

# Tru64 UNIX

---

## プログラミング・サポートツール・ガイド

Part Number: AA-RK3NB-TE

**2002 年 11 月**

オペレーティング・システム: Tru64 UNIX Version 5.1B 以降

本書では、プログラム開発のために利用することができる HP Tru64 UNIX のコマンドおよびユーティリティについて説明します。

---

© 2002 日本ヒューレット・パッカート株式会社

本書の著作権は日本ヒューレット・パッカート株式会社が保有しており、本書中の解説および図、表は日本ヒューレット・パッカートの文書による許可なしに、その全体または一部を、いかなる場合にも再版あるいは複製することを禁じます。

また、本書に記載されている事項は、予告なく変更されることがありますので、あらかじめご承知おきください。万一、本書の記述に誤りがあった場合でも、弊社は一切その責任を負いかねます。

日本ヒューレット・パッカートは、弊社または弊社の指定する会社から納入された機器以外の機器で対象ソフトウェアを使用した場合、その性能あるいは信頼性について一切責任を負いかねます。

本書で解説するソフトウェア(対象ソフトウェア)は、所定のライセンス契約が締結された場合に限り、その使用あるいは複製が許可されます。

UNIX は、米国ならびに他の国における The Open Group の商標です。このドキュメントに記載されているその他の会社名および製品名は、各社の商標または登録商標です。

原典    Programming Support Tools (AA-RH9WB-TE)  
         Copyright ©2002 Hewlett-Packard Company

---

# 目次

## まえがき

## 1 正規表現と **grep** コマンドによる情報の検索

1.1	正規表現 .....	1-1
1.1.1	基本正規表現 .....	1-2
1.1.2	拡張正規表現 .....	1-4
1.1.3	正規表現の繰り返しとの照合 .....	1-6
1.1.4	選択した文字のみの照合 .....	1-7
1.1.5	複数の正規表現の指定 .....	1-8
1.1.6	正規表現における特別な照合 .....	1-8
1.2	<b>grep</b> コマンドの使用 .....	1-10

## 2 **awk** によるパターンの照合と情報の処理

2.1	<b>awk</b> プログラムの実行 .....	2-2
2.2	<b>awk</b> における出力 .....	2-5
2.3	<b>awk</b> 変数の使用 .....	2-6
2.3.1	単変数 .....	2-7
2.3.2	フィールド変数 .....	2-7
2.3.3	配列変数 .....	2-8
2.3.4	組み込み <b>awk</b> 変数 .....	2-9
2.4	パターンとしての正規表現 .....	2-11
2.5	パターンとしての関係式および論理式 .....	2-12
2.6	パターン・レンジの使用 .....	2-13
2.7	<b>awk</b> のアクション .....	2-14
2.8	アクション内での演算子の使用 .....	2-15

2.9	アクション内での関数の使用 .....	2-17
2.10	awk プログラムでの制御構造の使用 .....	2-20
2.11	入力の処理前および処理後に実行するアクション .....	2-23
2.12	文字列の連結 .....	2-23
2.13	リダイレクションとパイプ .....	2-24
<b>3</b>	<b>sed エディタによるファイルの編集</b>	
3.1	sed エディタの概要 .....	3-1
3.2	sed エディタの実行 .....	3-2
3.3	編集行の選択 .....	3-4
3.4	sed コマンドの要約 .....	3-6
3.5	文字列の置換 .....	3-11
<b>4</b>	<b>入力言語解析プログラムおよびパーサの作成</b>	
4.1	字句解析プログラムの動作 .....	4-2
4.2	lex による字句解析プログラムの作成 .....	4-3
4.3	lex 仕様ファイル .....	4-4
4.3.1	置換文字列の定義 .....	4-5
4.3.2	規則 .....	4-6
4.3.2.1	正規表現 .....	4-7
4.3.2.2	照合規則 .....	4-8
4.3.2.2.1	文字列を照合するためのワイルドカード文字の使用 .....	4-8
4.3.2.2.2	文字列内の文字列の検索 .....	4-9
4.3.2.3	アクション .....	4-10
4.3.2.3.1	ヌル・アクション .....	4-10
4.3.2.3.2	複数の式に対する同一アクションの使用 .....	4-10
4.3.2.3.3	一致した文字列のプリント .....	4-11
4.3.2.3.4	一致した文字列の長さの検出 .....	4-11

4.3.2.3.5	入力の追加 .....	4-12
4.3.2.3.6	入力へ文字を戻す .....	4-12
4.3.3	標準の入出力ルーチンの使用または変更 .....	4-13
4.3.4	ファイルの終わりの処理 .....	4-15
4.3.5	生成されたプログラムへのコードの引き渡し .....	4-15
4.3.6	開始条件 .....	4-16
4.4	字句解析プログラムの生成 .....	4-17
4.5	yacc と lex の併用 .....	4-18
4.6	yacc によるパーサの作成 .....	4-20
4.6.1	main および yyerror の関数 .....	4-21
4.6.2	yylex 関数 .....	4-22
4.7	文法ファイル .....	4-23
4.7.1	宣言 .....	4-23
4.7.1.1	グローバル変数の定義 .....	4-25
4.7.1.2	開始記号 .....	4-25
4.7.1.3	トークン番号 .....	4-25
4.7.2	文法規則 .....	4-26
4.7.2.1	空文字列 .....	4-27
4.7.2.2	入力の終わりマーカ .....	4-27
4.7.2.3	yacc パーサのアクション .....	4-27
4.7.3	プログラム .....	4-29
4.7.4	文法ファイルの使用上のガイドライン .....	4-29
4.7.4.1	コメントの使用 .....	4-29
4.7.4.2	リテラル文字列の使用 .....	4-30
4.7.4.3	文法ファイルのフォーマットに関するガイドライン ..	4-30
4.7.4.4	文法ファイル中の再帰の使用 .....	4-31
4.7.4.5	文法ファイルのエラー .....	4-31
4.7.5	パーサによるエラー処理 .....	4-31
4.7.5.1	エラーの訂正 .....	4-32

4.7.5.2	先読みトークンの消去 .....	4-33
4.8	パーサの動作 .....	4-33
4.8.1	shift アクション .....	4-34
4.8.2	reduce アクション .....	4-35
4.8.3	あいまいな規則およびパーサの競合 .....	4-36
4.9	デバッグ・モードの使用 .....	4-38
4.10	簡易計算機プログラムの作成 .....	4-38
4.10.1	パーサのソース・コード .....	4-40
4.10.2	字句解析プログラムのソース・コード .....	4-43

## 5 プログラムにおける m4 マクロの使用

5.1	マクロの使用 .....	5-1
5.2	マクロの定義 .....	5-3
5.2.1	引用符の使用 .....	5-4
5.2.2	マクロ引数 .....	5-6
5.3	他の m4 マクロの使用 .....	5-7
5.3.1	コメント文字の変更 .....	5-10
5.3.2	引用符文字の変更 .....	5-10
5.3.3	マクロ定義の取り消し .....	5-11
5.3.4	定義されたマクロのチェック .....	5-11
5.3.5	整数演算の使用 .....	5-11
5.3.6	ファイルの操作 .....	5-12
5.3.7	出力のリダイレクト .....	5-12
5.3.8	プログラムでのシステム・プログラムの使用 .....	5-13
5.3.9	固有のファイル名の使用 .....	5-13
5.3.10	条件式の使用 .....	5-13
5.3.11	文字列操作 .....	5-14
5.3.12	表示 .....	5-15

## 6 RCS や SCCS によるソース・ファイルの管理

6.1	リビジョン制御の概要 .....	6-2
6.2	バージョン・コントロールの概念 .....	6-3
6.3	ファイルの複数バージョンの管理 .....	6-7
6.4	バージョン・コントロール・ライブラリの作成 .....	6-9
6.5	RCS の使用方法 .....	6-9
6.5.1	RCS ライブラリへの新しいファイルの配置 .....	6-12
6.5.2	RCS によるファイル識別情報の記録 .....	6-12
6.5.3	RCS ライブラリからのファイルの取り出し .....	6-14
6.5.4	編集済みファイルの RCS ライブラリへのチェックイン ..	6-15
6.5.5	複数バージョンを持つファイルの操作 .....	6-15
6.5.6	RCS ファイルの差異の表示 .....	6-17
6.5.7	RCS ファイルのリビジョン・ヒストリの報告 .....	6-17
6.5.8	構成の制御の概念 .....	6-18
6.6	SCCS の使用方法 .....	6-20
6.6.1	SCCS ライブラリへの新しいファイルの配置 .....	6-22
6.6.2	SCCS によるファイル識別情報の記録 .....	6-23
6.6.3	SCCS ライブラリからのファイルの取り出し .....	6-25
6.6.3.1	編集以外の目的でのファイルの取り出し .....	6-25
6.6.3.2	編集のためのファイルの取り出し .....	6-26
6.6.3.3	複数ファイルおよび新しいリリースの管理 .....	6-26
6.6.4	編集済みファイルの SCCS ライブラリへのチェックイン .	6-27
6.6.5	複数バージョンを持つファイルの操作 .....	6-28
6.6.6	SCCS ファイルの差異の表示 .....	6-29
6.6.7	SCCS ファイルのリビジョン・ヒストリの報告 .....	6-30
6.6.8	管理機能の実行 .....	6-31
6.6.9	SCCS オプションの使用 .....	6-33
6.6.10	独立した SCCS コマンドの一覧 .....	6-34

6.7	RCS コマンドと SCCS コマンドの機能比較 .....	6-36
<b>7</b>	<b>make ユーティリティによるプログラムのビルド</b>	
7.1	make ユーティリティの動作 .....	7-2
7.2	記述ファイル .....	7-4
7.2.1	記述ファイル・エントリの形式 .....	7-5
7.2.2	記述ファイルでのコマンドの使用 .....	7-6
7.2.3	make ユーティリティのコマンド・エコーの防止 .....	7-8
7.2.4	make ユーティリティのエラーによる停止の防止 .....	7-8
7.2.5	省略時の条件の定義 .....	7-9
7.2.6	make によるファイル削除の防止 .....	7-9
7.2.7	簡単な記述ファイル .....	7-9
7.2.8	記述ファイルの簡素化 .....	7-10
7.2.9	マクロの定義 .....	7-11
7.2.10	記述ファイルでのマクロの使用 .....	7-11
7.2.10.1	マクロの置換 .....	7-12
7.2.10.2	条件付きマクロ .....	7-15
7.2.11	記述ファイルからの make ユーティリティの呼び出し ....	7-15
7.2.12	内部マクロ .....	7-15
7.2.12.1	ターゲット・ファイル名内部マクロ .....	7-16
7.2.12.2	ラベル名内部マクロ .....	7-17
7.2.12.3	younger ファイル内部マクロ .....	7-18
7.2.12.4	最初の旧ファイル内部マクロ .....	7-18
7.2.12.5	現在のファイル名プレフィックスのための内部マクロ	7-18
7.2.13	make による環境変数の使用 .....	7-18
7.2.14	内部規則 .....	7-19
7.2.14.1	単一サフィックスの規則 .....	7-21
7.2.14.2	make の組み込みマクロの変更 .....	7-22
7.2.15	他のファイルのインクルード .....	7-24



7.2.16	記述ファイルのテスト .....	7-24
7.2.17	記述ファイル .....	7-24

## 用語集

## 索引

## 例

4-1	計算機のパーサ・ソース・コード .....	4-40
4-2	計算機の子句解析プログラム・ソース・コード .....	4-43
7-1	記述ファイルの例 .....	7-9
7-2	省略時の規則ファイル .....	7-21
7-3	make ユーティリティの makefile .....	7-25

## 図

2-1	awk 処理の順序 .....	2-5
3-1	sed 処理の順序 .....	3-4
4-1	簡単な有限状態のモデル .....	4-3
4-2	lex および yacc による入力パーサの生成 .....	4-19
6-1	バージョン・コントロール・ファイルの内容 .....	6-4
6-2	典型的な RCS ライブラリ .....	6-5
6-3	典型的な SCCS ライブラリ .....	6-7
6-4	バージョン・コントロール・ファイルのツリー構造 .....	6-8

## 表

1-1	基本正規表現の規則 .....	1-3
1-2	拡張正規表現の規則 .....	1-4
1-3	grep コマンドの動作 .....	1-10
1-4	grep コマンドのオプション .....	1-11
2-1	awk コマンドのフラグ .....	2-2

2-2	awk の組み込み変数 .....	2-10
2-3	awk アクションの演算子 .....	2-15
2-4	組み込み awk 算術関数 .....	2-17
2-5	組み込み awk 文字列関数 .....	2-18
2-6	その他の組み込み awk 関数 .....	2-19
2-7	awk の制御構造 .....	2-21
3-1	sed コマンドのフラグ .....	3-2
3-2	sed で認識される正規表現 .....	3-5
3-3	テキスト編集および移動のコマンド .....	3-7
3-4	バッファ操作用コマンド .....	3-9
3-5	フロー制御のコマンド .....	3-10
4-1	lex の正規表現演算子 .....	4-7
4-2	lex コマンドのオプション .....	4-17
4-3	yacc の処理条件の定義キーワード .....	4-24
5-1	組み込みの m4 マクロ .....	5-8
6-1	RCS と SCCS の機能 .....	6-3
6-2	RCS コマンド機能の要約 .....	6-10
6-3	RCS ID キーワード .....	6-13
6-4	sccs コマンド機能の要約 .....	6-21
6-5	SCCS ID キーワード .....	6-24
6-6	SCCS admin コマンドのオプション .....	6-31
6-7	admin コマンドのフラグ .....	6-32
6-8	SCCS コマンド・オプション .....	6-34
6-9	独立した SCCS コマンド .....	6-34
6-10	機能比較: RCS と SCCS コマンド .....	6-36
7-1	make の内部マクロ .....	7-16

---

## まえがき

本書では、システムで以下のような処理を行うための HP Tru64 UNIX のコマンドおよびユーティリティについて説明します。

- テキストおよび文字列の処理
- マクロおよびプログラムの作成
- ソース・ファイルの管理

### 本書の対象読者

本書は主としてプログラマを対象としたものですが、`grep` (第 1 章)、`awk` (第 2 章)、`sed` (第 3 章)、および `RCS` と `SCCS` (第 6 章) で説明するコマンドとユーティリティは、プログラミングだけでなく、文書作成やほかの作業のために使用しても便利です。

### 新しい機能と変更された機能

バージョン 5.0 リリースからの変更点は、以下のとおりです。

- 第 2 章 が変更され、`awk` での照合パターンと処理の指定方法が明確になりました。
  - 2.3.3 項 では、配列要素を保守するために `awk` で使用するハッシュ・テーブルについて、Tru64 UNIX でのサポートを説明しています。
  - 2.3.4 項 の表 2-2 では、例を新しくしています。
  - 2.4 節 では、`awk` での正規表現の使い方を明確にしました。
  - 2.8 節 の表 2-3 では、漏れていた演算子を追加し、また優先順位を追加しました。
  - 2.9 節 の表 2-4 では、例を明確にしました。
  - 2.9 節 の表 2-5 では、正規表現を明確にしました。
  - 2.9 節 の表 2-6 では、`delete` 関数を 2.10 節 の表 2-7 に移動し、例を追加しました。

- 2.10 節の表 2-7 では、else if 文と if 文の使い方を明確にしました。
- 2.11 節では、awk の END パターンの使い方を明確にしました。
- 2.13 節では、リダイレクションとパイプの例を追加しました。
- 第 6 章の 6.5.2 項では、例を修正しました。

本書の以前のバージョンは、次の Web サイトにあります。

<http://tru64unix.compaq.co.jp/document/index.html>

本書の改訂状況を知りたい場合は、各バージョンの「新しい機能と変更された機能」の項を参照してください。

## 本書の構成

本書の構成は以下のとおりです。

- |       |   |
|-------|---|
| 第 1 章 | 正規表現の概念と規則を紹介し、正規表現を使用してテキスト・ファイルを探索する <code>grep</code> コマンドについて説明します。                                     |
| 第 2 章 | <code>awk</code> コマンドの概要と <code>awk</code> コマンドで処理するテキスト処理プログラムの作成方法について説明します。                              |
| 第 3 章 | <code>sed</code> ストリーム・エディタについて説明します。このツールは、複雑で、繰り返しの多い編集作業を高速に実行するための非対話型ツールです。                            |
| 第 4 章 | <code>lex</code> プログラムと <code>yacc</code> プログラムについて説明します。これらのプログラムは、プログラムへの入力を処理するための文法解析プログラムおよびパーサを作成します。 |
| 第 5 章 | <code>m4</code> マクロ・プリプロセッサについて説明します。また、プログラムや、ドキュメント・ソースなど、その他のファイルにも使用可能なマクロの作成方法についても説明します。              |
| 第 6 章 | ソース・コード・コントロール・システム (SCCS) またはリビジョン・コントロール・システム (RCS) を使用した、ソース・ファイル・ライブラリの管理方法について説明します。                   |
| 第 7 章 | 複雑なプログラムおよびアプリケーションの作成および保守を行う、 <code>make</code> ユーティリティの使用方法について説明します。                                    |

## 関連資料

本書は、『プログラミング・ガイド』を補足するものです。いずれのマニュアルの場合も、その内容を使用するにあたって、他方のマニュアルを参照する必要はありません。

Tru64 UNIX のドキュメントは次の URL でアクセスできます:

<http://tru64unix.compaq.co.jp/document/index.html>

## 本書で使用する表記法

本書では、次の表記規約を使用します。

%

\$

パーセント記号は、C シェルのシステム・プロンプトを表します。ドル記号は、Bourne シェル、Korn シェル、および POSIX シェルの場合のシステム・プロンプトを表します。

#

番号記号は root としてログインした場合のシステム・プロンプトを表します。

% cat

対話式の例における太字(ボールド体)は、ユーザが入力する文字を示します。

*file*

イタリック体(斜体)は、変数値、プレースホルダ、および関数の引数名を示します。

[ | ]

{ | }

構文定義では、大カッコはオプションの項目を示し、中カッコは必須項目を示します。大カッコまたは中カッコの中の項目を縦線で区切っている場合は、そこに併記されている項目の中から1つの項目を選択することを示します。

...

構文定義では、水平の反復記号は、前の項目を1回以上繰り返して使用できることを示します。

cat(1)

リファレンス・ページの参照には、該当するセクション番号をカッコ内に示します。たとえば、cat(1) は、cat コマンドについての情報が、リファレンス・ページのセクション1に記載されていることを示します。

Return

四角で囲まれたキー名はユーザがそのキーを押すことを示します。

Ctrl/x

この記号は、スラッシュの前に指定されているキーを押しながら、スラッシュの後のキーまたはマウス・ボタンを押すことを示します。例中では、このようなキーの組み合わせは、四角あるいは大カッコで囲まれて示されます(たとえば、Ctrl/C)。

---

## 正規表現と `grep` コマンドによる情報の検索

この章では、正規表現とその使用方法について説明します。一般に正規表現は、`grep` コマンドでパターンを照合するために使用されていますが、テキスト処理やフィルタ処理のためのその他のユーティリティやコマンドでも使用されます。`grep` コマンドについては、1.2 節で説明します。

この章は、次の事項について説明します。

- 正規表現 (1.1 節)
- `grep` コマンドの使用 (1.2 節)

### 1.1 正規表現

この節では、次の項目について説明します。

- 基本正規表現 (1.1.1 項)
- 拡張正規表現 (1.1.2 項)
- 正規表現の繰り返しとの照合 (1.1.3 項)
- 選択した文字のみの照合 (1.1.4 項)
- 複数の正規表現の指定 (1.1.5 項)
- 正規表現における特別な照合 (1.1.6 項)

正規表現は、どのような文字列を照合するかを指定します。正規表現は、テキスト文字列と演算子から構成されます。テキスト文字列には照合する文字列を、演算子には反復、選択などの機能を指定します。

正規表現は次の 2 つのカテゴリに分けられます。

- 基本正規表現
- 拡張正規表現

1.1.1 項および 1.1.2 項で、これらの各正規表現について説明します。これらの項で説明する正規表現の他に、文字クラス、照合順序、等価クラスに関連する特殊な正規表現があります。これらのクラスについての詳細は 1.1.6 項を参照してください。

これら 3 つの項で説明する正規表現演算子の優先順位は、次のとおりです。

1. 照合関連のカッコ

[*=*], [*.*], [*:*]

2. エスケープ文字

*\char*

3. カッコ式

[*expr*]

4. サブ正規表現

基本正規表現の場合: *\(expr\)*, *\n*

拡張正規表現の場合: (*expr*)

5. 繰り返し

基本正規表現の場合: \*, *\{i\}*, *\{i,\}*, *\{i,j\}*

拡張正規表現の場合: \*, ?, +, *{i}*, *{i,}*, *{i,j}*

6. 連結

7. 行頭, 行末

*^*, *\$*

8. 二者択一 (拡張正規表現の場合)

|

### 1.1.1 基本正規表現

基本正規表現は、より単純な基本正規表現を連結して構成されます。アルファベット文字は普通のテキスト文字として扱われます。テキスト文字は、常にその文字自身を照合します。数字もテキスト文字として扱われます。ただし、数字の前にバックスラッシュがある場合は反復式として処理されます。反復式については、表 1-1 を参照してください。



たとえば，正規表現 `rabbit` は文字列 `rabbit` と一致し，正規表現 `a57D` は文字列 `a57D` と一致します。

文字と演算子で単純な基本正規表現が構成されます。いくつかの単純な基本正規表現を組み合わせると，複雑な正規表現が構成されます。

表 1-1 に基本正規表現の規則を示します。

表 1-1: 基本正規表現の規則

正規表現	名称	説明
文字，数字，ほとんどの句読点	テキスト文字	その文字自身と一致します。
<code>.</code>	ピリオド (ドット)	改行文字以外の任意の 1 文字と一致します。
<code>*</code>	アスタリスク	直前の単純正規表現の 0 回以上の繰り返しと一致します。
<code>\{i,j\}</code>	カウント式	直前の単一正規表現の繰り返し個数を指定します。たとえば， <code>ab\{3\}c</code> は， <code>abbbc</code> のみと一致します。 <code>ab\{2,3\}c</code> は， <code>abbc</code> または <code>abbbc</code> と一致し， <code>abc</code> または <code>abbbbc</code> とは一致しません。
<code>\(expr\)</code>	サブ正規表現 (保存デリミタ)	<code>expr</code> に指定した複数の基本正規表現演算子を 1 つの単位として扱います。たとえば， <code>a\{2,3\}d</code> は <code>abcd</code> あるいは <code>abcbcbcd</code> と一致します。 <code>abcd</code> あるいは <code>abcbcbcbcd</code> とは一致しません。この正規表現は，後で再使用できるように，番号付き保存領域に格納されます。
<code>\n</code>	反復式	正規表現内の <code>n</code> 番目の保存デリミタの内容を繰り返します。
<code>[chars]</code>	大カッコ	大カッコ内の文字の任意の 1 文字と一致します。ハイフンを使用すれば範囲指定ができます。たとえば， <code>[0-9a-z]</code> は任意の桁の数字または小文字と一致します。大カッコ内では，ハイフンおよび山形記号を除いて，すべての文字をテキスト文字として処理します。
<code>^</code>	山形記号	正規表現の最初に使用された場合，行頭を意味します。大カッコ内で最初の文字に使用された場合，カッコ内の文字以外と一致します。それ以外の場所では，特別な意味を持ちません。

表 1-1: 基本正規表現の規則 (続き)

正規表現	名称	説明
\$	ドル記号	正規表現の最後で使用された場合、行末を意味します。それ以外の場所では、特別な意味を持ちません。
\char	バックスラッシュ	大カッコ内にある場合を除いて、後続の 1 文字をエスケープすることにより、通常は正規表現演算子として使用される文字の照合を行いません。
expr expr ...	連結	連結した正規表現と一致する文字列を照合します。

1.1.2 拡張正規表現

基本的には、拡張正規表現は 1.1.1 項で説明した基本正規表現と同じです。ただし拡張正規表現の場合、grep (-Eフラグを指定しない場合) や sed などのプログラムよりも強力なファイル操作およびフィルタ操作を行うためのプログラムである awk などの特定のプログラムで使用する、拡張された正規表現を含みます。基本正規表現と拡張正規表現のそれぞれに含まれる正規表現の多くは同じものですが、これらの 2 つのタイプの正規表現は別のものとして考えるのがよいでしょう。表 1-2 に拡張正規表現の規則を示します。

表 1-2: 拡張正規表現の規則

正規表現	名称	説明
文字, 数字, ほとんどの句読点	テキスト文字	その文字自身と一致します。
.	ピリオド (ドット)	改行文字以外の任意の 1 文字と一致します。
*	アスタリスク	直前の単純正規表現の 0 回以上の繰り返しと一致します。
?	疑問符	直前の単一正規表現と一致します。0 回の場合も含みます。
+	プラス記号	直前の単一正規表現の 1 回以上の繰り返しと一致します。

表 1-2: 拡張正規表現の規則 (続き)

正規表現	名称	説明
<code>{i,j}</code>	カウント式	直前の単一正規表現の繰り返し個数を指定します。たとえば、 <code>ab{3}c</code> は、 <code>abbbc</code> のみと一致します。 <code>ab{2,3}c</code> は、 <code>abbc</code> または <code>abbbc</code> と一致し、 <code>abc</code> または <code>abbbbc</code> とは一致しません。基本正規表現のカウント式はエスケープされたカッコで範囲指定しますが、拡張正規表現の場合エスケープは不要です。拡張正規表現を使用してカウント式形式のリテラル文字列を照合する場合は、左側のカッコをエスケープします。たとえば、正規表現 <code>\{2,3}</code> は文字列 <code>{2,3}</code> を照合します。
<code>(expr)</code>	サブ正規表現	<code>expr</code> に指定した複数の拡張正規表現演算子を 1 つの単位として扱います。たとえば、 <code>a(bc)?d</code> は <code>ad</code> あるいは <code>abcd</code> を照合します。 <code>abcbcd</code> あるいは <code>abcbcbcd</code> などは照合しません。基本正規表現の場合はエスケープされたカッコで範囲指定されますが、拡張正規表現の場合エスケープは不要です。拡張正規表現を使用して、カッコで囲まれたリテラル文字列を照合する場合は、左側のカッコをエスケープします。たとえば、正規表現 <code>\(abc)</code> は文字列 <code>(abc)</code> を照合します。
<code>[chars]</code>	大カッコ	大カッコ内の文字の任意の 1 文字と一致します。ハイフンを使用すれば範囲指定ができます。たとえば、 <code>[0-9a-z]</code> は任意の桁の数字または小文字と一致します。大カッコ内では、ハイフンおよび山形記号を除いて、すべての文字をテキスト文字として処理します。
<code>^</code>	山形記号	正規表現の最初に使用された場合、行頭を意味します。大カッコ内で最初の文字に使用された場合、カッコ内の文字以外と一致します。それ以外の場所では、特別な意味を持ちません。
<code>\$</code>	ドル記号	正規表現の最後で使用された場合、行末を意味します。それ以外の場所では、特別な意味を持ちません。
<code>\char</code>	バックスラッシュ	大カッコ内にある場合を除いて、後続の 1 文字をエスケープすることにより、通常は正規表現演算子として使用される文字の照合を行いません。

表 1-2: 拡張正規表現の規則 (続き)

正規表現	名称	説明
<code>expr expr ...</code>	連結	連結した正規表現と一致する文字列を照合します。
<code>expr expr ...</code>	縦線	連結した正規表現と一致する文字列を照合します。

1.1.3 正規表現の繰り返しとの照合

アスタリスク (\*) は、直前の正規表現に作用し、照合パターンの 0 回以上の繰り返しを照合します。ピリオドの後にアスタリスクを使用するとヌルを含む任意の文字列に照合します。ピリオドとアスタリスクは、常に最長のテキストを照合しようとします。次の例で考えてみましょう。

```
% echo "A B C D" | sed 's/^.*/E/'
ED
```

この例の sed ストリーム・エディタ・コマンドは、sed に、1 番目と 2 番目のスラッシュの間の正規表現を照合させ、照合したパターンを 2 番目と 3 番目のスラッシュの間の文字列に置換するものです。この正規表現が照合するのは、行頭から始まり、任意の文字列を含み、スペースで終了する任意の文字列です。文字列 “A ” もこの式を満たしていますが、最長の照合パターンは “A B C ” です。したがって、sed は、“A B C ” を “E” に置換し、ED を出力します。sed ストリーム・エディタについての詳細は、第 3 章を参照してください。

基本正規表現の場合も拡張正規表現の場合も、アスタリスクは直前の正規表現の任意回数の繰り返しを照合します。プラス記号 (+) または疑問符 (?) を使用すると、特定の拡張正規表現が照合する反復回数を限定することができます。プラス記号を指定した場合、照合パターンの繰り返し回数は 1 回以上でなければなりません。疑問符の場合、2 回以上の繰り返しは照合しません。

次の表は、アスタリスク、プラス記号、疑問符が照合する文字の例を示したものです。

正規表現	照合する文字列		
<code>ab?c</code>	<code>ac</code>	<code>abc</code>	
<code>ab*c</code>	<code>ac</code>	<code>abc</code>	<code>abbc, abbbc, ...</code>
<code>ab+c</code>		<code>abc</code>	<code>abbc, abbbc, ...</code>

正規表現の繰り返し回数をさらに限定して指定することもできます。次に示すのは、基本正規表現の繰り返し回数を指定するいくつかの方法です。

- `expr\{i\}`  
`expr` の繰り返し回数が `i` 回の文字列のみを照合します。たとえば、`ab\{3\}c` は `abbbc` を照合し、`abbc` や `abbbbc` は照合しません。
- `\{i,\}`  
`i` 回以上の繰り返しを照合します。たとえば、`ab\{3,\}c` は、`abbbc` や `abbbbc` を照合します。`ac`、`abc`、および `abbc` は照合しません。
- `\{i,j\}`  
`i ~ j` 回の繰り返しを照合します。たとえば、`ab\{2,4\}c` は、`abbc`、`abbbc`、または `abbbbc` を照合します。`abc` や `abbbbbc` は照合しません。`i` に 0 を指定することもできます。

拡張正規表現の場合、バックスラッシュは不要です。たとえば、上記の例の回数制限はそれぞれ `ab{3}c`、`ab{3,}c`、`ab{2,4}c` と指定します。

サブ正規表現デリミタ `\` (および `\`) を使用すれば、サブ正規表現パターンとして 1 行で最高 9 つまでの基本正規表現を保管することができます。行の左から右に数えて、1 番目に保管されたパターンは 1 番目の保管領域に、2 番目に保管されたパターンは 2 番目の保管領域に保管されます。

文字列 `\n` (`n` は 1 ~ 9 までの 1 桁の数字) は、`n` 番目に保存されたパターンを照合します。次のパターン例で考えてみましょう。

```
\(A\) \(B\) C\2\1
```

このパターンは文字列 `ABCBA` に照合します。保管領域に保管するパターンはネストしてもかまいません。カッコで囲まれているパターンがネストしているかどうかにかかわらず、`n` は、左から数えて `n` 番目に出現したデリミタを参照します。また、正規表現 `\n` は、アドレス・パターンに使用することも、`ed` や `sed` などのエディタで置換文字列として使用することもできます。

#### 1.1.4 選択した文字のみの照合

正規表現のピリオドは、改行文字以外の任意の文字を照合します。照合する文字を限定するには、限定する文字を大カッコ (`[]`) で囲みます。大カッコで囲まれた文字列は単一の正規表現であり、カッコで囲まれた任意の 1 文字を照合します。山形記号 (^) を除いて、大カッコ内の正規表現演算子はテキスト文字として解釈されます。大カッコの中の最初の文字として山形記号を

使用すると、カッコ内に指定した文字以外を照合します。その他の場所に指定した山形記号は、特別な意味を持ちません。

ダッシュを使用して文字の範囲を指定した場合 (たとえば [a-z])、指定範囲に含まれる文字は、LC\_CTYPE 環境変数で設定されている現在の照合順序によって決定されます。国際化機能および照合順序の使用方法についての詳細は、『*Tru64 UNIX ユーザーズ・ガイド*』を参照してください。ダッシュがカッコで囲まれた文字列の最初または最後の文字である場合、あるいは、山形記号がカッコ内の最初の文字で、ダッシュの直前にある場合には、ダッシュは特別な意味を持ちません。カッコで囲む文字列の中に、右大カッコを含める場合は、右大カッコを最初に、あるいは先頭の山形記号の直後に指定してください。

grep ユーティリティで -i オプションを使用すると、大文字と小文字を区別しないで照合を行うことができます。-y オプションは、-i オプションと同一です。その他のユーティリティで、大文字と小文字を区別しない正規表現を使用する場合や、部分的に大文字と小文字を区別しない正規表現を使用する場合には、カッコ内に指定する文字を大文字と小文字の両方で記述した正規表現を使用します。次に例を示します。

```
% grep '[Jj]ones' group-list
```

### 1.1.5 複数の正規表現の指定

grep (-E フラグを指定した場合) や awk など、ユーティリティによっては縦線で正規表現を区分することによって、複数の正規表現を同時に指定することができます。次に例を示します。

```
% awk '/[Bb]lack|[Ww]hite/ {print NR ":", $0}' .Xdefaults
55: sm.pointer_foreground: black
56: sm.pointer_background: white
```

### 1.1.6 正規表現における特別な照合

大カッコによるデリミタを使用して、クラスと呼ばれる特別なタイプの正規表現を指定することができます。

- 文字クラス  
たとえば大文字などの、文字のタイプを指定します。
- 照合シンボル・クラス

国際化機能。複数の文字を単一の文字としてソートするように指定します。

- 等価クラス

国際化機能。同じ第 1 ソート値を持つ文字の集合を指定します。

大カッコとコロンによるデリミタ [: と :] で文字クラス名を指定すると、指定した文字クラスに含まれる文字セットを照合します。各文字セットのメンバは、LC\_CTYPE 環境変数の現在の設定によって決定されます。サポートされる文字クラスは、alnum, alpha, cntrl, digit, graph, lower, print, punct, space, upper, xdigit です。たとえば、[:lower:] は現在のロケールのすべての小文字を照合します。

使用する照合順序によっては、複数の文字を 1 つの文字としてソート場合があります。たとえばハンガリー語の場合、文字列 cs, dz などそれぞれ照合シンボルです (ハンガリー語の場合、第 1 ソート順序は a, á, b, c, cs, d, dz, e, ... です)。このような特別な文字列を照合シンボルと呼び、大カッコとピリオドによるデリミタ [. と .] で囲むことによって指定することができます。正規表現構文でこのデリミタを使用すると、複数文字の照合要素 (照合シンボル) と、その照合要素を構成する各文字とを区別することができます。

ハンガリー語の照合規則を使用する場合、[.cs.] は cs を照合する 1 つの正規表現と見なされます。一方、[cs] は c または s を照合する正規表現と見なされます。また、[a-.cs.] は、a, á, b, c, および cs を照合します。

照合要素を大カッコと等号によるデリミタ [= と =] で囲むことにより、文字に対して等価クラスを定義することができます。等価クラスとして定義された文字は、すべて同じ第 1 ロケーションにソートされます。等価クラスは、一般に 2 段階ソート进行处理するために設計されています。つまり、フランス語などのように、ある文字グループを同じ第 1 ロケーションにソートし、次にそれらの文字の第 2 次ソートを行うように定義するためのものです。たとえば、e, é, ê が同じ等価クラスに属している場合、[=[e=] fg], [=[é=] fg], および [=[ê=] fg] は、それぞれ、[eéêfg] に相当します。照合順序とその使用方法、および国際化機能の使用法についての詳細は、『*Tru64 UNIX ユーザーズ・ガイド*』を参照してください。

## 1.2 grep コマンドの使用

grep とは global regular expression printer の略です。egrep コマンドおよび fgrep コマンドの機能は、それぞれ grep -E および grep -F で実現されています。これらのフラグを使用した場合の grep コマンドの動作について、表 1-3 で説明します。

表 1-3: grep コマンドの動作

grep のバージョン	説明
grep	基本 (basic) grep。パターンを基本正規表現として解釈します。
grep -E (egrep)	拡張 (extended) grep。パターンを拡張正規表現として解釈します。
grep -F (fgrep)	固定 (fixed) grep。すべての正規表現演算子をリテラル文字として解釈します。

すべての形式の grep コマンドに対して、1 つ以上のパターンを複数行に渡って指定することができます。この場合、次の Bourne シェルを使用した例のように、パターンをアポストロフィで囲み、改行文字で文字列を区分します。

```
$ strings hpcalc | grep -F 'math.h
> fatal.h'
```

C シェルの場合は、各改行文字の前にバックスラッシュを入力してください。

```
% strings hpcalc | grep -F 'math.h\
fatal.h'
```

また、-e フラグを使用して、複数のパターンを 1 行に指定することもできます。

```
% grep -e 'ab*c' -e 'de*f' myfile
```

省略時の設定では、grep コマンドは、指定したパターンを含む行をすべて検出します。表 1-4 に、検索時の動作を指定するための grep コマンドのオプションを示します。



表 1-4: **grep** コマンドのオプション

フラグ	説明
-b	各出力行の先頭にディスク・ブロック番号を出力します。このフラグは主に、特定のブロック内にある情報を検索することによって、プログラマがディスク上の特定ブロックを調べるために使用されます。
-c	指定したパターンが含まれる行の行数のみを出力します。
-e <i>pattern_list</i>	<i>pattern_list</i> をパターンとして使用します。複数のパターンを指定する場合は、改行で区切ります。 <i>pattern_list</i> が - で始まる場合に便利です。
-f <i>pattern_file</i>	パターンをファイルによって指定します。 <i>pattern_file</i> では、1 行に 1 つずつパターンを指定します。
-h	複数のファイル进行处理している場合にファイル名を出力しません。
-l	パターンを含むファイルのファイル名のみを表示します。1 つのファイル内で複数行に渡ってパターンが存在する場合も、ファイル名は 1 回だけ表示します。このフラグとともに、処理するファイルとして標準入力指定されている場合、ファイル名の代わりに (standard input) というメッセージを出力します。
-n	各出力行の先頭に行番号を出力します。
-p <i>paragraph_sep</i>	<i>paragraph_sep</i> をパラグラフ・セパレータとして使用し、一致した行を含むパラグラフ全体を出力します。パラグラフ・セパレータ行は出力されません。省略時のパラグラフ・セパレータは空白行です。
-q	“quiet” モードで動作します。エラー・メッセージ以外は何も出力しません。 <sup>a</sup>
-s	ファイルが存在しない場合あるいはファイルに到達できないために発生するエラー・メッセージの出力を制限します。その他のエラー・メッセージは出力されます。 <sup>a</sup>
-v	指定したパターンと一致しない行のみを出力します。
-w <i>expr</i>	<i>expr</i> がテキスト中のワードとして検索された場合にのみその行を出力します。ワードとは、句読点、ホワイト・スペースなどの英数字以外の文字で区切られた、(英字、数字、および下線で構成される) 英数文字列のことです。たとえば、 <code>word1</code> は、ワードですが、 <code>A+B</code> はワードではありません。

表 1-4: `grep` コマンドのオプション (続き)

フラグ	説明
<code>-x</code>	行全体が一致する場合のみ出力します。
<code>-y</code>	<code>-i</code> フラグと同じです。

<sup>a</sup>シェル・スクリプトのように成功状態あるいは失敗状態のみを必要とする場合にすべての出力を制限するには、標準出力および標準エラーをクローズするか、あるいはそれらを `/dev/null` へリダイレクトします。

正規表現の詳細については、`grep(1)` を参照してください。

---

## awk によるパターンの照合と情報の処理

この章では、awk コマンドについて説明します。awk コマンドは、ファイル内のテキスト行を照合するためのツールであり、照合した行を操作するための一連のコマンドを提供しています。

awk コマンドは、第 1 章で説明した拡張正規表現に従ってテキストを照合するだけでなく、各行すなわちレコードをフィールドの集合として扱い、各フィールドを個別にまたは組み合わせて処理することができます。そのため、awk コマンドを使用して、次のような複雑な操作を実行することもできます。

- レコード内の特定のフィールドへの書き込み
- レコードの内容の並び換え、または置き換え  
たとえば、別のシステムに移行する際にプログラム・ソース・ファイル内の構文を変更したり、システム・コールを変更する場合などに使用します。
- 入力処理し、数値の計算、合計または小計の算出を行う
- 所定のフィールドに数字情報だけが含まれていることを確認する
- プログラミング・ファイル内でデリミタのバランスがとれていることを確認する
- レコード内のフィールドに含まれるデータの処理
- あるプログラムのデータを別のプログラムで使用可能な形式へ変更する

この章では、次の事項について説明します。

- awk プログラムの実行 (2.1 節)
- awk における出力 (2.2 節)
- awk 変数の使用 (2.3 節)
- パターンとしての正規表現 (2.4 節)
- パターンとしての関係式および論理式 (2.5 節)

- パターン・レンジの使用 ( 2.6 節)
- awk のアクション ( 2.7 節)
- アクション内での演算子の使用 ( 2.8 節)
- アクション内での関数の使用 ( 2.9 節)
- awk プログラムでの制御構造の使用 ( 2.10 節)
- 入力の処理前および処理後に実行するアクション ( 2.11 節)
- 文字列の連結 ( 2.12 節)
- リダイレクションとパイプ ( 2.13 節)

## 2.1 awk プログラムの実行

awk コマンドの構文は、次のとおりです。

```
awk [[-F ERE]] [[-v var= val]] {[ -f prog_file ] | [ prog_textf] } [ file1 [ file2 ...]]
```

表 2-1 に、awk コマンドのフラグを説明します。

表 2-1: awk コマンドのフラグ

フラグ	説明
-F ERE	フィールド・セパレータとして使用する拡張正規表現を指定します。省略時の設定では, awk は空白 (タブまたはスペース) を使用してレコード内のフィールドを区切ります。空白またはシェル・メタキャラクタを含む別のセパレータを使用する場合は、次のようにオプション全体を引用符 (') で囲みます。  %echo \$PATH   awk -F' : ' '{for (n=1;n<=NF;n++)print \$n}'
-v var=val	変数 var に val を割り当てます。このような割り当ては、プログラムの BEGIN ブロックに対してのみ有効です。awk コマンドでは、複数の -v フラグを指定することができます。
-f prog_file	awk プログラムを含むファイルの名前を指定します。awk コマンドの場合、複数の -f フラグを指定することができます。この場合、指定したすべてのプログラムを連結して単一のプログラムとして処理します。

awk プログラムは、-f prog\_file フラグを使用してを実行することも、コマンド行にプログラムを記述して実行することもできます。ファイル名の展開あるいは変数の代入が必要な場合は、コマンド行プログラムを引用符

( ' あるいは " ) で囲みます。一重引用符 ( ' ' ) を使用し、`-v var=val` オプションを使用して `awk` プログラムへシェル変数を引き渡すと、`awk` プログラムが読みやすくなります。

通常、`awk` を実行する前に `awk` プログラム・ファイルを作成します。プログラム・ファイルの内容は、次のような文の連続です。

```
pattern { action }
```

*pattern* とは、照合するテキストを定義する 1 つまたは複数の式です。パターンは次のものから構成されます。

- BEGIN または END
- 論理演算子 ! (NOT) , || (論理 OR) , および && (AND) を使用した正規表現のブール演算の組み合わせ。カッコを付けて、式をグループにまとめます。
- 関係演算式による文字列、数字、フィールド、および変数のブール演算の組み合わせ。
- レコードの範囲。次のように指定します。

```
pattern1,pattern2
```

*action* とは、`awk` コマンド、オペランド、および演算子によって指定される、1 つまたは複数の処理です。アクションは次のものから構成されます。

- 代入文
- データのフォーマットおよびプリントを指定する文
- 制御の流れを選択するためのテスト
- if-else , while , および for 文などの制御構造
- 標準出力以外の、1 つまたは複数の出力ストリームへの出力リダイレクション
- 入出力のパイプ処理

中カッコ ( { } ) は、検索パターンとアクションを区別するためのデリミタです。アクションは、1 行にまとめて指定しても、見やすいように複数行にまたがって指定してもかまいません。複数のコマンドから構成されるアクションを 1 行で記述する場合は、セミコロンでコマンドを区切ります。たとえば、次の 2 つのプログラムはどちらも、'Gunther' または 'gunther' を含むす

すべてのレコードを検出します。awk は、検出したすべてのレコードに関して、次のような 2 行の情報を出力します。1 行目には照合したレコードの番号を、2 行目には照合したレコードの最初の 2 フィールドを出力します。

#### プログラム 1:

```
/[Gg]unther/ { print "Record:", NR ; print $1, $2 }
```

#### プログラム 2:

```
/[Gg]unther/ {  
    print "Record:", NR  
    print $1, $2  
}
```

このプログラムからの出力は、次のようになります。

```
Record: 382  
Schuller Gunther  
Record: 397  
schwarz gunther
```

パターンおよびアクションはいずれも、プログラム行ではオプションの要素です。パターンを省略した場合、awk はファイル内のすべてのレコードにアクションを実行します。アクションを省略した場合、awk はレコードを標準出力にコピーします。パターンとアクションの両方を省略した空プログラムの場合は、入力を変更しない状態で出力に渡します。

プログラム・ファイルを作成した後、次のように awk コマンドをコマンド行に入力してください。

```
$ awk -f progfile infile > outfile
```

このコマンドは、progfile のプログラムを使用して、infile を処理し、出力を outfile に書き込みます。入力ファイルは変更されません。

プログラムが短い場合は、入力ファイル名の前にコマンド行上にプログラムを入力することによって、同じジョブを実行することができます。たとえば、次のようにします。

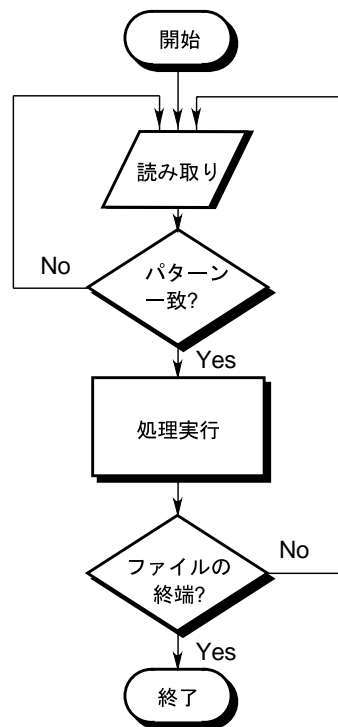
```
$ awk '/[Gg]unther/ { print $1, $2 }' infile
```

この方法で awk を使用する場合は、一重引用符 ( ' ' ) でプログラムを囲み、-v var=val オプションを使用してすべてのシェル変数を引き渡します。

awk を実行すると、awk は、プログラムを読み取り、構文をチェックします。次に、awk は入力ファイルの最初のレコードを読み取り、パターンの出

現順にプログラム・ファイルのそれぞれのパターンとレコードを突き合わせます。awk が、レコードに照合するパターンを検出した場合は、awk は対応するアクションを実行します。次に awk は、プログラム・ファイルの中で照合を検索し続けます。awk は、プログラム・ファイルのすべてのパターンに対して最初の入力レコードを突き合わせて、必要なすべてのアクションをレコードに実行した後、次の入力レコードを読み取り、そのレコードに対してプログラムを繰り返します。このように、処理は入力ファイルの終わりまで継続されます。図 2-1 は、この順序のフローチャートです。awk の動作と、図 3-1 で示した sed エディタの類似の動作とを比較してください。

図 2-1: awk 処理の順序



ZK0471UR

## 2.2 awk における出力

print コマンドまたは printf コマンドを使用して、awk で出力を生成することができます。print コマンドの構文では、引数をコンマまたは空白で区切ることができます。コンマで区切られた引数は、現在の出力フィールド・

セパレータ (OFS, 省略時は空白) を使って出力されます。空白で区切られた引数は, 連結されて出力されます。たとえば, 次のようになります。

```
awk 'BEGIN{ x=22; print "ABC" x, "DEF" }'  
ABC22 DEF
```

```
printf( "format", value1 [, value2, ...] )
```

このコマンドは, *format* 文字列の定義に従って *value1*, *value2* などの引数をフォーマットし, 出力します。フォーマット指定子の構文については, awk(1) および printf(3) を参照してください。

## 2.3 awk 変数の使用

awk プログラムでは, 情報を処理するために変数を使用します。変数には次の 3 つのタイプがあります。

- 単純変数 (2.3.1 項)
- フィールド変数 (2.3.2 項)
- 配列変数 (2.3.3 項)
- 組み込み awk 変数 (2.3.4 項)

awk 言語は表 2-2 に示す組み込み変数をサポートしています。また, 上記の 3 つのタイプの変数は, 作成したり変更したりすることもできます。たとえば, 次の代入文では, `var` という名前の変数を作成しています。変数の値は, 現在のレコードの第 3 および第 4 フィールドを合計したものです。

```
var = $3 + $4
```

変数をパターンの一部として使用したり, その変数をアクションで処理したりすることもできます。たとえば, 次のプログラムは, 変数 `tst` に値を割り当て, 以後のアクションのパターンの一部として `tst` を使用しています。

```
{ tst = $1 }  
tst == $3 { print }
```

2.3.1 項, 2.3.2 項, および 2.3.3 項で, 3 つのタイプの変数について説明します。これらの項で使用する例では, 2.4 節以降で説明するいくつかの awk 機能についても触れます。



### 2.3.1 単純変数

単純 (スカラ) 変数は、必要に応じて値を割り当てることによって、任意の数だけ作成することができます。明示的に値を割り当てる前に変数を参照した場合、`awk` は、その変数を作成し、空文字列値 ("") を代入します。変数には、アクション式内での使用方法に応じて、数値 (浮動小数点) または文字列値を指定することができます。たとえば、`x = 1` という式では、`x` は数値変数になります。同様に、式 `x = "smith"` では、`x` は文字列変数になります。しかし、`awk` では、必要に応じて、文字列と数値は、互いに自由に変換されます。その結果、たとえば `x = "3"+"4"` という式では、引数がりテラル文字列であるにもかかわらず、`awk` は `x` に値 7 を代入します。数値以外の値を含む変数を数値式で使用した場合には、`awk` は、数値 0 を変数に代入します。次に例を示します。

```
y = 0
z = "ABC"
x = y+z
print x, z
```

この例の場合、“00” が出力されます。これは、`y` には値 0 が代入され、`z` に代入された値 ABC も数値 0 に変換されるためです。

たとえば、`x = 2 ""` のように、変数値にヌル文字列 ("") を連結することによって、変数を強制的に文字列として処理することができます。文字列を連結する方法についての詳細は、2.12 節を参照してください。変数値の先頭に 0 を追加すれば、変数を強制的に数値として処理することができます。これらの方法は、変数のタイプを固定したい場合に便利です。たとえば、`x` の値が 0100 で `y` の値が 1 の場合、`awk` は通常、両方の変数を数値として処理して、`x` を `y` よりも大きいものと見なします。両方の変数を強制的に文字列として処理すると、`x` は `y` よりも小さくなります。これは、ASCII コードで 0 が 1 よりも前にあるためです。

### 2.3.2 フィールド変数

現在のレコード内の各フィールドはフィールド変数とも呼ばれ、単純変数の特性を共有しています。これらのフィールドは、算術または文字列演算に使用され、数字または文字列の値を代入することができます。`awk` では、現在のレコード (\$0) を明示的に変更することができます。次のアクションは、第 1 フィールドをレコード番号に置き換えて、結果のレコードを出力します。

```
{ $1 = NR; print }
```

次の例は、第 2 および第 3 フィールドを加算して、結果を第 1 フィールドに保存します。

```
{ $1 = $2 + $3; print $0 }
```

\$0 をプリントすることは、引数を付けずにプリントすることと同じです。

フィールド参照に数値式を使用することができます。次の例は、第 1、第 2、および第 6 フィールドを出力します。

```
i = 1  
n = 5  
{ print $i, $(i+1), $(i+n) }
```

2.3.1 項で説明されているように、awk は、文字列と数値間の変換を行います。フィールドの使用方法によって、awk がそのフィールドを文字列または数値のいずれで処理するかが決まります。指定されたフィールドの使用方法が決められていない場合には、awk はフィールドを文字列として処理します。

awk プログラムは、必要に応じて入力レコードを複数のフィールドに分割します。

### 2.3.3 配列変数

フィールド変数と同じように、配列変数も単純変数の特性を持っています。配列変数は、算術演算または文字列演算で使用し、数字または文字列の値を代入することができます。配列要素を宣言したり、初期化する必要はありません。awk によって配列要素が作成され、最初に参照された時点で、配列要素は空文字列("")に初期化されます。delete 文を使って不要な配列要素を削除することができます。詳細は、表 2-7 を参照してください。

添字は、大カッコで囲むことによって示します。添字には、文字列の値を含む、ヌル以外の値を使用することができます。数字の添字の例を次に示します。

```
x[NR] = $0
```

この式は、配列 *x* の *NR* 番目の要素を作成し、現在の入力レコードの内容をその要素に代入します。次に、文字列の添字を使用する方法の例を示します。

```
/apple/ { x["apple"]++ }  
/orange/ { x["orange"]++ }  
END { print x["apple"], x["orange"] }
```

このプログラムは、apple を含むそれぞれの入力レコードに対して、配列 *x* の *apple* 番目の要素をインクリメントします。orange についても同じ処理

を行います。その結果、このプログラムは、これらのそれぞれの単語を含むレコードの合計を計算し、出力します (単語はレコード内に複数含まれることがあるため、これは単語の数ではありません)。

配列要素を配置するために `if` または `while` 文を使用した場合、問題が発生する可能性があります。制御構造については 2.10 節を参照してください。配列の添字が存在しない場合は、これらの文でヌル値の配列要素を持った新しいハッシュ・テーブル・エントリとして添字が追加されてしまいます。たとえば、次のような場合です。

```
if (exists[$2] == 1) print i
```

このような問題を避けるには、次のようなコードを使用してください。このコードでは、配列要素が存在し、かつ配列要素の値が 1 の場合のみ `i` がプリントされます。

```
if (i in exists) {
    if (exists[i] == 1) print i
}
```

配列の全要素を処理するには、次のような `for` ループを使用します。

```
for(i in exists) {
    print exists[i]
}
```

また、`while` と関係演算子を併用する場合にも、このようなコーディングを使用してください。

リテラル文字列や文字列変数の値は、`split` 関数を使用して分割し、配列に格納することができます。たとえば次のようにします。

```
n = split("Thu Mar 18 11:19:40 EST 1999", array1)
m = split(array1[4], array2, ":")
```

この例の 1 行目では、リテラル文字列を分割して配列 `array1` の要素 (`array1[1]` から `array1[n]`) に格納します。このとき、`n` は文字列内のフィールドの数です。2 行目では、コロン (":") をセパレータとして、変数 `array1[4]` を分割し、`array2` に格納します (2.9 節を参照)。

### 2.3.4 組み込み `awk` 変数

`awk` プログラムでは、表 2-2 に示す組み込み変数を使用できます。

表 2-2: `awk` の組み込み変数

変数名	説明
<code>\$0</code>	現在のレコードの内容
<code>\$n</code>	入力レコードのフィールド <code>n</code> の内容 -- <code>awk</code> の場合レコード全体 ( <code>\$0</code> ) を変更できます。
<code>ARGC</code>	<code>awk</code> コマンド行の引数の数 -- この変数は変更可能です。コマンド名、マイナス記号付きのオプション、スクリプト・ファイル名 (もしあれば)、または変数割り当ては含みません。
<code>ARGV</code>	<code>awk</code> コマンド行の、コマンド名とそれに続く引数の配列 ( <code>ARGV[0]</code> から <code>ARGV[ARGC-1]</code> ) -- この配列の要素は変更可能です。マイナス記号が前に付いたオプション、スクリプト・ファイル名 (もしあれば)、または変数への代入は含みません。
<code>CONVFMT</code>	数値の変換フォーマット -- 省略時の設定は <code>%.6g</code> です。
<code>ENVIRON</code>	<code>ENVIRON["name"]</code> でアクセス可能な現在の環境変数を含む変更可能な配列 -- <code>"name"</code> は環境変数の名前を持つ変数またはリテラルです。この配列内の要素を変更しても、 <code>awk</code> がリダイレクション、パイプ処理、 <code>system()</code> 関数で生成したコマンドを渡す環境には影響しません。
<code>FILENAME</code>	現在の入力ファイルの名前 -- 入力ファイル名が指定されていない場合は、 <code>FILENAME</code> にはマイナス記号が含まれます。 <code>BEGIN</code> アクション内では、 <code>FILENAME</code> は定義されません。 <code>END</code> アクション内では、 <code>FILENAME</code> は最後のファイル読み取りを使用します。
<code>FNR</code>	現在のファイル内の現在のレコードの番号 -- 複数のファイルを処理しているときに、現在のファイルが複数ファイルの最初のファイルではない場合は、 <code>NR</code> とは異なります。
<code>FS</code>	フィールド・セパレータとして使用する文字または式 -- 省略時の設定は任意の数の空白です。 <code>awk</code> は、フィールド・セパレータとして複数バイトの正規表現を指定したり、複数のフィールド・セパレータを定義することができます。たとえば、次の例では、任意の数の空白が続くコンマ、または最低 1 つの空白を、フィールド・セパレータとして定義しています。  <code>FS = ", [ \t]*   [ \t]+"</code>
<code>NF</code>	現在のレコードのフィールド数

表 2-2: `awk` の組み込み変数 (続き)

変数名	説明
NR	最初に読み取られたファイルの先頭から順番に数えた場合の、現在のレコード番号 -- 複数のファイルを処理しているときに、現在のファイルが最初の読み取りファイルではない場合、 <code>FNR</code> とは異なります。
OFMT	数値の出力書式の指定 -- 省略時の設定は <code>%.6g</code> です。
OFS	出力フィールド・セパレータ -- データ書き込み時にフィールド間に出力される文字または文字列です。省略時の設定は空白です。
ORS	出力レコード・セパレータ -- データ書き込み時にレコード間に出力される文字です。省略時の設定は改行文字です。
RLENGTH	<code>match()</code> によって照合した文字列の長さ -- 一致するものがない場合は、 <code>-1</code> を設定します。
RS	入力レコード・セパレータとして使用する文字
RSTART	<code>match()</code> で照合した最初の文字のインデックス (文字列内の位置) -- 一致するものがない場合は、 <code>0</code> を設定します。
SUBSEP	配列要素内の複数の添字用のセパレータ -- 省略時の設定は、ASCII FS 文字 <code>\034</code> です。

これらの変数の詳細については、`awk(1)` を参照してください。

## 2.4 パターンとしての正規表現

最も単純な正規表現は、リテラル文字列です。`awk` では、正規表現をスラッシュで囲まなければなりません。正規表現の中にスラッシュを入れる場合には、バックスラッシュでスラッシュをエスケープします。たとえば、`/\usr\share/` は、文字列 `/usr/share` と一致する正規表現です。

次に示すのは、文字列 `the` を含むすべてのレコードを出力する `awk` プログラムの例です。

```
/the/
```

この正規表現には空白または他の修飾子が指定されていないため、プログラムは、単語として "the" を含むレコードと、northern のように単語の一部として文字列 "the" を含むレコードの両方を表示します。

正規表現は、大文字と小文字を区別します。"The" または "the" のいずれかを検出するには、次のように正規表現をカッコで囲ってください。

```
/[Tt]he/
```

awk プログラムは、第 1 章 で説明した拡張正規表現をサポートします。また、^ 記号および \$ 記号も、行全体とともに、特定のフィールドまたは変数に適用できます。次の例は、文字列 cat または cats のフィールドを照合しますが、これらの文字列を含む単語 (たとえば concatenate など) は照合しません。

```
{ for (i=1;i<=NF;i++) if ($i ~ /^cats?$/) print }
```

## 2.5 パターンとしての関係式および論理式

関係式を使用すると、レコードの特定のフィールドだけを照合したり、あるいは、数字または文字列に関連した他の照合を行うことができます。2.3 節では、パターンに関係式を使用する例を示しました。awk プログラムでは、パターンの作成に使用できるように、次の関係演算子を定義しています。

==	等しい
!=	等しくない
<	より小さい
>	より大きい
<=	より小さいか等しい
>=	より大きい等しい
~	正規表現と照合する
!~	正規表現と照合しない

リテラル文字列と数値を照合する場合には、== (等しい) および != (等しくない) 演算子を使用します。次に例を示します。

```
str == "literal string"
num != 23
$NF == 1991
```

この例の最後の行は、`$n` 構文に組み込み変数 `NF` を組み合わせて、レコードの最後のフィールドの値と照合しています。正規表現と照合させるには、次のように `~` (正規表現と照合する) および `!~` (正規表現と照合しない) 演算子を使用します。

```
str ~ /[Ll]iteral/
```

組み立て式に対しても、関係式をテストすることができます。たとえば、次のパターンは、第 2 フィールド (`$2`) が第 1 フィールド (`$1`) よりも 100 以上大きいレコードを検出します。

```
$2 > $1 + 100
```

次のパターンは、偶数個のフィールドからなるレコードを検出します。

```
NF % 2 == 0
```

式には、2.8 節に示されている演算子を使用します。

絶対比較演算子を使用して、文字列と照合させることができます。たとえば、次のパターンは、`"s"` で始まるレコードまたは文字セットの中で `s` の後にある任意の 1 文字 (`t, u, v, w, x, y, z`) を検出します。

```
$0 >= "s"
```

次のブール演算子を使用すれば、2 つ以上のパターンを結合することができます。

<code>&amp;&amp;</code>	論理積 (AND)
<code>  </code>	論理和 (OR)
<code>!</code>	否定 (NOT)

たとえば、前述した例で英数字以外の記号と照合しないようにするためには、次のように 2 つの式を組み合わせることもできます。

```
($0 >= "s" && $0 < "{")
```

左中カッコは、ASCII コードの英字 `z` の次の文字です。

## 2.6 パターン・レンジの使用

操作するレコードのグループを選択する場合には、パターン・レンジを使用します。パターン・レンジは、コンマによって区切られた 2 つのパターンから構成されています。最初のパターンは、レンジの開始を指定し、2 番目のパターンはレンジの終了を指定します。`awk` プログラムは、2 つのパターン

と照合するレコードも含め、レンジ内のすべてのレコードに対して対応するアクションを実行します。たとえば、次のとおりです。

```
NR==100,NR==200 { print }
```

このプログラムは、レコード 100 で始まり、レコード 200 で終わる 101 個のレコードを入力ファイルからプリントします。

パターン・レンジを使用した場合に、他のパターンが、レンジ内のレコードと照合しなくなることはありません。ただし、入力ファイルは 1 レコードずつ処理され、次のレコードが処理される前に、それぞれのレコードに対する適切なすべてのアクションが行われるため、次の例で示すように、順序どおりにアクションが行われていないようにみえることもあります。

```
2,4 { print }  
/share/ { print "Found share" }
```

このプログラムを次の入力ファイルに適用します。

```
This is a test file  
Line two  
Try to share things  
Line four  
Last line of file
```

このファイルが `awk` によって処理されると、出力は次のようになります。

```
Line two  
Try to share things  
Found share  
Line four
```

レコード 4 が最初のパターンに照合するかどうかを検証される前に、2 番目のアクションがレコード 3 に適用されます。

## 2.7 `awk` のアクション

アクションには、たとえば `print` のように単一のコマンドを指定することも、複数のコマンドを指定することもできます。アクションとして、レコードまたはレコードの一部の選択を指定することもできます。アクション内の関係式を代わりに使用して、明示的にパターンを指定しないプログラムを作成することもできます。このようなプログラムは、次の例に示すように、C プログラムに非常に似ています。

```
{  
    if ($1 == 0) {  
        print;  
    }
```



```

    printf("%5.2f\n", $2+$3)
  } else {
    printf("%5.2f\n", $1+$2)
  }
}

```

#### 注意

print コマンドの後にあるセミicolonは、C プログラムでは必須ですが、awk には必須ではありません。しかし、このセミicolonがあっても、エラーの原因となることはありません。

## 2.8 アクション内での演算子の使用

アクション文内の式を作成する場合には、表 2-3 に示されている演算子を使用します。

表 2-3: awk アクションの演算子

優先順位	演算子	説明	例
1	()	カッコ	$3+x*4 = 3+(x*4)$
2	\$	フィールド参照	$\$(NF-1)$ = 最後のフィールドの次
3	++	インクリメント	この表の後の説明を参照
3	--	デクリメント	この表の後の説明を参照
4	^	べき乗	$2^3 = 8$
5	!	論理否定	$!x$ は $x$ と等しくない
6	+	単項演算子のプラス	$+4 = 4$
6	-	単項演算子のマイナス	$-4$ は負の 4
7	*	乗算	$2*4 = 8$
7	/	除算	$6/3 = 2$
7	%	モジュロ (剰余)	$7\%3 = 1$
8	+	加算	$2+3 = 5$
8	-	減算	$7-3 = 4$

表 2-3: awk アクションの演算子 (続き)

優先順位	演算子	説明	例
9	空白	連結	"a" "b" = "ab"
10	<	より小さい	5 < 6
10	>	より大きい	"qrs" > "abc"
10	<=	より小さいか 等しい	3 <= 3
10	>=	より大きいか 等しい	4 >= 2
10	==	等しい	9 == 9
10	!=	等しくない	"xyz" != "abc"
11	~	正規表現と一致	"tmp.c" ~ /[a-z]+\.[ch]/
11	!~	正規表現と一致 しない	"tmp.o" !~ /[a-z]+\.[ch]/
12	in	配列のメンバ シップ	for (j in arr) print arr[j]
13	&&	論理積 (AND)	X
14		論理和 (OR)	X
15	?:	条件式	x == -1 ? "error" : "OK"
16	=	代入	x = 3
16	^=	値をべき乗	x^=3 は x = x^3 と同等
16	*=	値を乗算	x*=y は x = x*y と同等
16	/=	値を除算	x/=y は x = x/y と同等
16	%=	値のモジュロ	x%=y は x = x%y と同等
16	+=	値をインクリ メント	x+=y は x = x+y と同等
16	-=	値をデクリメント	x-=y は x = x-y と同等

入力ファイルのすべての第 1 フィールドの合計，およびすべての第 2 フィールドの合計をプリントする例を次に示します。

```
{ s1 += $1; s2 += $2 }
END { print s1,s2 }
```

インクリメントおよびデクリメント演算子の位置は、解釈に大きな影響を与えます。 `i++` という式は、`i` の現在の内容を評価してから、`i` をインクリメントします。 `++i` という式を使用すると、`awk` は評価の前に `i` をインクリメントします。 次に例を示します。

```
$ echo "3 3" | awk '{
>   print "$1 =", $1 "; $1++ =", $1++; new $1 =", $1
>   print "$2 =", $2 "; ++$2 =", ++$2 "; new $2 =", $2
> }'
$1 = 3; $1++ = 3; new $1 = 4
$2 = 3; ++$2 = 4; new $2 = 4
```

## 2.9 アクション内での関数の使用

`awk` は、表 2-4 に示す組み込み算術関数をサポートしています。

表 2-4: 組み込み `awk` 算術関数

関数	説明
<code>atan2(x, y)</code>	<code>x/y</code> で指定された値の逆正接を返します。
<code>cos(expr)</code>	<code>expr</code> で指定された値 (ラジアン) の余弦を返します。
<code>exp(arg)</code>	<code>arg</code> の自然真数 (底は <code>e</code> ) を返します。たとえば、 <code>exp(0.693147)</code> は値 <code>2</code> を返します。 <code>log(arg)</code> を参照。
<code>int(arg)</code>	<code>arg</code> の整数部を返します。
<code>log(arg)</code>	<code>arg</code> の自然対数 (底は <code>e</code> ) を返します。たとえば、 <code>log(2)</code> は <code>0.693147</code> を返します。 <code>exp(arg)</code> を参照。
<code>rand</code>	偽似乱数 ( $0 \leq n < 1$ ) を返します。
<code>sin(arg)</code>	<code>arg</code> で指定された値 (ラジアン) の正弦を返します。
<code>sqrt(arg)</code>	<code>arg</code> の平方根を返します。
<code>srand(seed)</code>	<code>rand</code> に引き続くコールに対する偽似乱数のシードとして <code>seed</code> を使用します。シードが何も指定されない場合には、日付の時間が使用されます。返す値は、その前のシードです。

`awk` は、表 2-5 に示す組み込み文字列関数をサポートしています。

表 2-5: 組み込み awk 文字列関数

関数	説明
<code>gsub(expr, s1, s2)</code>	文字列 <i>s2</i> 内で正規表現 <i>expr</i> に一致するすべての文字列を, <i>s1</i> で指定された文字列に置換します。 <i>s2</i> が指定されていない場合には, 現在の入力レコードが使用されます。正規表現 <i>expr</i> は照合のたびに再評価されます。この関数は置換した数を表わす値を返します。 <code>sub(expr, s1, s2)</code> を参照。
<code>index(s1, s2)</code>	文字列 <i>s2</i> 中の部分文字列 <i>s1</i> の文字位置を返します。 <i>s2</i> が <i>s1</i> に含まれない場合, この関数は 0 を返します。
<code>length</code>	現在のレコードの文字列の文字数を返します。
<code>length(arg)</code>	<i>arg</i> で指定された文字列の長さを返します。 <code>length</code> を参照。
<code>match(s, expr)</code>	文字列 <i>s</i> 内で正規表現 <i>expr</i> との一致が見つかった文字位置を返します。 <code>RSTART</code> 変数を照合が始まる文字の位置に設定し, <code>RLENGTH</code> 変数を照合文字列の長さを表す値に設定します。一致するものがない場合, この関数はゼロ (0) に返します。
<code>split(s, array, sep)</code>	文字列 <i>s</i> を <code>array[1] ... [n]</code> の連続する要素に分割し, 要素の数を返します。 オプションの <i>sep</i> 引数には, 現在使用されているフィールド・セパレータ (省略時は <code>FS</code> 変数の内容) 以外のフィールド・セパレータを指定することができます。
<code>sprintf(f, e1, e2, ...)</code>	引数 <i>e1</i> 等を含む文字列を <code>printf</code> コマンドと同じ方法でフォーマットして返します。プリントは行いません。
<code>sub(expr, s1, s2)</code>	文字列 <i>s2</i> 内で正規表現 <i>expr</i> に一致する最初の文字列を, <i>s1</i> で指定される文字列と置換します。 <i>s2</i> が指定されていない場合は, 現在の入力レコードが使用されます。この関数は置換した数 (0 か 1) を返します。 <code>gsub(expr, s1, s2)</code> を参照。
<code>substr(s, m, n)</code>	文字位置 <i>m</i> から始まり, 長さが <i>n</i> 文字の, 文字列 <i>s</i> の部分文字列を返します。 <i>s</i> の最初の文字の文字位置は 1 です。 <i>n</i> が省略された場合, あるいは文字列の長さが <i>n</i> 文字未満の場合は, 残りの文字列が返されます。
<code>tolower(s)</code>	文字列 <i>s</i> 内の大文字をすべて小文字に変換する。引数がない場合には, 現在のレコードを処理します。
<code>toupper(s)</code>	文字列 <i>s</i> 内の小文字をすべて大文字に変換する。引数がない場合は, 現在のレコードを処理します。

awk は、表 2-6 に示すその他の組み込み awk 関数をサポートしています。

表 2-6: その他の組み込み awk 関数

関数	説明
<code>close(arg)</code>	<code>arg</code> で指定したファイルまたはパイプをクローズします。
<code>system("command")</code>	システム・コマンドを実行し、終了 (Exit) 状態を返します。変数名として awk が解釈しないように、コマンド全体を " 記号で囲みます。

awk プログラムでは、次の構文で関数を作成することもできます。

```
function name ( parameter-list ) {
    statements
}
```

function の代わりに func を使用することもできます。

新たに作成した関数の場合、関数を定義する場合も関数を作成する場合も、関数名の後ろの左カッコと関数名との間にはスペースを入れないで記述する必要があります。

関数宣言のパラメータのリストで使用した名前は、その関数内で使用する形式的なものです。実際に関数を呼び出す際に、これらの形式的なパラメータは呼び出し文で指定された値に awk によって置き換えられます。関数は再帰的に使用できます。

関数エントリ上で形式的なパラメータを特別に宣言することによって、関数に対してローカル変数を定義することができます。すべてのローカル変数は空の文字列あるいは 0 に初期化されます。実パラメータとローカル変数とを混同しないように、次の例のように余分なスペースを使用してローカル変数を区別することができます。

```
function foo(in, out,      local1, local2) {
    local1 = "foo"
    local2 = "bar"
    :
}
```

## 2.10 awk プログラムでの制御構造の使用

awk 言語は、表 2-7 に示す制御構造をサポートしています。特に明記しない限り、これらの制御構造は C 言語の場合と同じように動作します。単一の制御構造のアクションとしていくつかの文を実行する場合は、それらの文を中カッコで囲みます。制御構造内で実行する文が 1 つだけの場合は、中カッコは省略できます。

次の例の最初の 2 つの if 制御構造には、実行する文がそれぞれ 1 つ含まれています。これらの制御構造の機能は、どちらも同じです。

```
{
  if (x == y) print
  if (x == y) {
    print
  }
  if (x == y) {
    print $3
    printf("Sum = %d\n", x+z)
  }
}
```

表 2-7: **awk** の制御構造

制御構造	説明
if-else	<p>if-else 制御構造の小カッコ内の条件が評価されます。真の場合は、if に続く文が実行されます。偽の場合は、else に続く文が実行されます。else キーワードは省略することもできます。連続した if 文は、else if 文で指定できます。</p> <p>次のように、"else" と "if" を記述する順番は重要です。</p> <pre>if ( \$1 == "abc" ) {      print("found abc\n"); } else if ( \$1 == "qrs" ) {     print("found qrs\n"); } else if ( \$1 == "xyz" ) {     print("found xyz\n"); } else {     print("did not find \"abc\", \"qrs\", or \"xyz\"\n"); }</pre>
delete	<p>delete 文を使用すると、配列要素を削除することができます。たとえば次の例では、配列 x の要素がすべて削除されます。</p> <pre>{     for(j in x)         delete x[j] }</pre>
while	<p>テストされた条件が真でなくなるまでの間、while 文に続く文が実行されます。次の例では、入力レコードのすべてのフィールドを、1 行に 1 フィールドずつ出力します。</p> <pre>{     i = 1     while(i&lt;=NF) print \$i++ }</pre>

表 2-7: **awk** の制御構造 (続き)

制御構造	説明
for	<p><code>for(expr1;expr2;expr3) statements</code> の形式の制御構造は、次の <code>while</code> 制御構造と同じ動作をします。</p> <pre>{   expr1   while(expr2) {     statements   }   expr3 }</pre> <p>たとえば、上記の <code>while</code> の例を、次のように記述することもできます。</p> <pre>{   for(i=1;i&lt;=NF;++i) print \$i }</pre> <p><code>for(i in array)</code> 文は、配列のすべての要素を処理します。</p> <pre>\$2==""&gt;{name_value_pairs[\$1]=\$3} end{   for (i in name_value_pairs)     print name_value_pairs[i] }</pre>
break	<p><code>break</code> 文を実行すると、<code>while</code> ループまたは <code>for</code> ループから即座に抜けます。</p>
コメント	<p><code>awk</code> プログラム・ファイル内にプログラムの説明を記述します。コメントは、<code>(#)</code> 記号で始まり、行の最後で終了します。次に例を示します。</p> <pre>{   print x,y      # This is a comment }</pre>
continue	<p><code>continue</code> 文は、ループを最初から繰り返します。</p>
getline	<p><code>getline</code> 文を実行すると、<code>awk</code> は、現在の入力レコードを捨て、次の入力レコードを読み取り、現在の位置からパターンの走査を開始します。<code>getline var</code> を使用すると、<code>getline</code> の入力を変数に割り当てることができます。<code>var</code> を指定しなければ、入力は現在のレコードに割り当てられます。</p>



表 2-7: `awk` の制御構造 (続き)

制御構造	説明
<code>next</code>	<code>next</code> 文を実行すると、 <code>awk</code> は、現在の入力レコードを捨て、次の入力レコードを読み取り、プログラム・ファイルの先頭からパターンの走査を開始します。
<code>exit</code>	<code>exit</code> 文は、入力ファイルの終端に到達したときのようにプログラムを停止します。

## 2.11 入力の処理前および処理後に実行するアクション

`awk` プログラムは、入力ファイルの開始および終了を定義するための 2 つの特別なパターン・キーワード `BEGIN` および `END` を識別します。

`BEGIN` は、最初のレコードを読み取る前に、入力の先頭と一致します。このため `awk` は、入力ファイル进行处理する前に、このパターン (`BEGIN`) に対応するアクションを 1 回実行することになります。たとえば、ファイル内のすべてのレコードのフィールド・セパレータをコロン (:) に変更する場合は、プログラム・ファイルの最初に次の 1 行を記述します。

```
BEGIN { FS = ":" }
```

この例のアクションは、コマンド行で `-F:` フラグを使用した場合と同じです。

同様に、`END` は、最後のレコードを読み取った後に、入力ファイルの最後と一致します。このため、`awk` は、入力ファイル进行处理した後、このパターン (`END`) に対応するアクションを 1 回実行することになります。たとえば、入力ファイルのレコードの総数を出力するためにはプログラム・ファイルに次のように記述します。

```
END { print NR }
```

## 2.12 文字列の連結

変数名を 1 つの式にまとめることによって、文字列を連結します。たとえば、コマンド `print $1 $2` は、現在のレコードの最初の 2 つのフィールドを構成する文字列を、空白を入れないでプリントします。文字列を連結する場合は、変数、数値演算子、関数を使用することができます。変数と数値演算子についての詳細は、2.3.1 項および 2.8 節を参照してください。関数 `length($1 $2 $3)` は、最初の 3 つのフィールドの長さを文字数で返します。`awk` 関数のリストについては、2.9 節を参照してください。連結する文

字列がフィールド変数 (2.3.2 項参照) である場合は、空白で名前を区切る必要はありません。たとえば、式 `$1$2` は、`$1 $2` と同一です。

## 2.13 リダイレクションとパイプ

特に指定しない限り、`print` 文および `printf` 文は、出力を標準出力ファイルに書き込みます。標準リダイレクション演算子を使用すれば、プリント文の出力をリダイレクトすることができます。次に例を示します。

```
print $0, $3, amt >> "reportfile"
```

この例は、その出力を標準出力に書き込まないで、`reportfile` という名前のファイルに追加します。リダイレクションの最初のインスタンスの前に `reportfile` がない場合は、`reportfile` が作成されます。この例の出力ファイル名は引用符で囲まれています。引用符は、ファイル名を変数名と区別するために必要です。書き込みは、指定されたファイル、標準出力のいずれにも行うことができます。

`print` と `printf` 文は常に、その出力を標準出力に送ります。次の例では、出力を標準エラー出力に送ります。

```
print "oops: did not find expected input" | " cat 1>&2"
```

また、プリントされる出力を他のコマンドにパイプすることもできます。次に、`awk` の出力を `tr` コマンドにパイプして、大文字をすべて小文字に変換する例を示します。

```
print | "tr ' [A-Z]' '[a-z]'"
```

リダイレクションを使用する場合と同様に、出力のパイプ先のコマンドは、引用符で囲まなければなりません。

`awk` は、標準リダイレクト演算子を使用して `getline` への入力をリダイレクトすることができます。そのため、次のように `getline` 文への入力をパイプで提供することができます。

```
expr | getline
```

この場合、`expr` はシステム・コマンドとして解釈されます。

次の例では、システム・コマンドからの出力を読み取ります。

```
BEGIN {  
    cmd = "ps aux"  
    while( cmd | getline > 0 ) {  
        if ( $2 == "PID" ) continue
```

```

unique_users[$1]++
}
close(cmd)

for(i in unique_users) {
    printf("%3d %s\n", unique_users[i], i)
}
}

```

出力用にオープンできるファイルの数は制限されています。awk プログラムには、ユーザのオープン・ファイル記述子の省略時の制限が適用されます。しかし、効率を高めるために、`close(arg)` 文を使用して、出力用にオープンしたファイルのうち不要なものをクローズすることもできます。たとえば、次のようにします。

```

{
if ( cur_file != "/tmp/" $1 ) {
    close(cur_file)
    cur_file = "/tmp/" $1
}
print $2 >cur_file
}
END { close(cur_file) }

```



---

## sed エディタによるファイルの編集

この章では、ed と同じ機能を持ったプログラムである sed ストリーム・エディタについて説明します。この章で説明されている操作を行うために、ed ライン・エディタの使用方法を知っている必要はありません。次の各節で、sed エディタについて説明します。

- sed エディタの概要 (3.1 節)
- sed エディタの実行 (3.2 節)
- 編集行の選択 (3.3 節)
- sed コマンドの要約 (3.4 節)
- 文字列の置換 (3.5 節)

ed と異なる点は、sed はユーザと対話するのではなく、スクリプトと呼ばれるコマンド・リストを使用してファイルの編集を行う点です。このような編集形式をとるため、sed は次のような作業に最適です。

- サイズの大きいファイルを編集する。
- 長時間にわたるタイプ入力の繰り返しやカーソルの移動をすることなく、複雑な編集操作を何度も実行する。
- 1 度の入力でグローバルな変更を行う。

### 3.1 sed エディタの概要

sed ストリーム・エディタは、標準入力または指定したファイルから入力を受け取り、コマンド・ファイルまたはコマンド行のコマンドによって入力を変更し、その結果のストリームを標準出力に書き込みます。2 つ以上の入力ファイルを指定すると、sed は、各ファイルを順番に処理して、その結果を標準出力に連結します。コマンド・ファイルを指定せずに、sed コマンドオプションも使用しなかった場合、sed は、標準入力を変更せず標準出力にコピーします。このエディタは、編集中のファイルの 2、3 行をメモリに保持す

るだけなので、一時ファイルを使用しません。したがって、編集されるファイルのサイズは、使用可能なディスク・スペースによってだけ制限されます。

sed で使用するコマンド・スクリプトは、エディタを実行する前に作成しておいたファイルでも、また、コマンド・オプションとして入力した一連のコマンドでも、またはその両方でもかまいません。エディタは、1 回の呼び出しで 99 個までのコマンドしか処理することができません。このような制約のため、また、かなり複雑な編集作業を実行するために、sed からの出力を sed の別のインスタンス にパイプする必要が生じることがあります。

### 3.2 sed エディタの実行

sed コマンドの構文は次のとおりです。

```
sed [[-n]] [[[-e]][ script]] [[-f script_file]] [[source_file1 ]][[source_file2 ...]]
```

表 3-1 に、sed コマンドのフラグを示します。

表 3-1: sed コマンドのフラグ

フラグ	説明
<code>-e <i>script</i></code>	文字列スクリプトで指定された編集コマンドを、編集コマンドのスクリプトの終わりに追加する。 <code>-e</code> フラグを 1 つだけ使用し、 <code>-f</code> フラグを指定しない場合は、 <code>-e</code> フラグを省略すると、1 つの <i>script</i> をコマンド行に sed の引数として指定することが可能。
<code>-f <i>script_file</i></code>	<i>script_file</i> を編集スクリプトのソースとして使用する。 <i>script_file</i> は、入力に適用される編集コマンドの集合。
<code>-n</code>	通常では標準出力へ書き込まれる情報を抑制する。

`-e` および `-f` オプションの表記順は重要です。通常、sed を実行する前に、必要な編集コマンドを記述したコマンド・ファイルを作成します。sed エディタのコマンド・セットは強力で、ほとんど入力の必要はありません。コマンド・ファイル内の各コマンドは、個別の行に入力することも、セミコロン (;) を使用して各コマンドを区切り、複数のコマンドを 1 行にタイプすることもできます。たとえば、次の 2 つのスクリプトは、いずれも `.ne`、`.RE`、または `.RS` で始まるすべての行を削除します。

スクリプト 1:

```
/^\.ne/d
/^\.R[ES]/d
```

#### 3-2 sed エディタによるファイルの編集

## スクリプト 2:

```
/^\.ne/d;/^\.R[ES]/d
```

コマンド・ファイル (次の例では `cmdfile`) を作成した後、次の例のように `sed` コマンドを入力します。

```
$ sed -f cmdfile infile > outfile
```

このコマンドは、`cmdfile` に含まれているコマンドを使用して、`infile` を編集し、出力を `outfile` に書き込みます。入力ファイルは変更されません。

短い編集スクリプトの場合には、`-e` オプションの引数として編集コマンドをコマンド行に入力しても同じジョブを実行できます。

```
$ sed -e '/^\.ne/d;/^\.R[ES]/d' infile > outfile
```

同一のコマンド行で `-e` オプションと `-f` オプションを同時に使用すると、両方のオプションで指定されたすべてのコマンドが実行されます。実行される順番は、オプションがコマンド行に指定されている順番です。次に例を示します。

```
$ echo "s/line/foo/" > sedx
$ echo "Test line" | sed -f sedx -e 's/line/bar/'
Test foo
$ echo "Test line" | sed -e 's/line/bar/' -f sedx
Test bar
```

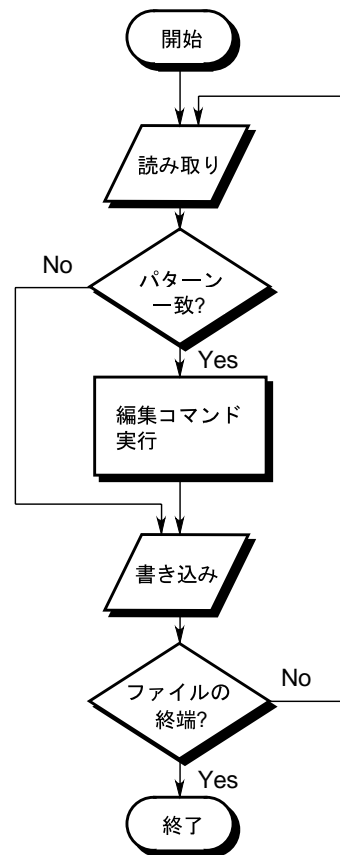
たとえば、次のように、1 個の `sed` コマンドに `-e` オプションと `-f` オプションを 2 回以上使用することができます。

```
$ sed -f script1 -e 's/foo/bar/' -f script2 msgs > msgs2
```

`sed` を実行すると、このエディタは構文のチェックおよび効率をよくするためのコマンドの編成を行いながら、コマンド・スクリプトを読み込んでコンパイルします。その後、入力ファイルを 1 行ずつパターン・スペースというメモリ領域に読み込みます。次に、エディタはスクリプトに記述してあるコマンドで指定されたアドレスを、パターン・スペースの行に対して次々に照合させます。コマンドのアドレスがパターン・スペース内の 1 行または複数行と照合した場合、`sed` は必ず照合したテキストにその編集コマンドを実行します。各コマンドはテキストに対して順番に実行されます。各コマンドで変更した結果は、後続するコマンドの入力として使用されます。パターン・スペース内の所定の行がそれ以上コマンドと照合なくなると、`sed` は出力にその行を書き込み、次の入力を読み込みます。この処理は繰り返し行われ

まず、この動作の順序のフローチャートは、図 3-1 のとおりです。sed の動作は、図 2-1 に記載した awk プログラムの動作と非常によく似ています。

図 3-1: sed 処理の順序



ZK0453UR

編集コマンドによっては、エディタに他のスクリプト・コマンドをバイパスさせたり、または (それらを削除させることで) ある行の書き込みを禁止させたり、または処理を中断させたりして、編集処理の動作方法を変更するものがあります。

### 3.3 編集行の選択

sed エディタは、アドレスの照合によって編集する行を識別します。アドレスは、行番号でもコンテキスト・アドレスでもかまいません。

- 行番号

#### 3-4 sed エディタによるファイルの編集



入力ストリームの開始行は 1 行目になります。ライン・カウンタは後続する各行ごとに 1 ずつ増分します。ドル記号 (\$) は、入力ストリームの最終行を指定するための略字です。1 回の sed の呼び出しで 2 つ以上のファイルを編集する場合、ライン・カウンタは編集されたすべてのファイルの行数を加算していきます。たとえば、最初のファイルが 100 行の場合は、2 番目のファイルの開始行は 101 行目となります。

• コンテキスト・アドレス

コンテキスト・アドレスは、パターン・デミリタ (通常はスラッシュ) で囲まれた正規表現です。たとえば `/^\.R[ES]/` は、`.RE` または `.RS` で始まるすべての行と照合します。

パターン・デリミタとしてスラッシュ以外の文字を使用したい場合は、最初に使用する際にその文字の前にバックスラッシュを付けることによって可能になります。たとえば、次の 2 つのパターンは同じものとして解釈されます。

`/abc/`  
`\xabcx`

2 つ目のパターンでは、パターン・デミリタとして `x` を使用しています。パターン・デミリタとして使用している文字をリテラル文字として使用したい場合には、前にバックスラッシュを付けます。たとえば、`\x\xyzx` は、文字列 `xyz` と解釈されます。

sed エディタは、第 1 章で説明した基本正規表現に加えて、表 3-2 に示す正規表現も認識します。

表 3-2: sed で認識される正規表現

正規表現	名称	規則
<code>\n</code>	改行	埋め込まれた改行文字を照合します。
<code>//</code>	空のパターン・デミリタ	直前に指定された正規表現と一致したテキストを照合します。

sed コマンドのなかには、アドレスを受け入れないものがあります。アドレスを受け入れるコマンドは、指定されるアドレスの数によって次のように異なった動作をします。

- アドレスが指定されなかった場合、そのコマンドは入力ストリームのすべての行に対して実行されます。

- 1つのアドレスが指定された場合は、コマンドはそのアドレスと照合する各行に対して実行されます。
- 2つのアドレスが指定された場合、最初のアドレスが照合する行から2番目のアドレスが照合する行までのすべての行に対してコマンドが実行されます。その後で、対象となる別の範囲を見つけるために、エディタは再び最初のアドレスでの照合を行います。

---

#### 注意

---

2つのアドレスが指定された場合に、`sed` が終わりのアドレスと照合する行を見つけられないと、`sed` はファイルの最初のアドレスからファイルの最後まですべての行に対して実行されます。

---

### 3.4 `sed` コマンドの要約

各 `sed` コマンドは、オプションのアドレスを持つ1文字の英字で構成されています。コマンドのなかには、引数を必要とし、コマンドの動作を変更させる修飾子を受け入れるものがあります。アドレスと英字の間には、スペースを入れてはなりません。1つのコマンドに2つのアドレスを使用する場合は、コンマでアドレスを区切ってください。`r` コマンドと `w` コマンド、および、`s` コマンド用の `w` フラグを使用する場合は、英字と引数の間にスペースが1つ必要です。それ以外では、英字と引数の間にスペースを入れてはなりません。

表 3-3、表 3-4、および表 3-5 では、個々の `sed` コマンドについて説明し、それぞれの構文を記述します。これらの表については、次の規約が適用されます。

- 行範囲という用語は、コマンドに与えられたアドレスの数で指定される1行、複数行、またはすべての行を意味します。
- 大カッコ [ ] はオプション要素を囲みます。ネストした大カッコは、大カッコで囲まれた要素が存在する場合のみネストした要素を使用できることを表します。
- イタリック体 (斜体) は、指定する対象の一般名を示します。たとえば *file* は、コマンド引数としてファイル名を使用することを表します。

すべてのオプションの要素を含む `s` コマンドの例を次に示します。

```
1,/^$/s/vizier//g
```

この例は、メール・メッセージのヘッダ (1 行目から最初に出現する空白行まで) を処理します。指定された範囲内の文字列 `vizier` をすべて削除します。

表 3-3 はテキスト編集および移動の `sed` コマンドについて説明し、構文を示します。

表 3-3: テキスト編集および移動のコマンド

コマンド	説明
テキストの追加	
<code>[addr1]a\ text[\ text...]</code>	<code>addr1</code> で指定された行の後ろに、指定されたテキスト <sup>a</sup> を書き込んで出力する。i コマンドを参照。
行の変更	
<code>[addr1 [,addr2]]c\ text[\ text...]</code>	アドレスで指定された行範囲を削除して、指定されたテキスト <sup>a</sup> を、削除された所書き込んで出力する。 <sup>b</sup>
行の削除	
<code>[addr1 [,addr2]]d</code>	指定された行範囲を削除する。 <sup>b</sup>
パターン・スペースの最初の行の削除	
<code>[addr1 [,addr2]]D</code>	最初の改行文字が表れるまでのパターン・スペースのすべてのテキストを、その改行文字を含めて削除する。パターン・スペースに 1 行しかない場合は、このコマンドにより別の行が入力からパターン・スペースに読み込まれる。これらの動作の後で、このコマンドは編集コマンドのリスト全体を先頭から再開する。
行の挿入	
<code>[addr1 ]i\ text[\ text...]</code>	指定されたテキスト <sup>a</sup> を、 <code>addr1</code> で指定された行の前に書き込んで出力する。a コマンドを参照。
次の行の読み込み	
<code>[addr1 [,addr2]]n</code>	パターン・スペースの中の、指定された範囲を (削除されていないければ) 書き込んで出力する。次に、入力から次の行をパターン・スペースに読み込む。
行の結合	

表 3-3: テキスト編集および移動のコマンド (続き)

コマンド	説明
<code>[addr1 [,addr2 ]]N</code>	指定された複数の行を、埋め込まれた改行文字を持つ 1 行となるように結合させる。アドレスが 1 つしか与えられていない場合、このコマンドは指定された行を入力ストリームの次の行と結合する。アドレスの指定、または文字列置換のためのパターン照合は、埋め込まれた改行文字をまたがって拡張することができる。照合用に埋め込まれた改行文字を示すには、 <code>\n</code> を使用する。
行の表示	
<code>[addr1 [,addr2 ]]p</code>	編集処理の実行中に <code>p</code> コマンドが出現した時点で、指定された行範囲を出力に書き込む。このコマンドは、ファイルのセクションを並べ換えるために使用できる。
パターン・スペースの開始行の表示	
<code>[addr1 [,addr2 ]]P</code>	編集処理の実行中に <code>P</code> コマンドが出現した時点で、最初の改行文字が出現するまでのパターン・スペースのすべてのテキストの内容を、その改行文字を含めて出力へ書き込む。
ファイルの読み込みと追加	
<code>[addr1]r file</code>	指定されたファイル <sup>c</sup> を読み込んで、そのファイルの内容を <code>addr1</code> に続けて出力へ書き込む。
テキストの置換	
<code>[addr1 [,addr2 ]]s/expr /string /[flags ]</code>	指定された行の中から、 <code>expr</code> で定義した正規表現と照合する文字列を見つけ出し、その文字列を <code>string</code> と置換する。このコマンドの動作は、 <code>g</code> 、 <code>p</code> 、および <code>w</code> の <code>file</code> フラグで修飾される。 <code>expr</code> または <code>string</code> にスラッシュ (/) が入っている場合は、バックスラッシュ ( <code>s/path/path\file/</code> ) を使用して、文字としてのスラッシュを避ける必要がある。あるいは、アット符号 (@) または疑問符 (?) などの別のデリミタを使用する。たとえば、 <code>s@path@path/file@</code> は、 <code>path</code> を <code>path/file</code> に置換する。 <sup>d</sup>
指定したファイルへの書き込み	
<code>[addr1 [,addr2 ]]</code> <code>w file</code>	編集処理の実行中に、 <code>w</code> コマンドが出現した時点で指定された行範囲を指定されたファイル <sup>e</sup> に書き込む。

表 3-3: テキスト編集および移動のコマンド (続き)

コマンド	説明
行番号の表示	
<code>[addr1 ]=</code>	指定された行の行番号を出力に書き込む。
<sup>a</sup> 書き込まれるテキストが複数行からなる場合は、最終行以外の各行には、行末の改行文字の前に、バックスラッシュ (\) を付けなければならない。書き込まれるテキストは、その原因となった行に対して後続するコマンド行を、その行の削除も含めて実行したとしても、テキストは常に書き込まれる。書き込まれるテキストは、アドレスを照合するために走査されることも、後続する編集コマンドの影響を受けることもない。また、エディタのライン・カウンタにも影響を与えない。	
<sup>b</sup> アドレスが全く与えられない場合は、 <code>d</code> コマンドによりパターン・スペースの行はすべて削除される。つまり、中カッコの中の範囲を制御するコマンドのグループによって強制されていない限り、このコマンドによりファイルの全体の内容が削除される。	
<sup>c</sup> <code>r</code> コマンドとファイル名の間には、必ずスペースを 1 つ入れること。 <i>file</i> をアクセスできない場合、 <code>sed</code> は空のファイルを読み込んだものとして動作し、異常を表示しない。1 回の編集プロセスで、最大 10 個までのファイルを読み込みおよび書き込みを行うために指定することができる。	
<sup>d</sup> <code>s</code> コマンドのオプションのフラグについての詳細は、3.5 節を参照。	
<sup>e</sup> <code>w</code> コマンドとファイル名の間には、スペースを必ず 1 つ入れること。 <i>file</i> が存在する場合、そのファイルは上書きされる。存在しない場合、そのファイルは作成される。一回の編集プロセスで、最大 10 個までのファイルを読み込みおよび書き込みを行うために指定することができる。	

表 3-4 はバッファ操作作用の `sed` コマンドについて説明し、構文を示します。

表 3-4: バッファ操作作用コマンド

コマンド	説明
ホールド・エリアからのテキストの取り出し	
[addr1[,addr2]]g	ホールド・エリアの内容が存在する場合は、その内容を addr1 と addr2 で指定されたパターン・スペースにコピーする。g コマンドは、パターン・スペースの以前の内容を破壊する。G コマンドは、ホールドされたテキストをパターン・スペースの内容に追加する。改行文字を使用して、以前のテキストと追加されたテキストを分割する。
[addr1[,addr2]]G	
ホールド・エリアへのテキストの移動	
[addr1[,addr2]]h	パターン・スペースの指定された範囲をホールド・エリアにコピーする。h コマンドは、ホールド・エリアの以前の内容を破壊する。H コマンドは、パターン・スペースのテキストを、ホールド・エリアの内容に追加する。改行文字を使用して、以前のテキストと追加されたテキストを分割する。
[addr1[,addr2]]H	

表 3-4: バッファ操作作用コマンド (続き)

コマンド	説明
パターン・スペースとホールド・エリアの交換	
<code>[addr1 [,addr2]]x</code>	パターン・スペースの内容とホールド・エリアの内容を交換する。

表 3-5 はフロー制御の `sed` コマンドについて説明し、構文を示します。

表 3-5: フロー制御のコマンド

コマンド	説明
範囲の否定(「以外」)	
<code>[addr1 [,addr2]]!cmd</code>	感嘆符(!) を指定することによって、 <code>sed</code> は、入力ファイルの <code>addr1</code> と <code>addr2</code> で指定された範囲以外の部分に対して、感嘆符に続くコマンドを実行する。
コマンドのグループ化	
<code>[addr1[,addr2]]{ nested commands }</code>	左右の中カッコによって、実行するコマンドのグループを囲む。このコマンドのグループは、 <code>addr1</code> と <code>addr2</code> で指定された範囲に対して、1 つのセットとして適用される。最初のコマンドは、この表で示されているように、左の中カッコの次の行にあってよいし、中カッコと同じ行にあってよい。右の中カッコは 1 行に単独で書かなければならない。グループは他のグループ内にネストすることができる。
ラベル	
<code>:label</code>	分岐コマンドの宛先として使用される、編集コマンドのストリーム内の任意の場所を定義する。ラベルは最高 8 バイトまでの文字列である。編集ストリームの各ラベルは、独自のものでなければならない。詳細は、 <code>sed(1)</code> を参照。
分岐	
<code>label</code>	<code>label</code> に指定された編集スクリプトの特定の場所に分岐し、ラベルの次のコマンドを使用して、現在の入力行の処理を続行する。 <code>label</code> が省略されている場合は、 <code>b</code> コマンドは残りの編集スクリプトをバイパスし、新しい入力行を読み込み、スクリプトの先頭から編集を開始する。
条件付き分岐	

表 3-5: フロー制御のコマンド (続き)

コマンド	説明
<code>tlabel</code>	現在の入力行で、置換が正常終了した場合に、 <code>label</code> へ分岐する。それ以外は、コマンドは分岐しない。どちらの場合でも、このコマンドにより、置換が実行されたことを示すフラグがクリアされる。このフラグは、個々の新しい入力行の開始時点でもクリアされる。 <code>label</code> が省略されていて、分岐が必要とされる場合は、 <code>t</code> コマンドは残りの編集スクリプトをバイパスし、新しい入力行を読み込み、スクリプトの先頭から編集を開始する。
終了	
<code>[addr1]q</code>	現在の行を出力に書き込み、追加されたまたは読み込まれたテキストを出力に書き込んでから終了する、という規則正しい順で編集を終了する。

### 3.5 文字列の置換

`s` コマンドは、入力ファイルの指定された行で文字列を置換します。エディタは正規表現の `expr` を満たしている入力ファイルの文字列を見つけた場合は、その文字列を `string` で指定された文字セットと置換します。`string` 引数は正規表現ではなく、走査されることも解釈されることもありません。ただし、次のような例外があります。

- `string` 中のすべてのバックスラッシュ文字 (`\`) をエスケープしなければならない。`string` 中のスラッシュ文字 (`/`) の取り扱いについては、表 3-3 の説明を参照。
- 次の 2 つの特殊記号を、`string` 中で使用することができる。
  - アンパサンド (`&`)

`string` 内のこの記号は、入力行の中の `expr` と照合した文字列自身と交換されます。たとえば、`s/[Bb]oy/&s/` というコマンドを次の行に実行するとします。

The boy watched the game.

このコマンドは、入力行の中から `Boy` または `boy` のいずれかを見つけ出し、いずれを見つけた場合にも、`s` を追加して出力にコピーするように `sed` に指示します。このコマンドは `boy` を見つけたので、その文字列を変更して出力に移します。この結果は次のとおりです。

The boys watched the game.

– 反復式 ( $\backslash n$ )

数値  $n$  は 1 桁の数字です。 *string* 内のこのシンボルは、 *expr* の中の  $n$  番目のサブ正規表現と一致する入力行の文字列と置換されます。サブ正規表現は、バックスラッシュとカッコ ( $\backslash ($  および  $\backslash )$ ) によって区切られます。たとえば、コマンド `s/\(stu\)\(dy\)/\1r\2/` を次の行に適用する場合を考えてみましょう。

The study chair.

このコマンドは、入力行の中から `study` を見つけて、中間に `r` を挿入して、そのパターンを出力にコピーするように `sed` に指示します。この結果は次のとおりです。

The sturdy chair.

フラグを使用して、`s` コマンドの動作を次のように変更することができます。

- 通常、各行の範囲の中で、最初に照合した文字列だけが置換されるが、`g` (グローバル) フラグを使用すれば、`sed` はその範囲内のすべての行に対して、すべての照合した文字列を置換する。照合する文字列が同一である必要はない。正規表現 *expr* は、すべての可能な照合に対して、毎回評価される。
- `p` (プリント) フラグは、置換を実行した後に必ず指定された行を明示的に書き込むように `sed` に指示する。この書き込みは、`sed` の通常の動作を補足する。
- `w file` (書き込み) フラグは、置換を実行した後に必ず指定された行を指定したファイルに書き込むように `sed` に指示する。`w` フラグとファイル名の間には、1 つのスペースを入れなければならない。

これらのフラグは、1 度の `s` コマンドで使用できます。組み合わせる場合には、`w` フラグは、指定するすべてのフラグの最後のフラグとして指定されなければなりません。



## 入力言語解析プログラムおよびパーサの作成

プログラムが入力を受け取りその入力を処理する場合、処理を行う前にその入力を解析する手段が必要です。プログラム内の1つまたは複数のルーチンを使用して入力の解析を行うことも、入力をメイン・プログラムに渡す前に、フィルタ処理を行う別のプログラムを使用して入力の解析を行うこともできます。入力の複雑さに合わせて、入力インタフェースも複雑になります。複雑な入力を解析するには、かなり大量のコードが必要とされます。解析を行う際には、プログラムにとって有効なかたまりに入力を分割します。

この章では、入力インタフェースを開発するのに有効な次の2つのツールについて説明します。

- lex ツール

lex ツールは、一連の規則を使用して字句解析プログラムを生成します。これは、入力を解析し、数字、英字、または演算子などのカテゴリに分割するものです。

- yacc ツール

yacc ツールは、一連の規則を使用してパーサを生成します。パーサとは、字句解析プログラムによって識別されたカテゴリを使用して入力を解析し、入力の処理方法を決定するプログラムです。yacc ツールによって、左結合左復帰 (LALR) パーサが生成されます。

LALR 文法についての詳細は、『*Compilers: Principles, Techniques, and Tools*』<sup>1</sup>などのコンパイラに関する書籍を参照してください。

lex および yacc プログラムと、それらのプログラムが生成するプログラムとの混同を避けるため、この章では、lex と yacc をツールと呼びます。

この章では、次の事項について説明します。

---

<sup>1</sup> Alfred Aho, Ravi Sethi, and Jeffrey Ullman. 『*Compilers: Principles, Techniques, and Tools*』, Reading, MA, U.S.A.: Addison-Wesley Publishing Co., 1986.

- 字句解析プログラムの動作 (4.1 節)
- `lex` による字句解析プログラムの作成 (4.2 節)
- `lex` 仕様ファイル (4.3 節)
- 字句解析プログラムの生成 (4.4 節)
- `lex` と `yacc` の併用 (4.5 節)
- `yacc` によるパーサの作成 (4.6 節)
- 文法ファイル (4.7 節)
- パーサの動作 (4.8 節)
- デバッグ・モードの使用 (4.9 節)
- 簡易計算機プログラムの作成 (4.10 節)

## 4.1 字句解析プログラムの動作

`lex` が生成する字句解析プログラムは、決定性有限状態オートマトンと呼ばれます。これは、字句解析プログラムが存在可能な状態の数を限定し、また、次の入力文字を読み取り、解析した時点で字句解析プログラムが移行する状態を決定する規則を提供します。

コンパイルされた字句解析プログラムには、次の機能があります。

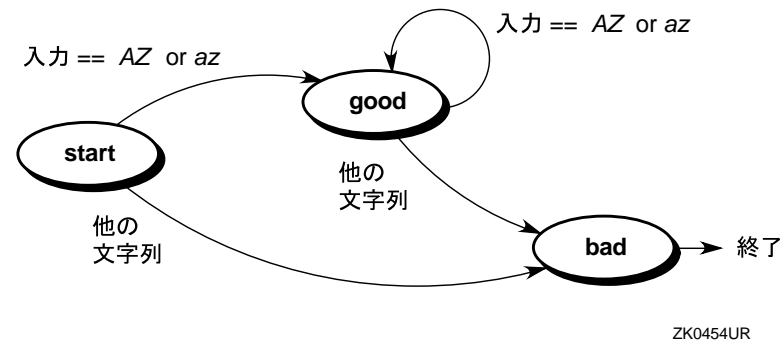
- 文字の入力ストリームを読み取る。
- 入力ストリームを出力ストリームにコピーする。
- `lex` 仕様ファイルの正規表現に照合するより小さな文字列に入力ストリームを分割する。
- 字句解析プログラムが認識した各正規表現にアクションを実行する。

アクションは `lex` 仕様ファイル中の C 言語プログラムのフラグメントです。アクション・フラグメントはそれ自体で完結している必要はなく、サブルーチンまたは他のアクションを呼び出すことができます。

図 4-1 に、`start`、`good`、および `bad` という、3 つの状態を持つ簡単な字句解析プログラムを示します。字句解析プログラムは、文字の入力ストリームを読み取ります。まず、`start` の状態で開始され、最初の文字を受け取ると、規則と比較します。文字が英文字である (規則と適合している) 場合、プログラムは `good` 状態に移行し、英文字でない場合には、`bad` 状

態に移行します。プログラムは条件に適合しない文字を見つけるまでは，good 状態で，bad 状態に移行すると終了します。

図 4-1: 簡単な有限状態のモデル



オートマトンは，生成された字句解析プログラムによる入力ストリームの 1 つまたは 2 つ以上の文字の先読みを許可します。たとえば，lex 仕様ファイルが，文字列 ab を見つける規則と，文字列 abcdefg を探す別の規則を定義しているとします。abcdefgh という文字列が入力されると，字句解析プログラムは abcdefg と照合を行うために十分な文字 (abcdef) を読み取ります。g ではなく h があるために abcdefg との照合が無効になると，字句解析プログラムは，ab を探す規則に戻ります。入力された最初の 2 文字が ab と照合するので，その規則に指定されたアクションを実行し，cdefh という残りの入力と照合する規則を探します。

## 4.2 lex による字句解析プログラムの作成

lex ツールを使用して，文字ストリーム入力を受け取りその入力をプログラムアクションに解釈する C 言語字句解析プログラムを作成することができます。lex を使用するには，次の記述が含まれた仕様ファイルを作成しなければなりません。

- 正規表現 -- 生成された字句解析プログラムによって識別される文字パターン
- アクション文 -- 生成された字句解析プログラムが，認識する正規表現に対して実行するアクションを定義する C 言語プログラム・フラグメント

実際に仕様ファイルに記述するフォーマットと論理については，4.3 節を参照してください。

lex ツールは、仕様ファイル内の情報を使用して字句解析プログラムを生成します。このツールによって生成された解析プログラムは `yy.lex.c` という名前になります。`yy.lex.c` プログラムには、仕様ファイルから生成される解析コードとともに標準の関数セットが含まれています。解析コードは、`yylex` 関数に含まれています。lex によって生成された字句解析プログラムは、簡単な文法構造と正規表現を認識します。次のコマンドを使用して、簡単な lex 解析プログラムをコンパイルすることができます。

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

`-ll` オプションは、コンパイラに lex 関数ライブラリを使用するように指定します。このコマンドにより実行可能な字句解析プログラムが生成されます。プログラムに複雑な文法規則が使用されているか、または文法規則がまったく使用されていない場合は、入力の適切な処理を確実にするため (lex と yacc ツールを併用して) パーサを作成してください。詳細については、4.6 節を参照してください。

`yy.lex.c` 出力ファイルは、lex ライブラリ関数をサポートする C コンパイラを持つシステムならどのシステムにでも移植することができます。

## 4.3 lex 仕様ファイル

lex 仕様ファイルのフォーマットは次のとおりです。

```
[ { definitions } ]
%%
[ { rules } ]
[ %%
{ user subroutines } ]
```

最初の 2 つのパーセント記号 (%%) は、規則の始まりを示すものです。仕様ファイルのそれ以外の部分は、すべてオプションです。最小の lex 仕様ファイルには、定義、規則、およびユーザ・サブルーチンは含まれていません。パターン照合のアクションが指定されていない場合は、字句解析プログラムは、入力パターンを変更することなく出力パターンにコピーします。したがって、この最小の仕様ファイルは、すべての入力をそのまま出力にコピーする解析プログラムを生成します。

この節では、次の事項について説明します。

- 置換文字列の定義 (4.3.1 項)
- 規則 (4.3.2 項)

- 標準の入出力ルーチンの使用または変更 (4.3.3 項)
- ファイルの終わりの処理 (4.3.4 項)
- 生成されたプログラムへのコードの引き渡し (4.3.5 項)
- 開始条件 (4.3.6 項)

#### 4.3.1 置換文字列の定義

lex 仕様ファイル中の最初の 2 つのパーセント記号の前に、文字列マクロを定義することができます。lex ツールは、字句解析プログラムを生成する際に、そこで定義されたマクロを展開します。この部の行の中で、カラム 1 から開始され、かつ `%{ デリミタおよび % }` デリミタに囲まれていない行は、lex 置換文字列を定義しています。置換文字列の定義は、一般的に次のようなフォーマットで記されます。

*name translation*

*name* と *translation* の要素は、少なくとも 1 つの空白またはタブで区切り、*name* は英文字で開始します。仕様ファイルの規則の部分で、中カッコ (`{ }`) で囲まれた文字列 *name* を発見すると、lex は、*name* を *translation* で定義された文字列に変更して、中カッコを削除します。

たとえば、`D` と `E` の名前を指定するには、仕様ファイルの中で最初の `%%` デリミタの前に次の定義を指定します。

```
D          [0-9]
E          [DEde] [-+] {D}+
```

上記の定義を規則部で使用して、整数と実数の識別をより簡潔にすることができます。

```
{D}+          printf("integer");
{D}+"." {D}* ({E})? |
{D}*"." {D}+ ({E})? |
{D}+{E}       printf("real");
```

また、定義部には次の項目を含めることもできます。

- 文字セット・テーブル (4.3.3 項を参照)
- 開始条件のリスト (4.3.6 項を参照)
- より大きなソース・プログラムと適応させるための配列サイズの変更

### 4.3.2 規則

仕様ファイルの規則部には、lex が生成する字句解析プログラムの制御を定義する決定が含まれています。その規則は、2 カラムのテーブルの形になっています。テーブルの左のカラムには正規表現が、テーブルの右のカラムには正規表現に対応するアクションが記述されています。アクションは C 言語プログラム・フラグメントで、最も単純なものとしてセミコロンだけ (nul 文) を指定することも、また必要に応じてより複雑なものを指定することもできます。lex が作成する字句解析プログラムには、正規表現とアクションの両方が含まれています。正規表現に照合した文字列を検索すると、それに対応するアクションを実行します。

たとえば、integer という文字列を探し、発見した場合はメッセージをプリントする字句解析プログラムを作成するには、次のような規則を定義します。

```
integer      printf ("found keyword integer");
```

上記の例では、メッセージの文字列をプリントするために、C 言語ライブラリ関数の printf を使用しています。規則の最初の空白またはタブは、正規表現の終わりを示します。1 つのアクションに 1 つの文だけを使用する場合は、正規表現 (上記の例では integer) の右に文を記述します。2 つ以上の文を使用する場合、または文が 2 行以上にわたる場合には、C 言語プログラムと同様に、アクションを中カッコで囲ってください。たとえば、次のように記述します。

```
integer      {
              printf ("found keyword integer");
              hits++;
              }
```

ファイル内の単語を英国式スペリングから米国式スペリングに変更する字句解析プログラムには、次のような規則が記述されている仕様ファイルがあります。

```
colour      printf("color");
mechanise   printf("mechanize");
petrol      printf("gas");
```

ただし、上記の仕様ファイルは完全なものではありません。petroleum という単語が、gaseum に変更されるためです。

### 4.3.2.1 正規表現

標準の拡張正規表現に加えて、lex は表 4-1 に示す正規表現を解釈します。

表 4-1: lex の正規表現演算子

演算子	名前	解説
<code>{name}</code>	中カッコ	中カッコで囲まれた変数は、仕様ファイルの最初に定義されている文字列を意味します。たとえば、 <code>{digit}</code> は <code>digit</code> として定義されている文字列と置き換えられます。カウント式と混同しないください。ともに <code>lex</code> で識別されます。
<code>" "</code>	引用符	引用符でリテラル文字列を囲むとテキスト文字列として解釈します。たとえば、 <code>"\$"</code> と指定すると、 <code>lex</code> はドル記号を演算子として解釈しません。引用符は、文字列の中で部分的に使用することもできます。たとえば、 <code>"abc++"</code> と <code>abc"++"</code> は、どちらもリテラル文字列 <code>abc++</code> を照合します。
<code>a/&lt;b</code>	スラッシュ	2 番目の文字 ( <code>b</code> ) が最初の文字 ( <code>a</code> ) の直後に続く場合のみ、最初の文字 ( <code>a</code> ) を照合します。たとえば、 <code>dog/cat</code> は、 <code>cat</code> が <code>dog</code> の直後に続いている場合のみ、 <code>dog</code> を照合します。
<code>&lt;x&gt;</code>	山カッコ	開始条件を囲みます。指定された開始条件 <code>&lt;x&gt;</code> に字句解析プログラムが適合する場合にのみ、アクションを実行します。開始条件 <code>ONE</code> が行頭にあるという条件を意味する場合、 <code>&lt;ONE&gt;</code> は山形記号 (^) 演算子と同じ意味になります。
<code>\n</code>	改行	正規表現の中では実際の改行文字は使用しません。基本正規表現の <code>\n</code> と混同しないよう注意してください。
<code>\t</code>	タブ	リテラルのタブ文字 (16 進数の 09) を照合します。
<code>\b</code>	バックスペース	リテラルのバックスペース文字 (16 進数の 08) を照合します。
<code>\\</code>	バックスラッシュ	リテラルのバックスラッシュ文字を照合します。
<code>\digits</code>	Digits	エンコードが 3 桁の 8 進数で表される文字です。
<code>\xdigits</code>	xDigits	エンコードが 16 進の整数で表される文字です。

通常、ホワイト・スペース (空白またはタブ) は、正規表現の終わりとそれに関連するアクションの始まりを区切るために使用されます。式に空白やタブを入れる場合には、それらを引用符 (" ") で囲みます。大カッコ ([ ]) で囲まれていない正規表現の中の空白にはすべて引用符を使用してください。

#### 4.3.2.2 照合規則

仕様ファイルの規則部内にある 2 つ以上の正規表現が現在の入力と照合する場合には、字句解析プログラムは次の基準を使用して、適用する規則を決定します。

1. 一致する文字数が最も多い文字列に適用する。
2. 一致する文字数が同じ場合は、最初に出現した規則を適用する。

たとえば、次のような規則がある場合を考えてみましょう。

```
integer      printf("found int keyword");  
[a-z]+      printf("found identifier");
```

上記の順に規則が適用され、`integers` という語が入力された場合、`[a-z]+` ワードの中の 8 文字と照合しますが、一方 `integer` は、7 文字にしか照合しないために、解析プログラムは、入力を `"identifier"` とみなします。しかし、入力が `integer` の場合には、両方の規則とも照合します。この場合、`keyword` の規則が先に出現するために、`lex` はこれを選択します。`int` のような短い入力は、`integer` という文字列には照合しないので、`lex` では `"identifier"` の規則を選択します。

##### 4.3.2.2.1 文字列を照合するためのワイルドカード文字の使用

字句解析プログラムでは、照合するもののうち最長の文字列が最初を選択されるために、意図した目的以上の影響をもたらす正規表現を使用しないように注意しなければなりません。たとえば、ピリオドの後にアスタリスクを指定してアポストロフィで囲んだ (`'.*'`) 正規表現は、アポストロフィで囲まれるすべての文字列を認識するために適切であるように見えますが、字句解析プログラムは、可能な限り長い文字列に照合させようと、さらに先を探し、離れたアポストロフィを検出します。次は、その例です。

```
'first' quoted string here, 'second' here
```

このような入力があると、字句解析プログラムは、次の文字列を照合します。

```
'first' quoted string here, 'second'
```



ピリオド演算子は改行文字を照合しないので、それほど広範囲に影響を与えることはありません。.\* のような式は、現在の行で停止します。次のような式を使用して、このようなアクションを予防しようとししないでください。

```
[.\n]+
```

このような式を指定すると、字句解析プログラムは、入力ファイルの全体を読み取り、内部バッファのオーバーフローが発生します。

次の規則は、前述のテキスト例から引用符で囲まれたより小さな範囲の文字列 'first' および 'second' を検索します。

```
'[^'\n]*''
```

この規則では、'first' を照合した後、検索を停止します。検索されるアポストロフィには、別のアポストロフィまたは改行文字を除く任意の文字数が続き、次に、2 番目のアポストロフィが続いているためです。次に、解析プログラムは再び該当する文字列を探し、'second' を検出します。また、この文字列は、引用符の付いた空の式 ( ' ' ) も照合します。

#### 4.3.2.2.2 文字列内の文字列の検索

字句解析プログラムは、通常、入力ストリームを区切ります。それぞれの式と照合する文字列を、可能な限りすべて探すわけではありません。文字はそれぞれ、1 回だけカウントされます。たとえば、入力テキストで 'she' と 'he' と照合するものをカウントするためには、次の規則が考えられます。

```
she      s++;
he       h++;
\n       ;
.        ;
```

最後の 2 つの規則は検索したい 2 つの文字列を除いたすべてを無視します。ただし、'she' には 'he' が含まれているため、字句解析プログラムは、'she' に含まれる 'he' は認識しません。

特殊なアクション REJECT は、この動作を抑止するためのものです。この指示は解析プログラムに、REJECT を含む規則を実行させ、次の規則を実行する前に、入力ポインタの位置を最初の規則が実行される前に戻します。たとえば 'she' に含まれる 'he' と照合するものをカウントするためには、次の規則を使用します。

```
she      {s++; REJECT;}
he       {h++; REJECT;}
\n       ;
.        ;
```

字句解析プログラムは、'she' をカウントした後に、入力ストリームをリジェクトして、'she' に含まれる 'he' をカウントします。この例では、'she' は 'he' を含んでいますが、その逆はあり得ないため、'he' での REJECT アクションは省略することができます。しかし他の場合、たとえば、ワイルドカードの正規表現と照合した場合、両方のクラスに属する入力文字を決定することは、困難になります。

REJECT が有効なのは、通常、入力ストリームを区切ることが目的ではなく、むしろオーバーラップしたり、相互に含む合うことができるような、入力の中の項目のインスタンスをすべて検出することが目的である場合です。

#### 4.3.2.3 アクション

字句解析プログラムが、仕様ファイルの規則部中の正規表現の中の 1 つを照合した場合は、その正規表現に対応するアクションを実行します。入力ストリームの中の文字列すべてを照合する規則を使用しない限り、字句解析プログラムは、入力を標準出力にコピーします。したがって、ただ入力を出力にコピーするだけの規則は作成しないでください。規則に含まれない条件を見つけるために、この省略時の出力を使用してください。

lex で生成された解析プログラムを使用して yacc が生成するパーサのための入力を処理する場合は、入力文字列のすべてを照合する規則を作成してください。その規則は、yacc が解釈できるような出力を生成しなければなりません。yacc および lex を使用する際の詳細については、4.5 節を参照してください。

##### 4.3.2.3.1 ノル・アクション

ある正規表現に対応する入力を無視するには、C 言語の空文であるセミコロン (;) をアクションとして使用してください。たとえば、次のようにします。

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

この規則では、3 つのスペーシング文字 (空白、タブ、および改行文字) を無視しています。

##### 4.3.2.3.2 複数の式に対する同一アクションの使用

いくつかの異なった式に対して同じアクションを使用するには、最後の式以外の各式に対して 1 つのアクションが縦線文字 (|) のみで構成されている連続した規則を作成します。それらの規則の最後の式には、通常どおり、そのアクションを指定します。縦線文字は、それを含む規則に対するアクショ

ンが、次の規則に対するアクションと同じであることを示しています。たとえば、空白、タブ、および改行文字 (4.3.2.3.1 項を参照) を無視するには、次のような規則を使用します。

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

この例の特殊文字シーケンス (\n と \t) を囲む引用符は必要ではありません。

#### 4.3.2.3.3 一致した文字列のプリント

仕様ファイルの規則部の中の正規表現に、どのテキストが一致するかを検出するには、その式に対するアクションのうちの 1 つとして、C 言語の `printf` 関数を入力します。字句解析プログラムが入力ストリーム中から照合する文字列を検出すると、解析プログラムはその照合した文字列を、`yytext` と呼ばれる外部文字配列に加えます。照合した文字列をプリントするには、次のような規則を使用します。

```
[a-z]+      printf("%s", yytext);
```

この方法で出力をプリントすることはよくあります。仕様ファイルの定義部にマクロとして、この `printf` 文などの式を定義することもできます。このアクションが `ECHO` として定義されている場合には、規則部のエントリは、次のようになります。

```
[a-z]+      ECHO;
```

マクロの定義についての詳細は、4.3.1 項を参照してください。

#### 4.3.2.3.4 一致した文字列の長さの検出

字句解析プログラムに特定の正規表現に一致する文字数を検出させるには、外部変数 `yylen` を使用します。たとえば、次の規則は、入力ワード中のワード数および文字数の両方をカウントします。

```
[a-zA-Z]+    {words++; chars += yylen;}
```

このアクションによって、一致したワードの中の文字数が合計され、その値を `chars` 変数に与えます。

次の式では、一致した文字列の中から、最後の文字を検出します。

```
yytext[yylen-1]
```

#### 4.3.2.3.5 入力の追加

字句解析プログラムでは、規則ファイルの中の正規表現と完全に照合する前に、入力が不足することがあります。この場合、その規則に対するアクションの中に、lex 関数の `yymore` への呼び出しを入力します。通常は、入力ストリームからの次の文字列が、`yytext` の現在のエントリに重ね書きされます。`yymore` アクションによって、次の文字列は、入力ストリームから `yytext` の現在のエントリの最後に追加されます。たとえば、次の構文を含む文字列を考えてみてください。

- 文字列は、引用符 (" ") で囲まれる任意の文字の集合である。
- バックスラッシュ (\) が次の文字をエスケープすることによって、その文字は文字列の一部とみなされる。たとえば、バックスラッシュと引用符の組み合わせ (\") は、引用符がその文字列の終わりのデリミタではなく、文字列の 1 部になっていることを示している。

次の規則によって、このような字句特性を処理することができます。

```
\"[^"]*" {
    if (yytext[yyleng-1] == '\\')
        yymore();
    else
        ... normal user processing
}
```

この字句解析プログラムによって、`"abc\def"` 文字列 (引用符はこのとおり) が検索されると、まず、最初の 5 文字 `"abc\"` を照合します。バックスラッシュによって、`yymore` の呼び出しが行われ、文字列の `"def"` の部分が `"abc"` の後に追加されます。`"normal user processing"` というラベルの付いたアクション・コードの部分では、文字列を終了する引用符を処理しなければなりません。

#### 4.3.2.3.6 入力へ文字を戻す

字句解析プログラムは、正規表現との照合が完了している文字すべてを必要としない場合があります。または、再び別の照合をチェックするために、照合した文字を入力ストリームに戻す必要が生じる場合もあります。文字を入力ストリームに戻すには、`yylless(n)` 呼び出しを使用します。変数 `n` には、現在の文字列で戻さない文字数を入れます。ストリームの中の `n` 番目を超えた文字は、入力ストリームに戻ります。この関数によって、スラッシュ演算子 (/) と同様に先読みを行います。が、`yylless` には先読みに対してより強い制御機能があります。`yylless(0)` を使用した場合、`REJECT` と同じ機能を持ちます。

2 回以上テキストを処理するには、`yylless` 関数を使用してください。たとえば、`x=-a` などの C 言語の式は複数の意味に解釈されあいまいです。`x = -a` であるとも、または `x -= a` として計算される、一般的には使用されない `x = x - a` という式とも解釈されます。このあいまいな式を `x = -a` として扱い、警告メッセージをプリントするには、次のような規則を使用してください。

```
--[a-zA-Z]      {
    printf("Operator (=-) ambiguous\n");
    yyless(yyleng-1);
    ... action for -=...
}
```

### 4.3.3 標準の入出力ルーチンの使用または変更

`lex` プログラムは、使用する字句解析プログラム用の入出力 (I/O) ルーチンを提供します。仕様ファイルの中の C コード・フラグメントに、次のようなルーチンへの呼び出しを含めてください。

- `input --` 次の入力文字を返す。
- `output(c) --` 出力に文字 `c` を書き込む。
- `unput(c) --` 後で `input` によって読み取るために、入力ストリームに文字 `c` を戻す。

上記のルーチンは、マクロ定義として実装されています。サブルーチン部の中で、ユーザ自身のコードを同名のルーチンとして記述することによって、マクロ指定を変更することができます。これらのルーチンによって、外部ファイルと内部の文字の関係が定義されています。それらの定義を変更する場合は、すべて同じ方法で変更してください。変更する際は、次の規則に従ってください。

- すべてのルーチンは同じ文字セットを使用する。
- 入力ルーチンでは、0 の値を返して、ファイルの終端を示す。

ユーザ自身のコードを記述する場合は、仕様ファイルの定義部の中で、ユーザの定義するコードの前に、これらのマクロ定義を削除する必要があります。

```
%{
#undef      input
#undef      unput
#undef      output
}%
```

---

### 注意

---

unput と input の関係を変更すると、先読み機能が動作しません。

---

lex で生成した字句解析プログラムを、標準入力から標準出力に引き渡すための簡易変換プログラムおよびまたは認識プログラムとして使用している場合は、lex ライブラリ (libl.a) を使用すれば、フレームワークに書き込まないですみます。このライブラリには、main ルーチンがあり、yylex 関数の呼び出しを行います。標準の lex ライブラリによって、字句解析プログラムは、最大 100 までの文字をバックアップできます。

空文字 (16 進の 00) を含む入力ファイルを読み取る必要がある場合は、input ルーチンの別のバージョンを作成しなければなりません。input の標準のバージョンでは、空を読み取ると 0 の値が返され、この値はファイルの最後を示すものとして解釈されます。

lex が生成する字句解析プログラムは、input ルーチン、output ルーチン、および unput ルーチンを使用して、文字 I/O を処理します。したがって、yytext で値を返すため、解析プログラムでは、これらのルーチンが使用する文字表現を使用します。ところが、各文字は、内部では小さい整数で表されています。標準ライブラリについては、この整数は、コンピュータが文字を表すために使用するビット・パターンの値です。

通常、a という文字は、文字定数 a と同じ形式で表されます。別の I/O ルーチンによって、この解釈を変更する場合は、仕様ファイルの定義部に変換テーブルを含めなければなりません。変換テーブルでは、始めの行と終わりの行が %T キーワードのみで、次のような書式の行が含まれます。

*[[ integer] { character string}]*

次の例では、英字の A および数字の 0 (ゼロ) と、それらの標準の値を関連づけるテーブル・エントリを示しています。

```
%T
{65} {A}
{48} {0}
%T
```

#### 4.3.4 ファイルの終わりの処理

字句解析プログラムがファイルの終わりに到達すると、`yywrap` というライブラリ・ルーチン呼び出します。このルーチンは、1 の値を返して、字句解析プログラムに通常のラップアップ (処理終了時に行われる動作) を続けるように示します。しかし、解析プログラムが2 つ以上のソースから入力を受け取る場合は、`yywrap` 関数を変更しなければなりません。新しい関数は、新しい入力を得て、0 の値を字句解析プログラムに返さなければなりません。0 のリターン値は、プログラムが処理し続けることを示します。

コマンド行で複数ファイルを指定した場合は、ファイルの終端の処理に関しては、1 つの入力ファイルとして扱われます。

また、`yywrap` には、特に要約レポートとテーブルをプリントするコードを含めることもできます。`yywrap` 関数は、`yylex` に強制的に入力の終わりを認識させる唯一の方法です。

#### 4.3.5 生成されたプログラムへのコードの引き渡し

仕様ファイルの定義部または規則部のいずれにも変数を定義することができます。仕様ファイル処理すると、`lex` は、指定ファイル中の文を字句解析プログラムに変更します。`lex` が解釈できない仕様ファイルの行はすべて、変更されずに字句解析プログラムに渡されます。次の4つのエントリのタイプは、変更されずに字句解析プログラムに渡されます。

- `lex` のいずれの規則にも含まれない、空白またはタブで始まる行は、字句解析プログラムにコピーされる。仕様ファイル中の最初の2つのパーセント記号(%%)の前にこのエントリが出現した場合、エントリは、コードの関数に対しても外部的なものとなる。最初の%%の後にエントリが出現する場合には、変数を定義するためのC言語プログラム・フラグメントになる。仕様ファイル中では、最初の`lex`規則の前に、それらの文を定義しなければならない。
- プログラムのコメントであり、空白またはタブで始まる行は、生成された字句解析プログラムでもコメントとして含まれる。コメントは、C言語のコメントのフォーマットでなければならない。
- `%{` と `%}` だけの行の間にある行はすべて、字句解析プログラムにコピーされる。`%{` と `%}` の記号はコピーされない。カラム1で始まらなければならないプリプロセッサ文を入力したり、プログラム文としては認識されない行をコピーする場合には、このフォーマットを使用する。

- 3 番目の %% デリミタの後に出現する行はいずれも、フォーマットの制限条項なしで字句解析プログラムにコピーされる。

#### 4.3.6 開始条件

どんな規則でも、開始条件に指定することができます。字句解析プログラムは、開始条件に含まれている場合にのみ、その規則を認識します。現在の開始条件は、常時変更可能です。

仕様ファイルの定義部の中の開始条件を定義するには、次のフォーマットの行を使用します。

**% Start** [*name1*] [[*name2* ...]]

*name1* 記号および *name2* 記号は、条件を表します。条件の数には制限がなく、どのような順番でもかまいません。Start は、s または S のいずれかに省略することができます。開始条件の名前には、C の予約語や、変数、フィールドなどの名前として定義されたものを使用することはできません。

仕様ファイルの規則部の中で開始条件を使用する場合は、その規則の最初に、山カッコ (< >) で開始条件の名前を囲みます。次のフォーマットで開始条件を含む規則を定義します。

[<*name1*] [[*name2* ...]] [> *expression*]

字句解析プログラムは、名前のうちのいずれかと対応する条件にある場合にのみ、*expression* を認識します。特定の開始条件に lex を移行させるには、次のアクション文を規則のアクション部分で実行してください。

```
BEGIN name;
```

開始条件はこの文によって *name* に移行します。通常の状態に戻すには、次のアクションを使用してください。

```
BEGIN 0;
```

前述の構文ダイアグラムに示したように、複数の開始条件で、1 つの規則がアクティブになります。たとえば、次のようにします。

```
<start1,start2,start3> [0-9]+ printf("integer");
```

この規則では、3 つの名前の開始条件のうちいずれかで、整数を検出した場合にのみ、integer をプリントします。開始条件で始まらない規則は常にアクティブです。



## 4.4 字句解析プログラムの生成

lex に基づく字句解析プログラムの生成は、次の 2 つの手順で行います。

1. lex を実行して仕様ファイルを C 言語プログラムに変更する。その結果生成されたプログラムは、lex.yy.c というファイルに存在する。
2. lex.yy.c を cc -ll コマンドで処理して、プログラムにコンパイルし、それを lex サブルーチンのライブラリとリンクする。結果として生成された実行可能プログラムは、a.out という名前になる。

たとえば、lex 仕様ファイルが lextest である場合は、次のコマンドを入力します。

```
% lex lextest
% cc lex.yy.c -ll
```

省略時の設定である lex I/O ルーチンは、C 言語標準ライブラリを使用しますが、lex が生成する字句解析プログラムでは必須ではありません。I/O ルーチンをライブラリで使わないようにするには、input、output、および unput のルーチンの別のコピーをいれます。詳細については、4.3.3 項を参照してください。

lex コマンドのオプションについては、表 4-2 を参照してください。

表 4-2: lex コマンドのオプション

オプション	説明
-n	有限状態マシン用にユーザがテーブル・サイズを設定した場合に、省略時の値として生成される統計値の要約を出力しない。状態マシンの指定についての詳細は、lex(1) を参照。
-t	生成された字句解析プログラム・コードを lex.yy.c ファイルではなく、標準出力に書き込む。
-v	有限状態マシンの一般的な統計値の要約を 1 行表示する。

lex は、中間ファイルおよび出力ファイルに決まった名前を使用するため、所定のディレクトリ内には、lex で生成されたプログラムが 1 つしか存在できません。ただし、-t オプションを使用して代わりのファイル名を指定した場合は、この限りではありません。

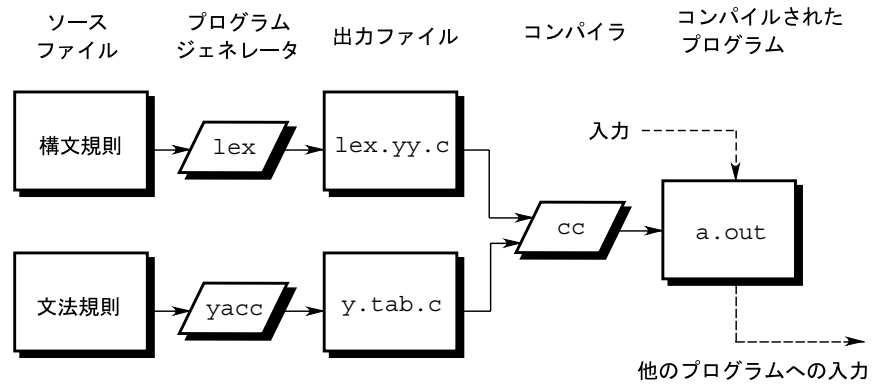
## 4.5 yacc と lex の併用

lex ツールは、単独で使用された場合、簡単な 1 ワードの入力を認識するか、または統計の入力を受け取る字句解析プログラムを作成します。また、yacc などのパーサ生成プログラムとあわせて lex を使用することもできます。yacc ツールは、パーサという、複数ワードの入力の構造を解析するプログラムを生成します。このパーサ・プログラムは、lex が生成する字句解析プログラムと組み合わせても効率よく動作します。字句解析プログラムは、正規表現だけを認識し、それらをトークンと呼ばれる文字パッケージの形式に分離します。

トークンは、パーサまたは字句解析プログラムのいずれかで定義される、最小の独立した意味単位のことです。トークンには、データ、言語キーワード、識別子、または他の言語構文のパーツが含まれます。トークンは、どのような文字列でもかまいません。トークンは、ワードの 1 部分またはそのワード全体、あるいは連続した複数のワードでもかまいません。yacc ツールによって、分脈に関係なく多種多様な文法を認識するパーサが生成されます。このようなパーサが入力トークンを認識するために、lex で生成された字句解析プログラムなどのプリプロセッサが必要になります。

lex で生成された字句解析プログラムが、yacc で生成されたパーサ用のプリプロセッサとして使用される場合は、字句解析プログラムが入力ストリームを区切ります。パーサは、その結果として生成された断片に構造を割り当てます。lex および yacc がプログラムを生成する方式、および、そのプログラムが組み合わされて動作する方式については図 4-2 を参照してください。lex や yacc によって生成されたプログラムと他のプログラムを併用することも可能です。

図 4-2: lex および yacc による入力パーサの生成



ZK0455UR

パーサ・プログラムは、そのプログラム用のプリプロセッサ (字句解析機能) に `yyllex` という名前を付けなければなりません。この名前は、生成する字句解析プログラムの解析コードに `lex` が指定する名前です。字句解析プログラムを単独で使用する場合には、`lex` ライブラリの省略時の設定である `main` プログラムが、`yyllex` ルーチン呼び出します。しかし、`yacc` が生成するパーサがロードされてそのパーサの `main` プログラムが使用される場合は、パーサによって `yyllex` が呼び出されます。この場合、各 `lex` の規則は、次に示した行で終わらなければなりません。ここでは、該当するトークン値が返されます。

```
return(token);
```

`yacc` で使用されるトークン名を検出するためには、`yacc` 文法ファイルの最後の部分に、次の行を入れることによって、パーサの一部 (`yacc` 出力ファイル) として字句解析プログラム (`lex` 出力ファイル) をコンパイルします。

```
#include lex.yy.c
```

別の方法として、`yacc` 出力 (`y.tab.h` ファイル) を `lex` プログラム仕様ファイルに含めることができます。これによって、`y.tab.h` で定義されているトークン名を使用することもできます。たとえば、文法ファイルが `good` という名前、仕様ファイルが `better` という名前の場合は、次のコマンド・シーケンスによって、最終プログラムが作成されます。

```
% yacc good
% lex better
% cc y.tab.c -ly -ll
```

yacc パーサを呼び出す main プログラムを作成するには、yacc ライブラリ (上記の例の `-ly`) を、lex ライブラリの前にロードしなければなりません。lex と yacc プログラムは、どちらを先に生成してもかまいません。

## 4.6 yacc によるパーサの作成

yacc を使用してパーサを生成するには、入力データ・ストリーム、およびパーサがそのデータに対して行う処理を指定する文法ファイルを記述する必要があります。文法ファイルには、入力の構造、規則が認識される際に呼び出されるコード、および基本入力を行うルーチンが含まれています。

yacc ツールは、文法ファイルの情報を使用することによって、yyparse を生成します。このプログラムによって、入力処理が制御されます。このパーサは、yylex 入力ルーチン (字句解析プログラム) を呼び出して、入力ストリームからトークンを取り出します。このパーサは、文法ファイルの構造規則に従って、これらのトークンを組織化します。この構造規則は、文法規則と呼ばれます。パーサが文法規則を認識すると、パーサは、その規則に与えられたユーザ・コード (アクション) を実行します。アクションは、値を返し、また、他のアクションによって返された値を使用します。

yacc が認識および使用する仕様以外に、文法ファイルには次の関数を含めることができます。

- `main` -- 最小限の `yyparse` 関数の呼び出しが含まれる C 言語関数。  
`yyparse` 関数は、yacc によって生成される。この関数の限定は、yacc ライブラリに含まれる。
- `yyerror` -- パーサの動作中に発生するエラーを処理する C 言語関数。  
この関数の簡易版は、yacc ライブラリに含まれる。
- `yylex` -- 入力ストリームについて字句解析を実行し、必要に応じて値を付けたトークンをパーサに渡す C 言語関数。この関数では、読み取られたトークンの種類を表す整数が返されなければなりません。この整数は、トークン番号と呼ばれています。さらに、値がトークンに対応する場合、字句解析プログラムは、その値を外部変数 `yyval` に割り当てなければなりません。トークン番号についての詳細は、4.7.1.3 項を参照してください。yacc で生成されたパーサと組み合わせて、正常に動作する字句解析プログラムを作成するには、lex ツールを使用してください。詳細は、4.3 節を参照してください。

yacc ツールは、文法ファイル进行处理して `y.tab.c` という、C 言語関数およびデータを含むファイルを生成します。cc コマンドを使用してコンパイルすると、この関数は、`yyparse` という結合された関数を形成します。この `yyparse` 関数は、入力トークンを得るために、`yylex` という字句解析プログラムを呼び出します。この解析プログラムは、パーサがエラーを検出するか、または解析プログラムが演算の終わりを示す `endmarker` トークンを返すまで入力を与え続けます。エラーが発生して、`yyparse` を復旧できない場合、`yyparse` は、`main` 関数に 1 の値を返します。`endmarker` トークンが検出された場合には、`yyparse` は `main` 関数に 0 の値を返します。

アクション・コードおよび他のサブルーチンを記述するには、C プログラミング言語を使用してください。yacc プログラムは、文法ファイル内で多くの C 言語構文規約を使用します。

#### 4.6.1 main および yyerror の関数

文法ファイルの中には、`main` および `yyerror` という関数ルーチンが含まれていなければなりません。yacc の初期導入作業を少なくするために、yacc ライブラリには、`main` と `yyerror` ルーチンの簡易版が備えられています。ロードまたは cc コマンドで `-ly` オプションを使用することによって、これらのルーチンをインクルードすることができます。`main` ライブラリ関数のソース・コードは次のとおりです。

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

`yyerror` ライブラリ関数のソース・コードは、次のとおりです。

```
#include <stdio.h>

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

`yyerror` の引数には、通常、文字列構文エラーなどのエラー・メッセージを含む文字列を指定します。

これらは、かなり制約のあるプログラムです。入力行番号を保持しておき、構文エラーの検出時にエラー・メッセージとともにそれをプリントするなど、これらのルーチンをより精巧なものにしてください。また、外部整数変

数 `yychar` の値も使用することができます。この変数には、エラーが検出されたときの、先読みトークン番号が含まれます。

### 4.6.2 `yylex` 関数

指定する `yylex` プログラムの入力ルーチンは、次の処理が実行できなければなりません。

- 入力ストリームの読み取り
- 入力ストリーム中の基本パターンの認識
- パターンを識別したトークンとともに `yyparse` にパターンを渡す

トークンは、入力ルーチンによって送られたパターンを `yyparse` に通知する記号または名前です。記号は、次の 2 つのクラスのいずれかにあります。

- 終端記号 -- 文法の基本単位を表すために、`yylex` により返される値
- 非終端記号 -- より複雑な順序、または終端記号の集合体を記述するために、`yacc` 文法で使用される複合記号

たとえば、字句解析プログラムが、数字、名前、および演算子のいずれでも認識する場合は、これらの要素が終端記号とみなされます。`yacc` 文法が認識する非終端記号は、`EXPR`、`TERM`、および `FACTOR` などの要素です。入力ルーチンが入力ストリームを `WORD`、`NUMBER`、および `PUNCTUATION` のトークンに分ける例を、“I have 9 turkeys.” という入力文の例で考えてみましょう。解析プログラムは、次の文字列とトークンをパーサに渡します。

文字列	トークン
I	WORD
have	WORD
9	NUMBER
turkeys	WORD
.	PUNCTUATION

`yyparse` 関数には、入力ルーチンが渡すトークンの定義が含まれてなければなりません。`yacc` コマンドの `-d` オプションを使用することによって、プログラムが `y.tab.h` というファイル名のトークンのリストを生成します。このリストは、`#define` 文のセットで、これによって、`yylex` で、パーサと同じトークンが使用可能になります。

パーサとの競合を回避するために、`yy` の英字で始まる名前は使用しないでください。`lex` を使用して入力ルーチンを生成しても、`C` 言語で記述してもかまいません。`lex` の使用方法についての詳細は、4.3 節を参照してください。

## 4.7 文法ファイル

`yacc` 文法ファイルは、次の 3 つの部で構成されています。

- 宣言 (4.7.1 項)
- 規則 (4.7.2 項)
- プログラム (4.7.3 項)

2 つのパーセント記号 (`%%`) は、文法ファイルの部を分けます。ファイルを読みやすくするために、パーセント記号は、単独で 1 行に書きます。文法ファイルのフォーマットは、次のとおりです。

```
[ declarations ]
%%
rules
[ %%
programs ]
```

規則の始まりと規則それ自体を示す (`%%`) 以外の、文法ファイルの各部分は、すべてオプションです。最小の `yacc` 文法ファイルには、次に示すように、定義およびプログラムはまったく含まれていません。

```
%%
rules
```

`yacc` プログラムでは、名前または予約記号の中を除いて、文法ファイルの中の空白、タブ、および改行文字は無視されます。これらの文字を使用することによって、文法ファイルを読みやすくすることができます。名前または予約記号の中では、空白、タブ、改行文字は、いずれも使用しないでください。

### 4.7.1 宣言

`yacc` 文法ファイルの宣言部には、次の要素が含まれます。

- 文法ファイルの他の部分で使用するすべての変数宣言または定数宣言
- 文法ファイル (ライブラリ・ヘッダ・ファイルに使用) の一部として他のファイルを呼び出す `#include` 文

- 生成されたパーサの処理条件を定義する文

変数宣言または定数宣言は、次のように、C プログラミング言語の構文に準拠します。

*type-specifier declarator;*

この構文では、*type-specifier* はデータ型キーワードで、*declarator* は変数または定数の名前です。名前の長さに制限はなく、英字、ドット、下線、および数字で構成されています。名前を数字で始めることはできません。大文字と小文字は区別されます。文法規則の本体で使用される名前は、トークンまたは非終端記号を表します。

宣言部で名前を宣言しない場合は、非終端記号としてのみその名前を使用することができます。それぞれの非終端記号は、規則部中で、少なくとも1つの規則の左側にその名前を使用して定義してください。#include 文はC言語構文の場合と同じで、同じ機能を持ちます。

yacc ツールには、表 4-3 にリストしたような、キーワードのセットがあります。これらのキーワードは、生成されたパーサの処理条件を定義しています。各キーワードは、パーセント記号(%)で始まり、その後にトークンのリストがあります。

表 4-3: yacc の処理条件の定義キーワード

キーワード	機能
%left	他のトークンと左結合するトークンを識別する。
%nonassoc	他のトークンと結合しないトークンを識別する。
%right	他のトークンと右結合するトークンを識別する。
%start	開始記号の名前を識別する。
%token	yacc が受け入れるトークン名を識別する。宣言部ですべてのトークン名を宣言すること。

同じ行でリストされたすべてのトークンは、同じ優先順位のレベルと結合性を持っています。ファイルの中の行は後に宣言されたものほど優先順位、または結合力が高くなります。たとえば、次のようになります。

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



この例は、4つの算術演算子の優先順位と結合性を説明しています。加算(+), 減算(-)の演算子は、左結合であり、同じく左結合である乗算(\*), 除算(/)の演算子よりも、優先順位が低いことを表しています。

#### 4.7.1.1 グローバル変数の定義

字句解析プログラムだけでなく、一部またはすべてのパーサ・アクションで使用されるグローバル変数も定義することができます。パーセント記号と中カッコ(%{ と %}) からなる2つの照合した記号で変数に対する宣言を囲むことによって、定義します。たとえば、プログラム全体のすべての部分で var 変数を使用可能にするためには、文法ファイルの宣言部の中に次のエントリを含んでください。

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

#### 4.7.1.2 開始記号

パーサは、開始記号と呼ばれる特殊記号を認識します。開始記号は、文法規則に付けられた名前です。この文法規則には、解析される言語の中で最も一般的な構造が記述されています。これが最も一般的な構造であるため、入力ストリームの解析をトップ・ダウンでパーサを開始させる構造になっています。%start キーワードを使用することによって、宣言部の中で開始記号を宣言します。開始記号を宣言しない場合には、パーサは、ファイル中の最初の文法規則の名前を使用します。

たとえば、C 言語の手順を解析する際に、パーサが認識する最も一般的な構造は次のようになります。

```
main()
{
    code_segment
}
```

開始記号は、この構造を記述する規則を示さなければなりません。ファイル内の残りのすべての規則は、処理の中でより低いレベルの構造体を識別する方法を記述します。

#### 4.7.1.3 トークン番号

トークン番号は、トークン名を表す、負でない整数になります。字句解析プログラムによって、トークン番号は、実際のトークン名ではなくパーサに渡られるので、プログラムはトークンに同じ番号を割り当てなければなりません。

yacc 文法ファイルで使用するトークンにトークン番号を割り当てることができます。トークン番号をトークンに割り当てなかった場合、yacc は、次の規則を使用してトークン番号を割り当てます。

- リテラル文字には、ASCII 文字セットの文字の数値が割り当てられる。
- 他の名前には、257 以降のトークン番号が割り当てられる。

---

#### 注意

---

0 (ゼロ) のトークン番号は使用できません。この番号は `endmarker` トークンに割り当てられます。この番号を再定義することはできません。

---

文法ファイルの宣言部の中のトークン (リテラル文字を含む) に番号を割り当てるためには、`%token` 行のトークン名の直後にゼロを含まない正の整数をいれます。この整数は、名前か、またはリテラル文字のトークン番号です。各番号は、ユニークでなければなりません。yacc と一緒に使用される字句解析プログラムは、入力の終わりに到達する際のトークンに、0 (ゼロ) または負の値を返さなくてはなりません。

### 4.7.2 文法規則

yacc 文法ファイルの規則には、1 つまたは複数の文法規則が含まれます。各規則には、構造が記述され、名前が付けられます。文法規則のフォーマットは、次のとおりです。

`[ nonterminal-name : BODY ;`

この構文では、*BODY* は 0 以上の名前とリテラルのシーケンスになります。コロンとセミコロンは、yacc の句読点として必要です。

同じ非終端名を伴う文法規則がいくつかある場合は、(|) 縦線を使用すれば、左の非終端名を何度も入力する必要がなくなります。また、縦線で結合されたすべての規則の最後にだけ、セミコロン (;) を使用してください。次の文法規則の 2 つの例は、ともに同じ意味になります。

#### 例 1

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

## 例 2

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

### 4.7.2.1 空文字列

空文字列と照合する非終端記号を示すためには、次のように、セミコロンだけを規則の本体に使用してください。

```
nullstr : ;
```

### 4.7.2.2 入力の終わりマーカ

字句解析プログラムが入力ストリームの終わりに達すると、パーサに `endmarker` と呼ばれる特別なトークンを送ります。このトークンは 0 のトークン値を持ち、入力の終わりをシグナル通知します。パーサが `endmarker` トークンを受け取ると、すべての入力定義された文法規則に割り当てられているかどうか、また、処理された入力が `yacc` 文法ファイルに定義されているとおり完全な単位を形成しているかどうかを、パーサがチェックをします。入力が完全な単位である場合は、パーサは停止します。入力が完全な単位ではない場合は、パーサによってエラーと停止がシグナル通知されます。

字句解析プログラムでは、ファイルやレコードなどが終わる時点で、`endmarker` トークンを送らなければなりません。

### 4.7.2.3 yacc パーサのアクション

それぞれの文法規則に、パーサが入力ストリームの中でその規則を認識するたびに、実行するアクションを指定することができます。アクションが値を返し、次に、前のアクションによって返された値を得ます。また、字句解析プログラムは、トークンのための値を返すこともできます。

アクションは、C 言語文で入出力を行ったり、サブプログラムを呼び出したり、外部ベクトルと変数の変更を行ったりします。文法ファイルのアクションは、中カッコ (`{ }`) で囲んだ 1 つ以上の文を使用して、指定します。次の例は、アクションが含まれている文法規則です。

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

```

{
    printf("a message\n");
    flag = 25;
}

```

アクションは、番号の付いた yacc パラメータ・キーワード (\$1, \$2 など) を使用することによって、他のアクションで生成された値を受け取ることができます。これらのキーワードは、左から右に読み、規則の右側の構成要素によって返された値を表示します。たとえば、次のようになります。

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

このコードが実行された場合、\$1 は、B を認識した規則によって返される値を持ち、\$2 は、C を認識した規則によって返される値を持ち、また、\$3 は、D を認識した規則によって返された値を持ちます。

値を返すためには、アクションで擬似変数 \$\$ をある値に設定します。たとえば、次のアクションでは、1 の値を返します。

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

省略時の設定では、ある規則の値は、\$1 の最初の要素の値になります。そのため、次のようなフォーマットの規則には、アクションは必要ありません。

```
A : B ;
```

規則が終了する前に解析処理を制御するには、規則の中央にアクションを記述します。この規則で、\$n パラメータを使用して値が返された場合は、その後のアクションで、その値を使用することができます。したがって、次の規則は、x を 1 に設定し、y を、C によって返された値に設定します。

```

A : B
    {
        $$ = 1;
    }
    C
    {
        x = $2;
        y = $3;
    }
;

```

内部的には、yacc は、中央にあるアクションの新しい非終端記号名を作成して、この名前と空文字列を照合させる新しい規則を作成します。したがって yacc は、前述のプログラムを、次のフォーマットで記述されているかのように処理します。ここでは、\$ACT は、空のアクションになります。

```

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

```

### 4.7.3 プログラム

yacc 文法ファイルのプログラム部には、規則部の中のアクションで使うことができる C 言語関数が含まれています。また、字句解析プログラム `yylex` を記述する場合は、プログラム部に追記してください。

### 4.7.4 文法ファイルの使用上のガイドライン

この項では、yacc 文法ファイルを使用する際の一般的なガイドラインを説明します。次の処理に関して解説しています。

- コメントの使用 (4.7.4.1 項)
- リテラル文字列の使用 (4.7.4.2 項)
- 文法ファイルのフォーマット (4.7.4.3 項)
- 再帰の使用 (4.7.4.4 項)
- エラーの訂正 (4.7.4.5 項)

#### 4.7.4.1 コメントの使用

文法ファイルのコメントには、プログラムの実行内容が説明されています。名前を指定できるファイルであれば文法ファイルのどこにでも、コメントを書くことができます。ただし、ファイルを読みやすくするために、規則の中の関数ブロックの最初の部分に独立した行としてコメントを記述するようにしてください。yacc 文法ファイルのコメントは、C 言語プログラムのコメントと、まったく同じフォーマットを持っています。つまり、スラッシュとアスタリスク (`/*`) で始まり、アスタリスクとスラッシュ (`*/`) で終わります。その例を次に示します。

```
/* This is a comment on a line by itself. */
```

#### 4.7.4.2 リテラル文字列の使用

リテラル文字列は、アポストロフィ、すなわち一重引用符 ( ' ' ) で囲まれた、1 つまたは複数の文字です。バックスラッシュ ( \ ) は、C 言語のように、リテラル文字列の中ではエスケープ文字になり、C 言語の特別文字シーケンスは、次のように認識されます。

<code>\n</code>	改行文字
<code>\r</code>	リターン
<code>\'</code>	アポストロフィまたは一重引用符
<code>\\</code>	バックスラッシュ
<code>\t</code>	タブ
<code>\b</code>	バックスペース
<code>\f</code>	改ページ
<code>\nnn</code>	8進法の値 <i>nnn</i>

`\0` または `0` (空文字) は、絶対に文法規則では使用しないでください。

#### 4.7.4.3 文法ファイルのフォーマットに関するガイドライン

次のガイドラインを使用すると、`yacc` 文法ファイルをより読みやすくすることができます。

- トークン名には大文字を、非終端記号名には小文字を使用する。
- 文法規則とアクションは、いずれかを変更する場合に片方を変更するだけですむように別の行に置く。
- 規則はすべて、左側の位置を合わせる。左側から入力を始め、縦線 ( | ) を使用してその残りの規則を始める。
- 左側が同じである各規則のセットについては、左の最後の規則の後に、1 行にセミコロン ( ; ) を 1 つだけ入力する。そうすると、新しい規則が容易に追加できる。
- 規則本体はタブで 2 つ分字下げし、またアクション本体はタブ 3 つ分字下げする。

#### 4.7.4.4 文法ファイル中の再帰の使用

再帰とは、規則自身を定義するために関数を使用する処理のことです。言語定義では、通常、これらの規則は次のフォーマットになります。

```
rule      :      end case
          |      rule, end case
```

rule の最も簡単な例は、end case で、rule は、2 つ以上の end case で構成することもできます。rule の定義に rule を使用する 2 行目のエントリが、再帰です。この規則が与えらると、パーサは、ストリームが最終の end case に減少するまで入力を繰り返します。

yacc ツールでは、左再帰の文法をサポートしています。右再帰はサポートしていません。規則中に再帰を使用する場合は、常にその規則の一番左のエントリとして (例に示したように)、規則名への呼び出しを含めてください。次の例で示すように、規則名の呼び出しが行の後方に出現した場合、パーサの内部のスタック・スペースが不足し、クラッシュすることがあります。

```
rule      :      end case
          |      end case, rule
```

#### 4.7.4.5 文法ファイルのエラー

yacc ツールは、すべての文法指定のセットに対応するパーサを生成することはできません。文法規則自身が矛盾している場合、または yacc が持つ方法とは別の方法で照合させる必要がある場合は、yacc は、パーサを生成しません。通常、yacc は、エラーを示すメッセージを表示します。これらのエラーを訂正するためには、文法ファイルの規則を設計し直すか、または yacc では処理できないパターンを認識する字句解析プログラムを生成してください。

#### 4.7.5 パーサによるエラー処理

パーサが、入力ストリームを読み取る場合には、その入力ストリームが、文法ファイルの規則に一致しないことがあります。文法ファイルにエラー処理ルーチンがある場合、パーサは、データを再入力させたり、不正データをスキップしたりすることができます。または、データの削除および回復のアクションを行うことができます。たとえば、パーサがエラーを検出した場合には、構文解析ツリー記憶を再生したり、シンボル・テーブル・エントリのクリーンアップあるいは変更を行ったり、またはこれ以上の出力を生成しないようにスイッチの設定を行ったりする必要があります。

エラーが発生すると、エラー処理ルーチンを使用するまでパーサは停止しています。さらにエラーを検出するために入力の処理を続けるには、パーサがさらに入力を認識できるような入力ストリームのある場所からパーサを再起動してください。エラーの発生後にパーサを再起動する方法としては、エラーに続くトークンのうちのいくつかを破棄して、入力ストリームのその場所からパーサを再起動する方法があります。

yacc ツールには、`error` という、特殊なトークン名があります。これはエラー処理のために使用されます。文法ファイルの入力エラーが発生する可能性があるところにこのトークンを置いてください。そうすれば、回復ルーチン `error` を使用することができます。入力エラーが、`error` トークンによって保護された位置に発生した場合は、パーサは、正常なアクションではなく、`error` トークンのアクションを実行します。

1 つのエラーで多数のエラー・メッセージが生成されないように、パーサは、エラーの後の 3 つのトークンが正常に処理されるまで、エラー状態を保持します。パーサがこのエラー状態の間に、別のエラーが発生した場合には、パーサは入力トークンを破棄し、メッセージを表示しません。また、`error` アクションで引数を使用すると、パーサが処理を再開する時点を指定することができます。たとえば、次のように入力します。

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

エラーが発生すると、この規則によりパーサは、そのトークンおよびそれに続くトークンをすべて飛ばして、次のセミコロンを探すように指示されます。そのエラーの後のトークンおよび次のセミコロンまでのトークンは、すべて破棄されます。パーサがセミコロンを検出すると、この規則を修正してこの規則に関連したクリーンアップ・アクションをすべて実行します。

#### 4.7.5.1 エラーの訂正

対話型の環境で入力ストリームを入力した場合、データ・ストリームの行を再入力することによって、入力エラーを訂正することができます。たとえば、次のように入力します。

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



```
;
```

この例では、パーサは、エラーの後の3つの入力トークンに対してエラー状態のままになっています。最初の3つのトークンでエラーが検出された場合は、パーサはトークンを削除して、メッセージを表示しません。回復するには、`yyerrok`; 文を使用してください。パーサは `yyerrok`; 文を検出すると、エラー状態をそのままにして通常の処理を開始します。回復の例は次のとおりです。

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

#### 4.7.5.2 先読みトークンの消去

先読みトークンは、次にパーサによってチェックされるトークンです。エラーが発生すると、先読みトークンは、エラーが検出された箇所のトークンになります。しかし、エラー回復アクションに、処理を再開するための正確な位置を検出するコードが含まれている場合は、そのコードの先読みトークンも変更しなければなりません。先読みトークンを消去するには、`yyclearin`; 文をエラー回復アクションに含みます。

## 4.8 パーサの動作

`yacc` プログラムによって、文法ファイルは、C 言語プログラムに変更されます。C 言語プログラムは、コンパイルされて実行されると、文法規則に従って入力を解析します。

パーサは、スタックを持つ、有限状態マシンです。パーサは次の入力トークン(先読みトークン)を読み取り、記憶することができます。現在の状態は、常に、スタックの最上部にある状態です。有限状態マシンの状態は、小さい整数によって表されます。最初は、マシンは0(ゼロ)の状態です、スタックには0(ゼロ)のみが含まれています。先読みトークンはまったく読み取りません。

マシンは、次の4つのアクションのうちのいずれかを実行することができます。

shift $n$	パーサは、現在の状態をスタックにプッシュし、 $n$ を現在の状態にし、先読みトークンを消去する。
reduce $r$	引数 $r$ は規則の番号。パーサは、入力ストリームの $r$ 番の規則に照合するトークン・シーケンスを検出すると、出力ストリームの規則番号を持つシーケンスに置換される。
accept	パーサによってすべての入力チェックされ、それらの入力は文法指定と照合し、その入力、最も高いレベルの構造 (開始記号として定義) を満たすように認識される。このアクションは、先読みトークンがエンド・マーカで、しかもパーサが正常に完了したことを示している場合にのみ有効。
error	パーサは入力ストリームの処理を続行できず、文法指定で定義された規則への正常な照合を続けられない。パーサが探している入力トークンは、先読みトークンとともに、正当な結果を出力できない。パーサはエラーを報告し、その状況を回復して、解析を再開しようとする。

パーサは、1つの処理手順の中で、次のアクションを実行します。

1. パーサは、現在の状態に基づき、次のアクションを決めるのに先読みトークンが必要かどうかを決定する。パーサに先読みトークンが必要となるとき、それがなければ、パーサは、次のトークンを得るために `yylex` を呼び出す。
2. 現在の状態と、必要であれば先読みトークンを使用して、パーサはその次のアクションを決定し、実行する。これによって、スタックにプッシュされるか、あるいはスタックからはじき出される状態で終了し、先読みトークンは処理されるか、あるいは処理されずに残される場合もある。

#### 4.8.1 shift アクション

shift アクションは、パーサの使用する最も一般的なアクションです。パーサが shift を実行すると、常に、先読みトークンがあります。次のパーサ・アクションの規則を考えてみます。

IF shift 34

パーサが、この規則を含む状態で、しかも先読みトークンが IF である場合は、パーサは次の手順を実行します。

1. 現在の状態をスタックにプッシュする。
2. 状態 34 を現在の状態に指定する (スタックの最上部に置く)。

3. 先読みトークンを消去する。

#### 4.8.2 reduce アクション

`reduce` アクションを使用すると、スタックが肥大化するのを防ぐことができます。パーサは、入力ストリームに対して規則の右側に照合し、入力ストリームのトークンを規則の左側に置換する準備ができると、縮小アクションを使用します。パーサは、先読みトークンを使用してパターンが完全に照合したかを決定する必要があります。

縮小アクションは、個々の文法規則によって関連付けられています。また、文法規則もまた、小さい整数を持つために、`shift` アクションと `reduce` アクションの番号の意味は混乱が生じやすくなっています。たとえば、次の2つのアクションの1番目は、文法 18 を表し、2番目はマシン状態 34 を表しています。

```
reduce 18
IF shift 34
```

たとえば、次の規則の縮小を考えてみてください。

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

パーサによって、スタックから一番上の3つの状態が取り出されます。取り出された状態の番号は、規則の右側の記号の番号と等しくなります。これらの状態は、`x`、`y`、および `z` を認識している間、スタックに置かれるものです。上記の状態が取り出されると、パーサは、規則が処理される前 (規則 `A` を認識して、その規則を満たすのが必要な状態) のパーサの状態になります。パーサは、元に戻された状態と規則の左側の記号を使用して、`goto` アクションを実行します。このアクションは、`A` の `shift` と似ています。新しい状態が得られると、スタックにプッシュされて、解析が続きます。

`goto` アクションは、トークンの通常の `shift` とは異なります。先読みトークンは `shift` で消去されますが、`goto` によっては作用されません。上記の例で3つの状態が取り出されると、元に戻された状態には、次のようなエントリが含まれています。

```
A goto 20
```

このエントリによって、状態 20 がスタックにプッシュされて、現在の状態となります。

また、`reduce` アクションは、ユーザが指定するアクションや値の取り扱いにおいても重要です。規則が縮小されると、パーサは、スタックを調整する

前にユーザが規則に入れたコードを実行します。状態を保持するスタックだけでなく、そのスタックと平行して実行される別のスタックによって、字句解析プログラムとアクションから返された値が保持されます。shift 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。ユーザが指定するコードを実行した後、パーサが縮小を行います。パーサによって goto 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。名前がドル記号 (\$) で始まる yacc 変数は、値スタックを表します。

### 4.8.3 あいまいな規則およびパーサの競合

入力文字列を 2 つ以上の違った方法で構造化できる場合は、文法規則はあいまいになります。次は、その例です。

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

上記の規則では、2 つの式の間に - (マイナス) 符号が入っているので、算術式になりますが、この文法規則では、すべての複雑な入力をどのように構成するかは指定されません。たとえば、次のような場合です。

```
expr - expr - expr
```

上記の規則を使用すると、プログラムは、この入力を左結合にするか、または右結合にするのかのどちらかで構成します。

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

または

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

上記の 2 つの数式を評価すると、別の結果が生じます。

パーサがあいまいな規則を処理する場合は、入力を処理する際に 4 つのアクションのうち、いずれを実行するかを混同することがあります。次の 2 種類の競合が生じる可能性があります。

Shift/reduce 競合	ある規則が、shift アクションまたは reduce アクションのいずれを使用しても正しく評価できる、別の結果を出力する。
Reduce/reduce 競合	ある規則が、2 つの異った reduce アクションのいずれを使用しても正確に評価できる、2 つの異ったアクションを生成する。

shift/shift 競合はあり得ません。

このような競合は、規則があまり完全でない場合に生じます。たとえば、次の入力と前述のあいまいな規則を考えてみてください。

a - b - c

入力の最初の 3 つの部分を読み取った後、パーサは、次の部分を持ちます。

a - b

この入力、文法規則の右側に照合します。パーサは、この規則を適用して、入力を縮小させることができます。この規則を適用した後に、入力は次のようになります。

*expr*

これは規則の左側です。次に、パーサは入力の最後の部分を次のように読み取ります。

- c

この時点で、パーサは次の部分を持ちます。

*expr* - c

この入力を縮小することによって、左結合の解釈が生まれます。

ただし、パーサは入力ストリームで先読みを行うこともできます。入力の最初の 3 つの部分を受け取った後に、入力ストリームの読み取りを続行して、次の 2 つの部分を得た場合は、パーサは、次の入力を持ちます。

a - b - c

この規則を一番右の 3 つの部分に適用することによって、b - c を *expr* に縮小します。このとき、パーサは、次のものを持ちます。

a - *expr*

再度、式を縮小することによって、右結合の解釈になります。

したがって、パーサによって最初の 3 つの部分を読み取られたところで、2 つの正当なアクション、すなわち、*shift* か、*reduce* のうちの 1 つを選択することができます。パーサにアクションを決定する規則がまったくない場合は、*shift/reduce* 競合が生じます。

パーサが 2 つの有効な *reduce* アクションの中から選択が可能な場合は、同様の状態が発生します。このような状態を *reduce/reduce* 競合といいます。

*shift/reduce* 競合あるいは *reduce/reduce* 競合が発生する場合、*yacc* は選択可能な場合は、有効な方法を選んで、パーサを生成します。選択するための規則がない場合、*yacc* は次の規則を使用します。

- *shift/reduce* 競合では、*shift* を実行する。

- `reduce/reduce` 競合では、入力ストリーム中の最初に出現する文法規則により縮小する。

パーサがどの規則を認識しているか認識する前にアクションを実行しなければならない場合には、規則内のアクションを使用することによって、競合が発生することがあります。このような場合、前述の規則を使用すると、間違ったパーサが生成されます。このために、`yacc` は、前述の 2 つの規則を適用することによって解決された、`shift/reduce` 競合と `reduce/reduce` 競合の数を報告します。

## 4.9 デバッグ・モードの使用

通常の操作では、外部整数変数 `yydebug` は 0 に設定されています。しかし、ゼロ以外の値に設定した場合、パーサは、実行記述ファイルを生成します。この記述ファイルには、パーサが受け取る入力トークン、および各トークンに対して行うアクションが含まれています。次の 2 つの中のいずれかの方法で、`yydebug` 変数を設定することができます。

- `yacc` 文法ファイルの宣言部に次の C 言語文を入れることによって `yydebug` 関数を使用する。  

```
yydebug = 1;
```
- デバッガを使用して最後のパーサを起動し、デバッガ・コマンドを使用して、`yydebug` 変数のオンまたはオフを設定する。`dbx` などのデバッガの使用についての詳細は、各デバッガのリファレンス・ページを参照。

## 4.10 簡易計算機プログラムの作成

例 4-1 および例 4-2 に示す `lex` で生成された字句解析プログラムおよび `yacc` で生成されたパーサのプログラムを使用して、加算、減算、乗算、および除算の演算を実行する簡単な卓上計算機のプログラムを作成することができます。また、この計算機プログラムでは、1 つの小文字で指定した変数に値を代入して、計算することもできます。プログラムを含むファイルは次のとおりです。

- `calc.l` – 字句解析規則を定義する `lex` 仕様ファイル
- `calc.y` – 解析規則を定義する `yacc` 文法ファイル。入力を提供するために、`lex` によって作成された `yylex` 関数を呼び出す。

慣例で、lex と yacc のプログラムには、ファイル名の接尾文字として、それぞれ .l と .y の英字が使用されます。例 4-1 と 例 4-2 は、実際に入力するためのプログラム・フラグメントの例です。この項の処理手順は、ファイルが現在のディレクトリにあることを前提としています。

示された順序どおりに、次の手順を実行して、lex と yacc を使用した計算機プログラムを作成してください。

1. 次のコマンドを使用して、yacc 文法ファイル进行处理する。

yacc では、-d オプションを指定すると、C 言語ソース・コードに加えて、使用するトークンを定義するファイルを作成します。

```
% yacc -d calc.y
```

このコマンドで、次のファイルが作成されます。

- y.tab.c – パーサ用に yacc が作成した C 言語ソース・ファイル
- y.tab.h – パーサで使用されるトークンのための define 文を含むヘッダ・ファイル

2. 次のコマンドを使用して lex 仕様ファイル进行处理する。

```
% lex calc.l
```

このコマンドは lex.yy.c ファイルを作成し、lex が字句解析プログラム用に作成した C 言語ソース・ファイルを含んでいます。

3. 次のコマンドを使用して 2 つの C 言語ソース・ファイルをコンパイルし、リンクする。

```
% cc -o calc y.tab.c lex.yy.c
```

4. ls コマンドを使用して次のファイルが作成されたことを確認する。

- y.tab.o – y.tab.c から作成されたオブジェクト・ファイル
- lex.yy.o – lex.yy.c から作成されたオブジェクト・ファイル
- calc – 実行可能プログラム・ファイル

calc コマンドを入力すると、プログラムを実行することができます。その後で、数字と演算子を代数的に入力することができます。リターンを押すと、このプログラムは演算の結果を表示します。次のように、変数に値を与えることができます。

```
m=4
```

次のように、計算で変数を使用することができます。

m+5  
9

#### 4.10.1 パーサのソース・コード

例 4-1 は、calc.y ファイルの内容です。このファイルには、yacc 文法ファイルの 3 つの部である、宣言、規則、およびプログラムのエントリが入っています。このファイルで定義された文法には、演算子の優先順位による通常の代数学の階層がサポートされています。

ファイルのさまざまな要素の記述とそれらの関数は、例の後に示されています。

##### 例 4-1: 計算機のパーサ・ソース・コード

---

```
%{  
#include <stdio.h> 1  
  
int regs[26]; 2  
int base;  
  
%}  
  
%start list 3  
  
%token DIGIT LETTER 4  
  
%left '|' 5  
%left '&  
%left '+' '-'  
%left '*' '/' '%'  
%left UMINUS /*supplies precedence for unary minus */  
  
%%  
/* beginning of rules section */  
  
list: /*empty */  
    | list stat'\n'  
    | list error'\n'  
    {  
        yyerrok;  
    }  
    ;  
stat: expr  
    {  
        printf("%d\n", $1);  
    }  
    |  
    LETTER '=' expr
```



#### 例 4-1: 計算機のパーサ・ソース・コード (続き)

---

```
    {
        regs[$1] = $3;
    }
    ;
expr:  '(' expr ')'
    {
        $$ = $2;
    }
    |
    expr '*' expr
    {
        $$ = $1 * $3;
    }
    |
    expr '/' expr
    {
        $$ = $1 / $3;
    }
    |
    expr '%' expr
    {
        $$ = $1 % $3;
    }
    |
    expr '+' expr
    {
        $$ = $1 + $3;
    }
    |
    expr '-' expr
    {
        $$ = $1 - $3;
    }
    |
    expr '&' expr
    {
        $$ = $1 & $3;
    }
    |
    expr '|' expr
    {
        $$ = $1 | $3;
    }
    |
    '-' expr %prec UMINUS
    {
        $$ = -$2;
    }
```

#### 例 4-1: 計算機のパーサ・ソース・コード (続き)

---

```
    }
    |
    LETTER
    {
        $$ = regs[$1];
    }
    |
    number
    ;
number: DIGIT
    {
        $$ = $1;
        base = ($1==0) ? 8:10;
    }
    |
    number DIGIT
    {
        $$ = base * $1 + $2;
    }
    ;

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

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

yywrap()
{
    return(1);
}
```

---

宣言部には、次の機能を実行するエントリが含まれています。

- ❶ 標準 I/O ヘッダ・ファイルをインクルードする。
- ❷ グローバル変数を定義する。
- ❸ 処理を開始する位置として規則のリストを定義する。
- ❹ パーサで使用するトークンを定義する。

⑤ 演算子およびその優先順位を定義する。

規則部には、入力ストリームを解析する規則が定義されています。

プログラム部には、次のルーチンが含まれています。これらのルーチンが含まれているため、このファイルの処理には、yacc ライブラリを使用する必要がありません。

- `main( )` – プログラムを開始するために `yyparse( )` を呼び出す、必須メイン・プログラム
- `yyerror(s)` – 構文エラー・メッセージをプリントするエラー処理ルーチン
- `yywrap( )` – 入力が終了した場合に 1 の値を返す、ラップアップ・ルーチン

#### 4.10.2 字句解析プログラムのソース・コード

例 4-2 は、`calc.l` ファイルの内容です。このファイルには、`y.tab.h` ファイルが使用する、標準入出力用の `#include` 文が含まれています。`y.tab.h` ファイルは、`calc.l` で `lex` を実行する前に、`yacc` によって生成されたファイルです。`y.tab.h` ファイルは、パーサ・プログラムで使用されるトークンを定義します。また、`calc.l` は、入力ストリームからトークンを生成する規則を定義します。

例 4-2: 計算機的字句解析プログラム・ソース・コード

---

```
%{

#include <stdio.h>
#include "y.tab.h"
int c;
extern int yylval;
}%
%%
" "      ;
[a-z]    {
          c = yytext[0];
          yylval = c - 'a';
          return(LETTER);
        }
[0-9]    {
          c = yytext[0];
          yylval = c - '0';
          return(DIGIT);
        }
```

#### 例 4-2: 計算機の子句解析プログラム・ソース・コード (続き)

---

```
    }  
    [^a-z0-9\b]    {  
                    c = yytext[0];  
                    return(c);  
    }
```

---

## プログラムにおける m4 マクロの使用

この章では、m4 マクロ・プリプロセッサについて説明します。この m4 マクロ・プリプロセッサは、ソース・ファイルの最初の部分に m4 マクロ定義を設定することにより、ユーザのマクロ定義を可能にするフロント・エンド・フィルタです。m4 プリプロセッサは、プログラム・ソース・ファイルにも、文書ソース・ファイルにも使用することができます。

この章では、マクロについての次の情報について説明します。

- マクロの使用 (5.1 節)
- マクロの定義 (5.2 節)
- 他の m4 マクロの使用 (5.3 節)

### 5.1 マクロの使用

マクロを使用すると、大量の内容を 1 語または 2 語で代用することができるため、プログラミングや入力作業が容易になります。ソース・ファイルのマクロ呼び出しには、次のフォーマットを使用します。

```
name[( arg1[, arg2] )]
```

たとえば、何度か同じメッセージを表示する C プログラムがある場合には、次のように、printf 文をコーディングすることができます。

```
printf("\nThese %d files are in %s:\n",cnt,dir);
```

プログラムの改良に伴って、メッセージの内容を変更することにした場合、メッセージの各インスタンスを編集する必要があります。次のようにマクロを定義すると、大量の作業を省略することができます。

```
define(filmsg,'printf("\nThese %d files are in %s:\n",$1,$2)')
```

このメッセージを出力するすべての場所で、マクロを次のように使用します。

```
filmsg(cnt,dir);
```

このように指定されていれば、メッセージを 1 箇所で編集するだけですべての表示が変更されます。

マクロ定義は、記号名 (トークン) と、それに置き代わる文字列によって構成されます。トークンは、英字または下線で始まる英数字 (英文字、数字、および下線) の文字列で、英数字以外の文字 (句読点またはホワイト・スペース) で区切られます。たとえば、N12 と N はどちらもトークンですが、A+B はトークンではありません。m4 を使用してファイルを処理すると、マクロが認識されるたびにマクロがその定義と置換されます。シンボリック・ネームをテキストと置換するだけでなく、m4 は、次のような動作を実行することもできます。

- 算術計算
- ファイル操作
- 条件付きマクロ展開
- 文字列および部分文字列関数
- システム・コマンドの実行

m4 プログラムは、ファイルの各トークンを読み取り、トークンがマクロ名であるかどうか調べます。他のトークンに埋め込まれたマクロ名は認識されません。たとえば、m4 は、N12 にトークン N が含まれているとは解釈しません。トークンがマクロ名の場合、m4 はそれを定義されたテキストに置換し、結果として生じた文字列を入力に戻し、再走査を実行します。

このように、マクロ展開は再帰的です。マクロ定義には、どんなに深くネストした他のマクロでもインクルードすることができます。引数を付けてマクロを呼び出すこともできます。この場合、引数はまとめられ、定義されたテキストを再走査する前に、定義テキストの適切な位置に代入されます。

m4 プリプロセッサは標準の UNIX フィルタです。標準入力、またはリクエストされた入力ファイルからの入力を受け入れ、その出力を標準出力に書き込みます。次に、正しい m4 の使用例を示します。

```
% grep -v '#include' file1 file2 | m4 > outfile
% m4 file1 file2 | cc
```

m4 プログラムは、各引数を順番に処理します。引数が全くないか、または引数がマイナス記号 (-) である場合、m4 は入力ファイルとして標準入力から読み取ります。

## 5.2 マクロの定義

m4 に実装された約 20 個の組み込みマクロの 1 つである、`define` コマンドでマクロ定義を作成します。たとえば次のように定義します。

```
define(N,100)
```

開き (左) カッコとワード `define` の間にスペースを入れてはなりません。

このマクロ定義によって、トークン `N` は処理されたファイル内のどこに現れても `100` に置換されます。定義テキストは、どんなテキストでもかまいませんが、テキストにカッコが含まれる場合を除きます。この場合は、対応していないカッコを引用符で囲って保護しない限り、開き (左) カッコの数と閉じ (右) カッコの数が一致していなければなりません。引用についての詳細は、5.2.1 項を参照してください。

組み込みマクロとユーザ定義マクロは、組み込みマクロの一部が処理の状態を変更することを除いて、同じ方法で動作します。組み込みマクロの一覧については、5.3 節を参照してください。

マクロを他のマクロとして定義することができます。たとえば、次のようになります。

```
define(N,100)
define(M,N)
```

この例では、`M` および `N` が `100` になるように定義しています。後で `N` の定義を変更し、新しい値を割り当てた場合、`M` は、ユーザが `N` に与えた新しい値ではなく、`100` の値を保持します。m4 プリプロセッサが可能な限り早い時点でマクロ名をそれらの定義テキストに展開するため、`M` の値は `N` の値に置き換えられません。全体の結果は、`M` に関する限り、最初から `define(M,100)` のように定義した場合と同じになります。`M` の値を `N` の値に置き換える場合は、次のように定義の順序を入れ替えてください。

```
define(M,N)
define(N,100)
```

この時点で、`M` は文字列 `N` であると定義されています。後で `M` の値が要求されると、`M` は `N` に置換され、`N` の値は再走査され、その時点の `N` の値に置換されます。

`define` コマンドで作成したマクロ定義は、閉じカッコの次の文字を削除しません。たとえば、次のように定義します。

```
Now is the time for all good persons.  
define(N,100)  
Testing N definition.
```

上記の例の結果は、次のようになります。

```
Now is the time for all good persons.  
  
Testing 100 definition.
```

空白行が `define` マクロを含んだ行の最後に改行文字がある結果として出力されます。組み込みの `dnl` マクロは、次の改行文字まで後続する文字を次の改行文字を含めてすべて削除します。空白行を削除するには、このマクロを使用してください。たとえば、次のように指定します。

```
Now is the time for all good persons.  
define(N,100)dnl  
Testing N definition.
```

上記の例の結果は、次のようになります。

```
Now is the time for all good persons.  
Testing 100 definition.
```

この節では、次の情報について説明します。

- 引用符の使用 (5.2.1 項)
- マクロ引数 (5.2.2 項)

### 5.2.1 引用符の使用

`define` マクロの引数の展開を遅らせるには、マクロを対応する引用符で囲みます。引用符として使用される文字の省略時の設定は、左右の一重引用符 ( `'` および `'` ) ですが、組み込みの `changequote` マクロを使用すれば、別の文字を指定することができます (5.3 節を参照)。引用符で囲まれたテキストはすぐには展開されませんが、引用符自体は削除されます。引用符で囲まれた値は、引用符を削除された状態の文字列です。次の例を参照してください。

```
define(N,100)  
define(M,'N')
```

引数が収集されると、`N` のまわりの引用符は削除されます。引用符を使用すると、`M` は `100` ではなく、文字列 `N` として定義されます。この例では、`M` の値として `N` の値が置換されています。したがってこの方法も、5.2 節に示した次の定義と同じ結果を得るための 1 つの方法です。

```
define(M,N)  
define(N,100)
```

## 5-4 プログラムにおける `m4` マクロの使用



一般規則として、`m4` が何かを評価するときには、必ず 1 レベルの引用符文字を削除します。このことは外部マクロでも同じです。たとえば、ワード `define` を出力に表示するには、次のように引用符で文字を囲んでください。

```
'define' = 1
```

`m4` が引用符を処理するので、ネストされたマクロには注意する必要があります。次の例を考えてみましょう。

```
define(dog,canine)
define(cat,animal chased by 'dog')
define(mouse,animal chased by cat)
```

`cat` の定義を処理した時点では、引用符で囲まれているために `dog` は直ちに `canine` に置換されませんが、`mouse` が処理される場合には、`cat` (`animal chased by dog`) の定義が使用されます。この時点で、`dog` は引用符で囲まれていないため、`mouse` の定義は `animal chased by animal chased by canine` になります。上記の例が、`infile` というファイルに含まれている場合は、次のようになります。

```
% cat infile
define(dog,canine)
define(cat,animal chased by 'dog')
define(mouse,animal chased by cat)

dog
cat
mouse
% m4 infile
canine
animal chased by canine
animal chased by animal chased by canine
```

既存のマクロを再定義する場合は、最初の引数(マクロ名)を次のように引用符で囲む必要があります。

```
define(N,100)
:
define('N',200)
```

引用符で囲まれていない場合には、2 番目の `define` マクロは `N` を検出して認識し、値を置換して次のような結果になります。

```
define(100,200)
```

`m4` プログラムでは、名前だけが定義できて数値は定義できないため、この文は無視されます。

## 5.2.2 マクロ引数

最も簡単なマクロ処理の形式は、これまでの説明にあるように、1つの文字列を別の(固定した)文字列に置換することです。ただし、所定のマクロによって異なった場所で異なった結果を得るために、マクロに引数を付けることができます。引数がマクロの置き換えテキスト(その定義の2番目の引数)内で使用される位置を示すには、 $n$ 番目の引数を示す  $\$n$  記号を使用してください。たとえば、記号  $\$1$  はマクロの最初の引数を参照します。マクロが使用された場合、 $m4$  は、記号を示された引数の値に置換します。次の例を参照してください。

```
define (bump, $1=$1+1)
:
```

```
bump (x) ;
```

この例では、 $m4$  は `bump (x)` 文を `x=x+1` に置換します。

マクロには必要なだけ引数を付けることができますが、 $\$n$  記号 ( $\$1$  から  $\$9$  まで) を使用してアクセスできるのは9個の引数だけです。9番目の引数以降の引数にアクセスするには、`shift` マクロを使用してください。`shift` マクロは、最初の引数を手放し、残りの引数を  $\$n$  記号 (2番目の引数を  $\$1$  に、3番目の引数を  $\$2$  に) に再び割り当てます。`shift` マクロを1回以上使用すると、マクロに使用されるすべての引数にアクセスすることができます。

記号  $\$0$  はマクロ名を返します。引数が指定されなかった場合は、空文字列に置換されます。このため、次のように引数を連結してマクロを定義することができます。

```
define (cat, $1$2$3$4$5$6$7$8$9)
:
```

```
cat (x,y,z)
```

この例では、`cat (x,y,z)` 文が `xyz` に置換されます。対応する引数が提供されないため、この例の  $\$4 \sim \$9$  の引数は `null` になります。

マクロを走査する時点で、 $m4$  プログラムは、引用符で囲まれていない前の空白、タブ、および引数の中の改行文字を破棄しますが、他のすべてのホワイト・スペースは保持します。次の例を参照してください。

```
define(a,      "$1 $2$3")
:
```

```
a(b,  
c,  
d)
```

この例では、a マクロを b cd となるように展開しています。ただし、define マクロでは、改行文字は有効です。次の例を参照してください。

```
define(a,$1  
$2$3)  
:  
:  
  
a(b,c,d)
```

この 2 番目の例では、a マクロは次のように展開されます。

```
b  
cd
```

マクロ引数はコンマで区切られます。引数にコンマが含まれている場合には、引数の途中で終了していると誤解されないように、カッコを使用してコンマを含む引数を囲んでください。たとえば、次の文の引数は 2 つだけです。

```
define(a, (b,c))
```

最初の引数は a で、2 番目の引数は (b,c) です。引数内に単一のカッコを指定する場合には、一重引用符で囲んでください。

```
define(a,b`)'c)
```

この例では、b)c が 2 番目の引数になります。

## 5.3 他の m4 マクロの使用

m4 プログラムは、定義済みのマクロ (組み込みマクロ) のセットを提供しています。表 5-1 ではこれらのマクロのすべてをリストし、概略を説明しています。

以下の項では、大部分のマクロとその使用法について、詳細に説明します。

- コメント文字の変更 (5.3.1 項)
- 引用符文字の変更 (5.3.2 項)
- マクロ定義の取り消し (5.3.3 項)
- 定義されたマクロのチェック (5.3.4 項)
- 整数演算の使用 (5.3.5 項)
- ファイルの操作 (5.3.6 項)

- 出力のリダイレクト ( 5.3.7 項)
- プログラムでのシステム・プログラムの使用 ( 5.3.8 項)
- 固有のファイル名の使用 ( 5.3.9 項)
- 条件式の使用 ( 5.3.10 項)
- 文字列操作 ( 5.3.11 項)
- 表示 ( 5.3.12 項)

表 5-1: 組み込みの m4 マクロ

マクロ	説明
<code>changeocom(<i>l</i>, <i>r</i>)</code>	左右のコメント文字を <i>l</i> および <i>r</i> で表された文字に変更する。2 つの文字は異ならなければならない。
<code>changequote(<i>l</i>, <i>r</i>)</code>	左右の引用符文字を <i>l</i> および <i>r</i> で表された文字に変更する。2 つの文字は異ならなければならない。
<code>decr(<i>n</i>)</code>	<i>n</i> -1 の値を返す。
<code>define(<i>name</i>, <i>replacement</i>)</code>	<i>replacement</i> の値を持つ <i>name</i> という名称の新しいマクロを定義する。
<code>defn(<i>name</i>)</code>	引用符で囲まれた <i>name</i> の定義を返す。
<code>divert(<i>n</i>)</code>	出力ストリームを番号 <i>n</i> の一時ファイルに変更。
<code>divnum</code>	現在アクティブな一時ファイルの番号を返す。
<code>dnl</code>	改行文字までのテキストを削除する。
<code>dumpdef('name'[, 'name'...])</code>	指定したマクロの名称および現在の定義をプリントする。
<code>errprint(<i>str</i>)</code>	<i>str</i> を標準エラー・ファイルにプリントする。
<code>eval(<i>expr</i>)</code>	<i>expr</i> を 32 ビット算術式として評価する。
<code>ifdef('name', <i>arg1</i>, <i>arg2</i>)</code>	マクロ <i>name</i> が定義されている場合は <i>arg1</i> を返し、そうでない場合は <i>arg2</i> を返す。

表 5-1: 組み込みの m4 マクロ (続き)

マクロ	説明
<code>ifelse(str1, str2, arg1, arg2)</code>	文字列 <code>str1</code> および <code>str2</code> を比較する。 <code>ifelse</code> は、それらが一致していた場合は <code>arg1</code> の値を、そうでない場合は <code>arg2</code> の値を返す。
<code>include(file)</code> <code>sinclude(file)</code>	<code>file</code> の内容を返す。ファイルにアクセスすることができない場合は、 <code>sinclude</code> マクロはエラーを報告しない。
<code>incr(n)</code>	<code>n+1</code> の値を返す。
<code>index(str1, str2)</code>	文字列 <code>str1</code> 内の <code>str2</code> が開始する文字位置を返し、 <code>str1</code> が <code>str2</code> を含まない場合は -1 を返す。
<code>len(str)</code> <code>dlen(str)</code>	<code>str</code> の文字数を返す。 <code>dlen</code> マクロは、国際化文字の 2 バイト表現を含む文字列で動作する。
<code>m4exit(code)</code>	<code>code</code> をリターン・コードとして <code>m4</code> を終了する。
<code>m4wrap(name)</code>	他のすべての処理を完了した後で、終了前にマクロ <code>name</code> を実行する。
<code>maketemp(strXXXXXstr)</code>	引数文字列のリテラル文字列 <code>XXXXX</code> をカレントのプロセス ID に置換し、ユニークなファイル名を作成する。
<code>popdef(name)</code>	<code>name</code> の現在の定義を、 <code>pushdef</code> マクロでセーブされた定義に置換する。
<code>pushdef(name, replacement)</code>	<code>name</code> の現在の定義をセーブし、 <code>define</code> と同じ方法で <code>name</code> が <code>replacement</code> になるように定義する。
<code>shift(param_list)</code>	パラメータ・リストの位置を左に 1 つシフトし、元のリストの最初の要素を削除する。
<code>substr(string, pos, len)</code>	文字位置 <code>pos</code> で始まり、 <code>len</code> 文字長である <code>string</code> の部分文字列を返す。
<code>syscmd(command)</code>	指定されたシステム・コマンドをリターン値なしで実行する。
<code>sysval</code>	直前に使用された <code>syscmd</code> マクロからリターン・コードを獲得する。
<code>traceoff(macro_list)</code>	リスト内のすべてのマクロのトレースをオフにする。 <code>macro_list</code> が指定されなかった場合は、すべてのトレースをオフにする。

表 5-1: 組み込みの m4 マクロ (続き)

マクロ	説明
<code>traceon(name)</code>	指定されたマクロのトレースをオンにする。 <code>name</code> が指定されなかった場合は、すべてのマクロのトレースをオンにする。
<code>translit(string, set1, set2)</code>	<code>string</code> 中の <code>set1</code> のすべての文字を <code>set2</code> の対応する文字に置換する。
<code>undefine('name')</code>	<code>'name'</code> で指定されたマクロ定義を取り消す。
<code>undivert(n, n[, n...])</code>	指定された一時ファイルの内容を、現在の一時ファイルに追加する。

5.3.1 コメント文字の変更

m4 プログラムにコメントを付ける場合には、コメント文字でコメント行を区切る必要があります。省略時の値の左コメント文字は、番号記号 (#) です。省略時の値の右コメント文字は改行文字です。これらの文字で問題がある場合は、組み込みの `changeocom` マクロを次のように使用してください。

```
changeocom({,})
```

この例では左右の中カッコを新しいコメント文字にしています。元のコメント文字に戻すには、次のように `changeocom` を使用してください。

```
changeocom(##,
)
```

引数なしで `changeocom` を使用すると、コメントは無効になります。

5.3.2 引用符文字の変更

引用符文字の省略時の値は左右の一重引用符 ( ` ` および ` ` ) です。これらの文字で問題がある場合は、組み込みの `changequote` マクロで引用符文字を、次のように変更してください。

```
changequote([,])
```

この例では、左右の大カッコ用の新しい引用符で囲む文字が作成されます。元の引用符文字に戻すには、次のように引数なしで `changequote` を使用してください。

```
changequote
```

### 5.3.3 マクロ定義の取り消し

`undefine` マクロはマクロ定義を取り消します。次の例を参照してください。

```
undefine('N')
```

この例では、`N` の定義が取り消されています。定義を取り消すマクロ名は引用符で囲まなければなりません。`undefine` を使用すれば組み込みのマクロを取り消すこともできますが、組み込みのマクロを取り消すと、マクロを復旧して使用することはできなくなります。

### 5.3.4 定義されたマクロのチェック

組み込みの `ifdef` マクロは、現在マクロが定義されているかどうか確認します。`ifdef` マクロは 3 つの引数を受け入れます。最初の引数が定義されている場合、`ifdef` の値は 2 番目の引数です。最初の引数が定義されていない場合、`ifdef` の値は 3 番目の引数です。3 番目の引数がない場合、`ifdef` の値は空になります。

### 5.3.5 整数演算の使用

`m4` プログラムは、整数のみの演算を可能にする、次の組み込み関数を提供しています。

<code>incr</code>	数値引数を 1 つ増分させる
<code>decr</code>	数値引数を 1 つ減少させる
<code>eval</code>	算術式を評価する

たとえば、その値がいつでも `N` より 1 大きくなるような、変数 `N1` を作成することができます。

```
define(N,100)
define(N1,'incr(N)')
```

`eval` 関数を使用して、次の演算子を含む式を評価することができます。優先順位の高い順にリストされています。

- 単項 + (プラス), 単項 - (マイナス)
- `**` または `^` (べき乗)
- `*`, `/`, `%` (剰余)
- `+`, `-`

- == , != , < , <= , > , >=
- ! (論理否定)
- & または && (論理 AND)
- | または || (論理 OR)

カッコを使用して、必要な位置で演算をグループ化してください。式のすべてのオペランドは  $1 > 0$  などの真の関係の数値は 1 で、偽は 0 (ゼロ) です。eval の精度は 32 ビットです。たとえば、M を  $2 == N + 1$  として定義するには、次のように eval を使用してください。

```
define(N,3)
define(M,'eval(2 == N+1)')
```

テキストが単純で、マクロ名のインスタンスを含まない場合を除いて、マクロを定義するテキストは引用符文字で囲んでください。

### 5.3.6 ファイルの操作

入力に新しいファイルを挿入するには、組み込みの include マクロを次のように使用してください。

```
include(myfile)
```

この例では、include コマンドの場所に myfile の内容が挿入されています。組み込まれたファイルが読み込まれると、m4 は、それが最初の入力ファイルの一部であるかのように、そのファイルの中にマクロがあるか走査します。

include マクロで指定されたファイルにアクセスできない場合には、致命的なエラーが発生します。エラーを回避するには、代替りのフォーマットである `sinclude` (`silent include`) を使用してください。指定されたファイルにアクセスできない場合にも `sinclude` マクロは、エラーなしで続きます。

### 5.3.7 出力のリダイレクト

処理中に m4 の出力を一時ファイルにリダイレクトし、収集したデータをコマンドで出力することができます。m4 プログラムは、1~9 までの番号を付けて最大 9 個の一時ファイルを保守することができます。出力をリダイレクトするには、以下の例のように `divert` マクロを使用します。

```
divert(4)
```

このコマンドが出現すると、m4 はその出力を一時ファイル 4 の終わりまで書き出し始めます。出力点を 1~9 以外の一時的ファイルにリダイレクトする



と、m4 プログラムは出力を破棄します。この機能を利用して、m4 は入力ファイルの一部を割合することができます。出力を標準出力ストリームに返すには、`divert(0)` または引数なしで `divert` を使用してください。

処理の終了時に、m4 は、すべてのリダイレクトされた出力を、標準出力ストリームに書き出し、番号順に一時ファイルから読み込んで、一時ファイルを削除します。処理の終了前にも、番号順にすべての一時ファイルから情報を取り出したい場合には、引数なしで組み込みの `undivert` マクロを使用します。特定の順番で一時ファイルを選んで取り出す場合には、引数を付けて `undivert` を使用します。`undivert` を使用した場合、m4 は回収した一時ファイルを破棄し、回収した情報に対してはマクロを探索しません。

`undivert` の値は、切り替えられたテキストではありません。

組み込みの `divnum` マクロは、現在使用中の一時ファイルの番号を返します。`divert` マクロで出力ファイルを変更しない場合、m4 は、すべての出力を一時ファイル 0 (ゼロ) にいれます。

### 5.3.8 プログラムでのシステム・プログラムの使用

組み込みの `syscmd` マクロを使用すると、プログラム内から任意のオペレーティング・システムのプログラムを実行することができます。システム・コマンドが情報を返す場合は、その情報は `syscmd` マクロの値になり、そうでない場合はマクロの値は空になります。たとえば次のように使用します。

```
syscmd(date)
```

### 5.3.9 固有のファイル名の使用

プログラムから固有のファイル名を作成するには、組み込みの `maketemp` マクロを使用してください。マクロの引数にリテラル文字列 `XXXXXX` がある場合、m4 は `XXXXXX` を現在の処理プロセス ID に置換します。たとえば次のように指定します。

```
maketemp(myfileXXXXXX)
```

現在のプロセス ID が 23498 である場合は、この例は `myfile23498` を返します。この文字列を使用して一時ファイルに名称を付けることができます。

### 5.3.10 条件式の使用

組み込みの `ifelse` マクロは条件付きのテストを実行します。最も簡単な形式は次のような形式です。

```
ifelse(a,b,c,d)
```

この例では、2つの文字列 a および b を比較します。a および b が同一である場合、ifelse は文字列 c を返します。異なる場合は、文字列 d を返します。たとえば、compare というマクロを次のように定義して2つの文字列を比較することができます。2つの文字列が同じである場合は yes を、異なる場合は no を返します。

```
define(compare, `ifelse($1,$2,yes,no)`)
```

引用符を付けることによって、ifelse の評価が早すぎないようにします。4番目の引数を省略すると、空として処理されます。

ifelse マクロは引数をいくつでも付けることができます。したがって、多岐選択決定機能の限定された形式を提供します。たとえば次のようになります。

```
ifelse(a,b,c,d,e,f,g)
```

この文は次のフラグメントと論理的に同じです。

```
if(a == b) x = c;  
else if(d == e) x = f;  
else x = g;  
return(x);
```

最後の引数が省略された場合は、その結果は空になります。

### 5.3.11 文字列操作

組み込みの len マクロは、その引数を構成する文字列のバイト長を返します。たとえば、len(abcdef) は 6 で、len((a,b)) は 5 になります。

組み込みの dlen マクロは、文字列で表示可能な文字列の長さを返します。国際化の使用環境では、2 バイト・コードを 1 文字として表示するものもあります。したがって、文字列に 2 バイトの国際文字コードが含まれる場合、dlen の結果は len の結果と異なります。

組み込みの substr マクロは、指定された文字列 (最初の引数) から部分文字列 (2 番目の引数によって指定された文字位置で始まる) を返します。3 番目の引数は、返される部分文字列のバイトの長さを指定します。たとえば次のように指定します。

```
substr(Krazy Kat,6,5)
```

この例では、文字列 "Krazy Kat" の文字位置 6 から始まる 3 つの部分文字列 "Kat" を返します。文字列の最初の文字は位置 0 (ゼロ) にあります。3 番目

の引数が省略されるか、あるいはこの例のように文字列の長さが 3 番目の引数が必要とする長さに満たない場合、残りの文字列が返されます。

組み込みの `index` マクロは、部分文字列 (2 番目の引数) が始まる文字列 (最初の引数) 内でバイト位置すなわちインデックスを返します。部分文字列がない場合、`index` は -1 を返します。`substr` を使用すると、文字列の起点は 0 (ゼロ) になります。次の例を参照してください。

```
index(Krazy Kat,Kat)
```

この例は 6 を返します。

組み込みの `translit` マクロは、1 対 1 の文字置換すなわち字訳を実行します。最初の引数は、処理される文字列です。2 番目と 3 番目の引数は文字のリストです。文字列で見つけることのできる 2 番目の引数からの文字の各インスタンスは、3 番目の引数からの対応する文字に置換されます。次の例を参照してください。

```
translit(the quick brown fox jumps over the lazy dog,aeiou,AEIOU)
```

この例では次のように返されます。

```
thE qUICK brOwn fOx jUmPs OvEr thE lAzY dOg
```

3 番目の引数が 2 番目の引数より短い場合、3 番目の引数に存在しない 2 番目の引数に対する文字は削除されます。3 番目の引数がない場合、2 番目の引数に表示されているすべての文字が削除されます。

---

#### 注意

---

`substr`、`index`、および `translit` マクロは、1 バイトと 2 バイトの表示可能文字の区別を行わないので、国際化コードを使用している場合に予期しない結果を返すことがあります。

---

### 5.3.12 表示

組み込みの `errprint` マクロは、引数を標準エラー・ファイルに書き込みます。たとえば次のように指定します。

```
errprint ('error')
```

組み込みの `dumpdef` マクロは、現在の名前および引数として指定された項目の定義をダンプします。名前は必ず引用符で囲まなければなりません。引数を指定しない場合、`dumpdef` はすべての現在の名前および定義を表示します。`dumpdef` マクロは、標準エラー・ファイルに書き込みます。



---

## RCS や SCCS によるソース・ファイルの管理

この章では、バージョン・コントロール・システムを使用して、プログラムまたはドキュメントのソース・ファイルを適切に管理する方法について説明します。バージョン・コントロール・システムは、ドキュメント・リビジョンの保存、取り出し、ロギング、識別、およびマージを自動的に行います。バージョン・コントロールは、プログラム、ドキュメント、グラフィックス、論文などの、頻繁に改訂されるテキストを管理する場合に有効です。Tru64 UNIX では、機能の若干異なった、次の 2 つのシステムを提供します。

- リビジョン・コントロール・システム (RCS)
- ソース・コード・コントロール・システム (SCCS)

この章では、基本的なバージョン・コントロールの概念を紹介し、RCS コマンド、SCCS コマンド、およびユーティリティの使用方法について説明します。概念を紹介した後に、各システムの応用的な使用方法を説明します。

- リビジョン制御の概要 (6.1 節)
- バージョン・コントロールの概念 (6.2 節)
- ファイルの複数バージョンの管理 (6.3 節)
- バージョン・コントロール・ライブラリの作成 (6.4 節)
- RCS の使用方法 (6.5 節)
- SCCS の使用方法 (6.6 節)
- RCS コマンドと SCCS コマンドの機能比較 (6.7 節)

この章の例は、「Orpheus Authoring Tools」と呼ばれる製品のためのキットを想定して説明しています。例の中のキットも Orpheus 製品の 1 つであると考えてください。このキットはドキュメント・ビルダ (document builder) なので、キット名は DCB と省略されます。また、プロジェクト・ディレクトリは `dcb_tools` になります。

## 6.1 リビジョン制御の概要

RCS または SCCS を使用すると、ソース・ファイルをシェアード・ライブラリに保存して管理することができます。どちらのシステムにも、簡便なコマンド行インタフェースが用意されています。基本的なコマンドで、修正したソース・ファイルをチェックインしてバージョン・コントロール・ファイルを作成することができます。バージョン・コントロール・ファイルには、ソース・ファイルの各リビジョンがすべて含まれます。バージョン・コントロール・ファイルからファイルをチェックアウトして編集する場合には、システムは指定した 1 つまたは複数のリビジョンをライブラリから読み出して、編集に使用するための作業ファイルを作成します。

さらに高度なインタフェース・コマンドを使用すると、次のことができます。

- ファイルの現在の状態とそれを編集したユーザ名を確認する。
- ファイルの古いバージョンを復元する。  
各バージョンごとに、システムは、そのバージョンの変更内容、変更したユーザの名前、および変更の理由を格納しています。
- 2 人のユーザが、知らずに同一のファイルを同時に変更することによる問題を防止する。
- 複数の分岐したファイル・バージョンを保守する。  
分岐したバージョンは、分岐前のシーケンスとマージすることができます。
- ファイルの不正な変更を防止する。
- RCS では、リリースおよび構成の制御も可能である。  
各リビジョンに、シンボル名を付けることもできます。また、ファイルの状態 (リリース済、確定、テスト中など) に応じたマークを付けることもできます。

RCS と SCCS のいずれをバージョン・コントロール・システムとして採用するかは、開発環境および各ユーザに固有のリビジョン・コントロールに関する要求によって決定します。最終的には、必要とされるセキュリティと多様性の度合いによって決定することになります。表 6-1 に、それぞれのシステムで一般的に使用されているいくつかの機能の一覧を示します。

表 6-1: RCS と SCCS の機能

機能	説明
テキストの複数リビジョンの保存と取り出し	いずれのシステムの場合も、簡単な方法でファイルに対するすべての変更内容を保存し、取り出すことができる。さらに RCS では、リビジョン番号のレンジ、シンボル名、日付、著者、および状態に基づいてリビジョンを取り出すことができる。
変更内容の完全な履歴の保守	いずれのシステムの場合も、自動的に変更内容のログを生成する。いずれのシステムも、各リビジョンのテキストだけでなく、著者、チェックインの日付と時刻、および変更内容を要約したログ・メッセージを格納する。
アクセスの矛盾の解決	いずれのシステムの場合も、2 人のユーザが互いに気づかずに同じファイルを変更することを防ぐ。
リビジョンのツリー構造の保守	いずれのシステムの場合も、各ファイルでの複数の開発ラインを保守できる。
複数の改訂ファイルを矛盾解決後にマージ	いずれのシステムの場合も、2 つの開発ラインから変更内容をマージしてファイルを作成できる。また、RCS の場合は、ファイルの複数のバージョンで変更が重複している場合、ユーザに警告する。
リリースおよび構成の制御 (RCS のみ)	RCS では、リビジョンにシンボル名を付けて、モジュールの構成を簡単かつ直接的に説明できるようにしている。
各リビジョン識別の自動化	いずれのシステムの場合も、キーワードを使用して、ファイルのリビジョンに、名前、リビジョン番号、時刻、著者などを加えることができる。

## 6.2 バージョン・コントロールの概念

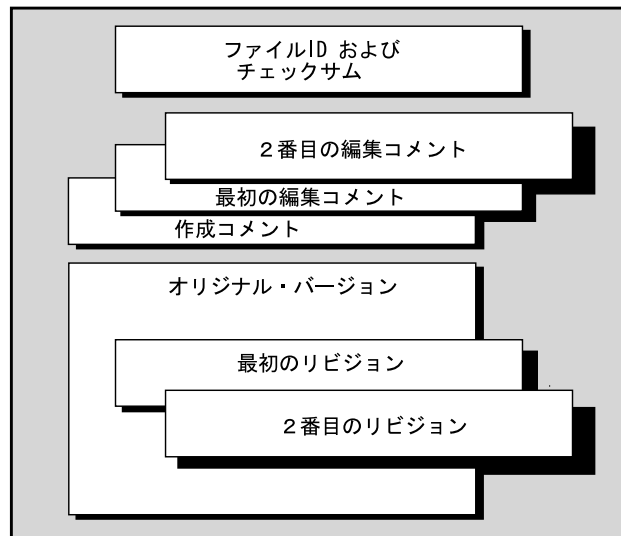
RCS と SCCS では、バージョン・コントロール・ライブラリと呼ばれる予約領域ディレクトリにファイルを保存します。各ソース・ファイルの内容は、1 つのバージョン・コントロール・ファイルとして格納されます。バージョン・コントロール・ファイルは、RCS では RCS ファイル、SCCS では s ファイルと呼ばれます。バージョン・コントロール・ファイルには、元のファイル (SCCS では g ファイル) と、元のファイルに対応するすべての変更内容 (すなわちデルタ) が含まれます。各デルタの内容は、誰が何のために変更を行ったのかを説明するテキストです。変更情報そのものは、行テキストにマークを付けた形式で格納されます。削除行または変更行は、削除のマー

クを付けられますが、実際には削除されません。新規の行は、古い行を編集したバージョンか、または適切な場所に完全に新しいテキストが挿入されてマークされたもののいずれかになります。バージョン・コントロール・システムは、必要なバージョンに至るまでのすべての削除行と追加行を適用し、それ以降のバージョンをすべて無視することによって、ファイルの任意のバージョンを復元することができます。

RCS では、ファイル名にサフィックス、`v` を付加して RCS ファイルを識別します。たとえば、`attr,v` は、`attr` という名前のソース・ファイルに対する RCS ファイルです。

SCCS では、ファイル名にプレフィックス `s.` を付加して `s` ファイルを識別します。たとえば、`s.attr` は、`attr` という名前のソース・ファイルの `s` ファイルです。図 6-1 に、典型的なバージョン・コントロール・ファイルの内容を示します。RCS ファイルおよび SCCS ファイルは、同じ種類の情報を含んでいますが、その構成方法は異なります。

図 6-1: バージョン・コントロール・ファイルの内容



ZK0456UR

バージョン・コントロール・ファイルの各リビジョンには、バージョン識別番号が割り当てられます。SCCS では、この番号は SID と呼ばれます。SCCS の識別番号には、最高 4 つの要素を含むことができます。RCS には、その他の要素も規定されています。最初の 2 つの要素は、そのリリース内の

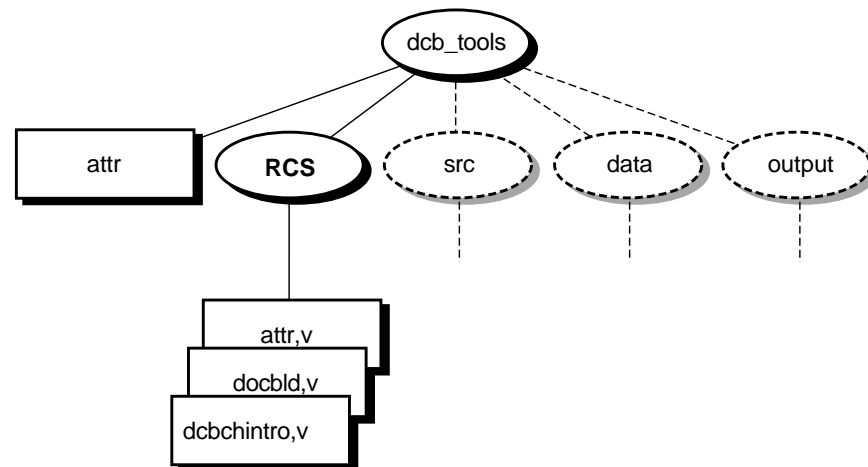


リリース番号とレベル番号であり、3 番目と 4 番目の要素には、ファイルの分岐バージョンに対する同じ項目、すなわちリリース番号とレベル番号 (分岐およびシーケンスと呼ばれる) を示します。6.3 節を参照してください。リリース識別番号は 1 から始まります。レベルの識別番号は、.1 から始まり、.1 ずつ増加します。つまり、ファイルの最初のバージョンは 1.1、2 番目のバージョンは 1.2、というようになります。6.3 節の図 6-4 に、1 つのファイルのデルタに付けられるシーケンス番号を示します。

バージョン・コントロール・ライブラリは、1 つのプロジェクトのバージョン・コントロール・ファイルがすべて保存されるディレクトリです。RCS と SCCS には、ライブラリからファイルを取り出す場合に、2 人のユーザが同一ファイルに同時にアクセスすることを防止するロック機構があります。ファイル・ロックについての詳細は、この節以降で説明します。

通常、使用するシステムによりライブラリの名前は RCS または SCCS になります。図 6-2 と図 6-3 は、プロジェクトのメイン・ディレクトリの下に、RCS ライブラリまたは SCCS ライブラリが配置された場合に、プロジェクトのディレクトリ・ツリーがどのようなようになるかを示します。

図 6-2: 典型的な RCS ライブラリ



ZK0621UR

図 6-2 は、3 つの RCS ファイルを示しています。ファイルを編集するためにライブラリからチェックアウトする場合は、図の `attr` ファイルに例示されるように、RCS はすべてのデルタを相互に関連付けて、指定されたバージョンのコピーを取り出します。RCS は、RCS ファイルを編集して、ファ

イルをチェックアウトした人の名前を挿入します。この情報は `$Locker$` キーワードに保存されます。RCS のキーワードの使用方法についての詳細は、6.5.2 項を参照してください。

RCS が SCCS と異なる点は、RCS がチェックイン時に強制的にファイルをロックするということです。2 人以上のユーザが 1 つのファイルをチェックアウトできますが、最初にチェックアウトした (ロックを保持している) 人だけが、そのファイルをライブラリにチェックインすることができます。リビジョンがロックされても、読み取り、コンパイルなどのためにチェックアウトすることはできます。ロックを行うことによって、一度に 1 人の開発者だけしか次の更新ファイルをチェックインできないように設定できます。すなわち、ロックを行うと、ロックした人 (ファイルを最初にチェックアウトした人) 以外がチェックインすることを防止します。

RCS ファイルがプライベートのファイルで、そのファイルにリビジョンを行っているユーザが他にいない場合は、RCS の完全ロック機能を解除することができます。ファイルがチェックインされる際に、RCS は、ユーザの名前を `$Locker$` キーワードから取り去ります。完全ロックが解除されると、ファイルの所有者は、チェックイン時にロックする必要がなくなります。ただし、他のユーザは必要です。次のコマンドを実行して、完全ロックの解除と設定ができます。

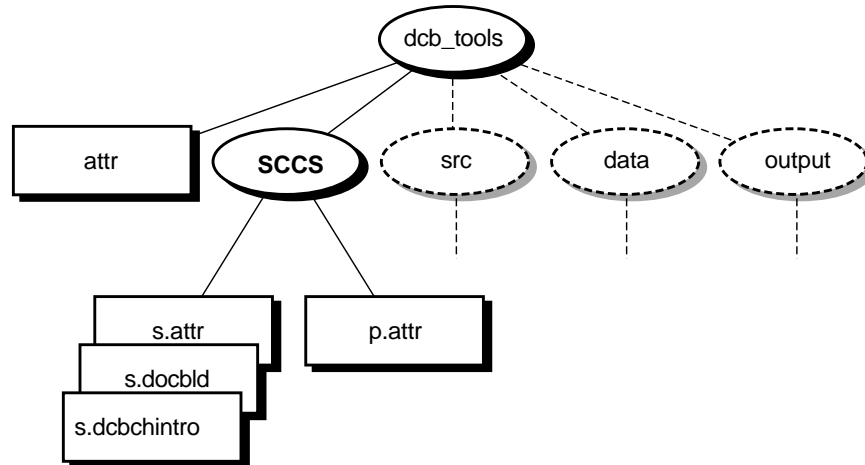
```
% rcs -U filename
```

および

```
% rcs -L filename
```

ファイル・ロックについての詳細は、6.5.3 項、6.5.5 項、および `co(1)` を参照してください。

図 6-3: 典型的な SCCS ライブラリ



ZK0457UR

図 6-3 には、SCCS ライブラリ内の、3 つの s ファイルと p.attr という名前のその他のファイルを示しています。編集を行うためにライブラリからファイルをチェックアウトする場合は、図の attr ファイルで例示されているように、SCCS は、すべてのデルタを相互に関連付けて、指定されたバージョンのコピーを取り出します。SCCS は、p ファイルと呼ばれるロック・ファイルも作成します。別のユーザが同じファイルを編集するためにチェックアウトしようとする、SCCS はファイルが編集中であることを報告し、他のユーザのアクセスを拒否します。p ファイルには、ファイル名にプレフィックスとして英字 p が付加されます。ファイルがライブラリにチェックインされると、SCCS は p ファイルを削除します。

## 6.3 ファイルの複数バージョンの管理

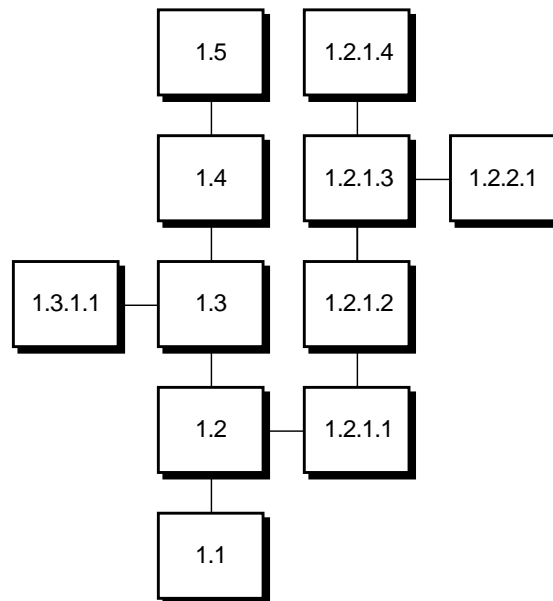
通常、ファイルのバージョンは、直線的に進化します。つまり、現在のバージョンは 1 つしかありません。この場合、ファイル識別番号は、2 つの要素で構成され、「.1」ずつ増加します。つまり、ファイルに割り当てられる最初のバージョン番号は 1.1 であり、たとえば 8 番目は 1.8 になります。

並行してプロジェクトが行われ、同じプログラムを基本として新しいバージョンを開発している場合は、同じバージョン・コントロール・ファイルを使用することができます。いろいろなバージョンが 1 つのライブラリにあるので、ツリー構造が分岐します。たとえば、2 つのチームがファイ

ルまたはモジュールの最新のリリースバージョンから別々のバージョンの開発を始めると仮定します。

2つの開発の流れが継続されるため、図 6-4 に示すように、デルタのツリー構造は複雑なものになります。

図 6-4: バージョン・コントロール・ファイルのツリー構造



ZK0458UR

分岐の1つからファイルを取り出したり、編集したりするためには、その分岐番号を指定する必要があります。図 6-4 には、(1.1, 1.2, および 1.3 などのリリース番号を含む) 主幹部とその分岐からなるバージョン・コントロール・ファイルのツリー構造が示されています。図のデルタ番号の最初の2つの要素は、分岐元のバージョン番号を反映しています。次の2つの要素は、新しい要素のバージョン番号を反映しています。

例として、2つの開発チームが 1.2 のリリース番号を持つファイルで作業を行っていると仮定します。RCS と SCCS のどちらであっても、最初にファイルをアクセスしたチームに番号 1.3 を割り当てます。バージョン・コントロール・システムは、2番目にアクセスしたチームに番号 1.2.1.1 のデルタを作成します。このデルタは、番号 1.2 から分岐する最初のデルタなので、バージョン番号の後ろ2つの要素は 1.1 になります。

2つのバージョンが開発されるにつれて、どちらのバージョンからでも分岐を作成することは可能です。たとえば、リビジョン番号 1.2.1.4 の作成後に、リビジョン番号 1.2.1.3 から新しいファイルを分岐することができます。

RCS と SCCS の分岐についての詳細および個々の例については、6.5.5 項と 6.6.5 項を参照してください。

## 6.4 バージョン・コントロール・ライブラリの作成

開発プロジェクトで使用するバージョン・コントロール・システムを選択した後、RCS ファイルまたは SCCS ファイルを格納するためのディレクトリを作成する必要があります。開発プロジェクトのサイズや複雑さによっては、ディレクトリおよびソース・ファイルの所有権と保護の設定についてシステム管理者に助言を求めることも必要になるでしょう。

ディレクトリをセットアップする際に、ディレクトリの所有権を `rccs` または `sccs` のユーザ ID に設定すると、`rccs` または `sccs` 以外のユーザがディレクトリに書き込むことを防止するようなパーミッションを設定できます。この方法により、RCS または SCCS だけがライブラリ内のファイルを直接変更できるため、セキュリティは向上します。

`sccs` コマンドを実行する場合は、図 6-3 に示すように、ライブラリのディレクトリは `sccs` という名前であればなりません。ライブラリのディレクトリが `sccs` という名前でない場合は、`sccs` コマンドに `-d` オプションを付けて、ライブラリのファイルをアクセスする必要があります。SCCS のオプションについては、表 6-8 を参照してください。RCS では、ディレクトリ名は `RCS` でなければなりません。それ以外の場合は、RCS ファイルに完全なパス（絶対パスまたは相対パス）を指定する必要があります。

## 6.5 RCS の使用方法

以降の項では、RCS の次の機能について説明します。

- RCS ライブラリへの新しいファイルの配置
- RCS によるファイル識別情報の記録
- RCS ライブラリからのファイルの取り出し
- 編集済みファイルの RCS ライブラリへのチェックイン
- 複数バージョンを持つファイルの操作

- RCS ファイルの差異の表示
- RCS ファイルのリビジョン・履歴の報告
- 構成の制御の概念

RCS システムは、テキスト・ファイルのバージョン・コントロールのタスクを補助する、1 組の UNIX コマンドのセットを提供します。それは、生産および開発の両方の環境で使用されるために設計されているため、柔軟性とファイル・アクセスの制御に優先順位が置かれています。生産環境の場合は、アクセス制御によって更新時の矛盾を検出したり、変更の重複を防止することができます。変更が頻繁に起こる開発環境の場合は、さほど厳密な制御は必要ないため、個々のプロジェクトのニーズに合わせて容易に制御を変更することができます。

RCS システムは、1 組の独立したコマンドのセットで構成されています。表 6-2 に、提供されている RCS コマンドをリストします。使用可能なコマンド・オプションについての詳細は、該当するリファレンス・ページを参照してください。

表 6-2: RCS コマンド機能の要約

コマンド	機能								
ci	リビジョンのチェックイン。作業ファイルの内容を、対応する RCS ファイルの新しいリビジョンとして保存する。								
	<table><tr><th>オプション</th><th>機能</th></tr><tr><td>-u または -1</td><td>これらのオプションのいずれかを使用すると、チェックイン時に作業ファイルは削除されない。</td></tr><tr><td>-r</td><td>チェックインするファイルにリビジョン番号を割り当てる。</td></tr><tr><td>-k</td><td>チェックインされたファイル内の識別マーカを検索し、新しいリビジョンを割り当てる。</td></tr></table>	オプション	機能	-u または -1	これらのオプションのいずれかを使用すると、チェックイン時に作業ファイルは削除されない。	-r	チェックインするファイルにリビジョン番号を割り当てる。	-k	チェックインされたファイル内の識別マーカを検索し、新しいリビジョンを割り当てる。
オプション	機能								
-u または -1	これらのオプションのいずれかを使用すると、チェックイン時に作業ファイルは削除されない。								
-r	チェックインするファイルにリビジョン番号を割り当てる。								
-k	チェックインされたファイル内の識別マーカを検索し、新しいリビジョンを割り当てる。								
co	リビジョンのチェックアウト。リビジョン番号、日付、著者、および状態の属性に応じてリビジョンを検索する。常に識別マーカ (キーワード) を展開する。								
	<table><tr><th>オプション</th><th>機能</th></tr><tr><td>-1</td><td>ファイルのチェックアウト時にリビジョンをロックし、複数のユーザが同じファイルに作業を行った場合の変更の重複を防止する。</td></tr></table>	オプション	機能	-1	ファイルのチェックアウト時にリビジョンをロックし、複数のユーザが同じファイルに作業を行った場合の変更の重複を防止する。				
オプション	機能								
-1	ファイルのチェックアウト時にリビジョンをロックし、複数のユーザが同じファイルに作業を行った場合の変更の重複を防止する。								

表 6-2: RCS コマンド機能の要約 (続き)

コマンド	機能
ident	ファイルから識別マークを取り出し、プリントする。識別マーク (キーワード) は常に co によって展開される。
rcs	RCS ファイル属性の変更。変更は管理者が行わなければならない。アクセス・リストの変更、ファイルのロック属性の変更、状態属性と文字によるリビジョン番号の設定、記述の変更、およびリビジョンの削除を行う。リビジョンが削除できるのは、その他の分岐の起点となっていない場合のみ。
<div>オプション      機能</div>	
<div>-L      完全なファイル・ロック・モードを設定する。これは、RCS ファイルの所有者が、チェックイン時に RCS ファイルをロックしなければならないことを意味する。省略時の設定はシステム管理者が決定する。</div>	
<div>-U      不完全なファイル・ロック・モードを設定する。これは、ファイルの所有者が、チェックイン時にファイルをロックする必要がないことを意味する。省略時の設定はシステム管理者が決定する。</div>	
rcclean	作業ディレクトリを整理する。チェックアウトされたが、変更が行われなかった作業ファイルを削除する。
rcsdiff	コマンドを使用して、2 つのリビジョンを比較し、その違いをプリントする。比較されるリビジョンのうちの 1 つはチェックアウトされているものでもかまわない。このコマンドは、変更内容の検索に有効である。
rcsfreeze	構成を凍結する。すべての RCS ファイルの指定されたリビジョンに、同一名で、文字によるリビジョン番号を割り当てる。このコマンドは、構成の正確な記録に有効である。
rcsmerge	共通の祖先に基づいて、2 つのリビジョン、rev1 および rev2 をマージする。3 つのファイルの比較は、行の 1 部分に対して、3 つのリビジョンがすべて同じか、2 つのリビジョンが同じか、あるいは、3 つのリビジョンがすべて違うかを判断することによって行われる。重複する変更内容にはフラグを立て、ユーザに報告する。
<div>オプション      機能</div>	
<div>-p      マージしたファイルの結果を標準出力にプリントする。このオプションを指定しないと、マージした結果のファイルは作業ファイルに重ね書きされる。</div>	

表 6-2: RCS コマンド機能の要約 (続き)

コマンド	機能						
rlog	ログ・メッセージを読み取る。RCS ファイルのログ・メッセージおよびその他の情報をプリントする。たとえば、RCS ファイル名、作業ファイル名、ヘッド (主幹部の最新のリビジョン番号)、省略時の分岐、アクセス・リスト、ロック、シンボル名、リビジョン番号、および記述テキスト。						
	<table><tr><th>オプション</th><th>機能</th></tr><tr><td>-h</td><td>RCS ファイル名、作業ファイル名、ヘッド、省略時の分岐、アクセス・リスト、ロック、シンボル名、およびサフィックスのみをプリントする。</td></tr><tr><td>-t</td><td>-h オプションでプリントする情報の他に、記述テキストもプリントする。</td></tr></table>	オプション	機能	-h	RCS ファイル名、作業ファイル名、ヘッド、省略時の分岐、アクセス・リスト、ロック、シンボル名、およびサフィックスのみをプリントする。	-t	-h オプションでプリントする情報の他に、記述テキストもプリントする。
オプション	機能						
-h	RCS ファイル名、作業ファイル名、ヘッド、省略時の分岐、アクセス・リスト、ロック、シンボル名、およびサフィックスのみをプリントする。						
-t	-h オプションでプリントする情報の他に、記述テキストもプリントする。						

6.5.1 RCS ライブラリへの新しいファイルの配置

ci コマンドを実行すれば、新しいファイルをライブラリに置くことができます。次の例では、ライブラリの親ディレクトリにユーザがいて、attr ファイルをライブラリに追加すると仮定しています。

```
% ci attr
RCS/attr,v <---- attr
enter description, terminated with single '.' or end of file:
>> Orpheus Authoring Tools attr command
>> .
initial revision: 1.1
done
```

ci コマンドは、RCS ファイル attr,v を作成し、その中に attr をリビジョン 1.1 として保存します。ci コマンドは、記述を入力するように要求してきます。記述とは、ファイルの内容の概要です。これ以降にチェックイン・コマンドを実行すると、ログ・エントリの入力を要求してきます。ログ・エントリとは、変更内容の要約のことです。

次に示すように、1 回の操作で複数のファイルを入力することができます。

```
% ci attr docbld dcb.ch-intro
```

6.5.2 RCS によるファイル識別情報の記録

RCS システムでは、ファイル識別情報を提供するためのキーワードまたは ID マーカをソース・ファイルに含めるための構文を提供しています。ID マーカは、キーワードをドル符号 (\$) で囲んだものです。RCS ライブラリが



らファイルを取り出す際に、RCS は、キーワードをファイル名またはリビジョン番号などの適切な情報に置き換えて展開します。

RCS では、ファイルの任意の場所にキーワード・マーカをリテラル文字列またはコメントとして使用し、リビジョンを確認することができます。たとえば、テキスト・ファイル内にマーカ `$Header$` を置いた場合は、(co コマンドを実行すると) RCS は、このキーワードを次のような情報に置き換えます。

```
$Header: full_pathname/filename rev_num yyyy/mm/dd HH:mm:ss author
state $
```

表 6-3 に、RCS キーワードとそれに対応する値をリストします。

表 6-3: RCS ID キーワード

キーワード	説明
<code>\$Author\$</code>	当該リビジョンをチェックインしたユーザのログイン名。
<code>\$Date\$</code>	当該リビジョンをチェックインした日付と時刻。
<code>\$Header\$</code>	RCS ファイルの完全パス名、リビジョン番号、日付、著者、状態。ロックされている場合は、ロックしたユーザ名からなる標準のヘッダ。
<code>\$Id\$</code>	RCS ファイル名がパスなしで示される以外は標準ヘッダと同じ。
<code>\$Locker\$</code>	当該リビジョンをロックしているユーザのログイン名。ロックしていない場合は空。
<code>\$Log\$</code>	チェックイン時に出力されるログ・メッセージ。先頭に RCS ファイル名、リビジョン番号、著者、および日付を含むヘッダが付く。既存のログ・メッセージは置換されず、代わりに、新しいログ・メッセージが <code>\$Log:...\$</code> の後に挿入される。
<code>\$RCSfile\$</code>	パスなしの RCS ファイルの名前。
<code>\$Revision\$</code>	当該リビジョンに付けられたリビジョン番号。
<code>\$Source\$</code>	RCS ファイルの完全パス名
<code>\$State\$</code>	<code>rsc</code> または <code>ci</code> の <code>-s</code> オプションによって当該リビジョンに付けられた状態。

`ident` コマンドは、オブジェクト・ファイルやダンプなど、どのようなファイルからでもキーワードを見つけて、抽出することができます。`ident` コマンドは、指定したファイル内で、パターン `$keyword:...$` の出現をすべて検索します。たとえば、C プログラム `myfile.c` に、次の情報が含まれていると仮定します。

```
char resid [] = "$Header: Header information$"
```

ident コマンドを実行すると、次のようにプリントされます。

```
myfile.c : $Header: Header information$"
```

RCS でのキーワードの使用についての詳細は、co(1) を参照してください。

### 6.5.3 RCS ライブラリからのファイルの取り出し

RCS ファイルから、特定のファイルのリビジョンを取り出すには、co コマンドを実行して RCS ライブラリからそのリビジョンをチェックアウトします。co コマンドは RCS ファイルから当該リビジョンを取り出して、それに対応する作業ファイルに保存します。

RCS ファイルのリビジョンは、ロックして、あるいはロックせずにチェックアウトすることができます。リビジョンをロックすると、更新の重複を防止できます。編集以外 (たとえば、参照または実行) のためにファイルをチェックアウトする場合は、リビジョンをロックする必要はありません。編集のためにチェックアウトし、後でチェックインをするリビジョンは、必ずロックしておかなければなりません。次に例を示します。

```
% cd /usr/projects/dcb_tools

% co -u attr
RCS/attr,v ----> attr
revision 1.6 (unlocked)
done
```

このコマンドは、RCS ファイルの最新のバージョンの (キーワード情報を含んだ) コピーを作成し、現在のディレクトリ (この例では /usr/projects/dcb\_tools) に取り出します。-u オプションを付けると、RCS はファイルをロックしません。任意の古いバージョンのコピーを取り出すには、-r オプションを使用します。たとえば、現在のバージョンが 1.8 のファイルのバージョン 1.5 を取り出す場合は、次のようなコマンドを実行します。

```
% cd /usr/projects/dcb_tools

% co -r1.5 attr
RCS/attr,v ----> attr
revision 1.5
done
```

1 回の co コマンドで、複数のファイルを取り出すこともできます。たとえば、次のように実行します。

```
% co attr unstamp
RCS/attr,v ----> attr
revision 1.5
done
```

```
RCS/unstamp,v ----> unstamp
revision 1.2
done
```

#### 6.5.4 編集済みファイルの RCS ライブラリへのチェックイン

1 つまたは複数の編集済みファイルを戻すには、`ci` コマンドを実行します。このコマンドは、各作業ファイルの内容を、対応する RCS ファイルに置き換えます。通常、RCS では、ライブラリに再配置されるリビジョンが元のファイルと異なっているかどうかをチェックし、ユーザに警告します。

また、`ci` コマンドはチェックイン時に作業ファイルを削除してしまうため、チェックアウト処理を暗黙に実行するために作業ファイルを保持するには、`-l` オプションまたは `-u` オプションを使用します。現在のリビジョンをセーブした後に編集を続ける場合は、`-l` オプションまたは `-u` オプションとともに `ci` コマンドを使用してください。

#### 6.5.5 複数バージョンを持つファイルの操作

6.3 節では、バージョン・コントロール・システムの分岐概念の概要を説明しています。この項の解説では、RCS が複数ファイルの分岐を処理する例をいくつか示します。

RCS は、ファイル・リビジョンをデルタのツリー構造として配置します。リビジョン・ツリー構造の各ファイルに含まれる情報は、リビジョン番号、チェックイン時刻および日付、著者の識別情報、ログ・エントリ、状態、ならびに実際のテキストです。これらのファイル属性はすべて、リビジョンがライブラリにチェックインされる際に決定されます。状態属性は、リビジョンの状態を示します。チェックイン時には「テスト中」に設定されますが、後で確定またはリリース済に変えることもできます。

`ci` コマンドは、通常は 1.1 の番号を割り当てられたルート・リビジョンを持ったリビジョン・ツリー構造を作成します。リビジョン番号を明示しない限り、`ci` は 1.2, 1.3, 1.4 というように直前のリビジョンのレベル番号を増分して新しいリビジョン番号を付けます。新しいリリースを開始するには、次の 2 つのコマンドのいずれかを使用します。

```
% ci -r2.1 unstamp
```

または

```
% ci -r2 unstamp
```

このアクションによって、新しいリビジョンに 2.1 という番号が割り当てられます。-r オプションを付けずにファイルをライブラリにチェックインすると、ファイルのリビジョンが新しくなるにつれ、自動的に 2.2, 2.3 のように番号が割り当てられます。

2 つの開発チームが unstamp コマンドのリビジョン番号 1.2 から始まる別々なリリースで開発を始めると仮定します。この時点では、両方のチームとも、-l オプションを付けて co コマンドを実行し、最新リビジョンをチェックアウトすることができます。次のように入力します。

```
% co -l unstamp
```

ファイルの編集後に、最初のチームは、ci コマンドを使用してファイルをチェックインすることができます。RCS は、新しいリビジョン番号が 1.3 であるという警告を発します。次の例のようになります。

```
% ci unstamp
RCS/unstamp,v <---- unstamp
new revision 1.3; previous revision 1.2
enter log message:
(terminate with a ^D or single '.')
>> Changed defaults check.
>> .
done
```

2 番目のチームが同じアクションでファイルをチェックインしようとする、RCS は次のメッセージを表示します。

```
RCS/unstamp,v <----- unstamp
ci error: no lock set by user-name
```

この時点で、2 番目のチームは、次のように ci コマンドを使用して分岐を作成することができます。

```
% ci -r1.3.1 unstamp
```

このアクションの結果、リビジョン番号 1.3.1.1 を持つ分岐が作成されます。この分岐に沿って開発を続けるために、2 番目のチームは、現在の分岐のリビジョン番号を、以降に行うすべてのファイルのチェックアウトに使用する必要があります。次に例を示します。

```
% co -r1.3.1.1 unstamp
```

RCS での新しい分岐の作成は、ci コマンドを使用して実行されます。特定の分岐に従って開発を続けるには、co コマンドで -r オプションを使用します。

ここまでの解説では、RCS が各ファイルのリビジョンを処理する方法について説明しました。RCS システムでは、ファイルをグループまたはセット単位

で指定して作業することもできます。RCS でのファイル構成についての詳細は、6.5.8 項を参照してください。

### 6.5.6 RCS ファイルの差異の表示

`rcsdiff` コマンドを実行すると、RCS ファイルのバージョン間の差異を確認できます。

`rcsdiff` コマンドは、`diff(1)` を実行して、任意の RCS ファイルの 2 つのリビジョンを比較します。たとえば、(1.8、編集すると 1.9 になる) `attr` ファイルの最新バージョンと直前のバージョン (1.7) の差異を見つけるには、次のコマンドを実行します。

```
% rcsdiff -r1.7 attr
=====
RCS file: RCS/attr,v
retrieving revision 1.7
diff -r1.7 attr
31d30
<# and is version linked to the docbld command
```

`attr` ファイルのバージョン 1.3 と 1.4 の差異を確認する場合は、次のコマンドを実行します。

```
% rcsdiff -r1.3 -r1.4 attr
=====
RCS file: RCS/attr,v
retrieving revision 1.3
retrieving revision 1.4
diff -r1.3 -r1.4
5a6
< uts=-04
---
> uts=-05
```

### 6.5.7 RCS ファイルのリビジョン・ヒストリの報告

`rlog` コマンドを実行して、ファイルのリビジョン・ヒストリを調べます。たとえば、`rlog` コマンドによって次のような詳細な情報が提供されます。

```
% rlog unstamp
RCS file:      RCS/unstamp,v; Working file:      unstamp
head:          1.2
branch:
locks:         ; strict
access list:
symbolic names:
comment leader: "# "
total revisions: 2; selected revisions: 2
description:
```

```
unstamp source file

revision 1.2
date: 92/06/09 15:51:16; author:gunther; state:Exp; lines
added/del:
Fixed copyright notice

revision 1.1
date: 92/06/09 15:49:16; author:gunther; state:Exp;
Initial revision
```

rlog コマンドを実行した場合に、使用可能となる情報のタイプと量に注意してください。RCS は、各 RCS ファイルに対して次の情報をプリントします。

- RCS ファイル名
- 作業ファイル名
- ヘッド (主幹部の最新のリビジョン番号)
- 省略時の分岐
- アクセス・リスト
- ファイルのロック
- シンボル名 (ある場合)
- サフィックス
- リビジョンの総数
- プリントされたリビジョンの番号
- 記述テキスト

この情報の後には、各分岐ごとに、選択されたリビジョンのエントリが逆の年代順で続いています。オプションを指定せずに入力した場合、rlog は、選択したファイルの情報をすべてプリントします。rlog コマンドの出力を制限するためのオプションの使用方法についての詳細は、rlog(1) を参照してください。

### 6.5.8 構成の制御の概念

RCS での構成とは、ファイル・リビジョンのグループまたはセットを指します。その中では、各リビジョンは、異なるファイル・リビジョンのグループから構成されています。ファイル・リビジョンは、一定の基準に従って選択 (チェックアウト) されます。次の選択基準に基づいて RCS ライブラリからファイルのセットをチェックアウトすることができます。

- 省略時の選択

省略時の設定では、RCS は、すべてのファイルの中で最新のリリースを選択します。たとえば、次のコマンドを使用すると、ライブラリ中の全 RCS ファイルの省略時の分岐から、最新のリリースを検索します。

```
% co *,v
```

- リリースに基づく選択

リリース番号または分岐番号を指定して、リリースまたは分岐の中の最新のリリースを選択することもできます。たとえば、次のコマンドは、各 RCS ファイルからリリース番号が 2 である最新のリリースを検索します。

```
% co -r2 *,v
```

- 状態および作成者による選択

RCS では、状態属性に従ってファイルを選択することができます。たとえば、リリース番号が 2 で、状態属性が「リリース済」の最新リリースを検索するには、次のコマンドを入力します。

```
% co -r2 -sReleased *,v
```

-w オプションを使用すると、作成者によってリリースを選択することもできます。

- 日付による選択

日付によってリリースを選択することもできます。システム全体のリリースが完了し、現在の時刻は 6 月 15 日 2:00 p.m であると仮定します。次のコマンドは、-d オプションで切り捨てる日付を 6 月 15 日と指定し、そのリリースのファイルをすべてチェックアウトします。

```
% co -d "June 15,2:00 pm" *,v
```

- 名前による選択 (シンボル名を使用)

RCS では、リリースと分岐にシンボル名を付けることができます。大規模なシステムの開発プロジェクトでは、すべてのグループから適切なリリースをすべて集めるには、1 種類のリリース番号または日付では不十分なことがあります。

たとえば、1 つのサブシステムのリリース 2 を別のサブシステムのリリース 10 と結合する必要があると仮定します。おそらく、これらのリリースの作成日は異なっています。したがって、この場合は 1 種類のリリース番号または日付を co コマンドに指定しても、適切なリリースを

得ることはできません。文字によるリビジョン名を使用すると、この問題を回避できます。これは、各 RCS ファイルが、数字のリビジョン番号にマップされたシンボリック名のセットを持つことができるためです。

たとえば、ファイル `attr,v` のリリース番号 2 およびファイル `unstamp,v` のリビジョン番号 10.2 に、シンボル名として IFT2 を設定したと仮定します。この場合、次のように 1 回の `co` コマンドで、`attr,v` からリリース 2 の最新リビジョンを、`unstamp,v` からリビジョン 10.2 の最新リビジョンを検索することができます。

```
% co -rIFT2 attr,v unstamp,v
```

`rcsfreeze` コマンドを実行すれば、特定の構成に含まれる一連の RCS ファイルに、文字によるリビジョン名を割り当てることができます。主幹部にある各 RCS ファイルの最新リビジョンに、固有の文字によるリビジョン名を付けるには、新しいバージョンをチェックインするたびに `rcsfreeze` コマンドを実行します。RCS ファイルにシンボル名を付ける際についての詳細は、`rcsfreeze(1)` を参照してください。

大規模なソフトウェア開発プロジェクトでは、1 つのコマンドですべてのリビジョンを取り出す能力が、構成管理を、組織的かつ効率的な作業にしてくれます。

## 6.6 SCCS の使用方法

SCCS システムは、いくつかの独立したコマンドで構成されています。それぞれのコマンドは、単独で 사용할 こともできます。`sccs` コマンドは統合されたインタフェースです。最も一般的な SCCS コマンドの使用方法を簡素化したもので、複数のコマンドの動作を結合して、いくつかの機能を追加しています。ただし、個々のコマンドのすべての機能をサポートしているわけではありません。

`sccs` コマンドの形式には必ず、キーワード `sccs` と、`edit` のような関数の名前が含まれます。その後ろに、オプションと処理するファイルの名前を指定します。表 6-4 は `sccs` コマンドです。表の後の項で、これらのコマンドのいくつかについて詳しく説明します。この解説では、必要な場合を除きコマンド・オプションが省略されています。また、独立した下位レベル・コマンドが存在するコマンドについては、表中で示されています。個々のすべてのコマンドの一覧は、表 6-9 に示されています。使用方法についての詳細な情報、およびオプションについての説明は、各リファレンス・ページを参照してください。



表 6-4: sccs コマンド機能の要約

コマンド名	下位レベル	説明
admin	あり	s ファイルの作成または既存の s ファイルの特性の変更。
check	なし	編集中のファイルおよび編集しているユーザ名の報告。info と異なり, check は, 意味のある終了ステータスを返す。また, 編集中のファイルがない場合は, レポートを表示しない。
clean	なし	指定された s ファイルから復元可能なすべてのファイルを, ディレクトリから削除する。
create	なし	g ファイルを削除せずに s ファイルを作成する。
deledit	なし	delta を行った後, 同じファイルに対して edit を行う。
delget	なし	delta を行った後, 同じファイルに対して get を行う。
delta	あり	編集済みの g ファイルをライブラリにチェックインさせる。その際, 変更内容および履歴を記録する。p ファイルを削除する。
diffs	なし	編集のためにチェックアウトされた g ファイルと, s ファイルから復元された古いバージョンを比較する。
edit	なし	編集のために s ファイルをチェックアウトする。g ファイルを再作成してユーザのディレクトリに置く。p ファイルを作成する。
fix	なし	最新のデルタを削除し, 再編集のために g ファイルを取り出す。rm del に続けて edit を入力することと同じ。
get	あり	g ファイルの再作成。通常は, 編集以外の目的に使用する。編集のために g ファイルを再作成する場合は, 通常, sccs get -e と同じ機能を持つ sccs edit コマンドを実行する。
help	あり	コマンド名または SCCS メッセージ番号を指定すると, その項目についての情報を表示する。独立したコマンド形式は, sccshelp である。各 SCCS メッセージには, 識別コードがある。たとえば, 「no ID keywords」メッセージの識別コードは cm7 である。sccs help cm7 コマンドは, このエラーについての記述を表示する。sccshelp delta コマンドは, delta コマンドの構文ダイアグラムを返す。
info	なし	編集中のファイルと編集しているユーザの名前を報告する。

表 6-4: sccs コマンド機能の要約 (続き)

コマンド名	下位レベル	説明
print	なし	指定した 1 つまたは複数のファイルのリビジョン・ヒストリを表示する。さらに、各行の先頭に ID 情報を追加した SCCS ファイルを表示する。
prs	あり	指定した 1 つまたは複数のファイルのリビジョン・ヒストリを表示する。
prt	なし	prs と同じ。
rmDEL	あり	指定の s ファイルの指定の分岐から最新のデルタを削除する。
sccsdiff	あり	s ファイルの 2 つのバージョンを比較する。s ファイル名は、明示的に指定しなければならない。
tell	なし	編集中のファイルについての報告をする。info と異なり、ファイル名だけが報告される。
unedit	なし	g ファイルの編集を打ち切る。p ファイルを削除して、他のユーザがチェックアウトできるように s ファイルを解放する。作業ディレクトリに g ファイルが存在する場合は、sccs unedit は、それを削除し、s ファイルに対して get コマンドを実行する。g ファイルが存在しない場合は、get コマンドは実行されない。unget コマンドと同じ。
what	あり	ファイル中の SCCS ID パターンを探索し、それに続くテキストを表示する。

6.6.1 SCCS ライブラリへの新しいファイルの配置

sccs create コマンドを実行して、新しいファイルをライブラリに配置することができます。次の例では、現在ライブラリの親ディレクトリから attr ファイルをライブラリに追加すると仮定します。

```
% sccs create attr
attr:
1.1
141 lines
```

指定するファイル名にプレフィックス s を含めてはなりません。SCCS では自動的に適用されるためです。

1 回の操作で複数のファイルを入力することができます。次に例を示します。

```
% sccs create attr docbld dcb.ch-intro
```

ライブラリに *s* ファイルを作成した後に、`sccs create` コマンドは元のファイル名にプレフィックスとしてコンマを付加します。たとえば、`attr` は、`attr` になります。このアクションは、キーワード (もしあれば) を展開しないままで、元の *g* ファイルを保持します。さらに、SCCS では、`get` コマンドを実行してファイルのコピーを取り出すことができます。取り出されたバージョンは、配布することができます。

次の構文を使用して、`sccs admin -i` コマンドでライブラリにファイルを挿入することもできます。

```
sccs admin -i [[path/] [input-file] [[path/] [s-filename]
```

次に例を示します。

```
% sccs admin -iunstamp unstamp
```

*path/input-file* には、入力ファイルを指定します。このファイル名のいかにかわらず、*s* ファイル名は、「*s.s-filename*」となります。`-i` オプションと *path/input-file* の間にはホワイト・スペースを入れな  
いください。*s-filename* にプレフィックス *s* を付けてはなりません。SCCS では自動的に適用されます。`admin -i` コマンドを実行すると、元の *g* ファイル名の書き換えと、キーワードを展開したバージョンの取り出しを行いません。`admin` コマンドの使用方法についての詳細は、6.6.8 項を参照してください。

短いシェル・スクリプトで複数のファイルを入力するには、`admin -i` オプションを使用します。次のコマンド行の例は、`csh` で実現されます。

```
% foreach x (attr docbld dcb.ch-intro)  
? sccs admin -i$x $x  
? end
```

## 6.6.2 SCCS によるファイル識別情報の記録

SCCS システムでは、ソース・ファイルに ID キーワードを指定するための構文を用意しており、ファイル識別情報を提供します。ID キーワードは、パーセント記号 (%) で囲まれた単一の英字で構成されます。編集以外の目的でファイルを取り出す場合は、SCCS は、各 ID キーワードを SID またはファイル名などのような適切な情報に置換して展開します。表 6-5 は SCCS ID キーワードのリストです。

表 6-5: SCCS ID キーワード

キーワード	説明
%B%	取り出された g ファイルの分岐番号。
%C%	g ファイルの現在の行の番号。プログラムから出力されたメッセージを確認することを目的としている。
%D%	get コマンドにより取り出された g ファイルの取り出し日付。フォーマットは, <i>yy/ mm/ dd</i> 。
%E%	デルタの作成日。フォーマットは, <i>yy/ mm/ dd</i> 。
%F%	s ファイルのファイル名。
%G%	デルタの作成日。フォーマットは, <i>mm/ dd/ yy</i> 。
%H%	get コマンドにより取り出された g ファイルの取り出し日付。フォーマットは, <i>mm/ dd/ yy</i> 。
%I%	取り出されたファイルに適用された最も高い SID。
%L%	取り出された g ファイルのレベル番号。
%M%	現在のモジュール (ファイル) 名。たとえば, <i>prog.c</i> 。
%P%	s ファイルの完全パス名。
%Q%	s ファイルの q フラグの値。
%R%	取り出された g ファイルのリリース番号。
%S%	取り出された g ファイルのシーケンス番号。
%T%	get コマンドにより取り出された g ファイルの取り出し時刻。フォーマットは, <i>hh: mm: ss</i> 。
%U%	デルタの作成時刻。フォーマットは, <i>hh: mm: ss</i> 。
%W%	%Z%%M% Tab %I% の簡易入力。
%Y%	admin コマンドで設定される t フラグの値のための位置保持記号。SCCS では無効。
%Z%	what コマンドが検出できるようにするための文字列 @(#) に展開される位置保持記号。

SCCS は、ファイル上の任意の場所にある ID キーワードを処理します。SCCS `what` コマンドの目的は、ファイル中の展開された ID キーワードを検出して表示することです。what コマンドは、文字列 @(#) を含む行を検索し、それらの行を表示します。文字列 @(#) は、%Z% キーワードまたは %W% 簡易入力キーワードで生成されます。what コマンドの例を、次に示します。

```
% what /usr/bin/attr
/usr/bin/attr:
    attr 1.8 of 4/15/92
```

この例で表示された行は、シェル・スクリプトの一部で、次のようにコード化されています。

```
# SCCSID: %Z%M% %I% of %G%
```

ファイルに ID キーワードが含まれていない場合、SCCS はファイルがライブラリに配置された時点と、そのファイルが取り出された時点に、このことを報告します。ファイルに `i` フラグを設定すると、ID キーワードが含まれていないということが致命的なエラーになるように指定することができます。ファイル・フラグについての詳細は、6.6.8 項を参照してください。`i` フラグの目的は、`delta` コマンドが、展開されたキーワードを持つ、あるいはキーワードがない `g` ファイルと `s` ファイルとをマージするのを防ぐことです。

### 6.6.3 SCCS ライブラリからのファイルの取り出し

SCCS ライブラリからファイルを取り出すのには 2 つの理由があります。配布など編集以外で使用するため、および編集のためです。

ファイルは、直線的に変化するバージョン・ヒストリとして編集すること、あるいは、分岐したツリー構造を作成して編集することもできます。同じ `s` ファイルに、複数の平行したバージョンと一緒に保存される場合のツリー構造の作成方法を、6.6.5 項に説明します。

#### 6.6.3.1 編集以外の目的でのファイルの取り出し

編集以外の目的で SCCS ファイルを取り出す場合は、`sccs get` コマンドを実行します。次に例を示します。

```
% cd /usr/projects/dcb_tools

% sccs get attr
1.8
126 lines
```

このコマンドは、展開された SCCS キーワード (表 6-5 を参照) を持つ、`s` ファイルの最新のバージョンのコピーを作成し、それを現在のディレクトリ (この例では `/usr/projects/dcb_tools`) に置きます。任意の古いバージョンのコピーを取り出すには、`-r SID` オプションを使用します。たとえば、現在バージョン 1.8 であるファイルのバージョン 1.5 を取り出すには、次のコマンドを実行します。

```
% cd /usr/projects/dcb_tools
```

```
% sccs get -r1.5 attr
1.5
128 lines
```

SCCS ファイルのより複雑なツリー構造の管理については、6.6.5 項を参照してください。

-p オプションを使用すると、ファイルを取り出した後で、暗黙に s ファイルと同じ名前の g ファイルを作成する代わりに、標準出力に書き込むことができます。詳細については、get(1) を参照してください。

### 6.6.3.2 編集のためのファイルの取り出し

ファイルを編集する場合は、sccs edit コマンドでライブラリからファイルをチェックアウトします。たとえば、次のようにします。

```
% sccs edit attr
1.8
new delta 1.9
126 lines
```

sccs edit コマンドは、SCCS キーワードを展開しないで s ファイルの最新のバージョンのコピーを作成し (表 6-5 を参照)、編集用として現在のディレクトリに格納します。sccs edit コマンドはまた、編集用にファイルをチェックアウトしたユーザを識別する p ファイルも作成します。

sccs info コマンドを実行すれば、ファイルの状態をチェックすることができます。次に例を示します。

```
% sccs info
unstamp: is being edited: 1.4 1.5 gunther 99/03/07 10:42:19
```

get -e コマンドを実行して、編集のためにファイルを取り出すこともできます。

### 6.6.3.3 複数ファイルおよび新しいリリースの管理

1 回の get コマンドまたは edit コマンドで、複数のファイルを取り出すことができます。次に例を示します。

```
% sccs get attr unstamp
SCCS/s.attr:
1.8
126 lines
SCCS/s.unstamp:
```

```
1.2
55 lines
```

1 つまたは複数のファイル名の代わりに `SCCS` という名前を指定した場合は、`SCCS` はライブラリのすべての `s` ファイルを取り出します。

ファイルの新しいリリースを作成するには、`sccs edit` コマンドに `-r` オプションを指定して、新しいリリース番号を取り出します。たとえば、次のコマンドでは `docbld` ファイルのリリース 2 が作成されます。

```
% sccs edit -r2 SCCS
SCCS/s.docbld:
1.50
new delta 2.1
1042 lines
SCCS/s.dcb_defaults:
1.50
new delta 2.1
63 lines
SCCS/s.dcb_diag.sed:
1.50
new delta 2.1
188 lines
```

#### 6.6.4 編集済みファイルの `SCCS` ライブラリへのチェックイン

ライブラリに編集済みファイルを再配置するには、`sccs delta` コマンドを実行します。`SCCS` はコメントの入力を求めます。次に例を示します。

```
% sccs delta attr

Comments? (^D to end)

Changed defaults check.  Now looks only for "flc="
```

**Ctrl/D**

```
1.9
4 inserted
4 deleted
124 unchanged
```

1 つまたは複数のファイル名の代わりに `SCCS` という名前を指定した場合には、`SCCS` は、ライブラリのすべての `s` ファイルに対して `delta` コマンドを実行します。この機能は、同様の指定をした `edit` コマンドと一緒に使用すると、セットのうちいくつかのファイルだけを編集し、セット全体を同じ

バージョンで保存する場合に有効です。SCCS は一度だけコメントを要求し、各ファイルに同じコメントが適用されます。

`sccs delget` コマンドおよび `sccs deledit` コマンドは、`delta` を実行し、`get` または `edit` の各処理を実行します。

### 6.6.5 複数バージョンを持つファイルの操作

6.3 節に、バージョン・コントロール・システムの分岐概念の概要を説明しました。この項では、ファイルの複数バージョンの分岐が SCCS によって処理される例を、いくつか示します。

2 つの開発チームが、`unstamp` ファイルの `SID1.2` を分岐点として、別々のバージョンで開発を始める場合を例にします。分岐を使用するためには、`sccs admin -fb` コマンドを次のように実行します。

```
% sccs admin -fb unstamp
```

最初のチームは、次のような `edit` コマンドを実行して、バージョン 1.3 を作成します。

```
% sccs edit unstamp
1.2
new delta 1.3
55 lines
```

2 番目のチームは、次のような `edit -b` コマンドを実行して、分岐を作成します。

```
% sccs edit -b unstamp
1.2
new delta 1.2.1.1
55 lines
```

現在、`unstamp` ファイルのツリー構造が、主幹部と番号 1.2.1、1.2.2、1.3.1 の分岐からなると仮定します。配布のために、分岐 1.2.2 から最新バージョンを取り出すには、次のコマンドを実行します。

```
% sccs get -r1.2.2 unstamp
1.2.2.1
55 lines
```

SCCS ツリー構造がより複雑になると、取り出すデルタをあらかじめ知る必要があるため、編集用の最新のデルタを確実に取り出すことが困難になります。`sccs get` コマンドと `sccs edit` コマンドに `-t` オプションを付けて使用すると、SID によらず、確実に最新のデルタを指定することができます。



`sccs edit -i` コマンドでマージしたいバージョンを指定すると、分岐した SCCS ファイルを主幹部にマージして戻すことができます。たとえば、次のコマンドは、1.2.1.1 から 1.2.1.3 の範囲のすべてのデルタを含んだ、`unstamp` コマンドのバージョン 1.5 を作成します。指定された変更内容がすべて含まれるように、各デルタは、相互の関連付けが行われます。

```
% sccs edit -i1.2.1.1-1.2.1.3 unstamp
Included:
1.2.1.1
1.2.1.2
1.2.1.3
1.4
new delta 1.5
55 lines
```

## 6.6.6 SCCS ファイルの差異の表示

SCCS ファイルのバージョン間の差異を確認することができます。ファイルの形式によって、`sccs diffs` コマンドまたは `sccsdiff` コマンドのいずれかを使用します。

`sccs diffs` コマンドは、`g` ファイルを、`s` ファイルの指定されたバージョンと比較します。たとえば、`attr` ファイルの最新バージョン (1.8 であり、編集後 1.9) と直前のバージョン (1.7) の差異を検出する場合には、次のコマンドを実行します。

```
% sccs diffs -r1.7 attr

----- attr -----
31d30
<#      and is version-linked to the docbld command
```

`attr` ファイルのバージョン 1.3 と 1.4 の差異を確認する場合には、次のコマンドを実行します。

```
% sccs sccsdiff -r1.3 -r1.4 SCCS/s.attr
< uts=-04
---
> uts=-05
```

この例のように、`s` ファイルは、パス名で指定することができます。このように設計されているため、SCCS ライブラリのあるディレクトリに移動しなくても、任意のディレクトリからこのコマンドを実行することができます。

## 6.6.7 SCCS ファイルのリビジョン・ヒストリの報告

`sccs prs` コマンドを実行して、ファイルのリビジョン・ヒストリを確認することができます。次に例を示します。

```
% sccs prs unstamp
SCCS/s.unstamp:

D 1.2 99/03/20 11:23:36 gunther 2 1      00000/00006/00055  1
MRs:                                     2
COMMENTS:                               3
Fixed copyright notice

D 1.1 99/03/19 09:39:11 gunther 1 0      00061/00000/00000
MRs:
COMMENTS:
date and time created 99/03/19 09:39:11 by gunther
```

上の例のコールアウトで示された、`D`、`MRs`、および `COMMENTS` の各キーワードは、SCCS キーワードの一部です。`sccs help` コマンドを実行すると、キーワードおよびその意味のリストが表示されます。

- ❶ `D` キーワードは、デルタ情報を示します。`gunther` (プログラマのユーザ名) の後の 2 つの数字は、新旧のリビジョン・レベルを示します。スラッシュで区切られた数字はそれぞれ、追加行、削除行、変更なしの行の数を示します。
- ❷ キーワード `MRs` によって、主リビジョンが表示されます。ここで、主リビジョンとは、ファイルの `SID` の最初の要素のことです。
- ❸ `COMMENTS` キーワードは、自由な書式での履歴情報を記録する場所を提供します。

`sccs get -m` コマンドを実行してファイルを取り出すと、プレフィックスとして各行に `SID` 番号が付加されたファイルが取り出されます。この方法で取り出されたファイルは、取り出したバージョンで、どのデルタがどの行を生成しているかを示します。デルタは、新しいデルタによって重ね書きされている可能性があることに注意してください。特定の変更を見るためには、`-r` オプションを使用する必要があります。

### 6.6.8 管理機能の実行

sccs admin コマンドには、いくつかの管理用機能があります。各機能は、表 6-6 に示されているように、admin コマンドに付けるオプションで指定します。

#### 注意

システム管理者は、管理者だけが使用できるように admin コマンドの許可を設定することができます。

表 6-6: SCCS admin コマンドのオプション

オプション	説明
<code>-auser s-file</code>	指定されたユーザを、指定の <i>s-file</i> を変更できるユーザのリストに追加する。ユーザ名はグループ ID でもよい。その場合、そのグループのユーザすべてが追加される。
<code>-dflag s-file</code>	<i>s-file</i> の、指定されたフラグをオフに (削除) する。
<code>-euser s-file</code>	指定されたユーザを、指定の <i>s-file</i> を変更できるユーザのリストから削除する。ユーザ名はグループ ID でもよい。その場合、そのグループのユーザはすべて削除される。
<code>-fflag s-file</code>	<i>s-file</i> の、指定されたフラグをオンにする。
<code>-h s-file</code>	指定の <i>s-file</i> の構造をチェックし、新しく計算されたチェックサムと、 <i>s-file</i> に保存されているチェックサムを比較する。このオプションは、事故による損傷、および、SCCS コマンド以外で直接 SCCS ファイルを変更した際に起こる損傷の発見に役立つ。
<code>-iinput-file s-file</code>	<i>input-file</i> を初期内容として、SCCS/ <i>s.s-file</i> を作成する。sccs create と異なり、admin <code>-i</code> は、 <i>g-file</i> の名前の変更や、 <i>s-file</i> のコピーを取り出さない。 <i>g-file</i> は、未変更のまま現在のディレクトリに格納される。
<code>-mMR-list s-file</code>	変更要求 (MR) 番号のリストを指定すると、初期デルタの作成理由として SCCS ファイルに挿入される。
<code>-ns-file</code>	空の <i>s-file</i> を作成する。
<code>-rSID s-file</code>	<i>s-file</i> を作成する際の初期の SID を指定する。

表 6-6: SCCS admin コマンドのオプション (続き)

オプション	説明
<code>-tfile s-file</code>	<i>file</i> の内容を <i>s-file</i> に追加する。その際、 <i>s-file</i> には、追加されたテキストとしてのフラグを立てる。 <i>file</i> が省略された場合は、そのようにして追加されたテキストは削除される。 <i>s-file</i> とともに確実に配布したいドキュメントを指定する場合に有効である。
<code>-y"comment" s-file</code>	delta コマンドの動作と同じ方法で、初期デルタにコメント・テキストを挿入する。 <code>-y</code> オプションを使用しない場合、省略時のコメントとして、ファイルの作成日付と時刻、およびユーザ名を含んだ 1 行が挿入される。
<code>-z s-file</code>	ファイルが破損した場合に、 <i>s-file</i> のチェックサムを再計算する。

注意

`val` コマンドと `admin -z` コマンドを使用して、破損した `s` ファイルを修復することは危険です。システム管理者または指定の SCCS ライブラリ管理者に依頼してください。

`admin -f` オプションと `admin -d` オプションのフラグは、表 6-7 で示されているとおりです。

表 6-7: admin コマンドのフラグ

フラグ	説明
<code>b</code>	<code>edit</code> コマンドに <code>-b</code> フラグを使用すると、分岐が作成される。
<code>cSID</code>	<i>SID</i> を <code>get -e</code> コマンドが使用できる最も高位のデルタとして指定する。
<code>dSID</code>	<code>get</code> コマンドまたは <code>edit</code> コマンドで省略時の <i>SID</i> を使用することを指定する。
<code>fSID</code>	<i>SID</i> を <code>get -e</code> コマンドで使用できる最も低位のデルタとして指定する。
<code>i</code>	「no Id keywords」エラー・メッセージを、警告ではなく致命的エラーにする。

表 6-7: admin コマンドのフラグ (続き)

フラグ	説明
j	2 人以上による <i>s-file</i> の同時編集を可能にする。
lSID [,SID..] ]	指定の <i>SID</i> が編集用として取り出されないようにロックする。 <i>-fla</i> フラグですべてのデルタをロックできる。 <i>-d</i> フラグで指定のデルタのロックを解除できる。
mname	get コマンドでキーワードが展開された場合、出現する %M% キーワードをすべて <i>name</i> で置換する。省略時の名前は、 <i>s</i> プレフィックスのない <i>s-file</i> 名。
n	delta コマンドは、デルタが新しいリリースで作成された場合に、すべてのスキップされたリリースを空のデルタとして作成する。たとえば、デルタ 5.1 をデルタ 2.7 の後に作成すると、リリース 3 と 4 は空になる。その結果作られた空のデルタは、分岐デルタを作り上げる際の起点とすることができる。このフラグを使用しない限り、省略されたリリースは <i>s-file</i> に現れない。
q"text"	get コマンドでキーワードが展開された場合、出現するすべての %Q% キーワードを <i>text</i> に置換する。
ttype	get コマンドでキーワードが展開された場合、出現するすべての %Y% キーワードを <i>type</i> に置換する。
v[program]	デルタは、デルタを作成する理由としての変更要求 (MR) 番号の入力を要求する。 <i>program</i> 名は、MR 番号の妥当性チェック・プログラムの名前を指定する。詳細は、delta(1) を参照。

たとえば、次のコマンドは、既存のテキスト・ファイルの内容を使用して、SID 2.1 で始まる *s-file* を作成します。さらに、コメントを付けて識別できるようにしています。*s-file* の *i* フラグが設定されています。このコマンドでは、結果の *s-file* を、ユーザの作業ディレクトリの下の SCCS ライブラリに格納します。

```
% sccs admin -iunstamp -fi -r2 -y "Initial release" unstamp
```

この例では、元のファイルは破壊されません。

## 6.6.9 SCCS オプションの使用

sccs コマンドでは、表 6-8 に掲載したオプションがサポートされています。これらのオプションには、表に例示されているような、SCCS 機能コマンドのキーワードが含まれている必要があります。オプションと引数の間にはスペースを入れてはなりません。

表 6-8: SCCS コマンド・オプション

オプション	説明
-ddirname	SCCS ライブラリの親として使用するディレクトリを指定する。 現在の作業ディレクトリが親でなくても、SCCS ライブラリにアクセスできる。次に例を示す。  % pwd /usr/users/gunther % sccs -d/usr/src/dcb_tools get attr 1.8 126 lines
-ppath	path を、指定したファイルのパス名の最終要素に付加する。 省略時、SCCS は sccs を付加する。-d の例では、指定されたパスが /usr/src/dcb_tools/SCCS/s.attr になる。SCCS ライブラリが sccs という名前でない場合は、パスの構成要素を変更するために、-d オプションを使用する。
-r	sccs UID に変更せずに、ユーザの実際の UID で実行する。セキュリティ上、SCCS は、通常、SCCS ライブラリのファイルの所有権を、sccs UID に属するように設定する。自分のライブラリを管理するためだけに SCCS を使用する場合に、このオプションは有効である。つまり、-r オプションを使用すると、特別の許可がなくても sccs ディレクトリを作成でき、SCCS は、ユーザの UID を使用してそのディレクトリ内のファイルを変更する。

6.6.10 独立した SCCS コマンドの一覧

表 6-9 に、独立した SCCS コマンドを簡単に説明します。これらのコマンドのうちのいくつかは、sccs コマンドではサポートされていません。詳細については、該当するコマンドのリファレンス・ページを参照してください。

表 6-9: 独立した SCCS コマンド

コマンド名	sccs コマンドによるサポート	説明
admin	あり	s-file の作成、または既存の s-file の特性の変更。
cdc	なし	デルタに関連するコメントを変更する。
comb	なし	s-file の 2 つ以上の連続したデルタを 1 つのデルタに統合する。デルタを結合することによって、必要な記憶領域を減少できる。

表 6-9: 独立した SCCS コマンド (続き)

コマンド名	SCCS コマンドによるサポート	説明
delta	あり	編集済みの <i>g-file</i> をライブラリにチェックインさせ、変更内容と履歴を記録する。p ファイルを削除する。
get	あり	<i>s-file</i> の指定されたバージョンを取り出す。編集またはコンパイルを行うためにファイルのコピーを取得する場合には、このコマンドを実行する。編集する場合は、 <code>get -e</code> コマンドを実行する。その場合は、編集用の <i>s-file</i> をチェックアウトし、 <i>g-file</i> を再作成し、それを現在のディレクトリに置いて、p ファイルを作成する。
prs	あり	指定の <i>s-file</i> のリビジョン・履歴を表示する。
rmDEL	あり	指定の <i>s-file</i> の指定された分岐から最新のデルタを削除する。
sccsdiff	あり	<i>s-file</i> の 2 つのバージョンを比較する。 <i>s-file</i> 名を明示して指定する必要がある。
sccshelp	なし	診断メッセージまたは SCCS コマンド名の説明を出力する。
unget	なし	p ファイルを削除して、展開された ID キーワードを持つコピーへ <i>g-file</i> を置き換え、前回 <code>get -e</code> コマンドを実行したときの影響を削除する。sccs unedit コマンドと同じ。
val	なし	<i>s-file</i> のチェックサムを計算して、その結果が、ファイルに保存されているチェックサムと一致するかどうかを確認する。sccs admin -z コマンドとこのコマンドを一緒に使用すると、破損したファイルを修復できる。
what	あり	ファイル内の SCCS ID パターンを探索して、それに続くテキストを表示する。このコマンドを実行すると、(SCCS 制御の下に置かれている) プログラムを構成するソース・バージョンの識別情報を検出できる。

## 注意

`val` コマンドと `admin -z` コマンドを使用して、破損した `s` ファイルを修復することは危険です。システム管理者または指定の SCCS ライブラリ管理者に依頼してください。

## 6.7 RCS コマンドと SCCS コマンドの機能比較

表 6-10 は、RCS および SCCS の動作の簡単な比較と、類似した機能を実行するコマンドです。各コマンドの使用方法についての詳細は、リファレンス・ページを参照してください。

表 6-10: 機能比較: RCS と SCCS コマンド

タスク	RCS コマンド	SCCS コマンド
元になるファイルから新しいファイルを作成する。	<code>ci file</code>	<code>sccs create file sccs admin -isfile gfile admin -ipath/sfile gfile</code>
展開されたキーワードを持つファイルのコピーを取り出す。	<code>co -u file</code>	<code>sccs get file get file</code>
展開されていないキーワードを持つファイルのコピーを取り出す。		<code>sccs get -k file get -k file</code>
ファイルをチェックアウトする。	<code>co -l file</code>	<code>sccs edit file get -e file</code>
編集済みのファイルをチェックインする。	<code>ci file</code>	<code>sccs delta file delta file</code>
ファイルのレビジョン・ヒストリを表示する。	<code>rlog file</code>	<code>sccs prs file prs file</code>



表 6-10: 機能比較: RCS と SCCS コマンド (続き)

タスク	RCS コマンド	SCCS コマンド
ファイル・リビジョン間の差異を確認する。	<code>rcsdiff -rrev file</code>	<code>sccs diffs -rrev file sccsdiff -rrev -rrev file</code>
ファイル・リビジョンをマージする。	<code>rcsmerge -rrevs file</code>	<code>sccs edit -irevs file</code>
確認情報を検索する。	<code>ident</code>	<code>sccs what what</code>
管理用のタスクを実行する。	<code>rcs</code>	<code>admin</code>
現在のディレクトリを整理する (変更されていないファイルを削除する)。	<code>rcsclean</code>	<code>sccs clean</code>



---

## make ユーティリティによるプログラムのビルド

make ユーティリティは、プログラムの最新バージョンを作成します。make ユーティリティは、大規模なプログラミング・プロジェクトで、多数のソース・ファイルを組み合わせて1つのプログラムを作成したり、1つの製品またはアプリケーションを構成するプログラム群を作成したりする場合などに最適です。

この章では、次の情報について説明します。

- make ユーティリティの動作 (7.1 節)
- 記述ファイル (7.2 節)

make コマンドには、ビルド処理の実行を制御したり変更したりするためのオプションが用意されています。make ユーティリティは、同じソース・ファイルの2つ以上のバージョンを保守するという問題は扱いません。

プログラムの保守に make ユーティリティを使用すると、次のことが実行できます。

- 大規模なプログラムを作成するための命令群を1つのファイルにまとめる。
- マクロを定義して make 記述ファイル内で使用する。
- シェル・コマンドを使用する。
- ライブラリを作成したり更新する。
- 別のプログラムからファイルをインクルードする。

オペレーティング・システムは、いくつかのバージョンの make コマンドを提供します。この章では、省略時のバージョンの make(1) コマンドについて説明します。この他のバージョンとしては、make(1) ではサポートしない機能を提供する make(1u) および make(1p) があります。make(1p) は、POSIX 準拠の make コマンドです。

`make(1)` および `make(1u)` は、ベース・オペレーティング・システムのサブセットに含まれています。`make(1p)` は、ソフトウェア開発環境 (Software Development Environment) サブセットに含まれています。

詳細については、`make(1)`、`make(1u)` および `make(1p)` を参照してください。この章では、`make(1)` の機能について説明します。

## 7.1 `make` ユーティリティの動作

`make` ユーティリティは、ターゲットまたはターゲット・ファイルと呼ばれるプログラムの作成日と、従属ファイルまたは従属と呼ばれるターゲットを構成する各ファイルの日付とを比較することによって作動します。ターゲットの従属ファイルの中にターゲットよりも新しいものがあれば、`make` は既存のターゲットを古いものと見なします。この場合、`make` はコンパイルやリンクなど、必要な手順を実行してターゲットを再作成します。従属ファイルはまた、同時にターゲットでもあります。たとえば、実行可能プログラムは複数のオブジェクト・モジュールから作成されます。オブジェクト・モジュールは、複数のソース・ファイルから作成されます。ターゲットよりも新しい従属は、若い (younger) ファイルと呼ばれます。

`make` ユーティリティは、次の情報を使用します。

- ユーザの作成する記述ファイル
- ファイル名
- ファイル・システム内のファイルのタイム・スタンプ
- ファイルのビルド方法を `make` に指示するための規則

`make` ユーティリティはファイルのタイム・スタンプを使用します。分散システム上で `make` を正しく作動させるには、ネットワーク上のすべてのシステムの日付および時刻が一致していなければなりません。

`make` ユーティリティは、次の段階的な手順でターゲット・ファイルをビルドします。

1. 記述ファイル内のターゲット・ファイルの名前を見つける。
2. 従属行と呼ばれるターゲットの従属を示す行を見つける。
3. ターゲットの従属ファイルのすべてが存在することおよび最新であることを確認する。

4. ターゲットがその従属に対して最新であるかどうかを判定する。
5. ターゲットまたは従属のいずれかが古いものであった場合には、次のいずれかの方法を使用してターゲットをビルドする。
  - 記述ファイルからコマンドを実行する。
  - 内部規則が適用される場合は、その規則を使用してファイルをビルドする。
  - 記述ファイルから省略時の規則を使用する。

`make` が実行された時点で、従属行に記されたすべてのファイルが最新のものであれば、`make` は、ターゲットが最新であることを示して実行を停止します。ターゲットよりも新しい従属がある場合、`make` は、古いターゲットだけを再ビルドします。存在していないファイルは、古いものと見なされます。

所定のターゲットに従属がない場合には、ターゲットは常に古いものと見なされ、`make` を実行するたびにターゲットは作り直されます。`make` の処理は、再ビルドする必要のあるターゲットを決定する際にはトップダウン方式で行われ、実際の再ビルド時にはボトムアップ方式で作動します。

ターゲットをビルドするために、`make` ユーティリティがコマンドを実行する時点で、`make` は各マクロをその値に置き換えて各コマンド行を標準出力にエコーし、コマンドを実行します。マクロについての詳細は、7.2.9 項を参照してください。`make` ユーティリティは、`rm` や `cc` などの直接実行できるコマンドを新しいシェルを呼び出さずに実行します。`make` ユーティリティは、新しいシェルのパイプやリダイレクションなどのシェル関数を指定した各コマンド行を実行します。

`make` ユーティリティは、記述ファイルのあるディレクトリで起動します。`make` コマンドの構文は、次のとおりです。

**make** *[[ -f ] makefile] [ options] [ targets] [ macro definitions]*

`make` ユーティリティは、コマンド行のエントリを確認して実行する動作を判断します。まず `make` ユーティリティは、コマンド行 (等号を含むエントリ) のマクロ定義がある場合にはこれに値を割り当て、次に記述ファイルのマクロ定義に値を割り当てます。コマンド行に定義されたマクロ名が記述ファイルにも定義されている場合、`make` はコマンド行の定義を使用し、記述ファイルの定義を無視します。

次に、make はオプションを確認します。make がサポートするオプションの一覧については、make(1) を参照してください。

make ユーティリティは、残りのコマンド行エントリをターゲット名であると解釈して、左から右の順にそれらのターゲットを処理します。コマンド行にターゲットがまったく存在しない場合には、make は記述ファイルで最初に指定されたターゲットを処理して実行を停止します。

## 7.2 記述ファイル

記述ファイルには、ターゲットに含める従属とそれらの従属とプロシージャ内の他のファイルとの関係を定義して、ターゲットのビルド方法を make に指示します。記述ファイルには、次の情報が含まれます。

- 記述ファイル内のマクロの定義
- 1 つまたは複数のターゲット名
- ターゲット・ファイルを構成する従属ファイル名
- 従属からターゲット・ファイルをビルドするためのコマンド
- .DEFAULT, .IGNORE, .PRECIOUS, .SILENT, または .SUFFIXES のいずれかの擬似ターゲット

これらの識別子は、実際のターゲットではないために擬似ターゲットと呼ばれます。これらは、make が特別な方法で解釈する組み込み名です。たとえば .SILENT は、make がコマンド行を実行する際にエコーしないように指示します。擬似ターゲットを実際のターゲット名として使用してはなりません。擬似ターゲットの詳細は、make(1) を参照してください。

make ユーティリティは、従属ファイルの日付を確認することによって、ターゲットを最新に保つためにビルドするファイルを決定します。前回ターゲットをビルドした後に従属ファイルが変更されている場合、make は、そのターゲットを含めて、従属ファイルの変更によって影響を受けるすべてのファイルをビルドします。ほとんどの場合記述ファイルは簡単に記述でき、何度も変更されることはありません。

通常 make ユーティリティは、makefile または Makefile と名付けられた記述ファイルを探します。記述ファイル名として makefile または Makefile を使用し、その記述ファイルが存在するディレクトリで作業している場合には、オプションおよび引数を省略して make コマンドを入力する

と、`make` が前回ターゲットをビルドしてから変更されたファイルの数にはかわらず、最初のターゲットおよびその従属ファイルを更新します。次の例のように、`make` ユーティリティに `-f` オプションを使用すると、省略時のファイル名の代わりに任意の記述ファイル名を指定することができます。

```
% make -f my_makefile
```

このオプションを使用することにより、複数の記述ファイルを同じディレクトリに保存することができます。

この節では、記述ファイルの次の事項について説明します。

- 記述ファイル・エントリの形式 (7.2.1 項)
- 記述ファイルでのコマンドの使用 (7.2.2 項)
- `make` ユーティリティのコマンド・エコーの防止 (7.2.3 項)
- `make` ユーティリティのエラーによる停止の防止 (7.2.4 項)
- 省略時の条件の定義 (7.2.5 項)
- `make` によるファイル削除の防止 (7.2.6 項)
- 簡単な記述ファイル (7.2.7 項)
- 記述ファイルの簡素化 (7.2.8 項)
- マクロの定義 (7.2.9 項)
- 記述ファイルでのマクロの使用 (7.2.10 項)
- 記述ファイルからの `make` ユーティリティの呼び出し (7.2.11 項)
- 内部マクロ (7.2.12 項)
- `make` ユーティリティによる環境変数の使用 (7.2.13 項)
- 内部規則 (7.2.14 項)
- 他のファイルのインクルード (7.2.15 項)
- 記述ファイルのテスト (7.2.16 項)
- 記述ファイル (7.2.17 項)

## 7.2.1 記述ファイル・エントリの形式

記述ファイル・エントリの一般的な形式は、次のとおりです。

```
[target1 [target2...]] [:] [[:]] [[ dependent...]] [[:] commands] [[#  
] comment...]
```

大カッコ内の項目は省略可能です。ターゲット (*target*) および従属 (*dependent*) は、文字、数字、ピリオド、あるいはスラッシュで構成される文字列によるファイル名です。make コマンドは、アスタリスク (\*) および疑問符 (?) などのワイルドカード文字を認識します。ターゲット名を含む記述ファイルの各行は、従属行と呼ばれます。従属行では、ターゲット名の後に、ターゲットをビルドするための処理手順を指定する 1 つまたは複数のコマンド行 (*commands*) が続きます。

make では、マクロの指定にドル記号 (\$) を使用するため、ドル記号 (\$) をターゲットおよび従属のファイル名に使用してはなりません。同様に、定義した make マクロを参照する場合以外には、記述ファイル内のコマンドにドル記号を使用してはなりません。マクロについては、7.2.9 項、7.2.10 項、および 7.2.12 項を参照してください。

記述ファイル内にコメント (*comment*) を書く場合には、番号記号 (#) を使用してコメント・テキストを開始してください。make ユーティリティは、番号記号以降にある同じ行上の番号記号とすべての文字を無視します。空白行も無視されます。

バックスラッシュ (\) を行の終わりに置くことによって、入力デバイスの行幅よりも長い行を入力することができますが、この方法でコメント行を拡張してはなりません。各行に 1 つの番号記号を付けて新しいコメント行を開始してください。

## 7.2.2 記述ファイルでのコマンドの使用

コマンドは、番号記号や改行文字を除いた、任意の文字からなる文字列です。コマンドは、従属行のセミコロン (;) の後または従属行の直後の行に現れます。従属行の後の各コマンド行は、単一のタブ文字で開始する必要があります。

記述ファイル内でターゲットにコマンド・シーケンスを定義する場合には、各ターゲットにコマンド・シーケンスを 1 つ指定するか、または特定の従属の集合に別のコマンド・シーケンスを指定します。

各ターゲットを使用するたびにコマンド・シーケンスを 1 つ使用するには、従属行のターゲット名に続けてコロン (:) を 1 つ使用します。たとえば、次



の例では、`test` というターゲット名とターゲットを作成するための従属ファイルの集合およびコマンドの集合が定義されています。

```
test: dependency list1...
      command list...
:
test: dependency list2...
```

ここに示されているように、1つのターゲット名が、記述ファイル内の複数の箇所に、異なる従属リストを伴って現れますが、このターゲット名に関連付けられるコマンド・リストは1つだけです。`make` ユーティリティは、所定のターゲットのすべての従属行を見つけて、それらの従属リストのすべてを1つのリストに連結します。従属のいずれかが変更された場合に、`make` は当該ターゲットをビルドするために1つのコマンド・リストにあるコマンド群を実行します。

特定のターゲット・ファイルをビルドするために、コマンドの集合を2つ以上指定する場合には、2つ以上の従属定義を入力します。各従属行には、ターゲット名の後にコロンを2つ(`::`)加え、その後に1つの従属リスト、および、従属リストのファイルのいずれかが変更された場合に `make` が使用するコマンド・リストを続けなければなりません。たとえば、次の例ではターゲット・ファイル `test` をビルドするために2つの異なる処理を定義しています。

```
test:: dependency list1...
      command list1...
:
test:: dependency list2...
      command list2...
```

`make` は、従属リスト `list1` のファイルのいずれかが変更された場合にはコマンド `list1` を、従属リスト `list2` のファイルのいずれかが変更された場合には `list2` を実行します。衝突を避けるため、1つの従属ファイルを従属リストの `list1` および `list2` の両方に指定することはできません。

#### 注意

`make` は、先行する各コマンド行および後続のコマンド行から独立したものとして各コマンドを実行するので、たとえば、一定のコマンド (`cd` など) を使用する場合には注意が必要です。次の例では、`cd` コマンドは、後続の `cc` コマンドには影響を与えていません。

```
test: test.o
      cd /u/tom/newtest
      cc main.o subs.o -o test
```

cc コマンドに cd コマンドを作用させるには、次のように、両方のコマンドを同じ行に置いてセミコロンで区切ります。

```
test: test.o
      cd /u/tom/newtest; cc main.o subs.o -o test
```

次のように、バックスラッシュを後続する行の先頭に置くことによって、複数の行からなるシェル・スクリプトをシミュレートすることができます。

```
test: test.o
      cd /u/tom/newtest; \
      cc main.o subs.o -o test
```

これは、直前の例とまったく同じ作用をします。バックスラッシュによって接続する各行 (この例では cd の行) には、バックスラッシュの前にセミコロンを置かなければなりません。

---

### 7.2.3 make ユーティリティのコマンド・エコーの防止

make が実行中のコマンドを標準出力にエコーしないようにするには、次の手順のうちの 1 つを実行します。

- make コマンドで -s フラグを指定する。
- 記述ファイルに擬似ターゲット名 `.SILENT:` という単独の 1 行を置く。  
擬似ターゲットの詳細については、7.2 節を参照してください。
- 記述ファイル内で、make がエコーする必要のない各コマンド行の文字開始位置 (タブの直後) にアットマーク (@) を入力する。

### 7.2.4 make ユーティリティのエラーによる停止の防止

いずれかのコマンドがゼロ以外の状態コードを返した場合、make ユーティリティは通常、実行を停止してエラーを表示します。

エラーで make が終了しないようにするには、次の手順のいずれかを実行します。

- make コマンドで -i フラグを指定する。
- 記述ファイルに擬似ターゲット名 `.IGNORE:` という単独の 1 行を置く。

擬似ターゲットの詳細については、7.2 節を参照してください。

- 記述ファイル内で、make をエラーで停止させたくない、各コマンド行の文字開始位置 (タブの直後) に、ハイフン (-) を入力する。

### 7.2.5 省略時の条件の定義

make がターゲットをビルドするにあたって、明示されたコマンド行や内部規則を見つけることができない場合には、記述ファイル中の省略時の条件を調べます。この場合に make が実行するコマンドを定義するには、記述ファイル中に擬似ターゲット名 `.DEFAULT:` を使用し、その他の任意のターゲットの場合と同様に、省略時のコマンド・シーケンスを入力します。

擬似ターゲット `.DEFAULT:` は、エラー回復の手順や make ユーティリティの内部規則に定義されていないプログラムで、すべてのファイルをビルドするための一般的な手順として使用することができます。

### 7.2.6 make によるファイル削除の防止

不正のおそれのあるターゲット・ファイルを使用してビルドしないように、make は通常、途中でエラーが返された場合にはターゲット・ファイルを削除します。エラーが検出された場合にも、make がファイルを削除しないようにするには、記述ファイル内に擬似ターゲット `.PRECIOUS:` を使用してください。擬似ターゲット名の後に、セーブする必要のあるターゲット名を挙げてください。コマンド行に `-u` オプションを指定した場合、make はチェックアウトされた RCS ファイルを削除しません。make が RCS と対話する方法についての詳細は、`make(1)` を参照してください。

### 7.2.7 簡単な記述ファイル

例 7-1 では、`x.c`、`y.c`、および `z.c` の 3 つの C 言語ファイルをロードしてコンパイルすることによって、`prog` と名付けられたプログラムがビルドされます。ファイル `x.c` および `y.c` は、`defs` という名前のファイルの中でいくつかの宣言を共有しますが、ファイル `z.c` はこれを共有しません。

例 7-1: 記述ファイルの例

---

```
# Make prog from 3 object files
prog: x.o y.o z.o
# Use the cc program to make prog
    cc x.o y.o z.o -o prog
```

### 例 7-1: 記述ファイルの例 (続き)

---

```
# Make x.o from 2 other files
x.o:  x.c defs
# Use the cc program to make x.o
    cc -c x.c

# Make y.o from 2 other files
y.o:  y.c defs
# Use the cc program to make y.o
    cc -c y.c

# Make z.o from z.c
z.o:  z.c
# Use the cc program to make z.o
    cc -c z.c
```

---

このファイルを `makefile` と名付けた場合、4 つのソース・ファイル `x.c`、`y.c`、`z.c`、および `defs` のいずれかのファイルを変更した後では、オプションも引数も使用しないで `make` コマンドを入力するだけで、`prog` の最新コピーを作成することができます。

## 7.2.8 記述ファイルの簡素化

記述ファイルをより簡単にするには、`make` ユーティリティの内部規則を使用します。`make` は、ファイル・システムの命名規則によって、必要な `.o` ファイルと対応する 3 つの `.c` ファイルを認識します。`make` には、ソース・ファイルからオブジェクト・ファイルを生成する処理 (`cc -c` コマンドの実行) も組み込まれています。これらの内部規則を活用することによって、前述の記述ファイルは次のように簡素化されます。

```
# Make prog from 3 object files
prog:  x.o y.o z.o
# Use the cc program to make prog
    cc x.o y.o z.o -o prog

# Use the file defs and the appropriate .c file
# when making x.o and y.o
x.o y.o:  defs
```

`make` が使用する内部規則については、7.2.14 項を参照してください。

## 7.2.9 マクロの定義

マクロとは、1 つまたは複数の名前の代わりに使用される別名のことです。長い文字列を短縮するために使用されます。マクロは、記述ファイル内またはコマンド行上に定義することができます。記述ファイル内にマクロを定義する方法は次のとおりです。

1. 新しい行をマクロ名で開始する。
2. 名前の後に等号 (=) を入力する。
3. 等号の右にマクロ名が表す文字列を入力する。文字列には空白を含めることができる。

マクロ定義では、等号の前後に空白を指定しても結果には影響を与えません。ただし、等号の前にコロンの(:) またはタブを指定することはできません。make ユーティリティは、マクロ定義の文字列に先行および後続する空白を無視します。次にマクロ定義の例を示します。

```
# Macro ABC has a value of "ls -la"
ABC = ls -la

# Macro LIBES has a null value
LIBES =

# Macro DIRECT includes the definition of macro ROOT
# The expanded value of DIRECT is "/usr/home/fred"
ROOT   = /usr/home
DIRECT = $(ROOT)/fred
```

この例にある DIRECT マクロは、自身の定義の一部として別の定義を使用します。マクロを使用する際の手順については、7.2.10 項を参照してください。

コマンド行上でマクロを定義するには、記述ファイル内でマクロを定義する場合と同じ構文に従いますが、同じ行の中にすべてのマクロを定義してください。コマンド行上で空白を含んだマクロを定義する場合には、("name = definition") のように引用符で定義を囲みます。引用符を付けない場合、シェルは、空白がパラメータ・セパレータであり、マクロの一部ではないと解釈します。

## 7.2.10 記述ファイルでのマクロの使用

記述ファイルでマクロを定義した後に、マクロ名の前にドル記号 (\$) を置くことによって記述ファイルのマクロの値を参照します。マクロ名が 1 文

字より長い場合には、次の例で示されるように、カッコまたは中カッコで囲んでください。

```
$(CFLAGS)
${xy}
$Z
$(Z)
```

最後の 2 つの例は、同じ結果になります。

#### 7.2.10.1 マクロの置換

マクロの定義値の一部またはすべてを、異なる値に置換することができます。次に、マクロ置換の 3 つの形式を示します。

- 形式 1

*MACRO* の定義された値の中のすべての *string1* を *string2* に置換します。

```
[$( MACRO: string1= string2 )]
```

次に例を示します。

```
# Define macro MAC1
MAC1 = xxx yyy zzz
:
# Evaluate MAC1
project:
    @ echo $(MAC1:yyy=abc)
```

この記述ファイルを使用して `make` を実行した場合、`make` は、`yyy` を `abc` に置換して、次の行を表示します。

```
xxx abc zzz
```

- 形式 2

定義された値の各ワードを置換します。*location* パラメータは、ワードのどの部分が *string* に置換されるかを指定します。

```
[$( MACRO/ location/ string )]
```

*location* パラメータは、次の値に限定されています。

- 山形記号 (^)

*string* の値が、定義された各ワードにプレフィックスとして追加されます。次にその例を示します。

```

# Define macro MAC1
MAC1 = abc def ghi
:

# Evaluate MAC1
project:
    @ echo $(MAC1/^/xyz)

```

この記述ファイルを使用して `make` を実行した場合、`make` は、定義された各ワードの先頭に `xyz` を追加して、次の行を表示します。

```
xyzabc xyzdef xyzghi
```

#### - アスタリスク (\*)

定義されたワードのすべてが、それぞれ *string* の値に置換されます。次にその例を示します。

```

# Define macro MAC1
MAC1 = abc def ghi
:

# Evaluate MAC1
project:
    @ echo $(MAC1/*/xyz)

```

この記述ファイルを使用して `make` を実行した場合、`make` は定義された各ワードを `xyz` と置換して、次の行を表示します。

```
xyz xyz xyz
```

アスタリスクを指定した場合、*string* の値にアンパサンド (&) を使用することができます。アンパサンドは置換される定義されたワードを示しており、そのワードは結果の中に挿入されます。次にその例を示します。

```

# Define macro MAC1
MAC1 = abc def ghi
:

# Evaluate MAC1
project:
    @ echo $(MAC1/*/x&z)

```

この記述ファイルを使用して `make` を実行した場合、`make` はアンパサンドに定義されたワードを挿入することによって各ワードを `x&z` に置換し、次の行を表示します。

```
xabcz xdefz xghiz
```

– ドル記号 (\$)

定義された各ワードに *string* の値が追加されます。次にその例を示します。

```
# Define macro MAC1
MAC1 = abc def ghi
```

⋮

```
# Evaluate MAC1
project:
    @ echo $(MAC1/$/xyz)
```

この記述ファイルを使用して `make` を実行した場合、`make` は定義された各ワードの後に `xyz` を追加して、次の行を表示します。

```
abcxyz defxyz ghixyz
```

• 形式 3

*MACRO* が定義されているかどうかにより、2 つの中からいずれかの置換を実行することができます。

`[$(MACRO? string1: string2)]`

*MACRO* が定義されていた場合、定義された値のすべてを *string1* と置換します。*MACRO* が定義されていない場合は、*string2* を使用します。次にその例を示します。

```
# Define macro MAC1
MAC1 = abc def ghi
```

⋮

```
# Evaluate MAC1 and MAC2.  MAC2 is not defined.
project:
    @ echo $(MAC1?uvw:xyz)
    @ echo $(MAC2?123:456)
```

この記述ファイルを使用して `make` を実行した場合、`make` は *MAC1* の値を `uvw` に置換し、定義されていない *MAC2* を `456` に置換して、次の行を表示します。

```
uvw
456
```

*MACRO* が定義されていない場合、形式 1 および 2 では空文字列が生成されます。



### 7.2.10.2 条件付きマクロ

マクロの値は、先在する条件に基づいて割り当てることができます。このタイプのマクロを条件付きマクロと呼びます。コマンド行で条件付きマクロを定義することはできません。条件付きマクロ定義は、すべて記述ファイルに定義する必要があります。条件付きマクロの構文は、次のとおりです。

```
[target:=MACRO = string]
```

指定されたターゲットが `make` コマンドの現在のターゲットである場合には、条件付きマクロには、文字列の値が割り当てられます。そうでない場合は、マクロの値は空になります。次の記述ファイルでは、`MAC1` に条件付き置換を使用しています。

```
# Define the conditional macro MAC1
target2:=MAC1 = xxx yy xxxyyy
:

#list targets and command lines
#
target1:;@echo $(MAC1)
target2:;@echo $(MAC1)
```

この記述ファイルを使用して `make` を実行した場合は、次のような結果になります。

```
% make target1
% make target2
xxx yy xxxyyy
```

### 7.2.11 記述ファイルからの `make` ユーティリティの呼び出し

`make` 記述ファイル内のコマンド行の 1 つに `$(MAKE)` マクロを指定することによって、`make` ユーティリティの呼び出しをネストすることができます。このマクロが存在する場合、`-n` オプションが設定された場合にも、`make` は、別の `make` のコピーを実行します。`-n` オプションについての詳細は、7.2.16 項を参照してください。

### 7.2.12 内部マクロ

`make` ユーティリティには、記述ファイル内で使用できる組み込みマクロ定義があります。これらのマクロは、記述ファイル内での変数の指定を補助するものです。`make` ユーティリティは、マクロを表 7-1 に示された値に置換します。

表 7-1: make の内部マクロ

マクロ	値
<code>\$@</code>	現在のターゲット・ファイル名
<code>\$\$@</code>	従属行のターゲット名
<code>\$?</code>	ターゲットよりも新しく変更された従属ファイル名
<code>\$&lt;</code>	新しいターゲット・ファイルがビルドされる原因となった古いファイルの名前
<code>\$*</code>	サフィックスなしの現在の従属ファイル名

上記の内部マクロは、実際に `make` が使用する際に、単一のファイル名に変更されます。サフィックス `D` を使用すれば上記マクロのいずれかの解釈を変更し、ファイル名のディレクトリ部分だけを指定することができます。たとえば、現在のターゲットが `/u/tom/bin/fred` である場合に、`$(@D)` マクロは、名前の `/u/tom/bin` という部分だけを返します。同様に、サフィックス `F` では、ファイル名の部分だけを返します。たとえば、`$(@F)` マクロに同じターゲットを与えた場合には、`fred` を返します。`$?` マクロを除くすべての内部マクロは、サフィックス `D` あるいは `F` をとることができます。

分散ファイル・システムで内部マクロを使用する場合には、その前に、`make` が処理するファイルを含むすべてのノードで、システム・クロックが同じ日付と時間を示していることを確認する必要があります。

`make` ユーティリティは、記述ファイルからコマンドを実行してターゲット・ファイルをビルドする場合にだけ、これらの記号を変更します。次の項では、これらのマクロの詳細について説明します。

7.2.12.1 ターゲット・ファイル名内部マクロ

`make` ユーティリティは、ターゲットをビルドするコマンド・シーケンスの中に出現するすべての `$@` マクロを、現在のターゲットのフルネームに置換します。この置換は、コマンドを実行する前に行われます。次にその例を示します。

```
/u/tom/bin/test: test.o
                cc test.o -o $@
```

この例では、`/u/tom/bin/test` という名前の実行可能ファイルが作成されます。

### 7.2.12.2 ラベル名内部マクロ

\$\$@ マクロが記述ファイルの従属行のコロンの右側で使用された場合，  
make はこの記号を従属行のコロンの左側にあるラベル名と置換します。  
この名前は，ターゲット名や別のマクロ名であってもかまいません。次に  
その例を示します。

```
cat:  $$@.c
```

make ユーティリティは，この行を次のように解釈します。

```
cat:  cat.c
```

このマクロを使用すると，各ファイルが単一のソース・ファイルを持つファイルのグループを作成できます。たとえば，システム・コマンドのディレクトリを保守するには，次のような記述ファイルを使用します。

```
# Define macro CMDs as a series of command names
CMDs = cat dd echo date cc cmp comm ar ld chown
```

```
# Each command depends on a .c file
$(CMDs):  $$@.c
```

```
# Create the new command set by compiling the out of
# date files ($?) to the current target file name ($@)
    cc -O $? -o $@
```

make ユーティリティは，実行時に \$\$(@F) マクロを \$@ のファイルの部分に変更します。たとえば，記述ファイルを別のディレクトリで使いながら  
usr/include ディレクトリを保守する場合に，この記号を使用することができます。その記述ファイルは，次に示されるようになります。

```
# Define directory name macro INCDIR
INCDIR = /usr/include
```

```
# Define a group of files in the directory
# with the macro name INCLUDES
INCLUDES = \
    $(INCDIR)/stdio.h \
    $(INCDIR)/pwd.h \
    $(INCDIR)/dir.h \
    $(INCDIR)/a.out.h
```

```
# Each file in the list depends on a file
# of the same name in the current directory
$(INCLUDES):  $$(@F)
```

```
# Copy the younger files from the current
# directory to /usr/include
    cp $? $@
```

```
# Set the target files to read only status
chmod 0444 $@
```

この記述ファイルは、現在のディレクトリで対応するファイルが変更された場合に、`/usr/include` ディレクトリでファイルを作成します。

#### 7.2.12.3 **younger** ファイル内部マクロ

`$?` マクロが記述ファイルのコマンド・シーケンスにある場合、`make` は `$?` をターゲット・ファイルの最後の変更後に更新された従属ファイルのリストに置換します。

#### 7.2.12.4 最初の旧ファイル内部マクロ

`$<` マクロが記述ファイルのコマンド・シーケンスにある場合、`make` は、`$<` をファイルの作成を開始させたファイル名に置換します。置換するファイル名は、当該ターゲット・ファイルに対して古くなった特定の従属ファイル、すなわち、`make` が当該ターゲット・ファイルを再度作成する原因となったファイルの名前です。これが、新ファイルの一覧を返す `$?` 記号との違いです。

`make` ユーティリティは、内部規則から、あるいは、`.DEFAULT:` リストからコマンドを実行する場合に限り、この記号を置換します。`$<` 記号は、明示的に指定されたコマンド行には影響を与えません。

#### 7.2.12.5 現在のファイル名プレフィックスのための内部マクロ

`$*` マクロが記述ファイルのコマンド・シーケンスにある場合、`make` は `$*` を現在 `make` が使用してターゲット・ファイルをビルドしている従属ファイル名のサフィックスを除く部分に置換します。たとえば、`make` がターゲット `test.c` を作成している場合、`$*` 記号はファイル名 `test` に相当します。

`make` ユーティリティは、内部規則から、あるいは、`.DEFAULT:` リストからコマンドを実行する場合に限り、この記号を置換します。`$*` 記号は、明示的に指定されたコマンド行には影響を与えません。

### 7.2.13 **make** による環境変数の使用

`make` は、実行されるたびに現在の環境変数を読み取って、定義済みのマクロに追加します。さらに、`MAKEFLAGS` という新しいマクロを作成します。`MAKEFLAGS` マクロは、コマンド行に入力されたすべてのオプションの集合です。コマンド行オプションおよび記述ファイルの割り当てによっても、

MAKEFLAGS マクロの値を変更することができます。make が別の処理を開始する場合には、別の処理に対して MAKEFLAGS をエクスポートします。MAKEFLAGS マクロの再帰的な make 処理の影響については、7.2.16 項を参照してください。

make ユーティリティは、次の順序でマクロ定義を割り当てます。矛盾が発生する場合は上位の設定が優先します。

1. MAKEFLAGS 環境変数を読み取り、変数に指定されたオプションを設定する。

MAKEFLAGS が存在しないか、あるいは値が空の場合、make は内部 MAKEFLAGS マクロを空文字列に設定します。そうでない場合、make は MAKEFLAGS の各文字が入力オプションであると見なします。make ユーティリティは、`-f`、`-p`、および `-r` を除くこれらのオプションを使用して、動作条件を決定します。

2. コマンド行にある入力フラグを読み取り、設定する。

コマンド行で明示的に指定されたすべてのオプションが、MAKEFLAGS 環境変数によって設定に追加されます。

3. コマンド行からマクロ定義を読み取る。

これらの定義は、記述ファイル内の同じ名前の定義を変更します。

4. 内部マクロ定義を読み取る。

5. MAKEFLAGS マクロを含む環境変数を読み取る。

make ユーティリティは、すべての環境変数をマクロ定義として扱い、コマンドを実行するために呼び出すシェルにそれらを渡します。

## 7.2.14 内部規則

make ユーティリティは、内部規則の集合を持ち、これを使用してターゲットをビルドする方法を決定します。`-r` オプションを付けて make を呼び出すことによって、これらの規則を変更することができます。この場合、記述ファイル内のターゲットのビルドに必要なすべての規則を提供する必要があります。内部規則には、擬似ターゲット `.SUFFIXES:` を使用して定義されるファイル名のサフィックスのリストが含まれます。また、内部規則には、あるサフィックスを持つファイルから別のサフィックスを持つファイルを make に作成させる規則も含まれます。make の内部規則でサポートされる変換対象項目の一覧を参照するには、次のコマンドを実行してください。

```
% make -p | more
```

省略時の設定によるリストを変更しない場合，make は次のサフィックスがあるものと解釈します。

サフィックス	ファイル・タイプ
.o	オブジェクト・ファイル
.c	C ソース・ファイル
.e	efl ソース・ファイル
.r	Ratfor ソース・ファイル
.f あるいは .F	FORTRAN ソース・ファイル
.s	アセンブラ・ソース・ファイル
.y	yacc C ソース文法
.yr	yacc Ratfor ソース文法
.ye	yacc efl ソース文法
.l	lex ソース文法
.out	実行可能ファイル
.p	Pascal ソース・ファイル
.sh	Bourne シェル・スクリプト
.csh	C シェル・スクリプト
.h	C ヘッダ・ファイル

1 つまたは複数のサフィックスをスペースで区切り，記述ファイルの .SUFFIXES: 行に加えると，上記のリストにサフィックスを追加することができます。たとえば，次の行の例では，サフィックス .f77 および .ksh が既存のリストに追加されます。

```
.SUFFIXES: .f77 .ksh
```

make のサフィックスの省略時のリストを消去するには，何も名前を指定しないで .SUFFIXES: 行を挿入します。1 行目に空のリストを，2 行目に新しいリストを入力すると，省略時のリストを完全に新しいものにすることができます。

```
.SUFFIXES:
.SUFFIXES: .o .c .p .sh .ksh .csh
```

make は、サフィックス・リストを左から右の順に調べるため、エントリの順序は重要です。上記の例では、make がオブジェクト・ファイルを、次に C ソース・ファイルを検索します。

make ユーティリティは、リスト中にあり、次の 2 つの必要条件を満たす最初のエントリを使用します。

- 入出力のサフィックス要求と照合するエントリ
- 当該エントリに割り当てられた規則を持つエントリ

make が認識するリストにサフィックスを追加する場合には、従属からターゲットをビルドする方法を説明する規則を提供する必要があります。規則は従属行とそれに対応する一連のコマンドと似ています。make ユーティリティは、その規則が定義する 2 つのファイルのサフィックスから規則の名前を作成します。たとえば、.r ファイルを .o ファイルに変える規則の名前は、.r.o です。例 7-2 は、標準の省略時の規則ファイルの一部です。

#### 7.2.14.1 単一サフィックスの規則

make ユーティリティには、コマンド・ファイルなどのサフィックスを持たないターゲットをビルドするための、単一サフィックスの規則もあります。make ユーティリティには、次のサフィックスを持つソース・ファイルを、サフィックスを持たないオブジェクト・ファイルに変更する規則があります。

##### 例 7-2: 省略時の規則ファイル

---

```
# Create a .o file from a .c
# file with the cc program
.c.o
    $(CC) $(CFLAGS) -c $<

# Create a .o file from either a
# .e , a .r , or a .f
# file with the efl compiler
$(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<

# Create a .o file from
# a .s file with the assembler
.s.o:

    $(AS) -o $@ $<

.y.o:
# Use yacc to create an intermediate file
```

例 7-2: 省略時の規則ファイル (続き)

```
$(YACC) $(YFLAGS) $<
# Use cc compiler

$(CC) $(CFLAGS) -c y.tab.c
# Erase the intermediate file

rm y.tab.c
# Move to target file

mv y.tab.o $@

.y.c:
# Use yacc to create an intermediate file

$(YACC) $(YFLAGS) $<
# Move to target file

mv y.tab.c $@
```

サフィックス	ソース・ファイルのタイプ
.c	C 言語ソース・ファイルから
.sh	シェル・ファイルから

たとえば，必要なすべてのファイルが現在のディレクトリにある場合に，`cat` のようなプログラムを保守するには，次のコマンドを入力します。

```
% make cat
```

7.2.14.2 `make` の組み込みマクロの変更

`make` ユーティリティは，内部規則としてマクロ定義を使用します。これらのマクロ定義を変更するには，コマンド行または記述ファイルに，それらのマクロの新しい定義を入力してください。`make` ユーティリティは，コマンドおよび言語プロセッサに次のマクロ名を使用します。



コマンドまたは関数	コマンド・マクロ	コマンド・オプションまたは他のマクロ
アーカイブ・プログラム (ar)	AR	ARFLAGS
アーカイブの目次作成		RANLIB
アセンブラ	AS	ASFLAGS
C コンパイラ	CC	CFLAGS
C ライブラリ		LOADLIBS
RCS チェックアウト	CO	COFLAGS
コピー・コマンド (cp)	CP	CPFLAGS
efl コンパイラ	EC	EFLAGS
リンカ・コマンド (ld)	LD	LDFLAGS
lex コマンド	LEX	LFLAGS
lint コマンド	LINT	LINTFLAGS
make コマンド	MAKE	
再帰的 make 呼び出しフラグ		MAKEFLAGS
mv コマンド	MV	MVFLAGS
pc コマンド	PC	PFLAGS
f77 コンパイラ	RC	FFLAGS
Ratfor コンパイラ・フラグ		RFLAGS
rm コマンド	RM	RMFLAGS
従属ファイルと関係する ファイルの配置		VPATH
yacc コマンド	YACC	YFLAGS
yacc -e コマンド	YACCE	YFLAGS
yacc -r コマンド	YACCR	YFLAGS

たとえば、次のコマンドは、`make` を実行して、既定の C 言語コンパイラを `newcc` プログラムに置換します。

```
% make CC=newcc
```

同様に、次のコマンドは、`make` に C 言語コンパイラによって作成された最終的なオブジェクト・コードを最適化するように指示します。

```
% make "CFLAGS=-O"
```

make が使用する内部規則を調べるには、Bourne シェルから次のコマンドを入力します。

```
$ make -fp -< /dev/null 2>/dev/null
```

出力は、標準出力に表示されます。

### 7.2.15 他のファイルのインクルード

記述ファイルの任意の行に最初のワードとして `include` を入力すると、現在の記述ファイルにファイルをインクルードすることができます。`include` の後に空白またはタブを続けて、次に `make` が実行中にインクルードするファイル名を入力します。次にその例を示します。

```
include      /u/tom/temp /u/tom/sample
```

### 7.2.16 記述ファイルのテスト

記述ファイルをテストするには、`-n` コマンド・オプションで `make` を実行してください。このオプションは、`make` にコマンド行を実行しないでエコーだけするように指示します。`make` が実行するプロセス全体がユーザに見えるように、アットマーク (@) で始まるコマンドもエコーされます。`-n` オプションで実行中の場合でも、`$(MAKE)` マクロは、その他のコマンドとは違い実際に実行されます。

記述ファイルに `$(MAKE)` マクロのインスタンスが含まれる場合、`make` は `-n` を含む、コマンド行に入力したオプションのリストを `MAKEFLAGS` マクロに設定した状態で、`make` の新しいコピーを呼び出します。`make` の新しいコピーは、`-n` オプションが設定されているため、新しいコピーを呼び出した `make` と同じ方法でコマンドの実行を無視します。`make` コマンドを 1 度実行するだけで、`make` を再帰的呼び出しする記述ファイルのセットをテストすることができます。

### 7.2.17 記述ファイル

例 7-3 は、`make` ユーティリティを保守するための記述ファイルの例です。`make` のソース・コードには、多くの C 言語ソース・ファイルおよび `yacc` 文法ファイルが含まれます。`yacc` についての詳細は、第 4 章を参照してください。

### 例 7-3: make ユーティリティの makefile

---

```
# Description file for the Make program

# Macro def: send to be printed
P = lpr

# Macro def: source file names used
FILES = Makefile version.c defs main.c

        doname.c misc.c files.c
        dosy.c gram.y lex.c gcos.c

# Macro def: object file names used
OBJECTS = version.o main.o doname.o \
        misc.o files.o dosys.o gram.o

# Macro def: lint program and flags
LINT = lint -p

# Macro def: C compiler flags
CFLAGS = -O

# make depends on the files specified
# in the OBJECTS macro definition
make:    $(OBJECTS)
# Build make with the cc program
        cc $(CFLAGS) $(OBJECTS) -o make
# Show the file sizes
        size make

# The object files depend on a file
# named defs
$(OBJECTS):  defs

# The file gram.o depends on lex.c
# uses internal rules to build gram.o
gram.o:  lex.c

# Clean up the intermediate files
clean:
        rm *.o gram.c

# Copy the newly created program
# to /usr/bin and deletes the program
# from the current directory
install:
        cp make /usr/bin/make ; rm make

# Empty file ''print'' depends on the
```

### 例 7-3: make ユーティリティの makefile (続き)

---

```
# files included in the macro FILES
print: $(FILES)
# Print the recently changed files
    lpr $?
# Change the date on the empty file,
# print, to show the date of the last
# printing
    touch print

# Check the date of the old
# file against the date
# of the newly created file
test:
    make -dp | grep -v TIME >1zap
    /usr/bin/make -dp | grep -v TIME >2zap
    diff 1zap 2zap
    rm 1zap 2zap

# The program, lint, depends on the
# files that are listed
lint: dosys.c doname.c files.c main.c misc.c \
    version.c gram.c
# Run lint on the files listed
# LINT is an internal macro
    $(LINT) dosys. doname.c files.c main.c \
    misc.c version.c gram.c
    rm gram.c

# Archive the files that build make
arch:
    ar uv /sys/source/s2/make.a $(FILES)
```

---

---

## 用語集

この用語集には、本書で使用されている用語の定義が掲載されています。

### G

#### g ファイル

ソース・コード・コントロール・システム (SCCS) において、s ファイルを作成するか、あるいは、s ファイルにデルタを適用するために、その内容が使用されるファイル。

### I

#### ID キーワード

ソース・コード・コントロール・システム (SCCS) において、単一文字をパーセント記号 (%) で囲んだもの。リビジョン・コントロール・システム (RCS) では、キーワード名をドル符号 (\$) で囲んだものである。キーワードは、展開された形式では、日付、バージョン番号、名前などのファイルに関する識別情報を提供する。

### P

#### p ファイル

ソース・コード・コントロール・システム (SCCS) におけるロック・ファイル。このファイルの存在は、同名の s ファイルが現在編集集中であることを示す。

### R

#### RCS (リビジョン・コントロール・システム)

プログラムおよびドキュメントのソース・ファイルを管理し、特定のファイルの任意のリビジョンを検索することができる一組のプログラム。ファイルに対する改訂は、すべてのバージョンの完全なコピーではなく、元のバージョンに対する一連の付加的な変更 (デルタ) として保存される。このシステムはロック機構を提供するため、どの時点でも一人のユーザだけが特定のファイルに対する変更を行うことができる。

*SCCS* (ソース・コード・コントロール・システム) も参照

#### **RCS** ファイル

リビジョン・コントロール・システム (RCS) において、RCS ライブラリに保管されていて、元のファイルのテキストおよびそれに適用されたデルタのリストを含むファイルのこと。

#### **RCS** ライブラリ

リビジョン・コントロール・システム (RCS) において、RCS ファイルが保存されるディレクトリ。

## **S**

#### **SCCS** (ソース・コード・コントロール・システム)

プログラムおよびドキュメントのソース・ファイルを管理し、特定のファイルの任意のリビジョンを検索することができる一組のプログラム。ファイルに対する改訂は、すべてのバージョンの完全なコピーではなく、元のバージョンに対する一連の付加的な変更 (デルタ) として保存される。このシステムはロック機構を提供するため、どの時点でも一人のユーザだけが特定のファイルに対する変更を行うことができる。

*RCS* (リビジョン・コントロール・システム) も参照

#### **SCCS** ライブラリ

ソース・コード・コントロール・システム (SCCS) において、SCCS の *s* ファイルおよび *p* ファイルが保存されているディレクトリ。

#### **SID**

SCCS において、特定のデルタに適用される数値による識別子。

*SCCS* (ソース・コード・コントロール・システム) も参照

#### **s** ファイル

ソース・コード・コントロール・システム (SCCS) において、SCCS ライブラリに保存されていて、元のファイルのテキストおよびそれに適用されたデルタのリストを含むファイルのこと。

## え

### 演算子

正規表現において、リテラルな文字としてではなく、別の意味を表すと解釈される文字。たとえば、1組の大カッコ ( [ ] ) は、大カッコで囲まれた文字の中の任意の 1 文字に対する照合を行う演算子である。

## か

### 改行

次に続くテキストを新しい行の左マージンに移動させる文字、あるいはユーザ入力最後の最後を示す文字。通常 Return キーが改行のために使用される。改行文字は、`awk` などのレコードに対応したプログラムのための省略時のレコード・セパレータである。

## し

### 字句解析プログラム

入力を解析するだけでなく、入力の解析を支援するために要素の分類を行う、プログラムまたはプログラム・フラグメント。`lex` プログラムは、字句解析プログラムの作成を支援する。

パーサ も参照

### 従属

従属ファイルとも呼ばれる。`make` ユーティリティで、作成されるファイル (target) が従属するエンティティ。たとえば、ソース・ファイルは、オブジェクト・モジュールの従属である。

### 従属行

`make` ユーティリティにおいて、記述ファイル中で所定のターゲットが従属している従属ファイルを記述している行。

### 従属ファイル

従属 を参照

### 照合記号

正規表現で、多文字文字列を使用して 1 文字を表す照合順序の中で、たとえば、小文字などの、使用可能な文字の特定のサブセットを定義する名前。

## す

### スクリプト

sed エディタにおいて、入力ファイルに適用されるエディット・コマンドのリスト。

## そ

### ソース・コード・コントロール・システム

SCCS (ソース・コード・コントロール・システム) を参照

## た

### ターゲット

ターゲット・ファイルとも呼ばれる。make ユーティリティでは、その従属から組み立てられるエンティティのこと。実行可能プログラムは、1 つまたは複数のオブジェクト・モジュールから組み立てられるターゲットである。

## ち

### チェックアウト

リビジョン・コントロール・システム (RCS) において、RCS ライブラリからファイルまたはリビジョンを取り出すこと。

### チェックイン

リビジョン・コントロール・システム (RCS) において、ファイルまたはリビジョンを RCS ライブラリに保存すること。

## て

### デルタ

RCS ファイルまたは SCCS ファイルの特定のバージョンを構成するための変更内容の集合。

## と

### トークン

m4 マクロ・プロセッサの場合は、マクロ名になり得る認識可能なエンティティのこと。トークンは、英数字以外の文字で区切られた英数字で構成されている。他のトークンを中に含むことはできない。



lex によって生成された字句解析プログラムおよび yacc によって生成されたパーサの場合は、パーサまたは字句解析プログラムによって定義された、独立した最小の意味単位。トークンとしては、データ、言語キーワード、識別子、または言語構文のその他の要素がある。

## は

### パーサ

入力を解釈し、それに対する処理方法を決定するプログラムまたはプログラム・フラグメント。yacc プログラムは、パーサの作成を支援する。

### バージョン・コントロール・システム

ファイルのリビジョンおよび構成の、組織化および保守を支援するためのソフトウェア・ツール。特に、ソース・プログラム、ドキュメント、およびデータ・ファイルなどのリビジョンの、格納、ロギング、取り出し、および識別を自動化する。

### バージョン・コントロール・ファイル

バージョン・コントロール・システムにおいて、元のテキストおよび作成された一連のリビジョン (デルタ) によって構成されるファイル。このファイルは、RCS では RCS ファイル、SCCS では s ファイルと呼ばれる。

### バージョン・コントロール・ライブラリ

RCS または SCCS などのバージョン・コントロール・システムの下で構成および保守されるファイルがあるディレクトリ。

### パターン・スペース

sed エディタで現在編集されている行の範囲を示す。パターン・スペースは、アドレスまたはアドレスのペアによって選択される。

## ふ

### フィールド

awk における入力レコードの 1 要素。各フィールドは、フィールド・セパレータで区切られる。フィールド・セパレータは、ユーザによる指定が可能であり、省略時の設定は任意の数のホワイト・スペースである。レコードの始まりおよび終了もフィールド・セパレータである。

レコード も参照

### フィールド変数

awk の入力レコードのフィールドである変数。フィールド変数は他のどの変数としてでも処理することができる。

## ま

### マクロ定義

m4 マクロ・プロセッサまたは make ユーティリティで使用される。マクロ名を作成するとともに、マクロに行わせるテキストおよび引数の置換を定義する文。

## れ

### レコード

awk において、連続した 2 つのレコード・セパレータの出現の間にある情報。レコード・セパレータは、ユーザ指定が可能であり、省略時の設定は改行文字である。ほとんどの場合、1 レコードは、入力ファイルの 1 行と考えてよい。ファイルの先頭および終端もレコード・セパレータである。

## ろ

### ロック

バージョン・コントロール・システムでは、バージョン・コントロール・ファイルが編集のためにチェックアウトされているというフラグを立て、そのフラグを使用することを指す。

### ロック機構

バージョン・コントロール・システムにおいて、ファイルに重複および並列した変更が行われないようにする方法。SCCS は、p ファイルを使用して現在どのファイルが編集中であるかを示す。RCS は、RCS ファイルにロック情報を挿入することによってロックする。

## わ

### 若いファイル

make ユーティリティにおいて、ターゲットの作成以降に変更された従属ファイル。

## 索引

### 数字および記号

\$  
（ドル記号 を参照）  
\*  
（アスタリスク を参照）  
+  
（プラス記号 を参照）  
.  
（ピリオド を参照）  
/  
（スラッシュ を参照）  
:  
（コロンの記号 を参照）  
;  
（セミコロン を参照）  
?  
（疑問符 を参照）  
@  
（アットマーク を参照）  
\  
（バックスラッシュ を参照）  
^  
（山形記号 を参照）  
|  
（縦線 を参照）  
( )  
（カッコ を参照）  
[ ]  
（大カッコ を参照）

{ }  
（中カッコ を参照）  
&  
（アンパサンド を参照）  
< >  
（山カッコ を参照）

### A

**admin**コマンド ..... 6-23, 6-31  
**awk**  
    printfコマンド ..... 2-5  
    printコマンド ..... 2-5  
**awk**コマンド ..... 2-1, 2-23  
    pipe ..... 2-24  
    split関数 ..... 2-9  
    アクション ..... 2-3, 2-14  
    アクションとパターンの区別 ..... 2-3  
    アクションの演算子 ..... 2-15  
    関係式 ..... 2-12  
    関数 ..... 2-17  
        その他の ..... 2-18  
        文字列 ..... 2-17  
    組み込み ..... 2-9  
    構文 ..... 2-3  
    コマンド行構文 ..... 2-2  
    コマンド行に入力 ..... 2-4  
    作成 ..... 2-7  
    省略 ..... 2-4

初期化された値 ..... 2-7  
 処理 ..... 2-7  
 スラッシュ ..... 2-11  
 正規表現の開始フィールド... 2-12  
 正規表現フィールドの終了... 2-12  
 制御構造 ..... 2-20  
 単純 ..... 2-7  
 動作の順序 ..... 2-4, 2-14  
 内部 ..... 2-9  
 配列 ..... 2-8, 2-9  
 パターン ..... 2-3  
 パターンとしての正規表現... 2-11  
 バックスラッシュ ..... 2-11  
 フィールド ..... 2-1, 2-7  
 フィールド・セパレータ ..... 2-2  
 フィールド変数 ..... 2-7  
 フラグ ..... 2-2  
 プログラム ..... 2-3  
 プログラム構造 ..... 2-4  
 プログラムのセミコロン 2-3, 2-15  
 プログラムの注釈 ..... 2-20  
 変数 ..... 2-6  
     RLENGTH ..... 2-17  
     RSTART ..... 2-17  
 文字の連結 ..... 2-23  
 文字列 ..... 2-7  
 文字列処理 ..... 2-9  
 文字列の操作 ..... 2-13, 2-23  
 リダイレクト ..... 2-24  
 レコード ..... 2-1  
 レコード・レンジ ..... 2-13  
 レコード・レンジの指定 ..... 2-13

## B

**BEGIN**文 ..... 2-23  
     lex ..... 4-16

## C

**changecom**マクロ, **m4** ..... 5-10  
**changequote**コマンド, **m4** .... 5-4  
**ci**コマンド ..... 6-12, 6-15  
**create**コマンド ..... 6-22

## D

**deledit**コマンド ..... 6-28  
**delget**コマンド ..... 6-28  
**delta**コマンド ..... 6-27  
**diffs**コマンド ..... 6-29  
**divert**マクロ, **m4** ..... 5-12  
**divnum**マクロ, **m4** ..... 5-13  
**dlen**マクロ, **m4** ..... 5-14  
**dnl**コマンド, **m4** ..... 5-4  
**dumpdef**マクロ, **m4** ..... 5-15

## E

**edit**コマンド ..... 6-26  
     -rオプション ..... 6-27  
     分岐のマージ ..... 6-29  
**egrep**コマンド  
     ( **grep**コマンド を参照 )  
**endmarker**トークン .... 4-21, 4-27  
     値 ..... 4-26  
**error**トークン, **yacc** ..... 4-32  
**eval**マクロ, **m4** ..... 5-11

## F

**fgrep** コマンド  
( **grep** コマンド を参照 )

## G

**gawk** コマンド  
( **awk** コマンド を参照 )  
**get** コマンド ..... 6-25  
  -p オプション ..... 6-26  
**grep** コマンド ..... 1-10  
  オプション ..... 1-10  
**g** ファイル ..... 6-3

## H

**help** コマンド  
  SCCS ..... 6-30

## I

**ifdef** マクロ , **m4** ..... 5-11  
**ifndef** マクロ , **m4** ..... 5-13  
**include** マクロ , **m4** ..... 5-12  
**index** マクロ , **m4** ..... 5-15  
**info** コマンド ..... 6-26

## L

**LC\_TYPE** 環境変数  
( 照合順序 を参照 )  
**len** マクロ , **m4** ..... 5-14  
**lex** プログラム ..... 4-1  
  **yacc** と併用 ..... 4-18

**yylex** 関数の処理 ..... 4-12  
引用符 ..... 4-7  
エスケープ文字 ..... 4-7  
計算機の例 ..... 4-38  
使用方法 ..... 4-17  
代替 ..... 4-9, 4-12  
入力ストリームへ入力を戻す . 4-9  
部分列の検索 ..... 4-9  
ワイルドカードとの照合 ..... 4-8

**lex** ユーティリティ  
  置換文字列 ..... 4-5  
  展開 ..... 4-5  
  マクロ ..... 4-5  
**lex** ライブラリ ..... 4-4, 4-14

## M

**m4** 出力の空白行(擬似) ..... 5-4  
**m4** マクロ・プリプロセッサ ..... 5-1  
  **change** マクロ ..... 5-10  
  **divert** マクロ ..... 5-12  
  **divnum** マクロ ..... 5-13  
  **dlen** マクロ ..... 5-14  
  **dnl** マクロ ..... 5-4  
  **dumpdef** ..... 5-15  
  **eval** マクロ ..... 5-11  
  **ifdef** マクロ ..... 5-11  
  **ifndef** マクロ ..... 5-13  
  **index** マクロ ..... 5-15  
  **len** マクロ ..... 5-14  
  **maketemp** マクロ ..... 5-13  
  **print** マクロ ..... 5-15  
  **substr** マクロ ..... 5-14  
  **translit** マクロ ..... 5-15

undivertマクロ .....	5-13	例 .....	7-24
一時ファイル .....	5-12, 5-13	記述ファイルのテスト .....	7-24
引用符で囲む文字 .....	5-4	規則	
引用符変更マクロ .....	5-10	単一サフィックス .....	7-21
演算 .....	5-11	定義 .....	7-21
再帰 .....	5-2	内部 .....	7-9, 7-19
システム・プログラムの使用 .....	5-13	内部, 簡素化 .....	7-10
出力内の擬似空白行 .....	5-4	規則ファイルの例 .....	7-21e
条件付きアクション .....	5-13	コマンド構文 .....	7-3
ネストされたマクロでの引用符 .....	5-5	コマンドの実行 .....	7-3
表示 .....	5-15	再帰 .....	7-15
ファイルのインクルード .....	5-12	サフィックス .....	7-19
マクロ		置換 .....	7-20
組み込み .....	5-8	追加 .....	7-20
内部 .....	5-8	シェルの呼び出し .....	7-3
マクロ構文 .....	5-1	使用方法 .....	7-3
マクロ再定義 .....	5-5	従属がないターゲット・ファイ	
マクロ定義 .....	5-3	ル .....	7-3
他のマクロの置き換え .....	5-3	従属リスト .....	7-7
他のマクロの項 .....	5-3	条件付きアクション .....	7-15
マクロ定義の取り消し .....	5-11	操作 .....	7-2
マクロ内の空白文字 .....	5-6	他のファイルのインクルード .....	7-24
マクロ引数 .....	5-6, 5-7	ターゲット・ファイルビルド手	
文字列変更 .....	5-14	順 .....	7-2
リダイレクト .....	5-12	内部 .....	7-9
main関数, yacc .....	4-20, 4-21	内部マクロ .....	7-15
\$(MAKE)マクロ .....	7-15	最初の旧ファイル .....	7-18
記述ファイルのテスト .....	7-24	ターゲット・ファイル名... ..	7-16
MAKEFLAGSマクロ .....	7-18	ターゲット・ファイル名, 従属	
maketempマクロ, m4 .....	5-13	行 .....	7-17
makeの使用法 .....	7-3	ファイル名プレフィックス .....	7-18
makeユーティリティ .....	7-1	古いファイルのリスト .....	7-18
環境変数 .....	7-18	ファイルの更新 .....	7-3
記述ファイル .....	7-4,	ファイルのビルド .....	7-3
7-5, 7-11, 7-25e		古いファイル .....	7-3, 7-18

分散システム .....	7-2
マクロ定義.....	7-3, 7-11, 7-15
マクロの置換 .....	7-12
優先順位 .....	7-3
呼び出しのネスト .....	7-15
例.....	7-9

## N

<code>\n</code>	(埋め込まれた改行文字 を参照)
-----------------	------------------

## P

<b>printf</b> コマンド	
awk内への出力 .....	2-5
<b>print</b> コマンド	
awk内への出力 .....	2-5
<b>print</b> マクロ, <b>m4</b> .....	5-15
<b>prs</b> コマンド.....	6-30
<b>p</b> ファイル.....	6-7

## R

<b>RCS</b> .....	6-1
ciコマンド .....	6-12, 6-15
IDキーワード .....	6-13
rcsdiffコマンド .....	6-17
新しいリリースの作成 .....	6-15
ファイル記憶領域 .....	6-3
ファイルの同時編集の防止....	6-6
ファイルの取り出し	
バージョン指定.....	6-14

ファイルのバージョン .....	6-3
識別 .....	6-4
ファイル名.....	6-12
ライブラリ.....	6-3
作成 .....	6-9
名前 .....	6-5
ファイルの取り出し .....	6-14
ファイルの配置.....	6-12
ライブラリのセキュリティ....	6-9
ライブラリへの新しいファイルの配	
置 .....	6-12
ライブラリからのファイルの取り出	
し .....	6-14
バージョン指定.....	6-14
<b>rcsdiff</b> コマンド.....	6-17
<b>rsc</b> コマンド	
昨日 .....	6-10
<b>RCS</b> ファイル .....	6-3
説明 .....	6-4
<b>RCS</b> ライブラリからのファイルの取り	
出し .....	6-14
バージョン指定 .....	6-14
<b>RCS</b> ライブラリのセキュリティ .	6-9
<b>RCS</b> ライブラリへのファイルの配	
置.....	6-12
<b>reduce</b> アクション, <b>yacc</b> .....	4-35
<b>REJECT</b> アクション, <b>lex</b> .....	4-9
<b>REJECT</b> アクション, <b>lex</b>	
代替 .....	4-12
<b>return</b> 文, <b>lex</b> .....	4-19
<b>RLENGTH</b> 変数, <b>awk</b> .....	2-17
<b>RSTART</b> 変数, <b>awk</b> .....	2-17

## S

- SCCS** ..... 6-1
  - adminコマンド ..... 6-23, 6-31
  - deleditコマンド ..... 6-28
  - delgetコマンド ..... 6-28
  - deltaコマンド ..... 6-27
  - diffsコマンド ..... 6-29
  - editコマンド ..... 6-26
    - rオプション ..... 6-27
    - 分岐のマージ ..... 6-29
  - getコマンド ..... 6-25
    - pオプション ..... 6-26
  - gファイル ..... 6-3
  - helpコマンド ..... 6-30
  - IDキーワード ..... 6-23
    - 位置づけ ..... 6-24
    - 要求 ..... 6-25
  - infoコマンド ..... 6-26
  - prsコマンド ..... 6-30
  - pファイル ..... 6-7
  - sccsdiffコマンド ..... 6-29
  - 新しいリリースの作成 ..... 6-27
  - キーワード ..... 6-23
  - コマンド ..... 6-34
  - コマンドの作成 ..... 6-22
  - ファイル ..... 6-3, 6-7
  - ファイル記憶領域 ..... 6-3
  - ファイルの状態の取り出し... 6-26
  - ファイルのバージョン ..... 6-3
    - 識別 ..... 6-4
  - ファイル名 ..... 6-22, 6-23
  - 複数ファイルの取り出し ..... 6-26
  - ライブラリ ..... 6-3
    - 作成 ..... 6-9
  - セキュリティ ..... 6-9, 6-33
  - 名前 ..... 6-5, 6-9
  - パス指定 ..... 6-9
  - ファイルの取り出し ..... 6-25
  - ファイルの取り出し, バージョン指定 ..... 6-25
  - ファイルの取り出し, 標準出力への書き込み ..... 6-26
  - ファイルの取り出し, 編集用 ..... 6-26
  - ファイルの配置 ..... 6-22
  - ライブラリへのファイルの配置 ..... 6-22
  - ライブラリからのファイルの取り出し ..... 6-25
  - バージョン指定 ..... 6-25
  - 標準出力への書き込み ..... 6-26
  - 編集用 ..... 6-26
- sccsdiff**コマンド ..... 6-29
- sccs**コマンド ..... 6-20
  - dオプション ..... 6-9
  - オプション, リスト ..... 6-33
  - 機能 ..... 6-20
- SCCSのIDキーワード** ..... 6-23
- SCCSファイルの状態の取り出し** ..... 6-26
- SCCSファイルの分岐のマージ** ..... 6-29
- SCCSライブラリからのファイルの取り出し** ..... 6-25
- バージョン指定 ..... 6-25
- 標準出力への書き込み ..... 6-26
- 編集用 ..... 6-26
- SCCSライブラリのセキュリティ** 6-9, 6-33
- SCCSライブラリへのファイルの配置** ..... 6-22



<b>sed</b> エディタ .....	3-1	編集行の選択 .....	3-4
アドレス .....	3-4	ホールド・エリア .....	3-9
アンパサンドの使用 .....	3-11	ホールド・エリアの使用 .....	3-9
エスケープ文字 .....	3-11	文字列の操作 .....	3-11
オプションの組み合わせ .....	3-3	<b>sed</b> エディタ内の照合の繰り返し .....	3-12
行番号 .....	3-5	<b>sed</b> エディタ内の複数の照合 .....	3-12
コマンド行構文 .....	3-2	<b>shift</b> アクション, <b>yacc</b> .....	4-34
コマンド		<b>shift</b> コマンド, <b>m4</b> .....	5-6
バッファ操作 .....	3-9	<b>SID</b> .....	6-4
フロー制御 .....	3-10	<b>sinclude</b> マクロ, <b>m4</b> .....	5-12
編集 .....	3-7	<b>split</b> 関数, <b>awk</b> .....	2-9
コマンド行 .....	3-3	<b>substr</b> マクロ, <b>m4</b> .....	5-14
コマンド構文 .....	3-6, 3-7	<b>syscmd</b> マクロ, <b>m4</b> .....	5-13
コマンド・スクリプト .....	3-2	<b>s</b> ファイル .....	6-3
コマンドの修飾 .....	3-12		
コンテキスト・アドレス .....	3-5	<b>T</b>	
照合の繰り返し .....	3-12	<b>translit</b> マクロ, <b>m4</b> .....	5-15
使用の制限 .....	3-5		
スラッシュの使用 .....	3-5	<b>U</b>	
制御構造 .....	3-10	<b>undivert</b> マクロ, <b>m4</b> .....	5-13
制限 .....	3-2		
セミコロンの使用 .....	3-2	<b>W</b>	
テキストの置換後 .....	3-12	<b>what</b> コマンド .....	6-24
動作手順 .....	3-3		
入出力 .....	3-1	<b>Y</b>	
入力ファイル, 処理 .....	3-3	<b>y.tab.c</b> ファイル .....	4-21
パターン・スペース .....	3-3	<b>y.tab.h</b> ファイル .....	4-39, 4-43
バックスラッシュ .....	3-11	lex仕様ファイルでの使用方法 .....	4-19
バックスラッシュの使用 .....	3-7t	トークンのリスト .....	4-22
ファイルへの書き込み .....	3-12	<b>yacc</b> プログラム .....	4-18
複数 .....	3-3		
フラグ .....	3-2		
フラグの使用 .....	3-12		

global変数 ..... 4-25  
 lexと併用 ..... 4-18  
 エラー ..... 4-31  
 開始記号 ..... 4-25  
 ガイドライン ..... 4-30  
 規則 ..... 4-26  
 キーワードの処理 ..... 4-24  
 空文字 ..... 4-30  
 空文字列 ..... 4-27  
 計算機の例 ..... 4-38  
 結合性 ..... 4-24  
 再帰 ..... 4-31  
 先読みトークン, 消去 ..... 4-33  
 省略時の設定 ..... 4-28  
 宣言 ..... 4-23  
 デバッグ・モード ..... 4-38  
 トークン番号 ..... 4-25  
 トークン名の検出 ..... 4-19  
 パラメータ・キーワード ..... 4-28  
 文法ファイル ..... 4-23  
   プログラム部 ..... 4-29  
 優先順位 ..... 4-24  
 ライブラリ・ルーチン ..... 4-21  
**yy.lex.c**ファイル ..... 4-4, 4-17  
**yychar**変数 ..... 4-22  
**yyerror**関数 ..... 4-20, 4-21  
**yy leng**変数 ..... 4-11  
**yyless**関数, **lex** ..... 4-12  
**yy lex**関数 ..... 4-4, 4-20  
   yyparseによる呼び出し ..... 4-20  
   パーサに含まれる関数 ..... 4-29  
   必要条件 ..... 4-22  
**yy lval**変数 ..... 4-20, 4-36  
**yy more**関数 ..... 4-12  
**yy parse**関数 ..... 4-20, 4-21

**yytext**変数 ..... 4-11  
**yywrap**関数  
   lex ..... 4-15

## あ

### アクション

awk ..... 2-3, 2-14  
 reduce ..... 4-35  
 yacc  
   shift ..... 4-34  
   あいまいな ..... 4-36  
   競合 ..... 4-36  
 yaccパーサ ..... 4-20, 4-27  
 解決策 ..... 4-37  
 字句解析プログラム ..... 4-2,  
   4-3, 4-6, 4-10  
   1つの正規表現に対する複数のア  
     クション ..... 4-10  
   ヌル・アクション ..... 4-10

### アスタリスク

make ..... 7-6, 7-13  
   \$\*マクロ ..... 7-18  
 拡張正規表現 ..... 1-4  
 基本正規表現 ..... 1-3  
 正規表現 ..... 1-6

### 新しいリリースの作成

RCS ..... 6-15  
 SCCS ..... 6-27

### アットマーク

make ..... 7-8  
   \$@マクロ ..... 7-16  
   \$\$@マクロ ..... 7-17  
 アドレス, **sed**エディタ ..... 3-4  
 アンパサンド  
   make ..... 7-13

sed ..... 3-11

## い

一時ファイル, **m4** ..... 5-12, 5-13

引用符で囲まれた文字列

lex ..... 4-7

引用符で囲む文字, **m4** ..... 5-10

引用符で囲む文字列

m4 ..... 5-4

引用符変更マクロ, **m4** ..... 5-10

## う

埋め込まれた改行文字

sed ..... 3-5

## え

エスケープ文字

lex ..... 4-7

sed ..... 3-11

拡張正規表現 ..... 1-4

基本正規表現 ..... 1-3

演算, **m4** ..... 5-11

演算子

アクション, **awk** ..... 2-15

関係, **awk** ..... 2-12

正規表現の定義 ..... 1-1

ブール, **awk** ..... 2-3, 2-13

## か

開始記号, **yacc** ..... 4-25

開始条件, **lex** ..... 4-16

設定 ..... 4-16

拡張正規表現 ..... 1-4

カッコ

**awk** ..... 2-3

**m4** ..... 5-7, 5-12

**make** ..... 7-12

拡張正規表現 ..... 1-4

基本正規表現 ..... 1-3

カレット

(山形記号を参照)

環境変数, **make** ..... 7-18

関係式

**awk** ..... 2-12

関数, **awk** ..... 2-17

## き

記号, **yacc** ..... 4-22, 4-24

開始 ..... 4-25

記述ファイル

**make** ..... 7-25e

テスト ..... 7-24

コマンド... 7-6, 7-8, 7-11, 7-15

規則

**lex** ..... 4-6

矛盾 ..... 4-8

**make** ..... 7-2

内部 ..... 7-9

**yacc** ..... 4-26

入力との照合 ..... 4-8

基本正規表現 ..... 1-2

疑問符

**make** ..... 7-6

\$?マクロ ..... 7-18

拡張正規表現 .....	1-4
正規表現 .....	1-6
行番号	
sed .....	3-5
キーワード, 処理, <b>yacc</b> .....	4-24
結合性 .....	4-24
優先順位 .....	4-24

## く

空文字	
lex .....	4-14
文法規則 .....	4-30
空文字列, <b>yacc</b> .....	4-27
組み込みマクロ	
(マクロを参照)	

## こ

構文	
(個々のユーティリティ・エン	
トリを参照)	
コマンド定義, <b>m4</b> .....	5-3
コメント文字, <b>m4</b> .....	5-10
コロソ, <b>yacc</b> .....	4-26
コンテキスト・アドレス	
sed .....	3-5

## さ

再帰	
<b>m4</b> .....	5-2
<b>make</b> .....	7-15
<b>yacc</b> .....	4-31
先読み	
字句解析プログラム .....	4-3

先読みトークン, <b>yacc</b>	
消去 .....	4-33
番号 .....	4-22

## し

字句解析プログラム .....	4-2
BEGIN文 .....	4-16
endmarkerトークン .....	4-21
lexで解釈できない行 .....	4-15
lexライブラリ .....	4-4, 4-14
printf関数 .....	4-11
return文 .....	4-19
<b>yacc</b> パーサとの併用 .....	4-10
yy <sub>leng</sub> 変数 .....	4-11
yy <sub>lval</sub> 変数 .....	4-20
yy <sub>more</sub> 関数 .....	4-12
yy <sub>text</sub> 変数 .....	4-11
yy <sub>wrap</sub> 関数 .....	4-15
アクション... 4-2, 4-3, 4-6, 4-10	
1つの正規表現に対する複数のア	
クション .....	4-10
一致した文字列の長さ .....	4-11
開始条件 .....	4-16
規則 .....	4-6
規則部 .....	4-6
規則を指定しない場合のアクショ	
ン .....	4-4
空文字 .....	4-14
照合した文字列のプリント... 4-11	
仕様ファイル	
定義, y.tab.hの使用方法... 4-19	
省略時のアクション .....	4-4
正規表現 .....	4-3, 4-6, 4-7
生成 .....	4-17



照合の制限.....	1-6, 1-8
照合留意事項 .....	1-8
選択文字の照合 .....	1-8
等価クラス.....	1-9
パターンの保存と再使用 .....	1-7
複数の指定.....	1-8
複数の連結.....	1-3
文字クラス.....	1-8
制御構造	
awk .....	2-20
sed .....	3-10
セミコロン	
awk .....	2-3, 2-15
lex .....	4-6
sed .....	3-2
yacc.....	4-26
宣言, <b>yacc</b> .....	4-24

## そ

ソース・コード・コントロール・シス テム .....	6-1, 6-36
(SCCS も参照)	
ソース・ファイルの保管 (RCS, SCCS を参照)	
ソース・ファイルのリビジョン・コン トロール (RCS, SCCS を参照)	

## た

大カッコ	
拡張正規表現 .....	1-4
基本正規表現 .....	1-3
タイム・スタンプ	
makeユーティリティが使用...	7-2

## 縦線

lex .....	4-10
yacc.....	4-26
拡張正規表現 .....	1-4
ターゲット・ファイル	
makeでのビルド手順 .....	7-2
従属なし.....	7-3
定義 .....	7-2

## ち

### 中カッコ

awk .....	2-3
lex .....	4-5, 4-15
make .....	7-12
yacc.....	4-25
拡張正規表現 .....	1-4
基本正規表現 .....	1-3

## て

### テキストの検索

grepコマンド .....	1-10
テキスト・ファイルの処理 (awkコマンド, m4マクロ・ブ リプロセッサ, sedエディタを 参照)	
デルタ.....	6-3

## と

### ドル記号

awk .....	2-12
m4 .....	5-6
make .....	7-6, 7-14
sed.....	3-5

拡張正規表現 ..... 1-4  
 基本正規表現 ..... 1-3  
 トークン  
   m4  
     解釈 ..... 5-2  
     定義 ..... 5-2  
     定義 ..... 4-18  
     名前の検出 ..... 4-19  
     リスト ..... 4-22  
     トークン番号, **yacc** ..... 4-25

## な

内部マクロ  
 ( マクロ を参照 )

## に

入出力ルーチン, **lex** ..... 4-13  
   空文字 ..... 4-14  
   指定変更 ..... 4-13  
   変換テーブル ..... 4-14

## は

パイプ, **awk** ..... 2-24  
 配列, **awk** ..... 2-8, 2-9  
 パターン  
   awk ..... 2-4, 2-11  
   レコード・レンジ ..... 2-13  
 パターン・スペース ..... 3-3  
 バックスラッシュ  
   awk ..... 2-11  
   sed ..... 3-11

sedでの ..... 3-7t  
   拡張正規表現 ..... 1-4  
   基本正規表現 ..... 1-3  
   正規表現内 ..... 1-2  
 パーサ ..... 4-20  
   endmarkerトークン ..... 4-27  
   main関数 ..... 4-19, 4-21  
   reduceアクション ..... 4-35  
   shiftアクション ..... 4-34  
   yychar変数 ..... 4-22  
   yyerror関数 ..... 4-20, 4-21  
   yylex関数 ..... 4-20, 4-29  
   yylval変数 ..... 4-20  
   あいまいなアクション ..... 4-36  
   アクション ..... 4-27  
   エラーの処理 ..... 4-31  
   解決策 ..... 4-37  
   規則のアクション中の制御... 4-28  
   競合するアクション ..... 4-36  
   字句解析プログラムと併用... 4-18  
   訂正が可能 ..... 4-32  
 パーセント記号 ..... 4-23  
   lex ..... 4-5, 4-15  
   SCCS ..... 6-23  
   yacc ..... 4-25

## ひ

非終端記号 ..... 4-22, 4-24, 4-27  
   内部 ..... 4-28  
 非対話型編集  
   ( sedエディタ を参照 )  
 ピリオド  
   拡張正規表現 ..... 1-4

基本正規表現 ..... 1-3

## ふ

### ファイル

RCSでの名前 ..... 6-12

RCSでのバージョン ..... 6-3

RCSまたはSCCSでのバージョン

識別 ..... 6-4

SCCSでの名前 ..... 6-22

SCCSでのバージョン ..... 6-3

作成

RCS ..... 6-12

SCCS ..... 6-22

状態の取り出し, SCCS ..... 6-26

取り出し, SCCS ..... 6-25

複数ファイルの取り出し,

SCCS ..... 6-26

編集, SCCS ..... 6-26

ファイルの終わり

lex ..... 4-15

sed ..... 3-5

ファイルの同時編集, **RCS**による管

理 ..... 6-6

ファイルの同時編集, **SCCS**による防

止 ..... 6-7

ファイルの編集, 同時, **RCS**による管

理 ..... 6-6

ファイルの編集, 同時, **SCCS**による

防止 ..... 6-7

ファイル名

SCCS ..... 6-23

フィールド

awk ..... 2-1, 2-7

複数の**SCCS**ファイルの取り出

し ..... 6-26

部分列 ..... 4-9

フラグ

SCCSファイル ..... 6-25

リスト ..... 6-32

sed ..... 3-12

プラス記号

拡張正規表現 ..... 1-4

正規表現 ..... 1-6

プログラムの作成

( lexプログラム, yaccプログラ

ムを参照 )

プログラムのビルド

( makeユーティリティを参照 )

文法ファイル, **yacc** ..... 4-23

エラー ..... 4-31

ガイドライン ..... 4-30

規則 ..... 4-26

宣言 ..... 4-23

内容 ..... 4-23

プログラム部 ..... 4-29

## へ

変換テーブル, **lex** ..... 4-14

変数

awk ..... 2-6

global, yacc ..... 4-25

組み込み ..... 2-9

作成 ..... 2-7

初期化された値 ..... 2-7

処理 ..... 2-7

数値 ..... 2-7

単純 ..... 2-7



内部 ..... 2-9  
配列 ..... 2-8, 2-9  
フィールド ..... 2-7

## ま

マクロ ..... 5-2  
  (m4マクロ・プリプロセッサも参照)  
  lex ..... 4-5  
  展開 ..... 4-5  
  make ..... 7-11  
  古いファイルのリスト ..... 7-18  
  引用符で囲む ..... 5-5  
  組み込み  
    m4 ..... 5-8  
    make ..... 7-15  
  再定義, m4 ..... 5-5  
  遅延 ..... 5-4  
  置換, make ..... 7-12  
  定義  
    make ..... 7-11  
  定義, m4 ..... 5-2, 5-3  
  他のマクロの置き換え ..... 5-3  
  他のマクロの項 ..... 5-3  
  定義, make ..... 7-3  
  定義のチェック, m4 ..... 5-11  
  定義の優先順位, make ..... 7-3  
  展開, m4  
    自然再帰 ..... 5-2  
  取り消し, m4 ..... 5-11  
  内部  
    m4 ..... 5-8

make ..... 7-15  
make, ターゲット・ファイル, 最初の旧ファイル ..... 7-18  
make, ターゲット・ファイル名 ..... 7-16  
make, ターゲット・ファイル名, 従属行 ..... 7-17  
make, ファイル名プレフィックス ..... 7-18  
  引数, m4 ..... 5-6, 5-7  
マクロ再定義, **m4** ..... 5-5  
マクロ定義  
  (makeユーティリティを参照)  
  m4マクロ・プリプロセッサ ... 5-3  
マクロ定義の取り消し, **m4** .... 5-11  
マクロ内の空白文字, **m4** ..... 5-6  
マクロの取り消し, **m4** ..... 5-11

## も

文字クラス  
  正規表現 ..... 1-8  
文字列処理  
  awk ..... 2-9  
文字列の処理  
  lex ..... 4-5  
文字列の操作  
  awk ..... 2-13, 2-23  
  sed ..... 3-11  
文字列変更  
  m4 ..... 5-14  
文字列変数, **awk** ..... 2-7

## や

---

山形記号  
    awk ..... 2-12  
    make ..... 7-12  
    拡張正規表現 ..... 1-4  
    基本正規表現 ..... 1-3  
山カッコ  
    lex ..... 4-7  
    make ..... 7-18

## ゆ

---

有限状態オートマトン ..... 4-2,  
    4-3, 4-33  
    スタックの使用 ..... 4-33

## ら

---

ライブラリ, **RCS**  
    (RCS を参照)  
ライブラリ, **SCCS**  
    (SCCS を参照)

## り

---

リダイレクション  
    awk ..... 2-24  
リダイレクト  
    m4 ..... 5-12  
リテラル文字列, **yacc** ..... 4-30  
リビジョン・コントロール・システ  
    ム ..... 6-1, 6-36  
    (RCS も参照)  
リリース, 新しいリリースの作  
    成 ..... 6-27  
    RCS ..... 6-15

## れ

---

レコード  
    awk ..... 2-1

## わ

---

若いファイル  
    定義 ..... 7-2

## Tru64 UNIX ドキュメントの購入方法

Tru64 UNIX ドキュメントのご購入については、弊社担当営業または日本ヒューレット・パッカートの各営業所/代理店にお問い合わせください。

各ドキュメント・キットの注文番号は以下のとおりです。ドキュメント・キットに含まれるマニュアルの内容については『ドキュメント概要』を参照してください。

キット名	注文番号
Tru64 UNIX Documentation CD-ROM	QA-6ADAA-G8
Tru64 UNIX Documentation Kit	QA-6ADAA-GZ
End User Documentation Kit	QA-6ADAB-GZ
- Startup Documentation Kit	QA-6ADAC-GZ
- General User Documentation Kit	QA-6ADAD-GZ
- System and Network Management Documentation Kit	QA-6ADAE-GZ
Developer's Documentation Kit	QA-6ADAF-GZ
Reference Pages Documentation Kit	QA-6ADAG-GZ
TruCluster Server Documentation Kit	QA-6BRAA-GZ
Tru64 UNIX 日本語ドキュメント・キット	QA-6ADJB-GZ
スタートアップ・ドキュメント・キット	QA-6ADJC-GZ
一般ユーザ・ドキュメント・キット	QA-6ADJD-GZ
システム/ネットワーク管理ドキュメント・キット	QA-6ADJE-GZ
プログラミング・ドキュメント・キット	QA-6ADJF-GZ
CDE 翻訳ドキュメント・キット	QA-6ADJG-GZ
TruCluster Server 日本語ドキュメント・キット	QA-05SJA-GZ
Advanced Server for UNIX 日本語ドキュメント・キット	QA-5U2JA-GZ



# マニュアルに対するご意見

## Tru64 UNIX

プログラミング・サポートツール・ガイド

AA-RK3NB-TE

弊社のマニュアルに関して、ご意見、ご要望、または内容の不明確な部分など、お気づきの点がございましたら、下記にご記入の上、弊社社員にお渡しくださるようお願い申し上げます。

マニュアルの採点：

	大変良い	良い	普通	良くない
正確さ(説明どおりに動作するか)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
情報量(十分か)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
分かり易さ	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
マニュアルの構成	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
図(役立つか)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
例(役立つか)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
索引(項目の検索性)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
ページ・レイアウト(情報の検索性)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

内容の不明確な部分がありましたら、以下にご記入ください：

ペ ー ジ


その他お気づきの点がございましたら、以下にご記入ください：


ご使用のソフトウェアのバージョン： \_\_\_\_\_

貴社名/部課名 \_\_\_\_\_

御名前 \_\_\_\_\_

記入日 \_\_\_\_\_

(注) 当用紙を受け取った弊社社員は、すみやかに下記にお送りください。

ビジネスクリティカルシステム統括本部 **BCS** 技術本部 **Alpha** ソフトウェア技術部