

Flex

Flex

字句スキャナ生成プログラム

Flex 2.3.7、1.03 版

1993 年 2 月

G. T. Nicol 著

Flex Copyright © by The Regents of the University of California Berkeley.
All rights reserved.

Flex is copyrighted by The Regents of the University of California Berkeley and is not covered by the GNU General Public License. It is distributed by the Free Software Foundation under the terms of the original copyright. Please read the file ‘COPYING’ in the Flex distribution for details on the Flex copyright.

This manual
Copyright © 1992, 1993 Free Software Foundation

Published by the Free Software Foundation
59 Temple Place – Suite 330
Boston, MA 02111-1307 USA
Printed copies are available for \$20 each.
ISBN 1-882114-21-3

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

Cover art by Etienne Suvasa.

序

このマニュアルでは Flex の字句スキャナ生成機能が持つ重要な点を網羅して説明しています。これには、パターン・マッチング・ルールの記述方法、スキャナの最適化方法、および、どのように Flex が POSIX に適合しているか、という点が含まれています。本書中いたるところに実例を示してありますし、有用なコードを集めた特別なセクションもあります。

このマニュアルは、チュートリアルというよりはむしろリファレンス・マニュアルになっています。題材や説明はなるべくシンプルになるよう努めました、かなり専門的になってしまったところもあります。Flex の使用経験が浅いユーザでも、最初の 2、3 章は問題なく読むことができます。しかし、実例を含む章を理解するには、少なくとも実用的な C 言語の知識と、ある程度のプログラミング経験が必要になります。特に、Flex と Bison を組み合わせて使う方法を説明した章は、前提として yacc/Bison と BNF (BNF が何であるかを知らない読者は、それを勉強する必要がでてくるでしょう) に関する知識を必要とします。

字句解析、コンパイラ・デザイン、プログラミング全般について経験のない読者には、このマニュアルの後半部分を読む前に、各領域における一般的な問題について知識を得ておくようお勧めします。この目的に役立つ良書をいくつか挙げておきます。

Compiler Design in C — Allen I. Hollub 著

Compiler Design and Construction — Arthur B. Pyster 著

Compilers: Principles, Techniques and Tools —

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman 共著

ここで、このマニュアルを書く作業を様々な方法でより容易にしてくれた以下の人々に感謝したいと思います。

Robert J. Chassell (bob@gnu.ai.mit.edu)

正しい方向へ導いてくれる多くの助言と役に立つコメントを提供してくれたこと、また、このマニュアルを整形するのに使われた Texinfo パッケージに関する作業を行ってくれたことに対して感謝します。

Vern Paxson (vern@ee.lbl.gov)

Flex を作成してくれたこと、自らの作業を中断してこのマニュアルを徹底的に校正してくれたこと、技術的な面と一般的な面の両方において役に立つコメントを提供してくれたこと、このマニュアルの多くの部分 - 特に性能に関する部分 - に影響を与えた優れたドキュメント flexdoc.1 を書いてくれたことに対して感謝します。

Brendan Kehoe (brendan@cygnus.com)

初期の草稿に対して広範な注とコメントを提供してくれたこと、リファレンス・カードの作成を手伝ってくれたことに対して感謝します。

さらにここで、全くの無報酬であるにもかかわらず、なぜ私がこのマニュアルを書くために多くの時間を割いたのか、その理由について説明しておきたいと思います。主な理由はこういことです。私自身には Free Software Foundation (FSF) ほど強い政治的指向性はありませんが、それでもフリー・ソフトウェアはこの世界に

とって非常に重要な資産であり、FSF が一貫して目的としているものは、私たちすべてがその実現のために努力すべきものであると感じるのです。

プログラミングをしている時、私はフリーのツールを多く使います。私がこうしたツールを使うのは、それがフリー（無料）であるからではなく、他の同等の製品と比較して優れているからです。例えば、gcc は入手可能な ANSI C コンパイラの中で最も良いものの 1 つであると主張できます。emacs、TeX、groff、ghostscript はいずれもきわめて役に立つテキスト処理ツールです。X11、f2c、p2c はそれぞれの分野において注目に値するものです。そして Flex はほとんどのマシン上にある標準の Lex より格段に優れています。

このことは、フリー・ソフトウェアが役に立ち、かつ、十分にサポートされていることを証明しています。こうしたプログラムのソース・コードにアクセスできたおかげで、私はそれらを研究し、そこから学ことができました。

このマニュアルは、FSF とすべてのフリー・ソフトウェアの作者の努力に対して恩返しをするための、私流の方法なのです。もし読者が GNU のツールやその他のフリーのプログラムを使っているのであれば、ご自分でも何か貢献することを試みてはいかがでしょうか？ 現在、フリーのオペレーティング・システムが 3 つ（Linux、386BSD、GNU Hurd）開発中です。そこでは、あらゆる援助、特にプログラミングとは関係ない領域での援助が必要とされています。

最後に、このマニュアル中の情報に誤りがないようにするための努力は払いましたが、なんらかの誤りや矛盾がまだ含まれているであろうということを一言断っておきたいと思います。このマニュアル中の誤りやその他の不都合な点の責任はすべて私にあり、私以外の何人の見解もしくは能力の限界を表すものではありません。また、このマニュアル中の実例の多くは C 言語の使い方として最も効率的なものではないことを私は認識していますが、このマニュアル中では性能よりも明瞭さの方がはるかに重要であると考えました。

最後にもう一度、すべてのフリー・ソフトウェアの作者に対して感謝いたします。

G.T. Nicol、1993 年 1 月 11 日

プログラムの実例について

このマニュアルの標準的な電子ディストリビューションには、このマニュアルだけではなく、マニュアルの中で示されたり言及されたりしたプログラムの実例とその他の様々な実例が、あるサブディレクトリ配下に収められています。¹ こうした実例は、読者が、すべてのコードを入力するという誤りの起りやすい作業に煩わされることなく、実例の研究や修正に集中できるようにするために提供されています。コンピュータ上に GCC と Bison がインストールされていれば、ほとんど問題なくこれらの実例をコンパイルできるはずです。しかし、ANSI C コンパイラがない場合には、問題に直面するかもしれません。

実例のプログラムをビルドするには、`examples` という名のサブディレクトリに移動して以下を実行するだけです。

```
make
```

これですべての実例がコンパイルされます。あるいは、以下のように実行することもできます。

```
make example name
```

また、実例を手作業でコンパイルすることもできます。

ディストリビューション全体を持っていない読者は、インターネット(`prep.ai.mit.edu`) から入手することができますし、Free Software Foundation に直接注文することもできます。

¹ このマニュアルの日本語訳のディストリビューションには、マニュアルの中で使われているサンプル・プログラムは含まれていません。また、Flex 2.5 のディストリビューションに含まれているサンプル・プログラムは、このマニュアルで使われているものとは異なるものです。

1 Flex 入門

この章では字句スキャン処理の概念を紹介し、Flex のようなツールの必要性を指摘します。この章の後半部分で Flex を紹介し、Flex を使うことのできる状況の実例をいくつか紹介します。

Unix および C の世界では、ファイルは通常個々のバイトが連続したものとして扱われます。個々のバイトを集めてどのようにグループ化するかという点は、プログラマが決めることです。このような抽象化は非常に強力です。というのは、どのようなファイルであってもこの抽象化方法によって表現することができるからです。しかしこの方法には短所もあり、プログラマはほとんど常に生のファイルに対して構造をあてはめなければならないと、言葉を変えると、ファイルをより意味のある部分に分割しなければならないということです。例えば、コンパイラのある部分はファイルから連続した文字を受け取り、構文チェッカが理解することのできる構成要素、例えば、数値、キーワード、文字列などにグループ化します。このようなことを行う理由は、コンパイラの言語パーサが処理を行うのは、連続した文字に対してではなく、その言語のシンボルが連続したものに対してだからです。

データベース・アプリケーションや、バイナリ・ファイルを扱うアプリケーションは、扱うデータに対してある固定されたフォーマットというものを持っていることが多く、そのフォーマットを使って入力データから意味を導き出します。テキストを入力するプログラムは通常これとは反対で、入力を単語やシンボルに分割しなければならないことが多いのですが、通常これらの単語やシンボルがどのように配置されているかを示す決まった構造というものは存在しません。したがって、テキスト処理を行うプログラムは、入力された情報を意味のあるシンボルに分割するために、字句解析もしくは字句スキャンと呼ばれる処理を行う部分を持っていることが多く、そこで入力情報の分割が行われます。このようなことを行う関数群のことを字句アナライザもしくは字句スキャナ、あるいは短く「スキャナ」と呼びます。

一般的に、スキャナを作成するのは、プログラマにとって難しいことでも面白いことでもないのですが、時間のかかる作業になることはあります。普及しているプログラミング言語のほとんどは、スキャナの作成を支援する機能が不十分です。というのは、連続した文字を単語、トークン、シンボルに分割する組み込みの機能を持っていないからです。通常はこのような仕事を行うライブラリ・ルーチンが存在しますが、柔軟でなかったり、使いにくいものであったり、あるいは、ルーチンとのやりとりにより多くのコードが必要になったりすることが多いために、実装上の細かな点によって根本的な問題が不明瞭にされてしまいます。

1 つの良い例が、C 言語で許されているすべての数値型（浮動小数、10 進整数、16 進整数、8 進整数）を処理するスキャナを C 言語で記述する場合です。これは非常に難しいということはありませんが、出来上がったコードは通常美しいとはとても言えないものでしょうし、その保守や拡張は容易でないことが多いのです。

ほとんどのプログラマが即座に断言するように、他人の書いたコードを保守するのは通常あまり楽しい作業ではありません。さらに、美しくないコードを保守するのは、楽しいというにはほど遠いものです。このように、スキャナを書くことが退屈で、その保守が難しいとなると、スキャナの作成をより容易にしてくれる方法を考えようとするに足る理由のあったことが読者にもお分かりいただけるでしょう。

1.1 問題解決手段としての Flex

ここで Flex が登場することになります。Flex はプログラマに対して、字句解析処理部分をきれいに記述し、その記述にしたがった効率的な字句スキャナを生成する方法を提供します。プログラマは Flex に対して、必要なスキャナに関する記述情報を提供します。Flex はその記述情報を使って、C 言語で書かれたスキャナを生成します。記述に使われる言語は上級言語であり、スキャナの記述に関しては C 言語よりもはるかに適しています。その上級言語を使うことで、プログラマは、文字をどのようにグループ化し、グループ化が完了した時にどのようなアクションを発生させるかを指定することができます。

注：このマニュアルのほとんどの部分は *Flex*、*Lex* の両方を対象にしています。*Lex* は (*Flex* には劣りますが) ほとんどの *Unix* システム上にある標準のスキャナ生成ユーティリティです。両者の間に違いがある場合には、*Flex* を優先させています。*Lex* については *Section 8.2 [標準 Lex]*, *page 113* で簡単に説明してあります。

ここでも 1 つの良い例がコンパイラです。前に議論したように、コンパイラの構文チェッカは、文字が連続したものではなく、言語文法の構成要素を表すトークンが連続したものを、入力として受け取る必要があります。Flex はこのような場合に最適です。Flex によって生成されたスキャナが構文チェッカとファイルの仲介役となり、構文チェッカが次の意味のあるトークンを要求するのを待ちます。Flex はファイルを読み、プログラマが与える記述にしたがって文字をグループ化して、マッチしたトークンを返却します。この処理は、スキャナもしくはパーサが終了するまで続きます。

C のコンパイラを作成する場合、このようなことを行うために必要となる Flex の記述情報は、コードの行数にして 100 行から 300 行くらいになるかもしれません。この記述情報のほとんどは、シンボル・テーブルの管理、識別子の検索、型のマッピング、ある数の値等の追加情報の返却を行うための補助的な C コードになるでしょう。こうしたコード自体は記述情報の一部ではありませんが、通常、コンパイラによって必要とされるものです。

概念的には Flex は、原材料 (文字) を取り込み、消費者 (パーサ等) がすぐに使うことができる完成品 (トークン) を製造する工場のようなものです。

1.2 一般的なプログラミング・ツールとしての Flex

Flex はコンパイラにしか使えないということはありません。読者のコンピュータ上にある、ファイルを読み込んだり、なんらかの形で文字のグループを処理する必要のあるすべてのプログラム、特に、変換フィルタや言語ツールのことを考えてみてください。こうしたプログラムのほとんどすべてを、Flex 単体、もしくは Flex と他のツールの組み合わせによって作成することができます。

1 つの良い例が文字数のカウントです。例えば、ファイルの中の全行数、個々の文字の出現回数、`'foo'` という単語の出現回数を調べるプログラムを作成したいとしましょう。標準的なツール (`grep`、`sed`、`awk`、`perl` 等) で作成することも、C 言語のプログラムを書いて作成することもできますが、Flex で作成することも可能です。他の例として、特定のキーワードを探す必要のある、メール・リーダーがあります。この場合でも、標準ツールで実現することも、Flex と C 言語で実現することもできます。

Flex を使うことで、プログラマは、スキャナの開発やファイルを構成する文字の処理にかかる時間を大幅に減らすことができます。ほとんどの場合、Flex に対する入力情報は、既存のプログラミング言語で記述されたコードと比較して、より理解しやすく、少なくとも同程度の移植性があり、保守もより簡単です。それだけでなく、Flex でスキャナを開発するのにかかる時間は、既存のプログラミング言語で同等のスキャナを開発する場合と比べきわめて短くてすむので、Flex は、プロトタイプングや、一度しか使わないプログラムやフィルタの開発に最適です。

2 Flex の起動

この章では *Flex* を起動する基本的な方法を説明し、*Flex* で使用可能なコマンドライン・オプションを簡単に説明します。

Flex は記述情報を含むファイルを入力として受け取り、スキャナ機能を持つ C のファイルに変換します。Flex を起動するためのコマンド行は以下のようになります。

```
flex [-bcdfinpstvFILT8] [-C[efmF]] [-Sskeleton] [file ...]
```

一般的には、単に 'flex' に続けて処理すべきファイル名を入力することで実行されます。

```
flex myfile.l
```

記述情報ファイル名の末尾の '.l' は、'myfile' が Flex もしくは Lex の記述ファイルであることを示唆する慣例的な方法です。名前付けの慣例としてもう一つよく見られるのが、末尾に '.lex' を使うことです。例えば、以下のようになります。

```
flex myfile.lex
```

Flex は記述情報ファイル (myfile.l) を読み込み、そこに記述されたパターンを認識するスキャナ機能を持つ 'lex.yy.c' という名前の C 言語ファイルを生成します。記述情報の中になんらかのエラーがあれば、Flex は対応するエラー・メッセージを stderr に出力します。

2.1 コマンドライン・オプション

Flex のコマンドライン・オプションは以下のような意味を持ちます。

- b '-b' オプションを指定すると 'lex.backtrack' というファイルが生成されます。このファイルはスキャナの記述情報を最適化の際に使用されます。詳細については、See Section 6.1 [Optimizing for Speed], page 85。
- c このオプションは POSIX との互換性のために提供されているだけで、実際には何もしません。POSIX では、'-c' オプションは C 言語によるアクションが使用されることを意味します。
- d このオプションを指定するとデバッグが可能になります。これにより生成されるスキャナは、実行中にスキャナの状態情報を stderr に出力します。
- f Flex に対してファスト・スキャナ (*fast scanner*) とフル・スキャナ (*full scanner*) のどちらを生成するかを指示します。詳細については、See Section 5.3 [Table Compression and Scanner Speed], page 77。'-f' (小文字) オプションと '-F' (大文字) オプションとは異なる効果を持つ点に注意してください。
- i Flex に対して大文字、小文字を区別しないスキャナを生成するよう指示します。詳細については、See Section 5.1 [Case Insensitive Scanners], page 75。

- n このオプションは Flex にとっては何の意味も持たず、POSIX との互換性のためにのみ提供されています。POSIX では、このフラグは ‘-v’ オプションによる出力を抑制するために使用されます。POSIX でのデフォルトは、テーブル・サイズが指定されない限りこのような出力を抑制するというものです。Flex ではテーブル・サイズは意味を持たないので、このフラグは冗長です。
- p ‘-p’ オプションが指定されると、Flex は性能レポートを `stderr` に出力します。スキャナの性能を向上させる方法に関する議論については、See Section 6.1 [Optimizing for Speed], page 85。
- s Flex スキャナがマッチするものを見つけることができなかった場合のデフォルトのアクションは、そのマッチされなかった入力情報を `stdout` に出力することです。‘-s’ オプションはこのようなアクションを抑制し、その代わりに入力情報がマッチしなかった時点でスキャナを異常終了させます。
- t このオプションが指定された場合、Flex は生成されたスキャナをファイル ‘`lex.yy.c`’ ではなく `stdout` に出力します。
- v Flex に対して冗長モードで動作するよう指示します。
- F Flex に対してファスト・スキャナ (*fast scanner*) を生成するよう指示します。詳細については、See Chapter 6 [スキャナの最適化], page 85。‘-F’ (大文字) は ‘-f’ (小文字) とは異なる効果を持つ点に注意してください。‘-f’ と ‘-F’ の相違点に関する情報については、See Section 5.3 [Table Compression and Scanner Speed], page 77。
- I このオプションは Flex に対して対話型スキャナを生成するよう指示します。詳細については、See Section 5.2 [Interactive Scanners], page 76。
- L デフォルトでは、デバッグを支援するために、Flex は生成されたスキャナのコード中に `#line` 指示子を書き込みます。このオプションによって `#line` 指示子の書き込みは行われなくなります。
- T Flex に対してトレース・モードで動作するよう指示します。Flex は多くのメッセージを `stderr` に出力するようになります。こうしたメッセージは、Flex を非常によく理解しているユーザ以外には無意味でしょう。
- 8 このオプションは、Flex に対して 8 ビット入力を受け付けるスキャナを生成するよう指示します。
- C[*efmF*] これらのオプションは、スキャン処理用のテーブルをどのように圧縮するかを制御します。詳細については、See Chapter 6 [スキャナの最適化], page 85。
- S*skeleton_file* Flex に対して、生成するスキャナのベースとして *skeleton_file* により指定されるスケルトン・ファイルを使用するよう指示します。主に、Flex 自体をデバッグするために使用されます。

2.2 コマンドライン・オプション (Flex 2.5 の補足情報)

Flex 2.5 では、前節 (Section 2.1 [Command Line Switches], page 9) で説明されていない、以下のオプションもサポートされています。

- h Flex に対してコマンドライン・オプションの要約情報を出力するよう指示します。
- l AT&T により実装された lex との互換性を最大限に提供します。このオプションは、性能面でかなりの悪影響を及ぼします。また、このオプションを、'-f'、'-F'、'-Cf'、'-CF'、'-+' オプションと同時に指定することはできません。Flex と Lex の (非) 互換性の問題については、Chapter 8 [Flex and Lex], page 111 を参照してください。
- w このオプションが指定されると、Flex は、警告メッセージを出力しません。
- B Flex に対してバッチ・スキャナを生成するよう指示します。これは、対話型スキャナを生成するよう指示する '-I' オプションの否定です。
- V Flex に対してバージョン番号を出力するよう指示します。
- 7 Flex に対して 7 ビット・スキャナを生成するよう指示します。これは、'-8' オプションの否定です。内部的に生成されるテーブルのサイズは、'-8' オプションが指定された場合と比較して半分になりますが、生成されるスキャナは、8 ビット文字を含む入力进行处理することができなくなります。'-Cf'、'-CF' が指定されていない場合は、明示的に '-7' を指定しない限り、8 ビット・スキャナが生成されます。
- ++ Flex に対して C++ スキャナ・クラスを生成するよう指示します。C++ スキャナについては、Section 4.6 [Flex and C++ (Flex 2.5)], page 71 を参照してください。
- ? Flex に対してコマンドライン・オプションの要約情報を出力するよう指示します。 ('-h' オプションと同じです) 。
- Ca このオプションは、スキャン処理用のテーブルを long int の配列として定義するよう Flex に通知します (デフォルトでは short int 型の配列となります) 。 RISC マシンによっては、long int の方が高速に処理されるため、スキャナの性能向上が期待できますが、その反面、テーブルのサイズは大きくなります。
- Cr このオプションを指定して生成されたスキャナは、入力に read() システム・コールを使います。デフォルトでは、対話型スキャナの場合は getc() が、バッチ (非対話型) ・スキャナの場合は fread() が使われます。
- ofile このオプションが指定されると、Flex は生成されたスキャナを file により指定されるファイルに出力します。デフォルトでは、スキャナはファイル 'lex.yy.c' に出力されます。

-Pprefix Flexにより生成されるスキャナのソース・ファイルの中では、大域変数や大域関数の名前の先頭に接頭辞 'yy' が付けられます。このオプションが指定されると、'yy' の代わりに、*prefix* により指定される文字列が接頭辞として使用されます。また、'-o' オプションが指定されない場合のスキャナ・ファイル名 'lex.yy.c' も、'lex.prefix.c' となります。

以下に、このオプションにより影響を受ける名前の一覧を示します。

```
yy_create_buffer
yy_delete_buffer
yy_scan_buffer
yy_scan_string
yy_scan_bytes
yy_flex_debug
yy_init_buffer
yy_flush_buffer
yy_load_buffer_state
yy_switch_to_buffer
yyin
yylen
yylex
yylineno
yyout
yyrestart
yytext
yywrap
```

'-+' オプションが指定されている場合は、影響を受けるのは yywrap と yyFlexLexer の 2 つだけです。

このオプションにより、yywrap() の名前が変更されてしまう点に注意してください。プログラムをリンクするためには、'prefixwrap' という名前の関数を作成する必要があります。この関数を作成したくない場合には、スキャナ定義ファイルの中で、'%option noyywrap' を指定して、リンク時に '-lfl' オプションを指定します。%option 指示子については、Section 3.9 [%option (Flex 2.5)], page 37 を参照してください。

--help Flex に対してコマンドライン・オプションの要約情報を出力するよう指示します。('-h' オプションと同じです)

--version Flex に対してバージョン番号を出力するよう指示します。('-V' オプションと同じです)

3 Flex 記述言語

この章では、スキャナ定義の構成要素を説明し、その使用例を示します。*Flex*を効率的に使用するためには、定義の個々の要素を完全に理解することが非常に重要です。したがって、初めて *Flex* を使うユーザには、時間をかけてこの章を読むことをお勧めします。

Flex スキャナ定義のほとんどの要素は、必須要素ではありません。全体的な定義フォーマットは以下のようになります。

```
定義、初期 C コード
%%
ルール
%%
他の C コード
```

各々について、以下において詳細に説明します。

3.1 コメント

C のコードが記述できるところには、どこにでもコメントを記述することができます。コメントの書式は、C のコメントの規則に従います。コメントは、記述情報に影響を与えることはありません。C スタイルのコメントは以下のようになります。

```
... /*
    */
```

これに加えて、*Flex* では '#' で始まるコメントも許されます。このようなコメントは '*lex.yy.c*' にはコピーされませんので、この形式のコメントを使うことはお勧めできません。

注: C 以外の言語 (例えば *Pascal*) のコードを生成する *Lex* も存在します。このような *Lex* ではコメントの書式はおそらく異なるでしょう。*Flex* の場合は C のコードしか生成しません。

3.2 オプションの C コード

プログラマは、2つの異なる方法を用いて、スキャナの中に直接 C のコードを含めることができます。第1の方法は、「定義、初期 C コード」セクション (最初の '%' より前の部分) にコードを含めることです。第2の方法は、「他の C コード」セクション (2 番目の '%' より後ろの部分) にコードを含めることです。どちらの場合も、コードはそのまま '*lex.yy.c*' にコピーされますので、正当なコードでなければなりません。

第1のセクション中の C コードは以下の形式になります。

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

ここで ‘%{...%}’ というペアが、C コード・ブロックの先頭と末尾を示すために使われています。この形式のコードと定義は、「定義、初期 C コード」セクションのどこにでも自由に記述することができます。定義については次の節で説明します。

C のコードが最初のカラムから始まるのでなければ、「%{...%}」というペアは必要ありません。しかし普通は、分かりやすくするために記述しておいた方が良いでしょう。もう1つのポイントは、`#ifdef` 等のように最左端のカラムから始まらなければならない、かつ、通常はスキャナ記述情報の先頭に置かれる必要のあるものが存在するということです。こうした場合、「%{...%}」に囲まれていないと、Flex はそれを定義の一部であると見なすでしょう。これが、常に ‘%{...%}’ を使うもう1つの理由です。

最後の（「他の C コード」）セクション内のコードは、そのままコピーされます。ここには特別な宣言は必要ありません。

3.3 定義

定義セクションにおいて、プログラマは、ある文字のグループに一意な識別子を与え、その識別子がその文字グループに置き換えられるようにすることができます。定義は以下のような形式になります。

```
definition_name    definition
```

`definition_name` は最初のカラムから始まらなければならない、そうしないとその定義は ‘lex.yy.c’ にそのままコピーされてしまうということに注意してください。以下に一般的な定義をいくつか挙げます。

```
DIGIT      [0-9]
LETTER     [a-z]
IDENT      [a-z_][a-z0-9_]*
ALPHANUM   {LETTER}|{DIGIT}
```

`definition name` は、そのグループの一意な識別子でなければなりません。また、`definition` はルール・セクション（後述）において正当なものであれば何でも構いません。ルール・セクションや（上の例の `ALPHANUM` の定義において示されるように）別の定義中において使われる場合には、定義は ‘{ }’ によって囲まれていなければならない。

Flex と Lex の非常に重要な相違点に、定義を展開する時、Flex は字義どおりに丸括弧 () で囲むのに対して、Lex は囲まないという点があります。¹ これは、‘^’、‘<<EOF>>’、‘\$’、‘/’ を定義中に入れることができないことを意味しています。というのは、前述の文字は丸括弧 () で囲まれた部分に入れることができないからです。

¹ 訳注：Flex 2.5 では、‘-1’ オプションを指定して生成されたスキャナは、Lex の場合と同じように、定義を展開する時に丸括弧 () で囲みません。

詳細は、Section 3.6.1 [Characters], page 17 および Chapter 8 [Flex and Lex], page 111 において説明します。

例えば、

```
FUNCTION ^[a-zA-Z_][a-zA-Z0-9_]*("
%%
{FUNCTION} printf("got a function\n");
```

は、以下のようなプログラミング・スタイルを使っている場合の、C の関数宣言にマッチするように見えます。

```
int
foo()
{
    ...
}
```

しかし実際にはうまくいきません。というのは、{FUNCTION}が展開されると、

```
(^[a-zA-Z_][a-zA-Z0-9_]*)
```

のようになりますが、これは不正だからです。このような種類の問題に関する説明については、Section 8.1.1 [Flex and POSIX], page 111 を参照してください。

3.4 %%

2つのパーセント記号が、スキャナ記述情報のルール・セクションの先頭と末尾を示します。すべての Flex 記述情報は、少なくともルール・セクションの先頭を示す '%%' を含んでいなければなりません。

3.5 ルール

ルールは Flex の心臓部です。ルールを書くことによって、プログラマは、スキャナが何を実行するべきであるかを Flex に通知します。

通常、ルールは2つの部分から構成されます。

```
pattern          actions
```

このうち *pattern* が何を認識するべきかを定義し、*actions* がその何かを認識した時に何を実行するべきであるかをスキャナに知らせます。*pattern* の部分は空白によって区切られます。これは、空白をマッチさせたい場合には、それを引用符で囲む必要があるということを意味しています。

スキャナは、マッチするものを2つ以上見つけた場合、以下の2つのルールを使ってどれを受け入れるかを決めます。

1. 後続コンテキスト (trailing context) も含めて最も長いものを受け入れる。
2. マッチするものがすべて同じ長さの場合、スキャナ定義中に最初に記述されたものを受け入れる。

actions は、空 (コードなし) にするか、もしくは、1 つ以上の C の文を含む単一のコード行、`{...}` または `%{...}%` で囲まれた 1 行以上のコード、単一の垂直棒 (‘|’) のいずれかを記述することができます。以下にいくつか例を挙げます。

```

hi          |
bonjour     |
hello       printf("hello!\n");
goodbye     { printf("goodbye!\n"); }
konnichiwa {
              line 1
              ...
              line n
            }
sayonara    printf("lex will not "); printf("print this\n");

```

どの行も複数の文を含むことができます。‘|’は、そのルールにマッチするものが見つかった場合、次に現れるルールのアクション部に記述されているアクションが実行されるべきであることを Flex に通知します。

注:ほとんどのバージョンの *Lex* は、‘{’と ‘}’ のペアの外部では単一の文しか許しません (例えば上の *sayonara* ルールは許されません)。また、C 以外の言語をターゲットにしている *Lex* では、‘{’と ‘}’ のペアは、例えば *Pascal* の場合の *begin...end* のように、異なるシンボルに置き換える必要があるかもしれません。

ルールにマッチしなかった入力に対するデフォルトのアクションは、それを *stdout* に出力することであり、一方、マッチしたパターンに対するデフォルトのアクションは、それを破棄することであるという点に注意してください。これは、最も単純な Flex の定義が

```
%%
```

であることを意味しています。これは、入力を変更せずそのまま *stdout* へ出力するものです。別の単純な例として以下のようなものがあります。

```
%%
foobar
```

この場合、入力の中から ‘foobar’ という文字の並びをすべて取り除き、取り除いた結果を *stdout* に出力します。

3.6 パターン・セクション

パターン・セクションは、正規表現と呼ばれる仕組みを使って実際のマッチング処理を実行します。正規表現は、文字列、文字、文字集合 (クラス)、および演算子から構成されています。正規表現を構成する要素については次節以降で説明します。また正規表現自体については、Section 3.7 [Regular Expressions], page 23 において議論します。

3.6.1 文字

いくつかの文字は Flex にとって特別の意味があり、その文字を単独で使ったのは、その文字自体を表すことができません。以下に、Flex における特殊文字とその意味を表にして示します。

文字	Flex による解釈						
.	ピリオドは改行 (‘\n’) 以外の任意の文字を表します。						
\	バックスラッシュはエスケープ文字です。エスケープ・シーケンスは ANSI C のものと同一です。						
[]	角括弧 [] は複数の文字を文字クラスにグループ化します。詳細については、See Section 3.6.3 [Flex における文字のグループ化], page 21。						
^	文字クラスの内部では ^ は否定を意味します。詳細については、See Section 3.6.3 [Flex における文字のグループ化], page 21。文字クラスの外部では、^ は行の先頭を意味し、ルール先頭にのみ置くことができます。例を以下に示します。 <table> <tr> <td>[^AB]</td><td>否定クラスです。</td></tr> <tr> <td>^foo</td><td>行の先頭にある ‘foo’ という文字の並びにのみマッチします。</td></tr> <tr> <td>foo^</td><td>この場合、‘^’ は普通の文字であるとみなされます。このような時には、希望どおりの結果が確実に得られるようにするために、特別な意味を持つ文字の前にバックスラッシュ ‘\’ を置くのが良いでしょう。このような文字の並びをエスケープ・シーケンスと呼びます。エスケープ・シーケンスについてはこの節の最後で説明します。</td></tr> </table>	[^AB]	否定クラスです。	^foo	行の先頭にある ‘foo’ という文字の並びにのみマッチします。	foo^	この場合、‘^’ は普通の文字であるとみなされます。このような時には、希望どおりの結果が確実に得られるようにするために、特別な意味を持つ文字の前にバックスラッシュ ‘\’ を置くのが良いでしょう。このような文字の並びをエスケープ・シーケンスと呼びます。エスケープ・シーケンスについてはこの節の最後で説明します。
[^AB]	否定クラスです。						
^foo	行の先頭にある ‘foo’ という文字の並びにのみマッチします。						
foo^	この場合、‘^’ は普通の文字であるとみなされます。このような時には、希望どおりの結果が確実に得られるようにするために、特別な意味を持つ文字の前にバックスラッシュ ‘\’ を置くのが良いでしょう。このような文字の並びをエスケープ・シーケンスと呼びます。エスケープ・シーケンスについてはこの節の最後で説明します。						
-	ハイフンは文字クラスの内部において文字の範囲を表します。文字クラスの外部では、ハイフンはそれ自身を表します。詳細については、See Section 3.6.3 [Flex における文字のグループ化], page 21。						
{ }	大括弧 { } は、定義の参照、複数行のアクションの囲み、またはある範囲にわたる繰り返しの定義を行います。例を挙げると、定義 F00 があって、それをルールの中で参照したい場合に {F00} を使います。 与えられたパターンのある範囲にわたる繰り返しを定義するには、以下のような ‘{ repetition list }’ を使います。						

```
%%
f{2,5} /* f の 2 回以上 5 回以下の繰り返し */
/* にマッチ */
f{2,} /* f の 2 回以上の繰り返しにマッチ */
f{2} /* f の 2 回の繰り返しにマッチ */
```

この用法の解釈において、Flex と Lex の間にはいくつかの相違点があります。詳細については、Section 8.1.1 [Flex and POSIX], page 111 を参照してください。

() 丸括弧 () を使って優先順位を変更することができます。また、定義が展開される時には、その定義は暗黙のうちに丸括弧 () で囲まれることに注意してください。² このため、Lex とは非互換なところがあります。この点については、Section 8.1.1 [Flex and POSIX], page 111 と Section 3.3 [Definitions], page 14 で説明しています。

"" 二重引用符記号は文字列を表します。引用符の内側の文字列だけがマッチの対象になります。したがって、

```
%%
"string"
```

は "string" ではなく、string にマッチします。

/ スラッシュは後続コンテキスト (trailing context) を設定します。これは、あるパターンの後ろに特定の文字の並びが続く場合のみ、そのパターンを認識したいという状況です。つまり、スラッシュ '/' は「ルック・アヘッド (その先を見る)」演算子として機能するということです。例を挙げると、

```
abcDEF      'abcDEF' を認識します。
abc/DEF      'abc' の後ろに 'DEF' が続く場合に限り、
              'abc' を認識します。'DEF' の部分は、
              あたかもまだ読まれてはいないかのように扱われ、
              マッチの対象になりません。
```

注：1 つのルールの中では '/' は 1 つだけ許されます。つまり、

```
abc/def/hijkl
```

は不正です。

< > かぎ括弧 < > はスタート状態を参照します。また、EOF シンボル ('<<EOF>>') にも使われます。完全な説明については、Section 3.8 [Start States], page 24 と Section 5.6 [End-Of-File Rules], page 83 を参照してください。

² 訳注：Flex 2.5 では、'-1' オプションを指定して生成されたスキャナは、Lex の場合と同じように、定義を展開する時に丸括弧 () で囲みません。

? + * ‘?’、‘+’、‘*’は、ある正規表現が現れることのできる回数を設定します。‘?’は 0 回もしくは 1 回（その正規表現が現れることは必須ではないということ）を意味します。‘+’は 1 回以上を意味します。‘*’は 0 回以上を意味します。例えば、

a? 0 個もしくは 1 個の a にマッチします。

a+ 1 個以上の a にマッチします。

a* 0 個以上の a にマッチします。

(ABC)+ ‘ABC’ という文字の並びが 1 回以上続くものにマッチします。

[abQrS]? 0 個もしくは 1 個の、(5 つの文字 ‘abQrS’ から構成される) この文字クラスのメンバにマッチします。文字クラスに関する詳細については、Section 3.6.3 [Flex における文字のグループ化], page 21 を参照してください。

{NUM}* 0 個以上の NUM にマッチします。ここでの NUM は定義です。定義に関する詳細については、Section 3.3 [Definitions], page 14 を参照してください。

| OR 演算子、および、特別なアクションを表します。例えば、

apples|oranges

は apples もしくは oranges のいずれかにマッチし、

```
apples      |
oranges      printf("fruit!\n");
```

は、apples と oranges の両方に対して同一のアクションを実行します。

\$ ドル記号は行末を意味します。例えば、

end\$

はその直後が行末である場合にのみ ‘end’ という文字の並びにマッチします。これは、後ろに続くのが行末のマーカである場合のみ ‘end’ にマッチする

end/\n

とまったく同じです。

こうした文字のいずれかをその文字自身として表したい場合には、引用符で囲むか、(後に示す表で説明する) エスケープ・シーケンスとして表さなければなりません。

Flex には 3 種類のエスケープ・シーケンスがあります。バックスラッシュ‘\’に続けて 8 進数を使うもの、‘\x’に続けて 16 進数を使うもの、‘\letter’という表記法によってある 1 文字、または、特別な表示不可の文字を表すものの 3 つです。C をよく知っている人であれば、この 3 つが ANSI C のエスケープ・シーケンスであることに気がつくことでしょう。数値によるエスケープ・シーケンスは、100 パーセント移植性があるわけではなく、保守を困難にするので、避けるべきです。

以下に、文字の使用に関する要約を示します。この表中では、‘c’が単一の文字を、‘NNN’が 8 進定数を、‘HH’が 16 進定数を表します。

Flex における文字

文字	意味	例
c	c が演算子でない場合は、文字 c 自体	a
"c"	c	" "
.	改行以外の任意の文字	Un.x
\b	バックスペース (BS)	\b
\t	水平タブ (HT)	\t
\n	改行 (NL)	\n
\v	垂直タブ (VT)	\v
\f	頁送り (FF)	\f
\r	キャリッジ・リターン (CR)	\r
"	単一引用符	"
\\	単一バックスラッシュ	\\
\0	NUL 文字	\0
\c	c が上記以外の文字の場合、文字 c 自体	*
\xHH	16 進数 HH を値として持つ文字	\x1B
\NNN	8 進数 NNN を値として持つ文字	\033

注：いくつかのバージョンの *Lex* では、‘\0’を正しく認識、またはマッチしません。これは、‘\0’が *NUL*、つまり *C* 文字列の終端文字だからです。*Flex* では、*NUL* をマッチの対象にしても問題はありますが、性能には若干影響します。

さらに付け加えると、‘^’演算子と‘<<EOF>>’はルールの先頭にのみ置くことができます。また、これらと‘\$’、‘/’は丸括弧 () の内部に置くことはできません。このことはまた、定義の正当性にも影響を及ぼします。というのは、展開される時に定義は字義どおりに丸括弧 () で囲まれるからです。³ 詳細については、Section 3.3 [Definitions], page 14 と Section 8.1.1 [Flex and POSIX], page 111 を参照してください。

³ 訳注：Flex 2.5 では、‘-1’オプションを指定して生成されたスキナは、Lex の場合と同じように、定義を展開する時に丸括弧 () で囲みません。

3.6.2 Flex における文字列

文字列とは、(常に、というわけではありませんが) 多くの場合、引用符によって囲まれる文字のグループです。エスケープ・シーケンスが使われない限り、文字列には改行や表示不可の文字を含めることはできません。

‘-i’ オプション(詳細については、See Section 5.1 [Case Insensitive Scanners], page 75) を使わない限り、大文字・小文字の区別も含めた字義どおりの文字列に対してマッチが行われます。引用符付きの文字列については、引用符は認識される文字列には含まれません。

例えば、

```
string
StrING
"STRING"
\"string\"
```

はすべて正当な文字列であり、最後のものは引用符も含めてマッチされます。Flex においては文字列には引用符は必須ではありません。したがって、キーワードのグループにマッチさせる場合、

```
begin
end
pointer
...
```

と

```
"begin"
"end"
"pointer"
...
```

のいずれも正当です。

3.6.3 Flex における文字のグループ化

Flex では、文字をグループ化して文字クラスにすることができます。文字クラスは、文字のグループを角括弧 [] で囲むことにより作成されます。どのような文字でも正当です (表示不可の文字についてはエスケープ・シーケンスを使います)。また、文字の範囲をハイフン ‘-’ を使って指定することができます。文字クラスがルールの中で使われている場合には、Flex はそのクラスの任意のメンバとマッチさせ、あたかも単一文字が使われているかのように振る舞います。例えば、

```
[a-z]
[A-Z]*
```

において、最初の例は ‘a’ から ‘z’ までの任意の単一文字にマッチします。第 2 の例は ‘A’ から ‘Z’ までの任意の文字が 0 個以上並んだものにマッチします。

否定文字クラスを表す正規表現を書くこともできます。否定文字クラスは、(‘\n’ も含めて) 文字クラスのメンバ以外であれば何にでもマッチします。これを行うには、

否定すべきクラスの先頭に '^' を置きます。(クラスの外部では '^' は異なる意味を持つことに注意してください。) 以下に、正当なクラスの例をいくつか挙げます。

```
[abc]      'a'、'b'、'c'のいずれかにマッチします。
[abc\n]    'a'、'b'、'c'、'\n'のいずれかにマッチします。
[a-z]      ASCII 値が 'a' から 'z' までの範囲にある任意の文字、すなわち、
            任意の英小文字にマッチします。
[^a-z]     'a' から 'z' までの範囲にある文字以外の任意の文字にマッチします。
[ABcd]     'A'、'B'、'c'、'd'のいずれかにマッチします。
```

注: *Flex*、およびいくつかのバージョンの *Lex* は、クラス内における逆方向の範囲を扱うことができません。したがって、

```
%%
[z-a9-0]
```

はエラー・メッセージを出力します。逆方向の範囲は指定しないでください。

3.6.4 Flex における文字のグループ化 (Flex 2.5 の補足情報)

Flex 2.5 では、文字クラスの中に文字クラス式を含めることができます。

文字クラス式は、形式的には、ある文字集合を識別する名前を '[' と ':' で囲んだものです。Flex 2.5 では、以下の文字クラス式が有効です。

```
[ :alnum:] [ :alpha:] [ :blank:]
[ :cntrl:] [ :digit:] [ :graph:]
[ :lower:] [ :print:] [ :punct:]
[ :space:] [ :upper:] [ :xdigit:]
```

文字クラス式 `[:XXX:]` は、ANSI C の `'isXXX()'` 関数がゼロ以外の値を返す文字の集合に対応します。唯一の例外は `[:blank:]` で、`isblank` は (POSIX では定義されているものの) ANSI C では定義されていないため、Flex では、マクロ `IS_BLANK` を

```
#define IS_BLANK(c) ((c) == ' ' || (c) == '\t')
```

のように定義して、これが真となる文字の集合 (すなわち、スペースとタブ) を文字クラス式 `[:blank:]` に対応させています。

文字クラス式を使えば、

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

を

```
[[:alpha:]]
[[:digit:]]
```

と書くことができます。

また、文字クラスの中に複数の文字クラス式を含めることができますので、

```
[a-zA-Z0-9]
```

を、

```
[[[:alpha:]][:digit:]]
```

と書くこともできます（もっとも、この例の場合は、`[[[:alnum:]]]` と書くほうが良いでしょう）。

3.7 正規表現

Flex の文字、文字列、クラス、定義、および演算子を組み合わせることで、正規表現として知られているものが作られます。（基本単位が数と演算子である）数学表現と同じように、基本的な要素は単純なもの（文字、演算子、文字列、クラス、および定義）ですが、要素を組み合わせることでより複雑な表現式を作ることができます。例えば、`'c'` は単一文字の正規表現で、`'c'` にマッチします。`'cc'` は 2 つの正規表現をつないだものを含む正規表現で、`'cc'` にマッチします。`'c*'` は、単一文字の正規表現 `'c'` と、それに続く演算子 `'*'` から構成される正規表現で、0 個以上の `c` にマッチします。正規表現の真のパワーは、個々の要素よりもむしろ、組み合わせ可能な方法の中にあります。

次の表は、Flex で利用可能な正規表現をすべて示したものです。表中において、`'c'` は（エスケープ・シーケンスを含む）任意の単一文字を、`'r'` は任意の正規表現を、`'s'` は文字列を表します。表はグループ別に編成しており、優先度の最も高いものが一番上にあります。

Flex における正規表現

正規表現	マッチの対象	例
<code>c</code>	特殊文字を除く任意の文字	<code>A</code> または <code>\n</code>
<code>.</code>	改行 (<code>\n</code>) を除く任意の文字	<code>efg.*</code>
<code>[s]</code>	クラス <code>s</code> 中にある任意の文字	<code>[efg]</code>
<code>[^s]</code>	クラス <code>s</code> 中にない任意の文字	<code>[^mnqsl]</code>
<code>r*</code>	0 個以上の <code>r</code>	<code>(a e-f)*</code>
<code>r+</code>	1 個以上の <code>r</code>	<code>(a e-f)+</code>
<code>r?</code>	0 個または 1 個の <code>r</code>	<code>(a e-f)?</code>
<code>r{x,y}</code>	<code>x</code> 個以上 <code>y</code> 個以下の <code>r</code> (<code>abc{1,3}</code> は、 <code>ab</code> と 1 個以上 3 個以下の <code>c</code>)	<code>abc{1,5}</code>
<code>"s"</code>	字義どおりの文字列 <code>s</code>	<code>"****"</code>
<code>\c</code>	(<code>\c</code> が ANSI C において特別な意味を持たない場合) <code>c</code>	<code>\</code> または <code>*</code>
<code>(r)</code>	<code>r</code> - 丸括弧 <code>()</code> はグループ化のためのもの	<code>(Ab Bb)</code>
<code>r1r2</code>	<code>r1</code> の後ろに <code>r2</code> が続くもの	<code>Aa</code>
<code>r1 r2</code>	<code>r1</code> または <code>r2</code>	<code>A B</code>
<code>r1/r2</code>	<code>r2</code> が後ろに続くという条件を満足する <code>r1</code>	<code>abc/123</code>
<code>^</code>	行頭	<code>^abc</code>
<code>\$</code>	行末	<code>abc\$</code>
<code><start>r</code>	スタート状態 (<code>start</code> 状態の時、 <code>r</code> がアクティブ)	<code><comment>"*/"</code>
<code><<EOF>></code>	ファイルの終端 (End-Of-File ルールを参照)	<code><<EOF>></code>

Unix においてパターン検索が必要な場合には正規表現がよく使われますが、アプリケーションが異なると、正規表現もよく似てはいるもののまったく同一ではないという点に注意してください。例えば、Flex、egrep、Emacs はいずれもパターン検索のテンプレートとして正規表現を使いますが、それぞれが理解する正規表現は少しずつ異なります。特に、Flex では定義が使われますが、egrep や Emacs では使われませんし、egrep や Emacs は単語の先頭と末尾にマッチさせるための '`\<`' と '`\>`' とを提供していますが、Flex は提供していません。さらに、Emacs はバッファの先頭に対するマッチングや「ファジー」なマッチング等を行うための、特別な '`\letter`' シーケンスをほかにも数多く提供しています。

3.8 スタート状態

なんらかの条件に基づいて、パターン・マッチング処理のルールを活性化することが便利な時があります。例えば、いくつかのコンピュータ言語では、重複しているスキャン・ルールの曖昧さを取り除くのを支援するために、パース状態を使います。別の例としては、ある特定の入力が見つかったあとでだけ、あるルールを活性化したいという場合があります。このような状況に対処するために、Flex はスタート条件またはスタート状態と呼ばれる単純なシステムを提供しています。

3.8.1 スタート状態の説明

スタート状態は、あるルールがアクティブになるのはいつであるかを Flex に通知するブール値のようなものです。スタート状態は、定義セクションにおいて（排他的スタート状態の場合）`%x`、または（包含的スタート状態の場合）`%s`を使って宣言されます。

```
%x start_state_name
%s start_state_name
```

`start_state_name` は一意な名前でなければならない点に注意してください。つまり、他のスタート状態や定義が同じ名前を持つてはならないということです。スタート状態は、1つの状態の名前、または、カンマで区切られた複数の状態の名前をかぎ括弧<>で囲むことによって、ルール・セクションで参照されます。スタート状態の参照はルールの先頭になければならず、1つのルール中には1対のかぎ括弧<>のみ許されます。このことは、

```
%x state1
%s state2
%x state3 state4
%%
<state1>"foo"
<state2,state3,state4>"bar"
```

が正当であり、

```
integer [-+]?[0-9]*
%x integer
%s state1,state2,state3
%%
<integer>"foo"
"bar"<state1>
<state1>"bar"<state2,state3>
```

はすべて不当であることを意味しています。`integer`については同じ名前を持つ定義が存在し、それ以外のものについてはスタート状態の参照の位置が正しくないか、複数の参照が存在するからです。

これまでのところでは、Flexが異なる2種類のスタート状態をサポートしている事実から目をそらしてきました。2つのスタート状態とは、包含的スタート状態（`%s`）と排他的スタート状態（`%x`）のことです。これら2つの相違点は、排他的スタート状態が活性化された場合は、その状態に属するルールだけが活性化されるのに対して、包含的スタート状態の場合は、その状態に属するルールとスタート状態への参照を持たないルールの両方が活性化されるという点にあります。この違いを示す例を挙げると、以下のようになります。

```
%s state1
%%
<state1>"one" printf("two");
"three"      printf("four");
```

この場合、state1状態が活性化されている場合は‘one’を‘two’に置き換え、state1状態が活性化されているか否かにかかわらず‘three’を‘four’に置き換えます。デフォルトのルールにより、その他のテキストは stdoutに出力されます。これに対して、

```
%x state1
%%
<state1>"one" printf("two");
"three"      printf("four");
```

は、state1状態が活性化されている時は‘one’を‘two’に置き換え、state1状態が活性化されていない時のみ‘three’を‘four’に置き換えます。デフォルトのルールにより、その他のテキストは stdoutに出力されます。

このことは、排他的スタート状態が使われる場合には、マッチしないテキストが stdoutに出力されてはならないのであれば、すべての可能な入力にマッチするルールを、個々の排他的スタート状態が持たなければならないことを意味しています。包含的スタート状態の場合は、あらゆる状態において有効な、スタート状態への参照を持たないルールを1つ持つ必要があります。

注：排他的スタート状態は *POSIX* の一部であるにもかかわらず、*Lex* ではサポートしていません。

3.8.2 状態の活性化

スタート状態の名前を並べただけではあまり役に立ちません。つまり、スタート状態がいつ活性化されるのかということも制御しなければなりません。活性化は、アクションの中、または、記述情報内の追加的な C コードを記述する領域の中において、BEGINを使うことで実現されます。使い方は以下のとおりです。

```
BEGIN(start_state_name);
```

例を挙げると、以下のようになります。

```
%x COMMENT
%%
"{ "          BEGIN(COMMENT);
<COMMENT>"$R"
<COMMENT>"$I"
<COMMENT>"$M"
...
<COMMENT>"}"  BEGIN(INITIAL);
```

この場合、Pascal のコメントの先頭部分を見つけると COMMENT状態に移行し、コンパイラ・オプションを認識するようになります。BEGINは最初の‘%%’の直後(最初のルールの前)において使うこともでき、この場合は *yylex()* は常に指定された状態で開始されます。

上の例においては、定義されていない INITIALという状態があることに注意してください。この状態は常に利用可能で、活性化された状態が1つも存在しない時のスキナの初期状態を表します。つまり、BEGIN(INITIAL)によって、スキナの状態

が効果的に（もちろん、その時点においてスキャンしている箇所を維持したまま）リセットされることを意味しています。

3.8.3 スタート状態に関する注

以下に、スタート状態の使用に関する注をいくつか示します。

- 特殊文字

1つのルールにおいては、単一のスタート状態、または、カンマで区切られたスタート状態のリストのみを使用することができます。また、こうしたスタート状態の指定はルールの先頭になければなりません。次に示すものは正当です。

```
%x state1
%s state2
%%
<state1> "something"
<state2> "another thing"
<state, state2> "something else"
```

しかし、次に示すものは不当です。

```
%x state1
%s state2
%%
wrong<state1>
<state1><state2>"wrong"
<state2>"wrong"<state1>
```

- 排他的スタート状態

排他的スタート状態は、他のすべての状態を「無効」にするので、強力です。これは、スキャナの内部においてもう1つのスキャナを効果的に定義することができることを意味しています。これにより例えば、スタート状態次第で、CとPascalの両方をスキャンするスキャナを定義することが、理論的には可能になります。以下のようなコードが持つ効果を想像してみてください。

```
%x PASCAL
%x C
%%
<PASCAL>begin      return(OPEN_BLOCK);
<PASCAL>end        return(CLOSE_BLOCK);
<C>{               return(OPEN_BLOCK);
<C>}               return(CLOSE_BLOCK);
```

- スタート状態の名前

前述のとおり、スタート状態はそれ自身の名前空間を持っていません。その理由は、スタート状態が#defineとほとんど同じ方法で整数値として定義されているからです。このことは、整数値と同様、スタート状態の「スタック」のようなものを作成することが可能であることを意味しています。例えば、

```

%{
    int last_state[MAX_STATES]
    int state_count = 0;
}%
%x FOO BAR baz
%%
FOO      {
            last_state[state_count] = FOO;
            state_count++;
            BEGIN(baz);
        }
BAR      {
            last_state[state_count] = BAR;
            state_count++;
            BEGIN(baz);
        }
<baz>rule 1
    ...
<baz>rule n
<baz>END  {
            statecount--;
            BEGIN(last_state[statecount]);
        }

```

は FOO と BAR の両方によって baz 状態を活性化させ、<baz>END というルールによって 1 つ前の状態に戻します。こうした「状態スタック」は将来、Flex の特徴的な機能になるかもしれません。⁴

3.8.4 スタート状態に関する注 (Flex 2.5 の補足情報)

Flex 2.5 では、以下の新機能を利用することができます。

- ワイルド・カード

スタート状態 '<*>' には特殊な意味があり、すべてのスタート状態にマッチします。

例えば、

```

%x state1
%%
<state1>"one" printf("two");
<*>"three"   printf("four");

```

⁴ 訳注: Flex 2.5 は、スタート状態スタックをサポートしています。次節 (Section 3.8.4 [Start State Notes (Flex 2.5)], page 28) を参照してください。

は、

```
%s state1
%%
<state1>"one" printf("two");
"three"      printf("four");
```

と同じ意味になります。

- カレントなスタート状態

マクロ YY_STARTにより、カレントなスタート状態を参照することができます。

前節 (Section 3.8.3 [Start State Notes], page 27) の「スタート状態の名前」に示した例では、カレントなスタート状態を配列 `last_state` に格納する処理を、以下のように記述していますが、

```
FOO      {
          last_state[state_count] = FOO;
          ...
        }
```

この代入は、

```
last_state[state_count] = YY_START;
```

のように書き換えることができます。

Lex との互換性のために、YYSTATEが、YY_STARTの別名として定義されています。

- スタート状態のスコープ

スタート状態のスコープを定義することができます。これにより、同じスタート状態において複数のルールが存在する時に、その個々のルールにスタート状態を指定する必要がなくなります。

スタート状態のスコープの形式は、以下のとおりです。

```
...    <start_states>{
        }
...    }
```

ここで、`start_states` は、単一のスタート状態、または、カンマで区切られたスタート状態のリストです。スタート状態のスコープの境界は、‘{’と‘}’によって指定されます。スタート状態のスコープを入れ子にすることも可能です。

Section 3.8.2 [Activating States], page 26 に示した例を、スタート状態のスコープを使って書き直すと、以下のようになります。

```

%x COMMENT
%%
"{ "          BEGIN (COMMENT);
<COMMENT>{
    "$R"
    "$I"
    "$M"
    ...
    "}"          BEGIN (INITIAL);
}

```

- スタート状態スタック

スキャナ定義ファイルで ‘%option stack’ を指定すると、スタート状態スタックを利用できます。スタート状態スタックを操作するために、以下の関数が提供されています。

```
void yy_push_state(int new_state)
```

カレントなスタート状態をスタート状態スタックにプッシュし、
new_state 状態に移移します。

```
void yy_pop_state()
```

スタート状態スタックからスタート状態をポップし、そのポップされたスタート状態に移移します。

```
int yy_top_state()
```

スタート状態スタックの先頭にあるスタート状態を返します (スタート状態スタックの内容は変更されません)

前節 (Section 3.8.3 [Start State Notes], page 27) の「スタート状態の名前」に示した例を、スタート状態スタックを使って書き直すと以下のようになります。

```

%option stack
%x FOO BAR baz
%%
FOO      {
                yy_push_state(baz);
        }
BAR      {
                yy_push_state(baz);
        }
<baz>rule 1
        ...
<baz>rule n
<baz>END  {
                yy_pop_state();
        }

```

3.8.5 スタート状態の使用例

プログラミングにおいて、何かをする方法を学ぶのに最良の方法は、実際にそれを行ってみることです。そのことに留意し、スタート状態をどのように使うことができるかを示す実例を以下に挙げます。

```
/*
 * dates.lex: 日付の異なる形式を識別するために
 *             スタート状態を使用する例
 */

%{
#include <ctype.h>

char month[20], dow[20], day[20], year[20];

%}

skip of|the|[ \t,]* /* この文字の並びを無視する */

mon  (mon(day)?)    /* 曜日の名前の長い形式と短い形式の */
tue  (tue(sday)?)   /* どちらにもマッチするよう設定する */
wed  (wed(nesday)?)
thu  (thu(rsdays)?)
fri  (fri(day)?)
sat  (sat(urday)?)
sun  (sun(day)?)
```

```

/* 以下はすべての可能な曜日を表す */

day_of_the_week ({mon}|{tue}|{wed}|{thu}|{fri}|{sat}|{sun})

jan  (jan(uary)?) /* すべての月について同様のことを行う */
feb  (feb(ruary)?)
mar  (mar(ch)?)
apr  (apr(il)?)
may  (may)
jun  (jun(e)?)
jul  (jul(y)?)
aug  (aug(ust)?)
sep  (sep(tember)?)
oct  (oct(ober)?)
nov  (nov(ember)?)
dec  (dec(ember)?)

/* 以下はすべての可能な月の名前を表す */

first_half  ({jan}|{feb}|{mar}|{apr}|{may}|{jun})
second_half ({jul}|{aug}|{sep}|{oct}|{nov}|{dec})
month       {first_half}|{second_half}

/*
 * 日、月、年の数値形式
 * これらは重複しているため、正しくパースするには、
 * スタート状態と日付の形式に関するある程度の知識
 * が必要であることに注意
 */

nday      [1-9] | [1-2] [0-9] | 3 [0-1]
nmonth    [1-9] | 1 [0-2]
nyear     [0-9] {1,4}

/* 年と日のための拡張子 */

year_ext  (ad|AD|bc|BC)?
day_ext   (st|nd|rd|th)?

```

```
/*
 * このプログラムの最後にあるルールを使ってすべての無関係な
 * テキストを処理するために、非排他的なスタート状態を使う。
 * こうしないと、個々のスタート状態のブロックにルールを追加
 * しなければならなくなる。規模の大きいスキャナにおいては、
 * これは実行可能な選択肢であることが多い。なぜなら、ルール
 * の追加はスキャナのスピードに影響を与えないからである。
 * ここでは、簡潔さを優先させることにする
 */

%s LONG SHORT
%s DAY MONTH /* 長い形式の日付のために追加した状態 */
%s YEAR_FIRST YEAR_LAST YFMONTH YLMONTH

%%

/*
 * 曜日は常に最初に置かれ、後ろに続く日付の修飾子として
 * 機能するものと仮定される。よって、曜日は複数の日付形式
 * の間で共用可能である
 */

<LONG>{day_of_the_week} strcpy(dow,yytext);

/*
 * 月-日-年という形式の日付を処理する
 * パース状態は
 * LONG->[月にマッチ]->DAY->LONG
 * のように遷移する
 */

<LONG>{month}          strcpy(month,yytext); BEGIN(DAY);
<DAY>{nday}{day_ext}   strcpy(day,yytext);   BEGIN(LONG);
```

```

/*
 * 日-月-年という形式の日付を処理する
 * パース状態は
 * LONG->[日にマッチ]->MONTH->LONG
 * のように遷移する
 */

<LONG>{nday}{day_ext} strcpy(day,yytext); BEGIN(MONTH);
<MONTH>{month}          strcpy(month,yytext); BEGIN(LONG);

/*
 * 日付の年の部分は最後に置かれるものと考えられる。したがって、
 * 年を見つけたらパースされた日付を表示することができる。もち
 * ろん、日付として不当なものであればゴミが出力されることになる
 */

<LONG>{nyear}{year_ext} {
    printf("Long:\n");
    printf("  DOW   : %s \n",dow);
    printf("  Day    : %s \n",day);
    printf("  Month  : %s \n",month);
    printf("  Year   : %s \n",yytext);
    strcpy(dow,"");
    strcpy(day,"");
    strcpy(month,"");
}

/*
 * 日-月-年という形式の日付を処理する
 * SHORT 状態で数値形式の日を見つけた場合は、年が日付の最後の部分
 * になると仮定する
 * パース状態は
 * SHORT->[日にマッチ]->YEAR_LAST->YLMONTH->SHORT
 * のように遷移する
 */

<SHORT>{nday}          strcpy(day,yytext); BEGIN(YEAR_LAST);
<YEAR_LAST>{nmonth}    strcpy(month,yytext);BEGIN(YLMONTH);
<YLMONTH>{nyear}       strcpy(year,yytext); BEGIN(SHORT);

```

```

/*
 * 年-月-日という形式の日付を処理する
 * SHORT 状態で数値形式の年を見つけた場合は、日が日付の最後の部分
 * になると仮定する
 * パース状態は
 * SHORT->[年にマッチ]->YEAR_FIRST->YFMONTH->SHORT
 * のように遷移する
 */

<SHORT>{nyear}      strcpy(year,yytext); BEGIN(YEAR_FIRST);
<YEAR_FIRST>{nmonth}  strcpy(month,yytext);BEGIN(YFMONTH);
<YFMONTH>{nday}      strcpy(day,yytext);  BEGIN(SHORT);

/*
 * 数値形式の日付では、年は最初になることも最後になることも可能。
 * したがって、パースしたものをいつ表示すべきかを示すのに改行を使う
 */

<SHORT>\n            {
                        printf("Short:\n");
                        printf("  Day   : %s \n",day);
                        printf("  Month : %s \n",month);
                        printf("  Year  : %s \n",year);
                        strcpy(year,"");
                        strcpy(day,"");
                        strcpy(month,"");
                        }

/*
 * 以下により、短い( 数字 )形式と長い( 英数字 )形式とを切り換える
 */

long\n      BEGIN(LONG);
short\n     BEGIN(SHORT);

```

```

/*
 * 以下のルールは、無関係なテキストを見つけて破棄する
 * ( マッチされたテキストはデフォルトでは ECHO されないが、
 *   マッチされなかったテキストは ECHO される。ピリオドは
 *   改行以外のすべての文字を見つける。改行は\nによって
 *   見つけられる )
 */

{skip}*
\n
.

```

この実例は、様々な形式の日付をスキャンし、構成単位に分割します。例えば、以下のものを正しくスキャンし、日付の個々の部分を識別します。

```

short
1989:12:23
1989:11:12
23:12:1989
11:12:1989
1989/12/23
1989/11/12
23/12/1989
11/12/1989
1989-12-23
1989-11-12
23-12-1989
11-12-1989
long
Friday the 5th of January, 1989
Friday, 5th of January, 1989
Friday, January 5th, 1989
Fri, January 5th, 1989
Fri, Jan 5th, 1989
Fri, Jan 5, 1989
FriJan 5, 1989
FriJan5, 1989
FriJan51989
Jan51989

```

ファイルの最初の部分では、月、および、日付の異なる部分に使われる数字形式を単に定義しています。この実例では、ある特定の方法でスキャン処理が進行するよう強制するために、スタート状態を使います。例えば、行の先頭で‘1989’を見つければ、それが日と月の組み合わせではなく年であり、したがって、日付の次の部分が月に違いないことが分り、スキャン処理がそのとおりに進むよう強制します。このことにより、非常に単純な状態駆動のパースを効果的に作成したことになり、日付をその

構成要素にうまく分割することができるようになります（このスキナナの内部で起こっていることをフロー・チャートに描いてみれば、このことが明瞭に見てとれるでしょう）

このマニュアル中の他の実例と同様に、この実例も

```
flex -i dates.lex
cc -o dates lex.yy.c -lfl
```

を実行することによりコンパイルすることができます。また、‘examples’サブディレクトリにおいて単に ‘make dates’を実行することにより、コンパイルすることもできます。

3.9 %option (Flex 2.5 の補足情報)

Flex 2.5 では、スキナナ定義ファイルの中で様々なオプションを指定することができます。オプションを指定するには、スキナナ定義ファイルの先頭（最初の ‘%%’ よりも前の部分）に、%option指示子を記述します。

ほとんどの%option指示子は、以下の形式で指定されます。

```
%option option_name
```

オプション *option_name* の指定を無効にするためには、オプション名の前に ‘no’ を付けます。

```
%option nooption_name
```

以下に、コマンドライン・オプションと同等の効果を持つ%option指示子を示します。各コマンドライン・オプションの意味については、Section 2.1 [Command Line Switches], page 9 と Section 2.2 [Command Line Switches (Flex 2.5)], page 11 を参照してください。

```
%option 7bit
    -7 オプション

%option 8bit
    -8 オプション

%option align
    -Ca オプション

%option backup
    -b オプション

%option batch
    -B オプション

%option c++
    -+ オプション

%option caseful
    -i オプションの否定
```

```
%option case-sensitive
        -i オプションの否定

%option case-insensitive
        -i オプション

%option caseless
        -i オプション

%option debug
        -d オプション

%option default
        -s オプションの否定

%option ecs
        -Ce オプション

%option fast
        -F オプション

%option full
        -f オプション

%option interactive
        -I オプション

%option lex-compat
        -l オプション

%option meta-ecs
        -Cm オプション

%option output="file"
        -ofile オプション

%option perf-report
        -p オプション

%option prefix="prefix"
        -Pprefix オプション

%option read
        -Cr オプション

%option stdout
        -t オプション

%option verbose
        -v オプション

%option warn
        -w オプションの否定
```

次に、コマンドライン・オプションでは代替できない%option指示子を示します。

%option array

yytextを charの配列として定義します。これは、‘%array’と同じです。

%option always-interactive

入力を常に対話的に扱うスキャナを生成するよう指示します。これと‘%option never-interactive’のどちらも指定されない場合、生成されたスキャナは、ファイルをオープンするたびに isatty()を呼び出して、入力を対話的に(1文字ずつ)読み込むべきか否かを決定します。

%option main

生成されるスキャナに、以下のような main()関数を組み込むよう指示します。

```
int main()
{
    yylex();
    return 0;
}
```

これは、暗黙のうちに ‘%option noyywrap’を指定します。

%option never-interactive

入力を常に対話的に扱わないスキャナを生成するよう指示します。これと ‘%option always-interactive’のどちらも指定されない場合、生成されたスキャナは、ファイルをオープンするたびに isatty()を呼び出して、入力を対話的に(1文字ずつ)読み込むべきか否かを決定します。

%option pointer

yytextを charに対するポインタとして定義します。これは、‘%pointer’と同じです。

%option reject

スキャナ定義ファイルの中で REJECTが使われていることを、Flex に通知します。Flex は通常、定義ファイルの中で REJECTが使われているか否かを自分で調査しますが、この%option指示子の指定は、Flex 自身による判定結果に優先します。

%option stack

スタート状態スタック(Section 3.8.4 [Start State Notes (Flex 2.5)], page 28 を参照)を使用するためには、この%option指示子を指定しなければなりません。

%option stdinit

yyinを stdinで、yyoutを stdoutで、それぞれ初期化します。この%option指示子が指定されない場合、あるいは、‘%option

`nostdinit`が指定された場合、`yyin`と`yyout`は、`(FILE *)0`(`NULL`ポインタ)で初期化されます。

`%option unput`

‘`%option noinput`’が指定されると、生成されるスキナの中に、関数 `unput()` が組み込まれません。⁵

`%option yy_pop_state`

‘`%option noyy_pop_state`’が指定されると、生成されるスキナの中に、関数 `yy_pop_state()` が組み込まれません。ただし、‘`%option stack`’が指定されていない場合は、‘`%option noyy_pop_state`’の指定の有無にかかわらず、関数 `yy_pop_state()` は組み込まれません。

`%option yy_push_state`

‘`%option noyy_push_state`’が指定されると、生成されるスキナの中に、関数 `yy_push_state()` が組み込まれません。ただし、‘`%option stack`’が指定されていない場合は、‘`%option noyy_push_state`’の指定の有無にかかわらず、関数 `yy_push_state()` は組み込まれません。

`%option yy_scan_buffer`

‘`%option noyy_scan_buffer`’が指定されると、生成されるスキナの中に、関数 `yy_scan_buffer()` が組み込まれません。

`%option yy_scan_bytes`

‘`%option noyy_scan_bytes`’が指定されると、生成されるスキナの中に、関数 `yy_scan_bytes()` が組み込まれません。

`%option yy_scan_string`

‘`%option noyy_scan_string`’が指定されると、生成されるスキナの中に、関数 `yy_scan_string()` が組み込まれません。

`%option yy_top_state`

‘`%option noyy_top_state`’が指定されると、生成されるスキナの中に、関数 `yy_top_state()` が組み込まれません。ただし、‘`%option stack`’が指定されていない場合は、‘`%option noyy_top_state`’の指定の有無にかかわらず、関数 `yy_top_state()` は組み込まれません。

`%option yyclass="classname"`

これは、‘`-+`’オプションが指定されている場合(すなわち、C++スキナを生成する場合)のみ有効です。これにより、`classname`により指定される名前のクラスが、`yyFlexLexer`のサブクラスとして定義されます。実際にスキャン処理を実行するコードは、クラス `classname` のメンバ関数 `yylex()` (`classname::yylex()`) に実装されます。詳細については、Section 4.6 [Flex and C++ (Flex 2.5)], page 71 を参照してください。

⁵ Flex 2.5.4 に付属のドキュメント ‘`flex.texi`’には、関数 `input()` についても同様のことが記載されていますが、実際に ‘`%option noinput`’を指定してみると、生成されるスキナの中に、関数 `input()` が組み込まれます。

%option yylineno

入力の行数をカウントして大域変数 `yylineno` に保持するスキャナを生成するよう指示します。

%option yymore

スキャナ定義ファイルの中で `yymore()` が使われていることを、Flex に通知します。Flex は通常、定義ファイルの中で `yymore()` が使われているか否かを自分で調査しますが、この `%option` 指示子の指定は、Flex 自身による判定結果に優先します。

%option yywrap

`'%option noyywrap'` が指定されると、`yywrap()` はマクロとして、

```
#define yywrap() 1
```

のように定義されます。この結果、ファイルの終端を検出した時に、スキャナは、ほかにスキャンすべきファイルは存在しないと判断するようになります。

4 Flex とのインターフェイス

この章では *C* および *Bison* と一緒に *Flex* を使う方法を説明します。¹ *C*、*Bison* のそれぞれが非常に多くの細目を含むため、本章は 2 つの部分に分割されています。その両方に、全般的なインターフェイス概念に関する節と実例を示す節があります。

4.1 Flex と C

Flex に対する C の主要なインターフェイスは、以下に挙げるルーチンと変数によるものです。以下の節を読む際には、いくつかの細かな部分で Flex と Lex との間に相違点があるということを意識しておいてください。Lex が提供していない関数がいくつかありますし、宣言の内容が違うものもあります。こうした相違点は、通常大きな問題にはなりません。というのは、相違のある関数は一般的にはあまり使われていないからです。相違点に関する詳細については、Chapter 8 [Flex and Lex], page 111 および Section 8.1.1 [Flex and POSIX], page 111 を参照してください。

関数	説明と実例
yylex()	yylex() は実際のスキャン処理を行う関数です。ファイル (デフォルトは stdin) を読み込み、パターン・マッチングを行い、パターンに関連付けされたアクションを実行します。デフォルトでは、入力の終端に達するまでマッチングを行い、終端に達したところでゼロを返します。(return を使って、呼び出し側のプログラムにほかの値を返すことは可能です。これは、Section 4.4 [Flex and Bison], page 61 で説明しています。) したがって、インターフェイスを提供する簡単な方法の 1 つは、オプションの C コード領域の 1 つに以下のようなコードを追加することです。

```
#include <stdio.h>

int main(argc, argv)
int argc;
char *argv;
{
    yylex();
}
```

しかしこのような場合には、Flex ライブラリ (‘-lfl’) もしくは Lex ライブラリ (‘-ll’) のいずれかをリンクして、そこからこれと同じような main() 関数を取り込むことができます。この場合は、スキャナは単にファイルをスキャンして、ルールに関連付けされたアクションを実行するだけであるという点に注意してください。

¹ 訳注：この章の最後で、C++ の使い方についても説明します。

`yylex()`の使い方としてもう1つよく見られるのが、マッチされたものが何であることを示す値を返させることです。これは、アクションに `return` 文を追加することで行われます。`return` 文を見つければ、`yylex()`は指定された値を返します。これが、Bison によるパーサが Flex によるスキャナから情報を獲得する方法です。

ルールの中に、マッチされたテキストが何を表しているかを示すコードを返す `return` 文があれば、以下のようなインターフェイスを使うことができます。

```
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv;
{
    int return_code;
    while((return_code = yylex()) != 0){
        switch(return_code){
            case KEYWORD1:
                /* 何かを行う */
                break;
            case KEYWORD2:
                /* 何か別のことを行う */
                break;
            ...
            case KEYWORDn:
            }
        }
    }
}
```

`yylex`のデフォルトの定義は `'int yylex(void)'` ですが、これは `YY_DECL` マクロを使うことによって変更することができます。例を示すと、以下のコードは `yylex()` の名前を `gettok()` に、型を `char` に対するポインタ型に変更し、パラメータ `buffer` を受け取るように指定します。

```
#undef YY_DECL
#define YY_DECL char *gettok(char *buffer)
```

注：ANSI 対応でない *C* を使っている場合は以下のように定義しなければなりません。

```
#define YY_DECL char *gettok(buffer) \
    char *buffer;
```

言葉を変えると、再宣言はターゲットとなる *C* コンパイラにとって正当な関数宣言でなければなりません。さらに、この再宣言は、

ファイルの先頭にあるオプションの C コード領域になければならないという点に注意してください。

yyin yyinは、yylex()が文字を読み込む元となるファイルです。デフォルトはstdinですが、fopen()を使って変更することができます。yyinを読み込むデフォルトの方法は、複数文字から成るブロックを一度に読むというものです。これは、YY_INPUTマクロによって変更できます。YY_INPUTマクロは、ファイルではなく文字列をスキャンするためのスキナを生成するのに便利です。YY_INPUTを定義する方法は以下のとおりです。

```
YY_INPUT(buffer,result,max_size)
```

ここで、bufferは入力バッファ、resultは読み込まれた文字数がセットされる変数、max_sizeはbufferのサイズです。以下に、一度に1文字ずつ読み込むという入力方法に変更する再定義の例を示します。この方法を使うとかなり遅くなるので、お勧めはできません。

```
#undef YY_INPUT
#define YY_INPUT(buffer,result,max_size) \
{ \
    buffer[0] = getchar(); \
    if(buffer[0] == EOF) \
        result = YY_NULL; \
    else \
        result = 1; \
}
```

注：この再宣言は、ファイルの先頭にあるオプションの C コード領域になければなりません。

yyout yyoutはスキナがECHOの出力を書き込むファイルです。デフォルトはstdoutですが、これもfopen()を呼び出すことで変更できます。

yytext yytextは最後にマッチされた文字列、つまり最後に認識されたトークンを含む大域変数です。yytextの正しい外部定義は、Lexの場合のcharの配列とは異なり、charに対するポインタ型である点に注意してください。² つまり、yytextは

```
extern char yytext[];
```

ではなく、常に

² 訳注：Flex 2.5 では、‘%pointer’と‘%array’により、yytextの型を選択できるようになりました。‘%pointer’を指定した場合はchar *yytext、‘%array’を指定した場合はchar yytext[YYLMAX]となります。デフォルトは‘%pointer’です。‘%array’を指定した場合の配列のサイズは、YYLMAXを再定義することによって変更可能です。

```
extern char *yytext;
```

のように宣言されなければならないということです。

このようになっている理由は性能です。yytextが配列であると、スキャナ中でそれを操作するコードは、コピー処理をたくさん行う必要があります。これに対して yytextがポインタである場合には、このようなことは必要ありません。

通常は、yytextは変更すべきではありません。yytextの内容が変更される必要がある場合には、代わりのバッファが使われるべきです。(‘examples’サブディレクトリの yymore2.lex ファイルでは、yytextを直接操作する技法が実演されています。ただし、このようなやり方はお勧めできません。)

yylen yylenは、最後に認識されたトークンの長さを保持する大域変数です。

yywrap() yywrapは、yyinの終端に達した時に呼び出される関数です。この関数が TRUE(ゼロ以外)を返すとスキャナは終了し、FALSE(ゼロ)を返すと、yyinが次の入力ファイルを指すように設定されたものと仮定して、スキャン処理が続行されます。

現在のところ yywrap()は、常に 1 を返すよう定義されているマクロです。このため、再定義するには、まず最初に #undef で定義解除しなければなりません。Lex では、yywrap()は関数です。Flex も将来のある時点で、これを関数として定義することになるでしょう。³

yymore() yymore() は、次に認識されるトークンで yytextの内容を更新するのではなく、その時点の yytextの内容の後ろにそのトークンを追加するよう Flex に通知する関数です。したがって、以下の例に対して ‘foobar’ という文字の並びを入力として与えると、stdoutに ‘foofoobar’ という文字の並びが書き込まれます。

```
%%
foo    ECHO; yymore();
bar    ECHO;
```

これは、まず foo ルールによって ‘foo’ という文字の並びが認識されて ECHO され、次に ‘bar’ という文字の並びが認識されて yytextの内容の後ろに追加された後に、‘foobar’ という文字の並びが ECHO されるからです。

もう少し現実的な例を取り上げましょう。以下のコードは複数行の文字列を処理するのに yymore() を使っています。

³ 訳注：Flex 2.5 では、‘%option noyywrap’が指定されない限り、yywrap()は関数です。再定義をするのに、#undefで定義解除する必要はありません。

```

/*
 * yymore.lex: yymore() を有効に使う例
 */

%{
#include <memory.h>
void yyerror(char *message)
{
    printf("Error: %s\n",message);
}

%}

%x STRING

%%
\"    BEGIN(STRING);

<STRING>[^\n"]* yymore();
<STRING><<EOF>> {
    yyerror("EOF in string.");
    BEGIN(INITIAL);
}
<STRING>\n {
    yyerror("Unterminated string.");
    BEGIN(INITIAL);
}
<STRING>\\n yymore(); /* 複数行にわたる
                        * 文字列を処理する
                        */

<STRING>\\" {
    yytext[yytext-1] = '\\0';
    printf("string = \"%s\"",yytext);
    BEGIN(INITIAL);
}

%%

```

この例では、エスケープ・シーケンスの変換がまったく行われていないので、文字列に対してさらに処理が必要である点に注意してください。この例は、Section 9.2 [文字列リテラルの処理], page 116 において、エスケープ・シーケンスを処理する、より役に立つ形式に拡張されます。

`yylless(n)`

`yylless()`は、`yymore()`とほぼ反対のを行うものです。この関数は、最初の n 文字以外のすべてを戻します。戻された文字の並びは、次のトークンをマッチするのに使われ、`yyleng`と `ytext`には、この変化を反映した値が設定されます。引数 n にゼロを指定して `yylless()` を呼び出すと、全入力データが戻され、スキャナは (`BEGIN`、またはそれに類似のものでデフォルトの動作が変更されない限り) 無限ループに入ります。例えば、次のコードに `'foobar'` という文字の並びを入力として与えると、`'foobarbar'` という文字の並びが出力されます。

```
%%
foobar      ECHO; yylless(3);
[a-z]+      ECHO;
```

これは、`'foobar'` が認識され `ECHO` された後に、`'bar'` が戻されるからです。となると、次にマッチするのは (`'[a-z]+'` というルールでマッチされる) `'bar'` だけで、これが次に `ECHO` されることになります。

`input()`

`input()` は、`yyin` から次の文字を取って返す関数です。これは、標準的な Flex ルール・システムを使ったのではうまく扱えないケースを処理するのによく使われます。例えば、ほとんどの言語におけるコメントは、これを使って処理することができます。これを使う理由は、

```
%%
"/*".*"/"
```

が、ピリオドが改行以外の任意の文字にマッチしてしまうために複数行にわたるコメントをうまく処理できず、また、

```
%%
"/*"[.\n]*"/"
```

は、文字クラスが任意の文字にマッチしてしまうために、バッファをオーバーフローさせるか、さもなければファイルの内容をすべて読み込んでしまうからです。(実際には、排他的スタート状態を使うことで、こうしたことを非常にエレガントな方法で処理することができます。実例については、Chapter 9 [役に立つコードの抜粋], page 115 を参照してください。しかし、POSIX によりサポートされているにもかかわらず、ここで必要になるいくつかの機能を Lex が提供していないために、この方法には移植性がありません。) C のコメントは以下のようにして移植性のある方法で処理することができます。

```

%%
"/*" {
    int a,b;

    a = input();
    while(a != EOF){
        b = input();
        if(a == '*' && b == '/'){
            break;
        }else{
            a = b;
        }
    }
    if(a == EOF){
        error_message("EOF in comment");
    }
}

```

注: スキャナが C++ コンパイラを使ってコンパイルされる場合は、この関数 `input` は `yyinput` という名前になります。これは、`input` という名前が同一名の C++ ストリームと衝突するからです。また、*Flex* では `input()` は `yytext` の内容を破壊しますが、*Lex* では `yytext` は変更されずそのまま残ります。これは将来のリリースで修正される予定です。

`unput(c)` `unput()` は、文字 `c` が次にスキャンされる文字になるように、文字 `c` を入力ストリームに置く関数です。例えば、

```

%%
foo unput('b');

```

は `'foo'` を `'b'` で置き換えます。これは、`'foo'` にマッチして `'b'` を戻し、この `'b'` が次にスキャンされる文字になるからです。デフォルトのルールにより、`'b'` は `stdout` に書き込まれます。

1 つの文字が次にスキャンされる文字になるということには 1 つ微妙な点があって、それは、文字列を入力ストリームに置きたい場合には、逆順に行わなければならないということです。以下に例を示します。

```

foobar {
    char *baz = "baz";
    int i = strlen(baz)-1;

    while(i >= 0){
        unput(baz[i]);
        i--;
    }
}

```

これは、‘foobar’がマッチされた時に、入力ストリームに ‘baz’を置きます。以下は、してはならないことを示す例です。

```

/*
 * unput.l : unput() を使って行ってはならない
 *           処理の例
 */

%{
#include <stdio.h>

void putback_yytext(void);
%}

%%
foobar    putback_yytext();
raboof    putback_yytext();
%%

void putback_yytext(void)
{
    int i;
    int l = strlen(yytext);
    char buffer[YY_BUF_SIZE];

    strcpy(buffer,yytext);
    printf("Got: %s\n",yytext);
    for(i=0; i<l; i++){
        unput(buffer[i]);
    }
}

```

この例に ‘foobar’を入力として与えると、まず ‘foobar’にマッチし、次に ‘raboof’にマッチする無限ループに陥ります。

注: `input()`と同様に `unput()`も `yytext`の内容を破壊します。⁴
つまり、`yytext`から文字情報を返したい場合には、上の例に示されるように、まず `yytext`の内容をコピーしなければならないことを意味しています。

`yyterminate()`

アクションの中で呼び出されると、`yyterminate()`はスキナの実行を終了させ、その後に `yylex()`が 0 を返します。この後は、`yyrestart()`(下記参照)が呼び出されない限り、`yylex()`を呼び出してもすぐに復帰してしまいます。

`yyrestart(file)`

`yyrestart()`は、スキナの実行を再開するよう Flex に通知する関数です。これは引数を 1 つだけ、すなわち、スキナの対象となるファイル(通常は `yyin`)を取ります。これは、EOF を処理するために使うこともできますし、また、Flex に割り込みをかけ、その後に再開させることができるようにするために使うこともできます。(Flex スキナは再入可能ではないので、このようなことが必要になります。)

`YY_NEW_FILE`

`yyin`が新しいファイルを指すよう変更され、処理が継続されるべきであるということを Flex に通知するマクロです。⁵ 以下に例を示します。

```
/*
 * cat.lex: YY_NEW_FILE の実演
 */

%{
#include <stdio.h>

#define ERRORMESS "Unable to open %s\n"

char **names = NULL;
int current = 1;
%}
```

⁴ 訳注: Flex 2.5 では、`%array`が指定された場合は、`unput()`は `yytext`の内容を破壊しません。

⁵ 訳注: Flex 2.5 では、`yyin`を変更した後に `YY_NEW_FILE`を実行する必要はなくなりました。

```

%%
<<EOF>> {
    current += 1;
    if(names[current] != NULL){
        yyin = fopen(names[current], "r");
        if(yyin == NULL){
            fprintf(stderr, ERRORMESS,
                    names[current]);
            yyterminate();
        }
        YY_NEW_FILE;
    } else {
        yyterminate();
    }
}

%%

int main(int argc, char **argv)
{
    if(argc < 2){
        fprintf(stderr, "Usage: cat files....\n");
        exit(1);
    }
    names = argv;

    yyin = fopen(names[current], "r");
    if(yyin == NULL){
        fprintf(stderr, ERRORMESS, names[current]);
        yyterminate();
    }
    yylex();
}

```

ECHO yytextの内容を yyoutに書き込むマクロです。

REJECT REJECTは、その時点においてマッチしているものを受け入れず、次に最もよくマッチするものを受け入れるようスキナに通知するマクロです。スキナはマッチするものの中で最長のものを探し、マッチするものが2つあってその長さが同じ場合は、記述ファイルにおいて最初に定義されている方を選択します。つまり、認識されるテキストの長さは、同一の長さになることもあり、または短くなることもあるということを意味しています。REJECTを使った後は、yytextとyylengは新しい値を取ります。REJECTに関して知っておくべき重要な点が2つあります。1つめは、REJECTは分岐命令

であり、決して戻ってこないのです。REJECTの後ろに記述されたアクションは実行されないということです。2 つめは、REJECTとファスト・テーブル (fast table / ‘-F’) は一緒に使うことはできないということです。以下に簡単な例を示します。

```
/*
 * reject.lex: REJECT と unput() を悪用する実例
 */

%%
UNIX    {
            unput('U'); unput('N');
            unput('G'); unput('\0');
            REJECT;
        }
GNU     printf("GNU is Not Unix!\n");
%%
```

この例は、新式のテキスト代替の技法を示しています。‘UNIX’にマッチするものが見つかったら、unput()によって‘GNU’という文字の並びが戻され、その時点におけるスキャン・バッファの内容が上書きされます。次に REJECTにより分岐が行われ、別のものにマッチするようスキャナに対して通知が行われます。‘GNU’がバッファに書き込まれたので、これが次にマッチされ、そのアクションが実行されます。以下に、その結果こうなるであろうと思われる例を示します。

```
UNIX return
GNU is Not Unix!
```

実際のところは、Flex において REJECTの用途はほんの少ししかありません。上記以外では、重複するパターンや状態の変更に使うことができます。例を示すと、以下のようになります。

```
nday      [1-9] | [1-2] [0-9] | 3 [0-1]
nmonth    [1-9] | 1 [0-2]
nyear     [0-9] {1,4}

%x DAY MONTH YEAR
```

```

%%

{nday}          BEGIN(DAY);   REJECT;
<DAY>{nday}
...
{nmonth}        BEGIN(MONTH); REJECT;
<MONTH>{nday}
...
{nyear}         BEGIN(YEAR);  REJECT;
<YEAR>{nday}
...

```

この例では、日付の形式は重複しており、最初に認識された構成要素によって、どのように日付をパースするかを決定します。しかし、この例は少々不自然な感じがします。というのは、少し考えれば、REJECTを使わずに、より効率的なスキャナにすることができるからです。これは、Section 3.8.5 [スタート状態の使用例], page 31 に示しています。

BEGIN BEGINは、スキャナをある特定のスタート状態にするためのマクロです。BEGINに続く名前はスタート状態の名前です。例えば、

```

%x FLOAT
%%
floats    BEGIN(FLOAT)
<FLOAT>some_rule some_action
...

```

は、‘floats’という単語がマッチした時に、スタート状態を FLOAT に設定します (詳細については、see Section 3.8.1 [Start States Explained], page 25)

YY_USER_ACTION

YY_USER_ACTIONは、ルール・セクション中のどのアクションよりも前に実行されるアクションを定義するマクロです。これは、以下の例で示すように、yytextの内容の小文字から大文字への変換等を行うのに役に立ちます。

```

/*
 * user_act.lex: YY_USER_ACTION を使う
 *               ユーザ・アクションの例
 */

%{

#include <ctype.h>

void user_action(void);

#define YY_USER_ACTION user_action();

%}

%%

.*      ECHO;
\n      ECHO;

%%

/*
 * このユーザ・アクションはすべての文字を
 * 単に大文字に変換する
 */

void user_action(void)
{
    int loop;

    for(loop=0; loop<yyleng; loop++){
        if(islower(yytext[loop])){
            yytext[loop] = toupper(yytext[loop]);
        }
    }
}

```

これは、すべての入力文字を単に大文字に変換して ECHO します。
YY_USER_ACTION のデフォルトの設定では、何も実行されません。

YY_USER_INIT

YY_USER_INIT は、スキャン処理が開始される前に実行されるアクションを定義するマクロです。基本的には、main() 関数の中で、

`yylex()`を呼び出す文の前に同様のコードを記述するのと同じことです。以下に簡単な例を示します。

```
/*
 * userinit.lex: YY_USER_INITを使う例
 */

%{
#define YY_USER_INIT open_input_file()

extern FILE *yyin;

void open_input_file(void)
{
    char *file_name,buffer[1024];

    yyin      = NULL;

    while(yyin == NULL){
        printf("Input file: ");
        file_name = fgets(buffer,1024,stdin);
        if(file_name){
            file_name[strlen(file_name)-1] = '\0';
            yyin = fopen(file_name,"r");
            if(yyin == NULL){
                printf("Unable to open \"%s\"\n",
                    file_name);
            }
        } else {
            printf("stdin\n");
            yyin = stdin;
            break;
        }
    }
}

%}
%%
```

これは、ファイルがオープンされるか EOF が検出されるまで、入力ファイル名を入力するようユーザに催促します。EOF が検出された場合は、入力元はデフォルトで `stdin` になります。これは以下と同じことです。

```

/*
 * この例は、前の例と同じことを YY_USER_INIT を
 * 使わずに行う
 */

%{
void open_input_file(void)
{
    char *file_name,buffer[1024];

    yyin      = NULL;

    while(yyin == NULL){
        printf("Input file: ");
        file_name = fgets(buffer,1024,stdin);
        if(file_name){
            file_name[strlen(file_name)-1] = '\0';
            yyin = fopen(file_name,"r");
            if(yyin == NULL){
                printf("Unable to open \"%s\"\n",
                    file_name);
            }
        } else {
            printf("stdin\n");
            yyin = stdin;
            break;
        }
    }
}

}%
%%
%%

int main(int argc, char *argv[])
{
    open_input_file();
    yylex();
}

```

YY_BREAK YY_BREAKはマクロです。インターフェイス的な機能というよりも、むしろ生成されるコードを変更するために使うことができるものです。

スキャナ中において、すべてのアクションは1つの大きな switch 文の要素であり、デフォルトで C の `break;` 文に置き換えられる `YY_BREAK` によって区切られます。ルールのアクション部が多く の `return` 文を含んでいる場合、コンパイラが `'statement not reached'` というエラー・メッセージをたくさん出力するかもしれません。 `YY_BREAK` を再定義することによって、この警告メッセージの出力を止めることが可能です。再定義は、セミ・コロンを含む 正当な C の文でなければなりません。

注： `YY_BREAK` を再定義して空にするのであれば、アクションの最後は必ず `return;` か `break;` になるようにしてください。

4.2 Flex と C (Flex 2.5 の補足情報)

Flex 2.5 では、前節 (Section 4.1 [Flex and C], page 43) で説明されていない、以下のマクロもサポートされています。

`yy_set_interactive(is_interactive)`

カレント・バッファを、対話的なものと見なすか、非対話的なものと見なすかを制御します。引数 `is_interactive` にゼロ以外の値を渡すと、カレント・バッファは対話的なものと見なされ、ゼロを渡すと、非対話的なものと見なされます。 `yy_set_interactive()` による指定は、 `'%option always-interactive'` や `'%option never-interactive'` による指定に優先します。このマクロは、バッファからのスキャン処理が始まるよりも前に呼び出されなければなりません。

`yy_set_bol(at_bol)`

バッファは、様々なコンテキスト情報を保持しています。例えば、行頭を表す `^` を含むルールが適用されるのは、バッファ内のカレントな位置が実際に行の先頭である場合だけですが、カレントな位置が行の先頭にあるか否かという情報は、バッファのコンテキスト情報として保持されています。

マクロ `yy_set_bol()` は、バッファ内のカレントな位置が行の先頭にあるか否かを表すコンテキスト情報を設定します。引数にゼロ以外の値を渡すと、バッファ内のカレントな位置は行の先頭である、というコンテキスト情報がセットされます。したがって、次にトークンのマッチ処理が行われる時には、行頭を表す `^` を含むルールの適用が試みられます。逆に、引数にゼロを渡すと、バッファ内のカレントな位置は行の先頭ではないことになり、次にトークンのマッチ処理が行われる時には、行頭を表す `^` を含むルールの適用が試みられなくなります。

`YY_AT_BOL()`

次にトークンのマッチ処理が行われる時に、行頭を表す `^` を含むルールの適用が試みられるようなコンテキスト情報がセットされ

ている場合には、ゼロ以外の値を返します。それ以外の場合は、ゼロを返します。

4.3 Flex と C の簡単な実例

ある単語が現れた時に、それを別の単語に置き換える必要の生じることがよくあります。例えば、ある名前が現れるたびに、それをある 1 つの環境変数の値で置き換えてくれるユーティリティを作りたいとしましょう。そして、以下のようなことができるように、そのユーティリティがフィルタとして動作するようにさせたいとします。

```
nick% myname    < infile | more
nick% myname    < infile > outfile
```

以下に、こうしたことを実現する方法を示す Flex ファイルの簡単な例を挙げます。

```
/*
 * myname.lex : トークンの置き換えを行う Flex プログラム
 *              のサンプル
 */
```

```
%%
```

```
%NAME      { printf("%s",getenv("LOGNAME")); }
%HOST       { printf("%s",getenv("HOST"));    }
%HOSTTYPE   { printf("%s",getenv("HOSTTYPE")); }
%HOME       { printf("%s",getenv("HOME"));    }
```

```
%%
```

このソース・ファイルは 'examples' サブディレクトリにあり、その名前は 'myname.lex' です。これをビルドするには、'examples' サブディレクトリに移動して 'make myname' を実行するか、以下を実行します。

```
flex myname.lex
cc lex.yy.c -o myname -lfl
```

ここで '-lfl' は、リンカに対して Flex ライブラリをリンクするよう通知します。現在のところ、Flex ライブラリにはデフォルトの main() 関数だけが含まれています。将来のバージョンの Flex では、他の関数も含まれるようになるでしょう。Flex ライブラリがインストールされていない場合は、この部分は '-ll' でなければなりません。

いずれの場合でも、最終的には 'myname' という名前の実行ファイルが生成されるはずです。これは、以下のような変換処理を実行するフィルタです。

```
%NAME      ユーザのログイン名に置き換えられます。
%HOST       ユーザのホスト・コンピュータ名に置き換えられます。
%HOSTTYPE   ユーザのホスト・コンピュータのマシン・タイプに置き換えられます。
```

%HOME ユーザのホーム・ディレクトリを表すパスに置き換えられます。

したがって、以下のような内容を持つファイル 'myname.txt' を作成して、

```
Hello, my name is %NAME.  Actually
"%NAME" isn't my real name, it is the
alias I use when I'm on %HOST, which
is the %HOSTTYPE I use.  My HOME
directory is %HOME.
```

以下を実行すると、

```
myname < myname.txt
```

以下のテキストに似たものが stdout へ書き込まれます。

```
Hello, my name is foobar.  Actually
"foobar" isn't my real name, it is the
alias I use when I'm on baz, which
is the cray I use.  My HOME
directory is /home/foo/foobar.
```

このプログラムがうまく動作するのは、yyin と yyout がデフォルトでは stdin、stdout にそれぞれ割り当てられ、かつ、デフォルトのアクションが yyin の内容を yyout にコピーするからです。また、個々のルールに対応する唯一のアクションが単一行で記述されているため、'{' は必要ではないことに注意してください。このような場合には、アクションを '{' で囲むか否かは、個人的な好みの問題になります。

これが、引用符で囲まれた部分にあるものも含めて、指定された名前が現れるところすべてにマッチしたことに気がつきましたか？ Flex においては、引用符で囲まれた部分にあるものにマッチさせたくない場合には、それに対応するルールを作成することにより、そうしないよう明示的に Flex に通知しなければなりません。以下に例を示します。

```
/*
 * myname2.lex : トークンの置き換えを行う Flex プログラムの例
 */

%{
#include <stdio.h>
%}

%x STRING
%%
\"          ECHO; BEGIN(STRING);
<STRING>[^\\"\\n]* ECHO;
<STRING>\"  ECHO; BEGIN(INITIAL);
```



```
%NAME      { printf("%s",getenv("LOGNAME")); }
%HOST       { printf("%s",getenv("HOST"));    }
%HOSTTYPE   { printf("%s",getenv("HOSTTYPE")); }
%HOME       { printf("%s",getenv("HOME"));    }
```

この例では、排他的スタート状態を使って、文字列中のテキストが変更されることのないようにしています。この例も 'examples' サブディレクトリにあるもので、その名前は 'myname2.lex' です。

4.4 Flex と Bison

Bison は、Flex と同様、ある記述情報を受け取って、それをもとに C のコードを生成するプログラムです。両者の違いは、Bison が C や Pascal のような言語の文法に関する記述情報を入力として受け取り、その記述情報からパーサを生成する点にあります。Flex と Bison を結合することにより、言語の字句解析と構文解析の両方を処理することができるようになります。(これは、コンパイラ・デザインにおいて最も容易に自動化できる部分です。)

生成されるパーサが機能するためには、Bison は `yylex()` という関数が必要とします。この関数はユーザによって提供され、呼び出された時に、パースされている言語のある要素を表す整数値を Bison に返します。Flex においてスキャン処理を行うルーチンは `yylex()` であり、デフォルトでは整数値を返します。これにより、Flex と Bison を一緒に使うのは非常に簡単になります。

警告： 以下の節では、読者が *Bison* の基本的なパーサの宣言を理解しているものと仮定します。*Bison* を使った経験のない人には、パーサの定義は混乱をもたらす可能性がありますので、先に進む前に是非 *Bison* のマニュアルを読んでください。*Bison* に興味のない人は、この節全体を飛ばしても構いません。

4.4.1 Flex と Bison のインターフェイス

Flex と Bison の間で情報を渡す基本的な方法は、関数 '`yylex()`' を使うことです。これは、Flex により生成されるスキャナにおいて、スキャン処理を実行する関数の名前です。Flex の入力ファイルのアクション部分において '`return`' 文を使うことによって、単なる 0 や 1 以外の値を返すことができます。この方法で、`yylex()` は最後に認識されたトークンを表す整数値を返すことができます。

Bison を '`-d`' オプション付きで使うと、Bison は '`.tab.h`' という拡張子を持つファイルを生成します。このファイルには、記述情報中にある正当なトークンの 1 つ 1 つに対する一意な定義情報が含まれます。この出力情報は、特にスキャナによって使用されることを想定して設計されています。このファイルを Flex により生成されたスキャナに含めることで、2 つのプログラムの間に非常に明確なインターフェイスを作ることができます。例として、以下に Bison のファイルを示します。このファイルの名前を '`expr.y`' としましょう。

```
/*
 * expr.y : Bison マニュアル中の例に基づく
 *          Bison による簡単な表現式パーサ
 */

%{
#include <stdio.h>
#include <math.h>

%}

%union {
    float val;
}

%token NUMBER
%token PLUS MINUS MULT DIV EXPON
%token EOL
%token LB RB

%left MINUS PLUS
%left MULT DIV
%right EXPON

%type <val> exp NUMBER

%%
input    :
          | input line
          ;

line     : EOL
          | exp EOL { printf("%g\n", $1); }
```

```

exp      : NUMBER                { $$ = $1;      }
        | exp PLUS exp          { $$ = $1 + $3;  }
        | exp MINUS exp         { $$ = $1 - $3;  }
        | exp MULT exp          { $$ = $1 * $3;  }
        | exp DIV exp           { $$ = $1 / $3;  }
        | MINUS exp %prec MINUS { $$ = -$2;    }
        | exp EXPON exp         { $$ = pow($1,$3); }
        | LB exp RB             { $$ = $2;      }
        ;

```

```
%%
```

```

void yyerror(char *s)
{
    printf("%s\n",s);
}

```

```

int main()
{
    yyparse();
}

```

これは非常に簡単な計算機の文法定義です。

‘-y -d’オプション付きで呼び出されると、Bison は ‘y.tab.h’ というファイルを生成します。このファイルには以下のような定義が、それにきわめてよく似た定義が含まれます。

```

typedef union {
    float val;
} YYSTYPE;
extern YYSTYPE yylval;
#define NUMBER 258
#define PLUS 259
#define MINUS 260
#define MULT 261
#define DIV 262
#define EXPON 263
#define EOL 264
#define LB 265
#define RB 266

```

Flex がトークンの値を正しく Bison に返すことができるように、(#include を使って) これをスキナナに含めることができます。そのコードは以下のようなものになります。

```

/*
 * expr.lex : 簡単な表現式パーサのためのスキャナ
 */

%{
#include "y.tab.h"
%}

%%

[0-9]+      { yyval.val = atof(yytext);
              return(NUMBER);
            }
[0-9]+\.[0-9]+ {
              sscanf(yytext,"%f",&yyval.val);
              return(NUMBER);
            }
"+"         return(PLUS);
"-"         return(MINUS);
"*"         return(MULT);
"/"         return(DIV);
"^"         return(EXPON);
"("         return(LB);
")"         return(RB);
\n          return(EOL);
.           { yyerror("Illegal character");
              return(EOL);
            }

%%

```

上記のファイルは、以下のようにしてコンパイルすることができます。

```

bison -d -y expr.y
flex -I expr.lex
cc -o expr y.tab.c lex.yy.c alloca.c

```

また、この例のソースが手元にあれば、`examples` サブディレクトリにおいて `make expr` を実行するだけでコンパイルできます。どちらの方法でも、`expr` という名前の簡単な計算機が生成されます。これは以下のような表現式をパースして、その結果を出力します。

```
1 + 2 * (199*2)
```

これを見てお分かりのように、この種のインターフェイスは非常に柔軟であり、かつ、保守も非常に容易です。(トークンを定義する名前が変わらない限り) Bison と Flex の間のインターフェイスを変更することなく、Flex、Bison いずれの入力情報においても、機能の追加や削除、定義やコードの変更を行うことが可能です。

この例では、Flex と Bison の間で情報を渡すための別の方法を導入していることに注意してください。この例では、数字の値を Bison に返すのに `yylval` を使っています。これについては次の節でより詳細に説明します。ここではとりあえず、`return` 文の使い方を学んでおいてください。

注：これは単純な例です。表現式のパーズ処理についてより詳しく知りたい人は、*Bison* のマニュアルを参照してください。

4.4.2 YYSTYPE と `yylval`

Flex から Bison に対して、単なる整数値以上の情報を渡す必要の生じることがよくあります。例えば、コンパイラにおいては、どのような種類のトークンが認識されたかだけでなく、そのトークンの値についても知る必要がある場合がときどきあります。文字列、文字、および数値定数などが良い例です。ここで問題なのは、どのようにして Flex にこうした情報を返させるかです。

その答は、Bison が持っている `%union` 文です。これは、`YYSTYPE` という型を定義するものです。`YYSTYPE` は、パーサ定義中において使われるすべての正当なデータ型の共用体 (`union`) です。Bison がカレントなパーズ状態に関連づけたデータを保存するために使う、`YYSTYPE` 型の変数 `yylval` というものがあり、Flex から `yylval` に値を設定することができるので、トークンの型だけでなく、それ以上の情報を Bison に返すことができます。

Bison において `%union` を宣言して `-d` オプションを使うと、Bison は `‘.tab.h’` という拡張子を持つファイルを作成して、そこにトークンの定義情報だけでなく、`YYSTYPE` と `yylval` の宣言も含めます。したがって、`yylval` にアクセスするためにしなければならないことは、Flex の定義情報の中にこの `‘.tab.h’` ファイルをインクルードすることだけです。これは、追加の C コード・セクションにおける定義の先頭でインクルードしなければなりません (see Section 4.4.1 [Interfacing Flex and Bison], page 61)。

注：初期のバージョンの *Bison* は、自動的に `YYSTYPE` と `yylval` の宣言を生成しません。この場合には、より新しいバージョンの *Bison* を入手するか、もしくは、Flex の定義ファイルの先頭において `YYSTYPE` と `yylval` を宣言する必要があります。

4.5 Flex と Bison のもう1つの実例

コードを読むのは、プログラミングの方法を学ぶ良い方法です。そこで、Flex、Bison のインターフェイス例をもう1つ示すことにします。下の例では、拡張してデータベースを操作するために使うことができるような、小規模な言語のための簡単なパーサを作ります。

4.5.1 インターフェイス言語

データベースとのインターフェイス言語は、英語の非常に小さなサブセットになります。文法はおおよそ以下のようになります。

```

command_list      ::= sentence {sentence ...}
sentence          ::= verb_phrase noun_phrase position_phrase
                    adverb period
verb_phrase       ::= VERB | adverb VERB
noun_phrase       ::= declared_noun | qualified_noun | noun
declared_noun     ::= declarator NOUN
declarator        ::= THIS | THAT | THE | THOSE
qualified_noun    ::= qualifier NOUN
qualifier         ::= SOME | MANY | ALL { declarator } NOUN
position_phrase   ::= position declarator NOUN | empty
position          ::= IN | ON | AT
adverb            ::= ADVERB | empty

```

結果として作成されるプログラムは、以下のような文章を受け付けます。

```

FIND MEN
QUICKLY FIND MEN
FIND ALL MEN ON THE NETWORK
QUICKLY FIND ALL MEN ON THE NETWORK
FIND ALL MEN ON THE NETWORK QUICKLY

```

この例では、Bison と Flex の間のインターフェイスが明確に示されるよう、文章の簡単な解析結果が表示されます。このプログラムを試しに実行してみると、その表示結果は大体以下のようになります。

```

% front
FIND MEN
I understand that sentence.
VP = FIND
NP = MEN
PP =
AD =
QUICKLY FIND ALL THE MEN ON THE NETWORK
I understand that sentence.
VP = QUICKLY FIND
NP = ALL THE MEN
PP = ON THE NETWORK
AD =
^C
%

```

これは特別便利なものではありません。というのは、これは文章の構成要素を表示する以外に何も行わないからです。しかし、そこには拡張のためのフックもありますし、一般的な技法も示されています。より一般的な形式の文章を受け付けるよう、この例を拡張してみてください。ほとんどの場合、文章は動詞句 (VERB) と名詞句 (NOUN) に分割することができますが、所有格名詞、名詞の後ろに名詞が続く場合など、文章を構成する他の要素も許容されるようにする必要があります。(‘FIND ALL

JONE'S CAT NAMES'のような文章をどうやってパースするかを想像してみてください。) Bison の文法やその使い方に関する詳しい説明については、*Bison* のマニュアルを参照してください。

4.5.2 実装：コマンド文パーサ

上の節で、小規模な言語について説明しました。次にそれを実装してみることにしましょう。以下のファイルがこれを実現します。

注：これはあくまでも 1 つの例として見てください。特に文法の部分は、英語のパース処理としてはあまり良い例ではありません。

以下は *Bison* のファイルです。%union の部分、および、yylval にアクセスするために \$\$ と \$n を使う方法に注目してください。

```
/* C コードはファイルの先頭で提供する */

%{

#include <stdio.h>
#include <string.h>

extern int  yylexlinenum; /* lex.yy.c に存在する */
extern char *yytext;      /* カレント・トークン */

%}

/* キーワードと予約語がここから始まる */

%union{
    char    name[128];      /* これはデータの共用体 */
                        /* 名前          */
}

/*----- 予約語 -----*/

%token PERIOD
%token NEWLINE
%token POSITIONAL

%token VERB
%token ADVERB

%token PROPER_NOUN
%token NOUN
```

```

%token DECLARATIVE
%token CONDITIONAL

%type <name> declarative
%type <name> verb_phrase
%type <name> noun_phrase
%type <name> position_phrase
%type <name> adverb

%type <name> POSITIONAL VERB ADVERB PROPER_NOUN
%type <name> NOUN DECLARATIVE CONDITIONAL

%%

sentence_list : sentence
              | sentence_list NEWLINE sentence
              ;

sentence : verb_phrase noun_phrase position_phrase
         adverb period
         {
           printf("I understand that sentence.\n");
           printf("VP = %s \n",$1);
           printf("NP = %s \n",$2);
           printf("PP = %s \n",$3);
           printf("AD = %s \n",$4);
         }
         | { yyerror("That's a strange sentence!"); }
         ;

position_phrase : POSITIONAL declarative PROPER_NOUN
               {
                 sprintf($$, "%s %s %s", $1, $2, $3);
               }
               | /* 空 */ { strcpy($$, ""); }
               ;

verb_phrase : VERB { strcpy($$, $1); strcat($$, " "); }
            | adverb VERB
            {
              sprintf($$, "%s %s", $1, $2);
            }
            ;

```



```

adverb : ADVERB      { strcpy($$, $1); }
      | /* 空 */      { strcpy($$, ""); }
      ;

noun_phrase : DECLARATIVE NOUN
            {
                sprintf($$, "%s %s", $1, $2);
            }
      | CONDITIONAL declarative NOUN
            {
                sprintf($$, "%s %s %s", $1, $2, $3);
            }
      | NOUN { strcpy($$, $1); strcat($$, " "); }
      ;

declarative : DECLARATIVE { strcpy($$, $1); }
          | /* 空 */      { strcpy($$, ""); }
          ;

period : /* 空 */
      | PERIOD
      ;

%%

/* main() および yyerror() 関数を提供する */

void main(int argc, char **argv)
{
    yyparse();          /* ファイルをパースする */
}

int yyerror(char *message)
{
    extern FILE *yyout;

    fprintf(yyout, "\nError at line %5d. (%s) \n",
            yylexlinenum, message);
}

```

以下は *Flex* のファイルです。文字列が渡される方法に注意してください。これは最適化された方法ではありませんが、最も理解しやすい方法です。

```

%{
#include <stdio.h>
#include <string.h>
#include "y.tab.h"      /* これは Bison により生成される */

#define TRUE 1
#define FALSE 0

#define copy_and_return(token_type) \
    { \
        strcpy(yylval.name,yytext);\
        return(token_type); \
    }

int yylexlinenum = 0; /* 行数カウント用 */
%}

%%
/* 字句解析ルールがここから始まる */

MEN|WOMEN|STOCKS|TREES      copy_and_return(NOUN)
MISTAKES|GNUS|EMPLOYEES    copy_and_return(NOUN)
LOSERS|USERS|CARS|WINDOWS  copy_and_return(NOUN)

DATABASE|NETWORK|FSF|GNU   copy_and_return(PROPER_NOUN)
COMPANY|HOUSE|OFFICE|LPF   copy_and_return(PROPER_NOUN)

THE|THIS|THAT|THOSE       copy_and_return(DECLARATIVE)

ALL|FIRST|LAST            copy_and_return(CONDITIONAL)

FIND|SEARCH|SORT|ERASE|KILL copy_and_return(VERB)
ADD|REMOVE|DELETE|PRINT    copy_and_return(VERB)

QUICKLY|SLOWLY|CAREFULLY   copy_and_return(ADVERB)

IN|AT|ON|AROUND|INSIDE|ON  copy_and_return(POSITIONAL)

"."                        return(PERIOD);
"\n"                      yylexlinenum++; return(NEWLINE);
.
%%

```

これらのファイルは、以下を実行することでコンパイルできます。

```
% bison -d front.y
% flex -I front.lex
% cc -o front alloca.c front.tab.c lex.yy.c
```

または、この例のソースが手元にあれば、`examples` サブディレクトリにおいて `make front` を実行することでもコンパイルできます。

注: *Bison* パーサは `alloca.c` というファイルが必要とします。このファイルは `examples` サブディレクトリにあります。*Bison* の代わりに *yacc* を使うのであれば、このファイルは必要ありません。

4.5.3 実装に関する注

以下に実装に関する注を示します。

- **YYSTYPE と yylval**
yylval が Flex からアクセスされる方法に注目してください。Bison 文法においてパース・ツリーの上位にデータを渡す方法については、*Bison* のマニュアルに説明されていますが、Flex に対しては何の影響も持ちません。整数値、浮動小数点数値、および他の任意の型のデータも同様の方法で返すことができます。
- **トークン値の返却**
この例では、トークンの型と値の両方が Bison からアクセスできるように、トークンの値と文字列の値の両方が Bison に返されていることに注意してください。
- **Bison と Flex**
Bison と Flex がいかにうまく調和しているかに注目してください。データを交換するためのコード以外に、インターフェイスのためのコードは一切ありません。Bison は `yyllex()` を呼び出し、スキャナがトークン定義を提供しています。

4.6 Flex と C++ (Flex 2.5 の補足情報)

Flex 2.5 では、Flex に対する C++ インターフェイスが提供されています。

Flex の C++ インターフェイスを使うためには、Flex 実行時に `-+` オプションを指定するか、スキャナ定義ファイルの中で `%option c++` を指定する必要があります。これにより、C++ のスキャナ・クラスを実装する `lex.yy.cc` というファイルが生成されます。

`lex.yy.cc` は、Flex が提供する `FlexLexer.h` をインクルードします。この `FlexLexer.h` の中に、C++ スキャナ・クラスの実装に利用される 2 つの C++ クラス (`FlexLexer` と `yyFlexLexer`) が定義されています。

`FlexLexer` は、C++ スキャナ・クラスが実装すべきインターフェイスを構成する抽象仮想関数を定義するクラスです。

`FlexLexer` の持つメンバを以下に示します。

```
char* yytext
    最後に認識された文字列 ( トークン ) を保持します。
```

```
int yy leng
    最後に認識された文字列 ( トークン ) の長さを保持します。

int yylineno
    ‘%option yylineno’が指定されている場合は、入力された行数
    を保持します。それ以外の場合は、固定値 1 を持ちます。

int yy_flex_debug
    この値がゼロ以外の場合、C++スキャナはデバッグ出力を行います。
```

次に、FlexLexerの持つメンバ関数のうち、抽象仮想関数ではないものを以下に示します。

```
const char* YYText()
    メンバ yytextの値を返します。

int YYLeng()
    メンバ yy lengの値を返します。

int yylex(istream* new_in, ostream* new_out = 0)
    new_in と new_out を引数に指定して、メンバ関数 switch_
    streams()を呼び出した後、メンバ関数 int yylex(void)を呼
    び出します。

int lineno() const
    メンバ yylinenoの値を返します。

int debug() const
    メンバ yy_flex_debugの値を返します。

void set_debug(int flag)
    flag をメンバ yy_flex_debugに代入します。6
```

次に、FlexLexerの持つ抽象仮想メンバ関数を列挙します。

```
void yy_switch_to_buffer(struct yy_buffer_state* new_buffer)
struct yy_buffer_state* yy_create_buffer(istream* s, int size)
void yy_delete_buffer(struct yy_buffer_state* b)
void yyrestart(istream* s)
int yylex()
void switch_streams(istream* new_in = 0, ostream* new_out = 0)
```

最初の 5 つのメンバ関数は、Flex の C インターフェイスにおける同名の関数と同等の機能を実現します。C インターフェイスでは、FILE*となっていた引数の型が、

⁶ Flex の仕様では、Flex の起動時に ‘-d’ オプションを指定するか、スキャナ定義ファイルの中に ‘%option debug’ を指定すると、スキャナ実行時にデバッグ情報が出力されることになっていますが、Flex 2.5.4 で C++ スキャナを生成した場合は、いずれの方法でもデバッグ情報は出力されません。デバッグ情報を出力するためには、C++ プログラムから明示的にこの set_debug() メンバ関数を (引数にゼロ以外の値を指定して) 呼び出す必要があります。

istream*となっている点に注意してください。最後の `switch_streams()` は、入出力ストリームの切り替えを行います。これらの抽象仮想メンバ関数の定義は、サブクラス `yyFlexLexer` において与えられ、そのコードは `'lex.yy.cc'` の中に生成されます。

`yyFlexLexer` は、`FlexLexer` のサブクラスです。デフォルトの状態では、`yyFlexLexer` のインスタンスを生成して、`yylex()` メンバ関数を呼び出すことによって、スキナ処理が実行されます。以下に例を示します。

```
int main( int /* argc */, char** /* argv */ )
{
    FlexLexer* lexer = new yyFlexLexer;
    while(lexer->yylex() != 0)
        ;
    return 0;
}
```

これは、C インターフェイスにおける、以下のコードに対応します。

```
int main( int /* argc */, char** /* argv */ )
{
    yylex();
    return 0;
}
```

スキナ定義ファイルの中に `'%option yyclass="classname"'` を指定すると、`'lex.yy.cc'` に `classname::yylex()` が生成されます。クラス `classname` を `yyFlexLexer` のサブクラスとして定義することによって、`classname` のインスタンスを使ってスキナ処理を実行することができます。クラス `classname` を定義する際、以下に示す、`yyFlexLexer` の持つ `protected` メンバ関数を再定義することによって、スキナの振る舞いを変更することができます。

```
int LexerInput(char* buf, int max_size)
    これを再定義することによって、スキナの入力処理を変更することができます。最大 max_size バイトの文字を入力して buf の指す領域にセットし、実際に入力したバイト数を戻り値とします。入力を対話的に扱う場合と扱わない場合で、処理内容を変更する必要がある場合は、#ifdef YY_INTERACTIVE を使います。

void LexerOutput(const char* buf, int size)
    これを再定義することによって、スキナの出力処理を変更することができます。buf の指す領域にある size バイトの文字を出力します。

void LexerError(const char* msg)
    これを再定義することによって、エラー・メッセージの出力処理を変更することができます。エラー・メッセージは、引数 msg で渡されます。
```

スキャン処理に関わるすべてのコンテキスト情報は、yyFlexLexerのインスタンスの内部に閉じています。このことは、C++スキャナ・クラスを使うことによって、再入可能なスキャナを生成することが可能であることを意味しています。

複数の C++スキャナ・クラスを生成して、1つの実行プログラムにリンクすることも可能です。これを行うには、Flex 起動時に `-Pprefix` オプションを指定するか、スキャナ定義ファイルの中に `%option prefix="prefix"` を指定することによって、yyFlexLexerの名前を `prefixFlexLexer` に変更します。`prefixFlexLexer` クラスを使うソース・ファイルの中では、以下のようにして `FlexLexer.h` をインクルードすることによって、`prefixFlexLexer` (実際には `yyFlexLexer`) の定義を参照する必要があります。

```
#undef yyFlexLexer
#define yyFlexLexer prefixFlexLexer
#include <FlexLexer.h>
```

5 Flex の他の特徴

ここでは、*Lex* が提供していない機能や一般にはあまり使われない機能を説明します。*Flex* はほぼ 100 パーセント *Lex* 互換ですが、*Lex* よりも後に実装されたため、性能的により優れており、また、広範な用途に使えるスキャナをより簡単に作成することができるよう、特別な機能を提供しています。

5.1 大文字・小文字を区別しないスキャナ

多くの言語は、その識別子において大文字・小文字を区別しません (Pascal、BASIC、FORTRAN 等)。Lex にも、大文字・小文字を区別しないスキャナを指定するための方法がありますが、それらは概して美しくなく、理解するのも困難です。個々の文字を置き換えてくれる定義を、長いリストにして作成することも可能ですし、すべての識別子を受け付ける 1 つのルールを作成し、そのルールにおいて大文字・小文字を変換してから、トークンの種類を返すようにすることも可能です。以下のコードは、この 2 つの方法を示すものです。定義を使うのであれば、以下のようになります。

```
A [aA]
B [bB]
...
Z [zZ]

%%
{B}{E}{G}{I}{N}      return(BEGIN_SYM);
{E}{N}{D}             return(END_SYM);
```

これに似た操作をサブルーチンで実行するのであれば、以下のようになります。

```
ALPHA      [a-zA-Z]
NUM        [0-9]
ALPHANUM   {ALPHA}|{NUM}

%%
{ALPHA}{ALPHANUM}*    return(convert_and_lookup(yytext));
```

もっともこれは、関数呼び出しの必要があるため、効率が悪くなります (Flex では、パターンの複雑さは大した影響をもたらしません)。

ほかにもこれと同じことを行う方法がありますが、いずれもエレガントではありません。

5.1.1 ‘-i’ オプション

Flex は、この問題を簡単に解決するための方法を提供しています。コマンドラインで ‘-i’ オプションを使うことによって、入力情報の大文字・小文字を区別しないスキャナを生成するよう、Flex に対して通知することができます。つまり、Flex では上記のようなテクニックを使う必要がないということを意味しています。例えば、

```
%%
begin      return(BEGIN_SYM);
end        return(END_SYM);
```

は、`‘-i’`オプションを使うことによって、`‘BEGIN’`、`‘begin’`、`‘BeGiN’`、およびこれ以外のすべての大文字・小文字の組み合わせにマッチします。これは、Lexにおいて同様のことを行うための方法よりも、はるかに簡単です。

`‘-i’`オプションには1つ注意すべき点があります。それは、スキャナが大文字・小文字を区別しないだけで、その変換まではしてくれないということです。つまり、Pascalにおいてシンボル名をハッシュしたいような場合、自分でシンボル名を大文字または小文字に変換しなければならないことを意味しています。そうしないと、`‘F00’`と`‘foo’`は異なるものとして扱われます。これは、シンボルを保存するルーチンの中で対処することもできますし、YY_USER_ACTIONを使うことによって対処することもできます。これを実現する方法の例については、Section 4.1 [Flex and C], page 43におけるYY_USER_ACTIONの説明を参照してください。

5.2 `‘-I’`オプション：対話型スキャナ

Flexの問題として、どのルールを適用するかを決定する前に、入力情報中の次の1文字を先読みする必要があることがあります。対話的ではない使い方をする場合には問題になりませんが、Flexを使ってユーザから直接入力文字を受け取るような場合には、問題になることがあります。

このような場合を2つ挙げると、1つはシェルとやりとりする場合、もう1つはデータベースのフロント・エンドとやりとりする場合です。通常のアクションは、改行が入力の終わりを表すというもので、改行自身は一種の「中身の無い文」として受け付けるのが望ましいのですが、通常のFlexスキャナではこれは可能ではありません。Flexが常に先読みをするという事実は、改行が認識されるためにはユーザが次の行を入力しなければならないということを意味しています（すなわち、単一の改行は、それだけでは認識されず、他の文字が入力される必要があるということです）。これはシェル上ではまったく望ましくありません。

Flexにはこれを回避する方法があります。コマンドラインで`‘-I’`オプションを使うと、Flexは、必要な場合にしか先読みをしない特別な対話型スキャナを生成します。この種のスキャナは、ユーザからの入力を直接受け取るのに適していますが、若干の性能低下を引き起こすかもしれません。

注：`‘-I’`オプションは、`‘-f’`、`‘-F’`、`‘-Cf’`、または`‘-CF’`フラグと一緒に使うことはできません。つまり、先読みができないことから来る性能低下に加えて、パーサも性能向上のために最適化することができないということを意味しています。

`‘-I’`オプションに関連するマイナス面は、通常はきわめて小さいので、入力情報がどこから来るのか確かではなく、性能向上のための最適化を施す可能性を諦めても構わないのであれば、コマンドラインにおいて`‘-I’`オプションを使う方が良いでしょう。

5.3 テーブルの圧縮とスキャナのスピード

テーブルの圧縮とスピードの領域では、Flex は Lex の能力をはるかに上回っています。Flex は、使われるオプションに応じて、Lex よりもはるかに高速なテーブル、あるいは、はるかに小さいテーブルを生成することができます。この節では、利用可能なオプションと各オプションがスピードにもたらす影響について説明します。一般的には、テーブルが圧縮されるほど、そのスピードは遅くなります。Flex では、こうしたオプションをコマンドラインで指定します。オプションは以下のとおりです。¹

-f または -Cf

このオプションは、Flex がフル・テーブル (*full table*) を生成すべきであることを指定します。このテーブルはまったく圧縮されず、サイズが大きくなりますが、スピードは速くなります。このオプションが指定された場合は、アクションの部分に REJECT を使うことはできない点に注意してください。

注：'-f' フラグと '-F' フラグは、Flex が生成するテーブルにおいて相違をもたらします。'-f' フラグはフル・テーブル (*full table*) を生成し、'-F' フラグはファスト・テーブル (*fast table*) を生成します。ファスト・テーブルとは、スピードを最大限にするよう最適化されたテーブル形式であり、一方、フル・テーブルには最適化は一切施されません。もたらされる結果はよく似ていますが、テーブルのサイズは大きく異なるものになる可能性があります。

-F または -CF

このオプションはファスト・テーブル (*fast table*) 形式を用いてテーブルを生成するよう Flex に通知します。一般的には、このテーブルのスピードは先に説明したフル・テーブル (*full table*) とほとんど同じですが、使われるパターンに応じて、そのサイズは小さくも大きくもなる可能性があります。原則として、すべての識別子をキャッチするルールのほかにはキーワードの一覧も持つファイルに対しては、'-f' オプションを使うべきです。例えば、

```
ALPHA      [a-zA-Z]
NUM        [0-9]
ALPHANUM   {ALPHA}|{NUM}
%%
begin      return(BEGIN_SYM);
... rules and actions ...
end        return(END_SYM);
{ALPHA}{ALPHANUM}* return(IDENTIFIER);
```

は '-f' フラグを使って処理すべきであり、

¹ 訳注：Flex 2.5 では、ここに列挙されているもの以外に、'-Ca' オプションをサポートしています。これについては、Section 2.2 [Command Line Switches (Flex 2.5)], page 11 を参照してください。

```
{ALPHA}{ALPHANUM}* {ECHO;
                                return(lookup(ytext));}
```

は ‘-F’ フラグを使って処理すべきです。これらのオプションが指定されている場合は、アクションの部分に REJECT を使うことができない点に注意してください。

-Ce このオプションを使うと、性能にはわずかしかな影響を及ぼさずに、テーブルのサイズをかなり小さくすることができます。‘-Ce’ が使われると、Flex は同等クラス (*equivalence classes*) を作成します。同等クラスとは、同一の方法で使われる文字のグループです。例えば、使われる数字が集合 [0-9] の範囲に限定されるのであれば、0 から 9 までの数は同等クラスの中に置かれることになります。

-Cfe, -CFe 同等クラスを持つファスト・テーブルです。このオプションによって生成されたスキャナもまた高速であり、かつ、‘-Cf’ あるいは ‘-CF’ を指定して生成されたスキャナと比較して、サイズもはるかに小さくなる可能性があります。サイズ、またはスピードの一方が他方に比べてはるかに重要であるということがないのであれば、これは良い組み合わせです。

-Cm Flex に対してメタ同等クラス (*meta-equivalence classes*) を使うよう通知します。これは、一緒に使われることが多い文字の集合、または (同等クラスが使われている場合には) 同等クラスです。同等クラスを使う場合よりも性能はさらに悪くなりますが、これは多くの場合、テーブル・サイズを小さくするのに非常に効果的な方法です。

-Cem デフォルトのテーブル圧縮です。このオプションで生成されるスキャナは、Flex が生成するスキャナの中で事実上最も小さく、かつ、最も性能の劣るものになります。

-C ‘-C’ オプション単体では、同等クラスやメタ同等クラスを使わずにテーブルを圧縮するよう、Flex に対して通知します。

注: ‘-Cxx’ オプションは、コマンドライン上には 1 つだけ指定すべきです。というのは、このうち最後に見つかったオプションだけが実際の効果を持つからです。したがって、

```
flex -Cf -Cem foo.l
```

は、Flex に ‘-Cem’ オプションを使わせることになります。

Flex のデフォルトの動作は、コマンドライン上で ‘-Cem’ オプションを使った場合に相当します。この動作では圧縮を最大限に行うことになり、一般的には最も遅いスキャナが生成されることになります。こうした小さなテーブルはより速く生成され、コンパイルもより速く実行されるので、デフォルトは、開発段階では非常に便利です。スキャナのデバッグが終了した後は、より高速な (そして通常はよりサイズの大きい) スキャナを作成することができます。

5.4 翻訳テーブル

翻訳テーブルは、文字をグループにマップするのに使われます。このテーブルは Lex の持つ機能の 1 つですが、POSIX では定義されていません。Flex でも翻訳テーブルを使うことはできますが、サポート対象外の機能です。Flex においては翻訳テーブルは不要です。というのは、Flex には ‘-i’ オプションによる同等クラスというものがあ、これが翻訳テーブルと同等の機能を実現しているからです (see Section 5.1.1 [‘-i’ オプション], page 75)。翻訳テーブルの機能は、互換性のためだけに存在する余分な機能です。翻訳テーブルを使うことはお勧めできません。翻訳テーブルを使いたいのであれば、定義ファイルの先頭の定義セクションにおいて定義しなければなりません。

翻訳テーブルの一般的な形式は以下のとおりです。

```
%t
1 ABCDEFGHIJKLMNOPQRSTUVWXYZ
2 0123456789
%t
%%
```

これは、‘A’から‘Z’までの任意の文字がルールの中で使われている場合、そのパターンは‘A’から‘Z’までのどの文字にもマッチするということを意味しています。したがって、‘A(BC)’と‘X(YZ)’はまったく同一であるということになります。

5.5 複数の入力バッファ

スキャナが、複数のファイルからの入力进行处理することができるということが必要になる状況は、たくさんあります。例えば、多くの Pascal の実装では、コンパイル時に複数のファイルを取り込むことを許していますし、C では、スキャナもしくはプリプロセッサが #include 文进行处理できなければなりません。このことが意味しているのは、スキャナは、カレントなスキャン処理のコンテキストを保存してから新しいコンテキストに変更し、その後で、以前の状態と完全に一致する状態に復帰することができなければならないということです。

Flex スキャナは、スキャン処理のコンテキストを維持するために余分の処理が必要になるような、大きな入力バッファを使っています。しかし Flex は、複数の入力バッファの作成、切り替え、削除が非常に簡単に行えるような特別な機能を提供しています。

5.5.1 バッファを操作する関数

Flex は、複数の入力バッファを取り扱うために、以下のような関数やマクロを提供しています。

```
YY_BUFFER_STATE yy_create_buffer( FILE *file, int size)
    fileで指定されるファイルのために、sizeで指定される数の文字
    を格納するのに十分な大きさのバッファを作成します。この関数
```

は、後に複数のバッファ間の切り替え、または新規に作成されたバッファの削除に使うことのできるハンドルを返します。

YY_BUF_SIZE

デフォルトのバッファ・サイズを定義するマクロです。yy_create_buffer()に渡すべきサイズが分からない場合に、これを使うことができます。

void yy_switch_to_buffer(YY_BUFFER_STATE new_buffer)

バッファを切り替えます。次に読み込まれるトークンは、new_bufferで指定されるバッファから取られます。ファイルの終端(EOF)に達するか、次に yy_switch_to_buffer()が呼び出されるまで、new_bufferからトークンが読み込まれます。new_bufferが EOF に達すると、新しいバッファに切り替えることができます。

void yy_delete_buffer(YY_BUFFER_STATE buffer)

bufferで指定されるバッファを削除し、それに割り当てられたメモリを解放します。

YY_CURRENT_BUFFER

使用中のカレントなバッファを返すマクロです。

上記が、複数の入力バッファを取り扱うのに必要なすべての機能を提供しています。

5.5.2 バッファを操作する関数 (Flex 2.5 の補足情報)

Flex 2.5 では、以下のバッファ操作関数もサポートされています。

YY_BUFFER_STATE yy_new_buffer(FILE *file, int size)

yy_create_bufferの別名です。

void yy_flush_buffer(YY_BUFFER_STATE buffer)

引数で指定されたバッファの内容を破棄し、バッファの先頭2バイトに YY_END_OF_BUFFER_CHAR ('\0') をセットします。

YY_FLUSH_BUFFER

引数にカレント・バッファを指定して yy_flush_buffer()を呼び出すよう定義されたマクロです。

さらに、Flex 2.5 では、メモリ上の文字列を操作するための入力バッファを作成する関数が提供されています。いずれも、新しく作成された入力バッファに対応する YY_BUFFER_STATE型のハンドルを戻り値とします。入力バッファを使い終わったら、このハンドルを引数に指定して yy_delete_buffer()を呼び出す必要があります。

YY_BUFFER_STATE yy_scan_string(const char *str)

NULL 文字で終端する文字列をスキャンするための入力バッファを作成します。実際には、引数 str に指定された文字列の長さを調べて、次に説明する yy_scan_bytes()を呼び出し、その戻り値を返します。

`YY_BUFFER_STATE yy_scan_bytes(const char *bytes, int len)`
bytes から始まる *len* バイトのメモリ領域をスキャンするためのバッファを作成します。実際には、次に説明する `yy_scan_buffer()` を呼び出し、その戻り値を返します。

`yy_scan_buffer()` の第 1 引数には、*bytes* ではなく、この関数の内部で獲得された *len* + 2 バイトの領域へのポインタが渡される点に注意してください。`yy_scan_buffer()` が呼び出される前に、*bytes* から始まる *len* バイトのデータが、新たに獲得した領域にコピーされ、さらに、末尾の 2 バイトに `YY_END_OF_BUFFER_CHAR` (`'\0'`) がセットされます。

`YY_BUFFER_STATE yy_scan_buffer(char *base, yy_size_t size)`
base から始まる *size* バイトのメモリ領域をスキャンするためのバッファを作成します。メモリ領域の末尾 2 バイトは、`YY_END_OF_BUFFER_CHAR` (`'\0'`) でなければなりません。この末尾 2 バイトは、スキャン処理の対象になりません。

引数で指定されたメモリ領域の末尾 2 バイトが `YY_END_OF_BUFFER_CHAR` でない場合は、`yy_scan_buffer()` はバッファを作成せず、`NULL` ポインタを返します。

5.5.3 複数バッファを使う実例

複数のバッファを使うというアイデアを理解するための手助けとして、インクルードすべきファイルを探す C のスキャナの一部を以下に示します。これは C の `#include` のうち、引用符で囲まれた文字列のみを受け付けます。例えば、

```
#include "file1.c"
#include "file2.c"
#include " file3.c"
```

は、最後の例のファイル名が空白を含むことになりましたが、いずれも正当な入力です。ここでの例はまた、EOF ルールとスタート状態の使用法を実演する良い例でもあります。

```
/*
 * eof_rules.lex : 複数バッファ、EOF ルール、スタート状態
 *               の使い方の例
 */

%{

#define MAX_NEST 10
```

```

YY_BUFFER_STATE include_stack[MAX_NEST];
int             include_count = -1;

%}

%x INCLUDE

%%

^"#include"[ \t]*\" BEGIN(INCLUDE);
<INCLUDE>\" BEGIN(INITIAL);
<INCLUDE>[^\"]+ { /* インクルード・ファイルの名前を獲得する */
    if ( include_count >= MAX_NEST){
        fprintf( stderr, "Too many include files" );
        exit( 1 );
    }

    include_stack[++include_count] = YY_CURRENT_BUFFER;

    yyin = fopen( yytext, "r" );
    if ( ! yyin ){
        fprintf(stderr,"Unable to open \"%s\"\n",yytext);
        exit( 1 );
    }

    yy_switch_to_buffer(
        yy_create_buffer(yyin,YY_BUF_SIZE));

    BEGIN(INITIAL);
}
<INCLUDE><<EOF>> {
    fprintf( stderr, "EOF in include" );
    yyterminate();
}
<<EOF>> {
    if ( include_count <= 0 ){
        yyterminate();
    }
}

```

```

    } else {
        yy_delete_buffer(include_stack[include_count--] );
        yy_switch_to_buffer(include_stack[include_count] );
        BEGIN(INCLUDE);
    }
}
[a-z]+          ECHO;
.|\n           ECHO;

```

スタート状態を使ってファイル名のスキナを生成する方法や、バッファの切り替えを発生させる方法に注目してください。ほかに注目すべき重要な点は、`<<EOF>>`を取り扱うセクション、および、古いバッファに復帰する際に `BEGIN` を使って確実に正しい状態に遷移するようにする点です。これを怠ると、状態は `INITIAL` にリセットされ、`#include` の最後の `'"` が `ECHO` されてしまいます。

注: `<<EOF>>` 機能は次の節で説明します。`<<EOF>>` が何であり、何を行うものかという点に関する詳細な議論については、*See Section 3.8 [Start States], page 24.*

5.6 ファイルの終端 (End-Of-File) ルール

ファイルの終端 (EOF) が見つかると、Flex は `yywrap()` を呼び出し、ほかに処理できる状態のファイルが存在するか調べます。`yywrap()` が 0 以外の値を返すと、もうこれ以上ファイルはないということを意味し、したがって、これがまさに入力最後の状態であるということになります。状況によっては、この時点でさらに処理を行う必要がある場合があります (例えば、入力のために別のファイルをセットアップしたいということがあるかもしれません)。このような場合のために、Flex は `<<EOF>>` 演算子を提供しています。これを使うことで、EOF が見つかった時に実行すべきことを定義することができます。See Section 5.5.3 [複数バッファを使う実例], page 81。EOF ルールを使って、終わりのないコメントやインクルードされているファイルの終端を見つける、良い例が示されています。

`<<EOF>>` 演算子の使用にはいくつか制限があります。制限事項を以下に示します。

- パターンと一緒に使用することは不可

EOF ルールは、スタート状態とのみ一緒に使うことができます。スタート状態が指定されていない場合 (すなわち、`<<EOF>>` ルールが状態により制限されない場合)、`<<EOF>>` が使われていないすべての (排他的スタート状態を含む) 状態が影響を受けます。つまり、

```
"foo"<<EOF>>
```

が不当である一方で、

```

<<EOF>>          /* <<EOF>>が使われていないすべての */
                  /* 状態における EOF                */
<indent><<EOF>>   /* indent 状態における EOF          */
<comment><<EOF>>  /* コメント内の EOF                */

```

はすべて正当であることを意味しています。

- アクションの終端

1つ注意しなければならない点は、EOF ルールは入力の最後で呼び出されるという点です。したがって、EOF ルールのアクションは、(1) (`yy_switch_to_buffer()`、または `YY_NEW_FILE` を使って) 新しい入力ストリームを確立する、(2) (`return` 文を使って) 復帰する、(3) (`yyterminate()`、または `exit()` を使って) スキャナの実行を終了させる、のいずれかを実行しなければなりません。

See Section 5.5.3 [複数バッファを使う実例], page 81. `yy_terminate()` と `yy_switch_to_buffer()` を使う例が示されています。また、`yyterminate()` の説明については、Section 4.1 [Flex and C], page 43 を参照してください。

6 スキャナの最適化

デバッグをしている間は、スキャナの性能は通常それほど重要ではなく、*Flex*のデフォルトの設定で十分です。しかしデバッグ終了後は、スピード、またはサイズの面でスキャナを最適化したくなることもあるでしょう。ここでは、スキャナを最適化するのによく使われる手法をいくつか紹介します。

6.1 スピードの最適化

多くのプログラムは、字句解析の処理に多くの時間を費やします。したがって、スキャナの最適化はかなり大きな性能改善に結びつくことが多いのです。*Flex*によるスキャナは、*Lex*によるスキャナと比較するとかなり高速になる傾向がありますが、特定の構成もしくはアクションによって、性能に大きな影響を与えることができます。注意すべき点は以下のとおりです。

1. テーブルの圧縮

どのような圧縮も結果的にスキャナを遅くします。したがって、スピードのことが心配であるならば、常にコマンドラインで `-f` オプション、または `-F` オプションを使ってください。テーブルの圧縮とスピードに関連するオプションに関する詳細な議論については、See Section 5.3 [Table Compression and Scanner Speed], page 77。

2. REJECT

スピードに対して最も大きな影響を及ぼします。これが使われるとすべてのマッチ処理が遅くなります。というのは、スキャナは、マッチする前の状態に自身を復旧する必要があるからで、このようなことが必要ない場合と比較して、より多くの内部的な保守作業を行わなければならないからです。スピードが重要な場合には、使わないようにしてください。

3. バックトラッキング

スキャナがあるテキストにマッチするために「逆行」しなければならないことを、バックトラッキングといいます。これは、スキャナの性能に悪い影響を及ぼしますので、スピードが最も重要である場合には避けるべきです。圧縮されたテーブルは常にバックトラッキングを発生させるので、`-f` オプション、または `-F` オプションを使わない場合は、ルールからバックトラッキングを削除しようとするのは時間の無駄です。スキャナからバックトラッキングを削除することに関する詳細な情報については、See Section 6.1.1 [Removing Backtracking], page 86。

4. 可変長後続コンテキスト (variable trailing context)

可変長後続コンテキストとは、あるルールの先頭部分と後続部分の両方が固定長でないような場合を指します。性能の観点からは REJECT と同じくらい悪影響を及ぼすもので、可能な場合にはいつでも避けるべきです。この例を示すと、以下のようになります。

```
%%
linux|hurd/(OS|"Operating system")
```

これは、以下のように分割すべきです。

```
linux/OS|"Operating system"
hurd/OS|"Operating system"
```

こうすることによって、問題は解消されます。

5. 行の先頭を表す演算子
‘^’演算子は、性能に不利な影響を及ぼします。スピードが最も重要な場合には、使わないでください。
6. `yymore()`
`yymore`を使うと性能を低下させます。スピードが最も重要な場合には、使わないでください。
7. テキスト長
スキャナの性能は、マッチするテキストの長さによっても影響を受けます。常に長い文字列にマッチするような場合には、スキャナは高速に実行されます。というのは、`yytext`環境をセットアップする必要がないからです。スキャナの実行時間のほとんどは、内部の高速なマッチング・ループの中で費やされることになります。
8. NUL
Flex は、NULを含むトークンをマッチするのに時間がかかります。この場合には、短いテキストにマッチするようルールを記述するほうが良いでしょう。

6.1.1 バックトラッキングの削除

スキャナからバックトラッキングを削除することは、スキャナの性能にかなりの影響をもたらします。残念ながら、バックトラッキングの削除はかなり複雑な作業になる可能性があります。例えば、

```
%%
hurd      return(GNU_OS);
hurdle    return(JUMP);
hurdled   return(JUMPED);
```

では、バックトラッキングが発生します。スキャナが‘hu’をマッチし、次の文字が‘r’ではない場合、マッチされなかったテキストを ECHOするデフォルトのルールを使って‘h’と‘u’を処理するために、スキャナはバックトラッキングを行わなければなりません。同じことが‘d’と‘e’についても適用されます。(これは、何かにマッチするようスキャナが努力を継続するというのが、もはやできないからです。この場合、スキャナはデフォルトのルールを適用し、`yyext`環境をリセットしなければなりません。が、いずれも時間のかかる処理です。)

コマンドライン・オプション‘-b’を使うことで、バックトラッキングを発生させている原因に関する情報を知ることができます。これにより、バックトラッキングに関する情報を含む‘lex.backtrack’というファイルが生成されます。上記の例の場合、このファイルは以下のような情報を含みます。

```

State #6 is non-accepting -
  associated rule line numbers:
        2          3          4
out-transitions: [ r ]
jam-transitions: EOF [ \000-q s-\177 ]

```

```

State #7 is non-accepting -
  associated rule line numbers:
        2          3          4
out-transitions: [ d ]
jam-transitions: EOF [ \000-c e-\177 ]

```

```

State #9 is non-accepting -
  associated rule line numbers:
        3          4
out-transitions: [ e ]
jam-transitions: EOF [ \000-d f-\177 ]

```

Compressed tables always backtrack.

バックトラッキング情報はセクションに分割され、個々のセクションにおいて、バックトラッキングを引き起こしている 1 つの状態のことが記述されています。個々のセクションの最初の行から、状態番号を知ることができます。2 行目からは、記述ファイルの何行目が関連しているのかを知ることができます。3 行目からは、バックトラッキングを発生させた文字を知ることができます。よって、最初のブロックからは、文字 'r' でバックトラッキングが発生し、それは記述ファイルの 2 , 3 , 4 行目に関連していることを見てとることができます。最後の行は、圧縮されたテーブルは常にバックトラッキングを発生させるので、テーブル圧縮を引き起こすようなコマンドライン・オプションを使う場合には、バックトラッキングを削除しようとして時間を費やすべきではないことを思い出させるためのものです。

バックトラッキングを削除するためには、バックトラッキングが関与している状態をキャッチするルールを加える必要があります。これは、スキャナのスピードには影響を与えないということに注意してください。スキャナのスピードは、ルールの数や複雑さとはまったくといえるほど無関係です。

バックトラッキングを削除するためにルールを追加する方法は、2 種類あります。第 1 の方法は、以下のようなルールを追加することです。

```

%%
hurd      return(GNU_OS);
hurdle    return(JUMP);
hurdl     return(JUMPED);
hu        return(OTHER);
hur       return(OTHER);
hurdl     return(OTHER);

```

別の方法として、すべてをキャッチするようなルールを追加することもできます。

```
%%
hurd      return(GNU_OS);
hurdle    return(JUMP);
hurdled   return(JUMPED);
[a-z]+    return(OTHER);
```

この第2の方法を適用できる場合は、常にこれを使うべきです。上記のどちらかと ‘-b’ オプションを一緒に使うと、

```
Compressed tables always backtrack.
```

というメッセージだけが出力されるようになります。これは、バックトラッキング状態が存在しないことを示唆しています。

これに付随する問題の1つとして、複雑なスキャナではバックトラッキング問題はカスケードする傾向があるので、lex.backtrack内の情報が混乱をもたらすものになる可能性があります。しかし、バックトラッキングの原因は通常2、3個のルールにしばることが可能なので、バックトラック・データを調べようと努力するだけの値打ちはあります。

6.2 サイズの最適化

Flexは、サイズの小さいスキャナよりも、むしろ非常に高速なスキャナを作成することを目標としていますが、いずれにしても、作成されるテーブルのサイズはLexによるそれと比較しても、通常はかなり小さいものになります。

デフォルトでは、Flexは可能な限りサイズの小さいスキャナを作成します。これは、コマンドラインで ‘-Cem’ を使うのと同様です。デフォルトを使うのであれば、コマンドライン・オプションを気にする必要はありません。

さらにテーブルのサイズを小さくするには、より大きなテキスト・グループにマッチするルールを使い、字句の値を認識するためにCのサブルーチンを使うのが最も良い方法です。この良い例がコンパイラで、以下のようなルールを与えることができます。

```
%%
begin     return(BEGINSYM);
end       return(ENDSYM);
program   return(PROGSYM);
...
```

あるいは、以下のようにテーブル検索を使うことも可能です。

```
[a-zA-Z][a-zA-Z0-9]* return(lookup(yytext));
```

ここでは、一般的なルールが指定されていて、lookup()がテキストをキーワードにマッチさせ、そのトークンが何であるかを示す整数値を返します。これにより、サイズのより小さいテーブルが生成されますが、性能は悪くなる傾向があります。また、数が少なく複雑ではないルール集合については、テーブル・サイズを縮小することの効果は、シンボル・マッピング用の情報をプログラム中の他の領域に格納しなければならないという事実によって、相殺されるかもしれません。というのは、シンボル・

マッピング用の情報は、Flex テーブルと比較して、より多くのスペースを必要とする可能性があるからです。

7 Flex を使うその他の実例

ここでは、*Flex* の使用例をさらにいくつか紹介します。ここでの例も、必ずしも最適な実装ではありませんが、一般的な *Flex* の使い方を示してくれるはずです。

7.1 単語数、文字数、行数のカウント

以下の定義は、与えられたファイルの中の単語数、文字数、行数をカウントするのに *Flex* を使う方法を示す、簡単な例です。実際に *Flex* に関係のある部分は、非常に少ないことに注意してください。以下のコードのほとんどは、コマンドライン・パラメータを処理したり、カウントの合計を保持したりするものです。

```
/*
 * wc.lex : wc のようなユーティリティを、
 *          Flex を使って作成する簡単な例
 */

%{
int  numchars = 0;
int  numwords = 0;
int  numlines = 0;
int  totchars = 0;
int  totwords = 0;
int  totlines = 0;
}%

/*
 * ルールはここから始まる
 */

%%

[\\n]      { numchars++; numlines++;      }
[^\t\\n]+  { numwords++;  numchars += yyleng; }
.          { numchars++;                  }

%%

/*
 * 追加的な C コードがここから始まる。
 * ここで、すべての引数処理等を行うコードが提供される
 */
```

```

void main(int argc, char **argv)
{
    int  loop;
    int  lflag = 0; /* 行数をカウントする場合は 1          */
    int  wflag = 0; /* 単語数をカウントする場合は 1          */
    int  cflag = 0; /* 文字数をカウントする場合は 1          */
    int  fflag = 0; /* ファイル名が指定されている場合は 1 */

    for(loop=1; loop<argc; loop++){
        char *tmp = argv[loop];
        if(tmp[0] == '-'){
            switch(tmp[1]){
                case 'l':
                    lflag = 1;
                    break;
                case 'w':
                    wflag = 1;
                    break;
                case 'c':
                    cflag = 1;
                    break;
                default:
                    fprintf(stderr,"unknown option -%c\n",tmp[1]);
            }
        } else {
            fflag = 1;
            numlines = numchars = numwords = 0;
            if((yyin = fopen(tmp,"rb")) != 0){
                (void) yylex();
                fclose(yyin);
                totwords += numwords;
                totchars += numchars;
                totlines += numlines;
                printf("file   : %25s :",tmp) ;
                if(lflag){
                    fprintf(stdout,"lines %5d ",numlines);
                }
                if(cflag){
                    fprintf(stdout,"characters %5d ",numchars);
                }
            }
        }
    }
}

```



```

        if(wflag){
            fprintf(stdout,"words %5d ",numwords);
        }
        fprintf(stdout,"\n");
    }else{
        fprintf(stderr,"wc : file not found %s\n",tmp);
    }
}
}
if(!fflag){
    fprintf(stderr,"usage : wc [-l -w -c] file [file...]\n");
    fprintf(stderr,"-l = count lines\n");
    fprintf(stderr,"-c = count characters\n");
    fprintf(stderr,"-w = count words\n");
    exit(1);
}
for(loop=0;loop<79; loop++){
    fprintf(stdout,"-");
}
fprintf(stdout,"\n");
fprintf(stdout,"total : %25s  ","") ;
if(lflag){
    fprintf(stdout,"lines %5d ",totlines);
}
if(cflag){
    fprintf(stdout,"characters %5d ",totchars);
}
if(wflag){
    fprintf(stdout,"words %5d ",totwords);
}
fprintf(stdout,"\n");
}

```

7.2 Pascal のサブセット用の字句スキャナ

ここでは、Pascal のような言語用の字句スキャナを作る方法を示します。このスキャナ定義では、個々のキーワードがルールとしてリストされています。(一般的には、すべてのキーワードをテーブルに格納してからテーブル検索を使う手法がよく見られますが、)ここでの方法は、キーワードと識別子とを区別するための方法としては、一般的に最も簡単なものです。また、識別子用にただ 1 つのルールがあるという点に注意してください。多くの場合、このルールはシンボル・テーブルを管理するためのサブルーチンを呼び出します。

もう1つ注意すべき点は、‘_FILE’と‘_BEGIN’が先頭にアンダースコアを持つという点です。Flex、またはCで定義済みの名前は、追加的な工夫なしでは使えないということを示すために、このようにしてあります。これよりもっと一般的に使われる手法は、すべてのトークンの先頭もしくは末尾に何らかの文字列を付加するというもので、こうすることによって問題は発生しなくなります。‘TOK’や‘SYM’が一般的によく使われる拡張子です。

```
/*
 * pascal.lex : PASCAL スキャナの例
 */

%{
#include <stdio.h>
#include "y.tab.h"

int line_number = 0;

void yyerror(char *message);

%}

%x COMMENT1 COMMENT2

white_space      [ \t]*
digit            [0-9]
alpha            [A-Za-z_]
alpha_num        ({alpha}|{digit})
hex_digit        [0-9A-F]
identifier       {alpha}{alpha_num}*
unsigned_integer {digit}+
hex_integer      ${hex_digit}{hex_digit}*
exponent         e[+-]?{digit}+
i                {unsigned_integer}
real             ({i}\.{i}?|{i}?\.{i}){exponent}?
string           \'([^\n]|\'\'')+\'
bad_string       \'([^\n]|\'\'')+\'
```

```

%%
"{ "                                BEGIN(COMMENT1);
<COMMENT1>[^]\n]+
<COMMENT1>\n                        ++line_number;
<COMMENT1><<EOF>>                    yyerror("EOF in comment");
<COMMENT1>" }"                      BEGIN(INITIAL);

"(* "                                BEGIN(COMMENT2);
<COMMENT2>[^)*\n]+
<COMMENT2>\n                        ++line_number;
<COMMENT2><<EOF>>                    yyerror("EOF in comment");
<COMMENT2>"*)"                      BEGIN(INITIAL);
<COMMENT2>[*)]

/* FILE と BEGIN は、Flex や C においては既に定義されているため
 * 使うことができない点に注意。これは、すべてのトークンの
 * 先頭に TOK_ やその他の接頭辞を付加することによって、より
 * すっきりと克服することができる
 */

and                                return(AND);
array                              return(ARRAY);
begin                              return(_BEGIN);
case                               return(CASE);
const                              return(CONST);
div                                return(DIV);
do                                 return(DO);
downto                            return(DOWNT0);
else                               return(ELSE);
end                                return(END);
file                               return(_FILE);
for                                return(FOR);
function                           return(FUNCTION);
goto                               return(GOTO);
if                                 return(IF);
in                                 return(IN);
label                             return(LABEL);
mod                                return(MOD);
nil                               return(NIL);
not                                return(NOT);
of                                 return(OF);
packed                            return(PACKED);
procedure                          return(PROCEDURE);

```

```

program          return(PROGRAM);
record           return(RECORD);
repeat           return(REPEAT);
set              return(SET);
then             return(THEN);
to               return(TO);
type             return(TYPE);
until           return(UNTIL);
var              return(VAR);
while            return(WHILE);
with             return(WITH);

"<=" | "=<"      return(LEQ);
">=" | ">="      return(GEQ);
"<>"            return(NEQ);
"="             return(EQ);

".."            return(DOUBLEDOT);

{unsigned_integer} return(UNSIGNED_INTEGER);
{real}           return(REAL);
{hex_integer}    return(HEX_INTEGER);
{string}         return{STRING};
{bad_string}     yyerror("Unterminated string");

{identifier}     return(IDENTIFIER);

[/+\\-,^.;:()\\[\\]] return(yytext[0]);

{white_space}    /* 何もしない */
\\n              line_number += 1;
.                yyerror("Illegal input");

%%
void yyerror(char *message)
{
    fprintf(stderr,"Error: \"%s\" in line %d.  Token = %s\\n",
        message,line_number,yytext);
    exit(1);
}

```

7.3 専門用語の変換

ここでは、スタート状態を使って、Flexにより生成されるスキヤナの内部に小規模のパーサを作る方法の例を示します。このコードは **The New Hackers Dictionary** (‘prep.ai.mit.edu’、およびその他の多くのインターネット FTP サイトから入手可能なテキスト形式のもの) を入力として受け取り、すぐに製版および印刷できる状態の Texinfo フォーマットのドキュメントに変換するものです。このコードは ‘jargon2910.ascii’ を使ってテスト済みです。

典型的な使い方は以下のとおりです。

```
j2t < jargon > jargon.texi
tex jargon.texi
lpr -d jargon.dvi
```

このプログラムは、使用に耐える info ファイルに変換可能なファイルは作成しませんが、こうした機能は大した困難もなく追加することが可能です。この例は非常に長いものですが、大して複雑でもないの、尻込みしないで研究してみてください。

```
/*
 * j2t.lex : スタート状態を利用 ( ひょっとして悪用 ! ) する例
 */

%{
#define MAX_STATES 1024
#define TRUE 1
#define FALSE 0

#define CHAPTER "@chapter"
#define SECTION "@section"
#define SSECTION "@subsection"
#define SSSECTION "@subsubsection"

int states[MAX_STATES];
int statep = 0;

int need_closing = FALSE;

char buffer[YY_BUF_SIZE];

extern char *yytext;
```

```

/*
 * このプログラムが生成する*.texinfo ファイルの先頭部分を作る。
 * これは標準的な Texinfo ヘッダである
 */

void print_header(void)
{
    printf("\\input texinfo @c -*-texinfo-*-\\n");
    printf("@c          %c**start of header\\n",'%');
    printf("@setfilename      jargon.info\\n");
    printf("@settitle          The New Hackers Dictionary\\n");
    printf("@synindex          fn cp\\n");
    printf("@synindex          vr cp\\n");
    printf("@c          %c**end of header\\n",'%');
    printf("@setchapternewpage odd\\n");
    printf("@finalout\\n");
    printf("@c @smallbook\\n");
    printf("\\n");
    printf("@c =====\\n\\n");
    printf("@c This file was produced by j2t. Any mistakes are *not*\\n");
    printf("@c the fault of the jargon file editors.\\n");
    printf("@c =====\\n\\n");
    printf("@titlepage\\n");
    printf("@title      The New Hackers Dictionary\\n");
    printf("@subtitle Version 2.9.10\\n");
    printf("@subtitle Generated by j2t\\n");
    printf("@author Eric S. Raymond, Guy L. Steel, and Mark Crispin\\n");
    printf("@end titlepage\\n");
    printf("@page\\n");
    printf("@c =====\\n\\n");
    printf("\\n\\n");
    printf("@unnumbered Preface\\n");
    printf("@c          *****\\n");
}

```

```
/*
 * 生成される Texinfo ファイルの末尾の部分を作成する
 */

void print_trailer(void)
{
    printf("\n");
    printf("@c =====\n");
    printf("@contents\n"); /* 目次を表示する */
    printf("@bye\n\n");
}

/*
 * 後でそれを見つけることができるよう、節または章に下線を引く
 */

void write_underline(int len, int space, char ch)
{
    int loop;

    printf("@c ");

    for(loop=3; loop<space; loop++){
        printf(" ");
    }

    while(len--){
        printf("%c",ch);
    }
    printf("\n\n");
}

/*
 * Texinfo において特殊な意味を持つ文字をチェックし、エスケープする
 */

char *check_and_convert(char *string)
{
    int buffpos = 0;
    int len,loop;
```

```

    len = strlen(string);
    for(loop=0; loop<len; loop++){
        if(string[loop] == '@' ||
           string[loop] == '{' ||
           string[loop] == '}')
        {
            buffer[bufpos++] = '@';
            buffer[bufpos++] = string[loop];
        } else {
            buffer[bufpos++] = string[loop];
        }
    }
    buffer[bufpos] = '\0';
    return(buffer);
}

/*
 * 章、節、項のヘッダを書き出す
 */

void write_block_header(char *type)
{
    int loop;
    int len;

    (void)check_and_convert(ytext);
    len = strlen(buffer);
    for(loop=0; buffer[loop] != '\n'; loop++)

        buffer[loop] = '\0';
    printf("%s %s\n", type, buffer);
    write_underline(strlen(buffer), strlen(type)+1, '*');
}

%}

```



```

/*
 * Flex の記述情報がここから始まる
 */

%x HEADING EXAMPLE ENUM EXAMPLE2
%x BITEM BITEM_ITEM
%s LITEM LITEM2

%%

^[^#]*"#" /* ヘッダとフッタをスキップする */
/*
 * 章は、その下にアスタリスクを持ち、コロンで終わる
 */
^[^\\n:]+\\n[*]+\\n write_block_header(CHAPTER);

"= "[A-Z]" =\\n"=* { /* 個々のカテゴリごとに節を作成する */
    if(need_closing == TRUE){
        printf("@end table\\n\\n\\n");
    }
    need_closing = TRUE;
    write_block_header(SECTION);
    printf("\\n\\n@table @b\\n");
}

"Examples:"[^\\.]+ ECHO;

"*"^[^*\\n]+ "*" { /* @emph{} (強調された) テキスト */
    yytext[yytext-1] = '\\0';
    (void)check_and_convert(&yytext[1]);
    printf("@i{%s}",buffer);
}

"{{"[^{]+"}" { /* 特別な強調 */
    yytext[yytext-2] = '\\0';
    (void)check_and_convert(&yytext[2]);
    printf("@strong{%s}",buffer);
}

```

```

"["^"]+"} " { /* 特別な強調 */
                yytext[yytext-1] = '\0';
                (void)check_and_convert(&yytext[1]);
                printf("@b{%s}",buffer);
            }

/* 特殊な Texinfo 文字をエスケープする */
<INITIAL,LITEM,LITEM2,BITEM,ENUM,EXAMPLE,EXAMPLE2>"@ " printf("@@");
<INITIAL,LITEM,LITEM2,BITEM,ENUM,EXAMPLE,EXAMPLE2> "{" printf("@{");
<INITIAL,LITEM,LITEM2,BITEM,ENUM,EXAMPLE,EXAMPLE2>"}" printf("@}");

/*
 * @example コードを再生成する
 */

": "\n+ ["^ \n0-9*]+ \n" " ["^ ] {
                int loop;
                int len;
                int cnt;

                printf(":\n\n@example \n");
                strcpy(buffer,yytext);
                len = strlen(buffer);
                cnt = 0;
                for(loop=len; loop > 0;loop--){
                    if(buffer[loop] == '\n')
                        cnt++;
                    if(cnt == 2)
                        break;
                }
                yyless(loop+1);
                statep++;
                states[statep] = EXAMPLE2;
                BEGIN(EXAMPLE2);
            }
<EXAMPLE,EXAMPLE2>^ \n {
                printf("@end example\n\n");
                statep--;
                BEGIN(states[statep]);
            }

```

```

/*
 * @enumerate リストを再生成する
 */

":\n+[ \t]*[0-9]+". " {
    int loop;
    int len;

    printf(":\n\n@enumerate \n");
    strcpy(buffer,yytext);
    len = strlen(buffer);
    for(loop=len; loop > 0;loop--){
        if(buffer[loop] == '\n')
            break;
    }
    yyless(loop);
    statep++;
    states[statep] = ENUM;
    BEGIN(ENUM);
}

<ENUM>"@"          printf("@@");
<ENUM>":\n+"        "[^0-9]    {
    printf(":\n\n@example\n");
    statep++;
    states[statep] = EXAMPLE;
    BEGIN(EXAMPLE);
}

<ENUM>\n[ \t]+[0-9]+". " {
    printf("\n\n@item ");
}

<ENUM>^[^ ] |
<ENUM>\n\n\n[ \t]+[^0-9] {
    printf("\n\n@end enumerate\n\n");
    statep--;
    BEGIN(states[statep]);
}

```

```

/*
 * 1 種類の@itemize リストを再生成する
 */

": "\n+": {
    int loop;
    int len;

    printf(":\n\n@itemize @bullet \n");
    yyless(2);
    statep++;
    states[statep] = LITEM2;
    BEGIN(LITEM2);
}

<LITEM2>^": ".+": {
    (void)check_and_convert(&yytext[1]);
    buffer[strlen(buffer)-1]='\0';
    printf("@item @b{%s:}\n",buffer);
}

<LITEM2>\n\n\n+[^:\n] {
    printf("\n\n@end itemize\n\n");
    ECHO;
    statep--;
    BEGIN(states[statep]);
}

/*
 * リビジョン・ヒストリ部からリストを作成する。
 * ここで"Version"が必要なのは、そうしないと他のルール
 * と衝突するからである
 */

:[\n]+"Version"[^:\n*]+": {
    int loop;
    int len;

    printf(":\n\n@itemize @bullet \n");
    strcpy(buffer,yytext);
    len = strlen(buffer);
    for(loop=len; loop > 0;loop--){
        if(buffer[loop] == '\n')
            break;
    }
}

```

```

        yyless(loop);
        statep++;
        states[statep] = LITEM;
        BEGIN(LITEM);
    }

<LITEM>^\.+;"    {
    (void)check_and_convert(yytext);
    buffer[strlen(buffer)-1]='\0';
    printf("@item @b{%s}\n\n",buffer);
}

<LITEM>^[^\n]+\n\n[^\n]+\n    {
    int loop;

    strcpy(buffer,yytext);
    for(loop=0; buffer[loop] != '\n'; loop++);
    buffer[loop] = '\0';
    printf("%s\n",buffer);
    printf("@end itemize\n\n");
    printf("%s",&buffer[loop+1]);
    statep--;
    BEGIN(states[statep]);
}

/*
 * @itemize @bullet リストを再生成する
 */

";"\n[ ]*"    {
    int loop;
    int len;

    printf(":\n\n@itemize @bullet \n");
    len = strlen(buffer);
    for(loop=0; loop < len;loop++){
        if(buffer[loop] == '\n')
            break;
    }
    yyless((len-loop)+2);
    statep++;
    states[statep] = BITEM;
    BEGIN(BITEM);
}

```

```

<BITEM>^" "*"*" {
    printf("@item");
    statep++;
    states[statep] = BITEM_ITEM;
    BEGIN(BITEM_ITEM);
}

<BITEM>"@"
<BITEM>^\n {
    printf("@end itemize\n\n");
    statep--;
    BEGIN(states[statep]);
}

<BITEM_ITEM>[^\:]* {
    printf(" @b{%s}\n\n",check_and_convert(yytext));
}

<BITEM_ITEM>": " {
    statep--;
    BEGIN(states[statep]);
}

/*
 * @chapter、@section 等を再作成する
 */

^:[^:]* {
    (void)check_and_convert(&yytext[1]);
    statep++;
    states[statep] = HEADING;
    BEGIN(HEADING);
}

<HEADING>:[^\n] {
    printf("@item @b{%s}\n",buffer);
    write_underline(strlen(buffer),6,'~');
    statep--;
    BEGIN(states[statep]);
}

```

```

<HEADING>:\n"*" {
    if(need_closing == TRUE){
        printf("@end table\n\n\n");
        need_closing = FALSE;
    }
    printf("@chapter %s\n",buffer);
    write_underline(strlen(buffer),9,'*');
    statep--;
    BEGIN(states[statep]);
}

<HEADING>:\n"="* {
    if(need_closing == TRUE){
        printf("@end table\n\n\n");
        need_closing = FALSE;
    }
    printf("@section %s\n",buffer);
    write_underline(strlen(buffer),9,'=');
    statep--;
    BEGIN(states[statep]);
}

<HEADING>"@" printf("@@");
<HEADING>:\n"-"* {
    if(need_closing == TRUE){
        printf("@end table\n\n\n");
        need_closing = FALSE;
    }
    printf("@subsection %s\n",buffer);
    write_underline(strlen(buffer),12,'-');
    statep--;
    BEGIN(states[statep]);
}

```

```

/*
 * @example テキストを再作成する
 */

~"      "      {
    printf("@example\n");
    statep++;
    states[statep] = EXAMPLE;
    BEGIN(EXAMPLE);
}
<EXAMPLE>~"    "
.              ECHO;

%%

/*
 * 初期化して実行する
 */

int main(int argc, char *argv[])
{
    states[0] = INITIAL;
    statep    = 0;
    print_header();
    yylex();
    print_trailer();
    return(0);
}

```

このプログラムは、ASCII の専門用語ファイルを読み込んで、いくつかのよく見られるパターンを検索します。このパターンは、オリジナルの Texinfo 形式の専門用語ファイルを単なる ASCII テキストに変換した際に作成されたものです。この変換の過程で、多くのマークアップ情報が失われているために、ある出力結果の元になったオリジナルの情報が何であったか、あるいは、そのオリジナルの候補が 2 つ 3 つあったとしても、そのうちのどれがその出力結果をもたらしたかを正確に決定することが困難であるという事情のため、この検索作業はいくらか複雑なものになります。よく見られるパターンをいくつか挙げると、以下のようになります。

章、節、項 これらの先頭にはいずれも同じパターンが来ます。

```
:some text:\n
```

この後ろに、(章の場合は) アスタリスクによる下線、(節の場合は) 等号による下線、(項の場合は) マイナス記号による下線が続きます。

強調 これは少し難しいのですが、一般的には強調は (イタリックの場合は) *...**、(強調文字 (strong) の場合は) **{...}**、(太字 (bold) の場

合は) { ... } の対によって示されます。ここでは、この 3 種類を検索して、コマンドを出力します。

実例、および列挙されたリスト

ともにコロンで始まり、その後ろに、1 つ以上の改行、少なくとも 5 つの空白、そして最後に数字もしくは何らかのテキストが続きます。例えば、列挙されたリストは以下のようになります。

```
...
    enumerated:

        0. some text
        1. some more text
```

また、実例は以下のようになります。¹

```
...
    example:

        some text
```

項目化されマークを付けられたリスト

実例、および列挙されたリストによく似ていますが、違いは、項目の先頭にコロン、またはアスタリスクがあり、末尾にコロンがあるという点です。

ここでの例は、パースされているものが何であることを示すヒントとしてこのようなパターンを使い、その特定のセクション用の部分的なパーサを（ほとんどの場合、排他的）スタート状態を使って作ります。ASCII 版の専門用語ファイルを持っているのであれば、スキャナのどの部分とそのファイル中の何にマッチするかを検証してみる値打ちがあります。例えば、HEADING 状態において @item を生成するルールが、すべての専門用語のエントリを処理するルールでもあるということは、おそらく一見だけでは明らかではないでしょう。

¹ 訳注： *some text* の部分に、インデントされたテキストが記されます。

8 Flex と Lex

ここで非常に簡単ではありますが、*Flex* と *Lex* の両方を概観してみます。*Flex*、*Lex* それぞれの性能と、*Lex* のようなユーティリティに関する *POSIX* 標準への準拠度についても、いくつか一般的なコメントを示します。

8.1 Flex

Flex は、*Lex* のより優れた再実装であり、*Lex* と同様、パターンとアクションの記述情報を入力として受け取って、そのパターンにマッチする能力を持つ C のスキャナに変換するものです。しかしながら、*Flex* はより少ない時間でテーブルを生成しますし、*Flex* により生成されるテーブルは、*Lex* により生成されるテーブルと比較して、はるかに効率的なものです。(*Flex* が正確には何を生成するのかという説明については、このマニュアルの冒頭で言及した書籍を参照してください。)

Flex は、*Lex* および *POSIX* と十分に互換性があり、それ独自の特別な機能もいくつか追加しています。

8.1.1 Flex と POSIX

Flex は、大体のところ *Lex* および *POSIX* の両方と互換性があります。将来は (*Flex*、*POSIX* のどちらかが変わるによって) さらに *POSIX* との互換性を高めていくでしょう。しかし、*Flex*、*Lex*、*POSIX* には、異なる部分もいくつかあります。それを以下に示します。

排他的スタート状態

Flex と *POSIX* は排他的スタート状態をサポートしていますが、*Lex* はサポートしていません。

定義

Lex と *Flex* では定義の展開の方法が違います。*Flex* (および *POSIX* のドラフト仕様) は、定義を展開する時に丸括弧 () で囲みますが、*Lex* は囲みません。¹ このことは、*Flex* 定義では演算子 '^'、'\$'、'/'、'<<EOF>>'、および '<start state>' は使うことができないということを意味しています。

このことがもたらす主要な問題の 1 つに、マッチの優先順位に影響を与え、*Flex* と *Lex* の間でスキャン処理に微妙な差異が出てくるといことがあります。この問題の例については、Section 3.6 [パターン・セクション], page 16 を参照してください。

input()

Flex および *POSIX* のドラフト仕様では、*input()* は再定義可能ではありません。*Flex* で入力を制御するためには、*input()* を再定義する代わりに、YY_INPUT という拡張機能を使います (これは現在のところ *POSIX* ではサポートされていません)。また、*Lex* とは異なり、*Flex* の *input()* は *yytext* の値を変更するという点に注意してください。

¹ 訳注 : *Flex* 2.5 では、'-1' オプションを指定して生成されたスキャナは、*Lex* の場合と同じように、定義を展開する時に丸括弧 () で囲みません。

`output()` Flex は `output()` ルーチンをサポートしていません。ECHO の出力は `yyout` 経由で行われます。この `yyout` のデフォルトは `stdout` です。これを使うように `output()` を書くことも可能ですが、現在の POSIX のドラフト仕様は、`output()` が正確には何をすべきなのかを示していません。

Ratfor スキャナ

Flex、POSIX のどちらも、Lex の Ratfor² スキャナ・オプション (`%r`) をサポートしていません。

`yylineno` これは、Flex や POSIX には存在しない、ドキュメント化されていない Lex の機能です。³ しかし、Flex で行数をカウントする機能を実装するのは難しくありません。定義中に行数カウント機能を組み込む方法の例については、See Section 9.5 [Miscellaneous], page 128 を参照してください。

`yywrap()` 現在のところ `yywrap()` はマクロです。POSIX のドラフト仕様では、これは関数であるべきとされていますので、おそらく将来は変更されることになるでしょう。⁴

`unput()` 現在のところ `unput()` は `yytext` と `yyleng` の値を破壊しますが、次のトークンがマッチされるまでは、これは不当です。Lex と POSIX では、`yytext` と `yyleng` は `unput()` の影響を受けません。⁵

数値範囲 POSIX によると、`'abc{1,3}'` は「`ab` の後ろに 1 個、2 個、または 3 個の `c` が続くもの」にマッチすべきとなっています。Flex はこのとおりに動きますが、Lex はこれを「1 個、2 個、または 3 個の `abc`」と解釈します。

`yytext` Flex において `yytext` の正しい定義は `'extern char *yytext'` ですが、Lex では `'extern char yytext[]'` です。⁶ 配列によるアクセス方法は、性能にかなりの影響を及ぼすので、Flex では `'extern char *yytext'` を使い続けるでしょう。

最新の POSIX ドラフト仕様は、`%array` と `%pointer` を導入することによって、両方の方法をサポートしています。これは、Flex と Lex のいずれにもまだ組み込まれていません。⁷

² 訳注：Rational Fortran

³ 訳注：Flex 2.5 では、Flex 起動時に `'-l'` オプションを指定するか、スキャナ定義ファイルの中に `'%option yylineno'` を指定することによって、変数 `yylineno` を利用することができます。

⁴ 訳注：Flex 2.5 では、`'%option noyywrap'` が指定されない限り、`yywrap()` は関数です。

⁵ 訳注：Flex 2.5 では、`%array` を指定すれば、`unput()` は `yytext` の内容を破壊しません。

⁶ 訳注：Flex 2.5 では、`%pointer` と `%array` により、`yytext` の型を選択できるようになりました。デフォルトは `%pointer` です。

⁷ 訳注：Flex 2.5 は、`%pointer` と `%array` をサポートしています。

テーブル・サイズ

Lex にはテーブル・サイズ宣言子 (%p、%a等) がありますが、Flex では必要ありません。互換性のために認識はされますが、無視されるだけです。

FLEX_SCANNER

スキャナが Flex と Lex のどちらにより生成されたかによって、コードをインクルードしたりしなかったりすることができるように、FLEX_SCANNERが#defineによって定義されています。

アクション Flex では、大括弧の対{...}を使うことなく、単一行において複数の文を置くことができます。これに対して Lex は、そのような行を単一行に切り詰めてしまいます。

コメント Flex ではコメントを '#' で始めることができますが、Lex と POSIX ではできません。ただし、この形式のコメントを使うことはお勧めできません。

yyterminate()、yyrestart()、<<EOF>>、YY_DECL、#line 指示子

これらはいずれも Lex ではサポートされていませんし、POSIX において明示的に定義されてもいません。#line指示子の説明に関しては、See Section 10.1 [Flex コマンドライン・オプションの要約], page 129。

8.1.2 Flex と POSIX (Flex 2.5 の補足情報)

Flex 2.5 でサポートされている新しい機能のうち、POSIX の仕様 (および、Lex) に存在しないものを以下に列挙します。

C++ スキャナ
 %option 指示子
 スタート状態スコープ
 スタート状態スタック
 yy_scan_string()、yy_scan_bytes()、yy_scan_buffer()
 yy_set_interactive()
 yy_set_bol()
 YY_AT_BOL()
 <*>
 YY_START

8.2 標準 Lex

Lex はスキャナを作成するための標準的な Unix ユーティリティであり、長い歴史を持っています。Lex は Flex と非常によく似ていますが、スキャナを生成するのにより多くの時間がかかりますし、Lex の生成するスキャナは Flex の生成するスキャナよりも通常は遅いものです。Lex は、特に多くの POSIX 機能を提供していないという理由から、置き換える必要が大いにあります。Flex はこうした POSIX 機能を提供しています。より多くのコンピュータ・システムが POSIX 互換になるにつれ

て、Flexの提供する多くの機能をサポートしなければならなくなり、このために、おそらくはFlexがLexの代わりにインストールされるようになるでしょう（例えば、4.4 BSD リリースはFlexを使うことになります）。しかし、Lexがインストールされている少数のシステムがあるために、しばらくの間はLexの存在は確実に維持されるでしょう。

FlexとLexの大きな違いは、Flexが性能を考慮して書かれたという点にあります。一般的には、Flexを持っているのであればそれを使うべきです。両者の性能差は、無視するにはあまりにも大きすぎます。しかし、移植性が最も重要なのであれば、スキャナ定義は可能な限りLexのものに近づけるべきです。というのは、Lexは事実上すべてのUnixマシンに入っていることが保証されていますが、Flexは入っていない可能性があるからです（しかし、Flexのインストールは通常は取るに足りない作業です）。このような場合に残念なのは、FlexとPOSIXが持っている排他的スタート状態のような、より便利な拡張機能を使うことができなくなるということです。

この問題を回避するためのもう1つの方法は、Flexでスキャナを作成して、作成されたスキャナを配布することです。スキャナというものは一度書かれるとほとんど変更されることがないので、多くの場合この方法は実行可能です。仮に変更が必要になったとしても、プログラムの他の部分も相当変更しなければならない可能性があり、よってプログラムを更新するための努力全体から見れば、Flexをインストールすることなどはほんの些細なものでしょう。

9 役に立つコードの抜粋

ここでは、読者がプログラムの中で使うことのできる、ちょっとした *Flex* 定義を一覧にして示します。多くは、このマニュアルを読んだあとでは、かなり自明のものになるはずです。しかし、読者がこうしたコードを最初から作らずに済むように、ここに入れてあります。

9.1 コメントの処理

Section 4.1 [Flex and C], page 43 において述べたように、コメントは `input()` を使って処理することができます。これを行うためのコードは以下のようになります。

```
%%
"/*" {
    int a,b;

    a = input();
    while(a != EOF){
        b = input();
        if(a == '*' && b == '/'){
            break;
        }else{
            a = b;
        }
    }
    if(a == EOF){
        error_message("EOF in comment");
    }
}
```

これは、Flex と Lex の両方で正当なコードです。コメントは排他的スタート状態を使って処理することも可能で、こちらの方がより美しく、より効率的です。スタート状態を使ってコメントを処理するコードは、以下のようになります。

```
%x COMMENT
%%
"/*"          BEGIN(COMMENT);
<COMMENT>[^\\n]
<COMMENT>\\n
<COMMENT><<EOF>>      yyerror("EOF in comment");
<COMMENT>"*/"        BEGIN(INITIAL);
```

改行の 1 つ前までと改行とを別々に処理した方が良いのは、そうしないと、内部のマッチ処理用のバッファをオーバーフローさせてしまうようなルールを作ることになってしまうからです。Lex は排他的スタート状態をサポートしていないので、このコードは Lex では動きません。この例は分かりやすいのですが、実際には単一文字

をマッチするのに多くの時間を無駄に消費するため、非効率的です。もっと長いテキストにマッチするように変更することで、スピードをかなり向上させることができます。例えば、以下のように書き直すことができます。

```
%x COMMENT
%%
"/*"                               BEGIN(COMMENT);
<COMMENT>[^\n]*
<COMMENT>[^\n]*\n
<COMMENT>"*"+[^\n]*      /* 余分な*を探す */
<COMMENT>"*"+[^\n]*\n
<COMMENT><<EOF>>          yyerror("EOF in comment");
<COMMENT>"*"+"/"        BEGIN(INITIAL);
```

これは、Flexと一緒に配布されている flexdoc.1 の中にある例とほとんど同一です。より長いテキスト・ブロックにマッチするため、はるかに高速ですし、ルールの中で改行のみにマッチさせる必要もありません。

9.2 文字列リテラルの処理

文字列は、それが入力として与えられた時に破棄されないという点で、コメントとは若干異なります。しかし、基本的なアプローチは同じです。第1の方法としては、input()を使って文字列を処理することができます。コードは以下のようになります。

```
/*
 * string1.lex: input() を使って文字列を処理する
 */

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

#define ALLOC_SIZE 32 /* バッファの(再)割り当て用 */

#define isodigit(x) ((x) >= '0' && (x) <= '7')
#define hextoint(x) (isdigit((x)) ? (x) - '0'\
                        : ((x) - 'A') + 10)
```



```
void yyerror(char *message)
{
    printf("\nError: %s\n",message);
}

%}

%%

\" {
    int  inch,count,max_size;
    char *buffer;
    int  temp;

    buffer  = malloc(ALLOC_SIZE);
    max_size = ALLOC_SIZE;
    inch    = input();
    count   = 0;
    while(inch != EOF && inch != '"' && inch != '\\n'){
        if(inch == '\\'){
            inch = input();
            switch(inch){
                case '\\n': inch = input(); break;
                case 'b' : inch = '\\b';    break;
                case 't' : inch = '\\t';    break;
                case 'n' : inch = '\\n';    break;
                case 'v' : inch = '\\v';    break;
                case 'f' : inch = '\\f';    break;
                case 'r' : inch = '\\r';    break;
                case 'X' :
                case 'x' : inch = input();
                    if(isxdigit(inch)){
                        temp = hextoint(toupper(inch));
                        inch = input();
                    }
            }
        }
        buffer[count] = inch;
        count++;
        inch = input();
    }
    buffer[count] = '\0';
    printf("%s\n",buffer);
}
```

```

        if(isxdigit(inch)){
            temp = (temp << 4) +
                hextoint(toupper(inch));
        } else {
            unput(inch);
        }
        inch = temp;
    } else {
        unput(inch);
        inch = 'x';
    }
    break;
default:
    if(isodigit(inch)){
        temp = inch - '0';
        inch = input();
        if(isodigit(inch)){
            temp = (temp << 3) + (inch - '0');
        } else {
            unput(inch);
            goto done;
        }
        inch = input();
        if(isodigit(inch)){
            temp = (temp << 3) + (inch - '0');
        } else {
            unput(inch);
        }
    done:
        inch = temp;
    }
}
buffer[count++] = inch;
if(count >= max_size){
    buffer = realloc(buffer,max_size + ALLOC_SIZE);
    max_size += ALLOC_SIZE;
}
inch = input();
}

```

```

    if(inch == EOF || inch == '\n'){
        yyerror("Unterminated string.");
    }
    buffer[count] = '\0';
    printf("String = \"%s\"\n",buffer);
    free(buffer);
}
.
\n
%%

```

このスキャナは、複数行にわたる文字列や、様々なエスケープ・シーケンスを処理します。また、文字列がどのような長さでも構わないように、動的バッファを使っています。これと同じことをスタート状態を使って行うコードは、以下のようになります。

```

/*
 * string2.lex: スタート状態を使って文字列をスキャンする例
 */

%{
#include <ctype.h>

#define isodigit(x) ((x) >= '0' && (x) <= '7')
#define hextoint(x) (isodigit((x)) ? (x) - '0' \
                    : ((x) - 'A') + 10)

char *buffer      = NULL;
int  buffer_size  = 0;

void yyerror(char *message)
{
    printf("\nError: %s\n",message);
}

%}

%x STRING

hex  (x|X)[0-9a-fA-F]{1,2}
oct  [0-7]{1,3}

```

```

%%

\"          {
    buffer      = malloc(1);
    buffer_size = 1; strcpy(buffer,"");
    BEGIN(String);
}

<String>\n  {
    yyerror("Unterminated string");
    free(buffer);
    BEGIN(Initial);
}

<String><<EOF>> {
    yyerror("EOF in string");
    free(buffer);
    BEGIN(Initial);
}

<String>[^\\n"] {
    buffer_size += yyleng;
    buffer = realloc(buffer,buffer_size+1);
    strcat(buffer,yytext);
}

<String>\\\n /* エスケープされた改行を無視する */
<String>\\{hex} {
    int temp =0, loop = 0, foo;
    for(loop=yyleng-2; loop>0; loop--){
        temp <= 4;
        foo    = toupper(yytext[yyleng-loop]);
        temp += hextoint(foo);
    }
    buffer = realloc(buffer,buffer_size+1);
    buffer[buffer_size-1] = temp;
    buffer[buffer_size]   = '\\0';
    buffer_size += 1;
}

```

```

<STRING>\\{oct} {
    int temp =0, loop = 0;
    for(loop=yytext[yytext[0]-1]; loop>0; loop--){
        temp <+= 3;
        temp += (yytext[yytext[0]-loop] - '0');
    }
    buffer = realloc(buffer,buffer_size+1);
    buffer[buffer_size-1] = temp;
    buffer[buffer_size] = '\\0';
    buffer_size += 1;
}
<STRING>\\[^\n] {
    buffer = realloc(buffer,buffer_size+1);
    switch(yytext[yytext[0]-1]){
        case 'b' : buffer[buffer_size-1] = '\\b';
                    break;
        case 't' : buffer[buffer_size-1] = '\\t';
                    break;
        case 'n' : buffer[buffer_size-1] = '\\n';
                    break;
        case 'v' : buffer[buffer_size-1] = '\\v';
                    break;
        case 'f' : buffer[buffer_size-1] = '\\f';
                    break;
        case 'r' : buffer[buffer_size-1] = '\\r';
                    break;
        default  : buffer[buffer_size-1] =
                    yytext[yytext[0]-1];
    }
    buffer[buffer_size] = '\\0';
    buffer_size += 1;
}
<STRING>\" {
    printf("string = \"%s\\\"",buffer);
    free(buffer);
    BEGIN(INITIAL);
}

%%

```

このスキャナは、string1.lexよりもモジュール化されていて、おそらくはより分かりやすいでしょう。エラーのルールは、INITIAL状態に戻るようになっていることに注意してください。こうしないと、スキャナは不当な文字列と正当な文字列とを結合してしまいます。ここでも、Flexのバッファ(YY_BUF_SIZE)が十分に大きいと

いうことをあてにせず、動的バッファを使いました。内部バッファが十分に大きいという確信が持てるのであれば、yytextだけを使うことも可能です。この場合には、yytextの右端が確実に最初の位置に留まるようにすることが重要です。より詳しい情報については、Section 4.1 [Flex and C], page 43 の yymoreの項を参照してください。

9.3 数字の処理

ここでは、Cに見られる様々な数値形式に対してよく使われる定義をいくつか示し、さらにその使い方の実例を1つ示します。注目すべき主要な点は、数の値を獲得するためにscanf()を使っている点と、オーバーフローが発生しないようlong型の値をスキャンするデフォルトのルールです。一般的には、yytextを数に変換する最良の方法は、sscanf()を使うことです。

```
/*
 * numbers.lex : 数をスキャンするための定義およびテクニックの実例
 */

%{
#include <stdio.h>

#define UNSIGNED_LONG_SYM    1
#define SIGNED_LONG_SYM     2
#define UNSIGNED_SYM        3
#define SIGNED_SYM          4
#define LONG_DOUBLE_SYM     5
#define FLOAT_SYM           6

union _yylval {
    long double    ylong_double;
    float          yfloat;
    unsigned long  yunsigned_long;
    unsigned       yunsigned;
    long           ysigned_long;
    int            ysigned;
} yylval;

%}

digit      [0-9]
hex_digit  [0-9a-fA-F]
oct_digit  [0-7]
```

```

exponent      [eE] [+]? {digit}+
i              {digit}+
float_constant ({i} \. {i}? | {i}? \. {i}) {exponent}?
hex_constant  0[xX] {hex_digit}+
oct_constant  0{oct_digit}*
int_constant  {digit}+
long_ext      [lL]
unsigned_ext   [uU]
float_ext      [fF]
ulong_ext     [lL] [uU] | [uU] [lL]

%%

{hex_constant}{ulong_ext} { /* 0xの部分をスキップする */
    sscanf(&yytext[2], "%lx",
           &yylval.unsigned_long);
    return(UNSIGNED_LONG_SYM);
}
{hex_constant}{long_ext} {
    sscanf(&yytext[2], "%lx",
           &yylval.signed_long);
    return(SIGNED_LONG_SYM);
}
{hex_constant}{unsigned_ext} {
    sscanf(&yytext[2], "%x",
           &yylval.unsigned);
    return(UNSIGNED_SYM);
}
{hex_constant} { /* オーバーフローを回避するために%lxを使う */
    sscanf(&yytext[2], "%lx",
           &yylval.signed_long);
    return(SIGNED_LONG_SYM);
}
{oct_constant}{ulong_ext} {
    sscanf(yytext, "%lo",
           &yylval.unsigned_long);
    return(UNSIGNED_LONG_SYM);
}

```

```

{oct_constant}{long_ext} {
    sscanf(yytext,"%lo",
           &yylval.ysigned_long);
    return(SIGNED_LONG_SYM);
}
{oct_constant}{unsigned_ext} {
    sscanf(yytext,"%o",
           &yylval.yunsigned);
    return(UNSIGNED_SYM);
}
{oct_constant} { /* オーバーフローを回避するために%loを使う */
    sscanf(yytext,"%lo",
           &yylval.ysigned_long);
    return(SIGNED_LONG_SYM);
}
{int_constant}{ulong_ext} {
    sscanf(yytext,"%ld",
           &yylval.yunsigned_long);
    return(UNSIGNED_LONG_SYM);
}
{int_constant}{long_ext} {
    sscanf(yytext,"%ld",
           &yylval.ysigned_long);
    return(SIGNED_LONG_SYM);
}
{int_constant}{unsigned_ext} {
    sscanf(yytext,"%d",
           &yylval.yunsigned);
    return(UNSIGNED_SYM);
}
{int_constant} { /* オーバーフローを回避するために%ldを使う */
    sscanf(yytext,"%ld",
           &yylval.ysigned_long);
    return(SIGNED_LONG_SYM);
}
{float_constant}{long_ext} {
    sscanf(yytext,"%lf",
           &yylval.ylong_double);
    return(LONG_DOUBLE_SYM);
}

```



```

{float_constant}{float_ext} {
    sscanf(yytext,"%f",
           &yylval.yfloat);
    return(FLOAT_SYM);
}
{float_constant} { /* オーバーフローを回避するために%lfを使う */
    sscanf(yytext,"%lf",
           &yylval.ylong_double);
    return(LONG_DOUBLE_SYM);
}

%%

int main(void)
{
    int code;

    while((code = yylex())){
        printf("yytext          : %s\n",yytext);
        switch(code){
            case UNSIGNED_LONG_SYM:
                printf("Type of number  : UNSIGNED LONG\n");
                printf("Value of number : %lu\n",
                      yylval.yunsigned_long);
                break;
            case SIGNED_LONG_SYM:
                printf("Type of number  : SIGNED LONG\n");
                printf("Value of number : %ld\n",
                      yylval.ysigned_long);
                break;
            case UNSIGNED_SYM:
                printf("Type of number  : UNSIGNED\n");
                printf("Value of number : %u\n",
                      yylval.yunsigned);
                break;
            case SIGNED_SYM:
                printf("Type of number  : SIGNED\n");
                printf("Value of number : %d\n",
                      yylval.ysigned);
                break;

```

```

    case LONG_DOUBLE_SYM:
        printf("Type of number  : LONG DOUBLE\n");
        printf("Value of number : %lf\n",
            yylval.ylong_double);
        break;
    case FLOAT_SYM:
        printf("Type of number  : FLOAT\n");
        printf("Value of number : %f\n",
            yylval.yfloat);
        break;
    default:
        printf("Type of number  : UNDEFINED\n");
        printf("Value of number : UNDEFINED\n");
        break;
}
}
return(0);
}

```

16 進定数については、変換する前に先頭の '0x' をスキップする必要がある点に注意してください。これは `sscanf()` の仕様です。

9.4 複数のスキャナ

時には、1つのプログラムの中で複数のスキャナを持つ必要がある場合がありますが、こうすると、2回以上現れる関数や変数について、リンカが文句を言ってきます。これを回避するためには、スキャナとそれに関連するすべてのものの名前を変更する必要があります。すべてのスキャナ関数、マクロ、およびデータの名前は 'yy' もしくは 'YY' で始まりますので、これはきわめて簡単です。しなければならないことは、名前の接頭辞を変更することだけです。これは `sed` を使って簡単に行うことができますが、ここではおもしろ半分で、これを行う flex スキャナを示しましょう。¹

¹ 訳注: Flex 2.5 では、Flex 起動時に '-Pprefix' オプションを指定するか、スキャナ定義ファイルの中に '%option prefix="prefix"' を指定することによって、接頭辞 'yy' を別の文字列に変更することができます。

```

/*
 * replace.lex : flex により生成されたスキャナや
 *               bison により生成されたパーサの
 *               一部の名前を変更する簡単なフィルタ
 */

%{
#include <stdio.h>

char lower_replace[1024];
char upper_replace[1024];

%}

%%

"yy"    fprintf(yyout,"%s",lower_replace);
"YY"    fprintf(yyout,"%s",upper_replace);

%%

int main(argc,argv)
int  argc;
char **argv;
{
    if(argc < 3){
        printf("Usage %s lower UPPER\n",argv[0]);
        exit(1);
    }
    strcpy(lower_replace,argv[1]);
    strcpy(upper_replace,argv[2]);
    yylex();
    return(0);
}

```

すべてのスキャナ関数の名前を変更するには、コマンドライン上で以下のように実行するだけです。

```
replace myscan_ MYSCAN_ < lex.yy.c > myscan.c
```

これにより、好きなだけ多くのスキャナを含めることができるようになります。ほとんど同じことを、排他的スタート状態と複数のバッファを使って実現することも可能ですが、その方法は多少複雑になります。

注：いくつかの *Flex* 内部ルーチンは、将来 *Flex* ライブラリ (‘-lf1’) の中に移されるでしょう。そうになると、このテクニックは機能しなくなります。しかし、この

変更が行われる時には、変更する必要のある関数名を変更する方法を、*Flex* 自身がサポートするようになるでしょう。²

9.5 その他

- 行数のカウント

行数をカウントしたいのであれば、ファイルの先頭のオプションの C コード・セクションに変数を定義して、改行をチェックします。スタート状態の中でも改行をチェックするのを忘れないようにしてください。さもないと、行数のカウントはうまくいきません。例えば、以下のようにします。

```
%{
int line_number = 0;
%}
%x COMMENT STRING
%%
"/*"          BEGIN(COMMENT);
<COMMENT>\n    line_number += 1;
<COMMENT>[^\n]*
<COMMENT>"*/"   BEGIN(INITIAL);
\"             BEGIN(STRING)
<STRING>\\\n    line_number += 1;
<STRING>[^\n\\"]*
<STRING>\\"     BEGIN(INITIAL);
\n            line_number += 1;
```

- スキャナとソケット

yyinとyyoutをリダイレクトすることによって、スキャナを(したがってBisonによるパーサをも)ソケットにアタッチすることができます。これは、fdopen()を呼び出すことによって行います。例えば、以下ようになります。

```
yyin  = fdopen(connection, "r");
yyout = fdopen(connection, "w");
```

ここで connection は、確立されたソケット・コネクションのファイル・ディスクリプタです。

² 訳注: Flex 2.5 では、'-Pprefix' オプションや '%option prefix="prefix"' を指定することにより、関数名を変更することができます。

10 要約

ここでは、*Flex*の使用に関連するすべての情報を要約します。この情報は、クイック・リファレンスとして使うことができます。

10.1 Flex コマンドライン・オプションの要約

Flex には、以下のコマンドライン・オプションがあります。

- b このオプションは、バックトラッキングを必要とする状態をもたらすルールに関する情報を含む、`'lex.backtrack'`というファイルを生成します。なぜこの情報が重要なのか、また、この情報をどのように使うかという点に関する詳細については、Section 6.1 [Optimizing for Speed], page 85 と Section 6.1.1 [Removing Backtracking], page 86 を参照してください。
- c このオプションは、POSIX との互換性のためだけに提供されており、実際には何もしません。以前は、テーブル圧縮を制御するために使われていましたが、その機能は `'-C'` オプションに移されました。このフラグを見つけると、Flex はユーザがテーブル圧縮を希望しているものと想定し、警告メッセージを出力します。将来、この警告メッセージは出力されないようになるかもしれません。¹
- d デバッグに使われます。実行中に自身の状態情報を `yyout` に書き込むスキャナを生成します。あるルールがマッチするたびに、バックトラッキングに関する情報、検出されたバッファの終端、NUL に関する情報に加えて、以下のような情報が書き込まれます。
 --accepting rule at line 行番号 ("マッチしたテキスト")
 この中の行番号は (`'-L'` オプションが使われていない場合には)、生成されたファイル `'lex.yy.c'` ではなく、スキャナを生成するのに使われた記述ファイルの行番号を指します。
- f フル・スキャナ (*full scanner*) を生成します。圧縮は一切行われません。これは、`'-Cf'` と同等です (詳細については、See Section 5.3 [Table Compression and Scanner Speed], page 77)。
- i 大文字・小文字の区別を無視するスキャナを作成するよう、Flex に通知します。ルールのマッチ処理において大文字・小文字の区別は無視されますが、個々の文字は大文字または小文字に変換されないで、`yytext` には大文字・小文字が混在した文字の並びが入ることになります。
- n このオプションは、Flex に対してはまったく意味を持たず、単に POSIX との互換性のためだけに提供されています。

¹ 訳注 : Flex 2.5 では、この警告メッセージは出力されません。

- p 性能に関する情報を `stderr` に書き込むよう、Flex に通知します。報告される情報は、性能を低下させるようなスキナ記述情報の機能に関するコメントによって構成されます。
- s マッチするものがなかった場合の Flex スキナのデフォルトのアクションは、マッチしなかった入力情報を `stdout` に書き込むことです。'-s' オプションはこのアクションを抑制し、その代わりに、入力がマッチしないとすぐにスキナを異常終了させます。
- t このオプションが指定されると、Flex は生成されたスキナをファイル '`lex.yy.c`' ではなく、`stdout` に書き込みます。
- v 冗長モードで動作するよう、Flex に通知します。Flex は、生成されたスキナに関する統計情報の要約を生成して、`stdout` に出力します。要約情報の第 1 行には Flex のバージョン番号、次の行には日付と時刻、さらに次の行には実際に使われているオプションが示されます。要約情報のこれ以外の部分は、Flex やその他の同様のプログラムの動作の詳細を理解している人以外にはほとんど意味を持ちません。
- F ファスト・スキナ (*fast scanner*) を生成するよう、Flex に通知します。これは、'-CF' と同等です。詳細については、See Chapter 6 [スキナの最適化], page 85。
- I このオプションは、シェル上や、型を持つ入力情報を受け付ける必要のあるプログラム内で使うことのできる対話型スキナを生成するよう、Flex に通知します。詳細については、See Section 5.2 [Interactive Scanners], page 76。
注：'-I' オプションは、'-Cf'、'-f'、'-CF'、'-F' の各オプションと一緒に使うことはできません。
- L デフォルトでは Flex は、エラーがスキナ定義のどこで発生したのかを追跡できるように、生成されたスキナのコード中に `#line` 指示子を生成します。'-L' オプションは、この `#line` 指示子を生成する機能を抑制します。
- T Flex をトレース・モードで実行させます。Flex は、入力情報、スキナ処理テーブル、同等クラス (*equivalence class*)、およびメタ同等クラス (*meta-equivalence class*) に関するメッセージを生成して、(`stderr` に) 書き込みます。この情報は、Flex の内部的な動作を理解していない人には、ほとんど意味を持たないでしょう。
- 8 このオプションは、8 ビットの入力情報を受け付けることのできるスキナを生成するよう、Flex に通知します。7 ビットの入力情報しか受け付けないスキナに 8 ビットの入力情報を与えた場合の結果は、予測不能です。

- C[efmF] スキャン処理テーブルをどのように圧縮するかを、Flexに通知します。詳細については、See Chapter 6 [スキヤナの最適化], page 85 を参照してください。
- Skeleton_file
生成されるスキヤナのベースとして、*skeleton_file* で指定されるファイルを使うよう、Flexに通知します。これを使うことはほとんどありませんが、MS-DOS 上ではこれによって標準のスキヤナ・スケルトンへのパスを設定することができます。

10.2 Flex コマンドライン・オプションの要約 (Flex 2.5 の補足情報)

Flex 2.5 では、前節 (Section 10.1 [Switches Summary], page 129) で説明されていない、以下のオプションもサポートされています。

- h Flex に対してコマンドライン・オプションの要約情報を出力するよう指示します。
- l AT&T により実装された lex との互換性を最大限に提供します。このオプションは、性能面でかなりの悪影響を及ぼします。また、このオプションを、'-f'、'-F'、'-Cf'、'-CF'、'-+' オプションと同時に指定することはできません。
- w このオプションが指定されると、Flex は、警告メッセージを出力しません。
- B Flex に対してバッチ・スキヤナを生成するよう指示します。これは、対話型スキヤナを生成するよう指示する '-I' オプションの否定です。
- V Flex に対してバージョン番号を出力するよう指示します。
- 7 Flex に対して 7 ビット・スキヤナを生成するよう指示します。これは、'-8' オプションの否定です。
- + Flex に対して C++ スキヤナ・クラスを生成するよう指示します。
- ? Flex に対してコマンドライン・オプションの要約情報を出力するよう指示します。 ('-h' オプションと同じです)。
- Ca このオプションは、スキャン処理用のテーブルを long int の配列として定義するよう Flex に通知します (デフォルトでは short int 型の配列となります)。
- Cr このオプションを指定して生成されたスキヤナは、入力に read() システム・コールを使います。デフォルトでは、対話型スキヤナの場合は getc() が、バッチ (非対話型) スキヤナの場合は fread() が使われます。

<code>-ofile</code>	このオプションが指定されると、Flexは生成されたスキナを <i>file</i> により指定されるファイルに出力します。デフォルトでは、スキナはファイル <code>'lex.yy.c'</code> に出力されます。
<code>-Pprefix</code>	Flexにより生成されるスキナのソース・ファイルの中では、大域変数や大域関数の名前の先頭に接頭辞 <code>'yy'</code> が付けられます。このオプションが指定されると、 <code>'yy'</code> の代わりに、 <i>prefix</i> により指定される文字列が接頭辞として使用されます。また、 <code>'-o'</code> オプションが指定されない場合のスキナ・ファイル名 <code>'lex.yy.c'</code> も、 <code>'lex.prefix.c'</code> となります。
<code>--help</code>	Flex に対してコマンドライン・オプションの要約情報を出力するよう指示します。(<code>'-h'</code> オプションと同じです)
<code>--version</code>	Flex に対してバージョン番号を出力するよう指示します。(<code>'-V'</code> オプションと同じです)

10.3 Flex 変数および Flex 関数の要約

Flex に対する主要な C インターフェイスは、以下のルーチンおよび変数を通じて実現されます。個々のルーチン、変数に関する完全な説明については、See Chapter 4 [Interfacing to Flex], page 43。

<code>yylex()</code>	主要なインターフェイスです。これが実際のスキャン処理を行う関数です。
<code>yyin</code>	<code>yylex()</code> が文字を読み込む元となるファイルです。このデフォルトは <code>stdin</code> です。
<code>yyout</code>	スキナの出力ファイルです。デフォルトは <code>stdout</code> です。
<code>yytext</code>	最後にマッチした文字列を保持する大域変数です。つまり、最後に認識されたトークンを保持しています。
<code>yylen</code>	最後に認識されたトークンの長さを保持する大域変数です。
<code>yywrap()</code>	この関数は、 <code>yyin</code> の終端に達した時に呼び出されます。これが <code>TRUE</code> (ゼロ以外) を返すとスキナは実行を終了しますが、 <code>FALSE</code> (ゼロ) を返すと、 <code>yyin</code> が次の入力ファイルを指すよう設定されたものと想定し、スキャン処理は継続されます。
<code>yyomore()</code>	次に認識されるトークンで <code>yytext</code> の内容を上書きするのではなく、そのトークンを <code>yytext</code> の末尾に付加するよう Flex に通知する関数です。
<code>yyless(<i>n</i>)</code>	<code>yyomore()</code> とほぼ反対のを行います。この関数は、最初の <i>n</i> 文字を除くすべての文字を戻します。戻された文字の並びは、次のトークンをマッチするのに使われます。 <code>yylen</code> と <code>yytext</code> の内容には、この変更が反映されます。

<code>input()</code>	入力から次の1文字を返します。これは、標準のFlex記述言語や特にLex記述言語を使ったのではうまく処理できないようなスキナにおいて、よく使われます。
<code>unput(<i>c</i>)</code>	この関数は、文字 <i>c</i> を入力ストリームに戻します。この後、この文字は次にスキャンされる文字になります。
<code>yyterminate()</code>	この関数は、アクションの中で使われると、スキナ(<code>yylex()</code>)の実行を終了させます。終了したスキナは0を返します。この後 <code>yyrestart()</code> が呼び出されない間は、 <code>yylex()</code> を呼び出してもすぐに復帰してしまいます。
<code>yyrestart(<i>file</i>)</code>	この関数は、スキナの実行を再開するようFlexに通知します。これは、スキャンすべきファイル(通常は <code>yyin</code>)を表す引数を1つ取ります。EOFを処理するのに使うことができますし、また、Flexに割り込みをかけ、その後で再開することを可能にするために使うこともできます。(Flexが再入可能ではないので、このようなことが必要になります。)
ECHO	<code>yytext</code> の内容を <code>yyout</code> にコピーするマクロです。
REJECT	カレントなトークンを認識しないで、次に最もよくマッチするものを選択するよう、スキナに通知します。スキナは、マッチするもののうち最も長いものを探します。マッチするものが2つあってその長さが同じ場合には、スキナ記述の中で最初に定義されているものを選択します。
<code>BEGIN(<i>state</i>)</code>	スキナをある特定のスタート状態に置くために使われます。 <code>BEGIN</code> の後ろの名前は、スタート状態の名前です。これは、スキナ記述の先頭の定義セクションにおいて宣言されているものでなければなりません。
<code>YY_USER_INIT</code>	スキナが初期化される前に実行されるべきアクションを定義します。詳細については、Section 4.1 [Flex and C], page 43 を参照してください。
<code>YY_USER_ACTION</code>	マッチが発生した後で、ルール・セクションに定義されたアクションが実行される前に、実行されるべきアクションを定義します。例えば、 <code>yytext</code> の内容を小文字から大文字へ変換する等を行うのに使うことができます。デフォルトのルールでは何も実行されません。詳細については、Section 4.1 [Flex and C], page 43 を参照してください。
<code>YY_BREAK</code>	実際にはインターフェイス機能ではなく、むしろ生成されるコードを変更するために使うことができるものです。

スキャナの中では、すべてのアクションは1つの大きな switch文の構成要素であり、個々のアクションの区切りは、デフォルトで 'break;' 文に変換される YY_BREAK で与えられます。もし、ほとんどのルールのアクション部が 'return;' 文を含んでいると、コンパイラは 'statement not reached' というエラーをたくさん表示することになるでしょう（表示するはずです）。YY_BREAK を再定義することによって、この警告メッセージを表示させないようにすることが可能です。

注：YY_BREAK を再定義する場合は、アクションが必ず 'return;' か 'break;' で終わるようにしてください。

- YY_DECL スキャン処理を実行する関数の名前を定義するマクロです。デフォルトは `yylex` ですが、再定義することができます。再定義した名前は、関数のプロトタイプとして正当なものでなければなりません。
- YY_INPUT 入力ルーチンの名前を定義するマクロです。必要があれば、この名前は再定義することができます。例えば、文字列や、標準的ではない何らかの入力デバイスを入力として、スキャン処理を行う場合に役に立ちます。
- YY_NEW_FILE `yyin` が新しいファイルを指すよう設定されたこと、および、処理が継続されるべきであることを Flex に通知するマクロです。²
- YY_CURRENT_BUFFER カレントな入力バッファを返すマクロです。
- `yy_create_buffer()` 新しい入力バッファを作成するのに使われます。この関数と、この後の2つの関数を使うことにより、複数のバッファを作成し、バッファ間で切り替えることが可能になります。See Section 5.5.1 [バッファを操作する関数], page 79。
- `yy_delete_buffer()` 以前に作成された入力バッファを削除するのに使われます。
- `yy_switch_to_buffer()` 複数の入力バッファの間で切り替えを行うのに使われます。
- YY_BUFFER_STATE バッファを処理するのに使われる型です。バッファのカレントなコンテキストを保持します。複数のバッファ間で切り替えを行う時には、この型の変数が必要になります。
- YYSTYPE Bison ファイル中の %union の型です。これは、Flex と Bison の間のインターフェイスで使われます。

² 訳注：Flex 2.5 では、`yyin` を変更した後に YY_NEW_FILE を実行する必要はなくなりました。

`yyval` Bison パーサのカレントなパース状態に関連するデータを保持する、Bison パーサ中の変数です。この変数を使うことで、データを Flex と Bison の間で渡すことができます。

10.4 Flex 変数および Flex 関数の要約 (Flex 2.5 の補足情報)

Flex 2.5 では、前節 (Section 10.3 [Summary of Flex Variables and Functions], page 132) で説明されていない、以下の関数やマクロもサポートされています。

`yy_set_interactive()`
カレント・バッファを、対話的なものと見なすか、非対話的なものと見なすかを制御します。引数にゼロ以外の値を渡すと、カレント・バッファは対話的なものと見なされ、ゼロを渡すと、非対話的なものと見なされます。

`yy_set_bol()`
バッファ内のカレントな位置が行の先頭にあるか否かを表すコンテキスト情報を設定します。引数にゼロ以外の値を渡すと、バッファ内のカレントな位置は行の先頭である、というコンテキスト情報がセットされます。したがって、次にトークンのマッチ処理が行われる時には、行頭を表す '^' を含むルールの適用が試みられます。逆に、引数にゼロを渡すと、バッファ内のカレントな位置は行の先頭ではないことになり、次にトークンのマッチ処理が行われる時には、行頭を表す '^' を含むルールの適用が試みられなくなります。

`YY_AT_BOL()`
次にトークンのマッチ処理が行われる時に、行頭を表す '^' を含むルールの適用が試みられるようなコンテキスト情報がセットされている場合には、ゼロ以外の値を返します。それ以外の場合は、ゼロを返します。

`yy_new_buffer()`
`yy_create_buffer` の別名です。

`yy_flush_buffer()`
引数で指定されたバッファの内容を破棄し、バッファの先頭 2 バイトに `YY_END_OF_BUFFER_CHAR ('\0')` をセットします。

`YY_FLUSH_BUFFER`
引数にカレント・バッファを指定して `yy_flush_buffer()` を呼び出すよう定義されたマクロです。

`yy_scan_string()`
NULL 文字で終端する文字列をスキャンするための入力バッファを作成します。実際には、引数で渡された文字列のコピーがスキャンされます。

`yy_scan_bytes()`
引数で指定されたメモリ領域をスキャンするためのバッファを作成します。実際には、メモリ領域上のデータのコピーがスキャンされます。

`yy_scan_buffer()`
引数で指定されたメモリ領域をスキャンするためのバッファを作成します。メモリ領域上のデータはコピーされません。

`yy_push_state()`
カレントなスタート状態をスタート状態スタックにプッシュし、引数で指定された状態に遷移します。

`yy_pop_state()`
スタート状態スタックからスタート状態をポップし、そのポップされたスタート状態に遷移します。

`yy_top_state()`
スタート状態スタックの先頭にあるスタート状態を返します（スタート状態スタックの内容は変更されません）。

`yyFlexLexer::yylex()`
C++ スキャナにおいて実際にスキャン処理を行う関数です。

`yyFlexLexer::LexerInput()`
`yyFlexLexer` のサブクラスにおいて再定義することによって、C++ スキャナの入力処理を変更することができます。

`yyFlexLexer::LexerOutput()`
`yyFlexLexer` のサブクラスにおいて再定義することによって、C++ スキャナの出力処理を変更することができます。

`yyFlexLexer::LexerError()`
`yyFlexLexer` のサブクラスにおいて再定義することによって、C++ スキャナのエラー・メッセージ出力処理を変更することができます。

10.5 Flex 文字の要約

Flex における基本的な構成要素の 1 つに、文字があります。基本的に Flex は、演算子、特殊文字、エスケープ・シーケンスを除いて、文字をそのまま受け付けます。エスケープ・シーケンスは、ANSI C に見られるものと同一です。Flex の演算子と特殊文字は以下のとおりです。

文字	Flex の解釈
\	バックスラッシュは、ANSI C のエスケープ・シーケンスで使われるのと同様の、エスケープ文字です。

- [] 角括弧 [] は、文字の集合を文字クラスにグループ化するのに使われます。詳細については、See Section 3.6.3 [Flex における文字のグループ化], page 21。
- ^ 文字クラスの中では、'^'は否定を意味します。詳細については、See Section 3.6.3 [Flex における文字のグループ化], page 21。一方、文字クラスの外部では、行の先頭を意味し、(エスケープされていない場合は) ルールの先頭にのみ置くことができます。
- ハイフンは、文字クラスの中で文字の範囲を設定するのに使われます。文字クラスの外部では、ハイフン自身を表します。詳細については、See Section 3.6.3 [Flex における文字のグループ化], page 21。
- { } 大括弧 { } は、定義の参照、複数行にわたるアクションの先頭と末尾の指定、またはパターンの繰り返し回数の範囲の定義を行います。
- () 丸括弧 () は、優先順位の変更に使われます。また、定義は展開される時に、暗黙のうちに丸括弧で囲まれることに注意してください。
- "" 二重引用符は、文字列の範囲を示します。引用符で囲まれた範囲の中にある文字だけがマッチされます。
- / スラッシュは、後続コンテキスト (trailing context) を設定します。これは、あるパターンを認識するのを、その後ろに別のパターンが続く場合に限定したい、という場合です。これは、スラッシュ '/' が一種の「ルック・アヘッド (その先を見る)」演算子として機能することを意味します。
- < > かぎ括弧 < > は、スタート状態の参照、またはスタート状態のグループの参照を行い、さらに EOF シンボル ('<<EOF>>') で使われます。これに関する完全な説明については、Section 3.8 [Start States], page 24 と Section 5.6 [End-Of-File Rules], page 83 を参照してください。
- ? + * '?', '+', '*' の各文字は、ある正規表現が何回出現することができるかを指定するのに使われます。 '?' は、ゼロ回もしくは 1 回 (つまり、オプションであるということ) を、 '+' は 1 回以上を、 '*' はゼロ回以上をそれぞれ意味します。
- | OR 演算子を表します。また、カレントなルールに対するマッチが発生した場合、次に記述されているルールのアクションを実行するよう Flex に通知する、特別なアクションを表します。
- \$ ドル記号は行末を意味します。

ここに挙げた文字を、その文字自身として表したい場合には、その文字を引用符で囲む (例えば "*") か、または、エスケープ・シーケンスとして表す必要があります。詳細については、See Section 3.6.1 [Characters], page 17。

10.6 Flex ルールの要約

Flex におけるルールには 2 つの部分があります。パターン・マッチング用の表現式とアクション部です。この 2 つは、以下のように配置されます。

pattern actions

Flex がマッチするパターンは、正規表現を使って作られます。そしてその正規表現は、文字、文字列、定義、スタート状態、および演算子から作られます。下の表は、種々の正当な正規表現を示します。表中において、`'c'` は (エスケープ・シーケンスを含む) 任意の単一文字を、`'r'` は任意の正規表現を、`'s'` は文字列を、それぞれ表します。表はグループ別に編成されていて、優先度の最も高いものが一番上にあります。

Flex における正規表現

正規表現	マッチの対象	例
<code>c</code>	特殊文字を除く任意の文字	<code>A</code> または <code>\n</code>
<code>.</code>	改行 (<code>\n</code>) を除く任意の文字	<code>efg.*</code>
<code>[s]</code>	クラス <code>s</code> 中にある任意の文字	<code>[efg]</code>
<code>[^s]</code>	クラス <code>s</code> 中不在任意の文字	<code>[^moqs]</code>
<code>r*</code>	0 個以上の <code>r</code>	<code>(a [e-f])*</code>
<code>r+</code>	1 個以上の <code>r</code>	<code>(a [e-f])+</code>
<code>r?</code>	0 個または 1 個の <code>r</code>	<code>(a [e-f])?</code>
<code>r{x,y}</code>	<code>x</code> 個以上 <code>y</code> 個以下の <code>r</code> (<code>abc{1,3}</code> は、 <code>ab</code> と 1 個以上 3 個以下の <code>c</code>)	<code>foo{1,5}</code>
<code>"s"</code>	字義どおりの文字列 <code>s</code>	<code>"****"</code>
<code>\c</code>	<code>c</code> (<code>\c</code> が ANSI C において特別な意味を持たない場合)	<code>\</code> または <code>*</code>
<code>(r)</code>	<code>r</code> - 丸括弧 <code>()</code> はグループ化のためのもの	<code>(Ab Bb)</code>
<code>r1r2</code>	<code>r1</code> の後ろに <code>r2</code> が続くもの	<code>Aa</code>
<code>r1 r2</code>	<code>r1</code> または <code>r2</code>	<code>A B</code>
<code>r1/r2</code>	<code>r2</code> が後ろに続くという条件を満足する <code>r1</code>	<code>abc/123</code>
<code>^</code>	行頭	<code>^foo</code>
<code>\$</code>	行末	<code>foo\$</code>
<code><start>r</code>	スタート状態: <code>start</code> 状態の時、 <code>r</code> がアクティブ	<code><comment>"*/"</code>
<code><<EOF>></code>	ファイルの終端 (End-Of-File ルールを参照)	<code><<EOF>></code>

これは、`sed`、`grep`、`Emacs` や正規表現を使う他の一般的なプログラムにおいて使われる正規表現と完全に同一ではないことに注意してください。

ルールのアクション部は、任意の正当な C コードです。単一行に複数の文を書くことも可能ですし、括弧の対 `{...}` で囲むことで、複数の文のブロックを複数行にわたって書くことも可能です。

インデックス	139
インデックス	
例、スタート状態..... 31	対話型スキャナ 76
名前、不当..... 94	状態の振る舞いの設定 24
	条件スキャン処理 24
翻訳テーブル..... 79	序 1
文字クラス..... 21	謝辞..... 1
文字クラス式..... 22	
文字のグループ化..... 21	字句スキャナ..... 5
複数バッファを使う実例..... 81	
不当な名前..... 94	最適化..... 85
標準出力..... 45	再入可能性とスキャナ 133
	再帰..... 47, 122
排他的スタート状態..... 27	行数のカウント 94
	行数、カウント 94
入力バッファ..... 79	型付けされた入力 76
入力バッファ、複数の使用..... 81	型付けされた入力のスキャン処理 76
入力ストリームへのテキストの追加.. 49	
入力ストリームへのアクセス..... 48	記述、スキャナ 13
入力の変更..... 79	
日付のスキャン処理..... 31	関数と変数、要約..... 132
日付、スキャン処理..... 31	
同等クラス..... 78	ルール..... 15
定義..... 14	ルール定義 15
定義ファイル中のコメント..... 13	ルールを使用した定義 14
定義、コメント..... 13	ユーザ定義の初期化..... 55
大文字・小文字の無視..... 75	

ユーザ定義のアクション	54	スキヤナの再入可能性、yyrestart()	51
メタ同等クラス	78	コード・ブロック	15
プログラムの実例について	2	エスケープ・シーケンス	20
プログラム、実例	2	コマンドライン・オプション (Flex 2.5 補足)	11
パターンのマッチング	16		
パターン・マッチの拒絶	52	定義への C コードの追加	13
パターン、マッチング	16		
パース言語	65	特殊な Flex 機能	75
バッファの操作	79		
バッファ、複数	79	記述、Flex	13
デフォルトの振る舞い、スキヤナ	60		
テキスト長	46	#	
テキストの末尾への追加	46	#line 指示子、制御	130
テキストの返却	47	#line 指示子と Lex	113
テーブルの圧縮	77	%	
スピードとテーブル・サイズ	85	%array	39
スタート状態	24	%option	37
スタート状態スタック	30	%pointer	39
スタート状態スコープ	29	%s、詳細	25
スタート状態の宣言	24	%x、詳細	25
スタート状態の使用	25	%x 複数行文字列を使用したサンプル	119
スタート状態、排他的	27	%x より長いテキストにマッチするサン プル	116
スタート状態、使用	31	%x EOF を使用したサンプル	115
スタート状態、活性化	26	-	
スキャン処理のスピード	77	--help	132
スキャンされたテキスト	45	--version	132
スキャンされたテキストへのアクセス	45	-?	131
スキヤナの生成結果のリダイレクト、 ‘-t’	130	-+	131
スキヤナの振る舞いの動的な変更	24	-7	131
スキヤナの最適化	85	-8	130
スキヤナのデフォルト・アクション ..	16	-b	129
スキヤナ、対話型	76	-B	131
コマンドライン・オプション	129	-c	129
エスケープ・シーケンスの取り扱い	116	-C[efmF]	130
エスケープ・シーケンス、スキャン処理	116		
スキャンされたテキストの表示、ECHO	52		
コンテキスト依存スキャン処理	24		
コマンドライン・オプション	9		

-Ca	131
'-Ce'、詳細	78
-Cem、詳細	78
-Cf、詳細	78
-CF、詳細	78
-Cm、詳細	78
-Cr	131
-d	129
-f	129
'-f'、詳細	77
-F	130
'-F'、詳細	77
-h	131
-i	129
-i、詳細	75
-I	130
-I、詳細	76
-l	131
-L	130
-n	129
-o	131
-p	129
-P	132
-s	130
-S	131
-t	130
-T	130
-v	130
-V	131
-w	131
..	
.l、lex ファイル	9

<

<*>	28
<<EOF>>、詳細	83
<<EOF>>と Lex	113

1

16 進定数	119
--------------	-----

7

7bit、%option	37
--------------------	----

8

8 進定数	119
8bit、%option	37

A

align、%option	37
always-interactive、%option	39
array、%option	39

B

backup、%option	37
batch、%option	37
BEGIN、要約	133
BEGIN、詳細	54
BEGIN、スタート状態の活性化	26
Bison	61
Bison 文法の例	65

C

C コメントのスキャン処理	115
C コメント、スキャン処理	115
C コード、追加	13
C とのインターフェイス	13
C と Flex のインターフェイス	43
C と Flex のインターフェイス (Flex2.5 補足)	58
c++、%option	37
C++と Flex のインターフェイス	71
case-insensitive、%option	38
case-sensitive、%option	37
caseful、%option	37
caseless、%option	38

D

debug、%option	38
debug()、FlexLexer	72
default、%option	38

E

ECHO	133
ECHO、サンプル	77
ECHOと output()	111
ecs、%option	38

End-Of-File、取り扱い..... 83

F

fast、%option..... 38
 Flex 文字..... 17
 Flex 文字列..... 21
 Flex 文字クラス..... 21
 Flex 文字、要約..... 136
 Flex 入門..... 5
 Flex 正規表現..... 138
 Flex 記述..... 13
 Flex ルール..... 16
 Flex ルール、要約..... 138
 Flex データ型..... 16
 Flex コマンドライン・オプション.. 129
 Flex の使用..... 9
 Flex の起動..... 9
 Flex における文字..... 17
 Flex における文字列..... 21
 Flex における文字、要約..... 136
 Flex における正規表現..... 138
 Flex コマンドライン・オプション (Flex 2.5 補足)..... 131
 Flex におけるデータ型..... 16
 Flex からのデータの返却..... 61
 Flex、入門..... 5
 Flex、使用法..... 9
 Flex、起動..... 9
 Flex、パターン・マッチング・ルール..... 15
 Flex 関数と Flex 変数..... 43
 Flex と Bison のインターフェイス.... 61
 Flex と C のインターフェイス..... 132
 Flex と C のインターフェイス (Flex2.5 補足)..... 135
 Flex と Lex の相違点..... 111
 Flex と Lex、相違点..... 111
 FlexLexer..... 71
 front.lex..... 69
 front.y..... 67
 full、%option..... 38

I

input()、要約..... 132

input()、詳細..... 48
 input()、サンプル..... 115
 input()、Lex と Flex の相違点..... 111
 interactive、%option..... 38

L

Lex..... 113
 lex に対する C のインターフェイス.. 43
 lex-compat、%option..... 38
 lex.yy.c..... 9
 lex.yy.cc..... 71
 LexerError()、yyFlexLexer..... 73
 LexerInput()、yyFlexLexer..... 73
 LexerOutput()、yyFlexLexer..... 73
 lineno()、FlexLexer..... 72

M

main、%option..... 39
 meta-ecs、%option..... 38

N

never-interactive、%option..... 39

O

output、%option..... 38
 output()..... 111

P

Pascal コメントのスキャン処理..... 94
 Pascal コメント、スキャン処理..... 94
 perf-report、%option..... 38
 pointer、%option..... 39
 POSIX..... 111
 prefix、%option..... 38

R

Ratfor スキャナ..... 112
 read、%option..... 38
 REJECT、詳細..... 52
 reject、%option..... 39

S

set_debug()、FlexLexer..... 72

stack、%option	39	yy_pop_state()、要約	136
stdinit、%option	39	yy_pop_state()、詳細	30
stdout、%option	38	yy_push_state、%option	40
switch_streams()、FlexLexer	72	yy_push_state()、要約	136
		yy_push_state()、詳細	30
		yy_scan_buffer、%option	40
		yy_scan_buffer()、要約	136
		yy_scan_buffer()、詳細	81
		yy_scan_bytes、%option	40
		yy_scan_bytes()、要約	135
		yy_scan_bytes()、詳細	80
		yy_scan_string、%option	40
		yy_scan_string()、要約	135
		yy_scan_string()、詳細	80
		yy_set_bol()、要約	135
		yy_set_bol()、詳細	58
		yy_set_interactive()、要約	135
		yy_set_interactive()、詳細	58
		YY_START	29
		yy_switch_to_buffer()、要約	134
		yy_switch_to_buffer()、詳細	80
		yy_switch_to_buffer()、FlexLexer ..	72
		yy_top_state、%option	40
		yy_top_state()、要約	136
		yy_top_state()、詳細	30
		YY_USER_ACTION、要約	133
		YY_USER_ACTION、詳細	54
		YY_USER_INIT、要約	133
		YY_USER_INIT、詳細	55
		YYBREAKとともに使用される break ..	57
		yyclass、%option	40
		yyFlexLexer	73
		yyFlexLexer::LexerError()、要約 ..	136
		yyFlexLexer::LexerError()、詳細 ..	73
		yyFlexLexer::LexerInput()、要約 ..	136
		yyFlexLexer::LexerInput()、詳細 ..	73
		yyFlexLexer::LexerOutput()、要約 ..	136
		yyFlexLexer::LexerOutput()、詳細 ..	73
		yyFlexLexer::yylex()、要約	136
		yyFlexLexer::yylex()、詳細	73
		yyin	45
		yyinとソケット	128
		yyin、要約	132
U			
unput、%option	40		
unput()、要約	133		
unput()、詳細	49		
V			
verbose、%option	38		
W			
warn、%option	38		
Y			
Yacc	61		
YY_AT_BOL()、要約	135		
YY_AT_BOL()、詳細	58		
YY_BREAK、要約	133		
YY_BREAK、詳細	57		
YY_BUF_SIZE	80		
YY_BUFFER_STATE	134		
yy_create_buffer()	79		
yy_create_buffer()、FlexLexer	72		
YY_CURRENT_BUFFER	80		
YY_DECL、要約	134		
YY_DECL、詳細	44		
yy_delete_buffer()	80		
yy_delete_buffer()、FlexLexer	72		
yy_flex_debug、FlexLexer	72		
YY_FLUSH_BUFFER、要約	135		
YY_FLUSH_BUFFER、詳細	80		
yy_flush_buffer()、要約	135		
yy_flush_buffer()、詳細	80		
YY_INPUTの再定義	45		
YY_INPUT、要約	134		
YY_INPUT、再定義	45		
yy_new_buffer()、要約	135		
yy_new_buffer()、詳細	80		
YY_NEW_FILE、要約	134		
YY_NEW_FILE、詳細	51		
yy_pop_state、%option	40		

yyin、リセットの例.....	91	yyrestart()、要約.....	133
yy leng.....	46	yyrestart()、詳細.....	51
yy leng、要約.....	132	yyrestart()、FlexLexer.....	72
yy leng、FlexLexer.....	71	yyrestart()と Lex.....	113
yy leng、unput().....	112	YYSTYPE.....	71
YYLeng()、FlexLexer.....	72	YYSTYPE、Bison ファイルにおける%union 型.....	134
yyless()、要約.....	132	YYSTYPE、Bison との関連.....	65
yyless()、詳細.....	47	yyterminate()、要約.....	133
yy lex()の再定義.....	44	yyterminate()、詳細.....	51
yy lex()、要約.....	132	yyterminate()と Lex.....	113
yy lex()、詳細.....	43	yytext非互換性.....	112
yy lex()、再定義.....	44	yytextの長さへのアクセス.....	46
yy lex()、FlexLexer.....	72	yytext、要約.....	132
yy lex()と yyterminate().....	51	yytext、表示.....	52
yylineno.....	112	yytext、詳細.....	45
yylineno、%option.....	40	yytext、FlexLexer.....	71
yylineno、FlexLexer.....	72	yytext、unput().....	112
yy lval、要約.....	134	YYText()、FlexLexer.....	72
yy lval、説明.....	65	yywrap、%option.....	41
yy more、%option.....	41	yywrap()、要約.....	132
yy more()、要約.....	132	yywrap()、詳細.....	46
yy more()、詳細.....	46	yywrap()と EOF.....	46
yyout.....	45	yywrap()と POSIX.....	112
yyoutとソケット.....	128		
yyout、要約.....	132		

Short Contents

序	1
プログラムの実例について	3
1 Flex 入門	5
2 Flex の起動	9
3 Flex 記述言語	13
4 Flex とのインターフェイス	43
5 Flex の他の特徴	75
6 スキャナの最適化	85
7 Flex を使うその他の実例	91
8 Flex と Lex	111
9 役に立つコードの抜粋	115
10 要約	129
インデックス	139

Table of Contents

序	1
プログラムの実例について	3
1 Flex 入門.....	5
1.1 問題解決手段としての Flex.....	6
1.2 一般的なプログラミング・ツールとしての Flex.....	6
2 Flex の起動	9
2.1 コマンドライン・オプション	9
2.2 コマンドライン・オプション (Flex 2.5 の補足情報)	11
3 Flex 記述言語	13
3.1 コメント	13
3.2 オプションの C コード	13
3.3 定義	14
3.4 %%	15
3.5 ルール	15
3.6 パターン・セクション	16
3.6.1 文字	17
3.6.2 Flex における文字列	21
3.6.3 Flex における文字のグループ化	21
3.6.4 Flex における文字のグループ化 (Flex 2.5 の補足情報)	22
3.7 正規表現	23
3.8 スタート状態	24
3.8.1 スタート状態の説明	25
3.8.2 状態の活性化	26
3.8.3 スタート状態に関する注	27
3.8.4 スタート状態に関する注 (Flex 2.5 の補足情報)	28
3.8.5 スタート状態の使用例	31
3.9 %option (Flex 2.5 の補足情報)	37
4 Flex とのインターフェイス	43
4.1 Flex と C	43
4.2 Flex と C (Flex 2.5 の補足情報)	58
4.3 Flex と C の簡単な実例	59
4.4 Flex と Bison	61

4.4.1	Flex と Bison のインターフェイス	61
4.4.2	YYSTYPE と yyval	65
4.5	Flex と Bison のもう 1 つの実例	65
4.5.1	インターフェイス言語	65
4.5.2	実装：コマンド文パーサ	67
4.5.3	実装に関する注	71
4.6	Flex と C++ (Flex 2.5 の補足情報)	71
5	Flex の他の特徴	75
5.1	大文字・小文字を区別しないスキャナ	75
5.1.1	‘-i’ オプション	75
5.2	‘-I’ オプション：対話型スキャナ	76
5.3	テーブルの圧縮とスキャナのスピード	77
5.4	翻訳テーブル	79
5.5	複数の入力バッファ	79
5.5.1	バッファを操作する関数	79
5.5.2	バッファを操作する関数 (Flex 2.5 の補足情報)	80
5.5.3	複数バッファを使う実例	81
5.6	ファイルの終端 (End-Of-File) ルール	83
6	スキャナの最適化	85
6.1	スピードの最適化	85
6.1.1	バックトラッキングの削除	86
6.2	サイズの最適化	88
7	Flex を使うその他の実例	91
7.1	単語数、文字数、行数のカウント	91
7.2	Pascal のサブセット用の字句スキャナ	93
7.3	専門用語の変換	97
8	Flex と Lex	111
8.1	Flex	111
8.1.1	Flex と POSIX	111
8.1.2	Flex と POSIX (Flex 2.5 の補足情報)	113
8.2	標準 Lex	113
9	役に立つコードの抜粋	115
9.1	コメントの処理	115
9.2	文字列リテラルの処理	116
9.3	数字の処理	122
9.4	複数のスキャナ	126
9.5	その他	128

10	要約	129
10.1	Flex コマンドライン・オプションの要約	129
10.2	Flex コマンドライン・オプションの要約 (Flex 2.5 の補足情報)	131
10.3	Flex 変数および Flex 関数の要約	132
10.4	Flex 変数および Flex 関数の要約 (Flex 2.5 の補足情報)	135
10.5	Flex 文字の要約	136
10.6	Flex ルールの要約	138
	インデックス	139

