プログラミングユーティリティ

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650–960–1300

Part No: 805–5827–10
1998 年 11 月

原典: *Programming Utilities Guide*

Part No: 805-4035-10

Revision A

Adobe PostScript

目次

はじめに

本書『プログラミングユーティリティ』には、SunOS システムで使用可能な特別な組み込み型プログラミングツールに関する開発者向けの情報が記載されています。

対象読者

本書は、Solaris 2.x システムを使用するアプリケーションプログラマを対象としています。

お読みになる前に

このマニュアルは、Solaris 2.x オペレーティングシステム、プログラミング、ネットワーキングについて読者が理解していることを前提としています。

このマニュアルの構成

このマニュアルは、以下に示す章で構成されており、プログラミングを支援する各ツールについて説明します。

第 1 章

TNF は、プログラムの実行に関するトレース情報を収集します。Trace Implementation Format を使用することにより、プローブポイントをソースコードに挿入して、分析用のデータを収集できます。

第 2 章

`lex` は、テキストの簡単な字句解析に使用されるプログラムを生成します。この `lex` は、さまざまな文字列を識別することによって問題を解決するツールです。

第 3 章

yacc は、言語パーサーを生成します。この yacc は、コンピュータ入力をある特定の構造にして、それを入力ストリームを検査する C 言語の関数に変換します。

第 4 章

`make` は、関連するプログラムとファイルの保守、更新、再生成を自動的に行います。

第 5 章

SCCS (Source Code Control System) によって、共有ファイルに対するアクセスを制御したり、プロジェクトに加えられた変更の履歴を保存することができます。

第 6 章

`m4` マクロ言語プロセッサは、ライブラリアーカイブを作成し、ファイルの追加や抽出を行います。

付録 A

System V `make` は、旧バージョンの `make` と互換性のあるバージョンの `make(1)` です。

以下、その他の重要なツールについて簡単に説明します。これらのツールについての詳細は、該当するマニュアルページを参照してください。

`ar(1)`	移植性のあるライブラリまたはアーカイブの作成と保守を行います。
`cpp(1)`	C 言語の命令を前処理します。

`dis(1)`	COFF のオブジェクトコードを逆アセンブルします。
`dump(1)`	オブジェクトファイルの選択部分をダンプ (表示) します。
`lorder(1)`	オブジェクトのライブラリまたはアーカイブの順序関係を検索します。
`mcs(1)`	ELF オブジェクトファイルのコメント部を操作します。
`nm(1)`	オブジェクトファイルの名前一覧を出力します。
`size(1)`	オブジェクトファイルのサイズを表示します。
`strip(1)`	オブジェクトファイルからシンボルと再配置ビットを削除します。
`tsort(1)`	トポロジによるソートを実行します。
`unifdef(1)`	C 言語のソースから `ifdef` 行を展開して削除します。

関連文書

- 各ユーティリティコマンドのマニュアルページ

マニュアルの注文方法

SunDocs™ プログラムでは、米国 Sun Microsystems™, Inc. (以降、Sun™ とします) の 250 冊以上のマニュアルを扱っています。このプログラムを利用して、マニュアルのセットまたは個々のマニュアルをご注文いただけます。

マニュアルのリストと注文方法については、米国 SunExpress™, Inc. のインターネットホームページ `http://www.sun.com/sunexpress` にあるカタログセクションを参照してください。

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 P–1 表記上の規則

字体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、およびディレクトリ名を示します。または、画面上のコンピュータ出力を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 system%
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力とは区別して示します。	system% **su** password:
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm *filename* と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「 」	参照する章や節を示します。また、ボタンやメニューなど、強調する単語を囲む場合にも使用します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。

ただし AnswerBook2™ では、ユーザーが入力する文字と画面上のコンピュータ出力は区別して表示されません。

コード例は次のように表示されます。

- C シェルプロンプト

```
system% command y|n [filename]
```

- Bourne シェルおよび Korn シェルのプロンプト

```
system$ command y|n [filename]
```

■ スーパーユーザーのプロンプト

```
system# command y|n [filename]
```

[ ]は省略可能な項目を示します。上記の場合、*filename* は省略してもよいことを示します。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: Shift キーを押します)。ただし、キーボードによっては Enter キーが Return キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、Ctrl-D は Control キーを押したまま D キーを押すことを意味します。

一般規則

- このマニュアルでは、英語環境での画面イメージを使っています。このため、実際に日本語環境で表示される画面イメージとこのマニュアルで使っている画面イメージが異なる場合があります。本文中で画面イメージを説明する場合には、日本語のメニュー、ボタン名などの項目名と英語の項目名が適宜、併記されています。
- 「x86」という用語は、一般に Intel 8086 ファミリに属するマイクロプロセッサを意味します。これには、Pentium、Pentium Pro の各プロセッサ、および AMD と Cyrix が提供する互換マイクロプロセッサチップが含まれます。このマニュアルでは、このプラットフォームのアーキテクチャ全体を指すときに「x86」という用語を使用し、製品名では「Intel 版」という表記で統一しています。

第 1 章

TNF ユーティリティによるプログラム実行のトレース

この章では、TNF (Trace Normal Form) ユーティリティを使用してプログラムからデータを収集する方法について説明します。

TNF ユーティリティは、TNF バイナリファイルの作成、処理、読み取りを行う、ライブラリおよびプログラム群で構成されています。この TNF ユーティリティを使用して、以下の作業を行うことができます。

- C または C++ プログラムの実行をトレースする
- C または C++ プログラムをデバッグする
- C または C++ プログラムのパフォーマンスデータを収集する
- アプリケーション実行中のカーネルの活動状況をトレースする

TNF ユーティリティは、初級プログラマが実行中のプログラムのイベントをトレースする場合にも、上級プログラマがマルチスレッド化されたプログラムにおける複数のスレッドとカーネルとの間の相互作用をトレースする場合にも、便利なユーティリティです。

- 2ページの「ユーザーの分類」
- 3ページの「TNF ユーティリティの使い方」
- 21ページの「カーネルのトレース」
- 35ページの「上級者向きのトピック」

この章の始めの部分では、TNF ユーティリティの概要について説明しています。その後に、詳細な情報が記述されています。初級プログラマの方は、目的の作業を

行うのに必要な部分だけを読んでください。上級プログラマの方は、この章をすべて一読することをお勧めします。

ユーザーの分類

初級プログラマと上級プログラマの区別とは別に、TNF ユーティリティの用途によってユーザーを以下のように分類できます。

既存のプローブポイントを使用

プログラムまたはライブラリに付属しているプローブポイントを使用する場合は、`prex` および `tnfdump` に関する節をお読みください。プローブの挿入方法についての説明はとばしても構いません。

プログラムをデバッグ

プログラムをデバッグについては、この章の必要な部分を読んでください。実行するトレースの複雑さによって、必要な箇所は異なります。

ライブラリへプローブポイントを挿入

作成中のライブラリにプローブポイントを挿入する場合は、この章全体をお読みください。また、ライブラリのユーザー用に、プローブポイントに関する情報を必ず提供してください。

カーネル実行をトレース

カーネル内のいくつかの TNF プローブは、システムコール、スレッドの状態の移り変わり、ページフォルト、スワッピング、および入出力を記録します。カーネルをトレースする方法の大部分は、ユーザーレベルのプロセスをトレースする場合と同じです。

TNF ユーティリティの使い方

TNF ユーティリティは、`libtnfprobe` ライブラリ、`prex(1)` コマンド、および `tnfdump(1)` コマンドで構成されています。

- `libtnfprobe` - プログラムにリンクされているライブラリで、トレース記録をバイナリの TNF トレースファイルに生成します。
- `prex` - プローブポイントと、プローブポイントによって収集された各種データを操作するユーティリティです。
- `tnfdump` - バイナリの TNF トレースファイルを ASCII ファイルに変換するユーティリティです。

コード内のデータを収集する箇所 (プローブポイント) に、コマンドを挿入することができます。`prex` を使用して実行可能オブジェクトを実行すると、プローブポイントの操作が可能になります。収集された情報は、Trace Normal Form (TNF) と呼ばれる形式のトレースファイルに書き込まれます。`tnfdump` は、そのバイナリの TNF トレースファイルを ASCII ファイルに変換します。

プローブポイントは、`.init()` セクション、`.fini()` セクション、マルチスレッド化されたコード、共有オブジェクト、`dlopen(3X)` によってオープンされた共有オブジェクトなど、C または C++ コードの任意の場所に挿入できます。

また、プローブ関数と呼ばれる関数をこれらのプローブポイントに接続できます。プローブ関数は、収集された情報に基づいてアクションを実行します (現在使用できるプローブ関数は、出力を `stderr` に書き込むデバッグ関数だけです) 。

通常の TNF セッションでは、以下の操作を行います。

1. プローブポイントをプログラムに挿入します。ソースコードには、プローブポイントがすでに組み込まれている場合もあります。
2. `prex` を使用して、プログラムを起動します。`libtnfprobe` は自動的に `prex` によって起動前にロード (プリロード) されます。プログラムを起動するときに、そのプログラムを `libtnfprobe` にリンクすることもできます。その場合は、実行中のプログラムに `prex` を接続できます。5ページの「実行中のプロセスへの `prex` の接続」を参照してください。
3. `prex` でプローブポイントを操作して、必要な情報をプログラムから収集します。
4. `tnfdump` を使用して、ターゲットプログラムから生成されたバイナリの TNF 情報トレースファイルを ASCII ファイルに変換します。20ページの「読み取り可能な形式へのバイナリファイルの変換」を参照してください。

5. ASCII ファイル中の情報を確認します。

プローブポイントの挿入

プローブポイントの挿入についての詳細は、35ページの「上級者向きのトピック」で説明しています。

15ページの「C プログラムのサンプル」では、プローブポイントの設計と配置について説明しています。17ページの「prex セッションのサンプル」と19ページの「トレースファイルの読み取り」では、プローブポイントの操作方法と、それぞれのプローブポイントでどのような情報が収集されるかについて説明しています。

prex の起動

ユーザーは、prex を使用してプログラムをロードするか、もしくは実行中のプロセスに prex を接続することができます。以下の 2 つの節で、その手順を説明します。

prex によるプログラムの起動

prex を使用してプログラムを起動すると、ターゲットのプログラムに libtnfprove ライブラリがプリロードされます。したがって、プログラムを明示的に libtnfprobe にリンクする必要はありません。

prex は、プログラムを起動し、コードが実行される前にそのプログラムを停止します。.init() セクションが実行される前であってもプログラムの実行は停止されるため、プローブポイントを .init() セクションに配置して初期化コードをトレースすることができます。

プローブポイントが含まれていて、引数を受け取らない a.out という単純な実行可能プログラムの場合には、以下のコマンドを使用できます。

```
$ prex a.out
```

注 - (prex は、環境変数 $PATH を使用して実行可能プログラムを検索します。)

オプションを指定するには、以下の書式で指定します。

```
prex [-o outfilename] [-s kbytes_size]
     [-l sharedobjs] cmd [cmdargs...]
```

–o、–s、–l オプションについては、6ページの「prex のコマンド行オプション」で説明しています。

実行中のプロセスへの prex の接続

a.out という名前のプログラムを実行中に、その a.out のプロセス ID (PID) が 2374 である (ps(1) コマンドを使用して確認できます) が示した場合には、以下のコマンドによって prex を a.out に接続できます。

```
$ prex -p 2374
```

実行中のプログラムに prex を接続する場合には、先に libtnfprobe をそのプログラムにリンクする必要があります。libtnfprobe にリンクしていないと、libtnfprobe がプログラムにリンクされていないというエラーメッセージが表示され、prex が終了します。

libtnfprobe とのリンク

libtnfprobe ライブラリにリンクすると、割り当てられたトレースファイルのサイズ (prex のコマンド行オプションで制御可能) に約 33 KB (25 KB は共有可能) を加えた分だけプログラムのサイズが、増加します。prex を使用して実行中のプログラムに接続しない場合は、libtnfprobe にリンクしないでください。

ただし、再起動が容易ではなく、長時間連続して実行するプログラムの場合で、将来そのプログラムに関する情報を収集する可能性がある場合には、libtnfprobe をプログラムにリンクしてください。ウィンドウサーバーなどが、その例です。このような停止や再起動を頻繁に行わない種類のプログラムの場合には、プログラムの実行を開始する前に libtnfprobe にリンクする必要があります。

以下のいずれかの方法で libtnfprobe に、リンクすることができます。

これからプログラムのコンパイルを行う場合には、cc 行に –ltnfprobe を追加します (–lthread を使用している場合には、必ず –lthread の前に –ltnfprobe を入力してください)。

```
$ cc -ltnfprobe -lthread -o cookie cookie.c
```

プログラムがコンパイル済みの場合、または libtnfprobe に明示的に依存するプログラムを作成しない場合には、以下のコマンドを使用します。

```
$ LD_PRELOAD=libtnfprobe.so.1 executable_object_name
```

LD_PRELOAD については、『リンカーとライブラリ』を参照してください。

prex の停止と再開

prex を使用してプログラムをロードするか、または実行中のプログラムに prex を接続したら、Control-c キーを押してプログラムを停止し、制御を prex に渡します。

prex が実行されて、プログラムが停止すると、prex のプロンプトが表示されます。

```
prex>
```

このプロンプトに prex コマンドを入力して、プローブポイントを制御します。たとえば、プローブポイントの一覧表示や、トレースの開始、プログラムのデバッグを行うことができます。

プログラムの実行を再開するには、continue コマンドを入力します。

```
prex> continue
```

プログラムを停止して prex に戻るには、Control-c キーを押します。

```
$ ^c
prex>
```

prex の標準のコマンド行オプションを表 1–1 に示します。

prex のコマンド行オプション

表 1–1　prex のコマンド行オプション

オプション	定義
–o*trace_file*	トレースファイル *trace_file* には、prex コマンドによって収集された情報が格納されます。*trace_file* の場所は、prex を起動したディレクトリに対する相対パスとみなされます。引数として *trace_file* を指定しなかった場合は、$TMPDIR/trace-*pid* がデフォルトの場所になります。この *pid* には、プログラムのプロセス ID が入ります。$TMPDIR が設定されていない場合は、/tmp が使用されます。 プログラムをトレースするときには、起動時に設定されたトレースファイル (デフォルトのファイル、または –o ファイル名で指定されたファイル) がそのプログラム用として使用されます。
–l *libraries*	引数 *libraries* には、プログラムにプリロードするライブラリの名前を指定します。ライブラリの指定方法、およびライブラリの検索場所については、LD_PRELOAD (ld(1) を参照) の規則が適用されます。複数のライブラリを指定するには、各ライブラリ名を空白文字で区切り、そのリスト全体を二重引用符で囲んでください。 prex がプログラムをロードするときには、libtnfprobe.so.1 がデフォルトのライブラリになります。実行中のプロセスに prex を接続するときには、このオプションを使用することはできません。
–s *size*	*size* は、キロバイト (2^{10} バイト) で表したトレースファイルのサイズです。トレースファイルのデフォルトのサイズは、4 MB です。指定可能な最小サイズは、128 KB です。

prex の実行

prex をプログラムに接続後、prex コマンドを使用して、収集するデータのパラメータを変更できます。prex list コマンドを使用して、プログラム内のプローブをすべて表示できます。プログラムの実行中にプローブポイントが実行されると、その実行ごとに情報がトレースファイルに記録されます。収集された情報は、tnfdump() を使用して表示します。tnfdump() については、20ページの「tnfdump ファイルの読み取り」を参照してください。

ユーザーは、プログラムに対する prex の接続と切り離しを繰り返し行なって、複数のセッションを作成できます。たとえば、最初のセッションでは、prex を使用し

てターゲットプログラムを起動し、プローブを設定してから prex コマンドの quit resume を実行することができます。

この操作は、prex の起動、収集する情報の定義、prex の終了を行い、最後にプログラムの実行を再開することによって情報の収集を開始します。これらの処理を行うコマンドを表 1–2 に示します。

2 回目のセッションでは、prex を同じプログラムに接続して、プローブを再設定してから、再度 prex コマンドの continue を実行することができます。この操作は繰り返し何度でも行うことができます。

注 - すべてのトレース結果は、最初のセッションで名前を指定したトレースファイルに書き込まれます。このトレースファイル名を変更することはできません。

表 1–2 基本的な prex コマンド

コマンド	結果
% **prex a.out**	プログラムに prex を接続し、prex を起動します。
prex> **enable $all**	すべてのプローブを有効にします。
prex> **quit resume**	prex を終了して、プログラムを再実行します。

prex コマンドの構成要素

プローブポイントをソースコードに挿入するときに、さまざまな属性や (必要に応じて) 値を各プローブポイントに割り当てることができます。たとえば、任意の名前を "name" 属性の値に割り当てることができます。

定義済みの属性とその値の意味を表 1–3 に示します。prex コマンドを使用して、プローブポイントの属性または値に一致する複数のプローブポイントをグループとして選択できます。また、選択したプローブポイントのトレースやデバッグを行うこともできます。

prex コマンドとそのアクションは、表 1–5 を参照してください。

コード例 1–1 では、TNF_PROBE マクロによって work_start という名前のプローブポイントが定義されています。このプローブポイントには、state と message

という 2 つの引数があります。work_start という名前のプローブが検出されるたびに、変数 state と message のタイムスタンプと値がトレースファイルに記録されます。

コード例 1–1　prex の属性と値

```
#include <tnf/probe.h>

int
work(int state, char *message)
{
   TNF_PROBE_2(work_start, "work_module work"
               "sunw%debug in function work",
               tnf_long, int_input, state,
               tnf_string, string_input, message);
  ...
  ...
}
```

TNF_PROBE マクロとこのコード例についての詳細は、36ページの「TNF_PROBE マクロの使い方」を参照してください。

属性

属性は、コード内のプローブポイントを識別するための特性です。定義済みの属性を表 1–3 に示します。ユーザーは、TNF_PROBE を使用して別の属性を追加定義することができます。36ページの「TNF_PROBE マクロの使い方」を参照してください。

表 1–3　定義済みの属性

属性	特性	値
enable	プローブポイントは、そのプローブポイントが有効な場合にだけ、設定されているアクションを実行します。たとえば、トレースを行うように設定しても、プローブポイントが有効になっている場合にしかトレースは行われません。	OFF (デフォルト)
file	プローブポイントが含まれているファイルの名前。	work.c
funcs	このプローブに接続されたプローブ関数のリストを表示します。現在使用できるのはデバッグ関数だけです。	<値なし> (デフォルト)
keys	プローブポイントが属するグループ。プローブポイントのいずれかのキーが有効になると、そのプローブポイントは有効になります。	work_module work
line	プローブポイントが現われるコード内の行番号	10

表 1–3 定義済みの属性 続く

属性	特性	値
name	プローブポイントの名前	work_start
object	プローブポイントが含まれている共有オブジェクトまたは実行可能プログラムの名前。特定のモジュール中のプローブをすべて選択する場合に便利な属性です。	work
slots	プローブポイントの引数 (arg_name_n については、37ページの「TNF_PROBE_1 ～ TNF_PROBE_5」を参照) の名前。	int_input string_input
trace	プローブポイントのトレースがオンの場合、プログラムのプローブポイントを実行するたびに、データが 1 行ずつトレースファイルに書き込まれます。	ON (デフォルト)

プローブの照合構文

プローブの属性と値に基づいてプローブを選択します。 *selector_list* に選択条件を指定します。それぞれの属性または値は、以下のいずれかで表現できます。

- 識別子－英字、数字、_ \ . % (下線、バックスラッシュ、ピリオド、パーセント記号) を使用した文字列。識別子の最初の文字を数字にすることはできません。
- 引用符付き文字列－単一引用符で囲んだ文字列。この文字列は文字どおりに解釈される直定数とみなされます。照合する文字列が予約語の場合に便利な表現です。予約語の一覧については、表 1–4 を参照してください。
- 正規表現－スラッシュ (/) で囲んだ文字列。ed(1) の規則に従って照合用に拡張されています。正規表現においてパス名などを表すためにスラッシュを使用する場合は、/\/tmp\/*filename* のように、バックスラッシュでそのスラッシュをエスケープしてください。

selector_list は、1 つ以上の *selector*= という形式の選択条件で構成されています。最初の *selector*= が指定されていない場合は、keys= がデフォルトになります。たとえば、enable コマンドは以下の書式で指定します。

enable *selector_list*

たとえば、以下のように指定します。

```
enable name=/first/ file='sampleZ.c'
```

このコマンドは、値 first が含まれている name 属性 (正規表現による一致) をもつプローブポイント、または値が sampleZ.c である file 属性を持つプローブポイントをすべて有効にします。なお、トレースは、論理積ではなく論理和であることに注意してください。

選択条件 *selector_list* の簡略名として、$*set_name* のように変数を設定することができます。以下の例では、myprobes がその *set_name* に当たります。

```
create $myprobes name=/first/ file='sampleZ.c'
enable $myprobes
```

これは、前述の例と同じ処理を行います。set_name は、識別子の命名規則に従っています。デフォルト設定の $all は、プログラムのプローブをすべて選択します。

予約語

予約語を表 1–4 に示します。これらの予約語を使用して属性または値を選択する場合は、その予約語を単一引用符で囲んでください。

表 1–4 予約語

add	alloc	buffer
clear	connect	continue
create	dealloc	delete
disable	enable	fcns
filter	help	history
kill	ktrace	list
off	on	pfilter
probes	quit	resume
sets	source	suspend
trace	untrace	values

たとえば、以下のコマンドは、trace 属性の値が ON のプローブをすべて有効にします。trace と on はどちらも予約語なので、単一引用符で囲む必要があります。

```
enable 'trace'='on'
```

表 1–5 prex のコマンド

コマンド	アクション
`clear $`*set_name* `clear` *selector_list*	接続されているプローブ関数を切り離します。
`connect &debug $`*set_name* `connect &debug` *selector_list*	デバック関数をプローブポイントに接続します。このコマンドを実行しても、プローブポイントは有効になりません。デバッグ関数は、トレースファイルにではなく `stderr` に出力を送ります。
`continue`	`prex` を接続したまま、プログラムの実行を再開します。
`create $`*set_name selector_list*	*selector_list* と一致するプローブポイントのグループを作成します。同時に、*selector_list* のエイリアス *$set_name* を作成します。
`enable $`*set_name* `enable` *selector_list* `disable $`*set_name* `disable` *selector_list*	プローブポイントに設定されているアクションをそのプローブポイントに実行させるかどうかを制御します。デフォルトでは、プローブポイントは無効になっています。つまり、`prex` はトレースをオンにしません。プローブポイントのトレースを停止する最も効率的な方法 (プローブポイントにおける実行時間の面から考えた場合) は、`disable` コマンドを使用する方法です。 `enable` と `disable` コマンドは、主スイッチで、トレースを実行するかどうかを決定する最も優先度が高い要素です。プローブポイントが無効の場合は、そのプローブポイントが `debug()` 関数に接続されていて `trace` 属性がオンであっても、`stderr` やトレースファイルに情報は送られません。
`help`	使用可能な `prex` コマンドをすべて表示します。
`list` *attribute*`s probes` *selector_list* `list` *attribute*`s probes$` *set_name*	指定されたプローブポイントが有効/無効のどちらであるか、そのプローブポイントのトレースがオン/オフのどちらであるか、および接続されているプローブ関数を表示します。9ページの「属性」で説明したように、属性が選択条件になります。たとえば、以下のコマンドは、一致したプローブポイントの *name* 属性と *file* 属性の値だけを表示します。 **`list name file probes $all`** 以下のコマンドは、デフォルトの属性とその値 (*name*、*enable*、*trace*、*file*、*line*、*funcs*) をすべて表示します。 **`list probes $all`**
`list fcns`	定義済みの関数 (現在は、*&debug* だけが定義されています) を表示します。

表 1–5 prex のコマンド 続く

コマンド	アクション
`list history`	制御コマンドの履歴を表示します。`dlopen()` によって新しい共有オブジェクトがプログラムに取り込まれるたびに `connect`、`clear`、`trace`、`untrace`、`enable`、`disable` で使用したコマンドの履歴が記録されます。44ページの「`dlopen()`、`dlclose()`、履歴」を参照してください。
`list sets`	定義されたセットを表示します。
`list values` *attributes*	*attributes* に指定された属性に関連付けられた固有値を表示します。たとえば **`list values keys`** を実行すると、プログラム内の固有キーがすべて表示されます。
`source` *filename*	`prex` コマンドのファイルを読み込みます。ファイル名は引用符で囲んでください。
`trace $`*set_name* `>trace` *selector_list* `untrace $`*set_name*	プローブポイントのトレース動作を制御します。`trace` と `untrace` を使用して、プローブポイントを実行したときにそのプローブポイントにトレース記録を生成させるかどうかを切り替えます。`trace` と `untrace` のどちらを実行しても、プローブポイントは有効になりません。
`untrace` *selector_list*	デフォルトでは、トレースはオンになります。 `untrace` コマンドは、デバッグ出力だけを得たいときに利用できます。このコマンドを使用する際には、トレースをオフ、デバッグをオンにして、プローブを有効にする必要があります。 `debug` は `stderr` に書き込みを行いますが、`trace` は `mmap` された (メモリーが割り当てられた) ファイルに書き込みを行います。そのため、トレースの方がデバッグ関数よりもプログラムに与える負荷が少なくなります。
`quit`	`prex` を終了します。`prex` を使用してプログラムがロードされている場合は、そのプログラムを強制終了します。`prex` にプログラムが接続されている場合は、そのプログラムが再実行されます。
`quit kill`	`prex` を終了して、プログラムを強制終了します。
`quit resume`	`prex` を終了して、プログラムを再実行します。
`quit suspend`	プログラムを中断したまま、`prex` を終了します。

トレース、プローブポイント、デバッグ関数の設定状態

プローブポイントをトレースするには、プローブポイントのトレースをオンにして、プローブポイントを有効にする必要があります。プローブポイントでデバッグするには、プローブポイントをデバッグ関数に接続して、そのプローブポイントを有効にする必要があります。

トレース、プローブポイント、デバッグ関数の設定状態の組み合わせと、その結果を表 1–6 に示します。

表 1–6 トレース、プローブポイント、デバッグ関数の設定状態の組み合わせとその結果

プローブポイント (有効または無効)	トレースの状態 (オン/オフ)	デバッグの状態 (接続/未接続)	結果
有効	オン	接続	トレースとデバッグ
有効	オン	未接続	トレースのみ
有効	オフ	接続	デバッグのみ
有効	オフ	未接続	なし
無効	オン	接続	なし
無効	オン	未接続	なし
無効	オフ	接続	なし
無効	オフ	未接続	なし

スクリプトによる prex の制御

prex へのコマンドは、prex コマンド行から、または prex コマンドが含まれているファイルから入力します。

prex を起動すると、prex は .prexrc という名前のファイルをまず $HOME/ ディレクトリ内で探し、その次に、prex を起動したディレクトリ内で探します。コマンドは、検索されたすべてのファイルから読み取られます。したがって、ホームディ

レクトリの .prexrc ファイルで設定されたデフォルト値が、現在のディレクトリの .prexrc ファイルによって上書きされる可能性もあります。

検索された .prexrc ファイルを読み取った後は、prex コマンド行から入力されたコマンドが実行されます。.prexrc ファイルを使用して作業のすべてを設定する場合は、.prexrc ファイル中の最後の文を quit resume にします。quit resume は、prex を終了してプログラムを再実行します。

また、prex の実行中には、source *filename* コマンドを使用して、prex が読み取るコマンドが記述されているファイル *filename* を指定できます。このファイルは、どのような名前でも構いません。

スクリプト内のコマンドは、prex コマンド行から入力するコマンドと同様に、すべて ASCII 形式にしてください。スクリプトには以下の規則が適用されます。

- 各コマンドの最後には、改行文字を入力します。
- 次の行までコマンドが続く場合は、1 行目の終わりにバックスラッシュ (\) を入力します。
- トークンは、空白 (1 つ以上の空白文字またはタブ) で区切ります。
- コメントの先頭には、ハッシュ記号 (#) を入力します。

  コマンド行から入力する prex コマンドとスクリプトのコマンドとでコマンド言語は同じですが、スクリプトでは出力が stdout に送られるため、出力を返すコマンド (たとえば、list probes...) をスクリプトで使用してもあまり意味がありません。

C プログラムのサンプル

このプログラム cookie.c は give me a COOKIE! というメッセージを表示します。それに対する応答は、大文字で入力する必要があります。小文字の場合は誤りとみなされます。また、数値の素因数を計算することもできます。これら 2 つの処理過程を通して、トレース機能がどのように動作しているかを見てみましょう。

このプログラムをコンパイルして cookie() という実行可能プログラムを作成するには、以下のコマンドを実行します。

```
$ cc -o cookie cookie.c
```

このプログラムでは、5 つのプローブポイントが定義 (大文字で表記) されています。各プローブポイントの名前は、start (17 行目)、inloop (33 行目)、factor_start (60 行目)、found_a_factor (65 行目)、factor_end (72 行

目)となっています。これらのプローブポイントについての詳細は、17ページの「prex セッションのサンプル」で説明しています。

コード例 **1–2**　cookie.c のコード

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <tnf/probe.h>

#define MAX_RESPONSE_SIZE 256

static void find_factor(int n);

int
main(int argc, char **argv)
{
    boolean_t shouldexit = B_FALSE;
    int       sum = 0, max_loop = 5;
    int       i;

    TNF_PROBE_0(start, "cookie main",
         "sunw%debug starting main");

    while (!shouldexit) {
       char response[MAX_RESPONSE_SIZE];
       int factor_input;

       (void) printf("give me a COOKIE! ");
       (void) scanf(``%s'', response);

       if (!strcmp(response, "COOKIE")) {
            (void) printf("thanks!\n");
            shouldexit = B_TRUE;
       }

       else if (!strcmp(response, "loop")) {
            for (i = 0; i < max_loop; i++) {
               TNF_PROBE_2(inloop, "cookie main
                           loop","sunw%debug in the loop",
                           tnf_long, loop_count,  i,
                           tnf_long, total_iterations,  sum);
               sum++;
            }
            max_loop += 2 ;
       }

      else if (!strcmp(response, "factor")) {
            (void) printf("number you want factored? ");
            (void) scanf("%d", &factor_input);
            find_factor(factor_input);
      }

      else {
            (void) printf("not a %s, ", response);
      }
    }
    return 0;
```

```
}   /* end main */
static void
find_factor(int n)
{
    int i;

    (void) printf("\tfactors of %d = ", n);
    TNF_PROBE_1(factor_start, "factor", "",
                tnf_long, input_number, n);

    for (i=2; i <= n; i++) {
        while (n % i == 0) {
           TNF_PROBE_2(found_a_factor, "cookie find_factor",
                       "", tnf_long, searching_for, n,
                       tnf_long, factor, i);
           (void) printf("%d ", i);
           n /= i;
        }
    }
    TNF_PROBE_0(factor_end, "factor", "");
    (void) printf("\n");
}
```

prex セッションのサンプル

この prex セッションのサンプルは、prex のさまざまな機能を紹介できるように作られています。cookie を実行するときに収集されるデータは、20ページの「tnfdump ファイルの読み取り」に示しています。

```
% prex cookie     /* prex によって、実行可能プログラム cookie がロードされます。*/
Target process stopped
Type "continue" to resume the target, "help" for help ...
prex> list sets
$all      'keys'=/.*/    /* エイリアス $all が 1 つ定義されています
                                --- (すべてのプローブです)。*/
prex> list fcns
&debug    tnf_probe_debug      /* デバッグ関数は、使用可能な唯一の関数です。*/
prex> list probes $all
name=inloop enable=off trace=on file=cookie.c line=35 funcs=<no value>
name=factor_end enable=off trace=on file=cookie.c line=72 funcs=<no value>
name=factor_start enable=off trace=on file=cookie.c line=61 funcs=<no value>
name=found_a_factor enable=off trace=on file=cookie.c line=67 funcs=<no value>
name=start enable=off trace=on file=cookie.c line=17 funcs=<no value>
prex>/* 行番号は、5 つのプローブのそれぞれの最終行を示しています。*/
prex> create $factor /factor/   /* "keys" 属性に文字列 "factor" が含まれている
                          プローブと一致する新しいセットを作成します。*/
prex>
prex> list sets
$all      'keys'=/.*/
$factor   'keys'=/factor/
/* "factor" という名前の新しいセットが作成されて、そのセットが表示されています。*/
prex>
  /* セットの一覧 */
prex> list probes $factor
/* この行によって、どのプローブがそのセットと一致しているかがわかります。*/
name=factor_end enable=off trace=on file=cookie.c line=72 funcs=<no value>
name=factor_start enable=off trace=on file=cookie.c line=61 funcs=<no value>
name=found_a_factor enable=off trace=on file=cookie.c line=67 funcs=<no value>
```

```
prex> list probes $all    /* 有効なプローブがあるかどうか検査します。*/
name=inloop enable=off trace=on file=cookie.c line=35 funcs=<no value>
name=factor_end enable=off trace=on file=cookie.c line=72 funcs=<no value>
name=factor_start enable=off trace=on file=cookie.c line=61 funcs=<no value>
name=found_a_factor enable=off trace=on file=cookie.c line=67 funcs=<no value>
name=start enable=off trace=on file=cookie.c line=17 funcs=<no value>
prex>              /* 有効なプローブはありませんが、
                                  どのプローブもトレースはオンになっています。*/

prex> enable $all             /* すべてのプローブを有効にします。*/
prex> list probes $all        /* 有効なプローブがあるかどうか、再度検査します。*/
name=inloop enable=on trace=on file=cookie.c line=35 funcs=<no value>
name=factor_end enable=on trace=on file=cookie.c line=72 funcs=<no value>
name=factor_start enable=on trace=on file=cookie.c line=61 funcs=<no value>
name=found_a_factor enable=on trace=on file=cookie.c line=67 funcs=<no value>
name=start enable=on trace=on file=cookie.c line=17 funcs=<no value
prex> list values name              /* プローブの名前を調べます。*/
name =
    factor_end
    factor_start
    found_a_factor
    inloop
    start
prex> list values /.*/          /* 定義済みの属性とその値をすべて表示します。*/
enable =              /* 固有の属性だけを表示します。*/
    on
file =
    cookie.c
funcs =

keys =
    cookie
    factor
    find_factor
    loop
    main
line =
    17
    35
    61
    67
    72
name =
    factor_end
    factor_start
    found_a_factor
    inloop
    start
object =
    cookie
slots =
    factor
    input_number
    loop_count
    searching_for
    total_iterations
sunw%debug =          /* ユーザー定義のマクロ sunw%debug も表示されています。*/
    in     /* このマクロは、cookie.c の 17 行目で定義されています。*/
    loop
    main
```

```
    starting
    the
trace =
    on
prex> list values object
object =
    cookie
prex> connect &debug name=inloop
prex> list /.*/ probes $all         /* すべてのプローブの情報をすべて表示します。*/
enable=on trace=on object=cookie funcs=<no value> name=inloop slots=loop_count
total_iterations keys=cookie main loop file=cookie.c line=35 sunw%debug=in the loop
enable=on trace=on object=cookie funcs=<no value> name=factor_end slots=<no value>
keys=factor file=cookie.c line=72
enable=on trace=on object=cookie funcs=<no value> name=factor_start slots=input_number
keys=factor file=cookie.c line=61
enable=on trace=on object=cookie funcs=<no value> name=found_a_factor slots=searching_for
factor keys=cookie find_factor file=cookie.c line=67
enable=on trace=on object=cookie funcs=<no value> name=start slots=<no value>
keys=cookie
main file=cookie.c line=17 sunw%debug=starting main
prex> continue
give me a COOKIE! loop
 /* ループカウントの例 */
probe inloop; sunw%debug ``in the loop''; loop_count=0; total_iterations=0;
probe inloop; sunw%debug ``in the loop''; loop_count=1; total_iterations=1;
probe inloop; sunw%debug ``in the loop''; loop_count=2; total_iterations=2;
probe inloop; sunw%debug ``in the loop''; loop_count=3; total_iterations=3;
probe inloop; sunw%debug ``in the loop''; loop_count=4; total_iterations=4;
give me a COOKIE! factor
number you want factored? 25
 factors of 25 = 5 5
give me a COOKIE! factor
number you want factored? 43645729
 factors of 43645729 = 43645729
give me a COOKIE! ^C Target process stopped
Type ``continue'' to resume the target, ``help'' for help...
prex> continue
give me a COOKIE! biscuit
not a biscuit, give me a COOKIE!  cookie
not a cookie, give me a COOKIE! COOKIE
thanks!
prex: target process finished
```

トレースファイルの読み取り

prex で作成したバイナリのトレースファイルには、ユーザーが選択した prex コマンドによって判定された情報が含まれています (7ページの「prex の実行」を参照)。

デフォルトでは、トレースファイルは /$TMPDIR/trace-*pid* に作成されます。この *pid* は、ターゲットのプログラムのプロセス ID です。$TMPDIR が設定されていない場合は、/tmp/trace-*pid* にファイルが作成されます。トレースファイルの場所は、prex コマンドに –o オプションを付けて変更できます (詳細については、6ページの「prex のコマンド行オプション」を参照してください)。

トレースファイルが一杯になると、新しいイベントによって古いイベントが上書きされます。トレースファイルのデフォルトサイズは、4 MB です。このサイズは、prex の –s オプションで変更することができます。

注 - プログラムのトレース結果がファイルに書き込まれた後で、そのプログラムが存在する間にはそのトレースファイルを削除したり、トレースファイルの名前を変更することはできません。トレースファイルをターゲットプログラムから切り離し、別の名前に変更したトレースファイルを再接続しても、新しい名前は無視されます。

読み取り可能な形式へのバイナリファイルの変換

バイナリのトレースファイルを ASCII ファイルに変換するには、tnfdump コマンドとバイナリのトレースファイルの名前を使用します。デフォルトで tnfdump の出力は stdout に送られるため、その出力をファイルにリダイレクトする必要があります。

```
$ tnfdump filename > newfile
```

tnfdump に –r オプションを付けると、詳細な (生の) TNF 出力が得られます。この詳細な TNF 出力を解釈するには、この章の説明範囲を超えた TNF の知識が必要となります。

tnfdump ファイルの読み取り

17ページの「prex セッションのサンプル」で説明した prex cookie コマンドの出力を以下に示します。tnfdump ファイルの出力は非常に幅が広いので、出力を表示する場合は幅の広いウィンドウを開いてください。

```
probe  tnf_name: ''start'' tnf_string: ''keys cookie main;file cookie.c;line 17;sunw%debug
starting main''
probe  tnf_name: ''factor_start'' tnf_string: ''keys factor;file cookie.c;line 61;''
probe  tnf_name: ''found_a_factor'' tnf_string: ''keys cookie find_factor;file cookie.c;line
67;''
probe  tnf_name: ''factor_end'' tnf_string: ''keys factor;file cookie.c;line 72;''
---------------- ---------------- ----- ----- ---------- --- -------------------------
------------------------
    Elapsed (ms)  Delta (ms)     PID  LWPID  TID  CPU Probe Name        Data / Description . . .
---------------- ---------------- ----- ----- ---------- --- -------------------------
------------------------
  0.000000 0.000000 5354 1 0 - start
 4551.625000 4551.625000 5354 1 0 - factor_start input_number: 25
 4571.278000 19.653000 5354 1 0 - found_a_factor searching_for: 25 factor: 5
 4571.543000 0.265000 5354 1 0 - found_a_factor searching_for: 5 factor: 5
 4571.732000 0.189000 5354 1 0 - factor_end
  23151.434000 18579.702000 5354 1 0 - factor_start input_number: 101247
```

```
  23151.509000 0.075000 5354 1 0 - found_a_factor searching_for: 101247 factor: 3
  23228.090000 76.581000 5354 1 0 - found_a_factor searching_for: 33749 factor: 33749
  23228.250000 0.160000 5354 1 0 - factor_end
  89041.868000 65813.618000 5354 1 0 - factor_start input_number: -1690908149
  89041.920000 0.052000 5354 1 0 - factor_end
 108271.852000 19229.932000 5354 1 0 - factor_start input_number: 43645729
 208857.756000  100585.904000 5354 1 0 - found_a_factor searching_for: 43645729 factor:
43645729
 208857.960000 0.204000 5354 1 0 - factor_end
 334511.548000  125653.588000 5354 1 0 - factor_start input_number: 12
 334511.618000 0.070000 5354 1 0 - found_a_factor searching_for: 12 factor: 2
 334511.689000 0.071000 5354 1 0 - found_a_factor searching_for: 6 factor: 2
 334511.750000 0.061000 5354 1 0 - found_a_factor searching_for: 3 factor: 3
 334511.808000 0.058000 5354 1 0 - factor_end
```

tnfdump の表示を見て、factor_end の時間から factor_start の時間を引くと、因数を見つけるまでに要した時間がわかります。43645729 の因数分解には、208857.960000 - 108271.852000 (つまり、100586.11) ミリ秒間かかっています。12 の因数分解には、334511.808000 - 334511.548000 (つまり、0.260000) ミリ秒間かかっています。

注 - どのハードウェアプラットフォームでも、結果はナノ秒の精度で報告されます。ただし、結果の正確度は、使用しているハードウェアプラットフォームによって異なります。

カーネルのトレース

Solaris 2.5 リリースから、SunOS のカーネルには TNF プローブがいくつか組み込まれています。このプローブは、システムコール、スレッド状態の移り変わり、ページフォルト、スワッピング、入出力などのカーネルイベントを記録します。ユーザーは、これらのプローブを使用して、アプリケーション処理実行中におけるカーネルの活動状況を詳細にトレースできます。無効になっているプローブは、実行中のカーネルのパフォーマンスにはほとんど影響を与えません。

カーネルをトレースする方法は、いくつか相違点はあるものの、ユーザーレベルのプロセスをトレースする方法と似ています。以降の説明を理解するためには、あらかじめ 7ページの「prex の実行」と 19ページの「トレースファイルの読み取り」をお読みください。カーネルをトレースするにはスーパーユーザーの特権が必要です。

カーネルのプローブは、prex ユーティリティを使用して制御します。ユーザーは、カーネルのトレースの設定と管理を行うコマンドに加え、プローブの表示と操作を行う標準の prex コマンドも使用することができます。

カーネルのプローブは、トレース記録をカーネルトレースバッファーに書き込みます。このトレース記録を後で処理するためには、tnfxtract ユーティリティを使用してそのバッファーを TNF ファイルにコピーする必要があります。

カーネルトレースファイルを検証するには、tnfdump ユーティリティを使用して行います。これは、ユーザーレベルのトレースファイルを検証する場合とまったく同じ方法です。

カーネルのトレースを行う場合の標準的な手順を以下に示します。

1. スーパーユーザーになります (su)。
2. 必要なサイズのカーネルトレースバッファーを割り当てます (prex)。
3. トレースするプローブを選択して、そのプローブを有効にします (prex)。
4. カーネルのトレースをオンにします (prex)。
5. アプリケーションを起動します。
6. カーネルのトレースをオフにします (prex)。
7. カーネルトレースバッファーからデータを抽出します (tnfxtract)。
8. すべてのプローブを無効にします (prex)。
9. カーネルトレースバッファーの割り当てを解除します (prex)。
10. トレースファイルを検証します (tnfdump)。

これらの手順は、2 つのシェルウィンドウを使用すると簡単に実行できます。一方のウィンドウで対話型の prex セッションを実行して、もう一方のウィンドウでアプリケーションと tnfxtract を実行します。32ページの「カーネルをトレースするためのシェルスクリプト」に示した ktrace シェルスクリプトを使用すると、上記の作業をさらに簡単にすることができます。

カーネルのトレース制御 (prex)

–k フラグを付けてカーネルに対して、prex を起動します (必ず root として実行してください)。prex が正常にカーネルに接続されると、prex のコマンドプロンプトが表示されます。

```
# prex -k
prex>
```

注 - カーネルには、一度に 1 つの prex セッションしか接続できません。

バッファーの割り当て

カーネルのトレースを行うための最初の手順は、カーネルトレースバッファーの割り当てです。トレースバッファーが一杯になると、古いデータの上に新しいデータが書き込まれます。

バッファーのデフォルトサイズは 384 KB ですが、バッファーを割り当てるときにこのサイズを変更することができます。このバッファーは物理メモリーとカーネルの仮想メモリーの両方を使用するため、バッファーサイズを決める際には、トレース対象のシステムにその影響が及ぶことを考慮してください。RAM の容量が大きいほど、トレース対象の処理に対する影響は少なくなり、より大きな容量のバッファーを割り当てることができます。

prex で buffer コマンドを使用して、バッファーを割り当てます。たとえば、512 KB のバッファーを割り当てる場合は、以下のように行います。

```
prex> buffer          #バッファーが割り当てられているか?
No trace buffer allocated
prex> buffer alloc 512k# 512 KB のバッファーを割り当てる
Buffer of size 524288 bytes allocated
prex>
```

注 - バッファーの最小サイズは 128 KB です。128 KB よりも小さいバッファーを割り当てようとした場合には、prex によって 128 KB のバッファーが割り当てられます。

プローブの選択と有効化

カーネルのプローブとその属性は、prex の標準リストコマンドを使用して一覧表示します。たとえば、pagein という名前のプローブの name 属性と keys 属性を表示するには、以下のコマンドを実行します。

```
prex> list name=pagein keys
name=pagein keys=vm pageio io
```

io グループに含まれるすべてのプローブの name 属性と keys 属性を表示するには、以下のコマンドを実行します。

```
prex> list name keys probes io
name=biodone keys=io blockio
name=physio_start keys=io rawio
name=pagein keys=vm pageio io
name=pageout keys=vm pageio io
name=physio_end keys=io rawio
name=strategy keys=io blockio
prex>
```

次に、必要なプローブをトレースして、そのプローブを有効にします。たとえば、thread、vm、io を指定するキーを持ったプローブをすべて選択して、そのプローブをトレースできるようにするには、以下のコマンドを実行します。

```
prex> trace thread vm io      #トレース関数をプローブに接続します。
prex> enable thread vm io     #プローブを有効にします。
prex>
```

注 - ユーザーレベルのプローブとは違って、カーネルのプローブは、prex をカーネルに接続しても自動的にはトレースされません。上記のように、明示的に trace コマンドと enable コマンドを実行する必要があります。

これで、関連するプローブはトレースできる状態になりますが、カーネルのトレースは広域的には許可されていないので、トレース記録は書き込まれません。

プロセスのフィルタリング

システムの動作をすべてトレースする場合 (たとえば、ビジーサーバー上など) には、25ページの「トレースの有効と無効」に進んでください。これは、カーネルトレースの最も一般的な使い方であり、また、通常は最も有用な使い方です。

ただし、選択したプロセスに関するトレースデータの生成を制限するオプションも用意されています。このオプションを使用すると、収集して分析しなければならないデータの量を減らすことができます。

prex では、2 つの抽出条件によって、プロセスのフィルタリングを実現しています。

- プロセスのフィルタセットは、トレースを有効にするプロセスのプロセス識別子 (PID) のリストです。このフィルタセットに含まれていないプロセス (に属するスレッド) は、トレースデータの書き込みを行いません。デフォルトのフィルタセットは空です。
- プロセスのフィルタモードは、カーネルにおけるプロセスのフィルタリングの有効/無効を選択するための広域フラグです。デフォルトでは、プロセスのフィルタリングは無効にされています。つまり、すべてのプロセス (とスレッド) がトレース記録を書き込みます。フィルタリングを有効にすると、フィルタセット内のプロセス (に属するスレッド) だけがトレース記録を書き込みます。

プロセスのフィルタリングを制御するには、prex pfilter コマンドを使用します。

```
prex> pfilter                 # フィルタリングは行われているか?
Process filtering is off
Process filter set is empty.
prex> pfilter add 408         # PID 408 をフィルタセットに追加
prex> pfilter
Process filtering is off
Process filter set is {408}
prex> pfilter on              # プロセスのフィルタリングを有効にする
prex>
```

注 - システムスレッド (割り込みスレッドなど) は、プロセス 0 に属するスレッドとして扱われます。

トレースの有効と無効

トレースを開始するために行う最後の手順は、広域的にカーネルのトレースを有効にすることです。この手順を実行すると、有効にしたプローブが検出されるたびに、そのプローブはカーネルトレースバッファーに記録を書き込みます。

```
prex> ktrace              # トレースの状態を検査
Tracing is off
prex> ktrace on           # カーネルのトレースを有効にする
prex>
```

アプリケーションのカーネル動作をトレース (同時に、ユーザーレベルのトレースも開始) するには、この時点でそのアプリケーションを起動します。アプリケーションが終了したとき、またはアプリケーションの動作をサンプリングできるだけの十分なトレースデータを得たときには、広域的にカーネルのトレースを無効にします。

```
prex> ktrace off          # カーネルのトレースを無効にする
prex>
```

トレースバッファーは、カーネル内にそのまま残っています。`tnfxtract` を使用して、そのトレースバッファーを TNF ファイルにコピーしてください。この操作の手順については、26ページの「カーネルトレースデータの抽出 (tnfxtract)」を参照してください。

カーネルのトレースのリセット

トレースバッファーを TNF ファイルにコピーしたら、すべてのプローブを無効にして、バッファーの割り当てを解除することによって、カーネルのトレースをリセットします。リセットすることによって、カーネルのパフォーマンスはトレースを開始する前の状態に戻るため、この操作は重要です。

次のように標準の prex コマンドを使用して、すべてのプローブを無効にし、そのトレースも解除します。

```
prex> disable $all       # すべてのプローブを無効にする
prex> untrace $all       # すべてのプローブのトレースを解除
prex>
```

最後に、prex buffer コマンドを使用して、トレースバッファーの割り当てを解除します。

注意 - トレースバッファーをトレースファイルにすべてコピーしてから、トレースバッファーの割り当てを解除してください。コピーを行う前にトレースバッファーの割り当てを解除すると、検証用に収集したトレースデータがすべて失われてしまいます。

```
prex> buffer dealloc     # バッファーの割り当てを解除

buffer deallocated
prex>
```

バッファーの割り当てを解除したら、prex を終了して、収集したトレースデータを検証できます。

```
prex> quit
#
```

カーネルトレースデータの抽出 (tnfxtract)

tnfxtract ユーティリティを使用して、アクティブなカーネルトレースバッファーのコピーを外部の TNF トレースファイルに作成します。トレースと並行して tnfxtract ユーティリティを実行することもできますが、通常はトレースを無効にしてから実行します。このユーティリティでは、整合性のある TNF データについてのみ、読み取りおよび書き込みを行うことができます。

抽出したカーネルトレースデータを保持するために tnfxtract が使用するファイルの名前を指定してください。このファイルは、上書きされ、カーネルトレースバッファーと同じサイズに縮小されます。たとえば、カーネルトレースバッファーを ktrace.tnf という名前の一時ファイルに抽出するには、以下のコマンドを実行します。

```
# tnfxtract /tmp/ktrace.tnf
# ls -l /tmp/ktrace.tnf
-rw------- 1 root other 524288 Aug 15 16:00 /tmp/ktrace.tnf
#
```

トレースファイルのアクセス権を変更して、すべてのユーザーが読み取れるようにしておくと、スーパーユーザー以外のユーザーでもそのファイルに対して解析ツールを実行できるようになります。

注 - カーネルクラッシュダンプに対して `tnfxtract` を使用することもできます。詳細は、`tnfxtract(1)` を参照してください。

カーネルトレースデータの検証 (`tnfdump`)

`tnfdump` を使用して、カーネル trace-file の時刻順の ASCII ダンプを `stdout` に出力します (この操作は、ユーザーレベルのトレースの場合とまったく同じです)。トレース出力のサイズは非常に大きくなる場合もあるので、通常はページングユーティリティ (`more` など) を使用して `tnfdump` を実行するか、または出力ファイルにリダイレクトします。

```
# tnfdump /tmp/ktrace.tnf | more
```

注 - `tnfdump` には、複数の TNF ファイルを入力として指定できます。複数の TNF ファイルを入力に指定すると、指定した入力ファイル内のイベント記録がすべて時刻順にソートされて出力されます。このため、カーネルトレースファイルと複数のユーザーレベルのトレースファイルを入力として指定することによって、それらのファイルを 1 つに結合することができます。

使用可能なカーネルプローブ (`tnf_probes`)

SunOS のカーネルプローブは、`vmstat`、`iostat`、`sar` などの主に統計を行う Solaris パフォーマンス監視ツールとほぼ同程度の情報を提供します。ただし、プローブの方が上記の監視ツールよりも詳しい情報を提供します。各プローブは、高精度のタイムスタンプ、スレッド (LWP (軽量プロセス)、スレッド、プロセス識別子)、CPU を記録し、イベントに関連したシステムリソース (ディスク、ファイル、CPU など) の識別もします。

ユーザーは、カーネルのトレースとユーザーレベルのトレースを併用して、アプリケーションまたはライブラリのプローブによって記録されたイベントと、カーネルのプローブによって記録されたイベントを相互に関連付けることができます。これにより、コードがカーネルのサービスをどのように使用しているか、また、シス

テムリソースに対してアプリケーションが行う要求と他のクライアントによって行われる要求がどのように相互に作用しているかを詳しく知ることができます。

以下の節では、SunOS のカーネルプローブについて簡単に説明します。イベント記録フィールドについての詳細は、tnf_probes(4) を参照してください。

スレッドプローブ

thread_create プローブは、カーネルスレッドの作成をトレースします。このプローブは、プロセス識別子、カーネルスレッド識別子、新しいスレッドの開始ルーチンのカーネルアドレスを記録します。

thread_exit プローブは、現在のスレッドの終了を記録します。

スレッド状態の移り変わりをトレースするプローブは、マイクロステートプローブと呼ばれます。このプローブは、スレッド状態とスレッド識別子 (オプション) を記録します。スレッド識別子が指定されている場合は、状態変化はそのスレッドに適用されます。スレッド識別子が指定されていない場合は、状態変化は書き込みスレッドに適用されます。

スレッド状態の値には、<sys/msacct.h> で定義されたマイクロステート定数が使用されます。記録されるスレッド状態を表 1–7 に示します。

表 1–7 スレッドマイクロステート定数

状態	説明
user	ユーザーモードで実行中
system	システムモードで実行中
tfault	ユーザーテキスト障害の初期状態
dfault	ユーザーデータ障害の初期状態
trap	その他のトラップの初期状態
user_lock	ユーザーモードのロックを待つための休眠
sleep	その他の理由による休眠

表 1–7　スレッドマイクロステート定数　続く

状態	説明
wait_cpu	CPU 待ち (実行可能)
stopped	停止 (/proc、jobcontrol、lwp_stop)

注 - トレース量を減らすため、カーネルは、システムコールによって暗黙に示されているシステム状態とユーザー状態との間の遷移はトレースしません。この情報を得るには、システムコールプローブを有効にする必要があります。thread キーを使用してプローブを有効にすると、システムコールプローブは自動的に有効になります。

システムコールプローブ

システムコールの入口および出口のプローブは、ユーザーコードによって明示的に要求されたシステム操作を識別します。

syscall_start プローブは、システムコールの開始とそのシステムコール番号を記録します。システムコールの引数を捕捉するのはかなり大きな負荷がかかるので、このプローブはシステムコールの引数は捕捉しません (ユーザーレベルで C ライブラリのエントリポイントにプローブを挿入すると、システムコールの引数についての情報の一部を捕捉できます)。また、この syscall_start プローブは、system 状態に入ったときに現在のスレッドを暗黙的に記録します。

syscall_end プローブは、システムコールの終了、そのシステムコールの 2 つの戻り値、および errno 値を記録します。また、この syscall_end プローブは、user 状態に入ったときに現在のスレッドを暗黙に記録します。

注 - このレベルでのシステムコールの実装は、Solaris のリリースによって異なります。各システムコールに対応する番号がどのリリースでも同じとは考えないでください。

VM プローブ

仮想メモリーサブシステム (VM) プローブは、ページフォルト、ページ入出力、ページデーモン、スワッパーに関する情報を提供します。

ページフォルト

ページフォルトプローブは、仮想アドレスを障害の種類とファイル (vnodes) に関係付けます。

`address_fault` プローブは、アドレス空間障害をトレースして、障害が発生した仮想アドレス、障害の種類、要求されたアクセス権を記録します。

障害の種類およびアクセス権の種類を表す値には、`<vm/seg_enum.h>` で定義された定数が使用されます。障害の種類には、無効なページ (軽い障害)、アクセス権障害、物理的な入出力用のページのロック/ロック解除を行うソフトウェア要求 (softlock と softunlock) があります。アクセス権の種類には、読み取り、書き込み、実行、作成があります。

`major_fault` プローブは、深刻なページフォルトをトレースします。このプローブは、障害を解決する際に必要となる vnode とオフセットを記録します。この 2 つの組み合わせによって、ファイルシステムのページが識別されます。このデータを現在のスレッドの直前の `address_fault` イベントに相互に関連付けて、障害が発生した仮想アドレスを知ることができます。

`anon_private` プローブは、コピー時の書き込み障害をトレースします。

`anon_zero` プローブは、zero-fill 障害をトレースします。

`page_unmap` プローブは、物理ページとファイルシステムページとの間の分離を記録します (たとえば、ページの名前が変更されたときや、ページが削除されたときなど)。

ページ入出力

`pagein` プローブは、ページイン要求の開始をトレースして、vnode、オフセット、ページインのサイズを記録します。

`pageout` プローブは、ページアウト要求の完了をトレースして、ページアウトされたページ数、開放されたページ数、ページアウト後に再要求されたページ数を記録します。

ページデーモン

ページデーモン (ページスティーラ) は、2 つのプローブ pageout_scan_start と pageout_scan_end によってトレースされます。これらのプローブは、検査前に必要な空きページ数、検査されたページ数、検査前と検査後の空きページ数を報告します。検査によってキューに入れられた pageout 要求がすべて完了すれば、さらに多くのページが開放される可能性があります。

スワッパー

以下の 3 つのプローブによって、スワッパーの活動状況がトレースされます。

- swapout_process プローブは、プロセスアドレス空間のスワップアウトをトレースして、プロセス識別子と、開放されたページまたは出力用にキューに入れられたページの総数を記録します。
- swapout_lwp プローブは、LWP のスタックページのスワップアウトをトレースして、LWP の識別情報と出力用にキューに入れられたページ数を記録します。
- swapin_lwp プローブは、LWP のスタックページのスワップインをトレースして、LWP の識別情報と取り込まれたスタックページ数を記録します。

ローカル入出力プローブ

strategy プローブは、カーネルによるローカルブロックデバイス入出力の開始をトレースして、転送に関係するデバイス番号、論理ブロック番号、サイズ、バッファーポインタ、バッファーフラグを記録します。このフラグ値は、<sys/buf.h> で定義されているバッファー状態フラグです。

biodone プローブは、バッファー付き入出力転送の完了、つまりカーネルの biodone(9f) ルーチンの呼び出しをトレースします。このプローブは、転送に関係するデバイス番号、論理ブロック番号、バッファーポインタを記録します。

物理 (生の) 入出力は、physio(9f) の 2 つのプローブ physio_start と physio_end によってトレースされます。これらのプローブは、デバイス番号、オフセット、サイズ、入出力転送の方向を記録します。

その他のプローブ

thread_queue プローブは、スレッドのスケジューリングをトレースします。スケジューリングされているスレッドのスレッド識別子、スレッドが配置されている

ディスパッチキューに関連付けられている CPU、スレッドのディスパッチ優先順位、ディスパッチキュー上にある実行可能スレッドの現在の数を記録します。

カーネルをトレースするためのシェルスクリプト

コード例 **1–3**　ktrace スクリプト

```
#!/bin/sh
#
# ktrace
# カーネルのトレースをリセットし有効にする
# 指定されている場合はコマンドをバックグラウンドで実行する
# 指定されている場合は休眠（スリープ）する
# コマンドの実行完了、タイムアウト、キーボードの割り込みを待機
# （注 : キーボード割り込みによってコマンドは終了する）
# カーネルのトレースを無効にする
# カーネルトレースバッファーを抽出する
# 指定されている場合はカーネルのトレースをリセットする
#

TMPDIR=${TMPDIR:-/tmp}

output_file=$TMPDIR/ktrace.tnf
buffer_size=512k
duration=
events=
command=
do_reset=y
child=
alarm=

# 使用法のメッセージ
usage() {
 echo ````
 echo $1
 echo ``
Usage: ktrace [-o <output_file>] # デフォルトでは /tmp/ktrace.tnf
              [-s <buffer_size>] # デフォルトでは 512k
              -e <events>        # 有効にするカーネルプローブ（キー）
              [-r]                  # カーネルのトレースをリセットしない
                                    # デフォルトでは、コマンド実行後にリセットする
              <cmd> | -t <seconds>

Eg,
 # ktrace -e 'thread vm io' -t 10
 # ktrace -e 'thread' -s 256k myapp ...
``
 exit 1
}
# 失敗時のメッセージ
fail() {
 while [ $# -gt 0 ]
 do
```

(続く)

```
  echo $1
  shift
 done
 echo ``ktrace failed''
 exit 2
}

# カーネルのトレースをリセットする
ktrace_reset() {
 if [ $1 = ``y'' ]; then
  echo ``Resetting kernel tracing''
  prex -k >/dev/null 2>&1 <<-EOF
   ktrace off
   untrace /.*/
   disable /.*/
   buffer dealloc
   EOF
  test $? -ne 0 && fail ``Could not reset kernel tracing'' \
   ```su root' and retry''
 fi
}

# カーネルのトレースを有効にする
ktrace_on() {
 echo ``Enabling kernel tracing''
 prex -k >/dev/null 2>&1 <<-EOF
  buffer alloc $buffer_size
  trace $events
  enable $events
  ktrace on
  EOF
 test $? -ne 0 && fail ``Could not enable kernel tracing'' \
  ``Check syntax of `-e' argument'' \
  ``Check buffer size is not too high''
}

# カーネルのトレースを無効にする
ktrace_off() {
 prex -k >/dev/null 2>&1 <<-EOF
  ktrace off
  EOF
 test $? -ne 0 && fail ``Could not disable kernel tracing''
 echo ``Kernel tracing disabled''
}

# カーネルトレースバッファーを抽出する
ktrace_xtract() {
 echo ``Extracting kernel trace buffer''
 tnfxtract $output_file || fail ``Could not extract kernel trace
 buffer''
 ls -l $output_file
}
```

(続く)
```

```
# コマンドを実行、休眠、またはコマンド実行完了を待機する
run_command() {
 trap 'interrupt' 0 1 2 3 15
 if [ ''$command'' ]; then
  $command &
  child=$!
  echo '''$command' pid is $child''
 fi

 if [ ''$duration'' ]; then
  sleep $duration &
  alarm=$!
  wait $alarm
  # XXX test -z ''$child'' || kill -15 $child
 else
  wait $child
 fi
 trap 0 1 2 3 15
}

# キーボード割り込み
interrupt() {
 test -z ''$alarm'' || kill -15 $alarm
 test -z ''$child'' || kill -15 $child
}

# オプションを読み取る
while getopts o:s:t:e:r opt
do
 case $opt in
  o) output_file=''$OPTARG'';;
  s) buffer_size=''$OPTARG'';;
  t) duration=''$OPTARG'';;
  e) events=''$OPTARG'';;
  r) do_reset=''n'';;
  \?) usage;;
 esac
done

shift `expr $OPTIND - 1`

# 実行するコマンドを取得する
test $# -gt 0 && command=''$*''

# オプションを検査
test -z ''$events'' && usage ''No kernel events specified''
test -z ''$command'' && test -z ''$duration'' && \
usage ''No command or time duration specified''

# 検査を実行する
ktrace_reset y          # カーネルのトレースをリセット
ktrace_on               # カーネルのトレースを有効に
```

(続く)

```
run_command               # コマンド実行、待機、または休眠
ktrace_off                # カーネルのトレースを無効に
ktrace_xtract             # バッファを抽出
ktrace_reset $do_reset    # カーネルのトレースを再度リセット
exit 0
```

上級者向きのトピック

プローブポイントの挿入

以下のような場合に、プローブポイントをコードに挿入します。

- プログラムの変数値をトレースする。
- エクスポートされたインタフェースからは得ることができない内部状態の情報を提供する。これは、デバッグやパフォーマンス分析を行う場合に役立ちます。

  たとえば、プローブポイントを使用して、C++ の private クラスに隠されているパフォーマンス関連の情報を表示したり、ハッシュテーブル内の衝突率などの処理データを表示できます。ハッシュテーブルのコード中にプローブポイントを配置しておくと、衝突が検出されるたびにそのプローブポイントはトレースファイルに書き込みを行うことができます。

  プローブを挿入するためのインタフェースは、`TNF_PROBE_0` から `TNF_PROBE_5` までの `TNF_PROBE` マクロによって定義されます。0 ～ 5 の数字は、マクロによってトレースされる変数の数です。

  これらのマクロを使用して、コードの任意の場所にプローブポイントを挿入することによって、変数の値を取得したり、プログラムの実行をトレースすることができます。`libtnfprobe` ライブラリでは標準のスカラー型 (int、long、float など) が定義されていますが、`TNF_DECLARE_RECORD` マクロと `TNF_DEFINE_RECORD` マクロを使用すると、さらに複雑な構造を定義することもできます。40ページの「プローブポイントのユーザー定義型」を参照してください。

TNF_PROBE マクロの使い方

単純な例として、TNF_PROBE_0 を以下に示します。このマクロには、引数の型は指定しません。

```
TNF_PROBE_0 (name, keys, detail);
```

各変数について説明します。

- *name* - プローブの名前。この名前は、ANSI C の識別子の構文規則にすべて従います。*name* は、使用したときに宣言されるので、あらかじめ別に宣言する必要はありません。この宣言はブロックスコープ宣言なので、プログラムの名前空間には影響を与えません。
- *keys* - プローブが属するグループのリスト。このリスト中には複数のキーワードが含まれており、各キーワードは空白文字で区切られています。このリストの中では、セミコロン (;)、等号 (=)、単一引用符 (') は使用できません。いずれかのグループが有効になると、そのプローブポイントも有効になります。*keys* を変数にすることはできません。*keys* はインライン文字列にしてください。
- *detail* - 独自の属性と値を定義する手段を提供します。*detail* 文字列には、複数の組の属性と値が含まれており、各組はセミコロンで区切られています。ただし、この値は省略可能です。最初のワード (空白文字まで) は属性、残りの文字列 (セミコロンまで) は値とみなされます。区切り文字のセミコロンの前後には、スペースを入れることができます。*detail* 文中では単一引用符と等号は使用できません。

  属性名の前にベンダー固有の記号と % 記号を付けて、名前の衝突を避けてください。以下の例では、
  sunw%debug、comX%exception、comY%func_entry、comY%color という 4 つの属性が定義されています。prex は空白文字の値をトークン化するので、複数の語から成る値の場合はいずれかの語で照合できますが、文字列全体で照合することはできません。たとえば、以下のような例が考えられます。

```
sunw%debug entering function A;
   comX%exception no file;
   comY%func_entry;
   comY%color red blue
```

上記のコマンドで照合される値を表 1–8 に示します。

表 1–8　ユーザー定義属性の例

属性	値	prex が照合する値
sunw%debug	entering function A	entering または function または A
comX%exception	no file	no または file
comY%func_entry		/.*/ (正規表現)
comY%color	red blue	red または blue

libtnfprobe は、ベンダー記号が前に付いていない属性名をすべて予約します (つまり、名前に % 記号が含まれていない属性をすべて予約します)。15ページの「C プログラムのサンプル」の cookie.c では、以下の TNF_PROBE_0 を使用しています。

```
TNF_PROBE_0(start, ''cookie main'', ''sunw%debug starting main'');
```

注 - プリプロセッサオプションの –DNPROBE (cc(1) のマニュアルページを参照) を使用してコンパイルする場合、またはプリプロセッサ制御文 #include <tnf/probe.h> の前に #define NPROBE を置いてコンパイルする場合には、プローブポイントと TNF の型拡張コードがプログラムにコンパイルされないようにしてください。

TNF_PROBE_1 ～ TNF_PROBE_5

引数の名前に含まれている数字 1 ～ 5 は、プローブポイントに指定する変数の数を表します。たとえば、TNF_PROBE_1 の構文は次のようになっています。

```
TNF_PROBE_1(name, keys, detail,
arg_type_1, arg_name_1, arg_value_1);
```

また、TNF_PROBE_5 の構文は次のようになっています。

```
TNF_PROBE_5(name, keys, detail,
arg_type_1, arg_name_1, arg_value_1
arg_type_2, arg_name_2, arg_value_2
arg_type_3, arg_name_3, arg_value_3
arg_type_4, arg_name_4, arg_value_4
```

arg_type_5, arg_name_5, arg_value_5);

以下、各引数について説明します。

- *arg_type_n* − *n* 番目の引数の型です。*n* には 1 ～ 5 の数字が入ります。定義済みの型を表 1–9 に示します。ユーザー独自の型の定義については、40ページの「プローブポイントのユーザー定義型」 を参照してください。
- *arg_name_n* − *n* 番目の引数に与える名前です。 ANSI C の識別子に関する規則に従ってください。また、引数の名前は、引用符で囲まないでください。(9ページの「属性」で取り上げた `slots` 属性には、この名前の文字列の場合が含まれています。)
- *arg_value_n* − トレースファイルに含まれている値に対して評価される式です。*value_n* に含まれている任意の変数に対して読み取り (アクセス) を行うことができます。マルチスレッドのプログラムで、読み取りを禁止する必要があるデータが *value_n* に含まれている場合には、マクロ `TNF_PROBE_n` の前後にロックを配置してください。

表 1–9 定義済みの型

型	対応する **C** 言語の型と意味
`tnf_long`	int、long
`tnf_ulong`	unsigned int、unsigned long
`tnf_longlong`	long long (コンパイラに実装されている場合)
`tnf_ulonglong`	unsigned long long (コンパイラに実装されている場合)
`tnf_float`	float
`tnf_double`	double
`tnf_string`	char *
`tnf_opaque`	void *

たとえば、15ページの「C プログラムのサンプル」の `cookie.c` では、以下のように `TNF_PROBE_2` を使用しています。

```
TNF_PROBE_2(inloop, "cookie main loop","sunw%debug in the loop",
            tnf_long, loop_count,   i,
            tnf_long, total_iterations,  sum);
```

cookie.c のマクロ定義の一部について、表 1–10 で説明します。

表 1–10 cookie.c の TNF マクロ定義

TNF_PROBE_0 (	引数の型が指定されていないプローブ
start,	プローブの名前
"cookie main",	プローブが cookie と main (keys 属性の値) に属しているグループのリスト
"sunw%debug starting main");	ユーザー定義の属性 = sunw%debug、値 = starting main (デバッグプローブ関数で使用されます)
TNF_PROBE_2 (	2 つの変数を持ったプローブ
inloop,	プローブの名前
"cookie main loop",	keys - cookie、main、loop
"sunw%debug in the loop",	デバッグプローブ関数の値
tnf_long,	最初の変数の型
loop_count,	最初の変数の名前 (slots 属性の値)
i,	最初の変数
tnf_long,	2 番目の変数の型
total_iterations,	2 番目の変数の名前 (slots 属性の値)
sum);	2 番目の変数
"");	

例 ー 関数の所要時間の測定

コード例 1–4 では、関数の入口と出口にプローブポイントを配置して、関数内で費やされる時間を測定しています。関数の入口に配置されたプローブは、その関数への引数も記録します。

prex は、トレースが許可されているプローブポイントをプログラム実行時に検出すると、トレースファイルに記録を書き込みます。各プローブポイントは、検出された時刻を記録し、ファイル名、行番号、name、keys などのプローブポイントの詳細情報が含まれたタグ記録を参照します。これらのタグの記録は、トレースファイルに 1 度だけ書き込まれ、上書きされることはありません。

以下のコード例において、最初のプローブポイント work_args は、そのプローブポイントの 2 つの変数値 (state と message) も記録します。

コード例 1–4 関数の入口と出口に配置されたプローブポイント

```
#include <tnf/probe.h>

int
work(int state, char *message)
{
    TNF_PROBE_2(work_start, ``work_module work''
                ``sunw%debug in function work'',
                tnf_long, int_input, state,
                tnf_string, string_input, message);
   ...
    TNF_PROBE_0(work_end, ``work_module work'', ````);
}
```

プローブポイントのユーザー定義型

プログラムの構造体をトレースするには、TNF_DECLARE_RECORD マクロと TNF_DEFINE_RECORD_n マクロを使用して新しい型を定義します。これらのマクロは、プローブポイントに渡される型を拡張するためのコンパイル時インタフェースの一部です。

```
TNF_DECLARE_RECORD(c_type, tnf_type);


TNF_DEFINE_RECORD_1(c_type, tnf_type,
                    tnf_member_type_1,
                    tnf_member_name_1)
TNF_DEFINE_RECORD_2(c_type, tnf_type,
                    tnf_member_type_1,
                    tnf_member_name_1,
                    tnf_member_type_2,
                    tnf_member_name_2)
TNF_DEFINE_RECORD_3(c_type, tnf_type,
                    tnf_member_type_1,
                    tnf_member_name_1,
```

```
                        tnf_member_type_2,
                        tnf_member_name_2,
                        tnf_member_type_3,
                        tnf_member_name_3)
TNF_DEFINE_RECORD_4 (c_type, tnf_type,
                        tnf_member_type_1,
                        tnf_member_name_1,
                        tnf_member_type_2,
                        tnf_member_name_2,
                        tnf_member_type_3,
                        tnf_member_name_3,
                        tnf_member_type_4,
                        tnf_member_name_4)
TNF_DEFINE_RECORD_5 (c_type, tnf_type,
                        tnf_member_type_1,
                        tnf_member_name_1,
                        tnf_member_type_2,
                        tnf_member_name_2,
                        tnf_member_type_3,
                        tnf_member_name_3,
                        tnf_member_type_4,
                        tnf_member_name_4,
                        tnf_member_type_5,
                        tnf_member_name_5)
```

TNF_DECLARE_RECORD と TNF_DEFINE_RECORD は、新しく定義する型ごとに 1 つだけ作成してください。TNF_DECLARE_RECORD は、TNF_DEFINE_RECORD よりも前に置く必要があります。定義されている tnf_type を複数のソースファイルで使用する必要がある場合は、その複数のファイルで共有するヘッダーファイル中に TNF_DECLARE_RECORD を宣言することができます。TNF_DEFINE_RECORD は、いずれか 1 つのソースファイルだけに作成する必要があります。

TNF_DEFINE_RECORD マクロインタフェースは、1 つの関数といくつかのデータ構造体を定義します。したがってこのインタフェースは、関数の内部ではなく、ソースファイル (.c ファイルまたは .cc ファイル) のファイルスコープ (有効範囲) 内で使用してください。

注 - TNF_DEFINE_RECORD 文の後にセミコロンを置かないでください。セミコロンを置くと、コンパイル時に警告が出力されます。

以下、各変数について説明します。

- *c_type* − 新しい tnf_type を作成するためのテンプレートです。C 構造体のすべての要素を、定義する TNF 型で使用する必要はありません。*c_type* は、C の構造体型にしてください。
- *tnf_type* − 新しく作成した型に与えられる名前です。このインタフェースは、tnf_type を接頭辞とした名前空間を使用します。したがって、ライブラリ

が xxx_type という新しい型を定義した場合には、そのライブラリは、xxx_type を接頭辞に使用して別のシンボルを定義することはできません。

型の名前空間の管理方法は、ライブラリ内のその他の名前空間の管理方法と同じです。つまり、ライブラリ内の型以外のシンボルが使用している固有の接頭辞を新しい TNF 型の名前の接頭辞としてください。これによって、新しい TNF 型が定義されている複数のライブラリをリンクしても、名前空間が衝突することはなくなります。

たとえば、libpalloc.so というライブラリが名前の接頭辞として pal を使用してシンボルを定義している場合には、この libpalloc.so は、新しく定義する TNF 型の名前にも pal を接頭辞として使用する必要があります。

- *tnf_member_type_n* − C 構造体の *n* 番目のメンバーの TNF 型です。
- *tnf_member_name_n* − C 構造体の *n* 番目のメンバーの名前です。

例 − TNF 型の定義

新しい TNF 型の定義方法と、プローブ内でのその TNF 型の使い方をコード例 1–5 に示します。

このコード例 1–5 は、すべてのシンボルの接頭辞に pal を使用している架空のライブラリ libpalloc.so の一部であると仮定しています。

コード例 **1–5**　新しい TNF 型の定義

```
#include <tnf/probe.h>

typedef struct pal_header {
        long    size;
        char * descriptor;
        struct pal_header *next;
} pal_header_t;

TNF_DECLARE_RECORD(pal_header_t, pal_tnf_header);
TNF_DEFINE_RECORD_2(pal_header_t, pal_tnf_header,
                        tnf_long,   size,
                        tnf_string, descriptor)

/*
 * 注 : 接頭辞 pal_tnf_header が付いている名前空間は、
 *      このクライアントではもう使用できない。
 */

void
pal_free(pal_header_t *header_p)
{
        int state;

        TNF_PROBE_2(pal_free_start, ``palloc pal_free'',
```

```
                ``sunw%debug entering pal_free'',
                tnf_long,       state_var, state,
                pal_tnf_header, header_var, header_p);
        . . .
}
```

next フィールドを使用して自分自身を指し示す構造体 (リンクされたリスト) 中などで、tnf_type を再帰的または相互再帰的に定義することができます。

このような構造体が TNF_PROBE に渡されると、リンクされたリストの全体がトレースファイルに記録されます (next フィールドが NULL になるまで続きます)。ただし、そのリストが循環リストの場合には、処理が無限ループに入ります。この無限ループを中断するには、tnf_type から next フィールドを削除するか、または next メンバーの型を tnf_opaque として定義します。

パフォーマンスについて

頻繁に使用される mutex ロックのような、頻繁に通過するコード内のセクションには、プローブポイントを配置しないでください。

SPARCStation10 の場合、各プローブが使用するワーキングセットメモリーは約 30 ワード (データが 10 ワード、テキストが 20 ワード)、無効な各プローブで費やされる時間は約 200 ns (ナノ秒) です。アプリケーションのパフォーマンスは、配置するプローブの数を調整することによって改善できます。

プローブポイントを配置したライブラリを出荷する場合は、ベンチマークを実行して、パフォーマンスが著しく低下していないことを確認してください。パフォーマンスを改善するにはプローブの数を減らすか、プローブの配置を変更してください。

/proc

dbx、truss、および prex は、/proc を使用してターゲットプロセスを制御します。/proc では、1 つのクライアントだけが安全にターゲットプロセスを制御できます。このため、dbx や prex のようなユーティリティを同じターゲットプログラム上で同時に実行することはできません。dbx または truss が実行されているターゲットプログラム上で prex を実行しようとすると、prex は「cannot attach to target (ターゲットに接続できません)」というメッセージを表示します。

ただし、以下の手順によって prex と dbx を交互に実行できます。

1. prex を起動します。

2. プローブポイントの状態を設定します。
3. quit suspend コマンドを実行します。
4. dbx を起動します。
5. 中断されているプログラムに接続します。

   ターゲットは、prex と dbx 以外のコードは実行しません。

   ターゲットに SIGSTOP シグナルを送ってそのターゲットを一時停止させてから、"quit resume" を prex に入力することもできます。これを行なった場合には、停止されたプロセス上で dbx を呼び出してから、SIGCONT シグナルを送る必要があります (そうしない場合、dbx がハングします)。

dlopen()、dlclose()、履歴

dlopen(3X) によって取り込まれた共有オブジェクト内のプローブは、prex のコマンド履歴に従って自動的に設定されます。dlclose(3X) によって共有オブジェクトが削除されると、prex はターゲットプログラム内のプローブの情報をリフレッシュします。これは、dlopen と dlclose に必要な作業がほかにもあるということを意味しています。したがって、dlopen と dlclose の所要時間は多少長くなります。

この機能を必要とせず、また dlopen と dlclose の処理に影響を与えたくない場合は、ターゲットから prex を分離してください。

シグナル

prex は、ターゲットプログラムに直接送られるシグナルを妨げません。ただし、prex は Control-c (SIGINT) や Control-z (SIGSTOP) などの端末から生成されたシグナルをすべて受信しますが、それをターゲットプログラムには転送しません。

ターゲットプログラムにシグナルを送るには、シェルから kill(1) コマンドを実行してください。

イベント書き込み操作の障害

トレースファイルにイベントを書き込む際には、システムコール障害などの障害がいくつか発生する可能性があります。これらの障害が発生すると、ターゲットプロセス内で障害コードが設定されます。ターゲットプロセスは正常に実行を継続しますが、トレース記録は書き込まれません。

Control-c を prex に入力して prex プロンプトを表示させると、prex はターゲット内の障害コードを検査して、トレース障害が発生したかどうかを通知します。

ターゲットによる fork() または exec() の実行

プログラムが fork() を実行した場合には、子プロセスが検出したプローブはすべて同じトレースファイルに記録されます。イベントにはプロセス ID 付きで記録されるので、そのイベントがどのプロセスから発生したのかを判定できます。

マルチスレッドのプログラムで、あるスレッドが fork を行なったときに他のスレッドが実行されていると、競合状態が発生することがあります。トレースファイルが破損しないようにするには、fork を実行するときに他のスレッドが停止しているようにするか、または fork1(2) を使用してください。

ターゲットプログラム自身 (ターゲットプログラムがフォークした子プロセスではない) が exec(2) を実行した場合には、prex がターゲットから分離して終了します。ユーザーは、以下のコマンドを使用して、prex を再接続することができます。

```
$ prex -p pid
```

第 2 章

字句解析

lex(1) ソフトウェアツールを使用して、テキスト処理、コード暗号化、コンパイラ書き込みにおける問題を解決できます。たとえばテキスト処理では、ワードのスペルを検査してエラーを検出できます。コード暗号化では、特定の文字パターンを別のパターンに変換できます。コンパイラ書き込みでは、プログラム内のどのトークン (意味を持った最小の文字列) をコンパイルするかを決定できます。

これらの問題すべてに共通の処理は字句解析で、ある特定の特徴をもったさまざまな文字列を識別します。このツールの名前 lex は、字句解析 (lexical analysys) からきています。lex を使用して、字句解析を扱う必要はありません。C などの標準言語で作成されたプログラムを使用して、字句解析を扱うことができます。lex が行うのはそのような C プログラムを作成することなので、lex はプログラムジェネレータと呼ばれています。

lex を使用すると、上記の処理を実行するプログラムを、短い時間で簡単に作成することができます。lex の短所は、手作業で記述した場合よりも長い C プログラムが作成されたり、実行速度が遅い C プログラムが作成されることがあるという点です。ただし、多くのアプリケーションにとっては、この問題は深刻なものではなく、lex を使用する利点の方が大きいです。

また、lex を使用して、文字数、ワード長、ワードの出現回数などの、入力テキストの特徴に関する統計データを収集することもできます。この章は、以下の節で構成されています。

- 字句アナライザプログラムの生成
- lex ソースの作成
- lex ソースの変換
- C++ 符号化 (mangle) シンボル

- lex と yacc の併用
- オートマトン
- ソース形式の概要

国際化

英語以外の言語でアプリケーションを開発する場合の lex の使用については、lex(1) を参照してください。

字句アナライザプログラムの生成

lex は、ユーザーが作成したソース仕様から C 言語スキャナを生成します。この仕様には、入力テキスト内で検索される文字列 (式) と、式が見つかったときに実行するアクションが示す、規則のリストが含まれています。lex 仕様の作成方法については、50ページの「lex ソースの作成」を参照してください。

字句アナライザの C ソースコードは、以下のコマンドを実行すると生成されます。

```
$ lex lex.l
```

lex.l は、lex 仕様が記述されているファイルです。(慣例的に lex.l という名前が使用されていますが、この名前は自由に選択できます。ただし接尾辞の .l は、他のシステムツール、特に make によって認識される規約なので、注意してください。) ソースコードは、デフォルトで、lex.yy.c という名前の出力ファイルに書き込まれます。このファイルには、指定した式が入力ファイル内で見つかるごとに 1 を返し、ファイルの終わりを検出すると 0 を返す yylex() という関数の定義が含まれています。yylex() を呼び出すと、トークンが 1 つ構文解析されます (結果は yylex() に返されます)。yylex() を再度呼び出すと、yylex() は最後に構文解析したトークンの次のトークンから構文解析を再開します。

以下の例のように、複数のファイルにまたがった仕様に対して lex を実行しても、lex.yy.c というファイルが 1 つ生成されます。

```
$ lex lex1.l lex2.l lex3.l
```

–t オプションを付けて lex を呼び出すと、lex.yy.c の代わりに stdout へ lex の出力が書き込まれるため、以下のように出力をリダイレクトすることができます。

```
$ lex -t lex.l > lex.c
```

lex のオプションは、コマンド名とファイル名の引数の間に入力します。

入力テキストの字句解析を行う実行可能オブジェクトプログラム (スキャナ) を生成するためには、lex.yy.c (または、lex.yy.c のリダイレクト先の .c ファイル) に保存された字句アナライザのコードをコンパイルする必要があります。

lex ライブラリは、関数 yylex() を呼び出すデフォルトの main() を提供します。したがって、ユーザーは独自の main() を用意する必要はありません。このライブラリにアクセスするには、以下のように cc に –ll オプションを付けてコンパイルします。

```
$ cc lex.yy.c -ll
```

また、ユーザーは独自のドライバを作成することもできます。lex ライブラリのドライバに類似したコードを以下に示します。

```
extern int yylex();

int yywrap()
{
     return(1);
}

main()
{
     while (yylex());
         ;
}
```

関数 yywrap() についての詳細は、50ページの「lex ソースの作成」を参照してください。以下のように、lex.yy.c と共にドライバファイルをコンパイルすると、lex ライブラリがロードされているかのように、その main() は実行時に yylex() を呼び出します。

```
$ cc lex.yy.c driver.c
```

生成された実行可能ファイルは、stdin を読み取り、出力を stdout に書き込みます。lex による処理の流れを図 2–1 に示します。

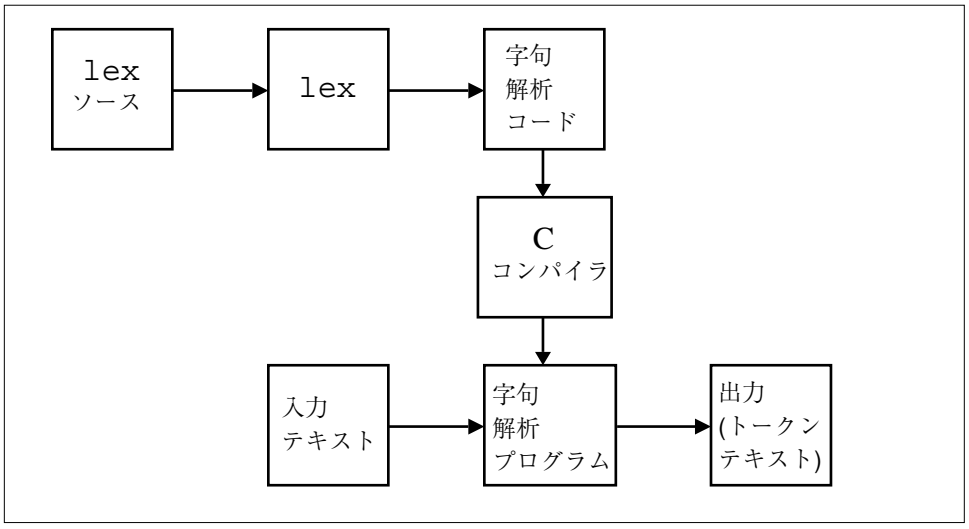


図 2–1　lex による字句アナライザの作成と使用

lex ソースの作成

lex のソースは、3 つまでのセクションで構成されています。その 3 つとは、定義、規則、ユーザー定義ルーチンです。規則セクションは必須です。定義とユーザー定義ルーチンのセクションは省略できますが、使用する場合には、以下に示した順に配置する必要があります。

```
定義
%%
規則
%%
ユーザー定義ルーチン
```

lex 規則の基礎

必須の規則セクションは、区切り記号 %% で開始します。ルーチンのセクションを後に続ける場合は、規則の最後の行に %% を記述して規則のセクションをそこで終了します。区切り記号 %% は、行の先頭に入力する必要があります。%% の前に空白文字は入れないでください。2 つ目の区切り記号がない場合には、規則セクションがプログラムの終わりまで続いているとみなされます。

規則セクション内の最初の規則の前にある空白で始まる行は、関数 yylex() の開始部分の最初の中括弧のすぐ後ろにコピーされます。この機能を利用すれば、yylex() の局所変数を宣言できます。

各規則によって、検索するパターンと、そのパターンが見つかったときに実行するアクションが指定されます。パターン仕様は、行の先頭から入力する必要があります。スキャナは、パターンと一致しない入力を出力ファイルに直接書き込みます。したがって、最も単純な字句アナライザプログラムは、規則セクションの開始を示す区切り記号 %% だけが入っているプログラムです。このプログラムは、すべての入力をまったく変更を加えずにそのまま出力に書き込みます。

正規表現

検索するパターンは、正規表現と呼ばれる表記法を使用して指定します。正規表現は、文字や演算子をつなぎ合せて形成します。最も単純な正規表現は、以下のような演算子が 1 つも含まれていないテキスト文字の列です。

```
apple
orange
pluto
```

これら 3 つの正規表現は、入力テキストに現われるこれと同じ文字列すべてと一致します。入力テキストに現われる orange をスキャナにすべて削除させたい場合には、以下のように規則を指定することができます。

```
orange ;
```

セミコロンの後にアクションが指定されていないため、スキャナは何も行わずに、元の入力テキストからこの正規表現に一致した文字列をすべて削除したテキストを出力します。つまり、出力されるテキストには、文字列 orange はまったく含まれません。

演算子

ほとんどの式は、orange のように簡単に指定することはできません。式自体は非常に長いものであったり、あるいは (こちらの方がよくある事例ですが) その式のクラスが非常に大きい可能性があります。実際、無限の大きさである可能性も考えられます。

lex 演算子 (表 2–1 を参照) を使用して、特定のクラスのあらゆる式を表すための正規表現を形成できます。たとえば + 演算子は、直前の式が 1 個以上存在することを表します (つまり、直前の式は省略可能です)。? は、直前の式が 0 個または 1 個存在

することを表します。* は、直前の式が 0 個以上存在することを表します。したがって m+ は、1 個以上の m で構成される文字列すべてと一致する正規表現になります。

```
mmm
m
mmmmm
```

また、7* は、0 個以上の 7 で構成される文字列すべてと一致する正規表現となります。

```
77
77777

777
```

空白の 3 行目が正規表現に一致したのは、その行に 7 が 1 つも含まれていないためです。

| 演算子は、二者択一を表します。したがって、ab|cd は ab または cd と一致します。演算子 {} は、繰り返しを指定します。したがって a{1,5} は、a が 1 ～ 5 個含まれている文字列と一致します。A(B{1,4}) は、ABC、ABBC、ABBBC、ABBBBC と一致します (グループ化するための記号として丸括弧 () を使用している点に注目してください)。

角括弧 [] は、角括弧内に指定された文字列のいずれかの文字を表します。したがって [dgka] は、1 個の d、g、k、または a と一致します。角括弧内の文字は、間に空白文字や句読点を入れないで隣接して並べてください。

^ 演算子は、左角括弧の直後の文字として使用したときには、角括弧内に指定した文字を除く標準セットのすべての文字を表します。|、{、}、および ^ は、他の目的で使用する場合もあるので注意してください。

標準のアルファベットや数字 (A ～ Z、a ～ z、0 ～ 9) の範囲は、ハイフンを使用して指定します。たとえば、[a-z] は、任意の小文字を表します。

```
[A-Za-z0-9*&#]
```

これは、任意の文字 (大文字または小文字)、任意の数字、アスタリスク、アンパサント、または # と一致する正規表現の式です。

以下に示した入力テキストを与えられると、上記の仕様を規則の 1 つとして持つ字句アナライザは、*、&、r、# を認識して、規則が指定するアクションをその認識した各文字に対して実行し、残りのテキストをそのまま出力します。

```
$$$$?? ????!!!*$$ $$$$$$&+====r~~# ((
```

ハイフン文字をクラスに加えるには、[-A-Z] または [A-Z-] のように、角括弧内の最初または最後の文字として入力してください。

演算子は、組み合わせて使用すると非常に効果的です。たとえば、複数のプログラミング言語の識別子を認識するための正規表現は、次のようになります。

```
[a-zA-Z][0-9a-zA-Z]*
```

これらの言語における識別子は、文字の後にゼロ、文字、または数字が続く文字列として定義されており、上記の正規表現は、その識別子を表しています。 1 組目の角括弧は、任意の文字に一致します。2 組目の角括弧は、その後の * がなければ、任意の数字または文字に一致します。

2 つの角括弧の組とそれに囲まれた文字は、後ろに 1 個の数字または文字が続く任意の文字と一致します。しかし * が付いているため、この例は後ろに任意の数の文字または数字が続く任意の文字と一致します。たとえば、この例は以下の文字列を識別子として認識します。

```
e
not
idenTIFIER
pH
EngineNo99
R2D2
```

一方、以下の文字列は識別子として認識されません。not_idenTIFIER には下線が含まれており、5times は文字でなく数字で始まっており、$hello は特殊文字で始まっているためです。

```
not_idenTIFIER
5times
$hello
```

演算子文字に潜在する問題は、検索する文字としてどのようにその演算子文字を検索パターン内で指定するかということです。たとえば、前述の例は * を含むテキストを認識しません。lex では、2 つの方法のいずれかでこの問題を解決します。
バックスラッシュの後の演算子文字、二重引用符で囲まれた文字 (バックスラッシュは除く) は、そのまま文字として扱われます。つまり、検索されるテキストの一部となります。

たとえば、バックスラッシュによる方法を使用して、* の後に任意の数の数字が続く文字列を認識するには、以下のパターンを使用できます。

```
\*[1-9]*
```

\ 自体を認識するには、バックスラッシュを 2 つ並べる必要があります (\\)。同様に、"x*x" は x*x と一致し、"y\"z" は y"z と一致します。その他の lex 演算子については、表 2–1 で説明しています。

表 2–1 lex 演算子

式	説明
\x	x が lex 演算子の場合は x
"xy"	x または y が lex 演算子 (\ は除く) の場合でも xy
[xy]	x または y
[x-z]	x、y、または z
[^x]	x 以外の文字
.	復帰改行以外の文字
^x	行の先頭の x
<y>x	lex が開始状態 y にあるときは x
x$	行の最後の x
x?	0 個または 1 個の x
x*	0 個以上の x
x+	1 個以上の x
x{m,n}	m ～ n 個の x
xx|yy	xx または yy
x |	x に対するアクションは、次の規則のアクション
(x)	x
x/y	x の後に y が続く場合にのみ x
{xx}	定義セクションからの xx の変換

アクション

スキャナは、規則の開始部にある正規表現に一致する文字列を認識すると、その規則の右側を検査してアクションを実行します。ユーザーは、このアクションを指定します。

見つかったトークンの型とその値を記録する、トークンを別のトークンと置換する、トークンまたはトークンの型のインスタンス数 (出現する数) をカウントする、などがアクションとして考えられます。これらのアクションは、C のプログラムとして作成します。

アクションは、任意の数の文を使用して構成できます。何らかの方法でテキストを変更したり、テキストが見つかったことを示すメッセージを出力したい場合があります。たとえば、式 `Amelia Earhart` を認識して、認識したことを通知するには、以下の規則を適用します。

```
"Amelia Earhart" printf("found Amelia");
```

テキスト内の長い医学用語を同義の略語に置換する処理は、以下のような規則で実現できます。

```
Electroencephalogram printf("EEG");
```

テキスト内の行数をカウントするには、行の終わりを認識して、行カウンタを増分します。

`lex` では、復帰改行を表す `\n` などの、標準の C エスケープシーケンスを使用します。たとえば行数をカウントするには、以下のような構文を使用できます。ここで使用している `lineno` は、他の C 変数と同じようにして、62ページの「定義」で宣言されています。

```
\n  lineno++;
```

C 言語の NULL 文 (1 個のコロン `;`) を指定すると、入力は無視されます。したがって、以下の規則を使用した場合には、空白文字、タブ、復帰改行は無視されます。

```
[ \t\n] ;
```

また、択一演算子 | は、ある規則とその次の規則のアクションが同じであることを表す際に使用することもできます。たとえば、前述の例は以下のように記述しても同じ結果になります。

```
" " |
\t
\n ;
```

スキャナは、式と一致したテキストを yytext[] という文字配列内に格納します。必要に応じて、この配列の内容を出力したり操作できます。実際に、lex は printf ("%s", yytext) と同義の ECHO というマクロを備えています。

C 言語の長い文または複数の文でアクションを構成する場合には、その文を複数の行に渡って記述できます。アクションが、1 つの規則だけに対するものであることを lex に伝えるには、その C コードを中括弧で囲みます。

たとえば、数字列が見つかるごとにその数字列とそれまでに見つかった数字列の総数を出力しながら、入力テキスト全体の数字列の総数をカウントするには、次のような lex コードを使用します。

```
\+?[1-9]+      { digstrngcount++;
                 printf("%d",digstrngcount);
                 printf("%s", yytext); }
```

? は、その直前の文字が 0 個または 1 個存在することを表しているので、この仕様は 0 個または 1 個のプラス記号が直前に付いた数字列と一致します。また、負の数字列の場合でも、マイナス記号の後の部分はこの仕様に一致するので、同様に数値が捕捉されます。

lex の高度な機能

lex に提供されている一連の機能を使用することによって、複雑なパターンで入力テキストを処理できます。考えられる複数の仕様のうちどの仕様が一番適しているかを決定する規則、一致パターンを別の一致パターンに変換する関数、定義とサブルーチンの使用なども含まれています。

以下の例には、これまで取り上げてきた事項がまとめて記述されています。

```
%%
-[0-9]+     printf("negative integer");
\+?[0-9]+   printf("positive integer");
-0.[0-9]+   printf("negative fraction, no whole number part");
rail[ \t]+road   printf("railroad is one word");
crook       printf("Here's a crook");
function    subprogcount++;
G[a-zA-Z]*  { printf("may have a G word here:%s", yytext);
                    Gstringcount++; }
```

最初の 3 つの規則では、それぞれ負の整数、正の整数、0 ～ -1 の間の負の小数が認識されます。各仕様の終わりには + が付いているので、これらの数字は 1 桁または複数桁の数字で構成されることになります。

その後の規則は、それぞれ以下に示す特定のパターンを認識します。

- railroad 用の仕様は、rail と road の 2 つの音節の間に 1 個以上の空白がある場合に一致します。railroad と crook については、メッセージの代わりに同義語を出力させる仕様にすることもできます。
- 関数を認識する規則は、カウンタを増分します。

  この規則には、いくつか重要なポイントがあります。

  - この規則では、複数行に渡る連続したアクションを中括弧で囲んでいます。
  - このアクションでは、認識した文字列を格納する lex 配列 yytext[] を使用しています。
  - この仕様では、* を使用して、G の後に 0 個以上の文字が続くことを表しています。

特殊な機能

スキャナは、一致した入力テキストを yytext[] に格納する以外にも、一致した入力テキストの文字数をカウントしてその値を変数 yyleng に自動的に格納します。ユーザーは、この変数を使用して、配列 yytext[] に配置された任意の文字を参照できます。

C 言語の配列インデックスは 0 から始まります。したがって、認識した整数の 3 桁目の数字 (3 桁目の数字がある場合) を出力するには、以下のように記述します。

```
[1-9]+    {if (yyleng > 2)
             printf("%c", yytext[2]);
           }
```

記述した一連の規則によってあいまいさが生じた場合、lex は上位のルールに従ってそのあまいさを解決します。以下の字句アナライザの例では、「予約語」end は 2 番目の規則と、識別子用の 8 番目の規則に一致します。

```
begin       return(BEGIN);
end         return(END);
while       return(WHILE);
if          return(IF);
package     return(PACKAGE);
reverse     return(REVERSE);
loop        return(LOOP);
[a-zA-Z][a-zA-Z0-9]*  { tokval = put_in_tabl();
                        return(IDENTIFIER); }
[0-9]+   { tokval = put_in_tabl();
           return(INTEGER); }
\+       { tokval = PLUS;
           return(ARITHOP); }
\-       { tokval = MINUS;
           return(ARITHOP); }
>        { tokval = GREATER;
```

```
            return(RELOP); }
>=        { tokval = GREATEREQL;
            return(RELOP); }
```

lex は、仕様内の 2 個以上の規則と一致する場合には、最初の規則のアクションが実行されるというルールに従います。識別子用の規則の前に end やその他の予約語用の規則を配置すれば、予約語を確実に認識できます。

検索するパターンが別のパターンの接頭辞になっている場合には、また別の問題が発生する可能性があります。たとえば、上記の字句アナライザの例における最後の 2 つの規則は、> と >= を認識するようにデザインされています。

lex は、できる限り長い文字列と照合してその文字列用の規則を実行する、というルールに従います。テキストの一部に文字列 >= が含まれている場合には、スキャナは > で止まって > の規則を実行するのではなく、>= を認識して >= の規則を実行します。このルールによって、C プログラムの + と ++ が区別されます。

字句アナライザが、検索する文字列を超えて読み取る必要がある場合には、後方コンテキストを使用してください。その典型的な例が、FORTRAN の DO 文です。以下の DO 文の最初の 1 は、最初のコンマを読み取るまではインデックス k の初期値のように見えます。

```
DO 50 k = 1 , 20, 1
```

つまり、最初のコンマを読み取るまでは、この文は代入文のように見えます。

```
DO50k = 1
```

FORTRAN では空白はすべて無視されます。後ろに続く文字列が後方コンテキストであることを示すには、スラッシュ / を使用します。スラッシュはパターンの一部ではないため、この後方コンテキストは yytext[] には格納されません。

FORTRAN の DO 文を認識する規則は、以下のように記述できます。

```
DO/([ ]*[0-9]+[ ]*[a-zA-Z0-9]+=[a-zA-Z0-9]+,) {
        printf("found DO");
        }
```

別バージョンの FORTRAN で識別子 (この例ではインデックス名) のサイズが制限されていても、この規則のように任意の長さのインデックス名を受け入れると、記述を簡素化することができます。後方コンテキストと類似した前方コンテキストの取り扱いについては、63ページの「開始条件」を参照してください。

lex では、特別な後方コンテキスト (行の終わり) を表す演算子として $ 記号を使用します。例として、行の終わりの空白文字とタブをすべて無視する規則を以下に示します。

```
[ \t]+$ ;
```

上記の例は、以下のように記述することもできます。

```
[ \t]+/\n ;
```

パターンが行またはファイルの先頭にあるときにだけそのパターンと一致させるには、^ 演算子を使用します。たとえば、あるテキスト書式整形プログラムでは、空白文字以外の文字で行を開始する必要があるとします。その場合には、以下の規則を使用して、そのプログラムへの入力を検査できます。

```
^[ ]    printf("error: remove leading blank");
```

左角括弧の内側に ^ 演算子がある場合との意味の違いに注意してください。

lex ルーチン

以下のマクロを使用して特殊なアクションを実行できます。

- input() は、文字をもう 1 つ読み取ります。
- unput() は、少し後でもう一度読み取る文字を元へ返します。
- output() は、文字を出力デバイスに書き込みます。

2 つの特殊文字 (たとえば、二重引用符など) の間にある文字をすべて無視する方法の 1 つとして、以下のように input() を使用する方法があります。

```
\"      while (input() != '"');
```

スキャナは、最初の二重引用符を読み取ってから 2 番目の二重引用符を読み取るまでは、照合を行わずに後続の文字をすべて読み取ります。(input() と unput() の使い方の詳しい例については、64ページの「ユーザールーチン」を参照してください。)

これらデフォルトのマクロでは対応していない特殊な入出力 (複数ファイルへの書き込みなど) が必要な場合には、C の標準入出力ルーチンを使用してマクロ関数を書き直してください。

ただし、これらのルーチンは一貫性をもって変更する必要があります。特に、文字セットはすべてのルーチンで同じものを使用し、input() によって返される値 0 はファイルの終わりを意味していなければなりません。また、input() と unput() との間の関係も維持する必要があります。関係が維持されていないと、lex の先読み処理は機能しなくなります。

独自の input()、output(c)、または unput(c) を作成する場合には、まず初めに #undef input、#undef output、または #undef unput を定義セクションに記述してください。

```
#undef input
#undef output
  . . .
#define input() ...  etc.
more declarations
  . . .
```

標準のルーチンが新しいルーチンへ置換されます。詳しくは、62ページの「定義」を参照してください。

ユーザーが再定義できる lex ライブラリルーチンは、スキャナがファイルの終わりに到達すると必ず呼び出される yywrap() です。yywrap() が 1 を返した場合には、スキャナは入力の終わりで通常の後処理を行います。新しいソースから引き続き入力を受け取るようにするには、yywrap() を再定義して、継続して処理が必要なときには 0 を返すようにします。デフォルトの yywrap() は必ず 1 を返します。

ファイルの終わりを認識する標準の規則を記述することはできないので注意してください。yywrap() からのみ、ファイルの終わりを認識する標準の規則を変更することができます。input() によって返される値 0 はファイルの終わりとみなされるので、ユーザーが独自の input() を用意しないかぎり、NULL を含んだファイルを取り扱うことはできません。

複数の方法で処理される文字列を扱うための lex ルーチンとしては、yymore()、yyless(n)、REJECT があります。 与えられた仕様に一致したテキストは配列 yytext[] に格納されることに注意してください。一般に、その仕様のアクションが実行されると、yytext[] 内の文字は、その次に一致した入力ストリーム内の文字に置換されます。

一方、関数 yymore() の場合には、認識された後続の文字は yytext[] 内にすでに格納されている文字の後に追加されます。この関数を使用すれば、ある文字列が有効で、その文字列を含んだ別の文字列も有効なときなどに、処理を連続して行うことができます。

二重引用符で囲まれた文字の集まりとして文字列が定義されている言語において、文字列に二重引用符を含める指定を行うには、その引用符の直前にバックスラッシュを置く必要があります。これに一致する正規表現は多少複雑であるため、次のように記述してください。

```
\"[^"]* {
     if (yytext[yyleng-2] == '\\')
          yymore();
     else
```

```
        ... 通常処理
    }
```

スキャナは、文字列 "abc\"def" に出会うと、まず文字列 "abc\" と照合します。次に、yymore() を呼び出して、文字列の次の部分 "def を先の文字列の後ろに追加します。文字列の終わりを示す二重引用符は、「通常処理」と記されている部分のコードで取り扱われます。

関数 yyless(n) を使用して、アクションの実行対象となる一致文字の数を指定できます。yytext[] には、式の最初の n 文字だけが保持されます。後続の処理は、nth+1 番目の文字から再開されます。

たとえば、コードの解読を行なっていて、特定の文字 (この例では、小文字または大文字の Z) で終了する文字列の半分だけを処理する場合には、以下のように記述します。

```
[a-yA-Y]+[Zz] { yyless(yyleng/2);
                 ... 文字列の前半分を処理
                 ...
               }
```

また、REJECT 関数を使用すれば、文字列が重複していたり、文字列の一部に別の文字列が含まれている場合であっても、文字列の処理をさらに容易に行うことができます。REJECT 関数では、yytext[] の内容を変更せずに次の規則とその仕様に直接ジャンプすることによって、この処理を実現しています。たとえば、入力テキスト内の正規表現 snapdragon とその部分式 dragon の両方の出現回数をカウントするには、以下のように記述します。

```
snapdragon    {countflowers++; REJECT;}
dragon         countmonsters++;
```

パターンが別のパターンと重複しているときの例を以下に示します。この例では、入力テキストに comediana などの文字列が含まれている場合でも、式 comedian と diana の出現回数をカウントします。

```
comedian    {comiccount++; REJECT;}
diana        princesscount++;
```

この例のアクションはカウンタの値を増やしていくことですが、それよりもずっと複雑なアクションを指定しなければならない場合もあります。カウンタやその他必要な変数は、常に lex 仕様の開始部分の定義セクションで宣言してください。

定義

lex の定義セクションには、複数のクラスの項目を入れることができます。最も重要なのは、外部定義、#include などの前処理文、および略語です。正式な lex ソースではこのセクションは省略可能ですが、ほとんどの場合において、これらの項目のいくつかは必要になります。前処理文と C ソースコードは、%{ と %} の行の間に記述します。

この区切り記号の間の行 (空白で始まる行も含む) は、すべて yylex() の定義の直前に lex.yy.c にコピーされます。(区切り記号で囲まれていない定義セクション内の行は、同じ場所にコピーされ、その先頭には空白が入れられます。)

定義セクションは通常、規則セクション内のアクションまたは外部にリンクされたルーチンによってアクセスされるオブジェクトの C 定義を配置する場所です。

たとえば、字句アナライザを呼び出すパーサーを生成する yacc と共に lex を使用するときには、ファイル y.tab.h をインクルードします。このファイル中で、#define を使用してトークン名を定義することができます。

```
%{
#include "y.tab.h"
extern int tokval;
int lineno;
%}
```

#include 文と宣言の終わりを示す %} の後には、規則セクション内の正規表現の略語を置いてください。行の左側には略語、行の右側にはその定義または変換後の正規表現を記述して、両者を 1 つ以上の空白文字で区切ります。

規則の中で略語を使用するときには、その略語を中括弧で必ず囲んでください。略語を使用することにより、仕様を繰り返し記述する手間が省け、見た目にも読みやすくなります。

たとえば、56ページの「lex の高度な機能」で取り上げた lex ソースについて再検討します。定義を使用することによって、数字、文字、空白文字が後で見た時にわかりやすくなります。

これは、同じ仕様が何度も現われる場合に特に効果的です。

```
D                   [0-9]
L                   [a-zA-Z]
B                   [ \t]+
%%
-{D}+               printf("negative integer");
\+?{D}+             printf("positive integer");
-0.{D}+             printf("negative fraction");
G{L}*               printf("may have a G word
                            here");
rail{B}road         printf("railroad is one word");
```

```
crook             printf("criminal");
 ...
 ...
```

開始条件

開始条件を使用すると、単独の ^ 演算子よりも細かく前方コンテキストを取り扱うことができます。たとえば、行の終わりまたはファイルの始まりよりも複雑な前方コンテキストに従って、異なる規則を式に適用する場合があります。

その場合には、規則の適用条件となるコンテキスト内の変化を示すフラグを設定して、そのフラグをテストするコードを作成できます。あるいは、各規則の適用条件となるさまざまな「開始条件」を lex 用に定義することもできます。

以下の問題について考えます。

- 入力を出力にコピーする。ただし、文字 a で始まる行にあるワード magic をすべて first に変更する。
- b で始まる行にある magic をすべて second に変更する。
- c で始まる行にある magic をすべて third に変更する。フラグを使用してこの問題を扱った場合のコード例を以下に示します。

  すでに説明したように、ECHO は printf ("%s", yytext) と同義の lex マクロです。

```
int flag
   %%
   ^a {flag = 'a'; ECHO;}
   ^b {flag = 'b'; ECHO;}
   ^c {flag = 'c'; ECHO;}
   \n {flag = 0; ECHO;}
   magic {
   switch (flag)
         {
               case 'a': printf("first"); break;
               case 'b': printf("second"); break;
               case 'c': printf("third"); break;
               default: ECHO; break;
         }
   }
```

開始条件を使用してこれと同じ問題を扱うには、以下のような行を lex の定義セクションに記述する必要があります。定義セクションには、任意の順序で条件を記述することができます。

```
%Start name1 name2 ...
```

ワード Start は、S または s と短縮できます。この条件を参照する場合は、規則の先頭で角括弧 <> を使用します。たとえば、スキャナの開始条件が name1 のときにだけ認識される規則は、以下のようになります。

```
<name1>式
```

開始条件に入るには、以下のアクションの文を実行します。

```
BEGIN name1;
```

上記の文は、開始条件を name1 に変更します。通常の状態に戻す場合は、以下の文を使用します。

```
BEGIN 0;
```

この文は、スキャナの初期条件をリセットします。

規則は、複数の開始条件によってアクティブにすることができます。たとえば、以下のような開始条件にすることもできます。

```
<name1,name2,name3>
```

前置演算子 <> で開始されていない規則は、すべてアクティブです。

開始条件を使用した場合には、前述の例は以下のように記述できます。

```
%Start AA BB CC
%%
^a    {ECHO; BEGIN AA;}
^b    {ECHO; BEGIN BB;}
^c    {ECHO; BEGIN CC;}
\n    {ECHO; BEGIN 0;}
<AA>magic    printf("first");
<BB>magic    printf("second");
<CC>magic    printf("third");
```

ユーザールーチン

ユーザーは、他のプログラミング言語でルーチンを使用する場合と同じようにして lex ルーチンを使用できます。複数の規則で使用されるアクションのコードは、1 度だけ作成して、必要なときに呼び出すことができます。定義の場合と同様に、このルーチンによってプログラムの記述が簡素化され、コードも読みやすくなります。

67ページの「lex と yacc の併用」で説明している put_in_tab1() 関数は、lex 仕様のユーザールーチンのセクションで使用するのに適した関数です。

このセクションにルーチンを配置するもう 1 つの理由は、コードを使用する規則が 1 つしかない場合であっても、重要なコードを強調したり規則セクションを簡素化

するためです。例として、/* と */ に挟まれた部分がコメントになっている C のような言語のコメントを無視する以下のルーチンについて説明します。

```
%{
static skipcmnts();
%}
%%
"/*"        skipcmnts();
 ...
 ...     /* 残りの規則 */
%%
static
skipcmnts()
{
       for(;;)
       {
            while (input() != '*')
                  ;

            if (input() != '/')
                unput(yytext[yyleng-1])
            else return;
       }
}
```

この例には、重要なポイントが 3 つあります。

- マクロ unput(c) は、読み取った最後の文字を元に返して、コメントが **/ で終わっている場合でも最後の / を読み取ることができるようにします。

  その場合、スキャナは、* を読み取った後にその次の文字が終端の / でないことを検知して読み取りを継続します。

- 式 yytext[yyleng-1] は、読み取った最後の文字を取り出します。
- このルーチンでは、C 言語の場合のようにコメントが入れ子になっていないものと仮定しています。

C++ 符号化シンボル

関数名が C++ の符号化 (mangle) シンボルの場合には、lex はそれを復号化された (demangle) 形式で出力します。符号化 C++ シンボルは、すべて復号化シンボルの後で [] でくくられます。通常の符号化 C++ 関数名 (メンバー関数と非メンバー関数を含む) の場合には、関数プロトタイプがその復号化 (demangle) の形式として使用されます。

たとえば、以下の関数名は、

```
_ct_13Iostream_initFv
```

以下のように出力されます。

```
Iostream_init::Iostream_init()
```

C++ の静的コンストラクタと静的デストラクタは、復号化されて以下の形式で出力されます。

```
static constructor function for
```

または、

```
static destructor function for
```

たとえば、以下のデストラクタは、

```
_std_stream_in_c_Fv
```

以下のように復号化されます。

```
static destructor function for _stream_in_c
```

C++ 仮想テーブルシンボルが符号化された時の名前は以下の形式になります。

```
_vtbl_class
```

```
_vtbl_root_class_derived_class
```

lex の出力では、仮想テーブルシンボルが復号化された時のの名前は以下のように出力されます。

```
virtual table for class
```

```
virtual table for class derived_class derived from root_class
```

たとえば、以下を復号化すると、

```
_vtbl_7fstream
```

以下の形式になります。

```
virtual table for fstreamH
```

また、以下を復号化すると、

```
_vtbl_3ios_18ostream_withassign
```

以下の形式になります。

```
virtual table for class ostream_withassign derived from ios
```

一部の C++ シンボルは、仮想テーブルに対するポインタです。符号化された時の名前は、以下の形式になります。

```
_ptbl_class_filename

_ptbl_root_class_derived_class_filename
```

lex の出力では、これらのシンボルが復号化された時の名前は以下のように出力されます。

```
pointer to virtual table for class in filename

pointer to virtual table for class derived class derived from
  root_class in filename
```

たとえば、以下を復号化すると、

```
_ptbl_3ios_stream_fstream_c
```

以下の形式になります。

```
pointer to the virtual table for ios in _stream_fstream_c
```

また、以下を復号化すると、

```
_ptbl_3ios_11fstreambase_stream_fstream_c
```

以下の形式になります。

```
_stream_fstream_c

pointer to the virtual table for class fstreambase derived
  from ios in _stream_fstream_c
```

lex と yacc の併用

コンパイラプロジェクトに取り組んでいる場合、または入力言語の妥当性を検査するプログラムを開発する場合には、システムツールの yacc (第 3 章を参照) を使用することがあります。yacc は、パーサーを生成します。パーサーは、入力を解析してその入力の構文に誤りがないことを検証するプログラムです。

多くの場合、コンパイラの開発では lex と yacc を併用して作業を行うことができます。

すでに説明したように、プログラムは、関数 yylex() を繰り返し呼び出して、lex が生成したスキャナを使用します。yacc が生成したパーサーはこれと同じ名前を使用して字句アナライザを呼び出すので、lex と yacc を併用する場合にこの名前は好都合です。

lex を使用してコンパイラ用の字句アナライザを作成する場合は、lex の各アクションはトークンを返す文で終了してください。トークンとは、整数値を使用して定義された語句です。

返されるトークンの整数値によって、字句アナライザが何を見つけたかがパーサーに示されます。その後、パーサー (yacc によって呼び出された yyparse()) は制御を再開して、字句アナライザに対して別の呼び出しを行い、次のトークンを取得します。

コンパイラでは、見つかったその言語の予約語 (予約語がある場合) によって、または識別子、定数、算術演算子、関係演算子のどれが見つかったのかによって、トークンの値は異なります。後者の場合は、アナライザはトークンの厳密な値も指定する必要があります。つまり、識別子、定数の値 (9 や 888 など)、算術演算子の種類 (+ や * など)、関係演算子の種類 (= や > など) を指定する必要があります。

以下に、C のような言語のトークンを認識するスキャナの lex ソースの一部を示します。

表 2–2 トークンを認識する lex ソースの例

```
begin        return(BEGIN);
end         return(END);
while        return(WHILE);
if          return(IF);
package       return(PACKAGE);
reverse       return(REVERSE);
loop         return(LOOP);
[a-zA-Z][a-zA-Z0-9]*
          { tokval = put_in_tabl();
                return(IDENTIFIER); }
[0-9]+      { tokval = put_in_tabl();
                return(INTEGER); }
\+         { tokval = PLUS;
                return(ARITHOP); }
\-         { tokval = MINUS;
                return(ARITHOP); }
>          { tokval = GREATER;
                return(RELOP); }
>=         { tokval = GREATEREQL;
                return(RELOP); }
```

返されるトークンと、tokval に割り当てられる値は、整数です。プログラムを理解しやすくするために、整数値をそのまま使用するのではなく、BEGIN、END、WHILE などの説明的な語句を使用してパーサーが解釈する整数を表してください。

整数値と語句の関連付けは、C パーサーを呼び出すルーチンの #define 文を使用して定義します。たとえば、以下のように定義します。

```
#define BEGIN 1
#define END 2
 ...
#define PLUS 7
 ...
```

また、いずれかのトークン型の整数を変更する場合は、その特定の整数が現われる個所をすべて変更するのではなく、パーサーの #define 文を変更してください。

yacc を使用してパーサーを生成するには、lex ソースの定義セクションに以下の文を挿入します。

```
#include "y.tab.h"
```

–d オプションを付けて yacc を呼び出したときに作成されるファイル y.tab.h は、BEGIN や END などのトークン名を有効な整数に関連付ける #define 文を、生成されたパーサーに提供します。

表 2–2 の予約語は、返される整数値だけで十分表すことができます。他のトークン型については、変数 tokval にその整数値が保存されます。

この変数は広域的に定義されているので、パーサーと字句アナライザはその変数にアクセスできます。yacc も、同じ用途の変数 yylval を提供します。

表 2–2 には tokval に値を割り当てる方法が 2 種類示されている点に注目してください。

- 1 つは、関数 put_in_tabl() による方法です。関数 put_in_tabl() は、識別子または定数の名前と型をシンボルテーブルに配置するので、コンパイラはその名前と型を参照できます。

  put_in_tabl() は、現在位置に加えて型の値も tokval に割り当てるので、パーサーはその情報をすぐに使用して入力テキストの構文上の正当性を判定できます。関数 put_in_tabl() は、コンパイラの作成者がパーサーのユーザールーチンセクションに配置するルーチンです。

- もう 1 つは、特定の整数を tokval に割り当てる方法です。tokval には、スキャナが認識した算術演算子または関係演算子を表す特定の整数が割り当てられます。

たとえば、#define 文によって変数 PLUS が整数 7 に関連付けられている場合には、+ が認識されると、そのアクションは + を表す値 7 を tokval に割り当てます。

スキャナは、演算子の一般クラス (つまり、ARITHOP または RELOP で表される整数) を、パーサーに返す値で示します。

lex と yacc を併用するときには、どちらを先に実行しても構いません。以下のコマンドを実行すると、ファイル y.tab.c にパーサーが生成されます。

```
$ yacc d grammar.y
```

すでに説明したように、–d オプションを使用すると、ファイル y.tab.h が作成されます。このファイルには、yacc によって割り当てられた整数のトークン値とユーザー定義のトークン名を関連付ける #define 文が含まれています。lex は以下のコマンドで呼び出すことができます。

```
$ lex lex.l
```

次に、以下のコマンドを使用して出力ファイルのコンパイルとリンクを行うことができます。

```
$ cc lex.yy.c y.tab.c -ly -ll
```

–ll オプションで lex ライブラリをロードする前に、–ly オプションで yacc ライブラリをロードして、提供されている main() が yacc パーサーを呼び出すようにしています。

また、CC と共に yacc を使用するには (特に、.l ファイル内の yyback()、yywrap()、yylook()() などのルーチンが外部 C 関数になる場合)、コマンド行に以下のコマンドを入力する必要があります。

```
$ CC -D__EXTERN_C__ ... ファイル名
```

オートマトン

入力テキスト内の式の認識は、lex が生成する決定性有限オートマトンによって実行されます。–v オプション を使用すると、有限オートマトンを表す少量の統計情報が出力されます。(有限オートマトンの詳細な説明と、lex における有限オートマトンの重要性については、『*Aho, Sethi, and Ullman text, Compilers: Principles, Techniques, and Tools*』(Addison-Wesley, 1986) を参照してください。)

lex は、テーブルを使用してその lex の有限オートマトンを表します。有限オートマトンで使用できる状態の最大数は、デフォルトで 500 に設定されています。多数の規則または非常に複雑な規則が lex ソースに含まれている場合には、lex ソースの定義セクションに以下のようなエントリをもう 1 つ置くことによって、そのデフォルト値を大きくすることができます。

```
%n 700
```

このエントリは、700 個の状態を扱うことができる大きさのテーブルを作成するように lex に指示します。–v オプションを使用すると、選択する必要のある状態の数が表示されます。

状態遷移の最大数を 2000 よりも多くするには、指定パラメータを a にします。

```
%a 2800
```

lex コマンドで使用できるオプションの一覧については、lex(1) を参照してください。

ソース形式のまとめ

lex ソースファイルの一般形式を以下に示します。

```
定義
%
規則
%
ユーザールーチン
```

定義セクションでは、以下の項目を自由に組み合わせて使用できます。

- 以下の形式による略語の定義

```
 name space translation
```

- 以下の形式で取り込まれたコード

```
% {
   C code
   % }
```

- 以下の形式による開始条件

```
Start name1 name2 ...
```

- 以下の形式による内部配列サイズの変更

```
%x nnn
```

nnn は配列サイズを表す 10 進整数で、x はそのパラメータを選択します。

内部配列サイズの変更は、以下のように表すことができます。

表 2–3 内部配列サイズ

p	位置
n	状態
e	ツリーノード
a	遷移
k	パック文字クラス
o	出力配列サイズ

規則セクションの行は以下の形式になります。

```
式 アクション
```

アクションは、中括弧で囲むことによって、次の行まで続けることができます。

lex 演算子用の文字を以下に示します。

```
" \ [] ^ - ? .  * | () $ / {} <> +
```

重要な lex 変数、関数、マクロを以下に示します。

表 2–4 lex 変数、関数、およびマクロ

yytext[]	char の配列
yyleng	int
yylex()	関数
yywrap()	関数
yymore()	関数

表 2–4　lex 変数、関数、およびマクロ　続く

yyless(n)	関数
REJECT	マクロ
ECHO	マクロ
input()	マクロ
unput(c)	マクロ
output(c)	マクロ

第 3 章

yacc - コンパイラコンパイラ

yacc (yet another compiler compiler) ユーティリティは、コンピュータプログラムに対する入力の構造を指定するための汎用ツールで構成されています。yacc を使用する場合は、以下の内容が含まれている仕様をあらかじめ用意する必要があります。

- 入力の要素を表す規則
- 規則が認識されたときに呼び出されるコード
- 入力を調べるための低レベルのスキャナの定義もしくは宣言

yacc は、この仕様を、入力ストリームを調べる C 言語の関数に変換します。パーサーと呼ばれるこの関数は、低レベルスキャナを呼び出すことによって機能します。

字句アナライザと呼ばれるスキャナは、入力ストリームから項目を拾い上げます。選択されたこの項目はトークンと呼ばれます。トークンは、構文規則と比較検査されます。

規則として認識されると、その規則に対して設定したコードが呼び出されます。このコードはアクションと呼ばれます。このアクションは、C 言語のコードです。アクションは、値を返したり、他のアクションが返す値を使用できます。

yacc 仕様の核となるのは、構文規則の集まりです。各規則には構成を記述し、その構成に名前が付けられます。構文規則の例を以下に示します。

```
date: month_name day ',' year ;
```

date、month_name、day、year がこの構文規則の構成になります。month_name、day、year は別の場所で詳細に定義されているものと仮定します。

この例では、コンマを単一引用符で囲んでいます。これは、コンマがそのまま入力に現われるということを意味しています。コロンとセミコロンは規則内では句読点で、入力の評価では無視されます。正しく定義されると、以下の入力はその規則によって照合されます。

```
July 4, 1776
```

字句アナライザは、構文解析関数の重要な要素です。ユーザーが作成するこのルーチンは、入力ストリームを読み取り、低レベルの構成を認識して、その構成をトークンとしてパーサーに伝えます。字句アナライザは、入力ストリームの構成を終端記号として認識します。一方、パーサーは、構成を非終端記号として認識します。混乱を避けるため、終端記号をトークンとして読み進めてください。

字句アナライザと構文規則のどちらを使用して構成を認識するかは、かなり自由に選択できます。たとえば、上記の例では以下のような規則を使用することもできます。

```
month_name : 'J' 'a' 'n' ;

month_name : 'F' 'e' 'b' ;
...
month_name : 'D' 'e' 'c' ;
```

字句アナライザは個々の文字を単純に認識するだけですが、このような低レベルの規則は、時間や空間を浪費する傾向があり、また yacc では扱えないほど仕様を複雑化する可能性があります。

通常、字句アナライザは、月の名前を認識して、month_name が見つかったことを示す通知を返します。この場合は、month_name がトークンであり、詳細な規則は不要です。

字句アナライザを通り過ぎる必要があるコンマなどのリテラル文字も、同様にトークンとみなされます。

仕様ファイルには柔軟性があります。前述の例には、比較的簡単に以下の規則を加えることができます。

```
date : month '/' day '/' year ;
```

この規則により、入力では

```
7/4/1776
```

の同義語として

```
July 4, 1776
```

も使用できるようになります。ほとんどの場合、この新しい規則は、既存の記述を変更せずに最低限の労力でシステムに取り入れることができます。

読み取られた入力は、仕様と一致していない場合もあります。左から右に検査することによって、入力エラーは理論上可能な最も早い段階で検出されます。したがって、無効な入力データの読み取りや計算の回数が実際に減少するだけでなく、通常は無効なデータも瞬時に見つけることができます。

入力仕様の一部として用意されているエラー処理では、無効データを再入力したり、無効データを読み飛ばした後に入力処理を再開できます。複数の仕様を一度に指定すると、yacc はパーサーの生成に失敗することがあります。たとえば、仕様どうしが矛盾している場合や、yacc の認識メカニズムよりも強力な認識メカニズムを必要とする仕様などの場合に失敗します。

前者の場合は、設計エラーであることを表しています。後者の場合は、さらに機能が豊富な字句アナライザを作成するか、または構文規則の一部を書き直すことによって、問題を解決することができます。

yacc は、どんな仕様でも扱えるわけではありませんが、他の類似したシステムよりも優れています。また、yacc で扱うのが難しい構成は、ほとんどの場合、人間でも扱うのが難しい構成です。入力を検査する有効な yacc 仕様によって、プログラム開発の初期で概念や設計のエラーが明らかになったという、ユーザーからの報告があります。

この章の残りの部分では、以下の主題について説明します。

- yacc 仕様を準備するための基本処理
- パーサーの操作
- あいまいさの取り扱い
- 算術式における演算子の優先度の取り扱い
- エラー検出と回復
- yacc が生成したパーサーの操作環境と特殊機能
- 仕様のスタイルと有効性を改善するためのヒント
- 高度なトピック

また、yacc 入力構文のまとめと 2 つの例を記載しています。

国際化

英語以外の言語でのアプリケーション開発における yacc の使用についての詳細は、yacc(1) を参照してください。

基本仕様

名前とは、トークンまたは非終端記号のことを指します。yacc では、トークン名をトークンとして宣言する必要があります。字句アナライザは、仕様ファイルの一部として取り込むこともできますが、モジュール化し独立したファイルとして保存する場合の方が多くあります。字句アナライザと同様に、他のサブルーチンを取り込むこともできます。

すべての仕様ファイルは、理論上は 3 つのセクションで構成されます。その 3 つのセクションとは、宣言、(構文) 規則、サブルーチンです。これらのセクションは、2 つのパーセント記号 %% (パーセント記号は、yacc 仕様では通常はエスケープ文字として使用されます) で区切られます。

すべてのセクションを使用したときの仕様全体は以下のようになります。

```
宣言
%%
規則
%%
サブルーチン
```

宣言とサブルーチンのセクションは省略可能です。最も短い正当な yacc 仕様の例を以下に示します。

```
%%
S:;
```

空白文字、タブ、復帰改行は無視されますが、名前や複数文字から成る予約シンボルでこれらを使用することはできません。コメントは、名前が正当である任意の場所で使用できます。C 言語の場合と同じように、コメントは /* と */ で囲みます。

規則セクションは、1 つ以上の構文規則で構成されます。構文規則は以下の形式になっています。

```
A: BODY ;
```

A は非終端記号を表しており、BODY はゼロ、または 1 つ以上の名前やリテラルを表しています。コロンとセミコロンは、yacc の句読点です。

名前は、任意の長さにすることが可能で、文字、ピリオド、下線、および数字で構成できます。ただし、数字を名前の先頭の文字に使用することはできません。大文字と小文字は区別されます。構文規則の本体で使用される名前では、トークンまたは非終端記号を表すことができます。

リテラルは、単一引用符で囲まれた文字で構成されます。C 言語の場合と同じように、リテラル内では、バックスラッシュはエスケープ文字です。yacc は、C 言語のエスケープシーケンスをすべて認識します。技術的ないくつかの理由により、NULL 文字を構文規則の中で使用することはできません。

左側の記述が同じ構文規則が複数ある場合には、縦棒を使用して左側の部分を繰り返し記述しなくても済むようにできます。また、規則の終わりにあるセミコロンは、縦棒の前方にあるものについては消去します。

したがって、以下の構文規則は、

```
A : B C D ;
A : E F ;
A : G ;
```

縦棒を使用した以下の形式で yacc に与えることができます。

```
A     : B C D
      | E F
      | G
;
```

左側の部分が同一の構文規則を、すべてまとめて構文規則セクションに記述する必要はありません。ただし、そうした方が入力は読みやすくなり、変更するのも容易になります。

非終端記号が空の文字列に一致させたい場合には、以下の規則によってそれを示すことができます。

```
epsilon : ;
```

コロンの後の空白は、yacc によって非終端記号 epsilon の構成として認識されます。

トークンを表す名前は宣言する必要があります。宣言セクションに以下のように記述するのが、最も簡単な方法です。

```
$token name1 name2 name3
```

宣言セクションで定義されていない名前は、すべて非終端記号を表しているとみなされます。非終端記号は、少なくとも 1 つの規則の左側に記述する必要があります。

すべての非終端記号において、開始シンボルは非常に重要な意味を持っています。このシンボルは、デフォルトでは、規則セクションの最初の構文規則の左側にあるとみなされます。開始シンボルは、宣言セクションで `%start` キーワードを使用して明示的に宣言できます。また、このように宣言することをお勧めします。

```
%start シンボル
```

パーサーに対する入力の終わりは、エンドマーカーと呼ばれる特殊なトークンで示されます。エンドマーカーは、ゼロまたは負の数値で表されます。

エンドマーカーまでのトークン (ただし、エンドマーカーは含まない) が開始シンボルと一致する構成を形成している場合には、パーサー関数はエンドマーカーを認識して、その入力を受け取った後に呼び出し元に戻ります。それ以外のコンテキストでエンドマーカーが認識された場合は、そのエンドマーカーはエラーです。

適切なときにエンドマーカーを返すことは、ユーザーが作成した字句アナライザの役割です。エンドマーカーは、通常はファイルの終わりやレコードの終わりなどの論理的に明白ないくつかの入出力状態を表します。

アクション

ユーザーは、規則が認識されたときに実行するアクションを各構文規則に関連付けることができます。アクションは、値を返したり、他のアクションによって返された値を取得できます。また、字句アナライザは、必要であれば、トークンの値を返すことができます。

アクションは、C 言語の文として入力および出力を行なったり、サブルーチンを呼び出したり、配列や変数を変更できます。アクションは、{ と } で囲まれた 1 つ以上の文によって指定されます。アクションを持った構文規則の例を以下に 2 つ示します。

```
A    : '(' B ')'
     {
         hello( 1, "abc" );
     }


XXX : YYY ZZZ
     {
         (void) printf("a message\n");
         flag = 25;
      }
```

$ 記号は、アクションとパーサーの間の伝達を容易にするために使用されます。擬似変数 $$ は、完了したアクションによって返される値を表します。

たとえば、以下のアクションは値 1 を返します。実際、このアクションが行うことはそれだけです。

```
{ $$ = 1; }
```

アクションは擬似変数 $1、$2、... $n を使用して、他のアクションと字句アナライザによって返された値を取得できます。これらの擬似変数は、規則の右側にある要素 1 ～ *n* によって返された値を参照します。要素には、左から右の順に番号が割り当てられます。以下のような規則の場合には、$2 は C によって返された値、$3 は D によって返された値を持つことになります。

```
A      : B C D ;
```

以下の規則で、一般的な例を示します。

```
expr    : '(' expr ')' ;
```

この規則で値は、括弧内の expr の値に意味があります。アクションの最初の要素はリテラルの左括弧なので、目的の論理的な結果は以下の記述によって表すことができます。

```
expr      : '(' expr ')'
          {
              $$ = $2 ;
          }
```

デフォルトでは、規則の値はその規則内の最初の要素の値 ($1) になります。したがって、以下の形式の構文規則では、明示的なアクションを持つ必要性はほとんどありません。

```
A : B ;
```

前述の例では、アクションは規則の終わりにすべて記述されています。場合によっては、規則が完全に構文解析される前に制御を行う必要があります。yacc では、規則の終了部だけでなく、規則の途中にもアクションを記述できます。

このアクションは、通常の $ メカニズムでアクセスできる値がこのアクションの中の記述によって返されると仮定しています。同様に、このアクションは、その左側のシンボルによって返された値にアクセスできます。したがって、以下の規則の場合には、x は 1 に設定され、y は C によって返される値に設定されます。

```
A     : B
      {
          $$ = 1;
      }
```

```
C
      {
          x = $2;
          y = $3;
      }
      ;
```

規則を終端していないアクションは、yacc によって扱われます。yacc は、新しい非終端記号の名前と、その名前を空の文字列に一致させる新しい規則を作成します。内部のアクションは、この追加された規則を認識することによって起動されるアクションです。

yacc は、以下のように記述されているかのように上記の例を取り扱います。

```
$ACT      : /* 空 */
          {
               $$ = 1;
          }
          ;
A         :  B $ACT C
          {
               x = $2;
               y = $3;
          }
          ;
```

$ACT は空のアクションです。

多くのアプリケーションでは、出力はアクションの直接的な結果ではありません。構文解析ツリーなどのデータ構造がメモリー内で構築され、そのデータ構造に変換処理が適用されてから出力が生成されます。ツリー構造の作成と保守を行うために必要なルーチンは用意されているので、構文解析ツリーは非常に簡単に構築できます。

たとえば、以下のようにして呼び出した場合に、ラベル L とその派生 n1 および n2 を持ったノードを作成し、その新しく作成したノードのインデックスを返す node という C 関数があると仮定します。

```
node( L, n1, n2 )
```

その場合、構文解析ツリーは、以下のような仕様のアクションを用意することによって構築できます。

```
expr      : expr '+' expr
          {
              $$ = node( '+', $1, $3 );
          }
```

ユーザーは、アクションで使用する他の変数を定義できます。宣言と定義は、宣言セクションで %{ と %} の間に記述します。これらの宣言と定義の適用範囲はグローバルスコープであるため、アクション文に認識されます。また、字句アナライザにも認

識されるようにすることも可能です。たとえば、以下の定義を宣言セクションに配置すれば、すべてのアクションで変数 (variable) にアクセスできるようになります。

```
%{ int variable = 0; %}
```

yacc パーサーは yy で始まる名前だけを使用しているので、yy で始まる名前は避ける必要があります。また、これまでに示した例では値がすべて整数であることに注意してください。

値については、106ページの「高度なトピック」で説明しています。最後に、以下のように定義した場合には、yacc は %{ の後からコピーを開始して、最初に検出した %}、つまり printf() の中の %}でコピーを終了するので注意してください。逆に、printf() の中で %{ を検出した場合には、その %{ からコピーが開始されます。

```
%{
      int i;
      printf("%}");
%}
```

字句解析

ユーザーは、字句アナライザを用意して、入力ストリームを読み取った後にトークン (必要であれば、値も共に) をパーサーに伝達する必要があります。字句アナライザは、yylex() と呼ばれる整数値の関数です。この関数は、読み取ったトークンの種類を表す整数値のトークン番号を返します。値がそのトークンに関連付けられている場合には、その値を外部変数 *yylval* に割り当てる必要があります。

パーサーと字句アナライザの間で伝達を行うためには、これらのトークン番号が両者で共通の番号でなければなりません。この番号は、yacc で選択することも、ユーザーが選択することもできます。いずれの場合でも、C 言語の #define メカニズムを使用して、字句アナライザがこれらの番号をシンボルとして返せるようにします。

たとえば、yacc 仕様ファイルの宣言セクションにトークン名 DIGIT が定義されているとします。適切なトークンを返すための字句アナライザの関連部分は、以下のように記述できます。

```
int yylex()
{
     extern int yylval;
     int c;
       ...
     c = getchar();
       ...
     switch (c)
     {
         ...
       case '0':
       case '1':
```

```
        ...
      case '9':
          yylval = c - '0';
          return (DIGIT);
        ...
    }
        ...
}
```

このコードは、DIGIT のトークン番号を返します。字句アナライザのコードはサブルーチンセクションに、DIGIT の宣言は宣言セクションに置くことができます。字句アナライザのコードは、以下のようにして、別にコンパイル済みファイルに置くこともできます。

- -d オプションを付けて yacc を実行します。これにより、トークンの #define 文を含んだ y.tab.h というファイルが生成されます。
- 独立したコンパイル済み字句アナライザに #include y.tab.h を追加します。

  このメカニズムによって、明示的で変更が容易にできる字句アナライザが作成されます。ただし、C 言語またはパーサーで予約済みのトークン名や重要なトークン名を使用しないでください。

  たとえば、トークン名 if または while を使用すると、字句アナライザのコンパイル時に深刻な問題が必ず発生します。トークン名 error はエラー処理用に予約されているので、安易に使用しないでください。

  デフォルトの状態では、トークン番号は yacc によって選択されます。リテラル文字のデフォルトのトークン番号は、文字セットの文字の数値です。他の名前には、257 以降のトークン番号が割り当てられます。

  自分でトークン番号を割り当てたい場合には、宣言セクションで最初に現われるトークン名またはリテラルの直後に負でない整数を置く必要があります。この整数は、名前またはリテラルのトークン番号とみなされます。この方法で定義されていない名前とリテラルには、yacc によってデフォルトの定義が割り当てられます。その場合には、トークン番号が重複する可能性があります。トークン番号が重複していないことを必ず確認してください。

  エンドマーカーは 0 または負のトークン番号を持つ必要があります。このトークン番号を再定義することはできません。したがって、字句アナライザは、入力の終わりに達したときに必ず 0 または 負の数字をトークンとして返すようになっていなければなりません。

  第 2 章で説明したように、lex によって生成された字句アナライザは、yacc と協調して動作するように設計されています。これらの字句アナライザの仕様では、構文規則の代わりに正規表現が使用されます。lex を使用して非常に複雑な

字句アナライザを生成できますが、理論上仕様が適合しない言語がいくつかあります。そのような言語の場合には、手作業で字句アナライザを作成する必要があります。

パーサーの操作

yacc を使用して、仕様ファイルを C 言語の手続きに変換します。この手続きは、与えられた仕様に従って入力を構文解析します。仕様からパーサーへの変換で使用されるアルゴリズムは非常に複雑なので、ここでは説明しません。ただし、パーサー自体は比較的シンプルであり、その使い方を理解すればエラー回復やあいまいさの取扱いが容易になります。

yacc によって生成されたパーサーは、スタックを持った有限状態マシンで構成されています。パーサーは、lookahead トークンと呼ばれる次の入力トークンを読み取ってそれを記憶することもできます。現在の状態は常にスタックの 1 番上にあります。有限状態マシンの状態には、小さい整数のラベルが付けられます。初期状態では、マシンは状態 0 (スタックには状態 0 しか入っていません) になっており、lookahead トークンもまだ読み取られていません。

マシンで実行できるアクションは、shift (シフト)、reduce (還元)、accept (受理)、error (エラー) の 4 つだけです。パーサーの動作手順を以下に示します。

1. パーサーは、実行するアクションを選択するためには lookahead トークンが必要かどうかを現在の状態に基づいて判断します。lookahead トークンが必要で、まだそのトークンを持っていない場合には、パーサーは yylex() を呼び出して次のトークンを取得します。
2. パーサーは、必要であれば現在の状態と lookahead トークンを使用して、次のアクションを決定し、それを実行します。この処理が行われると、状態はスタックにプッシュされたり、スタックからポップされることがあります。lookahead トークンは、処理されたり、処理されずにそのまま残されたりすることがあります。

   shift アクションは、パーサーで最も頻繁に使用されるアクションです。shift アクションを行うときには、必ず lookahead トークンが伴います。たとえば、状態 56 で以下のようなアクションを行なったとします。

```
IF shift 34
```

これは、状態 56 で lookahead トークンが IF の場合には、現在の状態 (56) がスタックにプッシュされ、状態 34 が現在の状態 (スタックの一番上の状態) になるということを意味しています。lookahead トークンは消去されます。

reduce アクションは、スタックが際限なく大きくなるのを防ぎます。reduce アクションは、パーサーが構文規則の右側を認識し、右側を左側で置換して、規則のインスタンスを認識したことを通知する準備ができたときに適用できます。lookahead トークンを調べて reduce を実行するかどうかを判断しなければならない場合もあります。実際には、ほとんどのデフォルトアクション (. で表されます) が reduce アクションです。

reduce アクションは、個々の構文規則に関連付けられます。構文規則にも小さい整数の番号が付けられるため、多少の混乱が生じます。以下のアクションは、構文規則 18 を参照します。

```
.  reduce 18
```

一方、以下のアクションの場合には、状態 34 を参照します。

```
IF shift 34
```

たとえば、以下の規則を還元すると仮定します。

```
A : x y z ;
```

reduce アクションは、左側のシンボル (この場合は A) と右側のシンボルの数 (この場合は 3) に依存します。reduce を実行するには、まずスタックの上位 3 つの状態をポップします (通常は、ポップされる状態の数は規則の右側のシンボルの数と一致します)。

これらの状態は、*x*、*y*、*z* を認識するときにスタックに置かれたものなので、すでにその必要性はなくなっています。これらの状態がポップされると、規則を処理する前のパーサーの状態が取り出されます。

この取り出された状態と、規則の左側のシンボルを使用して、結果的に *A* がシフトされます。新しい状態が取得され、スタックにプッシュされると、構文解析が再開されます。ただし、左側のシンボルの処理と通常のトークンのシフト処理はかなり異なっています。そのため、このアクションは goto アクションと呼ばれています。特に、lookahead トークンは goto アクションでは何の影響も受けませんが、shift アクションでは消去されので、注意してください。取り出された状態には、必ず以下のようなエントリが含まれています。この例の場合には、状態 20 がスタックにプッシュされて現在の状態になります。

```
A goto 20
```

実際には、reduce アクションは、状態をスタックからポップすることによって構文解析を逆戻しして、規則の左側を最初に認識したときの状態に戻します。そして、パーサーはその時点で左側を認識したかのように動作します。規則の右側が空の場合は、スタックから状態はポップされません。取り出された状態は実際の現在の状態です。

reduce アクションは、ユーザーが作成したアクションと値を扱う場合にも重要になります。規則を還元すると、スタックが調整される前に、規則と共に用意されたコードが実行されます。状態を保持するスタックのほかに、そのスタックと並行して動作する別のスタックが用意されています。このスタックは、字句アナライザとアクションから返された値を保持します。

shift アクションが実行されると、外部変数 *yylval* は、値スタックにコピーされます。還元は、ユーザーコードから戻った後に実行されます。goto アクションが実行されると、外部変数 *yyval* が値スタックにコピーされます。擬似変数 $1、$2 等は、その値スタックを参照します。

他の 2 つのパーサーアクションは、概念的にはさらに単純なアクションになっています。accept アクションは、入力全体が認識され、その入力が仕様と一致していることを示します。このアクションは、lookahead トークンがエンドマーカーのときにだけ使用されて、パーサーが作業を正常に終了したことを示します。

一方、error アクションは、パーサーが仕様に従って構文解析を続けられなくなった場所を表します。パーサーが (lookahead トークンと共に) 認識した入力トークンの後に、不正となる可能性がある入力を続けることはできません。パーサーは、エラーを報告し、その状態を回復してから構文解析を再開します。

エラー回復 (エラー検出との対応) については、99ページの「エラー処理」で説明しています。

以下の yacc 仕様について考えます。

```
$token      DING DONG DELL
%%
rhyme     : sound place
          ;
sound     : DING DONG
          ;
place     : DELL
          ;
```

–v (冗長) オプションを付けて yacc を呼び出すと、パーサーが記述された y.output というファイルが生成されます。

上記の文法に対応した y.output ファイルは以下のようになります (ただし、ファイルの後ろの統計情報は一部省略されています)。

```
state 0
         $accept : _rhyme $end

         DING shift 3
         .  error
         rhyme goto 1
         sound goto 2

state 1
         $accept : rhyme_$end

         $end accept
         .  error

state 2
         rhyme : sound_place

         DELL shift 5
         .  error

         place goto 4

state 3
         sound : DING_DONG

         DONG shift 6
         .  error

state 4
         rhyme : sound place_  (1)
         .  reduce 1

state 5
         place : DELL_  (3)

         .  reduce 3

state 6
         sound : DING DONG_  (2)
         .  reduce 2
```

このファイルでは、各状態のアクションが指定されており、各状態で処理される構文解析規則も記述されています。下線 (_) は、各規則において何が認識され、何がまだ認識されていないかを示すために使用されています。以下のように入力をすると、パーサーの動作を追跡できます。

```
DING DONG DELL
```

初期状態での現在状態は `state 0` です。

パーサーは、入力を参照して、`state 0` で使用可能なアクションのいずれかを選択します。したがって、最初のトークン `DING` が読み取られて、`lookahead` トークンになります。`DING` に対する `state 0` のアクションは `shift 3` です。この `state 3` はスタックにプッシュされ、`lookahead` トークンは消去されます。これ

により、state 3 が現在の状態になります。そして、その次のトークン DONG が読み取られて、lookahead トークンになります。

DONG に対する state 3 のアクションは、shift 6 です。state 6 はスタックにプッシュされ、lookahead トークンは消去されます。この時点で、スタックには 0、3、6 が含まれています。state 6 では、パーサーは、lookahead を調べることなく、以下の規則 2 によって還元を行います。

```
sound : DING DONG
```

state 6 と state 3 がスタックからポップされ、state 0 が取り出されます。そして、state 0 の記述を参照して (sound に対する goto を探して)、以下の記述を見つけ出します。

```
sound goto 2
```

state 2 は、スタックにプッシュされて、現在の状態になります。

state 2 では、次のトークン DELL を読み取る必要があります。DELL に対するアクションは shift 5 なので、state 5 がスタックにプッシュされ、lookahead トークンはクリアされます。この時点のスタックには、0、2、5 が入っています。state 5 には、規則 3 によって還元するアクションしかありません。この規則の右側には 1 つのシンボルしかないので、state 5 だけがポップされ、state 2 が取り出されます。

place (規則 3 の左側) に対する state 2 の goto は state 4 です。この時点のスタックには、0、2、4 が含まれています。state 4 には、規則 1 によって還元するアクションしかありません。規則 1 の右側にはシンボルが 2 つあるので、上位 2 つの状態がポップされ、state 0 が再度取り出されます。

- state 0 には rhyme に対する goto があるので、パーサーはstate 1 に移行します。
- state 1 では、入力が読み取られ、y.output ファイルの $end によって示されているエンドマーカーが取得されます。エンドマーカーを取得した場合には、state 1 のアクションによって構文解析は正常に終了します。

あいまいさと衝突

入力文字列が 2 種類以上の方法で構成できる場合には、構文規則のセットはあいまいになります。たとえば、2 つの式の間にマイナス記号を置いたものが算術式の 1 つであることを表す場合は、普通は以下の構文規則を使用します。

```
expr : expr '-' expr
```

この構文規則では、複雑な入力のすべてを構成する方法は完全には指定されていないので注意してください。たとえば、以下のような入力の場合には、

```
expr - expr - expr
```

この規則では以下のいずれかでその入力を構成できます。

```
( expr - expr ) - expr
```

```
expr - ( expr - expr )
```

最初のものは左結合、2 番目のものは右結合と呼ばれています。

yacc は、パーサーを構築する際にこのようなあいまいさを検出します。以下のような入力を与えられたときにパーサーが直面する問題について考えます。

```
expr - expr - expr
```

パーサーが 2 番目の expr を読み取ったときには、この入力は以下のように認識されます。

```
expr - expr
```

これは、前述の構文規則の右側と一致します。パーサーは、この規則を適用することによって入力を還元 (reduce) できます。この規則を適用すると、入力は還元されて expr (規則の左側) になります。その後、パーサーは入力の最後の部分 (その部分を以下に示します) を読み取って、再度 reduce を実行します。

```
- expr
```

この結果、入力は左結合であるとみなされます。

あるいは、以下の入力を認識した場合に、

```
expr - expr
```

パーサーは、すぐにその規則を適用しないで、以下の入力を認識するまで読み取りを継続することもできます。

```
expr - expr - expr
```

この入力を認識した後で、パーサーは右端の 3 つのシンボルに対して規則を適用し、それを expr に還元できます。その結果、以下の入力が残ります。

```
expr - expr
```

これで、この規則をもう一度還元することが可能になります。この結果、入力は右結合であるとみなされます。したがって、以下の入力を読み取ったパーサーは、shift または reduce のいずれかの正当なアクションを実行できます。

```
expr - expr
```

shift と reduce を選択する方法はありません。これは、shift-reduce 衝突と呼ばれています。また、正当な 2 つの還元の選択肢をパーサーが持っている可能性もあります。これは、reduce-reduce 衝突と呼ばれています。shift-shift 衝突というものは存在しません。

shift-reduce 衝突や reduce-reduce 衝突が生じた場合でも、yacc はパーサーを生成します。yacc は、選択肢がある場所で有効なステップの 1 つを選択することによって、パーサーの生成を行います。所定の状況において行う選択を表した規則は、あいまいさ解決規則と呼ばれます。

yacc は、デフォルトの 2 つのあいまいさ解決規則を呼び出します。

1. shift-reduce 衝突の場合は、デフォルトは shift になります。
2. reduce-reduce 衝突の場合は、デフォルトは、読み込まれた yacc 仕様中で先に定義された文法規則による reduce になります。

   規則 1 は、選択肢があるところでは還元はシフトの方に従うことを意味しています。規則 2 では、ユーザーがこの状況において行えるパーサーの動作の制御は多少おおまかなものになりますが、reduce-reduce 衝突は可能なかぎり避ける必要があります。

   衝突は、入力や論理に誤りがあるとき、または yacc では構築できない複雑なパーサーが (矛盾のない) 構文規則で必要になったときに発生します。また、どの規則を認識しているのかをパーサーが確認する前にアクションを実行する必要がある場合には、規則内のアクションを使用すると衝突が発生することがあります。

これらの場合にあいまいさ解決規則を適用するのは不適切であり、パーサーも不正なものになります。そのため、yacc は上記の規則 1 と規則 2 によって解決された shift-reduce 衝突と reduce-reduce 衝突の数を常に報告します。

一般に、あいまいさ解決規則を適用して正しいパーサーを生成できるときには、構文規則を書き直して同じ入力を読み取るようにすることもできます。構文規則を書き直した場合には、衝突は発生しません。そのため、初期のパーサージェネレータでは、衝突は重大なエラーとして扱われていました。

このような書き直しは、やや不自然である上に、低速のパーサーが生成されることになります。そのため、yacc は衝突が存在している場合であってもパーサーを生成します。

あいまいさ解決規則の効果を表した例を以下に示します。

```
stat    : IF '(' cond ')' stat
        | IF '(' cond ')' stat ELSE stat
        ;
```

これは、if-then-else 文を含んだプログラミング言語のコードの一部です。これらの規則では、IF と ELSE はトークン、cond は条件 (論理) 式を表す非終端記号、stat は文を表す非終端記号になっています。最初の規則は単純 if 規則、2 番目の規則は if-else 規則と呼ばれます。

この 2 つの規則は、あいまいな構造を生じます。その理由は、以下のような形式の入力は、

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

その 2 つの規則に従った場合に以下の 2 つの方法で構成できるからです。

```
IF ( C1 )
{
     IF ( C2 )
         S1
}
ELSE
     S2


IF ( C1 )
{
    IF ( C2 )
        S1
    ELSE
        S2
}
```

後者の解釈は、この構造を持った大部分のプログラミング言語で用いられる解釈です。つまり、各 ELSE は、ELSE に関連付けられていない直前の IF と関連付けられ

ます。この例で、パーサーが以下の入力を認識してから ELSE を探す場合について考えます。

```
IF ( C1 ) IF ( C2 ) S1
```

この入力は、単純 if 規則によってすぐに以下のように還元されます。

```
IF ( C1 ) stat
```

そして、以下に示す残りの入力が読み取られます。

```
ELSE S2
```

次に、if-else 規則によって以下の入力が還元されます。

```
IF ( C1 ) stat ELSE S2
```

この結果、前述した前者の方の入力のグループ化が行われます。

一方、ELSE をシフトして、S2 を読み取ってから、以下の右側の部分に if-else 規則を適用すると、

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

以下のように還元できます。

```
IF ( C1 ) stat
```

これは、単純 if 規則で還元できます。

この結果、前述した後者の方の入力のグループ化が行われます。通常はこちらが望ましい方法です。

パーサーが行うことができる有効な処理が 2 つあるので、shift-reduce 衝突が存在します。この場合にあいまいさ解決規則 1 を適用すると、パーサーは shift を実行し、その結果目的のグループ化が行われます。この shift-reduce 衝突は、現在の入力シンボルが ELSE で、以下のような特定の入力が認識済みのときにだけ発生します。

```
IF ( C1 ) IF ( C2 ) S1
```

一般に、衝突は多数発生する可能性があり、各衝突は入力シンボルと以前に読み取った入力のセットに関連付けられます。以前に読み取った入力は、パーサーの状態によって特徴づけられます。

yacc の衝突メッセージの内容は、–v を指定した場合の出力を検討するとよくわかります。例として、上記の衝突状態に相当する出力を以下に示します。

```
23: shift-reduce conflict (shift 45, reduce 18) on ELSE

state 23
      stat :   IF ( cond ) stat_ (18)
      stat :   IF ( cond ) stat_ELSE stat

      ELSE     shift 45
           .   reduce 18
```

最初の行は、衝突に関する説明で、状態と入力シンボルが表示されています。通常の状態の場合には、その状態でアクティブな構文規則とパーサーのアクションが表示されます。下線記号はすでに認識されている構文規則の一部を表しています。

したがって、state 23 のこの例では、パーサーは以下に対応する入力をすでに認識しています。

```
IF ( cond ) stat
```

また、この時点では、表示されている 2 つの構文規則はアクティブです。パーサーは 2 通りの処理を行うことができます。入力シンボルが ELSE の場合には、state 45 にシフトできます。

ELSE は state 45 においてすでにシフトされているので、state 45 にはその記述の一部として以下の行が含まれています。

```
stat : IF ( cond ) stat ELSE_stat
```

state 23 では、入力シンボルがアクションの中に明示的に記述されていない場合には、代替アクション (. によって指定されたアクション) が実行されます。つまり、入力シンボルが ELSE 以外の場合には、パーサーは構文規則 18 によって以下のように還元を行います。

```
stat : IF '(' cond ')' stat
```

shift コマンドの後の数字は別の状態を参照しており、reduce コマンドの後の数字は構文規則番号を参照しています。y.output ファイルでは、規則番号は還元可能な規則の後の括弧内に出力されます。大部分の状態では、reduce が使用可能であり、reduce がデフォルトのコマンドになっています。予期しない shift-reduce が発生した場合には、–v オプションによる出力を調べて、デフォルトのアクションが適切かどうかを判断してください。

優先度

衝突を解決する規則が不十分である状況が 1 つあります。それは、算術式の構文解析における状況です。一般に使用される算術式の構成のほとんどは、演算子の優先度の概念と、左または右の結合規則に関する情報によって自然に表現できます。適切なあいまいさ解決規則を伴ったあいまいな文法を使用すれば、あいまいでない文法で構築されたパーサーよりも高速かつ容易に書き込めるパーサーを作成できます。基本的な概念は、必要なすべての二項演算子と単項演算子について、以下の形式で構文規則を作成するということです。

```
expr : expr OP expr
```

次に :

```
expr : UNARY expr
```

これにより、構文解析の衝突を多数伴った、非常にあいまいな文法が作成されます。

あいまいさ解決規則として、すべての演算子の優先度または結合の強さ、および二項演算子の結合規則を指定します。この情報によって、yacc がこれらの規則に従って構文解析の衝突を解決し、必要な優先度と結合規則を認識するパーサーを構築することができます。

優先度と結合規則は、宣言セクションでトークンに関連付けられます。この処理は、yacc キーワードの %left、%right、または %nonassoc の後にトークンのリストが続く一連の行によって行われます。同じ行にあるトークンは、すべて同じ優先度レベルと結合規則を持っているとみなされます。各行は、優先度または結合が強いものから順に記述されます。以下にその例を示します。

```
%left '+' '-'
%left '*' '/'
```

この例では、4 つの算術演算子の優先度と結合規則が記述されています。+ と - は、左結合で、* と / よりも低い優先度を持っています。* と / も左結合です。キーワード %right は、右結合の演算子を表すために使用します。キーワード %nonassoc は、FORTRAN の演算子 .LT. などのような、互いに結合できない演算子を表すために使用します。つまり、以下の記述は FORTRAN では無効なので、.LT. は yacc ではキーワード %nonassoc を使用して表すことになります。

```
A .LT.  B .LT.  C
```

これらの宣言の例を以下に示します。

```
%right '='
%left '+' '-
%left '*' '/'

%%

expr    : expr '=' expr
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | NAME
        ;
```

この記述を以下の入力に対して使用すると、

```
a = b = c * d - e - f * g
```

演算子の正しい優先度を実現するために、以下のように構成されます。

```
a = ( b = ( ((c * d) - e) - (f * g) ) )
```

このメカニズムを使用するときには、通常は単項演算子にも優先度を与える必要があります。単項演算子と二項演算子は、記号表現が同一の場合がありますが、優先度は必ず異なります。たとえば、単項演算子の - と二項演算子の - とでは、優先度が異なります。

単項演算子の - には、* (乗算) と同じか、それよりも高い優先度を与えることができますが、二項演算子の - は * よりも低い優先度になります。キーワード %prec は、特定の構文規則に関連付けられている優先度を変更します。この %prec は、構文規則の本体の直後の、アクションや終端のセミコロンの前に記述します。%prec の後にはトークン名またはリテラルを記述します。これにより、その構文規則の優先度は、後続のトークン名またはリテラルと同じ優先度になります。たとえば、以下の規則を使用すれば、単項演算子の - を * と同じ優先度にできます。

```
%left '+' '-'
%left '*' '/'

%%

expr     : expr '+' expr
         | expr '-' expr
         | expr '*' expr
         | expr '/' expr
         | '-' expr %prec '*'
         | NAME
         ;
```

%left、%right、%nonassoc によって宣言されたトークンは、再度 %token によって宣言する必要はありません。ただし、宣言しても構いません。

優先度と結合規則は、構文解析の衝突を解決する際に yacc によって使用されます。優先度と結合規則によって、以下のようなあいまいさ解決規則が生成されます。

1. 優先度と結合規則を持ったトークンやリテラルの優先度と結合規則が記録されます。
2. ある 1 組の優先度と結合規則が、各構文規則に関連付けられます。これらは、規則の本体にある最終的なトークンまたはリテラルの優先度と結合規則です。%prec 構文を使用した場合には、このデフォルト値はその構文の値に置換されます。その構文規則に関連付けられた優先度と結合規則を持っていない構文規則もあります。
3. reduce-reduce または shift-reduce 衝突が生じていて、入力シンボルまたは構文規則が優先度と結合規則を持っていない場合には、前節で示した 2 つのデフォルトのあいまいさ解決規則が使用され、その衝突が報告されます。
4. shift-reduce 衝突が生じていて、構文規則と入力文字の両方が共にそれぞれに関連付けられた優先度と結合規則を持っている場合には、さらに高い優先度に関連付けられたアクション (shift または reduce) を選択して、その衝突を解決します。優先度が等しい場合には、結合規則が使用されます。左結合は reduce、右結合は shift、非結合は error を意味します。

   優先度によって解決された衝突は、yacc が報告する shift-reduce と reduce-reduce 衝突の数には加えられません。これは、優先度の仕様が間違っていると、入力構文のエラーを検出できなくなる可能性があることを意味しています。ある程度慣れるまでは優先度の使用を控えることも 1 つの方法です。y.output ファイルは、実際にパーサーが指定通りに動作しているかどうかを判定する際に利用できます。

優先度キーワードを使用して shift-reduce 衝突を解決する方法をさらに詳しく説明するため、前節の例と類似した例を取り上げます。以下の C 言語の文について考えます。

```
if (flag) if (anotherflag) x = 1;
    else x = 2;
```

パーサーにとっての問題は、else に対応するのは最初の if なのか、それとも 2 番目の if なのかということです。C プログラマであれば、この字下げに惑わされることなく、else は 2 番目の if に対応していると判断するでしょう。if-then-else 構成の以下の yacc 文法では、この問題を抽象化しています。つまり、入力 *iises* (if-if.statement.else-statment) によってこれらの C の文を形成します。

```
%{
#include <stdio.h>
%}
%token SIMPLE IF ELSE
%%
S          ;  stmnt
           ;
stmnt      : SIMPLE
           | if_stmnt
           ;
if_stmnt   : IF stmnt
                { printf("simple if\n");}
           | IF stmnt ELSE stmnt
                { printf("if_then_else\n");}
           ;
%%
int
yylex() {
        int c;
        c=getchar();
        if (c= =EOF) return 0;
        else switch(c) {
            case 'i': return IF;
            case 's': return SIMPLE;
            case 'e': return ELSE;
            default: return c;
        }
}
```

この仕様が yacc に渡されると、以下のメッセージが表示されます。

```
conflicts: 1 shift/reduce
```

問題なのは、yacc が *iises* と照合するために *iis* を読み取ったときに、選択肢が 2 つあるということです。つまり、*is* を文として認識する (reduce) 場合と、さらに入力を読み取って (shift)、最終的に *ises* を文として認識する場合があります。

この問題を解決する方法の 1 つとして、新しいトークン REDUCE を作成して解決する方法があります。この REDUCE の用途は、正しい優先度を規則に与えることです。

```
%{
#include <stdio.h>
%}
%token SIMPLE IF
%nonassoc REDUCE
%nonassoc ELSE
%%
S       : stmnt '\n'
        ;
stmnt   : SIMPLE
        | if_stmnt
        ;
if_stmnt : IF stmnt %prec REDUCE
            { printf("simple if");}
        | IF stmnt ELSE stmnt
            { printf("if_then_else");}
        ;
```

```
%%
```

if_stmnt の 2 番目の形式に関連付けられている優先度が高くなっているので、yacc は最初にその規則との照合を試み、衝突も報告されません。

この単純な例では、実際には新しいトークンは必要ありません。

```
%nonassoc IF

%nonassoc ELSE
```

上記の 2 つも機能します。また、shift-reduce 衝突では yacc はデフォルトで shift を行うので、実際には上記のような方法で衝突を解決する必要はありません。ただし、正しい仕様ならば診断メッセージは表示されないという点では、衝突を解決するのも良い方法です。

エラー処理

エラー処理には、意味上の問題がいくつか含まれています。たとえば、エラーが見つかったときに、構文解析ツリーの記憶領域の再生、シンボルテーブルのエントリの削除や変更、スイッチを設定して出力の生成を止めたりしなければならない場合があります。

エラーが見つかったときに処理をすべて停止するのは、あまり好ましくありません。入力の検査を継続してその先の構文エラーを見つける方が便利です。その場合には、エラーの後でパーサーを再始動させるという問題が持ち上がります。再始動を行う一般的な部類のアルゴリズムには、入力文字列からトークンをいくつか破棄する処理や、入力を継続するためにパーサーを調整しようとする、という処理が含まれています。

この処理の一部をユーザーが制御できるようにするため、yacc はエラーにトークン名 error を設定します。この名前は、構文規則の中で使用できます。結果的に、この名前は、エラーの発生する箇所と、エラー回復を行うことができる箇所を示します。

パーサーは、トークンエラーが有効な状態に入るまで、そのパーサーのスタックをポップします。そして、トークンエラーが現在の lookahead トークンであるかのようにふるまって、次のアクションを実行します。その後、lookahead トークンはエラーを引き起こしたトークンにリセットされます。エラー規則が特に指定されていない場合は、エラーの検出時に処理は停止します。

エラーメッセージが連続して出力されるのを避けるため、パーサーは、エラー検出後に 3 つのトークンを正常に読み取ってシフトするまではエラー状態を継続します。すでにパーサーがエラー状態にあるときにエラーを検出した場合には、メッセージは表示されず、入力トークンが削除されます。

たとえば、以下の形式の規則は、構文エラー時にパーサーはエラーが見つかった文を読み飛ばそうとすることを表しています。

```
stat : error
```

正確に言うと、パーサーは前方を検査し、正当に文の後に続いていると考えられる 3 つのトークンを探して、その 3 つの内の最初のトークンの処理を開始します。文の開始部分が明確でない場合は、誤って文の途中から処理を開始して、実際にはエラーではないところで 2 つ目のエラーを報告することがあります。

アクションは、これらの特別なエラー規則と共に使用できます。これらのアクションは、テーブルの再初期化、シンボルテーブル空間の修正などを試みます。上記のようなエラー規則は、非常に一般的な規則ですが、制御するのは困難です。

以下のような規則であれば、制御は多少容易になります。

```
stat : error ';'
```

この規則の場合は、エラーが見つかるとパーサーは文を次のセミコロンまで読み飛ばします。エラー後の次のセミコロンより前のトークンは、シフトできず、すべて破棄されます。セミコロンが見つかると、この規則は還元されてそれに関連するアクションが実行されます。

もう 1 つの形式は、エラーの後に行を再入力できるようにした方が望ましい対話型アプリケーションで用いられます。それを行う方法の 1 つを以下に示します。

```
input     : error '\n'
            {
                (void) printf("Reenter last line: " );
            }
            input
            {
                $$ = $4;
            }
;
```

この方法には、潜在的な問題が 1 つあります。パーサーは、エラーの後で正しく再同期したことを確認する前に 3 つの入力トークンを正しく処理しなければなりません。再入力された行の最初の 2 つのトークンにエラーが含まれている場合には、パーサーはその不正なトークンを削除し、何もメッセージは表示しません。これは、好ましくない状態です。

そのために、エラーが回復されたことをパーサーに認識させるメカニズムが用意されています。アクションの中に以下の文があると、パーサーは通常のモードにリセットされます。

```
yyerrok ;
```

したがって、前述の例は、以下のように書き直すことができます。

```
input      : error '\n'
             {
                 yyerrok;
                 (void) printf("Reenter last line: " );
             }
             input
             {
                 $$ = $4;
             }
             ;
```

エラーシンボルの直後に認識されたトークンは、エラーが発見された入力トークンです。場合によっては、このことが不適切になることがあります。たとえば、エラー回復アクションは、入力を再開する正しい位置を探すジョブを実行する必要があるとします。その場合には、1 つ前の lookahead トークンを消去する必要があります。アクションに以下の文を記述すれば、そのトークンは消去されます。

```
yyclearin ;
```

たとえば、次の有効な文の先頭まで入力を進める (ユーザーが作成した) 複雑な再同期化ルーチンの呼び出しが、エラー後のアクションであると仮定します。このルーチンを呼び出した後は、yylex() によって返された次のトークンは、有効な文の最初のトークンであるとみなされます。古い無効なトークンは破棄し、エラー状態をリセットする必要があります。これを実行する規則の例を以下に示します。

```
stat      : error
          {
              resynch();
              yyerrok ;
              yyclearin;
          }
          ;
```

これらのメカニズムは、確かに不完全ですが、簡単かつ効果的にパーサーを多くのエラーから回復できます。また、ユーザーは、プログラムの他の部分で必要なエラーアクションの処理を制御することもできます。

yacc 環境

yacc パーサーは、以下のコマンドを使用して作成します。

```
$ yacc grammar.y
```

上記の grammar.y は yacc 仕様が含まれているファイルです (接尾辞 .y は、他のオペレーティングシステムによって認識される表記です。必ずこの表記にする必要はありません) 。 出力は、y.tab.c という C 言語サブルーチンのファイルになります。yacc が提供する関数は、yyparse() という整数値型の関数です。

yyparse() を呼び出すと、この関数は yylex() を繰り返し呼び出します。yylex() は、ユーザーが作成する、入力トークンを取得するための字句アナライザです (83ページの「字句解析」を参照)。エラーが検出されると、yyparse() は値 1 を返しますが、エラー回復を行うことはできません。または、字句アナライザがエンドマーカートークンを返して、パーサーがそれを受け取ります。その場合には、yyparse() は値 0 を返します。

実用的なプログラムを作成するためには、ある程度の環境をこのパーサーに提供する必要があります。たとえば、C 言語のプログラムの場合は、yyparse() を呼び出す main() というルーチンを必ず定義しなければなりません。また、構文エラーを検出したときにメッセージを出力するためには、yyerror() というルーチンが必要になります。

この 2 つのルーチンは、ユーザーがいずれかの形式で提供する必要があります。yacc をすぐに使えるようにするため、デフォルトの main() と yyerror() を備えたライブラリが用意されています。ライブラリには、cc コマンドに対する引数 –ly によってアクセスできます。これらのデフォルトプログラムのソースコードを以下に示します。

```
main()
{
     return (yyparse());
}


# include <stdio.h>

yyerror(s)
char *s;
{
     (void) fprintf(stderr, "%s\n", s);
}
```

このように、デフォルトプログラムは非常に単純な構造になっています。yyerror() に対する引数は、エラーメッセージ (通常は、構文エラーのメッセージ) を含んだ文字列です。平均的なアプリケーションでは、これよりも複雑な処理が求められます。通常は、プログラムは入力の行番号を監視して、構文エラーを検出したときにメッセージと共にその行番号を出力する必要があります。外部整数変数 *yychar* には、エラーが検出されたときの lookahead トークンの番号が含まれています (これは診断に役立つ機能です)。ほとんどの場合、main() ルーチン (たとえば引数を読み取るなど) はユーザーによって提供されるので、yacc ライブラリは小規模プロジェクトや大規模プロジェクトの初期段階でしか役に立ちません。

外部整数変数 *yydebug* は、通常は 0 に設定されます。これを 0 以外の値に設定した場合には、パーサーは、読み取った入力シンボルやパーサーのアクションなどを含む、アクションの詳しい説明を出力します。

仕様の準備に関するヒント

この節では、効果的および明瞭で、変更が容易な仕様を準備するためのさまざまなヒントについて説明します。

入力スタイル

規則に実際のアクションを与え、同時に仕様ファイルも読みやすくするのは簡単なことではありません。スタイルに関するヒントをいくつか以下に示します。

- トークン名にはすべて大文字、非終端名にはすべて小文字を使用してください。このように表記しておくと、デバッグの際に役立ちます。
- 構文規則とアクションを別々の行に置けば、編集が容易になります。
- 左側が同一の規則をまとめて置きます。1 つの規則の左側だけを残して、後の規則はすべて縦棒で始めてください。
- 左側の規則の最後の定義の後にだけセミコロンを置きます。そのセミコロンは別の行に置いてください。これにより、新しい規則を容易に追加できるようになります。
- 規則の本体は 1 個のタブ、アクションの本体は 2 個のタブを使用して字下げしてください。

- 複雑なアクションは、別のファイルで定義されたサブルーチンに入れてください。

  この節の例は、このスタイルに従って書かれています。要点は、アクションのコードの中から規則を認識しやすいように記述することです。

左再帰

yacc パーサーによって使用されるアルゴリズムには、いわゆる左再帰構文規則が頻繁に利用されます。以下の形式の規則はこのアルゴリズムに一致します。

```
name : name rest_of_rule ;
```

シーケンスとリストの仕様を作成するときには、以下のような規則を頻繁に使用します。

```
list     : item
         | list ',' item
         ;
```

次に :

```
seq      : item
         | seq item
         ;
```

どちらの場合でも、最初の規則は最初の項目についてだけ還元され、2 番目の規則は 2 番目とそれ以降のすべての項目について還元されます。

以下のような右再帰規則を使用した場合には、パーサーのサイズが多少大きくなります。認識された項目は、右から左に向かって還元されます。

```
seq     : item
        | item seq
        ;
```

問題点は、非常に長いシーケンスを読み取った場合に、パーサーの内部スタックがオーバーフローを起こす可能性があることです (ただし、現在の yacc は、かなり大きなスタックを処理できるようになっています)。したがって、なるべく左再帰の方を使用するようにしてください。

ゼロのシーケンスに意味があるかどうかについて考えてみましょう。たとえば、空の規則を使用して以下のシーケンス仕様を作成した場合について考えます。

```
seq      : /* 空 */
         | seq item
         ;
```

最初の規則は最初の項目が読み取られる前に必ず 1 回だけ還元されます。2 番目の規則は、読み取られた各項目ごとに 1 回還元されます。空のシーケンスを許可すると、一般性が高くなることがよくあります。ただし、どの空シーケンスを認識したのかを、まだ十分に認識していない段階で yacc が判断するように要求された場合には、衝突が発生する可能性があります。

C++ 符号化シンボル

この節の内容は、第 2 章の 65ページの「C++ 符号化シンボル」と全く同じ内容です。lex を yacc に置き換えて読んでください。

字句連結

一部の字句の判断はコンテキストに依存します。たとえば、字句アナライザは通常は空白文字を削除しますが、引用符で囲まれた文字列の空白文字は削除しません。また、名前は宣言の中のシンボルテーブルに入力できますが、式の中のシンボルテーブルには入力できません。これらの状況に対処する方法の 1 つは、字句アナライザによって検査され、アクションによって設定される広域フラグを作成することです。以下にその例を示します。

```
%{
        int dflag;
%}
...  その他の宣言 ...
%%
prog     : decls stats
         ;
decls    :      /* 空 */
         {
            dflag = 1;
         }
         | decls declaration
         ;
stats    :     /* 空 */
         {
            dflag = 0;
         }
         | stats statement
         ;

.  .  .  その他の規則  .  .  .
```

この例では、0 個以上の宣言とそれに続く 0 個以上の文で構成されるプログラムが指定されています。文を読み取ると *dflag* は 0 になり、宣言を読み取ると *dflag* は 1 になります。ただし、最初の文の最初のトークンは除きます。

パーサーは、宣言セクションが終了して文が始まったことを認識する前に、このトークンを読み取る必要があります。多くの場合、この 1 つのトークン例外は、字句の検査には影響しません。この考え方は、難しい構造を表現する場合の手がかりになります。

予約語

一部のプログラミング言語では、通常はラベル名や変数名として予約されている `if` などのワードを使用できます。これらのプログラミング言語では、そのような使い方をしても本来の意味で使用されている名前とは衝突しません。しかし、これを `yacc` で実現するのは非常に困難です。

`if` のあるインスタンスがキーワードであるか変数であるかを伝える情報を字句アナライザに渡すのは容易ではありません。前節の情報を読むと、105ページの「字句連結」がここで役に立つことがわかります。

高度なトピック

この節では、`yacc` の高度な機能についていくつか説明します。

アクションにおける **error** と **accept** のシミュレーション

`error` と `accept` の構文解析アクションは、マクロ `YYACCEPT` と `YYERROR` を使用してアクションのなかでシミュレーションを行うことができます。`YYACCEPT` マクロを使用すると、`yyparse()` は値 0 を返します。`YYERROR` を使用すると、パーサーは現在の入力シンボルが構文エラーであるかのように動作します。つまり、`yyerror()` を呼び出して、エラー回復を実行します。

これらのメカニズムは、複数のエンドマーカーを持つパーサーのシミュレーションや、コンテキスト文法の検査に使用できます。

囲み規則の中の値へのアクセス

あるアクションは、左側に記述された規則のアクションによって返された値を参照できます。メカニズムは通常のアクションと同じで、$ の後に数字が 1 つ続きます。

```
sent     : adj noun verb adj noun
           {
               look at the sentence ...
           }
           ;
adj        : THE
           {
               $$ = THE;
           }
           | YOUNG
           {
               $$ = YOUNG;
           }
              ...
               ;
noun       : DOG
           {
               $$ = DOG;
           }
           | CRONE
           {
               if ( $0 = = YOUNG )
               {
                   (void) printf( "what?\n" );
               }
               $$ = CRONE;
           }
           ;
...
```

この例では、その数字は 0 または 負の値にできます。ワード CRONE の後のアクションでは、シフトされた前のトークンが YOUNG かどうかを検査します。ただし、この検査は、入力内のシンボル noun の前のシンボルについて詳しい情報が得られている場合にのみ可能です。場合によっては、このメカニズムによって多くの問題が回避できます。特に、他の標準構造から結合をいくつか除外するときに役立ちます。

任意値型のサポート

デフォルトでは、アクションと字句アナライザによって返される値は整数です。yacc は、構造体などの他の型の値もサポートできます。また、生成されるパーサーの型を厳密に検査するために、yacc は型を監視して、適切な共用体のメンバー名を挿入します。yacc の値スタックは、必要なさまざまな型の値の共用体として宣言されます。共用体を宣言したら、値を持った各トークンと非終端記号に共用体の

メンバー名を関連付けます。$$ または $n 構成によって値が参照されると、yacc は、不要な変換が起きないように、適切な共用体の名前を自動的に挿入します。

これには 3 つのメカニズムが用意されています。1 つ目は、共用体を定義する方法です。他のサブルーチン (特に、字句アナライザ) は共用体のメンバー名を知る必要があるので、この定義はユーザーが行わなければなりません。2 つ目は、共用体のメンバー名をトークンと非終端記号に関連づける方法です。3 つ目は、yacc が容易に型を判定できないごく少数の値の型を表すメカニズムです。

共用体を宣言するには、以下の記述を宣言セクションに加えます。

```
%union
{
     共用体の本体
}
```

これにより、yacc の値スタックと外部変数 *yylval* および *yyval* は、この共用体と同じ型で宣言されます。–d オプションを付けて yacc を呼び出した場合には、共用体の宣言は y.tab.h ファイルに YYSTYPE としてコピーされます。

YYSTYPE を定義したら、共用体のメンバー名を各種の終端および非終端の名前に関連づける必要があります。以下の構文を使用して、共用体のメンバー名を指定します。

```
<name>
```

%token、%left、%right、%nonassoc のいずれかのキーワードの後に上記の指定が続いている場合は、共用体の名前は記述されているトークンに関連付けられます。

したがって、以下のように記述すると、これら 2 つのトークンによって返された値に対する参照には、すべて共用体のメンバー名 optype がタグとして付けられます。

```
%left <optype> '+' '-'
```

もう 1 つのキーワード %type は、共用体のメンバー名を非終端記号に関連づける際に使用します。たとえば、以下の規則を使用すれば、共用体メンバー nodetype を非終端記号 expr と stat に関連づけることができます。

```
%type <nodetype> expr stat
```

これらのメカニズムでは対応しきれないケースもいくつかあります。規則のなかにアクションがある場合には、そのアクションによって返される値には型がありません。同様に、左側のコンテキスト値 (たとえば、*$0* など) を参照した場合は、yacc はその型を容易に知ることができなくなります。この場合には、最初の $ の直後の < と > の間に共用体のメンバー名を挿入することによって、型を強制的に参照できます。その例を以下に示します。

```
rule        : aaa
            {
                 $<intval>$ = 3;
            }
            bbb
            {
                 fun( $<intval>2, $<other>0 );
            }
            ;
```

この構文には特に説明することはありませんが、このような状況は頻繁に発生します。

任意値型のサポート機能は、実際に使用されるまでは起動されません。特に %type の使用によって、これらのメカニズムは有効になります。これらの機能を使用するときには、非常に厳密な検査が行われます。

たとえば、型が定義されていないものを $n または $$ を使用して参照すると、型の診断が行われます。これらの機能を起動しない場合は、yacc の値スタックは int を保持するために使用されます。

yacc 入力構文

この節では、yacc 仕様としての yacc 入力構文について説明します。コンテキスト依存性などについては考慮していません。yacc では LALR(1) 文法を使用することもできますが、yacc 入力仕様言語は、LR(2) 文法として指定されています。この文法では、規則内のアクションの直後に識別子があると、問題が発生します。

この識別子の後にコロンが続いている場合には、次の規則の開始を表しています。コロンが続いていない場合は、現在の規則の続きであり、その規則のなかに偶然アクションが埋め込まれていただけに過ぎません。字句アナライザは、識別子を認識した後にその先を読み取り、次のトークン (空白文字、復帰改行、コメントなどは読み飛ばします) がコロンかどうかを判定します。コロンの場合には、トークン C_IDENTIFIER を返します。

コロンでない場合には、IDENTIFIER を返します。リテラル (引用符で囲んだ文字列) も C_IDENTIFIER としてではなく、必ず IDENTIFIER として返されます。

```
      /* yacc に対する入力の文法 */

      /* 基本エントリ */
%token IDENTIFIER     /* 識別子とリテラルの取り込み */

%token C_IDENTIFIER   /* 識別子 (リテラル以外) */
                      /* 後に a : が続く */
```

```
%token NUMBER          /* [0-9]+ */

      /* 予約語: %type=>TYPE %left=>LEFT、等 */

%token LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token MARK            /* %% 記号 */

%token LCURL           /* %{ 記号 */

%token RCURL           /* %) 記号 */

       /* 自分自身を表す ASCII 文字リテラル*/

%token spec t

%%

spec     : defs MARK rules tail
         ;
tail     : MARK
         {
            In this action,read in the rest of the file
         }
         |       /* 空: 2 番目の MARK は省略可 */
         ;
defs     :       /* 空 */
         | defs def
         ;
def      : START IDENTIFIER
         | UNION
         {
            Copy union definition to output
         }
         | LCURL
         {
           Copy C code to output file
         }
         RCURL
         | rword tag nlist
         ;
rword    : TOKEN
         | LEFT
         | RIGHT
         | NONASSOC
         | TYPE
         ;
tag      :           /* 空: 共用体タグは省略可 */
         | '<' IDENTIFIER '>'
         ;
nlist    : nmno
         | nlist nmno
         | nlist ',' nmno
         ;
nmno     : IDENTIFIER /* 注: リテラル % type では無効 */
         | IDENTIFIER NUMBER  /* 注: % type では無効 */
         ;

                    /* 規則セクション */
```

```
rules    : C_IDENTIFIER rbody prec
         | rules rule
         ;
rule     : C_IDENTIFIER rbody prec
         | '|' rbody prec
         ;
rbody    :          /* 空 */
         | rbody IDENTIFIER
         | rbody act
         ;
act      : '{'
         {
            Copy action translate $$ etc.
         }
         '}'
         ;
prec     :         /* 空 */
         | PREC IDENTIFIER
         | PREC IDENTIFIER act
         | prec ';'
         ;
```

例

簡単な例

以下のコード例では、小型電卓に対して yacc を完全に適用する例を示します。この電卓は、a ～ z の名前が付いた 26 個のレジスタを持っており、演算子 +、-、*、/、%、&、| と代入演算子で構成された算術式を使用できます。

最上位レベルの式が代入の場合は、その代入だけが行われます。それ以外の場合は、式が出力されます。C 言語の場合と同じように、0 で始まる整数は 8 進数とみなされます。それ以外は、10 進数とみなされます。

yacc 仕様の例として、この電卓プログラムでは、優先度とあいまいさの使い方を示し、簡単な回復処理も含まれています。大幅に簡略化されているのは字句アナライザの部分で、大部分のアプリケーションよりもかなり単純なものになっています。また出力は行ごとにすぐ生成されます。

構文規則によって 8 進 と 10 進の整数を読み取る方法に注目してください。このジョブは、字句アナライザで実行することもできます。

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;
%}

%start list
%token DIGIT LETTER
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' %'
%left UMINUS       /* 単項演算子 - の優先度を指定 */

%%                 /* 規則セクションの開始 */

list      :        /* 空 */
          | list stat '\n'
          | list error '\n'
          {
              yyerrok;
          }
          ;
stat      : expr
          {
              (void) printf( "%d\n", $1 );
          }
          | LETTER '=' expr
          {
              regs[$1] = $3;
          }
          ;
expr      : '(' expr ')'
          {
              $$ = $2;
          }
          | expr '+' expr
          {
              $$ = $1 + $3;
          }
          | expr '-' expr
          {
              $$ = $1 - $3;
          {
          | expr '*' expr
          {
              $$ = $1 * $3;
          }
          | expr '/' expr
          {
              $$ = $1 / $3;
          }
          | exp '%' expr
          {
              $$ = $1 % $3;
```

(続く)

```
            }
            | expr '&' expr
            {
                $$ = $1 & $3;
            }
            | expr '|' expr
            {
                $$ = $1 | $3;
            }
            | '-' expr %prec UMINUS
            {
                $$ = -$2;
            }
            | LETTER
            {
                $$ = reg[$1];
            }
            | number
            ;
number      : DIGIT
            {
                $$ = $1; base = ($1= =0) ? 8 ; 10;
            }
            | number DIGIT
            {
                $$ = base * $1 + $2;
            }
            ;

%%              /* サブルーチンセクションの開始 */
int yylex( )    /* 字句解析ルーチン */
{               /* 小文字の場合は LETTER を返す */
                /* yylval = 0 ～ 25、DIGIT を返す */
                /* 数字の場合、yylval = 0 ～ 9 */
                /* その他の文字はすぐに返される */
   int c;                       /* 空白は読み飛ばす */
   while ((c = getchar()) = = ' ')
       ;
                  /* c は空白以外の文字 */

   if (islower(c)) {
      yylval = c - 'a';
      return (LETTER);
   }
   if (isdigit(c)) {
      yylval = c - '0';
      return (DIGIT);
   }
   return (c);
}
```

高度な例

この節では、高度な機能をいくつか使用した文法の例を示します。例 1 の電卓を変更して、浮動小数点区間演算を実行できる電卓にしています。この電卓は、浮動小数点定数、算術演算子 +、-、*、/ 、および単項演算子 - を認識します。また、この電卓は、a ～ z のレジスタを使用します。さらに、次のような X が Y より小さいまたは等しい区間演算も理解できます。

```
(X、Y)
```

A ～ *Z* の 26 個の区間値変数を使用することもできます。使い方は例 1 の電卓と同様で、代入は値を返さず何も出力しませんが、式は (浮動小数点または区間) 値を出力します。

この例では、yacc と C の多数の機能を利用しています。区間は、double として保存されている左終点と右終点の値で構成される構造体で表されます。この構造体には、typedef によって型名 INTERVAL が与えられます。

また、yacc の値スタックは、浮動小数点のスカラーと整数 (変数の値を格納する配列を指し示すために使用されます) を含むことができます。このプログラム全体の設計は、C 言語の構造体と共用体を割り当てることが可能ということに強く依存しているので注意してください。実際に、多くのアクションは構造体を返す関数を呼び出しています。

YYERROR を使用して、0 を含んだ区間による除算や、誤った順序で表された区間などのエラー状態を処理している点に注目してください。また、yacc のエラー回復メカニズムを使用して、残りの不正な行を無視しています。

値スタックでの型の混在に加え、この文法では、中間式の型 (スカラーや区間など) を監視する構文を使用することもできます。また、コンテキストで区間値が必要になった場合には、スカラー値を自動的に区間値に昇格できます。ただしその場合には、yacc によって文法が実行されたときに多数の衝突 (18 個の shift-reduce、26 個の reduce-reduce) が発生することになります。

この問題は、以下の 2 つの入力行で確認できます。

```
2.5 + (3.5 - 4.)
```

次に :

```
2.5 + (3.5, 4)
```

2 番目の例では区間値の式で 2.5 が使用されていますが、区間値であることはコンマを読み取るまでわからないということに注意してください。2.5 を読み取ったパーサーは、後戻りすることはできず、別の処理を行うことになります。一般的に、任意の数のトークンを先読みし、スカラーを区間に変換するかどうかを判断する必要があります。

この問題は、2 項区間値の演算子の規則を 2 つ用意することによって回避できます。1 つは、左側のオペランドがスカラーのときの規則、もう 1 つは、左側のオペランドが区間のときの規則です。後者の場合には、右側のオペランドを区間にする必要があります。それにより、変換は自動的に適用されます。

この問題回避を行なっても、変換するかどうかを判断しなければならないために、上記の衝突が発生する場合が多数あります。これらの問題は、スカラーを先に生成する規則を仕様ファイルに記述することによって解決されます。これにより、スカラー値の式は強制的に区間に変換されるまではスカラー値のまま維持されるため、衝突は解決されます。

複数の型を扱うこの方法は、参考になります。2 つだけでなくさまざまな型の式がある場合には、必要になる規則の数は大幅に増加し、衝突の数もさらに増加します。したがって、この例は確かに有益ですが、もっと普通のプログラミング言語環境における実践で、文法の一部ではなく値の一部として型の情報を扱うことをお勧めします。

最後に、字句解析に関する事項を説明します。このプログラムの唯一の特殊な機能は、浮動小数点定数の処理です。C 言語のライブラリルーチン `atof()` を使用して、文字列から倍精度の値への実際の変換を行なっています。字句アナライザは、エラーを検出すると、文法内の無効なトークンを返すことで応答します。これにより、パーサー内で構文エラーが発生し、エラー回復が行われます。`yacc` 仕様のコード例を以下に示します。

```
%{
#include <stdio.h>
#include <ctype.h>

typedef struct interval {
      double lo, hi;
} INTERVAL;

INTERVAL vmul(), vdiv();
double   atof();
double   dreg[26];
INTERVAL vreg[26];
```

(続く)

```
%}

%start lines

%union {
       int ival;
       double dval;
       INTERVAL vval;
}

%token <ival> DREG VREG   /* dreg と vreg 配列へのインデックス */
%token <dval> CONST       /* 浮動小数点定数 */
%type  <dval> dexp        /* 式 */
%type  <vval> vexp        /* 区間式 */

         /* 演算子に関する優先順位の情報 */

%left '+' '/-'
%left '*' '/'

%%       /* 規則セクションの開始 */

lines    : /* 空 */
           | lines line
         ;
line     : dexp '\n'
           {
              (void)printf("%15.8f\n", $1);
           }
           | vexp '\n'
           {
            (void)printf("(%15.8f, %15.8f)\n", $1.lo, $1.hi);
           }
           | DREG '=' dexp '\n'
           {
            dreg[$1] = $3;
           }
           | VREG '=' vexp '\n'
           {
           vreg[$1] = $3;
           }
           | error '\n'
           {
           yyerrok;
           }
           ;
dexp     : CONST
           | DREG
           {
           $$ = dreg[$1];
           }
           | dexp '+' dexp
```

(続く)

```
            {
              $$ = $1 + $3;
            }
            | dexp '-' dexp
            {
             $$ = $1 - $3;
            }
            | dexp '*' dexp
            {
             $$ = $1 * $3;
            }
            | dexp '/' dexp
            {
             $$ = $1 / $3;
            }
            | '-' dexp
            {
              $$ = -$2;
            }
            | '(' dexp ')'
            {
             $$ = $2;
            }
            ;
vexp      : dexp
            {
           $$.hi = $$.lo = $1;
            }
            | '(' dexp ',' dexp ')'
            {
             $$.lo = $2;
              $$.hi = $4;
               if($$.lo > $$.hi) {
                 (void) printf("interval out of order\n");
               YYERROR;
            }
            }
            | VREG
            {
              $$ = vreg[$1];
            }
            | vexp '+' vexp
            {
              $$.hi = $1.hi + $3.hi;
              $$.lo = $1.lo + $3.lo;
            }
            | dexp '+' vexp
            {
              $$.hi = $1 + $3.hi;
              $$.lo = $1 + $3.lo;
            }
            | vexp '-' vexp
            {
```

(続く)

```
                $$.hi = $1.hi - $3.lo;
                $$.lo = $1.lo - $3.hi;
              }
              | dexp '-' vexp
              {
                $$.hi = $1 - $3.lo;
                $$.lo = $1 - $3.hi;
              }
              | vexp '*' vexp
              {
                $$ = vmul($1.lo, $1.hi, $3);
              }
              | dexp '*' vexp
              {
                $$ = vmul($1, $1, $3);
              }
              | vexp '/' vexp
              {
                if (dcheck($3)) YYERROR;
                $$ = vdiv($1.lo, $1.hi, $3);
              }
              | dexp '/' vexp
              {
                if (dcheck($3)) YYERROR;
                $$ = vdiv($1, $1, $3);
              }
              | '-' vexp
              {
                $$.hi = -$2.lo;
                $$.lo = -$2.hi;
              }
              | '(' vexp ')'
              {
                $$ = $2;
              }
              ;

%%                  /* サブルーチンセクションの開始 */

# define BSZ 50   /* 浮動小数点数値のバッファーサイズ */

/* 字句解析 */

int yylex()
{
        register int c;
                    /* 空白文字は読み飛ばす */
        while ((c=getchar()) = = ' ')
                ;
        if (isupper(c)) {
             yylval.ival = c - 'A';
              return(VREG);
        }
```

(続く)

```
        if (islower(c)) {
              yylval.ival = c - 'a';
             return(DREG);
        }

                     /* 数字、小数点、指数 */
        if (isdigit(c) || c = = '.') {
             char buf[BSZ + 1], *cp = buf;
             int dot = 0, exp = 0;
               for (;(cp - buf) < BSZ; ++cp, c = getchar()) {
                *cp = c;
                if (isdigit(c))
                continue;
                if (c = = '.') {
                    if (dot++ || exp)
                    return('.'); /* 構文エラーが発生する */
                    continue;
                }
                if (c = = 'e') {
                    if (exp++)
                    return('e'); /* 構文エラーが発生する */
                        continue;
                }
                /* 数字の終わり */
                break;
             }
         *cp = '\0';
         if (cp - buf >= BSZ)
            (void)printf("constant too long -- truncated\n");
           else
            ungetc(c, stdin); /* 最後に読み取った char を元に戻す */
         yylval.dval = atof(buf);
         return(CONST);
      }
      return(c);
}

INTERVAL
hilo(a, b, c, d)
double a, b, c, d;
   /* vmul と vdiv ルーチンで使用した a、b、c、d を含む最小区間を返す */

     INTERVAL v;

     if (a > b) {
        v.hi = a;
        v.lo = b;
     }
     else {
        v.hi = b;
        v.lo = a;
     }
     if (c > d) {
```

(続く)

```
        if (c > v.hi)
           v.hi = c;
           if (d < v.lo)
              v.lo = d;
     }
     else {
        if (d > v.hi)
           v.hi = d;
           if (c < v.lo)
              v.lo = c;:
     }
     return(v);
}

INTERVAL
vmul(a, b, v)
double a, b;
INTERVAL v;
{
      return(hilo(a * v.hi, a * v.lo, b * v.hi, b * v.lo));
}
dcheck(v)
INTERVAL v;
{
      if (v.hi >= 0.  && v.lo <= 0.) {
         (void) printf("divisor interval contains 0.\n");
         return(1);
      }
return(0);
}

INTERVAL
vdiv(a, b, v)
double a, b;
INTERVAL v;
{
        return(hilo(a / v.hi, a / v.lo, b / v.hi, b / v.lo));
}
```

第 4 章

make ユーティリティ

この章では、make ユーティリティに関する以下の項目を説明しています。

- 隠れた依存関係の検査
- コマンド依存関係の検査
- パターンマッチングの規則
- SCCS ファイルの自動取り出し

このバージョンの make ユーティリティは、旧バージョンの make (System V make) 用に記述されたメークファイルを使用して実行できます。ただし、メークファイルが拡張機能に依存している場合には、他のバージョンで使用できないことがあります (旧バージョンの make についての詳細は、付録 A を参照してください)。拡張機能および互換性の概要については、191ページの「make の拡張機能のまとめ」を参照してください。

make は、オブジェクトファイルおよび実行可能プログラムの生成および保守を簡素化します。make によって、プログラムを一貫してコンパイルすることが容易になります。また、ソースコードの変更の影響を受けないモジュールに対して、不要な再コンパイルを実行しません。make には、コンパイルを簡単にする機能が提供されています。対話式で行わない複雑な処理や繰り返し処理を自動化することができます。また、オブジェクトライブラリの更新および保守、テストの実行、ファイルシステムまたはテープへのファイルのインストールを行うことができます。make を SCCS とともに使用すると、ソースファイルの階層全体の任意のバージョンから、大規模なソフトウェアプロジェクトを構築することができます。

make は、ユーザーが作成したメークファイルというファイルを読み取ります。メークファイルには、構築するファイルとその構築方法が指定されています。ユー

ザーは、メークファイルを記述してそれをテストした後は、処理の詳細を記憶しておく必要はありません。処理はすべて make が行います。

依存関係の検査：make とシェルスクリプト

単純な処理の場合には、ソフトウェアの構築にシェルスクリプトを使用して整合性を保つことができますが、ソフトウェアプロジェクトの構築では多くの場合スクリプトは適していません。プログラムまたはオブジェクトモジュールのうちの 1 つだけが変更されている場合に、単純なスクリプトを使用してすべてのプログラムまたはオブジェクトモジュールをコンパイルするのは効率がよくありません。また、スクリプトを繰り返し編集しているうちに、整合性を保つことができない場合があります。再コンパイルが必要なモジュールだけを再コンパイルするスクリプトを記述することはできますが、それには make を使用する方が適しています。

make を使用すると、構築するファイルとその構築方法を簡潔かつ構造的に記述することができます。make は、依存関係の検査という仕組みを使用して、各モジュールとその派生元のソースファイルまたは中間ファイルを比較します。make は、モジュールが最後に構築された後にこれらの派生元のファイル (依存ファイルと呼びます) が変更されている場合にのみ、モジュールを再構築します。

派生したファイルがソースファイルよりも古いものかどうかを確認するため、make は (既存の) モジュールおよびその依存ファイルが修正された時間を比較します。モジュールがない、あるいはモジュールが依存ファイルよりも古い場合は、make はそのモジュールが最新でないとみなし、再構築に必要なコマンドを発行します。再構築に使用されたコマンドが変更されている場合に、モジュールが最新でないとみなされることがあります。

make は依存関係を完全に検査するため、1 つのソースファイルへの変更が任意の数の中間ファイルまたは処理過程に一貫して反映されます。このため、上位から下位へという階層で処理を指定することができます。

make は、メークファイルを読み取り、実行する必要のある処理を特定し、モジュールを完成させるために必要な処理のみを実行します。構築する各ファイルや実行するそれぞれの処理 (手順) をターゲットと呼びます。メークファイル中に記述するターゲットのエントリには、ターゲットの名前、そのターゲットが依存するターゲットのリスト、ターゲットを構築するためのコマンドのリストが記述されています。

コマンドのリストを規則と呼びます。make は、依存関係を必須ターゲットとして扱い、依存関係を必要に応じて更新してから現在のターゲットを処理します。ター

ゲットの規則によって生成されたファイルをターゲットファイルと呼びます。ターゲットの名前は、生成されるターゲットファイルの名前と対応付けられています。ターゲットの規則がファイルを生成しない場合もあります。ターゲットの派生元のファイル (ターゲットが依存しているファイルなど) を依存ファイルと呼びます。

ターゲットの規則がそのターゲットの名前のファイルを生成しない場合は、make は規則を実行し、それ以降の make の実行ではターゲットが最新のものであるとみなします。

make は、現在の実行で処理されているファイルに対しては、make だけが変更を行うことを前提としています。make の実行中に他の処理がソースファイルを変更すると、make が生成するファイルが不整合になる場合があります。

簡単なメークファイルの記述

メークファイルのターゲットのエントリの基本的な書式は、以下のとおりです。

表 4–1 メークファイルのターゲットのエントリ形式

```
target . . . : [ dependency . . . ]
        [ command ]
        . . .
```

最初の行では、ターゲット名のリストの後にコロンを付けます。依存関係がある場合にはコロンの後にそのリストを記述します。複数のターゲットが記述されている場合は、指定された規則を使用して各ターゲットが独立して構築されます。

タブで始まる次の行に示されているように、行はターゲットの規則を示すコマンド行になります。タブの代わりに空白文字を使用することはできません。

で始まる行は、次の (エスケープされていない) 復帰改行 (NEWLINE) までがコメントとして扱われ、ターゲットのエントリとして処理されません。次にタブと # 以外の文字で始まる空白でない行の直前まで、あるいはファイルの最後までが、ターゲットのエントリとみなされます。

メークファイルは、以下の例のように 1 つのターゲットだけで構成される簡単なものもあります。

表 4–2　簡単なメークファイル

```
test:
    ls test
    touch test
```

引数を指定せずに make を実行すると、make はまず makefile という名前のファイルを検索し、見つからない場合は Makefile という名前のファイルを検索します。いずれかの名前のメークファイルが SCCS 管理されている場合、make は SCCS の履歴ファイルを調べ、メークファイルが最新でない場合は、最新のバージョンを抽出します。

次に、make はメークファイル内の最初のターゲットのエントリについて依存関係の検査を開始します。メークファイルがない場合は、構築するターゲットのリストをコマンド行で引数として指定する必要があります。ターゲットの構築中には、make が実行中のコマンドが表示されます。

```
$ make
ls test
test not found
touch test
ls test
test
```

この例では、test というファイルが存在しない (つまり最新の状態でない) ため、make はターゲットのエントリに指定されている規則を実行しています。make をもう一度実行すると、ターゲットは最新になっているというメッセージが表示され、規則は実行されません。

```
$ make
'test' is up to date.
```

注 - セミコロン (;)、切り替え記号 (<、>、>>、|)、置換記号 (*、?、[]、$、=)、引用符、エスケープ文字、コメント ("、'、`、\、# など、:) などの、シェルのメタキャラクタがコマンド行に含まれている場合は、make は Bourne シェルを起動してコマンド行を処理します。シェルによるコマンド行の構文解析が必要ない場合は、make はコマンドを直接実行します。

各コマンド行は独立したプロセスまたはシェルによって実行されるため、規則内での行の区切りには重要な意味があります。

以下の規則を例にして説明します。

```
test:
	cd /tmp
	pwd
```

この規則の実行結果は、以下のようになります。

```
$ make test
cd /tmp
pwd
/usr/tutorial/waite/arcana/minor/pentangles
```

一連のコマンドをセミコロンで区切って指定すると、シェルを 1 回呼び出すだけですべてのコマンドが実行されます。

```
test:
    cd /tmp ; pwd
```

また、復帰改行をバックスラッシュ (\) でエスケープすると、2 行に渡って記述された入力コマンドが、1 行で続けて記述された場合と同等に扱われます。make は、エスケープされた復帰改行を空白文字として処理します。

バックスラッシュは、行の最後に記述する必要があります。バックスラッシュの前には、シェルが処理するためのセミコロンが必要です。

```
test:
	cd /tmp ; \
	pwd
```

暗黙の規則の基本的な用途

ターゲットに規則が指定されていない場合は、make は暗黙の規則を使用してターゲットを構築します。ターゲットが属するファイルのクラスに対する規則が make によって検出された場合は、暗黙の規則のターゲットエントリで定義されている規則が適用されます。

make は、ユーザーが指定したメークファイルの他に、デフォルトのメークファイル (/usr/share/lib/make/make.rules) も読み取ります。このデフォルトのメークファイルには、多数の暗黙の規則のためのターゲットエントリなどの情報が記述されています。

注 - System V make では、暗黙の規則は make プログラム中に定義 (ハードコード) されています。

暗黙の規則には、接尾辞の規則とパターンマッチングの規則の 2 種類があります。接尾辞の規則は、ある接尾辞を持つファイルを、ベース名は同じで接尾辞が異なる

別のファイルから構築するためのコマンドを指定します。パターンマッチングの規則は、ワイルドカードのパターンに一致するターゲットおよび依存関係に基づいて規則を選択します。デフォルトの暗黙の規則は、接尾辞の規則です。

接尾辞の規則を使用すると、メークファイルを記述する必要がなくなる場合があります。たとえば、以下のように make コマンドを実行して、functions.c という 1 つの C ソースファイルから functions.o という名前のオブジェクトファイルを構築することができます。

```
$ make functions.o
cc -c functions.c -o functions.o
```

makeは、nonesuch.c というソースファイルから nonesuch.o というオブジェクトファイルを構築する場合でも同様に機能します。

functions.c から functions という名前の (接尾辞なし) 実行可能ファイルを構築するには、以下のように make コマンドを実行します。

```
$ make functions
cc -o functions functions.c
```

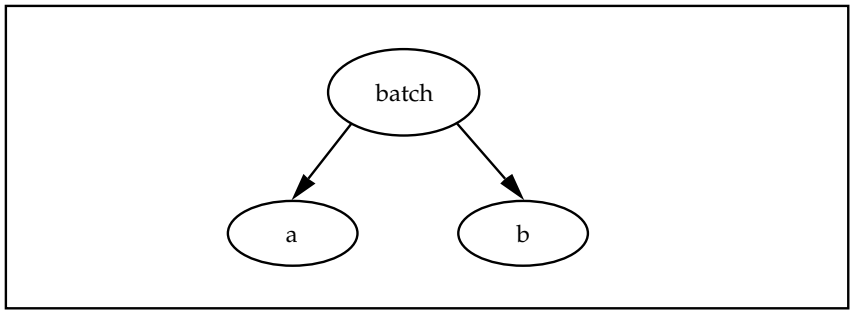
.c ファイルから .o という接尾辞のファイルを構築する際の規則を、.c.o 接尾辞の規則と呼びます。.c ファイルから実行可能プログラムを構築する際の規則を、.c 規則と呼びます。デフォルトの接尾辞のすべての規則は表 4–8 を参照してください。

依存関係の処理

make の開始後、make は最初の依存関係を検査し、ターゲットが検出されるとそのターゲットが処理されます。以下のメークファイルを例として説明します。

```
batch: a b
     touch batch
b:
     touch b
a:
     touch a
c:
     echo "you won't see me"
```

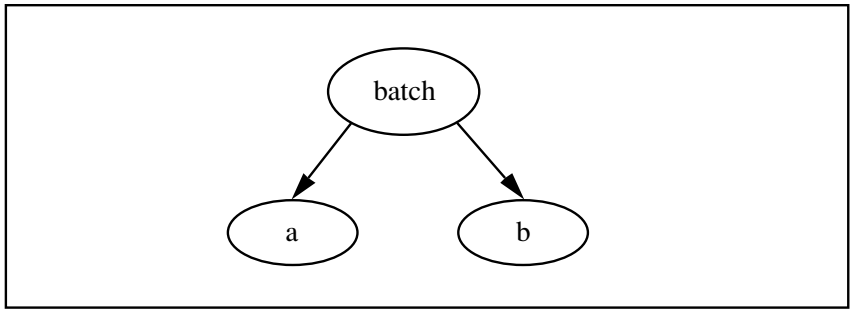
make は、batch というターゲットから処理を開始します。batch には検査されていない依存関係、つまり a と b があるため、make は a と b の依存関係の検査が終了するまで batch を実行しません。



a には依存関係がないため、make は a を処理します。ファイルがない場合は、make はターゲットのエントリの規則を実行します。

```
$ make
touch a
...
```

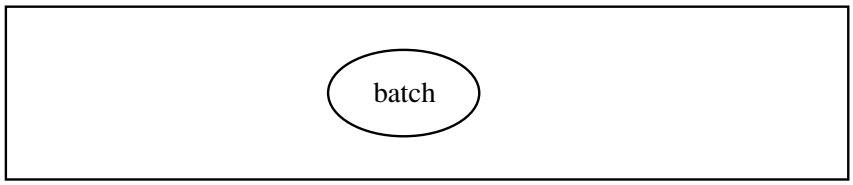
次に、make は親のターゲットである batch に戻ります。まだ検査されていない依存関係 b があるため、make は b に移って検査を行います。



b にも依存関係がないため、make はその規則を実行します。

```
...
touch b
...
```

これで batch のすべての依存関係が検査され (必要であれば) 構築されたため、最後に make は batch を検査します。



make が batch の依存関係のうち少なくとも 1 つを再構築したため、make は batch が最新ではないものとみなして再構築します。つまり make はタイムスタンプを比較する検査を行うので、a または b が、この make の実行では構築されていないが、ディレクトリ内には存在していて batch より新しい場合にも、 batch が再構築されます。

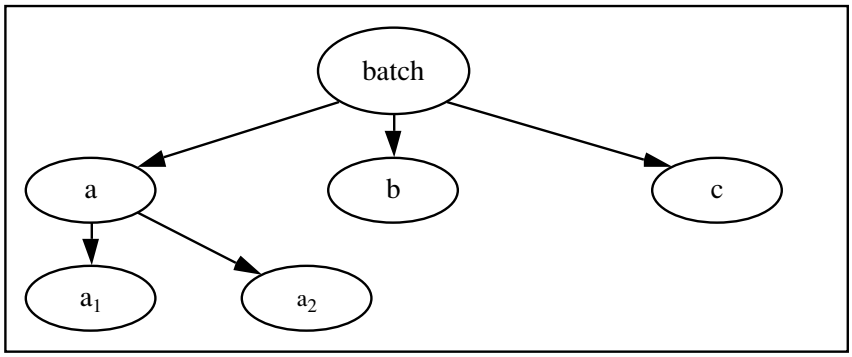
```
...
touch batch
```

依存関係の検査で検出されないターゲットのエントリは処理されません。c のターゲットエントリがメークファイル中にありますが、batch の依存関係の検査において c のエントリは検出されていないため、c の規則は実行されません。make コマンドの引数として入力すると、c を開始ターゲットとして指定することができます。

次の例では、batch のターゲットはファイルを生成せず、複数のターゲットを 1 つのグループとして示すラベルとして使用されます。

```
batch: a b c
a: a1 a2
        touch a
b:
        touch b
c:
        touch c
a1:
        touch a1
a2:
        touch a2
```

この場合は、下図のようにターゲットが検査および処理されます。



要約すると、make は以下のような処理を実行します。

1. batch の依存関係を検査して依存関係を 3 つ検出するので、batch の処理はまだ実行しません。

2. 最初の依存関係である a を検査します。依存関係を 2 つ検出します。同様に、make は次の処理を実行します。

   a. a1 を検査し、必要であればそれを再構築します。

   b. a2 を検査し、必要であればそれを再構築します。

3. a を再構築するかどうかを決定します。
4. b を検査し、必要であればそれを再構築します。
5. c を検査し、必要であれば それを再構築します。
6. make は、依存関係のツリー構造をすべて処理した後に、最上位のターゲットである batch の検査および処理を行います。batch に規則が含まれている場合は、make はその規則を実行します。この例では、batch には規則が含まれていないため、make は規則を実行せずにその batch を再構築します。batch に依存するすべてのターゲットも再構築されます。

空の (NULL) 規則

ターゲットに規則が含まれていない場合は、make は暗黙の規則を選択してターゲットを構築します。適切な暗黙の規則を make が検出できず、規則を取り出すための SCCS 履歴がない場合は、make はターゲットに対応するファイルがないものとし、規則が空であるとみなします。

注 - 規則が空である依存関係を使用して、ターゲットの規則を強制的に実行することができます。たとえば、FORCE という規則は空のターゲットを定義します。

以下のメークファイルを例として説明します。

```
haste: FORCE
        echo "haste makes waste"
FORCE:
```

make は、haste という名前のファイルが最新である場合でも、haste を作成するための規則を実行します。

```
$ touch haste
$ make haste
echo "haste makes waste"
haste makes waste
```

特殊ターゲット

make には、特別な機能を実行するための特殊ターゲットが組み込まれています。たとえば、.PRECIOUS という特殊ターゲットは、make が割り込まれたときにライブラリファイルを保存するように make に指示します。

特殊ターゲットには、以下のような特徴があります。

- 名前がピリオド (.) で始まる
- 依存関係がない
- メークファイルの任意の場所に記述できる

表 4–3 に、特殊ターゲットの一覧を示しています。

不明なターゲット

ターゲットの名前がコマンド行またはメークファイル中の依存関係リストで指定されており、かつ以下のいずれかに該当する場合には、make は処理を停止してエラーメッセージを表示します。

- 作業中のディレクトリにファイルがない
- ターゲットまたは依存関係のエントリがない
- 暗黙の規則のファイル名接尾辞の定義に一致しない
- SCCS 履歴ファイルがない
- .DEFAULT という特殊ターゲット用に指定された規則がない

```
$ make believe
make: Fatal error: Don't know how to make target 'believe'.
```

注 - ただし、–k オプションが有効である場合は、make はエラーが発生したターゲットに依存しない他のターゲットに対して処理を続行します。

重複するターゲット

メークファイル中で同じ名前のターゲットを複数回記述することができます。次に例を示します。

```
foo:  dep_1
foo:  dep_2
foo:
     touch foo
```

上記の例は、次の例と同じ結果になります。

```
foo:  dep_1 dep_2
      touch foo
```

ただし通常は、メークファイルをわかりやすくするために、ターゲットは 1 回だけ記述することをお勧めします。

make の予約語

以下の表に示す語は、make によって予約されています。

表 4–3 make の予約語

.BUILT_LAST_MAKE_RUN	.DEFAULT	.DERIVED_SRC
.DONE	.IGNORE	.INIT
.KEEP_STATE	.MAKE_VERSION	.NO_PARALLEL
.PRECIOUS	.RECURSIVE	.SCCS_GET
.SILENT	.SUFFIXES	.WAIT
FORCE	HOST_ARCH	HOST_MACH
KEEP_STATE	MAKE	MAKEFLAGS
MFLAGS	TARGET_ARCH	TARGET_MACH
VERSION_1.0	VIRTUAL_ROOT	VPATH

コマンドを表示せずに実行する

規則内のコマンド行の最初に @ を挿入すると、実行中にその行を表示しないようにできます。次に例を示します。

```
quiet:
        @echo you only see me once
```

結果は、以下のようになります。

```
$ make quiet
        you only see me once
```

make の実行時にコマンドを表示しないように指定する場合は、-s オプションを使用します。すべての実行でコマンド行の表示を禁止する場合は、.SILENT という特殊ターゲットをメークファイルに追加します。

```
.SILENT:
quiet:
        echo you only see me once
```

特殊な機能を持つターゲットは、名前の最初がドット (.) になっています。ドットで始まるターゲット名は、コマンド行の引数として明示的に指定した場合を除き、開始ターゲットとしては使用されません。コマンドがゼロ以外の終了コードを返した場合は、make は通常はエラーメッセージを出力して停止します。たとえば、次のようなターゲットがある場合を考えます。

```
rmxyz:
     rm xyz
```

この例で、xyz という名前のファイルがない場合は、make は rm が終了ステータスを返した後に停止します。

```
$ ls xyz
xyz not found
$ make rmxyz
rm xyz
rm: xyz: No such file or directory
*** Error code 1
make: Fatal error: Command failed for target `rmxyz'
```

注 - - と @ の両方を組み合わせて記述することも可能です。

コマンドの終了コードに関係なく処理を続行するには、タブの後に記述する最初の文字としてハイフン (-) を使用します。

```
rmxyz:
        -rm xyz
```

この例では、make が受け取った終了コードを示す警告メッセージが表示されます。

```
$ make rmxyz
rm xyz
rm: xyz: No such file or directory
*** Error code 1 (ignored)
```

注 - メークファイルをテストしている場合を除き、通常はゼロ以外のエラーコードをすべて無視することは避けてください。

–i オプションを使用すると、make がすべてのエラーコードを無視するように指定できます。ただし、通常はこのオプションは使用しないでください。また、.IGNORE という特殊ターゲットを記述すると、メークファイルの処理時に make が終了コードを無視するように指定できます。このオプションも、通常は使用しないでください。

ターゲットのリストを処理している際にゼロ以外のリターンコードが返されたときに、処理全体を停止するのではなく後に make が次のターゲットを続けて処理するように指定するには、–k オプションを使用します。

SCCS ファイルの自動取り出し

依存関係リスト中にソースファイルの名前が指定されている場合は、make はそのソースファイルを他のターゲットと同様に処理します。この場合は、ソースファイルはディレクトリ内にあることが前提になっているため、ソースファイルのエントリをメークファイルに追加する必要はありません。

ターゲットに依存関係がなく、ターゲットがディレクトリ内にある場合は、make はファイルを最新のものとみなします。ただしソースファイルが SCCS 管理されている場合は、make はさらにいくつかの検査を行い、ソースファイルが最新であることを確認します。ファイルがないあるいは履歴ファイルの方が新しい場合は、make は自動的に以下のコマンドを実行して、最新のバージョンを取り出します。

sccs get -s *filename* **-G***filename*

注 - 他のバージョンの make では、SCCS ファイルの自動取り出しは特定の暗黙の規則だけで使用できる機能です。また旧バージョンとは異なり、このバージョンの make は履歴ファイルを sccs ディレクトリ内でのみ検索します。作業中のディレクトリ内の履歴ファイルは無視されます。

ただしソースファイルが書き込み可能な場合は、make は新しいバージョンを取り出しません。

```
$ ls SCCS/*
SCCS/s.functions.c
$ rm -f functions.c
$ make functions
sccs get -s functions.c -Gfunctions.c
cc -o functions functions.c
```

make は、取り出したバージョンのタイムスタンプと履歴ファイルのタイムスタンプを調べます。ディレクトリ内のバージョンが最後にチェックインされた最新バージョンかどうかは調べません。したがって、他のユーザーが日付を指定してファイルを取り出した (sccs get -c) 場合でも、make はそれが最新のバージョンであるかどうかを調べないので、最新でないバージョンのプログラムまたはオブジェクトファイルが構築されることがあります。確実に最新バージョンを構築するには、sccs get SCCS または sccs clean のいずれかのコマンドを実行してから make を実行します。

SCCS ファイル取り出しの抑止

SCCS ファイルの取り出しコマンドは、デフォルトのメークファイル (/usr/share/lib/make/make.rule) の .SCCS_GET という特殊ターゲット用の規則で指定されています。SCCS ファイルの自動取り出しを抑止するには、メークファイルでこのターゲットのエントリに空の規則を指定します。

```
# Suppress sccs retrieval.
.SCCS_GET:
```

パラメータの引き渡し：make の簡単なマクロ

make のマクロ置換は、メークファイル内でパラメータをコマンド行に渡す際に便利です。cc –O オプションを使用して、program という最適化されたプログラムを、コンパイルする場合を例に挙げます。メークファイルに以下の例のようなマクロ参照を functions のターゲットに追加して、パラメータをコマンド行に渡すことができます。

```
functions: functions.c
        cc $(CFLAGS) -o functions functions.c
```

マクロ参照は、メークファイル内あるいは make コマンドの引数でユーザーが定義した値に対応する可変部分として機能します。make で CFLAGS マクロの値を定義すると、make はマクロ参照を指定された値に置換します。

```
$ rm functions
$ make functions "CFLAGS= -O"
cc -O -o functions functions.c
```

注 - CFLAGS マクロへの参照は、.c と .c.o の両方の暗黙の規則に含まれています。

注 - コマンド行で指定できる引数は 1 つだけです。この例では、二重引用符で囲んである部分が引数定義を示しています。

マクロが未定義の場合は、make はマクロの参照先を空白の文字列とみなします。

メークファイル中にマクロを定義することもできます。デフォルトで最適化されたオブジェクトコードを make が生成するように、CFLAGS を –O に設定するなどの用途があります。

```
CFLAGS= -O
functions: functions.c
        cc $(CFLAGS) -o functions functions.c
```

make のコマンド行の引数として指定されたマクロ定義は、条件付きマクロ定義を除き、メークファイル内の他の定義よりも優先されます。

たとえば、dbx または dbxtool を使用してデバッグするために functions をコンパイルするには、コマンド行で CFLAGS の値を –g として定義します。

```
$ rm functions
$ make CFLAGS=-g
cc -g -o functions functions.c
```

gprof を使用してプロファイル用に functions をコンパイルするには、–O および –pg の両方を CFLAGS の値として指定します。

マクロ参照は、マクロ名が 2 文字以上の長さの場合は、括弧で囲む必要があります。マクロ名が 1 文字だけの場合は、括弧を省略できます。括弧の代わりに中括弧 { と } を使用することもできます。たとえば、'$X'、'$(X)'、'${X}' はすべて同じ意味です。

.KEEP_STATE とコマンドの依存関係の検査

通常の依存関係の検査に加えて、特殊ターゲットの .KEEP_STATE を使用してコマンドの依存関係を検査することができます。この検査を実行すると、make は各ターゲットファイルと依存ファイルとを検査するだけでなく、規則中の各コマンド行と、ターゲットを最後に構築したときに実行したコマンド行の比較も行います。比較の結果は、現在のディレクトリの .make.state ファイルに保存されます (136 ページの「状態ファイル」を参照してください)。

以下のメークファイルを例として説明します。

```
CFLAGS= -O
.KEEP_STATE:
```

```
functions: functions.c
        cc -o functions functions.c
```

以下に、上記のメークファイルで make コマンドを実行した結果を示します。

```
$ make
cc -O -o functions functions.c
$ make CFLAGS=-g
cc -g -o functions functions.c
$ make "CFLAGS= -O -pg"
cc -O -pg -o functions functions.c
```

プログラムのコンパイル時には、ユーザーが指定したオプションが優先されて使用されます。

.KEEP_STATE を有効にしてから最初に make を実行したときには、必要な情報を収集して記録するため、すべてのターゲットが再コンパイルされます。KEEP_STATE 変数は、環境変数から取り込んだ場合には、.KEEP_STATE ターゲットと同様に機能します。

選択した行に対してコマンドの依存関係の検査を強制または抑止する

指定したコマンド行のコマンドの依存関係の検査を抑止するには、タブの後の最初の文字として疑問符を挿入します。

動的なマクロである $? を含む行では、コマンドの依存関係の検査が自動的に抑止されます。このマクロは、現在のターゲットよりも新しい依存関係のリストです。このリストの内容は、make の実行ごとに異なる場合があります。

このマクロを含む行に対してコマンドの依存関係の検査を強制的に実行するには、そのコマンド行の (タブの後) の最初の文字として、! を挿入します。

状態ファイル

.KEEP_STATE が有効な場合は、make は .make.state という名前の状態ファイルを現在のディレクトリに生成します。このファイルには、.KEEP_STATE が有効になっている間に処理されたすべてのターゲットおよびそのターゲットを構築する規則が、メークファイルの書式で記述されます。この状態ファイルの整合性を保つために、.KEEP_STATE をメークファイルに追加した後は、内容を変更しないでください。

注 - このターゲットは旧バージョンの make では無視されるため、互換性の問題は生じません。このターゲットは、依存するターゲットがなく、規則が空 (NULL) で、それ自身の依存関係がないので、旧バージョンでは余分なターゲットとして処理されます。名前の先頭にドットがあるため、開始ターゲットして使用されません。

.KEEP_STATE と隠れた依存関係

ヘッダーを挿入するための #include 指令が C のソースファイルに記述されている場合は、ターゲットはヘッダーファイルとソースファイルの両方に依存します。このようなヘッダーファイルは、コンパイルのコマンド行ではソースとして明示的に表示される場合があるため、これらの関係を隠れた依存関係と呼びます。.KEEP_STATE が有効の場合は、さまざまなコンパイラおよびコンパイルのプリプロセッサは、どの隠れた依存ファイルがどのターゲットに対応しているかを make に通知します。

.KEEP_STATE は、この情報を状態ファイルの依存関係のリストに追加します。以降の実行では、これらの追加された依存関係は通常の依存関係と同様に処理されます。この機能により、各ターゲットの隠れた依存関係のリストが常に正確で最新の状態で自動的に保持されます。また、完全な依存関係のリストを生成するために、旧バージョンのメークファイルでのように複雑なスキームを使用する必要がなくなります。

隠れた依存関係を持つターゲットを make が最初に処理する際には、状態ファイルにはまだそれらの記録がないため、若干不便な場合があります。ヘッダーがなく、make がその記録を持っていない場合は、ターゲットのコンパイル前に SCCS からヘッダーファイルを取り出す必要があることを make は認識できません。

SCCS の履歴ファイルがある場合でも、依存関係のリストや状態ファイルに履歴ファイルはまだ含まれていないため、現在のバージョンは取り出されません。C のプリプロセッサがヘッダーを挿入する際にはヘッダーが見つからないでのコンパイルは失敗します。

hidden.h というヘッダーを挿入する #include 指令が functions.c に追加されて、それ以降に何らかの理由で make を実行する前にファイル hidden.h が削除された場合は、make の実行結果は以下のようになります。

```
$ rm -f hidden.h
$ make functions
cc -O -o functions functions.c
functions.c: 2: Can't find include file hidden.h
make: Fatal error: Command failed for target 'functions'
```

これを回避するには、新しいヘッダーが存在することを確認してから make を実行してください。またコンパイルが失敗する場合でも、SCCS がヘッダーを管理していれば、SCCS から手動でヘッダーを取り出すことができます。

```
$ sccs get hidden.h
1.1
10 lines
$ make functions
cc -O -o functions functions.c
```

2 回目以降に make を実行する際には隠れた依存関係として状態ファイルに記述されているため、make は構築を実行またはヘッダーを取り出すことができます。

隠れた依存関係の検査では、$? マクロは隠れた依存関係の依存ファイル名をインクルードします。このため、$? を使用する既存のメークファイルで予期しない動作が生じる場合があります。

.INIT と隠れた依存関係

前述のどちらの方法でも、他の (インクルード) ディレクトリの状態が一定でないために、ローカルディレクトリで最初に実行する make が失敗する場合があるという問題があります。このため、最初に行う構築をだれかが監視する必要があります。これを回避するには、.INIT ターゲットを使用して、既知の隠れた依存ファイルを SCCS から取り出します。.INIT は、make の実行開始時に依存関係とともに構築される特殊ターゲットです。hidden.h が必ず存在するようにするには、以下の行をメークファイルに追加します。

```
.INIT:     hidden.h
```

make の実行に関する情報を表示する

–n オプションを使用して make を実行すると、make が実行するコマンドを実際には実行せずに表示だけを行います。このオプションは、メークファイルのマクロが意図しているとおりに展開されているかどうかを確認する際に便利です。以下のメークファイルを例に示します。

```
CFLAGS= -O

.KEEP_STATE:

functions: main.o data.o
        $(LINK.c) -o functions main.o data.o
```

make -n を実行すると、以下のように表示されます。

```
$ make -n
cc -O -c main.c
```

```
cc -O -c data.c
cc -O -o functions main.o data.o
```

注 - ただし、1 つ例外があります。MAKE マクロへの参照 ($(MAKE) や ${MAKE} など) を含むコマンド行は、make に –n オプションを付けた場合にも実行されます。以下のような行をメークファイルに記述することは避けてください。

```
$(MAKE) ; rm -f *
```

注 - MAKEFLAGS という環境変数を設定すると、その値が make オプションのリストに追加されるため、処理が複雑になる場合があります。混乱を防ぐために、この変数を設定することは避けてください。

---

make には、make の動作やその理由を確認できるオプションが他にもいくつかあります。

–d　ターゲットが最新でないと make が決定する条件を表示します。–n オプションとは異なり、以下の例のようにターゲットの処理は実行します。このオプションは、環境変数 (デフォルトでは NULL) から取り込んだ MAKEFLAGS マクロの値も表示します。MAKEFLAGS マクロについての詳細は、後述の節で説明しています。

```
$ make -d
MAKEFLAGS value:
    Building main.o using suffix rule for .c.o because
it is out of date relative to main.c
cc -O -c main.c
    Building functions because it is out of date
relative to main.o
    Building data.o using suffix rule for .c.o because
it is out of date relative to data.c
cc -O -c data.c
    Building functions because it is out of date
relative to data.o
cc -O -o functions main.o data.o
```

–dd　make が検査する、すべての依存関係 (隠れた依存関係も含む) を詳細に表示します。

–D　メークファイルのテキストをメークファイル読み取り時に表示します。

–DD	メークファイル、デフォルトのメークファイル、状態ファイル、実行中の make における隠れた依存関係を表示します。
–f ***makefile***	make が (makefile または Makefile ではなく) *makefile* に指定した名前のメークファイルを使用します。

注 - 複数のメークファイルを指定するには、–f オプションをその数だけ指定します。指定したメークファイルは連結されて処理されます。

–K ***makestatefile***	*makestatefile* がディレクトリの場合は、make はそのディレクトリ内の .make.state ファイルに KEEP_STATE の情報を書き込みます。*makestatefile* がファイルの場合は、make は KEEP_STATE 情報を *makestatefile* に書き込みます。
–p	すべてのマクロ定義およびターゲットのエントリを表示します。
–P	デフォルトのターゲットまたは指定したターゲットのすべての依存関係のツリーを表示します。

–t オプションを使用すると、make の処理を省略できます。–t を使用して make を実行すると、make はターゲットを構築する規則を実行しません。その代わりに touch を使用して、依存関係の検査で検出した各ターゲットの変更時間を更新します。また、構築したターゲットに応じて状態ファイルを更新します。ターゲットのエントリに対応するファイルがない場合は、touch によってファイルが作成されます。

注 - 副作用が生じる場合があるため、make の –t (touch) オプションは通常は使用しないでください。

---

make -t を使用しない例を以下に示します。make によって作成されたターゲットファイルを削除してディレクトリ内を掃除する clean というターゲットがあるとします。

```
clean:
        rm functions main.o data.o
```

注 - clean は、派生ファイルを削除するターゲットを示す慣用名です。clean は、新しく最初から構築を行う場合に便利です。

ここで、以下のような無意味なコマンドを実行したとします。

```
$ make -t clean
touch clean
$ make clean
'clean' is up to date
```

この場合、ディレクトリ内を掃除するターゲットを再度機能させるには、clean というファイルを削除する必要があります。

–q	ターゲットファイルが最新かどうかに応じて、ゼロまたはゼロ以外の終了ステータスを返します。
–r	デフォルトのメークファイル (/usr/share/lib/make/make.rules) の読み取りを抑止します。
–S	ゼロ以外の終了ステータスをコマンドが返したときに、処理を停止して –K オプションの効果を取り消します。

make を使用してプログラムをコンパイルする

前述の例では、明示的なターゲットのエントリと暗黙の規則を使用して簡単な 1 つのソースファイルから C プログラムをコンパイルする方法を説明しました。ただし、ほとんどの C プログラムは、複数のソースファイルからコンパイルされます。また、多くのプログラムでは、標準のシステムライブラリやユーザー定義のライブラリのルーチンがインクルードされています。

cc コマンドを 1 回実行して 1 つのプログラムを再コンパイルおよびリンクする方が簡単な場合がありますが、通常は、複数のソースファイルからプログラムを何回かに分けて段階的にコンパイルする方が便利です。各ソースファイルを別々のオブジェクト (.o) ファイルにコンパイルしてから、それらのオブジェクトファイルをリンクして実行可能ファイル (a.out) を作成します。この方法では、より多くのディスク容量が必要ですが、後で (繰り返して) 再コンパイルを行う際に、ソースが変更

されたオブジェクトファイルだけに対してコンパイルを実行すればよいため、コンパイルの時間を節約することができます。

簡単なメークファイルの例

単純ですが実用的なメークファイルの例を以下に示します。

表 4–4 C ソースのコンパイル用のメークファイル (すべて明示的)

```
# 2 つの C ソースファイルから 1 つのプログラムを
# コンパイルするメークファイル

.KEEP_STATE

functions: main.o data.o
	cc -O -o functions main.o data.o
main.o: main.c
	cc -O -c main.c
data.o: data.c
	cc -O -c data.c
clean:
	rm functions main.o data.o
```

この例では、make は main.o および data.o というオブジェクトファイルと、functions という実行可能ファイルを生成します。

```
$ make
cc -o functions main.o data.o
cc -O -c main.c
cc -O -c data.c
```

make の定義済みマクロを使用する

次の例は、前述の例と機能は同一ですが、コンパイルコマンドの代わりに make の定義済みマクロを使用しています。定義済みマクロを使用すると、コンパイル環境が変更されたときに、メークファイルを編集する必要がありません。また、コマンド行からコンパイラのオプションを指定するための CFLAGS マクロ (およびその他の FLAGS マクロ) を使用できます。定義済みマクロは、make の暗黙の規則でも頻繁に使用されます。後述のメークファイルで使用されている定義済みマクロを次に示します。これらの定義済みマクロ[1]は、C プログラムのコンパイル全般で便利です。

1. このバージョンの make では定義済みマクロが、旧バージョンよりも頻繁に使用されています。ここで示す定義済みマクロの一部は、旧バージョンでは使用できません。

COMPILE.c	cc コマンド行。以下のように CC、CFLAGS、CPPFLAGS と –c オプションで構成されます。 `COMPILE.c=$(CC) $(CFLAGS) $(CPPFLAGS) -c` マクロの名前の COMPILE という部分は、マクロがコンパイルの (オブジェクト、つまり .o ファイルを作成する) コマンド行を表すマクロであることを示しています。.c という接尾辞は、コマンド行が .c (C ソース) ファイルに適用されることを示すニーモニックです。

注 - 名前が FLAGS という文字列で終わるマクロは、関連するコンパイラコマンドのマクロにオプションを引き渡します。これらのマクロは、整合性と移植性を保つために使用します。また、メークファイル中にこれらのマクロに適したデフォルト値を記述しておくと便利です。すべての定義済みマクロのリストを、表 4–10 に示しています。

LINK.c	COMPILE.c などのオブジェクトファイルをリンクするための基本的な cc コマンド行です。–c オプションは使用せず、LDFLAGS マクロへの参照を使用します。 `LINK.c=$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS)`
CC	値は cc です (この値は、別の C コンパイラのパス名に再定義できます)。
CFLAGS	cc コマンドのオプションです。デフォルトでは空です。
CPPFLAGS	cpp のオプションです。デフォルトでは空です。
LDFLAGS	リンカー ld のオプションです。デフォルトでは空です。

表 4–5　定義済みマクロを使用して C ソースをコンパイルするメークファイル

```
# 2 つの C ソースをコンパイルするメークファイル
CFLAGS= -O
.KEEP_STATE:

functions: main.o data.o
	$(LINK.c) -o functions main.o data.o
main.o: main.c
	$(COMPILE.c) main.c
data.o: data.c
	$(COMPILE.c) data.c
clean:
	rm functions main.o data.o
```

暗黙の規則を使用してメークファイルを簡素化する：接尾辞の規則

main.o および data.o を .c ファイルからコンパイルするコマンド行は、このバージョンでは接尾辞 .c.o の規則と機能的に同一であるため、ターゲットのエントリが重複します。make は、そのようなコマンド行がメークファイルに記述されているかどうかに関係なく、同じようにコンパイルを実行します。次のメークファイルは、.c.o の規則によって各オブジェクトファイルをコンパイルし、各オブジェクト用のコマンド行を省略しています。

表 4–6　接尾辞の規則を使用して C ソースをコンパイルするメークファイル

```
# 接尾辞の規則を使用して 2 つの C ソースから
# 1 つのプログラムをコンパイルするメークファイル
CFLAGS= -O

.KEEP_STATE:

functions: main.o data.o
	$(LINK.c) -o functions main.o data.o
clean:
	rm functions main.o data.o
```

注 - 接尾辞のすべての規則は、表 4–8 を参照してください。

---

make が main.o および data.o という依存関係を処理する際に、これらの依存関係のターゲットのエントリがありません。make は、適切な暗黙の規則を調べて適用

します。この場合は、接尾辞 .c.o の規則が選択され、名前が同じで接尾辞が .c の依存ファイルから .o ファイルが構築されます。

注 - make は、接尾辞リスト中に記述されている順番に従って使用する依存ファイルおよび接尾辞の規則を決定します。たとえば、main.c と main.s の両方のファイルがディレクトリにある場合は、接尾辞のリストにおいて .c が .s よりも前に記述されているため、make は .c.o の規則を使用します。

まず、make は接尾辞リストにターゲットファイルの接尾辞が含まれているかどうかを調べます。main.o の場合は .o がリストに含まれています。次に、make はターゲットファイルを構築するための接尾辞の規則と、元になる依存ファイルを調べます。依存ファイルの名前は、ターゲットの名前と接尾辞を除いて同一です。この場合は、make は .c.o の規則を調べて main.c という名前の依存ファイルを検出するため、.c.o の規則を使用します。

接尾辞リストは、.SUFFIXES という名前の特殊ターゲットです。SUFFIXES マクロの定義にはさまざまな接尾辞が含まれています。.SUFFIXES は、このマクロに依存するように指定されています。

表 4–7 標準の接尾辞のリスト

```
SUFFIXES= .o .c .c~ .cc .cc~ .C .C~ .y .y~ .l .l~ .s .s~ .sh .sh~ .S .S~ .ln
\  .h .h~ .f .f~ .F .F~ .mod .mod~ .sym .def .def~ .p .p~ .r .r~ \  .cps
.cps~ .Y .Y~ .L .L~ .SUFFIXES: $(SUFFIXES)
```

以下の例は、すべての実行可能プログラムをコンパイルするメークファイルです。各プログラムには、ソースファイルが 1 つずつあります。各実行可能ファイルは、ベース名が同じで接尾辞が .c のソースファイルから構築されます。たとえば、demo_1 は demo_1.c から構築されます。

注 - clean と同様に、all はターゲットの慣用名です。all は、依存関係リストに含まれるすべてのターゲットを構築します。通常は、make と make all は同一の処理を行います。

```
# 1 組の C プログラムをコンパイルするメークファイル。
# 1 つのソース (.c ファイル) から 1 つのプログラムを生成。
CFLAGS= -O
.KEEP_STATE:

all: demo_1 demo_2 demo_3 demo_4 demo_5
```

この場合は、いずれのターゲットにも接尾辞がありません。そのため、make は各ターゲットに空の (NULL) 接尾辞が付いているとして処理します。次に、make は接尾辞の規則と、有効な接尾辞の付いた依存ファイルを検索します。demo_2 の場合は、demo_2.c という名前のファイルを検索します。接尾辞 .c の規則のターゲットエントリと、それに対応する .c ファイルがあるため、make は接尾辞 .c の規則を使用して demo_2.c から demo_2 を構築します。

空の接尾辞のターゲットに明示的な依存関係があるときは、あいまいさを避けるために、make は接尾辞の規則を使用したターゲットの構築は行いません。

```
program: zap
zap:
```

このメークファイルを使用して make を実行しても、何も出力されません。

```
$ make program
$
```

明示的なターゲットのエントリと暗黙の規則の使用について

複数の依存ファイルからターゲットを構築する際には、構築のための規則が含まれている明示的なターゲットを make で指定する必要があります。1 つの依存ファイルからターゲットを構築する際には、多くの場合は暗黙の規則を使用すると便利です。

前述の例で示すように、make は簡単に 1 つのソースファイルを対応するオブジェクトファイルまたは実行可能ファイルにコンパイルします。ただし、複数のオブジェクトファイルをリンクして実行可能ファイルを作成するための情報は make には組み込まれていません。また、make は依存関係の検査で検出したオブジェクトファイルだけをコンパイルします。make は、開始点、つまりリスト内の各オブジェクトファイル (最終的には各ソースファイル) が依存するターゲットを必要とします。

したがって、複数の依存ファイルからターゲットを構築するには、照合順と、ターゲットの依存ファイルを示す依存関係リストを提供する明示的な規則が必要です。各依存ファイルが 1 つのソースから構築される場合は、暗黙の規則を使用できます。

暗黙の規則と動的なマクロ

make は、複数のマクロをターゲット単位で動的に管理します。動的なマクロは、特に暗黙の規則の定義において頻繁に使用されます。マクロの意味を理解することは重要です。

注 - 動的なマクロは、メークファイルで明示的に定義されていないため、$ 記号を接頭辞として追加して (つまり、マクロ参照であることを示して) 記述します。

動的なマクロは、以下のとおりです。

$@	現在のターゲットの名前です。
$?	ターゲットよりも新しい依存関係のリストです。
$<	make が暗黙の規則を使用するときの依存ファイル名を示します。
$*	現在のターゲットのベース名 (接尾辞を除いたターゲット名) です。
$%	処理されるライブラリメンバーの名前を示します。詳細は、158ページの「オブジェクトライブラリの構築」を参照してください。

暗黙の規則は、ターゲット名または依存ファイル名を規則内のコマンド行に指定するために、これらの動的なマクロを使用します。以下の .c.o の規則を例として示します。

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

$< は、現在のターゲットの依存ファイル (この場合は .c ファイル) の名前に置換されます。

注 - OUTPUT_OPTION というマクロは、デフォルトでは値が空です。このマクロは、CFLAGS と同様の機能を持っていますが、引数を –o コンパイラオプションに渡してコンパイラの出力先ファイル名を指定するための独立したマクロとして提供されています。

この .c の規則では、$@ は現在のターゲット名に置換されます。

```
.c:
        $(LINK.c) $< -o $@
```

$< および $* のいずれのマクロも、接尾辞のリストでの順序に応じて値が異なるため、明示的なターゲットのエントリでこれらのマクロを使用すると予期しない結果

になる場合があります。関連するファイル名からファイル名を正確に取得する方法については、161ページの「マクロ参照での接尾辞の置換」を参照してください。

動的なマクロの修飾子

F および D を参照に含めることによって動的なマクロを修飾できます。処理されるターゲットがパス名で指定されている場合は、$(@F) はファイル名部分、$(@D) はディレクトリ部分をそれぞれ示します。ターゲット名に / という文字が含まれていない場合は、$(@D) にドット (.) が値として割り当てられます。たとえば、/tmp/test という名前のターゲットでは、$(@D) の値は /tmp、$(@F) の値は test になります。

動的なマクロと依存関係リスト：遅延マクロ参照

動的なマクロは、すべてのどのようなターゲットを処理する際にも割り当てられます。マクロは、ターゲットの規則ではそのまま、依存関係リストでは参照の先頭に $ を追加して使用できます。$$ で始まる参照を、マクロへの遅延参照と呼びます。以下に例を示します。

```
x.o y.o z.o: $$@.BAK
        cp $@.BAK $@
```

このエントリは、x.o.BAK、y.o.BAK、z.o.BAK からそれぞれ x.o、y.o、z.o を取得するために使用できます。

依存関係リストの読み取り

遅延マクロ参照を使用できるのは、make が、メークファイル全体を最初に読み込む際と、ターゲットの依存関係を処理する際の合計 2 回依存関係リストを読み込むためです。make は、リストを読み込むたびにマクロ展開を実行します。動的なマクロは、マクロへの参照が 2 回目の読み取りまで遅延される場合を除き、最初の読み取りではまだ定義されていないため、空の文字列に展開されます。

$$ という文字列は、定義済みマクロの '$' への参照です。このマクロは値が '$' になっています。make が最初の読み取りでこのマクロを展開する際には、$$@ は $@ に展開されます。依存関係の検査では、その $@ というマクロ参照に動的に値が割り当てられるときに、make は参照をその値に展開します。

make は、最初の読み取りではターゲットのターゲット名部分だけを評価します。ターゲット名として遅延マクロ参照を使用すると、正しい結果が得られません。

```
NONE= none
all: $(NONE)

$$(NONE):
        @: this target's name isn't 'none'
```

このメークファイルは、以下のような結果になります。

```
$ make
make: Fatal error: Don't know how to make target 'none'
```

規則の評価

make は、規則の実行時にそのコマンドの適用ごとに 1 回だけ、ターゲットエントリの規則部分を評価します。この場合も、make への遅延マクロ参照を使用すると誤った結果になります。

接尾辞規則の橋渡し

接尾辞の規則では、複数の規則の関係から自動的にそれらの中間の規則を生成するということはありません。たとえば、接尾辞が .X というファイルから接尾辞が .Y というファイルを構築する接尾辞の規則と、接尾辞が .Y というファイルから接尾辞が .Z というファイルを構築する規則がある場合に、make は .X ファイルから .Z ファイルを作成するという規則を、上記 2 つの規則から生成するということは行いません。中間の処理をターゲットとして指定する必要があります。中間の処理のエントリには空の規則を使用できます。

```
trans.Z:
trans.Y:
```

この例では、trans.Y が存在していれば trans.Z はそのファイルから構築されます。trans.Y がターゲットエントリとして記述されていない場合は、使用する依存ファイルがないために make が "don't know how to build" というエラーを表示して失敗する場合があります。trans.Y がターゲットエントリとして記述されている場合は、trans.Y が最新でないまたは存在しないときに、make が trans.Y を構築します。メークファイルで規則が指定されていないため、make は適切な暗黙の規則を使用します。この場合は .X.Y の規則が使用されます。trans.X が存在する (または SCCS から取り出すことができる) 場合に、make は trans.Y および trans.Z の両方を必要に応じて再構築します。

接尾辞の規則を追加する

make には多数の便利な接尾辞の規則が提供されていますが、ユーザー定義の規則を追加することもできます。ただし、暗黙の規則を追加する場合は、パターンマッチングの規則が適しています (151ページの「パターンマッチングの規則: 接尾辞の規則の代替」を参照してください)。この節では、暗黙の規則をメークファイルに追加する従来の方法を説明しています。旧バージョンの make と互換性がある暗黙の規則を記述する必要がない場合をは、この節をとばして次の節に進んでも構いません。

接尾辞の規則は、2 段階の手順で行います。まず、ターゲットと依存関係の両方のファイルの接尾辞を、.SUFFIXES という特殊ターゲットの依存関係として指定し、接尾辞のリストに追加します。依存関係リストには指定した内容が蓄積されていくため、以下のようにこのターゲットのエントリを追加すると、リストに接尾辞が追加されます。

```
.SUFFIXES:  .ms .tr
```

次に、接尾辞の規則を指定するターゲットエントリを追加します。

```
ms.tr:
        troff -t -ms $< > $@
```

これらのエントリを記述したメークファイルを使用して、ms というマクロ (.ms ファイル) が含まれている文書ソースファイルを troff 出力ファイル (.tr ファイル) にフォーマットできます。

```
$ make doc.tr
troff -t -ms doc.ms > doc.tr
```

接尾辞リストのエントリは、SUFFIXES マクロに指定されています。接尾辞をリストの最初に挿入するには、まず依存関係のない .SUFFIXES ターゲットエントリを指定して、その値を消去します。依存関係リストには蓄積されていきますが、この場合は例外的に値が上書きされます。以下のように依存関係および規則のないターゲットエントリを指定することによって、既存の定義を消去することができます。

```
.SUFFIXES
```

次に、以下のようにもう 1 つ別のエントリに新しい接尾辞、その後に SUFFIXES マクロへの参照を続けて記述して、接尾辞を追加することができます。

```
.SUFFIXES:
.SUFFIXES: .ms .tr $(SUFFIXES)
```

パターンマッチングの規則: 接尾辞の規則の代替

パターンマッチングの規則は、機能的には暗黙の規則と似ています。パターンマッチングの規則では、ターゲットと依存関係のファイルの関係を、接頭辞 (パス名を含む) と接尾辞のいずれかまたは両方を基準にして指定できるため、暗黙の規則よりも記述が簡単で強力です。パターンマッチングの規則は、次の形式で記述するターゲットエントリです。

```
tp%ts: dp%ds
     rule
```

ここで、tp および ts は、ターゲットの名前の接頭辞および接尾辞 (省略可能) です。dp および ds は、依存関係の名前の接頭辞および接尾辞 (省略可能) です。% は、両方に共通するベース名を示すワイルドカードです。

注 - make は、接尾辞の規則の前にパターンマッチングの規則を調べます。このため、標準の暗黙の規則を無効にすることもできますが、通常は無効にしないでください。

ターゲットを構築する規則がない場合は、make はパターンマッチングの規則を検索してから、接尾辞の規則を調べます。使用可能であれば、パターンマッチングの規則が使用されます。

パターンマッチングの規則のターゲットエントリに規則が含まれていない場合は、make は、規則のない明示的なターゲットエントリを持つものとして、ターゲットファイルを処理します。したがって、make は接尾辞の規則を検索し、SCCS からターゲットファイルのバージョンを取り出し、空の規則を持つものとして最終的にターゲットを処理します (ターゲットは、その実行においては更新されている (最新である) とみなされます)。

troff ソースファイルを troff 出力ファイルにフォーマットするパターンマッチングの規則は、以下のとおりです。

```
%.tr: %.ms
     troff -t -ms $< > $@
```

make のデフォルトの接尾辞の規則と定義済みマクロ

以下の表に、デフォルトのメークファイル (/usr/share/lib/make/make.rules) に記述されている標準の接尾辞の規則と定義済みマクロを示します。

表 4–8 標準の接尾辞の規則

用途	接尾辞規則の名前	コマンド行
アセンブリファイル	.s.o	$(COMPILE.s) -o $@ $<
	.s	$(COMPILE.s) -o $@ $<
		$(COMPILE.s) -o $% $<
	.s.a	$(AR) $(ARFLAGS) $@ $%
		$(RM) $%
	.S.o	$(COMPILE.S) -o $@ $<
		$(COMPILE.S) -o $% $
	.S.a	$(AR) $(ARFLAGS) $@ $%
		$(RM) $%
C ファイル (.c の規則)	.c	$(LINK.c) -o $@ $< $(LDLIBS)
	.c.ln	$(LINT.c) $(OUTPUT_OPTION) -i $<
	.c.o	$(COMPILE.c) $(OUTPUT_OPTION) $<
	.c.a	$(COMPILE.c) -o $% $<
		$(AR) $(ARFLAGS) $@ $%
		$(RM) $%
C++ ファイル	.cc	$(LINK.cc) -o $@ $< $(LDLIBS)
	.cc.o	$(COMPILE.cc) $(OUTPUT_OPTION) $<
	.cc.a	$(COMPILE.cc) -o $% $<
		$(AR) $(ARFLAGS) $@ $%
		$(RM) $%
C++ ファイル (SVR4 書式)	.C	$(LINK.C) -o $@ $< $(LDFLAGS) $*.c
	.C.o	$(COMPILE.C) $<
	.C.a	$(COMPILE.C) $<
		$(AR) $(ARFLAGS) $@ $*.o
		$(RM) -f $*.o

表 4–8 標準の接尾辞の規則 続く

用途	接尾辞規則の名前	コマンド行
FORTRAN 77 ファイル	.cc.o	$(LINK.f) -o $@ $< $(LDLIBS)
	.cc.a	$(COMPILE.f) $(OUTPUT_OPTION) $<
		$(COMPILE.f) -o $% $<
		$(AR) $(ARFLAGS) $@ $%
		$(RM) $%
	.F	$(LINK.F) -o $@ $< $(LDLIBS)
	.F.o	$(COMPILE.F) $(OUTPUT_OPTION) $<
	.F.a	$(COMPILE.F) -o $% $<
		$(AR) $(ARFLAGS) $@ $%
		$(RM) $%

表 4–9 標準の接尾辞の規則

用途	接尾辞規則の名前	コマンド行
lex ファイル	.l	$(RM) $*.c
		$(LEX.l) $< > $*.c
		$(LINK.c) -o $@ $*.c $(LDLIBS)
		$(RM) $*.c
	.l.c	$(RM) $@
		$(LEX.l) $< > $@
	.l.ln	$(RM) $*.c
		$(LEX.l) $< > $*.c
		$(LINT.c) -o $@ -i $*.c
		$(RM) $*.c
	.l.o	$(RM) $*.c
		$(LEX.l) $< > $*.c
		$(COMPILE.c) -o $@ $*.c
		$(RM) $*.c
	.L.C	$(LEX) $(LFLAGS) $<
	.L.o	$(LEX)(LFLAGS) $<
		$(COMPILE.C) lex.yy.c
	.L.o	rm -f lex.yy.c
		mv lex.yy.o $@
Modula 2 ファイル	.mod	$(COMPILE.mod) -o $@ -e $@ $<
	.mod.o	$(COMPILE.mod) -o $@ $<
	.def.sym	$(COMPILE.def) -o $@ $<
NeWS	.cps.h	$(CPS) $(CPSFLAGS) $*.cps
Pascal ファイル	.p	$(LINK.p) -o $@ $< $(LDLIBS)
	.p.o	$(COMPILE.p) $(OUTPUT_OPTION) $<
Ratfor ファイル	.r	$(LINK.r) -o $@ $< $(LDLIBS)
	.r.o	$(COMPILE.r) $(OUTPUT_OPTION) $<
	.r.a	$(COMPILE.r) -o $% $<
		$(AR) $(ARFLAGS) $@ $%
		$(RM) $%

表 4–9 標準の接尾辞の規則 続く

用途	接尾辞規則の名前	コマンド行
シェルスクリプト	.sh	$(RM) $@
		cat $< >$@
		chmod +x $@
yacc ファイル (.yc の規則)	.y	$(YACC.y) $<
		$(LINK.c) -o $@ y.tab.c $(LDLIBS)
		$(RM) y.tab.c
	.y.c	$(YACC.y) $<
		mv y.tab.c $@
	.y.ln	$(YACC.y) $<
		$(LINT.c) -o $@ -i y.tab.c
		$(RM) y.tab.c
	.y.o	$(YACC.y) $<
		$(COMPILE.c) -o $@ y.tab.c
		$(RM) y.tab.c
yacc ファイル (SVR4)	.Y.C	$(YACC) $(YFLAGS) $<
		mv y.tab.c $@
	.Y.o	$(YACC) $(YFLAGS) $<
		$(COMPILE.c) y.tab.c
		rm -f y.tab.c
		mv y.tab.o $@

表 4–10 定義済みの動的マクロ

用途	マクロ	デフォルト
ライブラリアーカイブ	AR	ar
	ARFLAGS	rv
アセンブラのコマンド	AS	as
	ASFLAGS	
	COMPILE.s	$(AS) $(ASFLAGS)
	COMPILE.S	$(CC) $(ASFLAGS) $(CPPFLAGS) -target -c

表 4–10 定義済みの動的マクロ 続く

用途	マクロ	デフォルト
C コンパイラのコマンド	CC	cc
	CFLAGS	
	CPPFLAGS	
	COMPILE.c	$(CC) $(CFLAGS) $(CPPFLAGS) -c
	LINK.c	$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS)
C++ コンパイラのコマンド[1]	CCC	CC
	CCFLAGS	
	COMPILE.cc	$(CCC) $(CCFLAGS) $(CPPFLAGS) -c
	LINK.cc	$(CCC) $(CCFLAGS) $(CPPFLAGS) $(LDFLAGS)
C++ SVR4 コンパイラのコマンド (SVR4)	(C++C)	CC
	(C++FLAGS)	-O
	COMPILE.C	$(C++C) $(C++FLAGS) $(CPPFLAGS) -c
	LINK.C	$(C++C) $(C++FLAGS) $(CPPFLAGS) $(LDFLAGS) -target
FORTRAN 77 コンパイラのコマンド	FC in SVr4	f77
	FFLAGS	
	COMPILE.f	$(FC) $(FFLAGS) -c
	LINK.f	$(FC) $(FFLAGS) $(LDFLAGS)
	COMPILE.F	$(FC) $(FFLAGS) $(CPPFLAGS) -c
	LINK.F	$(FC) $(FFLAGS) $(CPPFLAGS) $(LDFLAGS)
リンカーのコマンド	LD	ld
	LDFLAGS	
lex のコマンド	LEX	lex
	LFLAGS	
	LEX.l	$(LEX) $(LFLAGS) -t
lint のコマンド	LINT	lint
	LINTFLAGS	
	LINT.c	$(LINT) $(LINTFLAGS) $(CPPFLAGS)

表 4–10 定義済みの動的マクロ 続く

用途	マクロ	デフォルト
Modula 2 のコマンド	M2C	m2c
	M2FLAGS	
	MODFLAGS	
	DEFFLAGS	
	COMPILE.def	$(M2C) $(M2FLAGS) $(DEFFLAGS)
	COMPILE.mod	$(M2C) $(M2FLAGS) $(MODFLAGS)
NeWS	CPS	cps
	CPSFLAGS	
Pascal コンパイラのコマンド	PC	pc
	PFLAGS	
	COMPILE.p	$(PC) $(PFLAGS) $(CPPFLAGS) -c
	LINK.p	$(PC) $(PFLAGS) $(CPPFLAGS) $(LDFLAGS)
Ratfor のコンパイルコマンド	RFLAGS	
	COMPILE.r	$(FC) $(FFLAGS) $(RFLAGS) -c
	LINK.r	$(FC) $(FFLAGS) $(RFLAGS) $(LDFLAGS)
rm のコマンド	RM	rm -f
yacc のコマンド	YACC	yacc
	YFLAGS	
	YACC.y	$(YACC) $(YFLAGS)
接尾辞のリスト	SUFFIXES	.o .c .c~ .cc .cc~ .C .C~ .y .y~ .l .l~ .s .s~ .sh .sh~ .S .S~ .ln .h .h~ .f .f~ .F .F~ .mod .mod~ .sym .def .def~ .p .p~ .r .r~ .cps .cps~ .Y .Y~ .L .L~
SCCS get コマンド	.SCCS_GET	sccs $(SCCSFLAGS) get $(SCCSGETFLAGS) $@ -G$@
	SCCSGETFLAGS	-s

1. 下位互換性を持たせるため、C++ のマクロには上記の代わりに使用できる形式があります。C++C は CCC で、C++FLAGS は CCFLAGS で、COMPILE.C は COMPILE.cc で、LINK.cc は LINK.C で、それぞれ代用できます。これらの代用の形式は、将来のリリースでは削除されます。

オブジェクトライブラリの構築

ライブラリ、メンバー、シンボル

オブジェクトライブラリは、ar ライブラリアーカイブ に含まれている複数のオブジェクトファイルの集まりです (ライブラリアーカイブファイルについての詳細は、ar(1) および lorder(1) のマニュアルページを参照してください)。多くの言語では、C のライブラリにあるような汎用ユーティリティのコンパイル済み関数をオブジェクトライブラリとして保存します。

ar は、1 つまたは複数のファイルを読み込んでライブラリを作成します。ライブラリに含める各メンバーには、ヘッダーと 1 つのファイルのテキストが含まれます。メンバーのヘッダーには、ファイルディレクトリのエントリから取得した、変更時間などの情報が含まれます。これによって、make は依存関係の検査で、ライブラリのメンバーを独立した別々の構成要素として扱うことができます。

オブジェクトライブラリの関数を使用するプログラムを (ファイル名または cc の –l オプションにより適切なライブラリを指定して) コンパイルすると、リンカーは必要なシンボルを含むライブラリのメンバーを選択してリンクします。

ar を使用して、オブジェクトファイルのライブラリ用のシンボルテーブルを生成できます。ld がライブラリ内のシンボルへアクセスできるように、つまり関数が定義されたオブジェクトファイルを ld が検出してリンクできるようにするには、このシンボルテーブルが必要です。また、先に lorder と tsort を使用して、メンバーを呼び出し順にライブラリ内でソートすることもできます (詳細は、ar(1) および lorder(1) を参照してください)。大規模なライブラリでは、これら両方の処理を行なってください。

ライブラリのメンバーおよび依存関係の検査

make は、以下の形式で記述されたターゲットまたは依存関係を、ライブラリのメンバーへの参照または複数のメンバーを空白文字で区切ったリストへの参照として認識します。

```
lib.a (member . . . )
```

注 - 旧バージョンの make は、上記の書式を認識しますが括弧内の最初のメンバーだけが処理されます。

このバージョンの make では、括弧内に指定したすべてのメンバーが処理されます。たとえば次に示すターゲットエントリは、librpn.a というライブラリが、stacks.o および fifos.o というメンバーから構築されることを示しています。またパターンマッチングの規則は、各メンバーは対応するオブジェクトファイルに依存していて、オブジェクトファイルは対応するソースファイルから暗黙の規則を使用して構築されることを示しています。

```
librpn.a:   librpn.a (stacks.o fifos.o)
             ar rv $@ $?

             $@
librpn.a (%.o): %.o
             @true
```

ライブラリのメンバーを記述する際には、動的マクロの $? は対応するメンバーよりも新しいファイルのリストを示します。

```
$ make
cc -c stacks.c
cc -c fifos.c
ar rv librpn.a stacks.o fifos.o
a - stacks.o
a - fifos.o
```

ライブラリと動的マクロ $%

動的マクロ $% は、特にライブラリの指定に使用します。ライブラリのメンバーがターゲットの場合は、メンバーの名前が $% マクロに割り当てられます。たとえば、libx.a(demo.o) というターゲットが指定されている場合は、$% の値は demo.o になります。

.PRECIOUS：割り込みによる削除からライブラリを保護する

通常は、ターゲットの処理中に make に割り込みが発生すると、ターゲットファイルが削除されます。これは、変更時間が最新になった不完全なファイルがディレクトリに残らないようにするため、1 つずつ独立したライブラリファイルの場合に適しています。しかし、複数のメンバーで構成されるライブラリの場合は、最新でないメンバーがあってもライブラリファイルを削除せずにそのままにしておくことをお勧めします。以降に make を実行すると、前回の実行で処理が中断されたオブジェクトファイルまたはメンバーから処理が継続されるため、特に大規模なライブ

ラリの場合は、ライブラリファイルを削除しないでおくことをお勧めします。処理時間を節約することができます。

.PRECIOUS という特殊ターゲットは、割り込み時の削除から保護するファイルを指定するために使用します。make は、このターゲットの依存関係として指定されたターゲットは削除しません。

```
.PRECIOUS:  librpn.a
```

たとえば、この行を前述のメークファイルに追加して make を実行すると、librpn.a の処理に割り込んだ場合は、ライブラリは削除されません。

make を使用してライブラリおよびプログラムを管理する

これまでの節では、簡単なプログラムをコンパイルして簡単なライブラリを構築する際に make を利用する方法を説明しました。この節では、複雑なプログラムおよびライブラリを管理する make のより高度な機能について説明します。

マクロについて

マクロ定義は、メークファイルの任意の行に記述できます。マクロは、長いターゲットリストや式を短縮するために、あるいは繰り返し記述する必要のある長い文字列の代替として使用できます。

また、マクロを使用して、オブジェクトファイルのリストをソースファイルのリストから取得できます。マクロ名は、メークファイルの読み取り時に割り当てられます。マクロ参照の値は、最後に割り当てられた値によって決定されます。

注 - マクロの評価は、実際には上述よりも複雑に行われます。146ページの「暗黙の規則と動的なマクロ」を参照してください。

条件付きマクロおよび動的マクロを除き、make はマクロの値をマクロの定義順に割り当てます。

組み込みマクロ参照

マクロ参照は、他のマクロの参照箇所に組み込むことができます (旧バージョンの make では使用できません)。

```
$(CPPFLAGS$(TARGET_ARCH))
```

注 - 代入文の += は、指定された文字列をマクロの既存の値の後に追加します。

この場合は、マクロは内側から外側に展開されます。以下の定義例では、make は Sun-4 システム用の正しいシンボルを定義します。

```
CPPFLAGS-sun4 = -DSUN4
CPPFLAGS += $(CPPFLAGS-$(TARGET_ARCH))
```

マクロ参照での接尾辞の置換

make には、参照されるマクロの値に含まれる特定の接尾辞を置換する機能があります。慣例として、接尾辞はドット (.) で始まりますが、任意の文字列を接尾辞として指定することもできます。

```
$(macro:old-suffix=new-suffix)
```

上記の形式は、接尾辞を置換するマクロ参照の書式です。このような参照を使用することによって、以下のようにオブジェクトファイルのリストをソースファイルのリストで表わすことができます。

```
OBJECTS= $(SOURCES:.c=.o)
```

この例では、make は値に含まれる接尾辞 .c をすべて .o に置換します。指定された接尾辞が付いていない語に対しては置換は適用されません。

```
SOURCES= main.c data.c moon
OBJECTS= $(SOURCES:.c=.o)

all:
        @echo $(OBJECTS)
```

例えば、上記のメークファイルで make を実行すると次のような結果になります。

```
$ make
main.o data.o moon
```

make と lint を使用する

C プログラムをより容易にデバッグおよび管理するには、lint ツールを使用します。lint は、他のアーキテクチャへの移植性がない C の構造体の検査も行います。

lint は、移植性のある C プログラムを記述する際に非常に便利です。C プログラムの検査を行う lint は、発見および追跡するのが難しいバグを防ぐためのツールです。初期化されていないポインタ、関数呼び出しにおける引数の数の不一致、移植不可能な C の構造体の使用、などのバグがないかどうかを検査します。clean と同様に、lint はターゲットの慣用名です。C プログラムを構築するメークファイル中に、lint を含めることをお勧めします。lint は、cpp および lint の最初の (構文解析) 処理過程で処理された出力ファイルを生成します。出力ファイルは、接尾辞が .ln となります。また、接尾辞の置換によって、ソースファイルのリストから lint の出力ファイルを生成することもできます (旧バージョンの lint では不可能な場合があります)。

```
LINTFILES= $(SOURCES:.c=.ln)
```

lint のターゲットエントリは、以下のようになります。

```
lint: $(LINTFILES)
        $(LINT.c) $(LINTFILES)
$(LINTFILES):
        $(LINT.c) $@ -i
```

各 .ln ファイルは、対応する .c ファイルから構築するという暗黙の規則があるため、.ln ファイルに対するターゲットエントリは必要ありません。ソースが変更されると、以下のように make を実行するときに .ln ファイルが更新されます。

```
make lint
```

LINT.c という定義済みマクロに lint のオプション指定として LINTFLAGS マクロへの参照が含まれていますが、通常は特に指定しなくてよいでしょう。lint では cpp を使用するため、コンパイルのプリプロセッサオプション (–I など) に対して、通常は CFLAGS ではなく CPPFLAGS を使用してください。また、LINT.c マクロには CFLAGS への参照は含まれていません。

make clean を実行すると、このターゲットによって生成された .ln ファイルが削除されるようにしたいと考えられます。clean ターゲットには、そのようなマクロ参照を簡単に追加することができます。

システムが提供するライブラリとのリンク

次の例は、画面上でのカーソル動作を制御する curses および termlib というライブラリパッケージを使用したプログラムをコンパイルするメークファイルです。

表 4–11　システムが提供するライブラリを使用した C プログラム用のメークファイル

```
# curses および termlib を使用した
# C プログラムを生成するメークファイル

CFLAGS= -O

.KEEP_STATE:

functions: main.o data.o
      $(LINK.c) -o $@ main.o data.o -lcurses
      -ltermlib
lint: main.ln data.ln
      $(LINT.c) main.ln data.ln
main.ln data.ln:
      $(LINT.c) $@ -i
clean:
      rm -f functions main.o data.o main.ln \
      data.ln
```

リンカーは、未定義のシンボルを検出するとそれを解決するため、通常は、リンクするファイルのリストの最後に、ライブラリの参照を記述してください。

このメークファイルの実行結果は、以下のとおりです。

```
$ make
cc -O -c main.c
cc -O -c data.c
cc -O -o functions main.o data.o -lcurses -ltermlib
```

デバッグ用およびプロファイル用にプログラムをコンパイルする

デバッグ用およびプロファイル用のプログラムは同じソースコードを使用して生成しますが、構築の際に異なる C コンパイラオプションを使用します。デバッグ用のオブジェクトコードを生成するには、cc の –g オプションを使用します。プロファイル用には、–O および –pg という cc オプションを使用します。

コンパイル手順は同じなので、make のコマンド行で CFLAGS の定義を指定できます。このコマンド行での定義は、メークファイルでの定義を無効にします。また、.KEEP_STATE により、変更の影響を受けるすべてのコマンド行が実行されます。以下に例を示します。

```
$ make "CFLAGS= -O -pg"
cc -O -pg -c main.c
cc -O -pg -c data.c
cc -O -pg -o functions main.o data.o -lcurses -ltermlib
```

これらのオプションを暗記したり、このような複雑なコマンドを入力するのは煩雑です。その代わりに、これらの情報はメークファイルに記述できます。デバッグ用またはプロファイル用のコードの生成方法をメークファイルに記述し、make に伝える必要があります。1 つの方法として、debug および profile という名前の 2 つのターゲットエントリと、コマンド行へのそれぞれのコンパイルオプションを、メークファイルに追加します。

よりよい方法として、開始ターゲットに応じて CFLAGS の定義を変更する規則を指定する、debug および profile というターゲットエントリを追加します。次に、各ターゲットが既存のターゲットに依存するようにすることによって、make は既存のターゲットの規則と、指定されたオプションを使用することができます。

以下に例を示します。

```
make "CFLAGS= -g"
```

前述のようにメークファイルを記述すると、上記のように make を実行する代わりに、以下のようにしてデバッグ用のコードをコンパイルすることができるようになります。

```
make debug
```

ここで、ターゲット (およびその依存関係) ごとに異なる方法でマクロを定義することを、どのように make に指示するかが問題になります。

条件付きマクロ定義

条件付きマクロ定義は、以下のような形式で記述します。

```
target-list := macro = value
```

make が *target-list* に指定された名前のターゲットおよびその依存関係を処理する際に、指定されたマクロに指定された値を割り当てます。

注 - *target-list* 中のそれぞれの語は、% のパターンを 1 つずつ含めることができます。定義が適用されるターゲットを make が特定する必要があるため、条件付きマクロ定義を使用してターゲット名を変更することはできません。

以下の行は、デバッグ用およびプロファイル用の各プログラムコードを処理するための適切な値を CFLAGS に割り当てます。

```
debug := CFLAGS= -g
profile := CFLAGS= -pg -O
```

条件付きマクロへの参照を依存関係リスト中で使用する際には、遅延参照を使用する (先頭に $ を 1 つ追加する) 必要があります 。遅延参照を使用しないと、正しい値が割り当てられる前に make が参照を展開してしまいます。make は、このような正しくない (可能性のある) 参照を検出すると警告を表示します。

デバッグ用およびプロファイル用のプログラムをコンパイルする

以下のメークファイルは、指定したターゲットに応じて、最適化された C プログラム、デバッグ用の C プログラム、プロファイル用の C プログラムのいずれかの形式の C プログラム (デフォルトでは最適化されたプログラム) をコンパイルします。コマンドの依存関係が検査されるので、上記 3 つのうちで形式を切り換えると、プログラムおよびそのオブジェクトファイルが再コンパイルされます。

表 4–12 デバッグ形式またはプロファイル形式の C プログラムを生成するメークファイル

```
# デバッグ形式またはプロファイル形式の
# C プログラムを生成するメークファイル

CFLAGS= -O

.KEEP_STATE:

all debug profile: functions

debug := CFLAGS = -g
profile := CFLAGS = -pg -O

functions: main.o data.o
      $(LINK.c) -o $@ main.o data.o -lcurses
      -ltermlib
lint: main.ln data.ln
      $(LINT.c) main.ln data.ln
clean:
      rm -f functions main.o data.o main.ln data.ln
```

最初のターゲットエントリは、all によって 3 つのターゲットを指定しています。

注 - 通常最終的な完成プログラムには、デバッグ形式およびプロファイル形式のコードが含まれないようにします。

慣例として、メークファイルの最初のターゲットとして、代替の開始ターゲット (またはターゲットのリストを処理するターゲット) とともに all を記述します。all

の依存関係は、種類を問わず、最終的に構築されるすべてのターゲットになります。この場合は、最終的には最適化されたプログラムが構築されます。ターゲットエントリは、debug および profile が functions ($(PROGRAM) の値) に依存することも示しています。

次の 2 行は、CFLAGS の条件付きマクロ定義です。

次に、functions のターゲットエントリが記述されています。debug が functions に依存している場合は、–g オプションを使用してコンパイルされます。

次の例では、同様の手法を C のオブジェクトライブラリの管理に使用しています。

表 4–13 デバッグ形式またはプロファイル形式の C ライブラリを生成するメークファイル

```
# デバッグ形式またはプログラム形式の
# C ライブラリファイルを生成するメークファイル

CFLAGS= -O

.KEEP_STATE
.PRECIOUS: libpkg.a

all debug profile: libpkg.a
debug := CFLAGS= -g
profile := CFLAGS= -pg -O

libpkg.a: libpkg.a(calc.o map.o draw.o)
	ar rv $@ $?
	libpkg.a(%.o): %.o
	@true
lint: calc.ln map.ln draw.ln
	$(LINT.c) calc.ln map.ln draw.ln
clean:
	rm -f libpkg.a calc.o map.o draw.o \
	calc.ln map.ln draw.ln
```

複数のプログラムおよびライブラリの形式を管理する

前述の 2 つの例は、開発、デバッグ、プロファイルを別々に実行する場合には適しています。しかし、形式を切り替えるごとにすべてのオブジェクトファイルが再コンパイルされるので、コンパイル時間が長くなるという欠点があります。以下の 2 つの例は、3 つの形式すべてを独立した構成要素として別々に管理する方法を説明しています。

3 つの形式のオブジェクトファイルが同一のディレクトリにあることによる混乱を避けるため、デバッグ用およびプロファイル用のオブジェクトファイルと実行可能

ファイルをサブディレクトリに保存できます。ただし、オブジェクトファイルのリストの各エントリに、サブディレクトリ名を接頭辞として追加する必要があります。

パターン置換マクロ参照

パターン置換マクロ参照は、形式および機能が接尾辞置換の参照と同様です。パターン置換参照を使用して、接頭辞、接尾辞のいずれかまたは両方を、マクロの値中で一致する語に追加または置換できます。

注 - パターン置換マクロ参照は、パターンマッチングの規則と同様に、旧バージョンの make で使用することはできません。

パターン置換参照は、以下の形式で記述します。

```
$(macro:p%s =np%ns)
```

ここで、*p* は置換対象の既存の接頭辞 (ある場合)、*s* は置換対象の既存の接尾辞 (ある場合)、*np* および *ns* はそれぞれ新しい接頭辞および接尾辞、% はワイルドカードです。パターン置換は、値 *p%s* に一致するすべての語に適用されます。

```
SOURCES= old_main.c old_data.c moon
OBJECTS= $(SOURCES:old_%.c=new_%.o)
all:
        @echo $(OBJECTS)
```

この例は、以下のような結果になります。

```
$ make
new_main.o new_data.o moon
```

= 記号の右側 (置換後の語) では、ワイルドカードの % を任意の数だけ必要に応じて使用できます。以下に例を示します。

```
...
OBJECTS= $(SOURCES:old_%.c=%/%.o)
```

この置換は、以下のような結果になります。

```
main/main.o data/data.o moon
```

ただし、パターン置換マクロ参照は、パターンマッチングの規則を指定しているターゲットエントリの依存関係を示す行では使用しないでください。使用した場合は、マクロとターゲット (または依存関係) のどちらにワイルドカードが適用されるかを make が特定できないため、衝突が生じます。

```
OBJECT= .o
```

```
x:
x.Z:
        @echo correct
%: %$(OBJECT:%o=%Z)
```

このメークファイルは、make が x.Z から x を構築するために記述したものです。しかし、依存関係の行に複数含まれている % のうちどれをパターンマッチングの規則で使用するかを make が特定できないため、パターンマッチングの規則は認識されません。

形式が複数あるプログラム用のメークファイル

次に示すのは、複数の形式が別々に管理されている C プログラム用のメークファイルの例です。まず .INIT という特殊ターゲットが、debug_dir および profile_dir というサブディレクトリを (まだ作成されていなければ) 作成します。これらのサブディレクトリには、デバッグ用およびプロファイル用のオブジェクトファイルと実行可能ファイルが含まれます。

注 - make は、メークファイルが読み込まれた後に .INIT ターゲットの規則を実行します。

実行可能ファイルは、VARIANTS.o マクロで指定されているオブジェクトファイルに依存します。このマクロにはデフォルトでは OBJECTS の値が指定され、後に条件付きマクロ定義によって値が再度割り当てられます。このとき、debug_dir/ または profile_dir/ の接頭辞が追加されます。サブディレクトリ内の実行可能ファイルは、同じサブディレクトリ内に構築されるオブジェクトファイルに依存します。

次に、両方のサブディレクトリに含まれているオブジェクトファイルが作業中のディレクトリにあるソース (.c) ファイルに依存するように、パターンマッチングの規則が追加されます。これは、1 つのソースファイル群から 3 つの形式すべてを構築し管理するために必要です。

最後に、サブディレクトリの debug_dir と profile_dir は一時的に作成されたものであるため、clean ターゲットが更新されてこれらのサブディレクトリとその内容が再帰的に削除されます。これは、各形式用のサブディレクトリは一時的なものであるため、派生ファイルはそのソースと同一のディレクトリ内に構築するという慣例に従っています。

表 4–14　デバッグ用およびプロファイル用のプログラムを別々に扱うためのメークファイル

```
# デバッグ用およびプロファイル用のプログラムを
# 別々に扱うためのメークファイル

CFLAGS= -O

SOURCES= main.c rest.c
OBJECTS= $(SOURCES:%.c=$(VARIANT)/%.o)
VARIANT= .

functions profile debug: $$(OBJECTS)
 $(LINK.c) -o $(VARIANT)/$@ $(OBJECTS)

debug := VARIANT = debug_dir
debug := CFLAGS = -g
profile := VARIANT = profile_dir
profile := CFLAGS = -O -pg

.KEEP_STATE:
.INIT: profile_dir debug_dir
profile_dir debug_dir:
 test -d $@ || mkdir $@
 $$(VARIANT)/%.o: %.c
 $(COMPILE.c) $< -o $@
clean:
 rm -r profile_dir debug_dir $(OBJECTS) functions
```

複数の形式があるライブラリ用のメークファイル

複数の形式があるライブラリ用のメークファイルも、同様の方法で変更します。

```
# 複数の形式のライブラリを別々に扱うためのメークファイル

CFLAGS= -O

SOURCES= main.c rest.c
LIBRARY= lib.a
LSOURCES= fnc.c

OBJECTS= $(SOURCES:%.c=$(VARIANT)/%.o)
VLIBRARY= $(LIBRARY:%.a=$(VARIANT)/%.a)
LOBJECTS= $(LSOURCES:%.c=$(VARIANT)/%.o)
VARIANT= .

program profile debug: $$(OBJECTS) $$(VLIBRARY)
          $(LINK.c) -o $(VARIANT)/$@ $<

lib.a debug_dir/lib.a profile_dir/lib.a: $$(LOBJECTS)
          ar rv $@ $?
```

```
$$(VLIBRARY)($$(VARIANT)%.o): $$(VARIANT)%.o
        @true
profile := VARIANT = profile_dir
profile := CFLAGS = -O -pg

debug := VARIANT = debug_dir
debug := CFLAGS = -g

.KEEP_STATE:
profile_dir debug_dir:
        test -d $@ || mkdir $@
$$(VARIANT)/%.o: %.c
        $(COMPILE.c) $< -o $@
```

複数の形式を管理するこの手法は、便利ですがやや複雑です。説明を簡潔にするため、この手法は以下の例では省略しています。

ヘッダーファイルのディレクトリを管理する

ヘッダーのインクルードディレクトリを管理するためのメークファイルは、非常に簡単です。ヘッダーファイルはテキストファイルなので、記述する必要があるのは `all` というターゲットだけです。この `all` ターゲットで、ヘッダーを依存関係として指定します。その他の処理は、SCCS の自動取り出しによって行われます。ヘッダーのリストの代わりにマクロを使用すると、他のターゲットエントリで同一のリストを使用できます。

```
# インクルードディレクトリを管理するためのメークファイル

FILES.h= calc.h map.h draw.h

all: $(FILES.h)

clean:
        rm -f $(FILES.h)
```

ユーザー定義のライブラリをコンパイルおよびリンクする

ユーザー定義のライブラリパッケージを作成する際には、各ライブラリを、ライブラリのヘッダーおよびライブラリを使用するプログラムとは別の構成要素として扱ってください。プログラム、ライブラリ、ヘッダーをそれぞれ別のディレクトリに置くと、各モジュール用のメークファイルの作成が簡単になります。また、ソフトウェアプロジェクトの構造が明確になります。

注 - make を実行した後に、ファイルシステム上のいろいろな場所にファイルが生成されないようにしてください。

メークファイルは、作業中のディレクトリまたは一時的なサブディレクトリ内のファイルだけを構築するように記述する必要があります。make を使用して、何らかの理由で特定のファイルシステム上のディレクトリに意図的にファイルをインストールする場合を除き、他のディレクトリにファイルを作成するメークファイルは作成しない方がよいでしょう。

他のディレクトリにあるライブラリに依存するプログラムを構築する場合には、メークファイル中で修正が必要な点があります。これまでの例では、すべての必要なファイルは、同一のディレクトリ中か、または基本的には変更されない標準ディレクトリ中にあります。ただし、開発中のプロジェクトの一部であるユーザー定義のライブラリについては、場所が変更されることも考えられます。

これらのライブラリは自動的に構築されない (隠れた依存関係の検査に相当するものがライブラリにはない) ため、ライブラリのターゲットエントリを指定する必要があります。また、リンクするライブラリが最新のものであることを確認する必要があります。

また、ローカルディレクトリ内でのみファイルを管理するようにメークファイルを記述する必要があります。さらに、メークファイル中には、別のメークファイルにある内容と重複する情報を記述しないようにする必要があります。

入れ子にした make コマンド

以上の問題を解決するには、ライブラリがあるディレクトリで make コマンドを入れ子にして実行し、(そのディレクトリにあるメークファイル中のターゲットエントリに従って) ライブラリを構築します。

注 - デフォルトでは "make" という値に設定されている MAKE マクロは、make コマンドの –n オプションを無効にします。つまり MAKE マクロを参照しているコマンドは、–n オプションが指定されている場合でも実行されます。ただし MAKE マクロは make コマンドを呼び出すためにのみ使用され、このマクロによって呼び出された make は、特殊マクロ MAKEFLAGS から –n オプションを継承します。入れ子 (階層構造) になった make はそれぞれ MAKEFLAGS マクロによって –n オプションが指定されていることを認識していきます。このため、–n オプションを使用することで、入れ子になった make の動作を実際に実行せずに確認することができます。

```
# 他のディレクトリに生成されるターゲット用の最初のエントリ

../lib/libpkg.a:
        cd ../lib ; $(MAKE) libpkg.a
```

ライブラリは、現在のディレクトリからの相対パス名で指定します。プロジェクトが新しいルートディレクトリまたはマシンに移動された場合に、新しいルートディレクトリに対するディレクトリ構造がそのまま同じであれば、すべてのターゲットエントリが正しいファイルを示します。

入れ子にした make のコマンド行では、定義済みマクロ MAKE の場合と同様に、動的マクロの修飾子の F および D が便利です。処理されるターゲットがパス名で指定されている場合は、$(@F) はファイル名部分、$(@D) はディレクトリ部分をそれぞれ示します。ターゲット名に / という文字が含まれていない場合は、$(@D) の値としてドット (.) が値として割り当てられます。

ターゲットエントリは、次のように書き換えることができます。

```
# 2 番目のエントリ

../lib/libpkg.a:
        cd $(@D); $(MAKE) $(@F)
```

入れ子にした make コマンドを強制的に実行する

このターゲットには依存関係がないため、../lib/libpkg.a という名前のファイルがないときにだけこのターゲットが実行されます。ファイルが .PRECIOUS により保護されたライブラリアーカイブである場合は、../lib/libpkg.a ファイルがないということはほとんどありません。make はそのファイルの依存関係を認識する必要はないため、認識しません。ファイルを構築するかどうか、およびその構築方法は、入れ子にした呼び出しによって決定されます。

つまり、ファイルシステム内にファイルがあっても、そのファイルが最新でない場合もあります。したがって、ファイルがあるかどうかに関わらず、そのファイルを規則が空白の (および既存のファイルがない) 他のターゲットに依存させることによって、入れ子にした make コマンドを強制的に実行する必要があります。

表 4–15 入れ子にした make コマンド用のターゲットエントリ

```
# 入れ子にした make コマンド用
# ターゲットエントリ

../lib/libpkg.a: FORCE
      cd $(@D); $(MAKE) $(@F)
FORCE:
```

この方法により、make は、正しいディレクトリ ../lib に変更し、そのディレクトリにあるメークファイルに記述された命令に従って、必要であれば libpkg.a を構築します。入れ子にした make の実行結果は次のようになります。

```
$ make ../lib/libpkg.a
cd ../lib; make libpkg.a
make libpkg.a
'libpkg.a' is up to date.
```

以下のメークファイルは、入れ子にした make コマンドを使用して、プログラムが依存するユーザー定義のライブラリを処理します。

表 4–16 ユーザー定義のライブラリを使用した C プログラム用のメークファイル

```
# ユーザー定義の C ライブラリと
# 入れ子になった make コマンド用の Makefile

CFLAGS= -O

.KEEP_STATE:

functions: main.o data.o ../lib/libpkg.a
      $(LINK.c) -o $@ main.o data.o
      ../lib/libpkg.a -lcurses -ltermlib
      ../lib/libpkg.a: FORCE
      cd $(@D); $(MAKE) $(@F)
FORCE:

lint: main.ln data.ln
      $(LINT.c) main.ln data.ln
clean:
      rm -f functions main.o data.o main.ln data.ln
```

../lib/libpkg.a が最新である場合は、このメークファイルを使用する make の実行結果は以下のようになります。

```
$ make
cc -O -c main.c
cc -O -c data.c
cd ../lib; make libpkg.a
'libpkg.a' is up to date.
```

```
cc -O -o functions main.o data.o ../lib/libpkg.a -lcurses -l   termlib
```

MAKEFLAGS マクロ

MAKE マクロと同様に、MAKEFLAGS も特殊マクロです。

注 - MAKEFLAGS をメークファイル中に定義しないでください。

MAKEFLAGS には、make コマンド用のフラグ (1 文字のオプション) が含まれています。他の FLAGS マクロとは異なり、MAKEFLAGS の値は、フラグの冒頭に付いている - (ハイフン) を除いて連結したものになります。たとえば、eiknp という文字列は、MAKEFLAGS の値として認識されますが、-f x.mk や macro=value は値として認識されません。

MAKEFLAGS という環境変数が設定されている場合は、make は、コマンド行で指定されたフラグと、MAKEFLAGS に含まれるフラグを組み合わせて実行されます。

MAKEFLAGS の値は、環境変数で設定されているかどうかに関係なく常にエクスポートされ、MAKEFLAGS に含まれるオプションは、
($(MAKE)、make、/usr/bin/make のうちどれによって呼び出されたかに関係なく) 入れ子にした make コマンドに渡されます。これにより、親の make が呼び出された際のオプションが、入れ子にした make コマンドに渡されます。

入れ子にした make コマンドにパラメータを渡す

MAKEFLAGS を除いて、make は環境変数をインポートし、定義済みのマクロと同様に扱います。次に、make は呼び出したコマンド (入れ子になった make コマンドを含む) にそれらの環境変数およびその値を渡します。

注 - 環境変数 SHELL は、このバージョンの make にはインポートおよびエクスポートされません。

マクロは、メークファイルと同様に、コマンド行の引数としても指定できます。このため、マクロが複数の箇所で定義されている際に名前が衝突することがあります。make には、このような衝突を回避するための非常に複雑な優先順位の規則があります。

まず、条件付きマクロの定義は、定義されているターゲット (およびその依存関係) 内で常に有効です。

マクロ定義を引数として make を呼び出すと、その定義はメークファイル内のマクロ定義または環境変数からインポートされたマクロ定義よりも優先されます (ただし、入れ子にした make コマンドではこの優先順位とは異なる場合があります)。それ以外の場合は、メークファイルでマクロを定義 (または再定義) した場合は、最新の定義が適用されます。通常は、最新の定義が環境変数の定義よりも優先されます。

最後に、マクロがデフォルトのファイルだけで定義されている場合は、その値が使用されます。

入れ子にした make コマンドでは、通常はメークファイルでの定義が環境変数よりも優先されますが、これは定義がメークファイルに記述されている場合のみ該当します。対応する環境変数はこれとは無関係に伝達されます。

コマンド行での定義は、その定義を指定した make の実行でのみ、環境変数とメークファイルの定義を無効にします。コマンド行で指定した値は、入れ子にした make コマンドにも渡されますが、入れ子にした make コマンドでの定義と、入れ子にした make コマンドによりインポートされた環境変数によって無効になります。

–e オプションの動作はさらに一貫しています。環境変数は、メークファイルでのマクロ定義を無効にします。コマンド行での定義は、メークファイルおよび環境変数での定義よりも常に優先して使用されます。ただし、–e を使用すると、メークファイルに含まれていない情報により構築の成否が左右される可能性があります。

このような複雑さを避けるために、特定の値を make コマンドの階層全体に渡す際には、環境変数を (C シェルで) 設定して、サブシェルで make –e を実行してください。

```
% (unsetenv MAKEFLAGS LDFLAGS; setenv CFLAGS -g; make -e)
```

以下のメークファイルを使用して、さまざまな場合をテストすることができます。

```
# top.mk

MACRO= "Correct but unexpected."

top:
      @echo "------------------------------ top"
      echo $(MACRO)
      @echo "------------------------------"
      $(MAKE) -f nested.mk
      @echo "------------------------------ clean"
clean:
      rm nested


# nested.mk

MACRO=nested

nested:
```

```
@echo "------------------------------ nested"
touch nested
echo $(MACRO)
$(MAKE) -f top.mk
$(MAKE) -f top.mk clean
```

以下に、マクロの割り当て順序の一覧表を示します。

表 4–17 マクロの割り当て順序の一覧表

–e なし	–e を使用
最上位の make コマンド	
条件付き定義	条件付き定義
make のコマンド行	make のコマンド行
最新のメークファイルの定義	環境変数の値
環境変数の値	最新のメークファイルの定義
定義済みの値 (ある場合)	定義済みの値 (ある場合)
入れ子にした make コマンド	
条件付き定義	条件付き定義
make のコマンド行	make のコマンド行
最新のメークファイルの定義	親の make コマンド行
環境変数	環境変数
定義済みの値 (ある場合)	最新のメークファイルの定義
親の make コマンド行	定義済みの値 (ある場合)

その他のソースファイルをコンパイルする

アセンブリ言語のルーチンを使用した C プログラムのコンパイルおよびリンク

次の例のメークファイルは、アセンブリ言語のルーチンとリンクした C プログラムを管理します。アセンブリのソースファイルには、cpp プリプロセッサ指令を含むものと含まないものの 2 種類あります。

慣例として、プリプロセッサ指令を含まないアセンブリのソースファイルには .s という接尾辞が付きます。プリプロセッサ処理が必要なアセンブリのソースには、.S という接尾辞が付きます。

注 - ASFLAGS は、.s.o および .S.o の暗黙の規則に関するオプションを渡します。

アセンブリのソースは、C ソースをコンパイルするのと同様の方法でアセンブルされてオブジェクトファイルを形成します。そのオブジェクトファイルは、C プログラムにリンクできます。make には、.s および .S のファイルをオブジェクトファイルに変換するための暗黙の規則があるため、アセンブリのルーチンを持つ C プログラムのターゲットエントリでは、オブジェクトファイルのリンク方法を指定するだけです。アセンブラによって生成されたオブジェクトファイルをリンクするには、cc コマンドを使用できます。

表 4–18 アセンブルソースファイルから C プログラムを生成するメークファイル

```
CFLAGS= -O
ASFLAGS= -O

.KEEP_STATE:

driver: c_driver.o s_routines.o S_routines.o
      cc -o driver c_driver.o s_routines.o
      S_routines.o
```

.S ファイルは、cc コマンドを使用して処理されます。cc コマンドは、C のプリプロセッサ cpp およびアセンブラを呼び出します。

lex および yacc のソースをコンパイルする

lex および yacc は、C ソースファイルを出力します。lex および yacc のソースファイルは、接尾辞がそれぞれ .l、.y になります。lex および yacc のソースファイルを別々にコンパイルする場合のコンパイル処理は、C ソースだけからプログラムを生成する場合と同様です。

lex または yacc のソースを .c ファイルにコンパイルする暗黙の規則があります。.c ファイルは、C ソースからオブジェクトファイルをコンパイルする暗黙の規則を使用してさらに処理されます。ソースファイルに #include 文が含まれていない場合は、.c ファイルは中間ファイルとして使用されるため、保存しておく必要はありません。この場合は、.l.o の規則または .y.o の規則を使用してオブジェクトファイルを生成し、(派生した) .c ファイルを削除できます。

以下にメークファイルの例を示します。

```
CFLAGS= -O
.KEEP_STATE:

all: scanner parser
scanner: scanner.o
parser: parser.o
```

このメークファイルの結果は、以下のようになります。

```
$ make -n
rm -f scanner.c
lex -t scanner.l > scanner.c
cc -O -c -o scanner.o scanner.c
rm -f scanner.c
yacc parser.y
cc -O -c -o parser.o y.tab.c
rm -f y.tab.c
```

lex と yacc を組み合わせて使用する場合は、より複雑になります。オブジェクトファイルが正しく機能するには、lex の C コードに yacc が生成したヘッダーが含まれている必要があります。yacc のソースファイルが変更されたときは、lex が生成した C ソースファイルを再コンパイルする必要があります。この場合は、yacc のソースが変更されるごとに lex を実行しなくてもよいように、lex が生成した中間 (.c) ファイルと、yacc が生成した .h ファイルを保存しておきます。

注 - yacc は、y.tab.c と y.tab.h という名前の出力ファイルを生成します。出力ファイルのベース名をソースファイルと同じにしたい場合には、出力ファイルの名前を変更してください。

以下のメークファイルは、lex のソース、yacc のソース、C ソースファイルから構築したプログラムを管理します。

```
CFLAGS= -O
.KEEP_STATE:

a2z: c_functions.o scanner.o parser.o
         cc -o $@ c_functions.o scanner.o parser.o
scanner.c:

parser.c + parser.h: parser.y
         yacc -d parser.y
         mv y.tab.c parser.c
         mv y.tab.h parser.h
```

前述のように、複数の暗黙の規則の橋渡しをする規則は生成されないので、scanner.c のターゲットエントリを指定する必要があります。このエントリは、.l.c と .c.o の暗黙の規則の橋渡しをして scanner.o の依存関係リストから scanner.l を生成します。ターゲットエントリには規則がないため、暗黙の規則 .l.c を使用して scanner.c が構築されます。

次のターゲットエントリは、yacc の中間ファイルを生成する方法を指定しています。yacc –d を使用してヘッダーと C ソースファイルの両方を生成する暗黙の規則はないため、これを行う規則を指定するターゲットを記述する必要があります。

+ 記号を使用してターゲットグループを指定する

parser.c と parser.h のターゲットエントリにおいて、ターゲット名を区切る + 記号は、そのエントリがターゲットグループのエントリであることを示します。ターゲットグループとは複数のファイルの集まりで、それらすべてのファイルは規則の実行時に生成されます。 1 つのターゲットを構成している複数のファイルを、1 つのグループとして扱います。+ 記号がない場合は、リスト中の各項目は独立したターゲットになります。ターゲットグループを使用すると、make は各ターゲットファイルについて別々に変更日時を調べ、ターゲットの規則は、make の実行 1 回につき必要な場合に 1 度だけ実行されます。

make および SCCS を使用してシェルスクリプトを管理する

シェルスクリプトはテキストファイルですが、実行するには実行権が必要です。SCCS 管理のファイルからは実行権が削除されるため、SCCS ではシェルスクリプトとその「ソース」という区別を設けると便利です。make には、ソースからスクリプ

トを取り出すという暗黙の規則があります。シェルスクリプトのソースファイルの接尾辞は .sh です。スクリプトおよび .sh ソースファイルファイルの内容は同じですが、スクリプトには実行権が設定されており、.sh ソースファイルには設定されていません。make でのスクリプト用の暗黙の規則は、ソースファイルからスクリプトを取り出し、.sh ファイルを (必要に応じて取り出してから) 複製し、そのスクリプトファイルのアクセス権を変更して実行可能にします。以下に例を示します。

```
$ file script.sh
script.sh:    ascii text
$ make script
cat script.sh > script
chmod +x script
$ file script
script:    commands text
```

make を使用してテストを実行する

シェルスクリプトは、テストの実行や、対話式の (ユーザー入力が必要な) 処理あるいは make による依存関係の検査が不要な定型作業を行う際に便利です。特にテストでは、プログラムに対して端末から特定の入力を繰り返し行う必要があります。

ライブラリの場合は、さまざまな機能を実行するプログラムのセットを C で記述し、スクリプトからの特定の入力に応じて特定の順序で実行することができます。ユーティリティプログラムの場合は、機能を実行してその速度を測定するベンチマークプログラムを作成できます。いずれの場合も、各テストを実行するコマンドをシェルスクリプトに組み込み、繰り返しユーザーが行う処理をなくしたり、管理を簡単にすることができます。

テスト用スクリプトを開発した後、そのスクリプトを実行するためのターゲットは簡単に記述できます。スクリプト内では make の依存関係の検査は不要な場合がありますが、依存関係の検査を実行すると、テスト前にプログラムまたはライブラリを更新することができます。

以下に示すテストを実行するためのターゲットエントリでは、test が lib.a に依存しています。ライブラリが最新でない場合は、make はライブラリを再構築してテストを実行します。これにより、常に最新のバージョンを使用してテストが実行されます。

```
# テスト中のライブラリ
LIBRARY= lib.a

test: $(LIBRARY) testscript
         set -x ; testscript > /tmp/test.\$\$

testscript: testscript.sh test_1 test_2 test_3

# ライブラリ構築規則
$(LIBRARY):
```

```
        @ echo Building $(LIBRARY)
       (library-building rules here)

# test_1 ... test_3 で複数のライブラリ関数を検査する
test_1 test_2 test_3: $$@.c $(LIBRARY)
        $(LINK.c) -o $@ $<
```

test は、testscript にも依存しています。testscript は、3 つのテストプログラムに依存しています。

これにより、テストプログラムが make がテスト処理を実行する前に更新されます。lib.a は、メークファイルに含まれるターゲットエントリに従って構築されます。testscript は、.sh の暗黙の規則を使用して構築されます。各テストプログラム用にそれぞれソースファイルが 1 つずつある場合に、最後のターゲットエントリの規則を使用してテストプログラムが構築されます (これらのプログラムは、.c ファイルのほかに、適切なライブラリにもリンクする必要があるため、.c の暗黙の規則は適用されません)。

シェル変数へのエスケープ参照

テスト用の規則に含まれる \$\$ という文字列は、make が $ 記号を解釈しないように指定しています。make は、2 つの $ 記号をそのままシェルに渡します。シェルは、$$ をシェルのプロセス ID に展開します。これよって、各テストごとに異なる名前の一時ファイルに書き込むことが可能になります。set –x コマンドは、シェルが端末上で実行するコマンドをシェルに表示させます。これにより、テスト結果が記録されているファイルの実際の名前を確認することができます。

シェルコマンドの置換

以下の例のように、規則内でシェルコマンドの置換を指定できます。

```
do:
        @echo `cat Listfile`
```

マクロ中では、逆引用符で囲まれた式を指定することもできます。

```
DO= `cat Listfile`
do:
        @echo $(DO)
```

ただし、この形式のコマンド置換は規則内でのみ使用できます。

コマンド置換マクロ参照

シェルコマンドをマクロの定義として指定する例を以下に示します。

```
COMMAND= cat Listfile
```

コマンド置換マクロ参照を使用して、参照をマクロ値に含まれるコマンド出力に置換するように make に指示できます。このコマンド置換は、メークファイル中の任意の箇所に記述できます。

```
COMMAND= cat Listfile
$(COMMAND:sh): $$(@:=.c)
```

この例は、他のファイルからターゲットのリストを取り込み、各ターゲットが対応する .c ファイルに依存することを示します。

シェルコマンド置換と同様に、コマンド置換参照が評価されると、コマンドの標準出力結果に置換されます。復帰改行文字は、空白文字に変換されます。コマンドは、参照が検出されると実行されます。コマンドの標準エラーは無視されます。ただし、コマンドがゼロ以外の終了ステータスを返した場合は、make はエラーを表示して停止します。

これを回避するには、コマンド行の末尾に true コマンドを追加します。

```
COMMAND = cat Listfile ; true
```

コマンド置換マクロ代入

以下の形式のマクロ代入は、*command* の標準出力を *cmd_macro* に代入します。以下に例を示します。

```
cmd_macro:sh = command


COMMAND:sh = cat Listfile
$(COMMAND): $$(@:=.c)
```

この例は、前述の例と同じ結果になります。ただし、この例ではコマンドは make の実行ごとに 1 回だけ実行されます。この場合も、標準出力だけが使用され、復帰改行文字は空白文字に変換されます。また、コマンドがゼロ以外の終了ステータスを返した場合は、make はエラーを表示して停止します。

コマンド置換マクロ代入は、以下の形式で記述することもできます。

```
macro:sh += command
```

macro の値にコマンドの出力を追加します。

```
target := macro:sh = command
```

target およびその依存関係を処理するときに、command の出力として条件付き macro を定義します。

```
target := macro:sh += command
```

target およびその依存関係を処理するときに、*command* の出力を条件付き *macro* の値に追加します。

ソフトウェアプロジェクトの管理

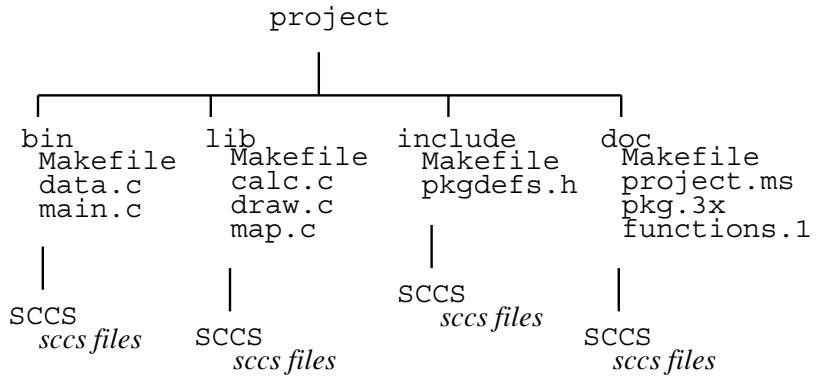
make は、ソフトウェアプロジェクトがプログラムとライブラリのシステムで構成されている場合に特に便利です。入れ子にした make コマンドを使用すると、ディレクトリ階層全体に渡って、オブジェクトファイル、実行可能ファイル、ライブラリを管理できます。make を SCCS と合わせて使用すると、ソースを一貫して管理し、そのソースから整合性のとれたプログラムを構築することができます。必要に応じて、ディレクトリ階層を複製して他のプログラマに提供し、複数のプログラマが並行して開発およびテストを同時に行うことができます (ただし、考慮すべき点はあります)。

また、make を使用して、プロジェクト全体を構築し、完成したさまざまなモジュールを統合して配布するために他のファイルシステムにコピーすることができます。

プロジェクトを整理して管理を簡単にする

前述のように、プロジェクトを整理するには、プロジェクトを主な構成要素ごとにいくつかのディレクトリに分割するのが適しています。このようにして分割したプロジェクトは、通常は 1 つのファイルシステム上またはディレクトリ階層内で管理します。ヘッダーファイル、ライブラリ、プログラムはそれぞれ別のサブディレクトリに置きます。参照マニュアルなどのドキュメントも、別のサブディレクトリに置いて管理します。

次の図に示すように、プロジェクトが 1 つの実行可能プログラム、1 つのユーザー定義ライブラリ、ライブラリのルーチン用の 1 組のヘッダー、複数のマニュアルで構成されているものとします。



各サブディレクトリにあるメークファイルは、これまでの節で説明したものを使用できますが、プロジェクト全体を管理するためのメークファイルがさらに必要です。プロジェクトのルートディレクトリにあるメークファイルには、プロジェクトを 1 つの構成要素として一括管理するためのターゲットエントリを指定します。

プロジェクトが大きくなると、簡単に使用できる整合性のあるメークファイルが必要になります。マクロおよびターゲット名は、どのメークファイルでも意味が同じである必要があります。出力形式を決定する条件付きマクロ定義およびコンパイルオプションは、プロジェクト全体で一貫している必要があります。

可能であれば、テンプレートを使用してメークファイルを記述します。テンプレートを使用して、プロジェクトがどのように構築されるかを監視します。モジュール用のディレクトリを作成し、適切なメークファイルをそのディレクトリにコピーして、数行を編集するだけで、新しい種類のモジュールを追加することができます。また、ルートのメークファイルで構築する新しいモジュールを追加する必要があります。

デフォルトのメークファイルなどで使用されるマクロおよびターゲットの名前の付け方は、プロジェクト全体で統一する必要があります。ニーモニック名とは、ターゲットの機能やマクロの値を正確に覚えていなくても、その名前から機能または値の種類を判断することができるような名前です。ニーモニックは、メークファイルを解読する際にも便利です。

メークファイルをインクルードする

一貫したコンパイル環境を保ちながら、メークファイルを簡潔にする方法として、以下のように `make` を使用します。

include *filename*

この include 指令は、*filename* に指定した名前のメークファイルの内容を読み取ります。指定した名前のファイルがない場合は、make は /etc/default でその名前のファイルを検査します。

たとえば、以下のようにターゲットエントリをインクルードすると、各メークファイルごとに troff ソースを処理するパターンマッチングの規則を重複して記述する必要はありません。

```
SOURCES= doc.ms spec.ms
...
clean: $(SOURCES)
include ../pm.rules.mk
```

この例では、make は ../pm.rules.mk ファイルの内容を読み取ります。

```
# pm.rules.mk
#
# パターンマッチング規則用の "include" makefile
#

%.tr: %.ms
        troff -t -ms $< > $@
%.nr: %.ms
        nroff -ms $< > $@
```

完成したプログラムおよびライブラリをインストールする

外部でのテストまたは通常の使用を目的として、プログラムをリリースする際には、make を使用してプログラムをインストールできます。インストールを行うための新しいターゲットおよびマクロ定義は、以下のように簡単に追加できます。

```
DESTDIR= /proto/project/bin

install: functions
        -mkdir $(DESTDIR)
        cp functions $(DESTDIR)
```

ライブラリまたはヘッダーをインストールする際にも、同様のターゲットを使用できます。

プロジェクト全体の構築

ある時点のソースおよびそれによって構築されるオブジェクトファイルを、プロジェクト開発中に何度か保存する必要があります。プロジェクト全体を構築するには、make を各サブディレクトリごとに実行して、各モジュールを構築し、それをイ

ンストールします。以下の例は、入れ子にした make コマンドを使用して単純なプロジェクトを構築する方法を示しています。

プロジェクトが bin と lib という 2 つの異なるサブディレクトリに置かれていて、両方のサブディレクトリで、make を使用して、プロジェクトのデバッグ、テスト、インストールを行うものとします。

最初に、プロジェクトのルートディレクトリ (プロジェクトの一番上のディレクトリ) に、以下のようなメークファイルを置きます。

```
# プロジェクトのルート makefile

TARGETS= debug test install
SUBDIRS= bin lib

all: $(TARGETS)
$(TARGETS):
         @for i in $(SUBDIRS) ; \
         do \
               cd $$i ; \
               echo "Current directory:  $$i" ;\
               $(MAKE) $@ ; \
               cd .. ; \
         done
```

次に、各サブディレクトリ (この場合は bin) に、以下の一般的な形式のメークファイルを作成します。

```
#サブディレクトリ中の makefile
debug:
         @echo "   Building debug target"
         @echo
test:
         @echo "   Building test target"
         @echo
install:
         @echo "   Building install target"
         @echo
```

プロジェクトのルートディレクトリで make と入力すると、以下のように出力されます。

```
$ make
Current directory:  bin
          Building debugging target

Current directory:  lib
          Building debugging target

Current directory:  bin
          Building testing target

Current directory:  lib
          Building testing target
```

```
Current directory:  bin
          Building install target

Current directory:  lib
          Building install target
$
```

再帰的なメークファイルを使用してディレクトリ階層を管理する

プロジェクトの階層を拡張する場合は、各中間ディレクトリのメークファイルが、ターゲットファイルを生成するだけでなく、その各メークファイルのサブディレクトリ用に入れ子にした make コマンドを呼び出す必要があります。

現在のディレクトリのファイルは、サブディレクトリのファイルに依存する場合があります。その場合は、ターゲットエントリはサブディレクトリにある対応するターゲットエントリに依存している必要があります。

各サブディレクトリ用の入れ子にした make コマンドは、ローカルディレクトリのコマンドの前に実行する必要があります。入れ子にしたコマンドとローカルディレクトリのコマンドに対して別々にエントリを作成すると、コマンドを正しい順序で実行できます。このような新しいターゲットを、元のターゲットの依存関係リストに追加すると、オリジナルのターゲットの動作と新しいターゲットの動作の両方が実行されます。

再帰的なターゲットの管理

ローカルディレクトリとサブディレクトリの両方で同一の動作を行うターゲットを、再帰的なターゲットと呼びます。

注 - 厳密には、ターゲット自身の名前を引数として指定して make を呼び出すターゲットは、すべて再帰的です。ただしここでは、入れ子になった動作とローカルの動作の両方を持つターゲットのみを「再帰的なターゲット」と呼びます。入れ子にした動作のみを持つターゲットは、「入れ子にしたターゲット」と呼びます。

再帰的なターゲットを含むメークファイルは、再帰的なメークファイルと呼びます。

以下の例の all の場合は、入れ子にした依存関係は NESTED_TARGETS、ローカルの依存関係は LOCAL_TARGETS です。

```
NESTED_TARGETS=  debug test install
SUBDIRS= bin lib
LOCAL_TARGETS= functions
```

```
all: $(NESTED_TARGETS) $(LOCAL_TARGETS)

$(NESTED_TARGETS):
         @ for i in $(SUBDIRS) ; \
         do \
               echo "Current directory:  $$i" ;\
               cd $$i ; \
               $(MAKE) $@ ; \
               cd .. ; \
         done

$(LOCAL_TARGETS):
         @ echo "Building $@ in local directory."
       (local directory commands)
```

入れ子にした make も、最下位の階層にある場合を除き、再帰的である必要があります。末端のディレクトリ (サブディレクトリを持たないディレクトリ) のメークファイルでは、ローカルターゲットのみを構築します。

大規模なライブラリを分割して管理する

大規模なライブラリは、複数の補助ライブラリに分割し、完全なパッケージを構築する際に make を使用して 1 つに結合すると、管理が簡単になる場合があります。ar を使用してライブラリを直接結合することはできませんが、以下の例に示すように、まず各補助ライブラリのメンバーファイルを取り出し、次に別の手順でそれらのファイルをアーカイブにまとめることができます。

```
$ ar xv libx.a
x - x1.o
x - x2.o
x - x3.o
$ ar xv liby.a
x - y1.o
x - y2.o
$ ar rv libz.a *.o
a - x1.o
a - x2.o
a - x3.o
a - y1.o
a - y2.o
ar: creating libz.a
```

補助ライブラリは、そのライブラリを構築する (オブジェクト) ファイルとともに、そのディレクトリにあるメークファイルを使用して管理します。完成ライブラリ用のメークファイルは、各補助ライブラリアーカイブへのシンボリックリンクを作成し、アーカイブの内容を一時的なサブディレクトリに展開し、生成されたファイルをアーカイブして完全なパッケージを作成します。

次の例は、補助ライブラリを更新し、展開したファイルを保存する一時ディレクトリを作成して、補助ライブラリを展開します。一時ディレクトリでは、ワイルドカード * (シェル) を使用して、照合したファイルのリストを生成します。通常は、ファイル名にワイルドカードは使用しない方がよいですが、この例ではターゲットが構築されるたびに新しいディレクトリが作成されるため、使用が可能です。これによって、実行中の make により展開されたファイルだけが一時ディレクトリに含まれます。

注 - 通常は、メークファイル中でシェルファイル名にワイルドカードを使用することは避けてください。使用する場合は、必要なファイルを一時サブディレクトリに移動することによって対称外のファイルを除外するような処理手順にしてください。

この例は、ディレクトリの命名規約を使用しています。ディレクトリ名は、そのディレクトリにあるライブラリのベース名から決定されます。たとえば、補助ライブラリの名前が libx.a の場合は、その補助ライブラリがあるディレクトリの名前は libx になります。

この例は、動的なマクロ参照での接尾辞置換を使用して、各サブディレクトリのディレクトリ名を取り出します。各ライブラリを順次取り出すためのループとしてシェルを使用します。また、ライブラリからパッケージを作成する際に、シェルコマンド置換を使用して、オブジェクトファイルを正しいリンク順に並び替えます (lorder と tsort を利用します)。最後に、一時ディレクトリおよびその内容を削除します。

```
# サブディレクトリで生成されたライブラリをまとめる Makefile

CFLAGS= -O

.KEEP_STATE:
.PRECIOUS:  libz.a

all: lib.a

libz.a: libx.a liby.a
         -rm -rf tmp
         -mkdir tmp
         set -x ; for i in libx.a liby.a ; \
                do ( cd tmp ; ar x ../$$i ) ; done
         ( cd tmp ; rm -f *_*_.SYMDEF ; ar cr ../$@ `lorder * | tsort` )
         -rm -rf tmp libx.a liby.a

libx.a liby.a: FORCE
         -cd $(@:.a=) ; $(MAKE) $@
         -ln -s $(@:.a=)/$@ $@
FORCE:
```

説明を簡潔にするため、この例では他の形式のプログラム
と、clean、install、test ターゲットはサポートしていません (ソースファイルがサブディレクトリにあるため、適用できません)。

インデントされた行にある rm -f *_*_.SYMDEF というコマンドは、補助ライブラリのシンボルテーブル (補助ライブラリに対して ar を実行すると生成されます) がこの完成ライブラリ libz.a にアーカイブされないようにします。

入れ子にした make コマンドは現在のライブラリを処理する前に補助ライブラリを構築するため、現在のディレクトリにある補助ライブラリとオブジェクトファイルの両方から構築されるライブラリ用にこのメークファイルを拡張できます。オブジェクトファイルのリストを、ライブラリの依存関係リストに追加する必要があります。また、そのオブジェクトファイルを、補助ライブラリから展開したオブジェクトファイルと照合するための一時サブディレクトリにコピーするためのコマンドを追加する必要があります。

```
# サブディレクトリで生成されたライブラリと、オブジェクトをまとめる Makefile

CFLAGS= -O

.KEEP_STATE:
.PRECIOUS:  libz.

OBJECTS= map.o calc.o draw.o

all: libz.a

libz.a: libx.a liby.a $(OBJECTS)
          -rm -rf tmp
          -mkdir tmp
          -cp $(OBJECTS) tmp
          set -x ; for i in libx.a liby.a ; \
                do ( cd tmp ; ar x ../$$i ) ; done
          ( cd tmp ; rm -f *_*_.SYMDEF ; ar cr ../$@ \
                `lorder * | tsort` )
          -rm -rf tmp lix.a liby.a

libx.a liby.a: FORCE
          -cd $(@:.a=) ; $(MAKE) $@
          -ln -s $(@:.a=)/$@ $@
FORCE:
```

隠れた依存関係を make にレポートする

たとえば、.so を要求して、troff ドキュメントにインクルードされているソースファイルをトレースする必要があるなどの場合は、隠れた依存関係を処理するコマンドを記述する必要があります。.KEEP_STATE が有効な場合は、make は環境変数 SUNPRO_DEPENDENCIES を次のように設定します。

```
SUNPRO_DEPENDENCIES='report-file target'
```

コマンドが終了すると、make はファイルが作成されたかどうかを調べ、作成されていれば、そのファイルを読み取り、報告された依存関係を以下の形式で .make.state に書き込みます。

```
target:dependency ...
```

ここで、target は環境変数に指定されているものと同じです。

make の拡張機能のまとめ

以下では、make に新しく追加された機能について説明しています。

デフォルトのメークファイル

make の暗黙の規則およびマクロ定義は、このバージョンでは make プログラム中に定義 (ハードコード) されずに、デフォルトのメークファイルである /usr/share/lib/make/make.rules に含まれています。ローカルディレクトリに make.rules というファイルがある場合を除き、このデフォルトのメークファイルを自動的に読み取ります。ローカルディレクトリにある make.rules ファイルを使用する際は、標準の暗黙の規則および定義済みマクロを取得するために、デフォルトの make.rules ファイルをインクルードする指示を追加する必要があります。

状態ファイル .make.state

make は、状態ファイル .make.state も読み取ります。特殊ターゲットの .KEEP_STATE がメークファイル中で使用されている場合は、make は、隠れた依存関係のリスト (cpp などのコンパイル処理によりレポートされます) と、各ターゲットを構築するために使用された最新の規則が記述されている、各ターゲットのについての記録をこのファイルに書き込み蓄積します。状態ファイルの書式は、通常のメークファイルの書式とほぼ同じです。

隠れた依存関係の検査

make が .KEEP_STATE ターゲットにより起動された場合は、cc、cpp、f77、ld、make、pc などのコンパイルコマンドによって報告された情報

を使用して、ターゲットファイルにインクルードされたヘッダーファイル (場合によってはライブラリ) に対して依存関係の検査を実行します。これらの隠れた依存ファイルは、依存関係リストには記述されていません。多くの場合はローカルディレクトリ以外の場所に存在しています。

コマンド依存関係の検査

.KEEP_STATE が有効な場合に、ターゲットを構築するために使用されるコマンド行が以前の make 実行時から変更された場合 (メークファイルを編集したりマクロの値を変更した場合など)、ターゲットは最新でないとして扱われ、(ターゲットが依存するファイルよりもターゲットが新しい場合でも) make はターゲットを再構築します。

SCCS ファイルの自動取り出し

この節では、sccs が管理しているファイルの自動取り出しの規則を説明します。

チルド規則について

このバージョンの make は、ターゲットファイルを構築する規則がない場合には、sccs getx を適宜自動的に実行します。接尾辞リストで接尾辞の後にチルド (~) が付いている場合は、sccs によって依存ファイルを取り出すのが適切であることを示します。このバージョンの make は、sccs ファイルの現在のバージョンを取り出すためのコマンドを含むチルドの接尾辞の規則をサポートしていません。

現在の sccs のバージョンの自動取り出しを禁止または許可するには、.SCCS_GET という特殊ターゲットを再定義します。このターゲットの規則を空白にすると、すべてのファイルに対して自動取り出しを禁止します。

パターンマッチングの規則

ユーザー定義の暗黙の規則を簡単にプロジェクトに追加する方法を簡素化するために、パターンマッチングの規則が追加されています。

```
tp%ts : dp%ds
      rule
```

この形式のターゲットエントリは、関連する依存ファイルからターゲットを構築するパターンマッチングの規則を定義します。tp はターゲット名の接頭辞、ts はその接尾辞です。dp は依存ファイル名の接頭辞、ds はその接尾辞です。% 記号は、ターゲットと依存ファイルの名前の両方に共通する、0 文字または 1 文字以上の連続した文字列を示すワイルドカードです。たとえば、以下のターゲットエントリは、接尾辞が .ms で –ms というマクロパッケージを使用するファイルから接尾辞が .tr の troff 出力ファイルを構築するパターンマッチングの規則を定義します。

```
%.tr: %.ms
  troff -t -ms $< > $@
```

メークファイルにこのエントリが含まれている場合に、以下のコマンドを実行します。

```
make doc.tr
```

結果は以下のようになります。

```
$ make doc.tr
troff -t -ms doc.ms > doc.tr
```

doc2.ms というファイルがある場合に同じエントリを使用して、以下のコマンドを実行します。

```
make doc2.tr
```

結果は以下のようになります。

```
$ make doc2.tr
troff -t -ms doc2.ms > doc2.tr
```

明示的なターゲットエントリは、ターゲットに適用されるパターンマッチングの規則よりも優先されます。パターンマッチングの規則は、通常は暗黙の規則よりも優先されます。例外的に、パターンマッチングの規則のターゲットエントリにおいて、規則の部分にコマンドが含まれていない場合は、make はターゲットを構築するための規則の検索を続行し、その依存関係として (依存関係) パターンに一致したファイルを使用します。

パターン置換マクロ参照

接尾辞の規則およびパターンマッチングの規則と同様に、指定したマクロ参照の語を変更する方法として、マクロ参照で接尾辞を置換する既存の方法よりもさらに汎用性が高い、パターン置換マクロ参照が追加されました。パターン置換マクロ参照は、以下の形式で記述します。

```
$(macro:p%s=np %ns)
```

ここで、p は既存の接頭辞 (ある場合)、s は既存の接尾辞 (ある場合)、np および ns は新しい接頭辞および接尾辞、% は 0 文字または 1 文字以上の一致文字列を示すワイルドカードです。

接頭辞および接尾辞の置換は、既存のパターンに一致するマクロの値中のすべての語に適用されます。この機能は、特にサブディレクトリの名前をそのサブディレクトリに含まれる各ファイルの接頭辞として使用する場合に便利です。以下にメークファイルの例を示します。

```
SOURCES= x.c y.c z.c
SUBFILES.o= $(SOURCES:%.c=subdir/%.o)

all:
        @echo $(SUBFILES.o)
```

結果は以下のようになります。

```
$ make
subdir/x.o subdir/y.o subdir/z.o
```

= 記号の右側 (置換後の語) では、任意の数だけワイルドカードの % を必要に応じて使用できます。以下に例を示します。

```
...
NEW_OBJS= $(SOURCES:%.c=%/%.o)
```

この置換は、以下のような結果になります。

```
...
x/x.o y/y.o z/z.o
```

パターン置換マクロ参照は、パターンマッチングの規則が記述されているターゲットエントリの依存関係を示す行では使用しないでください。使用した場合は、予期しない結果が生じます。以下にその例を示します。

```
OBJECT= .o

x:
%: %.$(OBJECT:%o=%Z)
        cp $< $@
```

このメークファイルは、make が x.Z というファイルから x を構築しようとして記述したものです。しかし、依存関係の行に複数含まれている % のうち、どれをパターンマッチングの規則で使用し、どれをマクロ参照に適用するかを、make が特定できないため、パターンマッチングの規則は認識されません。

したがって、x.Z のターゲットエントリは実行されません。このような問題を回避するため、他の行で中間マクロを使用できます。

```
OBJECT= .o
ZMAC= $(OBJECT:%o=%Z)

x:
%: %$(ZMAC)
        cp $< $@
```

新しいオプション

新しいオプションは、以下のとおりです。

–d	処理した各ターゲットの依存関係の検査結果を表示します。新しい依存関係、あるいはターゲットがコマンドの依存関係の結果構築されたことを示す依存関係をすべて表示します。
–dd	旧バージョンの make の –d と同一の機能を実行します。内部状態など、make のすべての実行情報を表示します。
–D	メークファイルのテキスト (内容) を読み取り時に表示します。
–DD	メークファイルおよび使用されているデフォルトのメークファイルのテキストを表示します。
–p	マクロ定義およびターゲットエントリを出力します。
–P	ターゲットを再構築せずに、その依存関係をすべてレポートします。

C++ および Modula-2 のサポート

このバージョンの make には、C++ プログラムをコンパイルするための定義済みマクロが含まれています。また、Modula-2 をコンパイルするための定義済みマクロおよび暗黙の規則も含まれています。

定義済みマクロの命名規約

定義済みマクロの命名規約が合理的になりました。新しい命名規約に合わせて暗黙の規則が変更されました。マクロおよび暗黙の規則は、既存のメークファイルに対して上位互換性があります。

標準のコンパイルコマンド用のマクロを例に示します。

`LINK.c`

これは、実行可能ファイルを生成するための標準の `cc` コマンド行です。

`COMPILE.c`

これは、オブジェクトファイルを生成するための標準の `cc` コマンド行です。

新しい特殊ターゲット

`.KEEP_STATE`	メークファイルに記述されている場合は、隠れた依存関係およびコマンドの依存関係の検査を有効にします。また、`make` は、実行後に状態ファイル `.make.state` を更新します。

注 - `.KEEP_STATE` ターゲットは、`make` の実行で使用したことがある場合は削除しないでください。

`.INIT` および `.DONE`	`make` 実行時に、最初と最後に実行するコマンドを指定します。
`.FAILED`	`make` が失敗したときに実行するコマンドを指定します。
`.PARALLEL`	ターゲットを並行処理するか逐次処理するかを指定します。
`.SCCS_GET`	このターゲットには、`sccs` 履歴ファイルから現在のバージョンのファイルを取り出す規則を指定します。
`.WAIT`	このターゲットが依存関係リストに記述されている場合は、並行処理の場合でも、`make` は先

行する依存関係の処理が終了するまでそれ以降の処理を行わずに待機します。

lint の新しい暗黙の規則

lint を使用した差分検査をサポートするため、暗黙の規則が追加されました。

マクロ処理の変更

このバージョンでは、マクロの値を任意の長さにすることができます。また、旧バージョンではマクロの値から最後の空白文字だけが削除されましたが、このバージョンでは最初と最後の両方の空白文字が削除されます。

マクロ：定義、置換、接尾辞置換

新しい追加演算子

`+=`

マクロの既存の値に、先頭にスペースが付いた文字列を追加します。

条件付きマクロ定義

`:=`

ターゲットの条件付きマクロ定義を示す演算子です。

```
target := macro = value
```

このようにメークファイルで記述すると、target およびその依存関係を処理する際には、macro が value に指定された値になります。

条件付きマクロのパターン

make は、条件付きマクロ定義のターゲット部分において、ワイルドカードパターンである % を認識します。以下に例を示します。

```
profile_% := CFLAGS += -pg
```

この例は、profile_ という接頭辞を持つすべてのターゲットについて、CFLAGS マクロを変更します。条件付き定義の値では、パターン置換を使用できます。以下に例を示します。

```
profile_% := OBJECTS = $(SOURCES:%.c=profile_%.o)
```

この例は、SOURCES 値に指定されているすべての .c ファイルのベース名の前後に、profile_ という接頭辞および .o という接尾辞を追加します。

接尾辞の置換の優先度

部分文字列の置換は、参照されるマクロが展開されてから実行されます。旧バージョンの make では、置換が先に実行されるため、結果を直観的に理解しにくくなっていました。

入れ子にしたマクロ参照

このバージョンの make は、内側の参照を展開してから外側の参照を展開します。以下に例を示します。

```
CFLAGS-g = -I../include
  OPTION = -g
  $(CFLAGS$(OPTION))
```

この例の入れ子にした参照の値は、旧バージョンでは NULL 値になりましたが、このバージョンでは -I../include になります。

クロスコンパイルマクロ

定義済みマクロの HOST_ARCH および TARGET_ARCH をクロスコンパイルで使用できます。デフォルトでは、*arch* マクロは、arch コマンドが返す値に設定されます。

マクロでのシェルコマンド出力

以下の定義は、command で指定されたコマンドの標準出力を MACRO の値として設定します。

```
MACRO :sh = command
```

出力に含まれる復帰改行は、空白文字に置換されます。コマンドは、定義が読み込まれたときに 1 回だけ実行されます。標準エラー出力は無視されます。コマンドがゼロ以外の終了ステータスを返した場合は、make はエラーを表示して停止します。

```
$(MACRO :sh)
```

このマクロ参照は、参照が評価されるときに MACRO に設定されているコマンド行出力に展開されます。復帰改行は、空白文字に置換されます。標準エラー出力は無視されます。コマンドがゼロ以外の終了ステータスを返した場合は、make はエラーを表示して停止します。

ar ライブラリのサポートについて

make は、ar フォーマットのライブラリのメンバーを、そのメンバーと名前が同じファイルから自動的に更新します。また、このバージョンの make では、以下の形式で、依存関係の名前としてメンバーのリストを指定できます。

```
lib.a: lib.a(member member ...)
```

ターゲットグループ

このバージョンでは、複数ターゲットファイルで構成されるグループを生成するように規則を指定することができます。ターゲットエントリ中で各ターゲット名が + 記号で区切られている場合は、指定したターゲットがグループを構成していることを示します。ターゲットグループの規則は、make の実行 1 回につき 1 度だけ実行されます。

旧バージョンとの非互換性

この節では、以下について簡単に説明しています。

- –d オプション
- 動的なマクロ
- チルド規則
- ターゲット名

–d オプション

–d オプションは、このバージョンでは、ターゲットが最新でないとされる理由を出力します。

動的なマクロ

動的なマクロの $< および $* は、旧バージョンでは、暗黙の規則と .DEFAULT という特殊ターゲットでのみ値が代入されると説明されていましたが、実際には、明示的なターゲットエントリにも値が代入される場合がありました。このバージョンでは、値の代入について正しく説明されています。

これらのマクロに実際に割り当てられる値は、暗黙の規則 (この規則は変更されていません) で使用されるのと同一の手順で取得されます。明示的なターゲットエントリでこれらのマクロを使用すると、予期しない結果になる場合があります。

明示的な依存関係を指定しても、make はその依存関係を使用して前述のマクロの値を取得することはせず、適切な暗黙の規則および依存ファイルを検索します。以下に例を示します。

```
test: test.f
        @echo $<
```

上記のように明示的なターゲットエントリを指定したときに、test.c と test.f というファイルがある場合には、$< には test.f という値ではなく test.c が割り当てられます。これは、接尾辞のリスト中で .c の方が .f よりも先にあるためです。

```
$ make test
test.c
```

明示的なエントリでは、厳密な指定方法でマクロ参照および接尾辞の置換を使用して、依存関ファイルの名前を取得することをお勧めします。たとえば、$< の代わりに $@.f を使用して、依存関係の名前を取得できます。.o のターゲットファイルのベース名を取得するには、接尾辞の置換マクロ参照である $(@:.o=) を $* の代わりに使用できます。

隠れた依存関係の検査が有効である場合は、$? という動的なマクロの値に、ヘッダーファイルなどの隠れた依存関係の名前が含まれます。このため、以下のようなターゲットエントリで、x.c がヘッダーファイルを展開すると、コンパイルが失敗する場合があります。

```
x: x.c
        $(LINK.c) -o $@ $?
```

これを解決するには、$? の代わりに $@.< を使用します。

チルド規則

チルド規則はサポートされていません。このバージョンの make は、SCCS でバージョンを取り出すためのチルド規則をサポートしていません。このため、SCCS でバージョンを取り出す必要があるときに特別な処理を実行するためのチルド規則が、旧バージョンのメークファイルで再定義されている場合には、問題が生じることがあります。

ターゲット名

./ で始まるターゲット名は、ローカルディレクトリにあるファイル名として扱われます。

./ で始まるターゲット名を make が検出した場合は、./ を削除します。以下に例を示します。

```
./filename
```

このターゲット名は、以下のように解釈されます。

```
filename
```

再帰的なターゲットで使用された場合は、無限状態になることがあります。これを回避するには、.. (親ディレクトリ) からの相対パスでターゲットを記述してください。

```
../dir/filename
```

第 5 章

SCCS ソースコード管理システム

ソースファイルが複数のユーザーによって変更される可能性がある場合は、ソースファイルへの書き込み権を調整する必要があります。更新記録を管理することによって、変更がいつどのような理由で行われたかを特定することができます。

ソースコード管理システム (SCCS) を使用して、ソースファイルへの書き込み権の管理およびソースファイルの変更の監視を行うことができます。SCCS は、同時に 1 人のユーザーだけにファイルの更新を許可し、履歴ファイルにすべての変更を記録します。

SCCS を使用して、以下のことを行うことができます。

- 編集するファイルの任意のバージョンのコピーを SCCS 履歴から取り出すことができます。
- ファイルのバージョンをチェックアウトおよびロックし、他のユーザーによる変更を禁止します。あるユーザーが行なった編集を消去してしまうような変更を、他のユーザーが勝手に行うことを防止します。
- ファイルに更新をチェックインします。ファイルをチェックインする際に、変更内容を要約したコメントを入力することもできます。
- チェックアウトしたコピーへの変更を取り消すこともできます。
- 編集可能なファイルを確認できます。
- 選択した複数のバージョン間の違いを確認できます。
- これまでにチェックインされた変更を要約したバージョンログを表示できます。

sccs コマンド

ソースコード管理システムは、sccs(1) コマンドで構成されます。このコマンドは、/usr/ccs/bin ディレクトリにあるユーティリティプログラムのフロントエンドです。SCCS のユーティリティプログラムの一覧は、表 5–2 を参照してください。

sccs create コマンド

sccs create コマンドは、ファイルを SCCS に管理させます。このコマンドは、新しい履歴ファイルを作成し、ソースファイルのすべての内容を最初のバージョンとして使用します。デフォルトでは、履歴ファイルは SCCS サブディレクトリにあります。

```
$ sccs create program.c
program.c:
1.1
87 lines
```

SCCS は、作成されたファイルの名前、バージョン番号 (1.1)、行数を出力します。

元のファイルが誤って喪失または損傷してしまったときのために、sccs create は新しいファイル名の先頭にコンマを付けたファイルに (これをコンマファイルと呼びます) 元のファイルへの 2 つ目のリンクを作成します。履歴ファイルの初期化に成功すると、SCCS は新たに読み取り専用バージョンを取り出します。このバージョンとそのコンマファイルとを比較して検証した後は、コンマファイルを削除してください。

```
$ cmp ,program.c program.c
(何も出力されない場合は、ファイルが完全に一致していることを意味します)
$ rm ,program.c
```

SCCS が取り出す読み取り専用バージョンは、まだ編集しないでください。ファイルを編集するには、後述の sccs edit コマンドを使用してファイルをチェックアウトする必要があります。

履歴ファイルと現在のバージョンとを区別するため、履歴ファイルの名前に s. という接頭辞が使用されます。この履歴ファイルはしばしば s. ファイル (*s* ドットファイル) とも呼ばれます。また、履歴ファイルは、慣用的に *SCCS* ファイルとも呼ばれます。

SCCS 履歴ファイルの形式については、sccsfile(4) を参照してください。

基本的な sccs サブコマンド

以下の sccs サブコマンドは、バージョン管理機能を実行します。以下にサブコマンドを要約しています。create 以外のサブコマンドについての詳細は、207ページの「sccs サブコマンド」で説明しています。

create	前述の履歴ファイルおよび最初のバージョンを初期化します。
edit	編集を行うために書き込み可能なバージョンをチェックアウトします。このコマンドを実行したユーザーを所有者として書き込み可能なコピーを取り出し、他のユーザーが変更をチェックインできないように、履歴ファイルをロックします。
delta	変更をチェックインします。このサブコマンドによって、sccs edit 処理が完了します。変更を記録する前に、コメントを入力するためのプロンプトが表示されます。コメントは、履歴ファイル中のバージョンログに保存されます。
get	ファイルの読み取り専用コピーを s. ファイルから取り出します。デフォルトでは、最新のバージョンが取り出されます。取り出されたバージョンは、コンパイル、フォーマット、表示を行うためのソースファイルとして使用できます。ただし、このバージョンは編集または変更を行うためのものではありません。読み取り専用バージョンのファイルアクセス権を変更すると、ファイルへの変更が無効になる場合があります)。 ディレクトリをファイル名引数として指定すると、sccs はそのディレクトリ内のすべての s. ファイルに対して get サブコマンドを実行します。 `sccs get SCCS` 上記のコマンドは、SCCS サブディレクトリ内のすべての s.file の読み取り専用バージョンを取り出します。

prt	バージョンログと、各バージョンに対応するコメントを表示します。

デルタとバージョン

バージョンをチェックインすると、SCCS は、チェックインしたテキストと直前のバージョンとの、行ごとの相違だけを記録します。この相違をデルタと呼びます。edit または get により取り出されるバージョンは、それまでにチェックインされたデルタから構成されます。

「デルタ」と「バージョン」は、しばしば同義語として使用されます。ただし、これらの意味は正確には同一ではありません。選択したデルタを省略したバージョンを取り出すこともできます (217ページの「取り出したバージョンからデルタを削除する」を参照してください)。

SID

SCCS のデルタ ID (SID) は、特定のデータを示すための番号です。SID は、ドット (.) で区切った 2 つの番号で構成されます。最初のデルタの SID は、デフォルトでは 1.1 です。SID の最初の部分はリリース番号、2 番目の部分はレベル番号です。デルタをチェックインすると、レベル番号が自動的に 1 つずつ増えます。リリース番号は、必要に応じて増やすことができます。SCCS は、この他に分岐デルタ用の 2 つのフィールドも認識します (分岐デルタについては 222ページの「分岐」を参照してください)。

厳密には、SID はデルタそのものを示しますが、デルタとそれ以前のバージョンから構築したバージョンのことを「デルタ」という場合もあります。

ID キーワード

SCCS は、ソースファイル中の特定のキーワードを認識して展開します。このキーワードを使用して、チェックインするバージョンのテキストにそのバージョン固有の情報 (SID など) を含めることができます。編集を行うファイルをチェックアウトするときに、ID キーワードは以下のようになります。

```
%C%
```

ここで、C は大文字です。ファイルをチェックインする際に、SCCS はキーワードを対応する情報に置換します。たとえば、%I% は、現在のバージョンの SID に展開されます。

ID キーワードは、一般的にはコメントまたは文字列の定義中に含めます。少なくとも 1 つの ID キーワードがソースファイルに含まれていない場合は、SCCS は以下のような診断を出力します。

```
No Id Keywords (cm7)
```

ID キーワードについての詳細は、211ページの「ID キーワードを使用してバージョン固有情報を組み込む」を参照してください。

SCCS サブコマンド

ファイルのチェックインおよびチェックアウト

以下のサブコマンドは、バージョンを取り出したり変更をチェックインする際に使用します。

編集用のファイルをチェックアウトする：sccs edit

ソースファイルを編集するには、まず sccs edit を使用してファイルをチェックアウトする必要があります (sccs edit コマンドは、sccs get で –e オプションを使用する場合と同じ結果になります)。

SCCS は、取り出したバージョンのデルタ ID と、変更をチェックインしたときに割り当てられる新しいデルタ ID を返します。

```
$ sccs edit program.c
1.1
new delta 1.2
87
```

取り出したファイルは、テキストエディタを使用して編集できます。ファイルの書き込み可能なコピーがある場合は、sccs edit はエラーメッセージを表示します。つまり sccs edit は、他のユーザーにもファイルへの書き込み権がある場合は、そのファイルを上書きしません。

新しいバージョンをチェックインする：sccs delta

ファイルをチェックアウトして編集が完了したら、sccs delta を使用して変更をチェックインできます。

ファイルをチェックインすることを、デルタを作成するということもあります。更新をチェックインする前に、コメントを入力するプロンプトが表示されます。コメントとして、変更についての要約を記述します。

```
$ sccs delta program.c
comments?
```

コメントは、後でそのファイルを使用する時のために、わかりやすい内容にする必要があります。

バックスラッシュ (\) と復帰改行を入力することによって、コメントを 2 行以上に渡って記述することができます。

```
$ sccs delta program.c
comments? corrected typo in widget(), \
null pointer in n_crunch()
1.2
5 inserted
3 deleted
84 unchanged
```

SCCS は、新しいバージョンの SID と、挿入された行数、削除された行数、変更されていない行数を表示します。変更された行数は、削除された行と挿入された行の合計になります。SCCS は、作業用のコピーを削除します。sccs get を使用して、読み取り専用バージョンを取り出すことができます。

バージョンをチェックインする際には注意が必要です。少量の編集を行うたびにデルタを作成していくと、デルタが多くなりすぎる場合があります。逆に長期間ファイルをチェックアウトしたままにすると、他のユーザーがそのファイルを編集できないので不便になる場合があります。

一般に使用されるモジュールをコンパイルまたはインストールする前に、変更したすべてのファイルをチェックインしてください。手順は以下のとおりです。

- 必要なファイルを編集します。
- 必要な変更およびテストを行います。
- ファイルのコンパイルおよびデバッグを行います。
- ファイルをチェックインし、get を使用して読み取り専用のコピーを取り出します。
- モジュールを再コンパイルします。

バージョンを取り出す：sccs get

ファイルの最新バージョンを取得するには、以下のコマンドを使用します。

```
sccs get filename
```

以下に例を示します。

```
$ sccs get program.c
1.2
86
```

この例では、program.c を取り出し、バージョン番号および取り出された行数が表示されています。取り出された program.c のコピーは、読み取り専用になっています。

SCCS は、ファイルをチェックアウトしない限り新しいデルタを作成しないため、この取り出した読み取り専用のコピーは変更 (編集) しないでください。取り出した読み取り専用のコピーをそのまま強制的に変更すると、他のユーザーが次に sccs get または sccs edit をそのファイルに対して実行したときに、そのファイルに加えた変更が無効になる場合があります。

保留中の変更を確認する：sccs diffs

チェックアウトしたバージョンの変更で、まだチェックインされていないものを、保留中の変更と呼びます。ファイルの編集中に、sccs diffs を使用して保留中の変更を確認できます。diffs サブコマンドは、diff(1) を使用して、作業用 (編集中) のコピーとチェックインされた最新のバージョンとを比較します。

```
$ sccs diffs program.c
------ program.c ------
37c37
<         if (((cmd_p - cmd) + 1) == l_lim) {
---
>         if (((cmd_p - cmd) - 1) == l_lim) {
```

diff コマンドのほとんどのオプションを使用することができます。diff の –c オプションを呼び出すには、sccs diffs に –C という引数を使用します。

保留中の変更を削除する：sccs unedit

sccs unedit は、保留中の変更を取り消します。これは、ファイルの編集を間違ってしまったときに編集を最初からやり直す場合に便利です。unedit は、チェックアウトしたバージョンを削除し、履歴ファイルのロックを解除し、チェッ

クインされている最新のバージョン (最後にチェックインしたバージョン) の読み取り専用コピーを取り出します。unedit を使用すると、ファイルをチェックアウトしなかったのと同じ状態になります。編集を再開するには、sccs edit を使用してファイルをもう一度チェックアウトします (211ページの「書き込み可能なコピーを修復する : sccs get -k -G」も参照してください)。

delta と get を組み合わせて実行する : sccs delget

sccs delget は、delta および get の動作を組み合わせて一度に実行します。sccs delget は、変更をチェックインし、チェックインした新しいバージョンの読み取り専用コピーを取り出します。ただし、SCCS が delta の実行中にエラーが発生した場合は、get は実行されません。ファイル名のリストを (複数のファイルを一度に) 処理する際は、delget は可能なすべての delta を実行し、delta でエラーが発生した場合は、すべての get の処理を実行しません。

delta と edit を組み合わせて実行する : sccs deledit

sccs deledit は、delta を実行した後に edit を実行します。バージョンをチェックインした後、すぐに編集を再開することができます。

SID を指定してバージョンを取り出す : sccs get -r

–r オプションを使用すると、取り出す SID を指定できます。

```
$ sccs get -r1.1 program.c
1.1
87
```

日時を指定してバージョンを取り出す : sccs get -c

必要なデルタの SID は不明でも、チェックインした日付ならわかるという場合があります。以下の形式で –c オプションと日時を指定して、その日時以前にチェックインされた最新のバージョンを取り出すことができます。

```
-cyy [mm [dd [hh [mm [ss ]]]]]
```

以下に例を示します。

```
$ sccs get -c880722120000 program.c
1.2
86
```

この例は、1988 年 7 月 22 日午後 12 時の時点で最新のバージョンを取り出します。年以外のフィールドは省略できます (デフォルトでは現在になります)。また、指定された箇所に区切りとして句切り文字を挿入できます。前述のコマンドは、以下のように書き換えることができます。

```
sccs get -c"88/07/22 12:00:00" program.c
```

注 - 2000 年問題について : SCCS は日付を表わす書式として 2 けたで年を表します。Sun は、年として 69 ～ 99 の値が指定されている場合は 1969 ～ 1999、00 ～ 68 の場合は 2000 ～ 2068 にそれぞれ解釈するという、X/Open グループにより提案された仕様 (XCU5) を採用しています。

書き込み可能なコピーを修復する：sccs get -k -G

`sccs get -k -G`*filename* は、新しいバージョンをチェックアウトせずに、テキストの書き込み可能なコピーを取り出し、`–G` で指定されたファイル名で保存します。このコマンドは、`diff` コマンドとテキストエディタを使用して、損傷した作業中のコピーを置き換えたりまたは修復したりする際に便利です。

ID キーワードを使用してバージョン固有情報を組み込む

前述のように、SCCS では、ID キーワード (SID) を使用して、チェックインするバージョンにバージョン固有の情報を組み込むことができます。ID キーワード (ファイル中に挿入します) は、変更をチェックインする際にそのバージョンに対応する情報に自動的に置換されます。SCCS ID キーワードは、以下のような形式になります。

```
%C%
```

ここで、*C* は大文字の一文字を表わしています。

たとえば、`%I%` は、最新デルタの SID に展開されます。`%W%` は、ファイル名、SID、一意の文字列 `@(#)` をファイルに取り込みます。この文字列 `@(#)` は、`what` コマンドによりテキストファイルと バイナリファイルの両方で検索されます。これによって、ファイルまたはプログラムがどのバージョンのソースから構築されたかを確認することができます。`%G%` キーワードは、最新のデルタの日付に展開されます。その他の ID キーワードとそれらが展開されたときの文字列については、表 5–1 を参照してください。

注 - この方法で文字列を定義すると、C のオブジェクトファイルにバージョン情報が組み込まれます。この方法を使用して ID キーワードをヘッダー (.h) ファイルに組み込む場合は、ヘッダーファイルごとに異なる変数を使用してください。これにより、静的 (static) 変数を再定義しようとするエラーを防止できます。

バージョン固有の情報を C プログラムに取り込むには、以下のようにします。

```
static char SccsId[ ] = "%W%\t%G%";
```

ファイル名が program.c の場合、この行はバージョン 1.2 が取り出されたときに以下のように展開されます。

```
static char SccsId[ ] = "@(#)program.c 1.2 08/29/80";
```

コンパイルされたプログラムで文字列が定義されているため、この方法によりコンパイルしたプログラムにソースファイルの情報を組み込むことができます。ソースファイルの情報は、以下のように what コマンドで確認することができます。

```
$ cd /usr/ucb
$ what sccs
sccs
sccs.c 1.13 88/02/08 SMI
```

シェルスクリプトなどのスクリプトでは、ID キーワードをコメント内に含めることができます。

```
# %W%  %G%
.  .  .
```

展開されたキーワードを含むバージョンをチェックインすると、バージョン固有の情報が更新されなくなります。これを通知するために、get、edit、create 実行時に ID キーワードを検出できないときには、以下のように SCCS は警告を表示します。

```
No Id Keywords (cm7)
```

さまざまな情報の確認

以下のサブコマンドは、ファイルまたはその履歴を問い合わせる際に便利です。

取り出されたバージョンを確認する：what コマンド

SCCS を使用することによって、ファイル履歴中の任意のバージョンを取り出すことができるため、ディレクトリ内にある作業中のコピーが必要なバージョンでないという可能性もあります。what コマンドは、ファイル中の SCCS ID キーワードを

検査します。また、バイナリファイルでもキーワードを検査するため、どのバージョンのソースからプログラムがコンパイルされたかを確認できます。

```
$ what program.c program
program.c:
      program.c 1.1 88/07/05 SMI;
program:
      program.c 1.1 88/07/05 SMI;
```

この例では、ファイルにはバージョン 1.1 の作業用のコピーが含まれています。

最新バージョンを確認する：sccs get -g

sccs get -g を使用すると、最新のデルタの SID を確認できます。

```
$ sccs get -g program.c
1.2
```

この例では、最新のデルタは 1.2 です。このバージョンは、前述の what の例でのバージョン 1.1 よりも新しいため、get を使用して最新のバージョンを取り出します。

ファイルをチェックアウトしたユーザーを確認する：sccs info

どのファイルが編集中であるかを確認するには、以下のように入力します。

```
$ sccs info
```

このサブコマンドは、編集中のすべてのファイルと、ファイルをチェックアウトしたユーザー名などの情報を表示します。同様に、以下のコマンドを使用できます。

```
$ sccs check
```

このコマンドは、編集されているファイルがある場合に、警告を出力せずにゼロ以外の終了ステータスを返します。このコマンドを makefile で使用して、ソースファイルがチェックアウトされている場合に make(1S) を強制的に停止することができます。

チェックアウトしたすべてのファイルをチェックインする場合は、以下のコマンドを使用できます。

```
$ sccs delta 'sccs tell -u'
```

tell は、編集中のファイルの名前だけを 1 行に 1 つずつ表示します。–u オプションを使用すると、tell を実行したユーザーがチェックアウトしたファイルの名前だけを出力します。–u の引数としてユーザー名を指定すると、そのユーザーがチェックアウトしたファイルの名前だけを出力します。

デルタのコメントを表示する：sccs prt

sccs prt は、SID、作成日時、各バージョンをチェックインしたユーザー名、挿入／削除／変更なしの行数、コメントが記録されているバージョンログ (デルタテーブルとも呼びます) のリストを出力します。

```
$ sccs prt program.c
D 1.2  80/08/29 12:35:31      pers  2 1 00005/00003/00084
corrected typo in widget(),
null pointer in n_crunch()

D 1.1  79/02/05 00:19:31      zeno  1 0 00087/00000/00000
date and time created 80/06/10 00:19:31 by zeno
```

最新バージョンのコメントだけを表示するには、–y オプションを使用します。

デルタのコメントを追加する：sccs cdc

コメントに大切な情報を記述し忘れたなどの場合に、以下のコマンドを使用して、コメントに情報を追加できます。

```
sccs cdc -r sid
```

このコマンドを実行するには、最新のデルタ (または分岐中で最新のデルタ、222 ページの「分岐」を参照) を指定する必要があります。また、そのデルタをチェックインしたユーザーであるか、履歴ファイルおよび SCCS サブディレクトリの両方に書き込み権を持つ所有ユーザーである必要があります。cdc は、コメントを入力するためのプロンプトを表示し、そこで入力された情報をコメントに追加します。

```
$ sccs cdc -r1.2 program.c
comments? also taught get_in() to handle control chars
```

prt を使用して新しいコメントを表示すると、以下のようになります。

```
$ sccs prt program.c
D 1.2  80/08/29 12:35:31       pers  2 1 00005/00003/00084
also taught get_in() to handle control chars
*** CHANGED *** 88/08/02 14:54:45 pers
corrected typo in widget(),
null pointer in n_crunch()

D 1.1  79/02/05 00:19:31       zeno  1 0 00087/00000/00000
```

```
date and time created 80/06/10 00:19:31 by zeno
```

チェックインしたバージョンを比較する：sccs sccsdiff

チェックインした 2 つのバージョン、たとえば 1.1 と 1.2 の 2 つのデルタを比較するには、以下のコマンドを使用して、両者の違いを確認します。

```
$ sccs sccsdiff -r1.1 -r1.2 program.c
```

全履歴を表示する：sccs get -m -p

ファイルの変更内容および変更を行なったデルタをすべて表示するには、–m と –p のオプションを使用します。

```
$ sccs get -m -p program.c
1.2
1.2 #define L_LEN 256
1.1
1.1 #include <stdio.h>
1.1
.  .  .
84
```

特定のデルタに対応する行を抽出するため、出力を grep(1V) にパイプすることができます。

```
$ sccs get -m -p program.c | grep '^1.2'
```

–p だけを使用すると、取り出したバージョンの中身を (ファイルではなく) 標準出力に送ることができます。

レポートを作成する：sccs prs -d

prs サブコマンドと –d *dataspec* オプションを使用すると、SCCS が管理するファイルに関する情報を取り出し、それをレポートとして表示することができます。*dataspec* 引数には、履歴ファイルの一部分に対応するデータキーワードのセットを指定します。データキーワードは、以下の書式で指定します。

```
:X :
```

表 5–3 にデータキーワードのリストを示しています。引数 *dataspec* では、データキーワードを任意の回数使用できます。有効な *dataspec* は、テキストおよびデータキーワードで構成される文字列を二重引用符で囲んだものです。prs は、認識した各キーワードを履歴ファイルの適切な値に置換します。

データキーワードの値の形式は、1 行または複数行のいずれかです。前者の場合は、展開された値は単純な文字列になります。後者の場合は、展開された値に復帰改行が含まれます。

タブは \t で、復帰改行 は \n でそれぞれ指定します。

以下に例を示します。

```
$ sccs prs -d"Users and/or user IDs for :F: are:\n:UN:" program.c
Users and/or user IDs for s.program.c are:
zeno
pers
$ sccs prs -d"Newest delta for :M:: :I:.  Created :D: by :P:." -r program.c
Newest delta for program.c: 1.3.  Created 88/07/22 by zeno.
```

確定した変更を削除する

デルタを置き換える：sccs fix

入力ミスなどの、修正する必要はあるけれども修正したファイルを新たなデルタとして追加する必要はない、というような小さなバグを含むデルタをチェックインしてしまうことがあります。または、ファイルの内容は正しいけれど、デルタのコメントが不完全であるあるいは誤っている場合があります。いずれの場合も、sccs fix を使用してファイルに対して変更を行い、最新のデルタに対応するバージョンログのエントリを置き換えることができます。

```
$ sccs fix -r 1.2 program.c
```

このコマンドは、program.c のバージョン 1.2 をチェックアウトします。修正後にこのファイルをチェックインすると、履歴ファイル中のデルタ 1.2 がファイルに加えた変更に置き換えられ、(新しい) コメントを入力するプロンプトが表示されます。sccs fix を使用するときには、–r オプションを使用して末端の (最新の) デルタの SID を指定する必要があります。

以前にチェックインしたデルタ 1.2 は実質的には削除されますが、SCCS はそれを削除されたデルタとして履歴ファイル中で記録します。

sccs fix を使用する前に、現在のバージョンのコピーを作成しておいてください。

デルタを削除する：sccs rmdel

最新のデルタを完全に削除するには、rmdel サブコマンドを使用します。–r を使用して SID を指定する必要があります。fix は削除されたデルタの記録を保持してい

ますが rmdel は保持しないので (詳細は、sccs rmdel(1) を参照してください)、通常は rmdel よりも fix を使用することをお勧めします。

以前のバージョンに戻す

以前のバージョンの書き込み可能なコピーを取り出すには、get -k を使用します。このコマンドは、過去の複数のデルタを調べる必要がある場合に便利です。

以前のバージョンのデルタを使用して新しいデルタを作成するには、以下の手順に従います。

1. sccs edit を使用して ファイルをチェックアウトします。
2. get –k を使用して、以前のバージョンの書き込み可能なコピーを取り出します。

```
sccs get -k -r sid -Goldname filename
```

oldname にはファイル名、*filename* には取り出した以前のバージョンを一時に保存するファイルの名前 (元のファイル名 *oldname* とは異なる名前) を指定します。

3. 現在のバージョンを以前のバージョンに置き換えます。

```
mv oldname filename
```

4. ファイルをチェックインします。

特定のデルタを削除した方が簡単な場合があります。また、SCCS を使用してファイルの複数の更新を管理する方法については、222ページの「分岐」を参照してください。

取り出したバージョンからデルタを削除する

デルタ 1.3 で行なった変更が、次のバージョンである 1.4 には適切でないとします。編集用のファイルを取り出す際に、–x オプションを使用して、作業用のコピーからデルタ 1.3 を削除することができます。

```
$ sccs edit -x1.3 program.c
```

デルタ 1.5 をチェックインする際には、そのデルタにはデルタ 1.4 での変更が含まれますが、デルタ 1.3 での変更は含まれません。さらに、削除するデルタをコンマで区切ったリストを –x に指定したり、あるいはハイフンで削除するデルタの範囲を指定して、複数のデルタを削除することができます。たとえば、1.3 と 1.4 を削除する場合は、以下のコマンドを使用できます。

```
$ sccs edit -x1.3,1.4 program.c

$ sccs edit -x1.3-1.4 program.c
```

次の例では、SCCS は 1.3 からリリース 1 において現時点で最新のデルタまでを削除します。

```
$ sccs edit -x 1.3-1 program.c
```

–x を使用する場合は、バージョン間で衝突が生じることがあります。たとえば、特定の行について挿入と削除の両方が指定されている場合があります。この場合は、SCCS は影響のある行の範囲を示すメッセージを表示します。このメッセージを十分に確認して、SCCS が取り出したバージョンが正しいかどうかを確認してください。

各デルタは、(「変更のセット」として) 削除できるため、関連する変更を各デルタごとにまとめると便利です。

バージョンを結合する：sccs comb

comb サブコマンドは、選択したデルタを結合または削除して新しい履歴ファイルを構成する Bourne シェルスクリプトを生成します。comb サブコマンドは、ディスク容量を節約したい場合に便利です。

注 - 複数のデルタを結合する際には、comb で生成したスクリプトは、ファイルのバージョンログ (コメントを含む) の一部を削除します。

–p*sid* オプションでは、スクリプト実行後の再構成で保存する最も古いデルタを指定します。

-c *sid-list*

このオプションを使用して、含めるデルタのリストを指定できます。*sid-list* には、各デルタをコンマで区切ったリストを指定します。2 つの SID をハイフン (-) で区切って、範囲を指定することができます。–p と –c を同時に使用することはできません。–o オプションは、再構成に含めるデルタの個数を最小限にします。

–s オプションは、再構成後の履歴と元の履歴のサイズ (ブロック数) を比較するスクリプトを生成します。比較結果では、再構成後の履歴が元の履歴の何パーセントであるかが表示されます。

注 - comb を使用する際は、元の履歴ファイルを保存してください。comb はディスク容量を節約するためのコマンドですが、節約されない場合もあります。場合によっては、生成された履歴ファイルが元の履歴ファイルよりも大きくなります。

オプションが指定されていない場合は、comb はそれまでの変更を保持するために必要な最小限の祖先を保存します。

バイナリファイルのバージョン管理

通常は、SCCS は ASCII テキストを含むソースファイルに対して使用しますが、このバージョンの SCCS では、バイナリファイル (NULL または制御文字を含むファイル、または最後が復帰改行でないファイル) のバージョン管理も行うことができます。バイナリファイルは、チェックインの際にテキストファイルにエンコードされます。作業用のコピーは、取り出した時にはデコードされます (詳細は、uuencode(1C) を参照してください)。

SCCS を使用して、アイコン、ラスタイメージ、画面フォントなどのファイルの変更を追跡することができます。

sccs create -b を使用すると、ファイルをバイナリファイルとして処理することができます。バイナリファイルに対して create または delta を使用すると、以下のような警告メッセージが表示されます。

```
Not a text file (ad31)
```

また、以下のメッセージも表示されます。

```
No id keywords (cm7)
```

これらのメッセージは無視して構いません。これらのメッセージ以外については、通常通り処理が行われます。

```
$ sccs create special.font

special.font:
Not a text file (ad31)
No id keywords (cm7)
1.1
20
No id keywords (cm7)
$ sccs get special.font
1.1
20
```

```
$ file special.font SCCS/s.special.font
special.font:    vfont definition
SCCS/s.special.font: sccs
```

注 - SCCS を使用してソースファイルの更新を管理し、それらのソースファイルをもとにしてオブジェクトを作成してください。

エンコードされたバイナリファイルは各バージョンごとに大きく異なるため、バイナリファイルのソースの履歴ファイルはテキストのソースの場合よりも急速に大きくなります。ただし、すべてのソースファイルで同一のバージョン管理システムを使用すると、管理がより簡単になります。

ソースのディレクトリを管理する

SCCS を使用している場合、実際のソースファイルは履歴ファイルであり、作業用のコピーではありません。

ソースディレクトリの複製

プロジェクトでの作業中に、テストまたはデバッグ用にソースの複製を作成したい場合は、自分の作業用ディレクトリに SCCS のサブディレクトリへのシンボリックリンクを作成できます。

```
$ cd /private/working/cmd.dir
$ ln -s /usr/src/cmd/SCCS SCCS
```

これにより、以下のコマンドを使用して、ソースファイルの作業用のコピーを複製して取り出すことができます。

```
sccs get SCCS
```

複製したディレクトリでの作業中にも、元のディレクトリでの場合と同様にファイルのチェックインまたはチェックアウトを行うことができます。

SCCS と make

多くの場合、SCCS を make(1s) とともに使用して、ソフトウェアプロジェクトを管理します。make を使用すると、ソースファイルの自動取り出しを行うことができ

ます (他のバージョンの make では、自動取り出しを行うために特別な規則を使用します)。また、すべてのソースファイルの以前のバージョンを取り出し、make を使用してプロジェクトの以前のバージョンを再構築することもできます。

```
$ mkdir old.release ; cd old.release
$ ln -s ../SCCS SCCS
$ sccs get -c"87/10/01" SCCS
SCCS/s.Makefile:
1.3
47
.  .  .
$ make
.  .  .
```

構築処理中にはソースファイルをチェックインしないでください。プロジェクトの完成が近くなったら、すべてのファイルをチェックインして構築するようにしてください。これによって、公開される完成プロジェクトのソースが安定します。

ファイルの SID の整合性を保つ

複数のファイルで構成されるソース間で SID の整合性を保つことができます。SID の整合性を維持するには、すべてのファイルをまとめて sccs edit します。変更は、必要に応じて任意のファイルに行うことができます。そしてすべてのファイル (変更されていないものも含む) をチェックインしてください。edit および delta の実行時に引数として SCCS のサブディレクトリを指定すると、すべてのファイルを簡単にチェックアウトおよびチェックインすることができます。

```
$ sccs edit SCCS
.  .  .
$ sccs delta SCCS
```

SCCS サブディレクトリを引数として delta サブコマンドを実行すると、コメントの入力を 1 回だけ要求されます。コメントは、チェックインされるすべてのファイルに適用されます。変更されたファイルを特定するには、各ファイルのバージョンログ (デルタテーブル) の "lines added, deleted, unchanged" のフィールドを比較します。

新しいリリースを作成する

プログラムの新しいリリースを作成するには、編集用のファイルをチェックアウトする際に、edit で *–rn* オプションを使用します。*n* に作成するリリース番号を指定します。

```
$ sccs edit -r 2 program.c
```

この場合は、新しいバージョンのデルタを使用するときに、そのデルタがリリース 2 での最初のデルタになり、SID が 2.1 となります。ディレクトリにあるすべての SCCS のファイルのリリース番号を変更するには、以下のコマンドを使用します。

```
$ sccs edit -r 2 SCCS
```

SCCS が使用する一時ファイル

SCCS が s. ファイル (履歴ファイル) を更新する際には、x. ファイルという一時コピーに書き込みます。更新が終了すると、SCCS は x. ファイルの内容を古い s. ファイルに上書きします。これにより、処理が異常終了した場合でも履歴ファイルは失われません。x. ファイルは、履歴ファイルと同一のディレクトリに作成され、履歴ファイルと同じアクセス権が与えられ、実効ユーザーによって所有されます。

SCCS ファイルが同時に更新されることを防止するために、履歴ファイルを更新するサブコマンドは、z. ファイルというロックファイルを作成します。z. ファイルには、更新を実行するプロセスの PID が含まれています。更新が終了すると、z. ファイルは削除されます。z. ファイルは、SCCS ファイルが含まれているディレクトリ中にモード 444 (読み取り専用) で作成され、実効ユーザーによって所有されます。

分岐

SCCS ファイルに適用されるデルタは、ツリーのノード (節) とみなすことができます。このツリーのルート (根) は、ファイルの最初のバージョンです。ルートのデルタ (ノード) の番号は、デフォルトでは「1.1」です。それ以降のデルタ (ノード) は、1.2、1.3 と順に番号が付きます。前述のように、SID はリリース番号とレベル番号を示します。後続のデルタでは、レベル番号が 1 ずつ増えていきます。また、ファイルを大幅に変更する際に、新しいリリースをチェックアウトする方法についてもすでに説明しました。前のリリースで新しいレベルを指定した場合を除き、新しいリリース番号は、後続のすべてのデルタにも適用されます。

したがって、ファイルは下図のように推移します。

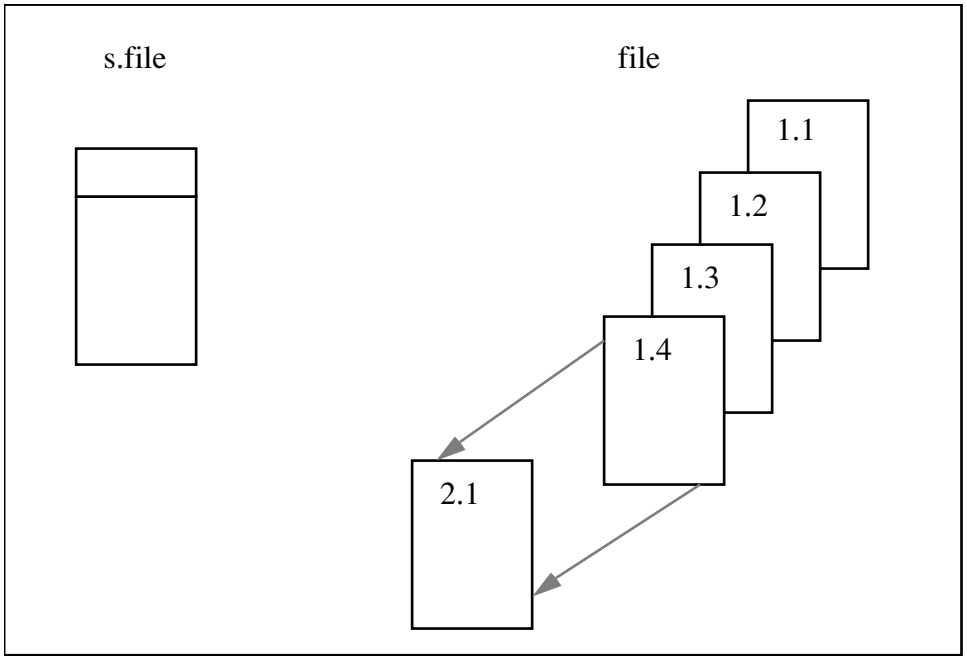


図 5–1　SCCS ファイルの推移

この構造は、SCCS デルタツリーの幹と呼びます。幹は、SCCS ファイルの通常の進展経過を表しています。あるデルタの変更は、それ以前のすべてのデルタに依存しています。

ただし、ツリーに分岐を作成すると便利な場合があります。たとえば、バージョン 1.3 が製品化されていて、リリース 2 の開発がすでに進行しているプログラムがあるとします。この場合は、リリース 2 のデルタがすでに存在している可能性があります。

ユーザー (顧客) から、バージョン 1.3 での問題点が報告され、リリース 2 の公開までその修正を延期できないとします。問題を修正するために必要な変更は、バージョン 1.3 のデルタに対して行う必要があります。このため、リリース 2 での作業とは別に、新しいバージョンを作成する必要があります。こうして、新しいデルタによってツリー上に新しい分岐が形成されます。

分岐デルタの SID は、次のようにリリース番号、レベル番号、分岐番号、シーケンス番号の 4 つの部分で構成されます。

```
release.level.branch.sequence
```

分岐番号は、特定の幹のデルタの子孫である (幹のデルタから派生した) 分岐それぞれに対して、最初の分岐は 1、次は 2 というように順に割り当てられま

す。シーケンス番号は、特定の分岐に属する各デルタに順次割り当てられます。したがって、下図のように、1.3.1.1 は、デルタ 1.3 から派生した最初の分岐の最初のデルタを示します。

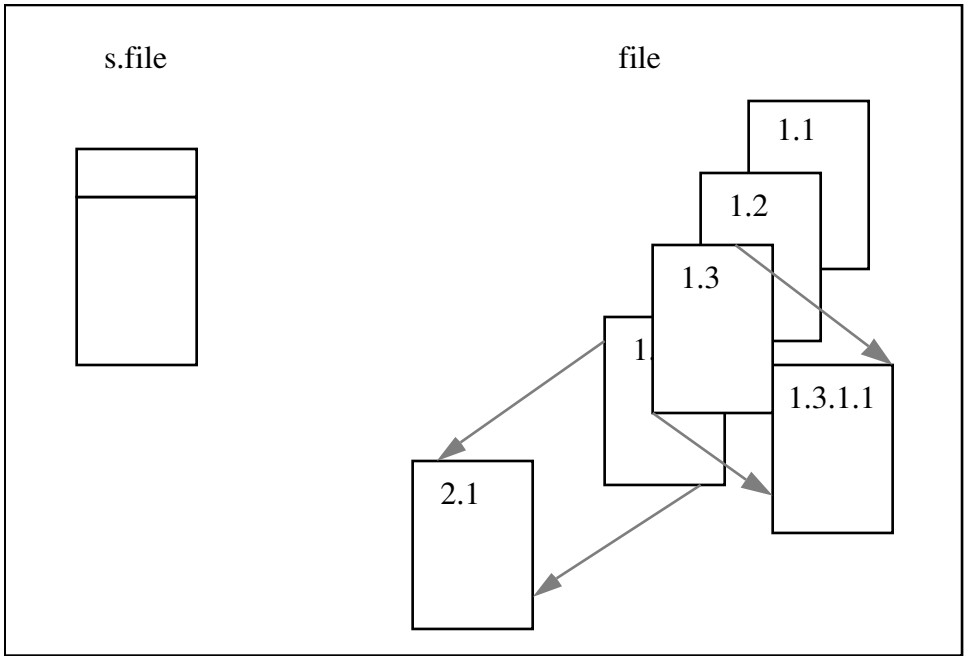


図 5–2　分岐デルタのツリー構造

分岐の概念は、ツリー中のその他のデルタにも適用されます。生成されたデルタの名前は、前述のように決定されます。分岐したデルタでは、リリース番号とレベル番号は、幹にある祖先のデルタと常に同一になります。

分岐番号は、幹との相対的な位置とは無関係に、分岐が作成された順で割り当てられます。したがって、分岐デルタは常に名前から識別できます。幹のデルタを分岐デルタの名前から識別できますが、幹のデルタから分岐デルタを特定することはできません。

たとえば、デルタ 1.3 から派生する分岐が 1 つある場合は、その分岐のすべてのデルタは 1.3.1.*n* という名前になります。この分岐のデルタからさらに派生する分岐が 1 つある場合は、新しい分岐のすべてのデルタは 1.3.2.*n* という名前になります。

デルタの名前が 1.3.2.2 の場合は、そのデルタが 2 番目に作成された分岐上の 2 番目のデルタで、その幹は デルタ 1.3 ということだけを特定できます。

特に、デルタ 1.3.2.2 の名前から、そのデルタと幹にある祖先のデルタ (1.3) の間にあるデルタをすべて特定することはできません。

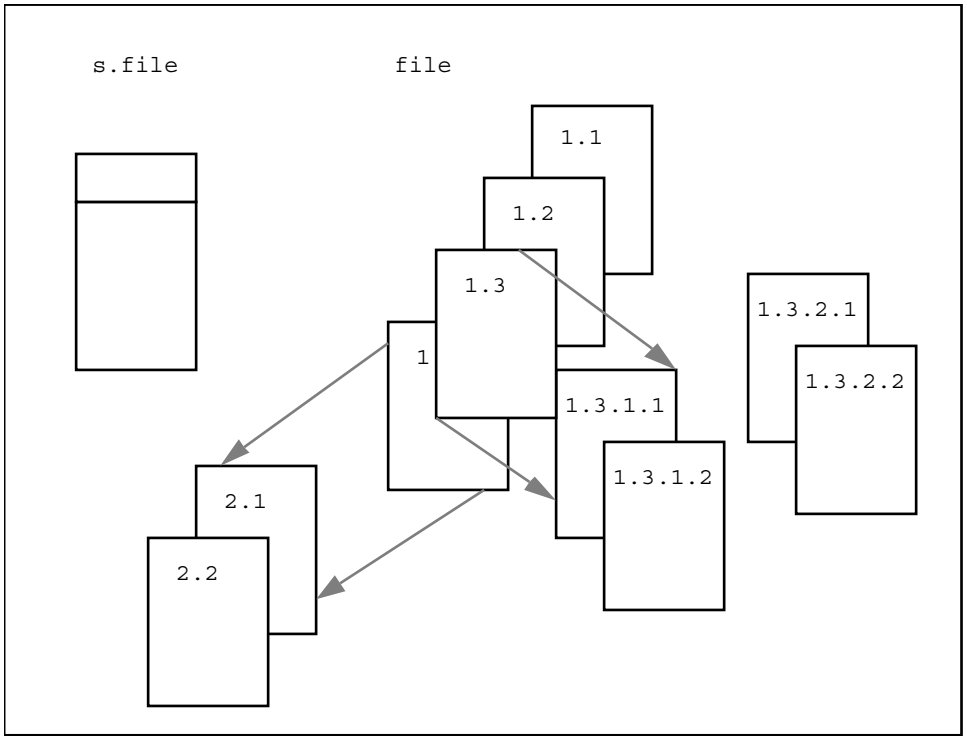


図 5–3　分岐の概念を拡張する

分岐デルタを使用してさらにツリー構造を形成できますが構造が複雑になるので、分岐はできるだけ少なくすることをお勧めします。

分岐を使用する

並行して開発している、バグの修正やテストを行うための別のバージョンを追跡する必要がある場合は、分岐を使用できます。ただし分岐を作成する前に、SCCS admin コマンドを以下のように使用して履歴ファイルの b (branch) フラグを有効にする必要があります。

```
$ sccs admin -f b program.c
```

–fb オプションは、履歴ファイル中に b フラグを設定します。

分岐デルタの作成

program.c のデルタ 1.3 から分岐を作成するには、次のように sccs edit サブコマンドを使用します。

```
$ sccs edit -r 1.3 -b program.c
```

編集したバージョンをチェックインすると、分岐デルタの SID に 1.3.1.1 が含まれます。この分岐から以降に作成するデルタは、1.3.1.2 から順に SID が割り当てられます。

分岐デルタからバージョンを取り出す

通常、get を使用して取り出したバージョンには分岐デルタは含まれていません。分岐バージョン (分岐デルタを関連付けたバージョン) を取り出すには、–r オプションを使用して明示的に指定する必要があります。次の例のようにシーケンス番号を省略すると、SCCS は分岐中で最上位 (最後) のデルタを取り出します。

```
$ sccs get -r 1.3.1 program.c
1.3.1.1
87
```

分岐をメインの幹に統合する

テストの完了後に、テストした機能を製品に組み込む場合があります。しかし、製品の開発は幹のバージョンで進行したために、分岐バージョンと幹での最新バージョンとの間に互換性がない場合があります。

この問題を解決する場合は、sccs edit の –i オプションを使用して、ファイルをチェックアウトする際に含めるデルタのリストを指定すると便利です。指定したデルタでの変更が衝突する場合は警告が表示されます。衝突は、あるデルタでは削除する必要がある行が、別のデルタでは挿入する必要がある場合に発生します。衝突の解決はユーザーが行いますが、衝突している箇所は特定することができます。

SCCS ファイルの管理

履歴ファイルおよびすべての一時 SCCS ファイルは、SCCS サブディレクトリに保存されます。標準のファイル保護機構に加えて、SCCS では、特定のリリースを凍結したり、リリースへのアクセスを特定のユーザーに制限できます (詳細は、sccs admin(1) を参照してください)。SCCS 以外のユーティリティによる修

正を禁止するため、通常は履歴ファイルのアクセス権は 444 (すべてのユーザーに対して読み取り専用) に設定されます。通常、履歴ファイルは編集しないでください。

履歴ファイルへのリンクは 1 つだけにする必要があります。SCCS ユーティリティは、修正用のコピー (x. ファイル) を変更してそのコピーの名前を変更することによって、履歴ファイルを更新します。

エラーメッセージを解釈する：sccs help

help サブコマンドは、SCCS のエラーメッセージおよびユーティリティに関する情報を表示します。

help では、引数として SCCS ユーティリティ名または SCCS のエラーメッセージのコード (括弧で示されます) を指定する必要があります。ディレクトリ /usr/ccs/lib/help には、help が表示するさまざまなメッセージが含まれているファイルがあります。

履歴ファイルのデフォルト値を変更する：sccs admin

admin コマンドを使用して、多数のパラメータ、特にフラグを設定できます。フラグは、–f オプションを使用して追加できます。

たとえば、以下のコマンドは、d フラグの値を 1 に設定します。

```
$ sccs admin -f d1 program.c
```

このフラグは、以下のコマンドを使用して削除できます。

```
$ sccs admin -d d program.c
```

最も便利なフラグを以下に示します。

b	sccs edit で –b オプションを使用して、分岐を作成することができます (222ページの「分岐」を参照してください)。
d*SID*	sccs get または sccs edit で使用されるデフォルトの SID です。これがリリース番号の場合は、特定のリリースだけにバージョンが制限されます。

i	ID キーワードがファイルに含まれていない場合に、エラーを表示します。これは、ID キーワードがないバージョン、または誤って ID キーワードが展開されているバージョンをチェックインすることを防ぐのに便利です。
y	%Y% という ID キーワードがこのフラグの値に置き換えられます。
–t*file*	*file* で指定したファイルに含まれるテキストを、s. ファイルにコメントとして保存します。マニュアルや設計と実装について説明するドキュメントなどを指定します。–t オプションを使用すると、s. ファイルを他のユーザーに渡す場合に、ドキュメントも確実に渡すことができます。*file* を省略すると、コメントは省略されます。コメントを表示するには、prt -t を使用します。

sccs admin コマンドは、ファイルに対して何度でも安全に使用できます。admin を使用する際に、現在のバージョンを取り出す必要はありません。

履歴ファイルの妥当性検査

val サブコマンドを使用すると、履歴ファイルに関する検査を行うことができます。val は、以下の状態になっていないかどうかを常に検査します。

- 履歴ファイルが壊れている。
- 履歴ファイルを開いて読み取ることができない。
- ファイルが SCCS 履歴に含まれていない。

val で –r オプションを使用すると、指定した SID が存在するかどうかを調べます。

履歴ファイルの復元

他のユーザーによる編集が原因で、履歴ファイル自体が破壊されることがあります。ファイルには検査合計 (チェックサム) が含まれるため、破壊されたファイルを読み込むたびにエラーが表示されます。検査合計を修正するには、以下のコマンドを使用します。

```
$ sccs admin -z program.c
```

注 - 履歴ファイルが破壊されていると SCCS で表示されたときは、検査合計が正しくないということよりも重大な障害を示している場合があります。履歴ファイルを修正する前に、現在の変更内容を保存してください。

参照用の一覧表

表 5–1 SCCS ID キーワード

キーワード	展開結果
%Z%	@(#) (what コマンド用の検索文字列)
%M%	現在のモジュール (ファイル) 名
%I%	割り当てられている最上位の SID
%W%	%Z% %M% タブ %I% の短縮形
%G%	%I% キーワードに対応するデルタの日付
%R%	現在のリリース番号
%Y%	-t フラグの値 (sccs admin により設定)

表 5–2 SCCS ユーティリティコマンド

SCCS ユーティリティプログラム	
コマンド	マニュアルページ名
admin	sccs-admin(1)
cdc	sccs-cdc(1)
comb	sccs-comb(1)

表 5–2　SCCS ユーティリティコマンド　続く

SCCS ユーティリティプログラム	
コマンド	マニュアルページ名
delta	sccs-delta(1)
get	sccs-get(1)
help	sccs-help(1)
prs	sccs-prs(1)
prt	sccs-prt(1)
rmdel	sccs-rmdel(1)
sact	sccs-sact(1)
sccsdiff	sccs-sccsdiff(1)
unget	sccs-unget(1)
val	sccs-val(1)
what	what(1)

表 5–3　prs –d で指定するデータキーワード

キーワード	データ項目	ファイルセクション	値	形式
:Dt:	デルタ情報	デルタテーブル	:Dt: = :DT: :I: :D: :T: :P: :DS: :DP:	1 行
:DL:	デルタ行の統計情報	デルタテーブル	:Dt: = :Li:/:Ld:/:Lu:	1 行
:Li:	delta により挿入された行	デルタテーブル	*nnnnn*	1 行
:Ld:	delta により削除された行	デルタテーブル	*nnnnn*	1 行
:Lu:	delta により変更されていない行	デルタテーブル	*nnnnn*	1 行
:DT:	デルタの種類	デルタテーブル	D または R	1 行

表 5–3 prs –d で指定するデータキーワード 続く

キーワード	データ項目	ファイルセクション	値	形式
:I:	SCCS ID 文字列 (SID)	デルタテーブル	:Rf3:.:Lf3:.:Bf3:.:S:	1 行
:R:	リリース番号	デルタテーブル	*nnnn*	1 行
:L:	レベル番号	デルタテーブル	*nnnn*	1 行
:B:	分岐番号	デルタテーブル	*nnnn*	1 行
:S:	シーケンス番号	デルタテーブル	*nnnn*	1 行
:D:	デルタが作成された日付	デルタテーブル	:Dy:/:Dm:/:Dd:	1 行
:Dy:	デルタが作成された年	デルタテーブル	*nn*	1 行
:Dm:	デルタが作成された月	デルタテーブル	*nn*	1 行
:Dd:	デルタが作成された日	デルタテーブル	*nn*	1 行
:T:	デルタが作成された時	デルタテーブル	:Th:::Tm:::Ts:	1 行
:Th:	デルタが作成された時間	デルタテーブル	*nn*	1 行
:Tm:	デルタが作成された分	デルタテーブル	*nn*	1 行
:Ts:	デルタが作成された秒	デルタテーブル	*nn*	1 行
:P:	デルタを作成したユーザー	デルタテーブル	ログ名	1 行

表 5–3 prs –d で指定するデータキーワード 続く

キーワード	データ項目	ファイルセクション	値	形式
:DS:	デルタのシーケンス番号	デルタテーブル	*nnnn*	1 行
:DP:	前のデルタのシーケンス番号	デルタテーブル	*nnnn*	1 行
:DI:	追加、削除、無視されたデルタのシーケンス番号	デルタテーブル	:Dn:/:Dx:/:Dg:	1 行
:Dn:	追加されたデルタ (シーケンス番号)	デルタテーブル	:DS: :DS: ...	1 行
:Dx:	削除されたデルタ (シーケンス番号)	デルタテーブル	:DS: :DS: ...	1 行
:Dg:	無視されたデルタ (シーケンス番号)	デルタテーブル	:DS: :DS: ...	1 行
:MR:	デルタの MR (変更要求) の数	デルタテーブル	テキスト	複数行
:C:	デルタのコメント	デルタテーブル	テキスト	複数行
:UN:	ユーザー名	ユーザー名	テキスト	複数行
:FL:	フラグのリスト	フラグ	テキスト	複数行
:Y:	モジュールの種類のフラグ	フラグ	テキスト	複数行
:MF:	MR の妥当性検査のフラグ	フラグ	yes または no	複数行
:MP:	MR の妥当性検査のプログラム名	フラグ	テキスト	1 行
:KF:	キーワードエラー / 警告フラグ	フラグ	yes または no	1 行
:BF:	分岐フラグ	フラグ	yes または no	1 行
:J:	連結編集のフラグ	フラグ	yes または no	1 行

表 5–3 prs –d で指定するデータキーワード 続く

キーワード	データ項目	ファイルセクション	値	形式
:LK:	ロックされたリリース	フラグ	:R: ...	1 行
:Q:	ユーザー定義キーワード	フラグ	テキスト	1 行
:M:	モジュール名	フラグ	テキスト	1 行
:FB:	切り下げの境界	フラグ	:R:	1 行
:CB:	切り上げの境界	フラグ	:R:	1 行
:Ds:	デフォルトの SID	フラグ	:I:	1 行
:ND:	NULL データフラグ	フラグ	yes または no	1 行
:FD:	ファイルのコメント	コメント	テキスト	複数行
:BD:	本体	本体	テキスト	複数行
:GB:	取得した本体	本体	テキスト	複数行
:W:	what(1) 文字列の形式	N/A	:Z: :M:\t:I:	1 行
:A:	what(1) 文字列の形式	N/A	:Z: :Y: :M: :I: :Z:	1 行
:A:	what(1) 文字列の区切り文字	N/A	@(#)	1 行
:F:	SCCS ファイル名	N/A	テキスト	1 行
:PN:	SCCS ファイルのパス名	N/A	テキスト	1 行

第 6 章

m4 マクロプロセッサ

概要

m4 は、C 言語とアセンブリ言語のプログラムを前処理する際に使用できる汎用マクロプロセッサです。テキストの文字列を別の文字列に置換する簡単な処理のほかに、m4 は以下の処理を行うことができます。

- 整数演算
- ファイルの取り込み
- 条件付きマクロ展開
- 文字列と部分文字列の操作

ユーザーは、組み込みマクロを使用してこれらの処理を実行したり、独自のマクロを定義できます。一部の組み込みマクロは処理の状態に副作用を及ぼしますが、それ以外は組み込みマクロとユーザー定義マクロはまったく同じように機能します。

m4 の基本動作は、英数字のトークン (文字と数字の文字列) をすべて読み取り、そのトークンがマクロの名前かどうかを判定することです。マクロの名前はそのマクロの定義テキストに置換され、その文字列は入力に戻されて再度読み取られます。マクロは、引数を使用して呼び出すことができます。引数は、収集されて、定義テキストが再度読み取られる前に適切な位置でその定義テキストに置換されます。

マクロ呼び出しの一般形式を以下に示します。

```
名前 (引数1, 引数2, ..., 引数n)
```

マクロ名の直後に左括弧がない場合は、引数がないものとみなされます。引数の収集時に引数の先頭にある引用符で囲まれていない空白文字、タブ、復帰改行は無視されます。左と右の単一引用符は、文字列を囲む際に使用します。引用符で囲まれた文字列の値は、引用符を取り除いた文字列になります。

マクロ名が認識されると、その引数は左括弧に対応する右括弧を検索することによって収集されます。与えられた引数がマクロ定義の引数の数よりも少ない場合には、足りない引数はすべて NULL であるとみなされます。引数を収集している間に、マクロ評価が行われます。また、入れ子にされた呼び出しの値に含まれているコンマや右括弧は、元の入力テキストのコンマや括弧と同様にすべて有効です。引数の収集が終わると、マクロの値は入力ストリームに返され、再度読み取られます。以下に、詳細を説明します。

以下の形式のコマンドを使用して m4 を呼び出します。

```
$ m4 file file file
```

各引数ファイルは、順番に処理されます。引数が 1 つも指定されていない場合、またはハイフンが引数として指定されている場合には、標準入力が読み取られます。最終的に m4 の出力をコンパイルする場合は、次のようなコマンドを使用してください。

```
$ m4 file1.m4 > file1.c
```

m4 コマンド行で –D オプションを使用すると、マクロを定義できます。たとえば、類似した 2 つのバージョンのプログラムがあり、その 2 つの出力ファイルを生成できる 1 つの m4 入力ファイルがあるとします。つまり、file4.m4 には以下のような行が含まれています。

```
if(VER, 1, do_something)
if(VER, 2, do_something)
```

このプログラムのメークファイルは以下のようになります。

```
file1.1.c : file1.m4
m4 -DVER=1 file1.m4 > file1.1.c
...
file1.2.c : file1.m4
m4 -DVER=2 file1.m4 > file1.2.c
...
```

–U オプションを使用すると、 VER の定義を解除することができます。file1.m4 に以下の行が含まれている場合には、

```
if(VER, 1, do_something)
if(VER, 2, do_something)
ifndef(VER, do_something)
```

そのメークファイルには以下のような行が含まれることになります。

```
file0.0.c : file1.m4
m4 -UVER file1.m4 > file1.0.c
...
file1.1.c : file1.m4
m4 -DVER=1 file1.m4 > file1.1.c
...
file1.2.c : file1.m4
m4 -DVER=2 file1.m4 > file1.2.c
...
```

m4 マクロ

マクロの定義

基本となる組み込み m4 マクロは、define() です。このマクロを使用して、新しいマクロを定義します。以下を入力すると、文字列 name が stuff として定義されます。

```
define(
name, stuff)
```

この定義以降に現われる name は、すべて stuff に置換されます。定義される文字列は、英数字にしてください。また、定義文字列の先頭は英数字 (下線は英数字とみなされます) にする必要があります。定義文字列は、1 組の括弧を含んだ任意のテキストです。この定義文字列は複数の行に渡っていても構いません。

典型的な例を以下に示します。

```
define(N, 100)
...
if (i > N)
```

この例では、N は 100 として定義されており、その定数 N が if 文で使用されています。

前述のように、define() が引数を持っていることを示すには、ワード define の直後に左括弧を置く必要があります。マクロ名の直後に左括弧が続いていない場合は、そのマクロは引数を持っていないとみなされます。したがって、上記の例の N は、引数を持たないマクロです。

マクロ名は、英数字以外の文字で囲まれている場合にだけ、そのマクロ名として認識されます。以下の例では、変数 *NNN* に N が含まれていますが、定義されたマクロ N とこの変数の間には何の関係もありません。

```
define(N, 100)
...
if (NNN > 100)
```

m4 は、できるかぎり早くにマクロ名を定義されているテキストに展開します。したがって、以下の例では、define(M, N) の引数を収集するときには N が 100 に置換されているので、M は 100 として定義されます。別の見方をすれば、N を再定義しても M の値は 100 のまま変わらないということになります。

```
define(N, 100)
define(M, N)
```

このような結果を回避する方法は 2 つあります。1 つは、定義の順序を変更して回避する方法です。この方法は、上記の例のような場合に特に有効です。

```
define(M, N)
define(N, 100)
```

これにより、M は文字列 N として定義されるので、M の値が後で要求されたときには、結果は常にその時点の N の値になります。つまり M は、N の値 100 に置換されます。

引用符で囲む

より一般的な解決方法は、define() の引数を引用符で囲んでその引数の展開を遅らせる方法です。左と右の単一引用符に囲まれたテキストは、すぐには展開されず、引数が収集されたときに引用符が取り除かれます。引用符で囲まれた文字列の値は、その引用符を取り除いた文字列です。

したがって、以下の例では、M は 100 ではなく文字列 N として定義されます。

```
define(N, 100)
define(M, 'N')
```

一般的なルールとして、m4 は評価時に 1 組の単一引用符を取り除きます。これは、マクロの外でも同じです。ワード define を出力に表示する場合には、入力ではそのワードを引用符で囲む必要があります。

```
'define' = 1;
```

通常は、マクロの引数を引用符で囲んで、マクロに定義した値が実際にそのマクロに割り当てられるようにする方法をお勧めします。たとえば、N を再定義するには、以下のように引用符を使用してその評価を遅らせます。

```
define(N, 100)
...
```

```
define(‘N’, 200)
```

引用符で囲まない場合は、2 番目の定義の N はすぐに 100 に置換されます。

```
define(N, 100)
...
define(N, 200)
```

このため結果は以下の定義と同じになります。

```
define(100, 200)
```

m4 は、名前と判断できるものしか定義できないので、この文を無視します。

左と右の単一引用符以外を引用符記号として使用したい場合には、以下のように組み込みマクロ changequote() を使用して、引用符記号を変更することができます。

```
changequote([, ])
```

この例では、マクロによって引用符記号を左と右の単一引用符から左と右の角括弧に変更します。引用符記号の最大文字長は 5 文字です。元の引用符記号に戻すには、引数のない changequote() を使用します。

```
changequote
```

undefine() は、マクロまたは組み込みマクロの定義を削除します。

```
undefine(‘N’)
```

この例では、マクロによって N の定義が削除されます。undefine() の引数は必ず引用符で囲んでください。同様に、組み込みマクロも undefine() を使用して削除できます。

```
undefine(‘define’)
```

組み込みマクロを削除したり再定義すると、そのマクロの元の定義は使用できなくなります。マクロの名前は、defn() を使用して変更することができます。たとえば、組み込みマクロの define() を XYZ() という名前に変更する場合には、以下のように指定します。

```
define(XYZ, defn(‘define’))
undefine(‘define’)
```

この指定を行うと、XYZ() は define() の元の意味を引き継ぎます。したがって、以下のマクロは A を 100 として定義します。

```
XYZ(A, 100)
```

組み込みマクロの ifdef() は、マクロが現在定義されているかどうかを判定することができます。以下のようにして、システムごとに特定のマシンに適した定義を行うことができます。

```
ifdef(‘pdp11’, ‘define(wordsize,16)’)
ifdef(‘u3b’, ‘define(wordsize,32)’)
```

ifdef() マクロには、3 つの引数を指定できます。最初の引数が定義されている場合は、ifdef() の値は 2 番目の引数になります。最初の引数が定義されていない場合は、ifdef() の値は 3 番目の引数になります。

```
ifdef(‘unix’, on UNIX, not on UNIX)
```

3 番目の引数がない場合は、ifdef() の値は NULL になります。

引数

ここまでは、文字列を別の (固定された) 文字列に置換するという、最も単純なマクロ処理について説明してきました。マクロは、呼び出しごとに結果が異なるように定義することもできます。マクロの置換テキスト (マクロ define() の 2 番目の引数) は、そのマクロが実際に使用されるときに、$n の部分が n 番目の引数に置換されます。したがって、以下のように定義された bump() は、bump(x) の場合、x = x + 1 と同義になります。

```
define(bump, $1 = $1 + 1)
```

マクロには、任意の数の引数を指定できますが、個別にアクセスできるのは最初の 9 個 ($1 ～ $9) だけです。$0 はそのマクロ自身の名前です。すでに説明したように、指定されていない引数は、NULL 文字列に置換されます。そのため、以下のように、引数を連結するマクロを定義できます。

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

つまり、cat (x、y、z) の結果は xyz になります。引数 $4 ～ $9 は、対応する引数が指定されていないため、NULL になります。

引数の収集時に引数の先頭にある引用符で囲まれていない空白文字、タブ、復帰改行は破棄されます。それ以外の空白はそのまま残ります。したがって、以下のマクロの場合には、a は b cとして定義されます。

```
define(a, b c)
```

引数はコンマによって区切られます。括弧で囲まれたコンマは引数の区切りとはみなされません。以下の例では、2 つの引数 a と (b,c) を使用しています。このように括弧で囲むことによって、コンマや括弧を引数として指定することができます。

```
define(a, (b,c))
```

以下の例では、$(** は、その後で呼び出すマクロに指定された引数のリストに置換されます。その引数リストの中の各項目はコンマで区切られます。したがって、以下の例の結果は、1, 2 になります。

```
define(a, 1)
define(b, 2)
define(star, '$(**')
star(a, b)
```

また、以下の例では m4 は star() の引数を収集するときに a と b から引用符を取り除き、star() を評価するときに a と b を展開するので、上記の例と結果は同じになります。

```
star('a', 'b')
```

$@ は、その後で呼び出すマクロの各引数が引用符で囲まれている以外は、$(** と同義です。したがって、以下の例では at() が評価されるときに引用符が再び引数に付けられるので、結果は a,b となります。

```
define(a, 1)
define(b, 2)
define(at, '$@')
at('a', 'b')
```

$# は、その後で呼び出すマクロの引数の総数に置換されます。したがって、以下の例の結果は、3 になります。

```
define(sharp, '$#')
sharp(1, 2, 3)
```

以下の場合は、結果は 1 になります。

```
sharp()
```

また以下の場合は、結果は 0 になります。

```
sharp
```

組み込みマクロの shift() は、1 つ目の引数以外のすべての引数を返します。1 つ目の引数以外の引数は、引用符で囲まれコンマで区切られて入力に返されます。最も単純な例を以下に示します。

```
shift(1, 2, 3)
```

この例の場合は、結果は 2、3 になります。$@ の場合と同様に、引用符で引数を囲むことによって、その引数の展開を遅らせることができます。したがって以下の例の場合は、shift() が評価されるときに引用符がまた引数に付けられるので、結果は b となります。

```
define(a, 100)
define(b, 200)
shift(‘a’, ‘b’)
```

演算用の組み込みマクロ

m4 には、整数演算を行う 3 つの組み込みマクロがあります。incr() は、その数値の引数を 1 ずつ増やします。decr() は、1 ずつ減らします。したがって、プログラミングでよく使われる「N より 1 大きい」変数を定義するには、以下のように incr() を使用します。

```
define(N, 100)
define(N1, ‘incr(N)’)
```

つまり、N1 は N の現在値よりも 1 大きい値として定義されます。

演算でよく用いられるメカニズムは、eval() という組み込みマクロです。このマクロでは、整数に対して任意の演算を行うことができます。以下に、このマクロの演算子を優先度が高いものから順に示します。

```
+ - (単項)
(**(**
(** / %
+ -
== != < <= > >=
! ~
&
| ^
&&
||
```

必要に応じて、括弧を使用して演算をグループ化できます。eval() に最終的に渡される式のオペランドは、すべて数値でなければなりません。真の関係 (たとえば、1 > 0) は 1、偽の関係は 0 になります。eval() の精度は 32 ビットです。

簡単な例を以下に示します。この例では、M を 2(**(**N+1 として定義しています。

```
define(M, ‘eval(2(**(**N+1)’)
```

この場合、以下の結果は 9 になります。

```
define(N, 3)
M(2)
```

ファイルの取り込み

組み込みマクロ include() を使用すると、新しいファイルを任意の時点に入力に取り込むことができます。

include(ファイル名)

このマクロは、指定したファイルの内容をマクロとその引数の位置に挿入します。include() の値 (include() の置換テキスト) は、ファイルの内容です。必要に応じて、ファイルの内容を定義などに取り込むことができます。

include() で指定されたファイルにアクセスできない場合には、重大なエラーが発生します。重大なエラーとしたくない場合には、 sinclude() (silent include) を使用することができます。指定されたファイルにアクセスできない場合でも、この組み込みマクロはメッセージを出力せずにそのまま処理を続けます。

出力の分割

処理中に、m4 の出力を一時ファイルにすることができます。収集されたデータはコマンドで出力できます。m4 は、出力を 1 番から 9 番の 9 つに分割することができます。組み込みマクロ divert(n) を使用した場合には、後続の出力はすべて n で参照される 1 つの一時ファイルに追加されます。このファイルへの出力を停止するには、divert() マクロまたは divert(0) マクロを使用します。これによって、通常の出力プロセスが再開されます。

分割された出力テキストは、通常は処理の終わりに番号順に配置されます。新しい分割出力を現在の出力の後に追加することによって、いつでも分割出力を元に戻すことができます。0 ～ 9 以外の番号で分割された出力は、破棄されます。組み込みマクロ undivert() は、すべての分割出力を番号順に元に戻します。undivert()に引数を指定すると、指定された分割出力を指定された順に元に戻します。引数なしで undivert() を使用すると、(divert() の場合と同様に) 分割された出力テキストは 0 ～ 9 以外の番号が付けられ破棄されます。

undivert() の値は、分割された出力テキストではありません。また、分割出力されたマクロのデータは、再度読み取られません。組み込みマクロ divnum() は、現在アクティブな分割出力の番号を返します。通常の出力方法で処理を行なっている場合、divnum() の値は 0 です。

システムコマンド

組み込みマクロ syscmd() を使用して、任意のプログラムを実行できます。以下に示した例では、オペレーティングシステムの date コマンドが呼び出されています。通常、後に続く include() 用のファイルを作成する際にこの syscmd() を使用します。

```
syscmd(date)
```

個々のファイルに容易に一意の名前を付けるために、組み込みマクロ maketemp() は、引数に含まれている文字列 *XXXXX* を現在のプロセスのプロセス ID に置換します。

条件付きテスト

任意の条件付きテストを、組み込みマクロ ifelse() を使用して実行することができます。このマクロの簡単な例を以下に示します。

```
ifelse(a,b,c,d)
```

この例では、文字列 a と b が比較されます。a と b が等しければ ifelse() は文字列 c を返します。等しくなければ d を返します。たとえば、2 つの文字列を比較して、等しい場合は yes、等しくない場合は no を返す compare() というマクロを定義するには、以下のようにします。

```
define(compare, `ifelse($1, $2, yes, no)')
```

ifelse() の評価を遅らせるために、引用符を使用している点に注目してください。最後の引数が省略されている場合は、結果は NULL になります。たとえば以下に示した例では、a と b が等しい場合は c、等しくない場合は NULL という結果になります。

```
ifelse(a,b,c)
```

ifelse() は、実際には任意の数の引数を持つことができ、単純な条件を判定して処理を分岐する機能も備えています。たとえば以下の例では、文字列 a と 文字列 b が等しければ結果は c になります。等しくない場合は d と e を比較して、等しければ結果は f になります。d と e が等しくない場合は、結果は g になります。

```
ifelse(a,b,c,d,e,f,g)
```

文字列の操作

len() マクロは、引数に含まれている文字列の長さ (文字数) を返します。たとえば、

```
len(abcdef)
```

は 6 を返し、

```
len((a,b))
```

は 5 を返します。

substr() マクロは、文字列の部分文字列を生成します。たとえば以下のマクロは、文字列 s の i 番目の位置 (起点は 0) から始まる n 文字の部分文字列を返します。

```
substr(s, i, n)
```

n が省略されている場合には、i 文字目以降の文字列が返されます。たとえば、以下のように入力すると、

```
substr(‘now is the time’,1)
```

以下の文字列が返されます。

```
ow is the time
```

i または n が範囲を超えている場合は、結果は不定です。

index(s1,s2) マクロは、文字列 s1 の中で文字列 s2 が現われる場所のインデックス (位置) を返します。文字列 s2 が含まれていない場合は -1 が返されます。substr() の場合と同様に、文字列の起点は 0 です。

translit() は、文字置換を行います。このマクロの一般形式を以下に示します。

```
translit(s,f,t)
```

このマクロは、f に指定された文字を t に指定された対応する文字に置換して s を変更します。

たとえば、以下の入力を使用すると、

```
translit(s, aeiou, 12345)
```

母音字はそれに対応する数字に置換されます。t が f より短い場合は、t に指定されていない文字は削除されます。t に何も指定されていない場合は、f の中の文字が s から削除されます。

したがって、以下の場合には、*s* から母音字 a,e,i,o,u がすべて削除されます。

```
translit(s, aeiou)
```

マクロ dnl() は、dnl() の後から復帰改行までの文字 (復帰改行も含む) をすべて削除します。このマクロは主に、m4 の出力に必要ない空の行を削除するために使用されます。たとえば、以下を入力すると、定義に含まれない行の終わりは復帰改行になります。

```
define(N, 100)
define(M, 200)
define(L, 300)
```

したがって、復帰改行を必要としない出力に復帰改行がコピーされます。これらの各行に dnl() を追加すると、復帰改行はなくなります。また、以下のように記述した場合も、同じ結果を得ることができます。

```
divert(-1)
define(...)
...
divert
```

出力

組み込みマクロ errprint() は、その引数を標準エラーファイルに書き込みます。以下にその例を示します。

```
errprint(‘fatal error’)
```

また、dumpdef() は、引数として指定した項目の現在の名前と定義を出力する、デバッグ支援用のマクロです。引数が指定されていない場合は、現在の名前と定義がすべて出力されます。

組み込み m4 マクロの一覧

表 6–1 組み込み m4 マクロの一覧

組み込み **m4** マクロ	説明
changequote(L, R)	左の引用符を L、右の引用符を R に変更します。
changecom	左と右のコメントマーカーをデフォルトの # と復帰改行から変更します。
decr	1 ずつ減分された引数の値を返します。

表 6–1 組み込み m4 マクロの一覧 続く

組み込み **m4** マクロ	説明
`define(name, stuff)`	`name` を `stuff` として定義します。
`defn('name')`	引用符で囲まれた引数の定義を返します。
`divert(number)`	出力を `number` 個に分割します。
`divnum`	現在アクティブな分割出力の番号を返します。
`dnl`	復帰改行までを削除します (復帰改行も含む)。
`dumpdef('name', 'name', . . .)`	指定された定義を出力します。
`errprint(s, s, . . .)`	引数を標準エラーに書き込みます。
`eval(numeric expression)`	数式 `numeric expression` を評価します。
`ifdef('name', true string, false string)`	`name` が定義されている場合は `true string` を返し、定義されていない場合は `false string` を返します。
`ifelse(a, b, c, d)`	`a` と `b` が等しい場合は `c` を返し、等しくない場合は `d` を返します。
`include(file)`	ファイルの内容を取り込みます。
`incr(number)`	number を 1 ずつ増分します。
`index(s1, s2)`	`s2` が現われる `s1` 中の位置を返します。`s2` が含まれていない場合は -1 を返します。
`len(string)`	文字列 `string` の長さを返します。
`maketemp(. . .XXXXX. . .)`	一時ファイルを作成します。
`m4 exit`	`m4` を直ちに終了します。
`m4 wrap`	最後の `EOF` で、引数 1 が入力ストリームに返されます。
`popdef`	引数の現在の定義を削除します。

表 6–1 組み込み m4 マクロの一覧 続く

組み込み **m4** マクロ	説明
pushdef	以前の定義をすべて保存します (define() の機能と同等)。
shift	1 つ目の引数以外のすべての引数を返します。
sinclude(file)	ファイルの内容を取り込みます。ファイルが見つからない場合は、無視して継続します。
substr(string, position, number)	文字列 string の position 番目の位置から始まる文字数が number の部分文字列を返します。
syscmd(command)	システムのコマンドを実行します。
sysval	syscmd() への最後の呼び出しのコードを返します。
traceoff	広域的にも、指定されたマクロに対しても、トレースをオフにします。
traceon	すべてのマクロに対してトレースをオンにします。また、引数を使用して、指定したマクロに対するトレースだけをオンにします。
translit(string, from, to)	from で指定された文字を to で指定された文字に、文字列 string 中の文字を置換します。
undefine('name')	定義のリストから名前を削除します。
undivert(number, number, ...)	number 番目の分割出力を現在の分割出力の後に追加します。

付録 **A**

System V の make

注 - /usr/ccs/lib/svr4.make は、旧バージョン System V make を使用するメークファイルのユーザー用に提供されています。ただし、できるだけデフォルトのバージョンの make (/usr/ccs/bin/make) を使用してください。また、第 4 章も参照してください。

このバージョンの make を使用するには、以下のようにシステム変数 USE_SVR4_MAKE を設定します。

```
$ USE_SVR4_MAKE='''' ; export USE_SVR4_MAKE  (Bourne シェルの場合)
% setenv USE_SVR4_MAKE  (C シェルの場合)
```

このバージョンの make の詳細は、sysV-make(1) のマニュアルページも参照してください。

プログラムのモジュールの数が多くなると、プロジェクト管理者はその分多くのファイルを管理する必要があります。多数のファイルから最終的に実行可能な製品を生成するには、さまざまな作成手順が必要です。

make には、さまざまな方法で作成された多数のファイルで構成されるプログラムの最新バージョンを管理するための機能があります。

make を使用すると、プログラマは以下のことを考慮する必要がなくなります。

- ファイルの依存関係
- 修正されたファイルおよび他のファイルへの影響
- 新しいバージョンのプログラムを生成するための正確な手順

make は、記述ファイル中に書かれているファイルおよびファイル間の関係を作成するコマンドを追跡します。プログラムを構成するファイルのいずれかに変更があると、make コマンドは、その変更によって直接的または間接的な影響がある部分だけを再コンパイルして、最終的なプログラムを作成します。

make が行う基本的な処理は、以下のとおりです。

- 記述ファイルでターゲットを検索します。
- ターゲットが依存するすべてのファイル (ターゲットを生成するために必要なファイル) が存在していてそれが最新のものであることを確認します。
- ターゲットが修正されたときよりも後にターゲットが依存するファイルのいずれかが修正されている場合は、ターゲットを (再) 作成します。

ファイル間の依存関係とコマンドについての情報が書かれている記述ファイルは、通常 makefile、Makefile、s.makefile、または s.Makefile という名前にします。記述ファイルの名前をこれら 4 つのいずれかにしておくと、最後に make を実行した後に編集されたファイル数とは無関係に、make コマンドを実行するだけでターゲットを再生成することができます。ほとんどの場合は、記述ファイルは簡単に記述でき、頻繁に変更する必要はありません。編集されたファイルが 1 つだけの場合、ターゲットを再生成するコマンドをすべて入力せずに make コマンドを入力するだけで、記述通りに再生成が実行されます。

基本的な機能

make の基本的な処理は、ターゲットファイルが依存するすべてのファイルが存在しており、それらのファイルが最新であることを確認して、ターゲットファイルを更新することです。依存関係の修正後にターゲットファイルが修正されていない場合、ターゲットファイルが再生成されます。make は、依存関係のグラフを検索します。ファイルが最後に修正された日時によって、make が実行する処理が決まります。

make は、以下の情報を使用して処理を実行します。

- ユーザー定義の記述ファイル
- ファイルシステムから取得する、ファイル名および最後の変更時間
- 情報を補足するための組み込み規則

例として、C 言語のファイルの x.c、y.c、z.c を数学ライブラリの libm とコンパイルして読み込むことにより作成した、prog という名前のプログラムを説明します。C 言語の規約によって、C 言語のコンパイル出力は、x.o、y.o、z.o という名

前のファイルになります。x.c および y.c が defs.h というファイル中の宣言のいくつかを共有していて、z.c は共有していないとします。この場合は、x.c および y.c に次の行が含まれます。

```
#include "defs.h"
```

以下は、ファイルの関係と処理を記述しています。

```
prog : x.o y.o z.o
       cc x.o y.o z.o -lm -o prog
x.o y.o : defs.h
```

この情報が makefile という名前のファイルに保存されている場合は、次のコマンドは、x.c、y.c、z.c、defs.h のいずれかが変更された後に prog を再生成するために必要な処理を実行します。

```
$ make
```

前述の例では、最初の行は prog が 3 つの .o ファイルに依存することを示しています。2 行目は、これらのオブジェクトが最新になった後に、それらのオブジェクトを読み込んで prog を作成する方法を示しています。3 番目の行は、x.o および y.o がファイル defs.h に依存することを示しています。make は、ファイルシステムを調べて、必要な .o ファイルに対応する 3 つの .c ファイルがあり、それらの .c ソースファイルからオブジェクトを生成する際に組み込み規則を使用する (つまり cc -c コマンドを実行する) ことを特定します。

このように必要な処理を自動的に特定する make 機能がないとすると、以下のような長い記述ファイルが必要です。

```
prog : x.o y.o z.o
       cc x.o y.o z.o -lm -o prog
x.o : x.c defs.h
       cc -c x.c
y.o : y.c defs.h
       cc -c y.c
z.o : z.c
       cc -c z.c
```

最後に prog を作成した後に変更されたソースファイルまたはオブジェクトファイルがなく、すべてのファイルが最新である場合は、make コマンドはその旨を表示して停止します。ただし、defs.h ファイルが編集されている場合は、x.c および y.c (z.c を除く) が再コンパイルされ、新しい x.o および y.o と既存の z.o から prog が作成されます。y.c だけが変更されている場合は、y.c だけが再コンパイルされますが、この場合も prog をもう一度読み込む必要があります。make コマンド行でターゲット名が指定されていない場合は、記述ファイルで最初に記述されているターゲットが作成されます。指定されている場合は、そのターゲットが作成

されます。以下のコマンドは、x.c または defs.h が変更された場合に、x.o を再生成します。

```
$ make x.o
```

プログラミングの際には、ニーモニック名を付けた規則と、そのニーモニック名とは異なる名前のファイルを生成するコマンドを記述し、それらをインクルードするという方法でプログラムを作成しておくと、make のファイル生成とマクロ置換の機能を利用することができます。(マクロについての詳細は、254ページの「記述ファイルと置換」を参照してください)。たとえば、ファイルをコピーするには save エントリ、不要な中間ファイルを削除するには clean などのエントリを記述します。

前述のコマンドを実行した後にファイルが存在している場合は、ファイルの最終変更時間によって先の処理が決定されます。コマンドを実行した後にファイルが存在していない場合は、現在時間によって先の処理が決定されます。

動作が実行された時間を記録するために、長さがゼロのファイルを使用することができます。この方法は、遠隔のアーカイブおよびリストを管理する際に便利です。

依存関係の行とコマンド文字列での置換を行う際には、make のマクロが使用されます。マクロは、コマンド行の引数として定義するか、記述ファイルに記述できます。いずれの場合も、マクロ名の後に = 記号を記述し、その後にマクロが表す内容を続けて記述します。マクロは、$ という記号を名前の最初に付けて呼び出します。1 文字以上の長さのマクロ名は、括弧で囲みます。以下に、有効なマクロ呼び出しの書式を示します。

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

最後の 2 つは、同一の結果になります。

$*、$@、$?、$< という特殊なマクロは、コマンドの実行中に値を変更します (これらの 4 つのマクロについては、254ページの「記述ファイルと置換」で説明しています)。以下の例は、マクロの割り当ておよび使用方法を示しています。

```
OBJECTS = x.o y.o z.o
LIBES = -lm
prog: $(OBJECTS)
        cc $(OBJECTS) $(LIBES) -o prog
...


$ make LIBES="-ll -lm"
```

このコマンドは、lex (–ll) ライブラリと math (–lm)

ライブラリとともに 3 つのオブジェクトを読み取ります。これは、コマンド行でのマクロ定義が、記述ファイル中の定義を無効にするためです (オペレーティングシステムのコマンドでは、空白文字を含む引数は引用符で囲む必要があります)。

make の使用例として、make コマンドそのものを管理するために使用する記述ファイルを示します。make のコードは、複数の C 言語のソースファイルに対して実行され、yacc の文法を使用します。記述ファイルの内容は、以下のとおりです。

```
# make コマンドの処理を定義している記述ファイル

FILES = Makefile defs.h main.c doname.c misc.c \
        files.c dosys.c gram.y
OBJECTS = main.o doname.o misc.o files.o dosys.o gram.o
LIBES =
LINT = lint -p
CFLAGS = -O
LP = lp

make: $(OBJECTS)
        $(CC) $(CFLAGS) -o make $(OBJECTS) $(LIBES)
        @size make

$(OBJECTS): defs.h

cleanup:
        -rm *.o gram.c
        -du

install:
        make
        @size make /usr/bin/make
        cp make /usr/bin/make && rm make

lint: dosys.c doname.c files.c main.c misc.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c gram.c

        # print files that are out-of-date
        # with respect to "print" file.

print: $(FILES)
        pr $? | $(LP)
        touch print
```

make プログラムは、各コマンドを実行する前にコマンドを表示します。

ソースファイルと記述ファイルだけが含まれるディレクトリで make コマンドを実行すると、以下のように出力されます。

```
cc -O -c main.c
cc -O -c doname.c
cc -O -c misc.c
cc -O -c files.c
cc -O -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -O -c gram.c
```

```
cc -o make main.o doname.o misc.o files.o dosys.o gram.o \
 13188 + 3348 + 3044 = 19580
```

最後の行は、size make コマンドの結果表示されたものです。コマンド行そのものの出力は、記述ファイル中で記号 @ が記述されているため、表示されません。

記述ファイルと置換

記述ファイルでよく使用する要素について説明しています。

コメント

記号は、コメントの開始を示し、同一行にあるこの記号以降の文字はすべて無視されます。空白行および # で始まる行は無視されます。

継続行

コメントでない行が長すぎる場合は、\ 記号を行の最後に挿入すると、行を継続できます。行の最後の文字が \ の場合は、その \ 記号とそれ以降の空白文字およびタブが 1 つの空白に置換されます。

マクロ定義

マクロ定義は、= という記号を続けて記述する識別子です。マクロ定義の先頭に、コロン (:) またはタブを使用することはできません。= の左側に指定したマクロ名 (文字および数字の文字列この名前の後の空白文字およびタブは削除されます) には、= の右側に指定した文字列 (この文字列の前にある空白文字およびタブは削除されます) が割り当てられます。以下に、有効なマクロ定義の例を示します。

```
2 = xyz
abc = -ll -ly -lm
LIBES =
```

最後の定義は、LIBES に NULL 文字列を割り当てます。明示的に値が割り当てられていないマクロは、値が NULL 文字列になります。ただし、make 規則で明示的に定義されているマクロがあります (269ページの「内部規則」を参照してください)。

一般的な形式

記述ファイルのエントリの通常の形式は、以下のとおりです。

```
target1 [target2 ...] :[:] [dependent1 ...] [; commands]
[# ...] [ \t commands] [# ...]
 .  .  .
```

角括弧内の項目は省略可能です。ターゲットおよび依存関係は、文字、数字、ピリオド、スラッシュで構成される文字列です。* や ? などのシェルのメタキャラクタは、コマンドの評価時に展開されます。コマンドは、依存関係の行のセミコロンの後、または依存関係の行の直後に続く、タブ (前述の例では \t と表示されています) で始まる行に記述できます。コマンドは、# 以外の任意の文字 (二重引用符で囲んだ文字列では # を使用できます) で構成される文字列です。

依存関係について

依存関係の行には、単一コロンまたは二重コロンが含まれます。ターゲット名は複数の依存関係の行に記述できますが、すべての行が同一の種類 (コロンの数が同一) である必要があります。単一コロンの行では、コマンド文字列を 1 つの依存関係の行にのみ関連付けることができます。いずれかの行で記述されている依存ファイルよりもターゲットが古く、コマンド (セミコロンまたはタブの後の NULL 文字列も含む) が指定されている場合は、そのコマンドが実行されます。それ以外の場合は、デフォルトの規則を呼び出すことができます。二重コロンの行では、コマンドを複数の依存関係の行に関連付けることができます。ターゲットが特定の行で記述されている依存ファイルよりも古い場合は、対応するコマンドが実行されます。組み込み規則も実行できます。二重コロンの行は、ターゲットがアーカイブライブラリである、アーカイブ形式のファイルを更新する際に特に便利です (260ページの「アーカイブライブラリ」で例を示しています)。

実行可能なコマンド

ターゲットを作成する必要がある場合は、一連のコマンドを実行します。通常は、各コマンド行が表示され、マクロ置換の後に、コマンドごとに別々のシェル呼び出しに渡されます。サイレントモード (make コマンドの –s オプション) の場合、または記述ファイル中のコマンド行が @ 記号で始まる場合には、コマンドは表示されません。コマンドがゼロ以外のエラーコードを返してエラーを示している場合は、通常 make は停止します。make コマンドで –i フラグが指定されているか、.IGNORE というターゲット名が記述ファイルに記述されているか、記述ファイル中のコマン

ド文字列がハイフン (-) で始まる場合には、エラーは無視されます。プログラムが無意味な状態を返すことがわかっている場合は、そのプログラムを呼び出す行の前にハイフンを付けるのが適しています。各コマンド行は、コマンドごとに呼び出されたシェルに渡されるため、1 つのシェルプロセスでのみ有効なコマンド (cd および shell の制御コマンドなど) を使用する場合は注意してください。これらのコマンドの結果は、次の行が実行される前に削除されます。

コマンドを実行する前に、内部的に管理されているマクロが設定されます。$@ マクロは、現在のターゲットの完全なターゲット名に設定されます。$@ マクロは、明示的に名前が指定された依存関係に対してのみ評価されます。$? マクロは、ターゲットよりも新しい名前の文字列に設定されます。$? マクロは、メークファイルに記述されている明示的な規則の評価時に評価されます。コマンドが暗黙の規則により生成された場合は、$< マクロはその動作を発生させた関連ファイルの名前になります。$* マクロは、現在のファイルと依存ファイルの名前に共通する接頭辞です。ファイルを作成する際に、明示的に指定されたコマンドや関連する組み込み規則がない場合は、.DEFAULT という名前に対応するコマンドが実行されます。.DEFAULT という名前がない場合は、make はメッセージを表示して停止します。

また、記述ファイルでも、関連するマクロ
($(@D)、$(@F)、$(*D)、$(*F)、$(<D)、$(<F)) を使用できます。

$*、$@、$< の拡張

内部的に生成されるマクロである $*、$@、$< は、現在のターゲットおよび最新でないターゲットを示す総称として使用します。このリストには、関連マクロ
($(@D)、$(@F)、$(*D)、$(*F)、$(<D)、$(<F)) が追加されています。D は、1 文字のマクロのディレクトリ部分を示します。F は、1 文字のマクロのファイル名部分を示します。これらの追加マクロは、メークファイルの階層を構築する際に便利です。これらのマクロを使用してディレクトリ名にアクセスし、シェルの cd コマンドを使用できます。コマンド例を以下に示します。

```
cd $(<D); $(MAKE) $(<F)
```

出力の変換

マクロの値は、評価時に置換されます。通常の形式を以下に示します。ここで、角括弧で囲んだ文字列は省略可能です。

```
$(macro[:string1=[string2]])
```

置換の指定がなくマクロ名が 1 文字の場合は、括弧を省略できます。置換文字列がある場合は、マクロの値は、空白文字、タブ、復帰改行で区切った複数の語が連なったものとして扱われます。したがって、語が string1 で終わる場合は、string1 が string2 に置換されます (string2 がない場合は NULL 文字列に置換されます)。

このような置換が行われるのは、make が接尾辞を区別するためです。この機能は、アーカイブライブラリを管理する際に便利です。最新でないメンバーを蓄積して、すべての C プログラム (接尾辞が .c のファイル) を処理することができるシェルスクリプトを記述するだけで管理を行うことができます。以下の例は、アーカイブライブラリの管理用に make の実行を最適化します。

```
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)(c.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        $(AR) $(ARFLAGS) $(LIB) $?
        rm $?
```

アーカイブライブラリを定義するソースファイルの種類 (接尾辞) ごとに、上記の形式の依存関係が必要です。これによって、make が生成するさまざまな情報をより汎用的に使用することができるようになります。

再帰的なメークファイル

make では、環境変数と再帰的呼び出しの機能があります。シェルのコマンド行で $(MAKE) が指定されている場合は、その行は –n フラグが設定されている場合でも実行されます。–n フラグは、(MAKEFLAGS 変数により) すべての make の呼び出しで適用されるため、実行されるのは make コマンドそのものだけです。この機能は、階層になったメークファイルでソフトウェアのサブシステムが記述されている場合に便利です。テストの目的で make –n を実行できます。この場合は、下位で呼び出された make の出力など、実行された処理がすべて出力されます。

接尾辞および変換の規則

make は、内部の規則テーブルを使用して、ある接尾辞を持つファイルを別の接尾辞を持つファイルに変換する方法を特定します。–r フラグが make コマンド行で使用されている場合は、内部の規則テーブルは使用されません。

接尾辞のリストは、実際には .SUFFIXES という名前の依存関係リストです。make は、リストに含まれるいずれかの接尾辞を持つファイルを検索します。ファイルが見つかった場合は、make はそのファイルを別の接尾辞のファイルに変換します。変

換規則の名前は、変換前と変換後の接尾辞を連結したものになります。したがって、.r ファイルを .o ファイルに変換する規則の名前は .r.o になります。規則があり、コマンドが記述ファイルで明示的に記述されていない場合は、.r.o の規則のコマンドが使用されます。コマンドがいずれかの接尾辞の規則を使用して生成される場合は、作成されるファイル名のベース名 (ファイル名から接尾辞を除いたもの) が $* マクロに割り当てられます。また、$< マクロは、変換の原因となった依存ファイルの名前になります。

接尾辞のリストは左から右に検査されるため、リストの順序は重要です。ファイルおよび対応する規則の両方を持つ最初の名前が使用されます。新しい名前を追加する場合は、.SUFFIXES のエントリを記述ファイルに追加します。依存関係は、通常のリストに追加されます。

依存関係を指定しない .SUFFIXES 行によって、現在の接尾辞リストを削除できます。名前の順序を変更する場合は、現在のリストを削除する必要があります。

暗黙の規則

make は、接尾辞のテーブルおよび複数の変換規則を使用して、デフォルトの依存関係およびコマンドを指定します。デフォルトの接尾辞のリストを、デフォルトの順序で以下に示します。

.o	オブジェクトファイル
.c	C ソースファイル
.c~	SCCS C ソースファイル
.y	yacc C ソースの文法
.y~	SCCS yacc C ソースの文法
.l	lex C ソースの文法
.l~	SCCS lex C ソースの文法
.s	アセンブラソースファイル
.s~	SCCS アセンブラソースファイル
.sh	シェルファイル
.sh~	SCCS シェルファイル

.h	ヘッダーファイル
.h~	SCCS ヘッダーファイル
.f	FORTRAN ソースファイル
.f~	SCCS FORTRAN ソースファイル
.C	C++ ソースファイル
.C~	SCCS C++ ソースファイル
.Y	yacc C++ ソースの文法
.Y~	SCCS yacc C++ ソースの文法
.L	lex C++ ソースの文法
.L~	SCCS lex C++ ソースの文法

図 A–1 は、デフォルトの変換過程を示しています。同じ接尾辞に接続する過程が 2 とおりある場合、2 つの段階を経る過程は、中間ファイルがある場合または記述ファイル中に指定されている場合にのみ使用されます。

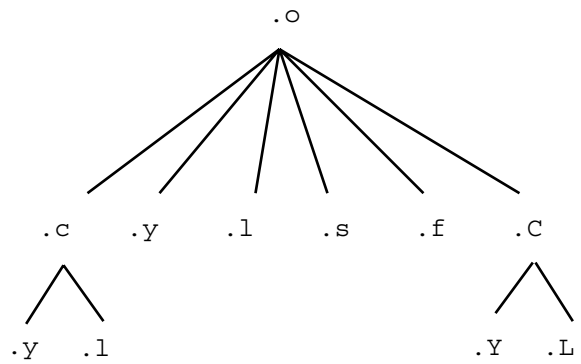


図 A–1　デフォルトの変換過程のまとめ

ファイル x.o が必要で、x.c が記述ファイル中またはディレクトリにある場合は、x.o がコンパイルされます。x.l というファイルもある場合は、そのソースファイルは lex で処理されてからコンパイルされます。ただし、x.c はないが x.l はある場合は、図 A–1 で示すように、make は C の中間ファイルを破棄して直接リンクを使用します。

マクロ名がわかっていれば、デフォルトで使用されるコンパイラ名またはコンパイラを呼び出す際に引数として使用するフラグを変更できます。コンパイラ名を示すマクロは、AS、CC、C++C、F77、YACC、LEX です。以下のコマンドは、通常の C コンパイラの代わりに newcc コマンドを使用するように指定します。

```
$ make CC=newcc
```

CFLAGS、YFLAGS、LFLAGS、ASFLAGS、FFLAGS、C++FLAGS の各マクロは、フラグを使用して実行できます。以下に例を示します。

```
$ make CFLAGS=-g
```

このコマンドは、cc コマンドがデバッグ情報を取り込むようにします。

アーカイブライブラリ

make プログラムには、アーカイブライブラリへのインタフェースがあります。ユーザーは、以下のようにライブラリのメンバーを指定できます。

```
projlib(object.o)
```

または

```
projlib((entry_pt))
```

2 番目の方法は、実際には、ライブラリ内のオブジェクトファイルのエントリポイントを指定します (make は、ライブラリを検索し、エントリポイントを特定し、正しいオブジェクトファイル名に変換します)。

この方法を使用してアーカイブライブラリを管理するには、以下のようなメークファイルが必要です。

```
projlib:: projlib(pfile1.o)
        $(CC) -c $(CFLAGS) pfile1.c
        $(AR) $(ARFLAGS) projlib pfile1.o
        rm pfile1.o
projlib:: projlib(pfile2.o)
        $(CC) -c $(CFLAGS) pfile2.c
        $(AR) $(ARFLAGS) projlib pfile2.o
        rm pfile2.o
```

メークファイルでは、オブジェクトごとに projlib 行を記述する必要があります。これには手間がかかるうえ、誤りが発生しやすくなります。ほとんどの場合、C ファイルをライブラリに追加するコマンド文字列は各ファイルで共通していて、ファイル名だけが異なります。

make コマンドは、ライブラリを構築する規則も使用できます。使用するには、.a という接尾辞を使用します。たとえば、.c.a の規則は、C ソースファイルをコンパイルしてライブラリに追加し、.o ファイルを削除します。同様に、.y.a、.s.a、.l.a の各規則は、それぞれ yacc、アセンブラ、lex のファイルを再構築します。内部定義されたアーカイブの規則は、.c.a、.c~.a、.f.a、.f~.a、.s~.a です (チルド (~) については後述)。ユーザーは、記述ファイルで他の必要な規則を定義できます。

したがって、前述のメンバーが 2 つのライブラリは、以下のメークファイルで管理できます。

```
projlib: projlib(pfile1.o) projlib(pfile2.o)
        @echo projlib up-to-date.
```

この例では、内部の規則を使用して、前述のライブラリ管理を行います。実際の .c.a の規則は、以下のとおりです。

```
c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o
```

したがって、$@ マクロは .a ターゲット (projlib) です。$< および $* マクロは、古い C ファイルと接尾辞を除いたファイル名 (pfile1.c および pfile1) に設定されます。前述の規則での $< マクロ は、$*.c に変更できます。

make が構築を行う際の処理の詳細について説明しています。

```
projlib: projlib(pfile1.o)
        @echo projlib up-to-date
```

ライブラリ内のオブジェクトが pfile1.c よりも古くなっていて、pfile1.o ファイルは存在しないとします。

1. make projlib を実行します。
2. make projlib を使用する前に、projlib の依存関係を検査します。
3. projlib(pfile1.o) が projlib の依存関係であるので、これを生成する必要があります。
4. projlib(pfile1.o) を生成する前に、projlib(pfile1.o) の依存関係を検査します (依存関係は存在していません)。
5. 内部の規則を使用して、projlib(pfile1.o) を作成します (明示的な規則はありません)。projlib(pfile1.o) では、ターゲットの接尾辞を .a に指定する括弧が名前に含まれています。projlib のライブラリ名の最後には、明示的な

.a がありません。括弧は、.a という接尾辞を示します。この意味で、.a は make に組み込まれています。

6. projlib(pfile.o) という名前を projlib および pfile1.o に分割します。$@(projlib) および $*(pfile1) という 2 つのマクロを定義します。
7. .X.a という規則と、$*.X というファイルを検索します。.SUFFIXES リストにおいてこれらの条件を最初に満たす .X は .c であるため、規則は .c.a、ファイルは pfile1.c になります。$< を pfile1.c に設定し、規則を実行します。実際には、make は pfile1.c をコンパイルする必要があります。
8. ライブラリが更新されます。projlib: の依存関係に対応する以下のコマンドを実行します。

```
@echo projlib up-to-date
```

pfile1.o が依存関係を持つには、以下の構文が必要です。

```
projlib(pfile1.o): $(INCDIR)/stdio.h pfile1.c
```

また、この構文を使用する際には、アーカイブのメンバー名を参照するマクロも使用できます。$% マクロは、$@ が評価されるごとに評価されます。現在のアーカイブのメンバーがない場合は、$% は NULL になります。アーカイブのメンバーがある場合は、$% は括弧内の式になります。

ソースコード管理システム (SCCS) ファイル名

make の構文では、接頭辞を直接参照できません。オペレーティングシステム上のほとんどの種類のファイルにおいて、ほぼすべてのユーザーがファイルを識別するために接尾辞を使用するため、問題はありません。SCCS ファイルは例外で、完全なパス名のファイル名部分の先頭に s. が付きます。

make が接頭辞 s. に簡単にアクセスできるようにするため、~ 記号が SCCS ファイルの識別子として使用されます。したがって、.c~.o は、SCCS C ソースファイルをオブジェクトファイルに変換する規則を示します。厳密には、内部の規則は以下のようになっています。

```
$(GET) $(GFLAGS) $< $(CC) $(CFLAGS) -c $*.c rm -f $*.c
```

したがって、接尾辞に付けられた ~ は、ドットからチルド記号の直前の文字までが示す実際の接尾辞を持つ SCCS ファイル名を検索します。

以下の SCCS 接尾辞が内部定義されています。

```
.c~                     .sh~                    .C~
.y~                     .h~                     .Y~
.l~                     .f~                     .L~
.s~
```

SCCS の変換について、以下の規則が内部定義されています。

```
.c~:                    .s~.s:                  .c~:
.c~.c:                  .s~.a:                  .C~.C:
.c~.a:                  .s~.o:                  .C~.a:
.c~.o:                  .sh~:                   .C~.o:
.y~.c:                  .sh~.sh:                .Y~.C:
.y~.o:                  .h~.h:                  .Y~.o:
.y~.y:                  .f~:                    .Y~.Y:
.l~.c                   .f~.f:                  .L~.C:
.l~.o:                  .f~.a:                  .L~.o:
.l~.l:                  .f~.o:                  .L~.L:
.s,:
```

ユーザーは、このほかの規則および接尾辞を任意に定義できます。~ は、SCCS ファイル名の形式を示します。

空接尾辞

多くのプログラムは、1 つのソースファイルで構成されます。make は、このようなプログラムを空接尾辞の規則により処理します。オペレーティングシステムのプログラムである cat を管理するため、以下の規則をメークファイルに記述する必要があります。

```
$(CC) -o $@ $(CFLAGS) $(LDFLAGS) $<
```

実際には、.c: という規則は、内部定義されているため、メークファイルは必要ありません。**$make cat dd echo date** を入力するだけで (これらはすべてオペレーティングシステムの単一ファイルのプログラムです)、.c: 規則に対応する前述のシェルコマンド行により 4 つの C ソースファイルが渡されます。内部定義されている単一の接尾辞の規則は、以下のとおりです。

```
.c:                     .sh:                    .f,:
.c,:                    .sh,:                   .C:
.s:                     .f:                     .C,:
.s,:
```

ユーザーは、このほかの規則をメークファイルに追加できます。

組み込みファイル

make には、C のプリプロセッサの #include 指令と同様の機能があります。include という文字列に続いて空白文字またはタブがメークファイルの行の最初に記述されている場合は、その行の残りの部分はファイル名とみなされます。実行中の make は、そのファイルを読み取ります。ファイル名にはマクロを使用できます。インクルードされたファイルの記述子はスタックされ、最高 16 レベルまでサポートされています。

SCCS メークファイル

make は、SCCS が管理するメークファイルにアクセスできます。つまり、s.makefile または s.Makefile がある場合に make を入力すると、 make はファイルに対して get を実行し、そのファイルを読み込んで削除します。

動的な依存関係のパラメータ

動的な依存関係のパラメータは、メークファイルの依存関係の行で使用した場合にのみ有効です。$$@ は、: 記号の左側にある文字列 ($@) を示します。また、$$(@F) という形式もあり、$@ のファイル部分にアクセスできます。以下に例を示します。

```
cat: $$@.c
```

この例では、依存関係は、実行時に文字列 cat.c に変換されます。これは、ソースファイルがそれぞれ 1 つだけの実行可能ファイルを多数構築する際に便利です。たとえば、オペレーティングシステムのソフトウェアコマンドのディレクトリには、以下のようなメークファイルを作成できます。

```
CMDS = cat dd echo date cmp comm chown

$(CMDS): $$@.c
```

```
        $(CC) $(CFLAGS) $? -o $@
```

これは、すべての 単一ファイルのプログラムのサブセットです。複数のファイルから成るプログラムでは、ディレクトリを割り当てて、プログラムごとにメークファイルを作成します。特別のコンパイル手順を持つファイルでは、メークファイルに特別なエントリが必要です。

もう 1 つの便利な依存関係のパラメータとして、$$(@F) があります。 これは、$$@ のファイル名部分を示します。この場合も、実行時に評価されます。/usr/src/head ディレクトリにある makefile を使用して /usr/include ディレクトリを管理するなどの場合に便利です。 この場合は、/usr/src/head/makefile は以下のようになります。

```
INCDIR = /usr/include

INCLUDES = \
        $(INCDIR)/stdio.h \
        $(INCDIR)/pwd.h \
        $(INCDIR)/dir.h \
        $(INCDIR)/a.out.h

$(INCLUDES): $$(@F)
        cp $? $@
        chmod 0444 $@
```

このメークファイルは、/usr/src/head にある前述のファイルのいずれかが更新されるたびに、/usr/include ディレクトリを完全に管理します。

コマンドの使用法

make コマンド

make コマンドでは、引数として、マクロ定義、オプション、記述ファイル名、ターゲットファイル名を以下の形式で指定します。

```
$ make [ options ] [ macro definitions and targets ]
```

以下に、各引数の意味を説明します。

まず、すべてのマクロ定義の引数 (= 記号が組み込まれた引数) が解析され、値が割り当てられます。コマンド行のマクロは、記述ファイル中の対応する定義を無効にします。次にオプションの引数が調べられます。使用可能なオプションは、以下のとおりです。

–i	呼び出したコマンドが返すエラーコードを無視します。.IGNORE というターゲット名が記述ファイルに記述されている場合に、このモードになります。
–s	サイレントモードです。実行前にコマンド行を出力しません。.SILENT というターゲット名が記述ファイルに記述されている場合にも、このモードになります。
–r	組み込み規則を使用しません。
–n	実行を行わないモードです。コマンドは出力しますが、コマンドを実際には実行しません。@ 記号で始まる行も出力されます。
–t	通常のコマンドは実行せずに、ターゲットファイルに touch コマンドを実行します (ファイルが更新されます)。
–q	make コマンドは、ターゲットファイルが最新かどうかにより、ゼロまたはゼロ以外の状態コードを返します。
–p	すべてのマクロ定義およびターゲットの記述を出力します。
–k	障害が発生した場合に現在のエントリでの処理を中止しますが、現在のエントリに依存しない他の分岐では処理を続行します。
–e	メークファイルでのマクロの割り当てよりも環境変数を優先します。
–f	引数は、記述ファイル名とみなされます。ファイル名が - の場合は、標準入力を示します。–f に引数がない場合は、現在のディレクトリにある makefile、Makefile、s.makefile、s.Makefile のいずれかの名前のファイルが読み込まれます。記述ファイルの内容 (ある場合) は、組み込

	み規則よりも優先されます。以下の 2 つのターゲット名は、フラグと同様に評価されます。
.DEFAULT	ファイルを作成する際に明示的なコマンドまたは関連する組み込み規則がない場合、.DEFAULT という名前があればそれに対応するコマンドが実行されます。
.PRECIOUS	Quit または Interrupt が実行された場合にこのターゲットの依存関係は削除されません。

その他の引数は、作成するターゲット名とみなされ、引数は左から右の順序で処理されます。他に引数がない場合は、記述ファイルに記述されている最初の文字が . 記号でない名前が使用されます。

環境変数

環境変数は、make の実行ごとに読み取られ、マクロ定義に追加されます。これを正しく実行するためには、優先度が重要です。以下では、make と環境変数との相互作用を説明しています。MAKEFLAGS というマクロは、make が管理しています。このマクロは、入力フラグ引数を (マイナス記号を除いて) 連結した文字列として定義されます。このマクロはエクスポートされるため、make の再帰的呼び出しでアクセスできます。メークファイルでのコマンド行のフラグおよび割り当てにより、MAKEFLAGS が更新されます。したがって、make による環境変数の処理を理解するには、MAKEFLAGS マクロ (環境変数) を考慮する必要があります。

make の実行時に、make は以下の順序でマクロ定義を割り当てます。

1. MAKEFLAGS 環境変数を読み取ります。この環境変数が存在しない、または NULL の場合は、make の内部変数の *MAKEFLAGS* が NULL 文字列に設定されます。それ以外の場合は、MAKEFLAGS に含まれる各文字が入力フラグ引数とみなされて処理されます (ただし、–f、–p、–r フラグを除きます)。
2. マクロ定義の内部リストを読み取ります。
3. 環境変数を読み取ります。環境変数はマクロ定義として扱われ、(シェルにおいて) エクスポート済みとして処理されます。
4. メークファイルを読み取ります。メークファイルでの割り当ては、環境変数よりも優先されます。これは、メークファイルが読み込まれて実行されたときに処理内容がわかるようにするためです。つまり、–e フラグが使用されている場合を除き、記述ファイルでの記述通りに処理が実行されます。–e は入力フラグ引数

で、環境変数がメークファイルでのマクロの割り当てよりも優先されるように指定します。したがって、–e を使用すると、環境変数がメークファイルでの定義よりも優先されます。また、MAKEFLAGS が割り当てられている場合は、それが環境変数よりも優先されます。これは、現在のメークファイルから make をさらに呼び出す場合に便利です。

以下に、割り当ての優先度を低いものから高いものの順に示します。

1. 内部定義
2. 環境変数
3. メークファイル
4. コマンド行

–e フラグが有効な場合は、優先度は以下のようになります。

1. 内部定義
2. メークファイル
3. 環境変数
4. コマンド行

通常はこの優先度によって、パラメータを動的に定義するメークファイルを定義することができます。

推奨および警告

最も頻繁に発生する問題は、make 固有の依存関係の意味が原因となっています。ファイル x.c に以下の行が記述されているとします。

```
#include "defs.h"
```

この場合は、オブジェクトファイル x.o は、defs.h に依存しますが、ソースファイル x.c は依存しません。defs.h が変更されると、x.c は変更されませんが、x.o は再作成する必要があります。

このとき、make が行う処理を確認するには、–n オプションが非常に便利です。

```
$ make -n
```

このコマンドは、make が実行するコマンドを、実際には実行せずに出力だけを行います。ファイルの変更の影響が少ない場合 (組み込みファイルにコメントを追加する場合など) は、–t (touch) オプションを使用すると時間を節約できます。make は、

不要な再コンパイルを何度も実行する代わりに、影響のあるファイルの変更時間だけを更新します。

```
$ make -ts
```

したがって、このコマンド (警告を表示しないで touch を実行する) は、関連するファイルを最新にします。このコマンドを実行すると、make の処理が無効になり以前の関係が破棄されるため、使用する際は十分に注意してください。

内部規則

make が使用する標準の内部規則を以下に示します。

make が認識する接尾辞は、以下のとおりです。

```
.o    .c    .c~   .y    .y~   .l    .l~   .s    .s~   .sh   .sh~
.h    .h~   .f    .f~   .C    .C~   .Y    .Y~   .L    .L~
```

定義済みマクロを以下に示します。

```
AR=ar ARFLAGS=-rv AS=as ASFLAGS= BUILD=build CC=cc CFLAGS= -0 C++C=CC
C++FLAGS= -0 F77=f77 FFLAGS= -0 GET=get GFLAGS= LEX=lex LFLAGS= LD=ld
LDFLAGS= MAKE=make MAKEFLAGS= YACC=yacc YFLAGS= $=$
```

特別な規則

この節では、接尾辞が 1 つまたは 2 つの特別な make の規則について説明しています。

```
markfile.o : markfile
     A=@; echo "static char _sccsid[]=\042‘grep $$A’(#)’ \
     markfile‘\042;" > markfile.c
     $(CC) -c markfile.c
     rm -f markfile.c
```

接尾辞が 1 つの規則

以下に、接尾辞が 1 つの規則を示します。

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


.c~:
  $(GET) $(GFLAGS) $<
```

```
  $(CC) $(CFLAGS) $(LDFLAGS) -o $* $*.c
  rm -f $*.c


.s:
  $(AS) $(AFLAGS) -o $@ $<


.s~:
  $(GET) $(GFLAGS) $<
  $(AS) $(AFLAGS) -o $@ $*.s
  rm -f $*.s


.sh:
  cp $< $@; chmod 0777 $@


.sh~:
 $(GET) $(GFLAGS) $<
  cp $*.sh $*; chmod 0777 $@
  rm -f $*.sh


.f:
  $(F77) $(FFLAGS) $(LDFLAGS) -o $@ $<


.f~:
  $(GET) $(GFLAGS) $<
  $(F77) $(FFLAGS) -o $@ $(LDFLAGS) $*.f
  rm -f $*.f


.C~:
  $(GET) $(GFLAGS) $<
  $(C++C) $(C++FLAGS) -o $@ $(LDFLAGS) $*.C
  rm -f $*.C


.C:
  $(C++C) $(C++FLAGS) -o $@ $(LDFLAGS) $<
```

接尾辞が 2 つの規則

以下に、接尾辞が 2 つの規則を示します。

```
 .c~.c          .y~.y          .l~.l          .s~.s          .sh~.sh
 .h~.h          .f~.f          .C~.C          .Y~.Y          .L~.L


$(GET) $(GFLAGS) $<


.c.a:
  $(CC) -c $(CFLAGS) $<
  $(AR) $(ARFLAGS) $@ $*.o
```

```
  rm -f $*.o


c.a~:
  $(GET) $(GFLAGS) $<
  $(CC) -c $(CFLAGS) $*.c
  $(AR) $(ARFLAGS) $@ $*.o
  rm -f $*.[co]


.c.o:
  $(CC) $(CFLAGS) -c $<


.c~.o:
  $(GET) $(GFLAGS) $<
  $(CC) $(CFLAGS) -c $*.c
  rm -f $*.c


.y.c:


 $(YACC) $(YFLAGS) $<
  mv y.tab.c $@


.y~.c:


 $(GET) $(GFLAGS) $<
  $(YACC) $(YFLAGS) $*.y
  mv y.tab.c $*.c
  rm -f $*.y


.y.o:
  $(YACC) $(YFLAGS) $<
  $(CC) $(CFLAGS) -c y.tab.c
  rm -f y.tab.c
  mv y.tab.o $@


.y~.o:
  $(GET) $(GFLAGS) $<
  $(YACC) $(YFLAGS) $*.y
  $(CC) $(CFLAGS) -c y.tab.c
  rm -f y.tab.c $*.y
  mv y.tab.o $*.o


.l.c:
  $(LEX) $(LFLAGS) $<
  mv lex.yy.c $@


.l~.c:
  $(GET) $(GFLAGS) $<
  $(LEX) $(LFLAGS) $*.l
  mv lex.yy.c $@
```

```
  rm -f $*.l


.l.o:
  $(LEX) $(LFLAGS) $<
  $(CC) $(CFLAGS) -c lex.yy.c
  rm -f lex.yy.c
  mv lex.yy.o $@


.l~.o:
  $(GET) $(GFLAGS) $<
  $(LEX) $(LFLAGS) $*.l
  $(CC) $(CFLAGS) -c lex.yy.c
  rm -f lex.yy.c $*.l
  mv lex.yy.o $@


.s.a:
  $(AS) $(ASFLAGS) -o $*.o $*.s
  $(AR) $(ARFLAGS) $@ $*.o


.s~.a:
  $(GET) $(GFLAGS) $<
  $(AS) $(ASFLAGS) -o $*.o $*.s
  $(AR) $(ARFLAGS) $@ $*.o
  rm -f $*.[so]


.s.o:
  $(AS) $(ASFLAGS) -o $@ $<


.s~.o:
  $(GET) $(GFLAGS) $<
  $(AS) $(ASFLAGS) -o $*.o $*.s
  rm -f $*.s


.f.a:
  $(F77) $(FFLAGS) -c $*.f
  $(AR) $(ARFLAGS) $@ $*.o
  rm -f $*.o


.f~.a:
  $(GET) $(GFLAGS) $<
  $(F77) $(FFLAGS) -c $*.f
  $(AR) $(ARFLAGS) $@ $*.o
  rm -f $*.[fo]


.f.o:
  $(F77) $(FFLAGS) -c $*.f


.f~.o:
  $(GET) $(GFLAGS) $<
  $(F77) $(FFLAGS) -c $*.f
```

```
  rm -f $*.f


.C.a:
  $(C++C) $(C++FLAGS) -c $<
  $(AR) $(ARFLAGS) $@ $*.o
  rm -f $*.o


.C~.a:
  $(GET) $(GFLAGS) $<
  $(C++C) $(C++FLAGS) -c $*.C
  $(AR) $(ARFLAGS) $@ $*.o
  rm -f $*.[Co]


.C.o:
  $(C++C) $(C++FLAGS) -c $<


.C~.o:
  $(GET) $(GFLAGS) $<
  $(C++C) $(C++FLAGS) -c $*.C
  rm -f $*.C


.Y.C:
  $(YACC) $(YFLAGS) $<
  mv y.tab.c $@


.Y~.C:
  $(GET) $(GFLAGS) $<
  $(YACC) $(YFLAGS) $*.Y
  mv y.tab.c $*.C
  rm -f $*.Y


.Y.o
  $(YACC) $(YFLAGS) $<
  $(C++C) $(C++FLAGS) -c y.tab.c
  rm -f y.tab.c
  mv y.tab.o $@


.Y~.o:
  $(GET) $(GFLAGS) $<
  $(YACC) $(YFLAGS) $*.Y
  $(C++C) $(C++FLAGS) -c y.tab.c
  rm -f y.tab.c $*.Y
  mv y.tab.o $*.o


.L.C:
  $(LEX) $(LFLAGS) $<
  mv lex.yy.c $@


.L~.C:
  $(GET) $(GFLAGS) $<
```

```
  $(LEX) $(LFLAGS) $*.L
  mv lex.yy.c $@
  rm -f $*.L


.L.o:
  $(LEX) $(LFLAGS) $<
  $(C++C) $(C++FLAGS) -c lex.yy.c
  rm -f lex.yy.c
  mv lex.yy.o $@


.L~.o:
  $(GET) $(GFLAGS) $<
  $(LEX) $(LFLAGS) $*.L
  $(C++C) $(C++FLAGS) -c lex.yy.c
  rm -f lex.yy.c $*.L
  mv lex.yy.o $@
```

索引

H

I

L

M

N

P

R

S

T

U

V

W

X

は

ひ

ふ