたいていの記号を含めほとんどの文字が使えます。空白が入ってもかまいませんが、普通に考えるなら使わないほうが無難です。変数名として実際に使えないものは、`:`と`#`と`=`だけです。

変数名は大文字と小文字を区別するので、ccとCCは異なる変数を表します。変数に入っている値を取り出すには、`$( )`で囲みます。特殊なケースとして変数名が1文字の場合には括弧を省略して「$文字」の形式が使えます。自動変数に括弧がないのは、この理由によります。一般的な規則として1文字変数を避けて、括弧で囲む形式を使うのがよいでしょう。

変数は`${CC}`のように中括弧で囲ってもかまいません。古いmakefileではこうした形式をよく見ると思います。形式を混ぜて使う利点はあまりないので、1つの形式を選んでそれを使い続けるのが上

† 訳注：本書の前身である『make改定版』ではここで扱うようなものをすべてマクロと表現していました。変数とマクロは本質的には同じものですが、本書では明確に区別し、これまでマクロと呼んでいたものをここでは変数と、GNU makeの機能であるdefine命令を使って作られたものをマクロと呼んでいます。

手な使い方です。人によってはシェルが行うのと同じように変数には中括弧を使い関数呼び出しには括弧を使う場合もあります。最近のmakefileの書き方は括弧を使うというもので、本書でもその方式にならいます。

慣例により、コマンドラインや環境変数で変更するかもしれない定数はすべて大文字で書くことになっています。単語は下線（アンダースコア）で区切ります。makefileの中だけで使われるものは下線で単語を区切った小文字の名前を持ちます。そして本書では変数やマクロで使われるユーザ定義関数にはマイナス記号（-）で単語を区切った小文字の名前を使います。他のネーミング規則は出てきた時点で説明します（下の例ではまだ扱っていない機能を含みます。ここでは命名規則を示すのが目的なので、現時点では各行の右辺についてはあまり深く考えないでください）。

```
# 簡単な定数
CC := gcc
MKDIR := mkdir -p

# 内部で使用する変数
sources = *.c
objects = $(subst .c,.o,$(sources))

# 関数を2つ
maybe-make-dir = $(if $(wildcard $1),,$(MKDIR) $1)
assert-not-null = $(if $1,,$(error Illegal null value.))
```

変数の値は代入記号の右辺にあるものから先頭の空白を取り除いたすべてです。後ろに続く空白は取り除かれません。例えば空白を含んだ変数がコマンドスクリプトで使われてしまうと、しばしばトラブルの種となります。

```
LIBRARY = libio.a # LIBRARY には後ろに空白が含まれている
missing_file:
        touch $(LIBRARY)
        ls -l | grep '$(LIBRARY)'
```

変数の代入に後続の空白が含まれることはコメントが後ろに続くことで明らかにしています（コメントがなくても空白を後ろに付けることはできます）。このmakefileを実行すると次のような結果になります。

```
$ make
touch libio.a
ls -l | grep 'libio.a '
make: *** [missing_file] Error 1
```

grepの検索文字列に空白が入ってしまったため、lsの出力から目的のファイルを見つけることができませんでした。空白の問題については後でじっくり扱います。今は変数に関しての話題を深く追って行きましょう。

3.1 変数を何に使うべきか

一般的には他のプログラムを表すのに変数を使うのはよい方法です。変数を使うことにより特殊な環境に対してmakefileを適合させやすくなります。例えばたいていのシステムにはawk、nawk、gawkなどいろいろなawkが存在します。AWKという変数を作りawkプログラムの名前を入れておくことで、そのmakefileは他のユーザにとっても使いやすいものになります。同様にセキュリティが問題になる場合、他のプログラムを絶対パスで指定することによりユーザのパスを使うことで生じる問題を回避するのは賢い方法です。絶対パスで指定することにより、ユーザパスのどこかにインストールされているかもしれないトロイの木馬プログラムを起動してしまうリスクを減少させることができます。もちろん絶対パスを指定することでmakefileの他のシステムに対するポータビリティは落ちてしまいます。どちらを選択するかはmakefileをどのように使うかによって変わります。

最初に見た変数は単純な定数を保持するものでしたが、次のようにディスクの空き容量を調べるためのユーザ定義コマンド列を保持することもできます†。

```
DF = df
AWK = awk
free-space := $(DF) . | $(AWK) 'NR == 2 { print $$4 }'
```

変数にはこれら以外にも数多くの用途で使用できます。

3.2 変数の種類

makeには2種類の変数があります。1つは**単純展開変数**でもう1つは**再帰展開変数**です。単純展開変数（または単純変数）は、:=代入演算子を使って定義されます。

```
MAKE_DEPEND := $(CC) -M
```

これが単純展開と呼ばれるのは、代入行がmakefileから読み込まれると演算子の右辺が即時評価されるためです。変数参照は値に展開され、その結果の値が変数に代入されます。この動作はたいていのプログラミング言語やスクリプト言語のふるまいと同じです。たとえばこの変数が普通に展開されると次のように評価されます。

```
gcc -M
```

しかしもしCCに値が設定されていなければ、上記代入により次の値を持つことになります。

```
<空白>-M
```

† dfコマンドはマウントされたファイルシステムの容量と使用情報の一覧を出力します。コマンドに引数があれば、指定されたファイルシステムの情報を出力します。出力の最初の行は、出力列のタイトルです。出力はawkによって検査され2行目以外は無視されます。そして4番目のカラムがファイルシステムの空きブロック数を示しています。

$(CC)はその値（この場合空の文字列）に展開され、結局何も起こりません。変数に値が設定されていないことはエラーではありません。事実このほうが好都合なのです。暗黙ルールのたいていのコマンドは、カスタマイズのためのプレースホルダとして設定されていない変数を使っています。もしカスタマイズが行われなければ、変数はなかったことと同じになります。ここで先頭の空白について説明しましょう。makeによって最初に評価されたとき、演算子の右辺は、$(CC) -Mになります。変数の参照が何も得ずに終わった時点で、makeは値の再走査も行わなければ空白の削除も行いません。そこで空白はそのまま残るのです。

もう1つの変数は再帰展開変数と呼ばれます。再帰展開変数は=代入演算子を使って定義されます。

```
MAKE_DEPEND = $(CC) -M
```

これが再帰展開と呼ばれるのは、演算子の右辺がmakeにより記録されるだけで値の評価や展開がまったく行われないことに由来しています。値の展開は変数が使われた時点で行われます。使われるまで評価が行われないことから、この種類の変数は**遅延評価変数**と呼んだほうがよいのかもしれません。この変数展開方式の驚くべき効果は、変数定義をどのような順に行ってもよいという点にあります。

```
MAKE_DEPEND = $(CC) -M
...
# どこか後ろのほうで
CC = gcc
```

変数CCはMAKE_DEPENDが代入された後で設定されていますが、MAKD_DEPENDの値はgcc -Mになります。

実際には再帰変数で遅延代入が行われているわけではありません（少なくとも普通の遅延代入という意味では）。再帰変数は使われるたびに演算子の右辺は再評価されます。上に示したようなMAKE_DEPENDのように単純な定数の場合には、演算子の右辺は単純な定数であるため、この指摘は意味がありません。しかし、右辺がdateのようなコマンドの実行を表していた場合を考えてみましょう。再帰変数が展開されるたびにdateプログラムは実行され、その結果毎回異なる値を持つことになります（少なくとも1秒以上は間を開けて実行した場合）。このことは状況によっては有益なのですが、場合によっては迷惑だったりします。

3.2.1 その他の代入

ここまでに2種類の変数代入を見てきました。=は再帰変数を作成し、:=は単純変数を作成します。これ以外にもう2種類の代入演算子をmakeは提供してます。

?=は**条件付き代入演算子**と呼ばれます。ちょっと長いので条件付き代入と呼びましょう。?=演算子は変数が値を持っていない場合にかぎり指定された代入を実行します。

```
# $(PROJECT_DIR)/outにすべての生成されたファイルを入れておく
OUTPUT_DIR ?= $(PROJECT_DIR)/out
```

これによりOUTPUT_DIRという出力ディレクトリ変数が値を持っていない場合に、代入が行われま

す。この機能は環境変数に対しとてもうまく協調します。「3.6　変数はどこからくるのか」でもっと詳しく解説します。

最後に紹介する+=代入演算子 は普通**アペンド**（追加）と呼ばれます。その名が示すようにこの演算子は変数にテキストを追加します。これは平凡な機能のように思えますが、再帰変数とともに使われる場合には注意が必要です。具体的にいうと、代入の右辺に変数の元の値が変更されずに使われることになります。「それがどうした。アペンドとはそういうものだろう？」という声が聞こえてきます。そのとおりなのですが、ここでは多少用心しなければなりません。

単純な変数に対する追加は、明らかです。+=演算子はおそらく次のように実現されているでしょう。

```
simple := $(simple) 追加分
```

simple変数の値はすでに決定しているので、makeは$(simple)を展開しテキストを加えて代入を終了することができます。しかし再帰変数の場合には問題が生じます。次のようには実現することができません。

```
recursive = $(recursive) 追加分
```

makeはこれをうまく扱う方法を持たないため、エラー（再帰変数‘recursive’は自分自身を参照している）となります。もしmakeが現在のrecursiveの定義に追加分を加えた値を記録すると、makeは実行時に再展開することができなくなってしまいます。さらにいうと自分自身への参照を含んだ再帰変数は無限ループを作り出してしまいます。

```
$ make
makefile:2: *** Recursive variable `recursive' references itself
(eventually). Stop.
```

そのため+=演算子は特に再帰変数に対しての追加ができるように適切に実現されています。この演算子は何らかの値を順次集めて保持するのに役立ちます。

3.3　マクロ

変数は1行のテキストを保持するには都合のよいものです。では、いろいろな場所で使われる複数行のコマンドを保持したいときにはどうすればよいでしょうか。次のコマンド列はJavaアーカイブ（またはjarファイル）をJavaのクラスファイルから作成する際に使われるものです。

```
echo Creating $@...
$(RM) $(TMP_JAR_DIR)
$(MKDIR) $(TMP_JAR_DIR)
$(CP) -r $^ $(TMP_JAR_DIR)
cd $(TMP_JAR_DIR) && $(JAR) $(JARFLAGS) $@ .
$(JAR) -ufm $@ $(MANIFEST)
$(RM) $(TMP_JAR_DIR)
```

このような長いコマンド列の最初には簡単なメッセージを出力するのはよい方法です。これによりmakeの出力が読みやすくなります。メッセージの後で、クラスファイルを一時ディレクトリに集めます。ここでは一時的なjarディレクトリを削除して古いファイルが残らないようにしてから†、新しいディレクトリを作成します。次に必須項目のファイルを（およびそのサブディレクトリも含めて）一時ディレクトリにコピーします。続いてカレントディレクトリを一時ディレクトリに移し、jarファイルをターゲットのファイル名で作成します。マニフェストファイルをjarファイルに加えて、最後に整理します。このコマンド列をあちらこちらにそれぞれ持つのは、将来の保守を考えると明らかによくありません。このコマンド列をすべて再帰変数に格納するという手もありますが、保守しにくいうえにmakeの行うコマンドライン出力が読みにくいものになってしまいます（すべてのコマンド列が非常に長い1行のテキストになります）。

かわりにdefine命令で作成されるGNU makeの「**慣用コマンド列**（canned sequence）」を使うことができます。「慣用コマンド列」という呼び方は多少不恰好なので、ここではマクロと呼ぶことにしましょう。マクロはmakeで変数を定義するもう1つの方法であり、改行を組み込むことのできる唯一の方法です。GNU makeのマニュアルでは変数とマクロという言葉を同義的に使用しています。本書ではdefine命令で作られる変数を特にマクロと呼び、代入により作られるもののみを変数と呼びます。

```
define create-jar
  @echo Creating $@...
  $(RM) $(TMP_JAR_DIR)
  $(MKDIR) $(TMP_JAR_DIR)
  $(CP) -r $^ $(TMP_JAR_DIR)
  cd $(TMP_JAR_DIR) && $(JAR) $(JARFLAGS) $@ .
  $(JAR) -ufm $@ $(MANIFEST)
  $(RM) $(TMP_JAR_DIR)
endef
```

define命令の後にはマクロ名と改行が続きます。endefキーワードだけを含む行までの文字列がすべてマクロの値となります。defineによって作られたマクロは他の変数とほとんど同じように展開されますが、コマンドスクリプトの中で使われた場合には、各行の頭にタブがあるかのように扱われます。以下は使い方の例です。

```
$(UI_JAR): $(UI_CLASSES)
        $(create-jar)
```

echoコマンドの前に@文字が置かれていることに注目してください。先頭に@のあるコマンド行はコマンドがmakeにより実行される際にコマンド自体は表示されなくなります。そのためmakeを実行した際にはechoコマンドを実行したことは表示されず、echoコマンドからの出力だけが表示されることになります。@プリフィックスがマクロの中で使われた場合、それぞれの行に対して効力を持つこ

† 削除をうまく働かせるためには、RM変数の値はrm -rfである必要があります。実際にはデフォルトの値は安全だけれども使いにくいrm -fになっています。さらにいうと、MKDIRはmkdir -pであるべきです。

とになります。一方、マクロを参照する側でプリフィックスを使った場合にはマクロ全体に対してコマンドの実行が表示されなくなります。

```
$(UI_JAR): $(UI_CLASSES)
        @$(create-jar)
```

この実行結果は次のようになります。

```
$ make
Creating ui.jar...
```

@の使い方は「5.1.2　コマンド修飾子」で詳しく解説します。

3.4　変数はいつ展開されるか

これまでに変数が展開される際の微妙な点について経験しました。それは、どこでどんなものが定義されていたかに大きく依存しています。make自体は何もエラーとしなかったにもかかわらず、思ったとおりにならないことがよくあります。変数展開のルールはどのようなものなのでしょうか。そしてそれはどのように働いているのでしょうか。

makeは実行された際に、2段階に分かれた作業を行います。最初の段階ではmakefileとそこからインクルードされたmakefileを読み込みます。このとき変数とルールがmakeの内部データベースに格納され、依存関係グラフが作成されます。次の段階では依存関係グラフを解析し更新すべきターゲットを特定した後、更新に必要なコマンドを実行します。

再帰変数かdefine命令を処理する際に、変数内の行またはマクロの本体が改行も含めて展開されないまま保存されます。マクロ定義の最後にある改行はマクロの一部としては保存されません。マクロが展開されるときにmakeが最後の改行を補います。

マクロが展開される際には、展開されたテキストが他のマクロや変数参照を含んでいないか直ちに検査され、見つかるとそれが展開されるという具合に再帰的な処理が行われます。もしマクロがコマンドの中で使われていた場合には、各行頭にタブが追加されます。

makefileの要素が展開されるルールをまとめてみました。

- 変数の代入における代入の左辺は、第1段階でmakeが代入行を読み込んだときに展開される
- =か?=の右辺は、第2段階にて使われるまで展開されない
- :=の右辺は、直ちに展開される
- +=の左辺が単純変数として定義されたものである場合には、右辺も直ちに展開される。そうでなければ、右辺の展開は後で行われる
- （define命令を使った）マクロ定義では、マクロ名は直ちに展開されるが、本体は使われるまで展開されない
- ルールでは常にコマンドが後で展開されるが、ターゲットと必須項目は直ちに展開される

表3-1に変数が展開される際の挙動をまとめました。

表3-1　展開が行われるタイミングのルール

代入文 定義	左辺 aの展開	右辺 bの展開
a = b	直ちに	後で
a ?= b	直ちに	後で
a := b	直ちに	後で
a += b	直ちに	後で、または直ちに

define a
  b...
  b...
  b...
endef
	直ちに	後で

一般的なルールとして変数やマクロは使われる前には定義されていなければなりません。特に変数がターゲットや必須項目で使われている場合には必須となります。

次の例を見ればはっきりするでしょう。ディスクの空きを調べるマクロを作り直してみます。個々の部分をそれぞれ調べてから、最後にまとめることにしましょう。

```
BIN := /usr/bin
PRINTF := $(BIN)/printf
DF := $(BIN)/df
AWK := $(BIN)/awk
```

マクロで使用するコマンド名を保持するために変数を3つ定義しました。コードの二重化を避けるためにbinディレクトリを抽出して4番目の変数に格納します。この4つの変数は単純変数なので、これらが読み込まれた際には定義の右辺が直ちに展開されます。BINはその他の変数に先立って定義されているため、その値はそれぞれの変数定義で適切に使われます。

次にfree-spaceマクロを定義します。

```
define free-space
  $(PRINTF) "Free disk space "
  $(DF) . | $(AWK) 'NR == 2 { print $$4 }'
endef
```

define命令の後の変数名は直ちに展開されます。この場合、展開は必要ありません。マクロの本体は展開されることなく保存されます。

最後に、このマクロをルールの中で使用します。

```
OUTPUT_DIR := /tmp

$(OUTPUT_DIR)/very_big_file:
        $(free-space)
```

$(OUTPUT_DIR)/very_big_fileの行が読み込まれ、ターゲットと必須項目で使われている変

数が直ちに展開されます。ここでは$(OUTPUT_DIR)が/tmpに展開され、ターゲットが/tmp/very_big_fileとなります。次にこのターゲットに関連づけられたコマンドスクリプトが読み込まれます。行頭にタブが置かれていることによりコマンド行として認識され、展開されないまま保存されます。

以下はmakefileの完成版です。各要素の順番はmakeの評価アルゴリズムを明らかにするため、意図的に変えてあります。

```
OUTPUT_DIR := /tmp

$(OUTPUT_DIR)/very_big_file:
        $(free-space)

define free-space
  $(PRINTF) "Free disk space "
  $(DF) . | $(AWK) 'NR == 2 { print $$4 }'
endef

BIN := /usr/bin
PRINTF := $(BIN)/printf
DF := $(BIN)/df
AWK := $(BIN)/awk
```

makefile中の順番は逆になっているように思えますが、これでうまく働きます。これは再帰的な変数の驚くべき効果の1つです。それは非常に有益であるとともに混乱の元でもあります。このmakefileがうまく働くのは、コマンドスクリプトとマクロ本体の展開が実際に使われるまで行われないためです。そのためmakefile内での順序は実行に何ら影響を及ぼさないのです。

makefileを読んだ後、処理の第2段階でmakeは依存関係の分析を行いターゲットを特定し、各ルールを実行します。ここでのターゲットは$(OUTPUT_DIR)/very_big_fileであり必須項目を持たないので、単に処理を実行します（ファイルが存在していないと想定しています）。コマンドは$(free-space)なので、makeはプログラマが次のように書いたかのごとく展開します。

```
/tmp/very_big_file:
        /usr/bin/printf "Free disk space "
        /usr/bin/df . | /usr/bin/awk 'NR == 2 { print $$4 }'
```

すべての変数が展開されると、そのコマンドを実行します。

順序が意味を持つ2つの部分について見てみましょう。以前説明したように、ターゲットである$(OUTPUT_DIR)/very_big_fileは直ちに展開されます。もしOUTPUT_DIRの定義がルールの後に置かれていた場合、ターゲットは/very_big_fileと展開されることになるでしょう。これは望んだ動作ではありません。同様にBINの定義がAWKの後で行われていた場合、:=による代入は右辺の評価も直ちに行われるため、3つの変数はそれぞれ/printf、/df、/awkと展開されることになるでしょう。しかしこの場合:=ではなく=を使って再帰変数として定義した場合にはこの問題を回避することができます。

最後に細かいことに触れます。OUTPUT_DIRとBINの定義を再帰変数に変えても、順序の問題は変わらないでしょう。$(OUTPUT_DIR)/very_big_fileやPRINTF、DF、AWK定義の右辺がいつ展開されるかが重要な問題なのですが、それらは直ちに展開されます。なので、参照している変数が先に定義されていなければならないのです。

3.5 ターゲットとパターンに固有の変数

makefile実行中の変数は通常1つの値を持ち続けます。これはmakefileの処理が2段階で行われることに由来します。第1段階ではmakefileが読み込まれ、変数の設定と展開が行われ、そして依存関係グラフが構成されます。第2段階では依存関係グラフが分析されトラバースされます。そのためコマンドスクリプトが実行される段階ではすべての変数処理は完了しているのです。しかし、1つのルールやパターンのためだけに変数の値を再設定したい場合があったらどうでしょう。

特定のファイルだけに追加のコマンドラインオプション-DUSE_NEW_MALLOC=1が必要だという例で考えてみましょう。

```
gui.o: gui.h
        $(COMPILE.c) -DUSE_NEW_MALLOC=1 $(OUTPUT_OPTION) $<
```

コンパイルコマンドスクリプトを別に持ち、そこで必要なオプションを追加することにより問題を解決しました。このアプローチはいくつかの観点から不十分なものだといえます。まず、コードを二重に持つことになります。もしこのルールが変更されたり、組み込みのルールをカスタムルールで置き換えた場合には、このコードも修正する必要がありますが、きっと忘れてしまうでしょう。次に、もし多くのファイルが特別の扱いを必要としたなら、コードの貼り付けを行わなければなりませんが、これはとても退屈で間違いを起こしやすい作業です（こういったファイルが100個あった場合を想像してください）。

この問題に対処するために、makeはターゲット固有の変数を提供しています。これはターゲットに付随した変数定義で、ターゲットとその必須項目を処理している間だけ有効になります。前出の例はこの機能を使って次のように書き換えることができます。

```
gui.o: CPPFLAGS += -DUSE_NEW_MALLOC=1
gui.o: gui.h
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

CPPFLAGS変数はデフォルトのCコンパイルルールに含まれ、Cプリプロセッサのオプションを保持するために使われます。+=形式の代入を使うことにより、既定の値に新しいオプションを追加することになります。これでコンパイルコマンドを撤廃することができました。

```
gui.o: CPPFLAGS += -DUSE_NEW_MALLOC=1
gui.o: gui.h
```

gui.oターゲットが処理されている間、CPPFLAGSは元の値に-DUSE_NEW_MALLOC=1が追加され

たものになります。gui.oターゲットの処理が終了するとCPPFLAGSの値は元に戻ります。パターンルールの中で指定されることを除けばパターン固有変数も同じものです（パターンルールについては「2.4　パターンルール」を参照してください）。

ターゲット固有変数の記法を以下に示します。

```
target...: variable = value
target...: variable := value
target...: variable += value
target...: variable ?= value
```

つまり、すべての代入形式はターゲット固有変数の定義で使うことができます。この代入が行われるよりも前に変数が定義されている必要はありません。

さらにいうとターゲットの処理が開始されるまでは変数の代入が行われません。したがって代入の右辺が他のターゲット固有変数であってもかまいません。変数は必須項目を処理している間も有効です。

3.6　変数はどこからくるのか

これまで説明に使ってきた変数のほとんどはmakefileの中で明示的に定義されたものでしたが、変数はいろいろなところから与えることができます。例えば、makeのコマンドラインで変数が定義できることはすでに見ています。実際のところmakeの変数は以下のものから与えることができます。

ファイル

当然のことながら、変数はmakefileやインクルードされたファイルの中で定義することができます（include命令については「3.7　条件判断とinclude命令」で説明します）。

コマンドライン

makeのコマンドライン上で直接、変数の定義または再定義を行うことができます。

```
$ make CFLAGS=-g CPPFLAGS='-DBSD -DDEBUG'
```

=を含むコマンドライン引数は、変数の代入です。コマンドライン上の代入はそれぞれシェルに対しては1つの引数でなければなりません。もし変数の値（もしくは、許されないことですが変数名自身）が空白を含んでいた場合、引数は引用符で囲むか空白をエスケープする必要があります。コマンドライン上で行われた代入は、makefile内で行われる代入と環境変数を元とする値を上書きします。コマンドライン上の代入は:=か=を使うことで単純変数としても再起変数としても定義することができます。override命令を使うことで、コマンドライン上で代入が行われても、makefile中の代入を優先させることができます。

```
# ビッグエンディアンのオブジェクトを使わないとプログラムがクラッシュする！
override LDFLAGS = -EB
```

環境

環境変数はすべてmakeの開始時にmake変数として定義されます。この変数の優先度は最も低いので、makefile内やコマンドラインで行われる代入により環境変数から来ている値は上書きされてしまいます。コマンドラインオプション--environment-overrides（-e）を使うことで、環境変数の値を優先させることができます。
makeの中から別のmakeを起動した場合、親make側の変数のいくつかは子makeに対して環境変数として渡されます。デフォルトでは、もともと環境変数だった変数が子makeにも環境変数として渡されますが、export命令を使えばその他の変数も渡すことができます。

```
export CLASSPATH := $(HOME)/classes:$(PROJECT)/classes
SHELLOPTS = -x
export SHELLOPTS
```

すべての変数を環境変数として渡す（export）には次のようにします。

```
export
```

ここで留意すべきなのは、makeがシェル変数としては無効な文字を含む名前を渡してしまう可能性があるという点です。例えば以下のような場合です。

```
export valid-variable-in-make = Neat!
show-vars:
        env | grep '^valid-'
        valid_variable_in_shell=Great
        invalid-variable-in-shell=Sorry
```

```
$ make
env | grep '^valid-'
valid-variable-in-make=Neat!
valid_variable_in_shell=Great
invalid-variable-in-shell=Sorry
/bin/sh: line 1: invalid-variable-in-shell=Sorry: command not found
make: *** [show-vars] Error 127
```

このようにシェルの命令として-（マイナス記号）を含んだシェル変数を扱うことはできませんが、makeから渡すことにより可能となります。このような変数はシェルに対する普通の方法では取り出すことができず、環境に対してgrepをかけるようなトリックを使わなければなりません。それでもこの変数は子makeに対しては有効な変数として渡すことができます。「再帰的」なmakeの使い方は6章以降で扱う予定です。
子プロセスに対して環境変数を渡さないようにすることも可能です。

```
unexport DISPLAY
```

exportとunexport命令はshが備えているexportとunset機能と同じ働きをします。

環境変数と非常に相性がよいのが条件付き代入です。makefileで既定の出力ディレクトリを設定しているとしましょう。しかしmakefileの利用者がその値を簡単に書き換えられるようにしたいとも考えています。こういった状況に条件付き代入はうってつけです。

```
# 出力ディレクトリは$(PROJECT_DIR)/outとする
OUTPUT_DIR ?= $(PROJECT_DIR)/out
```

これによりOUTPUT_DIRに値が設定されていなかった場合に代入が行われます。多少冗長ですが下記のようにしても、ほぼ同じ結果を得ることができます。

```
ifndef OUTPUT_DIR
  # 出力ディレクトリは$(PROJECT_DIR)/outとする
  OUTPUT_DIR = $(PROJECT_DIR)/out
endif
```

両者の違いは、条件付き代入では変数に設定されているのがたとえ空の値であっても代入が行われないのに対し、ifdefやifndefの場合は空ではない値かどうかがチェックされます。そのためOUTPUT_DIR =では条件付き代入に対しては値が設定されていると見なされますが、ifdefの場合には値が設定されていないと判断されます。

環境変数を必要以上に使うことでmakefileのポータビリティが損なわれてしまうことに注意する必要があります。なぜなら他のユーザが同じような環境変数を設定しているとはかぎらないからです。事実筆者は上記の理由によりこの機能を滅多に使いません。

自動変数

makeはルールのコマンドスクリプトを実行する直前に自動変数を設定します。

もともと環境変数は開発用コンピュータ間の差異を管理しやすくするために使われました。例えばmakefile内で参照する環境変数を元にして開発環境（ソースコードやコンパイル済みファイルのディレクトリツリーやツールなど）を構築するのが普通です。makefileでは各ディレクトリツリーのルートを環境変数として参照するようになっています。ソースファイル用のディレクトリツリーをPROJECT_SRCで、同様にバイナリの出力をPROJECT_BINで、ライブラリをPROJECT_LIBで参照していれば、各開発者はそれらのディレクトリをどこでも好きな場所に作ることができます。

この手法の（そして一般的に環境変数を利用した際の）潜在的な問題は、これらの元となる変数が設定されていなかった場合に顕在化します。解決策の1つは、?=形式の代入を利用してデフォルトの値を用意しておくというものです。

```
PROJECT_SRC ?= /dev/$(USER)/src
PROJECT_BIN ?= $(patsubst %/src,%/bin,$(PROJECT_SRC))
PROJECT_LIB ?= /net/server/project/lib
```

プロジェクトの各要素を指し示すための変数を使うことにより、さまざまなディレクトリ配置を持ったコンピュータに適合できる開発環境を作ることができます（6章以降で、より包括的な例を紹

介する予定です）。一方で環境変数に依存しすぎないようにも注意する必要があります。一般的にmakefileは開発者の環境からの情報が最小限であっても動作する必要があります。そのため妥当なデフォルト値を用意し、重要なものについては値の存在を確認しなければなりません。

3.7　条件判断とinclude命令

条件判断命令を使うことにより、makefileを読み込む際にmakefileの一部を省略したり選択することができます。選択を行うための条件指定には「設定されているか」とか「～と等しいか」などいくつかの形式があります。

```
# COMSPECはWindowsにおいてのみ設定されている
ifdef COMSPEC
  PATH_SEP := ;
  EXE_EXT := .exe
else
  PATH_SEP := :
  EXE_EXT :=
endif
```

この場合、COMSPEC変数が設定されている場合に最初の分岐が選択されます。条件判断命令の基本的な文法は以下のとおりです。

```
if-condition
  条件が真の場合に有効となるテキスト
endif
```

または、以下のようになります。

```
if-condition
  条件が真の場合に有効となるテキスト
else
  条件が偽の場合に有効となるテキスト
endif
```

if条件の形式は次のどれかになります。

```
ifdef variable-name
ifndef variable-name
ifeq test
ifneq test
```

ifdef/ifndefに対して、変数名（*variable-name*）は$(　)で囲む必要がありません。ifeq/ifneqの*test*部は以下の形式のどちらかです。

```
"a" "b"
(a,b)
```

単引用符と二重引用符はどちらでも使うことができます（ただし前後で一致させなければなりません）。

条件判断命令はmakefileのトップレベルだけでなくマクロ定義の中やコマンドスクリプトでも使用できます。

```
libGui.a: $(gui_objects)
        $(AR) $(ARFLAGS) $@ $<
    ifdef RANLIB
        $(RANLIB) $@
    endif
```

条件の字下げは好まれますが、注意を怠るとエラーの原因となります。上の例では条件判断命令は4つの空白で字下げされますが、命令で囲まれるコマンドではタブを先頭に置きます。もしこのコマンドがタブで始まっていなければmakeがコマンドであると認識しません。もし条件判断命令がタブで始まっていれば、それらはコマンドであると誤認されサブシェルに渡されてしまいます。

ifeqとifneqは引数が等しいか等しくないかをチェックします。条件内の空白は微妙な扱いを受けます。例えば、条件の指定に括弧を使った場合、カンマの後ろの空白は無視されますが、その他の空白は意味を持ちます。

```
ifeq (a, a)
  # 等しい
endif

ifeq ( b, b )
  # 等しくない - ' b' != 'b '
endif
```

個人的には、引用符を使う形式を使い続けています。

```
ifeq "a" "a"
    # 等しい
endif

ifeq 'b' 'b'
    # これも等しい
endif
```

このような形式を利用したとしても、予期しない空白が変数展開の過程で紛れ込むのはよくあることです。比較はすべての文字に対して行われるため、問題となることがあります。strip関数を使うことでmakefileをもっと頑強にできます。

```
ifeq "$(strip $(OPTIONS))" "-d"
  COMPILATION_FLAGS += -DDEBUG
endif
```

3.7.1 include命令

「2.7 自動的な依存関係の生成」でinclude命令を扱いましたが、ここではもう少し詳しく見ることにしましょう。

makefileは他のファイルをインクルードすることができます。この機能は共通の定義をmakeヘッダファイルに置いたり、自動的に生成された依存関係情報を取り込むのに使われます。include命令は次のように使います。

```
include definitions.mk
```

ファイルはいくつでも指定できる上にワイルドカードやmakeの変数を使うこともできます。

3.7.2 includeと依存関係

makeはinclude命令に出会うとワイルドカードと変数を展開してからインクルードファイルのファイル読み込みを試みます。ファイルが存在していれば、そのまま処理は続行しますが、ファイルが存在していない場合、makeはエラーを報告しmakefileの続きを読みます。すべて読み終わると、インクルードファイルを更新するためのルールがないかルールデータベースを探します。見つかった場合は通常のターゲットの更新として処理します。ルールによりインクルードファイルが更新されると、makeはいったん内部データベースをクリアした後、makefileをもう一度最初から読み直します。それらをすべて行ってもなおファイルが存在しないためにinclude命令が失敗する場合には、エラーとしてmakeは実行を終了します。

この過程はファイルを2つ使った次の例で見ることができます。簡単なメッセージを表示するためにwarning組み込み関数を使っています（これを含めて関数は4章で扱います）。makefileは次のようなものです。

```
# 生成したファイルをインクルードする簡単なmakefile
include foo.mk
$(warning Finished include)
foo.mk: bar.mk
        m4 --define=FILENAME=$@ bar.mk > $@
```

そして次がインクルードファイルの元となるbar.mkです。

```
# bar.mk - 読まれた際に報告します
$(warning Reading FILENAME)
```

これを実行すると、次のようになります。

```
$ make
Makefile:2: foo.mk: No such file or directory
Makefile:3: Finished include
m4 --define=FILENAME=foo.mk bar.mk > foo.mk
foo.mk:2: Reading foo.mk
Makefile:3: Finished include
```

```
make: `foo.mk' is up to date.
```

最初の行ではインクルードファイルが見つからなかったことを報告していますが、2行目ではmakefileの残りを読み続けていることがわかります。読み終わるとインクルードファイルであるfoo.mkを作成するルールを発見し実行します。その後最初からやり直し、今度はインクルードファイルが読み込めます。

makeはmakefile自体をもターゲットとして扱うことがあるということを明らかにするにはよいタイミングでしょう。makefile全体を読み込んだ後、makeは現在処理しているmakefileを作り直すルールを探します。それが見つかるとそのルールを実行しmakefileが更新されたかをチェックします。更新されていれば内部の状態をクリアしてmakefileを読み直し全体の解析をもう一度実行します。このふるまいを使って無限に繰り返すあまりおもしろくない例を次に示します。

```
.PHONY: dummy
makefile: dummy
        touch $@
```

このmakefileを実行するとmakefileは最新ではないと判断されるので（なぜなら、擬似ターゲットdummyが最新ではないからです）、makefileのタイムスタンプを更新するtouchコマンドが実行されます。そしてmakeはmakefileを再度読み直し、makefileが最新ではないことを再度発見し……、ということです。

makeはインクルードファイルをどこから探し出すのでしょうか。include命令の引数が絶対パスで指定されていたなら、そのファイルを読みます。もし相対的に指定されていた場合には、カレントディレクトリを探します。ファイルが見つけれらなければ、コマンドラインの--include-dir（-I）オプションで指定されているディレクトリを探しに行きます。その後、makeがコンパイルされたときに組み込まれた/usr/local/include、/usr/gnu/include、/usr/includeといった場所を探します。これらはmakeがどのようにコンパイルされたかで多少の違いがあります。

インクルードファイルを発見できずルールを使っても作成できなかった場合、makeはエラーを報告して実行を終了します。インクルードファイルが読み込めなくてもエラーとしたくない場合には、include命令の前にマイナス記号（-）を付けます。

```
-include i-may-not-exist.mk
```

他のmakeとの互換性維持のため、-includeの別名としてsincludeを使うことができます。

include命令を最初のターゲットの前に置くことで、デフォルトターゲットが変更されてしまうことは注目に値します。つまりこの場合インクルードファイルが何らかのターゲットを含んでいた場合、その中で最初のターゲットがmakefileのデフォルトターゲットになってしまいます。これはデフォルトターゲットにしたいものを（必須項目なしでもかまわないので）単にincludeの前に置くことで防ぐことができます†。

† 訳注：このテクニックは、例6-2の中でも使われています。

```
# allをデフォルトターゲットにする
all:

include support.mk

# これで変数が定義されたので、allターゲットを完成させる
all: $(programs)
```

3.8　標準的なmake変数

自動変数に加え、組み込みルールをカスタマイズするための変数と同じように、内部のこまごまとした状態を明らかにするための変数をmakeは保持しています。

MAKE_VERSION

GNU makeのバージョン番号です。本書の執筆時点での値は3.80で、CVSのリポジトリ内での値は3.81rc1でした。

前のバージョンである 3.79.1では（その他の差異に加えて）、今では非常によく使われているeval関数とvalue関数がサポートされていません。そこでこれらの関数を使う必要がある際には、この変数を使って現在動作しているmakeのバージョンをチェックすることにしています。この例は「4.2.4　実行制御」で目にすることができるでしょう。

CURDIR

この変数は現在実行しているmakeプロセスのカレント作業ディレクトリ(cwd)の値を保持します。--directoryオプション（-Cオプション）が使われていなければ、この値はmakeを起動したディレクトリ（そしてシェル変数PWDの値）とおそらく同じです。--directoryオプションはmakefileを探す前にディレクトリを変更するよう指示するためのものです。このオプションの使い方は、--directory=*directory-name*または-C *directory-name*です。--directoryが使われると、CURDIRの値は--directoryの引数と同じになります。

筆者はコーディング中、makeをemacsから起動します。例えば現在の筆者のプロジェクトではJavaを使っていますが、トップレベルのディレクトリ（コードのあるディレクトリである必要はありません）にmakefileが1つ置いてあります。この場合、--directoryを使うことでソースツリーのどこからmakeを起動してもmakefileを見つけることが可能となります。makefileの中ではすべてのパス名がmakefileのディレクトリからの相対パスで記述されています。場合によっては絶対パスが必要となることがありますが、その際にはCURDIRと合わせて使うことになります。

MAKEFILE_LIST

この変数にはデフォルトのmakefileやコマンドラインで指定されたもの、そしてinclude命令で読み込まれたものも含めてmakeが読んだファイルのリストが収められます。それぞれのファイルを読む前に、そのファイル名がMAKEFILE_LISTに追加されます。したがってこのリストの最後の単語を調べれば現在処理中であるmakefileのファイル名がわかります。

MAKECMDGOALS

MAKECMDGOALS変数には、現在実行しているmakeのコマンドラインで指定されたすべてのターゲットが入っています。そこにはコマンドラインオプションやコマンドラインで行われた変数定義は含まれません。例えば以下のようになります。

```
$ make -f- FOO=bar -k goal <<< 'goal:;# $(MAKECMDGOALS)'
# goal
```

この例では-f-オプション（--fileオプション）を使ってmakefileを標準入力から読み込むという仕掛けを使っています。標準入力は“<<<”を用いてbashの機能であるヒアストリング（ヒアドキュメントともいいます）を使ってコマンド上の文字列にリダイレクトされます。そのmakefileはというと、デフォルトターゲットであるgoalを持ち、コマンドスクリプトはセミコロンで区切られてターゲットと同じ行で指定されています。コマンドスクリプトは次の1行だけです†。

```
# $(MAKECMDGOALS)
```

MAKECMDGOALSはターゲットが特殊な扱いを必要とするときによく使われます。そのよい例がcleanターゲットです。cleanを実行する際、include命令で取り込まれる依存関係ファイルを作成（「2.7　自動的な依存関係の生成」で取り上げました）する必要はありません。そこでifneqとMAKECMDGOALSを使って作成が行われないようにします。

```
ifneq "$(MAKECMDGOALS)" "clean"
    -include $(subst .xml,.d,$(xml_src))
endif
```

.VARIABLES

この変数には、読み込んだ時点までにmakefileで定義されているすべての変数名のリストが入っています。ただしターゲット固有変数は含まれません。この変数は読み取り専用なので、この変数への代入は無視されます。

```
list:
        @echo "$(.VARIABLES)" | tr ' ' '\015' | grep MAKEF
$ make
MAKEFLAGS
MAKEFILE_LIST
MAKEFILES
```

これまで見てきたように、変数はmakeの組み込みルールをカスタマイズするためにも使われます。C/C++用のルールが典型ですが、こういった変数はその他のプログラミング言語用にも用意されてい

† この例をbash以外のシェル上で実行する場合には、次のようにします。

```
$ echo 'goal:;# $(MAKECMDGOALS)' | make -f- FOO=bar -k goal
```

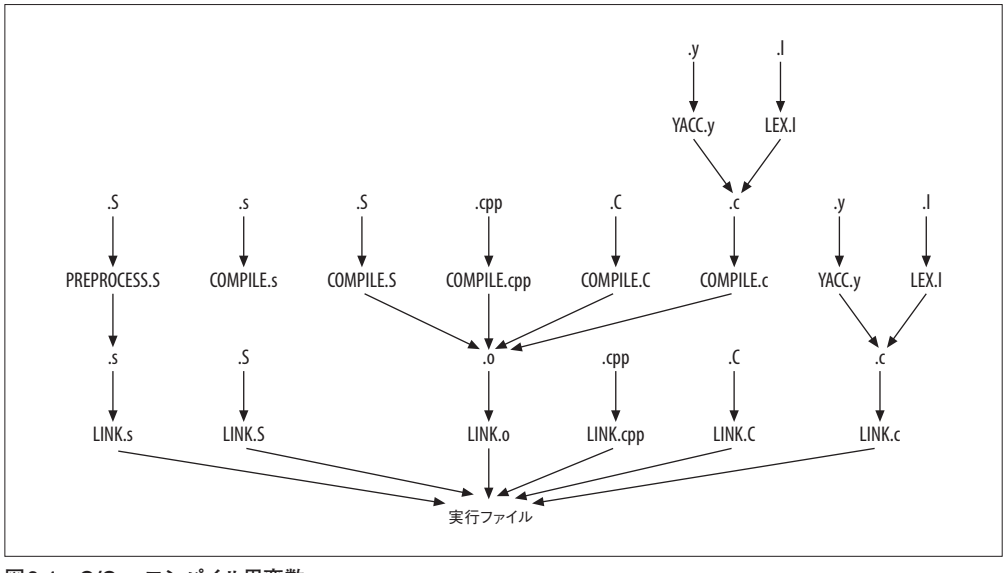

図3-1　C/C++コンパイル用変数

ます。図3-1ではファイルタイプの変換を制御する変数を示しています。

変数は*ACTION.suffix*という標準的な構造を持っています。*ACTION*部がCOMPILEのものはオブジェクトファイルを作成し、LINKのものは実行ファイルを作成します。その他特殊なものとしてPREPROCESS、YACC、LEXがそれぞれCプリプロセッサ、yacc、lexを表します。そして*suffix*は元となるファイルタイプを表しています。

例えばC++では、2つのルールを通ってこれらの処理が行われます。最初はC++のソースファイルをコンパイルしてオブジェクトファイルにするルールで、次がオブジェクトファイルをリンクして実行ファイルにするルールです。

```
%.o: %.C
        $(COMPILE.C) $(OUTPUT_OPTION) $<

%: %.o
        $(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

最初のルールでは以下の変数定義を使っています。

```
COMPILE.C = $(COMPILE.cc)
COMPILE.cc = $(CXX) $(CXXFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
CXX = g++
OUTPUT_OPTION = -o $@
```

GNU makeではC++ソースを表すサフィックスとして.cと.ccをサポートしています。CXX変数はC++コンパイラを表し、デフォルトの値はg++です。CXXFLAGS変数、CPPFLAGSTARGET_ARCH変

数にデフォルトの値はありません。これらの変数は構築処理をカスタマイズするために利用者が変更することを意図していて、それぞれC++コンパイラのフラグ、Cプリプロセッサのフラグそしてアーキテクチャ特有のコンパイルオプションを指定するために使います。OUTPUT_OPTIONは出力ファイル用のオプションを格納します。

リンクのルールはもっと簡単です。

```
LINK.o = $(CC) $(LDFLAGS) $(TARGET_ARCH)
CC = gcc
```

このルールではオブジェクトファイルを結合して実行ファイルにするのにCコンパイラを使います。デフォルトのCコンパイラはgccです。LDFLAGSとTARGET_ARCHにはデフォルトの値がありません。LDFLAGSには例えば-Lのようなリンク用のオプションを格納します。LOADLIBESとLDLIBSはリンクするライブラリのリストを保持します。2つ用意されているのは主に移植性を高めるためです。

以上でmakeの変数を巡る簡単なツアーを終了します。変数はもっと奥が深いのですが、どのように変数がルールと一体化しているかを伝えることはできたかと思います。TeX向けには別の変数群とひとそろいのルールが用意されています。再帰的makeは変数を活用する機能の1つですが、これについては6章で扱います。

4章 関数

GNU makeでは組み込みの関数とともにユーザ定義の関数もサポートしています。関数の呼び出しは変数参照のように見えますが、カンマで区切られた1つかそれ以上の引数を持ちます。ほとんどの組み込み関数は何らかの値に展開され、他の変数やサブシェルに渡されます。ユーザ定義関数は変数かマクロに格納されます。

4.1 ユーザ定義関数

コマンド列を変数に格納することで活用の幅が大きく広がります。例えば以下はプロセスを終了させる小さくて気の利いたマクロです†。

```
AWK := awk
KILL := kill

# $(kill-acroread)
define kill-acroread
  @ ps -W |                                        \
  $(AWK) 'BEGIN      { FIELDWIDTHS = "9 47 100" }  \
          /AcroRd32/ {                             \
                       print "Killing " $$3;       \
                       system( "$(KILL) -f " $$1 ) \
                     }'
endef
```

† 「なぜmakefileでこんなことをする必要があるのだろう」と感じるのはもっともです。Windowsではファイルを開くと他のプロセスからの書き込みを防ぐためにファイルをロックします。本書の執筆中、Acrobat ReaderがPDFファイルをロックして、makefileからPDFを更新できなくなることがよく起きました。そこでいくつかのターゲットに対してこのコマンドを追加し、ロックされたファイルを更新する前にAcrobat Readerを終了するようにしたのです。

（このマクロはCygwinツール†を使うように書いてあるので、探そうとしているプログラムの名前やpsコマンドやkillコマンドに対するオプションは標準的なUnixのものとは異なっています。）プロセスを終了させるために、psコマンドの出力をパイプでawkに渡しています。awkスクリプトではWindows版Acrobat Readerのプログラム名で検索を行い、動作していれば終了させます。プログラム名とその引数を1つのフィールドとして取り出すためにFIELDWIDTHSを使っています。これにより空白の有無にかかわらず、完全なプログラム名と引数を取り出すことができます。awkのフィールドは$1、$2のようにして参照しますが、そのまま使うとmakeの変数として扱われてしまいます。$nに対してドル記号を付加し$$nとすることで、$nを展開せずにawkに渡すことができます。makeは2つのドル記号を見つけると、1つのドル記号にしてからサブシェルに渡します。

よくできたマクロといえるでしょう。define命令を使っているので、何回使ってもコードが重複することがありません。しかし、完璧ではないのです。Acrobat Reader以外のプロセスを終了させる場合にはどうすればよいのでしょう。別のスクリプトをコピーして別のマクロを定義しなければならないのでしょうか。そんなことはしたくありません。

それぞれの展開結果を変えられるように、変数とマクロには引数を渡すことができます。マクロへの引数は、マクロ本体では$1、$2のように取り出すことができます。検索引数だけを加えることにより、kill-acroread機能をパラメータ化することができます。

```
AWK := awk
KILL := kill
KILL_FLAGS := -f
PS := ps
PS_FLAGS := -W
PS_FIELDS := "9 47 100"

# $(call kill-program,awk-pattern)
define kill-program
  @ $(PS) $(PS_FLAGS) |                                    \
  $(AWK) 'BEGIN { FIELDWIDTHS = $(PS_FIELDS) }             \
          /$1/ {                                           \
                  print "Killing " $$3;                    \
                  system( "$(KILL) $(KILL_FLAGS) " $$1 ) \
                }'
endef
```

awkの検索パターンであるAcroRd32を、引数参照の$1に置き換えました。マクロの引数である$1とawkのフィールド参照である$$1の微妙な違いに注意が必要です。変数を受け取るプログラムが何であるのかを意識するのは非常に重要です。この機能の改良作業では、引数に適切な名前を付け、加えてマクロに直接書き込まれていたCygwin特有の値を変数に移しました。これでプロセスを終了させるための使いやすいマクロになりました。

† CygwinツールはGNUとLinuxの標準的なプログラムの多くをWindowsに移植したツールです。それにはコンパイラ群、X11R6、sshに加えinetdまでもが含まれます。Windowsへの移植はUnixのシステムコールをWin32 APIで実装した互換ライブラリにより実現されています。これは技術上のすばらしい偉業であり、ぜひ使ってみることを勧めます。http://www.cygwin.comから入手できます。

では、実際にこれが使われる場面を見ましょう†。

```
FOP := org.apache.fop.apps.Fop
FOP_FLAGS := -q
FOP_OUTPUT := > /dev/null
%.pdf: %.fo
        $(call kill-program,AcroRd32)
        $(JAVA) $(FOP) $(FOP_FLAGS) $< $@ $(FOP_OUTPUT)
```

これはAcrobatプロセスが動作していれば終了させ、fo（Formatting Objects）ファイルをFopプロセッサ（http://xml.apache.org/fop）によりpdfファイルにするパターンルールです。以下は変数またはマクロを展開するための文法です。

```
$(call macro-name[, param1...])
```

callはmakeの組み込み関数で、最初の引数を展開し残りの引数を$1、$2に置き換えます（実際には実行制御を明け渡すという意味での「呼び出し（call）」を行うわけではなく、特殊な種類のマクロ展開を行うだけです）。macro-nameはマクロの名前でも変数の名前でもかまいません（マクロは単に改行を組み込める点を除いて変数と同じであったことを思い出しましょう）。マクロまたは変数の値に$*n*の参照がなくてもかまいませんが、それではcallを使う意味がなくなってしまいます。マクロ名に続く引数はカンマで区切られます。

callに対する最初の引数は変数名のみ（つまりドル記号で始まっていない）であることに注意が必要です。これはかなり珍しいことです。変数名のみを受け付ける組み込みの関数は、callとoriginだけです。もし最初の引数にドル記号を付けて括弧で囲むと、変数が展開され、その値がcallに渡されることになります。

callは必要最小限の引数チェックしか行いません。callにはいくつでも引数を付けることができます。マクロ内で$*n*を参照していてもcall側で対応する引数を与えなかった場合、その変数は空の文字列に置き換えられます。もしcallに付随する引数がマクロ内で参照される$*n*よりも多かった場合、余分な引数は使われません。

マクロの中からマクロを呼び出した場合、make 3.80では多少奇妙なふるまいをすることに注意する必要があります。call関数はマクロを展開する間に引数を普通のmake変数として設定します。そのためマクロから他のマクロを呼び出すと、呼び出されたマクロから呼び出し側の引数が見える可能性があります。

```
define parent
  echo "parent has two parameters: $1, $2"
  $(call child,$1)
endef

define child
  echo "child has one parameter: $1"
```

† 訳注：これは「11.1 Book makefile」で紹介するmakefileの一部です。

```
  echo "but child can also see parent's second parameter: $2!"
endef

scoping_issue:
        @$(call parent,one,two)
```

実行すると、マクロの実装にスコープ上の問題があることがわかります。

```
$ make
parent has two parameters: one, two
child has one parameter: one
but child can also see parent's second parameter: two!
```

この問題は3.81では修正され、呼び出されたマクロでは$2が展開されなくなっています。

ユーザ定義関数についてはこれからもあらゆるところで取り上げますが、おもしろい部分に進む前にもう少し基礎的なことを固める必要があります。

4.2 組み込み関数

単に値を格納するため以上の目的でmakeの変数を使い始めると、変数とその値をもっと複雑な方法で操作したくなるはずです。GNU makeには変数とその値を操作するための組み込み関数が多数用意されています。文字列操作、ファイル名操作、実行制御、ユーザ定義関数、その他の関数など大きくいくつかに分類できます。

でもその詳細を解説する前に、関数の文法についてもう少し学んでおきましょう。関数はすべて次の形式を持ちます。

```
$(function-name arg1[, argn])
```

$(の後に組み込み関数名が置かれ、関数への引数が続きます。最初の引数の前にある空白は取り除かれますが、その他の引数では前についている（もちろん途中や後ろにあるものも含めて）空白は残ります。

関数引数はカンマで区切られます。多くの関数が、1つの引数として単語を空白で区切ったリストを受け取ります。そういった関数では、単語間の空白は単語区切りと見なされますが、それ以外は無視されます。

多くの関数ではパターンを引数として受け付けます。このパターンの文法はパターンルールで使うものと同じです（「2.4 パターンルール」を参照してください）。パターンは1つの%と前置または後置（もしくは両方）の文字列で構成されます。%文字は0個以上のあらゆる文字列を表します。パターンが目的の文字列と一致していると見なされるには、どこか一部が合っているだけではなくすべてと一致している必要があります。具体例についてはfilter関数のところで触れます。パターンの中で%文字の使用は任意であり、必要がなければ使わなくてもかまいません。

4.2.1 文字列関数

ほとんどのmake組み込み関数はテキストを操作して、ある形式を別の形式に変更するものですが、ここで説明するいくつかの関数は特に文字列の操作に特化しています。

makeにおいてよく使われる文字列操作の1つに、リストの中から一連のファイルを選択するというものがあります。シェルスクリプトではgrepが使われますが、makeではfilter、filter-out、findstring関数が用意されています。

`$(filter` *`pattern...,text`*`)`

filter関数は*text*を空白で区切られた単語の列と見なし、*pattern*と一致したものを返します。例えばユーザインタフェース用コードのライブラリを作る際、uiサブディレクトリに存在しているオブジェクトファイルだけを一覧したいとしましょう。下の例ではファイル名のリストからui/で始まり.oで終わるものだけを取り出しています。%文字はその間にあるあらゆる文字列と一致します。

```
$(ui_library): $(filter ui/%.o,$(objects))
        $(AR) $(ARFLAGS) $@ $^
```

filter関数は空白で区切った複数のパターンを受け付けます。前述したようにパターンと完全に一致した単語が出力されます。例で示しましょう。

```
words := he the hen other the%
get-the:
        @echo he matches: $(filter he, $(words))
        @echo %he matches: $(filter %he, $(words))
        @echo he% matches: $(filter he%, $(words))
        @echo %he% matches: $(filter %he%, $(words))
```

makeを実行すると次の出力を得ます。

```
$ make
he matches: he
%he matches: he the
he% matches: he hen
%he% matches: the%
```

パターンは単語の一部ではなくすべてと合っていなければ一致したと見なされないため、最初のパターンはheとしか一致しません†。その他のパターンはheに加えheを適切な場所に含んでいる単語とも一致しています。

パターンには%が1つしか使えません。もしパターンに複数の%が使われていた場合には、最初の1つを除いてすべて%そのものとして扱われます。

† 訳注：この機能では単語と一致しているかが問題となり、grep的な動作とは異なっています。grepの場合、theは“he”を部分的に含んでいるので一致していると認識しますが、filter関数では単語の一致を見るため、theとheは一致しないことになります。

filter関数が単語の一部分との一致を許容しなかったり、複数のワイルドカード文字を受け付けなかったりすることを奇妙に思っていることでしょう。そのような機能が欠けていることを残念に思う人もいるでしょう。しかし、ループや条件判断を使って同様の機能を実現することはできます。それはもう少し後で示すことにしましょう†。

`$(filter-out` *`pattern...,text`*`)`

filter-out関数はfilterと逆の働き、つまりパターンに一致しなかった単語を取り出します。以下に示すのはCのヘッダファイル以外を選択する例です。

```
all_source := count_words.c counter.c lexer.l counter.h lexer.h
to_compile := $(filter-out %.h, $(all_source))
```

`$(findstring` *`string,text`*`)`

この関数は*text*中の*string*を検索します。もし見つかればその*string*自身を返し、見つからなければ何も返しません。

まず、これはfilter関数が本来そうであるべきgrep的な部分文字列検索を行う関数のように思えますが、実はそうではありません。まず最も重要な点として、この関数は見つかった単語ではなく検索文字列のほうを返します。そして検索文字列にはワイルドカードを指定することができません（つまり、%は%文字として扱われます）。

この関数は後で紹介するif関数と併せて使うのが便利なのですが、この関数単独で役に立つ状況が1つあります。

参照用のソースとか実験のためのソース、デバッグ用バイナリ、最適化バイナリなど複数のディレクトリツリーが並列に存在しているとしましょう。（ディレクトリツリーのルートからの相対パスではなく）カレントディレクトリが、現在どのディレクトリツリーの中なのか調べたいとします。次に示すのはこれを行うためのコード概要です。

```
find-tree:
        # PWD = $(PWD)
        # $(findstring /test/book/admin,$(PWD))
        # $(findstring /test/book/bin,$(PWD))
        # $(findstring /test/book/dblite_0.5,$(PWD))
        # $(findstring /test/book/examples,$(PWD))
        # $(findstring /test/book/out,$(PWD))
        # $(findstring /test/book/text,$(PWD))
```

（各行はタブとシェルのコメント文字で始まっているので、それぞれ別のサブシェルで「実行」されます。Bourne Again Shellつまりbashや他のBourne shellに似たシェルでは単にこの行を無視します。これはmakeが組み立てたものを出力するための@echoを使うより簡単な手法です。他のシェルでも使える演算子:を使っても、ほぼ同じことができますが、:演算子はリダイレクトを伴います。コマンド行には> *word*が含まれることになるので、*word*というファイルができるという副作用が生じます。）これを実行すると、次のように出力されます。

† 訳注：foreach関数の説明の中で、ユーザ定義関数grep-stringとして紹介されています。

```
$ make
# PWD = /test/book/out/ch03-findstring-1
#
#
#
#
# /test/book/out
#
```

ご覧のように、$(PWD)に対する各テストはカレントディレクトリの親ディレクトリに対するチェックを行うまでは何も返されていません。そしてカレントディレクトリの親ディレクトリだけが返されています。このコードは単にfindstring関数をデモしているにすぎませんが、カレントディレクトリが属しているディレクトリツリーを出力する関数を作る際に役立ちます。

検索と置換の関数は2つ存在します。

$(subst *search-string*,*replace-string*,*text*)

この関数は単純な検索と置換を行うもので、ワイルドカードも使えません。ファイル名のサフィックスを別のものに付け替える際によく使われます。

```
sources := count_words.c counter.c lexer.c
objects := $(subst .c,.o,$(sources))
```

$(sources)の中にあるすべての“.c”を“.o”に置換します。もっと一般的にいうと、すべての検索文字列（*search-string*）を置換文字列（*replace-string*）に置き換えます。この例は関数引数中の空白が持つ影響を説明する際によく使われます。ここではカンマの後に空白が入っていない点に注目してください。もし空白を入れてしまうと、

```
sources := count_words.c counter.c lexer.c
objects := $(subst .c, .o, $(sources))
```

（カンマの後の空白に注意）結果は次のようになります。

```
count_words .o counter .o lexer .o
```

これは望んだ結果ではありません。これは引数.oとカンマ間にある空白が置換文字列の一部として扱われ出力に現れた結果です。最初の引数の前に置かれた空白はmakeにより取り除かれるので、.cの前の空白は問題ありません。コマンド行の一部など先頭に空白が含まれていてもかまわない状況で$(objects)が使われるかぎり、$(sources)の前の空白もおそらく無害です。関数の実行が望んだとおりの結果になるとしても、カンマの後に空白を入れたり入れなかったりを混在させるのはよい方法ではありません。

```
# うーん、この空白の使い方はややこしい
objects := $(subst .c,.o, $(source))
```

substはファイル名やサフィックスというものを認識しているわけではなく、単に文字列として扱っている点に注意が必要です。もしファイル名の内部に.cが含まれていた場合、それも置き換えられてしまうでしょう。例えばファイル名がcar.cdr.cであった場合、結果はcar.odr.oになってしまいます。おそらくこれは望んだ結果ではありません。

「2.7 自動的な依存関係の生成」では依存関係を生成する方法を取り上げ、その最後で使ったmakefileには次のようにsubstが使われていました。

```
VPATH = src include
CPPFLAGS = -I include

SOURCES = count_words.c  \
          lexer.c        \
          counter.c

count_words: counter.o lexer.o -lfl
count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h
include $(subst .c,.d,$(SOURCES))
%.d: %.c
        $(CC) -M $(CPPFLAGS) $< > $@.$$$$;                    \
        sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@;   \
        rm -f $@.$$$$
```

subst関数はソースファイルのリストを依存関係ファイルのリストに変換するために使われています。依存関係ファイルはinclude命令の引数として使われるので、更新対象として%.dルールを使って更新されます。

`$(patsubst search-pattern,replace-pattern,text)`

こちらはワイルドカードが使える検索および置換の関数です。通常、パターンには%が1つ含まれます。*replace-pattern*で使われた%文字は、ワイルドカードに適合したテキストに展開されます。*search-patten*は*text*の値全体と一致する必要がある点に注意が必要です。例えば次の例は*text*の最後にあるスラッシュを取り除くものであり、*text*中の各スラッシュを取り除くものではありません。

```
strip-trailing-slash = $(patsubst %/,%,$(directory-path))
```

置換参照は同様の置換を行う別の手法です。以下は置換参照の文法です。

`$(variable:search=replace)`

*search*が単純な文字列である場合、単語の末尾に位置している場合には*replace*と置き換わります。つまり後ろに空白があるか*variable*の最後にある場合に置き換えられます。加えて、*search*にはワイルドカードを表す%を含めることが可能です。その場合、検索と置換のルールはpatsubstと同じになります。この記法はpatsubstと比べて不明瞭で読みにくいと思います。

これまでにも見てきたように、変数には単語のリストが格納されていることがあります。次に紹介する関数はリストから単語を選択するもの、リストの長さを数えるもの、などです。他のmake関数と同様に空白で区切られたものが単語として認識されます。

`$(words text)`

この関数は*text*中の単語数を返します。

```
CURRENT_PATH := $(subst /, ,$(HOME))
words:
        @echo My HOME path has $(words $(CURRENT_PATH)) directories.
```

後述するように、この関数には多くの用途がありますが、効果的に使えるようにするために他の関数もいくつか紹介します。

`$(word n,text)`

この関数は、*text*中*n*番目の単語を返します。1は最初の単語を表します。もし*text*中の単語数よりも大きい値を指定した場合には、関数は何も返しません。

```
version_list := $(subst ., ,$(MAKE_VERSION))
minor_version := $(word 2, $(version_list))
```

MAKE_VERSIONは組み込みの変数です（「3.8　標準的なmake変数」を参照してください）。次のように使うことで、リスト中最後の単語を取り出すことができます。

```
current := $(word $(words $(MAKEFILE_LIST)), $(MAKEFILE_LIST))
```

この結果は最後に読み込まれたmakefileのファイル名になります。

`$(firstword text)`

この関数は*text*中最初の単語を返します。`$(word 1,text)`と同じ働きをします。

```
version_list := $(subst ., ,$(MAKE_VERSION))
major_version := $(firstword $(version_list))
```

`$(wordlist start,end,text)`

この関数は*text*中の*start*から*end*までを含めた単語列を返します。word関数と同様に1は最初の単語を表します。もし*start*が総単語数よりも大きかった場合、関数は何も返しません。*start*が*end*よりも大きかった場合も同様です。*end*が総単語数よりも大きかった場合には*start*以降の全単語が返されます。

```
# $(call uid_gid, user-name)
uid_gid = $(wordlist 3, 4, \
              $(subst :, , \
                $(shell grep "^$1:" /etc/passwd)))
```

4.2.2 その他の重要な関数

ファイル名を扱う関数に進む前に、2つの非常に便利な関数sortとshellを紹介しましょう。

`$(sort list)`

sort関数は引数として与えられたリストの重複を取り除き並び替えます。結果は1つずつの空白に区切られた単語が重複なく辞書順に並んだものとなります。さらにいうと、sortは引数の前と後ろの空白を取り除きます。

```
$ make -f- <<< 'x:;@echo =$(sort d b s d t )='
=b d s t=
```

当然ながらsort関数はmakeが直接実装しているため、sortプログラムの持つオプションは装備していません。通常は、変数や別のmake関数の戻り値を引数として処理するだけです。

`$(shell command)`

shell関数は1つの引数を受け付け、（他の引数と同様に）必要な展開後、サブシェルによって実行します。コマンドの標準出力は関数の戻り値として返されます。出力中の改行列はそれぞれ1つの空白に置き換えられ、最後の改行は取り除かれます。標準エラー出力とコマンドのexitステータスは返されません。

```
stdout := $(shell echo normal message)
stderr := $(shell echo error message 1>&2)
shell-value:
        # $(stdout)
        # $(stderr)
```

ご覧のとおり標準エラー出力はいつものように画面に表示され、shell関数の戻り値にはなりません。

```
$ make
error message
# normal message
#
```

ここに一連のディレクトリを作るためのループがあります。

```
REQUIRED_DIRS = ...
_MKDIRS := $(shell for d in $(REQUIRED_DIRS);  \
             do                                \
               [[ -d $$d ]] || mkdir -p $$d;   \
             done)
```

コマンドスクリプトが実行される前に、必要としている出力先ディレクトリの存在が保証されているなら、makefileは簡単になります。この変数は一連のディレクトリが確実に存在するようbashのforループを使って必要なディレクトリを作成します。二重の角括弧は単語の分割と

パス名の展開が行われないという点を除き、testコマンドと同様の働きをするbashの記法です。そのため、変数中のファイル名が空白を含んでいたとしても（引用符で囲まなくても）このチェックはうまく動作します。この変数設定をmakefileの頭に置くことで、コマンドスクリプトや他の変数がディレクトリを使用する前に、その存在を保証することができます。変数_MKDIR自体には意味がなく、使われることもありません。

shell関数はあらゆる外部コマンドを実行することができるので、使用には注意が必要です。特に単純変数と再帰変数との違いは考慮しなければなりません。

```
START_TIME := $(shell date)
CURRENT_TIME = $(shell date)
```

START_TIMEは変数が定義された時点で1回だけdateコマンドが実行されます。CURRENT_TIMEは変数がmakefile中で使われるたびにdateが実行されます。

おもしろい機能を作るための道具はそろいました。次に紹介するのは値が重複していないかをチェックするためのものです。

```
# $(call has-duplicates, word-list)
has-duplicates = $(filter       \
                   $(words $1)  \
                   $(words $(sort $1)))
```

リスト中の単語数と重複を取り除いたリストの単語数を数え、比較します。makeの関数は数値を解釈しません。文字列だけです。2つの数値を比較するには文字列として比べなければなりません。それにはfilterを使うのが簡単です。このhas-duplicates関数は重複が存在した場合にはnull以外の値を返します。

次は日付からファイル名を作り出す簡単な方法です。

```
RELEASE_TAR := mpwm-$(shell date +%F).tar.gz
```

これにより次のファイル名が作られます。

```
mpwm-2003-11-11.tar.gz
```

dateコマンドをもう少し賢く使うことで同じファイル名を作ることもできます。

```
RELEASE_TAR := $(shell date +mpwm-%F.tar.gz)
```

次の関数は相対パス（おそらくcomディレクトリからのものです）を完全修飾されたJavaのクラス名に変換します。

```
# $(call file-to-class-name, file-name)
file-to-class-name := $(subst /,.,$(patsubst %.java,%,$1))
```

このパターンは2つのsubstを使っても実現できます。

```
# $(call file-to-class-name, file-name)
file-to-class-name := $(subst /,.,$(subst .java,,$1))
```

これを使えば次のようにJavaのクラスを実行することができます。

```
CALIBRATE_ELEVATOR := com/wonka/CalibrateElevator.java
calibrate:
        $(JAVA) $(call file-to-class-name,$(CALIBRATE_ELEVATOR))
```

$(sources)中のディレクトリ名がcomの上に親ディレクトリを持っていた場合には、ディレクトリのルートを第1引数とする次の関数により取り除くことができます†。

```
# $(call file-to-class-name, root-dir, file-name)
file-to-class-name := $(subst /,.,       \
                        $(subst .java,,  \
                          $(subst $1/,,$2)))
```

このような関数を読み解く際には、たいてい下から上に読むとわかりやすくなります。最初に最下層にあるsubstが$1/を取り除き、次に.javaを取り除き、最後にスラッシュをすべてピリオドに置き換えます。

4.2.3 ファイル名関数

makefileではファイルを扱うために多くの時間を費やします。そのため、この作業を行うmake関数が多く存在していても驚くには値しません。

$(wildcard *pattern...*)

ワイルドカードは2章でターゲット、必須項目、コマンドスクリプトを扱う際に取り上げました。しかしこの機能を変数の定義など他の場所で使うにはどうすればよいでしょう。shell関数を使えばパターンを展開するためだけにサブシェルを使うことができますが、頻繁に使うには時間がかかりすぎます。そこでwildcard関数が登場します。

```
sources := $(wildcard *.c *.h)
```

wildcard関数はパターンのリストを受け取り、それぞれを展開します††。パターンがどのファイルとも一致しなかった場合には空の文字列が返されます。ワイルドカードがターゲットや必須項目として展開されるときと同様に、「~、*、?、[...]、[^...]」といったシェル展開文字を使うことができます。

---

† Javaでは開発者のInternetドメインを逆順にした名前を含むパッケージの中で、すべてのクラスを宣言することを推奨しています。またディレクトリ構造はパッケージ構造を反映したものになります。そのためソースツリー構造はroot-dir/com/company-name/dirのようになります。

†† make 3.80のマニュアルの記述には、複数のパターンが受け付け可能であることが抜けています。

wildcardは条件に合ったファイルが存在するかどうかを調べるためにも使えます。if関数（後述します）と合わせて使う際には、ワイルドカードをまったく使っていないwildcard関数を目にすることも多いでしょう。例えば以下の記述は、ユーザのホームディレクトリに.emacsファイルが存在していなかった場合には空の文字列となります。

```
dot-emacs-exists := $(wildcard ~/.emacs)
```

$(dir *list...*)

dir関数は*list*にある各単語のディレクトリ部分を返します。次の例はCファイルがある各ディレクトリを戻します。

```
source-dirs := $(sort   \
                 $(dir  \
                   $(shell find . -name '*.c')))
```

findコマンドがすべてのソースファイル名を戻した後、dir関数がそれぞれのファイル名の部分を取り除き、最後にsortがディレクトリ名の重複を取り除きます。変数が使われる度にfindが実行されないように（makefileの実行中にファイルが生成されたり削除されたりしないことを想定しています）、この変数は単純変数として定義されていることに注意しましょう。次に示すものは再帰変数を必要とする関数の例です。

```
# $(call source-dirs, dir-list)
source-dirs = $(sort   \
                $(dir  \
                  $(shell find $1 -name '*.c')))
```

この例では検索先ディレクトリを空白で区切ったリストを最初の引数として受け取ります。findはマイナス記号の付いた引数より前の引数をディレクトリのリストとして認識します（このfindの機能を何十年も知りませんでした）。

$(notdir *name...*)

notdir関数はファイルのパス名からファイル名の部分だけを取り出します。次の例はJavaのソースファイル名からクラス名を取り出します。

```
# $(call get-java-class-name, file-name)
get-java-class-name = $(notdir $(subst .java,,$1))
```

dirとnotdirの両方を使って必要としている結果を得ているサンプルは無数にあります。例えばファイルの出力先と同じディレクトリで実行しなければならない特殊なシェルスクリプトがあるとしましょう。

```
$(OUT)/myfile.out: $(SRC)/source1.in $(SRC)/source2.in
        cd $(dir $@); \
        generate-myfile $^ > $(notdir $@)
```

ターゲットを表す自動変数$@を分解して、ターゲットのディレクトリとファイルそれぞれ別の値として取り出すことができます。実際にはOUTが絶対パスであった場合、notdirを使う必要はないのですが、このように記述したほうが出力が読みやすくなります。
コマンドスクリプトでは「2.2.1 自動変数」に登場した$(@D)と$(@F)を使うことでファイル名を分解することができます。

次に紹介するのはサフィックスを付けたり取り除いたりする関数などです。

$(suffix *name...*)

suffix関数は各引数からサフィックスのリストを作って返します。次の例は、全部同じサフィックスを持っているかをチェックする関数です。

```
# $(call same-suffix, file-list)
same-suffix = $(filter 1, $(words $(sort $(suffix $1))))
```

suffix関数は、findstring関数といっしょに条件判断の中で多く使われます。

$(basename *name...*)

basenameはsuffix関数を補完する関数です。ファイル名からサフィックスを取り除いた部分を返します。パス名の部分はそのまま残されます。前述のfile-to-class-name関数とget-java-class-name関数をbasenameを使って書き直すと次のようになります。

```
# $(call file-to-class-name, root-directory, file-name)
file-to-class-name := $(subst /,., \
                        $(basename \
                          $(subst $1/,,$2)))
# $(call get-java-class-name, file-name)
get-java-class-name = $(notdir $(basename $1))
```

$(addsuffix *suffix,name...*)

addsuffix関数は*name*中の各単語に対して指定されたサフィックス文字列を加えます。サフィックス文字列は何でもかまいません。次の例はPATH中に存在するファイルから指定されたパターンに合致するファイルを見つけ出します。

```
# $(call find-program, filter-pattern)
find-program = $(filter $1,                    \
                 $(wildcard                    \
                   $(addsuffix /*,             \
                     $(sort                    \
                       $(subst :, ,            \
                         $(subst ::,:.:,       \
                           $(patsubst :%,.:%,  \
                             $(patsubst %:,%:.,$(PATH)))))))))
find:
        @echo $(words $(call find-program, %))
```

最も下層にある3つの置換はシェル文法の特殊な場合を扱っています。パスの指定が抜けているところはカレントディレクトリとして解釈されます。これを処理するために、パス指定の最後が抜けている、最初が抜けている、中間が抜けているところ†をこの順に探し出し“.”と置き換えます。次にパスの区切り文字を空白に置き換えることで、個別の単語に分割します。sort関数が重複したパスを取り除きます。そして展開のためのサフィックスである/*を各単語に加え、wildcardを使って展開します。最後にfilter関数により指定したパターンだけを取り出します。

この関数は実行に時間がかかり（多分どんなシステムでも同様でしょう）、1.9GHz P4に512MBメモリのシステムで0.20秒かかって4,335個のファイルを見つけました。これは$1引数をwildcard関数の内側に移動することで改善することが可能です。次の例ではfilter関数を省略し、addsuffixの引数に実行時に指定されるパターンを使っています。

```
# $(call find-program,wildcard-pattern)
find-program = $(wildcard                          \
                 $(addsuffix /$1,                  \
                   $(sort                          \
                     $(subst :, ,                  \
                       $(subst ::,:.:,             \
                         $(patsubst :%,.:%,        \
                           $(patsubst %:,%:.,$(PATH))))))))
find:
        @echo $(words $(call find-program,*))
```

このようにすることで実行時間は0.17秒になりました。wildcard関数は、filter関数で取り除かれてしまうものも含めてすべてのファイルを返すわけではなくなった分だけ速くなりました。同様の例はGNU makeのマニュアルにも出てきます。最初の例ではfilter形式の展開パターン（%のみ）を使いますが、2番目の例ではwildcard形式のパターン（~、*、?、[...]、[^...]）を使う点に注意が必要です。

$(addprefix *prefix,name...*)

addprefixはaddsuffixを補完する関数です。次の例は一連のファイルが存在していて中身が入っていることをチェックします。

```
# $(call valid-files, file-list)
valid-files = test -s . $(addprefix -a -s ,$1)
```

この関数はこれまでのものと異なり、コマンドスクリプトの中で使われることを意図しています。シェルのtestコマンドを-sオプション（ファイルが存在して空でない場合に真となる）付きで使いチェックします。testコマンドは複数のファイルを指定する場合に-aオプション（and）を必要とするのでaddprefixで-aを各ファイル名の前に付加します。and連鎖の最初はドットであり、これは常に真となります。

† 訳注：それぞれパス指定の区切り文字であるコロンに対し、後に何もない、前に何もない、コロンとコロンの間に何もない状態を指しています。

`$(join prefix-list,suffix-list)`

joinはdirとnotdirを補完する関数です。これは2つのリストを受け取り、*prefix-list*の最初の要素と*suffix-list*の最初の要素を連結し、*prefix-list*の2番目の要素と*suffix-list*の2番目の要素を連結し、という作業を行います。これによりdirとnotdirにより分解されたリストを再構成することができます。

4.2.4 実行制御

これまで見てきた関数はリストに対して処理を行うものだったため、繰り返しの構造がなくてもうまく働きました。しかしながら実際のところ繰り返しやある種の条件判断ができないとmakeのマクロ言語は非常にかぎられた働きしかできません。ありがたいことにmakeはどちらの機能も提供しています。またここではerror関数を扱います。error関数もある意味で実行制御を行う関数です。

`$(if condition,then-part,else-part)`

if関数（3章に登場したifeq、ifneq、ifdef、ifndef命令と混同しないように）は、条件式の値により2つあるマクロの1つを選択します。*condition*の展開結果が何か（空白であっても）文字を含んでいた場合に条件は真となります。真の場合*then-part*が展開されることになります。もし*condition*の展開結果が空となった場合には条件が偽となり、*else-part*が展開されます†。

次の例ではmakeがWindow上で実行されているかどうかを簡単にチェックしています。Windowsでのみ定義されているCOMSPEC環境変数の存在を調べます。

```
PATH_SEP := $(if $(COMSPEC),;,:)
```

makeは*condition*を評価する際に、前と後ろについている空白を取り除いた後、値を展開します。展開した結果が（空白も含めて）何らかの文字列になった場合には条件は真となります。これによりmakeの実行がWindows上であってもUnix上であっても、PATH_SEPにはパス区切りとして正しい文字が設定されます。

3章で、（evalのような）最新の機能を使う前にmakeのバージョンをチェックする方法を扱いました。ifとfilterは文字列の値をチェックする際によく使われます。

```
$(if $(filter $(MAKE_VERSION),3.80),,\
  $(error This makefile requires GNU make version 3.80.))
```

新バージョンのmakeがリリースされた場合には、それを利用可能なバージョンとして追加しなければなりません。

```
$(if $(filter $(MAKE_VERSION),3.80 3.81 3.90 3.92),,\
  $(error This makefile requires one of GNU make version ….))
```

† 3章ではマクロ言語とその他のプログラミングとの違いを示しました。マクロの定義や展開を通して元の文字列を変換し結果の文字列を得るのがマクロ言語です。この違いはif関数がどのように働くかを見れば、より明確になるでしょう。

この手法は新しいmakeをインストールするたびに書き直さなければならないという欠点があります。しかしこのような状況は非常にまれです（3.80は2002年の10月以来ずっと正式リリース版です）。もし条件が真となれば何も実行されず、そうでなければerrorによりmakeの実行を終了させるので、このテストはmakefileの最初に置くことができます。

`$(error text)`

error関数は重大なエラーを示すメッセージを表示するために使います。メッセージを表示した後、makeは終了ステータス2で実行を終了します。メッセージ表示にはmakefileの名前と行番号が前置されます。以下はmake向けに一般的なassertの機能を実現します。

```
# $(call assert,condition,message)
define assert
  $(if $1,,$(error Assertion failed: $2))
endef
# $(call assert-file-exists,wildcard-pattern)
define assert-file-exists
  $(call assert,$(wildcard $1),$1 does not exist)
endef
# $(call assert-not-null,make-variable)
define assert-not-null
  $(call assert,$($1),The variable "$1" is null)
endef
error-exit:
        $(call assert-not-null,NON_EXISTENT)
```

最初のassert関数は最初の引数を評価し、偽であった場合には第2引数をエラーメッセージとして表示します。2番目の関数は最初の関数を利用してwildcardのパターンが実際に存在しているかをチェックします。ここでは、引数には複数のパターンを指定することができます。3番目の関数は変数の評価結果を利用する便利な検証機構です。makeの変数には別の変数の名前も含めて何でも格納できます。ところで別の変数名が格納されていたとして、その変数の値を取り出すにはどうすればよいのでしょう。単純に2回展開すればよいのです。

```
NO_SPACE_MSG := No space left on device.
NO_FILE_MSG := File not found.
…;
STATUS_MSG := NO_SPACE_MSG
$(error $($(STATUS_MSG)))
```

単純な例にするために少し不自然な部分があります。STATUS_MSGにエラーメッセージの変数名を格納することで、いくつかのエラーメッセージのうちの1つがセットされます。メッセージを出力するときには、$(STATUS_MSG)としてエラーメッセージの変数名を取り出し、$($(STATUS_MSG))としてその中身であるメッセージ文字列を取り出します。前述のassert-not-null関数では引数はmake変数の名前を想定しています。そのため$1で引数を展開して変数名を取り出し、$($1)で再度展開することにより値が入っているか調べます。空であった場合には$1に入っている変数名をエラーメッセージの中で使います。

```
$ make
Makefile:14: *** Assertion failed: The variable "NON_EXISTENT" is null. Stop.
```

同様にwarning関数も存在し、（詳しくは「4.2.5 比較的重要なその他の関数」を参照してください） makeを終了させずにerror関数と同じ形式でメッセージを表示します。

`$(foreach variable,list,body)`

foreach関数を使うと、異なる値に置換しながら文字列の展開を繰り返すことができます。これは異なる引数を用いながら関数を繰り返し実行するのとは違います（もちろん、そういうことも可能ですが）。例を示しましょう。

```
letters := $(foreach letter,a b c d,$(letter))
show-words:
        # letters has $(words $(letters)) words: '$(letters)'

$ make
# letters has 4 words: 'a b c d'
```

このforeachが実行されると、繰り返しを制御する変数letterに“a b c d”がそれぞれ設定され、繰り返しの本体である$(letter)がそのつど展開されます。展開された値は空白で区切られて蓄積されます。

次の例では、一連の変数に値が設定されているか調べます。

```
VARIABLE_LIST := SOURCES OBJECTS HOME
$(foreach i,$(VARIABLE_LIST), \
  $(if $($i),,                \
    $(shell echo $i has no value > /dev/stderr)))
```

（上記の例が動作するには、/dev/stderrを利用できるシステムである必要があります。）

ここではVARIABLE_LIST内の単語がiに設定されて繰り返しが実行されます。if関数の条件判断ではまず$iが評価され変数名が取り出されます。次いで再度$($i)として評価され値が入っているかどうかがチェックされます。値が入っていた場合にはif関数の第2引数が空なので何も実行されませんが、入っていなかった場合には警告が表示されます。echoコマンドの出力をリダイレクトしなかった場合には、shell関数の出力がmakefileに展開され文法エラーになります。ご覧のとおり、このforeachは何かを展開するわけではありません。

以前お約束したとおり、単語リストの中から指定した文字列を含む単語だけを集める関数を紹介しましょう†。

```
# $(call grep-string, search-string, word-list)
define grep-string
  $(strip                          \
    $(foreach w, $2,               \
      $(if $(findstring $1, $w),   \
        $w)))
```

† 訳注：strip関数は「4.2.5 比較的重要なその他の関数」で詳しく説明されています。

```
endef
words := count_words.c counter.c lexer.l lexer.h counter.h
find-words:
        @echo $(call grep-string,un,$(words))
```

残念ながらこの関数はパターンを扱えませんが、単純な文字列ならうまく働きます。

```
$ make
count_words.c counter.c counter.h
```

4.2.5 比較的重要なその他の関数

いくつかの雑多な（けれども重要な）文字列関数を紹介します。foreachやcallと比較すれば地味な存在ですが、気づくと至るところで使っている、そんな関数です。

`$(strip text)`

stripは*text*の前と後ろについている空白を取り除き、途中に入っている空白列を1つの空白に置き換えます。条件判断で使う変数をきれいにするというのがよくある使い方です。複数の行をまとめたマクロや変数から不必要な空白を取り除くために筆者は使います。余分な空白に影響を受ける関数に対して関数引数$1や$2などをstripで囲むのはよいアイディアといえるでしょう。関数引数を区切るカンマの後ろにある空白を次の引数の一部としてしまうというmakeの微妙なふるまいは、しばしば忘れられてしまうものです。

`$(origin variable)`

originは変数の出自を示す文字列を返します。これは変数の値を使用するかどうか決める際に役に立ちます。例えば変数の値が環境変数からのものであった場合には無視するけれども、コマンドラインで設定されたものなら使いたいというような場合です。次はもう少し具体的な例として、変数が定義されているかをチェックする新しいassert関数を紹介します。

```
# $(call assert-defined,variable-name)
define assert-defined
  $(call assert,                          \
  $(filter-out undefined,$(origin $1)),   \
  '$1' is undefined)
endef
```

originは次の値を返します。

undefined
: 変数が定義されていないことを表します。

default
: 変数の定義はmakeの組み込みデータベースからのものであることを表します。もし組み込みデータベースの値を変更していたなら、originは最新の定義について値を返します。

environment
: 変数は環境変数として定義されていたこと（そして--environment-overridesオプ

ションは指定されていない）を表します。

environment override
: 変数は環境変数として定義されていたこと（そして--environment-overridesオプションが指定されていた）を表します。

file
: 変数はmakefileで定義されていることを表します。

command line
: 変数はコマンドラインで定義されていることを表します。

override
: 変数はoverride命令で定義されていることを表します。

automatic
: 変数はmakeによって定義された自動変数であることを表します。

$(warning *text*)
: makeを終了させないという点を除けばwarning関数はerror関数と同じです。error関数のように、表示されるメッセージには現在のmakefileと行番号が前置されます。warning関数は展開されないので、ほとんどあらゆる場所で使うことができます。

```
$(if $(wildcard $(JAVAC)),,                                    \
  $(warning The java compile variable, JAVAC ($(JAVAC)), \
            is not properly set.))
```

4.3 高度なユーザ定義関数

今後もマクロ関数を書くために多くの時間を費やすことになると思いますが、残念なことに関数をデバッグするための機能は、それほど多く提供されているわけではありません。この状況を改善するために、ここでは簡単なデバッグ用トレースマクロを書いて見ましょう。

前述したように、call関数は引数を$1や$2といった番号の付いた変数に結び付けます。引数はいくつあってもかまいません。特殊なケースとして現在実行中の関数名（すなわち変数の名前）は$0として取り出すことができます。この情報を使って、マクロの展開を追跡する2つのデバッグ用関数を作ることができます。

```
# $(debug-enter)
debug-enter = $(if $(debug_trace),\
                $(warning Entering $0($(echo-args))))

# $(debug-leave)
debug-leave = $(if $(debug_trace),$(warning Leaving $0))

comma := ,
echo-args = $(subst ' ','$(comma) ',\
              $(foreach a,1 2 3 4 5 6 7 8 9,'$($a)'))
```

関数aとbがどのように実行されるのか調べたいときには、これらのトレース関数を使うことができます。

```
debug_trace = 1

define a
  $(debug-enter)
  @echo $1 $2 $3
  $(debug-leave)
endef

define b
  $(debug-enter)
  $(call a,$1,$2,hi)
  $(debug-leave)
endef

trace-macro:
        $(call b,5,$(MAKE))
```

関数の最初と最後にdebug-enterとdebug-leaveを置くことにより、自作関数がどのように展開されているかを追跡することができます。これらの関数は完全なものではありません。echo-args関数は最初の9つの引数しか表示できないうえに、callに与えられた引数の数も数えることができません（といっても、それはmakeにもできないのですが）。にもかかわらず、筆者はこのマクロをそのままデバッグのために使っています。実行すると次のようなトレース出力が行われます。

```
$ make
makefile:14: Entering b( '5', 'make', '', '', '', '', '', '', '')
makefile:14: Entering a( '5', 'make', 'hi', '', '', '', '', '', '')
makefile:14: Leaving a
makefile:14: Leaving b
5 make hi
```

筆者の知り合いは次のようにいっています、「以前はmakeがプログラム言語だなんて考えたこともなかった」と。GNU makeは、いにしえのmakeとは違うものになっているのです。

4.3.1 eval関数と値

evalはその他の関数とまったく異なっています。この関数は文字列を直接makeの構文解析に渡します。例えば次のように使います。

```
$(eval sources := foo.c bar.c)
```

evalの引数はまず（普通の関数と同じように）変数が展開され、次いでその文字列があたかもmakefileに書かれていたかのように構文解析と評価が行われます。この例はとても簡単なので、おそらくなぜこんなことにわざわざ関数を使うのか不思議に思うことでしょう。そこで、もう少しおもしろい例を紹介します。非常にたくさんのプログラムをコンパイルするmakefileがあるとします。そ

こでは各プログラムに対してそれぞれ変数（仮にsource、headers、objectsとしましょう）を定義するものとします。これらの変数の代入をそれぞれの値に対して何度も何度も繰り返す以下のコードを想像してください。

```
ls_sources := ls.c glob.c
ls_headers := ls.h glob.h
ls_objects := ls.o glob.o
…
```

このように何度も繰り返すのではなく、同じことを行うマクロが使えないでしょうか。

```
# $(call program-variables, variable-prefix, file-list)
define program-variables
  $1_sources = $(filter %.c,$2)
  $1_headers = $(filter %.h,$2)
  $1_objects = $(subst .c,.o,$(filter %.c,$2))
endef

$(call program-variables, ls, ls.c ls.h glob.c glob.h)

show-variables:
        # $(ls_sources)
        # $(ls_headers)
        # $(ls_objects)
```

program-variablesマクロは2つの引数を受け付けます。作成される3つの変数のプリフィックスと、マクロが選択して変数に設定するファイルのリストです。しかしこのマクロを実行するとエラーになってしまいます。

```
$ make
Makefile:7: *** missing separator. Stop.
```

これが思ったように動かないのは、makeのパーサの動作に関係があります。（最上位の構文解析のレベルでは）複数の行に展開されるマクロは構文エラーになります。

この場合パーサはこの行がルールもしくはコマンドスクリプトの一部であるけれども区切り文字が見つからないと認識します。まったくわかりにくいエラーメッセージです。eval関数はこの問題を扱うために導入されました。次のようにcallの行を変更すると、

```
$(eval $(call program-variables, ls, ls.c ls.h glob.c glob.h))
```

思ったとおりの結果を得られます。

```
$ make
# ls.c glob.c
# ls.h glob.h
# ls.o glob.o
```

eval関数は複数行のマクロを扱うことができるうえに、eval関数自体は展開結果を返さないので、こういった構文解析上の問題を解決できます†。

これで非常に単純な3つの変数を定義するマクロができました。$1_sourcesなどを使うことで関数の引数として渡したプリフィックスを使って代入先変数名を作っています。これは最初に説明したような純粋に生成された変数名というわけではありませんが、ほぼ同様のことを実現しています††。

この例についてもう少し見ると、ルールをマクロの中に入れることができると気づくでしょう。

```
# $(call program-variables,variable-prefix,file-list)
define program-variables
  $1_sources = $(filter %.c,$2)
  $1_headers = $(filter %.h,$2)
  $1_objects = $(subst .c,.o,$(filter %.c,$2))

  $($1_objects): $($1_headers)
endef

ls: $(ls_objects)

$(eval $(call program-variables,ls,ls.c ls.h glob.c glob.h))
```

これら2つのprogram-variablesマクロが引数に付随する空白に関する問題を明らかにするか説明しましょう。最初のprogram-variablesでは単純に引数を使用しているので、引数の前に付いている空白に影響を受けません。つまりこのコードは$1と$2について、引数の前に空白が付加されているかにかかわりなく動作します。ところが、2番目のprogram-variablesでは生成した変数を$($1_objects)と$($1_headers)として使っています。最初の引数（ls）の前に空白を入れてしまうと、生成した変数が空白で始まることになってしまいます。先頭に空白のある変数は定義してい

---

† 訳注：実はevalが必要なのは複数行に展開されるからではありません。まずマクロがdefineにより作成される特殊な形式の変数であったことを思い出してください。マクロと変数の違いは、前者が改行を中に含めることができるという点だけでした。ここで作られたマクロprogram-variablesは、内部で変数の代入を行っています。つまりprogram-variablesは変数の代入式を値として持つ変数だということができます。もしマクロの内部が1行の変数代入だけであれば、次の式と等価です。

```
program-variables = $1_sources = $(filter %.c,$2)
```

引数$1がlsに展開されたとして、このls_sources変数を作るためには、program-variables変数（マクロ）の評価を1回行ったうえで、その中身を変数の代入式として評価しなければなりません。1回は通常のmakefile構文解析中に行われますが、もう1回は明示的に指示しなければなりません。それがeval関数を必要とする理由です。また、makeの出したエラーは、変数（マクロ）の評価が1回行われると次にルールの評価が行われることを示しています。つまりエラーになった例では、program-variablesを1回だけ評価した結果得られた3行の変数代入式が、ターゲット行かコマンドスクリプトとして解釈されてしまったようです。そのため区切り文字（コロンまたはタブ）のない行が見つかったというエラーになりました。こういったことを毎回考えるのは複雑なので、ここで述べられているように複数の行に展開されるマクロをmakefileのトップレベルで使うときには必ずeval関数に渡すのがよいと思います。

†† 訳注：本来はソースファイルとしてls.cを渡したなら、そのソースファイル名からプリフィックスとなる部分を抽出し、それぞれの変数名（ls_sources、ls_headers、ls_objects）を作るべきですが、ここではプリフィックスを引数として指定しています。

ないので、これらの参照は値を持たないことになってしまいます。これは見つけにくい非常にやっかいな問題です。

このmakefileを実行すると、どうしたことか必須項目としての.hがmakeに無視されていることに気づきます。この問題を調査するために--print-data-baseオプションを使ってmakeの内部データベースをチェックします。すると何かおかしなことを見つけます。

```
$ make --print-database | grep ^ls
ls_headers = ls.h glob.h
ls_sources = ls.c glob.c
ls_objects = ls.o glob.o
ls.c:
ls.o: ls.c
ls: ls.o
```

ls.oに対する必須項目である.hファイルがなくなっています。これは生成された変数に何か問題があることを示しています。

makeがeval関数呼び出しを解析する際、最初にユーザ定義関数であるprogram-variablesを展開します。マクロの最初の行は次のようになります。

```
ls_sources = ls.c glob.c
```

マクロの各行は期待したとおりにその場で展開されるので、他の変数代入も同様に扱われます。その後次のルールが出てきます。

```
$($1_objects): $($1_headers)
```

これはまず変数名が展開され、次のようになります。

```
$(ls_objects): $(ls_headers)
```

そしてその外にある変数展開が行われて次のようになります。

```
:
```

定義した変数はどこに行ってしまったのでしょう。その前にある3つの変数代入は、展開されているけれどもmakeによってまだ評価されていなかった、というのがその答えです。どうなっているのかもう少し見てみましょう。program-variablesが展開された後、makeは次のような行を見ることになります。

```
$(eval ls_sources = ls.c glob.c
ls_headers = ls.h glob.h
ls_objects = ls.o glob.o

  :)
```

その結果evalの関数はそれぞれの行を評価して変数を定義します。結局、「ルール中の変数は実際に値が定義される前に参照された」というのが答えです。

3つの変数が定義されるまで、生成された変数に対する展開を明示的に遅らせることでこの問題を回避することができます。これは変数の前にドル記号を置くことで実現できます。

```
$$($1_objects): $$($1_headers)
```

今度は期待どおりに必須項目が表示されました。

```
$ make -p | grep ^ls
ls_headers = ls.h glob.h
ls_sources = ls.c glob.c
ls_objects = ls.o glob.o
ls.c:
ls.o: ls.c ls.h glob.h
ls: ls.o
```

まとめると、evalの引数は2回展開されることになります、つまり1回はmakeがevalの引数を展開する際に、もう1回はeval自身によって行われます。ここでは、生成された変数の評価を遅らせることにより最後の問題を解決しました。別の解決策として、変数の代入をそれぞれeval関数に渡すことで変数の評価を先に実行してしまうという手があります。

```
# $(call program-variables,variable-prefix,file-list)
define program-variables
  $(eval $1_sources = $(filter %.c,$2))
  $(eval $1_headers = $(filter %.h,$2))
  $(eval $1_objects = $(subst .c,.o,$(filter %.c,$2)))
  $($1_objects): $($1_headers)
endef

ls: $(ls_objects)

$(eval $(call program-variables,ls,ls.c ls.h glob.c glob.h))
```

変数の代入をそれぞれeval関数に渡すことにより、その実行をマクロが展開される処理の一部とすることができます。そのためそれらの変数はマクロの中で使うことができます。

makefileを改善したことで、マクロの中に別のルールも入れられるようになりました。ここでのプログラムはオブジェクトファイルに依存しています。このmakefileの仕上げとして最上位レベルのターゲットallと、このmakefileが管理できるプログラム名を保持するための変数を用意しましょう。

```
#$(call program-variables,variable-prefix,file-list)
define program-variables
  $(eval $1_sources = $(filter %.c,$2))
  $(eval $1_headers = $(filter %.h,$2))
  $(eval $1_objects = $(subst .c,.o,$(filter %.c,$2)))
```

```
    programs += $1

    $1: $($1_objects)

    $($1_objects): $($1_headers)
  endef

  # デフォルトターゲットなので、allはここに置く
  all:

  $(eval $(call program-variables,ls,ls.c ls.h glob.c glob.h))
  $(eval $(call program-variables,cp,...))
  $(eval $(call program-variables,mv,...))
  $(eval $(call program-variables,ln,...))
  $(eval $(call program-variables,rm,...))

  # ここでならprogramsも定義されているので、必須項目として指定する
  all: $(programs)
```

allターゲットとその必須項目を置く位置について説明しておきましょう。programs変数は5つのeval関数が実行されるまでは正しく定義されませんが、allはデフォルトターゲットなので、makefileの中で最初のターゲットでなければなりません。この制約を満たすために、まずallターゲットを最初のほうに置き、必須項目は後で指定することにします。

いくつかの変数がかなり初期の段階で評価されてしまうので、program-variables関数には問題がありました。実のところ、makeはこういった問題に対処するためにvalue関数を提供しています。value関数は引数として指定された変数の値を展開せずに返します。展開されなかった値はそのままeval関数に渡して処理することができます。展開しない値が返されることで、前述の例では変数参照を$を使って遅延させた問題を回避することができます。

残念なことに、この関数をprogram-variablesマクロに対して使うことはできません。なぜならvalueは与えられたものは全部処理するかまったく何もしない関数だからです。value関数を使ってしまうとprogram-variables中の変数は一切展開されなくなってしまいます。さらに、valueは複数の引数を受け取ることができない（もし受け取れたとしても、それらに対しても同様に何もしない）ので、プログラム名とファイルリストの引数は展開されなくなってしまいます。

このような制限により、本書ではvalue関数をあまり使っていません。

4.3.2 フック関数

ユーザ定義関数というのは、単に文字列を格納している変数に過ぎません。call関数は文字列の中に$1や$2といった参照が存在していれば、それを展開しているだけです。もし関数にそういった変数の参照がなければcallは何もしません。実際、変数に文字列が何も入っていなくても、callは何もしません。エラーも警告も出さないのです。これは関数名を間違えてしまったような状況ではイライラする動作ですが、一方ではとても使いやすいともいえるのです。

関数は再利用が可能です。関数をいろいろなところで使えば使うほど、上手に作ることの価値が出てきます。関数にフックを入れておくことで、もっと再利用度を上げることができます。フックとは別

の関数を呼び出すことで通常の処理の中に利用者の指定した独自の動作を入れることができるというものです。

多くのライブラリを構築するためのmakefileを想定してみましょう。あるシステムではranlibを実行しなければなりません。また別のシステムではchmodを実行しなければならないかもしれません。それらの操作を明示的にコマンドの中に入れるよりも関数を作りそこにフックを入れることもできます。

```
# $(call build-library, object-files)
define build-library
  $(AR) $(ARFLAGS) $@ $1
  $(call build-library-hook,$@)
endef
```

フックを使うには、build-library-hookという関数を用意します。

```
$(foo_lib): build-library-hook = $(RANLIB) $1
$(foo_lib): $(foo_objects)
        $(call build-library,$^)

$(bar_lib): build-library-hook = $(CHMOD) 444 $1
$(bar_lib): $(bar_objects)
        $(call build-library,$^)
```

4.3.3 関数に値を渡す

関数は、データを4つの方法で受け取ることができます。call関数を通して渡された引数、関数の外で定義された変数、自動変数、そしてターゲットに固有の変数です。引数を介することで関数外の変数に影響されなくなるため、引数を使うことが関数を独立した部品として扱うための最もよい選択です。しかし独立した部品化が最も重要な設計基準ではない場合もあります。

共通のmake関数を利用する複数のプロジェクトがあるとしましょう。各プロジェクトは変数のプリフィックス、例えばPROJECT1_で区別され、重要な変数はすべてこのプリフィックスに全プロジェクト共通のサフィックスで形成されます。3章の例に出てきたPROJECT_SRCのようなPROJECT1_SRC、PROJECT1_BIN、PROJECT1_LIBといった変数です。変数を3つ渡す必要のある関数を作る代わりに、プリフィックスだけを渡して内部で変数名を生成することができます。

```
# $(call process-xml,project-prefix,file-name)
define process-xml
  $($1_LIB)/xmlto -o $($1_BIN)/xml/$2 $($1_SRC)/xml/$2
endef
```

関数に値を渡す別の方法として、ターゲット固有変数を使う方法が考えられます。この方法は、たいていの場合において通常の値を使うけれども、まれに特殊な処理を必要とする場合には特に有効です。インクルードファイルに定義されているルールを、makefileで定義されている変数を使って呼び出す場合にもターゲット固有変数を利用できます。

```
release: MAKING_RELEASE = 1
release: libraries executables
…

$(foo_lib):
        $(call build-library,$^)
…

# $(call build-library, file-list)
define build-library
  $(AR) $(ARFLAGS) $@             \
    $(if $(MAKING_RELEASE),       \
      $(filter-out debug/%,$1),   \
      $1)
endef
```

このコードではリリースライブラリの構築であることを示すためにターゲット固有変数を使います。その場合build-library関数は、ライブラリにデバッグ用モジュールが一切入らないようにします†。

† 訳注：このコードでは、ターゲット固有変数を使っているので、releaseターゲットを構築する間はMAKING_RELEASE変数が定義されます。そのためreleaseターゲットの必須項目がbuild-library関数により作成されるときに、デバッグモジュールの排除を行うコードが有効になります。

5章
コマンド

makeのコマンドの基礎についての多くを学んできましたが、ここで同じ視点に立っているかを確認するために少し復習してみましょう。

コマンドは基本的に1行のシェルスクリプトです。makeは1行ごとに取り出し、サブシェルに渡して実行します。シェルの呼び出しを省略してもプログラムの動作が変わらないのであれば、比較的重い処理であるfork/execを最適化することができます。このチェックは各行を走査してワイルドカード文字や入出力リダイレクションなどシェルの特殊文字が使われているかを調べることで行われます。もし見つからなければ、makeはサブシェルに渡すのではなく、そのコマンドを直接実行します。

既定のシェルは/bin/shです。このシェルはmakeの変数SHELLにより変更できますが、この変数は環境変数から引き継ぐものではありません。makeが開始する際にユーザの環境変数をmakeの変数として取り込みますが、SHELLだけは例外です。ユーザの選択したシェルが（おそらく何らかのダウンロードしたソフトウェアパッケージの一部である）makefileの動作に影響を及ぼさないようにするためです。makeの使用する既定のシェルを変更したければmakefileで明示的にSHELL変数を設定しなければなりません。

5.1　コマンドの構文解析

ターゲットに続く行で最初の文字がタブである行は（バックスラッシュを使った前の行からの継続行でないかぎり）、コマンドであると認識されます。GNU makeはその他の文脈ではタブをできるかぎりインテリジェントに扱おうとします。例えばコメント、変数の代入、include命令などは明らかにタブが最初の文字にはなりません。makeはコマンド行をターゲットの直後以外の場所で読み込んだ場合、エラーメッセージ（「コマンド行が最初のターゲット以前に置かれています」の意）を出力します。

```
makefile:20: *** commands commence before first target. Stop.
```

このエラーメッセージは「最初」のターゲットよりもずっと後、つまりmakefileの中ほどであっても発生する場合があるので、奇妙に思うことがあるかもしれませんが、もう混乱することはないでしょう。このメッセージを改善するならば、「ターゲットの文脈以外でコマンドが見つかりました」ということになるでしょうか。

構文解析器が正常な場所にあるコマンドを見つけると、1行ずつスクリプトを構築する**コマンド解析モード**に変わります。そしてコマンド以外のものに突き当たると、停止します。そこでスクリプトは終了します。スクリプトには以下のものが含まれます。

- タブで始まるサブシェルによって実行されるコマンド行はスクリプトです。make要素の一部（例えばifdefやコメントやinclude命令など）でもかまいません。それらはコマンド解析モードの間コマンドとして扱われます。
- 空行は無視されます。サブシェルによって実行されることはありません。
- #で始まる行もスクリプトです。前に空白（タブではなくて）が付いていてもかまいません。これらはmakefileのコメントとして無視されます。
- ifdef、ifeqのような条件判断命令はコマンドスクリプトの一部として認識され処理されます。

先頭にタブがなければ、組み込みのmake関数はコマンド解析モードを終了させます。つまり組み込みの関数は有効なシェルコマンドになるか何も返さないかどちらかでなければなりません。warning関数とeval関数は展開文字列を返さない関数の例です。

空行やコメントをコマンドスクリプトの中に置けることに最初は戸惑うでしょう。次の例はそれらがどのように扱われるのかを示しています。

```
long-command:
        @echo Line 2: 空行が続く

        @echo Line 4: シェルのコメントが続く
        # これはシェルのコメント（先頭にタブがある）
        @echo Line 6: makeのコメントが続く
# これはmakeのコメント（行の先頭から始まっている）
        @echo Line 8: 時下げしたmakeのコメントが続く
  # これはmakeのコメント（空白で字下げされている）
        # これもmakeのコメント（空白で字下げされている）
        @echo Line 11: 条件判断が続く
    ifdef COMSPEC
        @echo Windowsで実行中
    endif
        @echo Line 15: コマンドとしてのwarningが続く
        $(warning 警告)
        @echo Line 17: コマンドとしてのevalが続く
        $(eval $(shell echo シェルのecho 1>&2))
```

5行目と10行目は見た目には同じですが、実態はまったく違います。5行目は先頭にタブが置かれているのでシェルのコメントになりますが、10行目は8個の空白で字下げされているだけなのでmake

のコメントです。makeのコメントをこのように書くのは（難解なmakefileコンテストに出場するのでなければ）明らかに推奨できません。下の出力結果からわかるように、たとえコマンドスクリプトの中に書かれていたとしても、makeのコメントは実行もされなければ画面に出力されることもありません。

```
$ make
makefile:2: 警告
シェルのecho
Line 2: 空行が続く
Line 4: シェルのコメントが続く
# これはシェルのコメント（先頭にタブがある）
Line 6: makeのコメントが続く
Line 8: 字下げしたmakeのコメントが続く
Line 11: 条件判断が続く
Windowsで実行中
Line 15: コマンドとしてのwarningが続く
Line 17: コマンドとしてのevalが続く
```

warning関数とeval関数からの出力が順番どおりではないように見えますが、これで正しいのです（「5.4　コマンドを評価する」にてどのような順番で評価されるかを扱います）。コマンドスクリプトには空行やコメントを何行でも入れられることが、イライラするようなわかりにくいエラーの原因となっています。偶然、行頭にタブのある行が入り込んだ場面を想像してください。その直前に置かれたターゲット（コマンドが付随していてもしていなくても）との間にコメントか空行しかなかった場合に、makeは誤って挿入されたタブで始まる行を、そのターゲットに対するコマンドの一部として扱います。ここで見てきたように、これはまったく正当な状態であり、makefile内の別の場所（またはインクルードファイルの中）で同じターゲットに対するルールが定義されていないかぎりエラーも警告も出ません。

もし幸運であったなら、偶然に入ったタブの行と直前のターゲットとの間に空行でもコメントでもないものが入っているかもしれません。その場合には、“commands commence before first target”（コマンド行が最初のターゲット以前に置かれています）というメッセージが表示されます。

ソフトウェアツールについて解説してもよい時期でしょう。コマンド行であることを示すために行頭にタブを置くという選択は不幸な選択であったことは万人が認めるところでしょう。しかしながら、これを変更することはもうできません。文法を認識する現代的なエディタを使えば、疑わしい個所を目に見える形で示すことで、潜在的な問題に対処できます。GNU emacsにはmakefileを編集するための便利なモードがあります。このモードでは、行継続の文字の後ろに入っている空白や行頭のタブと行頭の空白が混在している個所といった簡単な文法エラーを探し出し強調表示します。

5.1.1　長いコマンド

各コマンドはそれぞれ別のシェル（もしくはそう見えるように）で実行されるので、いっしょに実行されなければならないコマンド列は、特殊な扱いが必要になります。例えば、ファイルのリストが入っているファイルを生成しなければならないとしましょう。Javaコンパイラは大量のファイルをコンパイ

ルするときには、そういったファイルを受け付けることができます。その場合、次のようなコマンドスクリプトを作ることになるでしょう。

```
.INTERMEDIATE: file_list
file_list:
        for d in logic ui
        do
          echo $d/*.java
        done > $@
```

これがうまく働かないことは、もはや自明です。これはエラーになります。

```
$ make
for d in logic ui
/bin/sh: -c: line 2: syntax error: unexpected end of file
make: *** [file_list] Error 2
```

最初の修正では、各行末に行継続の文字を付加します。

```
.INTERMEDIATE: file_list
file_list:
        for d in logic ui \
        do                \
          echo $d/*.java  \
        done > $@
```

しかし次のようなエラーになります。

```
$ make
for d in logic ui \
do                \
  echo /*.java    \
done > file_list
/bin/sh: -c: line 1: syntax error near unexpected token `>'
/bin/sh: -c: line 1: `for d in logic ui do echo /*.java
make: *** [file_list] Error 2
```

何が起きたのでしょうか。問題は2つあります。まずループ制御変数dに対する参照をエスケープする必要があります。次にforループは1つの行としてサブシェルに渡すことになるので、ファイルのリストの後ろとforループ内の行にセミコロンを補わなければなりません。

```
.INTERMEDIATE: file_list
file_list:
        for d in logic ui; \
        do                 \
          echo $$d/*.java; \
        done > $@
```

これで期待したファイルが作られるようになりました。ターゲットは.INTERMEDIATEとして宣言されているので、コンパイルが終了した後でこの一時ファイルはmakeにより消去されます。

ディレクトリのリストがmakeの変数に入っているほうが、より現実的な例といえるでしょう。もしファイルの数が比較的少ないことが明らかならば、forループではなくmakeの関数を使って同じことができます。

```
.INTERMEDIATE: file_list
file_list:
        echo $(addsuffix /*.java,$(COMPILATION_DIRS)) > $@
```

しかしforループを使う方法なら、時間がたつにつれ扱うディレクトリが増えてもコマンドラインの長さ制限に抵触する可能性を低くできます。

ディレクトリの変更もmakeのコマンドスクリプトに関するよくある問題です。次のようなコマンドスクリプトでは、ctagsコマンドがsrcサブディレクトリで実行されないことは明らかです。

```
TAGS:
        cd src
        ctags --recurse
```

期待したとおりに動作させるには、両方のコマンドを1行で書くかバックスラッシュで改行をエスケープする必要があります（そしてコマンドの間にセミコロンを置きます）。

```
TAGS:
        cd src; \
        ctags --recurse
```

ctagsを実行する前にcdの実行結果を調べるのが、よりよい方法です。

```
TAGS:
    cd src && \
    ctags --recurse
```

ある種の状況下では、セミコロンがなくてもmakeやシェルがエラーを出さない場合があります。

```
disk-free = echo "Checking free disk space..." \
df . | awk '{ print $$4 }'
```

この例は簡単なメッセージの後に、ドライブの空きブロック数を表示するものです。本当にそうでしょうか。echoコマンドの後にセミコロンを置き忘れてしまいました。そのためdfプログラムは実行されません。その代わりに次のような内容がawkコマンドに渡され、律儀に4番目のフィールドが“space...”と表示されます。

```
Checking free disk space... df .
```

継続行ではなくdefine命令を使って複数行からなるコマンド列を作るときにも同じような状況が発生します。残念ながらこれは同じ問題ではありません。複数行のマクロが展開される際には、各行の先頭にタブが挿入され、makeによりそれぞれ別々に扱われます。マクロ内の各行が1つのサブシェルで実行されることはありません。したがって、マクロ内の行継続にも注意を払う必要があります。

5.1.2 コマンド修飾子

コマンドはいくつかのプリフィックスを置くことで修飾することができます。「寡黙」なプリフィックスである@についてはすでに多くの場所で使っています。以下はすべてのプリフィックスに対するリストの長所短所含めた解説です。

@

コマンドを表示しません。古いmakeとの互換性から、ターゲットを.SILENTの必須項目とすることで、そのターゲットに対するコマンドをすべて表示させないようにできます。けれどもコマンドスクリプト中、個々のコマンドに対して制御できるため@を使うほうが好まれています。この修飾子をすべてのターゲットに対して適用したい場合（いかなる理由からそうしたいのかは不明ですが）には、--silentオプション（-sオプション）を使えます。
コマンドを表示しないことでmakeからの出力は読みやすくなりますが、一方でコマンドスクリプトのデバッグは難しくなってしまいます。もし頻繁に@修飾子を取ったり付けたりするならば、QUIETという@修飾子の入った変数を作りコマンドの前に付けるという方法が使えます†。

```
QUIET = @
hairy_script:
        $(QUIET) complex script …
```

makeの実行する複雑なコマンドの状況を見たいときには、QUIET変数をコマンドラインからリセットします。

```
$ make QUIET= hairy_script
complex script …
```

-

マイナス記号はコマンドのエラーを無視するようにmakeに指示します。makeがコマンドを実行する際には、そのプログラムやパイプの終了ステータスをチェックします。終了ステータスが0以外（実行に失敗した）であった場合には、残りのコマンドを実行せずにmakeは終了します。この修飾子は、コマンドの終了ステータスを無視し、あたかもエラーが起きなかったかのように実行を継続するようmakeに指示します。この話題は「5.1.3　エラーと中断」で詳しく扱います。

† 訳注：hairy_scriptを直訳すると「込み入った_スクリプト」、complex scriptは「複雑なスクリプト」ということになります。

古いmakeとの互換性から、ターゲットを.IGNORE特殊ターゲットの必須項目とすることで、そのターゲットのコマンドスクリプトで発生したエラーを無視することができます。makefile中すべてのエラーを無視したい場合には、--ignore-errorsオプション（-iオプション）を使うことができます。これも使いやすいオプションとはいえません。

+

プラス修飾子は、--just-printオプション（-nオプション）が指定されている場合でも、そのコマンドを実行するようmakeに指示します。これは再帰的なmakefileを作るときに使います。詳しくは「6.1　再帰的make」で扱います。

これらの修飾子は各行に対して使います。コマンドが実行される前には取り除かれるということはいうまでもありません。

5.1.3　エラーと中断

makeが実行するコマンドは終了ステータスを返します。0はコマンドの実行が成功したことを表し、0以外は何らかのエラーが起きたことを表します。単にエラーが起きたこと以上の情報を終了ステータスとして返すプログラムもあります。例えばgrepは合致するものが見つかった場合には0（成功）を、見つからなかった場合には1を、そして何らかのエラーが起きたときには2を返します。

通常、プログラムの実行が失敗した（つまり0以外の終了ステータスが返された）場合、makeはコマンドスクリプトの実行を中断し、エラーの終了ステータスを返してmake自身も終了します。状況によっては、できるだけ多くのターゲットを構築すべく実行を継続させたい場合もあります。例えば、すべてのコンパイルエラーを1回のmake実行で吐き出させるために、できるだけ多くのファイルをコンパイルしたい場合などです。その場合には--keep-goingオプション（-kオプション）を使えます。

-修飾子により個々のコマンドのエラーを無視できますが、筆者はできるかぎりこの機能を使わないようにしています。というのも、この機能はエラー処理の自動化を難しくするうえに見た目にも不恰好だからです。

makeがエラーを無視する場合には、角括弧で囲ったターゲット名とともに警告メッセージが表示されます。例えば次はrmコマンドが存在しないファイルを削除しようとした際の表示です。

```
rm non-existent-file
rm: cannot remove `non-existent-file': No such file or directory †
make: [clean] Error 1 (ignored)
```

rmを含めいくつかのコマンドにはエラーの終了ステータスを抑制するためのオプションを持っています。-fオプションによりエラーメッセージを抑制しつつ成功を示す終了ステータスをrmが返すようになります。こういったオプションを活用するのはマイナス記号の修飾子に依存するよりもよい方法です。

† 訳注：「non-existent-fileが削除できません:ファイルが存在しませんでした」の意。

場合によってはコマンドを失敗させたり、プログラムが成功した場合でもエラーとしたいといった状況もあるはずです。そのような場合にはプログラムの終了ステータスを反転します。

```
# ソースの中にデバッグ用のコードが残っていないか確認する
.PHONY: no_debug_printf
no_debug_printf: $(sources)
        ! grep --line-number '"debug:' $^
```

残念ながら、make 3.80にはバグがあるため、こういった直感的な使い方ができません。makeは!文字をシェル経由の実行が必要な機能だと認識しません。コマンドを直接実行してしまうため、結果はエラーとなります。この場合makeに対してシェルの特殊文字を含んだ命令をヒントとして与えることで対処できます。

```
# ソースの中にデバッグ用のコードが残っていないか確認する
.PHONY: no_debug_printf
no_debug_printf: $(sources)
        ! grep --line-number '"debug:' $^ < /dev/null
```

シェルのif構文をelseなしで使うと、思いがけないエラーの原因となります。

```
$(config): $(config_template)
        if [ ! -d $(dir $@) ]; \
        then                   \
          $(MKDIR) $(dir $@);  \
        fi
        $(M4) $^ > $@
```

コマンドの最初で出力ディレクトリの存在をチェックし、存在していなければmkdirを呼びます。不運にもディレクトリが存在していた場合にはifコマンドは失敗を示す終了ステータス（チェック部分の終了ステータス）を返しスクリプトを終了させます。else節を加えることが解決方法の1つです。

```
$(config): $(config_template)
        if [ ! -d $(dir $@) ]; \
        then                   \
          $(MKDIR) $(dir $@);  \
        else                   \
          true;                \ †
        fi
        $(M4) $^ > $@
```

シェルに対するコロン（:）は、常にtrueを返すけれども何も実行しないコマンドを表し、trueの代わりに使うことができます。次に示す別の手法もうまく働きます。

† 訳注：trueは何も行わず、終了ステータスとして0（成功）を返すコマンドです。シェルスクリプトの中で正常終了するコマンドが必要な場合に使われます。

```
$(config): $(config_template)
        [[ -d $(dir $@) ]] || $(MKDIR) $(dir $@)
        $(M4) $^ > $@
```

ここでの最初のコマンドは、ディレクトリが存在しているか、mkdirが正常終了した場合に成功となります。別の選択肢としてmkdir -pが考えられます。これによりディレクトリが存在していた場合でもmkdirは正常終了したことになります。これらの方法ではディレクトリが存在していても何らかのコマンドがサブシェルで実行されます。しかしwildcard関数を使えばディレクトリが存在していた場合にはコマンドの実行を省略できます。

```
# $(call make-dir, directory)
make-dir = $(if $(wildcard $1),,$(MKDIR) -p $1)
$(config): $(config_template)
        $(call make-dir, $(dir $@))
        $(M4) $^ > $@
```

それぞれコマンドは個別のシェルで実行されるため、セミコロンで区切った複数行に渡るコマンドを使うのはごく普通のことです。このようなコマンドスクリプトの中で発生するエラーはスクリプトを終了させないということに注意しなければなりません。

```
    target:
            rm rm-fails; echo But the next command executes anyway †
```

コマンドスクリプトをできるだけ短くし、終了ステータスを判断してスクリプトを終了する機会をmakeに与えるのが最もよい方法です。例を使って説明しましょう。

```
path-fixup = -e "s;[a-zA-Z:/]*/src/;$(SOURCE_DIR)/;g" \
             -e "s;[a-zA-Z:/]*/bin/;$(OUTPUT_DIR)/;g"

# 正しく動作する手法
define fix-project-paths
  sed $(path-fixup) $1 > $2.fixed && \
  mv $2.fixed $2
endef

# さらに優れている方法
define fix-project-paths
  sed $(path-fixup) $1 > $2.fixed
  mv $2.fixed $2
endef
```

このマクロは（スラッシュを使った）DOS形式のパス名を、特定のソースディレクトリと出力ディレクトリのパス名に変換します。このマクロは入力用と出力用のファイル名を受け取ります。sedコ

† 訳注：rm-fails（失敗するrm）によりrmコマンドは失敗することになりますが、次の「しかし次のコマンドはどのみち実行されてしまう」という文を表示するechoコマンドは実行されるということを示す例です。

マンドが正常に終了した場合のみ出力ファイルを上書きするように注意する必要があります。「正しい」ほうではsedとmvコマンドを&&でつなぎ、1つのシェルで実行しています。「優れている」ほうでは別々のコマンドとして実行しsedが失敗した場合にはmakeがスクリプトを中断できるようになっています。「優れている」ほうではそれほど実行に負荷がからない（mvコマンドはシェル経由する必要がなく直接実行されます）うえに、読みやすくエラーが起きた際には多くの情報が表示されることになります（なぜならmakeはどのコマンドが失敗したかを示すからです）。

これはcdに関するよくある問題とは異なっているという点に留意しなければなりません。

```
TAGS:
        cd src && \
        ctags --recurse
```

この場合、2つのコマンドは1つのサブシェルで実行しなければなりません。そのため2つのコマンドは;や&&といった連結用の文字で区切らなければなりません。

ターゲットファイルの削除と保存

エラーが発生するとmakeはターゲットが再構築できなかったと判断します。そのターゲットを必須項目として持つ別のターゲットも同様に再構築できないため、makeはそのターゲットに関するコマンドスクリプトを実行せず再構築を続行しません。--keep-goingオプション（-kオプション）が指定されていた場合は次のターゲットの構築を試みますが、指定されていなければmakeは終了します。作業中のターゲットがファイルであった場合、コマンドが作業を完了せずに終了してしまうと、そのファイルは不完全な状態になっているかもしれません。残念ながら古いmakeとの互換性のため、makeはこういった不完全かもしれないファイルを残しておきます。ファイルのタイムスタンプは更新されているので、もう一度makeを実行しても正しい状態に作り直すことはしません。この問題を回避してエラーが起きた際には問題のありそうなファイルを削除するには、ターゲットを.DELETE_ON_ERRORの必須項目として指定します。.DELETE_ON_ERRORに必須項目が指定されていなければ、すべてのターゲット構築においてエラーが発生した際に、そのターゲットを削除します。

[Ctrl]-[C]のようなシグナルによりmakeが中断した場合にも似たような問題が起きます。作業中のターゲットファイルが更新されていた場合、makeはそのファイルを削除します。状況によってはファイルが削除されると困る場合があります。そのファイルはおそらく作成するのが大変で、全然ないよりは中途半端でも存在していたほうがよいか、もしかしたら構築の作業が進行するために存在していなければならないものなのかもしれません。このような場合にはターゲットを.PRECIOUSの必須項目に指定することで、ファイルの削除から保護することができます。

5.2 空のコマンド

空のコマンドは何もしません。

```
header.h: ;
```

必須項目のリストの後ろにはセミコロン（;）に続いてコマンドが置けることを思い出してください†。ここではセミコロンの後に何もないので、すなわちコマンドが存在しないことを示しています。代わりにタブだけの行をターゲットの次に置いても同じですが、タブだけの行は見えないのでわけがわからなくなってしまいます。空のコマンドはターゲットが意図しないパターンルールと合致してコマンドが実行されてしまうのを防ぐためによく使われます。

他のmakeでは空のターゲットは擬似ターゲットとして使われることがあります。GNU makeではその代わりに.PHONY特殊ターゲットを使います。そのほうが安全で明快でもあります。

5.3 コマンド環境

makeが実行するコマンドは、make自身の実行環境を受け継ぎます。この実行環境には、カレントディレクトリ、ファイルディスクリプタ、makeから渡される環境変数などが含まれます。

サブシェルが起動されるときに、makeはいくつかの環境変数を付加します。

```
MAKEFLAGS
MFLAGS
MAKELEVEL
```

MAKEFLAGSにはmakeに渡されたコマンドラインオプションが入っています。MFLAGSはMAKEFLAGSのコピーであり、これは歴史的な理由により用意されています。MAKELEVELには入れ子になったmakeの呼び出し回数が入っています。つまりmakeが再帰的にmakeを呼び出す際にはMAKELEVELの値を1つ増やします。最初の親makeから呼び出されたサブプロセスではMAKELEVELの値が1になります。通常この値は再帰的makeを管理するために使われます。詳しくは「6.1 再帰的make」で説明します。

もちろん使用者が独自にサブプロセスの環境に渡したい変数を加えることもexport命令を使えば可能です。

実行されるコマンドのカレントディレクトリは、呼び出し側makeのカレントディレクトリと同じです。普通これはmakeが実行されるディレクトリと同じになりますが、--directory=directoryオプション（-Cオプション）を使えば変更できます。--fileオプションを使って別のmakefileを指定する場合は、makefileだけが変更されカレントディレクトリは変わらないという点に留意する必

† 訳注：「3.8 標準的なmake変数」のMAKECMDGOALS変数使用例の中で出てきました。

要があります。

makeが起動するサブプロセスは3つの標準ファイルディスクリプタ、stdin（標準入力）、stdout（標準出力）、stderr（標準エラー出力）を引き継ぎます。コマンドスクリプトがstdinから読み込めることを除いて特に注目すべき点はありません。この動作は妥当であり、よく機能します。スクリプトの読み込みが終わると、残りのコマンドは期待したとおりに実行されます。しかし、makefileの動作では通常こういった処理を期待しません。利用者はmakeを実行すると、処理の進行には特に注意を払わなくても後で結果をチェックするだけでよいということを期待します。さらにいうとstdinの読み込みは、cronを使った自動構築処理に対してよくない相互作用をもたらしてしまいます。

よくあるエラーにstdinを偶然読むことになってしまうというものがあります。

```
$(DATA_FILE): $(RAW_DATA)
        grep pattern $(RAW_DATA_FILES) > $@
```

grepへの入力となるファイルは変数として（そしてこの例ではつづりが間違っている）与えられています。変数は何にも展開されないので、grepはstdinの読み込み状態のまま放置され、プロンプトもmakeが動作を停止している理由も何も表示されません。この問題に対する簡単な回答は、常に/dev/nullを追加のファイルとしてコマンドラインに指定することです。

```
$(DATA_FILE): $(RAW_DATA)
        grep pattern $(RAW_DATA_FILES) /dev/null > $@
```

このgrepコマンドがstdinを読み込むことはありません。もちろんこのような手法に頼らずにmakefileをデバッグしてもかまいません。

5.4　コマンドを評価する

コマンドスクリプトは4つの段階を通して処理されます、コードの読み込み、変数の展開、make式の評価、コマンドの実行です。これらの処理がどのようにして複雑なコマンドスクリプトに対して適用されるのか見ていきましょう。（いささか不自然ではありますが）次のmakefileで考えて見ます。プログラムをリンクした後、状況に応じてシンボルを取り除きupx実行ファイル圧縮ツール† で圧縮します。

```
# $(call strip-program, file)
define strip-program
  strip $1
endef
complex_script:
```

† 訳注：upxはDOS、Windows、Linuxなどの実行ファイルを圧縮してファイルサイズを減少させるフリーソフトウェアです。

http://upx.sourceforge.net/

```
        $(CC) $^ -o $@
    ifdef STRIP
        $(call strip-program, $@)
    endif
        $(if $(PACK), upx --best $@)
        $(warning Final size: $(shell ls -s $@))
```

コマンドスクリプトは実行されるときに評価されますが、ifdef命令は読み込まれたときに処理されます。そのためmakeはifdef　STRIPを読み込むまでは内容に関与せず各行を単に保存します。チェックを行い、もしSTRIPが定義されていなければ、対応するendifまでを読み捨てます。そしてmakeは引き続き残りのスクリプトを読み込み、保存します。

コマンドスクリプトが実行される際に、makeはまずスクリプトを走査して展開および評価する必要のある要素を探します。マクロが展開される際には、各行の先頭にタブが前置されます。展開や評価はコマンドが実行される前に行われるということを理解しておかないと、思いもよらない実行順序に見えてしまうことがあります。ここの例では、スクリプト最後の行には誤りがあります。shellおよびwarning関数はプログラムをリンクする前に実行されるため、チェックするファイルが更新される前にlsコマンドが実行されてしまいます。これが「5.1　コマンドの構文解析」で目にした、順序どおりに実行されないコマンドの理由です。

さらに、ifdef　STRIP行はファイルを読み込んだときに評価されますが、$(if...)の行はcomplex script中のコマンドが実行される直前に評価される点に注目しましょう。つまりif関数のほうがより柔軟であるといえます。というのも変数がいつ定義されるかで制御する機会をより多く持てるからです。しかし、コマンドの大きなブロックを制御するにはあまり向いていません。

この例が示すように、どのプログラム（例えばmakeかシェルか）が式を評価するか、それがいつ行われるのかに対して常に注意を払うのは重要なことです。

```
$(LINK.c) $(shell find $(if $(ALL),$(wildcard core ext*),core) -name '*.o')
```

この複雑なコマンドは一連のオブジェクトファイルをリンクするためのものです。評価の順番と処理を実行するプログラム（括弧の中）は次のようになります。

1. $(ALL)の展開（make）
2. ifの評価（make）
3. wildcardの評価、ここではALLが空でなかったと仮定します（make）
4. shellの評価（make）
5. findの実行（sh）
6. makeの要素をすべて展開し評価し終えた後で、linkコマンドを実行（sh）

5.5 コマンドラインの制限

大きなプロジェクトで作業していると、makeが実行するコマンド行の長さが制限を上回ることがまれに発生します。コマンド行長の制限はOSにより異なります。Red Hat 9 GNU/Linuxでは128K文字ですが、Windows XPでは32Kです。エラー発生時のメッセージもさまざまです。Windowsの上でCygwinを使っている場合には次（「無効な引数です」の意）のようになります。

```
C:\usr\cygwin\bin\bash: /usr/bin/ls: Invalid argument
```

lsが長すぎる引数リストを受け取ると、Red Hat 9では次のメッセージ（「引数リストが長すぎます」の意）がでます。

```
/bin/ls: argument list too long
```

32Kもあるとコマンド行長としては十分に思えますが、100のサブディレクトリに分散した3,000のファイルが存在するプロジェクトのすべてを操作しようとすれば、この制限に抵触することになるでしょう。

この状態を体験するには基本的に2つの方法があります。1つは簡単な値をシェルに展開させる方法、もう1つは非常に長い値を持つ変数をmakeに作らせる方法です。例えばすべてのソースを1つのコマンドラインでコンパイルしたい場合を考えましょう。

```
compile_all:
        $(JAVAC) $(wildcard $(addsuffix /*.java,$(source_dirs)))
```

source_dir変数には数百の単語が入っているとします。しかしJavaファイルを示すワイルドカードを加え、wildcard関数により展開させると、このリストの長さはシステムのコマンドライン長制限を軽く越えてしまうことでしょう。ちなみにmakeにはこういった制限がありません。仮想メモリが利用できるかぎり、必要な量のデータを好きなだけ作ることができます。

このような状況に陥ったときには、何か古いアドベンチャーゲームで「あなたは曲がりくねって見分けもつかない通路から成る迷路の中にいます（You are in a twisty maze of passages all alike）」といった感じがすることでしょう。例えばxargsは長いコマンドラインをシステム固有の長さまでの引数に分割してくれるので、この問題を解決するのに使えるのではないかと思うことでしょう。

```
compile_all:
        echo $(wildcard $(addsuffix /*.java,$(source_dirs))) | \
        xargs $(JAVAC)
```

残念ながら、単にコマンドライン長の問題をjavacからechoコマンドへ移動しただけです。データをファイルに書くためにechoやprintfコマンドを使うこともできません（コンパイラはソースファイルのリストをファイルから読み込むことができないと想定しています）。

この状況を解決するには最初にすべてのファイルリストを1回で作るのではなく、シェルを使ってディレクトリごとにファイルを展開します。

```
compile_all:
        for d in $(source_dirs); \
        do                       \
            $(JAVAC) $$d/*.java; \
        done
```

コンパイラの実行回数を減らすために、ファイルリストをxargsにパイプ経由で渡すという方法も使えます。

```
compile_all:
        for d in $(source_dirs); \
        do                       \
            echo $$d/*.java;     \
        done | \
        xargs $(JAVAC)
```

不幸にもこれらの方法ではコンパイルのエラーを扱うことができません。もしコンパイラがファイルから引数を読むことができるなら、改善された方法としてすべてのファイルリストをファイルに保存し、コンパイラにそのファイルを渡すという方法が使えます。そしてJavaのコンパイラはこの機能を持っています。

```
compile_all: $(FILE_LIST)
        $(JAVA) @$<

.INTERMEDIATE: $(FILE_LIST)
$(FILE_LIST):
        for d in $(source_dirs); \
        do                       \
            echo $$d/*.java;     \
        done > $@
```

forループ内のちょっとしたエラーに注意する必要があります。もしどこかのディレクトリにJavaファイルが存在していなかった場合、ファイルリストには*.javaという文字列が含まれることになりJavaコンパイラは“File not found”（ファイルがみつかりません）というエラーを出力します。nullglobオプションを設定することで、空のパターン展開をbashが行わないようにすることができます。

```
compile_all: $(FILE_LIST)
        $(JAVA) @$<

.INTERMEDIATE: $(FILE_LIST)
$(FILE_LIST):
        shopt -s nullglob;       \
        for d in $(source_dirs); \
        do                       \
            echo $$d/*.java;     \
        done > $@
```

多くのプロジェクトではファイルのリストを作る必要があります。次に紹介するのはファイルリストを作成するbashスクリプトのマクロです†。最初の引数は移動先のルートディレクトリを指定します。リスト中すべてのファイルはこのルートディレクトリからの相対パスを持ちます。2番目の引数はファイルを検索するディレクトリのリストです。3番目と4番目の引数はファイルサフィックスで、これらは指定してもしなくてもかまわない任意引数です。

```
# $(call collect-names, root-dir, dir-list, suffix1-opt, suffix2-opt)
define collect-names
  echo Making $@ from directory list...
  cd $1;                                                    \
  shopt -s nullglob;                                        \
  for f in $(foreach file,$2,'$(file)'); do                 \
    files=( $$f$(if $3,/*.{$3$(if $4,$(comma)$4)}) );       \
    if (( $${#files[@]} > 0 ));                             \
    then                                                    \
      printf '"%s"\n' $${files[@]};                         \
    else :; fi;                                             \
  done
endef
```

次の例は画像ファイルのリストを作成するパターンルールです。

```
%.images:
        @$(call collect-names,$(SOURCE_DIR),$^,gif,jpeg) > $@
```

このスクリプトは長い上に中のコードの一部を再利用する理由もほとんどないので、このマクロの

---

† 訳注：Cygwin bashではブレース展開の動作が多少異なるため、第4引数の指定は任意ですが、Unix環境では第3引数を指定した場合には第4引数も必須となります。これは、例えば次のマクロで解決できます。

```
# $(call collect-names, root-dir, dir-list, suffix1-opt, suffix2-opt)
define collect-names
  echo Making $@ from directory list...
  cd $1;                                                        \
  shopt -s nullglob;                                            \
  for f in $(foreach file,$2,'$(file)'); do                     \
    files=( $$f$(strip                                          \
                $(if $3,                                        \
                  $(if $(filter 1, $(words $3 $4)),             \
                    /*.$3,                                     \
                    /*.{$3,$4}))) );                           \
    if (( $${#files[@]} > 0 ));                                 \
    then                                                        \
      printf '"%s"\n' $${files[@]};                             \
    else :; fi;                                                 \
  done
endef
```

「Cygwin bashではブレース展開の動作が多少異なる」を少し説明しておきます。bashのブレース展開は、ブレース（{}）の中に少なくとも1つのカンマが必要ですが、Cygwinはなくてもかまわないようです。そのため、例えばa{b}ブレース展開の正しい形式ではないため、そのままa{b}となりますが、Cygwin bashはこれをブレース展開として解釈し、abと展開します。

実行は表示しません。ディレクトリのリストは必須項目として指定されます。ルートディレクトリに移動した後、nullglobオプションを設定します。残りは検索するディレクトリを個々に処理するためのforループです。forループが処理するのは、$2引数として渡された単語列です。ここでは単語を引用符で囲みます。というのも、ファイル名の中にはシェルの特殊文字を含むものがあるかもしれないからです。特にJavaのような言語では$記号が入る可能性があります。

```
for f in $(foreach file,$2,'$(file)'); do
```

ディレクトリ内の検索結果として、展開されたファイル名がfiles配列に格納されます†。filesに何か要素が入っていれば、printfを使い単語ごとに改行で区切って出力します。配列を使うことで、空白を含むパス名を適切に処理することが可能となります。printfコマンドでファイル名を二重引用符で囲んでいるのも同じ理由からです。ファイルリストは次の行により生成されます。

```
files=( $$f$(if $3,/*.{$3$(if $4,$(comma)$4)}) );
```

$$fはマクロにディレクトリまたはファイルの引数として渡されたものになります。それに続く式では3番目の引数が指定されているかをチェックしています。任意引数はこのように処理します。3番目の引数が指定されていない場合には、4番目も同様だと判断します。その場合利用者が指定したファイルはそのままファイルリストに入ります。これによりワイルドカードで指定するのが難しいファイルのリストも作ることができます。3番目の引数が指定されると、if関数により/*.{$3}が追加されます。4番目の引数が指定されていればこの$3の後に,$4が追加されます。ワイルドカードのパターンにカンマを入れるために使わざるをえない、ちょっとした技に注目しましょう。カンマをmakeの変数に入れることで、構文解析を通過させることができます。そうしないとカンマがif関数のthen部とelse部の区切りとして解釈されてしまいます。comma変数の定義は単純です。

```
comma := ,
```

これに先立つforループでもコマンドライン長の制限を考慮なければなりません。というのもワイルドカードの展開が行われるからです。ここでの違いは、展開するのが1つのディレクトリの中だけなので制限を超える可能性が低いという点です。

makeの変数にファイル名の長いリストが入っているとして、それをどうすればよいのでしょうか。ここでも面倒な問題に直面します。サブシェルに非常に長い値を渡す方法は、筆者の知るところ2つしかありません。最初の方法は変数の内容をふるいにかけて、その一部だけをサブシェルに渡します。

```
compile_all:
        $(JAVAC) $(wordlist 1, 499, $(all-source-files))
        $(JAVAC) $(wordlist 500, 999, $(all-source-files))
        $(JAVAC) $(wordlist 1000, 1499, $(all-source-files))
```

filter関数も同様に使うことができますが、選択されるファイルの数が指定するパターンの配分

† 訳注：bashでは1次元の配列を使うことができます。()で囲んだ単語列は、配列として格納されます。

により変化するため、どのように実行されるか予測できません。

```
compile_all:
    $(JAVAC) $(filter a%, $(all-source-files))
    $(JAVAC) $(filter b%, $(all-source-files))
```

ファイル名の持つ特徴をパターンに使うという方法もあるでしょう。

この作業を自動化するのは困難である点に留意しなければなりません。foreachループでアルファベット順に処理する方法が考えられますが、次のものはうまく働きません。

```
compile_all:
        $(foreach l,a b c d e ...,                  \
          $(if $(filter $l%, $(all-source-files)), \
            $(JAVAC) $(filter $l%, $(all-source-files));))
```

makeは1つの行に展開してしまうので、行の長さの問題が再燃してしまいます。かわりにevalを使います。

```
compile_all:
        $(foreach l,a b c d e ...,                  \
          $(if $(filter $l%, $(all-source-files)), \
            $(eval \
              $(shell \
                $(JAVAC) $(filter $l%, $(all-source-files));))))
```

evalによりshell関数は直ちに実行され展開されることがないので、これはうまく働きます。つまりforeachループは展開されません。しかしこの段階ではエラーの報告は意味を持たないので、コンパイルエラーがmakeにより正当に扱われないという点が問題となります。

wordlistを使った方法はこれよりも劣っています。数値を扱う機能の制限からwordlist手法をループの中に入れることができません。膨大な量のファイルを満足に扱う方法はほとんど存在しません。

6章
大きなプロジェクトの管理

大きなプロジェクトとはどの程度のものを指すのでしょうか？ 多数の開発者がいて、開発するプログラムは複数アーキテクチャ上で動作し、保守が必要な複数のリリースが存在するという程度をここでは意図しています。もちろんこれらすべてがそろっていないと、大きなプロジェクトといえないわけではありません。1つのアーキテクチャ用にテスト出荷されたC++のソースが何百万行もの大きさを持つようなプロジェクトもやはり大きいといえるでしょう。しかしソフトウェアが永遠にテスト出荷版にとどまっていることはめったにありません。もしそのソフトウェアが成功を収めたとすれば、別のプラットフォーム用にも提供することになるでしょう。結局大きなソフトウェアシステムはどれも同じような道筋をたどることになります。

個別のプログラムやライブラリといった主要なコンポーネントに分割することで、大きなソフトウェアプロジェクトは単純化されます。そういったコンポーネントは、それぞれ個別のディレクトリに格納され、専用の`makefile`によって管理されます。システム全体を構築するには、個々のコンポーネント用`makefile`を適切な順番で呼び出すトップレベルの`makefile`を使うという方法があります。この手法はトップレベルの`makefile`がそれぞれのコンポーネントに対して`make`を再帰的に実行するため再帰的`make`と呼ばれます。再帰的`make`はコンポーネント構築に対する標準的な手法です。1998年にPeter Millerは、それぞれのコンポーネントのディレクトリから必要な情報を収集する1つの`makefile`を使うという手法を提案し、再帰的`make`の持つ問題を解決しています†。

コンポーネント構築に目鼻がつくと、次には構築管理に関する更なる問題が存在していることに気づきます。問題には、複数のバージョン、異なるプラットフォームのサポート、ソースやバイナリの効率的な管理、構築の自動化などが含まれます。これらの問題は、本章の後半で取り上げます。

† Miller, P.A., Recursive Make Considered Harmful (再帰的makeは有害である), AUUGN Journal of AUUG Inc., 19(1), pp. 14–25 (1998)。http://aegis.sourceforge.net/auug97.pdf で読むことができます。

6.1　再帰的make

再帰的makeを使うに至る動機はとても単純です。makeは1つ（または少数）のディレクトリでは非常にうまく働きますが、ディレクトリの数が増えるととても複雑になってしまいます。そこで、単純かつ自己完結したmakefileをそれぞれのディレクトリに用意し、それらを個別に実行することでmakeを大きなプロジェクトで使えるようにします。個々のmakeの実行にスクリプト言語を使うこともできますが、より高いレベルから見るとそれぞれのmakeにも依存関係が存在するため、make自身を使うほうが効果的です。

例えばここに、MP3プレーヤアプリケーションがあるとしましょう。論理的には、ユーザインタフェース、コーデック、データベース管理などいくつかのコンポーネントに分けることができます。それらはlibui.a、libcodec.a、libdb.aといったライブラリとして存在するものとします。アプリケーションはそれらを組み合わせて作られます。これらコンポーネントをファイルシステムに当てはめると、直感的には図6–1のようになるでしょう。

伝統的な構造だと、アプリケーションの主要機能と組み合わせのためのコードはapp/playerサブディレクトリではなくトップレベルディレクトリに置くことになります。トップレベルのディレクトリ

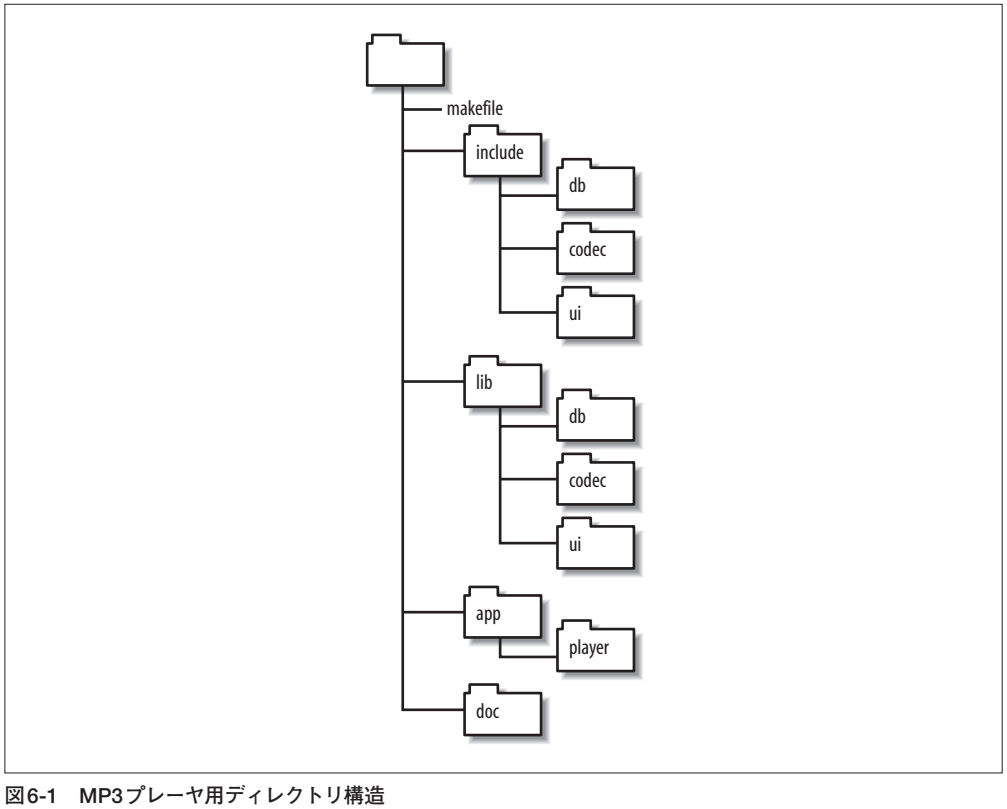

図6-1　MP3プレーヤ用ディレクトリ構造

が単純になることに加え機能が追加しやすくなるので、筆者はアプリケーションのコード自体を個別のディレクトリに置くほうを好みます。例えば後で全曲集を作るアプリケーションを追加する際には、app/catalogに置くとぴったりはまります。

lib/db、lib/codec、lib/ui、app/playerにそれぞれmakefileがあるなら、それらを実行するのがトップレベルmakefileの役目になります。

```
lib_codec := lib/codec
lib_db := lib/db
lib_ui := lib/ui
libraries := $(lib_ui) $(lib_db) $(lib_codec)
player := app/player

.PHONY: all $(player) $(libraries)
all: $(player)

$(player) $(libraries):
        $(MAKE) --directory=$@

$(player): $(libraries)
$(lib_ui): $(lib_db) $(lib_codec)
```

ターゲットがサブディレクトリのリストであり、コマンドがmakeの実行であるルールを通して、トップレベルmakefileはそれぞれのサブディレクトリでmakeを実行することになります。

```
$(player) $(libraries):
        $(MAKE) --directory=$@
```

MAKE変数は、makefileからmakeを実行する際に使われます。そのMAKE変数はmakeにより認識され、再帰的実行で同じmakeが使われるようにパスも含めた値が設定されます。コマンドラインオプション--touch（-t）、--just-print（-n）、--question（-q）が指定されていると、MAKE変数が使われている行は、特殊な扱いを受けます。この件については、「6.1.1　コマンドラインオプション」で詳しく扱います。

ターゲットが最新であってもルールが適用されるように、ターゲットディレクトリは.PHONYとしてマークされています。また、makeがmakefileを読み込む前にターゲットディレクトリに移動するよう--directoryオプション（-Cオプション）が使われています。

このルールはちょっと複雑ですが、下記の単純なコマンドスクリプトが引き起こすいくつかの問題を避けられます。

```
all:
        for d in $(player) $(libraries); \
        do \
          $(MAKE) --directory=$$d; \
        done
```

このコマンドスクリプトは呼び出し元のmakeにエラーを適切に返すことができません。またこの方

法では複数のサブディレクトリで並列にmakeを実行することもできません。この機能については10章で扱います。

makeが依存関係を処理しようと計画を立てるときに、1つのターゲットが依存する必須項目はそれぞれに関連がないことが判明します。さらにお互いに依存関係を持たないターゲットどうしもそれぞれに独立した存在です。例えばここで作られるライブラリは、本質的にいえばapp/playerターゲットとは関連していませんし、ライブラリどうしも関連がありません。このことはmakeがライブラリを構築する前にapp/playerのmakefileを実行してもかまわないということを意味しています。しかしアプリケーションのリンク時にはライブラリが必要なので、これでは明らかに構築が失敗してしまいます。この問題を解決するために依存関係情報を追加しています。

```
$(player): $(libraries)
$(lib_ui): $(lib_db) $(lib_codec)
```

これによりライブラリのサブディレクトリにあるmakefileはplayerディレクトリにあるmakefileよりも先に実行されなければならないことをmakeに伝えます。同様にlib/uiのコードはコンパイルする際にlib/dbとlib/codecのライブラリを必要としていることを知らせます。これで生成されるべきコード（例えばyaccやlexのファイル）がuiコードをコンパイルする前に生成されることを保証します。

必須項目を更新する際には順序に関するさらに微妙な問題が存在します。他の依存関係と同様に更新順序は依存関係グラフの分析で決まりますが、複数の必須項目が同じ行に書かれていた場合、GNU makeはそれらを左から右に更新します。例を使って説明しましょう。

```
all: a b c
all: d e f
```

考慮すべき他の依存関係がなければ、この6つの必須項目はどのような順番で更新されてもかまいません（例えば“d b a c e f”という順番）。しかしGNU makeは同じ行の中なら左から右という順序づけを行うので、更新順序は“a b c d e f”か“d e f a b c”になります。この順番は実装時の偶然によりますが、実行順は正しいように見えます。こうなると正しい順番になっているのは単なる幸運であることや、完全な依存関係を定義しなければならないことなどは忘れがちです。依存関係分析結果が最終的にこれまでと異なる順序を生成した場合には問題が表面化してしまいます。一連のターゲットを決まった順序で更新しなければならない場合には、適切な必須項目に対して適切な順序で実行されるようにする必要があります。

トップレベルmakefileを実行すると次の結果を得ます。

```
$ make
make --directory=lib/db
make[1]: Entering directory `/test/book/out/ch06-simple/lib/db'
Update db library...
make[1]: Leaving directory `/test/book/out/ch06-simple/lib/db'
make --directory=lib/codec
make[1]: Entering directory `/test/book/out/ch06-simple/lib/codec'
```

```
Update codec library...
make[1]: Leaving directory `/test/book/out/ch06-simple/lib/codec'
make --directory=lib/ui
make[1]: Entering directory `/test/book/out/ch06-simple/lib/ui'
Update ui library...
make[1]: Leaving directory `/test/book/out/ch06-simple/lib/ui'
make --directory=app/player
make[1]: Entering directory `/test/book/out/ch06-simple/app/player'
Update player application...
make[1]: Leaving directory `/test/book/out/ch06-simple/app/player'
```

再帰的にmakeを実行していることをmake自身が検知すると、Entering directoryとLeaving directoryメッセージを表示するために--printdirectoryオプション（-wオプション）が有効になります。このオプションは--directoryオプション（-Cオプション）が使われた際にも有効になります。各行には角括弧に囲まれてMAKELEVEL変数の値も表示されます。この例では、各コンポーネント用のmakefileにてコンポーネントを更新していることを示す簡単なメッセージを表示しています。

6.1.1 コマンドラインオプション

再帰的makeは単純なアイディアですが、すぐに複雑化してしまいます。理想的な再帰的makeなら、システム内すべてのmakefileがあたかも1つのmakefileであるかのようにふるまうことでしょう。このレベルの連携を達成するのは現実的には困難なので、妥協が必要です。この複雑な問題はコマンドラインオプションがどのように扱われなければならないかを見ることで、より明らかになります。

MP3プレーヤのヘッダファイルにコメントを加えたとしましょう。変更を加えたヘッダファイルに依存しているソースをすべて再コンパイルする代わりに、make --touchを使ってファイルのタイムスタンプを最新にすればよいことに気づくでしょう。トップレベルmakefileに対してmake --touchを実行することで、下位のmakeで管理されているすべてのファイルに対してタイムスタンプの更新を行いたいところです。これがどのように働くのかを見てみましょう。

--touchオプションが付いていると、通常のルール処理は行われません。代わりに依存関係グラフが走査され、指定されたターゲットとその必須項目の中で.PHONYとマークされていないもののタイムスタンプを更新することで最新の状態にします。例のmakefileではサブディレクトリが.PHONYになっているので、通常は無視されます（普通のファイルと同じようにタイムスタンプを更新しても意味がありませんし）。しかしこれらのターゲットが無視されると困ります。そのコマンドスクリプトを実行したいのです。makeは自動的にMAKEを含む行に+修飾子を付加し、--touchオプションの有無にかかわらず下位のmakeを実行します。

makeが下位のmakeを実行する際には、下位の作業でも使うために--touchフラグを付加します。これはMAKEFLAGS変数を通して行われます。makeが開始するときにほとんどのコマンドラインオプションをMAKEFLAGS変数に付加します。--directory（-C）、--file（-f）、--old-file（-o）、--new-file（-W）オプションは例外です。MAKEFLAGS変数は環境変数として公開され下位のmakeが開始するときに読み込まれます。

この特殊な機能により、下位のmakeはほぼ思ったとおりにふるまいます。$(MAKE)による再帰的な実行における--touchオプション（-tオプション）に対する特殊な動作は--just-print（-nオプション）と--question（-qオプション）でも適用されます。

6.1.2　値を渡す

すでに触れたように下位のmakeには環境変数を通して値が渡され、exportとunexport命令により制御されます。環境変数として渡された値はデフォルトの値として扱われますが、代入により上書きされてしまいます。--environment-overridesオプション（-eオプション）を使うことで代入された値を環境変数で上書きすることができます。override命令を使えば、（たとえ--environment-overridesオプションが使われていたとしても）特定の代入に対して明示的に環境変数の値を上書きするように指示できます。

```
override TMPDIR = ~/tmp
```

コマンドラインで定義された変数は、シェルの文法上誤っていたとしても、自動的に環境変数として公開されます。変数名が文字、数字、下線のみで構成されている場合には合法であると認識されます。コマンドライン上の変数定義は、コマンドラインオプションとともにMAKEFLAGS変数に格納されます。

6.1.3　エラー処理

再帰的makeがエラーになったときには、いったい何が起きているのでしょうか。実際には、これはよくあることです。makeがエラーの終了ステータスを受け取ると処理を中断して終了ステータス2を返します。呼び出し側のmakeも引き続いて終了し、このエラー状況は再帰的makeの処理全体に伝播します。もし--keep-goingオプション（-kオプション）がトップレベルのmakeに指定されていると、普通それは下位のmakeにも渡されます。そこで下位のmakeは通常行われていることを行います。つまりエラーを起こした現在のターゲットを放棄し、そのターゲットを必須項目としていない別のターゲットの処理を続行します。

例えば、MP3プレーヤのlib/dbの構築を行っている最中にコンパイルエラーが発生すると、そこでlib/dbの構築は中断され、呼び出し側のmakefileに対して終了ステータス2が返ります。もし--keep-goingオプション（-kオプション）が使われていた場合、呼び出し側makeは次にlib/dbとは直接関係のないlib/codecの構築作業を行います。その構築作業がどのような終了ステータスで完了したとしても、makeは最終的に終了ステータス2で終了します。なぜならlib/dbの処理がエラーとなったので、それ以上の処理を続行することができないからです。

--questionオプション（-qオプション）は非常に似た働きを持っています。このオプションにより、makeは最新ではないターゲットを見つけたときに1を、すべて最新であった場合に0を返します。再帰的makeに適用すると、プロジェクト全体が最新になっているかを再帰的に調べます。最新ではないファイルが見つかると、そこでmakeは処理を中断し呼び出し側へと戻ります。

6.1.4 その他のターゲットの構築

構築すべきターゲットは、どのような構築システムにおいても必要になるものですが、一方でclean、install、printなどといった補助ターゲットも同様に不可欠なものです。これらは.PHONYターゲットであるため、これまでに説明した手法ではうまく働きません。

例えば、次のような中途半端な方法がいくつかあります。

```
clean: $(player) $(libraries)
        $(MAKE) --directory=$@ clean
```

または

```
$(player) $(libraries):
        $(MAKE) --directory=$@ clean
```

最初の例が働かないのは、必須項目が指定されていることによりcleanターゲットの作業を行う前に$(player)や$(libraries)のmakefileのデフォルトターゲットの構築が始まってしまうからです。2番目の例はコマンドスクリプトを伴う同じターゲットが存在しているのでエラーとなります。

うまく働く方法の1つとしてシェルのforループを使うという方法があります。

```
clean:
        for d in $(player) $(libraries); \
        do                               \
          $(MAKE) --directory=$$d clean; \
        done
```

forループはすでに説明した理由により適切ではありませんが、解決への糸口を（前出のエラーとなる例とともに）提供してくれます。

```
$(player) $(libraries):
        $(MAKE) --directory=$@ $(TARGET)
```

再帰的make呼び出しの行に$(TARGET)変数を追加し、makeのコマンドラインでTARGET変数を定義することにより、下位のmakeに対する任意のターゲットを加えることができます。

```
$ make TARGET=clean
```

残念なことにこの方法では、トップレベルmakefile自体が持つ$(TARGET)の構築処理を呼び出すことができません。トップレベルのmakefileでは通常何も行わないので、たいていは大丈夫なのですが、どうしても必要ということであれば、ifで囲った別のmake呼び出しを加えることになります。

```
$(player) $(libraries):
        $(MAKE) --directory=$@ $(TARGET)
        $(if $(TARGET), $(MAKE) $(TARGET))
```

これでコマンドラインでTARGETに指定することによりclean（またはその他のターゲット）を呼び出すことができるようになりました。

6.1.5　makefileの相互依存

コマンドラインオプションや環境変数を通したやり取りに対する特殊な扱いは、再帰的makeを機能させるべく調整されていることを示しています。

再帰的makeの$(MAKE)コマンドは、分割されたmakefileの表層的な関連を記述しているにすぎません。残念ながらしばしば微妙な依存関係がディレクトリ間に埋もれていることがあります。

例えばdbのモジュールでは音楽データを読み込むためにyaccによる構文解析器を作成しているとしましょう。もしuiのモジュールであるui.cが、生成されたyaccのヘッダファイルを必要としていた場合、これらのモジュールには依存関係が成立します。依存関係が適切に記述されていれば、yaccの生成するヘッダファイルが更新された際にはmakeはuiのモジュールも再コンパイルするでしょう。以前説明した依存関係を自動生成する手法を使えば、これは難しくありません。しかしもしyaccのソースファイル自体が更新されていたらどうなるでしょうか？ この場合uiのmakefileが実行される際、ui.cをコンパイルする前にyaccを実行し構文解析器とヘッダファイルを生成するのが正しいありかたです。例の再帰的makeによる処理分割では、これを行うことができません。なぜならyaccを実行するのはuiのmakefileではなくdbのmakefileだからです。

最善の方法は、uiのmakefileが実行される前に必ずdbのmakefileが実行されるようにすることです。この上位レベルの依存関係は手作業で記述しなければなりません。例題のmakefileなら、こういった関係が記述されていることを認識することが可能ですが、通常これは大変難しい保守上の問題となります。修正が行われるうちにトップレベルのmakefileはモジュール間の依存関係を適切に記述できなくなってしまうでしょう。

例を続けましょう。dbに存在するyaccの文法ファイルが更新され、dbのmakefileが実行される前にuiのmakefileが（トップレベルmakefileから呼ばれるのではなく直接）実行されてしまうと、dbのmakefile中に更新すべき依存関係が存在しyaccを実行してヘッダファイルを更新しなければならないことをuiのmakefileは知るすべがありません。そのためuiのmakefileは古いyacc生成のヘッダファイルを使ってコンパイルを実行してしまいます。もし新しいシンボルが追加されていてプログラムの中から参照していた場合にはコンパイルエラーが発生します。このように再帰的makeの手法は、本質的に1つのmakefileを使う方法よりも脆弱なのです。

コード生成ツールが広範囲に使われると、問題はより深刻になります。RPCのスタブ生成ツールがuiで使われ、dbから参照されていたとしましょう。これにより相互参照に取り組む必要が出てきます。これを解決するにはdbに移動してyaccヘッダを生成し、次にuiに行ってRPCスタブを生成した後、dbでファイルをコンパイルを行い、最後にuiでコンパイルを完了させる必要があります。プロジェクトのソースを生成したりコンパイルするために必要な処理の数は、ソースコード構造や生成に使用するツールの種類により変化します。

現実世界での解決方法は、多くの場合その場しのぎです。すべてのファイルが最新であることを保証するためにトップレベルのmakefileが実行されたときにはすべてのmakefileを実行します。こ

れはまさに例題のMP3プレーヤのmakefileで行っていることに他なりません。トップレベルのmakefileが実行されると、4つの下位makefileが無条件に実行されます。複雑な状況では、最初にコード生成を行い次にコンパイルが行われるようmakefileを複数回実行します。普通このような実行の反復は時間を浪費するものですが、場合によっては必要なこともあります。

6.1.6 コードの重複を回避する

例ではディレクトリごとに3つのライブラリが存在します。これらのライブラリ用makefileは非常に似通っています。最終的に構築されるプログラムに対して、個々のライブラリはそれぞれ異なる目的を持っていますが、構築するためのコマンドは同じなので当然です。こういった機能の分割は大きなプロジェクトではよくあることですが、同時に多くの同じようなmakefileが作成され、多くの（makefileの）コードが重複してしまいます。

コードの重複は避けるべきです。それがmakefileのコードであってもです。コードの重複により保守にかかる費用は上昇し、バグをも増やしてしまいます。同時にアルゴリズムの理解や、ちょっとした違いの認識を難しくしてしまいます。そこで、可能なかぎり例題のmakefileからコードの重複を排除したいと思います。makefileの共通部分を共通のインクルードファイルに移動するというのが最も簡単な実現方法です。

例えば、codecのmakefileは次のようになっています。

```
lib_codec := libcodec.a
sources := codec.c
objects := $(subst .c,.o,$(sources))
dependencies := $(subst .c,.d,$(sources))

include_dirs := .. ../../include
CPPFLAGS += $(addprefix -I ,$(include_dirs))
vpath %.h $(include_dirs)

all: $(lib_codec)

$(lib_codec): $(objects)
        $(AR) $(ARFLAGS) $@ $^

.PHONY: clean
clean:
        $(RM) $(lib_codec) $(objects) $(dependencies)

ifneq "$(MAKECMDGOALS)" "clean"
  include $(dependencies)
endif

%.d: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $< | \
        sed 's,\($*\.o\) *:,\1 $@: ,' > $@.tmp
        mv $@.tmp $@
```

コードのほぼすべてがdbおよびuiのmakefileと重複しています。各ライブラリで異なっているのはライブラリ名自体とライブラリに含まれるファイルのソースリストだけです。重複したコードがcommon.mkに移動したなら、次のようにmakefileの内容を削ぎ落とすことができます。

```
library := libcodec.a
sources := codec.c

include ../../common.mk
```

共通のインクルードファイルに統合された内容を見てみましょう。

```
MV := mv -f
RM := rm -f
SED := sed

objects := $(subst .c,.o,$(sources))
dependencies := $(subst .c,.d,$(sources))
include_dirs := .. ../../include
CPPFLAGS += $(addprefix -I ,$(include_dirs))

vpath %.h $(include_dirs)

.PHONY: library
library: $(library)

$(library): $(objects)
        $(AR) $(ARFLAGS) $@ $^

.PHONY: clean
clean:
        $(RM) $(objects) $(program) $(library) $(dependencies) $(extra_clean)

ifneq "$(MAKECMDGOALS)" "clean"
  -include $(dependencies)
endif

%.c %.h: %.y
        $(YACC.y) --defines $<
        $(MV) y.tab.c $*.c
        $(MV) y.tab.h $*.h

%.d: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $< | \
        $(SED) 's,\($*\.o\) *:,\1 $@: ,' > $@.tmp
        $(MV) $@.tmp $@
```

すべてのライブラリが同じインクルードパスを使うように、ヘッダファイルのパスに手を加えたので、それぞれのmakefileで異なっていたinclude_dirs変数は、すべてのmakefileで同じになりました。

common.mkファイルは、ライブラリ用インクルードファイルのためのデフォルトターゲットも持ち

ます。元のmakefileではデフォルトのターゲットとしてallを使っていました。これはライブラリ構築以外のmakefileで、デフォルトターゲットに異なる種類の必須項目を加える際に問題が生じてしまいます。そこで共通コードではデフォルトターゲットをlibraryとしています。

この共通ファイルはターゲットを持っているので、ライブラリ構築用ではないmakefileではデフォルトターゲットの指定の後でインクルードされる必要があります。また、cleanのコマンドスクリプトはprogram、library、extra_clean変数を参照している点に注意が必要です。ライブラリ構築用makefileではprogram変数は空であり、一方プログラム構築用makefileではlibrary変数は空になっています。extra_clean変数はdbのmakefile用に追加してあります。この変数はyaccが生成したファイルを指定するために使われます。したがって次のようなmakefileになります。

```
library := libdb.a
sources := scanner.c playlist.c
extra_clean := $(sources) playlist.h

.SECONDARY: playlist.c playlist.h scanner.c

include ../../common.mk
```

この手法を使うことによりコードの重複は最小に抑えることができます。多くのコードを共通のmakefileに移動することにより、そのmakefileはプロジェクト全体の統合makefileへと進化します。makeの変数とユーザ定義関数を変更点としてカスタマイズすることで、統合makefileを個々のディレクトリに適合させることが可能となります。

6.2 非再帰的make

複数のディレクトリが関与するプロジェクトは、再帰的makeを使わなくても管理することが可能です。ここでの違いは、makefileにより操作されるファイルが複数のディレクトリに散在しているという点です。これに対応するために、サブディレクトリに存在するファイルへの参照には絶対パスか相対パスを含めなければなりません。

大きなプロジェクトを管理するためのmakefileはそれぞれプロジェクト内のモジュールに対応した多くのターゲットを持ちます。MP3プレーヤの例だと、ライブラリとプログラムそれぞれに対応するターゲットが必要です。モジュールの集合、例えばすべてのライブラリの集合を表すような擬似ターゲットを加えるのも有用です。典型的なデフォルトターゲットは、これらすべてのターゲットを構築するためのものです。デフォルトターゲットは、しばしばドキュメントの構築やテストプログラムの実行なども行います。

最も直接的な非再帰的makeの使い方は、ターゲット、オブジェクトファイルの参照、依存関係をすべて1つのmakefileに入れてしまうというものです。ソースファイルはファイルシステムの中に分散配置されているにもかかわらず、それらの情報は1つのファイルに集約してしまうため、この方法は再帰的makeに慣れた開発者の要求を満たすことができません。Millerの非再帰的makeに関する論文

では、ファイルのリストとモジュール特有のルールが書かれたインクルードファイルを各ディレクトリにそれぞれ用意するという手法が紹介されています。トップレベルのmakefileは下位のmakefileをインクルードします。

例6-1は、各サブディレクトリからモジュール単位のmakefileをインクルードするMP3プレーヤ用のmakefileを示しています。例6-2では、モジュール単位のmakefileの1つを示しています。

例6-1　非再帰的makefile

```
# 各モジュールの情報を、次の4つの変数に収集する
# ここでは、単純変数として初期化する
programs :=
sources :=
libraries :=
extra_clean :=

objects = $(subst .c,.o,$(sources))
dependencies = $(subst .c,.d,$(sources))

include_dirs := lib include
CPPFLAGS += $(addprefix -I ,$(include_dirs))
vpath %.h $(include_dirs)

MV := mv -f
RM := rm -f
SED := sed

all:

include lib/codec/module.mk
include lib/db/module.mk
include lib/ui/module.mk
include app/player/module.mk

.PHONY: all
all: $(programs)

.PHONY: libraries
libraries: $(libraries)

.PHONY: clean
clean:
        $(RM) $(objects) $(programs) $(libraries) \
              $(dependencies) $(extra_clean)

ifneq "$(MAKECMDGOALS)" "clean"
  include $(dependencies)
endif

%.c %.h: %.y
        $(YACC.y) --defines $<
        $(MV) y.tab.c $*.c
```

```
        $(MV) y.tab.h $*.h

%.d: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $< | \
        $(SED) 's,\($(notdir $*)\.o\) *:,$(dir $@)\1 $@: ,' > $@.tmp
        $(MV) $@.tmp $@
```

例6-2 非再帰的makefile向けのlib/codec用インクルードファイル

```
local_dir := lib/codec
local_lib := $(local_dir)/libcodec.a
local_src := $(addprefix $(local_dir)/,codec.c)
local_objs := $(subst .c,.o,$(local_src))

libraries += $(local_lib)
sources += $(local_src)

$(local_lib): $(local_objs)
        $(AR) $(ARFLAGS) $@ $^
```

モジュール固有の情報はモジュールのディレクトリに置いてあるインクルードファイルに入っています。トップレベルmakefileにはモジュールのリストとinclude命令だけが入っています。それではこのmakefileとmodule.mkについて詳しく見ていくことにしましょう。

module.mkはそれぞれ自分のライブラリ名をlibraries変数に追加し、自分のソースファイルをsources変数に追加します。local_変数は定数を保持したり、同じ値を何度も生成しないために使われます。これらの変数は再帰変数ではなく（:=で代入が行われる）単純変数を使います。というのも、複数のmakefileを結合して実行する構築作業にて他のmakefile中の変数が影響を及ぼすことがないようにするためです。ライブラリ名とソースファイルリストは前述したように相対パスで記述されます。最後にインクルードファイルにはそれぞれのライブラリを構築するためのルールが記述されます。ルールにおけるターゲットと必須項目の部分は直ちに評価されるので、local_変数をこのルールの中で使うのにまったく問題はありません。

トップレベルmakefileでは、まず各モジュールのファイル情報を蓄積する変数を定義します。個々のモジュールにてlocal変数を使って値を追加するので、この変数は単純変数にする必要があります。

```
local_src := $(addprefix $(local_dir)/,codec.c)
…
sources += $(local_src)
```

例えばsourcesを再帰変数にしてしまうと、最終的な値は最後のlocal_srcの値が何度も繰り返されたものになってしまいます。何もしないと変数が再帰変数になってしまうため、たとえ空の値の代入であっても明示的な初期化が必要です。

次にオブジェクトファイルのリストであるobjectsと依存関係ファイルのリストであるdependenciesをsources変数から生成しています。これらの変数は再帰変数です。なぜならこの時点ではsources変数が空だからです。インクルードファイルが読み込まれるまで、値を持つことは

ありません。このmakefileでは、これらの変数定義をファイルのインクルードの後に移動し、単純変数に変えてもまったく問題ありません。しかし基本的なファイルのリスト（例えばsources、libraries、objectsなど）を1か所に集めておくことにより、makefileを理解しやすくするよい方法だといえます。また、変数どうしが相互に参照しあうような状況では、再帰変数を使わなければなりません。

その次にはCPPFLAGSを設定することでC言語のインクルードファイルに対処しています。これによりコンパイラはヘッダファイルを見つけることができるようになります。コマンドラインオプションや環境変数や他のmake要素によりCPPFLAGSがすでに設定されているかもしれないので、値を追加しています。vpath命令により、makeが他のディレクトリに存在するヘッダファイルを認識することができます。include_dirs変数はインクルードディレクトリのリストを重複させないために使われます。

mv、rm、sedに対する変数は、直接プログラム名を使ってmakefileにコーディングするのを避けるために使われます。変数名で使う文字について補足しましょう。ここではmakeのマニュアルで推奨されている慣例に沿っています。makefile内だけで使われる変数は小文字を、コマンドライン上で設定される可能性のあるものは大文字を使います。

makefileの次の部分は、さらに興味深くなります。デフォルトターゲットallに関する明示的なルールを定義します。残念ながらallの必須項目はprograms変数です。この変数は直ちに評価されますが、設定されるのはインクルードファイルの中です。したがってallターゲットを定義する前にインクルードファイルを読み込まなければなりません。またまた残念なことに、インクルードファイルの中でもターゲットが存在し、最初のものがデフォルトターゲットとして認識されてしまいます。このジレンマに折り合いをつけるために、まずallターゲットを必須項目なしで指定し、インクルードファイルを読み込み、その後でallに必須項目を設定します。

残りはの部分はこれまでの例ですでにわかっていると思いますが、makeが暗黙ルールを適用する方法は注目に値します。ここではソースファイルがサブディレクトリの中に存在しています。makeが標準の%.o:%.cルールを適用する際に、必須項目はlib/ui/ui.cといった相対パスを持つものになります。makeはこの相対パスをターゲットファイルにも適用し、lib/ui/ui.oの更新を試みます。このようにmakeは魔法のごとく処理を進めます。

最後にもう1つ問題が残っています。makeはパスを正しく扱いますが、makefileから起動されるツールもすべてそうだとはかぎりません。特にgccを使って生成された依存関係ファイルにはオブジェクトファイルに対して相対パスが付きません。つまりgcc -Mの出力は次のようになります。

```
ui.o: lib/ui/ui.c include/ui/ui.h lib/db/playlist.h
```

しかし、本来期待していたのは次のようなものです。

```
lib/ui/ui.o: lib/ui/ui.c include/ui/ui.h lib/db/playlist.h
```

これではヘッダファイル必須項目を扱う手法がうまく働きません。この問題はsedコマンドを使って相対パス情報を付加することで解決できます。

```
$(SED) 's,\($(notdir $*)\.o\) *:,$(dir $@)\1 $@: ,'
```

いろいろなツールに起因する問題を解決するためにmakefileを調整することは、makeを使うために必要な作業の一部です。makeが依存せざるをえない広範にわたるツールが引き起こす予期しない問題のために、ポータビリティのあるmakefileを作ることは多くの場合とても複雑です。

さてこれで適正な非再帰的makefileができましが、まだ保守の問題が残っています。インクルードファイルであるmodule.mkは大部分が同じです。1つのファイルに対する変更は、結局全部修正することにつながります。MP3プレーヤのような小さなプロジェクトであっても、こういった作業は面倒なものです。何百ものインクルードファイルが存在する大きなプロジェクトでは致命的です。一貫した変数名とインクルードファイルの中身の標準化がこのような問題の処方箋です。リファクタリング後のlib/codecにおけるインクルードファイルは次のようになります。

```
local_src := $(wildcard $(subdirectory)/*.c)

$(eval $(call make-library, $(subdirectory)/libcodec.a, $(local_src)))
```

ソースファイルをファイル名で指定する代わりに、ディレクトリ内の.cファイルをすべてコンパイルするという方法があります。make-library関数はインクルードファイルで行われる作業の大部分を実行します。この関数はトップレベルmakefileで次のように定義されます。

```
# $(call make-library, library-name, source-file-list)
define make-library
  libraries += $1
  sources += $2
  $1: $(call source-to-object,$2)
    $(AR) $(ARFLAGS) $$@ $$^
endef
```

この関数はライブラリとソースをそれぞれの変数に追加し、ライブラリを構築するルールを定義します。自動変数がどのように使われるか注目しましょう。2つのドル記号により、$@と$^の評価を実際にルールが発動されるまで遅らせます。source-to-object関数はソースファイルのリストを対応するオブジェクトファイルのリストに変換します。

```
source-to-object = $(subst .c,.o,$(filter %.c,$1)) \
                   $(subst .y,.o,$(filter %.y,$1)) \
                   $(subst .l,.o,$(filter %.l,$1))
```

前例では、構文解析器と字句解析器のソースはplaylist.yとscanner.lであるという事実に目をつぶり、生成された.cファイルをソースであるとしてリストしていました。これにより、生成されたファイルを明示的にリストし格納するextra_cleanという変数の追加が必要になってしまいました。ここでは、この問題をsources変数に.yと.lファイルを直接追加することを許容し、source-to-object関数を変更してそれらも変換できるようにすることで解決します。

source-to-objectの変更に加え、cleanターゲットが適切にクリーンアップ作業を行えるよう

にyaccとlexの出力ファイル名を導出する関数が必要になります。generated-source関数はソースファイルのリストを受け取り、中間ファイル名を結果として返します。

```
# $(call generated-source, source-file-list)
generated-source = $(subst .y,.c,$(filter %.y,$1)) \
                   $(subst .y,.h,$(filter %.y,$1)) \
                   $(subst .l,.c,$(filter %.l,$1))
```

もう1つの補助関数subdirectoryを使えば、local_dirを省くことができます。

```
subdirectory = $(patsubst %/makefile,%, \
                 $(word \
                   $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST)))
```

「4.2.1　文字列関数」で説明したように、現在のmakefileのファイル名をMAKEFILE_LIST変数から取得できます。単純にpatsubstを使いトップレベルmakefileのある位置からの相対パスを取り出します。これによりよけいな変数を取り除きインクルードファイル間の違いを小さくできます。

（少なくともこの例に関しての）最後の最適化は、wildcardを使ってソースファイルリストを作り出すというものです。この方法はソースファイルツリーが常に整頓されているような環境でないとうまく働きません。しかし筆者がこれまで参加したプロジェクトはそういうものではありませんでした。古いコードは「万が一」のためにソースツリー内に残されていました。こういった古くて使われていないコードはディレクトリ内ファイルの文字列検索などで発見され、新しいプログラマ（もしくは、そのモジュールのことを知らない古株プログラマ）がもう使われていないコードをコンパイルやデバッグすることで、プログラマの苦悩と無駄な時間を増大させてしまいます。CVSのような現代的なソースコード管理システムを使えば、（リポジトリの中には残りますが）もう使わないコードをソースファイルツリーの中に置いておく必要はなくなり、wildcardを使う手法も現実的となります。

include命令も最適化できます。

```
modules := lib/codec lib/db lib/ui app/player
...
include $(addsuffix /module.mk,$(modules))
```

さらに大きなプロジェクトではモジュールのリストが数百数千にも膨れ上がるので、これでも保守上の問題となってしまいます。こういった状況下では、modulesをfindコマンドを使って定義するのが望ましいかもしれません。

```
modules := $(subst /module.mk,,$(shell find . -name module.mk))
...
include $(addsuffix /module.mk,$(modules))
```

modules変数がモジュールの汎用リストとして使えるようにファイル名を取り除いています。もちろんその必要がなければsubst関数とaddsuffix関数を省略し単にfind関数の出力をmodules変数に格納するだけでかまいません。例6-3は最終的なmakefileです。

例6-3 非再帰的makefileバージョン2

```
# $(call source-to-object, source-file-list)
source-to-object = $(subst .c,.o,$(filter %.c,$1)) \
                   $(subst .y,.o,$(filter %.y,$1)) \
                   $(subst .l,.o,$(filter %.l,$1))

# $(subdirectory)
subdirectory = $(patsubst %/module.mk,%,                          \
                 $(word                                           \
                   $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST)))

# $(call make-library, library-name, source-file-list)
define make-library
  libraries += $1
  sources   += $2

  $1: $(call source-to-object,$2)
        $(AR) $(ARFLAGS) $$@ $$^
endef

# $(call generated-source, source-file-list)
generated-source = $(subst .y,.c,$(filter %.y,$1))        \
                   $(subst .y,.h,$(filter %.y,$1))        \
                   $(subst .l,.c,$(filter %.l,$1))

# 各モジュールの情報を、次の4つの変数に収集する
# ここでは、単純変数として初期化する
modules      := lib/codec lib/db lib/ui app/player
programs     :=
libraries    :=
sources             :=

objects      =          $(call source-to-object,$(sources))
dependencies =          $(subst .o,.d,$(objects))

include_dirs := lib include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))
vpath %.h $(include_dirs)

MV  := mv -f
RM  := rm -f
SED := sed

all:

include $(addsuffix /module.mk,$(modules))

.PHONY: all
all: $(programs)

.PHONY: libraries
libraries: $(libraries)
```

```
.PHONY: clean
clean:
        $(RM) $(objects) $(programs) $(libraries) $(dependencies)        \
              $(call generated-source, $(sources))

ifneq "$(MAKECMDGOALS)" "clean"
  include $(dependencies)
endif

%.c %.h: %.y
        $(YACC.y) --defines $<
        $(MV) y.tab.c $*.c
        $(MV) y.tab.h $*.h

%.d: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $< | \
        $(SED) 's,\($(notdir $*)\.o\) *:,$(dir $@)\1 $@: ,' > $@.tmp
        $(MV) $@.tmp $@
```

モジュールごとにインクルードファイルを置くことは非常に有効であるだけでなく利点もありますが、そうする価値があるかどうかについては懐疑的です。大きなJavaプロジェクトにおける筆者の経験では、すべてのmodule.mkファイルを効率的に取り込んだ1つのトップレベルmakefileのほうが適切です。そのプロジェクトには997に分割されたモジュール、24のライブラリ、6つのアプリケーションが存在していました。独立した単位ごとにいくつかのmakefileが存在し、それぞれのmakefileはおおよそ2,500行の量がありました。共通のインクルードファイルには大域変数、ユーザ定義関数、パターンルール、その他を含み、これも2,500行程度ありました。

単一のmakefileを使うか、インクルードファイルに情報を分割するか、どちらを選択したとしても非再帰的makeは大きなプロジェクトを構築するための有効な手法です。また同様に、再帰的makeで問題となる点をも解決します。筆者が気にしている唯一の欠点は、再帰的makeに慣れている開発者に発想の転換が必要だということです。

6.3　巨大システムのコンポーネント

もちろん要件というものはプロジェクトや作業環境によりさまざまです。ここでは多くの商用ソフトウェア開発環境で重要だと考えられていることを広く取り上げます。

多くの開発チームが必要とする共通の機能は、ソースコードとバイナリコードの分離です。つまり、コンパイルによって生成されるオブジェクトファイルを、ソースとは別のバイナリツリーに置くべきだというものです。これにより多くの機能を加えることができるようになります。分離されたバイナリツリーには多くの利点があります。

- 大きなバイナリツリーの置き場所を指定することで、ディスク資源を管理しやすくなります。
- 複数のバイナリツリーを同時に管理することができます。例えば、1つのソースツリーに対して最適化バイナリ、デバッグバイナリ、性能評価（profiling）バイナリを持つことができます。

- 複数のプラットフォームを同時にサポートできます。適切に作られたソースツリーなら多くのプラットフォーム向けバイナリを並行に作ることができます。
- 開発者がソースツリーの一部を抜き出して、足りないファイルをソースやバイナリツリーを参照して自動的に取り出す構築システムを作ることができます。これは必ずしもソースとバイナリが分離されていなくてもよいのですが、分離しておかないと開発者の構築システムではバイナリをどこから見つけ出せばよいのかおそらく混乱することになるでしょう。
- ソースツリーを読み出し専用に設定することができます。これにより、構築作業がリポジトリ内のコードを反映していることを保証します。
- もし処理するファイルを探す代わりにツリー全体を一括で扱えるなら、cleanなどのいくつかのターゲットは単純に作ることができます（そして実行速度も劇的に速くなります）。

上にあげた点はそれ自体が重要な構築のための機能であり、プロジェクトが必要とするものとなるでしょう。

プロジェクトのリファレンスを維持することは、しばしば重要な機能となります。通常cronによって、毎晩チェックアウトと構築を行います。この結果として得られるのは、CVSの内容をそのまま反映したソースとバイナリのツリーなので、これを**リファレンスソースツリー**および**リファレンスバイナリツリー**と呼びます。これにはさまざまな用途があります。

まず、リファレンスソースツリーは、ソースを見る必要のある開発者やマネージャが使います。ささいなことのように思えますが、ファイルやリリースの数が多くなると、1つのファイルを調べるためだけにチェックアウトするのも面倒で非合理的に感じるくらい扱いにくくなります。CVSリポジトリを見るツールも一般的に使われますが、ソースツリー全体を簡易に検索する手段は提供されていないことが多いようです。このためtagsのテーブルを見たり、findやgrep（またはgrep -R†）を使うのが適切な手段となります。

次に、そして最も重要なことに、リファレンスバイナリツリーはソースをそのまま構築したものを表しています。開発者は毎朝システムがうまくできているのか、それとも失敗しているのかを知ることができます。もしバッチ形式のテスト機構が組み込まれていたなら、リファレンスを使ってテストを流すことができるでしょう。開発者は自分でテストを実行しなくてもシステムの状態がどうなっているのかをテスト結果をチェックするだけで把握することができます。開発者がそれぞれリファレンス用のソースをチェックアウトし構築する必要がないので、開発者が持つのは修正した分のソースだけとなり、さまざまな面で費用や時間を節約できます。そして最後に開発者は特定のコンポーネントの機能をテストし結果を比較するためにもリファレンスを使うことができます。

リファレンスは他の用途に使うこともできます。多くのライブラリで構成されるプロジェクトでは、毎晩コンパイルされるライブラリを開発者がそのまま各自のアプリケーションに使えます。このようにすることでソースツリーの大部分を開発者が各個にコンパイルする必要がなくなるため、開発の周期を短くすることができます。もちろんローカルなファイルサーバにプロジェクトのソースを置いておけばコードを調べる必要がありソースツリーの全体をチェックアウトしていない場合にも便利です。

---

† 訳注：grepの-Rオプションは GNU make 2.5から導入されたもので、意味は-rと同じです。

さまざまな用途で使われるようになると、リファレンスソースとバイナリツリーの整合が取れていることの重要性が増します。信頼性向上のための簡単で有効な方法の1つは、ソースツリーを読み出し専用にすることです。そうすることで、リファレンスソースのファイルはリポジトリからチェックアウトした状態を正確に保っていることを保証できます。構築作業のさまざまな局面にはソースツリーの中に何かを書き込もうとするものがあるかもしれないため、特別な配慮が必要になります。特にソースコードを生成したり、中間ファイルを作ろうとするなどです。ソースツリーを読み出し専用にすることは、粗こつな利用者が誤ってソースツリーを書き換えてしまうというよくある事故をも防ぎます。

プロジェクトの構築システムが必要とするもう1つの機能は、コンパイル、リンク、配置の異なる構成を簡単に扱えるというものです。構築システムはプロジェクトの異なるバージョン（たいていはソースリポジトリの分岐）を管理できなければなりません。

大きなプロジェクトの多くはリンク可能なライブラリやツールとしてサードパーティソフトウェアに深く依存しています。もしソフトウェアの構成管理を行う他のツールを持っていなければ（普通は持っていないのですが）、`makefile`を使い構築システムで構成の管理を行うというのは、多くの場合に妥当な選択となります。

ソフトウェアが顧客に対して出荷される際には、開発している状態から出荷用のパッケージが作られるのが普通です。この作業はWindows用に`setup.exe`を作成するような複雑なものから、`HTML`ファイルをフォーマットし`jar`ファイルにまとめるような簡単なものまでさまざまです。このインストーラを作る工程は構築作業の1つとなっていることもあります。構築作業とインストーラの作成はまったく異なる作業であるため、筆者はこれらを2つの別の工程として分けておくのがよいと考えています。ともかく、どちらの作業も構築システムに対しての影響を持つことは十分ありうることです。

6.4　ファイルシステムの配置

複数のバイナリツリーを持つことに決めたなら、ファイルシステム上にどのように配置すべきかが課題となります。複数のバイナリツリーが必要な環境では、通常多くのバイナリツリーを持つことになります。それらのすべてを正しく維持するためには、いくつか考慮すべき点があります。

このようなデータを組織的にまとめる一般的な方法は、バイナリツリーの「倉庫」として大きなディスクを用意することです。ディスクのトップレベル（もしくはその近く）に、各バイナリツリーごとにディレクトリを作ります。

バイナリツリーの妥当な配置方法は、バイナリツリーが対応しているベンダ名、ハードウェアプラットフォーム名、OSのシステム名および構築パラメータをディレクトリの名前に入れることです。

```
$ ls
hp-386-windows-optimized
hp-386-windows-debug
sgi-irix-optimzed
sgi-irix-debug
sun-solaris8-profiled
sun-solaris8-debug
```

複数の構築分を残しておかなければならない場合には、日付（場合により時間も）をディレクトリの名前に入れます。syymmddとかyymmddhhmmという形式が分類には便利です。

```
$ ls
hp-386-windows-optimized-040123
hp-386-windows-debug-040123
sgi-irix-optimzed-040127
sgi-irix-debug-040127
sun-solaris8-profiled-040127
sun-solaris8-debug-040127
```

もちろんファイル名はどのような順番で組み合わせてもかまいません。これらバイナリツリーのトップレベルディレクトリは、makefileやテストの記録を置いておくにはよい場所です。

この配置方法は多くの並行開発を保持するのに適しています。もし開発者が、おそらく内部の利用者に対する、出荷版を作成するなら、図6–2に示すようなバージョン番号と日時を持つ製品の集まりとしての倉庫を追加で用意してもよいと思います。

ここでの製品は、他の開発チームに使ってもらうためのライブラリの場合もあります。従来の考え方では、それを製品ということができます。

ファイルの配置や環境がどうあろうとも、同じ考え方が多くに適用できます。個々のツリーを識別することは簡単です。ファイルの整理方法は明白で、すばやく整理できます。ツリーをアーカイブツリーに簡単に移動できるので便利です。加えてファイルシステムの配置はおそらくその組織が持っている構築プロセスとよく整合が取れていることでしょう。このようにすることで、管理職や品質保証担当やドキュメント担当などのプログラマでない人たちが、この倉庫を容易に探索できるようになります。

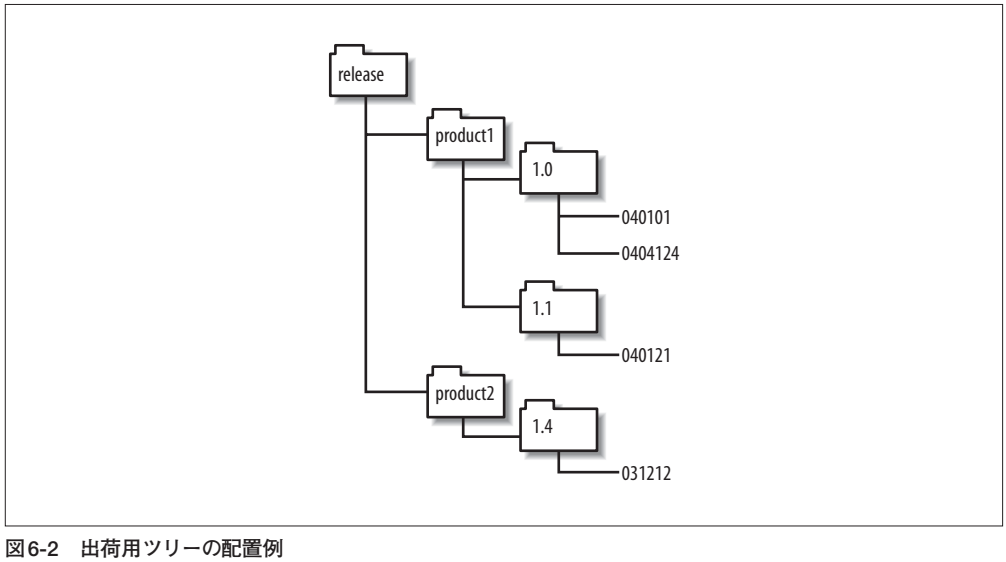

図6-2　出荷用ツリーの配置例

6.5　構築とテストの自動化

構築作業をできるかぎり自動化することは、たいていの場合で重要になります。これによりリファレンスツリーの構築を夜中に行うことが可能となり開発者の日中の時間を使わなくてすみます。同時に開発者が構築作業を行うのに自分のコンピュータについていなくてもよくなります。

生産を終了していないソフトウェアでは、異なる製品の異なるバージョンを別個に構築するという要求が生じます。構築環境の担当者にとって、複数の構築環境がすぐに準備できて手がかからないというのは、健全さを保ち要求を満足するために必須の機能となります。

テストの自動化には独特の課題が存在します。グラフィカルな表示を伴わないプログラムではテスト工程を管理するために単純なスクリプトを使うことができます。GNUツールの1つ`dejaGnu`を使うと、操作を必要とする非グラフィカルユーティリティをテストすることができます。もちろん`JUnit`（`http://www.junit.org`）といったテスト用フレームワークも非グラフィカルな単体テスト機能を提供します。

グラフィカルアプリケーションのテストには特別な問題があります。X11ベースのシステムでは、仮想フレームバッファ`Xvfb`を使って`cron`起動による人手を必要としないテストを実現できました。Windowsで人が介在しないテストを実現する方法を筆者は知りません。これらの手法は、テスト用のアカウントでログインしたままでスクリーンロックをかけない状態にする必要があります。

7章
ポータブルなmakefile

ポータブルなmakefileとは、どういったことを意味するのでしょうか。極端な例でいうと、GNU makeが使えるシステム上でなら一切修正しないで実行できるmakefileを指します。しかしOSはそれぞれ大きく異なっているため、実際には不可能です。もう少し現実的に解釈すれば、新しいプラットフォーム向けに簡単な修正で実行できるmakefileということになります。ここで加えた制約の重要な点は、新しいシステムのために修正を加えても以前のプラットフォームに対して影響しないというものです。

makefileに対するこのレベルのポータビリティは、カプセル化と抽象化という伝統的なプログラミングと同じテクニックを使うことで達成することができます。変数とユーザ定義関数を使えば、アルゴリズムとアプリケーションをカプセル化することができます。変数をコマンドライン引数やパラメータとして定義することで、固定的な要素からプラットフォームごとに異なる要素を抽出することができます。

次に各プラットフォームごとにどういったツールが使えるのか明らかにし、どれを使うのかを決めなければなりません。対象となるすべてのプラットフォームで利用可能なツールや機能だけを使うのが究極の姿です。これは通常**最小公倍数方式**と呼ばれ、非常にかぎられた機能だけを使って作業することになるのは明らかです。

最小公倍数方式の別の方法では、強力なツールを選びそれらをすべてのプラットフォームに移植します。これによりmakefileから実行されるコマンドは、どこでも同じふるまいとなります。これは管理的な側面からも、システムに手を加えるということに組織の協力が得られるかという観点からも成し遂げるのが困難です。しかし、これはうまく行きます。この後でWindows用Cygwinパッケージを使った例を紹介します。ツールの標準化ではすべての問題には対処できません。対応しなければならないOSの差異は必ず存在します。

システム間の相違点を受け入れ、マクロや関数を使ってそれらに対処することは可能です。本章では、そういった手法についても扱います。

つまり変数やユーザ定義関数を賢く使い、特殊な機能の使用をできるだけ避け、標準的なツールを活用することによりmakefileのポータビリティを高めることができます。冒頭でも書いたように、

ポータビリティを完璧に極めることはできず、ポータビリティとそれにかける労力のバランスを取ることが常に求められているのです。それでは、それぞれのテクニックを検討する前に、makefileをポータブルにする際に出てくる問題について見ていきましょう。

7.1　移植における問題点

移植における問題点を特色づけるのは困難です。というのも、パラダイムの変更（例えば旧式のMac OSに対するSystem V Unixなど）から、ちょっとしたバグの修正（例えば、あるプログラムの終了ステータスに関するバグの修正とか）までの広い範囲にわたって移植における問題は存在しているからです。とはいうものの、どのmakefileでもいつかは対処しなければならなくなる共通の問題を拾い上げることは可能です。

プログラム名

同一のまたは同様のプログラムに対してプラットフォームごとに名前が異なっていることはよくあることです。最も一般的なのはCまたはC++コンパイラの名前です（例えばccやxlc）。GNU版のプログラムをGNU以外のシステムにインストールする際に付けられるプリフィクス（例えばgmake、gawk）も同様です。

パス名

プログラムやファイルの置き場所はプラットフォームごとに異なります。例えばSolarisではX Window Systemのディレクトリは/usr/Xですが、その他多くのシステムでは/usr/X11R6となります。さらに/bin、/usr/bin、/sbin、/usr/sbinの違いは、慣れないシステムではよくわからないものです。

オプション

プログラムに対するコマンドラインオプションは、実装ごとにそれぞれ異なります。さらにいうと、もしプラットフォームにそのプログラムが置いていなかったり、バグのあるバージョンであった場合には、別のコマンドラインオプションを持つプログラムと置き換えなければならなくなるかもしれません。

シェル機能

既定では、makeは/bin/shを使ってコマンドスクリプトを実行しますが、シェル機能は実装により大きく違います。例えば、POSIX以前のシェルでは多くの機能を欠いており、現代的なシェルが受け付ける文法がエラーになります。

Open GroupはSystem VのシェルとPOSIXシェルとの違いに関する実用的な白書を提供しています。http://www.unix-systems.org/whitepapers/shdiffs.htmlから取り寄せることができます。より詳しい情報が欲しい場合には、http://www.opengroup.org/onlinepubs/007904975/utilities/xcu_chap02.htmlからPOSIXシェルの言語仕様書が得られます。

プログラムの挙動

ポータブルなmakefileを書くことは、異なる動作をするプログラムとの戦いでもあります。さ

まざまなプログラムの供給メーカがバグを修正したり作りこんだり、新しい機能を追加することは日常茶飯事です。ユーティリティの機能拡張が、プログラム供給メーカのリリースに含まれたり含まれなかったりします。例えば、1987年にawkプログラムは大きなバージョンアップが行われました。それから20年近くたちますが、いくつかのシステムではこの新しいバージョンを標準のawkとして提供していません。

OS

WindowsとUnix（Linux）とVMSのようにまったく異なるOSに起因する問題もあります。

7.2 Cygwin

Win32の上に移植されたmakeも存在しますが、これはWindowsにおける移植性の問題のほんの一部でしかありません。なぜならこの移植版が使うシェルはcmd.exe（またはcommand.com）だからです。これは、他のUnixツールの多くを欠いていることと合わせて、プラットフォーム間の移植問題を厄介なものにしています。幸いなことにCygwinプロジェクト（http://www.cygwin.com）はWindowsにLinux互換のライブラリを提供し、いまや多くのプログラム†がWindowsに移植されています。Linuxとの互換性が必要だったりGNUツールを使いたいWindowsの開発者が、これ以上のものを望むのは無理だと思います。

C++とLispが混在したCADアプリケーションから、pure Javaによるワークフロー管理システムまで、さまざまなプロジェクトにおいて筆者はCygwinを10年以上使い続けています。Cygwinツールには、各種プログラミング言語のコンパイラやインタプリタも含まれています。アプリケーション自身はCygwinのコンパイラやインタプリタを使っていなくても、Cygwinを使うのは有益です。Cygwinのツールは開発と構築プロセスを協調させる補助として、単体で使うことができます。つまり、Cygwinアプリケーションを書いたりCygwinの言語ツールを使わなくても、Cygwin環境から利益を得ることができるのです。

とはいうもののLinuxはWindowsではありませんし（本当によかった!）、Cygwinツールを生粋のWindowsアプリケーションに対して使う場合には問題が生じます。こうした問題のほとんどはファイルの行末文字と、CygwinとWindowsに渡されるパス名の形式に関連したものです。

7.2.1 行末文字

Windowsのファイルシステムにおいて、テキストファイルでは復帰文字とそれに続く改行文字の2文字（CRLF）で行末を表しています。POSIXシステムでは復帰文字（LFまたはnewline）の1文字を使います。時折この差異は、プログラムが文法エラーを起こしたりファイル中の誤った位置にあるデータを使ったりといったちょっとした混乱を引き起こします。しかしCygwinライブラリはこの件に関して、便利な機能を提供しています。Cygwinのインストール時（あるいは、mountコマンドの使用時）、CygwinがCRLFの行末を変換するかどうかを選択できます。DOSファイル形式が選択された場合には、CygwinはUnixベースのプログラムがDOS形式のテキストファイルを適切に扱えるように、

† 筆者の環境ではCygwinの/binディレクトリには1,343もの実行可能なファイルが存在します。

ファイル読み込み時にCRLFをLFに、書き出し時にはその逆に変換を行います。もしVisual C++やSunのJava SDKなどといったWindowsのツールを使うのであればDOSファイル形式を選択し、Cygwinのコンパイラを使うのであればUnix形式を使うのがよいでしょう（形式の選択はいつでも変更できます）。

加えて、Cygwinには明示的にファイル形式を変換するツールが提供されています。dos2unixとunix2dosは、必要ならばファイルの行末形式を変換します。

7.2.2 ファイルシステム

CygwinはWindowsのファイルシステムをPOSIX的に見せます。POSIXファイルシステムのルートは/であり、これはCygwinがインストールされたディレクトリになります。Windowsのドライブは仮想ディレクトリである/cygdrive/**ドライブ文字**を通して利用可能です。Cygwinが`C:\usr\sygwin`（筆者の好みの場所です）にインストールされていた場合のディレクトリ位置関係を表7–1に示します。

表7-1 Cygwinの既定ディレクトリ

Windowsパス名	Cygwinパス名	Cygwinの別名
c:\usr\cygwin	/	/cygdrive/c/usr/cygwin
c:\Program Files	/cygdrive/c/Program Files	
c:\usr\cygwin\bin	/bin	/cygdrive/c/usr/cygwin/bin

最初は混乱しますが、ツールに対しては何ら問題ありません。Cygwinは利用者がもっと簡便にファイルやディレクトリを使えるよう、mountコマンドも提供しています。mountコマンドのオプションである--change-cygdrive-prefixを使うことで、プリフィクスを変更できます。プリフィクスを単なる/に変更すれば、ドライブ文字がより自然に使えるようになります。

```
$ mount --change-cygdrive-prefix /
$ ls /c
AUTOEXEC.BAT            Home          Program Files             hp
BOOT.INI                I386          RECYCLER                  ntldr
CD                      IO.SYS        System Volume Information pagefile.sys
CONFIG.SYS              MSDOS.SYS     Temp                      tmp
C_DILLA                 NTDETECT.COM  WINDOWS                   usr
Documents and Settings  PERSIST       WUTemp                    work
```

この変更を行えば、前述のディレクトリ位置関係は表7–2のようになります。

表7-2 変更後のディレクトリ位置関係

Windowsパス名	Cygwinパス名	Cygwinの別名
c:\usr\cygwin	/	/c/usr/cygwin
c:\Program Files	/c/Program Files	
c:\usr\cygwin\bin	/bin	/c/usr/cygwin/bin

Visual C++コンパイラのようなWindowsプログラムにファイルのパス名を渡す必要があるときにはPOSIX形式のスラッシュを使った相対パス名を渡すことができます。Win32 APIは、スラッシュとバックスラッシュを区別しません。残念ながら、ユーティリティコマンドの中にはコマンドラインを独自に解析し、スラッシュをコマンドに対するオプションとして扱うものがあります。例えばDOSのprintコマンドがそうですし、netコマンドも同様です。

もし絶対パスを使う場合には、ドライブ文字の部分が常に問題を引き起こします。たいていのWindowsプログラムはスラッシュを渡されても問題ありませんが、/c形式の指定は理解できません。ドライブ文字は常にc:形式に戻さなければなりません。この問題を扱うと同時にスラッシュとバックスラッシュと交換しPOSIX形式のパス名をWindows形式にするためのcygpathユーティリティをCygwinは提供しています。

```
$ cygpath --windows /c/work/src/lib/foo.c
c:\work\src\lib\foo.c
$ cygpath --mixed /c/work/src/lib/foo.c
c:/work/src/lib/foo.c
$ cygpath --mixed --path "/c/work/src:/c/work/include"
c:/work/src;c:/work/include
```

--windowsオプションはコマンドライン上で指定されたPOSIX形式のパス名をWindows形式へ変換（または適切な引数により逆変換）します。筆者は、バックスラッシュではなくスラッシュを使ってWindows形式のパス名に変換する--mixedオプションをよく使います（ただしWindowsのユーティリティが受け付ける場合のみ）。これはCygwinシェルを使う場合に適切です。なぜならバックスラッシュはエスケープ文字として解釈されてしまうからです。cygpathユーティリティには多くのオプションが提供されていて、そのうちのいくつかはWindowsシステムのパスに対するポータビリティのあるアクセス手段を提供しています。

```
$ cygpath --desktop
/c/Documents and Settings/Owner/Desktop
$ cygpath --homeroot
/c/Documents and Settings
$ cygpath --smprograms
/c/Documents and Settings/Owner/Start Menu/Programs
$ cygpath --sysdir
/c/WINDOWS/SYSTEM32
$ cygpath --windir
/c/WINDOWS
```

cygpathをWindowsとUnixの両方で使うには、どちらでも使える関数の中に入れておきたくなるでしょう。

```
ifdef COMSPEC
  cygpath-mixed = $(shell cygpath -m "$1")
  cygpath-unix = $(shell cygpath -u "$1")
  drive-letter-to-slash = /$(subst :,,$1)
else
```

```
    cygpath-mixed = $1
    cygpath-unix = $1
    drive-letter-to-slash = $1
  endif
```

もし単にc:のようなドライブ文字をPOSIX形式に変換したいだけなら、cygpathプログラムを使うよりもdrive-letter-to-slash関数を使ったほうがより高速です。

しかしながら、CygwinはWindowsの持つ特異性をすべて覆い隠してくれるわけではありません。Windowsで無効のファイル名はCygwinに対しても無効です。aux.h、com1、prnといったファイル名は、たとえサフィックスが付いていたとしてもPOSIXパス名として使うことができません†。

7.2.3　プログラムの衝突

WindowsプログラムのいくつかはUnixのプログラムと同じ名前を持つものがあります。もちろんWindowsのプログラムはUnixのプログラムと同じコマンドライン引数を受け付けませんし、互換性のある動作をするわけでもありません。もし誤ってWindowsプログラムのほうを実行してしまうと、深刻な混乱を引き起こしてしまいます。紛らわしいのは、find、sort、ftp、telnetでしょう。できるかぎりポータビリティを持たせるために、Unix、Windows、Cygwinで移植する必要のあるプログラムに対しては、これらのプログラムを完全パス名で記述しておくのがよいでしょう。

Cygwinに深く傾倒し、かつWindows専用のツールを使う必要がないのならば、Cygwinの/binディレクトリをWindowsのパスの先頭に置いても大丈夫です。これにより、Windowsの提供しているツールではなくCygwinのツールが必ず使われることになります。

もしもmakefileの中でJavaのツールを使っているならば、標準的なSunのjarファイル形式と互換性のないjarプログラムをCygwinは含んでいることに注意しなければなりません。そこで、PATH変数の中でCygwinの/binの前にJDKのbinディレクトリを、置くことによりCygwinのjarプログラムが起動されることを防ぐことができます。

7.3　プログラムとファイルを管理する

プログラムを管理する一般的な方法は、変更される可能性のあるプログラム名やパス名を変数に入れることです。変数はすでに見てきたように次のように単純に定義します。

```
MV ?= mv -f
RM ?= rm -f
```

または次のように条件判断の中で行うこともあります。

```
ifdef COMSPEC
  MV ?= move
```

† 訳注：DOS時代からの歴史的理由により、aux、com、prnなどはWindowsのファイルシステムで予約された名前となっています。これらはコンピュータに接続された装置を表し、ファイル名として使うことはできません。

```
  RM ?= del
else
  MV ?= mv -f
  RM ?= rm -f
endif
```

単純に定義した場合、変数はコマンドラインで設定したりmakefileを書き換えたり、（ここでは条件付き代入である?=を使っているので）環境変数を設定することで変更できます。3章や4章でも述べたように、Windowsプラットフォームを使っているかどうかは、すべてのWindowsで使用されているCOMSPEC変数の存在を確認することでわかります。場合によってはパス名だけを変える必要が生じます。

```
ifdef COMSPEC
  OUTPUT_ROOT := d:
  GCC_HOME := c:/gnu/usr/bin
else
  OUTPUT_ROOT := $(HOME)
  GCC_HOME := /usr/bin
endif

OUTPUT_DIR := $(OUTPUT_ROOT)/work/binaries
CC := $(GCC_HOME)/gcc
```

このスタイルを用いることで、makefileの中で使われるほとんどのプログラムはmakeの変数を介して呼び出されることになります。慣れるまでは可読性が多少低下してしまいます。しかしながら、特に完全パスが使われる場合など、変数を使ったほうがプログラムの名前を書き下すよりも短く表現できるので、たいていは便利なのです。

同じ手法は、異なるコマンドオプションを管理するときにも使えます。例えば、組み込みのコンパイルルールは、特定のプラットフォーム用のフラグをセットするためのTARGET_ARCH変数を使っています。

```
ifeq "$(MACHINE)" "hpux-hppa"
  TARGET_ARCH := -mdisable-fpregs
endif
```

プログラム名用の変数を作る際にも、同じ手法が使えます。

```
MV := mv $(MV_FLAGS)
ifeq "$(MACHINE)" "solaris-sparc"
  MV_FLAGS := -f
endif
```

多くのプラットフォームに移植する場合、ifdefの中を変更するのは面倒で、保守も大変になります。ifdefを使う代わりにプラットフォーム固有の変数をそれぞれプラットフォームを示す名前の付いたファイルに入れておきます。例えばunameのパラメータを使ってプラットフォームを指定するなら、次のようにして適切なインクルードファイルを選択できます。

```
MACHINE := $(shell uname -smo | sed 's/ /-/g')
include $(MACHINE)-defines.mk
```

空白を含むファイル名は、特にmakeに関する不愉快な問題を引き起こします。構文解析を行う際に単語は空白で区切られているというという想定は、makeにとって基本的な了解事項です。word、filter、wildcardを始めとして多くの組み込み関数は、引数として渡されるのは空白で区切られた単語であることを想定しています。とはいうものの、ここで役に立つちょっとしたトリックをいくつか紹介します。最初のトリックは、「8.4　複数のバイナリツリーを利用する」で紹介するsubstを使って空白を別の文字に置き換えるというものです。

```
space = $(empty) $(empty)
# $(call space-to-question,file-name)
space-to-question = $(subst $(space),?,$1)
```

space-to-question関数は、すべての空白を展開ワイルドカード文字である疑問符（?）に置き換えます。これを使えば、空白を扱うことのできるwildcard関数とfile-exists関数を作れます。

```
# $(call wildcard-spaces,file-name)
wildcard-spaces = $(wildcard $(call space-to-question,$1))

# $(call file-exists
file-exists = $(strip                                   \
                $(if $1,,$(warning $1 has no value)) \
                  $(call wildcard-spaces,$1))
```

wildcard-spaces関数はspace-to-question関数を使い、空白を含むファイル名に対してwildcardの操作ができるようにします。wildcard-spacesはfile-exists関数を作成するために使えます。もちろん疑問符を使うことでwildcard-spaces関数はワイルドカードパターンと正確に合致していないファイル（例えば“my　document.doc”と“my-document.doc”）も返すようになってしまいますが、これが取りうる最善の手法です。

ターゲットと必須項目にも展開パターンを使うことができるので、space-to-question関数はスペースを含むファイル名を持つターゲットと必須項目に対しても使えます。

```
space := $(empty) $(empty)

# $(call space-to-question,file-name)
space-to-question = $(subst $(space),?,$1)

# $(call question-to-space,file-name)
question-to-space = $(subst ?,$(space),$1)

$(call space-to-question,foo bar): $(call space-to-question,bar baz)
        touch "$(call question-to-space,$@)"
```

“bar baz”というファイルがすでに存在しているとして、最初にmakefileが実行されたときには展開パターンが評価され必須項目として見つかります。しかしターゲットはまだ存在していないので、ターゲットの展開は失敗し$@はfoo?barという値を持つことになります。コマンドスクリプトはquestion-to-space関数を用い$@を空白を含む本来のファイル名に戻します。次回makefileを実行するときには展開パターンがスペースを含むファイルを見つけるので、ターゲットが見つかります。多少不恰好なのですが、こういったトリックは実際に使われるmakefileでは便利なものです。

7.4　ポータビリティのないツールで作業する

以前書いたように、最小公倍数方式でmakefileを書く際の選択肢として、いくつかの標準ツールを採用するという方法があります。もちろん必要なのは構築するアプリケーションと少なくとも同じ程度のポータビリティを、標準ツールが持っていることを確認することです。ポータビリティのあるツールといえばGNUプロジェクトのプログラムですが、その他にも非常に多くの選択肢があります。PerlとPythonの2つがすぐに思い浮かびます。

ポータビリティのあるツールがないときには、ポータビリティのないツールをmakeの関数で囲めば、場合によりちょうどよい機能となります。例えばエンタープライズJavaBeans向けのさまざまなコンパイラに対応（それぞれが少しずつ違った起動形式を持っています）するため、EJB jarをコンパイルする基本的な関数を作り、それを異なるコンパイラに対応するためにパラメータ化します。

```
EJB_TMP_JAR = $(TMPDIR)/temp.jar

# $(call compile-generic-bean, bean-type, jar-name,
#                              bean-files-wildcard, manifest-name-opt )
define compile-generic-bean
  $(RM) $(dir $(META_INF))
  $(MKDIR) $(META_INF)
  $(if $(filter %.xml %.xmi, $3),             \
    cp $(filter %.xml %.xmi, $3) $(META_INF))
  $(call compile-$1-bean-hook,$2)
  cd $(OUTPUT_DIR) &&                         \
  $(JAR) -cf0 $(EJB_TMP_JAR)                  \
         $(call jar-file-arg,$(META_INF))     \
         $(call bean-classes,$3)
  $(call $1-compile-command,$2)
  $(call create-manifest,$(if $4,$4,$2),,)
endef
```

この汎用（general）EJBコンパイル関数に対する最初の引数は、WeblogicやWebSphereといった使用するbeanコンパイラの種類を指定します。残りの引数はjar名、jarを構成するファイル名（構成ファイルも含みます）、そして最後にマニフェストファイル名である任意引数です。この雛形関数が最初に行うのは、一時ディレクトリを削除して再作成することで、一時ディレクトリをきれいにすることです。次に指定されたファイルの中からxmlファイル、xmiファイルを選んで$(META_INF)ディレクトリにコピーします。この時点でMETA-INFのファイルを整理したり.class

ファイルを用意するためのコンパイラごとの個別操作が必要となるでしょう。こういった作業を行うために、必要に応じて利用者が定義することのできるフック関数compile-$1-bean-hookを入れてあります。例えばWebSphereコンパイラはxslファイルという追加の制御ファイルを必要とするため、次のようなフック関数を用意します。

```
# $(call compile-websphere-bean-hook, file-list)
define compile-websphere-bean-hook
  cp $(filter %.xsl, $1) $(META_INF)
endef
```

単に関数を定義しておけば、compile-generic-bean中のcall関数が適切に展開されます。もしフック関数を用意しなければcall関数は何も展開されません。

次にjarファイルを作ります。補助関数jar-file-argはファイルのパス名を-Cオプションと相対パスに分解します。

```
# $(call jar-file-arg, file-name)
define jar-file-arg
  -C "$(patsubst %/,%,$(dir $1))" $(notdir $1)
endef
```

補助関数bean-classesはソースファイルのリストから、適切なclassファイルを抽出します(jarファイルはinterfaceクラスとhomeクラスだけを必要とします)。

```
# $(call bean-classes, bean-files-list)
define bean-classes
  $(subst $(SOURCE_DIR)/,,                  \
    $(filter %Interface.class %Home.class, \
      $(subst .java,.class,$1)))
endef
```

そして$(call $1-compilecommand,$2)により選択したコンパイラを起動します。

```
define weblogic-compile-command
  cd $(TMPDIR) && \
  $(JVM) weblogic.ejbc -compiler $(EJB_JAVAC) $(EJB_TMP_JAR) $1
endef
```

最後にマニフェストファイルを追加します。

compile-generic-bean関数は、サポートしたい環境に対する特定コンパイラ向けの関数内で使えます。

```
# $(call compile-weblogic-bean, jar-name,
#                               bean-files-wildcard, manifest-name-opt )
define compile-weblogic-bean
  $(call compile-generic-bean,weblogic,$1,$2,$3)
endef
```

7.4.1 標準シェル

あるシステムから別のシステムへと環境を移したときに遭遇する不快な非互換性の1つは、makeの既定シェルである/bin/shの機能であることをここで再確認することは重要です。もしmakefileのコマンドスクリプトをいろいろと調整し続けているなら、シェルを固定することを考えてもよいでしょう。もちろんmakefileがさまざまな環境で実行されるような典型的オープンソース開発プロジェクトでは妥当とはいえません。しかしながら、決められた構成のコンピュータ上で管理された設定の下で行われる開発であるなら理にかなっています。

シェルの非互換性を排除することに加えて、多数の小さなユーティリティの代わりに使える機能をシェルの多くは提供しています。例えばbashは%%や##といった機能強化した変数展開の機能を持っています。これらはsedやexprの代わりに使えます。

7.5 automake

本章が主題としていたのは、GNU makeと補助ツールを使ってポータビリティのある構築システムを作ることでした。しかしながら、こうしたささやかな要求でさえ状況によっては実現できないこともあります。もしGNU makeの拡張機能が使えず最小公倍数方式の機能だけに依存せざるをえない場合には、automakeツール（http://www.gnu.org/software/automake/automake.html）の使用を検討してもよいかもしれません。

automakeは様式化されたmakefileを入力すると昔ながらのmakefileを生成して出力します。automakeは非常に簡潔に記述された入力ファイル（makefile.amと呼びます）を処理するm4のマクロを中心に構成されています。automakeはC/C++プログラムのための移植サポートパッケージであるautoconfといっしょに使うことが多いのですが、autoconfが必須というわけではありません。

automakeは最大限のポータビリティが要求される構築システムに対する優れたツールですが、生成されるmakefileはアペンド演算子（+=）を除いてGNU makeの拡張機能を使いません。そしてautomakeへの入力は通常のmakefileと多少似ています。このようにautomakeを（autoconfなしで）使うことは、最小公倍数方式とそれほど大きく異なっているというわけではありません。

8章
CとC++

本章では6章で明らかにした問題と解法を掘り下げ、CとC++に対して適用します。MP3プレーヤを構築する非再帰的makefileの例を引き続き使用します。

8.1 ソースとバイナリの分離

もしも1つのソースツリーで複数のプラットフォームをサポートし、各プラットフォームごとに複数のバイナリツリーを持つ必要があるなら、ソースツリーとバイナリツリーを分離する必要があります。さてどのようにすればよいのでしょうか。もともとmakeは1つのディレクトリに入ったファイルに対してうまく働くように作られています。その後大幅な改良が行われましたが、本来の性質は受け継いでいます。makeは複数のディレクトリに対して動作しますが、更新するファイルがカレントディレクトリ（またはそのサブディレクトリ）にある場合に最もうまく働きます。

8.1.1 簡単な方法

makeがバイナリをソースとは別の場所に置けるようにする最も簡単な方法は、makeをバイナリディレクトリから起動することです。以前の章で見たように出力ファイルは相対パスを通して指定されますが、入力となるファイルは明示的に指定されたパスかvpath経由で探索できなければなりません。どちらの場合でも、複数のソースディレクトリを参照する必要があるため、それを保持する変数を用意することから始めます。

```
SOURCE_DIR := ../mp3_player
```

以前のmakefileからsource-to-object関数は変更しませんが、subdirectory関数はソースへの相対パスを考慮する必要があります。

```
# $(call source-to-object, source-file-list)
source-to-object = $(subst .c,.o,$(filter %.c,$1)) \
                   $(subst .y,.o,$(filter %.y,$1)) \
```

```
                    $(subst .l,.o,$(filter %.l,$1))

# $(subdirectory)
subdirectory = $(patsubst $(SOURCE_DIR)/%/module.mk,%, \
                  $(word                                \
                    $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST)))
```

変更後のmakefileではMAKEFILE_LISTはソースディレクトリへの相対パスを含むことになります。そのため、そのモジュールディレクトリへの相対パス部分を取り出すために、module.mkだけでなく、前置部も取り除かなければなりません。

次にmakeがソースを見つけられるよう、vpathを使用します。

```
vpath %.y $(SOURCE_DIR)
vpath %.l $(SOURCE_DIR)
vpath %.c $(SOURCE_DIR)
```

これにより出力ファイルと同様にソースファイルに対しても単純な相対パスを使うことができるようになります。makeはソースファイルが必要になると、出力ファイルツリーのカレントディレクトリにソースファイルが存在しなければ、SOURCE_DIRを探します。次にinclude_dirs変数を修正しなければなりません。

```
include_dirs := lib $(SOURCE_DIR)/lib $(SOURCE_DIR)/include
```

ソースのディレクトリに加え、バイナリツリーのlibディレクトリも加えています。なぜならyaccとlexが生成するヘッダファイルがそこに置かれるためです。

インクルードファイルの探索にvpathは使えないので、module.mkをソースファイルから取り出すようmakeのinclude命令を変更しなければなりません。

```
include $(patsubst %,$(SOURCE_DIR)/%/module.mk,$(modules))
```

そして出力ディレクトリ自体を作成します。

```
create-output-directories :=                          \
           $(shell for f in $(modules);               \
                   do                                 \
                     $(TEST) -d $$f || $(MKDIR) $$f;  \
                   done)
```

この代入により実際には使われないダミーの変数が作られますが、これは単純変数であるため、makeが他の作業を開始する前にディレクトリが作成されることを保証します。yaccやlexを使用したり、依存関係ファイルを生成しても、出力ファイルディレクトリ自体は作成されないため、このように「手動で」作る必要があります。

これらのディレクトリを確実に作成する別の方法は、ディレクトリを依存関係ファイルの必須項目にしてしまうことです。しかしディレクトリは本物の必須項目ではないため、これはよいアイディアと

はいえません。yaccやlex、そして依存関係ファイルはディレクトリの内容に依存してませんし、ディレクトリのタイムスタンプが更新されたから再生成されるわけでもありません。出力ファイルに対してファイルの増減がありプロジェクトが再構築される場合に多大なる非効率性をもたらします。

module.mkに対する変更はもっと簡単です。

```
local_src := $(addprefix $(subdirectory)/,playlist.y scanner.l)

$(eval $(call make-library, $(subdirectory)/libdb.a, $(local_src)))

.SECONDARY: $(call generated-source, $(local_src))

$(subdirectory)/scanner.d: $(subdirectory)/playlist.d
```

ここではソースを探すためのワイルドカードが省略されています。この機能を復活させるのは容易なので、読者への宿題としておきましょう。元のmakefileのバグかと思われるような不具合が1つあります。この例を実行するとscanner.d依存関係ファイルが、依存しているplaylist.hよりも先に生成されることがわかりました。この依存関係は元のmakefileにはなかったものですが、まったくの偶然からうまく動作してしまいます。小さなプロジェクトにおいてもすべての依存関係を適切に保つのは難しいことなのです。

ソースがmp3_playerにあるとした場合、新しいmakefileを使ったプロジェクトの構築は次のように行います。

```
$ mkdir mp3_player_out
$ cd mp3_player_out
$ make --file=../mp3_player/makefile
```

このmakefileは適切でうまく働きますが、出力ファイルに移動してから--fileオプション（-fオプション）を使わなければならないのは、多少面倒ではあります。これは簡単なシェルスクリプトを使って解決できます。

```
#! /bin/bash
if [[ ! -d $OUTPUT_DIR ]]
then
  if ! mkdir -p $OUTPUT_DIR
  then
    echo "Cannot create output directory" > /dev/stderr
    exit 1
  fi
fi
cd $OUTPUT_DIR
make --file=$SOURCE_DIR/makefile "$@"
```

このスクリプトは、ソースと出力ディレクトリがそれぞれ環境変数のSOURCE_DIRとOUTPUT_DIRに入っていることを想定します。これはディレクトリツリーを頻繁に移動する際に何度もパス名を入力しないための標準的な手法です。

最後に注意が1つあります。本来このmakefileはバイナリツリーから実行されるものですが、ソースツリーからの実行を防ぐ手段はmakeにもこのmakefile自体にもありません。これはよくある誤りであり、ある種のコマンドスクリプトは、非常にまずい動作になります。例えばcleanターゲットを次のように書くとソースツリー全体を削除してしまいます。

```
.PHONY: clean
clean:
        $(RM) -r *
```

makefileの最初にこの偶発事故をチェックするのは賢明だといえます。適切なチェック方法を次に示します。

```
$(if $(filter $(notdir $(SOURCE_DIR)),$(notdir $(CURDIR))),\
  $(error Please run the makefile from the binary tree.))
```

このコードは現在の作業ディレクトリの名前（$(notdir $(CURDIR))）がソースディレクトリ（$(notdir $(SOURCE_DIR))）と同じであるかを調べます。もしそうなら、エラーメッセージを出力し実行を終了します。ifとerror関数は展開されることがないので、この2行をSOURCE_DIR定義の直後に置くことができます。

8.1.2 難しい方法

バイナリツリーの中にcdで移動することが非常に面倒なので、それを回避するためならどんなことでもする開発者も中にはいるでしょうし、もしかしたらシェルスクリプトやシェルの別名を使ってmakeを呼び出すことが適切ではない環境で作業をしているmakefileの保守担当者がいるかもしれません。どちらにしても、makeをソースツリーから実行し出力ファイルにパス情報を付加することで別のツリーにファイルを出力できるようmakefileを修正することは可能です。今のところ、柔軟性の点で絶対パスを使うのがよいのですが、コマンドライン長の制限に関する問題を悪化させてしまいます。そこで入力ファイルにはmakefileディレクトリからの相対パスを使い続けることにします。

例8-1はソースツリーからmakeを実行し、バイナリファイルをバイナリツリーに出力するよう修正したmakefileです。

例8-1 ソースツリーから実行してソースとバイナリを分離するmakefile

```
SOURCE_DIR := /c/home/mecklen/book/examples/ch07-separate-binaries-1
BINARY_DIR := /c/home/mecklen/book/out/mp3_player_out

# $(call source-dir-to-binary-dir, directory-list)
source-dir-to-binary-dir = $(addprefix $(BINARY_DIR)/, $1)

# $(call source-to-object, source-file-list)
source-to-object = $(call source-dir-to-binary-dir,     \
                     $(subst .c,.o,$(filter %.c,$1))    \
                     $(subst .y,.o,$(filter %.y,$1))    \
                     $(subst .l,.o,$(filter %.l,$1)))
```

```
# $(subdirectory)
subdirectory = $(patsubst %/module.mk,%,                \
                 $(word                                 \
                   $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST)))

# $(call make-library, library-name, source-file-list)
define make-library
  libraries += $(BINARY_DIR)/$1
  sources   += $2

  $(BINARY_DIR)/$1: $(call source-dir-to-binary-dir,    \
                      $(subst .c,.o,$(filter %.c,$2))   \
                      $(subst .y,.o,$(filter %.y,$2))   \
                      $(subst .l,.o,$(filter %.l,$2)))
        $(AR) $(ARFLAGS) $$@ $$^
endef

# $(call generated-source, source-file-list)
generated-source = $(call source-dir-to-binary-dir,     \
                     $(subst .y,.c,$(filter %.y,$1))    \
                     $(subst .y,.h,$(filter %.y,$1))    \
                     $(subst .l,.c,$(filter %.l,$1)))   \
                   $(filter %.c,$1)

# $(compile-rules)
define compile-rules
  $(foreach f, $(local_src),\
    $(call one-compile-rule,$(call source-to-object,$f),$f))
endef

# $(call one-compile-rule, binary-file, source-files)
define one-compile-rule
  $1: $(call generated-source,$2)
        $(COMPILE.c) -o $$@ $$<

  $(subst .o,.d,$1): $(call generated-source,$2)
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $$< | \
        $(SED) 's,\($$(notdir $$*)\.o\) *:,$$(dir $$@)\1 $$@: ,' > $$@.tmp
        $(MV) $$@.tmp $$@

endef

modules      := lib/codec lib/db lib/ui app/player
programs     :=
libraries    :=
sources             :=

objects      =         $(call source-to-object,$(sources))
dependencies =         $(subst .o,.d,$(objects))

include_dirs := $(BINARY_DIR)/lib lib include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))
vpath %.h $(include_dirs)
```

```
MKDIR := mkdir -p
MV    := mv -f
RM    := rm -f
SED   := sed
TEST  := test

create-output-directories :=                                          \
        $(shell for f in $(call source-dir-to-binary-dir,$(modules));    \
                do                                                    \
                  $(TEST) -d $$f || $(MKDIR) $$f;                     \
                done)

all:

include $(addsuffix /module.mk,$(modules))

.PHONY: all
all: $(programs)

.PHONY: libraries
libraries: $(libraries)

.PHONY: clean
clean:
        $(RM) -r $(BINARY_DIR)

ifneq "$(MAKECMDGOALS)" "clean"
  include $(dependencies)
endif
```

ここでは、source-to-object関数を修正してバイナリツリーのパスを前置するようになっています。

```
SOURCE_DIR := /test/book/examples/ch07-separate-binaries-1
BINARY_DIR := /test/book/out/mp3_player_out

# $(call source-dir-to-binary-dir, directory-list)
source-dir-to-binary-dir = $(addprefix $(BINARY_DIR)/, $1)

# $(call source-to-object, source-file-list)
source-to-object = $(call source-dir-to-binary-dir,  \
                     $(subst .c,.o,$(filter %.c,$1)) \
                     $(subst .y,.o,$(filter %.y,$1)) \
                     $(subst .l,.o,$(filter %.l,$1)))
```

make-library関数も同様にBINARY_DIRを出力ファイルの先頭に付加します。includeパスは単純な相対パスに戻ったので、subdirectory関数も元に戻っています。小さな障害が1つあります。make 3.80のバグによりsource-to-object関数を修正後のmake-library関数から呼ぶことができません。このバグは3.81で修正されています。source-to-object関数を手で展開することにより、このバグに対処することができます。

さて、真に醜い個所にやってきました。出力ファイルがmakefileのディレクトリから相対的にたどり着くことができない場所にある場合には、もはや暗黙ルールは実行されません。例えば、基本的なコンパイルルール%.o: %.cは2つのファイルの位置が同じディレクトリであるか、Cファイルがlib/codec/codec.cのようなサブディレクトリにあるか、どちらかの場合にうまく働きます。ソースファイルがリモートディレクトリに存在している場合にはvpathを使ってmakeにソースを探す場所を教えることができます。しかしオブジェクトファイルがリモートディレクトリに存在する場合、makeはオブジェクトファイルがどこに存在するか知る方法がなくなり、ターゲットと必須項目の連鎖が壊れてしまいます。

出力ファイルの場所をmakeに知らせる唯一の方法は、ソースとオブジェクトファイルの関連を示す明示的なルールを提供することです。

```
$(BINARY_DIR)/lib/codec/codec.o: lib/codec/codec.c
```

これはすべてのオブジェクトファイルに対して行われる必要があります。

さらに悪いことには、このターゲットと必須項目の関係は暗黙ルールの%.o: %.cとは適合しません。つまりコマンドスクリプトも同様に指定する必要があり、暗黙データベースにあるあらゆるものが何度も繰り返されることを意味します。問題はこれまで使っていた依存関係の自動生成にまで及びます。各オブジェクトファイルに対してそれぞれ2つのルールをmakefileに指定するのを手作業で行うとしたら、それは保守に関する悪夢としかいいようがありません。しかしこれらのルールを生成する関数を書くことによりコードの重複と保守に関する問題を最小限にすることが可能です。

```
# $(call one-compile-rule, binary-file, source-files)
define one-compile-rule
    $1: $(call generated-source,$2)
            $(COMPILE.c) -o $$@ $$<

    $(subst .o,.d,$1): $(call generated-source,$2)
          $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $$< | \
          $(SED) 's,\($$(notdir $$*)\.o\) *:,$$(dir $$@)\1 $$@: ,' > $$@.tmp
          $(MV) $$@.tmp $$@
endef
```

最初の2行はオブジェクトとソースの依存関係を表す明示的ルールです。ソースファイルのいくつかはyaccとlexに対するもので、これらがコマンドスクリプトに出てくると（例えば$^が展開されたとき）、コンパイルエラーになってしまうので、このルールに関する必須項目は6章で作成したgenerated-source関数を使って生成しなければなりません。自動変数にドル記号が2つ付いているのは、ユーザ定義関数がevalによって評価されたときではなく、コマンドスクリプトが実行されたときに展開されるべきだからです。generated-source関数はyaccやlexが生成するファイルに加えて、Cファイルをそのまま返すように変更します。

```
# $(call generated-source, source-file-list)
generated-source = $(call source-dir-to-binary-dir,   \
                     $(subst .y,.c,$(filter %.y,$1))  \
```

```
                $(subst .y,.h,$(filter %.y,$1))  \
                $(subst .l,.c,$(filter %.l,$1))) \
              $(filter %.c,$1)
```

この変更により、この関数は次の出力を行うようになります。

```
引数                      結果
lib/db/playlist.y     /c/mp3_player_out/lib/db/playlist.c
                      /c/mp3_player_out/lib/db/playlist.h
lib/db/scanner.l      /c/mp3_player_out/lib/db/scanner.c
app/player/play_mp3.c app/player/play_mp3.c
```

依存関係を生成するための明示的なルールも同様です。繰り返しになりますが、依存関係用のスクリプトでも（2つのドル記号による）展開の遅延を施す必要があります。

この後、モジュールの各ソースに対して展開を実行しなければなりません。

```
# $(compile-rules)
define compile-rules
  $(foreach f, $(local_src),\
    $(call one-compile-rule,$(call source-to-object,$f),$f))
endef
```

この関数は、module.mkファイルで定義されているlocal_src変数に依存しています。もっと汎用にするには、変数の値であるファイルのリストを引数として渡すことですが、このプロジェクトでは必要ないと思います。これらの関数はmodule.mkファイルで簡単に使えます。

```
local_src := $(subdirectory)/codec.c

$(eval $(call make-library,$(subdirectory)/libcodec.a,$(local_src)))

$(eval $(compile-rules))
```

compile-rules関数は複数行のmakeコードに展開されるので、eval関数を使う必要があります†。

† 訳注：ここでeval関数を使わないと、“*** multiple target patterns.”というエラーになります。このエラーは、

```
ターゲット1: ターゲット2: 必須項目
```

のようにターゲットとコロンのパターンが複数存在しているときに出ます。compile-rulesマクロは結局.oと.dファイルのルールに展開されることになりますが、この2つのルールが1回で解釈されていることを示しています。
4章のevalに関する訳注の繰り返しになりますが、マクロは内部に改行を含めることができる変数なのです。展開された結果はmakefileのトップレベルにおいて改行を含む1つのmakefile要素、つまりルールのターゲット行として解釈されてしまいます。改行を含む非常に長いルール行です。本来別のルールである行も同じルール行として解釈されるので、ここではルール行にターゲットとコロンが2つあるというエラーになったわけです。これは望んだ結果ではありません。そのため、ここでもやはりeval関数に渡して改行が改行として扱われるように構文解析させる必要があります。

最後に1つ複雑な個所があります。もし標準的なCコンパイルルールがバイナリの出力パスに対して適用できなかった場合、暗黙のlexルールや、yaccに対するパターンルールも適用できません。これに対する変更は、手作業で簡単に行うことができます。そのルールはもはや他のlexやyaccファイルに対して適応しないので、lib/db/modue.mkに移動することにします。

```
local_dir := $(BINARY_DIR)/$(subdirectory)
local_src := $(addprefix $(subdirectory)/,playlist.y scanner.l)

$(eval $(call make-library,$(subdirectory)/libdb.a,$(local_src)))

$(eval $(compile-rules))

.SECONDARY: $(call generated-source, $(local_src))

$(local_dir)/scanner.d: $(local_dir)/playlist.d
        $(local_dir)/%.c $(local_dir)/%.h: $(subdirectory)/%.y
        $(YACC.y) --defines $<
        $(MV) y.tab.c $(dir $@)$*.c
        $(MV) y.tab.h $(dir $@)$*.h

$(local_dir)/scanner.c: $(subdirectory)/scanner.l
        @$(RM) $@
        $(LEX.l) $< > $@
```

lexのルールは普通のルールとして定義されていますが、yaccルールはパターンルールになっています。なぜでしょうか？ それはyaccのルールはCファイルとヘッダファイルという2つのターゲットに対して適用されるからです。もし普通のルールとして定義してしまうと、makeはそれぞれCファイルとヘッダファイルを作成するために、コマンドスクリプトを2回実行してしまいます。複数のターゲットに対するパターンルールにより、makeは1回の実行で複数のターゲットを構築できることを理解します。

もし可能であるなら、ここで示したmakefileを使うのではなく、バイナリツリーから実行する簡単な方法を筆者は使います。ここで見てきたように、ソースツリーからのコンパイルを行おうとすると、厄介な問題が（そして悪化する方向に）次々と表面化してしまいます。

8.2 読み出し専用のソース

構築作業により生成され出力ツリーに置かれるファイルがバイナリファイルだけであるなら、ソースとバイナリツリーが分離された後はリファレンスソースツリーを読み出し専用にする機能が多くの場合で必要になります。しかしソースファイルも生成されるなら、そのファイルがバイナリツリーに置かれることに対して配慮しなければなりません。

簡単な「バイナリツリーからのコンパイル」手法なら、yaccやlexはバイナリツリーで実行されるため、生成されるファイルは自動的にバイナリツリーに書き込まれることになります。「ソースツリーからのコンパイル手法」では、明示的にソースとターゲットファイルのパスを（yaccやlexに対して）

指定せざるを得ません。というのもバイナリツリーへのパスを指定しただけでは、makeはそのファイルに対して作業しなければならないことを覚えておく以外は、何もしてくれないからです。

リファレンスソースツリーを読み出し専用にすることは、多くの場合困難を内在しています。古い構築システムの多くはソースツリーにファイルを書き込むような作業を持っています。というのも構築システムの作成者は、ソースツリーを読み出し専用にする利点を考慮しなかったからです。生成されるドキュメント、ログファイル、一時ファイルなどが相当します。これらのファイルをバイナリツリーに移動するのは、場合により困難なので、もし1つのソースから複数のバイナリツリーを作る必要があるなら、代替手段として複数のまったく同じソースツリーを作成し、それらを同期させるという方法を使います。

8.3 依存関係の生成

依存関係の生成については、「2.7 自動的な依存関係の生成」で簡単に触れましたが、いくつかの問題に関してそのままになっています。そこで、ここでは2章の単純な方法に対する代替案をいくつか紹介します†。特に2章やGNU makeのマニュアルで紹介されている手法では以下の問題に悩まされます。

- まず非効率です。依存関係ファイルが存在しないか古いとmakeが判断したときには、.dファイルを更新し、自身を再スタートします。makefileの読み込みや依存関係グラフの解析で多くの作業が必要な場合には、makefileの再読み込みは非効率なものとなります。
- 最初にターゲットを構築するときと新しいソースファイルを加えたときに、makeは警告を出力します。新しいソースファイルに関連づけられた依存関係ファイルがまだ存在していない状態で、makeが（include命令により）依存関係ファイルを読み込もうとしたとき、依存関係ファイルを生成する前に警告を出します。これは大きな問題ではなく、単に不愉快なだけです。
- もしソースファイルを削除すると、その後の構築作業で重大なエラーとなりmakeは終了します。この状況では、削除したファイルを必須項目とする依存関係ファイルが残ることになります。makeは削除されたファイルを見つけることができず、作成する方法もわからないので、次のメッセージ（「foo.dが必要とするfoo.hを作成するルールがありません」の意）を出力します。

```
make: *** No rule to make target foo.h, needed by foo.d. Stop.
```

さらにこのエラーによりmakeは依存関係ファイルを再作成できなくなります。唯一頼れるのは手作業による依存関係の削除ですが、こういったファイルはとても見つけにくいため、通常すべての依存関係ファイルを削除し、全構築をやり直します。このエラーはファイル名を変更したときにも発生します。この問題は.cファイルよりもヘッダファイルを削除したり名前を変えた

† ここで扱う内容の多くはGNU automakeユーティリティのTom Tromey（tromey@cygnus.com）によるものであり、（GNU makeの保守担当である）Paul SmithのWebサイト（http://make.paulandlesley.org）に書かれている秀逸な概括記事からのものです。

ときに顕著です。なぜなら依存関係ファイルのリストは.cファイルのリストから自動的に作られるため問題とはなりません。

8.3.1 Tromeyの手法

これらの問題を個々に解決しましょう。まず、どのようにmakeの再スタートを抑制したらよいのでしょう。

よくよく考えると、makeの再スタートは必要ないことがわかります。依存関係ファイルの更新は、少なくとも1つの必須項目が変更されたことを示しており、すなわちターゲットを更新しなければならないことを意味します。これ以上の依存関係に関する情報はmakeのふるまいを変更しないので、makeの実行に際してこれ以上知る必要はありません。しかしmakeを次に実行した際に完全な依存情報を得られるように、依存関係ファイルは更新されなければなりません。

今回の実行では依存関係ファイルを必要としないので、ファイルの生成はターゲットの更新と同じタイミングで行うことができます。これはコンパイルのルールを変更して依存関係ファイルの更新も同時に行うようにすることで達成できます。

```
# $(call make-depend,source-file,object-file,depend-file)
define make-depend
  $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $1 | \
  $(SED) 's,\($$(notdir $2)\) *:,$$(dir $2) $3: ,' > $3.tmp
  $(MV) $3.tmp $3
endef

%.o: %.c
        $(call make-depend,$<,$@,$(subst .o,.d,$@))
        $(COMPILE.c) -o $@ $<
```

依存関係の生成機能は、ソース、オブジェクト、依存関係ファイルの名前を受け付けるmake-depend関数として実現されていて、別の状況でこの機能を必要とする場合に対する最大限の柔軟性を提供しています。このようにコンパイルルールを変更したので、依存関係ファイルを2回生成しないよう%.d:%cパターンルールを削除しなければなりません。

これでオブジェクトファイルと依存関係ファイルは、一方が存在していれば、もう一方も存在しなければならないという論理的なつながりを持つようになりました。そのため、依存関係ファイルが存在しない場合のことを心配する必要がなくなります。もし依存関係ファイルが存在していなければ、オブジェクトファイルも存在しないことになり、次回の構築で双方が作成されることになります。もう.dファイルが存在していないことによるあらゆる警告を無視することができるようになりました。

「3.7.2 includeと依存関係」で、エラーを無視して警告を出さないための-include（またはsinclude）という形式のinclude命令を紹介しました。

```
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(dependencies)
endif
```

これは2番目の問題、つまり依存関係ファイルが存在していない場合の気に障るメッセージを解決します。

そして必須項目が存在しないことが明らかになったことを示す警告を、ちょっとしたトリックを使って排除することができます。そのトリックとは存在しないファイルを必須項目もコマンドも持たないターゲットとすることです。例えば依存関係ファイル作成機能が、次のような依存関係を生成したとしましょう。

```
target.o target.d: header.h
```

このとき、コードのリファクタリングによりheader.hが必要なくなり削除されたとします。その後makeを実行すると次のようなエラー（「target.dが必要とするheader.hを作成するルールがありません」の意）になります。

```
make: *** No rule to make target header.h, needed by target.d. Stop.
```

ところが、header.hをコマンドを持たないターゲットとして依存関係ファイルに追加すれば、このようなエラーは起きません。

```
target.o target.d: header.h
header.h:
```

header.hが存在しなければ、単にそれが最新ではなく、header.hを必須項目とするターゲットが再構築されなければならないことを示すだけです。そこで、もはや参照されていないheader.hを含まない依存関係ファイルが再生成されます。もしheader.hが存在していれば、それが最新であり、処理を続行しようとmakeは考えます。ということから、行うべきことはすべての必須項目が空のルールと関連づけられていることを確実にすることだけです。次に示すのは、新しいターゲットを加えるよう変更したmake-depend関数です。

```
# $(call make-depend,source-file,object-file,depend-file)
define make-depend
  $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $1 | \
  $(SED) 's,\($$(notdir $2)\) *:,$$(dir $2) $3: ,' > $3.tmp
  $(SED) -e 's/#.*//'         \
         -e 's/^[^:]*: *//'   \
         -e 's/ *\\$$$$//'    \
         -e '/^$$$$/ d'       \
         -e 's/$$$$/ :/' $3.tmp >> $3.tmp
  $(MV) $3.tmp $3
endef
```

追加のルールを生成するために、依存関係ファイルをsedコマンドに通しています。ここでのsedコードは次の変換を行います。

1. コメントの削除
2. ターゲットファイル名とそれに続く空白の削除
3. 行末にある空白列の削除
4. 空白行の削除
5. 各行の最後にコロンを追加

（GNU sedは単一のコマンドで、読み込むファイルに対する追加を行うことができるので、2番目の一時ファイルを省略することができます。この機能は他のシステムでは使うことができないかもしれません）。このsedコマンドは、入力として次のようなものを受け取ります。

```
# 何かコメント
target.o target.d: prereq1 prereq2 prereq3 \
    prereq4
```

そして次のように変換します。

```
prereq1 prereq2 prereq3:
prereq4:
```

make-depend関数はこの内容を元の依存関係ファイルに追加します。これにより“No rule to make target”（「作成するためのルールがありません」の意）というエラーに関する問題は解決します。

8.3.2 makedependプログラム

ここまではたいていのコンパイラが提供している-Mオプションの機能に満足してきましたが、もしこのオプションが使えない場合はどうすればよいのでしょう。あるいは-Mよりももっとよいオプションが使える場合にはどうしましょう。

最近ではほとんどのCコンパイラがソースファイルから依存関係を生成する機能を提供していますが、そう遠くない昔はそうではありませんでした。X Window Systemプロジェクトの初期段階ではCとC++のソースから依存関係を導出するmakedependというツールが作成されました。このツールはインターネットからダウンロードして自由に使うことができます。makedependは出力をmakefileへ追加してしまうという我々の望まない動作をするため、使用にあたっては多少格好の悪い方法を採用しなければなりません。またmakedependの出力は、オブジェクトファイルがソースファイルと同じディレクトリに存在していることを仮定しています。そのためsedコードを再度変更しなければなりません。

```
# $(call make-depend,source-file,object-file,depend-file)
define make-depend
  $(MAKEDEPEND) -f- $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) $1 | \
  $(SED) 's,^.*/\([^/]*\.o\) *:,$(dir $2)\1 $3: ,' > $3.tmp
  $(SED) -e 's/#.*//'                                          \
         -e 's/^[^:]*: *//'                                    \
```

```
          -e 's/ *\\$$$$//'                                          \
          -e '/^$$$$/ d'                                             \
          -e 's/$$$$/ :/' $3.tmp >> $3.tmp
          $(MV) $3.tmp $3
endef
```

makedependが結果を標準出力に出力するよう-f-オプションを使います。

makedependや実際にプロジェクトで使うコンパイラ†に代わる手段としてgccがあります。gccは依存関係を生成するオプションをあきれるほど持っています。現在の要件に最も適している設定の1つは次のようになります。

```
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(dependencies)
endif

# $(call make-depend,source-file,object-file,depend-file)
define make-depend
  $(GCC) -MM            \
         -MF $3         \
         -MP            \
         -MT $2         \
         $(CFLAGS)      \
         $(CPPFLAGS)    \
         $(TARGET_ARCH) \
         $1
endef

%.o: %.c
        $(call make-depend,$<,$@,$(subst .o,.d,$@))
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

-MMオプションは「システム」のヘッダを必須項目からはずします。そういったファイルはめったに変更されませんし、構築システムが複雑化すると不要なものを削減することは意味があります。本来はパフォーマンスの理由から導入されたものです。現代的なプリプロセッサでは、実行速度の違いはほとんど変わりません。

-MFオプションでは依存関係ファイルの名前を指定します。これはオブジェクトファイル名の.oを.dに変えたものです。gccには同様の変換を行って出力ファイル名を自動的に決定する-MDや-MMDといったオプションがあります。こうしたオプションを使うのが理想的なのですが、変換を行う際にオブジェクトファイルディレクトリへの適切な相対パスを含めることができず、.dファイルを現在のディレクトリに置いてしまいます。そのため、この作業を-MFを使って自前で行うのです。

-MPオプションは各必須項目に対して擬似ターゲットを置くよう指示するものです。これによりmake-depend関数から複雑な5段階にもわたるsedコードを完全に排除することができます。おそ

† 訳注：gccももちろんコンパイラの1種ですが、ここではgcc以外のプラットフォームに特化したコンパイラをプロジェクトで使用していると仮定しています。原文は“native compiler”です。

らく擬似ターゲットのテクニックを考案したautomakeの開発者が、gccにこのオプションを加えるよう働きかけたのだと思います。

-MTオプションは依存関係ファイルのターゲットとして使う文字列を指定します。繰り返しになりますが、このオプションを使わないとgccはオブジェクトファイルディレクトリへの相対パスを含めることができません。

gccを使うことで、以前は依存関係生成で必要だった4つのコマンドを1つに削減することができました。たとえ特殊なコンパイラを使っていたとしても、依存関係管理のためにgccを使えます。

8.4　複数のバイナリツリーを利用する

バイナリファイルを別のツリーに置くようにmakefileを変更したなら、多くのバイナリツリーを使うのは非常に簡単になります。開発者がキーボードから入力して手動で行う構築作業でも、事前の準備は必要ないかあってもごくわずかです。開発者は出力ディレクトリを作成し、cdでそこに移動した後、makefileを実行します。

```
$ mkdir -p ~/work/mp3_player_out
$ cd ~/work/mp3_player_out
$ make -f ~/work/mp3_player/makefile
```

もし作業がこれ以上複雑なら、シェルスクリプトを用意するのが最善の方法です。シェルスクリプトではカレントディレクトリを解析しmakefileから使えるようにBINARY_DIRのような環境変数に値をセットします。

```
#! /bin/bash

# 現在ソースディレクトリにいると仮定
curr=$PWD
export SOURCE_DIR=$curr
while [[ $SOURCE_DIR ]]
do
  if [[ -e $SOURCE_DIR/[Mm]akefile ]] †
  then
    break;
  fi
  SOURCE_DIR=${SOURCE_DIR%/*}
done

# makefileが存在していなければエラーを表示
if [[ ! $SOURCE_DIR ]]
then
  printf "run-make: Cannot find a makefile" > /dev/stderr
```

† 訳注：bashのmanpageによると、[[と]]の間ではパス名展開が行われないため、上記スクリプトの中でif [[-e $SOURCE_DIR/[Mm]akefile]]による判断がうまく働きません。ここはif [-e $SOURCE_DIR/[Mm]akefile]にする必要があります。ただしCygwinのbashでは上記スクリプトでも意図したとおりに動作するようです。

```
  exit 1
fi

# もし出力ディレクトリが設定されていなければ、規定値を設定
if [[ ! $BINARY_DIR ]]
then
  BINARY_DIR=${SOURCE_DIR}_out
fi

# 出力ディレクトリを作成
mkdir --parents $BINARY_DIR

# makeを実行
make --directory="$BINARY_DIR" "$@"
```

このスクリプトは多少手が込んでいます。まず最初にカレントディレクトリから始めてmakefileが見つかるまでディレクトリを上にたどります。次にバイナリツリーを表す変数が設定されているかをチェックします。設定されていなければソースディレクトリに“_out”を付加した値を設定します。そして出力ディレクトリを作成しmakeを実行します。

もし異なるプラットフォームに対する構築作業も行う必要があるなら、プラットフォームを識別する手段が必要となります。最も簡単なのは、開発者が各プラットフォームを表す値を環境変数に設定しmakefileやソースに条件判断を加えるという方法です。もっとよい方法では、unameの出力を使って自動的に値を設定します。

```
space := $(empty) $(empty)
export MACHINE := $(subst $(space),-,$(shell uname -smo))
```

構築作業がcronから自動で実行される場合、シェルスクリプトを使ったほうがcronでmakeを直接起動するよりもよい手法だといえます。シェルスクリプトを使うことにより事前準備やエラーからの回復、そして自動構築の後始末などで必要な機能を提供することができます。またスクリプトは変数の設定やコマンドラインパラメータを置く場所としても適しています。

あらかじめ決められた分の出力ツリーやプラットフォームだけをプロジェクトがサポートするのならば、現在の構築作業を識別するのにディレクトリ名を使うことができます。例を示しましょう。

```
ALL_TREES := /builds/hp-386-windows-optimized \
             /builds/hp-386-windows-debug     \
             /builds/sgi-irix-optimzed        \
             /builds/sgi-irix-debug           \
             /builds/sun-solaris8-profiled    \
             /builds/sun-solaris8-debug

BINARY_DIR := $(foreach t,$(ALL_TREES),       \
                $(filter $(ALL_TREES)/%,$(CURDIR)))

BUILD_TYPE := $(notdir $(subst -,/,$(BINARY_DIR)))

MACHINE_TYPE := $(strip                       \
```

```
$(subst /,-,                    \
  $(patsubst %/,%,              \
    $(dir                       \
      $(subst -,/,              \
        $(notdir $(BINARY_DIR))))))
```

ALL_TREES変数は有効なバイナリツリーのリストを保持します。foreachループでカレントディレクトリと合致したものを取り出します。合致するのは1つだけです。バイナリツリーが識別されたなら、構築種別（例えば、最適化（optimized）、デバッグ（debug）、性能評価（profiled）など）をディレクトリ名から抜き出します。ここではマイナス記号区切りの単語列をスラッシュ区切りの単語列に変換し、notdir関数を使うことで最後の単語を取り出しています。同様にディレクトリ名の最後の部分に対して同じ手法を適用することでダッシュ区切りの最後の部分を取り除きマシンタイプとしています。

8.5 部分的なソースツリー

本当に大きなプロジェクトでは、ソースをチェックアウトして保守するのは開発者にとって重荷となってしまいます。もしシステムが多くのコンポーネントから構成されていて、特定の開発者がその一部だけを変更するような状況では、プロジェクトの全体をチェックアウトしてコンパイルするのは時間の大きな無駄です。そこで管理された構築を毎晩行い、開発者の持つソースやバイナリツリーの足りない部分を補うために使います。

このような場合、2種類の検索機能が必要になります。まず足りないヘッダファイルをコンパイラが必要としたときにリファレンスソースツリーを探すように指示しなければなりません。次に欠けているライブラリをmakefileが必要としたときにリファレンスバイナリツリーを探すよう指示する必要があります。コンパイラがソースを探す手助けをするには、開発者のディレクトリを指す-Iオプションの後に-Iオプションを単に追加するだけです。makeがライブラリを探す手助けをするにはvpathにディレクトリを追加します。

```
SOURCE_DIR := ../mp3_player
REF_SOURCE_DIR := /reftree/src/mp3_player
REF_BINARY_DIR := /binaries/mp3_player
…
include_dirs := lib $(SOURCE_DIR)/lib $(SOURCE_DIR)/include
CPPFLAGS += $(addprefix -I ,$(include_dirs)) \
            $(addprefix -I $(REF_SOURCE_DIR)/,$(include_dirs))
vpath %.h $(include_dirs) \
          $(addprefix $(REF_SOURCE_DIR)/,$(include_dirs))
vpath %.a $(addprefix $(REF_BINARY_DIR)/lib/, codec db ui)
```

この手法はCVSのチェックアウト粒度がライブラリやプログラムモジュール単位であることを想定しています。この場合、開発者がチェックアウトしていなければ、そういったライブラリやプログラムをスキップするようにmakeを仕向けることが可能です。ライブラリを必要とする時期がくれば、検索

パスに足りないファイルを自動的に埋め込みます。

このmakefileではmodules変数にmodule.mkファイルを探し出すサブディレクトリのリストが入っています。そのサブディレクトリがチェックアウトされていなければ、リストにはそのサブディレクトリが入らないように修正されなければなりません。そうする代わりにワイルドカードを使って設定することができます。

```
modules := $(dir $(wildcard lib/*/module.mk))
```

この式はmodule.mkの存在するサブディレクトリを見つけ出し、そのリストを返します。ここで留意すべきなのは、dir関数の働きにより各ディレクトリには最後にスラッシュが付くことです。

部分的なソースツリーを個々のファイル単位で管理することで、いくつかのオブジェクトファイルを開発者のディレクトリから、足りないものをリファレンスツリーから取り出してライブラリを構築することも可能です。しかしながら、これは非常に煩雑であり、筆者の経験からいうと開発者が満足するような手法ではありません。

8.6 リファレンスビルド、ライブラリ、インストーラ

リファレンスビルドを作成するために必要なことはほとんど説明してきました。この機能を実現するために行うトップレベルmakefileの変更は直感的です。単にSOURCE_DIRとBINARY_DIRの代入を?=に変更します。cronから実行するスクリプトではこの基本的な手法を使えます。

1. ログファイル名を設定し出力をリダイレクトします。
2. 前回の構築作業の後片づけを行いリファレンスソースツリーをきれいにします。
3. 最新のソースをチェックアウトします。
4. ソースとバイナリディレクトリを表す変数を設定します。
5. makeを実行します。
6. エラーが起きていないかログを検査します。
7. tagsファイルを生成し、場合によってファイル位置データベース†を更新します。
8. 構築が成功したか失敗したかを報告します。

リファレンスビルドを使う環境では、誤ったチェックインによりソースツリーが壊れてしまう場合に備えて古いビルドを保持しておくのが便利です。筆者は毎晩行う構築の通常7から14日分を保存しておきます。もちろん構築用スクリプトは、その記録を構築ツリー近くのファイルに出力し、古い構築と記録を削除します。ログを検査してエラーの有無を調べるのにawkスクリプトをよく用います。そして、筆者は最新版へのシンボリックリンクを作るスクリプトもたいていは用意します。構築が成功

† ファイル位置データベース（locate database）はファイルシステム上にあるすべてのファイル名を収集したものです。ファイル名でfindを実行するのが簡単に作る方法です。これは大きなソースツリーを管理するときに非常に役に立つことがわかっています。毎晩の構築作業が終了した後で更新するのが適しています。

したかチェックするために、各makefileにvalidate（検証）ターゲットを作ります。このターゲットはターゲットが構築されたか簡単な検証を行います。

```
.PHONY: validate_build
validate_build:
        test $(foreach f,$(RELEASE_FILES),-s $f -a) -e .
```

このコマンドスクリプトは、期待されるいくつかのファイルが存在していて中身があるかを調べます。もちろんこれはテストの代わりに使うことはできませんが、構築作業の健全性をチェックする便利な手段です。もしチェックが失敗したならmakeはエラーを返し、構築用のスクリプトはシンボリックリンクを古い構築を指したままにします。

サードパーティが提供しているライブラリは管理するのが多少厄介です。筆者は一般的にいわれている、CVSに大きなバイナリファイルを保存することはよくない、という意見に賛成します。これはCVSが前のバイナリファイルとの差分を保持できず、基盤としているRCSのファイルが非常に大きくなってしまうからです。

CVSリポジトリ内に巨大なファイルがあると、一般的なCVSの操作が遅くなってしまい、すべての開発作業に影響を及ぼしてしまいます。

もしサードパーティ提供のライブラリをCVSに保存しないのなら、別の方法で管理しなければなりません。筆者が現在望ましい方法だと思っているのは、図8-1に示すようにリファレンスツリーにライブラリディレクトリを作成し、ライブラリのバージョン番号をディレクトリ名として記録することです。

これらのディレクトリ名はmakefileから参照されます。

```
ORACLE_9011_DIR ?= /reftree/third_party/oracle-9.0.1.1/Ora90
ORACLE_9011_JAR ?= $(ORACLE_9011_DIR)/jdbc/lib/classes12.jar
```

ライブラリ提供者がライブラリを更新したときは新しいディレクトリをリファレンスツリー内に作成し新しい変数をmakefileに用意します。このようにCVSタグやブランチで適正に保守されているmakefileは、使っているライブラリのバージョンを明示的に反映することになります。

インストーラにも同じような難しい問題があります。基礎的な構築作業とインストールイメージの

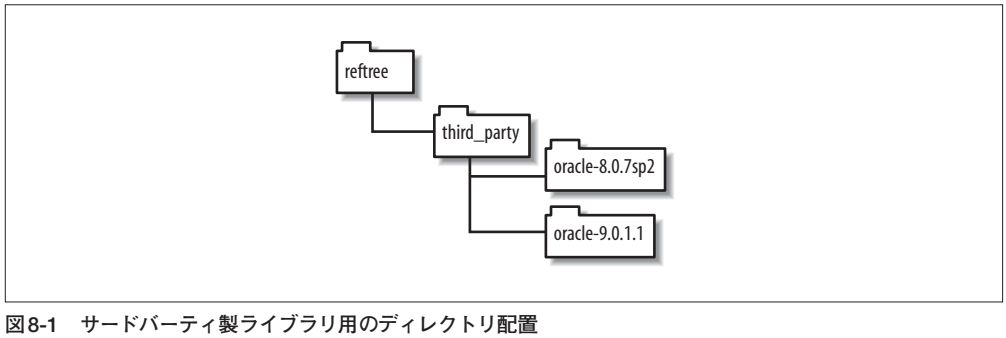

図8-1　サードバーティ製ライブラリ用のディレクトリ配置

作成を分離することはよいことだと筆者は考えています。現代的なインストーラツールは複雑なうえに扱いにくくなっています。それを（同じように複雑で扱いにくい）構築システムの中に組み込むことで、維持するのが難しいシステムを作り出してしまいます。代わりに基礎的な構築ではインストーラツールが必要とするデータがすべて入っているリリースディレクトリに結果を収めるだけにします。そしてインストーラーツールは最終的に実行可能なインストールイメージを作る別のmakefileから起動するようにします。

9章
Java

多くのJava開発者はEclipseのような統合開発環境（Integrated Development Environment：IDE)を好みます。Java IDEやAntなどがあるにもかかわらず、なぜJavaプロジェクトに対してmakeを使うことを考えなければならないのか疑問に思うことでしょう。本章では、同時にどんなJavaプロジェクトでも最小の修正で標準的な構築作業を行うことができるような汎用のmakefileを示し、そのような状況下でmakeを使うことの価値について論じていきます。

Javaに対してmakeを使うことでいくつかの問題が浮かび上がると同時に、いくつかの利点も明らかにします。これは主として次の3つの要因を背景としています。

- Javaコンパイラであるjavacは非常に高速に動作する
- 標準javaコンパイラはコマンドラインパラメータを@filename形式でファイルから読み込むことができる
- Javaのパッケージが指定されれば自動的に.classファイルへのパスが明らかになる

標準Javaコンパイラは非常に高速です。これは主にimport命令の働きのおかげです。C言語における#includeと同様に、この命令は外部で定義されたシンボルへの参照を行うために使います。しかしながら、構文解析や分析をそのつど行わなければならないソースコードを読み込むのではなく、classファイルを直接読み取ります†。classファイル内のシンボルはコンパイル中には変化しないので、コンパイラはclassファイルの内容をキャッシュすることができます。中程度のプロジェクトでは、Cと比べて何百万行ものソースコードを読み込んで構文解析を行い分析する作業を排除できます。地味ではありますが、コンパイラで実行される最適化が最小限であることもパフォーマンス向上に貢献しています。代わりにJavaは洗練されたジャストインタイム（Just-in-time：JIT）方式の最適化をJava仮想マシン (Java Virtual Machine：JVM）自身で行います。

非常に大きなJavaプロジェクトでは、Javaのpackage機能を広範囲に利用します。ファイル内で定義されたシンボルの有効範囲（scope）を形成するpackageの中にJavaのクラスはカプセル化され

† 訳注：Javaのソースをコンパイルすると、.classサフィックスを持つclassファイルができます。

ます。packageの名前は階層構造を持ち、自動的にファイルの階層構造を定義します。例えば、a.b.cというpackageは自動的にa/b/cというディレクトリ構造を持ちます。package a.b.cの中で宣言されたコードはコンパイルされるとa/b/cディレクトリの中のclassファイルとなります。これはつまりmakeが標準で行うソースファイルとバイナリファイルとの関連づけが使えないことを意味しています。しかし同時に出力ファイルをどこに置くべきかを示すための-oオプションを使わなくてもよいことになります。すべてのファイルで同じものを出力ファイルのルートとして示すだけでよいのです。同様に、異なるディレクトリに存在するソースファイルを1つの呼び出し形式でコンパイルできることも意味しています。

標準のJavaコンパイラはすべて@filename形式を使ってコマンドラインパラメータをファイルから読み込むことができます。プロジェクト内すべてのJavaソースを1回でコンパイルできるため、この機能はpackage機能とあわせて非常に重要な意味を持っているといえます。コンパイラ自身を読み込んで実行するの要する時間は構築時間の大きな部分を占めるので、この機能は非常に大きくパフォーマンス向上に寄与します。

簡単にいうと、適切にコマンド行を構成することにより400,000行ものJavaをコンパイルするのに要する時間は2.5GHzのPentium4プロセッサを使って3分ほどになります。同程度のC++だとおそらく数時間は必要となるでしょう。

9.1 makeの代替

冒頭で述べたようにJavaの開発者コミュニティは新しい技術を取り上げるのに熱心です。その中でAntとIDEについてmakeとどのような関係があるのかを見ることにしましょう。

9.1.1 Ant

Javaコミュニティは新しいツールやAPIを作り出すのに非常に積極的です。それも驚くべき速さで作り出します。そのうちの1つがJava開発プロセス中でmakeの代わりとなるべく作られた構築ツールAntです。makeと同様にAntはプロジェクトのターゲットと必須項目について記述したビルドファイルを使用します。一方makeと異なる点は、AntはJavaで書かれており、Antのビルドファイルは XMLで書かれていることです。

XMLによるビルドファイルの雰囲気をつかむため、Antのビルドファイルから少し引用してみます。

```
<target name="build"
        depends="prepare, check_for_optional_packages"
        description="--> compiles the source code">
  <mkdir dir="${build.dir}"/>
  <mkdir dir="${build.classes}"/>
  <mkdir dir="${build.lib}"/>

  <javac srcdir="${java.dir}"
         destdir="${build.classes}"
         debug="${debug}"
         deprecation="${deprecation}"
```

```
        target="${javac.target}"
        optimize="${optimize}" >
    <classpath refid="classpath"/>
  </javac>
  …
  <copy todir="${build.classes}">
    <fileset dir="${java.dir}">
      <include name="**/*.properties"/>
      <include name="**/*.dtd"/>
    </fileset>
  </copy>
</target>
```

ご覧のように、XMLの<target>タグによりターゲットは示されています。各ターゲットはname属性とdepends属性により示される名前（name）と依存関係（dependency）を持ちます。Antのタスク（task）によりコマンドは実行されます。タスクはJavaで書かれXMLのタグに関連づけられています。例えばディレクトリを作成するタスクは<mkdir>タグで指定され、Mkdir.executeというJavaメソッドを実行します、これは最終的にFile.mkdirを実行することになります。可能なかぎりすべてのタスクはJava APIを使って作られています。

これと同等の内容をmakeの文法で書くと次のようになります。

```
# ソースをコンパイルする
build: $(all_javas) prepare check_for_optional_packages
        $(MKDIR) -p $(build.dir) $(build.classes) $(build.lib)
        $(JAVAC) -sourcepath $(java.dir)                          \
                 -d $(build.classes)                              \
                 $(debug)                                         \
                 $(deprecation)                                   \
                 -target $(javac.target)                          \
                 $(optimize)                                      \
                 -classpath $(classpath)                          \
                 @$<
        …
        $(FIND) . \( -name '*.properties' -o -name '*.dtd' \) | \
        $(TAR) -c -f - -T - | $(TAR) -C $(build.classes) -x -f -
```

ここでは本書でまだ扱っていないテクニックを使っています。all_javas変数の値であるall.javasはコンパイルされるすべてのJavaファイルのリストが入っているファイルであるということを知っていれば今のところ十分です。Antタスクである<mkdir>、<javac>、<copy>は依存関係チェックも同様に行います。つまり、ディレクトリがすでに存在していれば、<mkdir>は何も行いません。同様にclassファイルがソースファイルよりも新しければコンパイルは行われません。とはいってもmakeのコマンドスクリプトも基本的には同じ機能を提供しています。またAntはプログラムを実行するための汎用タスクである<exec>も提供しています。

Antは構築ツールにとして巧妙で革新的な取り組みといえますが、考慮すべきいくつかの課題があります。

- AntはJavaコミュニティでは広く受け入れられていますが、その他ではあまり知られていません。また（ここで述べられる理由により）Java以上に広まるかどうかも懐疑的です。一方makeはこれまでも多くの場面で利用し続けられてきました。いくつか例をあげれば、ソフトウェア開発、文書処理、Webサイトやワークステーションの保守などです。makeを理解することは、広範囲なソフトウェアシステム上で仕事をするうえで重要なのです。
- 記述言語としてXMLを選択することはJavaをベースとしたツールでは適切であるといえます。しかしXMLは（多くの者にとって）特に読み書きという点において快適でありません。適切なXMLエディタを見つけるのは難しく、多くの場合で既存のツールとの統合もうまくできません（IDEがよいXMLエディタを統合しているか、IDEを捨てて別のツールを使わざるを得ないかどちらかです）。前述の例で見たように、XMLというかAnt方言はmakeやシェルの文法と比べて冗長です。またXMLは多くの特異性に満ちています。
- Antのビルドファイルを書く際には、ある種のあいまいさと格闘しなければなりません。Antの<mkdir>タスクはシステムのmkdirプログラムを起動しません。代わりにjava.io.Fileクラスのmkdir()メソッドを実行します。これは期待した動作かもしれませんし、そうでないかもしれません。プログラマが持っている一般的なツールがどのように動作するのかに関する知識は、Antに対しては本質的に怪しいので、Antのドキュメント、Javaのドキュメント、Antのソースを調べなければなりません。加えて、例えばJavaのコンパイラを実行する場合に、コンパイラのマニュアルには出ていないsrcdir、debugなどといったXMLの属性をいくつも使わなければなりません。一方makeのスクリプトは非常に率直です、つまりシェルに対するコマンドを普通はそのとおりに記述することができます。
- makeがそうであるようにAntもポータビリティがあります。7章で見てきたように、ポータブルなmakefileを書くことはポータブルなAntのビルドファイルを書くことに似て、経験と知識が欠かせません。ポータブルなmakefileを書くという作業は20年にも渡り続けられてきました。さらにポータビリティにかかわる問題としてUnixのシンボリックリンクおよびWindowsのロングファイルネームがあること、AppleのOSとしてMacOS Xしかサポートしていないこと、その他のプラットフォームでのサポートは保証していないことなどがAntのドキュメントには書かれています。またファイルに実行ビットを立てるといった基本的な操作はJava APIでは提供されておらず外部プログラムを使う必要があります。ポータビリティの向上は一筋縄ではいきません。
- Antは現在どういった作業を行っているのかを正確には表示しません。Antタスクはたいていの場合シェルのコマンドを実行するようには作られていないので、Antでは作業の内容を表示するのが難しいのです。タスクの作者がprint文を使って文章として画面へ出力するのが普通です。それに対してmakeが表示する内容は、利用者がコピーしてシェルから再実行する際にも使うことができます。つまり開発者が構築処理内容や使われるツールを理解しようと試みる際に、Antはあまり有益だとはいえません。さらに、作業の一部を再利用して緊急にキーボードから実行することもできません。
- 最も重要な点ですが、Antはスクリプト言語から非スクリプト言語へと構築の枠組みを変化さ

せます。Antのタスクは Javaで書かれます。タスクが存在しないとか希望の動作ではなかった場合には、独自のタスクを Javaで書くか<exec>タスクを使わなければなりません（もちろん<exec>タスクを頻繁に使うのなら、単純にmakeをマクロや関数などとともに簡潔な表現で使用したほうがずっと優れています）。一方スクリプト言語はこういった課題に対応するために考案され広く使われています。makeは30年近く存在し最も複雑な状況に対しても実装を変更せずに使われてきました。もちろんわずかばかりの拡張がこの30年の間に行われましたが、その多くはGNU makeで着想され実装されてきたものです。

Antは Javaコミュニティでは広く受け入れられているすばらしいツールです。しかし新しいプロジェクトに着手する前にはAntがその開発環境に適しているのか、慎重に検討しなければなりません。makeは Javaの構築に対しても非常に適していると、本章で立証できることを願っています。

9.1.2 IDE

エディタ、コンパイラ、デバッガ、コードブラウザなどが（たいていは）グラフィカルでまとまった環境に組み込まれている統合開発環境（Integrated Development Environment：IDE）を Java開発者の多くは使用しています。オープンソースの Eclipse（http://www.eclipse.org）や Emacs JDEE（http://jdee.sunsite.dk）など、商用のもので Sun Java Studio（http://wwws.sun.com/software/sundev/jde）や JBuilder（http://www.borland.com/jbuilder）などがあげられます。こういった開発環境では普通必要なファイルをコンパイルしアプリケーションが実行できるようにする、プロジェクトの構築という概念を持っています。

IDEがこうしたことを全部やってくれるのなら、なぜmakeの使用を考慮しなければならないのでしょうか。最も明白な理由はポータビリティです。もしプロジェクトの構築作業を別のプラットフォーム上で行う必要があるとき、その新しいプラットフォームで行う構築は失敗してしまうでしょう。Java自身は多くのプラットフォーム間でポータブルですが、たいていのサポートツールはそうではありません。例えばプロジェクトの設定ファイルが Unixまたは Windows形式のパス名を含んでいた場合、別の OS上で構築作業を行うとエラーになってしまうでしょう。makeを使うもう1つの理由は、無人構築です。IDEのいくつかはバッチ形式の構築機能を持っていますが、持っていないものもあります。この機能をどのように提供しているかもさまざまです。最後の理由として、組み込まれている構築機能にはたいてい制限があります。リリースディレクトリを独自の構造にする、外部アプリケーションからヘルプファイルを統合する、テストの自動化を組み込む、枝分かれたしたソースをもとに並行開発を行う、などを試みると組み込まれた構築機能では不十分であることに気づくでしょう。

筆者の経験では、IDEは小さい規模か内輪の開発には適していますが、製品の構築ではmakeが提供しているようなもっと総合的な機能が必要となります。筆者は通常デバッグ用のコードを IDEで作成し、製品構築とリリース用のmakefileを作成します。開発の期間中、デバッグを行う局面までは IDEを使ってプロジェクトのファイルをコンパイルします。しかし多くのファイルを変更した場合や、コード生成の入力となるファイルを変更した際にはmakefileを実行します。筆者が使用した IDEでは外部のソースコード生成ツールをうまく扱うことができませんでした。一般的に IDEでは内部や外部の利用者に対するリリースの構築に適していません。筆者はそのような作業ではmakeを使用します。

9.2 Java向け汎用makefile

例9-1はJava向けの汎用makefileです。本章の中で各部分について説明します。

例9-1 Java向けの汎用makefile

```
# Javaプロジェクト向け汎用makefile

VERSION_NUMBER := 1.0

# ディレクトリツリーの位置
SOURCE_DIR := src
OUTPUT_DIR := classes

# Unixツール
AWK := awk
FIND := /bin/find
MKDIR := mkdir -p
RM := rm -rf
SHELL := /bin/bash

# サポートツールへのパス
JAVA_HOME := /opt/j2sdk1.4.2_03
AXIS_HOME := /opt/axis-1_1
TOMCAT_HOME := /opt/jakarta-tomcat-5.0.18
XERCES_HOME := /opt/xerces-1_4_4
JUNIT_HOME := /opt/junit3.8.1

# Javaツール
JAVA := $(JAVA_HOME)/bin/java
JAVAC := $(JAVA_HOME)/bin/javac

JFLAGS := -sourcepath $(SOURCE_DIR)          \
          -d $(OUTPUT_DIR)                   \
          -source 1.4

JVMFLAGS := -ea                              \
            -esa                             \
            -Xfuture

JVM := $(JAVA) $(JVMFLAGS)

JAR := $(JAVA_HOME)/bin/jar
JARFLAGS := cf

JAVADOC := $(JAVA_HOME)/bin/javadoc
JDFLAGS := -sourcepath $(SOURCE_DIR)         \
           -d $(OUTPUT_DIR)                  \
           -link http://java.sun.com/products/jdk/1.4/docs/api

# Jars
COMMONS_LOGGING_JAR := $(AXIS_HOME)/lib/commons-logging.jar
LOG4J_JAR := $(AXIS_HOME)/lib/log4j-1.2.8.jar
```

```
XERCES_JAR := $(XERCES_HOME)/xerces.jar
JUNIT_JAR := $(JUNIT_HOME)/junit.jar

# Javaクラスパスを設定する
class_path := OUTPUT_DIR                    \
              XERCES_JAR                    \
              COMMONS_LOGGING_JAR           \
              LOG4J_JAR                     \
              JUNIT_JAR

# space - ただの空白
space := $(empty) $(empty)

# $(call build-classpath, variable-list)
define build-classpath
  $(strip \
    $(patsubst :%,%,                        \
      $(subst : ,:,                         \
        $(strip                             \
          $(foreach j,$1,$(call get-file,$j):)))))
endef

# $(call get-file, variable-name)
define get-file
  $(strip                                       \
    $($1)                                       \
    $(if $(call file-exists-eval,$1),,          \
      $(warning The file referenced by variable \
                '$1' ($($1)) cannot be found)))
endef

# $(call file-exists-eval, variable-name)
define file-exists-eval
  $(strip \
    $(if $($1),,$(warning '$1' has no value))   \
    $(wildcard $($1)))
endef

# $(call brief-help, makefile)
define brief-help
  $(AWK) '$$1 ~ /^[^.][-A-Za-z0-9]*:/                   \
         { print substr($$1, 1, length($$1)-1) }' $1 | \
  sort |                                               \
  pr -T -w 80 -4
endef

# $(call file-exists, wildcard-pattern)
file-exists = $(wildcard $1)

# $(call check-file, file-list)
define check-file
  $(foreach f, $1,
    $(if $(call file-exists, $($f)),,           \
      $(warning $f ($($f)) is missing)))
```

```
endef

# $(call make-temp-dir, root-opt)
define make-temp-dir
  mktemp -t $(if $1,$1,make).XXXXXXXXXX
endef

# MANIFEST_TEMPLATE - m4マクロプロセッサの入力となるマニフェストファイル
MANIFEST_TEMPLATE := src/manifest/manifest.mf
TMP_JAR_DIR := $(call make-temp-dir)
TMP_MANIFEST := $(TMP_JAR_DIR)/manifest.mf

# $(call add-manifest, jar, jar-name, manifest-file-opt)
define add-manifest
  $(RM) $(dir $(TMP_MANIFEST))
  $(MKDIR) $(dir $(TMP_MANIFEST))
  m4 --define=NAME="$(notdir $2)"                \
     --define=IMPL_VERSION=$(VERSION_NUMBER)     \
     --define=SPEC_VERSION=$(VERSION_NUMBER)     \
     $(if $3,$3,$(MANIFEST_TEMPLATE))            \
     > $(TMP_MANIFEST)
  $(JAR) -ufm $1 $(TMP_MANIFEST)
  $(RM) $(dir $(TMP_MANIFEST))
endef

# $(call make-jar,jar-variable-prefix)
define make-jar
  .PHONY: $1 $$($1_name)
  $1: $($1_name)
  $$($1_name):
        cd $(OUTPUT_DIR);                        \
        $(JAR) $(JARFLAGS) $$(notdir $$@) $$($1_packages)
        $$(call add-manifest, $$@, $$($1_name), $$($1_manifest))
endef

# CLASSPATHの設定
export CLASSPATH := $(call build-classpath, $(class_path))

# make-directories - 出力ディレクトリの存在を確実にする
make-directories := $(shell $(MKDIR) $(OUTPUT_DIR))

# help - デフォルトターゲット
.PHONY: help
help:
        @$(call brief-help, $(CURDIR)/Makefile)

# all - すべての構築作業を行う
.PHONY: all
all: compile jars javadoc
# all_javas - すべてのファイルのリストを保持する一時ファイル
all_javas := $(OUTPUT_DIR)/all.javas

# compile - ソースのコンパイル
.PHONY: compile
```

```
compile: $(all_javas)
        $(JAVAC) $(JFLAGS) @$<

# all_javas - ソースファイルのリストを収集する
.INTERMEDIATE: $(all_javas)
$(all_javas):
        $(FIND) $(SOURCE_DIR) -name '*.java' > $@

# jar_list - 作成すべき全jarのリスト
jar_list := server_jar ui_jar

# jars - すべてのjarを作成する
.PHONY: jars
jars: $(jar_list)

# server_jar - $(server_jar)を作成する
server_jar_name := $(OUTPUT_DIR)/lib/a.jar
server_jar_manifest := src/com/company/manifest/foo.mf
server_jar_packages := com/company/m com/company/n

# ui_jar - $(ui_jar)を作成する
ui_jar_name := $(OUTPUT_DIR)/lib/b.jar
ui_jar_manifest := src/com/company/manifest/bar.mf
ui_jar_packages := com/company/o com/company/p

# 各jarファイル向けの明示的ルールを作成する
# $(foreach j, $(jar_list), $(eval $(call make-jar,$j)))
$(eval $(call make-jar,server_jar))
$(eval $(call make-jar,ui_jar))

# javadoc - ソースからJava docを生成する
.PHONY: javadoc
javadoc: $(all_javas)
        $(JAVADOC) $(JDFLAGS) @$<

.PHONY: clean
clean:
        $(RM) $(OUTPUT_DIR)

.PHONY: classpath
classpath:
        @echo CLASSPATH='$(CLASSPATH)'

.PHONY: check-config
check-config:
        @echo Checking configuration...
        $(call check-file, $(class_path) JAVA_HOME)

.PHONY: print
print:
        $(foreach v, $(V), \
        $(warning $v = $($v)))
```

9.3 Javaのコンパイル

makeを使ったJavaのコンパイルは2つの方法で行うことができます。各ソースに対してそれぞれjavacを実行する伝統的な手法と、以前紹介した@filename文法を使用する高速な手法です。

9.3.1 高速な手法：一体型コンパイル

それでは高速な手法から始めましょう。汎用makefileで見たように以下のようになります。

```
# all_javas - すべてのファイルのリストを保持する一時ファイル
all_javas := $(OUTPUT_DIR)/all.javas

# compile - ソースのコンパイル
.PHONY: compile
compile: $(all_javas)
        $(JAVAC) $(JFLAGS) @$<

# all_javas - ソースファイルのリストを収集する
.INTERMEDIATE: $(all_javas)
$(all_javas):
        $(FIND) $(SOURCE_DIR) -name '*.java' > $@
```

擬似ターゲットcompileはプロジェクトの全ソースをコンパイルするためにjavacを1回だけ起動します。

$(all_javas)必須項目は、1行に1つずつすべてのJavaファイルが書き込まれたall.javasというファイルです。ファイルをそれぞれ別の行に書く必要はないのですが、こうすることにより必要に応じてgrep -vなどでふるいにかけることが簡単にできるようになります。makeを実行する度に消去されコンパイルのたびに新しいファイルが作られるように、all.javasは.INTERMEDIATEとしてマークされています。そのファイルを作るコマンドスクリプトは直感的です。保守性を最大限に高めるため、ソースツリーからJavaファイルをリストするのにfindコマンドを使用しています。このコマンドは動作が多少遅いのですが、ソースツリーが変更されてもコマンドを修正することなしに正しく動作します。

もしもmakefileの中からソースディレクトリのリストを容易に使えるのなら、all.javasを作るためのもっと高速なコマンドスクリプトがあります。そのソースディレクトリリストの長さが中くらいで、OSのコマンドライン長制限に抵触しない程度のものなら次のスクリプトが使えます†。

```
$(all_javas):
        shopt -s nullglob; \
        printf "%s\n" $(addsuffix /*.java,$(PACKAGE_DIRS)) > $@
```

このスクリプトは各ディレクトリ内にあるJavaファイルをリストするためにシェルのワイルドカードを使っています。ここでは、もしディレクトリ内にJavaファイルがなかった場合、ワイルドカードが

† 訳注：ここでは、PACKAGE_DIRS変数にディレクトリのリストが入っていると想定しています。

展開されずもともとのパターンが残る（多くのシェルにおける既定の動作）のではなく、空の文字列になることを期待します。そうするためにbashのオプションであるshopt -s nullglobを使用します。他の多くのシェルにも同様のオプションがあります。最後にls -1ではなくファイルの展開とprintfを併用しているのは、後者がbashの組み込みコマンドであり、パッケージディレクトリがいくつあっても起道されるコマンドを1つだけにするためです。

シェルの展開機能の代わりにwildcard関数を使うという別の方法もあります。

```
$(all_javas):
        print "%s\n" $(wildcard \
                       $(addsuffix /*.java,$(PACKAGE_DIRS))) > $@
```

もしソースディレクトリの数が非常に多い（またはパスが非常に長い）場合、上記のコマンドスクリプトはOSのコマンドライン長制限を超えてしまうかもしれません。その際には、次のコマンドスクリプトが適しています。

```
.INTERMEDIATE: $(all_javas)
$(all_javas):
        shopt -s nullglob;            \
        for f in $(PACKAGE_DIRS);     \
        do                            \
          printf "%s\n" $$f/*.java;   \
        done > $@
```

compileターゲットと、そのサポートルールは非再帰的手法に従っています。どれほど多くのサブディレクトリを使っていても1つのmakefileを使いコンパイラの実行は1回だけです。すべてのソースをコンパイルする必要があるのなら、この方法が考えられるかぎり最速です。

ここではすべての依存関係情報を無視しています。このルールの下ではどのファイルがどのファイルよりも新しいかをmakeは知りませんし考慮もしません。単にすべてのファイルを毎回コンパイルするだけです。さらにmakefileをバイナリツリーからでなくソースツリーから実行できるようになっています。makeの持つ依存関係を管理する能力を考慮するなら、このmakefileの使い方は間が抜けていると思うかもしれませんが、以下も考慮しなければなりません。

- 別の手段（この後すぐ検討しますが）では、標準的な依存関係を使います。これにより各ファイルに対してそれぞれjavacが起動され、多くのオーバーヘッドが加わってしまいます。しかしjavacコンパイラは非常に高速である一方でプロセスの生成には時間がかかります。したがって、もしプロジェクトが小さければすべてのファイルをコンパイルしてもファイルを少しだけコンパイルするのに比べてもそれほど多くの時間はかからないでしょう。どれほど多くの作業を行ったとしても、15秒以下の構築はならば基本的に等価といえます。およそ500のソースファイル（Antの配布ソース）では1.8GHz Pentium4に512MB RAMの筆者のPCで14秒かかりました。一方で1つのファイルをコンパイルするのに5秒かかります。
- たいていの開発者は個々のファイルを高速にコンパイルする手段を提供する、ある種の開発環境を使用するでしょう。makefileが使われるのは変更が広範囲におよび完全な再構築が必要

であるか、無人構築が必要な場合というのが適切です。

- これから見るように、依存関係を構成し維持するために必要な労力というのは、（8章で説明した）C/C++でソースツリーとバイナリツリーを分離するのと同じ程度です。過小評価できるような作業ではありません。

この後の例で見るようにPACKAGE_DIRS変数は単にall.javasファイルを作るため以外にも使われます。しかしこの変数を保守するのは多くの手間が必要になるうえに潜在的に困難な作業でもあります。小さなプロジェクトではディレクトリのリストを手作業でmakefileに書いておくことも可能ですが、その数が増えて100を超えるようになると、手作業では誤りが混入しやすい退屈な作業となってしまいます。ここに至ると、これらのディレクトリをfindを使って収集するのは賢明な方法だといえるでしょう。

```
# $(call find-compilation-dirs, root-directory)
find-compilation-dirs = \
  $(patsubst %/,%, \
    $(sort \
      $(dir \
        $(shell $(FIND) $1 -name '*.java'))))

PACKAGE_DIRS := $(call find-compilation-dirs, $(SOURCE_DIR))
```

findコマンドはファイルのリストを返し、dir関数はファイル名部分を削除しディレクトリ名だけを残します。sort関数はリストから重複を取り除き、patsubst関数は末尾のスラッシュを取り除きます。このfind-compilation-dirs関数はコンパイルすべきファイルを一覧にしファイル名部分だけを取り除きます。all.javasルールではワイルドカードを使ってファイル名部分を復元します。これは一見無駄なように見えますが、例えばEJBの設定ファイルを探し出す際にも使えるなど、ソースコードが置いてあるパッケージのリスト構築のための部品として使えるので有益なのです。パッケージの一覧が必要ないのであれば、all.javasを作る最初の単純な方法を使うべきです。

9.3.2 依存関係からコンパイルする

完全な依存関係チェックを経てコンパイルを行うには、Javaのソースに対して依存関係を調べるcc -Mと同様のツールがまず必要になります。Jikes（http://jikes.sourceforge.net/）は-makefileオプションまたは+Mオプションという同様の機能を提供しているオープンソースのJavaコンパイラです。Jikesは依存関係ファイルを常にソースと同じディレクトリに書いてしまうのでソースとバイナリの分離に適しているわけではありませんが、自由に利用できるだけでなくうまく動作します。さらによいことに、Jikesは依存関係ファイルをコンパイル中に生成するので、依存関係ファイルの生成のための作業を別に行う必要がありません。

以下は依存関係ファイル生成関数とそのルールです。

```
%.class: %.java
        $(JAVAC) $(JFLAGS) +M $<
        $(call java-process-depend,$<,$@)
```

```
# $(call java-process-depend, source-file, object-file)
define java-process-depend
  $(SED) -e 's/^.*\.class *:/$2 $(subst .class,.d,$2):/'  \
         $(subst .java,.u,$1) > $(subst .class,.tmp,$2)
  $(SED) -e 's/#.*//'                                     \
         -e 's/^[^:]*: *//'                               \
         -e 's/ *\\$$$$//'                                \
         -e '/^$$$$/ d'                                   \
         -e 's/$$$$/ :/' $(subst .class,.tmp,$2)          \
         >> $(subst .class,.tmp,$2)
  $(MV) $(subst .class,.tmp,$2).tmp $(subst .class,.d,$2)
endef
```

これを使うにはmakefileをバイナリツリーから実行しvpathがソースを見つけられるように設定されていなければなりません。Jikesコンパイラを依存関係生成のためだけに使い、別のコンパイラでコード生成を行うのであれば、+Bオプションを使ってJikesによるバイトコード生成を抑止することができます。

233のJavaファイルをコンパイルするテストでは、最初に紹介した1回だけコンパイラを使う手法が最も速く、筆者の環境では9.9秒でした。ファイルを個々にコンパイルする手法では411.6秒かかり41.5倍の違いがありました。さらに個別のコンパイルでは4つ以上のファイルを必要とするファイルでは1回のコンパイルですべてをコンパイルするよりも遅くなってしまいます。もし依存関係の生成とコンパイルを別のプログラムで行うなら、その差はもっと広がっているでしょう。

もちろん開発環境はそれぞれに異なっているのですが、目指すところを注意深く考慮することは重要です。コンパイルするファイルを最小にすることが常にシステムの構築時間を最小にするとはかぎりません。特にJavaでは完全な依存関係のチェックを行いコンパイルするファイルを最小化することは、通常の開発においては必要なこととはならないのです。

9.3.3 CLASSPATHの設定

Javaによるソフトウェア開発で最も重要な課題の1つがCLASSPATHを正しく設定することです。この変数はクラスの参照が行われたときにどこのコードをロードするかを決定するために使われます。Javaアプリケーションを正しくコンパイルするために、makefileは適切なCLASSPATHの設定を持たなければなりません。システムにJavaパッケージやAPIやサポートツールが追加されるにつれ、CLASSPATHはあっという間に長く複雑になってしまいます。もしCLASSPATHを適切に設定するのが難しいのなら、それを行う場所を1つにするのは理に適っています。

他のプログラムやmakeのためのCLASSPATH設定をmake自身にやらせるのは、筆者が便利だと思うテクニックの1つです。例えばclasspathターゲットはmakeを実行しているシェルにCLASSPATHを返します。

```
.PHONY: classpath
classpath:
        @echo "export CLASSPATH='$(CLASSPATH)'"
```

開発者は（もしbashを使っているなら）次のようにして各自のCLASSPATHを設定できます。

```
$ eval $(make classpath)
```

Windowsでは、次のようにしてCLASSPATHを設定できます。

```
.PHONY: windows_classpath
windows_classpath:
        regtool set /user/Environment/CLASSPATH "$(subst /,\\,$(CLASSPATH))"
        control sysdm.cpl,@1,3 &
        @echo "「環境変数」,「OK」,「OK」の順にクリックしてください。"
```

regtoolはWindowsのレジストリを操作するCygwinのユーティリティです。単にレジストリを設定しただけでは新しい値がWindowsに読み込まれません。そこで1つの方法として環境変数ダイアログを表示し、［OK］ボタンを押して終了します。

コマンドスクリプトの2行目で、「システムのプロパティ」ダイアログボックスの「詳細設定」タブを表示します。残念ながらこのコマンドでは「環境変数」ダイアログボックスを表示したり［OK］ボタンを押すことができないので、最後の行で利用者に対して処理を完了するための指示を表示します。

Emacs JDEEやJBuilderといった別のプログラムのプロジェクトファイルにCLASSPATHを渡すのは、それほど難しくありません。

CLASSPATHの設定もmakeにより管理できます。CLASSPATH変数を作るのが明らかに適切であり、次のように自明な方法で設定できます。

```
CLASSPATH = /third_party/toplink-2.5/TopLink.jar:/third_party/…
```

保守性を考えれば、変数を使うのが適しています。

```
CLASSPATH = $(TOPLINK_25_JAR):$(TOPLINKX_25_JAR):…
```

しかしもっとよい方法があります。汎用makefileにあるように、CLASSPATHを2段階に分けて設定します。最初にCLASSPATHの要素をmakeの変数として設定し、次にその変数を環境変数の文字列に変換します。

```
# Javaクラスパス設定
class_path := OUTPUT_DIR \
              XERCES_JAR \
              COMMONS_LOGGING_JAR \
              LOG4J_JAR \
              JUNIT_JAR
…
# CLASSPATH設定
export CLASSPATH := $(call build-classpath, $(class_path))
```

適切に書かれたbuild-classpath関数はいくつかのイライラさせられる問題を解決します（例9–1のCLASSPATHは利便性よりも実例を示す目的で書かれています）。

- CLASSPATHを各要素ごとに構成することが非常に簡単にできます。例えば、異なるアプリケーションサーバを使用している場合にCLASSPATHはそれに応じて変更する必要があります。異なるCLASSPATHをifdefによる選択の中に押し込め、makeの変数を使ってどちらかを選ぶことが可能となります。
- build-classpath関数が対応するので、保守担当者が空白を含む文字列や改行や行の継続に留意する必要がなくなります。
- パスの区切り文字はbuild-classpath関数が自動的に選択します。そのためUnixでもWindowsでも正しい値が使われることになります。
- パス要素の妥当性はbuild-classpath関数により検査されます。定義されていない変数がエラーにならず空の文字列になってしまうというmakeの不快な問題に対処しています。多くの場合でこの機能は有益なのですが、邪魔になることがまれに存在します。そのような場合、エラーも出さずに無効なCLASSPATHを作ってしまいます†。この問題に対処するために、build-classpath関数は空の値をチェックして警告を出します。また同時にそのファイルまたはディレクトリが実際に存在するかのチェックも行います。
- CLASSPATHを処理するフックを用意することで、パス名や検索パスに含まれる空白に対応するといった高度な機能を組み込むことができます。

次に示すのは、最初の3つの問題に対応しているbuild-classpathの一例です。

```
# $(call build-classpath, variable-list)
define build-classpath
  $(strip                                          \
    $(patsubst %:,%,                               \
      $(subst : ,:,                                \
        $(strip                                    \
          $(foreach c,$1,$(call get-file,$c):)))))
endef

# $(call get-file, variable-name)
define get-file
  $(strip                                          \
    $($1)                                          \
    $(if $(call file-exists-eval,$1),,             \
      $(warning The file referenced by variable    \
              '$1' ($($1)) cannot be found)))
endef

# $(call file-exists-eval, variable-name)
define file-exists-eval
$(strip                                            \
  $(if $($1),,$(warning '$1' has no value))        \
    $(wildcard $($1)))
endef
```

† --warn-undefined-variablesオプションを使うことで、このような状況を把握することができますが、意図して定義していない変数に対しても警告を出してしまいます。

build-classpath関数は引数内の単語それぞれに対して確認を行い、パス区切り文字（ここでは:）で連結します。パス区切り文字の自動選択はもはや簡単です。この関数では次にget-file関数とforeachループが加えた空白を除去します。続いてforeachループが加えたパス区切り文字のうち最後のものを除去します。そして行の継続により誤って混入した空白を取り除くために全体をstrip関数に渡します。

get-file関数は引数で指定された変数が実在のファイルを指しているか調べ、ファイル名を返します。実在していない場合には警告を表示します。この関数はファイルの有無にかかわらず変数の値を返します。というのもその値は呼び出し側にとっておそらく有益だからです。get-file関数は、これから生成されるまだ存在していないファイルに対しても使われることがあります。

最後のfile-exists-eval関数は、ファイル名が入っている変数の名前を受け付けます。もしその変数が空であったときには警告が表示されますが、そうでなければwildcard関数を使って値をファイル名（またはファイル名のリスト）に展開します。

build-classpath関数に無効な値を渡すと、次のようなエラーを生成します。

```
Makefile:37: The file referenced by variable 'TOPLINKX_25_JAR'
             (/usr/java/toplink-2.5/TopLinkX.jar) cannot be found
...
Makefile:37: 'XERCES_142_JAR' has no value
Makefile:37: The file referenced by variable
             'XERCES_142_JAR' ( ) cannot be found
```

警告も出ない単純な方法による設定と比べて、これは飛躍的な改善です。

get-file関数の存在は、CLASSPATHに加えるファイルの検索が一般化できることを示しています。

```
# $(call get-jar, variable-name)
define get-jar
  $(strip                                                    \
    $(if $($1),,$(warning '$1' is empty))                    \
    $(if $(JAR_PATH),,$(warning JAR_PATH is empty))          \
    $(foreach d, $(dir $($1)) $(JAR_PATH),                   \
      $(if $(wildcard $d/$(notdir $($1))),                   \
        $(if $(get-jar-return),,                             \
          $(eval get-jar-return := $d/$(notdir $($1))))))    \
    $(if $(get-jar-return),                                  \
      $(get-jar-return)                                      \
      $(eval get-jar-return :=),                             \
      $($1)                                                  \
      $(warning get-jar: File not found '$1' in $(JAR_PATH))))
endef
```

ここではファイルの検索パスを収めるためにJAR_PATHを定義しています。最初に見つかったファイルが返されます。この関数の引数はjarファイルへのパス名が入った変数の名前です。最初にこの変数で指定されたパスを探し、なければ次にJAR_PATHの中から探します。そのために、foreachループで使うディレクトリのリストは、変数の示すディレクトリに続いてJAR_PATHのディレクトリで

構成されます。jarファイルの名前とリスト中のパスを組み合わせられるように、引数をnotdir関数に入れて使います。そして、このforeachループからは途中で抜け出さないことに注意しましょう。

その代わりにeval関数を使い最初に見つけたファイル名をget-jar-return変数に設定しています。ループが終了した後で、変数に設定した値を返すか、何も見つからなかった場合には警告を表示します。マクロを終了する前に、返す値を入れておく変数を初期化しておくのを忘れてはいけません。

これはCLASSPATHを設定するという作業に対して、vpath機能と本質的に同じものを作ったに過ぎません。これを理解するために、vpathとは必須項目をカレントディレクトリからの相対パスでは見つけられないときにmakeが自動的に使用するパスであることを思い出しましょう。その場合makeはvpathの中から必須項目を探し出し、自動変数である$^、$?、$+に完全なパス名を供給します。CLASSPATHを設定するには、makeに各jarファイルのパスを検索させて、完全なパス名をCLASSPATH変数に入れればよいのですが、makeはそのための機能を組み込んでいないので、手作業で行わなければなりません。単純にJAR_PATH変数に適切なjarのファイル名を添えて展開し、その中からJava自身に選択させてもよいのですが、CLASSPATHはすぐに長くなってしまいます。OSによっては環境変数の長さに制限があり、長いCLASSPATHが切り詰められてしまう恐れがあります。Windows XPでは1つの環境変数の最大長は1,023文字です。加えて、たとえCLASSPATHが切り詰められなかったとしてもJava仮想マシンは必要なクラスをCLASSPATHから探し出すためアプリケーションの動作が遅くなってしまいます。

9.4 jarの管理

jarファイルの構築や管理ではC/C++のライブラリとは異なる問題が浮かび上がります。これには3つの理由が存在します。まずjarファイルに入るものは相対パスを持つので、jarプログラムに渡す正確なファイル名は注意深く管理されなければなりません。次にJavaでは1つのjarファイルが1つのプログラムを表すように構成する傾向があります。最後に、jarファイルの中にはマニフェストファイル、プロパティファイル、XMLなど、クラスファイル以外のものも入ります。

以下はGNU makeでjarファイルを作成する基本的なコマンドです。

```
JAR := jar
JARFLAGS := -cf

$(FOO_JAR): 必須項目…
        $(JAR) $(JARFLAGS) $@ $^
```

jarプログラムはファイル名の代わりにディレクトリ名も受け付けます。その場合、ディレクトリツリー内のすべてのファイルがjarファイルに入ります。これは-Cオプションを使ってディレクトリを変更する場合には特に便利な機能です†。

† 訳注：jarコマンドの-Cオプションは、続く引数を処理する際に一時的にディレクトリを変更するものです。

```
JAR := jar
JARFLAGS := -cf

.PHONY: $(FOO_JAR)
$(FOO_JAR):
        $(JAR) $(JARFLAGS) $@ -C $(OUTPUT_DIR) com
```

ここではjarファイルを.PHONYターゲットにしてあります。そうしないと、jarファイル自体は必須項目を持たないので、次回のmake実行ではファイルが再作成されません。以前arのところで説明したように、少なくともたいていの更新作業においてjarを最初から作るのと同じか長い時間がかかってしまうため、更新オプションである-uを使う必要性はあまりありません†。

jarファイルにはたいてい、供給者やAPIやバージョン番号を記述したマニフェストファイルが含まれます。簡単なマニフェストファイルの中身は次のようなものです。

```
Name: JAR_NAME
Specification-Title: SPEC_NAME
Implementation-Version: IMPL_VERSION
Specification-Vendor: Generic Innovative Company, Inc.
```

ここにはmakeの実行時にsedやm4またはその他のストリームエディタにより置き換えられる3つのプレースホルダ、JAR_NAME、SPEC_NAME、IMPL_VERSIONが配置されています。次はマニフェストファイルを処理する関数です。

```
MANIFEST_TEMPLATE := src/manifests/default.mf
TMP_JAR_DIR := $(call make-temp-dir)
TMP_MANIFEST := $(TMP_JAR_DIR)/manifest.mf

# $(call add-manifest, jar, jar-name, manifest-file-opt)
define add-manifest
  $(RM) $(dir $(TMP_MANIFEST))
  $(MKDIR) $(dir $(TMP_MANIFEST))
  m4 --define=NAME="$(notdir $2)"                \
     --define=IMPL_VERSION=$(VERSION_NUMBER)     \
     --define=SPEC_VERSION=$(VERSION_NUMBER)     \
     $(if $3,$3,$(MANIFEST_TEMPLATE))            \
     > $(TMP_MANIFEST)
  $(JAR) -ufm $1 $(TMP_MANIFEST)
  $(RM) $(dir $(TMP_MANIFEST))
endef
```

add-manifest関数は、上の例で示したようなマニフェストファイルを処理します。この関数は、まず一時ディレクトリを作成しサンプルのマニフェストを展開します。次にjarファイルを更新し、

† 訳注：ここではjarファイルに対して必須項目を指定せずに、あるディレクトリ下のファイルをすべてパッケージするという方法を採用しています。ターゲットのjarファイルを.PHONYにしておかないと、すでにそのjarファイルが存在していた場合、比較対象となる必須項目がないので、いつでも最新のファイルであると判断されてしまいます。

最後に一時ディレクトリを削除します。関数の最後のパラメータは任意パラメータで、マニフェストファイルのパス名が指定されなかった場合、MANIFEST_TEMPLETE変数の値を使用します。

汎用makefileでは、jarファイルを作成する明示的ルールを作る汎用関数の中にこの作業を組み込んであります。

```
# $(call make-jar,jar-variable-prefix)
define make-jar
  .PHONY: $1 $$($1_name)
  $1: $($1_name)
  $$($1_name):
        cd $(OUTPUT_DIR); \
        $(JAR) $(JARFLAGS) $$(notdir $$@) $$($1_packages)
        $$(call add-manifest, $$@, $$($1_name), $$($1_manifest))
endef
```

jarに対する4つのパラメータ、つまりターゲット名、jarファイル名、jar内のパッケージ名、jarのマニフェストファイル名を表すmakeの変数に付けるプリフィックスを、この関数は唯一のパラメータとして受け付けます。例えばui.jarに対しては、次のように使います。

```
ui_jar_name := $(OUTPUT_DIR)/lib/ui.jar
ui_jar_manifest := src/com/company/ui/manifest.mf
ui_jar_packages := src/com/company/ui \
                   src/com/company/lib

$(eval $(call make-jar,ui_jar))
```

変数名の合成を使うことにより、関数の呼び出しを短くすることができるだけでなく、関数の実装を柔軟にすることが可能となります。

多くのjarファイルを作らなければならない場合、jarの名前を変数に入れておくことで、この作業を自動化できます。

```
jar_list := server_jar ui_jar

.PHONY: jars $(jar_list)
jars: $(jar_list)

$(foreach j, $(jar_list),\
  $(eval $(call make-jar,$j)))
```

状況によってはjarファイルの中身を一時ディレクトリに展開する必要があります。その場合には次の関数を使うことができます。

```
# $(call burst-jar, jar-file, target-directory)
define burst-jar
  $(call make-dir,$2)
  cd $2; $(JAR) -xf $1
endef
```

9.5 リファレンスツリーとサードバーティ製jarファイル

共有の単一リファレンスツリーを作り、開発者が持つのをソースツリーの一部だけにするには、プロジェクトのjarを毎晩構築しJavaコンパイラのCLASSPATHにそのjarを置くことで可能になります。開発者は必要とするソースツリーの一部分だけをチェックアウトし、コンパイル（ソースファイルのリストはfindなどを使って実行時に作成することを想定しています）を実行します。開発者の持っていないソースのシンボルをJavaコンパイラが必要とすると、CLASSPATHを探しjarファイル中の.classファイルを見つけることになります。

リファレンスツリーからサードパーティ製のjarファイルを選択するのも簡単です。そのjarファイルへのパス名をCLASSPATHに含めるだけです。以前にも書いたようにmakeはこういった作業を管理するのに有益なツールとなります。JAR_PATH変数を設定することにより、ベータ版jarファイルか安定版jarファイル、ローカルのjarファイルかリモートにあるjarファイルといった切り替えをget-file関数を使って自動的に行うことができます。

10章
makeの性能改善

makeは開発プロセスの中で重要な役割を担っています。makeはプロジェクトの各要素を組み合わせてアプリケーションを構築するだけでなく、構築作業のどこかを誤って抜かしてしまったために生じる微妙なエラーをも排除します。しかしながら、もし開発者がmakefileの実行が遅いことを理由にmakeの使用をやめてしまったなら、makeの利点はすべて失われてしまいます。それ故、makefileをできるだけ効率よく巧みに記述することは重要なことなのです。

性能の問題はいつも扱いにくいものですが、利用者が性能をどのように認識するかを考慮したり、コードの中で複数の経路が考えられる場合はなおさらです。makefile中、すべてのターゲットを最適化する必要はありません。また非常によい改善であっても、状況により価値のないものになってしまいます。例えば処理の時間が90分から45分になったとしても、「昼食の時間」にかわりはないかもしれません。一方で開発者が特にやるべきこともなく過ごす時間が2分から1分になったとすれば、それは大歓迎されるでしょう。

効率よく動作するmakefileを書く際には、実行される操作と各操作のコストを把握することが非常に重要です。本章では、ここで述べたことを定量化するために簡単な比較を行うとともに、ボトルネックを特定するためのテクニックを紹介します。

性能を改善するための補足的な手法として、並列実行とローカルネットワーク配置の利点を活用することがあげられます。1つ以上のコマンドスクリプトを実行することにより（単一プロセッサ上であっても）、構築時間を短くすることができます。

10.1　ベンチマーク

makeの基本的な操作を比較してみました。表10-1はその結果を示しています。各テストの内容と普通のmakefileとどのように関連するかについては後で説明します。

表10-1 各操作のコスト

操作	実行回数	1回あたりの実行時間（Windows）	1秒あたりの実行回数（Windows）	1回あたりの実行時間（Linux）	1秒あたりの実行回数（Linux）
make（bash）	1000	0.0436	22	0.0162	61
make（ash）	1000	0.0413	24	0.0151	66
make（sh）	1000	0.0452	22	0.0159	62
assignment	10,000	0.0001	8130	0.0001	10,989
subst（short）	10,000	0.0003	3891	0.0003	3846
subst（long）	10,000	0.0018	547	0.0014	704
sed（bash）	1000	0.0910	10	0.0342	29
sed（ash）	1000	0.0699	14	0.0069	144
sed（sh）	1000	0.0911	10	0.0139	71
shell（bash）	1000	0.0398	25	0.0261	38
shell（ash）	1000	0.0253	39	0.0018	555
shell（sh）	1000	0.0399	25	0.0050	198

Windowsのテスト環境は1.9GHz Pentium4 (おおよそ3,578BogoMips †)、512MB RAM、WindowsXPで、Cygwin版のmake 3.80をrxvtウィンドウから実行しています。Linuxのテストでは450MHz Pentium2 (891 BogoMips)、256MB RAM、RedHat 9を使用しました。

makeから起動されるサブシェルは全体の性能に大きな影響を持っています。bashは複雑で多機能であるが故に巨大です。ashはずっと小さく機能も絞られていますが、たいていの作業を行うには十分です。厄介なことにbashは/bin/shとして実行されると、標準シェルに可能なかぎり適合するよう動作をまったく変えてしまいます。多くのLinuxシステムでは/bin/shはbashへのシンボリックリンクですが、Cygwinでは/bin/shが実際にはashです。これらの違いも計測するために、いくつかのテストでは異なるシェルを用いて3回実行しました。使用したシェルは括弧で囲んであります。“(sh)” とあるのは、/bin/shにリンクされているbashを意味します。

最初のmakeという名前の3回のテストでは、何もしないmakeを実行するのにどれくらいの処理時間がかかるかを示しています。makefileは次のようなものです。

```
SHELL := /bin/bash
.PHONY: x
x:
        $(MAKE) --no-print-directory --silent --question make-bash.mk; \
        …このコマンドを99回繰り返す…
```

bashと書かれた部分は必要に応じて別のシェルに書き換えます。

テスト結果をゆがめてしまう不必要な処理を排除するとともに、処理時間の表示を埋もれさせてし

† BogoMipsについては、http://www.clifton.nl/bogomips.htmlを参照してください。
訳注：多少古いですがJF Projectによる日本語訳がhttp://www.linux.or.jp/JF/JFdocs/BogoMips/から参照できます。

まう大量の出力を抑えるために--no-print-directoryオプションと--silentオプションを使いました。--questionオプションによりmakeはコマンドの実行を行わず依存関係だけをチェックし、ファイルが最新であった場合には0を返します。これでmakeは必要最小限の作業だけを行うことになります。このmakefileではコマンドが実行されず、依存関係は唯一.PHONYターゲットだけです。このコマンドスクリプトはmakeを100回呼び出します。このmake-bash.mkと呼ばれるmakefileは次のような親makefileから10回実行されます。

```
define ten-times
  TESTS += $1
  .PHONY: $1
  $1:
          @echo $(MAKE) --no-print-directory --silent $2; \
          time $(MAKE) --no-print-directory --silent $2;  \
          time $(MAKE) --no-print-directory --silent $2;  \
          time $(MAKE) --no-print-directory --silent $2;  \
          time $(MAKE) --no-print-directory --silent $2;  \
          time $(MAKE) --no-print-directory --silent $2;  \
          time $(MAKE) --no-print-directory --silent $2;  \
          time $(MAKE) --no-print-directory --silent $2;  \
          time $(MAKE) --no-print-directory --silent $2;  \
          time $(MAKE) --no-print-directory --silent $2;  \
          time $(MAKE) --no-print-directory --silent $2
endef

.PHONY: all
all:

$(eval $(call ten-times, make-bash, -f make-bash.mk))

all: $(TESTS)
```

そしてこの1,000回の実行時間の平均を取ります。

表からわかるように、Cygwinのmakeは1回の実行に0.044秒または1秒におおよそ22回実行できるのに対し、Linuxでは（大幅に遅いCPUであるにもかかわらず）1秒におよそ61回、1回あたり0.016秒かかることになっています。この結果を検証するためにWindows版のmakeを使って同様のテストを行い、大きな改善がなかったことを確認しました。つまり結果として、Cygwinのmakeは Windows版のmakeに比べて、プロセスの生成が多少遅いのですが、どちらにしてもLinuxと比べて劇的に遅いということになります。このことはWindows上で再帰的makeを使った構築作業はLinux上で行う同様の構築と比べて非常に遅いということを示しています。

このテストでは使用したシェルが結果に影響していないと指摘する方がいるかもしれません。ここでのコマンドスクリプトではシェルの特殊文字を使用していないので、シェルは一切起動されず、makeはコマンドを直接実行します。このことはSHELL変数にでたらめな値を入れてもテストが適切に実行されることで検証できます。3つのシェルで出た性能の違いは通常のシステム上におけるばらつきに属するものです。

次のベンチマークでは変数代入の速度を計ります。これは多くの基本的なmake処理の指標となり

ます。使用するmakefileであるassign.mkは次のような内容です。

```
# 10,000回代入
z := 10
…10,000回繰り返す…
.PHONY: x
x: ;
```

このmakefileはten-times関数を通して親makefileから実行されます。

代入は明らかに高速です。Cygwinのmakeでは1秒に8,130回、Linuxでは10,989回実行されます。makeプロセスを10回起動するために要する時間を正確に除外できないので、Windowsにおけるこうした操作の性能はベンチマークが示すものより実際はもっとよいと考えられます。結論として、平均的なmakefileでは代入を10,000回も行わないので、一般的なmakefileでは代入にかかるコストは無視することができます。

続く2つのベンチマークではsubst関数呼び出しのコストを測定します。最初は短い10文字の文字列で3か所の置換を行います。

```
# 10文字に対するsubst関数呼び出しを10,000回
dir := ab/cd/ef/g
x := $(subst /, ,$(dir))
…10,000回繰り返す…
.PHONY: x
x: ;
```

この処理は単純な代入と比べておおよそ2倍の時間で、もしくは秒あたり3,891回の処理がWindows上で実行できます。繰り返しになりますが、LinuxではWindowsと比較しても優れた性能を示しています。（LinuxはWindowsと比較して4分の1以下のクロック数で動作していることを思い出してください。）

もう一方では1,000文字の文字列に対しておおよそ100か所の置換が行われます。

```
# 10文字のファイル名
dir := ab/cd/ef/g
# 1000文字のパス名
p100 := $(dir);$(dir);$(dir);$(dir);$(dir);…
p1000 := $(p100)$(p100)$(p100)$(p100)$(p100)…

# 1,000文字に対するsubst関数呼び出しを10,000回
x := $(subst ;, ,$(p1000))
…repeated 10000 times…
.PHONY: x
x: ;
```

次の3つのベンチマークでは同様の置換をsedを用いて行います。ベンチマークでは次を使います。

```
# bashを経由してsedを100回実行
SHELL := /bin/bash
.PHONY: sed-bash
```

```
sed-bash:
        echo '$(p1000)' | sed 's/;/ /g' > /dev/null
        …100回繰り返し…
```

これまでのように、このmakefileはten-times関数を通して実行されます。Windowsではsubstと比較してsedを使うと50倍遅くなります。Linuxでは24倍程度です。

シェルの実行分を要素に含めると、Windowsではashを使うことにより手軽にスピードアップが望めます。ashとともに使うとsedはsubstに比べて、なんとたった39倍しか遅くなりません。Linuxではシェルがもっと強い影響を持ちます。ashを使うとsedによる操作はsubstの5倍程度遅くなるだけです。bashに関する興味深い点にも言及しておきましょう。Cygwinでは/bin/bashと/bin/shの間に違いはありませんが、Linuxでは/bin/shにリンクされているbashのほうがずいぶんよい性能を示します。

最後のベンチマークはサブシェルの実行コストを測定するために単純にshell関数を実行します。次のmakefileを使用します。

```
# bashを使って100回 $(shell ) を実行する
SHELL := /bin/bash
x := $(shell   :)
…100回繰り返し…
.PHONY: x
x: ;
```

この結果は驚くに値しません。bashよりも性能の優れているashを使っても、Windowsでは Linuxよりも遅いのです。ashによる性能の改善は約50%の高速化となり、顕著です。Linuxでは ashが最善で、bash（bashという名前を使う）が最悪です。

ベンチマークはきりがありませんが、ここで行った測定は有益な洞察を与えます。makefileの構造を明快にするなら変数をどれだけ多く作ってもかまいません、なぜなら基本的にそれにはコストがかからないからです。たとえコードの構造的にmakeの関数を繰り返し使わなければならないような場合でもコマンドを実行するよりも組み込みのmake関数を使うほうが望ましいのです。Windowsでは不必要なプロセスを生成したり再帰的makeの使用は避けましょう。一方Linuxでは多くのプロセスを生成するならashを使いましょう。

多くのmakefileでは実行にかかる時間のほとんどはプログラムを実行するためのもので、make自身やmakefileの構造に起因するものではありません。たいていの場合、実行するプログラムの数を減らすことがmakefileの実行時間の減少に最も多く寄与します。

10.2 ボトルネックの特定と対応

10.2.1 単純変数対再帰変数

処理性能に関連するよくある問題の1つが、単純変数の代わりに使われる再帰変数です。例えば下記のコードでは、:=ではなく、=演算子が使われているため、DATE変数が使われるたびにdateコマンドが実行されてしまいます。

```
DATE = $(shell date +%F)
```

+%Fオプションは日付を“yyyy-mm-dd”形式で返すように指示するものなので、利用者はdateコマンドが繰り返し実行されていることに気づきません。もちろん日付が変わるような深夜に働いているとびっくりしてしまうかもしれません。

makeはshell関数から起動されるコマンドを表示しないので、実際に何が実行されているかを把握するのは難しいのです。SHELL変数を/bin/sh -xに再設定することで、makeが実行するすべてのコマンドを明示できます。

次のmakefileは何かを実行する前に、その出力ディレクトリを作成します。出力ディレクトリの名前は日付に“out”を付加したものになります。

```
DATE = $(shell date +%F)
OUTPUT_DIR = out-$(DATE)

make-directories := $(shell [ -d $(OUTPUT_DIR) ] || mkdir -p $(OUTPUT_DIR))

all: ;
```

デバッグ用シェルとともに実行すると次のようになります。

```
$ make SHELL='/bin/sh -x'
+ date +%F
+ date +%F
+ '[' -d out-2004-03-30 ']'
+ mkdir -p out-2004-03-30
make: all is up to date.
```

dateコマンドが2回実行されていることが明白です。このようなシェルのトレースが頻繁に必要になるなら、次のようにすれば簡単です。

```
ifdef DEBUG_SHELL
  SHELL = /bin/sh -x
endif
```

10.2.2 @を無効にする

コマンド修飾子@を使うのが、コマンドを隠してしまう別の方法です。場合によりこの機能を無効化できれば便利です。@を保持するQUIET変数を用意し、コマンドでこの変数を使うのが簡単な方法です。

```
ifndef VERBOSE
  QUIET := @
endif
…
target:
        $(QUIET) echo Building target...
```

コマンド修飾子で隠されたコマンドを見る必要が生じた際には、コマンドライン上で単にVERBOSEを定義するだけです。

```
$ make VERBOSE=1
echo Building target...
Building target...
```

10.2.3 初期化の遅延

単純変数の定義にshell関数が使われていた場合、makeはmakefileを読み込むときにそれらすべてのshell関数を評価します。その数が多い、あるいは実行する処理が重いとmakeの動きが遅く見えてしまいます。makeの応答性は存在しないターゲットに対して実行することで計測することができます。

```
$ time make no-such-target
make: *** No rule to make target no-such-target. Stop. †

real 0m0.058s
user 0m0.062s
sys 0m0.015s
```

このコードはコマンドの実行に際してmakeが付加するオーバーヘッドを計測します。それがささいなコマンドやエラーを含むコマンドに対しても同様です。

再帰変数は代入の右辺をそのつど再評価するので、単純変数に比べて複雑な計算で使う傾向があります。しかしながらこれではすべてのターゲットに対してmakeの応答性を低下させてしまいます。変数が使われる前ではなく使われたときに、1回だけ代入の右側が評価されるような新しい種類の変数が必要に感じるでしょう。

こういった初期化の必要性を説明する例が、「9.3.1 高速な手法：一体型コンパイル」で紹介したfind-compilation-dirs関数です。

```
# $(call find-compilation-dirs, root-directory)
find-compilation-dirs = \
  $(patsubst %/,%, \
    $(sort \
      $(dir \
        $(shell $(FIND) $1 -name '*.java'))))

PACKAGE_DIRS := $(call find-compilation-dirs, $(SOURCE_DIR))
```

この中のfindはmakeの実行に対して1回だけ、ただしPACKAGE_DIRS変数が実際に使われたときに行うのが理想的です。これを**初期化の遅延**と呼びます。このような変数はevalを利用して次のように作ることができます。

† 訳注：「no-such-targetを構築するためのルールが見当たりません」の意。

```
PACKAGE_DIRS = $(redefine-package-dirs) $(PACKAGE_DIRS)

redefine-package-dirs = \
  $(eval PACKAGE_DIRS := $(call find-compilation-dirs, $(SOURCE_DIR)))
```

基本的には、まず最初にPACKAGE_DIRSを再帰変数として定義します。これが展開されるときに重い処理（ここではfind-compilation-dirsです）が評価され単純変数として再定義されます。最後に単純変数の値が本来の再帰変数定義の値として返されます。

もう少し詳しく見てみましょう。

1. この変数は再帰変数なので、makeがこの変数を読み込んだときには単にその右辺の記録だけを行います。
2. PACKAGE_DIRSが最初に使われたときにmakeは右辺の内容を取り出し、最初の変数であるredefine-package-dirsを展開します。
3. redefine-package-dirs変数の値は、eval関数の呼び出しです。
4. evalの中身は、PACKAGE_DIRを単純変数として定義し、find-compilation-dirs関数の返すディレクトリのリストを値として設定するというものです。これでPACKAGE_DIRはディレクトリのリストで初期化されました。
5. redefine-package-dirs変数は何らかの文字列に展開されるわけではありません（なぜならeval関数は空の文字列に展開されるからです）。
6. そしてmakeはもともとのPACKAGE_DIRの右辺を引き続き展開します。残っているのは変数PACKAGE_DIRSの展開です。makeはこの変数の値を探し出し単純変数として定義されている値を取り出します。

このコードの巧妙な点は、makeが再帰変数の右辺を評価する順番が左から右だということを利用している部分です。例えばもしmakeが$(PACKAGE_DIRS)の評価を$(redefine-package-dirs)よりも先に行った場合、このコードはうまく働きません。

この手法はlazy-init関数としてまとめることができます。

```
# $(call lazy-init,variable-name,value)
define lazy-init
  $1 = $$(redefine-$1) $$($1)
  redefine-$1 = $$(eval $1 := $2)
endef

# PACKAGE_DIRS - 初期化遅延によるディレクトリのリスト
$(eval \
  $(call lazy-init,PACKAGE_DIRS, \
    $$(call find-compilation-dirs,$(SOURCE_DIRS))))
```

10.3 並列make

構築性能を向上させる別の手法では、makeの解決する問題が生来もつ並列性をうまく利用します。ほとんどのmakefileではCソースをコンパイルしてオブジェクトファイルを作るとか、オブジェクトファイルからライブラリを作成するなど、同時並行的に実行できる作業を多く行います。さらに、非常によく記述されたmakefileは、同時処理を自動的に管理するために必要なすべての情報を提供します。

例10-1ではmp3_playerプログラムの構築を--jobs=2（または-j 2）オプション付きで実行しています。図10-1は、そのmakeの実行過程を擬似UMLシーケンス図で示しています。--jobs=2を指定することにより、可能なら2つのターゲットを同時に更新するようにmakeに指示することができます。makeが複数のターゲットを同時に更新するときには、実行順にコマンドを飛び飛びに表示します。これにより並列makeの出力が読みにくくなってしまいます。出力結果を注意深く見ていきましょう。

例10-1 --jobs=2でmakeを実行

```
$ make -f ../ch07-separate-binaries/makefile --jobs=2

1   bison -y  --defines ../ch07-separate-binaries/lib/db/playlist.y

2   flex  -t ../ch07-separate-binaries/lib/db/scanner.l > lib/db/scanner.c

3   gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -M
    ../ch07-separate-binaries/app/player/play_mp3.c | \
    sed 's,\(play_mp3\.o\) *:,app/player/\1 app/player/play_mp3.d: ,' > app/player/play_mp3.d.tmp

4   mv -f y.tab.c lib/db/playlist.c

5   mv -f y.tab.h lib/db/playlist.h

6   gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -M
    ../ch07-separate-binaries/lib/codec/codec.c | \
    sed 's,\(codec\.o\) *:,lib/codec/\1 lib/codec/codec.d: ,' > lib/codec/codec.d.tmp

7   mv -f app/player/play_mp3.d.tmp app/player/play_mp3.d

8   gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -M
    lib/db/playlist.c | \
    sed 's,\(playlist\.o\) *:,lib/db/\1 lib/db/playlist.d: ,' > lib/db/playlist.d.tmp

9   mv -f lib/codec/codec.d.tmp lib/codec/codec.d

10  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -M
    ../ch07-separate-binaries/lib/ui/ui.c | \
    sed 's,\(ui\.o\) *:,lib/ui/\1 lib/ui/ui.d: ,' > lib/ui/ui.d.tmp
```

```
11  mv -f lib/db/playlist.d.tmp lib/db/playlist.d

12  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -M
    lib/db/scanner.c | \
    sed 's,\(scanner\.o\) *:,lib/db/\1 lib/db/scanner.d: ,' > lib/db/scanner.d.tmp

13  mv -f lib/ui/ui.d.tmp lib/ui/ui.d

14  mv -f lib/db/scanner.d.tmp lib/db/scanner.d

15  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include
    -c -o app/player/play_mp3.o ../ch07-separate-binaries/app/player/play_mp3.c

16  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include
    -c -o lib/codec/codec.o ../ch07-separate-binaries/lib/codec/codec.c

17  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include
    -c -o lib/db/playlist.o lib/db/playlist.c

18  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include
    -c -o lib/db/scanner.o lib/db/scanner.c
    ../ch07-separate-binaries/lib/db/scanner.l: In function yylex:
    ../ch07-separate-binaries/lib/db/scanner.l:9: warning: return makes integer from
    pointer without a cast

19  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include
    -c -o lib/ui/ui.o ../ch07-separate-binaries/lib/ui/ui.c

20  ar rv lib/codec/libcodec.a lib/codec/codec.o
    ar: creating lib/codec/libcodec.a
    a - lib/codec/codec.o

21  ar rv lib/db/libdb.a lib/db/playlist.o lib/db/scanner.o
    ar: creating lib/db/libdb.a
    a - lib/db/playlist.o
    a - lib/db/scanner.o

22  ar rv lib/ui/libui.a lib/ui/ui.o
    ar: creating lib/ui/libui.a
    a - lib/ui/ui.o

23  gcc   app/player/play_mp3.o lib/codec/libcodec.a lib/db/libdb.a lib/ui/libui.a
    -o app/player/play_mp3
```

まずmakeはソースの生成と依存関係ファイルの作成を行う必要があります。2つの生成ソースはyaccとlexの出力です。コマンドの1と2がこれに相当します。3番目のコマンドはpaly_mp3.cの依存関係ファイルを作成するものですが、明らかにplaylist.cとscanner.cの依存関係ファイルの生成が完了する（コマンドの4、5、8、9、12、14が相当します）よりも前に実行されています。その結果コマンドラインオプションでは2つを指定していたにもかかわらず、このmakeは3つの作業を平行に行っています。

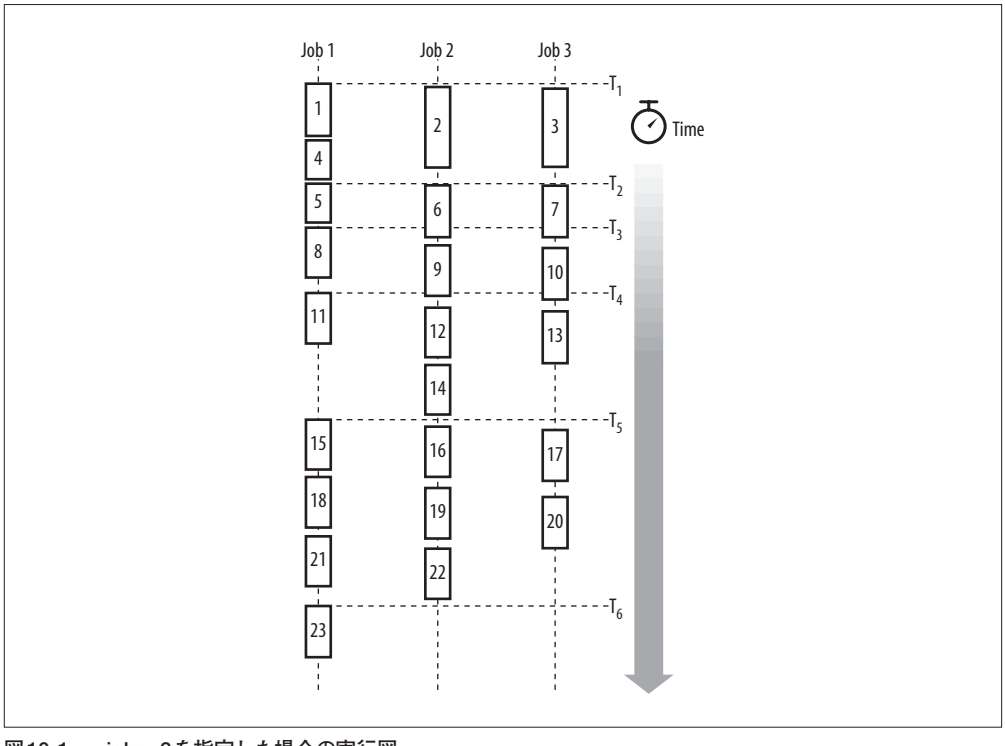

図10-1　--jobs=2を指定した場合の実行図

4と5のmvコマンドで、1のコマンドで開始されたplaylist.cのソース生成は完了します。6のコマンドは別の依存関係ファイル作成を開始します。どのコマンドスクリプトも1つのmakeが実行していますが、個々のターゲットと必須項目は別々の作業となります。そのため依存関係ファイルを作成する2番目のコマンドである7のコマンドは、おそらく3のコマンドと同じプロセスで実行されています。6のコマンドはおそらく1–4–5のコマンド（yacc文法の処理）の完了直後、しかし8のコマンドで依存関係ファイルの生成が行われる前に実行されています。

依存関係ファイルの作成は、このようにして14のコマンドまで続きます。すべての依存関係ファイルの作成は、makeが次の処理段階への移行、つまりmakefileの再読み込みを行う前に完了しておかなければなりません。ここでmakeは自動的に同期します。

makefileが依存関係情報の再読み込みを終えると、makeは構築処理の並列実行を再開します。ここでmakeはそれぞれのライブラリを作成する前に全オブジェクトファイルの作成を選択します。この順番はいつも同じではありません。つまりmakeを再度実行したときには、libcodec.aの作成がplaylist.cのコンパイルよりも前に行われるかもしれません。なぜならlibcodec.aはcodec.o以外のオブジェクトを必要としないからです。このように、例が示しているのは多くの可能性の中の1つなのです。

最後にプログラムがリンクされます。このmakefileではリンクでも同期が取られ常に最後に実行

されます。しかしもし目的とするものが1つのプログラムではなく複数のプログラムやライブラリであった場合には、最後のコマンドはそのときどきで変わることでしょう。

マルチプロセッサの上で複数の作業を同時に行うのは明らかに適切ですが、複数の作業を単一プロセッサの上で行うのもまた有益です。その理由は、たいていのシステムで発生するディスクI/Oや巨大なキャッシュの待ち時間です。例えばgccのようなプロセスがディスクのI/Oで待ちに入っている間でも、mvやyaccやarなどの別のプロセスのデータはメモリに読み込まれているかもしれません。このような場合、データが利用可能となっている作業を先に進めるのは理にかなっています。一般的に単一プロセッサ上で2つの作業を行うのは1つの場合に比べてほとんど常に高速で、3つまたは4つの作業が2つよりも速いことは、そう珍しくもありません。

--jobsオプションは数字の指定なしで使うこともできます。その場合、ターゲットを更新するためにできるだけ多くの作業を同時に行います。多数の同時作業はプロセッサの処理を圧倒してしまうことが多く、単一の作業だけを行っているときよりも遅くしてしまうため、これは通常よくない方法だといえます。

複数の作業を管理するもう1つの方法は、システムの**ロードアベレージ**（load average）を指定することです。ロードアベレージは一定の期間、通常は1分、5分、15分内の実行待ちプロセス数の平均です。ロードアベレージは浮動小数点数で表現されます。--load-averageオプション（-lオプション）で新しい作業を開始しない上限値を指定します。例えば次のコマンドは、ロードアベレージが3.5未満であった場合にかぎり新しい作業を開始するようmakeに指示します。

```
$ make --load-average=3.5
```

もし指定よりも大きかった場合には、makeはロードアベレージが指定値よりも下がるか他の作業がすべて終了するまで待ちます。

並列実行を行うためのmakefileを書く際には、適切に必須項目を指定することの重要性がさらに増します。前述したように--jobsが1の場合には必須項目のリストは左から右へと評価されます。--jobsが1よりも大きい場合には、必須項目が同時に評価されます。左から右という既定の方法で扱っている依存関係は、並列実行を行う際には明示的に指定しなければなりません。

並列makeの別の落とし穴として、共用中間ファイルの問題があります。例えばあるディレクトリにfoo.yとbar.yが存在している場合に、2つのyaccを同時に実行すると、一方の処理で作られたy.tab.cとy.tab.hのどちらか、もしくは両方がもう一方の結果として使われてしまうかもしれません。固定のファイル名を使う一時ファイルに情報をいったん格納するような処理では、同様の問題が発生します。

並列実行を妨げてしまう、よくある問題としてシェルのforループから起動される再帰的makeがあげられます。

```
dir:
        for d in $(SUBDIRS); \
        do \
          $(MAKE) --directory=$$d; \
        done
```

「6.1 再帰的make」で述べたように、makeはこの再帰的makeを並列に実行することができません。並列に実行するためには、ディレクトリを.PHONYとして指定したターゲットにしなければなりません。

```
.PHONY: $(SUBDIRS)
$(SUBDIRS):
        $(MAKE) --directory=$@
```

10.4 分散make

GNU makeはネットワークを介して複数のシステムを使用して構築を行うための、あまり知られていない（そして少ししかテストされていない）オプションを提供しています。この機能はpmakeとともに配布されたCustomsライブラリを利用しています。pmakeは1989年にAdam de Boorにより開発された（そしてその後ずっとAndreas Stolckeにより保守されている）SpriteというOS用のmakeです。Customsライブラリは多くのコンピュータ上でmakeを並列に実行するための支援をします。Version 3.77以降 GNU makeは分散makeのためのCustomsライブラリサポートを提供しています。

Customsライブラリサポートを使うには、makeをソースから再構築しなければなりません。この作業はmakeの配布物に含まれるREADME.customsファイルで説明されています。まず最初にpmakeの配布物をダウンロードしてpmakeを構築する必要があります。（URLはREADME.customsファイルに書かれています）その後makeを--with-customsオプション付きで構築します。

Customsライブラリの肝は分散makeネットワークに参加しているホスト上で動作するcustomsデーモンです。これらのホストはNFSが提供するような共通のファイルシステム構造を持たなければなりません。1つのcustomsデーモンをマスタとして指定します。マスタは参加ホストリストに入っているホストを監視し、作業をそれぞれに割り当てます。--jobsオプションに1より大きな数が指定された場合には、makeはマスタと協調して作業をネットワーク上利用可能なホストで実行します。

Customsライブラリは広範囲な機能を提供しています。ホストはアーキテクチャごとにグループ化され性能で格付けが行われます。任意の属性をホストに割り当てることが可能で、属性と論理演算子の組み合わせによる指定で作業を割り当てることが可能です。さらに遊休時間、空きディスク容量、空きスワップ容量、現在のロードアベレージなどのホストの状態も作業を割り当てる際に考慮されます。

C、C++、Objective-Cを使うプロジェクトなら、複数のホスト上での分散コンパイルを行うためにdistcc（http://distcc.samba.org）を使うという選択肢もあります。distccはMartin PoolらによってSambaの構築をスピードアップするために開発されました。これはC、C++、Objective-Cを使うプロジェクトにとって頑強で完結した解を提供しています。このツールは単にCコンパイラをdistccプログラムに置き換えることで動作します。

```
$ make --jobs=8 CC=distcc
```

各コンパイルに対してdistccはローカルのコンパイラで前処理を行い、処理済みのソースを空い

ているリモートコンピュータに送り出しコンパイルします。そしてリモートコンピュータはコンパイル済みオブジェクトファイルを送り返します。この手法だと共通化したファイルシステムは必要なく、インストールおよび構成を劇的に単純化することができます。

コンパイルに使うホストを指定する方法は何種類もあります。distccを開始する前に環境変数にホスト名を設定しておくのがいちばん簡単な方法です。

```
$ export DISTCC_HOSTS='localhost wasatch oops'
```

ホストリストの設定、コンパイラとの連携、圧縮、検索パス、エラーと回復の処理などの構成はオプションを通して指定することができます。

ccacheはSambaプロジェクトリーダのAndrew Tridgellにより開発されたコンパイル性能を向上させるツールです。アイディアは非常に単純で、以前のコンパイル結果をキャッシュに取っておくだけです。コンパイルを行う前に求めるオブジェクトファイルがキャッシュに入っているかどうか調べます。このツールは複数のホストやネットワークすら必要としません。通常のコンパイルでは5倍から10倍の性能改善が見られると作者は報告しています。ccacheをコンパイルコマンドに前置するのが最も簡単な使い方です。

```
$ make CC='ccache gcc'
```

性能をさらに改善するためにccacheとdistccを組み合わせて使うことができます。どちらもCygwinツールとして利用が可能です。

11章
makefileの実例

本書のいたるところで登場したmakefileは実際の業務で使えるとともに、先進的な要求にもこたえることができるものです。しかし実在のプロジェクトで使われているmakefileを観察し、成果物を提供するというプレッシャの下で人々はmakeをどのように使っているかに触れることは非常に重要です。ここではmakefileの実例2つを詳細に取り上げます。1つのmakefileは本書を作成するために用意したものです。もう1つのmakefileはLinuxの2.6.7カーネルを構築するためのものです。

11.1 Book makefile

プログラミングに関する書籍の執筆は、それ自体システムの構築のためのよい訓練だといえます。本のテキストは多くのファイルから構成されていて、それぞれいろいろな前処理を必要とします。プログラムがその1つの例であり、意図したとおりに動作し出力が正しくなければならないうえに、後処理を経て本文に（エラーが混入しないようカット＆ペーストなしで済むように）取り込まれなければなりません。文章の構成を考えている間に、テキストを別の形式で見られるようになっていると便利です。そしてリリースする前に、適切にパッケージ化されなければなりません。もちろん、これらの作業のすべては繰り返し行うことが可能で、簡単に保守できる必要があります。

何かmake向きの機能に思えてきました。これはmakeの非常に優れた点です。makeは驚くほど広範囲の問題に対して適用することができるのです。本書（原書）はDocBookフォーマット（つまりXML）で書かれています。TeX、LaTex、troffなどを使う際にmakeを活用するのは、標準的な手順だといえます。

例11-1は本書のためのmakefile全文です。約440行あります。makefileは以下の基本的な作業に分けることができます。

- 例題の管理
- XMLの前処理
- 各種出力フォーマットの生成

- ソースの検証
- 基本的な保守作業

例11-1 本書を構築するためのmakefile

```
# Build the book!
#
# 以下主要なターゲット:
#
# show_pdf  pdfを生成し、ビューアを起動
# pdf       pdfの生成
# print     pdfの印刷
# show_html htmlを生成し、ビューアを起動
# html      htmlの生成
# xml       xmlの生成
# release   リリース用tarボール作成
# clean     生成ファイルの後片付け
#

BOOK_DIR     := /test/book
SOURCE_DIR   := text
OUTPUT_DIR   := out
EXAMPLES_DIR := examples

QUIET            = @

SHELL            =  bash
AWK              := awk
CP               := cp
EGREP            := egrep
HTML_VIEWER      := cygstart
KILL             := /bin/kill
M4               := m4
MV               := mv
PDF_VIEWER       := cygstart
RM               := rm -f
MKDIR            := mkdir -p
LNDIR            := lndir
SED              := sed
SORT             := sort
TOUCH            := touch
XMLTO            := xmlto
XMLTO_FLAGS      =  -o $(OUTPUT_DIR) $(XML_VERBOSE)
process-pgm      := bin/process-includes
make-depend      := bin/make-depend

m4-macros        := text/macros.m4

# $(call process-includes, input-file, output-file)
# タブの除去、マクロの展開およびinclude命令の処理
define process-includes
  expand $1 |                                                    \
```

```
  $(M4) --prefix-builtins --include=text $(m4-macros) - |       \
  $(process-pgm) > $2
endef

# $(call file-exists, file-name)
# ファイルが存在した場合にはnull以外を返す
file-exists = $(wildcard $1)

# $(call maybe-mkdir, directory-name-opt)
# ディレクトリが存在しなければ作成する
# directory-name-optが省略された場合には、$@をディレクトリ名とする
maybe-mkdir = $(if $(call file-exists,          \
                    $(if $1,$1,$(dir $@))),,    \
                $(MKDIR) $(if $1,$1,$(dir $@)))

# $(kill-acroread)
# Acrobat Readerを終了させる
define kill-acroread
  $(QUIET) ps -W |                                      \
  $(AWK) 'BEGIN { FIELDWIDTHS = "9 47 100" }            \
          /AcroRd32/ {                                  \
                       print "Killing " $$3;            \
                       system( "$(KILL) -f " $$1 )      \
                     }'
endef

# $(call source-to-output, file-name)
# ソースツリーの参照を出力ツリーの参照に変換する
define source-to-output
$(subst $(SOURCE_DIR),$(OUTPUT_DIR),$1)
endef

# $(call run-script-example, script-name, output-file)
# makefileの例題を実行する
define run-script-example
  ( cd $(dir $1);                                       \
    $(notdir $1) 2>&1 |                                 \
    if $(EGREP) --silent '\$$\(MAKE\)' [mM]akefile;     \
    then                                                \
      $(SED) -e 's/^++*/$$/';                           \
    else                                                \
      $(SED) -e 's/^++*/$$/'                            \
             -e '/ing directory /d'                     \
             -e 's/\[[0-9]\]//';                        \
    fi )                                                \
  > $(TMP)/out.$$$$ &                                   \
  $(MV) $(TMP)/out.$$$$ $2
endef

# $(call generic-program-example,example-directory)
# 例題を構築するためのルールを作成する
define generic-program-example
  $(eval $1_dir      := $(OUTPUT_DIR)/$1)
```

```
  $(eval $1_make_out := $($1_dir)/make.out)
  $(eval $1_run_out  := $($1_dir)/run.out)
  $(eval $1_clean    := $($1_dir)/clean)
  $(eval $1_run_make := $($1_dir)/run-make)
  $(eval $1_run_run  := $($1_dir)/run-run)
  $(eval $1_sources  := $(filter-out %/CVS, $(wildcard $(EXAMPLES_DIR)/$1/*)))

  $($1_run_out): $($1_make_out) $($1_run_run)
        $$(call run-script-example, $($1_run_run), $$@)

  $($1_make_out): $($1_clean) $($1_run_make)
        $$(call run-script-example, $($1_run_make), $$@)

  $($1_clean): $($1_sources) Makefile
        $(RM) -r $($1_dir)
        $(MKDIR) $($1_dir)
        $(LNDIR) -silent ../../$(EXAMPLES_DIR)/$1 $($1_dir)
        $(TOUCH) $$@

  $($1_run_make):
        printf "#! /bin/bash -x\nmake\n" > $$@
endef

# 出力フォーマット
BOOK_XML_OUT      := $(OUTPUT_DIR)/book.xml
BOOK_HTML_OUT     := $(subst xml,html,$(BOOK_XML_OUT))
BOOK_FO_OUT       := $(subst xml,fo,$(BOOK_XML_OUT))
BOOK_PDF_OUT      := $(subst xml,pdf,$(BOOK_XML_OUT))
ALL_XML_SRC       := $(wildcard $(SOURCE_DIR)/*.xml)
ALL_XML_OUT       := $(call source-to-output,$(ALL_XML_SRC))
DEPENDENCY_FILES := $(call source-to-output,$(subst .xml,.d,$(ALL_XML_SRC)))

# xml/html/pdf - 指定したフォーマットの出力を生成する
.PHONY: xml html pdf
xml:  $(OUTPUT_DIR)/validate
html: $(BOOK_HTML_OUT)
pdf:  $(BOOK_PDF_OUT)

# show_pdf - pdfファイルを生成し、表示する
.PHONY: show_pdf show_html print
show_pdf: $(BOOK_PDF_OUT)
        $(kill-acroread)
        $(PDF_VIEWER) $(BOOK_PDF_OUT)

# show_html - htmlファイルを生成し、表示する
show_html: $(BOOK_HTML_OUT)
        $(HTML_VIEWER) $(BOOK_HTML_OUT)

# print - 指定したページを印刷する
print: $(BOOK_FO_OUT)
        $(kill-acroread)
        java -Dstart=15 -Dend=15 $(FOP) $< -print > /dev/null
```

```
# $(BOOK_PDF_OUT) - pdfファイルを生成する
$(BOOK_PDF_OUT): $(BOOK_FO_OUT) Makefile

# $(BOOK_HTML_OUT) - htmlファイルを生成する
$(BOOK_HTML_OUT): $(ALL_XML_OUT) $(OUTPUT_DIR)/validate Makefile

# $(BOOK_FO_OUT) - fo中間ファイルを生成する
.INTERMEDIATE: $(BOOK_FO_OUT)
$(BOOK_FO_OUT): $(ALL_XML_OUT) $(OUTPUT_DIR)/validate Makefile

# $(BOOK_XML_OUT) - xml入力ファイルをすべて処理する
$(BOOK_XML_OUT): Makefile

######################################################################
# FOP サポート
#
FOP := org.apache.fop.apps.Fop

# DEBUG_FOP - fopプロセッサの出力を見る際に設定する
ifndef DEBUG_FOP
  FOP_FLAGS  := -q
  FOP_OUTPUT := | $(SED) -e '/not implemented/d'         \
                         -e '/relative-align/d'          \
                         -e '/xsl-footnote-separator/d'
endif

# CLASSPATH - fopのための適切なCLASSPATHを定義する
export CLASSPATH
CLASSPATH = $(patsubst %;,%,                                          \
              $(subst ; ,;,                                           \
                $(addprefix c:/usr/xslt-process-2.2/java/,            \
                  $(addsuffix .jar;,                                  \
                    xalan                                             \
                    xercesImpl                                        \
                    batik                                             \
                    fop                                               \
                    jimi-1.0                                          \
                    avalon-framework-cvs-20020315))))

# %.pdf - foファイルからpdfを生成するためのパターンルール
%.pdf: %.fo
        $(kill-acroread)
        java -Xmx128M $(FOP) $(FOP_FLAGS) $< $@ $(FOP_OUTPUT)

# %.fo - xmlファイルからfoファイルを生成するためのパターンルール
PAPER_SIZE := letter
%.fo: %.xml
        XSLT_FLAGS="--stringparam paper.type $(PAPER_SIZE)" \
        $(XMLTO) $(XMLTO_FLAGS) fo $<

# %.html - xmlファイルからhtmlファイルを生成するためのパターンルール
%.html: %.xml
        $(XMLTO) $(XMLTO_FLAGS) html-nochunks $<
```

```
# fop_help - fopプロセッサのヘルプを表示する
.PHONY: fop_help
fop_help:
        -java org.apache.fop.apps.Fop -help
        -java org.apache.fop.apps.Fop -print help

######################################################################
# release - リリースパッケージの作成
#
RELEASE_TAR   := mpwm-$(shell date +%F).tar.gz
RELEASE_FILES := README Makefile *.pdf bin examples out text

.PHONY: release
release: $(BOOK_PDF_OUT)
        ln -sf $(BOOK_PDF_OUT) .
        tar --create                            \
            --gzip                              \
            --file=$(RELEASE_TAR)               \
            --exclude=CVS                       \
            --exclude=semantic.cache            \
            --exclude=*~                        \
            $(RELEASE_FILES)
        ls -l $(RELEASE_TAR)

######################################################################
# 1章の例題のためのルール
#

# 以下は例題が格納されたディレクトリ
EXAMPLES :=                                     \
                ch01-bogus-tab                  \
                ch01-cw1                        \
                ch01-hello                      \
                ch01-cw2                        \
                ch01-cw2a                       \
                ch02-cw3                        \
                ch02-cw4                        \
                ch02-cw4a                       \
                ch02-cw5                        \
                ch02-cw5a                       \
                ch02-cw5b                       \
                ch02-cw6                        \
                ch02-make-clean                 \
                ch03-assert-not-null            \
                ch03-debug-trace                \
                ch03-debug-trace-1              \
                ch03-debug-trace-2              \
                ch03-filter-failure             \
                ch03-find-program-1             \
                ch03-find-program-2             \
                ch03-findstring-1               \
                ch03-grep                       \
                ch03-include                    \
```

```
                ch03-invalid-variable           \
                ch03-kill-acroread              \
                ch03-kill-program               \
                ch03-letters                    \
                ch03-program-variables-1        \
                ch03-program-variables-2        \
                ch03-program-variables-3        \
                ch03-program-variables-5        \
                ch03-scoping-issue              \
                ch03-shell                      \
                ch03-trailing-space             \
                ch04-extent                     \
                ch04-for-loop-1                 \
                ch04-for-loop-2                 \
                ch04-for-loop-3                 \
                ch06-simple                     \
                appb-defstruct                  \
                appb-arithmetic

# 本当はforeachを使いたいところだが、3.80のバグにより致命的なエラーと
# なってしまう
#$(foreach e,$(EXAMPLES),$(eval $(call generic-program-example,$e)))

# そこで、展開済みのリストをここに置く
$(eval $(call generic-program-example,ch01-bogus-tab))
$(eval $(call generic-program-example,ch01-cw1))
$(eval $(call generic-program-example,ch01-hello))
$(eval $(call generic-program-example,ch01-cw2))
$(eval $(call generic-program-example,ch01-cw2a))
$(eval $(call generic-program-example,ch02-cw3))
$(eval $(call generic-program-example,ch02-cw4))
$(eval $(call generic-program-example,ch02-cw4a))
$(eval $(call generic-program-example,ch02-cw5))
$(eval $(call generic-program-example,ch02-cw5a))
$(eval $(call generic-program-example,ch02-cw5b))
$(eval $(call generic-program-example,ch02-cw6))
$(eval $(call generic-program-example,ch02-make-clean))
$(eval $(call generic-program-example,ch03-assert-not-null))
$(eval $(call generic-program-example,ch03-debug-trace))
$(eval $(call generic-program-example,ch03-debug-trace-1))
$(eval $(call generic-program-example,ch03-debug-trace-2))
$(eval $(call generic-program-example,ch03-filter-failure))
$(eval $(call generic-program-example,ch03-find-program-1))
$(eval $(call generic-program-example,ch03-find-program-2))
$(eval $(call generic-program-example,ch03-findstring-1))
$(eval $(call generic-program-example,ch03-grep))
$(eval $(call generic-program-example,ch03-include))
$(eval $(call generic-program-example,ch03-invalid-variable))
$(eval $(call generic-program-example,ch03-kill-acroread))
$(eval $(call generic-program-example,ch03-kill-program))
$(eval $(call generic-program-example,ch03-letters))
$(eval $(call generic-program-example,ch03-program-variables-1))
$(eval $(call generic-program-example,ch03-program-variables-2))
```

```
$(eval $(call generic-program-example,ch03-program-variables-3))
$(eval $(call generic-program-example,ch03-program-variables-5))
$(eval $(call generic-program-example,ch03-scoping-issue))
$(eval $(call generic-program-example,ch03-shell))
$(eval $(call generic-program-example,ch03-trailing-space))
$(eval $(call generic-program-example,ch04-extent))
$(eval $(call generic-program-example,ch04-for-loop-1))
$(eval $(call generic-program-example,ch04-for-loop-2))
$(eval $(call generic-program-example,ch04-for-loop-3))
$(eval $(call generic-program-example,ch06-simple))
$(eval $(call generic-program-example,ch10-echo-bash))
$(eval $(call generic-program-example,appb-defstruct))
$(eval $(call generic-program-example,appb-arithmetic))

######################################################################
# validate
#
# 次のチェックを行う a) 展開されなかったm4マクロ; b) タブ;
# c) FIXMEコメント; d) RM: Andyへの返答; e) 重複したm4マクロ
#
validation_checks := $(OUTPUT_DIR)/chk_macros_tabs              \
                     $(OUTPUT_DIR)/chk_fixme                    \
                     $(OUTPUT_DIR)/chk_duplicate_macros         \
                     $(OUTPUT_DIR)/chk_orphaned_examples

.PHONY: validate-only
validate-only: $(OUTPUT_DIR)/validate
$(OUTPUT_DIR)/validate: $(validation_checks)
        $(TOUCH) $@

$(OUTPUT_DIR)/chk_macros_tabs: $(ALL_XML_OUT)
        # マクロとタブを検索
        $(QUIET)! $(EGREP) --ignore-case                                \
                           --line-number                                \
                           --regexp='\b(m4_|mp_)'                       \
                           --regexp='\011'                              \
                           $^
        $(TOUCH) $@

$(OUTPUT_DIR)/chk_fixme: $(ALL_XML_OUT)
        # RM:とFIXME:を検索
        $(QUIET)$(AWK)                                                  \
                '/FIXME/ { printf "%s:%s: %s\n", FILENAME, NR, $$0 }    \
                 /^ *RM:/  {                                            \
                           if ( $$0 !~ /RM: Done/ )                     \
                           printf "%s:%s: %s\n", FILENAME, NR, $$0      \
                         }' $(subst $(OUTPUT_DIR)/,$(SOURCE_DIR)/,$^)
        $(TOUCH) $@

$(OUTPUT_DIR)/chk_duplicate_macros: $(SOURCE_DIR)/macros.m4
        # 重複したマクロを検索
        $(QUIET)! $(EGREP) --only-matching              \
              "\`[^']+'," $< |                          \
```

```
        $(SORT) |                                               \
        uniq -c |                                               \
        $(AWK) '$$1 > 1 { printf "$<:0: %s\n", $$0 }' | \
        $(EGREP) "^"
        $(TOUCH) $@

ALL_EXAMPLES := $(TMP)/all_examples

$(OUTPUT_DIR)/chk_orphaned_examples: $(ALL_EXAMPLES) $(DEPENDENCY_FILES)
        $(QUIET)$(AWK) -F/ '/(EXAMPLES|OUTPUT)_DIR/ { print $$3 }'      \
                $(filter %.d,$^) |                                      \
        $(SORT) -u |                                                    \
        comm -13 - $(filter-out %.d,$^)
        $(TOUCH) $@

.INTERMEDIATE: $(ALL_EXAMPLES)
$(ALL_EXAMPLES):
        # 使われていない例題を検索
        $(QUIET) ls -p $(EXAMPLES_DIR) |        \
        $(AWK) '/CVS/ { next }                  \
                /\//  { print substr($$0, 1, length - 1) }' > $@

######################################################################
# clean
#
clean:
        $(kill-acroread)
        $(RM) -r $(OUTPUT_DIR)
        $(RM) $(SOURCE_DIR)/*~ $(SOURCE_DIR)/*.log semantic.cache
        $(RM) book.pdf

######################################################################
# 依存関係管理
#
# cleanを行っているときにはインクルードファイルの読み込みや
# 再作成を行わない
#
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(DEPENDENCY_FILES)
endif

vpath %.xml $(SOURCE_DIR)
vpath %.tif $(SOURCE_DIR)
vpath %.eps $(SOURCE_DIR)

$(OUTPUT_DIR)/%.xml: %.xml $(process-pgm) $(m4-macros)
        $(call process-includes, $<, $@)

$(OUTPUT_DIR)/%.tif: %.tif
        $(CP) $< $@

$(OUTPUT_DIR)/%.eps: %.eps
        $(CP) $< $@
```

```
$(OUTPUT_DIR)/%.d: %.xml $(make-depend)
        $(make-depend) $< > $@

#################################################################
# Create Output Directory
#
# 必要があれば出力ディレクトリを作成する
#
DOCBOOK_IMAGES := $(OUTPUT_DIR)/release/images
DRAFT_PNG      := /usr/share/docbook-xsl/images/draft.png

ifneq "$(MAKECMDGOALS)" "clean"
  _CREATE_OUTPUT_DIR :=                                                  \
    $(shell                                                              \
      $(MKDIR) $(DOCBOOK_IMAGES) &                                       \
      $(CP) $(DRAFT_PNG) $(DOCBOOK_IMAGES);                              \
      if ! [[ $(foreach d,                                               \
                 $(notdir                                                \
                   $(wildcard $(EXAMPLES_DIR)/ch*)),                     \
                 -e $(OUTPUT_DIR)/$d &) -e . ]];                         \
      then                                                               \
        echo Linking examples... > /dev/stderr;                          \
        $(LNDIR) $(BOOK_DIR)/$(EXAMPLES_DIR) $(BOOK_DIR)/$(OUTPUT_DIR); \
      fi)
endif
```

このmakefileはCygwinの環境で動作するように作成され、Unix環境への移植はまじめに考えていません。しかしながら、変数の値を変えたり追加の変数を用意しても解決できないような非互換性はほとんどないものと思います。

最初にある変数定義セクションではルートディレクトリやテキスト、例題、出力ディレクトリなどの相対位置を定義しています。続いてmakefile内で使われるあまり重要でないプログラムの名前を変数として定義しています。

11.1.1 例題の管理

最初の作業である例題の管理が最も複雑です。各例題はそれぞれ固有の*book/examples/chn-<title>*というディレクトリ下に置かれています。例題はmakefileとその他必要なファイルおよびディレクトリで構成されています。makefileを実行することにより作成されるものをソースディレクトリの中に残さないように、ある例題を実行する前にはディレクトリを作り、その中にソースツリーへのシンボリックリンクを作成してから、例題をそこで実行します。さらに期待された出力を行うために、たいていの例題ではmakefileの存在するディレクトリをカレントディレクトリにする必要があります。ソースファイルへのシンボリックリンクを作成した後で、適切な引数を伴ってmakeを呼び出すためにrun-makeシェルスクリプトを実行します。もしシェルスクリプトが存在していなければ、既定のスクリプトを作ることができます。run-makeスクリプトの出力はmake.outに保存されます。いくつかの例題では実行ファイルも作成され、それも実行しなければなりません。これはrun-runシェルスクリプトを実行し出力をrun.outに保存することで行います。

シンボリックリンクの作成はmakefileの最後にある次のコードで行います。

```
ifneq "$(MAKECMDGOALS)" "clean"
  _CREATE_OUTPUT_DIR := \
    $(shell \
      …
      if ! [[ $(foreach d, \
                $(notdir \
                  $(wildcard $(EXAMPLES_DIR)/ch*)), \
                -e $(OUTPUT_DIR)/$d &&) -e . ]]; \
      then \
        echo Linking examples... > /dev/stderr; \
        $(LNDIR) $(BOOK_DIR)/$(EXAMPLES_DIR) $(BOOK_DIR)/$(OUTPUT_DIR); \
      fi)
endif
```

このコードはifneq条件判断に囲まれた1つの単純変数定義です。条件判断はmake cleanを実行する際には出力ディレクトリの作成を行わないために使われています。定義される変数はダミーであり、実際に使われることはありません。しかし、右辺にあるshell関数はmakeがmakefileを読み込んだときに即時実行されます。shell関数は例題のディレクトリが出力ディレクトリに存在するかを調べ、その中のどれかが存在していなければ、lndirコマンドが実行されシンボリックリンクツリーが更新されます†。

ifで使われている条件は詳しく見ておく必要があります。条件は各ディレクトリに対してそれぞれ1つずつの-eによるチェック（つまり、そのファイルは存在しているか？）で構成されています。実際のコードは次のように動作します。wildcard関数を使いすべての例題ディレクトリを洗い出し、notdir関数でディレクトリ部を取り出し、各例題ディレクトリに対して-e $(OUTPUT_DIR)/dir &&というテキストを生成します。それらを結合してbashの[[...]]条件の中に入れます。そして結果を反転します。foreachループの中で各要素の最後で単純に&&を加えることができるように、-e .が追加されます。

これにより新しいディレクトリが発見されるたびにそれが確実に追加されるようになります。

次のステップは、2つの出力ファイルmake.outとrun.outを更新するためのルール作成です。これは、それぞれの例題用.outファイルを作成するユーザ定義関数により行われます。

```
# $(call generic-program-example,example-directory)
# 例題を構築するためのルールを作成する
define generic-program-example
  $(eval $1_dir := $(OUTPUT_DIR)/$1)
  $(eval $1_make_out := $($1_dir)/make.out)
  $(eval $1_run_out := $($1_dir)/run.out)
  $(eval $1_clean := $($1_dir)/clean)
```

† 訳注：lndirはX Window Systemに付属しているユーティリティプログラムの1つです。lndirを使うと、あるディレクトリのコピーを作ることができますが、単純な複製ではなく、その中のファイルは元のファイルへのシンボリックリンクが使われます。これによりファイルを二重化させることなく、オリジナルとは別のディレクトリツリーを作成することが可能となります。

```
  $(eval $1_run_make := $($1_dir)/run-make)
  $(eval $1_run_run := $($1_dir)/run-run)
  $(eval $1_sources := $(filter-out %/CVS, $(wildcard $(EXAMPLES_DIR)/$1/*)))

  $($1_run_out): $($1_make_out) $($1_run_run)
        $$(call run-script-example, $($1_run_run), $$@)

  $($1_make_out): $($1_clean) $($1_run_make)
        $$(call run-script-example, $($1_run_make), $$@)

  $($1_clean): $($1_sources) Makefile
        $(RM) -r $($1_dir)
        $(MKDIR) $($1_dir)
        $(LNDIR) -silent ../../$(EXAMPLES_DIR)/$1 $($1_dir)
        $(TOUCH) $$@

  $($1_run_make):
        printf "#! /bin/bash -x\nmake\n" > $$@
endef
```

この関数はそれぞれの例題ディレクトリに対して1回ずつ次のように使われます。

```
$(eval $(call generic-program-example,ch01-bogus-tab))
$(eval $(call generic-program-example,ch01-cw1))
$(eval $(call generic-program-example,ch01-hello))
$(eval $(call generic-program-example,ch01-cw2))
```

関数の先頭にある変数定義の多くは、利便性と可読性を向上させるためのものです。さらに、それらの変数がマクロの中で別途$を付けた展開を行わなくても直接使えるように、evalの中で変数が定義されています。

この関数の肝は最初の2つのターゲット$($1_run_out)と$($1_make_out)にあります。これはそれぞれの例題に対してrun.outとmake.outを作成するためのものです。変数名は例題ディレクトリの名前から構成されており、_run_outと_make_outサフィックスが付けられます。

最初のルールでは、run.outがmake.outとrun-runスクリプトに依存していることを示しています。つまり、makeが実行されるか、run-runスクリプトが変更された際には、例題が再実行されます。このターゲットはrun-script-example関数により更新されます。

```
# $(call run-script-example, script-name, output-file)
# 例題のmakefileを実行する
define run-script-example
  ( cd $(dir $1); \
    $(notdir $1) 2>&1 | \
    if $(EGREP) --silent '\$$\(MAKE\)' [mM]akefile; \
    then \
      $(SED) -e 's/^++*/$$/'; \
    else \
      $(SED) -e 's/^++*/$$/' \
             -e '/ing directory /d' \
             -e 's/\[[0-9]\]//'; \
```

```
    fi ) \
  > $(TMP)/out.$$$$ && \
  $(MV) $(TMP)/out.$$$$ $2
endef
```

この関数はスクリプトへのパス名と出力先ファイル名を必要とします。カレントディレクトリをスクリプトのディレクトリに移動し、スクリプトを実行します。標準出力と標準エラー出力はパイプにつながっていて、フィルタを通して整形されます†。

make.outターゲットも同様ですが、もう少し複雑です。新しいファイルが例題に追加されると、その状況を検知して例題の再構築を行わなければなりません。_CREATE_OUTPUT_DIRのコードは新しいディレクトリが加わったときにシンボリックリンクを作り直すのであって、ファイルが追加されたときではありません。この状況を検知するには、各例題のディレクトリにタイムスタンプを表すファイルを配置し、最後にいつlndirが実行されたかを示すようにします。$($1_clean)ターゲットは（出力ディレクトリのシンボリックリックではなく）例題ディレクトリのファイルに依存しており、タイムスタンプファイルを更新します。makeの依存関係チェックによりタイムスタンプファイルよりも新しいファイルが例題ディレクトリに存在することが判明すると、コマンドスクリプトが実行されシンボリックリンクが入っている出力先ディレクトリが削除され、新しく作り直された後、新しくタイムスタンプファイルが作成されます。この作業はmakefile自身が更新されたときにも実行されます。

makefileを実行するためのrun-makeシェルスクリプトは、通常次のような2行のスクリプトになります。

```
#! /bin/bash -x
make
```

この程度の決まりきったスクリプトファイルを作るのは退屈な作業です。そこで$($1_make_out)の必須項目として$($1_run_make)ターゲットを用意し、そこでこのスクリプトを作成します。もしこのファイルが存在していなければ、出力ファイルに生成します。

各例題ディレクトリに対してgeneric-program-exampleを実行すると、例題の実行とXMLファイルに取り込まれる出力例作成のためのルールを、すべて作成します。これらのルールはmakefileにインクルードされる依存関係ファイルにより起動されます。例えば、1章に関する依存関係ファイルは次のようになります。

```
out/ch01.xml: $(EXAMPLES_DIR)/ch01-hello/Makefile
out/ch01.xml: $(OUTPUT_DIR)/ch01-hello/make.out
```

† 整形処理は複雑です。通常run-runとrun-makeスクリプトはbash -xを使って実際のmakeコマンド行が表示されるようにしています。-xオプションにより出力される各コマンドの前には++が置かれますが、整形スクリプトではシェルのプロンプトを表す$に置き換えます。しかしながら出力されるのはコマンド行だけではありません。ここではmakeの例題を実行しているので、結局別のmakeを実行することになります。単純なmakefileに対して、Entering directory...とかLeaving directory...といったよけいなメッセージがmakeの呼び出し回数を表す値とともに表示されてしまいます。makeを再帰的に実行しない単純なmakefileに対しては、あたかもシェルから直接起動されたかのように見せるため、これらの適切ではないメッセージを取り除いてしまいます。

```
out/ch01.xml: $(EXAMPLES_DIR)/ch01-cw1/count_words.c
out/ch01.xml: $(EXAMPLES_DIR)/ch01-cw1/lexer.l
out/ch01.xml: $(EXAMPLES_DIR)/ch01-cw1/Makefile
out/ch01.xml: $(OUTPUT_DIR)/ch01-cw1/make.out
out/ch01.xml: $(EXAMPLES_DIR)/ch01-cw2/lexer.l
out/ch01.xml: $(OUTPUT_DIR)/ch01-cw2/make.out
out/ch01.xml: $(OUTPUT_DIR)/ch01-cw2/run.out
out/ch01.xml: $(OUTPUT_DIR)/ch01-bogus-tab/make.out
```

この依存関係は、ひねりのない名前ですがmake-dependというファイル名の単純なawkスクリプトにより生成されます†。

```
#! /bin/awk -f
function generate_dependency( prereq )
{
  filename = FILENAME
  sub( /text/, "out", filename )
  print filename ": " prereq
}

/^ *include-program/ {
  generate_dependency( "$(EXAMPLES_DIR)/" $2 )
}

/^ *mp_program\(/ {
  match( $0, /\((.*)\)/, names )
  generate_dependency( "$(EXAMPLES_DIR)/" names[1] )
}

/^ *include-output/ {
  generate_dependency( "$(OUTPUT_DIR)/" $2 )
}

/^ *mp_output\(/ {
  match( $0, /\((.*)\)/, names )
  generate_dependency( "$(OUTPUT_DIR)/" names[1] )
}

/graphic fileref/ {
  match( $0, /"(.*)"/, out_file )
  generate_dependency( out_file[1] );
}
```

このスクリプトは、次のようなパターンを探します。

```
mp_program(ch01-hello/Makefile)
mp_output(ch01-hello/make.out)
```

† 訳注：このawkスクリプトは、match関数の拡張された機能を使っているので、実行するにはGNU awkが必要です。

このスクリプトはソースファイル名とfilenameパラメータより依存関係を作ります（mp_programマクロはプログラムリストフォーマットに変換し、mp_outputマクロはプログラムの出力フォーマットに変換するためのものです†）。

そして、依存関係ファイルの生成はいつものとおりinclude命令により開始されます。

```
# $(call source-to-output, file-name)
# ソースツリーの参照を出力ツリーの参照に変換する
define source-to-output
  $(subst $(SOURCE_DIR),$(OUTPUT_DIR),$1)
endef
...
ALL_XML_SRC := $(wildcard $(SOURCE_DIR)/*.xml)
DEPENDENCY_FILES := $(call source-to-output,$(subst .xml,.d,$(ALL_XML_SRC)))
...
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(DEPENDENCY_FILES)
endif
vpath %.xml $(SOURCE_DIR)
...
$(OUTPUT_DIR)/%.d: %.xml $(make-depend)
        $(make-depend) $< > $@
```

例題を処理するためのコードは以上です。複雑さの原因は、例題の出力とmakeからの出力とに加えて例題のmakefile自体も取り込もうとしたところにあります。

11.1.2 XMLの処理

この先ずっと筆者が俗物であるとレッテルを貼られることを覚悟のうえで、告白しなければならないことがあります。筆者はXMLが好きではありません。XMLは扱いにくくて冗長だと思います。ということから、原稿をDocBook形式で書かなければならないとわかったとき、この痛みを軽減するために、よく使われているツールを探してみました。m4マクロプロセッサとawkは、この点で非常に役に立ちます。

XMLの冗長な文法の排除と、XMLで相互参照のために使われる識別子の管理というDocBookとXMLに関する2つの問題を解決するのにm4はうってつけのツールです。例えば、ある単語をDocBookで強調するときには、次のように書く必要があります。

```
<emphasis>not</emphasis>
```

m4を使えば、これを次のように書くための簡単なマクロを作ることができます。

```
mp_em(not)
```

† 訳注：この少し後で出てきますが、mp_programやmp_outputはm4マクロの1種です。このawkスクリプトはテキストのソースファイル、例えばtext/ch01.xmlを入力としてその中で使われているファイル名をパラメータとするいくつかのm4マクロを走査して、依存関係を作り出します。

かなり気が楽になりました。さらにいうと、各構成要素に対してmp_variableとかmp_targetといったフォーマットスタイルシンボルを使っています。これにより、最初は簡単な書いたとおりのフォーマットを使っておいて、後で製作部門がフォーマットを変更するときに全文検索と置換を行わなくても済むようになっています。

相互参照で使うXMLの識別子を扱うのがm4のもう1つの仕事です。章、セクション、例題、表それぞれには識別子が付けられています。

```
<sect1 id="MPWM-CH-7-SECT-1">
```

章を参照するには、この識別子を使わなければなりません。これはプログラミングの観点からすると明らかに問題です。こういった識別子は「コード」中にばらまかれた複雑な定数です。さらにいうと、この名前に特別な意味はありません。7章に何が書かれているのか覚えてなんていられません。m4を使うことで、複雑な定数があちこちに直接書かれることを防ぎ、もっと意味のある名前を付けることができるようになります。

```
<sect1 id="mp_se_makedepend">
```

もっと重要なことに、これは頻繁に起きることですが、もし章やセクションがずれてしまった場合には1つのファイルにあるいくつかの定数を書き換えるだけで済みます。章のなかでセクションが入れ替わってしまったときのことを考えると、その利点は注目に値します。もしこういった変数による参照を使わなければ、すべてのファイルに対して5、6回は全文検索と置換を行わなければなりません。

m4マクロの例をいくつか紹介します†。

```
m4_define(`mp_tag', `<$1>`$2'</$1>')
m4_define(`mp_lit', `mp_tag(literal, `$1')')

m4_define(`mp_cmd', `mp_tag(command,`$1')')
m4_define(`mp_target', `mp_lit($1)')

m4_define(`mp_all', `mp_target(all)')
m4_define(`mp_bash', `mp_cmd(bash)')

m4_define(`mp_ch_examples', `MPWM-CH-11')
m4_define(`mp_se_book', `MPWM-CH-11.1')
m4_define(`mp_ex_book_makefile',`MPWM-CH-11-EX-1')
```

もう1つの前処理作業は、以前取り上げた例題のテキストを吸い上げるためのインクルード機能の作成です。このテキストはタブを空白に変換し（というのも、O'ReillyのDocBookコンバータはタブを扱うことができないのですが、makefileにはタブが多数含まれているためです）、[CDATA[...]]の中に入れて特殊文字を保護し、例題ファイルの最初と最後に入っている余分な改行を削除する必要があります。これはprocess-includesという名前の小さなawkプログラムで処理します。

† mpプリフィクスは“Managing Projects”（本書のタイトル）とか“macro processor”とか“make pretty”など、好きなように解釈していただいて結構です。

```
#! /usr/bin/awk -f
function expand_cdata( dir )
{
  start_place = match( $1, "include-" )
  if ( start_place > 0 )
  {
    prefix = substr( $1, 1, start_place - 1 )
  }
  else
  {
    print "Bogus include '" $0 "'" > "/dev/stderr"
  }

  end_place = match( $2, "(</(programlisting|screen)>.*)$", tag )
  if ( end_place > 0 )
  {
    file = dir substr( $2, 1, end_place - 1 )
  }
  else
  {
    print "Bogus include '" $0 "'" > "/dev/stderr"
  }

  command = "expand " file

  printf "%s>&33;&91;CDATA[", prefix
  tail = 0
  previous_line = ""
  while ( (command | getline line) > 0 )
  {
    if ( tail )
      print previous_line;

    tail = 1
    previous_line = line
  }

  printf "%s&93;&93;&62;%s\n", previous_line, tag[1]
  close( command )
}

/include-program/ {
  expand_cdata( "examples/" )
  next;
}

/include-output/ {
  expand_cdata( "out/" )
  next;
}

/<(programlisting|screen)> *$/ {
  # 現位置での字下げ数を数える
```

```
  offset = match( $0, "<(programlisting|screen)>" )

  # 改行の削除
  printf $0

  # プログラムの読み込み...
  tail = 0
  previous_line = ""
  while ( (getline line) > 0 )
  {
    if ( line ~ "</(programlisting|screen)>" )
    {
      gsub( /^ */, "", line )
      break
    }

    if ( tail )
      print previous_line

    tail = 1
    previous_line = substr( line, offset + 1 )
  }

  printf "%s%s\n", previous_line, line

  next
}

{
  print
}
```

このmakefileでは、XMLファイルをソースツリーから出力ツリーへとコピーしながら、タブの変換とマクロの展開とインクルードファイルの取り込みを行います。

```
process-pgm := bin/process-includes
m4-macros := text/macros.m4

# $(call process-includes, input-file, output-file)
# タブの除去、マクロの展開、インクルード命令の処理を行う
define process-includes
  expand $1 | \
  $(M4) --prefix-builtins --include=text $(m4-macros) - | \
  $(process-pgm) > $2
endef

vpath %.xml $(SOURCE_DIR)

$(OUTPUT_DIR)/%.xml: %.xml $(process-pgm) $(m4-macros)
        $(call process-includes, $<, $@)
```

このパターンルールはどのようにXMLファイルをソースツリーから取り出して出力ツリーに書き込むかを示しています。このルールではマクロやインクルード処理用のプログラムが変更された際には、XMLファイルが再処理されることも示しています。

11.1.3 出力の生成

テキストをフォーマットしたりプリントしたり画面に表示するためのものを、この時点まで何も生成していません。それらはこのmakefileが本をフォーマットするためのものなら、明らかに重要な機能のはずです。必要なフォーマット形式は2つあります。HTMLとPDFです。

まず最初にHTMLにフォーマットする方法を示します。小さいけれどもすばらしいプログラムがあります。xsltprocとサポートスクリプトであるxmltoで、筆者は仕事でよく使っていました。これを使うと、処理は非常に簡単になります。

```
# 出力フォーマット
BOOK_XML_OUT      := $(OUTPUT_DIR)/book.xml
BOOK_HTML_OUT     := $(subst xml,html,$(BOOK_XML_OUT))
ALL_XML_SRC       := $(wildcard $(SOURCE_DIR)/*.xml)
ALL_XML_OUT       := $(call source-to-output,$(ALL_XML_SRC))

# html - 指定したフォーマットの出力を生成する
.PHONY: html
html: $(BOOK_HTML_OUT)

# show_html - htmlファイルを生成し、表示する
.PHONY: show_html
show_html: $(BOOK_HTML_OUT)
        $(HTML_VIEWER) $(BOOK_HTML_OUT)

# $(BOOK_HTML_OUT) - htmlファイルを生成する
$(BOOK_HTML_OUT): $(ALL_XML_OUT) $(OUTPUT_DIR)/validate Makefile

# %.html - xmlファイルからhtmlファイルを生成するためのパターンルール
%.html: %.xml
        $(XMLTO) $(XMLTO_FLAGS) html-nochunks $<
```

このパターンルールがXMLファイルをHTMLファイルに変換する作業のほとんどを行います。本書のデータは1つのトップレベルファイルbook.xmlに集約され、各章をインクルードしています。トップレベルファイルはBOOK_XML_OUT変数で表されています。それに対応するHTMLファイルはターゲットにもなっているBOOK_HTML_OUTです。BOOK_HTML_OUTはインクルードされるXMLファイルを必須項目として持っています。利便性のために2つの擬似ターゲットが定義されています。htmlとshow_htmlはそれぞれHTMLの作成と、ブラウザによる表示を表しています。

原理上は簡単なのですが、PDFの生成はかなり複雑になります。xsltprocプログラムはPDFファイルを直接生成することができるはずなのですが、筆者の環境ではうまく動きませんでした。この作業はWindows上でCygwinの環境で行いますが、Cygwin版のxsltprocはPOSIX形式のパス名を使います。筆者の使うDocBookの改造版も原稿自体もWindows形式のパス名を使っています。おそら

くこの差異が、解決できないxsltprocの問題を引き起こしたのだと思います。そこでxsltprocを使ってXML Formatting Objectを生成し、Javaプログラム FOP（http://xml.apache.org/fop）を使ってPDFを生成します。

このような理由からPDFを生成するコードは多少長くなっています。

```
# 出力フォーマット
BOOK_XML_OUT      := $(OUTPUT_DIR)/book.xml
BOOK_FO_OUT       := $(subst xml,fo,$(BOOK_XML_OUT))
BOOK_PDF_OUT      := $(subst xml,pdf,$(BOOK_XML_OUT))
ALL_XML_SRC       := $(wildcard $(SOURCE_DIR)/*.xml)
ALL_XML_OUT       := $(call source-to-output,$(ALL_XML_SRC))

# pdf - 指定したフォーマットの出力を生成する
.PHONY: pdf
pdf:  $(BOOK_PDF_OUT)

# show_pdf - pdfファイルを生成し、表示する
.PHONY: show_pdf
show_pdf: $(BOOK_PDF_OUT)
        $(kill-acroread)
        $(PDF_VIEWER) $(BOOK_PDF_OUT)

# $(BOOK_PDF_OUT) - pdfファイルを生成する
$(BOOK_PDF_OUT): $(BOOK_FO_OUT) Makefile

# $(BOOK_FO_OUT) - fo中間ファイルを生成する
.INTERMEDIATE: $(BOOK_FO_OUT)
$(BOOK_FO_OUT): $(ALL_XML_OUT) $(OUTPUT_DIR)/validate Makefile

# FOP サポート
FOP := org.apache.fop.apps.Fop

# DEBUG_FOP - fopプロセッサの出力を見る際に設定する
ifndef DEBUG_FOP
  FOP_FLAGS  := -q
  FOP_OUTPUT := | $(SED) -e '/not implemented/d'        \
                         -e '/relative-align/d'         \
                         -e '/xsl-footnote-separator/d'
endif

# CLASSPATH - fopのための適切なCLASSPATHを定義する
export CLASSPATH
CLASSPATH = $(patsubst %;,%,                                        \
              $(subst ; ,;,                                         \
                $(addprefix c:/usr/xslt-process-2.2/java/,          \
                  $(addsuffix .jar;,                                \
                    xalan                                           \
                    xercesImpl                                      \
                    batik                                           \
                    fop                                             \
                    jimi-1.0                                        \
```

```
                    avalon-framework-cvs-20020315))))

# %.pdf - foファイルからpdfを生成するためのパターンルール
%.pdf: %.fo
        $(kill-acroread)
        java -Xmx128M $(FOP) $(FOP_FLAGS) $< $@ $(FOP_OUTPUT)

# %.fo - xmlファイルからfoファイルを生成するためのパターンルール
PAPER_SIZE := letter
%.fo: %.xml
        XSLT_FLAGS="--stringparam paper.type $(PAPER_SIZE)" \
        $(XMLTO) $(XMLTO_FLAGS) fo $<

# fop_help - fopプロセッサのヘルプを表示する
.PHONY: fop_help
fop_help:
        -java org.apache.fop.apps.Fop -help
        -java org.apache.fop.apps.Fop -print help
```

ご覧のとおり、ここで使う2段階の処理が2つのパターンルールで表されています。.xmlから.foへのルールはxmltoを起動し、.foから.pdfへのルールでは最初に稼動しているすべてのAcrobat Readerを止めてから（なぜなら、このプログラムがPDFファイルをロックしているので、FOPがPDFファイルへの書き込みができないからです）、FOPを起動します。FOPは非常におしゃべりなプログラムで、何百行もの取りとめもなく出力される警告に飽き飽きしてしまいます。そこで必要なデータだけを抜き出すための簡単なsedフィルタを加えるとともに、デバッグのためにそのフィルタを無効にする機能をDEBUG_FOPとして加えました。最後にHTMLのときと同様に、利便性のために全作業を開始する2つのターゲットpdfとshow_pdfを加えてあります。

11.1.4 ソーステキストの検証

タブ、マクロ、インクルードファイル、編集者のコメントなどをDocBookが拒絶するため、ソーステキストに誤りがないことを確認する作業は簡単ではありません。そこで4つの確認用ターゲットを作成し、それぞれの形式に関しての正しさを検証しています。

```
validation_checks := $(OUTPUT_DIR)/chk_macros_tabs      \
                     $(OUTPUT_DIR)/chk_fixme            \
                     $(OUTPUT_DIR)/chk_duplicate_macros \
                     $(OUTPUT_DIR)/chk_orphaned_examples

.PHONY: validate-only
validate-only: $(OUTPUT_DIR)/validate
$(OUTPUT_DIR)/validate: $(validation_checks)
        $(TOUCH) $@
```

それぞれのターゲットはタイムスタンプファイルを生成し、それらはトップレベルのタイムスタンプファイルvalidateの必須項目となっています。

```
$(OUTPUT_DIR)/chk_macros_tabs: $(ALL_XML_OUT)
        # マクロとタブの検索...
        $(QUIET)! $(EGREP) --ignore-case          \
                           --line-number          \
                           --regexp='\b(m4_|mp_)' \
                           --regexp='\011'        \
                           $^
        $(TOUCH) $@
```

最初は前処理で展開されなかったm4マクロをチェックします。これはマクロ名を間違えたか、定義されていないものを使ったかのどちらかです。ここでは同時にタブも探します。どちらもここでは起こりえないはずの誤りですが、実際には起きます。このスクリプトで興味深いのは、$(QUIET)の後の感嘆符です。これはegrepの終了ステータスを反転させるものです。つまりegrepが指定されたパターンのどれかを見つけたときに、このコマンドが失敗したとmakeは判断します。

```
$(OUTPUT_DIR)/chk_fixme: $(ALL_XML_OUT)
        # RM:とFIXMEの検索...
        $(QUIET)$(AWK) \
                '/FIXME/ { printf "%s:%s: %s\n", FILENAME, NR, $$0 } \
                 /^ *RM:/ { \
                           if ( $$0 !~ /RM: Done/ ) \
                           printf "%s:%s: %s\n", FILENAME, NR, $$0 \
                         }' $(subst $(OUTPUT_DIR)/,$(SOURCE_DIR)/,$^)
        $(TOUCH) $@
```

これは解決されていない筆者へのメモをチェックします。FIXMEラベルがついているテキストは修正され、ラベルを取り除かなければなりません。加えてRM:のうち直後にDoneがついていないものも修正すべき目印となります。ここでのprintf関数に指定したフォーマットは、標準的なコンパイラのエラーフォーマットに従っているので、コンパイラのエラーを処理できる標準的なツールでこれらの警告を扱うことができます。

```
$(OUTPUT_DIR)/chk_duplicate_macros: $(SOURCE_DIR)/macros.m4
        # 重複したマクロを検索...
        $(QUIET)! $(EGREP) --only-matching \
              "\[^]+'," $< | \
        $(SORT) | \
        uniq -c | \
        $(AWK) '$$1 > 1 { printf "$>:0: %s\n", $$0 }' | \
        $(EGREP) "^"
        $(TOUCH) $@
```

これはm4のマクロファイルから重複したマクロ定義を探します。m4プロセッサはマクロの再定義をエラーにしないため、このチェックを加えました。パイプラインは次のように処理を進めます。各マクロ中で定義されているシンボルを取り出し、ソート後重複を数えます。数えた結果が1である行を取り除き、最後に終了ステータスを見るためにegrepを使います。繰り返しになりますが、終了ステータスを反転させることで、何かが見つかったときにmakeはエラーだと判断します。

```
ALL_EXAMPLES := $(TMP)/all_examples

$(OUTPUT_DIR)/chk_orphaned_examples: $(ALL_EXAMPLES) $(DEPENDENCY_FILES)
        $(QUIET)$(AWK) -F/ '/(EXAMPLES|OUTPUT)_DIR/ { print $$3 }' \
                $(filter %.d,$^) | \
        $(SORT) -u | \
        comm -13 - $(filter-out %.d,$^)
        $(TOUCH) $@

.INTERMEDIATE: $(ALL_EXAMPLES)
$(ALL_EXAMPLES):
        # 使われていない例題の検索...
        $(QUIET) ls -p $(EXAMPLES_DIR) | \
        $(AWK) '/CVS/ { next } \
                /\// { print substr($$0, 1, length - 1) }' > $@
```

最後は、本文から参照されていない例題のチェックです。このターゲットではちょっとした技を使っています。ここではすべての例題ディレクトリのリストとXMLの依存関係ファイルの2つを入力として使います。これら必須項目は`filter`と`filter-out`関数を用いて2つに分けられます。例題ディレクトリのリストは`ls -p`（`-p`オプションによりディレクトリ名の後にスラッシュが付加されます）を使い、スラッシュを探すことで作成します。パイプラインはまず必須項目からXMLの依存関係ファイルを取り上げ、その中にある例題ディレクトリを抜き出して重複があれば取り除きます。これが実際に本文から参照されている例題となります。このリストを`comm`コマンドの標準入力に入れ、実在する例題ディレクトリのリストを`comm`の2番目のファイルとして指定します。`-13`オプションにより、`comm`コマンドは2番目のカラムにあるものだけを出力します（つまり、これが依存関係ファイルには存在しないディレクトリになります）。

11.2 Linuxカーネルのmakefile

Linuxカーネルの`makefile`は複雑な構築システムにおける`make`の優れた使用例です。Linuxカーネルがどのように構成され構築されるのかを説明するのは本書の範囲を超えるものですが、カーネルの構築に対して使われている興味深い`make`の利用法をいくつか調べることにします。2.5、2.6カーネルの構築システムに関する詳細と、2.4の手法からの進化については、`http://archive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Germaschewski-OLS2003.pdf`に書かれています。

この`makefile`には非常に多くの面を持っていますが、ここでは他のアプリケーションでも活用できそうないくつかの機能を取り上げます。まず最初に1文字の`make`変数が、1文字のコマンドラインオプションを表している部分を見ます。どのようにソースとバイナリツリーが分離され、ユーザが`make`をソースツリーから起動できるようにしているのかも調べます。次に`makefile`がどのように表示に関する冗長性を制御しているのかを調査します。そして最も興味深いユーザ定義関数がどのようにコードの重複を防いでいるのか、可読性を向上させているのか、カプセル化を提供しているのかを見ます。

設定、構築、インストールという多くのフリーソフトウェアが使っているパターンをLinuxカーネルの構築でも採用しています。フリーでオープンな多くのソフトウェアパッケージは個別の設定スクリプト（通常autoconfで作成されます）を使いますが、Linuxカーネルのmakefileでは、makeを使いスクリプトや他のプログラムを間接的に実行する設定方式を提供しています。

設定段階が終わった後でmakeまたはmake allを実行すれば、カーネル、全モジュールを構築し、圧縮カーネルイメージを作成します（それぞれvmlinux、modules、bzImageターゲットに相当します）。カーネルの構築ごとにそれぞれ異なるバージョン番号が与えられ、カーネルにリンクされるversion.oに入ります。この番号（とversion.o）はmakefileにより更新されます。

コマンドラインオプションの扱い、コマンドラインで指定されたターゲットの分析、構築ステータス、構築ステータスの記憶、makeの行う表示の管理などは、カーネルの構築ではない普通のmakefileにも適用できる機能でしょう。

11.2.1 コマンドラインオプション

makefileの最初はコマンドラインから一般的な構築オプションを取り込むコードです。次のコードはverboseフラグを制御する部分の抜粋です†。

```
# To put more focus on warnings, be less verbose as default
# Use 'make V=1' to see the full commands
# 警告に注目するために、低い冗長度がデフォルト
# すべてのコマンドを見るには、'make V=1'を使う
ifdef V
  ifeq ("$(origin V)", "command line")
    KBUILD_VERBOSE = $(V)
  endif
endif
ifndef KBUILD_VERBOSE
  KBUILD_VERBOSE = 0
endif
```

入れ子になったifdefとifeqの組みは、Vがコマンドライン上で設定されているときのみKBUILD_VERBOSE変数を設定するために使っています。環境変数やmakefile内での設定は無効です。続くifndef条件判断でKBUILD_VERBOSEが設定されなかったときにverboseオプションを落とします。環境変数かmakefile内での設定でverboseオプションを有効にするには、Vではなく KBUILD_VERBOSEを設定する必要があります。

コマンドライン上で直接行うKBUILD_VERBOSEの設定は期待したとおりにうまく働きます。これはmakeを実行するシェルスクリプト（またはalias）を書く際に有用です。こういったスクリプトではGNUのロングオプションと同様に自明な名前を使うことができます。

その他のオプションとして、sparseチェックを行うもの（C）と、外部モジュールを構築するためのもの（M）があります。どちらもmakefile内で誤って設定してしまわないためにチェックを慎重に

† 本書の以降のコードではオリジナルとの比較を容易にするために、原文のコメントと日本語のコメントを併記します。

行っています。

makefileの次のセクションは出力ディレクトリのオプションを処理するためのものです。次に示すのは関連しているコードです。必要な構造を際立たせるために、ところどころ省略しています。

```
# kbuild supports saving output files in a separate directory.
# To locate output files in a separate directory two syntax'es are supported.
# In both cases the working directory must be the root of the kernel src.
# 1) O=
# Use "make O=dir/to/store/output/files/"
#
# 2) Set KBUILD_OUTPUT
# Set the environment variable KBUILD_OUTPUT to point to the directory
# where the output files shall be placed.
# export KBUILD_OUTPUT=dir/to/store/output/files/
# make
#
# The O= assigment takes precedence over the KBUILD_OUTPUT environment variable.
# KBUILD_SRC is set on invocation of make in OBJ directory
# KBUILD_SRC is not intended to be used by the regular user (for now)

# kbuildは異なるディレクトリへの出力ファイル作成をサポートしている
# 出力ファイルの作成先は2つの方法で指定可能
# どちらの場合も、出力先はカーネルソースディレクトリの下でなければならない
# 1) O=
# "make O=dir/to/store/output/files/"
#
# 2) KBUILD_OUTPUTを設定
# ファイルの出力先をKBUILD_OUTPUT環境変数に設定する
# export KBUILD_OUTPUT=dir/to/store/output/files/
# make
#
# O=形式の指定はKBUILD_OUTPUT環境変数に優先する
# KBUILD_SRCはmakeをOBJディレクトリから起動した際に設定される
# KBUILD_SRCは通常のユーザによる使用を（今のところ）想定していない

ifeq ($(KBUILD_SRC),)

  # OK, Make called in directory where kernel src resides
  # Do we want to locate output files in a separate directory?

  # どうやら、makeはカーネルソースの場所から実行されている
  # 出力ファイルを別の場所に置きたいのかな？
  ifdef O
    ifeq ("$(origin O)", "command line")
      KBUILD_OUTPUT := $(O)
    endif
  endif
  …
  ifneq ($(KBUILD_OUTPUT),)
    …
    .PHONY: $(MAKECMDGOALS)
```

```
    $(filter-out _all,$(MAKECMDGOALS)) _all:
            $(if $(KBUILD_VERBOSE:1=),@)$(MAKE) -C $(KBUILD_OUTPUT) \
            KBUILD_SRC=$(CURDIR) KBUILD_VERBOSE=$(KBUILD_VERBOSE) \
            KBUILD_CHECK=$(KBUILD_CHECK) KBUILD_EXTMOD="$(KBUILD_EXTMOD)" \
            -f $(CURDIR)/Makefile $@
    # Leave processing to above invocation of make

    # 上記makeの実行により処理を終了する
    skip-makefile := 1
  endif # ifneq ($(KBUILD_OUTPUT),)
endif # ifeq ($(KBUILD_SRC),)

# We process the rest of the Makefile if this is the final invocation of make

# もしこれが最終的なmakeの実行であるなら、残りを処理する
ifeq ($(skip-makefile),)
  …the rest of the makefile here…
endif # skip-makefile
```

基本的にここでは、KBUILD_OUTPUTが設定されていればKBUILD_OUTPUTで定義された出力ディレクトリでmakeを再帰的に実行します。KBUILD_SRCはmakeが起動された場所に設定され、その場所にあるmakefileが使われます。skip-makefileが設定されるので、makefileの残りの部分をmakeが見ることはありません。再帰実行されたmakeは、この同じmakefileを読み込み、今度はKBUILD_SRCが設定されているのでskip-makefileが設定されることはなく、makefileの残りの部分が読み込まれ実行されます。

これでコマンドラインオプションの処理は終わりです。makefileの大半はifeq ($(skip-makefile),)セクションの中にあります。

11.2.2 設定と構築

makefileには設定ターゲットと構築ターゲットがあります。設定ターゲットはmenuconfig、defconfigなどといった形式を持っています。

```
$ make oldconfig all
```

この場合makefileは自分自身を再帰的に実行し、各ターゲットをそれぞれ処理することにより、設定ターゲットを構築ターゲットとを別に扱います。

設定、構築のターゲットとそれらが混在した場合の処理は以下のコードから始まります。

```
# To make sure we do not include .config for any of the *config targets
# catch them early, and hand them over to scripts/kconfig/Makefile
# It is allowed to specify more targets when calling make, including
# mixing *config targets and build targets.
# For example 'make oldconfig all'.
# Detect when mixed targets is specified, and make a second invocation
# of make so .config is not included in this case either (for *config).
```

```
# *configターゲットに対して.configをインクルードしないために初期の段階でターゲットを捕捉し、
# scripts/kconfig/Makefileに渡す
# makeを実行する際に*configターゲットと構築ターゲットを混在させて複数のターゲットを指定することが
# できる
# 例えば 'make oldconfig all'
# ターゲットの混在が発見されたとき、その後makeが（*configに対して）再帰起動されたときは、どちらも
# .configはインクルードされない
no-dot-config-targets := clean mrproper distclean \
                         cscope TAGS tags help %docs check%

config-targets := 0
mixed-targets := 0
dot-config := 1
```

no-dot-config-targets変数は.configファイルを必要としないターゲットのリストです。このコードでは次にconfig-targets、mixed-targets、dot-configを初期化します。何らかの設定ターゲットがコマンドライン上にあることをconfig-targetの値1で示します。構築ターゲットがあることはdot-configの値1で、同様にmixed-targetsの値1は設定ターゲットと構築ターゲットが混在していることを表します。

以下はdot-configを設定するコードです。

```
ifneq ($(filter $(no-dot-config-targets), $(MAKECMDGOALS)),)
  ifeq ($(filter-out $(no-dot-config-targets), $(MAKECMDGOALS)),)
    dot-config := 0
  endif
endif
```

設定ターゲットがMAKECMDGOALSの中にあると、最初のfilter関数は値を返します。filter関数が値を返すと、ifneqは真になります。これは二重否定的なコードなので、理解するのが多少難しいかもしれません。ifeqの行はMAKECMDGOALSに入っているのが設定ターゲットだけだった場合に真になります。つまり設定ターゲットがMAKECMDGOALSに入っていて、MAKECMDGOALSの中身が設定ターゲットだけである場合にdot-configの値が0になります。次のようにもっとていねいな書き方をすると、この2つの条件判断の意味が明確になります。

```
config-target-list := clean mrproper distclean \
                         cscope TAGS tags help %docs check%

config-target-goal := $(filter $(config-target-list), $(MAKECMDGOALS))
build-target-goal := $(filter-out $(config-target-list), $(MAKECMDGOALS))

ifdef config-target-goal
  ifndef build-target-goal
    dot-config := 0
  endif
endif
```

空の変数は未定義であると扱われるため、ここではifneqの代わりにifdefを使っていますが、変

数に（変数が定義されているとして扱われてしまう）空白列が入っていないことを確認しなければなりません†。

config-targets変数とmixed-targets変数は次のコードで設定されます。

```
ifeq ($(KBUILD_EXTMOD),)
  ifneq ($(filter config %config,$(MAKECMDGOALS)),)
    config-targets := 1
    ifneq ($(filter-out config %config,$(MAKECMDGOALS)),)
      mixed-targets := 1
    endif
  endif
endif
```

KBUILD_EXTMODは外部モジュールを構築することになっている場合に値を持ちますが、通常は値を持ちません。最初のifneqはMAKECMDGOALSにconfigというサフィックスを持っているターゲットが入っている場合に真となります。2番目のifneqはconfigサフィックスを持つターゲット以外が入っている場合に真となります。

いったん変数が設定されれば、4つに分岐するif-elseの中で使われます。以下のコードは、その構造を明確にするために必要な部分が凝縮されインデントが行われています。

```
ifeq ($(mixed-targets),1)
  # We're called with mixed targets (*config and build targets).
  # Handle them one by one.

  # ターゲットが混在している (*configと構築ターゲット)
  # それらを1つずつ処理する
  %:: FORCE
        $(Q)$(MAKE) -C $(srctree) KBUILD_SRC= $@
else
  ifeq ($(config-targets),1)
    # *config targets only - make sure prerequisites are updated, and descend
    # in scripts/kconfig to make the *config target

    # *configターゲットのみ - 必須項目が最新かを確認し、*configターゲットを構築するために
    # scripts/kconfigに入る
    %config: scripts_basic FORCE
            $(Q)$(MAKE) $(build)=scripts/kconfig $@
  else
    # Build targets only - this includes vmlinux, arch specific targets, clean
    # targets and others. In general all targets except *config targets.

    # 構築ターゲットのみ - ここにはvmlinux, アーキテクチャ固有のターゲット,cleanおよびその
    # 他のターゲットが含まれる
    # つまり*config以外のターゲットである
```

† 訳注：no-dot-config-targetsは、設定ターゲットというよりも.configファイルを必要としないターゲットのリストです。後で出てきますが、設定ターゲットとして扱われるのは*configターゲットであり、その他のターゲットは構築ターゲットの一部として扱われます。

```
...
ifeq ($(dot-config),1)
  # In this section, we need .config
  # Read in dependencies to all Kconfig* files, make sure to run
  # oldconfig if changes are detected.

  # このセクションでは.configが必要となる
  # 全Kconfig*ファイルに対する依存関係を読み込む変更を検知したときはoldconfigの実行を忘れずに
  -include .config.cmd
  include .config

  # If .config needs to be updated, it will be done via the dependency
  # that autoconf has on .config.
  # To avoid any implicit rule to kick in, define an empty command

  # もし.configを更新する必要があるならautoconfが持つ依存関係を通して更新される必要あり
  # その他の暗黙ルールが適用されないよう、ダミーを定義する
  .config: ;

  # If .config is newer than include/linux/autoconf.h, someone tinkered
  # with it and forgot to run make oldconfig

  # .configがinclude/linux/autoconf.hよりも新しいということは、だれかが手を入れたときに
  # make oldconfigを実行し忘れている
  include/linux/autoconf.h: .config
          $(Q)$(MAKE) -f $(srctree)/Makefile silentoldconfig
else
  # Dummy target needed, because used as prerequisite

  # 必須項目として使われるため、ダミーターゲットが必要
  include/linux/autoconf.h: ;
endif

include $(srctree)/arch/$(ARCH)/Makefile
... lots more make code ...
endif #ifeq ($(config-targets),1)
endif #ifeq ($(mixed-targets),1)
```

最初の分岐では`ifeq ($(mixed-targets),1)`が混在したコマンドラインターゲットを処理します。この分岐では完全汎用パターンルールが唯一のターゲットになります。ターゲットを処理する特定のルールは存在しないので（それらは他の分岐に入っています）、各ターゲットはパターンルールを1回ずつ実行します。これが設定ターゲットと構築ターゲットが混在したコマンドラインを単純なものに分割する方法です。この完全汎用パターンルールのコマンドスクリプトは各ターゲットに対して`make`を再帰的に実行して、ターゲットの混在をなくした上で同じロジックが実行されるようにします。必須項目には`.PHONY`の代わりに`FORCE`が使われています。というのも次のようなパターンルールは`.PHONY`では定義できないからです。

```
%:: FORCE
```

というわけで、FORCEを一貫してあらゆるところで使うことは理にかなっているように思います†。

2番目のif-elseによる分岐では、コマンドライン上に設定ターゲットだけが指定されていることをifeq ($(config-targets),1)が判断します。この分岐での唯一のターゲットは、%configに対するパターンルールです（これ以外のターゲットは、すでに除外されています）。コマンドスクリプトは、scripts/kconfigサブディレクトリで再帰的に実行するmakeにターゲットを引き渡します。興味深いのはmakefileの最後で定義されている$(build)です。

```
# Shorthand for $(Q)$(MAKE) -f scripts/Makefile.build obj=dir
# Usage:

# $(Q)$(MAKE) -f scripts/Makefile.build obj=dir の省略形
# 使い方:
# $(Q)$(MAKE) $(build)=dir
build := -f $(if $(KBUILD_SRC),$(srctree)/)scripts/Makefile.build obj
```

KBUILD_SRCが設定されていると、特定のmakefileへのフルパスが-fオプションに渡されます。そうでないときは単純に相対パスが使われます。次に等号の左辺にobj変数が設定されます††。

3番目のif-elseによる分岐では、ifeq ($(dot-config),1)が、.configと.config.cmdという生成される2つのファイルを必要とする構築ターゲットを扱います。分岐の最後は単にautoconf.hのダミーターゲットです。これにより、実際にファイルが存在していなくても、他のターゲットの必須項目として使うことができます。

makefileの残りの大部分は、3番目と4番目の分岐のためのコードで、カーネルとモジュールを構築するためのものです。

11.2.3 コマンド表示の管理

カーネルのmakefileではコマンドごとの表示レベルを管理する画期的な方法を用いています。重要な作業にはそれぞれverbose（冗長）とquiet（寡黙）という表現形式がありますverboseではcmd_actionという変数にコマンド名が収められ、コマンドが普通に実行されます。quietでは簡単な作業の説明がquiet_cmd_action変数に収められます。

```
quiet_cmd_TAGS = MAKE $@
cmd_TAGS = $(all-sources) | etags -
```

コマンドはcmd関数を通して実行されます。

---

† 訳注：FORCEはmakefileの最後で定義されている、必須項目とコマンドスクリプトを持たない1種のダミーターゲットです。.PHONYターゲットは他のターゲットの必須項目になれないのでこのようなダミーターゲットを使っています。FORCEはコマンドスクリプトを持たないので、ルールが実行されることはなくFORCEというファイルも作られることがありません。ということからFORCEを必須項目に持つターゲットは必ず構築作業が行われることになります。

†† 訳注：build変数は、コメント中にもあるように$(build)=dirという形式で代入の左辺として使われます。build変数の値の最後はobjであるので、$(build)変数が展開されると結局、-f scripts/Makefile.build obj=dirのように（-fオプションと）objへの代入式ができます。

```
# If quiet is set, only print short version of command
# quietが設定されていた場合、短いほうのコマンドを表示する
cmd = @$(if $($(quiet)cmd_$(1)),\
          echo ' $($(quiet)cmd_$(1))' &&) $(cmd_$(1))
```

emacsのtagsを作るコードを実行するには、makefileに次のように記述します。

```
    TAGS:
            $(call cmd,TAGS)
```

cmd関数は@で始まるので、関数が表示するのはechoコマンドの出力だけです。通常のモードではquiet変数は空になっているのでifの中の$($(quiet)cmd_$(1))が展開されると$(cmd_TAGS)になります。この変数は空ではないので、関数全体としては次のように展開されることになります†。

```
echo ' $(all-sources) | etags -' && $(all-sources) | etags -
```

もしquietのほうがよければquiet変数にはquiet_という値が入り、関数は次のように展開されます。

```
echo ' MAKE $@' && $(all-sources) | etags -
```

変数にはsilent_という値が設定される場合があります††。silent_cmd_TAGSというコマンドは存在しないので、コマンドの表示は一切行われなくなります。

コマンドの表示は場合により、特にコマンドに引用符が含まれるときにはもっと複雑になります。そのような場合に備えてmakefileには次のコードがあります。

```
$(if $($(quiet)cmd_$(1)),echo ' $(subst ','\'',$($(quiet)cmd_$(1)))';)
```

このechoコマンドには表示が正しく行われるように、単引用符からエスケープ文字付き単引用符への置換が入っています。cmd_とquiet_cmdを作る際に問題のあるいくつかのコマンドに対しては、値として@か空である$(Q)が前置されます。

```
ifeq ($(KBUILD_VERBOSE),1)
  quiet =
  Q =
else
  quiet=quiet_
  Q = @
endif

# If the user is running make -s (silent mode), suppress echoing of commands
```

† 訳注：実際には$(all-sources)も展開されます。all-sources変数は単純なファイルのリストではなく、すべてのファイルを探し出しファイル名を出力するfindコマンドが入っています。

†† 訳注：この後で出てきますが、silent_が設定されるのはmakeに-sオプションが付加された場合です。makefileではMAKEFLAGSをチェックして-sが指定されているかを見ています。

```
# make -s（サイレントモード）、で実行されたときには、コマンドを表示しません

ifneq ($(findstring s,$(MAKEFLAGS)),)
  quiet=silent_
endif
```

11.2.4 ユーザ定義関数

カーネルのmakefileには多数の関数が定義されています。ここでは最も興味深いものを取り上げます。コードは読みやすくするために整形してあります。check_gcc関数はgccのコマンドラインオプションを選択するために用意されたものです。

```
# $(call check_gcc,preferred-option,alternate-option)
check_gcc =
    $(shell if $(CC) $(CFLAGS) $(1) -S -o /dev/null \
               -xc /dev/null > /dev/null 2>&1;      \
            then                                    \
              echo "$(1)";                          \
            else                                    \
              echo "$(2)";                          \
            fi  ;)
```

この関数は/dev/nullを入力として、望ましいコマンドラインオプション（preferred-option）付きでgccを実行します。出力ファイル、標準出力、標準エラー出力はすべて捨てられます。gccコマンドがエラーにならなければ、すなわち指定したコマンドラインオプションが現在のアーキテクチャに対して有効であることになるので、そのまま関数の戻り値とします。そうでなければ指定したオプションが無効であるため、代替のオプション（alternate-option）が返されます。使用例はarch/i386/Makefileの中にあります。

```
# prevent gcc from keeping the stack 16 byte aligned
# gccがスタックを16バイト境界に揃えるのを阻止する
CFLAGS += $(call check_gcc,-mpreferred-stack-boundary=2,)
```

if_changed_dep関数は注目に値するテクニックを使って依存関係情報を生成します。

```
# execute the command and also postprocess generated
# .d dependencies file
# コマンドを実行し、生成された依存関係ファイル.dの後処理も行う
if_changed_dep =                                        \
    $(if                                                \
      $(strip $?                                        \
        $(filter-out FORCE $(wildcard $^),$^)           \
        $(filter-out $(cmd_$(1)),$(cmd_$@))             \
        $(filter-out $(cmd_$@),$(cmd_$(1)))),           \
      @set -e;                                          \
      $(if $($(quiet)cmd_$(1)),                         \
        echo ' $(subst ','\'',$($(quiet)cmd_$(1)))';)   \
      $(cmd_$(1));                                      \
```

```
scripts/basic/fixdep                                    \
  $(depfile)                                            \
  $@                                                    \
  '$(subst $$,$$$$,$(subst ','\'',$(cmd_$(1))))'        \
  > $(@D)/.$(@F).tmp;                                   \
rm -f $(depfile);                                       \
mv -f $(@D)/.$(@F).tmp $(@D)/.$(@F).cmd)
```

この関数に含まれるのは、1つのifだけです。依存関係情報は普通、ファイルの更新タイムスタンプに注意を払っています。カーネルの構築システムではこの作業に新しい趣向を持ち込みました。カーネルの構築ではコンポーネントの構造と挙動を制御するために広範囲にわたるコンパイラオプションを利用しています。構築を行う間にコマンドラインオプションが正しく適用されることを保証するため、ある特定のターゲットに対するコンパイラオプションが変更されたなら、そのファイルが再コンパイルされるようにmakefileは作られています。これがどのように達成されているかを見てみましょう。

本質的に、カーネルの各ファイルに対するコンパイルコマンドは.cmdファイルの中に入っています。構築を再度実行したとき、makeは.cmdファイルを読み、現在のコンパイルコマンドと前回行ったときのコマンドとを比較します。もしそれらが異なっていた場合にはオブジェクトファイルが再作成されるよう、依存している.cmdファイルを生成しなおします。.cmdファイルにはたいてい2つの要素が入っています。ターゲットファイルに対する依存関係とコマンドラインオプションを記録している1つの変数です。例えばarch/i386/kernel/cpu/mtrr/if.cに対するファイルは、次のような内容になります（一部省略されています）。

```
cmd_arch/i386/kernel/cpu/mtrr/if.o := gcc -Wp,-MD …; if.c

deps_arch/i386/kernel/cpu/mtrr/if.o := \
  arch/i386/kernel/cpu/mtrr/if.c \
  …

arch/i386/kernel/cpu/mtrr/if.o: $(deps_arch/i386/kernel/cpu/mtrr/if.o)
$(deps_arch/i386/kernel/cpu/mtrr/if.o):
```

if_changed_dep関数に戻りましょう。ターゲットよりも新しい必須項目がもし存在するなら、これがstrip関数の最初の引数になります。strip関数の第2引数は、全必須項目からファイル以外と空のターゲットであるFORCEを取り除いたものです。難解なのは次の2つのfilter-out関数でしょう。

```
$(filter-out $(cmd_$(1)),$(cmd_$@))
$(filter-out $(cmd_$@),$(cmd_$(1)))
```

コマンドラインオプションが変更されると、これら関数呼び出しの一方または両方が値を持つようになります。$(cmd_$(1))マクロは現在のコマンドに、$(cmd_$@)は前回実行したコマンドを表します。先の例でcmd_arch/i386/kernel/cpu/mtrr/if.o変数に入っているようなものです。も

し新しいコマンドにオプションが追加されていれば、最初のfilter-out関数は何も返しませんが、2番目のfilter-out関数は新しいオプションを値として持つようになります。新しいコマンドではオプションが削られていたなら、最初のfilter-out関数は削除されたオプションを値として持ち、2番目のfilter-out関数は値を返さなくなります。興味深いことにfilter-out関数は（それぞれ独立したパターンとして扱われる）単語のリストを受け取りますが、オプションの順番が変化してもfilter-out関数は正確にオプションの追加と削除を認識します。とても気が利いています。

コマンドスクリプトの最初の行はシェルのオプションを設定して、エラーが起きた際にはすぐに終了するようにします。これにより複数行に渡るスクリプトでエラーが起きても、ファイルを破壊するという事故を防ぐことができます。簡単なスクリプトでは、セミコロンの代わりに&&を使って連結することにより同じことができます。

次の文は、「11.2.3 コマンド表示の管理」で説明した手法を使ったechoと、依存関係を生成するコマンドの実行です。このコマンドはscripts/basic/fixdepにより変換される$(depfile)を書き出します。fixdepコマンド行にある入れ子になったsubst関数では、内側のsubstで単引用符をエスケープ付き単引用符に変換し、外側のsubst関数で$$（シェルの文法で現在のプロセス番号を表します）をエスケープします。

そしてエラーが起きなければ中間ファイルである$(depfile)が削除され、生成された（.cmdサフィックスルの付いた）依存関係ファイルが所定の位置に置かれます。

if_changed_dep関数がコマンドの実行を制御するのに使った手法を、次のif_changed_rule関数でも使っています。

```
# Usage: $(call if_changed_rule,foo)
# will check if $(cmd_foo) changed, or any of the prequisites changed,
# and if so will execute $(rule_foo)

# 使い方: $(call if_changed_rule,foo)
# $(cmd_foo)が変更されているか、もしくは何らかの必須項目が変更されているかを調べ、もしそうなら
# $(rule_foo)を実行する
if_changed_rule =                                       \
    $(if $(strip $?                                     \
           $(filter-out $(cmd_$(1)),$(cmd_$(@F)))   \
           $(filter-out $(cmd_$(@F)),$(cmd_$(1)))), \
      @$(rule_$(1)))
```

makefileの最上位レベルで、カーネルとマクロを結び付けるのにこの関数を使っています。

```
# This is a bit tricky: If we need to relink vmlinux, we want
# the version number incremented, which means recompile init/version.o
# and relink init/init.o. However, we cannot do this during the
# normal descending-into-subdirs phase, since at that time
# we cannot yet know if we will need to relink vmlinux.
# So we descend into init/ inside the rule for vmlinux again.

# ここはちょっと複雑: もしvmlinuxを再リンクする必要があるなら、バージョン番号をインクリメント
# しなければならない
```

```
# これはすなわちinit/version.oを再コンパイルしてinit/init.oを再リンクすることを意味している
# しかしながら、vmlinuxを再リンクする必要があるのか知るすべがないので、通常のdescending-into-
# subdirs（サブディレクトリに内に降下している）フェーズではこれを行うことができない
# そこで、vmlinuxに対するルールのなかで、もう1回init/に降りてゆくことにする
…

quiet_cmd_vmlinux__ = LD $@
define cmd_vmlinux__
  $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux) \
    …
endef

# set -e により、エラーが起きたときには直ちにルールが終了する

define rule_vmlinux__
  +set -e;                                                \
  $(if $(filter .tmp_kallsyms%,$^),,                      \
    echo ' GEN .version';                                 \
    . $(srctree)/scripts/mkversion > .tmp_version;        \
    mv -f .tmp_version .version;                          \
    $(MAKE) $(build)=init;)                               \
  $(if $($(quiet)cmd_vmlinux__),                          \
    echo ' $($(quiet)cmd_vmlinux__)' &&)                  \
  $(cmd_vmlinux__);                                       \
  echo 'cmd_$@ := $(cmd_vmlinux__)' > $(@D)/.$(@F).cmd
endef

define rule_vmlinux
  $(rule_vmlinux__);           \
  $(NM) $@ |                   \
  grep -v '\(compiled\)\|…' | \
  sort > System.map
endef
```

リンクと最終生成物であるSystem.mapの構築を行うためのrule_vmlinuxを起動するためにif_changed_rule関数が使われます。makefile内のコメントにあるようにrule_vmlinux__関数はvmlinuxを再リンクする前にカーネルのバージョンファイルを再生成してinit.oを再リンクします。これはrule_vmlinux__の最初のifで制御されています。2番目のifではリンクコマンドである$(cmd_vmlinux__)の表示を制御しています。リンクコマンドを実行した後で、実行されたコマンドを次回の構築での比較に用いるため、.cmdファイルに記録します。

12章
makefileのデバッグ

makefileのデバッグは、多少魔術的なところがあります。特定のルールがどのように評価されているかとか、変数がどのように展開されるのかを調べるためのデバッガは残念ながら存在しません。そこで、たいていのデバッグ作業では適当なところで表示を行い、makefileの動作を調べます。GNU makeはこういった作業のために、各種組み込み関数やコマンドラインオプションを提供しています。

makefileデバッグの最も優れた方法の1つは、デバッグ用のフックを用意するとともに、どこかでつまづいた場合にどこで何が起きたかをたどれるように防衛的なコードを書くことです。ここでは筆者が有益だと考える基本的なデバッグテクニックと防衛的コードの書き方をいくつか紹介します。

12.1 makeのデバッグ機能

問題のあるmakefileのデバッグに対してwarning関数は非常に便利な機能です。warning関数は何にも展開されないので、makefileのトップレベル、ターゲットや必須項目リスト、コマンドスクリプトなどどこにでも置くことができます。つまり変数の値を調べたいと思ったところのどこででも値を表示することが可能となります。例えば、次のmakefileを実行します。

```
$(warning A top-level warning)

FOO := $(warning Right-hand side of a simple variable)bar
BAZ = $(warning Right-hand side of a recursive variable)boo

$(warning A target)target: $(warning In a prerequisite list)makefile $(BAZ)
        $(warning In a command script)
        ls
$(BAZ):
```

すると、次ような表示になります。

```
$ make
makefile:1: A top-level warning
```

```
makefile:2: Right-hand side of a simple variable
makefile:5: A target
makefile:5: In a prerequisite list
makefile:5: Right-hand side of a recursive variable
makefile:8: Right-hand side of a recursive variable
makefile:6: In a command script
ls
makefile
```

warning関数の評価はmakeが行う即時評価と遅延評価の手順に従います。BAZ変数への代入にはwarning関数が含まれていますが、必須項目リストのなかでBAZが参照されるまではメッセージが出力されません。

warning関数がどこにでも使えるので、これは基本的なデバッグのための道具だといえます。

12.1.1 コマンドラインオプション

--just-print（-n）、--print-data-base（-p）、--warn-undefined-variablesは、デバッグ用として便利なコマンドラインオプションです。

--just-print

新しく追加したターゲットに対して筆者が最初に行うテストは、--just-printオプション（-nオプション）を付けてmakeを実行することです。これによりmakeはmakefileを読み込みターゲットを更新するために実行するコマンドのすべてを表示します。しかし実行はしません。利便性のためにGNU makeは、通常は表示されないはずの@修飾子を付けたコマンドも表示します。

このオプションを使うとすべてのコマンドを実行しないことになっています。これはある意味では正しいのですが、注意も必要です。makeはコマンドスクリプトを実行しませんが、直ちに評価される状況で使われたshell関数を実行してしまいます。例を示しましょう。

```
REQUIRED_DIRS = ...
_MKDIRS := $(shell for d in $(REQUIRED_DIRS); \
          do \
            [[ -d $$d ]] || mkdir -p $$d; \
          done)

$(objects) : $(sources)
```

以前見たように、_MKDIRS単純変数の目的は必要不可欠なディレクトリを作成するためのものです。これが--just-printオプション付きで実行された場合、makefileが読み込まれた時点でいつものようにshell関数を実行してしまうでしょう。そして$(objects)の中にあるファイルリストを更新するための各コンパイルコマンドを（実行せずに）表示します。

--print-data-base

--print-data-baseオプション（-pオプション）は、よく利用するもう1つのオプションです。これはmakefileを読み込み、GNUのコピーライトを表示してからmakeが実行したコマンドを表示

します。その後内部データベースの内容を洗いざらい出力します。データは、変数、ディレクトリ、暗黙ルール、パターンに固有の変数、ファイル（明示的ルール）、そしてvpathによる検索パスです。

```
# GNU Make 3.80
# Copyright (C) 2002 Free Software Foundation, Inc.
# This is free software; see the source for copying conditions.
# There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
# PARTICULAR PURPOSE.
normal command execution occurs here
# Make data base, printed on Thu Apr 29 20:58:13 2004
# Variables
…
# Directories
…
# Implicit Rules
…
# Pattern-specific variable values
…
# Files
…
# VPATH Search Paths
```

それぞれのセクションをもう少し詳しく見てみましょう。

変数（Variables）のセクションでは、変数がリストされ説明のためのコメントが添えられます。

```
# automatic
<D = $(patsubst %/,%,$(dir $<))
# environment
EMACS_DIR = C:/usr/emacs-21.3.50.7
# default
CWEAVE = cweave
# makefile (from `../mp3_player/makefile', line 35)
CPPFLAGS = $(addprefix -I ,$(include_dirs))
# makefile (from `../ch07-separate-binaries/makefile', line 44)
RM := rm -f
# makefile (from `../mp3_player/makefile', line 14)
define make-library
  libraries += $1
  sources += $2

  $1: $(call source-to-object,$2)
        $(AR) $(ARFLAGS) $$@ $$^
endef
```

自動変数は表示されませんが、自動変数から派生した$(<D)のような変数は表示されます。コメントはorigin関数（「4.2.5　比較的重要なその他の関数」を参照してください）が返す変数の種類を示しています。変数がファイルの中で定義されていれば、そのファイル名と定義されている場所の行番号も加えられます。単純変数と再帰変数は代入演算子で区別されます。単純変数の値は右辺を評価した状態で示されます。

次のディレクトリ（Directories）と書かれたセクションは、makeの利用者というよりもmakeの開発者にとって有用です。ここではmakeが調べたディレクトリがリストされます。たいていは存在しないSCCSやRCSのサブディレクトリも含まれます。個々のディレクトリに対して、デバイス番号、inode、合致したファイルパターンの数などの詳細情報も表示されます。

その後は、暗黙ルール（Implicit Rules）セクションが続きます。ここではmakeのデータベースに入っているすべての組み込みパターンルールとユーザ定義パターンルールが表示されます。繰り返しになりますが、ファイル内で定義されたルールはコメントにてそのファイル名と定義場所の行番号が示されます。

```
%.c %.h: %.y
# commands to execute (from `../mp3_player/makefile', line 73):
$(YACC.y) --defines $<
$(MV) y.tab.c $*.c
$(MV) y.tab.h $*.h
%: %.c
# commands to execute (built-in):
$(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@
%.o: %.c
# commands to execute (built-in):
$(COMPILE.c) $(OUTPUT_OPTION) $<
```

makeの組み込みルールの多様性や構造を熟知するためには、このセクションを精読するのがいちばんです。もちろんすべての暗黙ルールがパターンルールとして定義されているわけではありません。探しているルールが見つからなければ、古いサフィックスルール形式でリストされているファイル（Files）セクションを調べてみましょう。

次のセクションにはmakefileで定義されている、パターンに固有の変数が集められます。パターン固有の変数とは、関連しているパターンルールが実行される間だけ有効になる変数だったことを思い出してください。例えば次のように定義されているYYLEXFLAG変数を考えます。

```
%.c %.h: YYLEXFLAG := -d
%.c %.h: %.y
        $(YACC.y) --defines $<
        $(MV) y.tab.c $*.c
        $(MV) y.tab.h $*.h
```

これは次のように表示されます。

```
# Pattern-specific variable values
%.c :
# makefile (from `Makefile', line 1)
# YYLEXFLAG := -d
# variable set hash-table stats:
# Load=1/16=6%, Rehash=0, Collisions=0/1=0%
%.h :
# makefile (from `Makefile', line 1)
# YYLEXFLAG := -d
```

```
# variable set hash-table stats:
# Load=1/16=6%, Rehash=0, Collisions=0/1=0%

# 2 pattern-specific variable values
```

ファイル（Files）セクションでは特定のファイルに関連づけられるすべての明示的ルールとサフィックスルールがリストされます。

```
# Not a target:
.p.o:
# Implicit rule search has not been done.
# Modification time never checked.
# File has not been updated.
# commands to execute (built-in):
$(COMPILE.p) $(OUTPUT_OPTION) $<

lib/ui/libui.a: lib/ui/ui.o
# Implicit rule search has not been done.
# Last modified 2004-04-01 22:04:09.515625
# File has been updated.
# Successfully updated.
# commands to execute (from `../mp3_player/lib/ui/module.mk', line 3):
ar rv $@ $^

lib/codec/codec.o: ../mp3_player/lib/codec/codec.c ../mp3_player/lib/codec/codec.c ..
/mp3_player/include/codec/codec.h
# Implicit rule search has been done.
# Implicit/static pattern stem: `lib/codec/codec'
# Last modified 2004-04-01 22:04:08.40625
# File has been updated.
# Successfully updated.
# commands to execute (built-in):
$(COMPILE.c) $(OUTPUT_OPTION) $<
```

中間ファイルやサフィックスルールには“Not a target”（ターゲットではない）と表示され、それ以外はターゲットを表します。それぞれのファイルには、makeがどのようにそのルールを実行するかを示すコマンドが添えられます。また、通常のvpath検索で見つけられるファイルには、そのパス名もあわせて表示されます。

VPATH Search Pathsと書かれた最後のセクションには、VPATHの値とすべてのvpathパターンがリストされます。

複雑な変数やルールを作るためにユーザ定義関数やevalを広範囲に利用しているmakefileに対しては、多くの場合でこの出力を調べることがマクロ展開が期待した値を作っているかを確認する唯一の方法となります。

--warn-undefined-variables

このオプションを使うと、定義されていない変数を展開する際に警告が表示されるようになります。定義されていない変数は空の文字列になるので、タイプミスによる変数名の誤りが長い間発見されな

いまま残ってしまうことがよくあります。このオプションの問題点は、それは同時に筆者が余りこれを使わない理由でもありますが、利用者が使用するフックとして未定義の変数を使っている組み込みルールが非常に多く存在するからです。つまりこのオプションを付けてmakeを実行すると、エラーを示しているわけでもなくmakefileとも直接関係ない警告が必然的に大量に出力されてしまいます。例を見てみましょう。

```
$ make --warn-undefined-variables -n
makefile:35: warning: undefined variable MAKECMDGOALS
makefile:45: warning: undefined variable CFLAGS
makefile:45: warning: undefined variable TARGET_ARCH
...
makefile:35: warning: undefined variable MAKECMDGOALS
make: warning: undefined variable CFLAGS
make: warning: undefined variable TARGET_ARCH
make: warning: undefined variable CFLAGS
make: warning: undefined variable TARGET_ARCH
...
make: warning: undefined variable LDFLAGS
make: warning: undefined variable TARGET_ARCH
make: warning: undefined variable LOADLIBES
make: warning: undefined variable LDLIBS
```

そのような問題があるにしても、ここで述べたようなエラーを見つけるにはこのオプションが非常に有益です。

12.1.2 --debugオプション

makeが依存関係グラフをどのように解析したかを知りたければ、--debugオプションを使います。デバッガを使うことを除けば、これが最も詳細な情報を入手する手段です。--debugオプションには5つのデバッグオプションと1つの修飾子があります。それぞれbasic、verbose、implicit、jobs、all、makefileです。

--debugが指定されたときには、basicが使われます。-dが指定されたときにはallが使われます。他の組み合わせを指定するにはカンマで区切られた--debug=*option1,option2*という形式を使用します。*option*には次の単語を使うことができます（makeは実際には最初の1文字しか見ていません）。

basic

basicはデバッグの詳細度が最も低いオプションです。このオプションが指定されるとmakeは最新ではないと判断したターゲットと、その更新作業の状態を表示します。例を示しましょう。

```
File all does not exist.
  File app/player/play_mp3 does not exist.
    File app/player/play_mp3.o does not exist.
   Must remake target app/player/play_mp3.o.
gcc ... ../mp3_player/app/player/play_mp3.c
   Successfully remade target file app/player/play_mp3.o.
```

verbose

このオプションはbasicに加えて、どのファイルが解析されているのか、あるいは再構築する必要のない必須項目などの情報が表示されます。

```
File all does not exist.
 Considering target file app/player/play_mp3.
  File app/player/play_mp3 does not exist.
   Considering target file app/player/play_mp3.o.
    File app/player/play_mp3.o does not exist.
     Pruning file ../mp3_player/app/player/play_mp3.c.
     Pruning file ../mp3_player/app/player/play_mp3.c.
     Pruning file ../mp3_player/include/player/play_mp3.h.
    Finished prerequisites of target file app/player/play_mp3.o.
   Must remake target app/player/play_mp3.o.
gcc ... ../mp3_player/app/player/play_mp3.c
   Successfully remade target file app/player/play_mp3.o.
   Pruning file app/player/play_mp3.o.
```

implicit

このオプションはbasicに加えて、各ターゲットに対する暗黙ルールの検索についても表示されます。

```
File all does not exist.
  File app/player/play_mp3 does not exist.
  Looking for an implicit rule for app/player/play_mp3.
  Trying pattern rule with stem play_mp3.
  Trying implicit prerequisite app/player/play_mp3.o.
  Found an implicit rule for app/player/play_mp3.
    File app/player/play_mp3.o does not exist.
    Looking for an implicit rule for app/player/play_mp3.o.
    Trying pattern rule with stem play_mp3.
    Trying implicit prerequisite app/player/play_mp3.c.
    Found prerequisite app/player/play_mp3.c as VPATH
../mp3_player/app/player/
play_mp3.c
    Found an implicit rule for app/player/play_mp3.o.
   Must remake target app/player/play_mp3.o.
gcc ... ../mp3_player/app/player/play_mp3.c
   Successfully remade target file app/player/play_mp3.o.
```

jobs

このオプションではmakeが実行したサブプロセスの詳細が表示されます。このオプションではbasicオプションは有効になりません。

```
Got a SIGCHLD; 1 unreaped children.
gcc ... ../mp3_player/app/player/play_mp3.c
Putting child 0x10033800 (app/player/play_mp3.o) PID 576 on the chain.
Live child 0x10033800 (app/player/play_mp3.o) PID 576
Got a SIGCHLD; 1 unreaped children.
```

```
Reaping winning child 0x10033800 PID 576
Removing child 0x10033800 PID 576 from chain.
```

all
: このオプションは、以上のオプションをすべて有効にします。-dを指定した場合の既定値となっています。

makefile
: 通常デバッグ情報はmakefileが再構成されるまで出力されません。依存関係ファイルのようなインクルードされるファイルについても同様です。この修飾子を使うことで、makefileやインクルードファイルを再構成する間でも一部の情報は出力されます。このオプションはbasicオプションも有効にします。またallオプションが使われた場合には自動的に有効になります。

12.2 デバッグ用のコードを書く

12.2.1 優れたコーディングの習慣

筆者の経験上、多くのプログラマはmakefileの記述がプログラミングであるとは考えていませんし、C++やJavaのコードを書くときほどの注意も払いません。しかし、make言語は完全な非手続き型言語です。もしも構築システムの信頼性と保守性が重要なポイントであるなら、makefileを注意深く記述し可能なかぎり優れた方法を踏襲しなければなりません。

頑強なmakefileをプログラムする際の最も重要な点は、コマンドの終了ステータスのチェックを行うことです。もちろんmakeは単純なコマンドに対して自動的にチェックを行ないますが、makefileには黙ってエラーになってしまうような複合コマンドがしばしば含まれます。

```
do:
      cd i-dont-exist; \
      echo *.c
```

これを実行すると、エラーが起きてもエラーステータスによるmakefileの終了はありません。

```
$ make
cd i-dont-exist; \
echo *.c
/bin/sh: line 1: cd: i-dont-exist: No such file or directory
*.c
```

さらにいうと、.cファイルが1つもなければファイル名の展開は行われず、展開パターンがそのまま返されます。このコマンドスクリプトを記述する、もう少しましな方法はエラーを防いだりチェックするためのシェルの機能を使うことです。

```
SHELL = /bin/bash
do:
        cd i-dont-exist && \
        shopt -s nullglob &&
        echo *.c
```

これでcdのエラーは正常にmakeに伝えられ、echoコマンドは実行されず、makeはエラーステータスにより終了します。加えてbashのnullglobオプションにより、適応するファイルが存在しなかった場合には空の文字列が返るようになります（もちろん、特定のアプリケーションによっては、展開パターンそのままのほうがよい場合もあるでしょう）。

```
$ make
cd i-dont-exist && \
echo *.c
/bin/sh: line 1: cd: i-dont-exist: No such file or directory
make: *** [do] Error 1
```

次のポイントは、可能なかぎり読みやすくコードを整形するというものです。筆者がこれまでに見た多くのmakefileは、きれいに書かれておらず、そのため読みにくいものでした。次のコードのどちらが読みやすいでしょうか？

```
_MKDIRS := $(shell for d in $(REQUIRED_DIRS); do [[ -d $$d \
]] || mkdir -p $$d; done)
```

と

```
_MKDIRS := $(shell              \
for d in $(REQUIRED_DIRS);      \
do                              \
  [[ -d $$d ]] || mkdir -p $$d; \
done)
```

まともな人なら、最初の例では何が書かれているのか把握しづらく、セミコロンも見つけられず、いくつの文からできているのかも数えるのが難しいでしょう。これらは、つまらない問題ではありません。コマンドスクリプトの文法エラー中、非常に多くの割合を占めるのがセミコロン、バックスラッシュ、そしてパイプや論理演算子を表すその他の区切り文字の欠如です。

区切り文字の欠如すべてがエラーになるわけではない点に注意が必要です。例えば次の例はどちらもシェルの文法エラーにはなりません。

```
TAGS:
        cd src \
        ctags --recurse

disk_free:
        echo "Checking free disk space..." \
        df . | awk '{ print $$4 }'
```

コマンドを読みやすく整形することは、これらのエラーを見つけやすくします。ユーザ定義関数を整形する際には、字下げを使いましょう。時折、マクロ展開により混入する余分な空白がエラーの原因となります。そのような場合、strip関数の中に入れてしまいましょう。値の長いリストを整形する際には、それぞれの値を行に分割しましょう。各ターゲットの前にはコメントを置き、簡単な説明を加えパラメータのリストを書き出しておきましょう。

次のポイントは、よく使う値は変数にするというものです。普通のプログラムのように、あたりかまわず値を書き下してしまうと、コードは重複し、保守上の問題とバグを作り出してしまいます。変数を使うもう1つのすばらしい利点は、makeの実行中デバッグのために値を表示させることができるという点です。「12.2.3 デバッグテクニック」において、ちょっとしたコマンドライン上の技を紹介します。

12.2.2 防衛的コード

防衛的コードとは、あらかじめ想定していたことや何らかの期待が外れたときのために用意しておくコードで、決して真にならないifテストや、失敗しないassert関数、そしてトレースコードなどが相当します。もちろんこのような決して実行されないはずのコードの価値は、ときどき（通常は思いもしなかったときに）は実行されて警告やエラーを出力したり、トレースを有効にしてmakeの内部動作を確認することができるという点にあります。

このようなコードはすでに他の場所で目にしているはずですが、ここでまた見ることにしましょう。

妥当性の確認は防衛的コードのよい実例です。次のコードは実行中のmakeのバージョンが3.80であることを確認します。

```
NEED_VERSION := 3.80
$(if $(filter $(NEED_VERSION),$(MAKE_VERSION)),, \
  $(error You must be running make version $(NEED_VERSION).))
```

Javaアプリケーションに対しては、CLASSPATH中のファイルが存在することを確認するコードを含めるのが有益です。

妥当性の確認コードには単に何かが真であることを確認するというものも含まれます。先に見たディレクトリを作成するコードもこういった性質のものです。

防衛的コードのもう1つのテクニックは、「4.2.4 実行制御」で紹介したassert関数です。これにはいくつか異なる種類のものがありました。

```
# $(call assert,condition,message)
define assert
  $(if $1,,$(error Assertion failed: $2))
endef

# $(call assert-file-exists,wildcard-pattern)
define assert-file-exists
  $(call assert,$(wildcard $1),$1 does not exist)
endef

# $(call assert-not-null,make-variable)
define assert-not-null
  $(call assert,$($1),The variable "$1" is null)
endef
```

assertをmakefile中にちりばめることは、仮定が成り立っていないことを検知するだけでなく、

変数名の間違いなども発見することができる簡単で効果的な手法です。

4章では、ユーザ定義関数の展開を追跡するひと組の関数を紹介しました。

```
# $(debug-enter)
debug-enter = $(if $(debug_trace),\
                $(warning Entering $0($(echo-args))))

# $(debug-leave)
debug-leave = $(if $(debug_trace),$(warning Leaving $0))

comma := ,
echo-args = $(subst ' ','$(comma) ',\
              $(foreach a,1 2 3 4 5 6 7 8 9,'$($a)'))
```

自分で作った関数にこのマクロ呼び出しを配置して、デバッグで必要になるまでそのまま無効にしておくことができます。有効にするにはdebug_traceに何か値をセットします。

```
$ make debug_trace=1
```

4章でも述べたように、このマクロは完全ではありませんが、それでもバグを追跡するには有益です。

防衛的コードの最後の手法はコマンド修飾子@を直接書くのではなく、変数を通して設定し無効化できるようにするものです。

```
QUIET := @
…
target:
    $(QUIET) some command
```

この手法を使うと、通常は表示されないコマンドを、QUIET変数を再設定することで表示することができるようになります。

```
$ make QUIET=
```

12.2.3 デバッグテクニック

ここでは一般的なデバッグテクニックとその課題を扱います。結局のところデバッグでは、それぞれの状況に合っているなら何をやってもかまいません。ここで紹介する方法は筆者の状況には適しており、非常に簡単なmakefileの問題に対しても活用してきました。おそらく皆さんにとっても役に立つと思います。

make変数の値を調べるためのよく使われる最も簡単な方法は、ターゲットの構築中にその値を表示してみることです。warning関数を使って表示するための文を加えるのは簡単ですが、変数の値を表示するための汎用debugターゲットを作っておくと、長い眼で見たときには時間の節約になります。debugターゲットは次のようなものです。

```
debug:
        $(for v,$(V), \
          $(warning $v = $($v)))
```

これを使うには、表示したい変数のリストをコマンドラインで設定し、ターゲットとしてdebugを加えます。

```
$ make V="USERNAME SHELL" debug
makefile:2: USERNAME = Owner
makefile:2: SHELL = /bin/sh.exe
make: debug is up to date.
```

もっと巧妙なのが好みなら、MAKECMDGOALS変数を使うことでV変数の設定を省略することができます。

```
debug:
        $(for v,$(V) $(MAKECMDGOALS), \
          $(if $(filter debug,$v),,$(warning $v = $($v))))
```

これで変数をコマンドライン上に並べるだけで値を表示することができます。しかし同時に（ターゲットとしても扱われるので）それらの変数を更新する方法が不明であるという紛らわしいmakeからの警告も示されるため、この手法はあまりお勧めできません。

```
$ make debug USERNAME SHELL
makefile:2: USERNAME = Owner
makefile:2: SHELL = /bin/sh.exe
make: debug is up to date.
make: *** No rule to make target USERNAME. Stop.
```

10章でmakeの裏側で何が起こっているのかを理解するためにデバッグ用シェルを使う方法を示しました。makeはコマンドスクリプトに書かれているコマンドを実行する前に表示しますが、shell関数から実行されるコマンドは表示しません。通常こういったコマンドは微妙かつ複雑なものです。再帰変数に代入されていたりすると、即時に実行されることもあれば、評価されるまで実行が遅延させられることもあり、状況はさらに複雑になります。こういったコマンドを調べる1つの方法が、サブシェルに対してデバッグ表示を行うように指示することです。

```
DATE := $(shell date +%F)
OUTPUT_DIR = out-$(DATE)

make-directories := $(shell [ -d $(OUTPUT_DIR) ] || mkdir -p $(OUTPUT_DIR))

all: ;
```

シェルのデバッグオプションをつけて実行すると、次のようになります。

```
$ make SHELL="sh -x"
+ date +%F
```

```
+ '[' -d out-2004-05-11 ']'
+ mkdir -p out-2004-05-11
```

シェルにこのオプションを使うと展開された変数の値と実行される式も表示されるので、warning関数を使ったデバッグ情報以上のものを得ることができます。

本書の例の多くは、次のWindowsのCygwinシステムにおけるPATH変数を調べる関数のように深い入れ子構造を持っています。

```
$(if $(findstring /bin/,                                    \
       $(firstword                                          \
         $(wildcard                                         \
           $(addsuffix /sort$(if $(COMSPEC),.exe),          \
             $(subst :, ,$(PATH))))))),,                    \
  $(error Your PATH is wrong, c:/usr/cygwin/bin should      \
    precede c:/WINDOWS/system32))
```

こういった関数をデバッグする優れた方法はありません。適当な方法の1つは、バラバラに分解し、それぞれの値を表示することです。

```
$(warning $(subst :, ,$(PATH)))
$(warning /sort$(if $(COMSPEC),.exe))
$(warning $(addsuffix /sort$(if $(COMSPEC),.exe),   \
            $(subst :, ,$(PATH))))
$(warning $(wildcard \
            $(addsuffix /sort$(if $(COMSPEC),.exe), \
              $(subst :, ,$(PATH)))))
```

少々手間のかかる方法ではありますが、デバッガのない環境ではこれがいろいろな式の値を調べる最もよい（そして場合によっては唯一の）方法です。

12.3 よくあるエラーメッセージ

GNU make 3.81のマニュアルにはmakeのエラーメッセージとその理由をまとめたすばらしいセクションがあります。ここではよくあるエラーのいくつかを見てみます。ここで取り上げる問題の中にはコマンドスクリプトの文法エラーのように、厳密にいえばmakeのエラーではないものもありますが、それでも開発者にとってはおなじみのものです。すべてのmakeのエラーを知りたければ、makeのマニュアルを参照してください。

makeのエラーメッセージは次のような形式で標準化されています。

makefile:*n*: *** *message*. Stop.

または

make:*n*: *** *message*. Stop.

makefileの部分は、エラーの発生したmakefileのファイル名かインクルードファイルのファイル名になります。続いてエラーのある行番号、そして3つのアスタリスクの後にエラーメッセージが続きます。

他のプログラムを起動するのはmakeの仕事ですが、エラーが発生した場合に、起動したプログラムのエラーがmakefile内にある問題を明るみに出すことは非常によくあることです。例えば、コマンドスクリプトの形式が誤っているか、コンパイラへのコマンドライン引数が誤っているといったこともシェルのエラーの原因となります。どのプログラムがエラーを出しているのかを把握することが、問題を解決するための開発者としての最初の作業となります。幸いにもmakeのエラーメッセージはきわめて自明なものばかりです。

12.3.1 文法エラー

括弧のつけ忘れ、タブではなく空白を使ってしまったなどの、字句的な誤りは日常茶飯事です。make初心者の最も犯しやすい誤りは、変数名を括弧で囲まないことです。

```
foo:
        for f in $SOURCES; \
        do                 \
          ...              \
        done
```

結局$Sだけが空の文字列に展開され、シェルはfにOURCEという値を入れてforループを1回だけ回します。変数fをどのように使うかに依存しますが、場合により親切にも次のようなシェルのエラーメッセージが表示されることになります。

```
OURCES: No such file or directory
```

しかしメッセージが何も表示されない可能性もあります。makeの変数を括弧で囲むことを忘れてはいけません。

missing separator （区切り文字を欠いている）

次の2つメッセージは、通常コマンドスクリプトを書くのにタブではなく空白を使ったことを示しています。

```
makefile:2:missing separator. Stop.

makefile:2:missing separator (did you mean TAB instead of 8 spaces?). Stop.
```

もう少し正確に解釈すると、makeは:、=、タブといった区切り文字を探したけれども見つけることができず、代わりに理解不能なものに突き当たってしまったということになります。

commands commence before first target（最初のターゲットの前にコマンドが書かれている）

これもタブの問題です。このエラーは「5.1 コマンドの構文解析」で最初に出てきました。このエラーはmakefileの中ほどで、コマンドではない行をタブで始めてしまった場合に見られるようです。makeは可能なかぎり何が起きているのかを把握しようとしますが、変数の代入でもなく、条件判断でもなく、マクロ定義でもなければ、makeは誤ってコマンドが配置されていると解釈します。

unterminated variable reference（変数参照の終端がない）

これは単純ですが、よくあるエラーです。変数参照か関数の呼び出しが適切な数の閉じ括弧で終わっていないことを示しています。深く入れ子になった関数呼び出しや変数参照があるとmakefileはだんだんとLispのように見えてきます。Emacsのような括弧の対応を示してくれるエディタを使うことが、このエラーを防ぐ確実な方法です。

12.3.2 コマンドスクリプト中のエラー

コマンドスクリプトの代表的なエラーは次の3つです。複数行コマンドにセミコロンを入れ忘れている、不完全もしくは不正確なパス変数、そして単純な実行時のエラーです。セミコロンの問題については「12.2.1 優れたコーディングの習慣」ですでに扱ったので、ここでは詳しく取り上げません。

シェルがfooコマンドを見つけることができなかった場合は、以下の昔ながらのメッセージが表示されます。

```
bash: foo: command not found
```

つまりシェルはPATH変数に指定されている各ディレクトリから実行ファイルを探そうとしたけれども見つからなかったということになります。このエラーを修正するには、通常.profile（Bourneシェルの場合）または.bashrc（bashの場合）または.cshrc（Cシェルの場合）に書かれているPATH変数の定義を書き換えなければなりません。もちろんPATH変数はmakefileの中で定義してmakeを使ってPATH変数をexportすることもできます。

コマンドがエラーになると、0以外の終了ステータスを返して終了します。この場合makeはメッセージを表示してエラーが起きたことを示します。

```
$ make
touch /foo/bar
touch: creating /foo/bar: No such file or directory
make: *** [all] Error 1
```

この例では、エラーを起こしたtouchコマンドがまずその状況を示すエラーメッセージを表示しています。その次の行はmakeによるエラーの要約です。構築に失敗したターゲット名が角括弧内に表示され、続いてエラーとなったプログラムの終了ステータスが示されます。コマンドが単純に終了ステータスを返すのではなくシグナルで終了してしまった場合には、もう少し詳しいメッセージが表示されることもあります。

@修飾子が付けられ表示されないコマンドでもエラーは起きます。その場合、エラーメッセージは、あたかも何もないところから出力されたように見えます。

どちらの場合もエラーはmake自身ではなくmakeが起動したコマンドで起きたものです。

12.3.3 No Rule to Make Target（ターゲットを構築するためのルールがない）

このメッセージには以下の2つの形式があります。

```
make: *** No rule to make target XXX. Stop.

make: *** No rule to make target XXX, needed by YYY. Stop.
```

これは、makeがXXXファイルを更新する必要があると判断したけれども、そのためのルールを見つけられなかったことを示しています。makeはこのメッセージを表示して作業を断念する前に、内部データベースにあるすべての暗黙ルールと明示的ルールすべてを検索します。

このエラーには3つの原因が考えられます。

- そのファイルを更新するためのルールがmakefileに存在しません。この場合、そのターゲットを更新する方法を示すルールを追加しなければなりません。
- makefileにタイプミスがあります。makeは誤ったファイルを探すか、誤ったファイルを更新するためのルールを探します。変数を使用することによりタイプミスを見つけることが困難になります。場合により、複雑なファイル名を示す値が正しいのかを確かめる唯一の方法は、変数を表示させるか、makeの内部データベースを調べるしかありません。
- ファイルは存在するけれどもどこにあるかわからない、もしくはどこを探せばよいのかmakeが認識していません。もちろんmakeがまったく正しい場合もあるのです。CVSからチェックアウトし忘れていると、ファイルはおそらく見つかりません。もっとありがちなのが、ソースをどこか別のところに置いてしまったので、単にmakeが必要なファイルを見つけられないだけというものです。ソースは別のソースツリーに置いてあったり別のプログラムによりバイナリツリーの中に生成されているなどが考えられます。

12.3.4 Overriding Commands for Target（コマンドが上書きされている）

makeは1つのターゲットに対して1つのコマンドスクリプトだけを許容します（まれに使われることのある二重コロンルールは例外です）。もし1つのターゲットに複数のコマンドスクリプトが指定されているとmakeは警告（「ターゲットfooに対するコマンドが上書きされます」の意）を表示します。

```
makefile:5: warning: overriding commands for target foo
```

次の警告（「ターゲットfooに対する古いコマンドは無視されます」の意）も出力されるでしょう。

```
makefile:2: warning: ignoring old commands for target foo
```

最初の警告は2番目のコマンドが見つかった個所を示し、2番目の警告は上書きされてしまう最初のコマンドがある個所を示しています。

複雑なmakefileでは同じターゲットが複数回登場し、それぞれ必須項目を加えていきます。通常このうち1つだけにコマンドスクリプトが付随しますが、開発やデバッグを進めるうちに既存のコマンドを上書きすることなど意識せずに新たな別のコマンドを加えてしまうことはよくあります。

例えば、インクルードファイルに汎用のターゲットを定義して、それとは別にmakefileで個々に必須項目を加えたとします。

```
# jarファイルの作成
$(jar_file):
    $(JAR) $(JARFLAGS) -f $@ $^
```

makefileには次のように書くことになるでしょう。

```
# jarファイルのターゲットと、その必須項目を設定
jar_file = parser.jar
$(jar_file): $(class_files)
```

うっかりしてこのmakefileにコマンドスクリプトを書いてしまうと、makeはコマンドの上書きを警告するメッセージを表示します。

付録A
makeの実行

GNU makeは驚くほどのコマンドラインオプションを備えています。ほとんどのコマンドラインオプションには短い形式と長い形式があり、短い形式では1つのマイナス記号に続いて1文字、長い形式では2つのマイナス記号の後にマイナス記号で区切られた機能の完全な名前が続きます。これらオプションの形式は次のようなものです。

```
-o argument
--option-word=argument
```

以下にあげるのは、makeで最もよく使われるオプションです。すべてを知りたい場合は、GNU makeのマニュアルか、make --helpを実行してください。

--always-make
-B
: すべてのターゲットが最新ではないと仮定し、すべて更新します。

--directory=*directory*
-C *directory*
: 指定されたディレクトリに移動してから、makefileを読み込んだりその他の作業を行います。これにより指定された*directory*がCURDIR変数に設定されます。

--environment-overrides
-e
: 環境変数とmakefile内で定義された変数の両方が競合した場合、環境変数のほうを選択します。ただしmakefile内でoverride命令を付加した変数定義は、このオプションに優先して使われます。

--file=*makefile*
-f *makefile*
　指定されたファイルを規定のファイル（例えばmakefile、Makefile、GNUMakefile）に代わって読み込みます。

--help
-h
　コマンドラインオプションに関する概要を表示します。

--include-dir=*directory*
-I *directory*
　インクルードファイルが見つからないときに、makeのコンパイル時に指定された場所を探す前に見に行くディレクトリを指定します。--include-dirオプションはコマンドライン上でいくつでも使うことができます。

--keep-going
-k
　コマンドがエラーステータスを返してもmakeの処理を中止しません。代わりに、そのターゲットに関する残りの作業をスキップして他のターゲットの処理を続けます。

--just-print
-n
　makeが実行するはずのコマンド列を実行せずに表示します。makeが何を実行するのかを実行する前に知ることができるという点で非常に便利な機能です。このオプションにより実行されなくなるのはコマンドスクリプト内のコードだけであり、shell関数内のコード実行を妨げないという点に注意が必要です。

--old-file=*file*
-o *file*
　指定された*file*が非常に古いものであるとして扱い、それに対してターゲットを更新するための適切な処理を行います。誤ってファイルを更新してしまったときや、依存関係グラフ中におけるある必須項目の影響範囲を調べる際にこのオプションは有益です。この機能の反対が、--new-fileオプション（-Wオプション）です。

--print-data-base
-p
　makeの内部データベースを表示します。

`--touch`
`-t`

タイムスタンプを見て最新ではないと判断したファイルに対してtouchプログラムを実行します。この機能は依存関係グラフ中のファイルを最新状態にする際に有益です。例えば主要なヘッダファイル中のコメントを修正すると、不必要な大量のコードを再コンパイルする羽目になります。コンパイルを実行してコンピュータ資源を無駄に使う代わりに`--touch`オプションを使えば、すべてのファイルが最新の状態になります。

`--new-file=`*`file`*
`-W` *`file`*

指定した*`file`*をあらゆるターゲットよりも新しい物として扱います。あるターゲットを編集したりタイムスタンプを変更することなしに、ターゲットが更新されたものとして扱う際に使われます。この機能を補完するのが`--old-file`オプションです。

`--warn-undefined-variables`

定義されていない変数を展開する際に警告を表示します。定義されていない変数は、展開されずエラーにもならないので、この機能は問題解析ツールとして役立ちます。しかし定義されていない変数は、makefileのカスタマイズのためにも使われるため、値が入っていないすべてのカスタマイズ用変数に対する警告もこのオプションにより表示されてしまいます。

付録B
限界を超えて

ご覧になってきたように、GNU makeは非常にすばらしい機能を提供していますが、evalを使ってmake 3.80の制限を緩和する手法についてまだ十分に示していません。ここでは通常に提供している機能を拡張できるかどうかを見ていきます。

B.1 構造体

複雑なmakefileを書く際に困惑してしまう制限の1つが、makeに構造体の機能が欠けていることです。非常に限られた使用法ではありますが、ピリオドを変数名に含めることにより、（もしも我慢できるなら）構造体をシミュレートすることができます。

```
file.path = /foo/bar
file.type = unix
file.host = oscar
```

もし必要なら、変数名の合成を使えば関数に上記file構造体を渡すことができます。

```
define remote-file
  $(if $(filter unix,$($1.type)), \
    /net/$($1.host)/$($1.path), \
    //$($1.host)/$($1.path))
endef
```

しかしながら、いくつかの理由から、これは満足できるような解法ではありません。

- 構造体のインスタンスを作成するのが簡単ではありません。インスタンスの作成には新しい変数名の割り当てと各要素への値の設定とが含まれます。このようにして作成したそれぞれの擬似インスタンスが同じ要素（**スロット**（slot）と呼びます）を持っていることも保証できません。
- 構造体の構造は書いた人の頭の中にしかなく、名前により統合された実体というよりも、異なるmake変数の集合でしかありません。構造体自身には名前がないので、構造体への参照（ま

たはポインタ）も作ることができず、格好よく関数へ渡したり別の変数へ代入したりできません。

- 構造体のスロットを安全に操作する方法がありません。変数名のどの部分を書き間違えても、makeはエラーも出さずに誤った値（または値なし）に展開してしまいます。

しかし上記remote-file関数が、もう少し包括的な解へのヒントを与えてくれます。構造体インスタンスは変数名の合成を使って作ることとしましょう。初期の（いくつかは今でも使われている）Lispのオブジェクトシステムでは、同様のテクニックを使っています。構造体（ここではfile-infoとしましょう）はシンボリックな名前例えばfile_info_1で表されるインスタンスを持つことができます。

別のインスタンスはおそらくfile_info_2になるでしょう。この構造体のスロットは変数名を合成して次のように表すことができます。

```
file_info_1_path
file_info_1_type
file_info_1_host
```

このインスタンスは名前を持っているので、他の変数に代入することができます（いつものように再帰変数を使うか単純変数を使うかはプログラマの選択に委ねられています）。

```
before_foo = file_info_1
another_foo = $(before_foo)
```

file-infoの各要素はLisp風のgetとsetを使って読み書きできます。

```
path := $(call get-value,before_foo,path)
$(call set-value,before_foo,path,/usr/tmp/bar)
```

ここから発展させて、file-info構造体の新しいインスタンスを簡単に生成するためのテンプレートを作ることができます。

```
orig_foo := $(call new,file-info)
$(call set-value,orig_foo,path,/foo/bar)

tmp_foo := $(call new,file-info)
$(call set-value,tmp_foo,path,/tmp/bar)
```

これで2つのはっきりと区別されたfile-info構造体インスタンスができました。各スロットに初期値を持たせるという考え方も取り入れることができるでしょう。file-info構造体は次のように宣言します。

```
$(call defstruct,file-info,  \
  $(call defslot,path,),     \
  $(call defslot,type,unix), \
  $(call defslot,host,oscar))
```

defstruct関数には最初の引数に構造体の名前、そしてdefslot関数の呼び出し列が続きます。各defslotはひと組の（名前，初期値）を持ちます。例B-1はdefstruct関数と、その補助関数群です。

例B-1 makeの構造体定義

```
# $(next-id) - ユニークな値を返す
next_id_counter :=
define next-id
  $(words $(next_id_counter))$(eval next_id_counter += 1)
endef

# all_structs - 定義された構造体名のリスト
all_structs :=

value_sep := XxSepxX

# $(call defstruct, struct_name, $(call defslot, slot_name, value), ...)
define defstruct
  $(eval all_structs += $1)                                          \
  $(eval $1_def_slotnames :=)                                        \
  $(foreach v, $2 $3 $4 $5 $6 $7 $8 $9 $(10) $(11),                  \
    $(if $($v_name),                                                 \
      $(eval $1_def_slotnames          += $($v_name))                \
      $(eval $1_def_$($v_name)_default := $($v_value))))
endef

# $(call defslot,slot_name,slot_value)
define defslot
  $(eval tmp_id := $(next_id))
  $(eval $1_$(tmp_id)_name := $1)
  $(eval $1_$(tmp_id)_value := $2)
  $1_$(tmp_id)
endef

# all_instances - 全構造体における全インスタンスのリスト
all_instances :=

# $(call new, struct_name)
define new
$(strip                                                              \
  $(if $(filter $1,$(all_structs)),,                                 \
    $(error new on unknown struct '$(strip $1)'))                    \
  $(eval instance := $1@$(next-id))                                  \
  $(eval all_instances += $(instance))                               \
  $(foreach v, $($(strip $1)_def_slotnames),                         \
    $(eval $(instance)_$v := $($(strip $1)_def_$v_default)))         \
  $(instance))
endef

# $(call delete, variable)
define delete
```

```
$(strip                                                                    \
  $(if $(filter $($(strip $1)),$(all_instances)),,                         \
    $(error Invalid instance '$($(strip $1))'))                            \
  $(eval all_instances := $(filter-out $($(strip $1)),$(all_instances))) \
  $(foreach v, $($(strip $1)_def_slotnames),                               \
    $(eval $(instance)_$v := )))
endef

# $(call struct-name, instance_id)
define struct-name
  $(firstword $(subst @, ,$($(strip $1))))
endef

# $(call check-params, instance_id, slot_name)
define check-params
  $(if $(filter $($(strip $1)),$(all_instances)),,              \
    $(error Invalid instance '$(strip $1)'))                    \
  $(if $(filter $2,$($(call struct-name,$1)_def_slotnames)),,   \
    $(error Instance '$($(strip $1))' does not have slot '$(strip $2)'))
endef

# $(call get-value, instance_id, slot_name)
define get-value
$(strip                          \
  $(call check-params,$1,$2)     \
  $($($(strip $1))_$(strip $2)))
endef

# $(call set-value, instance_id, slot_name, value)
define set-value
  $(call check-params,$1,$2) \
  $(eval $($(strip $1))_$(strip $2) := $3)
endef

# $(call dump-struct, struct_name)
define dump-struct
{ $(strip $1)_def_slotnames "$($(strip $1)_def_slotnames)"      \
  $(foreach s,                                                  \
    $($(strip $1)_def_slotnames),$(strip                        \
    $(strip $1)_def_$s_default "$($(strip $1)_def_$s_default)")) }
endef

# $(call print-struct, struct_name)
define print-struct
{ $(foreach s,                                  \
    $($(strip $1)_def_slotnames),$(strip        \
    { "$s" "$($(strip $1)_def_$s_default)" })) }
endef

# $(call dump-instance, instance_id)
define dump-instance
{ $(eval tmp_name := $(call struct-name,$1))    \
  $(foreach s,                                  \
```

```
    $($(tmp_name)_def_slotnames),$(strip        \
    { $($(strip $1))_$s "$($($(strip $1))_$s)" })) }
endef

# $(call print-instance, instance_id)
define print-instance
{ $(foreach s,                                          \
    $($(call struct-name,$1)_def_slotnames),"$(strip    \
    $(call get-value,$1,$s))") }
endef
```

このコードをセクションごとに詳しく見ていきましょう。next-id関数の定義から始まります。これは単純なカウンタです。

```
# $(next-id) - ユニークな値を返す
next_id_counter :=
define next-id
  $(words $(next_id_counter))$(eval next_id_counter += 1)
endef
```

言語としては制限があるため、makeで数値計算を実行することはできないといわれています。一般的には正しいのですが、ここでのように限られた用途でなら必要なものを計算することはできます。この関数ではevalを使い単純変数の値を再設定しています。関数には2つの式があります。最初の式はnext_id_counterに入っている単語の数を返します。次の式では変数に別の単語を加えます。これは効率的な方法ではありませんが、数千くらいの数までならうまく働きます。

次のセクションではdefstract関数を定義するとともに、データ構造体の補助変数を作成しています。

```
# all_structs - 定義された構造体名のリスト
all_structs :=

value_sep := XxSepxX

# $(call defstruct, struct_name, $(call defslot, slot_name, value), ...)
define defstruct
  $(eval all_structs += $1) \
  $(eval $1_def_slotnames :=) \
  $(foreach v, $2 $3 $4 $5 $6 $7 $8 $9 $(10) $(11), \
    $(if $($v_name), \
      $(eval $1_def_slotnames += $($v_name)) \
      $(eval $1_def_$($v_name)_default := $($v_value))))
endef

# $(call defslot,slot_name,slot_value)
define defslot
  $(eval tmp_id := $(next_id))
  $(eval $1_$(tmp_id)_name := $1)
  $(eval $1_$(tmp_id)_value := $2)
  $1_$(tmp_id)
endef
```

all_structs変数はdefstruct関数により定義された全構造体のリストを保持します。これによりnew関数が新たに割り当てる構造体の型チェックを行うことができます。defstruct関数が定義するそれぞれの構造体をSとすると、それに対していくつかの変数を定義します。

```
S_def_slotnames
S_def_slotn_default †
```

最初の変数は、その構造体が持つ全スロットのリストです。次の変数が持つのは各スロットの初期値です。defstruct関数の最初の2行では、all_structs変数に値を追加し、スロット名リストの変数を初期化します。関数の残りの部分では、各スロットに対して順次スロット名リストへの追加と初期値の保存を行います。

各スロットの定義はdefslot関数が行います。この関数はスロットに対するidを定義し、スロット名と初期値をそれぞれの変数に保存し、その変数のプリフィックスを返します。プリフィックスを返すことで、各スロットを読み書きするときに使用するシンボルをdefstruct関数の引数のリストとすることが可能となります。後で属性を加える必要が生じても、それはスロット定義の中に押し込めるのはわかりやすい方法です。このテクニックにより、もっと単純な手法と比較しても、スロットの初期値に（空白を含む）さまざまな値を指定することができるようになります††。

defstruct中のforeachループでは、slotの最大許容数が決められています。ここでは10個までのスロットが使用可能となっています。foreachループの本体ではS_def_slotnamesにスロット名を追加し、初期値を各変数に設定します。例えば例で使用したfile-info構造体では次の変数が作られます。

```
file-info_def_slotnames := path type host
file-info_def_path_default :=
file-info_def_type_default := unix
file-info_def_host_default := oscar
```

以上で構造体の定義は終了です。

---

† 訳注：スロットの初期値を格納する変数は、スロットごとに作成されます。その際*slotn*の部分がスロット名に置き換えられます。例えばS構造体のpathというスロットでは、S_def_path_defaultになります。

†† 訳注：defslot関数‡で行われることを整理しましょう。$(call defslot, type, unix)という呼び出しを例に取って考えてみます。まずここでいうidはnext-idから得られるユニークな値です。ここでは例えば1としましょう。これをスロット名に連結してtype_1を作ります。これが変数のプリフィックスです。これを元に、スロット名を保持するtype_1_nameを作りtypeをセットします。次に値を保持するpath_1_valueを作り値にunixをセットします。最後にプリフィックスであるtype_1を返して終了します。呼び出し側であるdefstructでは、このプリフィックスを使い、type_1_name変数とtype_1_value変数の値を取り出します。

‡ 訳注：実は、このdefslot関数には誤りがあります。ユニークなidを得るのはnext_idではなく、next-idが正しい関数名です。defslotが返す値であるプリフィックスはスロット名と初期値をそれぞれ格納した2つの変数をいっしょに返すための苦肉の策といえるものであり、defstruct関数のforeachループが終了すると、意味がなくなってしまいます。実際にはこのプリフィックスをユニークにする理由はあまりありません。そのため関数名が誤っていてもこの例は正しく動作します。

これで構造体を作成することができます。構造体はインスタンスを個別に作ることができなければなりません。new関数がこの操作を行います。

```
# $(call new, struct_name)
define new
$(strip \
  $(if $(filter $1,$(all_structs)),, \
    $(error new on unknown struct '$(strip $1)')) \
  $(eval instance := $1@$(next-id)) \
  $(eval all_instances += $(instance)) \
  $(foreach v, $($(strip $1)_def_slotnames), \
    $(eval $(instance)_$v := $($(strip $1)_def_$v_default))) \
  $(instance))
endef
```

最初のifは指定された名前が実在の構造体を表しているのか調べています。all_structs変数の中に構造体名がなければ、エラーになります。次に、新たに作るインスタンス用に、構造体名にユニークな数をサフィックスとして加えユニークなidとして、これをインスタンス名とします。構造体名とサフィックスの間には、後で分離が容易になるようアットマーク（@）を区切り文字として入れます†。new関数はこの名前を記録して、値を読み書きする際のチェックに使います。そして構造体のスロットに初期値を設定します。興味深いのは、この初期化コードです。

```
$(foreach v, $($(strip $1)_def_slotnames), \
  $(eval $(instance)_$v := $($(strip $1)_def_$v_default)))
```

foreachループは構造体のスロットを順次処理します。構造体名をstrip関数に渡すことで、new関数呼び出し時にカンマの後に空白を置くことができるようになります。インスタンス名とスロット名を連結したもの（例えばfile_info@1_path）で各スロットは表されます。代入の右辺は構造体名とスロット名から合成された初期値の変数名です。そしてインスタンス名を関数の値として返します。

次に構造体の内部を読み書きする手段が必要です。そのために2つの関数を定義します。

```
# $(call get-value, instance_id, slot_name)
define get-value
$(strip \
  $(call check-params,$1,$2) \
  $($($(strip $1))_$(strip $2)))
endef

# $(call set-value, instance_id, slot_name, value)
define set-value
  $(call check-params,$1,$2) \
  $(eval $($(strip $1))_$(strip $2) := $3)
endef
```

† 訳注：file-infoのインスタンス名は、例えばfile-info@1となります。

スロットの値を読むために、get-value関数ではインスタンス名とスロット名からスロットの変数名を合成します。安全のために、まずインスタンス名とスロット名が正当であるかをcheck-params関数に渡して確認します。関数呼び出しでパラメータリストで空白を使っても、紛れ込んだ空白がスロットの値を壊さないようにするために、これらのパラメータをstrip関数に渡しています。

値を設定するset-value関数でも同様にパラメータを確認してから値を設定します。繰り返しになりますが、自由に空白を使って引数リストが書けるように引数をstrip関数に渡しています。スロットの値はstripに渡していないことに注意が必要です。なぜなら、値として空白が必要かもしれないからです。

```
# $(call check-params, instance_id, slot_name)
define check-params
  $(if $(filter $($(strip $1)),$(all_instances)),, \
    $(error Invalid instance '$(strip $1)')) \
  $(if $(filter $2,$($(call struct-name,$1)_def_slotnames)),, \
    $(error Instance '$($(strip $1))' does not have slot '$(strip $2)'))
endef

# $(call struct-name, instance_id)
define struct-name
  $(firstword $(subst @, ,$($(strip $1))))
endef
```

check-params関数は渡されたインスタンス名がインスタンス名リストに入っている値かどうかを単純に調べます。同様に、スロット名がこの構造体の一部であるかをチェックします。構造体名はインスタンス名を@の位置で分解した最初の文字列を使います。これはつまり、構造体名には@を使えないということを意味しています。

最後の仕上げとして表示とデバッグのための関数をふた組追加します。以下のように、print関数では構造体とインスタンスを単純で読みやすい形にして表示します。一方dump関数では、構造体とインスタンスの内部を詳細に表示します。

以下はfile-info構造体の定義と使用例です。

```
include defstruct.mk

$(call defstruct, file-info, \
  $(call defslot, path,), \
  $(call defslot, type,unix), \
  $(call defslot, host,oscar))

before := $(call new, file-info)
$(call set-value, before, path,/etc/password)
$(call set-value, before, host,wasatch)

after := $(call new,file-info)
$(call set-value, after, path,/etc/shadow)
$(call set-value, after, host,wasatch)
```

```
demo:
        # before = $(before)
        # before.path = $(call get-value, before, path)
        # before.type = $(call get-value, before, type)
        # before.host = $(call get-value, before, host)
        # print before = $(call print-instance, before)
        # dump before = $(call dump-instance, before)
        #
        # all_instances = $(all_instances)
        # all_structs = $(all_structs)
        # print file-info = $(call print-struct, file-info)
        # dump file-info = $(call dump-struct, file-info)
```

結果は次のように表示されます。

```
$ make
# before = file-info@0
# before.path = /etc/password
# before.type = unix
# before.host = wasatch
# print before = { "/etc/password" "unix" "wasatch" }
# dump before = { { file-info@0_path "/etc/password" } { file-info@0_type "unix" }
{ file-info@0_host "wasatch" } }
#
# all_instances = file-info@0 file-info@1
# all_structs = file-info
# print file-info = { { "path" "" } { "type" "unix" } { "host" "oscar" } }
# dump file-info = { file-info_def_slotnames " path type host" file-info_def_path_
default "" file-info_def_type_default "unix" file-info_def_host_default "oscar" }
```

また、以下は不正な構造体がどのようなエラー（「定義されていない'no-such-structure'がnew関数で使われた」の意と「インスタンス'foo@0'にはスロット'siz'が存在しない」の意）になるかを示しています。

```
$ cat badstruct.mk
include defstruct.mk
$(call new, no-such-structure)
$ make -f badstruct.mk
badstruct.mk:2: *** new on unknown struct 'no-such-structure'. Stop.

$ cat badslot.mk
include defstruct.mk
$(call defstruct, foo, defslot(size, 0))
bar := $(call new, foo)
$(call set-value, bar, siz, 10)
$ make -f badslot.mk
badslot.mk:4: *** Instance 'foo@0' does not have slot 'siz'. Stop.
```

もちろん改良すべき点はまだまだありますが、基本的なところは押さえました。以下に思いつく改良点を示します。

- スロットへの値設定に妥当性検査を加える。これはフック関数で行われますが、そのフック関数は設定が行われた後に影響を残さないように空文字列に展開されるものでなければなりません。フック関数は次のように使います。

```
# $(call set-value, instance_id, slot_name, value)
define set-value
  $(call check-params,$1,$2) \
  $(if $(call $(strip $1)_$(strip $2)_hook, value), \
    $(error set-value hook, $(strip $1)_$(strip $2)_hook, failed)) \
  $(eval $($(strip $1))_$(strip $2) := $3)
endef
```

- 継承のサポート。defstruct関数に他の構造体の名前をスーパークラスとして指定できるようにして、スーパークラスの全メンバをサブクラスでも使えるようにします。
- 構造体へのよりよい参照方法。現在のコードでは別の構造体をスロットに格納することもできますが、値を取り出すのは面倒です。（「構造体名@番号」を探すことで）参照関係をチェックし、自動的に参照を解決することができるようにget-value関数を改良することが可能です。

B.2　数値計算

先にmakeの組み込み機能だけで数値計算を行うのは不可能ではないと書きました。実際、リストへ単語を追加しそのリスト中の単語数を数えることで、単純なカウンタを実現しました。このカウント方法を発見した後、makeで整数を加算する限定的だけれどすばらしい手法がMichael Mounteneyから報告されました。

その手法とは、数列を操作して2つの自然数の合計を計算するというものです。どのように動作するかを見ましょう。次のような数列（number line）があるとします。

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

さて、（数列の順番が正しいとしたら）例えば「4＋5」を次のように計算することができます。まず数列の4番目から最後までの部分数列を作り、その5番目を取り出します。この操作はmakeの組み込み機能を使って実行することができます。

```
number_line = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
plus = $(word $2, $(wordlist $1, 15, $(number_line)))
four+five = $(call plus, 4, 5)
```

なんて賢いのでしょう。数列が0や1ではなく2から始まっていることに注目しましょう。これは「1＋1」を実行するために必要なことがわかるはずです。演算数がどちらも1なので、リストの最初の要素が使われることになり、それは2でなければなりません。wordとwordlist関数では要素の最初を0ではなく1で表すために、このようにする必要があります（わざわざ証明はしていません）。

さて、数列を使えば加算を実行できますが、手作業で書き込んだりシェルのプログラムを使わずに

どうすれば数列を生成できるのでしょうか。十の位の数と一の位の数の全組み合わせを作ることで00から99までの数値を生成できます。例を示しましょう†。

```
make -f - <<< '$(warning $(foreach i, 0 1 2, $(addprefix $i, 0 1 2)))'
/c/TEMP/Gm002568:1: 00 01 02 10 11 12 20 21 22
```

0から9までの数を使い00から99までの数値が作れます。foreachを百の位に組み合わせることにより、000から999までも作れます。後は必要に応じてstripを用いて前置された不必要な0を取り除くだけです。

次に示すのはMounteney氏のコードを手直しして、数列の生成とplusとgt演算ができるようにしたものです。

```
# combine - 数の組み合わせを作る
combine = $(foreach i, $1, $(addprefix $i, $2))

# stripzero - 先頭の0を取り除く
stripzero = $(patsubst 0%,%,$1)

# generate - 単語列を3回使いすべての組み合わせを作り出す
generate = $(call stripzero,                              \
             $(call stripzero,                            \
               $(call combine, $1,                        \
                 $(call combine, $1, $1))))

# number_line - 0から999までの数列を作る
number_line := $(call generate,0 1 2 3 4 5 6 7 8 9)
length := $(word $(words $(number_line)), $(number_line))

# plus - 数列を使い2つの整数の足し算を行う
plus = $(word $2,                                         \
         $(wordlist $1, $(length),                        \
           $(wordlist 3, $(length), $(number_line))))

# gt - 数列を使い$1が$2よりも大きいか調べる
gt = $(filter $1,                                         \
       $(wordlist 3, $(length),                           \
         $(wordlist $2, $(length), $(number_line))))

all:
        @echo $(call plus,4,7)
        @echo $(if $(call gt,4,7),is,is not)
        @echo $(if $(call gt,7,4),is,is not)
        @echo $(if $(call gt,7,7),is,is not)
```

† 訳注：これは3章のMAKECMDGOALSのところでも使ったテクニックですが、bashのヒアストリング機能を使ってmakefileを標準入力から与えています。warning関数が表示しているファイル名（ここでは/c/TEMP/Gm002568）は、bashが入力リダイレクトで使用している一時ファイル名になっています。そのため、実行する環境によりファイル名は変化しますし、同じ環境でも実行のたびにファイル名は異なります。

実行結果は次のようになります。

```
$ make
11
is not
is
is not
```

これを拡張して、逆順の数列を使い逆に数えるような操作を実現することで引き算ができます。例えば「7 − 4」を計算するために0から6までの部分数列を作り、逆順にして4番目の要素を取り出します。

```
number_line := 0 1 2 3 4 5 6 7 8 9...
1through6 := 0 1 2 3 4 5 6
reverse_it := 6 5 4 3 2 1 0
fourth_item := 3
```

makeの文法を使いアルゴリズムを記述してみます。

```
# backwards - 逆順の数列
backwards := $(call generate, 9 8 7 6 5 4 3 2 1 0)

# reverse - 単語列を逆順にする
reverse = $(strip                                       \
            $(foreach f,                                \
              $(wordlist 1, $(length), $(backwards)), \
              $(word $f, $1)))

# minus - $1引く$2を計算する
minus = $(word $2,                                      \
          $(call reverse,                               \
            $(wordlist 1, $1, $(number_line))))

minus:
        # $(call minus, 7, 4)
```

掛け算と割り算は読者への宿題としましょう。

付録C

GNU Free Documentation License
——GNU Project——
Free Software Foundation (FSF)†

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications

† 編集注：ライセンスの規定に従い、原文をそのまま掲載しています。

and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal

authors, if it has fewer than five), unless they release you from this requirement.

3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
8. Include an unaltered copy of this License.
9. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
10. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
11. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
12. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
13. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
14. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
15. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties-for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their

names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this

License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

索 引

記号

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X・Y

あ行

か行

さ行

た行

な行

は行

ま行

や行

ら行

わ行

●著者紹介

Robert Mecklenburg（ロバート・メクレンバーグ）

1977年、まだ学生だったときからUnixを使いはじめ、以来23年間プログラマとして働く。makeはNASAにいた1982年からのつきあい。その当時はUnixバージョン7だった。1991年、コンピュータ科学の博士号をユタ大学で取得。以来、CADからバイオインフォマティックスまで、広範囲にわたり活躍している。C++、Java、Lispなど言語の深い知識が執筆にも役に立った。

●監訳者紹介

矢吹 道郎（やぶき みちろう）

1981年 上智大学理工学研究科電気電子工学専攻終了

現在　明星大学情報学部情報学科助教授

●訳者紹介

菊池 彰（きくち あきら）

日本アイ・ビー・エム株式会社ソフトウェア開発研究所勤務

カバーの説明

本書のカバーに描かれている動物は、ロリス猿の一種で、ポットーと呼ばれるものです。西アフリカの熱帯林に生息する小型の霊長類で、体長は約40センチ。全身が赤茶けた柔らかい密集した毛で覆われています。ものを掴むのに適した手の形をしていて、木の上で生活するのに都合の良いように強い握力を持っています。昼間は木にあいた穴や裂け目の中で眠り、夜になると食料（昆虫、かたつむり、こうもり、果実など）を求めて出てきます。ほかの多くの霊長類とは異なり、通常単独で生活します。

GNU Make 第3版

2005年12月26日　初版第1刷発行
2014年 7 月31日　初版第8刷発行

著　　者　Robert Mecklenburg（ロバート・メクレンバーグ）
監 訳 者　矢吹 道郎（やぶき みちろう）
訳　　者　菊池 彰（きくち あきら）
発 行 人　ティム・オライリー
制　　作　株式会社 GARO
印刷・製本　株式会社ルナテック
発 行 所　株式会社オライリー・ジャパン
　〒160−0002　東京都新宿区坂町26番地27　インテリジェントプラザビル 1F
　Tel（03）3356−5227
　Fax（03）3356−5263
　電子メール japan@oreilly.co.jp
発 売 元　株式会社オーム社
　〒101−8460　東京都千代田区神田錦町3−1
　Tel（03）3233−0641（代表）
　Fax（03）3233−3440

Printed in Japan（ISBN4−87311−269−9）
乱丁本、落丁本はお取り替え致します。