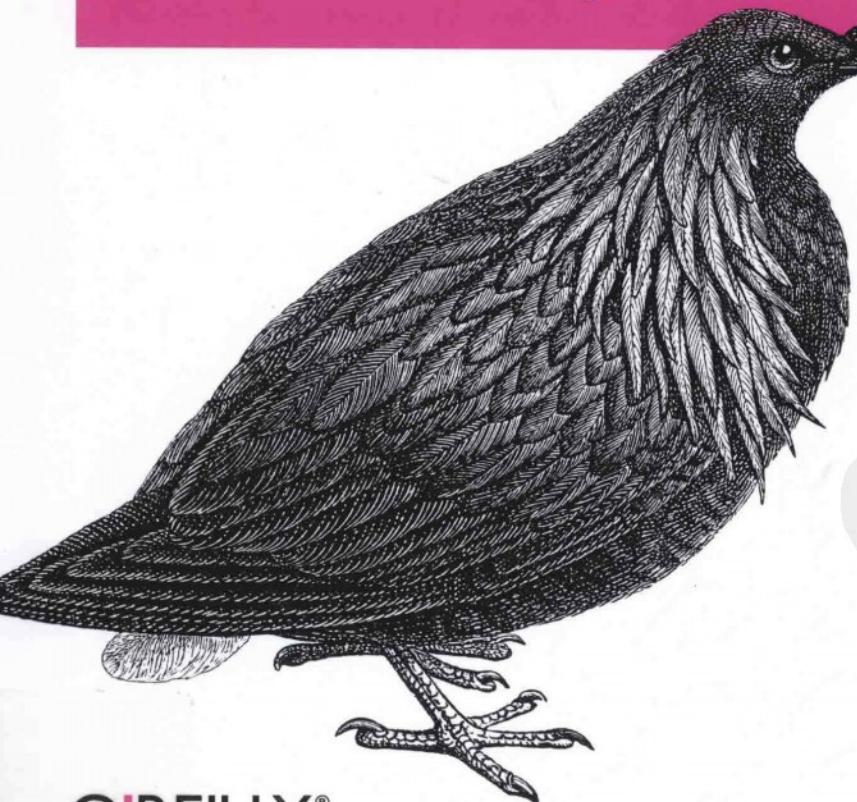


flex & bison



flex 与 bison

(中文版)



O'REILLY®

東南大學出版社

John Levine 著
陆军 译

赵文魁
陆军
PDG

flex与bison

如果你需要分析或处理Linux或Unix中的文本数据，这本有用的书籍就向你讲解了如何使用flex和bison迅速解决问题。《flex与bison》被期待已久，是经典O'Reilly系列书籍《lex & yacc》的续篇。在原书出版以来的近20年中，flex和bison已被证明比原来的Unix工具更可靠、更强大。

《flex与bison》一书涵盖了Linux和Unix程序开发中相同的重要核心功能，以及一些重要的新主题。你会找到适用于新手的修订教程和适用于高级用户的参考资料，以及对每个程序的基本用法的解释，并且运用它们创建简单、独立的应用程序。有了《flex与bison》，你会发现这些灵活的工具提供的广泛用途。

包括的主题有：

- 正则表达式工具无法处理的地址语法挤压（address syntax crunching）
- 生成编译器和解释器，并运用大范围的文本处理功能
- 解释代码、配置文件或任何其他结构化的格式
- 学习关键编程技术，包括抽象语法树和符号表
- 用完整的示例代码实现一个完善的SQL语法
- 使用新的功能，如纯（可重入）词法分析器（lexer）和语义分析器（parser）、功能强大的GLR分析器和C++的接口

INTRODUCTORY INTERMEDIATE ADVANCED

建议有Unix/Linux编程经验者阅读。

www.oreilly.com

O'Reilly Media, Inc.授权东南大学出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

“我很高兴看到John彻底详尽地重写这本经典书。他更新的示例和说明能够帮助老用户和新手摆脱模仿那些已经根深蒂固的旧lex和yacc。”

—Joel E. Denny

bison维护人员

John Levine, Taughannock Networks的创始人，著有20余本技术书籍，其中包括《lex & yacc》和《qmail》，均为O'Reilly出版。

O'REILLY®
oreilly.com

ISBN 978-7-5641-2605-6



9 787564 126056 >

定价：49.00元

flex与bison

John Levine 著

陆军 译



O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc.授权东南大学出版社出版

东南大学出版社

图书在版编目 (CIP) 数据

flex 与 bison/ (美) 利文 (Levine, J.) 编; 陆军译. —南京: 东南大学出版社, 2011.3

书名原文: flex & bison

ISBN 978-7-5641-2605-6

I. ① f… II. ①利… ②陆… III. ① Linux 操作系统 ② UNIX
操作系统 IV. ① TP316.8

中国版本图书馆 CIP 数据核字 (2011) 第 003003 号

江苏省版权局著作权合同登记

图字: 10-2010-274 号

©2009 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Southeast University Press, 2011. Authorized translation of the English edition, 2009 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form

英文原版由 O'Reilly Media, Inc. 出版 2009。

简体中文版由东南大学出版社出版 2011。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式重制。

flex 与 bison

出版发行: 东南大学出版社

地 址: 南京四牌楼 2 号 邮编: 210096

出 版 人: 江建中

网 址: <http://www.seupress.com>

电子邮件: press@seu.edu.cn

印 刷: 扬中市印刷有限公司

开 本: 787 毫米 × 980 毫米 16 开本

印 张: 17.5 印张

字 数: 343 千字

版 次: 2011 年 3 月第 1 版

印 次: 2011 年 3 月第 1 次印刷

书 号: ISBN 978-7-5641-2605-6

印 数: 1~4000 册

定 价: 49.00 元 (册)

本社图书若有印装质量问题, 请直接与读者服务部联系。电话 (传真): 025-83792328



O'Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc.授权东南大学出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc.是世界上在 Unix、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的*The Whole Internet User's Guide & Catalog*（被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc.一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc.具有深厚的计算机专业背景，这使得O'Reilly Media, Inc.形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc.所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc.依靠他们及时地推出图书。因为O'Reilly Media, Inc.紧密地与计算机业界联系着，所以O'Reilly Media, Inc.知道市场上真正需要什么图书。



目录

前言	1
第1章 Flex和Bison简介	7
词法分析和语法分析	7
正则表达式和词法分析	8
文法与语法分析	16
二义性文法：并不多见	20
添加更多的规则	21
Flex和Bison与手写的词法分析器和语法分析器的对比	22
练习	23
第2章 使用Flex	25
正则表达式	25
Flex词法分析器中的文件I/O操作	29
读取多个文件	30
Flex词法分析器的I/O结构	32
起始状态和嵌套输入文件	34
符号表和重要语汇索引生成器	38
C语言交叉引用	44
练习	51

第3章 使用Bison	53
Bison语法分析器如何匹配输入	53
移进/归约分析	54
Bison语法分析器	57
基于抽象语法树的改进的计算器	58
移进/归约冲突和操作符优先级	63
一个高级计算器	66
使用高级计算器	84
练习	85
第4章 分析SQL	86
SQL概述	86
关系操作	88
使用SQL的三种方法	88
从SQL到逆波兰式（RPN）	89
词法分析器	90
语法分析器	99
SQL分析器的Makefile	121
练习	122
第5章 Flex规范参考	123
Flex结构规范	123
BEGIN	124
C++词法分析器	125
上下文相关性	125
定义（替换）	126
ECHO	127
输入管理	127
Flex库	130
交互模式和批处理模式的词法分析器	130
行号和yylineno	130
文字块	131

单一程序中的多重词法分析器	131
编译词法分析器的选项	133
Flex词法分析器的可移植性	133
可重入词法分析器	134
正则表达式语法	137
REJECT	139
从yylex()返回值	140
起始状态	140
unput()	141
yyinput() yyunput()	142
yy leng	142
yy less()	142
yylex()和YY_DECL	143
ymore()	143
yyrestart()	144
yy_scan_string和yy_scan_buffer	144
YY_USER_ACTION	144
yywrap()	144
第6章 Bison规范参考	145
Bison语法结构	145
二义性和冲突	148
Bison程序的问题	150
C++语法分析器	151
%code块	151
结束标记	151
错误记号和错误恢复	152
继承属性 (\$0)	152
词法反馈	154
文字块	155
文字记号	155
位置	156

%parse-param	156
Bison语法分析器的可移植性	157
优先级和结合性声明	158
递归规则	160
规则	161
特殊字符	162
%start 声明	164
符号值	164
记号	165
可变语法和多重语法	168
多重语法分析器	169
y.output文件	170
Bison库文件	171
YYABORT	172
YYACCEPT	172
YYBACKUP	173
yyclearin	173
yydebug 和 YYDEBUG	173
YYERROR	174
yyerror()	175
yyparse()	175
YYRECOVERING()	175
第7章 二义性和冲突	177
指针模型和冲突	177
冲突类型	179
语法分析器状态	181
name.output 的内容	182
归约/归约冲突	183
移进/归约冲突	184
复习 name.output 中的冲突	187
常见的冲突例子	188

IF/THEN/ELSE.....	190
你如何解决冲突?	192
IF/THEN/ELSE（移进/归约冲突）	192
嵌套循环（移进/归约冲突）	194
表达式优先级（移进/归约冲突）	195
总结	199
练习	199
第8章 错误报告和恢复	200
错误报告	200
错误恢复	208
Bison错误恢复.....	208
编译器错误恢复	211
练习	212
第9章 Flex和Bison进阶	213
纯词法分析器和纯语法分析器	213
GLR分析	234
C++语法分析器	239
练习	246
附录 SQL语法分析器文法和交叉引用	247
术语表	263

前言

flex和**bison**是为编译器和解释器的编程人员特别设计的工具，不过它们在其他应用领域也非常有用，因此吸引了许多非编译器编程人员的注意。任何应用程序，只要它在其输入中寻找特定的模式，或者它使用命令语言作为输入，都适合使用**flex**和**bison**。而且，**flex**和**bison**允许快速的应用程序原型开发，修改简单，维护方便。让我来激发一下你的想象力吧，下面列出了一些使用**flex**和**bison**，或者它们的前身**lex**和**yacc**，来开发的程序：

- 桌面计算器**bc**
- 工具**eqn**和**pic**，用于数学方程式和复杂图片的排版预处理器
- 用于特定应用设计的大量领域特定语言
- **PCC**，在许多Unix系统中被使用的可移植的C编译器
- **flex**自身
- SQL数据库语言翻译器

本书内容

第1章，*Flex和Bison简介*，概述了**flex**和**bison**如何和为何可以被用来创建编译器和解释器，并且演示了一些简单的应用程序，包括内置于**flex**和**bison**的计算器。也介绍了一些我们这本书会用到的术语。

第2章，*使用Flex*，描述了如何使用**flex**。我们在本章开发了一些**flex**应用程序来计算文件中的单词数，处理多个和嵌套的输入文件，以及统计C程序中的交叉引用信息。

第3章，*使用Bison*，实现了一个完整的例子，它使用**flex**和**bison**开发了一个具有全部功能的桌面计算器，能够支持变量、过程、循环和条件表达式。本章展示了抽象语法树（AST）的用法，它具有强大而易用的数据结构来表示解析过的输入。

第4章，分析SQL，为SQL关系型数据库的MySQL方言开发了一种语法分析器。该分析器能够检查SQL语句的语法，然后把它们翻译成适合于特定解释器的内部形式。本章展示了逆波兰表示法（Reverse Polish Notation，RPN）的用法，它是另外一种高效的组织形式，用来表达和解释分析结果。

第5章，Flex规范参考，和第六章，Bison规范参考，为flex和bison程序员提供了详细的特性和选项描述。这两章和后续的两章为flex和bison的初学者提供了他们在开发flex和bison的应用程序时所需的技术信息。

第7章，二义性和冲突，解释了bison的二义性和冲突，它们会导致bison无法为特定语法创建语法分析器的问题。本章接着描述了一些可以被用来定位和解决这类问题的方法。

第8章，错误报告和恢复，讨论了编译器或者解释器的设计人员如何来定位、识别和报告编译器输入中的错误的技术。

第9章，Flex和Bison进阶，包括了可重入词法分析器和语法分析器、通用的自左向右语法分析器（它能够处理普通bison语法分析器无法处理的语法）以及C++的接口。

附录部分提供了第4章所讨论的SQL语法分析器的完整语法和交叉引用信息。

术语表列出了语言和编译器理论中的技术术语。

你需要熟悉C语言，因为大部分例子是由C语言、flex或者bison写成的，还有少量例子是由C++、SQL和其他基于特定目的而开发的语言来编写的。

排版约定

本书应用了如下一些印刷排版上的约定：

斜体字（*Italic*）

用于引入的新术语与概念。

等宽字（*Constant width*）

用于程序列表与段落中提到的程序元素，如语句、类、宏、状态、规则、所有代码项以及文件与目录。

等宽黑体字（*Constant width bold*）

用于命令或者其他用户需要逐字键入的文本类型。

等宽斜体字（*Constant width italic*）

用于需要使用用户提供的值或者由上下文确定的值替代的文本。

\$

命令行提示符。

[]

包围程序语法描述中的可选项。（方括号本身无需输入）

获取Flex和Bison

经典的lex和yacc由贝尔实验室在20世纪70年代开发，flex和bison则是它们的现代版本。 yacc由Stephen C. Johnson首先开发完成。lex由Mike Lesk和Eric Schmidt（现在他正领导着Google）设计，用来与bison协同工作。从第七版的Unix开始，lex和yacc就已经成为标准的Unix辅助工具。

自由软件基金会（Free Software Foundation）的GNU项目帮助发布bison，一种yacc的向前兼容版本。它最初由Robert Corbett和Richard Stallman编写。bison的帮助手册非常好用，特别是一些特性的参考。在BSD和Linux的发布版本中，你都可以找到bison，但是如果你希望使用最新的bison，你可以到它的主页下载：

<http://www.gnu.org/software/bison/>

BSD和GNU项目也发布flex（快速词法分析生成器），“它重写了lex来解决lex中存在的大量错误和缺陷。”flex最初由Jef Poskanzer编写，Vern Paxson和Van Jacobson相当多地改进了它。BSD和Linux的发布版本中包含flex的拷贝，但是如果你希望使用最新版本，它可以在SourceForge中找到：

<http://flex.sourceforge.net/>

本书的范例

本书的范例程序可以在线获得：

<ftp://ftp.iecc.com/pub/file/flexbison.zip>

它们可以通过任何浏览器和FTP客户端下载。zip压缩格式的文件可以通过在类Unix或者Linux系统中流行的自由软件unzip工具来解压，也可以在Windows XP或者更新的窗口系统中作为压缩文件夹来打开。

本书的范例程序在版本为2.5.35的flex和版本为2.4.1的bison上测试通过。

使用代码范例

希望本书能够帮助你完成工作。通常来说，你可以在你的程序和文档中使用本书的代码。除非你引用了显著数量的代码，否则你无需联系我们来获得许可。例如，编写一个使用了本书几段代码的程序就并不需要任何许可，但是销售或者发布带有O'Reilly书籍中的范例的光盘需要获得许可。引用本书和范例代码来回答问题也不需要许可，但把本书中显著数量的范例代码包含到你的产品文档中需要获得许可。

我们感谢，但并不要求，你给出出处。完整信息通常包括书名、作者、出版者和ISBN。例如：“书名、作者、copyright 2008 O'Reilly Media, Inc., 978-0-596-xxxx-x”。

如果你觉得你的范例代码使用方式超出常规的使用方式，或者并不在前面许可的范围内，请通过*permissions@oreilly.com*来联系我们。

如何联系我们

请将对本书的评价和存在的问题通过如下地址告知出版者：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly的每一本书都有专属网站，你可以在那找到关于本书的相关信息，包括勘误表、示例代码以及其他的信息。本书的网站地址是：

<http://www.oreilly.com/catalog/9780596155988/>

对于本书的评论和技术性的问题，请发送电子邮件到：

bookquestions@oreilly.com

关于本书的更多信息、会议、资料中心和O'Reilly网络，请访问以下网站：

<http://www.oreilly.com/>

<http://www.oreilly.com.cn/>

致谢

任何一本书都是众人的结晶，本书也不例外。我要感谢Tony Mason和Doug Brown，我的《lex & yacc》的共同作者，他们许可我摘录那本书的部分内容。许多人给我的草稿提供了宝贵意见，包括Lorenzo Bettini、Joel E. Denny、Danny Dubé、Ben Hanson、Jan Van Katwijk、Jeff Kenton、Timothy Knox、Chris Morley、Ken Rose和Juha Vihavainen。我要特别感谢Derek M. Jones，他在极短的时间里为每页提供了详尽的评审意见。Simon St. Laurent，为我长期受累的编辑，领导了本书的全部编辑和出版的过程而毫无怨言。



Flex和Bison简介

flex和bison是用来生成程序的工具，它们所生成的程序能够处理结构化输入。最初flex和bison是用来生成编译器的，但是后来它们被证明在其他很多领域也非常有效。在第1章里，我们将以它们背后的一部分（不是太多）原理作为介绍的开始，然后我们会深入讲解一些具体的使用例子。

词法分析和语法分析

最早的编译器可以追溯到20世纪50年代，这些编译器使用很特别的技术来分析它们所编译的程序源代码的语法。在60年代，这个领域获得了很多学术层面的关注，在70年代初，语法分析成为一个众所周知的领域。

在这个领域中，一个关键的想法是把分析工作分成两个部分：词法分析（*lexical analysis*，或称*scanning*）和语法分析（*syntax analysis*，或称*parsing*）。

简单来说，词法分析把输入分割成一个个有意义的词块，称为记号（*token*），而语法分析则确定这些记号是如何彼此关联的。例如，看一下这个C代码的片断：

```
alpha = beta + gamma ;
```

词法分析器把这段代码分解为这样一些记号：`alpha`、等号、`beta`、加号、`gamma`和分号。接着语法分析器确定了`beta + gamma`是一个表达式，而这个表达式被赋给了`alpha`。

获取Flex和Bison

大多数Linux和BSD系统自带flex和bison作为系统的基础部分。如果你的系统没有包含它们，或者包含的是过时的版本，安装它们也很容易。

flex是一个SourceForge项目 (<http://flex.sourceforge.net/>)。2009年初的版本是2.5.35。通常从一个版本到另一个版本的变化是很小的，如果你的版本接近.35，你就没有必要更新版本，不过有些系统自带的版本是2.5.4或者2.5.4a，这种版本就显得更老了。

bison可以从<http://www.gnu.org/software/bison>获得。2009年初的版本是2.4.1。bison还处在相当活跃的开发中，所以值得我们获取一个最新的版本来看看有哪些新引入的特性。比如，版本2.4就增加了对Java语法分析的支持。BSD用户通常可以使用Ports Collection来安装当前版本的flex和bison。Linux用户可以找到最新的RPM。如果没有的话，flex和bison都使用标准的GNU编译过程，因此为了安装它们，我们可以从这个网站下载并解压最新的flex和bison的压缩包，运行`./configure`以及`make`来编译它们，然后我们需要切换成超级用户并使用`make install`进行安装。

flex和bison都依赖于GNU m4 宏处理器。Linux和BSD都应该有m4，如果没有或者当前的版本太老的话，最新的m4也可以从<http://www.gnu.org/software/m4>获得。

对于Windows用户来说，bison和flex都被包含在Cygwin Linux模拟环境 (<http://www.cygwin.com>) 中。你可以使用由Cygwin开发工具或者Windows本地开发工具所生成的C或者C++代码。

正则表达式和词法分析

词法分析通常所做的就是在输入中寻找字符的模式 (pattern)。例如，在C程序中，整型常量是一个或多个数字的字符串，变量名称是一个字母后面跟着零个或者多个字母和数字，而各种各样的操作符则是一个或者成对的字符。一种简洁明了的模式描述方式就是正则表达式 (*regular expression*, 常简写为*regex*或*regexp*)。编辑器ed和vi以及搜索程序egrep就使用这种方式来描述其所需要查找的文本。flex程序主要由一系列带有指令的正则表达式组成，这些指令确定了正则表达式匹配后相应的动作 (*action*)。由flex生成的词法分析器可以读取输入，匹配输入与所有的正则表达式并且执行每次匹配后适当的关联动作。flex会把所有的正则表达式翻译成一种高效的内部格式，这使它几乎可以同时处理所有需要匹配的模式，因此它的速度可以成百倍地提高^(注1)。

注1： 内部格式使用确定性有穷自动机 (Deterministic Finite Automaton, DFA)。值得庆幸的是，对于确定性有穷自动机，你唯一需要知道的就是它们非常快，而且它们的处理速度与模式的个数和复杂程度无关。

我们的第一个Flex程序

Unix系统（同样也包括像Linux和BSD这样的类Unix系统）自带了一个字数统计（word count）程序，这个程序可以读入一个文件然后报告这个文件的行数、单词数和字符数。flex使我们能够用仅仅十几行就完成这个wc程序，见例1-1。

```
例1-1：字数统计 fbl-1.l
/* 正如 Unix的wc程序 */
%{
int chars = 0;
int words = 0;
int lines = 0;
%}
%%
[a-zA-Z]+ { words++; chars += strlen(yytext); }
\n      { chars++; lines++; }
.       { chars++; }
%%

main(int argc, char **argv)
{
    yylex();
    printf("%d%d%d\n", lines, words, chars);
}
```

这个程序的大部分内容对于C程序员来说是十分熟悉的，因为这些就是C代码。flex程序包含三个部分，各部分之间通过仅有%%的行来分割。第一个部分包含声明和选项设置，第二个部分是一系列的模式和动作，第三部分则是会被拷贝到生成的词法分析器里面的C代码，它们通常是一些与动作代码相关的例程。

在声明部分，%{和%}之间的代码会被原样照抄到生成的C文件的开头部分。在这个例子里面，它只是用来设定了行数、单词数和字符数的变量。

在第二部分，每个模式处在一行的开头处，接着是模式匹配时所需要执行的C代码。这儿的C代码是用{}括住的一行或者多行语句。（模式必须在行首出现，因为flex认为以空白开始的行都是代码而把它们照抄到生成的C程序中。）

这个程序只有三个模式。第一个模式，[a-zA-Z]+，用来匹配一个单词。在方括号里面的字符串是一种字符类（character class），能够匹配任意一个大小写字母，而+这个符号表示匹配一个或者多个前面的字符类，也就是一连串的字母，或者说一个单词。相关的动作更新匹配过的单词和字符的个数。在任意一个flex的动作中，变量yytext总是被设为指向本次匹配的输入文本。在这个例子里，我们所需要关心的是有多少个字符，因此我们可以借助这个变量来统计字符数。

第二个模式，\n，用来匹配换行符。相关的动作更新行数和字符数。

最后的模式是一个点号，它在正则表达式中代表任意一个字符（与shell脚本中的?相似）。关联的动作更新字符数。这些就是我们需要的所有模式^(注2)。

末尾的C代码是我们的主程序，它负责调用flex提供的词法分析例程`yylex()`，并输出结果。在没有任何其他改变的情况下，词法分析器将读取标准输入。我们来运行一下。

```
$ flex fb1-1.1
$ cc lex.yy.c -lfl
$ ./a.out
The boy stood on the burning deck
shelling peanuts by the peck
^D
2 12 63
$
```

首先我们用flex来翻译我们的程序，flex在没有任何错误的情况下默默地完成了翻译，正如Unix的经典设定那样。接着我们编译flex生成的C程序：`lex.yy.c`；将它与相应的flex库文件（-lfl）链接；运行它；然后提供一小段输入以便于程序进行统计。看起来一切运作正常。

真正的wc程序对单词的定义和这儿相比有些微小的差别，它的定义是没有空白字符的字符串。一旦我们知道哪些是空白字符，我们只需把匹配单词的那个模式替换成匹配没有空白字符的字符串：

```
[^ \t\n\r\f\v]+ { words++; chars += strlen(yytext); }
```

在字符类开始部分的符号`^`是指匹配任意一个不在字符类里面的字符，而符号`+`依然意味着匹配一个或者多个前面的模式。这展示了flex的强大之一——我们很容易在模式上做一些小的改动而让flex去担心它们可能怎样影响生成的代码。

纯Flex的程序

一些应用程序简单到你可以把所有的内容都写在flex里面，或者只是增加一点点C代码。例如，例1-2就实现了从英式英语到美式英语的翻译器框架。

例1-2：英式英语到美式英语fb1-2.1

```
/* 英式英语 -> 美式英语 */
%%
"colour" { printf("color"); }
```

注2：也许旁观的读者会问，如果一个点号代表所有的字符，难道它不会也匹配第一个模式所应该匹配的字母吗？答案是肯定的，但是flex棋高一着的地方是它总是选择更长的匹配，而且如果两个模式都匹配的话，flex会选择在程序里面首先出现的那个模式。这是一个精妙的设计，我们会发现它非常有用。

```
"flavour" { printf("flavor"); }
"clever" { printf("smart"); }
"smart" { printf("elegant"); }
"conservative" { printf("liberal"); }
... 其他更多的单词 ...
. { printf("%s", yytext); }
%%
```

这个程序读取其输入，当匹配到一个英语单词时就打印出其美式版本，否则就直接输出。这个例子在一定意义上显得并不真实（毕竟*smart*也有可能是*hurt*的意思），但是flex还是比较适合用来进行一定程度上的文本转换和输入统计的。当然，更多情况下你会用flex生成词法分析器——它可以把输入分解成为记号以便于你的程序的其他部分来使用。

让Flex和Bison协同工作

我们的第一个同时使用flex和bison的程序将是一个桌面计算器（desk calculator）。首先，我们编写一个词法分析器，接着我们编写一个语法分析器并把两者接合起来。

为了让事情变得简单，我们的计算器只需要识别整数、基本算术运算符和一元绝对值操作符（例1-3）。

例1-3：一个简单的flex词法分析器fb1-3.l

```
/* 识别出用于计算器的记号并把它们输出 */

%%
"+"   { printf("PLUS\n"); }
"-"   { printf("MINUS\n"); }
"*"   { printf("TIMES\n"); }
"/"   { printf("DIVIDE\n"); }
"|"   { printf("ABS\n"); }
[0-9]+ { printf("NUMBER %s\n", yytext); }
\n    { printf("NEWLINE\n"); }
[\t]   { }
.     { printf("Mystery character %s\n", yytext); }
```

前5个模式就是操作符本身，用引号引起，而目前的动作仅仅是打印出匹配的内容。引号告诉flex使用引号内文本的原义，而不是把它们解释成正则表达式。

第6个模式匹配一个整数。这种方括号括起的模式[0-9]可以匹配任意一个数字，接着的+这个符号表示匹配一个或者多个前面的项，也就是由一个或者多个数字组成的字符串。相关的动作利用词法分析器每次匹配后所设置的yytext来打印出匹配的字符串。

第7个模式匹配一个换行符，它使用C语言通常使用的序列：`\n`。

第8个模式用来忽略空白字符。它匹配任意一个空格或者tab (\t)，相关的动作无需做任何事情。

最后一个模式用来匹配其他模式所没有匹配的内容。相关的动作打印出恰当的抱怨信息。

这9个模式提供了一定的规则来匹配用户可能的输入。随着我们继续开发计算器，我们将添加更多的规则来匹配更多的记号，但这些已经可以作为我们的开始部分。

在这个简单的flex程序中，我们并没有第三段的C代码。flex库文件（-lfl）提供了一个极小的主程序来调用词法分析器，这对我们的例子来说已经足够。

让我们试验一下这个词法分析器：

```
$ flex fb1-3.1
$ cc lex.yy.c -lfl
$ ./a.out
12+34
NUMBER 12
PLUS
NUMBER 34
NEWLINE
5 6 / 7q
NUMBER 5
NUMBER 6
DIVIDE
NUMBER 7
Mystery character q
NEWLINE
^D
$
```

首先我们运行flex，它把词法分析器翻译成C程序（lex.yy.c），接着我们编译这个C程序并最终运行它。输出结果表明它可以识别数字、操作符以及最后一行的q（匹配最后的那个包罗万象的模式）。(^D是Unix/Linux的文件结束符。在Windows中，你可以输入^Z。）

作为协同程序的词法分析器

大多数包含flex词法分析器的程序使用词法分析器来获得一个记号流，这样可以方便语法分析器的处理。每当程序需要一个记号时，它调用yylex()来读取一小部分输入然后返回相应的记号。当程序需要下一个记号时，yylex()会被再次调用。词法分析器以协同程序的方式来运行，也就是说，每次它返回的时候，它会记住当前处理的位置，并从这个位置开始去处理下一次调用。

在词法分析器中，当动作代码识别出一个记号时，`yyelex()`将以这个记号作为返回值。当程序再次调用`yylex()`时，词法分析器会以后续的输入字符继续分析。相反，如果一个模式不能够产生一个用于调用程序且不可以返回的记号时，词法分析器会在这次`yylex()`的调用中继续分析接下来的输入字符。下面这个并不完整的片段包含了两个可以返回记号的模式，一个返回+操作符，另一个返回数字；它还包含一个不做任何事情的空白字符的模式，这个模式用来忽略它所匹配的内容。

```
"+" { return ADD; }
[0-9]+ { return NUMBER; }
[ \t] { /* 忽略空白字符 */ }
```

这种看起来并不确定的返回方式经常会迷惑flex的新手，但是这种规则事实上非常简单：如果动作有返回，词法分析会在下一次`yylex()`调用时继续；如果动作没有返回，词法分析将会立即继续进行。

现在我们将修改我们的词法分析器，它所返回的记号可以被语法分析器用来实现一个计算器。

记号编号和记号值

当flex词法分析器返回一个记号流时，每个记号实际上有两个组成部分，记号编号（token number）和记号值（token's value）。这里的记号编号是一个较小的整数。这个数字是随意确定的，但是零值总是意味着文件的结束。当bison创建一个语法分析器时，bison自动地从258（这样可以避免与文字字符记号产生冲突，后面会详细讨论）起指派每个记号编号，并且创建一个包含这些编号定义的.h文件。不过这次我们将手工定义一些编号：

```
NUMBER = 258,
ADD = 259,
SUB = 260,
MUL = 261,
DIV = 262,
ABS = 263,
EOL = 264 行结束
```

（当然，事实上，正如我们后面会看到的，它们也就是bison所创建的记号编号列表。但是这些编号一样好用。）

记号值可以区分一组相似的记号。在我们的词法分析器中，所有的数字都属于Number这个记号，而记号值则表明了具体的数字。当分析更复杂的输入时（名称、浮点数、字符串字面量等等），记号值将告诉我们具体的名称、数字、字面值和其他内容。我们第一版的计算器的词法分析器带有一个用于调试的主程序，参见例1-4。

例1-4：计算器词法分析器 *fb1-4.l*

```

/* 识别出用于计算器的记号并把它们输出 */
%{
    enum yytokentype {
        NUMBER = 258,
        ADD = 259,
        SUB = 260,
        MUL = 261,
        DIV = 262,
        ABS = 263,
        EOL = 264
    };
    int yyval;
}

%%
"+"
"-"
"*"
"/"
"|" 
[0-9]+
\n
[\t]
.
%%

main(int argc, char **argv)
{
    int tok;

    while(tok = yylex()) {
        printf("%d", tok);
        if(tok == NUMBER) printf(" = %d\n", yyval);
        else printf("\n");
    }
}

```

我们在一个C语言的enum中定义记号编号。接着我们把yyval（这个变量用来储存记号值）定义为整型，这对于我们第一版的计算器来说已经足够。（后面我们会看到记号值通常被定义为联合类型以便于不同类型的记号可以拥有不同类型的记号值，例如，符号表中的浮点数或者指向一个符号项的指针。）

这里的模式列表与前面的例子相同，但是动作代码有所不同。对于每个记号，词法分析器返回适当的记号代码；对于编号，它把数字字符串转成整数并在返回前储存到yyval中。匹配空白字符的模式并不返回，所以词法分析器会继续分析接下来的输入。

出于测试的目的，主程序将调用yylex()，打印出记号值，并且对于Number记号，也会输出yyval。

```
$ flex fb1-4.l
```

```
$ cc lex.yy.c -lfl  
$ ./a.out  
a / 34 + |45  
Mystery character a  
262  
258 = 34  
259  
263  
258 = 45  
264  
^D  
$
```

现在我们就有了一个能够工作的词法分析器，我们可以把注意力转向语法分析器。

Flex和Bison的起源

bison来源于yacc，一个由Stephen C. Johnson于1975年到1978年期间在贝尔实验室完成的语法分析器生成程序。正如它的名字（yacc是“yet another compiler compiler”的缩写）所暗示的那样，那时很多人都在编写语法分析器生成程序。Johnson的工具基于D.E.Knuth所研究的语法分析理论（因此yacc十分可靠）和方便的输入语法。这使得yacc在Unix用户中非常流行，尽管当时Unix所遵循的受限版权使它只能够被使用在学术界和贝尔系统里。大约在1985年，Bob Corbett，一个加州伯克利大学的研究生，使用改进的内部算法再次实现了yacc并演变成为伯克利yacc。由于这个版本比贝尔实验室的yacc更快并且使用了灵活的伯克利许可证，它很快成为最流行的yacc。来自自由软件基金会（Free Software Foundation）的Richard Stallman改写了Corbett的版本并把它用于GNU项目中，在那里，它被添加了大量的新特性并演化成为当前的bison。bison现在作为FSF的一个项目而被维护，且它基于GNU公共许可证进行发布。

在1975年，Mike Lesk和暑期实习生Eric Schmidt编写了lex，一个词法分析器生成程序，大部分编程工作由Schmidt完成。他们发现lex既可以作为一个独立的工具，也可以作为Johnson的yacc的协同程序。lex因此变得十分流行，尽管它运行起来有一点慢并且有很多错误。（不过Schmidt后来在计算机行业里拥有一份非常成功的职业，他现在是Google的CEO。）

大概在1987年，Lawrence Berkeley实验室的Vern Paxson把一种用ratfor（当时流行的一种扩展的Fortran语言）写成的lex版本改写为C语言的，被称为flex，意思是“快速词法分析器生成程序”（Fast Lexical Analyzer Generator）。由于它比AT&T的lex更快速和可靠，并且就像伯克利的yacc那样基于伯克利许可证，它最终也超越了原来的lex。flex现在是SourceForge的一个项目，依然基于伯克利许可证。

文法与语法分析

语法分析器的任务是找出输入记号之间的关系。一种常见的关系表达方式就是语法分析树（*parse tree*）。例如，基于通常的算术规则，算术表达式 $1 * 2 + 3 * 4 + 5$ 有如图 1-1 所示的语法分析树。

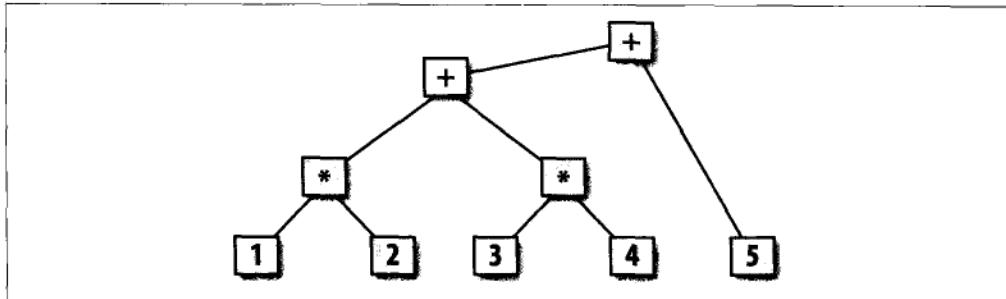


图1-1：表达式语法分析树

乘法比加法有更高的优先级，所以前两个表达式是 $1 * 2$ 和 $3 * 4$ 。接着这两个表达式被加在一起，它们的和再加上 5。这棵树的每个分支都显示了记号之间或者记号与下面子树的关系。这棵树的结构十分简单和普通，每个节点都有两个直接子孙（*descendant*）（这也是我们使用计算器作为第一个例子的原因），但是每一个 *bison* 语义分析器在分析其输入时都会构造一棵语法分析树。在有些应用里，它把整棵树作为一个数据结构创建在内存中以便于后续使用。在其他应用里，语法分析树只是隐式地包含在语义分析器进行的一系列操作中。

BNF 文法

为了编写一个语义分析器，我们需要一定的方式来描述语义分析器所使用的把一系列记号转化为语法分析树的规则。在计算机分析程序里最常用的语言就是上下文无关文法（*Context-Free Grammar*, CFG）^(注3)。书写上下文无关文法的标准格式就是 *Backus-Naur* 范式（*Backus-Naur Form*, BNF），大致创立于 1960 年，用来描述 Algol 60 语言，由两名 Algol 60 委员会的成员命名。

幸运的是，BNF 相当简单。这里描述简单算术表达式的 BNF 足以处理 $1 * 2 + 3 * 4 + 5$ ：

注3：上下文无关文法（CFG）也被称为短语结构文法（*Phrase-Structure Grammar*）或者 3 型语言（*Type-3 language*）。计算机理论学家和自然语言学家在 20 世纪 50 年代彼此独立地开发了它们。如果你是一名计算机科学家，你通常称呼它们为上下文无关文法（CFG）；如果你是语言学家，你会称呼它们为短语结构文法或者 3 型语法，但是它们是同一种文法。

```
<exp> ::= <factor>
    | <exp> + <factor>
<factor> ::= NUMBER
    | <factor> * NUMBER
```

每一行就是一条规则，用来说明如何创建语法分析树的分支。在BNF里，`::=`被读作“是”或者“变成”，`|`是“或者”，创建同类分支的另一种方式。规则左边的名称是语法符号 (*symbol*)。大致来说，所有的记号都被认为是语法符号，但是有一些语法符号并不是记号。

有效的BNF总是带有递归性的，规则会直接或者间接地指向自身。这些简单的规则被递归地使用来匹配任何极端复杂的加法和乘法序列。

Bison的规则描述语言

bison的规则基本上就是BNF，但是做了一点点简化以易于输入。例1-5显示了实现我们的第一版计算器的bison代码，里面就包含了BNF。

例1-5：简单的计算器 *fb1-5.y*

```
/* 计算器的最简版本 */
%{
#include <stdio.h>
%}

/* declare tokens */
%token NUMBER
%token ADD SUB MUL DIV ABS
%token EOL

%%

calclist: /* 空规则 */
    | calclist exp EOL { printf("= %d\n", $2); }      // 从输入开头进行匹配
    ;                                                       // EOL 代表一个表达式的结束

exp: factor default $$ = $1
    | exp ADD factor { $$ = $1 + $3; }
    | exp SUB factor { $$ = $1 - $3; }
    ;

factor: term default $$ = $1
    | factor MUL term { $$ = $1 * $3; }
    | factor DIV term { $$ = $1 / $3; }
    ;

term: NUMBER default $$ = $1
    | ABS term { $$ = $2 >= 0? $2 : - $2; }
    ;
%%

main(int argc, char **argv)
```

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

bison程序包含了（不是巧合）与flex程序相同的三部分结构：声明部分、规则部分和C代码部分。声明部分包含了会被原样拷贝到目标分析程序开头的C代码，同样也通过%{和%}来声明。随后是%token记号声明，以便于告诉bison在语法分析程序中记号的名称。通常来说，记号总是使用大写字母，虽然bison本身并不这么要求。任何没有声明为记号的语法符号必须出现在至少一条规则的左边。（如果一个语法符号既不是记号也没有出现在任何规则的左边，它会像C程序里没有被引用的变量。它并不会破坏任何事情，但是它可能意味着程序员犯了一个错误。）

第二部分包含了通过简单的BNF定义的规则。bison使用单一的冒号而不是::=，同时由于行间隔并不那么明显，分号被用来表示规则的结束。同样，像flex那样，C的动作代码在每条规则之后用花括号括起。

bison会自动帮你分析语法，记住每条被匹配的规则，所以动作代码只需要维护每个语法符号关联的语义值。bison语法分析器也执行一些额外的动作，比如创建数据结构以便后续使用（就像这个例子里面，我们会利用它来输出结果）。第一条规则左边的语法符号是语法起始符号 (*start symbol*)，整个输入必须被它匹配。当然，其他规则的左边也可以拥有相同的起始符号。

每个bison规则中的语法符号都有一个语义值，目标符号（冒号左边的语法符号）的值在动作中代码用\$\$代替，右边语法符号的语义值依次为\$1、\$2，直到这条规则的结束。当词法分析器返回记号时，记号值总是储存在yyval里，其他语法符号的语义值则在语法分析器的规则里进行设置。在本例中，`factor`、`term`和`exp`符号的语义值就是它们所描述的表达式的值。

在这个语法分析器里，头两条规则定义了`calclist`语法符号，它们通过循环来读入用换行符结束的表达式并且打印结果。`calclist`的定义使用一种常见的双规则递归定式来实现一个序列或者列表：第一个规则为空所以不进行任何匹配；第二个规则添加一个项目到列表中。第二个规则中的动作通过\$2打印出`exp`的值。

其余的规则实现了计算器。带有操作符的规则（例如，`exp ADD factor`和`ABS term`）在语义值上进行相应的算术操作。右边仅有一个语法符号的规则可以像黏合剂一样组合文法，例如，一种表达式（`exp`）就是一个因子（`factor`）。如果一个规则缺少显式的动

如何分析人类语言？

在很长一段时间里，甚至可以追溯到20世纪50年代，人们试图通过计算机程序来处理自然语言（*natural languages*），那些由人们说出的而不是由计算机使用的语言，这个任务显然极端困难。一种方式是像分析计算机语言那样分析它们，就像下面这段一样：

```
simple_sentence: subject verb object
    | subject verb object prep_phrase ;
subject: NOUN
    | PRONOUN
    | ADJECTIVE subject ;
verb: VERB
    | ADVERB VERB
    | verb VERB ;
object: NOUN
    | ADJECTIVE object ;
prep_phrase: PREPOSITION NOUN ;
```

不幸的是，它只能够处理很小的并且不现实的一部分自然语言。虽然自然语言也有语法，但这些语法异常复杂而且不容易写下来或者通过软件处理。这带来一个很有趣的开放问题。为什么我们为计算机发明的语言比我们所说的简单的多呢？

作，语法分析器将把\$1赋予\$\$.这是一个内部设定，不过很有用，因为在大部分情况下它总是对的。

联合编译Flex和Bison程序

在我们把词法分析器和语法分析器构造为同一个可以工作的程序前，我们需要对例1-4里面的词法分析器做一些小小的改动，以便于我们可以在语法分析器里面直接调用它。具体就是我们包含了bison为我们创建的头文件而不是在第一部分里定义显式的记号值，那个头文件包含了记号编号的定义和yyval的定义。我们同时也删除了词法分析器第三部分的测试主例程，因为语法分析器会调用词法分析器。现在词法分析器的第一部分类似于例1-6所示。

例1-6：计算器词法分析器fb1-5.l

```
%{
# include "fb1-5.tab.h"
%}
%% 与前面相同的规则，也不需要第三部分的代码
```

现在的编译过程已经足够复杂，我们最好把它放到一个Makefile中：

```
# makefile的一部分
```

```
fb1-5: fb1-5.l fb1-5.y
bison -d fb1-5.y
flex fb1-5.l
cc -o $@ fb1-5.tab.c lex.yy.c -lfl
```

首先它以-d（用于“定义”文件）标志运行bison，这会创建fb1-5.tab.c和fb1-5.tab.h，接着它运行flex来创建lex.yy.c。然后它把两者和flex的库文件编译在一起。测试一下生成的可执行程序，特别是验证它可以正确地处理操作符的优先级——乘法和除法会在加法和减法前执行：

```
$ ./fb1-5
2 + 3 * 4
= 14
2 * 3 + 4
= 10
20 / 4 - 2
= 3
20 - 4 / 2
= 18
```

二义性文法：并不多见

也许读者这时会想例1-5里的文法是否必须那么复杂。为什么不是写成这样呢？

```
exp: exp ADD exp
    | exp SUB exp
    | exp MUL exp
    | exp DIV exp
    | ABS exp
    | NUMBER
;
```

有两个原因：优先级和二义性。分开的Term、factor和exp的语法符号可以让bison首先处理ABS，接着是MUL和DIV，然后是ADD和SUB。通常来说，一旦一种文法有不同优先级，也就是一种操作符比另一种操作符绑定得更紧密时，语法分析器就需要为每种优先级制定一条规则。

既然如此，下面的文法又如何呢？

```
exp: exp ADD exp
    | exp SUB exp
    | factor
;
factor和term部分相似
```

bison最强大的地方之一，同时也是它最令人烦恼的特征之一，就是它并不分析二义性文法。也就是说，bison创建的任何语法分析器总是用同一种方式来分析输入，而且这些语

法分析器只接受相同的文法。前面的文法是有歧义的，因为类似 $1 - 2 + 3$ 的输入可以被分析为 $(1-2) + 3$ 或者 $1 - (2+3)$ ，两种不同的表达式其结果也不同。虽然有些情况下二义性并没有任何问题（例如 $1 + 2 + 3$ ），但在大多数情况下二义性是一种错误，这种文法需要被改正。我们在例1-5里面编写的文法使得表达式得以分组而不再带有歧义。如果一种文法是有歧义的，bison会报告冲突（conflicts），并且标示出针对给定的输入哪儿会有两种不同的分析。bison还是会创建出一种语法分析器，它会选择冲突中的一种分析方式，但这种选择意味着它所分析的语言并不一定是你想指定的那种。我们会在第7章做一定程度的讲述。

bison常用的分析算法可以向前看一个记号来决定用哪种规则来匹配输入。有的文法并不具有二义性但是有些地方可能需要向前看更多的记号来决定匹配的规则。这也可能导致冲突，尽管通常有可能重写文法来使得只要向前查看一个记号就足够了。

事实上，前面关于二义性的讨论并不是那么正确。由于表达式文法十分常见和有效，而且为每个优先级分开编写规则也是一件乏味的事情，bison有一些特性使你可以通过`exp OP exp`这样每个操作符一条规则的形式自然地写出表达式文法，妙诀在于事先告诉bison优先级和分组的规则以便于解决二义性。我们将会在第3章了解到具体的实现方法。bison还有另一种分析技巧，被称为GLR，它可以处理二义性文法和任意的向前查看，并且可以并行地追踪所有可能的输入匹配分析。我们将在第9章讲述它。

添加更多的规则

通过flex和bison来处理程序输入的一件很美妙的事情就是文法的少量调整总是很简单的。想象一下，如果我们的表达式语言能够处理圆括号以及注释（使用//语法）的话，它是否会变得更加有用和完美呢？为了实现这样的改进，我们仅仅需要为语法分析器和词法分析器分别添加一条和三条规则。

在语法分析器中我们定义两个新的记号，OP和CP，作为开始和结束的圆括号，并且添加一条规则使一个圆括号表达式成为一个term：

```
%token OP CP 定义部分
...
%%
term: NUMBER
| ABS term { $$ = $2 >= 0? $2 : - $2; }
| OP exp CP { $$ = $2; } 新规则
;
```

注意在新规则中动作代码把\$2（圆括号中表达式的值）赋给了\$\$。

词法分析器有两条新规则来识别两个新的记号，还有一条新规则忽略两个斜线之后的任意文本。由于点号匹配除了换行符之外的任意字符，.*可以匹配掉一行剩下的部分。

```
"("      { return OP; }
")"      { return CP; }
"/".*    /* 忽略注释 */
```

这就是我们所需要做的一切——重新编译这个计算器，现在它可以处理圆括号表达式和注释了。

Flex和Bison与手写的词法分析器和语法分析器的对比

这一章里的两个例子，字数统计和计算器，都是非常简单的，我们即使完全通过C语言来实现它们也并没有太大的问题。但有一些理由会支持我们通过这种方式来开发一个程序。flex所使用的模式匹配技术非常快，与手写的词法分析器速度基本一致。在拥有更多模式的更复杂的词法分析器中，flex的词法分析器可能更快，因为手写的代码通常需要对每个字符做很多比较，而flex只需要一次。flex版本的词法分析器永远比相应的C代码简短，这使它更容易调试。一般来说，如果一条规则需要分解输入流为一个个记号并且它可以通过正则表达式来描述，那么flex将是我们的选择。

一个手写的词法分析器

如果一种语言的词法分析不是太复杂的话，手写的词法分析器也可以作为flex词法分析器的替代品。这儿是一个手写的C代码，对应于例1-6的词法分析器。这个词法分析器可能会比flex版本快一点点，但是它非常难以修改来添加或者改变记号的类型。如果你打算使用一个手写的词法分析器，最好先通过flex来做一些原型(prototype)。

```
/*
 * 用于计算器的词法分析器的手写版本
 */
#include <stdio.h>
#include "fb1-5.tab.h"
FILE *yyin;
static int seeneof = 0;
int
yylex(void)
{
    if(!yyin) yyin = stdin;
    if(seeneof) return 0; /* 上次读到的是EOF */
    while(1) {
```

```

int c = getc(yyin);
if(isdigit(c)) {
    int i = c - '0';
    while(isdigit(c = getc(yyin)))
        i = (10*i) + c-'0';
    yyval = i;
    if(c == EOF) seeneof = 1;
        else ungetc(c, yyin);
    return NUMBER;
}
switch(c) {
    case '+': return ADD; case '-': return SUB;
    case '*': return MUL; case '|': return ABS;
    case '(': return OP; case ')': return CP;
    case '\n': return EOL;
    case ' ': case '\t': break; /* 忽略它们 */
    case EOF: return 0; /* 标准的文件结束符 */
    case '/': c = getc(yyin);
        if(c == '/') { /* 这是注释 */
            while((c = getc(yyin)) != '\n')
                if(c == EOF) return 0; /* EOF 出现在注释行里 */
            break;
        }
        if(c == EOF) seeneof = 1; /* 这是除法操作符 */
        else ungetc(c, yyin);
        return DIV;
    default: yyerror("Mystery character %c\n", c); break;
}
}
}

```

同样，bison语法分析器比对应的手写语法分析器更简短也更容易调试，特别是bison可以帮助验证分析的文法是否有二义性。

练习

1. 我们的计算器是否接受只有注释的行？为什么不接受？是在词法分析器里面修改来得简单还是在语法分析器里面简单？
2. 把这个计算器变成一个十六进制的计算器，使得它可以同时接受十六进制和十进制数字。在词法分析器中加入一个模式，比如0x[a-f0-9]+，来匹配十六进制数字，在动作代码中使用strtol把字符串转化成为数字并存储在yyval里，然后返回Number记号。调整输出函数printf来打印十进制和十六进制的结果。
3. (加分题) 在计算器中加入位操作符，比如AND和OR。最直接的用于OR的操作符就是竖线，但它已经是一元绝对值操作符。如果你同时也使用它作为二进制OR操作符会怎样呢，比如，exp ABS factor?

4. 在例1-4中手写版本的词法分析器是否识别与flex版本一致的记号呢？
5. 你能够想出一些并不适合使用flex作为词法分析器的语言吗？
6. 用C语言重写字数统计程序。用两种版本分别运行一些大文件输入。C语言版本的语法分析器是否明显地快？它的调试有多困难？



使用Flex

本章通过一些例子来进一步介绍flex作为独立工具的使用方法，这些例子运用了flex主要的C语言特性。第5章会讲解flex的所有功能，而第9章则会讲解在C++程序中使用flex词法分析器的方法。

正则表达式

任何flex词法分析器的模式都使用了强大的正则表达式语言。正则表达式使用元语言(*metalinguage*)来描述你想匹配的模式。flex的正则表达式语言本质上是扩展的POSIX正则表达式（考虑到它们继承了Unix的很多特性，你就不会惊讶了）。元语言使用标准的文本字符，一部分代表它们自身而另外一部分则代表模式。所有下文没有列出的字符，包括字母和数字，将匹配自身。

在正则表达式中有特殊意义的字符为：

匹配除换行符 (\n) 以外的任意单一字符。

{}]

字符类 (*character class*)，可以匹配方括号中的任意一个字符。如果字符类中的第一个字符是抑扬符号 (^)，则改变为匹配除方括号内字符以外的任何字符。字符类里的破折号表示字符的范围，例如，[0-9]意味着[0123456789]而[a-z]则表示任意小写字母。你可以把 - 或者] 作为第一个字符放在[之后，这样字符类也会包含它们。POSIX引入了其他一些特殊的字符类，它们在处理非英文字母时十分有效，我们会在本章稍后介绍。C语言中以 \ 开头的转义序列 (*escape sequence*) 在方括号中也会被识别，其他元字符就没有什么特殊意义了。字符范围根据当前使用的字符编码来解释，所以以ASCII字符编码的[A-z]将匹配所有大小写字母，同

时还匹配落在Z和a之间的6个标点符号。实际上，有用的字符范围通常包括数字范围、大写字母范围或者小写字母范围。

[a-z}{-}[jv]

一种不同的字符类，包含的字符是前一个字符类减去后一个字符类的结果（只有最近的flex版本才支持）。

^

如果它是正则表达式的第一字符就匹配行首。它也被用于方括号中表示补集。

\$

如果它是正则表达式的最后一个字符就匹配行尾。

{}

当花括号中带有一个或者两个数字时，它表示前一个模式可以匹配的最小和最大次数。例如，A{1,3}匹配一到三个字母A，而0{5}匹配00000。当花括号中带有名字时，它指向以这个名字命名的模式。

\

用来表示元字符自身和一部分常用的C语言转义序列。例如，\n表示换行符，而*则是字面意义上的星号。

*

匹配零个或者多个紧接在前面的表达式。例如，[\t]*可以匹配任意多个空格和tab，也就是空白字符，它能够匹配“”、“<tab><tab>”或者一个空字符串。

+

匹配一个或者多个紧接在前面的表达式。例如，[0-9]+可以匹配数字字符串，像1, 111或者123456，但不能是一个空字符串。

?

匹配零个或者一个紧接在前面的表达式。例如，-[0-9]+匹配一个有符号数字，它带有一个可选的前置负号。

/

选择(alternative)操作符，匹配紧接在前面的表达式或者紧跟在后面的表达式。例如，faith|hope|charity匹配这三个美德中的任意一个。

"..."

所有引号中的字符将基于字面意义被解释。不属于C转义序列的元字符将失去它的特殊意义。从书写风格上来说，比较好的做法是用引号引起所有需要基于字面意义匹配的标点符号。

()

把一系列的正则表达式组成一个新的正则表达式。例如，(01)匹配字符序列01，而a(bc|de)匹配abc或者ade。圆括号在建立带有*、+、?和|的复杂模式时很有用。

/

尾部上下文 (*trailing context*)，匹配斜线前的正则表达式，但是要求其后紧跟着斜线后的表达式。例如，0/1匹配字符串01中的0，但是不会匹配字符串0或者02。斜线后匹配的内容不会被“消耗掉”，它们会返还给输入以便于继续匹配。每个模式只允许一个尾部上下文操作符。

重复匹配的操作符总是针对邻近的表达式，所以abc+匹配带有一个或者多个c的ab。使用圆括号可以帮助你扩大邻近表达式的范围，例如(abc)+匹配一个或者多个abc。

正则表达式的例子

我们可以综合运用这些字符来表达相当复杂而又有用的正则表达式模式。例如，设计一个能够匹配Fortran风格的数值的模式就十分困难，这种数值有一个可选的正负号、一串包括小数点的数字以及可选指数，而指数又由字母E、可选的正负号和数字字符串组成。对于可选的正负号和数字字符串来说，下面的模式已经足够表达：

[-+]?[0-9]+

注意在这里我们的负号出现在[-+]首位，所以它并不是字符范围的意思。

匹配带有一个可选小数点的数字字符串的模式就相对麻烦一些，因为小数点可能出现在数值的头部或者尾部。以下的模式都有一定的问题：

[-+]?[0-9.]+	匹配了更多的可能性，比如 1.2.3.4
[-+]?[0-9]+\.\?[0-9]+	有些没有匹配到，比如 .12或12.
[-+]?[0-9]*\.\?[0-9]+	不能够匹配 12.
[-+]?[0-9]+\.\?[0-9]*	不能够匹配 .12
[-+]?[0-9]*\.\?[0-9]*	可能匹配空字符串和仅有一个小数点的情况

这表明了字符类的?、*和+的组合并不能够匹配带有一个可选小数点的数值。幸运的是选择操作符|可以帮助我们实现这一点，它允许两个版本联合出现在一个模式里，尽管每个版本都并不完备。

[-+]?(([0-9]*\.\?[0-9]+|[0-9]+\.\?)[-+]?(([0-9]*\.\?[0-9]+|[0-9]+\.\?[0-9]*))

第二个例子有一定的二义性，因为对于很多字符串来说，它的两个可选版本都可以匹配，不过对于flex的匹配算法来说这并不是问题（flex允许两个不同的模式匹配相同的输入，这非常有用，但也要求程序员加倍小心）。

现在我们添加一个可选的指数，它的模式很简单：

```
E(+|-)?[0-9]+
```

(我们把正负号字符作为选择项而不是字符类出现，这仅仅是实现的方法不同而已。) 现在我们把两个模式结合起来变成一个Fortran的数值匹配模式：

```
[+-]?([0-9]*\.[0-9]+|[0-9]+\.) (E(+|-)?[0-9]+)?
```

由于指数部分是可选的，所以我们使用圆括号和问号来使它成为整个模式的可选部分。注意我们的模式现在具有嵌套的可选部分，它可以正常工作并且也非常有用。

你会发现大多数flex词法分析器里都具有相同复杂程度的模式。值得再次重申的是复杂模式并不会使词法分析器变慢^(注1)。放心编写你需要匹配的模式吧，flex会帮你处理它。

Flex如何处理二义性模式

大多数flex程序都具有二义性，相同的输入可能被多种不同的模式匹配。flex通过两个简单的规则来解决它：

- 词法分析器匹配输入时匹配尽可能多的字符串。
- 如果两个模式都可以匹配的话，匹配在程序中更早出现的模式。

这两个规则在大量的实践中被证明是行之有效的。我们可以看以下这段词法分析器的代码片段：

```
"+" { return ADD; }
"=" { return ASSIGN; }
"+=" { return ASSIGNADD; }
"if" { return KEYWORDIF; }
"else" { return KEYWORDELSE; }
[a-zA-Z_][a-zA-Z0-9_]* { return IDENTIFIER; }
```

对于前三个模式来说，字符串+=被匹配为一个记号，因为+=比+更长。对于后三个模式来说，只要匹配关键字的模式先于匹配标识符的模式，词法分析器就可以正确地匹配关键字。

注1：有一个特例：如果在flex程序中出现 / 操作符，整个词法分析器将会变得有些慢，因为存在额外的逻辑去回退那些已经匹配但又不能消除的尾部上下文 (trailing context)。不过即使这样，词法分析器也已经足够快了。

上下文相关的记号

在一些语言里，词法分析是上下文相关的。例如，在Pascal中，1.通常是一个浮点数，但是在声明中，1..2就是通过..这个记号来分隔的两个整数。flex提供了起始状态(*start state*)的概念，它可以动态地开启和关闭针对特定模式的识别，这对于处理上述上下文相关的情况十分有用。我们会在本章稍后部分介绍起始状态。

Flex词法分析器中的文件I/O操作

除非你另行指定，否则flex词法分析器总是读取标准输入。实际上，大多数词法分析器都从文件读取输入。我们将修改例1-1里的字数统计程序以使它可以读取文件，就像真正的wc程序所做的那样。

对于由flex和它的前身lex所生成的词法分析器来说，I/O选项在过去的30年里有着广泛的发展。我们有多种不同的方法来管理词法分析器的输入和输出。除非你另做安排，否则词法分析器总是通过名为yyin的文件句柄读取输入。因此为了让它读取一个单一文件，你只需要在第一次调用yylex之前重新设定yyin。在例2-1中，我们修改了例1-1的字数统计程序，使它能够指定一个输入文件。

例2-1：字数统计，从文件中读取输入

```
/* 更像UNIX的wc程序 */
%option noyywrap
%{
int chars = 0;
int words = 0;
int lines = 0;
%}

%%
[a-zA-Z]+ { words++; chars += strlen(yytext); }
\n      { chars++; lines++; }
.      { chars++; }

%
main(argc, argv)
int argc;
char **argv;
{
    if(argc > 1) {
        if(!(yyin = fopen(argv[1], "r"))){
            perror(argv[1]);
            return (1);
        }
    }
}
```

```
yylex();
printf("%8d%8d%8d\n", lines, words, chars);
}
```

与例1-1的不同之处就在第三部分。如果用户在命令行中给出了文件名，主例程会打开这个文件并把相应的文件句柄赋给yyin；否则，yyin将保持未赋值的状态，这种情况下，yylex会自动把stdin赋给它。

flex库

flex和lex总是提供了一个小型的库，叫做-lfl，它定义了默认的main例程和早期lex遗留至今的鸡肋——默认版本的yywrap。

当lex词法分析器到达yyin的结束位置时，它将调用yywrap()。这样做的目的是，当有另一个输入文件时，yywrap可以调整yyin的值并且通过返回0来重新开始词法分析。如果当前就是真正的输入结束位置，它可以通过返回1来完成分析。虽然后续的lex和flex版本都如实地保留了yywrap，但是我现在30年里从未见过一个例子表明yywrap比flex的其他I/O管理特性来得好。事实上，每个人要么使用flex库中默认的yywrap——它总是返回1，要么就是使用一行相同逻辑的代码。flex的较新版本允许你在词法分析器的开头设定%option noyywrap来要求它不使用yywrap，而从现在开始，我们总会这么做。

从测试和快速上手的角度来说，默认的主程序还是有些作用的，不过，在一定规模的flex程序里，你总是会有自己的主例程的，它至少可以设定词法分析器的输入源。下面是库版本的主程序：

```
int main()
{
    while (yylex() != 0) ;
    return 0;
}
```

大多数flex程序使用%option noyywrap和自己的主例程，所以它们并不需要flex库。

读取多个文件

真正的wc程序可以处理多个文件，因此我们实现了改进版本的例2-2，它拥有相同的能力。对于那些只要求从头到尾读完整个输入文件的程序来说，flex提供了yyrestart(f)例程，它使词法分析器读取标准输入输出文件f。

例2-2：字数统计，读入多个文件

```
/* fb2-2 读取一些文件 */
%option noyywrap

%{
int chars = 0;
int words = 0;
int lines = 0;

int totchars = 0;
int totwords = 0;
int totlines = 0;
%}

%%

[a-zA-Z]+ { words++; chars += strlen(yytext); }
\n      { chars++; lines++; }
.        { chars++; }

%%

main(argc, argv)
int argc;
char **argv;
{
    int i;

    if(argc < 2) /* 读取标准输入 */
        yylex();
        printf("%8d%8d%8d\n", lines, words, chars);
        return 0;
    }

    for(i = 1; i < argc; i++) {
        FILE *f = fopen(argv[i], "r");

        if(!f) {
            perror(argv[i]);
            return (1);
        }
        yyrestart(f);
        yylex();
        fclose(f);
        printf("%8d%8d%8d %s\n", lines, words, chars, argv[i]);
        totchars += chars; chars = 0;
        totwords += words; words = 0;
        totlines += lines; lines = 0;
    }
    if(argc > 1) /* 超过一个文件时打印出总数 */
        printf("%8d%8d%8d total\n", totlines, totwords, totchars);
    return 0;
}
```

这个例子打开每个文件，使用`yyrestart()`把打开的文件作为词法分析器的输入，然后调用`yylex()`进行词法分析。就像真正的`wc`那样，如果有多个输入文件时，它也会统计和打印多个输入文件累计的项目。

Flex词法分析器的I/O结构

`flex`词法分析器默认使用标准输入和标准输出，但是就像我们所看到的那样，输入与输出也经常会被重新设定。

Flex词法分析器的输入

就一个包含词法分析器的程序而言，词法分析器的性能通常决定了整个程序的性能。早期的`lex`每次只能从`yyin`中读取一个字符。后来，`flex`提供了灵活的三层输入系统，让程序员可以自行选择来处理任何能够想象到的输入结构。

大多数情况下，`flex`词法分析器从文件或者标准输入（比如用户终端）中读取输入。从文件读取和从终端读取存在着一个微小但是重要的差异——预读机制。如果词法分析器从文件读取，它可以通过大段的读操作来提高工作效率。但是如果它从终端读取，用户可能一次只输入一行，并且期望每行输完后词法分析器能够立刻处理。在这种情况下，速度不再是一个问题，因为即使是一个非常缓慢的词法分析器也比打字能手来得快，所以它完全可以一次只读取一个字符。幸运的是，`flex`词法分析器会检查当前其输入是否来自终端并决定使用哪一种读取方式。^(注2)

`flex`词法分析器使用称为`YY_BUFFER_STATE`的数据结构来处理输入，该结构定义了一个单一的输入源。它包含一个字符串缓冲区以及一些变量和标记。通常它会有一个指向所读取的文件的`FILE*`，但是我们也可以创建一个与文件无关的`YY_BUFFER_STATE`来分析已经在内存中的字符串。

默认的`flex`词法分析器的输入行为大致如下：

```
YY_BUFFER_STATE bp;
extern FILE* yyin;

..... 任何在第一次调用词法分析器之前所需要做的事情

if(!yyin) yyin = stdin; 默认输入设备是stdin
bp = yy_create_buffer(yyin,YY_BUF_SIZE );
    YY_BUF_SIZE 由flex定义，大小通常是16K
yy_switch_to_buffer(bp); 告诉它使用我们刚刚创建的缓冲区
```

注2：交互式词法分析器还有一个是否需要向前查看的问题，我们会在本章稍后继续讨论。庆幸的是，`flex`对这种情况也处理得很好。

`yylex();` 或者是`yyparse()`或其他对词法分析器的调用

如果`yyin`还没有被设置，我们就把`stdin`设置给它。然后我们使用`yy_create_buffer`来创建一个读取`yyin`的新缓冲区，通过`yy_switch_to_buffer`来让词法分析器从缓冲区读入，接着开始分析。

当需要顺序读取多个文件时，我们每打开一个文件就调用一次`yyrestart(fp)`，把词法分析器的输入切换到标准输入输出文件`fp`。如果新的文件已经被赋给`yyin`，`YY_NEW_FILE`就等同于`yyrestart(yyin)`。^(注3)

其他一些函数也可以用来创建缓冲区，包括`yy_scan_string("string")`（分析以空字符结尾的字符串）和`yy_scan_buffer(char *base, size)`（分析长度确定的数据流）。本章稍后我们也会看到一些函数可以维护缓冲区的堆栈，它们对于处理嵌套的包含文件是比较方便的。这些函数都会在第5章列出，在“输入管理”这一节。

最后，出于最大灵活性的考虑，你可以重新定义flex用于读取输入到当前缓冲区的宏：

```
#define YY_INPUT(buf,result,max_size) ...
```

每当词法分析器的输入缓冲区为空时，它就调用`YY_INPUT`，`buf`是缓冲区，`maxsize`是缓冲区的大小，而`result`则用来放置实际读取的长度，如果位于EOF，那么`result`就等于0（因为这是一个宏，所以使用`result`而不是`result*`）。由于重新定义`YY_INPUT`的能力先于附加的`YY_BUFFER_STATE`，所以过去大多数人使用前者来实现，现在而言他们用后者来实现会更好。目前，自定义`YY_INPUT`主要用在事件驱动系统里，这种系统的输入来自于其他一些内容，而这些内容无法预先载入字符串缓冲区且无法支持标准输入输出。

当词法分析器达到文件末尾时，它将匹配到伪模式`<<EOF>>`，这个模式通常用来进行清理工作、切换到其他文件等等。

flex提供了两个在动作代码中比较有用的宏，`input()`和`unput()`。每次`input()`的调用将返回输入流的下一个字符。它可以帮助我们读取一小段输入而不用定义相应的模式。每次对`unput(c)`的调用把字符`c`推回到输入流。它与`/`操作符功能相似，可以向前查看输入但是并不做处理。

最后来总结一下，输入管理的三个层次是：

- 设置`yyin`来读取所需文件

注3：如果前一个`yyin`已经被读完，`YY_NEW_LINE`并不一定需要，但是使用它对程序严谨性有好处。特别是当词法分析器或者语法分析器出现错误而中途返回时，整个文件并没有被读完。

- 创建并使用YY_BUFFER_STATE输入缓冲区
- 重定义YY_INPUT

Flex词法分析器的输出

词法分析器的输出管理比输入管理简单得多，而且完全是可选的。同样可以追溯到最早的lex版本，除非你另行设定，否则flex总会执行一条默认的规则：所有没有被匹配的输入都拷贝到yyout。

```
. ECHO;  
#define ECHO fwrite( yytext, yylen, 1, yyout )
```

这对于那些仅处理一部分输入而保持剩余部分不变的flex程序来说可能有些作用，比如在例1-2中的英式英语到美式英语的翻译器，但在大多数情况下它更容易导致一些错误。flex允许你在词法分析器顶端设置%option nodefault，使它不要添加默认的规则，这样当输入无法被给定的规则完全匹配时，词法分析器可以报告一个错误。我建议词法分析器总是使用nodefault，并且在必要的情况下包含自己的默认规则。

起始状态和嵌套输入文件

我们将通过一个简单的程序来检验一下我们的flex I/O知识，这个程序需要处理嵌套的包含文件（include file）并且打印出它们。为了让程序变得有趣一些，我们同时打印出这些文件中每一行的行号。因此，这个程序需要维护一个包含嵌套输入文件和行号的堆栈，在每次遇到一个#include时压入当前文件和行号信息，在处理完包含文件后再把它们从堆栈弹出。

我们还会使用一个很强大的flex特性——起始状态（start state），它允许我们指定在特定时刻哪些模式可以被用来匹配。在靠近文件顶端的%行把FILE定义为起始状态，它将在我们寻找#include语句中的文件名时被使用。在任何时候，这个词法分析器都处在一个起始状态中，并且只匹配这个状态所激活的那些模式。实际上，起始状态定义了一个不同的词法分析器，拥有它自己的规则。

你可以定义任意多的起始状态，在这个程序中，除了flex本身会定义的INITIAL状态之外，我们只需要再额外定义一个状态。当一个模式紧随在尖括号括起的起始状态名字之后，表示这个模式只在该状态中被激活。%把FILE标记为一个独占（exclusive）的起始状态，这意味着当该状态被激活时，只有这个状态中的模式才可以进行匹配。（%可以用来声明包含（inclusive）的起始状态，它允许未标记为任何状态的模式也可以进行匹配。独占的状态通常更有用。）在动作代码里，宏BEGIN用来切换到另外一个起始状态。例2-3是一个词法分析器的代码框架，它能够处理包含文件。

例2-3：包含文件的框架

```
/* fb2-3 包含文件的框架 */
%option noyywrap
%x IFILE

%{
struct bufstack {
    struct bufstack *prev; /* 上一个文件信息 */
    YY_BUFFER_STATE bs; /* 保存的缓冲区 */
    int lineno; /* 保存的行号 */
    char *filename; /* 文件名 */
    FILE *f; /* 当前文件 */
} *curbs = 0;

char *curfilename; /* 当前输入文件的名称 */

int newfile(char *fn);
int popfile(void);
%}
%%

匹配#include语句直到引号或者<
^#"[" \t]*include[ \t]*[\"<] { BEGIN IFILE; }

处理文件名直到结束引号、>或者行结束符
<IFILE>[^ \t\n\>]+ {
    { int c;
        while((c = input()) && c != '\n') ;
    }
    yylineno++;
    if(!newfile(yytext))
        yyterminate(); /* no such file */
    BEGIN INITIAL;
}

处理IFILE状态中错误输入的情况
<IFILE>.|\n { fprintf(stderr, "%d bad include line\n", yylineno);
                  yyterminate();
}

文件结束时弹出文件堆栈，如果是最外层文件就结束
<<EOF>> { if(!popfile()) yyterminate(); }

在每一行的开始打印出行号
并且每遇到一个\n时就把行号加1
^. { fprintf(yyout, "%d %s", yylineno, yytext); }
^\\n { fprintf(yyout, "%d %s", yylineno++, yytext); }
\\n { ECHO; yylineno++; }
. { ECHO; }

%%

main(int argc, char **argv)
{
    if(argc < 2) {
        fprintf(stderr, "need filename\n");
    }
}
```

```
        return 1;
    }
    if(newfile(argv[1]))
        yylex();
}

int
newfile(char *fn)
{
    FILE *f = fopen(fn, "r");
    struct bufstack *bs = malloc(sizeof(struct bufstack));

    /* 如果文件打开失败或者没有足够空间时就退出 */
    if(!f) { perror(fn); return 0; }
    if(!bs) { perror("malloc"); exit(1); }

    /* 记住当前状态 */
    if(curbs)curbs->lineno = yylineno;
    bs->prev = curbs;

    /* 建立当前文件信息 */
    bs->bs = yy_create_buffer(f, YY_BUF_SIZE);
    bs->f = f;
    bs->filename = fn;
    yy_switch_to_buffer(bs->bs);
    curbs = bs;
    yylineno = 1;
    curfilename = fn;
    return 1;
}

int
popfile(void)
{
    struct bufstack *bs = curbs;
    struct bufstack *prevbs;

    if(!bs) return 0;

    /* 删除当前文件信息 */
    fclose(bs->f);
    yy_delete_buffer(bs->bs);

    /* 切换回上一个文件 */
    prevbs = bs->prev;
    free(bs);

    if(!prevbs) return 0;

    yy_switch_to_buffer(prevbs->bs);
    curbs = prevbs;
    yylineno = curbs->lineno;
    curfilename = curbs->filename;
    return 1;
}
```

}

程序的第一部分定义了起始状态并且声明了bufstack数据结构，bufstack拥有保存的输入文件列表中的一个项目。

第一个模式匹配#include语句，直到遇到文件名之前的双引号为止。这个模式允许语句中可能存在的空白字符。如果第一个模式得到匹配，词法分析器会切换到FILE状态，读取接下来的文件名。在FILE状态中，第二个模式匹配一个文件名，直到遇到结束引号、空白字符或者行结束符。然后文件名被传递给函数newfile，这将导致当前输入文件被压入堆栈，同时设置下一层的输入，不过在此之前我们需要处理一些#include行剩余的内容。一种方法是使用另外一个起始状态和模式来读取留下的部分，但这样做有点麻烦，因为随后的动作代码会转向新的包含文件，起始状态和模式将不得不在新的包含文件被处理完之后再使用。相反，另外一种基于函数input()的方法就显得简单的多。我们使用一个短循环来读完该行剩余的部分。这样，当词法分析器从包含文件返回时，它可以直接从当前文件的下一行开始继续处理。

由于独占的起始状态定义了一个自己的小型词法分析器，因此这个小型词法分析器必须处理任何可能的输入。下一个模式用来匹配在双引号后没有文件名的不规范#include行的情况。它只需要打印错误信息并调用宏yyterminate()来立即从词法分析器返回^(注4)。这里#include的定义是相当随意的，而且我们没有费力气来检查文件名前后的标点符号，也没有考虑是否会有额外的字符出现在文件名之后。不过，编写检查这些问题和诊断错误的代码并不困难，对于一个更完善的版本来说值得这么做^(注5)。

下面是特殊模式<<EOF>>，它匹配输入文件的结束。我们调用后面定义的popfile()来回到前一个输入文件。如果它返回0，则意味着这是最后一个文件，我们可以结束程序；否则，词法分析器将从上次的位置继续分析。

最后4个模式打印出前面带有行号的每行内容。flex提供了一个叫做yylineno的变量，它被用来记录行号，我们可以直接使用它。模式^.匹配任意一行的首字符，这样语义动作就可以打印出行号和这个字符。点号不能够匹配换行符，所以^\n被用来匹配首字符是换行符的情况，这种情况也就是一个空行，相应的代码打出行号和换行符，然后把行号加1。对于不在行首的换行符和其他字符，可以通过ECHO打印出来，并对每个换行符把行号加1。

例程newfile(fn)可以在读取名为fn的文件的同时保存前面所有输入文件的信息。它通

注4： 它返回YY_NULL，定义为0，bison语法分析器把它解释为输入的结束。

注5： 或者换而言之，错误诊断将作为一个练习留给读者。

过维护一个bufstack结构的链表来实现这一点，每个bufstack都有一个例程指向前一个bufstack的链接，其中包含了保存的yylineno和文件名的信息。这个例程打开文件，创建并切换到flex的缓冲区，而且保存前一个打开的文件、文件名和缓冲区。（这个程序里的文件名在文件被打开后就没有再使用，但我们本章稍后的程序会用到。）

例程popfile与newfile做的正好相反。它关闭打开的文件，删除当前的flex缓冲区，然后从前一个堆栈项恢复缓冲区、文件名和行号。注意当它恢复前一个缓冲区时它并没有调用yyrestart()，如果它这么做的话，它将会丢失已经读入缓冲区的输入。

这是一个简单但是相当典型的处理包含文件的例子。虽然flex可以通过例程yypush_buffer_state和yypop_buffer_state来管理输入缓冲区的堆栈，但我并不觉得它们有用，因为它们无法在堆栈的文件中关联其他信息。

符号表和重要语汇索引生成器

几乎每个flex和bison程序都使用符号表（*symbol table*）来记录输入中使用的名称。我们用一个非常简单的程序作为示例，它生成重要语汇索引（*concordance*），也就是输入中出现的各个单词所在行号的列表，然后我们把它修改为一个能够读取C源代码的交叉引用程序。

管理符号表

符号表的相关知识早就被很多编译相关的著作长篇大论过，我并不希望这本书也成为其中之一。重要语汇索引的符号表只需要简单地记录每个单词以及它们所在的文件和行号。例2-4是重要语汇索引生成器的声明部分。

例2-4：重要语汇索引生成器

```
/* fb2-4 文本重要语汇索引 */
%option noyywrap nodefault yylineno case-insensitive

/* 符号表 */
%{
    struct symbol { /* 单词 */
        char *name;
        struct ref *reflist;
    };

    struct ref {
        struct ref *next;
        char *filename;
        int flags;
        int lineno;
    };
}
```

```

/* 固定大小的简单的符号表 */
#define NHASH 9997
struct symbol symtab[NHASH];

struct symbol *lookup(char*);
void addref(int, char*, char*, int);

char *curfilename; /* 当前输入文件的名称 */

%}
%%

```

%option行有两个我们以前没有见过的选项，它们都相当有用。选项%yylineno告诉flex定义一个名为yylineno的整型变量来保存当前行号。每次词法分析器读到一个换行符时，yylineno就加1；如果词法分析器后退一个换行符时（我们后面会讨论到这种情况），yylineno就减1。不过你必须在每个文件的开头把yylineno初始化为1，对于包含文件的情况，你还需要额外的保存和恢复动作。即使有这样一些限制，它仍然比手动维护当前行号来得简单。（这个例子中我们只有一个需要匹配\n的模式，所以不太容易产生错误，但是有多个匹配\n的模式是很常见的情况，如果其中有一个忘了更新行号的话就会造成难以调试的缺陷。）

另一个新的选项是case-insensitive，它要求flex生成一个大小写无关的词法分析器。也就是像abc这样的模式将可以匹配abc、Abc、ABc、AbC等等。你不需要做额外的工作来实现这一点，特别是在yytext中被匹配的字符串并不会改变它的大小写。

这里的符号表也就是一个symbol结构的数组，每个结构包含一个指向名称的指针（也就是在重要语汇索引中的单词）以及引用列表。引用是一个包含行号和指向文件名的指针的链表。我们也定义了curfilename，一个指向当前文件的名称的静态指针，它将在添加引用时被使用。

```

%_
/*重要语汇索引生成器的规则 */
/* 忽略常用单词 */
a |
an |
and |
are |
as |
at |
be |
but |
for |
in |
is |
it |
of |
on |

```

```

or |
that |
the |
this |
to      /* 忽略 */

[a-z]+(\'(s|t))?      { addref(yylineno, curfilename, yytext, 0); }
.\n                  /* 忽略其他东西 */
%%

```

重要语汇索引通常并不索引常见的短词，因此第一个模式将忽略它们。仅包含竖线的语义动作告知它与下一条规则的语义动作一致。最后一个忽略单词的语义动作并不做任何事情，这样就忽略了单词。

下一条规则是词法分析器的主体，它大致匹配一个英语单词。这个规则由一连串的字母，也就是[a-z]+，后接可选的单引号以及s或者t（用来匹配类似于*owner's*和*can't*的单词）组成。每个匹配的单词都和当前文件名以及行号一起传递给稍后会讲解到的**addref()**。

最后的模式匹配所有前面的模式没有匹配的字符。

注意这个词法分析器是有二义性的，但是flex解决二义性的规则可以满足我们的要求。flex匹配尽可能长的字符串，所以单词*toad*将被主体的单词模式而不是*to*所匹配。当两个模式都可以匹配的时候，flex将使用程序中更早出现的那个，这也是我们把忽略规则前置和全匹配（catchall）后置的原因。

```

/* 重要语汇索引主例程 */
main(argc, argv)
int argc;
char **argv;
{
    int i;

    if(argc < 2) { /* 从标准输入读取 */
        curfilename = "(stdin)";
        yylineno = 1;
        yylex();
    } else
        for(i = 1; i < argc; i++) {
            FILE *f = fopen(argv[i], "r");

            if(!f) {
                perror(argv[1]);
                return (1);
            }
            curfilename = argv[i]; /* 用于addref */

            yyrestart(f);
            yylineno = 1;
    }
}

```

```
    yylex();
    fclose(f);
}

printrefs();
}
```

这个主例程和读取多个文件的字数统计程序十分相似。它依次打开每个文件，使用`yyrestart`来安排读取文件，然后调用`yylex`。额外的工作是把`curfilename`设置为文件名，便于构造引用列表，并且为每个文件设置`yylineno`为1（否则，行号将会从一个文件到另一个文件进行累加，这在有些场合可能适用，但却不是这儿）。最后，`printrefs`把符号表按字母顺序排列并打印出引用信息。

使用符号表

词法分析器的代码部分包含了一个简单的符号表例程、一个添加单词到符号表的例程以及一个在所有输入被运行后打印出重要语汇索引的例程。

这儿的符号表虽然小但是功能齐全。它带有一个例程，`lookup`，获取一个字符串并返回用于这个字符串名称的符号表条目的位置，如果名字不存在的话就创建它。名字查找技术称为线性探测的哈希算法 (*hashing with linear probing*)。它使用哈希函数把字符串转化为符号表中的条目号，然后检查这个条目，如果该条目被一个不同的符号占用时就顺序检查下一个条目直到有可用的为止。

哈希函数也很简单：对每个字符，首先用9乘以前一个哈希值，然后再`xor`这个字符，所有的操作都基于无符号数以避免溢出的情况。查找例程通过把哈希值对符号表的大小（一个素数）取模来得到符号表条目的索引。

```
/* 求符号的哈希值 */
static unsigned
symhash(char *sym)
{
    unsigned int hash = 0;
    unsigned c;

    while(c = *sym++) hash = hash*9 ^ c;

    return hash;
}

struct symbol *
lookup(char* sym)
{
    struct symbol *sp = &syntab[symhash(sym)%NHASH];
    int scount = NHASH; /* 查找次数 */

    while(--scount >= 0) {
```

```

    if(sp->name && !strcmp(sp->name, sym)) return sp;

    if(!sp->name) { /* 新条目 */
        sp->name = strdup(sym);
        sp->reflist = 0;
        return sp;
    }

    if(++sp >= symtab+NHASH) sp = symtab; /* 尝试下一个条目 */
}
fputs("symbol table overflow\n", stderr);
abort(); /* 所有条目都试过了，符号表已满 */
}

```

注意每当`lookup`创建一个新的条目时，它就调用`strdup`来产生字符串的拷贝并放到符号表条目中。`flex`和**bison**程序经常会存在难以跟踪的字符串存储管理问题，这是因为大家容易忘记一个事实，那就是`yytext`中的字符串会在下个词法记号被分析时被替换掉。

这个简单的哈希函数和查找例程工作得很好。我使用一个评测版本的重要语汇索引程序来分析一组文本文件，这些文件有4 429个不同的单词，该程序总共执行了70 775次查找。查找例程平均进行了1.32次探测，基本接近理想值1.0次。

```

void
addref(int lineno, char *filename, char *word, int flags)
{
    struct ref *r;
    struct symbol *sp = lookup(word);

    /* 不用复制相同的行和文件 */
    if(sp->reflist &&
       sp->reflist->lineno == lineno &&
       sp->reflist->filename == filename) return;

    r = malloc(sizeof(struct ref));
    if(!r) {fputs("out of space\n", stderr); abort(); }
    r->next = sp->reflist;
    r->filename = filename;
    r->lineno = lineno;
    r->flags = flags;
    sp->reflist = r;
}

```

下一个例子是`addref`，词法分析器调用它来添加对特定单词的引用，这个例程基于引用结构的链表来实现。为了让报告可以变得短一些，如果符号已经有一个对相同行号和文件名的引用，则它不会添加这个引用。注意在这个例程中，我们并不创建文件名的拷贝，因为我们知道这个字符串在调用方并不会改变。我们也不拷贝单词，因为`lookup`会帮我们处理它。每个引用都有一个`flag`值，它会在下个例子中被使用。

```
/* 打印引用信息
```

```

* 按字母顺序排列符号表
* 然后翻转每个条目的reflist使它变成正向排序
* 并进行打印
*/
/* 用于排序的辅助函数 */
static int
symcompare(const void *xa, const void *xb)
{
    const struct symbol *a = xa;
    const struct symbol *b = xb;

    if(!a->name) {
        if(!b->name) return 0;      /* 两个符号都为空 */
        return 1;                  /* 这样可以把空符号放到末尾 */
    }
    if(!b->name) return -1;
    return strcmp(a->name, b->name);
}

void
printrefs()
{
    struct symbol *sp;

    qsort(symtab, NHASH, sizeof(struct symbol), symcompare); /* 对符号表进行排序 */

    for(sp = symtab; sp->name && sp < symtab+NHASH; sp++) {
        char *prevfn = NULL; /* 前一个打印的文件名，用来跳过重复的文件 */

        /* 翻转引用列表 */
        struct ref *rp = sp->reflist;
        struct ref *rpp = 0;      /* 前一个引用 */
        struct ref *rpn; /* 后一个引用 */

        do {
            rpn = rp->next;
            rp->next = rpp;
            rpp = rp;
            rp = rpn;
        } while(rp);

        /* 打印单词和它的引用 */
        printf("%10s", sp->name);
        for(rp = rpp; rp; rp = rp->next) {
            if(rp->filename == prevfn) {
                printf(" %d", rp->lineno);
            } else {
                printf(" %s:%d", rp->filename, rp->lineno);
                prevfn = rp->filename;
            }
        }
        printf("\n");
    }
}

```

最后一部分例程对符号表进行排序和打印。符号表的创建顺序依赖于哈希函数，这种顺序并不适合人们阅读，所以我们使用标准`qsort`函数对符号表按字母顺序排序。由于符号表不一定被用完，所以排序函数把没有使用的条目放到使用的条目的后面，这样排序后的使用的条目在符号表的前端。

接着`printrefs`自上而下打印符号表中对每个单词的引用。引用被存储在一个链表中，但是由于每个引用被压入链表前端，实际上是倒序的。所以我们翻转整个链表来使它变成顺序排列，然后进行打印^(注6)。为了让重要语汇索引可读性更好一些，我们仅在文件名有变化时才打印文件名。同时我们只是比较文件名的指针，因为照理来说用于相同文件的条目将指向同一个文件名拷贝。

C语言交叉引用

本章最后的例子将运用到我们前面学习的所有知识，它是一个比较实际的C语言交叉引用程序（例2-5）。这个例子使用嵌套的输入文件来处理`#include`语句，用起始状态来处理包含文件和注释，用一个词汇标记来表明某个符号是作为定义还是引用出现，以及用一个符号表来记录所有的一切。

例2-5：C交叉引用程序

```
/* fb2-5 C交叉引用 */
%option noyywrap nodefault yylineno

%x COMMENT
%x IFILE

/* 一些复杂的命名模式 */
/* 通用字符名 */
UCN (\u[0-9a-fA-F]{4}|\U[0-9a-fA-F]{8})
/* 浮点数指数部分 */
EXP ([Ee][-+]?[0-9]+)
/* 整数长度 */
ILEN ([Uu](L|l|LL|ll)?|(L|l|LL|ll)[Uu]?)
```

选项部分与重要语汇索引例子的基本一致，但是大小写无关的选项被去掉了，因为C语言需要区分大小写。两个独占的起始状态分别为忽略C语言注释所需的`COMMENT`和处理`#include`所需的`IFILE`。

接着是三个命名模式，在后面的规则部分会用到。有种flex编程风格是把所有子模式都加以命名，例如，用`DIGIT`命名`[0-9]`。我并不觉得这种风格有效，但是对于相当复杂

注6： 这种基于错误的顺序来建立列表随后再做翻转的技巧是比较方便的，我们会在建立语法分析树的时候再次看到它。这种技巧被证明十分有效，因为翻转的步骤只需要过一遍列表而不需要用于每个条目的额外空间。

而且在其他大模式中都需要使用的模式来说，命名是比较有用的。第一个模式匹配通用字符名（Universal Character Name，UCN），一种在字符串和标识符中放置非ASCII码字符的比较笨拙的方法。通用字符名由跟在\u后的4个十六进制数或者跟在\U后的8个十六进制数表示。第二个模式是浮点数的指数部分，一个可以大写或者小写的字母E、一个可选的正负号以及一串数字。第三个模式匹配整型常量的长度和类型后缀，可选的U代表无符号数，可选的L或者LL代表常量长度，这两者都是大小写无关的，顺序也可以颠倒。每个模式都通过圆括号括起以避免与旧版的lex/flex不兼容。当flex使用命名模式时，它总是认为该模式是带有圆括号的，但lex并不这么认为，因此导致了一些不容易发现的错误。

```
/* 符号表 */
%{
    struct symbol { /* 变量名 */
        struct ref *reflist;
        char *name;
    };

    struct ref {
        struct ref *next;
        char *filename;
        int flags; /* 01 - definition */
        int lineno;
    };
};

/* 简单的固定大小的符号表 */
#define NHASH 9997
struct symbol symtab[NHASH];

struct symbol *lookup(char*);
void addref(int, char*, char*, int);

char *curfilename; /* 当前输入文件的名称 */

/* 包含文件的堆栈 */
struct bufstack {
    struct bufstack *prev; /* 前一条目 */
    YY_BUFFER_STATE bs; /* 保存的缓冲区 */
    int lineno; /* 保存的该文件中的行号 */
    char *filename; /* 文件名 */
    FILE *f; /* 当前文件 */
} *curbs;

int newfile(char *fn);
int popfile(void);

int defining; /* 名称是否是定义 */
%}
```

接下来的部分我们应该十分熟悉。符号表与前一个例子一致。文件堆栈与例2-3一致。

最后是一个新的变量，`defining`，当一个名称作为定义而不是引用出现时被设置。

接下来是规则部分，它比我们前面所见过的任何一个规则部分都要长，它更像一个实际意义上的flex程序。很多记号匹配规则既长又复杂，但通过用C语言来直译其BNF描述的方法会比较容易实现。虽然flex并不能够处理通常的BNF（你需要bison来实现），但这里的记号被特意设计以便于匹配正则表达式，所以直译其相应的BNF就可以。

```
%%
/* 注释 */
/*"
<COMMENT>"*/"      { BEGIN(COMMENT); }
<COMMENT>([^\*]|\\n)+|. { BEGIN(INITIAL); }
<COMMENT><<EOF>> { printf("%s:%d: Unterminated comment\\n",
    curfilename, yylineno); return 0; }

/* C++ 注释，一种常用的扩展 */
"//".*\n
```

独占的起始状态使该词法分析器更容易匹配注释。第一个规则在看到/*时激活COMMENT状态，而第二个规则在遇到*/时切换回正常的INITIAL状态，第三个规则匹配两者之间的一切字符。虽然模式的复杂度并不会影响flex词法分析器的速度，但是匹配一个长模式必然比匹配若干个短模式来得快。所以，这个规则可以只有.\n，不过([^*]|\\n)+却能够一次就匹配一长串文本。注意这个规则把*排除在外，因此第二个规则才能够匹配*/。规则<COMMENT><<EOF>>发现和报告没有终结的注释。下面是赠送的规则，它可以匹配C++格式的注释，一个常用的对C编译器的扩展。

C语言注释的模式

虽然单一的flex模式也可以匹配C语言注释，但是这样做并不是一个好主意。可以参考一下这个模式：

```
\/*([^\*]|\*+[^\*/])*\\\*/
```

它匹配开始一个注释的两个字符，\/*；接着是一连串的不带星号的字符或者是尾部不带星号和斜线的一连串星号，([^*]*+[^*/])*；最后是一个或者多个星号再加上一个结束的斜线，*/。

有两个原因让我们更倾向于使用多个模式和起始状态。一个原因是注释可能很长，在注释占据了若干页代码时会有问题，因为flex的记号有一定的输入缓冲的长度限制，通常是16K（这种错误不太会被测试到）。另外一个原因是多个模式更容易发现和诊断没有结束的注释。虽然通过修改前面的模式也可以匹配到，但这更容易发生超过16K长度限制的情况。

```
/* 声明关键字 */
_Bool |
_Complex |
_Imaginary |
auto |
char |
const |
double |
enum |
extern |
float |
inline |
int |
long |
register |
restrict |
short |
signed |
static |
struct |
typedef |
union |
unsigned |
void |
volatile { defining = 1; }

/* 关键字 */
break
case
continue
default
do
else
for
goto
if
return
sizeof
switch
while
```

下面是匹配所有C语言关键字的模式。对于引入声明或者定义的关键字我们将设置**defining**标志，对其他的关键字不需要做任何事情。

另外一种处理关键字的方法是把它们放到符号表中并加上标志以表明它们是关键字，在词法分析器中把它们作为普通的符号来看待，并且通过符号表的查找来识别它们。这种方法基本上就是以空间换时间。把它们直接放到词法分析器中会使词法分析器变大，但是并不需要额外的查找就可以识别出这些关键字。对于现代计算机来说，词法分析器本身的符号表的大小很少会成为问题，所以把它们放到词法分析器中会比较容易，而即使加了这些关键字，这个程序本身的符号表也不会超过18KB。

```

/* 常量 */

/* 整数 */
0[0-7]*{ILEN}?
[1-9][0-9]*{ILEN}?
0[Xx][0-9a-fA-F]+{ILEN}?

/* 十进制浮点数 */
([0-9]*\.[0-9]+|[0-9]+\.\.{EXP}?[\fFL]?
[0-9]+{EXP}[\fFL]?

/* 十六进制浮点数 */
0[Xx]([0-9a-fA-F]*\.[0-9a-fA-F]+|[0-9a-fA-F]+\.\.?)[Pp][-+]?[0-9]+[\fFL]?

```

接下来是数字的模式。C语言的数字格式异常复杂，不过我们的命名模式ILEN和EXP可以使处理变得简单。我们为整数、八进制数、十进制数和十六进制数分别定义了一个模式，每个模式都带有可选的正负号和类型后缀。（虽然它们可以实现为一个大模式，但这样设计更简单，而且处理速度是一样的。）

十进制浮点数很像本章前面讲到的Fortran例子。第一个模式匹配包含小数点和可选指数的数字。第二个模式匹配不带小数点的数字，这种情况下浮点指数是必需的（没有指数部分，它就和整数一样了）。

最后是十六进制格式的浮点数，它带有一个用P而不是E分隔的十进制指数，所以它不能使用命名模式EXP。

```

/* 字符常量 */
\'([^\\"]|\\["?\\abfnrtv]|\\[0-7]{1,3}|\\[Xx][0-9a-fA-F]+|{UCN})+\'
L?\"([^\\"]|\\["?\\abfnrtv]|\\[0-7]{1,3}|\\[Xx][0-9a-fA-F]+|{UCN})*\"

```

下面是非常晦涩的字符常量和字符串字面量。字符常量由一个单引号开始，其后可以是一个或者多个非单引号和反斜线的普通字符，或者是通过反斜线转义的字符如\n，或者是以转义符开始的最多三个八进制数字，或者是以转义符开始的一定数量的十六进制数字，或者是通用字符名，最后都以单引号结尾。字符串字面量基本类似，除了它通过双引号引起，并且带有一个可选的前缀L来表明一个宽字符串，而且它也可以为空。（注意字符常量和字符串字面量最后的+和*的区别。）

```

/* 标点符号 */
"|" "<%" ";" { defining = 0; }

"[" "|" "]" "(" ")" "|" "{" "}" "|" "." "|" "-" |
"++" "|" --" "|" &" "|" *" "+" "|" -" "|" ~" "|" !
"/" "|" %" "|" <<"|">"|"<"|">"|"<="|">"|"=="|"!="|"^"|"|"&"|"||"|
"?|" ":"|" ;"|" ..."|
"="|"*="|" /="|"%"|"+"|" -="|" <<="|">="|" &="|" ^="|" |="

```

```
" , " | "#" | "##"  
"<:" | ":" >" | "%>" | "%:" | "%:%:"
```

C语言把操作符和标点都称为标点符号 (*punctuator*)。这里我们额外定义了表明变量名或者函数定义结束的三种模式，其余的无需特别处理。

```
/* 标识符 */  
[_a-zA-Z] | {UCN} ) ( [_a-zA-Z0-9] | {UCN} ) * {  
    addref(yylineno, curfilename, yytext, defining); }  
  
/* 空白字符 */  
[ \t\n]+  
  
/* 续行符 */  
\$
```

C语言的标识符由字母、下划线与通用字符名开头，后面是可选的更多字母、下划线、通用字符名以及数字。当我们遇到一个标识符时，我们就在符号表中添加一个对其的引用。

其他两个模式匹配空白字符和行尾的反斜线。

```
/* 预处理器 */  
#" " *if.*\n  
#" " *else.*\n  
#" " *endif.*\n  
#" " *define.*\n  
#" " *line.*\n  
  
/* 识别包含文件 */  
^#" [ \t]*include[ \t]*["<"] { BEGIN IFILE; }  
<IFILE>[^>"]+ {  
    { int c;  
        while((c = input()) && c != '\n') ;  
    }  
    newfile(strdup(yytext));  
    BEGIN INITIAL;  
}  
  
<IFILE>.|\\n { fprintf(stderr, "%s:%d bad include line\\n",  
    curfilename, yylineno);  
    BEGIN INITIAL;  
}  
  
<<EOF>> { if(!popfile()) yyterminate(); }
```

在交叉引用程序中并没有一个大家都满意的解决方案来处理预处理器命令。一种方法是让源程序首先通过预处理器的处理，但这意味着交叉引用程序无法看到预处理器所处理的符号，而只有展开过的代码。这里我们采用了一个非常简单的方法，只处理#include而忽略其他的预处理器命令（更合理的方法是读取#define和#if之后的内容并且添加它们

到交叉引用）。处理包含文件的代码与本章前面的例子基本一致，只是针对C语言的包含语法做了微小的调整。一个模式匹配`#include`直到文件名前的引号或者`<`，然后切换到`I FILE`状态。下一个模式收集文件名，跳过该行剩余部分，接着处理新文件。这段代码并不健壮，因为它并没有检查文件名匹配前后的标点符号。下面两个模式检查是否有文件名以及在每个包含文件结束后回到前一个文件。

```
/* 无效字符 */
.     { printf("%s:%d: Mystery character '%s'\n",
    curfilename, yylineno, yytext);
}
%%
```

最后一个模式很简单，它匹配所有前面的模式没有处理的字符。由于这个模式覆盖了有效C程序中可能出现的每个记号，所以事实上它不应该被匹配到。

代码部分的内容与我们所看过的代码相似。前两个例程`symhash`和`lookup`和前例的版本完全一致，所以不再罗列。例程`addref`也没有差别。例程`printrefs`有一些微小的差异，它会为标志为定义的引用打印一个*。

```
void
printrefs()
{
    struct symbol *sp;

    qsort(symtab, NHASH, sizeof(struct symbol), symcompare); /* 对符号表进行排序 */

    for(sp = symtab; sp->name && sp < symtab+NHASH; sp++) {
        char *prevfn = NULL; /* 上一个打印的文件名，用来跳过重复的文件 */

        /* 翻转引用列表 */
        struct ref *rp = sp->reflist;
        struct ref *rpp = 0; /* 前一个引用 */
        struct ref *rpn; /* 后一个引用 */

        do {
            rpn = rp->next;
            rp->next = rpp;
            rpp = rp;
            rp = rpn;
        } while(rp);

        /* 打印单词和它的引用 */
        printf("%10s", sp->name);
        for(rp = rpp; rp; rp = rp->next) {
            if(rp->filename == prevfn) {
                printf(" %d", rp->lineno);
            } else {
                printf(" %s:%d", rp->filename, rp->lineno);
                prevfn = rp->filename;
            }
        }
    }
}
```

```
    if(rp->flags & 01) printf("*");
}
printf("\n");
}
```

例程newfile和popfile与本章早前的例子一样，所以这里不再重复。在这个程序里，当newfile无法打开一个文件时，它只是打印一条错误消息；它在能够打开文件时返回1，反之则返回0；而规则部分处理包含文件的代码将继续处理下一行。对于正常的编译器来说，这并不是一个好的做法。但对于交叉引用程序而言，它可以起到一定程度上的合理效果，与当前程序在同一目录的包含文件可以被正常处理，而在其他目录中的库文件则被跳过。

最后，主程序调用newfile，如果成功的话，为每个文件执行yylex。

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

    if(argc < 2) {
        fprintf(stderr, "need filename\n");
        return 1;
    }
    for(i = 1; i < argc; i++) {
        if(newfile(argv[i]))
            yylex();
    }

    printrefs();
    return 0;
}
```

这就结束了我们第一个现实意义上的大型flex程序。它有一些相当复杂的模式，一定复杂程度的文件I/O操作，并且对读到的文本进行了一些处理。

练习

1. 例2-3一次匹配一个字符。为什么它不使用类似`^.*\n?`的模式来一次匹配一行呢？请给出一个模式或者几个模式的结合，它能够匹配一大块文本，请记住`^.*`不能工作的原因。
2. 重要语汇索引程序区分了文本的大小写。请加以修改使它可以忽略大小写。你并不需要拷贝单词就可以做到这一点。在symhash()中，使用`tolower`来哈希字符串的小写版本，并且使用`strcasecmp()`来比较单词。

3. 在重要语汇索引程序和交叉引用程序中的符号表例程使用了固定大小的符号表，它在符号表填满时就会终止。请修改相应的例程以避免这种情况的发生。有两种标准技术可以实现改变哈希表的大小，它们是链地址法（*chaining*）和再哈希法（*rehashing*）。链地址法把哈希表变成指针表，每个指针指向一个符号条目列表。查找将顺着链查找符号，如果没有找到的话，就通过`malloc()`分配一个新条目并将其添加到链上。再哈希法创建一个初始固定大小的符号表，同样也用`malloc()`分配。当符号表填满时，创建一个更大的符号表并且拷贝当前所有的条目，拷贝中需要使用哈希函数来确定每个条目在新符号表中的位置。两种技术都可以解决我们的问题，但其中一个是会使交叉引用的生成变得零乱，你知道是哪一个吗？为什么？



使用Bison

前一章主要讲解了flex。本章我们将介绍bison，不过我们依然会使用flex来生成我们的词法分析器。flex可以识别正则表达式，而bison可以识别语法。flex把输入流分解为若干个片段（记号），而bison则分析这些记号并基于逻辑进行组合。本章我们将完成第1章的桌面计算器，首先加入简单的算术功能，接着是一些内置功能、用户变量，最后是用户自定义函数。

Bison语法分析器如何匹配输入

bison基于你所给定的语法来生成一个可以识别这个语法中有效“语句”的语法分析器。这里是广泛意义上的语句（sentence）——对于C语言语法，语句是指语法上有效的C程序。程序完全可能在语法上正确但是在语义上有问题，例如，一个把字符串赋值给整型变量的C程序。bison只处理语法，你需要保证其他部分的正确性。正如我们在第1章看到的那样，语法由一系列规则组成，语法分析器基于这些规则来识别语法上正确的输入。例如，下面是我们会在本章稍后介绍的计算器中的一段语法：

```
statement: NAME '=' expression  
expression: NUMBER '+' NUMBER  
           | NUMBER '?' NUMBER
```

竖线（|）意味着同一个语法符号有两种可能性，也就是说，一个表达式可以是加法或者是减法。：左边的语法符号称为规则的左部（*left-hand side*），通常简写为LHS，而右边的语法符号称为规则的右部（*right-hand side*），通常简写为RHS。几条规则可能会有相同的左部，这种情况下竖线是一种便捷的写法。在输入中出现并且被词法分析器返回的符号是终结符或称记号，而规则左部的语法符号是非终结符。终结符和非终结符必须不同，把记号写在规则左边是错误的。

表示语句分析的通常方法是一棵树。例如，如果我们用上面的语法分析输入`fred = 12 + 13`，这棵树将像图3-1那样。

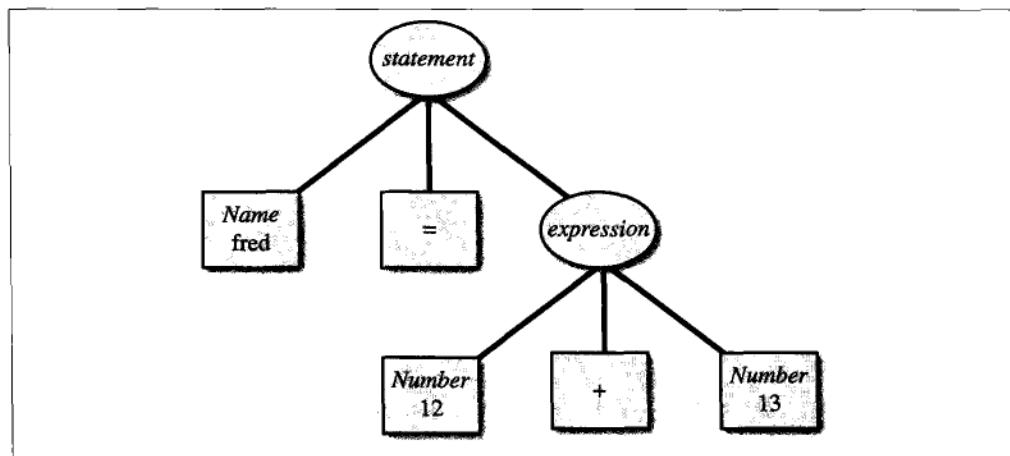


图3-1：表达式语法分析树

这个例子中，`12 + 13`是一个expression，而`fred = expression`是一个statement。`bison`语法分析器并不会自动把这棵树创建为一个数据结构，虽然我们后面会看到，我们自己做也不难。每个语法都包含了一个起始符号，它作为语法分析树的根出现。在这个语法中，`statement`是起始符号。规则可以直接或者间接地指向它们自身，这个重要特性使它可以分析很长的输入序列。让我们扩展我们的语法使它可以处理更长的算术表达式：

```
expression: NUMBER
| expression + NUMBER
| expression ? NUMBER
```

现在我们能够像图3-2所示的那样通过重复地运用表达式规则来分析像`fred = 14 + 23 - 11 + 7`这样的序列。`bison`能够有效地分析递归规则，所以我们会发现递归规则几乎出现在我们使用的每个语法中。

移进/归约分析

`bison`语法分析器通过查找能够匹配当前记号的规则来运作。当`bison`处理一个语法分析器时，它创建一组状态，每个状态都反映出一个或者多个部分分析过的规则中可能的位置。当语法分析器读取记号时，每当它读到的记号无法结束一条规则时，它将把这个记号压入一个内部堆栈，然后切换到一个新状态，这个状态能够反映出刚刚读取的记号。这种行为叫做移进 (*shift*)。当它发现压入的所有语法符号已经可以组成规则的右部时，它将把右部符号全部从堆栈中弹出，然后把左部语法符号压入堆栈。这种行为叫做

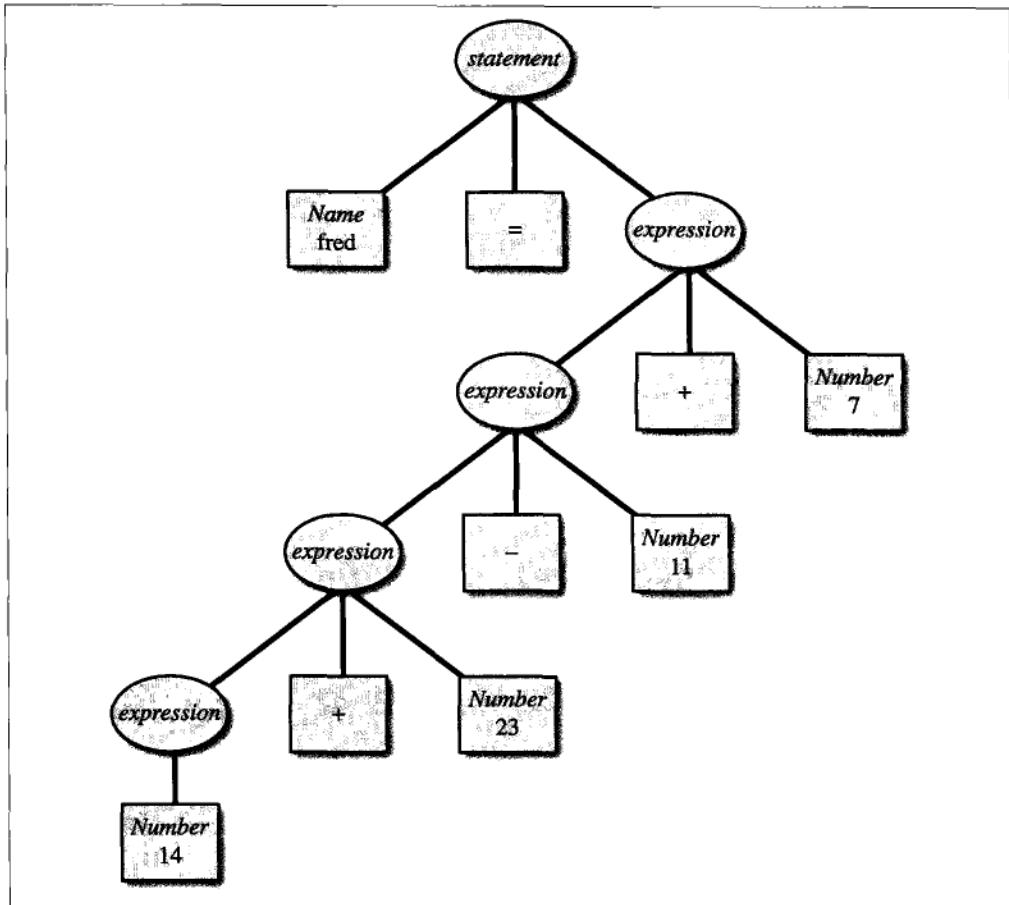


图3-2：带有递归规则的语法分析树

归约 (reduction)，因为它通常消减了堆栈中一定数量的符号^(注1)。每当bison归约一条规则时，它会执行该规则关联的用户代码。该代码也就是你对语法分析器分析的内容实际要做的事情。让我们看一下它是如何使用图3-1中的相同规则来分析输入fred = 12 + 13的。语法分析器开始时一次移进一个记号到内部堆栈中：

```

fred
fred =
fred = 12
fred = 12 +
fred = 12 + 13
  
```

注1： 有可能存在空的规则右部，这种情况下归约并不消减任何符号，但我们还是称其为归约。

这时它可以归约规则expression: NUMBER + NUMBER，所以它从堆栈中弹出12、加号和13，并把它们替换为expression：

```
fred = expression
```

现在它又可以归约规则statement: NAME = expression，所以它弹出fred、=和expression并把它们替换为statement。我们到达输入的末尾而堆栈被归约成起始符号，所以输入对于这个语法而言是有效的。

Bison的LALR(1)语法分析器无法分析的语法

bison语法分析器可以使用两种分析方法，一种是LALR(1)（自左向右向前查看一个记号），另一种是GLR（通用的自左向右）。大多数语法分析器使用LRLR(1)，它不如GLR强大但是被认为比GLR更快和更容易使用。本章我们将讲述LALR分析而把GLR放到第9章。

虽然LALR分析也很强大，但是你能够写出它无法处理的语法。它不能处理有歧义的语法，比如相同的输入可以匹配多棵语法分析树的情况。它也不能处理需要向前查看多个记号才能确定是否匹配规则的语法（但是bison有一个很奇妙的技巧来解决常见的二义性文法，我们会在这几页讲到）。考虑一下这个特意编排的例子：

```
phrase: cart_animal AND CART
       | work_animal AND PLOW

cart_animal: HORSE | GOAT

work_animal: HORSE | OX
```

这个语法并没有歧义，因为对于任何有效输入它只有一种可能的语法分析树，但bison无法处理它，因为它需要向前查看两个符号。具体来说，对于HORSE AND CART这种输入，在看到CART之前它无法区别HORSE是一个cart_animal还是一个work_animal，而bison不能够向前查看那么多。如果我们把第一条规则修改为：

```
phrase: cart_animal CART
       | work_animal PLOW
```

bison将没有任何问题，因为它能够向前查看一个记号来确定HORSE之后是否是CART，这样就是cart_animal；或者是PLOW，这样就是work_animal。实际上，bison能够处理的规则并不像这儿所看见的那么复杂和令人迷惑。一个原因是bison确实知道哪些语法它可以分析。如果你给它一个不能分析的语法，它会告诉你，所以并不存在过于复杂的语法分析器不带任何打印信息就出错的情况。另外一个原因是bison所处理的语法和人们实际写的语法是一一对应的。通常来说，bison认为有歧义的语法大家看了也会迷惑的。所以

如果你对所设计的语言还有一定的控制权的话，你一定要让它对bison和它的用户都易于理解。对于更多移进/归约分析的知识，请参见第7章。如果你希望了解bison是如何把你规范翻译成可以工作的C程序，一个很好的参考是Dick Grune的《Parsing Techniques: A Practical Guide》。他提供了一个虽然旧但是比较全的下载版本 (<http://www.cs.vu.nl/~dick/PTAPG.html>)。

Bison语法分析器

bison规范与flex规范一样由三部分组成（flex从lex复制了它的组织结构，而lex又是从yacc复制过来的，yacc是bison的前身）。第一部分是定义部分，处理语法分析器的控制信息，建立分析器操作所需要的执行环境。第二部分包含语法分析器的规则。而第三部分则是C代码，它们会被逐字拷贝到生成的C程序中去。

bison通过把每个片段插入到标准的框架文件中来创建C程序。规则会被编译成数组的形式，数组的内容代表了可以匹配输入记号的状态机。语义动作中的\$N和@N的值会首先翻译成C代码，然后被放置到`yyparse()`中的switch语句中，`yyparse()`会在归约发生时执行相应的动作。框架文件中存在着一些不同版本的代码，bison将基于当前使用的选项来决定哪个版本被使用，例如，如果语法分析器使用定位特性，它会把处理定位信息的代码包含进来。

本章我们将对第1章中简单的计算器例子做重要的扩展。首先，我们利用bison的一些便捷特性来重写它，使它产生一个可重用的数据结构，而无需立即计算数值。然后我们将添加更复杂的循环和函数的语法，展示它们是如何在一个简单的解释器里得以实现的^(注2)。

抽象语法树

在编译器中最强大的数据结构之一就是抽象语法树（*abstract syntax tree*）。在第1章里我们见过一棵语法分析树，这棵树对于每条分析输入串的规则来说都有一个节点。但在大多数实际语法中，有些规则只是用来管理分组，对于程序本身并没有太大的意义。比如在计算器的例子中，规则`exp: term`和`term: factor`仅仅是为了告诉语法分析器各个操作符的相对优先级。抽象语法树可以把分析树中这种我们不需要关注的规则节点移去。

一旦语法分析器创建了一棵抽象语法树，它就可以直接编写递归子程序来遍历整棵树。在这个例子中我们将看到几种遍历语法树的程序。

注2：接着我们就停下来了，现实世界里已经有很多脚本解释程序。出于练习的目的，你最好基于一个现有的测试充分的脚本语言，例如Python、Perl或者Lua，来添加扩展部分，而不是再写一个新的解释程序。

基于抽象语法树的改进的计算器

这个例子已经大得足以让我们把它分解为几个源文件，所以大部分的C代码将放到另外一个文件中，这意味着我们还需要一个C头文件来定义不同文件都要用到的函数和数据结构（例3-1）。

例3-1：创建基于抽象语法树的计算器：头文件 *fb3-1.h*

```
/*
 * fb3-1计算器的声明部分
 */

/* 与词法分析器的接口 */
extern int yylineno; /* 来自于词法分析器 */
void yyerror(char *s, ...);

/* 抽象语法树中的节点 */
struct ast {
    int nodetype;
    struct ast *l;
    struct ast *r;
};

struct numval {
    int nodetype; /* 类型K 表明常量 */
    double number;
};

/* 构造抽象语法树 */
struct ast *newast(int nodetype, struct ast *l, struct ast *r);
struct ast *newnum(double d);

/* 计算抽象语法树 */
double eval(struct ast *);

/* 删除和释放抽象语法树 */
void treefree(struct ast *);
```

变量yylineno和例程yyerror与flex的例子相似。我们的yyerror做了一点点增强，以便于允许接受像printf那样的多个参数。

抽象语法树由多个节点组成，每个节点都有一个节点类型。不同的节点可以有不同的域，但目前我们只有两种类型，一种是指向最多两个子节点的指针，另外一种包含一个数值。两个例程，newast和newnum，用来创建抽象语法树节点；eval遍历抽象语法树，返回它所代表的表达式的值；而treefree遍历抽象语法树，删除它的所有节点。

例3-2：基于抽象语法树的计算器的bison语法分析器

```
/* 基于抽象语法树的计算器 */
```

```
%{
```

```

#include <stdio.h>
#include <stdlib.h>
#include "fb3-1.h"
}

%union {
    struct ast *a;
    double d;
}

/* 声明记号 */
%token <d> NUMBER
%token EOL

%type <a> exp factor term

```

例3-2展示了基于抽象语法树的计算器所对应的bison语法分析器。这个分析器的第一部分使用`%union`来声明语法分析器中符号值的类型。在bison语法分析器中，每个语法符号，包括记号和非终结符，都可以有一个相应的值。默认情况下所有的符号值都是整数，但是真正有用的程序通常需要更多有价值的符号值。而`%union`，正如它的名字所暗示的那样，可以用来为符号值创建一个C语言的union（联合）类型。在这个例子中，联合类型有两个成员：一个是`a`，指向一个抽象语法树的指针；一个是`d`，它是一个双精度浮点数。

一旦联合类型被定义，我们需要告诉bison每种语法符号使用的值类型，这通过放置在尖括号(`<>`)中的联合类型的相应成员名字来确定。记号`NUMBER`代表了输入中的数字，它通过符号值`<d>`来保存具体的数值。新的声明`%type`把值`<a>`赋给`exp`、`factor`和`term`，当我们创建抽象语法树时会用到它们。

如果你不使用记号或者非终结符的值，你并不需要为它们声明类型。当声明中存在`%union`时，如果你试图使用一个没有被赋予类型的符号值，bison将会报错。记住如果没有显式的语义动作代码时，规则将使用默认语义动作`$$ = $1`；而当左部符号与右部符号具有不同的类型时，bison也会报错。

```

%%
calclist: /* 空 */
| calclist exp EOL {
    printf("= %.4g\n", eval($2));      计算抽象语法树并打印结果
    treefree($2);                      释放抽象语法树
    printf("> ");
}
| calclist EOL { printf("> "); } /* 空行或者注释 */
;

exp: factor
| exp '+' factor { $$ = newast('+', $1,$3); }

```

```

| exp '-' factor { $$ = newast('-', $1,$3); }
;

factor: term
| factor '*' term { $$ = newast('*', $1,$3); }
| factor '/' term { $$ = newast('/', $1,$3); }
;

term: NUMBER { $$ = newnum($1); }
| '\'' term { $$ = newast('\'', $2, NULL); }
| '(' exp ')' { $$ = $2; }
| '-' term { $$ = newast('M', $2, NULL); }
;
%%

```

文字字符记号

与第一章的版本相比，规则部分有两个重大改变。一个是规则现在使用文字字符（literal character）记号来表示操作符。我们并不需要为每个记号定义一个名字，相反，单引号引起的字符也可以作为记号，记号的ASCII值将成为记号的编号（bison对于命名记号的编号从258开始，所以不会有冲突）。按照惯例，文字字符记号用来代表具有相同字符的输入记号；例如，记号'+'代表了输入记号+，所以它们只是用在标点和操作符上面。除非你需要声明文字字符记号的类型，否则你不用显式地声明它们。

另外一个改变是它为每个表达式创建了一个抽象语法分析器，而没有立即计算。在这一版的计算器中，我们为每个表达式创建抽象语法树，然后计算这个抽象语法树，打印结果，最后释放抽象语法树。我们调用newast()来创建抽象语法树中的每个节点。每个节点都有一个操作类型，它们通常和记号名字一致。注意单目负号操作符创建了一个类型为M的节点，以便于区分它和双目减法操作符。

例3-3：基于抽象语法树的计算器对应的词法分析器

```

/* 识别计算器的记号 */
%option noyywrap nodefault yylineno
%{
#include "fb3-1.h"
#include "fb3-1.tab.h"
%}

/* 浮点数指数部分 */
EXP ([Ee][-+]?[0-9]+)

%%
"+"
"-"
"**"
"/"
"\\""
"(" |

```

```

    }" { return yytext[0]; }
[0-9]+."[0-9]*{EXP}? |
".?"[0-9]+{EXP}? { yylval.d = atof(yytext); return NUMBER; }

\n { return EOL; }
//".*
[ \t] { /* 忽略空白字符 */ }
. { yyerror("Mystery character %c\n", *yytext); }
%%

```

例3-3展示的词法分析器比第1章的版本相对简单。我们使用常见的手法来描述单字符操作符，通过同一条规则处理它们，然后把yytext[0]，也就是它们自身，作为记号返回。我们依然还有NUMBER和EOL的命名记号。我们使用第2章中的模式版本处理浮点数，并且把内部的数值类型改成了双精度浮点数。由于yyval现在是联合类型，双精度浮点数的值将不得不赋给yyval.d（flex无法像bison那样自动地管理记号值）。

例3-4：基于抽象语法树的计算器对应的C例程

```

# include <stdio.h>
# include <stdlib.h>
# include <stdarg.h>
# include "fb3-1.h"

struct ast *
newast(int nodetype, struct ast *l, struct ast *r)
{
    struct ast *a = malloc(sizeof(struct ast));

    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = nodetype;
    a->l = l;
    a->r = r;
    return a;
}

struct ast *
newnum(double d)
{
    struct numval *a = malloc(sizeof(struct numval));

    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = 'K';
    a->number = d;
    return (struct ast *)a;
}

```

最后，还有一个文件包含了语法分析器需要调用的一些例程，参见例3-4^(注3)。前两个例程创建抽象语法树节点，它们分配（malloc）一个节点并进行填充。所有的抽象语法树节点都把nodetype作为它们的第一个域，所以当我们遍历树的时候可以知道正在访问的节点类型。

```
double
eval(struct ast *a)
{
    double v; 子树的计算结果

    switch(a->nodetype) {
        case 'K': v = ((struct numval *)a)->number; break;

        case '+': v = eval(a->l) + eval(a->r); break;
        case '-': v = eval(a->l) - eval(a->r); break;
        case '*': v = eval(a->l) * eval(a->r); break;
        case '/': v = eval(a->l) / eval(a->r); break;
        case '|': v = eval(a->l); if(v < 0) v = -v; break;
        case 'M': v = -eval(a->l); break;
        default: printf("internal error: bad node %c\n", a->nodetype);
    }
    return v;
}

void
treefree(struct ast *a)
{
    switch(a->nodetype) {

        /* 两棵子树 */
        case '+':
        case '-':
        case '*':
        case '/':
            treefree(a->r);

        /* 一棵子树 */
        case '|':
        case 'M':
            treefree(a->l);

        /* 没有子树 */
        case 'K':
            free(a);
            break;
        default: printf("internal error: free bad node %c\n", a->nodetype);
    }
}
```

注3：这些例程在这个语法分析器的第三版中可以被消除，但是它们可以帮助你更容易地调试程序而不需要在语法分析器文件中加入大量的额外代码。

接下来我们有两个遍历树的例程。它们都采用了名为深度优先 (*depth-first*) 的算法，首先递归访问每个节点的所有子树，然后再访问节点本身。例程`eval`在每次调用时返回语法分析树或者子树的值，而例程`treefree`不需要返回任何值。

```
void
yyerror(char *s, ...)
{
    va_list ap;
    va_start(ap, s);

    fprintf(stderr, "%d: error: ", yylineno);
    vfprintf(stderr, s, ap);
    fprintf(stderr, "\n");
}

int
main()
{
    printf("> ");
    return yyparse();
}
```

最后是`yyerror`和`main`。这个版本的`yyerror`使用了可变长参数，它可以接受`printf`格式的参数列表，这在生成错误消息时十分方便。

编译基于抽象语法树的计算器

现在这个程序有三个源文件和一个头文件，所以理所当然我们使用`make`来编译它。

```
fb3-1: fb3-1.l fb3-1.y fb3-1.h
       bison -d fb3-1.y
       flex -ofb3-1.lex.c fb3-1.l
       cc -o $@ fb3-1.tab.c fb3-1.lex.c fb3-1funcs.c
```

注意`flex`的`-o`参数。`bison`自动基于`.y`文件来命名生成的C文件，但是`flex`总是把它的C文件命名为`lex.yy.c`，除非你另行指定。

移进/归约冲突和操作符优先级

表达式分析器使用了三种不同的语法符号，`exp`、`factor`和`term`，来设置操作符的优先级和结合性。虽然这个语法分析器目前还是比较清晰的，但是随着更多具有不同优先级的操作符被添加到语法中，整个语法会变得难以阅读和维护。`bison`提供了一个很聪明的方法，它可以在语法规则之外单独描述优先级，这使得语法和分析器都变得短小而且易于维护。首先，我们将让所有的表达式都使用语法符号`exp`：

```
%type <a> exp
```

```

%%

...
exp: exp '+' exp { $$ = newast('+', $1,$3); }
| exp '-' exp { $$ = newast('-', $1,$3); }
| exp '*' exp { $$ = newast('*', $1,$3); }
| exp '/' exp { $$ = newast('/', $1,$3); }
| '||' exp { $$ = newast('||', $2, NULL); }
| '(' exp ')' { $$ = $2; }
| '-' exp { $$ = newast('M', $2, NULL); }
NUMBER { $$ = newnum($1); }

;
%%

```

但这个语法有一个问题：它有很明显的歧义。例如，输入 $2+3*4$ 可能意味着 $(2+3)*4$ 或者 $2+(3*4)$ ，而输入 $3-4-5-6$ 可能意味着 $3-(4-(5-6))$ 或者 $(3-4)-(5-6)$ 或者其他更多的可能性。图3-3展示了针对 $2+3*4$ 的两种不同的分析树。

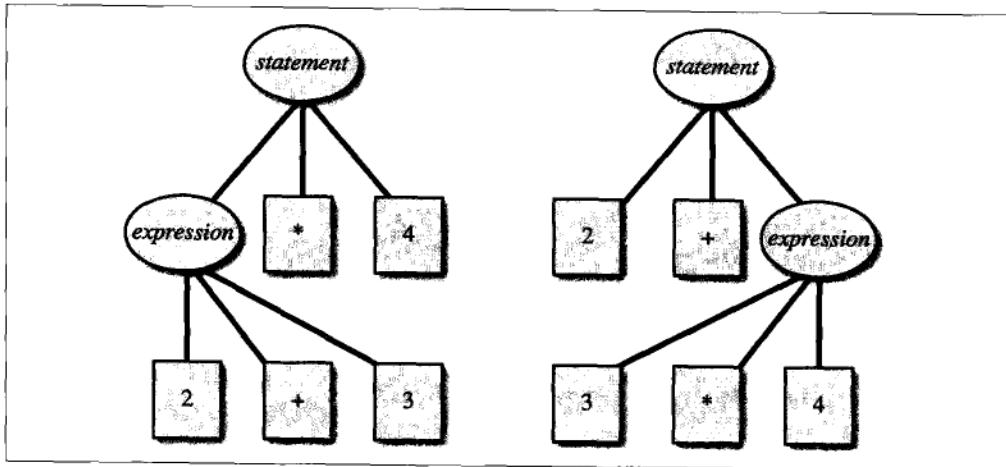


图3-3：两种表达式语法分析树

如果你编译上述语法的话，bison将报告24个移进/归约冲突（shift/reduce conflict），在这些状态里，bison无法确定应该首先移进记号，还是首先归约一条规则。例如，当分析 $2+3*4$ 时，语法分析器将进行以下这些步骤（这里我们缩写exp为E）：

```

2      移进 NUMBER
E      归约 E → NUMBER
E +
E + 3 移进 +
E + E  归约 E → NUMBER

```

这时，语法分析器看到了*，它可以使用下面的规则归约 $2+3$ 为一个表达式：

```
exp: exp '+' exp
```

或者它也可以先移进`*`，然后再使用下面的规则进行归约：

```
exp: exp '*' exp
```

这里面的问题在于我们并没有告诉**bison**操作符的优先级和结合性。优先级(*precedence*)决定了表达式中哪个操作符先执行。数学和编程的传统(可以追溯到1956年第一个Fortran编译器)表明乘法和除法的优先级比加法和减法高，所以`a+b*c`意味着`a+(b*c)`，而`d/e-f`意味着`(d/e)-f`。在任何表达式语法中，操作符总是被分成了若干组，每一组拥有从低到高的不同优先级。优先级数依据不同语言而不同。C语言拥有太多优先级是众所周知的，一共15级。

结合性决定了具有相同优先级的操作符的结合顺序。操作符可能自左向右结合，例如，`a-b-c`在C语言中意味着`(a-b)-c`；或者自右向左结合，例如，`a=b=c`意味着`a=(b=c)`。有些情况下操作符根本没有结合顺序，例如，在Fortran中，`A.LE.B.LE.C`就是非法的。

在语法中有两种方法来指定优先级和结合性：隐式定义和显式定义。以前我们都是隐式指定它们的，对不同的优先级使用了不同的非终结符。这种编写语法的方法非常合情合理，事实上，如果**bison**并不支持优先级的显式定义的话，这是唯一一种方法。

但**bison**允许你显式地指定优先级。我们添加一些行到定义部分，以便于告诉**bison**如何解决冲突：

```
%left '+' '-'
%left '*' '/'
%nonassoc '|' UMINUS

%type <a> exp

%%
...
exp: exp '+' exp { $$ = newast('+', $1,$3); }
| exp '-' exp { $$ = newast('-', $1,$3); }
| exp '*' exp { $$ = newast('*', $1,$3); }
| exp '/' exp { $$ = newast('/', $1,$3); }
| '|' exp { $$ = newast('|', $2, NULL); }
| '(' exp ')' { $$ = $2; }
| '-' exp %prec UMINUS{ $$ = newast('M', NULL, $2); }
| NUMBER { $$ = newnum($1); }
```

每个声明都定义了一种优先级，`%left`、`%right`和`%noassoc`的出现顺序决定了由低到高的优先级顺序。这些定义告诉**bison**`+`和`-`是左结合的，而且具有最低的优先级；`*`和`/`也是左结合的，但具有更高的优先级；而`|`和`UMINUS`(代表单目负号操作的伪记号)没有结合性，并且具有最高的优先级(在这里我们没有任何的右结合操作符，但如果有的话，应该使用`%right`)。**bison**把规则右部最右边记号的优先级赋给规则本身，如果这

个记号没有被赋予任何优先级，规则也将没有优先级。当bison遇到移进/归约冲突时，它将查询优先级表，如果冲突中的所有规则都具有优先级的话，它将使用优先级来解决冲突。

在我们的语法中，所有的冲突都发生在形式为`exp OP exp`的规则之间，所以为4个操作符设定优先级可以解决所有的冲突。使用优先级的语法分析器比前一个使用额外规则来做隐式优先级定义的语法分析器来得更小更快，因为它具有更少需要归约的规则。

负数规则包含`%prec UMINUS`。这个规则中唯一的操作符是`-`，它拥有较低的优先级，但我们希望单目负号操作符比乘法拥有更高的优先级。而`%prec`可以让bison把UMINUS的优先级赋予这个规则。

什么时候不应该使用优先级规则

你确实可以使用优先级规则解决语法中出现的任何移进/归约冲突。但这通常是一个很糟糕的想法。在表达式语法中，冲突的原因很容易理解，而优先级规则的效果也很清楚。在其他一些情况下，优先级规则解决了移进/归约的问题，但我们很难明白这种改动给语法造成的后果。

我们应该只在两种场合下使用优先级规则：表达式语法或者解决在`if/then/else`语言结构的语法中的“dangling else”冲突（参见稍后的第190页上的“IF/THEN/ELSE”一节的例子）。否则，只要有可能，你就应该通过修正语法来解决冲突。记住冲突仅仅表明bison无法为某个语法创建语法分析器，这可能是因为该语法确实有二义性。这就意味着对于相同的输入存在多种可能的分析方法，而bison选择了其中的一种来创建。如果不考虑前面两种场合的话，它通常表明你的语言定义有问题。有时候，如果一种语法对bison而言是有歧义的，那么它基本上对人类来说也是有歧义的。第8章里有更多的信息来帮助查找和修正冲突，同时你还可以了解到更高级的bison特性，它们允许你使用二义性文法，如果你真的希望那么做的话。

一个高级计算器

本章最后的例子扩展了前面的计算器，使它成为一个虽然短小但是有现实意义的编译器。我们将添加命名的变量和赋值，比较表达式（大于、小于、等于等等），`if/then/else`和`do/while`的流程控制，内置和用户自定义的函数以及一点点错误恢复机制。前一版本的计算器并没有充分利用抽象语法树的优点，但这一版本中，抽象语法树是实现流程控制和用户自定义函数的关键技术。下面是一个例子，它定义一个用户自定义函数，接着调用它，使用内置函数作为其中一个参数。

```
> let avg(a,b) = (a+b)/2;
定义好的avg
> avg(3, sqrt(25))
= 4
```

像前面那样，我们首先开始声明部分，参见例3-5。

例3-5：高级计算器头文件 *fb3-2.h*

```
/*
 * fb3-1计算器的声明部分
 */

/* 与词法分析器的接口 */
extern int yylineno; /* 词法分析器已经定义 */
void yyerror(char *s, ...);

/* 符号表 */
struct symbol { /* 变量名 */
    char *name;
    double value;
    struct ast *func; /* 函数体 */
    struct symlist *syms; /* 虚拟参数列表 */
};

/* 固定大小的简单的符号表 */
#define NHASH 9997
struct symbol symtab[NHASH];

struct symbol *lookup(char*);

/* 符号列表，作为参数列表 */
struct symlist {
    struct symbol *sym;
    struct symlist *next;
};

struct symlist *newsymlist(struct symbol *sym, struct symlist *next);
void symlistfree(struct symlist *sl);
```

符号表改编自前一章的例子。在这个计算器中，每个符号都可以有一个变量和一个用户自定义函数。`value`域用来保存符号的值，`func`域指向用抽象语法树表述的该函数的用户代码，而`syms`域指向任意多个虚拟参数（dummy argument）的链表，这些参数也是符号（在前面的例子中，`avg`是函数，`a`和`b`是虚拟参数）。函数`newsymlist`和`symlistfree`创建和释放符号。

```
/* 节点类型
 * + - * / |
 * 0-7 比较操作符，位编码： 04 等于， 02 小于， 01 大于
 * M 单目负号
 * L 表达式或者语句列表
 * I IF 语句
 * W WHILE 语句
```

```

/* N 符号引用
 * = 赋值
 * S 符号列表
 * F 内置函数调用
 * C 用户函数调用
 */
enum bifs { /* 内置函数 */
    B_sqrt = 1,
    B_exp,
    B_log,
    B_print
};

/* 抽象语法树节点 */
/* 所有节点都有公共的初始nodetype */

struct ast {
    int nodetype;
    struct ast *l;
    struct ast *r;
};

struct fncall { /* 内置函数 */
    int nodetype; /* 类型 F */
    struct ast *l;
    enum bifs functype;
};

struct ufcall { /* 用户自定义函数 */
    int nodetype; /* 类型 C */
    struct ast *l; /* 参数列表 */
    struct symbol *s;
};

struct flow {
    int nodetype; /* 类型 I 或者 W */
    struct ast *cond; /* 条件 */
    struct ast *tl; /* then 分支或者 do 语句 */
    struct ast *el; /* 可选的 else 分支 */
};

struct numval {
    int nodetype; /* 类型 K */
    double number;
};

struct symref {
    int nodetype; /* 类型 N */
    struct symbol *s;
};

struct symasgn {
    int nodetype; /* 类型 = */
    struct symbol *s;
    struct ast *v; /* 值 */
};

```

```

};

/* 构造抽象语法树 */
struct ast *newast(int nodetype, struct ast *l, struct ast *r);
struct ast *newcmp(int cmptype, struct ast *l, struct ast *r);
struct ast *newfunc(int functype, struct ast *l);
struct ast *newcall(struct symbol *s, struct ast *l);
struct ast *newref(struct symbol *s);
struct ast *newasgn(struct symbol *s, struct ast *v);
struct ast *newnum(double d);
struct ast *newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast *tr);

/* 定义函数 */
void dodef(struct symbol *name, struct symlist *syms, struct ast *stmts);

/* 计算抽象语法树 */
double eval(struct ast *);

/* 删除和释放抽象语法树 */
void treefree(struct ast *);

/* 与词法分析器的接口 */
extern int yylineno; /* 来自词法分析器 */
void yyerror(char *s, ...);

```

这个版本在抽象语法树中有相当多的节点类型。与前面一样，每个节点都有一个 `nodetype`，遍历树的代码使用这个变量来判断当前访问的节点^(注4)。基本的 `ast` 节点也被用来进行比较，基于各种比较（小于、小于等于、等于等等）变成不同类型的节点，`ast` 节点也被用于表达式列表。

内置函数使用 `funcall` 节点，该节点包含参数的抽象语法树（每个内置函数只有一个参数）和一个确定内置函数的枚举类型。这儿一共定义了三个标准函数，`sqrt`、`exp` 和 `log`，还有一个 `print` 函数打印出它的参数并把参数作为返回值。用户自定义函数使用 `ufcall` 节点，该节点包含一个指向自定义函数（在符号表中有定义）的指针和一个参数列表的抽象语法树。

流程控制表达式 `if/then/else` 和 `while/do` 使用 `flow` 节点，它包含控制表达式、`then` 分支或者 `do` 语句体以及可选的 `else` 分支。

常量与以前一样使用 `numval`；符号引用使用 `symref`，它具有指向符号表中特定符号的指针；赋值使用 `symasgn`，它有一个指向被赋值符号的指针和使用抽象语法树表示的值。

每个抽象语法树都有相应的值。对于 `if/then/else` 而言，它的值就是所选择的分支的值；`while/do` 的值则是 `do` 语句列表的最后一条语句的值；而表达式列表的值由最后一个表达

注4：这是经典C的定义方式，它并不适合于C++。如果你用C++来实现语法分析器的话（我们第9章会讲到），你需要使用在抽象语法树中的显式联合结构来达到相似的结果。

式确定^(注5)。最后，我们使用C过程来创建各种类型的抽象语法树节点，还有一个过程来创建用户自定义函数。

高级计算器的语法分析器

例3-6展示了基于抽象语法树的高级计算器的语法分析器。

例3-6：高级计算器语法分析器fb3-2.y

```
/* 基于抽象语法树的计算器 */
```

```
%{
# include <stdio.h>
# include <stdlib.h>
# include "fb3-2.h"
%}

%union {
    struct ast *a;
    double d;
    struct symbol *s; /* 指定符号 */
    struct symlist *sl;
    int fn;           /* 指定函数 */
}

/* 声明记号 */
%token <d> NUMBER
%token <s> NAME
%token <fn> FUNC
%token EOL

%token IF THEN ELSE WHILE DO LET

%nonassoc <fn> CMP
%right '='
%left '+' '-'
%left '*' '/'
%nonassoc '||' UMINUS

%type <a> exp stmt list explist
%type <sl> symlist

%start calclist
%%
```

这儿的%union定义了很多种符号值，这在实际的bison语法分析器中很常见。正如符号(symbol)值可以是指向抽象语法树的指针或者一个具体的数值那样，符号值也可以是指向符号表中特定用户符号(user symbol)、符号列表、比较子类型和函数记号的指

注5：这个设计大致基于古老的BLISS系统编程语言。虽然我们也可以设计出一个抽象语法树来区分表达式和语句，但目前这种实现更简单一些。

针。（这儿的单词*symbol*会有一定的迷惑性，它既代表了在bison语法中使用的名字，也代表了编译程序中的用户类型。所以当上下文不是很清晰的时候，我们使用*user symbol*来表示后者。）

这儿有一个新记号FUNC表示内置函数，它的值确定了具体的某个函数，另外还有6个保留字，从IF到LET。记号CMP是6种比较操作符之一，它的值确定了具体的操作符。（使用单一记号来表达多个语义上接近的操作符的技巧可以帮助我们缩短语法的长度。）

优先级声明列表从新的CMP和=操作符开始。

%start声明定义了顶层规则，所以我们不需要把这条规则放到语法分析器的开始部分。

计算器语句的语法

```
stmt: IF exp THEN list { $$ = newflow('I', $2, $4, NULL); }
    | IF exp THEN list ELSE list { $$ = newflow('I', $2, $4, $6); }
    | WHILE exp DO list { $$ = newflow('W', $2, $4, NULL); }
    | exp
    ;
list: /* 空 */ { $$ = NULL; }
    | stmt ';' list { if ($3 == NULL)
                      $$ = $1;
                      else
                          $$ = newast('L', $1, $3);
                }
    ;
;
```

我们的语法区分了语句（stmt）和表达式（exp）。语句是一个控制流（if/then/else或者while/do）或者一个表达式。if和while语句具有一连串语句，这里面的语句都用分号结尾。每当规则匹配到一条语句时，它调用相应的例程来创建合适的抽象语法树节点。

这儿的语法设计很随意，不过使用bison来构造语法分析器的优越性之一就在于它可以边试边改。语法的每个组成部分之间的影响是很微妙的，比如，当list的定义把分号放在每个语句之间而不是之后的话，语法将变得有歧义，除非语法也加入作为结束标志的FI和ENDDO记号来表明if/then和while/do语句的终止。

list的定义是右递归的，也就是说，是stmt ; list而不是list stmt ;。这对于语言识别没有任何差异，但我们因此会更容易构造一个从头到尾而不是相反方向的语句链表。每当stmt ; list规则被归约时，它创建一个链接来把语句添加到当前链表的头部。如果规则是list stmt ;，语句需要被添加到链表的尾部，这就要求我们使用一个更复杂的循环链表或者反转整个链表（就像我们在第1章中对引用列表的处理）。

使用右递归而不是左递归的缺点是它把所有需要被归约的语句都放在语法分析器的堆栈

里，直到链表的尾部才进行归约，而左递归在分析输入时每次遇到语句都会立刻将其添加到链表中。像我们现在的情况，链表可能只会有几条语句，这不会有什么问题，但在那种链表里可能有上千条语句的语言里，我们最好使用左递归然后翻转的方式来创建链表，这样可以避免堆栈溢出。有些程序员也会发现左递归更容易调试，因为它会倾向于在每条语句后产生输出，而不是在最后一次性处理所有语句。

计算器表达式的语法

```
exp: exp CMP exp { $$ = newcmp($2, $1, $3); }
| exp '+' exp { $$ = newast('+', $1,$3); }
| exp '-' exp { $$ = newast('-', $1,$3); }
| exp '*' exp { $$ = newast('*', $1,$3); }
| exp '/' exp { $$ = newast('/', $1,$3); }
| '||' exp { $$ = newast('||', $2, NULL); }
| '(' exp ')' { $$ = $2; }
| '-' exp %prec UMINUS { $$ = newast('M', $2, NULL); }
| NUMBER { $$ = newnum($1); }
| NAME { $$ = newref($1); }
| NAME '=' exp { $$ = newasgn($1, $3); }
| FUNC '(' explist ')' { $$ = newfunc($1, $3); }
| NAME '(' explist ')' { $$ = newcall($1, $3); }
;

explist: exp
| exp ',' explist { $$ = newast('L', $1, $3); }
;
symlist: NAME { $$ = newsymlist($1, NULL); }
| NAME ',' symlist { $$ = newsymlist($1, $3); }
;
```

这个表达式语法对前例做了适当的扩展。新的CMP规则处理6种比较操作符，它使用CMP的值来判定当前的操作符，还有一条赋值规则来创建赋值节点。

此外还有一些规则来处理由保留字（FUNC）标识的内置函数和由用户符号（NAME）标识的用户自定义函数。

规则explist定义了一个表达式列表，它创建这些表达式的抽象语法树来作为函数调用所需的实际参数。规则symlist定义了一个符号列表，它创建了这些符号的链表用于函数定义时所需要的虚拟参数。它们都是右递归的，以方便基于期望的顺序创建链表。

计算器的顶层规则

```
calclist: /* 空 */
| calclist stmt EOL {
  printf(= %4.4g\n> ", eval($2));
  treefree($2);
}
```

```
| calclist LET NAME '(' symlist ')' '=' list EOL {  
    dodef($3, $5, $8);  
    printf("Defined %s\n> ", $3->name); }  
| calclist error EOL { yyerrok; printf("> "); }  
;
```

语法的最后部分是顶层规则，它识别一个语句列表和函数声明。像前面那样，顶层规则计算每条语句的抽象语法树，打印结果，然后释放抽象语法树。在这里保存函数定义只是便于后续使用。

简单的语法分析器错误恢复

这个语法分析器的最后一条规则提供了一小部分错误恢复。考虑到bison语法分析器的工作原理，它并不太值得我们花精力来尝试修正错误，但是我们至少有可能在错误发生时把语法分析器恢复到可以继续工作的状态。在这里，特别的伪记号error确定了错误恢复点。当bison语法分析器遇到一个错误时，它开始从语法分析器堆栈里放弃各种语法符号，直到它到达一个记号error为有效的点；接着它开始忽略后续输入记号，直到它找到一个在当前状态可以被移进的记号，然后从这一点开始继续分析。如果又发生分析错误的话，它将放弃更多的堆栈中的语法符号和输入记号，直到它可以重新恢复分析，或者堆栈为空而分析失败。为了避免大段误导性的错误消息，语法分析器通常在第一个错误产生后就抑制后续的分析错误消息，直到它能够成功地在一行里移进三个记号。动作中的宏yyerrok告诉语法分析器恢复已经完成，这样后续的错误消息可以顺利产生。

虽然有可能在规则里加入大量的错误规则来尝试提供更多的错误恢复，但实际上大家很少使用超过两条规则。记号error几乎总是在顶层递归规则的标点符号处被用来进行同步，就像我们这儿所做的那样。

在错误恢复中，如果从堆栈中放弃的符号含有指向分配空间的值，错误恢复过程有可能导致内存泄漏，因为被放弃的值没有被释放。这个例子里我们并不需要关心这一点，不过bison确实提供了一个特性来告诉语法分析器，让它可以调用你的代码来释放被放弃的值，在第6章会有详细讲解。

高级计算器的词法分析器

例3-7：高级计算器的词法分析器 *fb3-2.l*

```
/* 识别计算器的记号 */  
%option noyywrap nodefault yylineno  
{  
# include "fb3-2.h"  
# include "fb3-2.tab.h"  
}  
  
/* 浮点数指数部分 */
```

```

EXP ([Ee][-+]?[0-9]+)

%%
/* 单一字符操作符 */
"+"
"-"
"*"
"/"
"="
"|"
","
";"
 "("
 ")" { return yytext[0]; }

/* 比较操作符, 所有返回值都是CMP记号 */
">" { yyval.fn = 1; return CMP; }
"<" { yyval.fn = 2; return CMP; }
"<>" { yyval.fn = 3; return CMP; }
"==" { yyval.fn = 4; return CMP; }
">=" { yyval.fn = 5; return CMP; }
"<=" { yyval.fn = 6; return CMP; }

/* 关键字 */

"if" { return IF; }
"then" { return THEN; }
"else" { return ELSE; }
"while" { return WHILE; }
"do" { return DO; }
"let" { return LET; }

/* 内置函数 */
"sqrt" { yyval.fn = B_sqrt; return FUNC; }
"exp" { yyval.fn = B_exp; return FUNC; }
"log" { yyval.fn = B_log; return FUNC; }
"print" { yyval.fn = B_print; return FUNC; }

/* 名字 */
[a-zA-Z][a-zA-Z0-9]* { yyval.s = lookup(yytext); return NAME; }

[0-9]+.[0-9]*{EXP}? |
."?[0-9]+{EXP}? { yyval.d = atof(yytext); return NUMBER; }

//.*
[\t] /* 忽略空白字符 */

\\n { printf("c> "); } /* 忽略续行符 */

\n { return EOL; }

. { yyerror("Mystery character %c\n", *yytext); }
%%

```

在例3-7中展示的词法分析器为前例添加了一小部分规则。里面有一些新的单字符操作

符。6个比较操作符都返回一个带有字面值以便于区分的CMP记号。这6个关键字和4个内置函数通过文字模式加以识别。注意它们必须被放在通用模式之前，以便优先于通用模式进行匹配。名字模式查找符号表中的名字并返回指向该名字的指针。

像前面那样，新行（EOL）标记了输入串的结束。由于函数或者表达式可能很长而无法在一行输完，我们允许续行。新的词法分析器规则匹配一个斜线和一个换行符，但是不返回任何值给分析器，这样续行符相对于分析器来说就是不可见的。不过它会打印一个提示符给用户。

保留字

在这个语法中，单词if、then、else、while、do、let、sqrt、exp、log和print都是保留字，它们不能够被作为用户符号来使用。是否允许用户在程序中使用相同名字来代表两种事物是一件有争论的事情。一方面，它使程序变得难以理解，但另一方面，用户将被强迫思考出新的名字，以便于不会和保留字发生冲突。

任何一种选择都可能变得极端，COBOL拥有超过300个保留字，所以没有人可以记住全部保留字，程序员开始借助于奇特的惯例，比如每个变量前都添加一个点号来保证它不会和保留字有冲突。另一方面，PL/I根本没有保留字，所以你可以写出下面的语句来：

```
IF IF = THEN THEN ELSE = THEN; ELSE ELSE = IF;
```

如果你保留关键字的话，bison语法分析器会更容易编写；否则，你需要很仔细地设计你的语言，以便于当词法分析器读取一个记号时，它要么是名字，要么是关键字，另外你还不得不提供额外的反馈给词法分析器，使它知道怎么做。我曾经写过Fortran的词法分析器，它没有任何的保留字，而且（在它的传统版本里）忽略所有的空白字符，我强烈建议你保留你的关键字。

构造和解释抽象语法树

最后是我们的辅助代码文件，参见例3-8。这个文件中的一部分与前例相同，例程main和yyerror没有变化所以就不再赘述。

例3-8：高级计算器辅助函数fb3-2func.c

```
/*
 * fb3-2的辅助函数
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <math.h>
```

```

# include "fb3-2.h"

/* 符号表 */
/* 哈希一个符号 */
static unsigned
symhash(char *sym)
{
    unsigned int hash = 0;
    unsigned c;

    while(c = *sym++) hash = hash*9 ^ c;

    return hash;
}

struct symbol *
lookup(char* sym)
{
    struct symbol *sp = &syntab[symhash(sym)%NHASH];
    int scount = NHASH; /* 需要查看的个数 */

    while(--scount >= 0) {
        if(sp->name && !strcmp(sp->name, sym)) { return sp; }

        if(!sp->name) { /* 新条目 */
            sp->name = strdup(sym);
            sp->value = 0;
            sp->func = NULL;
            sp->syms = NULL;
            return sp;
        }

        if(++sp >= syntab+NHASH) sp = syntab; /* 尝试下一个条目 */
    }
    yyerror("symbol table overflow\n");
    abort(); /* 尝试完所有的条目，符号表已满 */
}

```

下面是构造抽象语法树节点和符号列表的过程。它们都会分配一个节点，然后基于节点类型恰当地填充各个域。例程treefree的扩展版本递归地遍历一棵抽象语法树，同时释放这棵树的所有节点。

```

struct ast *
newast(int nodetype, struct ast *l, struct ast *r)
{
    struct ast *a = malloc(sizeof(struct ast));
    :
    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = nodetype;
    a->l = l;
}

```

```
a->r = r;
return a;
}

struct ast *
newnum(double d)
{
    struct numval *a = malloc(sizeof(struct numval));

    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = 'K';
    a->number = d;
    return (struct ast *)a;
}

struct ast *
newcmp(int cmptype, struct ast *l, struct ast *r)
{
    struct ast *a = malloc(sizeof(struct ast));

    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = 'o' + cmptype;
    a->l = l;
    a->r = r;
    return a;
}

struct ast *
newfunc(int functype, struct ast *l)
{
    struct fncall *a = malloc(sizeof(struct fncall));

    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = 'F';
    a->l = l;
    a->functype = functype;
    return (struct ast *)a;
}

struct ast *
newcall(struct symbol *s, struct ast *l)
{
    struct ufcall *a = malloc(sizeof(struct ufcall));

    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->s = s;
    a->l = l;
    return (struct ast *)a;
}
```

```
    exit(0);
}
a->nodetype = 'C';
a->l = l;
a->s = s;
return (struct ast *)a;
}

struct ast *
newref(struct symbol *s)
{
    struct symref *a = malloc(sizeof(struct symref));

    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = 'N';
    a->s = s;
    return (struct ast *)a;
}

struct ast *
newasgn(struct symbol *s, struct ast *v)
{
    struct symasgn *a = malloc(sizeof(struct symasgn));

    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = '=';
    a->s = s;
    a->v = v;
    return (struct ast *)a;
}

struct ast *
newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast *el)
{
    struct flow *a = malloc(sizeof(struct flow));

    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = nodetype;
    a->cond = cond;
    a->tl = tl;
    a->el = el;
    return (struct ast *)a;
}

/* 释放一棵抽象语法树 */
void
```

```

treefree(struct ast *a)
{
    switch(a->nodetype) {
        /* 两棵子树 */
        case '+':
        case '-':
        case '*':
        case '/':
        case '1': case '2': case '3': case '4': case '5': case '6':
        case 'L':
            treefree(a->r);
        /* 一棵子树 */
        case '|':
        case 'M': case 'C': case 'F':
            treefree(a->l);
        /* 没有子树 */
        case 'K': case 'N':
            break;
        case '=':
            free( ((struct symasgn *)a)->v);
            break;
        /* 最多三棵子树 */
        case 'I': case 'W':
            free( ((struct flow *)a)->cond);
            if( ((struct flow *)a)->tl) treefree( ((struct flow *)a)->tl);
            if( ((struct flow *)a)->el) treefree( ((struct flow *)a)->el);
            break;
        default: printf("internal error: free bad node %c\n", a->nodetype);
    }
    free(a); /* 总是释放节点自身*/
}

struct symlist *
newsymlist(struct symbol *sym, struct symlist *next)
{
    struct symlist *sl = malloc(sizeof(struct symlist));
    if(!sl) {
        yyerror("out of space");
        exit(0);
    }
    sl->sym = sym;
    sl->next = next;
    return sl;
}

/* 释放符号列表 */
void

```

```

symlistfree(struct symlist *sl)
{
    struct symlist *nsl;

    while(sl) {
        nsl = sl->next;
        free(sl);
        sl = nsl;
    }
}

```

计算器的核心是例程`eval`，它计算语法分析器中构造的抽象语法树。基于C语言的准则，比较表达式根据比较是否成功来返回1或者0，在if/then/else和while/do中的测试认为所有的非零值都为真。

我们采用相似的深度优先的树遍历算法来计算表达式的值。抽象语法树使if/then/else的实现变得很直接：计算条件表达式的抽象语法树来决定选择哪个分支，然后计算选择路径上的抽象语法树。对于do/while循环来说，`eval`中的一个循环会不断地计算条件表达式，然后执行语句体的抽象语法树，直到条件表达式的抽象语法树不再为真。当变量被赋值语句修改后，任何使用这个变量的抽象语法树都会基于新值来进行计算。

```

static double callbuiltin(struct fncall *);
static double calluser(struct ufncall *);

double
eval(struct ast *a)
{
    double v;

    if(!a) {
        yyerror("internal error, null eval");
        return 0.0;
    }

    switch(a->nodetype) {
        /* 常量 */
        case 'K': v = ((struct numval *)a)->number; break;

        /* 名字引用 */
        case 'N': v = ((struct symref *)a)->s->value; break;

        /* 赋值 */
        case '=': v = ((struct symasgn *)a)->s->value =
            eval(((struct symasgn *)a)->v); break;

        /* 表达式 */
        case '+': v = eval(a->l) + eval(a->r); break;
        case '-': v = eval(a->l) - eval(a->r); break;
        case '*': v = eval(a->l) * eval(a->r); break;
        case '/': v = eval(a->l) / eval(a->r); break;
        case '|': v = fabs(eval(a->l)); break;
    }
}

```

```

case 'M': v = -eval(a->l); break;

/* 比较 */
case '1': v = (eval(a->l) > eval(a->r))? 1 : 0; break;
case '2': v = (eval(a->l) < eval(a->r))? 1 : 0; break;
case '3': v = (eval(a->l) != eval(a->r))? 1 : 0; break;
case '4': v = (eval(a->l) == eval(a->r))? 1 : 0; break;
case '5': v = (eval(a->l) >= eval(a->r))? 1 : 0; break;
case '6': v = (eval(a->l) <= eval(a->r))? 1 : 0; break;

/* 控制流 */
/* 语法中允许空表达式, 所以需要检查这种可能性 */

/* if/then/else */
case 'I':
    if( eval( ((struct flow *)a)->cond) != 0) {    检查条件
        if( ((struct flow *)a)->tl) {                条件成立分支
            v = eval( ((struct flow *)a)->tl);
        } else
            v = 0.0; /* 默认值 */
    } else {
        if( ((struct flow *)a)->el) {                条件不成立分支
            v = eval(((struct flow *)a)->el);
        } else
            v = 0.0; /* 默认值 */
    }
    break;

/* while/do */
case 'W':
    v = 0.0; /* 默认值 */

    if( ((struct flow *)a)->tl) {
        while( eval(((struct flow *)a)->cond) != 0) 计算条件
            v = eval(((struct flow *)a)->tl);          执行目标语句体
    }
    break; /* 最后一条语句的值就是while/do的值 */

/* 语句列表 */
case 'L': eval(a->l); v = eval(a->r); break;

case 'F': v = callbuiltin((struct fncall *)a); break;
case 'C': v = calluser((struct ufcall *)a); break;

default: printf("internal error: bad node %c\n", a->nodetype);
}
return v;
}

```

执行计算器中的函数

计算器中最特别的部分是处理函数。内置函数的实现很简单：它们确定具体的函数，然后调用相应的代码来执行这个函数。

```
static double
callbuiltin(struct fnccall *f)
{
    enum bifs functype = f->functype;
    double v = eval(f->l);

    switch(functype) {
    case B_sqrt:
        return sqrt(v);
    case B_exp:
        return exp(v);
    case B_log:
        return log(v);
    case B_print:
        printf("= %4.4g\n", v);
        return v;
    default:
        yyerror("Unknown built-in function %d", functype);
        return 0.0;
    }
}
```

用户自定义函数

函数定义包含函数名、虚拟参数列表和代表函数体的抽象语法树。当函数被定义时，参数列表和抽象语法树将被简单地保存到符号表中函数名对应的条目中，同时替换了任何可能的旧版本。

```
/* 定义函数 */
void
dodef(struct symbol *name, struct symlist *syms, struct ast *func)
{
    if(name->syms) symlistfree(name->syms);
    if(name->func) treefree(name->func);
    name->syms = syms;
    name->func = func;
}
```

比如你定义一个函数来计算两个参数中的最大值：

```
> let max(x,y) = if x >= y then x; else y;;
> max(4+5,6+7)
```

这个函数有两个虚拟参数，x和y。当函数被调用时，求值程序将如下执行：

1. 计算实际参数值，这个例子中就是4+5和6+7。

2. 保存虚拟参数的当前值，然后把实际参数值赋予它们。
3. 执行函数体，在涉及到虚拟参数的地方都使用实际参数值。
4. 将虚拟参数恢复原值。
5. 返回函数体表达式的值。

具体的代码计算参数的个数，为虚拟参数分配了两个临时数组来储存旧值和新值，然后执行上述步骤。

```

static double
calluser(struct ufnccall *f)
{
    struct symbol *fn = f->s;      /* 函数名 */
    struct symlist *sl;             /* 虚拟参数 */
    struct ast *args = f->l;        /* 实际参数 */
    double *oldval, *newval;        /* 保存的参数值 */
    double v;
    int nargs;
    int i;

    if(!fn->func) {
        yyerror("call to undefined function", fn->name);
        return 0;
    }

    /* 计算参数个数 */
    sl = fn->syms;
    for(nargs = 0; sl; sl = sl->next)
        nargs++;

    /* 为保存参数值做准备 */
    oldval = (double *)malloc(nargs * sizeof(double));
    newval = (double *)malloc(nargs * sizeof(double));
    if(!oldval || !newval) {
        yyerror("Out of space in %s", fn->name); return 0.0;
    }

    /* 计算参数值 */
    for(i = 0; i < nargs; i++) {
        if(!args) {
            yyerror("too few args in call to %s", fn->name);
            free(oldval); free(newval);
            return 0.0;
        }

        if(args->nodetype == 'L') { /* 是否是节点列表 */
            newval[i] = eval(args->l);
            args = args->r;
        } else { /* 是否是列表末尾 */
            newval[i] = eval(args);
            args = NULL;
        }
    }
}

```

```

}

/* 保存虚拟参数的旧值，赋予新值 */
sl = fn->syms;
for(i = 0; i < nargs; i++) {
    struct symbol *s = sl->sym;

    oldval[i] = s->value;
    s->value = newval[i];
    sl = sl->next;
}

free(newval);

/* 计算函数 */
v = eval(fn->func);

/* 恢复虚拟参数的值 */
sl = fn->syms;
for(i = 0; i < nargs; i++) {
    struct symbol *s = sl->sym;

    s->value = oldval[i];
    sl = sl->next;
}

free(oldval);
return v;
}

```

使用高级计算器

这个计算器足够灵活，可做很多有用的计算。例3-9展示了函数sq，它使用Newton的方法来迭代计算平方根；此外还有函数avg，它可以计算两个数值的平均值。

例3-9：使用计算器计算平方根

```

> let sq(n)=e=1; while |((t=n/e)-e)|.001 do e=avg(e,t);;
Defined sq
> let avg(a,b)=(a+b)/2;
Defined avg
> sq(10)
= 3.162
> sqrt(10)
= 3.162
> sq(10)-sqrt(10)
= 0.000178 accurate to better than the .001 cutoff

```

练习

1. 请在高级计算器中尝试一些不同的语法。在前例中，函数`sq`不得不使用两个分号来分别结束`while`循环和`let`语句，这有些臃肿。你可以修改这个语法使它变得更为直观吗？如果你可以添加结束符号至条件语句`if/then/else/fi`和循环语句`while/do/done`，你能够使语句列表的语法更加灵活吗？
2. 在最后一个例子中，用户自定义函数计算所有的实际参数值，把它们放在一个临时数组中，然后赋予虚拟参数。为什么不是在计算实际参数值时就直接赋予虚拟参数呢？

第4章

分析SQL

SQL（Structured Query Language，结构化查询语言，发音为*sequel*）是最常用的处理关系型数据库的语言^(注1)。我们将开发一个SQL语法分析器，它可以处理简洁的记号化版本SQL语句。

这个语法分析器将基于流行的MySQL开源数据库所使用的SQL版本。MySQL实际上也使用bison语法分析器来分析它的SQL输入，不过出于各种原因，我们的语法分析器并没有建立在MySQL的语法分析器之上，而是基于手册中SQL语言的描述来实现的。

实际的MySQL的语法分析器更长更复杂，我们这儿出于讲解目的，在例子里省略了很多较少用到的部分。MySQL的语法分析器采用了特别的方式来开发，它通过bison来生成C语言的语法分析器，但是用C++编译器来编译，而词法分析器则是手写的C++代码。此外还有一些细节，不过MySQL的许可证不允许它们被摘录在这样的书里面。如果你感兴趣的话，你可以查阅文件sql/sql_yacc.yy，它是源代码 (<http://dev.mysql.com/downloads/mysql/5.1.html>) 的一部分。

SQL的最终定义可以参见ANSI和ISO发布的标准文档，其中包括ISO/IEC 9075-2:2003，它定义了SQL还有一些其他相关文档，这些文档定义了在其他编程语言和XML中嵌入SQL的方法。

SQL概述

SQL是为关系型数据库特别设计的语言。不同于操纵内存中的数据，它操纵数据表中的数据。

注1： SQL是数据库中的Fortran——很少有人喜欢它，这个语言既丑陋又古怪，但每种数据库都支持它，而且我们都使用它。

关系型数据库

一个数据库 (*database*) 是数据表 (*table*) 的集合，这儿的数据表类似于文件。每个数据表都具有行 (*row*) 和列 (*column*)，它们类似于记录和字段。数据表中的行并不基于任何特定顺序保存。你通过指定每列的名字和类型来创建表。

```
CREATE TABLE Foods (
    name CHAR(8) NOT NULL,
    type CHAR(5),
    flavor CHAR(6),
    PRIMARY KEY ( name )
)

CREATE TABLE Courses (
    course CHAR(8) NOT NULL PRIMARY KEY,
    flavor CHAR(6),
    sequence INTEGER
)
```

这种语法完全是自由格式，而且经常可以通过若干种不同的语法规则来写出相同的东西——注意对于PRIMAY KEY修饰符我们使用了两种不同的方法（表中的*primary key*可以是一个字段或者一组字段，它们唯一确定一条记录）。表4-1展示了我们刚刚创建的数据表在加载数据之后的情况。

表4-1：两个关系表

Foods			Courses		
<i>name</i>	<i>type</i>	<i>flavor</i>	<i>course</i>	<i>flavor</i>	<i>sequence</i>
peach	fruit	sweet	salad	savory	1
tomato	fruit	savory	main	savory	2
lemon	fruit	sour	dessert	sweet	3
lard	fat	bland			
cheddar	fat	savory			

SQL实现了元组演算 (*tuple calculus*)，元组 (*tuple*) 代表了记录中的关系部分，它是有序的字段或者表达式的列表。为了使用一个数据库，你需要告诉数据库具体从数据中抽取的元组。数据库会决定如何从它的数据表来获取元组（这就是演算部分）。指定一组需要的数据就是查询 (*query*)。例如，使用表4-1中的两个数据表来获得水果的列表，你可以如下实现：

```
SELECT name, flavor
FROM Foods
WHERE Foods.type = "fruit"
```

结果如表4-2所示。

表4-2：SQL结果表

name	flavor
peach	sweet
tomato	savory
lemon	sour

你也可以跨越多个数据表来进行询问。为了获得适合每道菜的食物列表，可以如下实现：

```
SELECT course, name, Foods.flavor, type  
FROM Courses, Foods  
WHERE Courses.flavor = Foods.flavor
```

结果显示在表4-3中。

表4-3：第二个SQL结果表

course	name	flavor	type
salad	tomato	savory	fruit
salad	cheddar	savory	fat
main	tomato	savory	fruit
main	cheddar	savory	fat
dessert	peach	sweet	fruit

当书写每个字段名时，在字段没有歧义的情况下，我们可以省略表名。

关系操作

SQL有很丰富的表操作命令。你可以使用SELECT、INSERT、UPDATE和DELETE命令来读和写不同的记录。SELECT语句有一个非常复杂的语法，它可以让你查找字段值、比较字段、进行算术运算，以及计算最小值、最大值、平均值和分组累计值。

使用SQL的三种方法

在SQL的最初版本中，用户首先在文件中准备需要执行的命令或者直接在终端上输入命令，结果都会立即返回。现在人们有时候依然使用这种方法来创建数据表和调试SQL，但对于绝大多数应用程序而言，SQL命令来自于程序，执行结果也返回给程序。SQL标

准定义了一种“模块化语言”来把SQL嵌入到各种各样的编程语言中，但是MySQL通过使用子例程调用的方式来实现这一点，这些例程在用户程序和数据库之间进行通信，而且它根本不使用模块化语言。由于SQL的语法非常长，我们在附录里记载了整个语法，并且为语法中的每个符号制作了交叉引用。

从SQL到逆波兰式 (RPN)

我们记号化的SQL版本将使用逆波兰式 (Reverse Polish Notation, RPN)，HP计算器的使用者会比较熟悉它。在1920年，波兰逻辑学家Jan Łukasiewicz^(注2) 意识到如果你在逻辑表达式中把操作符放在操作数之前的话，你将不需要任何圆括号或者其他标点符号来描述计算的顺序：

```
(a+b)*c * + a b c  
a+(b*c) + a * b c
```

它在反向的情况下也一样工作，你可以把操作符放到操作数之后：

```
(a+b)*c a b + c *  
a+(b*c) a b c * +
```

在计算机中，逆波兰式的价值在于它很容易使用堆栈进行解释。计算机按顺序处理每个记号。如果被处理记号是一个操作数，它把这个记号压入堆栈。如果被处理记号是一个操作符，正确数量的操作符将从堆栈弹出，进行计算，然后结果再压入堆栈。这个技巧众所周知，而且自1954年开始就被用来构造基于堆栈解释逆波兰式代码的软件和硬件。

对于编译器开发者来说，逆波兰式还有另外两个优点。一个是你使用自底向上的语法分析器（就像bison生成的分析器那样），它非常容易产生逆波兰式。如果你在识别动作代码的规则里发出 (emit) 相应操作符或操作数的动作代码，你的代码将是基于逆波兰式的顺序。以下是SQL语法分析器的部分掠影：

```
expr: NAME { emit("NAME %s", $1); }  
| INTNUM { emit("NUMBER %d", $1); }  
| expr '+' expr { emit("ADD"); }  
| expr '-' expr { emit("SUB"); }  
| expr '*' expr { emit("MUL"); }  
| expr '/' expr { emit("DIV"); }
```

当这个语法分析器分析 $a+2*3$ 时，它输出 NAME a, NUMBER 2, NUMBER 3, MUL, ADD。这个可爱的特性来自于LALR语法分析器的工作方式，部分分析的规则中的语法符号被

注2： 它被称为逆波兰式是因为大家不知道怎么发音Łukasiewicz。它的大致发音为WOOkashay-vits。

压入内部堆栈，然后在规则结束时弹出语法符号并压入新的左部符号，这一系列的操作正是逆波兰式分析器所做的事情。

另外一个优点是它很容易把逆波兰式记号的字符串转化为抽象语法树，倒过来也一样成立。为了把逆波兰式转化为抽象语法树，你只需要遍历整个逆波兰式，压入每个操作数，对于每个操作符弹出操作数，使用弹出的操作数和当前的操作符构造抽象语法树节点，然后压入新的树节点的地址。当你完成整个遍历后，堆栈将包含抽象语法树的根节点。对于从抽象语法树转化为逆波兰式的情况，你可以做一次抽象语法树的深度优先遍历。首先从抽象语法树的根节点开始，对于每个节点访问它的子节点（通过递归地调用树遍历例程），然后发出每个节点的操作符。对于叶节点，你只需要发出该节点的操作数。

经典的逆波兰式对每个操作符有固定数量的操作数，但对我们这个规则会稍稍放松一些，允许有些操作符可以有可变数量的操作数，而操作数的数量也将作为操作符的一部分。例如：

```
select a,b,c from d;  
  
rpn: NAME a  
rpn: NAME b  
rpn: NAME c  
rpn: TABLE d  
rpn: SELECT 3
```

SELECT语句中的3告诉逆波兰式解释器这个语句选择了三样事物，所以当它弹出表名后，它需要从堆栈中取出三个字段名。我写过逆波兰式代码的解释器，这个技巧使得代码变得更加简单。另外还有一种方法是使用额外的操作符，在主要操作符被处理之前先把多个操作数合并成一个，这种方法就复杂多了。

词法分析器

首先我们需要一个词法分析器来识别SQL所使用的记号。SQL的语法是自由格式的，空白字符仅仅用来分隔单词。它有相当长但是固定集合的保留字。其他记号都是很常见的：名称、字符串、数字和标点符号。注释是Ada风格的，从两个破折号开始到该行结束，MySQL的扩展也支持C语言风格的注释。

例4-1：MySQL词法分析器

```
/*  
 * mysql子集的词法分析器  
 * $Header: /usr/home/johnl/flnb/RCS/ch04.tr,v 1.7 2009/05/19 18:28:27 johnl Exp $  
 */  
  
%option noyywrap nodefault yylineno case-insensitive  
{
```

```

#include "pmysql.tab.h"
#include <stdarg.h>
#include <string.h>

void yyerror(char *s, ...);

int oldstate;

%}

%x COMMENT
%s BTWMODE

%%

```

例4-1中的词法分析器以一小段包含文件开始，注意这里的pmysql.tab.h是bison生成的记号名称定义文件。该词法分析器还定义了两个起始状态，独占的COMMENT状态被用在C语言风格的注释中，共享的BTWMODE状态被用于处理SQL表达式中AND关键字的一个特殊用法。

分析SQL关键字

SQL有大量的关键字：

```

/* 关键字 */

ADD { return ADD; }
ALL { return ALL; }
ALTER { return ALTER; }
ANALYZE { return ANALYZE; }

/* BETWEEN ... AND ...的处理技巧
 * 如果发现BETWEEN的话，返回特殊的AND记号
 */
<BTWMODE>AND { BEGIN INITIAL; return AND; }
AND { return ANDOP; }
ANY { return ANY; }
AS { return AS; }
ASC { return ASC; }
AUTO_INCREMENT { return AUTO_INCREMENT; }
BEFORE { return BEFORE; }
BETWEEN { BEGIN BTWMODE; return BETWEEN; }
INT8|BIGINT { return BIGINT; }
BINARY { return BINARY; }
BIT { return BIT; }
BLOB { return BLOB; }
BOTH { return BOTH; }
BY { return BY; }
CALL { return CALL; }
CASCADE { return CASCADE; }
CASE { return CASE; }
CHANGE { return CHANGE; }
CHAR(ACTER)? { return CHAR; }

```

```
CHECK { return CHECK; }
COLLATE { return COLLATE; }
COLUMN { return COLUMN; }
COMMENT { return COMMENT; }
CONDITION { return CONDITION; }
CONSTRAINT { return CONSTRAINT; }
CONTINUE { return CONTINUE; }
CONVERT { return CONVERT; }
CREATE { return CREATE; }
CROSS { return CROSS; }
CURRENT_DATE { return CURRENT_DATE; }
CURRENT_TIME { return CURRENT_TIME; }
CURRENT_TIMESTAMP { return CURRENT_TIMESTAMP; }
CURRENT_USER { return CURRENT_USER; }
CURSOR { return CURSOR; }
DATABASE { return DATABASE; }
DATABASES { return DATABASES; }
DATE { return DATE; }
DATETIME { return DATETIME; }
DAY_HOUR { return DAY_HOUR; }
DAY_MICROSECOND { return DAY_MICROSECOND; }
DAY_MINUTE { return DAY_MINUTE; }
DAY_SECOND { return DAY_SECOND; }
NUMERIC|DEC|DECIMAL { return DECIMAL; }
DECLARE { return DECLARE; }
DEFAULT { return DEFAULT; }
DELAYED { return DELAYED; }
DELETE { return DELETE; }
DESC { return DESC; }
DESCRIBE { return DESCRIBE; }
DETERMINISTIC { return DETERMINISTIC; }
DISTINCT { return DISTINCT; }
DISTINCTROW { return DISTINCTROW; }
DIV { return DIV; }
FLOAT8|DOUBLE { return DOUBLE; }
DROP { return DROP; }
DUAL { return DUAL; }
EACH { return EACH; }
ELSE { return ELSE; }
ELSEIF { return ELSEIF; }
END { return END; }
ENUM { return ENUM; }
ESCAPED { return ESCAPED; }
EXISTS { yylval.subtok = 0; return EXISTS; }
NOT[ \t\n]+EXISTS { yylval.subtok = 1; return EXISTS; }
EXIT { return EXIT; }
EXPLAIN { return EXPLAIN; }
FETCH { return FETCH; }
FLOAT4? { return FLOAT; }
FOR { return FOR; }
FORCE { return FORCE; }
FOREIGN { return FOREIGN; }
FROM { return FROM; }
FULLTEXT { return FULLTEXT; }
GRANT { return GRANT; }
```

```
GROUP { return GROUP; }
HAVING { return HAVING; }
HIGH_PRIORITY { return HIGH_PRIORITY; }
HOUR_MICROSECOND { return HOUR_MICROSECOND; }
HOUR_MINUTE { return HOUR_MINUTE; }
HOUR_SECOND { return HOUR_SECOND; }
IF { return IF; }
IGNORE { return IGNORE; }
IN { return IN; }
INFILE { return INFILE; }
INNER { return INNER; }
INOUT { return INOUT; }
INSENSITIVE { return INSENSITIVE; }
INSERT { return INSERT; }
INT4|INTEGER { return INTEGER; }
INTERVAL { return INTERVAL; }
INTO { return INTO; }
IS { return IS; }
ITERATE { return ITERATE; }
JOIN { return JOIN; }
INDEX|KEY { return KEY; }
KEYS { return KEYS; }
KILL { return KILL; }
LEADING { return LEADING; }
LEAVE { return LEAVE; }
LEFT { return LEFT; }
LIKE { return LIKE; }
LIMIT { return LIMIT; }
LINES { return LINES; }
LOAD { return LOAD; }
LOCALTIME { return LOCALTIME; }
LOCALTIMESTAMP { return LOCALTIMESTAMP; }
LOCK { return LOCK; }
LONG { return LONG; }
LONGBLOB { return LONGBLOB; }
LONGTEXT { return LONGTEXT; }
LOOP { return LOOP; }
LOW_PRIORITY { return LOW_PRIORITY; }
MATCH { return MATCH; }
MEDIUMBLOB { return MEDIUMBLOB; }
MIDDLEINT|MEDIUMINT { return MEDIUMINT; }
MEDIUMTEXT { return MEDIUMTEXT; }
MINUTE_MICROSECOND { return MINUTE_MICROSECOND; }
MINUTE_SECOND { return MINUTE_SECOND; }
MOD { return MOD; }
MODIFIES { return MODIFIES; }
NATURAL { return NATURAL; }
NOT { return NOT; }
NO_WRITE_TO_BINLOG { return NO_WRITE_TO_BINLOG; }
NULL { return NULLX; }
NUMBER { return NUMBER; }
ON { return ON; }
ON[ \t\n]+DUPLICATE { return ONDUPLICATE; } /* 由于受限的向前查看而使用的权宜之计 */
OPTIMIZE { return OPTIMIZE; }
OPTION { return OPTION; }
```

```
OPTIONALLY { return OPTIONALLY; }
OR { return OR; }
ORDER { return ORDER; }
OUT { return OUT; }
OUTER { return OUTER; }
OUTFILE { return OUTFILE; }
PRECISION { return PRECISION; }
PRIMARY { return PRIMARY; }
PROCEDURE { return PROCEDURE; }
PURGE { return PURGE; }
QUICK { return QUICK; }
READ { return READ; }
READS { return READS; }
REAL { return REAL; }
REFERENCES { return REFERENCES; }
REGEXP|RLIKE { return REGEXP; }
RELEASE { return RELEASE; }
RENAME { return RENAME; }
REPEAT { return REPEAT; }
REPLACE { return REPLACE; }
REQUIRE { return REQUIRE; }
RESTRICT { return RESTRICT; }
RETURN { return RETURN; }
REVOKE { return REVOKE; }
RIGHT { return RIGHT; }
ROLLUP { return ROLLUP; }
SCHEMA { return SCHEMA; }
SCHEMAS { return SCHEMAS; }
SECOND_MICROSECOND { return SECOND_MICROSECOND; }
SELECT { return SELECT; }
SENSITIVE { return SENSITIVE; }
SEPARATOR { return SEPARATOR; }
SET { return SET; }
SHOW { return SHOW; }
INT2|SMALLINT { return SMALLINT; }
SOME { return SOME; }
SONAME { return SONAME; }
SPATIAL { return SPATIAL; }
SPECIFIC { return SPECIFIC; }
SQL { return SQL; }
SQLEXCEPTION { return SQLEXCEPTION; }
SQLSTATE { return SQLSTATE; }
SQLWARNING { return SQLWARNING; }
SQL_BIG_RESULT { return SQL_BIG_RESULT; }
SQL_CALC_FOUND_ROWS { return SQL_CALC_FOUND_ROWS; }
SQL_SMALL_RESULT { return SQL_SMALL_RESULT; }
SSL { return SSL; }
STARTING { return STARTING; }
STRAIGHT_JOIN { return STRAIGHT_JOIN; }
TABLE { return TABLE; }
TEMPORARY { return TEMPORARY; }
TERMINATED { return TERMINATED; }
TEXT { return TEXT; }
THEN { return THEN; }
TIME { return TIME; }
```

```
TIMESTAMP { return TIMESTAMP; }
INT1|TINYINT { return TINYINT; }
TINYTEXT { return TINYTEXT; }
TO { return TO; }
TRAILING { return TRAILING; }
TRIGGER { return TRIGGER; }
UNDO { return UNDO; }
UNION { return UNION; }
UNIQUE { return UNIQUE; }
UNLOCK { return UNLOCK; }
UNSIGNED { return UNSIGNED; }
UPDATE { return UPDATE; }
USAGE { return USAGE; }
USE { return USE; }
USING { return USING; }
UTC_DATE { return UTC_DATE; }
UTC_TIME { return UTC_TIME; }
UTC_TIMESTAMP { return UTC_TIMESTAMP; }
VALUES? { return VALUES; }
VARBINARY { return VARBINARY; }
VARCHAR(ACTER)? { return VARCHAR; }
VARYING { return VARYING; }
WHEN { return WHEN; }
WHERE { return WHERE; }
WHILE { return WHILE; }
WITH { return WITH; }
WRITE { return WRITE; }
XOR { return XOR; }
YEAR { return YEAR; }
YEAR_MONTH { return YEAR_MONTH; }
ZEROFILL { return ZEROFILL; }
```

所有这些保留字都是词法分析器中独立的记号，因为这样做最简单。注意这里的CHARACTER和VARCHAR可以简写为CHAR和VARCHAR，而INDEX和KEY彼此等价。

关键字BETWEEN切换到起始状态BTWMODE，这个状态下单词AND返回记号AND而不是ANDOP。这是因为正常情况下AND被认为是&&逻辑与操作符，除了在SQL操作符BETWEEN… AND中：

```
IF(a && b, ...) 正常情况下这两者一致
IF(a AND b, ...)
... WHERE a BETWEEN c AND d, ... 例外
```

有多种方法可以处理类似的问题，但是这样处理最简单。

同时注意短语NOT EXISTS和ON DUPLICATE被识别为单一记号，这可以避免语法分析器中的移进/归约冲突，因为NOT和ON也可能出现在其他的上下文中。为了区分EXISTS和NOT EXISTS，词法分析器使用了语法分析器生成记号编码时所使用的值。虽然这两者并不会导致二义性，但是语法分析器分析它们时需要向前查看超过一个记号，而bison通常只向前查看一个记号。我们在第9章将使用GLR语法分析器来直接处理它们。

分析数字

数字可以有多种不同表达方式：

```
/* 数字 */
-[0-9]+ { yyval.intval = atoi(yytext); return INTNUM; }

-?[0-9]+."[0-9]* |
-?"."[0-9]+ |
-?[0-9]+E[-+]?[0-9]+ |
-?[0-9]+."[0-9]*E[-+]?[0-9]+ |
-?"."[0-9]+E[-+]?[0-9]+ { yyval.floatval = atof(yytext); return APPROXNUM; }

/* 布尔值 */
TRUE { yyval.intval = 1; return BOOL; }
UNKNOWN { yyval.intval = -1; return BOOL; }
FALSE { yyval.intval = 0; return BOOL; }

/* 字符串 */
'(\\".|''|[^\n])*' |
\"(\\".|\\\"|[^\n])*\" { yyval.strval = strdup(yytext); return STRING; }

'(\\".|[^\\n])*$ { yyerror("Unterminated string %s", yytext); }
\"(\\".|[^\\n])*$ { yyerror("Unterminated string %s", yytext); }

/* 十六进制字符串 */
X'[0-9A-F]+' |
oX[0-9A-F]+ { yyval.strval = strdup(yytext); return STRING; }

/* 二进制字符串 */
0B[01]+ |
B'[01]+' { yyval.strval = strdup(yytext); return STRING; }
```

SQL数字与前几章我们看到的数字相差无几。分析它们的规则把它们转化为C语言的整数或者双精度浮点数，然后存储到记号值中。布尔值包括true、false和unknown，它们被识别成保留字并返回为记号BOOL的变化。

SQL字符串使用单引号引起，字符串中的一对双引号代表单引号。MySQL对此做了扩展，它添加了双引号字符串以及字符串中以\x起始的转义字符。头两个字符串模式匹配不带有换行符的有效字符串，并且把字符串作为记号值返回，记住我们需要做一份拷贝，因为yytext的值并不会保留很久^(注3)。接下来两个模式匹配没有终结的字符串并且打印恰当的调试信息。

下面四个模式匹配十六进制和二进制数字，这两者都可以用两种方法实现。更现实的例子是把它们转化为二进制数，但我们这儿只需要把它们作为字符串返回就可以了。

注3： MySQL实际上允许多行字符串，但我们尽量让例子变得简单。

分析操作符和标点符号

操作符和标点符号可以被如下一些模式捕获：

```
/* 操作符 */
[-+&~|^/*(),.;!] { return yytext[0]; }

"&&" { return ANDOP; }
"||" { return OR; }

"==" { yylval.subtok = 4; return COMPARISON; }
"<=>" { yylval.subtok = 12; return COMPARISON; }
">=" { yylval.subtok = 6; return COMPARISON; }
">" { yylval.subtok = 2; return COMPARISON; }
"<=" { yylval.subtok = 5; return COMPARISON; }
"<" { yylval.subtok = 1; return COMPARISON; }
"!=" |
"<>" { yylval.subtok = 3; return COMPARISON; }

"><<" { yylval.subtok = 1; return SHIFT; }
">>>" { yylval.subtok = 2; return SHIFT; }

":=" { return ASSIGN; }
```

接下来是标点符号，它使用标准技巧在一个模式里匹配所有单一字符的操作符。MySQL 具有常见的比较操作符，它们都被认为是COMPARISON操作符，然后通过记号值来区分。我们在后面会看到这种方法并不能完美地工作，因为记号 = 也会出现在一些不是比较操作的地方，但我们会有一些应急之策。

分析函数和名字

最后需要捕获的部分是函数和名字：

```
/* 函数 */

SUBSTR(ING)?/ "(" { return FSUBSTRING; }
TRIM/"/ "(" { return FTRIM; }
DATE_ADD/"/ "(" { return FDATE_ADD; }
DATE_SUB/"/ "(" { return FDATE_SUB; }

/* check trailing context manually */
COUNT { int c = input(); unput(c);
        if(c == '(') return FCOUNT;
        yylval.strval = strdup(yytext);
        return NAME; }

/* 名字 */

[A-Za-z][A-Za-z0-9_]* { yylval.strval = strdup(yytext);
                        return NAME; }
```

```

`[^`\\.\n]+`      { yyval.strval = strdup(yytext+1);
                     yyval.strval[yyleng-2] = 0;
                     return NAME; }

`[^`\n]*$ { yyerror("unterminated quoted name %s", yytext); }

/* 用户变量 */
@[0-9a-zA-Z_.]+ |
@\"[^`\n]+\" |
@`[^`\n]+` |
@'[^`\n]+` { yyval.strval = strdup(yytext+1); return USERVAR; }

@\"[^`\n]*$ |
@`[^`\n]*$ |
@'[^`\n]*$ { yyerror("unterminated quoted user variable %s", yytext); }

```

标准SQL只有一小部分函数，它们的名字就是有效的关键字，但是MySQL加入了更多的函数并且允许你定义自己的函数，所以它们不得不在分析器中通过上下文来识别。MySQL通常认为函数名后必须带有左圆括号，所以我们可以使用尾部上下文规则来进行检查。不过，MySQL提供了一个选项来关闭左圆括号的检查。COUNT的模式提供了一种备选方案来实现尾部上下文，它通过input()和unput()来查看下一个字符。这种方法比起尾部上下文模式来说不那么优美，但是它的优点是你的代码可以在运行时来决定是否进行测试和返回何种记号。

名字由字母开始，后面可以跟随字母、数字和下划线。这个模式应该在所有的保留字匹配完之后才能够进行匹配，所以保留字模式放在它的前面。当一个名字被识别时，词法分析器返回它的一份拷贝。名字也可以通过反引号引起，这种情况下名字允许由任意字符串组成。词法分析器把引起的名字当作没有引起一样看待，反引号会被消除。下一个模式捕获没有结束反引号的情况，也就是一个字符串以反引号开始，但是一直到该行结束都没有出现另外一个反引号。

用户变量是MySQL针对标准SQL的扩展，它隶属于用户会话而不是整个数据库。它们的名字以符号@起始，而且可以使用三种引号中的任何一种来包含任意字符。

我们还有一些模式来匹配没有结束引号的用户变量名。它们使用\n而不是\$来结尾，这样可以避免flex的“危险的尾部上下文”警告。模式尾部的\$等价于/\n，对于使用同一动作的带有尾部上下文的多个模式来说，其处理被证明是很低效的。在这个例子里，\n不会用于其他用途，所以我们可以让模式匹配新行，但是如果确实需要把\n用于其他目的的话，我们可能不得不为这三个模式都拷贝一份动作代码。

注释和其他

```

/*注释 */
#.* ;

```

```

"--"[ \t].* ;

/*          { oldstate = YY_START; BEGIN COMMENT; }
<COMMENT> /* { BEGIN oldstate; }
<COMMENT>.|\n ;
<COMMENT><<EOF>> { yyerror("unclosed comment"); }

/* 其他一切 */
[ \t\n]      /* whitespace */
.           { yyerror("mystery character '%c'", *yytext); }

%%

```

最后一部分模式忽略空白字符，统计空白字符为换行符的行数，忽略注释，并且当输入中存在无效字符时进行警告。C语言注释的模式使用独占的起始状态COMMENT来消除注释内容。模式<<EOF>>捕获所有直到输入文件末尾都没有结束的C语言风格的注释。

语法分析器

例4-2中展示的SQL语法分析器比我们之前看到的任何分析器都要长，但我们可以逐步理解它。

例4-2：MySQL子集语法分析器

```

/*
 * 针对MySQL子集的语法分析器
 * $Header: /usr/home/johnl/flnb/RCS/ch04.tr,v 1.7 2009/05/19 18:28:27 johnl Exp $
 */
%{
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>

void yyerror(char *s, ...);
void emit(char *s, ...);
%}

%union {
    int intval;
    double floatval;
    char *strval;
    int subtok;
}

```

语法分析器以常见的包含语句和两个函数原型开始，一个函数是yyerror()，与第3章中的函数一致；另外一个函数是emit()，用来产生逆波兰式代码，它使用printf风格的格式字符串与参数。

%union有四个成员，都在前面的词法分析器中出现过：整数、浮点数、指向字符串拷贝

的指针以及用于带有子类型的记号的subtok。由于intval和subtok都是整数，所以即使我们使用单一域来表示两者的话也行得通，但分开表示可以帮助理解它们源自两种不同的目的：数值和子类型。

```
/* 名字和字面值 */

%token <strval> NAME
%token <strval> STRING
%token <intval> INTNUM
%token <intval> BOOL
%token <floatval> APPROXNUM

/* 使用类似@abc的名字 */

%token <strval> USERVAR

/* 操作符和优先级 */

%right ASSIGN
%left OR
%left XOR
%left ANDOP
%nonassoc IN IS LIKE REGEXP
%left NOT '!'
%left BETWEEN
%left <subtok> COMPARISON /* = <> <> <= >= <=> */
%left '|'
%left '&'
%left <subtok> SHIFT /* << >> */
%left '+' '-'
%left '*' '/' '%' MOD
%left '^'
%nonassoc UMINUS
```

下面是记号声明，与词法分析器中使用的记号保持一致。像C语言一样，MySQL也有大量的优先级，但是bison对于处理它们没有任何问题，只要你能够定义它们。对于记号COMPARISON和SHIFT而言，它们都拥有从词法分析器返回的针对特定操作符和移进方向的subtok值。

接下来是一长串的保留字。其中有些与前面定义的记号重复。bison允许重复的记号声明，而且把所有的保留字按字母顺序放在同一处声明也比较方便。全部的记号都列在附录的交叉引用中，所以这儿我们只写了几个有代表性的记号。注意EXISTS的定义的特殊之处，根据词法分析器读取的输入，它可能对应于EXISTS，也可能对应于NOT EXISTS。

```
%token ADD
%token ALL
...
%token ESCAPED
%token <subtok> EXISTS /* NOT EXISTS 或者 EXISTS */
```

```
...
/* 具有特殊语法的函数 */
%token FSUBSTRING
%token FTRIM
%token FDATE_ADD FDATE_SUB
%token FCOUNT
```

还有一些像';'那样的记号，它们不是操作符，因此并没有需要定义的优先级。

我们以非终结符的语义值类型定义作为定义部分的结束。由于我们在语法分析器中生成逆波兰式代码，这些语义值要么是位掩码，这样一个非终结符可以匹配一组选项；要么是数量，这样非终结符可以匹配长度可变的项目列表。

```
%type <intval> select_opts select_expr_list
%type <intval> val_list opt_val_list case_list
%type <intval> groupby_list opt_with_rollup opt_asc_desc
%type <intval> table_references opt_inner_cross opt_outer
%type <intval> left_or_right opt_left_or_right_outer column_list
%type <intval> index_list opt_for_join

%type <intval> delete_opts delete_list
%type <intval> insert_opts insert_vals insert_vals_list
%type <intval> insert_asgn_list opt_if_not_exists update_opts update_asgn_list
%type <intval> opt_temporary opt_length opt_binary opt_uz enum_list
%type <intval> column_atts data_type opt_ignore_replace create_col_list
%start stmt_list
%%
```

顶层分析规则

```
stmt_list: stmt ';'
| stmt_list stmt ';'
;
```

顶层规则也就是一系列的以分号结尾的语句，基本上与mysql命令行工具所接受的内容一致。对于stmt而言，每条语句都将定义一个或者多个可能的语法。

SQL表达式

在我们定义具体语句的语法之前，我们先定义MySQL表达式的语法，它与C语言或者Fortran语言的表达式相比，做了一定的扩展。

```
***** 表达式 ****/
expr: NAME { emit("NAME %s", $1); free($1); }
| NAME '.' NAME { emit("FIELDNAME %s.%s", $1, $3); free($1); free($3); }
| USERVAR { emit("USERVAR %s", $1); free($1); }
| STRING { emit("STRING %s", $1); free($1); }
| INTNUM { emit("NUMBER %d", $1); }
```

```
| APPROXNUM { emit("FLOAT %g", $1); }
| BOOL { emit("BOOL %d", $1); }
;
```

最简单的表达式是变量名和常量。由于在SQL表达式中的名字通常也就是数据表中的字段名，所以当语句中的多个表都使用相同的字段名时（这种情况在多个表基于相同值进行连接时十分常见），名字也可以被修饰为table.name。其他简单的表达式包括以符号@（词法分析器中已经做了处理，所以这儿不可见）开始的用户变量以及常量字符串、固定值和浮点数、布尔值。对于每一种情况，代码都仅仅发出相应的逆波兰式语句。当词法分析器返回的项目是字符串时，代码也需要释放词法分析器创建的字符串以避免内存泄漏。在一个更实际的语法分析器里，名字将首先被放入一个符号表中而不是作为字符串来传递。

```
expr: expr '+' expr { emit("ADD"); }
| expr '-' expr { emit("SUB"); }
| expr '*' expr { emit("MUL"); }
| expr '/' expr { emit("DIV"); }
| expr '%' expr { emit("MOD"); }
| expr MOD expr { emit("MOD"); }
| '-' expr %prec UMINUS { emit("NEG"); }
| expr ANDOP expr { emit("AND"); }
| expr OR expr { emit("OR"); }
| expr XOR expr { emit("XOR"); }
| expr '|' expr { emit("BITOR"); }
| expr '&' expr { emit("BITAND"); }
| expr '^' expr { emit("BITXOR"); }
| expr SHIFT expr { emit("SHIFT %s", $2==1?"left":"right"); }
| NOT expr { emit("NOT"); }
| '!' expr { emit("NOT"); }
| expr COMPARISON expr { emit("CMP %d", $2); }

/* 递归select和比较表达式 */
| expr COMPARISON '(' select_stmt ')' { emit("CMPSELECT %d", $2); }
| expr COMPARISON ANY '(' select_stmt ')' { emit("CMPANYSELECT %d", $2); }
| expr COMPARISON SOME '(' select_stmt ')' { emit("CMPANYSELECT %d", $2); }
| expr COMPARISON ALL '(' select_stmt ')' { emit("CMPALLSELECT %d", $2); }

expr: expr IS NULLX { emit("ISNULL"); }
| expr IS NOT NULLX { emit("ISNULL"); emit("NOT"); }
| expr IS BOOL { emit("ISBOOL %d", $3); }
| expr IS NOT BOOL { emit("ISBOOL %d", $4); emit("NOT"); }

| USERVAR ASSIGN expr { emit("ASSIGN @%s", $1); free($1); }
;

expr: expr BETWEEN expr AND expr %prec BETWEEN { emit("BETWEEN"); }
```

单目和双目表达式的处理很简单，只需要针对相应的操作符发出代码。比较表达式需要

额外发出一个子代码来区分具体的比较符号（子代码是基于位编码的，1代表小于，2代表大于，4代表等于）。

SQL允许递归SELECT语句，内部SELECT语句返回的值将作为外部SELECT的条件检查部分。如果内部SELECT返回多个值，SQL将检查比较表达式是某一条（ANY）、全部（ALL）还是部分（SOME）成立。虽然这可能导致很复杂的语句，但是分析它却是很简单的，因为语法分析器只需要根据稍后定义的select_stmt来分析。发出的逆波兰式代码首先是用来比较的表达式的代码，接着是SELECT代码，然后是操作符CMPSELECT、COMPANYSELECT或者CMPALLSELECT来表明它是前面表达式和SELECT的比较。

SQL有一些后缀操作符，包括IS NULL、IS TRUE和IS FALSE以及它们的取非版本，例如IS NOT FALSE（记住BOOL是指布尔常量，包括TRUE、FALSE或者UNKNOWN）。我们并没有为取非版本也发出对应的逆波兰式代码，我们仅仅发出额外的NOT操作符来逆转测试结果。

下面是MySQL针对标准SQL的扩展：内部赋值给用户变量。它们使用:=赋值操作符，词法分析器对该操作符返回的是记号ASSIGN，这样可以避免与相等比较操作符产生歧义。

在语法上不常见的BETWEEN...AND操作符测试值是否在给定的范围内。它需要一个前面提到的词法技巧，因为这个操作符中的AND和逻辑操作符中的AND存在二义性（就像其他技巧一样，这个技巧也不是十全十美的，但是它确实可以奏效）。由于bison的规则优先级通常由规则的最右记号的优先级确定，所以我们需要一个%prec来告诉这个规则使用BETWEEN的优先级。

```
val_list: expr { $$ = 1; }
  | expr ',' val_list { $$ = 1 + $3; }
  ;

opt_val_list: /* nil */ { $$ = 0 }
  | val_list
  ;

expr: expr IN '(' val_list ')' { emit("ISIN %d", $4); }
  | expr NOT IN '(' val_list ')' { emit("ISIN %d", $5); emit("NOT"); }
  | expr IN '(' select_stmt ')' { emit("CMPANYSELECT 4"); }
  | expr NOT IN '(' select_stmt ')' { emit("CMPALLSELECT 3"); }
  | EXISTS '(' select_stmt ')' { emit("EXISTSELECT"); if($1)emit("NOT"); }
  ;
```

下一组操作符使用可变长的表达式列表（MySQL手册把它们称为值列表或者va_list）。在第3章里我们需要创建语法分析树来管理多个表达式，但是逆波兰式让实现变得更加简单。由于逆波兰式解释器把每个逆波兰式的结果都放在内部堆栈里，所以

对于一个处理多个操作数的操作符而言，它只需要知道从堆栈中弹出的操作数的个数。因此在我们的逆波兰式代码中，这样的操作符包含一个表达式计数。

这也就意味着分析变长表达式的bison规则仅仅需要维护一个计数，这样就可以知道具体处理了多少表达式，而这些处理过的表达式则被储存在左部符号的值列表中，这个例子里也就是`val_list`。只有单个表达式的列表的长度为1，多个表达式的列表则比它的子列表多一个表达式。有些情况下值列表是可选的，所以`opt_val_list`要么为空，计数为零；要么就是`val_list`，它的计数代表了`val_list`具体拥有的表达式个数（注意没有显式动作的规则自动拥有默认动作`$$ = $1`）。

一旦我们有了值列表，我们就可以分析操作符`IN`和`NOT IN`，它们测试一个表达式是否在或者不在值列表中。注意发出的代码包含值列表的计数。SQL也允许值列表的内容从另一个`SELECT`语句获得。对于这样的情况，`IN`和`NOT IN`等价于`= ANY`和`!= ALL`，所以我们需要发出相应的代码。

函数

SQL只有一小部分函数，而MySQL做了大量的扩展。分析正常的函数调用是非常简单的事情，因为我们可以使用`opt_val_list`规则，其逆波兰式的`CALL`命令则带有参数的具体数量，但是有些函数的分析会比较复杂，它们具有诡异的可选语法。

```
/* 常规函数 */
expr: NAME '(' opt_val_list ')' { emit("CALL %d %s", $3, $1); free($1); }
;

/* 具有特殊语法的函数 */
expr: FCOUNT '(' '*' ')' { emit("COUNTALL") }
| FCOUNT '(' expr ')' { emit("CALL 1 COUNT"); }

expr: FSUBSTRING '(' val_list ')' { emit("CALL %d SUBSTR", $3); }
| FSUBSTRING '(' expr FROM expr ')' { emit("CALL 2 SUBSTR"); }
| FSUBSTRING '(' expr FROM expr FOR expr ')' { emit("CALL 3 SUBSTR"); }
| FTRIM '(' val_list ')' { emit("CALL %d TRIM", $3); }
| FTRIM '(' trim_ltb expr FROM val_list ')' { emit("CALL 3 TRIM"); }
;

trim_ltb: LEADING { emit("NUMBER 1"); }
| TRAILING { emit("NUMBER 2"); }
| BOTH { emit("NUMBER 3"); }
;

expr: FDATE_ADD '(' expr ',' interval_exp ')' { emit("CALL 3 DATE_ADD"); }
| FDATE_SUB '(' expr ',' interval_exp ')' { emit("CALL 3 DATE_SUB"); }
;

interval_exp: INTERVAL expr DAY_HOUR { emit("NUMBER 1"); }
| INTERVAL expr DAY_MICROSECOND { emit("NUMBER 2"); }
```

```

| INTERVAL expr DAY_MINUTE { emit("NUMBER 3"); }
| INTERVAL expr DAY_SECOND { emit("NUMBER 4"); }
| INTERVAL expr YEAR_MONTH { emit("NUMBER 5"); }
| INTERVAL expr YEAR { emit("NUMBER 6"); }
| INTERVAL expr HOUR_MICROSECOND { emit("NUMBER 7"); }
| INTERVAL expr HOUR_MINUTE { emit("NUMBER 8"); }
| INTERVAL expr HOUR_SECOND { emit("NUMBER 9"); }
;

```

在这里我们处理了5种具有特殊语法的函数：COUNT、SUBSTRING、TRIM、DATE_ADD和DATE_SUB。COUNT具有一个特殊的格式，COUNT(*)，被用来方便地计算SELECT语句返回的记录个数，在正常情况下它被用来计算表达式的个数。我们为这个特殊格式设计了一条规则，它产生一个特殊的COUNTALL操作符；而第二条规则是为正常格式准备的，它产生一个常规函数的调用。SUBSTRING是一个常见的子串操作符，它基于原始字符串、起始位置和截取字符个数来进行处理。它可以使用常规调用语法，也可以使用保留字FROM和FOR来分隔参数。我们为每种格式都定义了一条规则，它们都产生相似的代码，因为它们都具有相同的两个或者三个参数。同样地，TRIM也可以使用常规语法或者类似于TRIM(LEADING 'x' FROM a)的特殊语法。我们也分析了每种格式并产生规则。在三个参数的格式里，关键字LEADING、TRAILING和BOTH将被转化为整数值1到3并作为第一个参数传递。DATE_ADD和DATE_SUB在指定日期上加或减一段时间。这个特殊格式支持不同的时间尺度，这些时间尺度也被转化为整数传递给函数。

bison在处理这样的复杂语法时很有价值，主要有两个原因：一个原因是，一旦你写下你所需要的诸如此类的规则后它们就可以发挥作用；更为重要的原因是，bison自动诊断具有二义性的语法，如果它没有报告任何冲突的话，你就知道你并没有意外地破坏现有的规则。

其他表达式

我们把其他表达式都混杂在了一起。

```

expr: CASE expr case_list END { emit("CASEVAL %d 0", $3); }
    | CASE expr case_list ELSE expr END { emit("CASEVAL %d 1", $3); }
    | CASE case_list END { emit("CASE %d 0", $2); }
    | CASE case_list ELSE expr END { emit("CASE %d 1", $2); }
;

case_list: WHEN expr THEN expr { $$ = 1; }
        | case_list WHEN expr THEN expr { $$ = $1+1; }
;

expr: expr LIKE expr { emit("LIKE"); }
    | expr NOT LIKE expr { emit("LIKE"); emit("NOT"); }
;
expr: expr REGEXP expr { emit("REGEXP"); }
;
```

```

| expr NOT REGEXP expr { emit("REGEXP"); emit("NOT"); }
;

expr: CURRENT_TIMESTAMP { emit("NOW") };
| CURRENT_DATE { emit("NOW") };
| CURRENT_TIME { emit("NOW") };
;

expr: BINARY expr %prec UMINUS { emit("STRTOBIN"); }
;

```

语句CASE有两种格式。第一种格式里，CASE后紧跟着一个值，它被用来和一系列表达式进行比较，此外还有一个可选的ELSE作为默认选择，例如CASE a WHEN 100 THEN 1 WHEN 200 THEN 2 ELSE 3 END。另外一个格式只是一系列的条件表达式，例如CASE WHEN a=100 THEN 1 WHEN a=200 THEN 2 END。我们使用规则case_list来构造WHEN/THEN的成对表达式列表，并把它用在CASE的四种情况里，也就是前面两种格式带或者不带ELSE的情况。对于有或者没有初始值，其相应的逆波兰式是CASEVAL和CASE，参数包括WHEN/THEN成对表达式的个数，以及由有无ELSE分支决定的1或者0。操作符LIKE和REGEXP也拥有自己的匹配模式。它们基本上就是双目操作符，此外它们也允许一个前缀NOT来反转测试目的。最后是三个版本的当前时间关键字，以及一个单目BINARY操作符，它可以迫使一个表达式作为二进制数而不是文本数据被看待。

Select语句

迄今为止SQL中最复杂的语句就是SELECT，它从SQL数据表中检索数据并加以整理。我们之所以首先处理它是因为它使用了几个子规则，这些子规则可以在我们分析其他语句时得到重用。

```

/* 语句: select语句 */

stmt: select_stmt { emit("STMT"); }
;

select_stmt: SELECT select_opts select_expr_list 简单的无数据表select
            { emit("SELECTNODATA %d %d", $2, $3); } ;
| SELECT select_opts select_expr_list 有数据表select
  FROM table_references
  opt_where opt_groupby opt_having opt_orderby opt_limit
  opt_into_list { emit("SELECT %d %d %d", $2, $3, $5); } ;
;
```

第一个规则表明select_stmt是一条语句，它产生一个逆波兰式STMT作为语句之间的分隔符。SELECT语句列出了SQL计算其检索的每条记录（元组）所需的表达式；一个可选（但通常总是会有）的FROM来指定包含表达式数据的表；可选的限定符，比如WHERE、

`GROUP BY`和`HAVING`，它们限定、合并和排序所检索的记录，每个限定符都有它自己的规则。

```
opt_where: /* 空 */
| WHERE expr { emit("WHERE"); };

opt_groupby: /* 空 */
| GROUP BY groupby_list opt_with_rollup
    { emit("GROUPBYLIST %d %d", $3, $4); }
;

groupby_list: expr opt_asc_desc
    { emit("GROUPBY %d", $2); $$ = 1; }
| groupby_list ',' expr opt_asc_desc
    { emit("GROUPBY %d", $4); $$ = $1 + 1; }
;

opt_asc_desc: /* 空 */ { $$ = 0; }
| ASC { $$ = 0; }
| DESC { $$ = 1; }
;

opt_with_rollup: /* 空 */ { $$ = 0; }
| WITH ROLLUP { $$ = 1; }
;

opt_having: /* 空 */
| HAVING expr { emit("HAVING"); };

opt_orderby: /* 空 */
| ORDER BY groupby_list { emit("ORDERBY %d", $3); }
;

opt_limit: /* 空 */ | LIMIT expr { emit("LIMIT 1"); }
| LIMIT expr ',' expr { emit("LIMIT 2"); }
;

opt_into_list: /* 空 */
| INTO column_list { emit("INTO %d", $2); }
;

column_list: NAME { emit("COLUMN %s", $1); free($1); $$ = 1; }
| column_list ',' NAME { emit("COLUMN %s", $3); free($3); $$ = $1 + 1; }
;
```

有些选项，比如`WHERE`、`GROUPBY`和`HAVING`，使用固定数量的表达式，而`LIMIT`则使用一个或者两个表达式。它们都有着简单的规则来匹配该选项和相应的表达式，它们会产生一个逆波兰式来表明如何处理其表达式。

`GROUP BY`和`ORDER BY`接受表达式列表，这些表达式通常是字段名，每个字段名后带有可选的`ASC`或者`DESC`来指定排列顺序。规则`groupby_list`产生一个被计数过的表达式列

表，它发出GROUPBY操作符以及表明排列顺序的操作符。规则GROUP BY和ORDER BY接着发出带有计数的GROUPLIST和ORDERBY操作符，对于GROUP BY来说，还有一个标志（flag）来表明是否使用WITH ROLLUP选项，该选项在结果中添加额外的摘要字段。

INTO操作符接受一连串的名字，它们称为column_list，也就是用来保存选中数据的字段名列表。INTO的使用并不常见，但我们在后面的好几个地方重用column_list，这些地方的语法都带有字段名列表。

Select选项和表引用

现在我们开始处理起始选项和SELECT中的主表达式列表。

```
select_opts: { $$ = 0; }
| select_opts ALL
  { if($1 & 01) yyerror("duplicate ALL option"); $$ = $1 | 01; }
| select_opts DISTINCT
  { if($1 & 02) yyerror("duplicate DISTINCT option"); $$ = $1 | 02; }
| select_opts DISTINCTROW
  { if($1 & 04) yyerror("duplicate DISTINCTROW option"); $$ = $1 | 04; }
| select_opts HIGH_PRIORITY
  { if($1 & 010) yyerror("duplicate HIGH_PRIORITY option"); $$ = $1 | 010; }
| select_opts STRAIGHT_JOIN
  { if($1 & 020) yyerror("duplicate STRAIGHT_JOIN option"); $$ = $1 | 020; }
| select_opts SQL_SMALL_RESULT
  { if($1 & 040) yyerror("duplicate SQL_SMALL_RESULT option"); $$ = $1 | 040; }
| select_opts SQL_BIG_RESULT
  { if($1 & 0100) yyerror("duplicate SQL_BIG_RESULT option"); $$ = $1 | 0100; }
| select_opts SQL_CALC_FOUND_ROWS
  { if($1 & 0200) yyerror("duplicate SQL_CALC_FOUND_ROWS option"); $$ =
    $1 | 0200; }
;

select_expr_list: select_expr { $$ = 1; }
| select_expr_list ',' select_expr { $$ = $1 + 1; }
| '*' { emit("SELECTALL"); $$ = 1; }
;

select_expr: expr opt_as_alias ;

opt_as_alias: AS NAME { emit ("ALIAS %s", $2); free($2); }
| NAME { emit ("ALIAS %s", $1); free($1); }
| /* nil */
;
```

这里的选项就是可以影响SELECT如何被处理的一些标志。关于选项是否彼此兼容的规则过于复杂，所以我们仅仅实现一部分选项，我们为它们建立了相应的位掩码，这可以让我们有能力诊断是否存在重复的选项。（选项可以以任何顺序出现，所以在语法层面上并没有什么好方法来屏蔽重复选项，通常你自己来检测会更容易，就像我们这儿做的那样。）

SELECT表达式列表是由逗号分隔的表达式列表，每个表达式都带有可选的AS子句来给表达式命名，以便于**SELECT**语句在其他地方使用。我们在逆波兰式中产生相应的ALIAS操作符。作为一个特殊例子，*意味着源记录中的所有字段，这种情况下我们发出**SELECTALL**。

SELECT表引用

SELECT中最复杂也最强大的部分，同时也是SQL最强大的部分，就是它可以指向多个表。在**SELECT**中，你可以让它创建概念上的联接表（joined table），表中数据来自于很多的实际表，可以通过显式联接或者递归的**SELECT**语句来创建。由于数据表可能会相当大，所以会有一些方法来指定如何有效地进行联接。

```
table_references: table_reference { $$ = 1; }
| table_references ',' table_reference { $$ = $1 + 1; }
;

table_reference: table_factor
| join_table
;

table_factor:
NAME opt_as_alias index_hint { emit("TABLE %s", $1); free($1); }
| NAME '.' NAME opt_as_alias index_hint { emit("TABLE %s.%s", $1, $3);
                                         free($1); free($3); }
| table_subquery opt_as NAME { emit("SUBQUERYAS %s", $3); free($3); }
| '(' table_references ')' { emit("TABLEREFERENCES %d", $2); }
;

opt_as: AS
| /* 空 */
;

join_table:
table_reference opt_inner_cross JOIN table_factor opt_join_condition
{ emit("JOIN %d", 100+$2); }
| table_reference STRAIGHT_JOIN table_factor
{ emit("JOIN %d", 200); }
| table_reference STRAIGHT_JOIN table_factor ON expr
{ emit("JOIN %d", 200); }
| table_reference left_or_right opt_outer JOIN table_factor join_condition
{ emit("JOIN %d", 300+$2+$3); }
| table_reference NATURAL opt_left_or_right_outer JOIN table_factor
{ emit("JOIN %d", 400+$3); }
;

opt_inner_cross: /* nil */ { $$ = 0; }
| INNER { $$ = 1; }
| CROSS { $$ = 2; }
;

opt_outer: /* nil */ { $$ = 0; }
```

```

| OUTER {$$ = 4; }
;

left_or_right: LEFT { $$ = 1; }
| RIGHT { $$ = 2; }
;

opt_left_or_right_outer: LEFT opt_outer { $$ = 1 + $2; }
| RIGHT opt_outer { $$ = 2 + $2; }
| /* 空 */ { $$ = 0; }
;

opt_join_condition: /* nil */
| join_condition ;

join_condition:
ON expr { emit("ONEXPR"); }
| USING '(' column_list ')' { emit("USING %d", $3); }
;

index_hint:
USE KEY opt_for_join '(' index_list ')'
{ emit("INDEXHINT %d %d", $5, 10+$3); }
| IGNORE KEY opt_for_join '(' index_list ')'
{ emit("INDEXHINT %d %d", $5, 20+$3); }
| FORCE KEY opt_for_join '(' index_list ')'
{ emit("INDEXHINT %d %d", $5, 30+$3); }
| /* 空 */
;

opt_for_join: FOR JOIN { $$ = 1; }
| /* 空 */ { $$ = 0; }
;

index_list: NAME { emit("INDEX %s", $1); free($1); $$ = 1; }
| index_list ',' NAME { emit("INDEX %s", $3); free($3); $$ = $1 + 1; }
;

table_subquery: '(' select_stmt ')' { emit("SUBQUERY"); }
;

```

虽然数据表子语言的语法很长，但是它一点都不复杂，它由主要的条目列表和大量的可选子句构成。每个table_reference可以是一个table_factor（简单表、嵌套SELECT或者括起的列表）或者一个join_table，表明显式的联接。一个简单表的引用包括该表的名字，这个名字可以带有或者不带该表所属数据库的名字；一个可选的AS子句来给定别名（一张表可能在同一条SELECT中出现多次，而别名可以帮助SELECT识别表达式中的表名所指向的具体实例）；可选的提示来指定使用哪个索引，稍后会讲到。

嵌套的SELECT是由括号括起的SELECT语句，它必须被赋予一个名字，即使名字前的AS是可选的。table_factor也可以是括号括起的table_reference的列表，这在创建联接时很有用。

每个table_factor都可以有一个索引提示。SQL表允许索引出现在任何的字段组合上，这使它可以更快地基于这些字段进行搜索。每个索引都有一个名字，例如对于字段foo会是foo_index。通常MySQL自动使用合适的索引，但是你也可以通过USE KEY、FORCE KEY或者IGNORE KEY来更改它所选择的索引。

联接（join）指定了如何合并两组表。它带来了多种选择，从而改变了表间原本的匹配秩序，它给出了那些在一组中存在但是在另一组中无法得到匹配的记录的处理方法，它还指定了其他一些细节。每个联接也都通过各式各样的语法来显式或者隐式指定用于表间匹配的字段，例如：

```
SELECT * FROM a JOIN b ON a.foo=b.bar  
SELECT * FROM a JOIN b USING (foo) a.foo=b.foo
```

在NATUAL联接中，联接匹配具有相同名字的字段，而在常规联接中，如果没有任何列出的字段，它将导致一个叉积，也就是针对第一组的每条记录来联接第二组的每条记录。在后一种情况中，结果通常会通过WHERE和HAVING子句来进行削减。对于不同类型的联接，我们产生一个JOIN操作符以及相应的子域来指定具体的联接类型。

注意这儿被分开的规则table_factor和table_reference。它们分别设置JOIN操作符的结合性和解决一些表达式的二义性，例如a JOIN b JOIN c，它意味着(a JOIN b) JOIN c而不是a JOIN (b JOIN c)。在join_table规则中，每个联接的左边有一个table_reference，而其右边则是table_factor，这样可以保证语法的左结合性。由于table_factor可以是一个被括号括起的table_reference，所以如果它并不是你期望的情况，你也可以使用括号。这个例子里，我们本来可以让table_factor也成为table_reference，然后使用优先级来解决二义性，但是这个语法是从SQL标准中来的，我们并没有什么理由去改变它。

删除语句

一旦我们掌握了SELECT语句，其他数据操纵语句的分析将变得十分简单。当DELETE删除一张表中的记录时，它使用与SELECT中WHERE子句一致的WHERE子句来指定需要删除的记录，也可以像SELECT那样指定一组表来选择删除记录。

```
/* 语句: delete 语句 */  
  
stmt: delete_stmt { emit("STMT"); }  
;  
  
/* 单表删除 */  
delete_stmt: DELETE delete_opts FROM NAME  
    opt_where opt_orderby opt_limit  
    { emit("DELETEONE %d %s", $2, $4); free($4); }
```

```

;

delete_opts: delete_opts LOW_PRIORITY { $$ = $1 + 01; }
| delete_opts QUICK { $$ = $1 + 02; }
| delete_opts IGNORE { $$ = $1 + 04; }
| /* 空 */ { $$ = 0; }
;

```

DELETE语句重用了我们为SELECT所写的几条规则：针对可选WHERE子句的opt_where，针对可选ORDER BY子句的opt_orderby，以及针对可选LIMIT子句的opt_limit。由于针对这些规则的每条子句都会产生自己的逆波兰式，我们只需要对诸如DELETE、QUICK和IGNORE这样的特定关键字以及DELETE语句自身来编写规则。

```

/* 多表删除，第一版 */
delete_stmt: DELETE delete_opts
    delete_list
        FROM table_references opt_where
            { emit("DELETEMULTI %d %d %d", $2, $3, $5); }

delete_list: NAME opt_dot_star { emit("TABLE %s", $1); free($1); $$ = 1; }
| delete_list ',' NAME opt_dot_star
    { emit("TABLE %s", $3); free($3); $$ = $1 + 1; }
;

opt_dot_star: /* nil */ | '.' '*' ;

/* 多表删除，第二版 */
delete_stmt: DELETE delete_opts
    FROM delete_list
        USING table_references opt_where
            { emit("DELETEMULTI %d %d %d", $2, $4, $6); }
;

```

我们为多表DELETE定义了两种不同的语法，这样可以兼容于SQL的其他不同实现。一种语法列出多张表，其后紧跟着FROM和table_references；另外一种语法首先是FROM，然后是多张表、USING以及table_references。bison很容易处理这种差异，我们会为这两者产生相同的逆波兰式。delete_list有一个小小的可供选择的“语法甜头”，它允许你使用name.*来指定哪张表的记录将被删除，这和简单的name是一致的，但它可以提醒读者记录中的所有字段都会被删除。

插入和替换语句

语句INSERT和REPLACE向一张表中添加记录。它们之间的差异在于如果新记录的主键值在原有记录中已经存在的话，INSERT将直接失败并报告错误，除非它还具有一条ON DUPLICATE KEY子句，这样REPLACE会替换现有的记录。INSERT和DELETE一样，也有两个等价的格式来插入新记录，同时它还有第三种格式允许插入由SELECT创建的记录。

```

INSERT INTO a(b,c) values (1,2),(3,DEFAULT)

/* 语句: insert 语句 */

stmt: insert_stmt { emit("STMT"); }

insert_stmt: INSERT insert_opts opt_into NAME
    opt_col_names
    VALUES insert_vals_list
    opt_on_duplicate { emit("INSERTVALS %d %d %s", $2, $7, $4); free($4) }
;

opt_on_duplicate: /* 空 */
    | ONDUPLICATE KEY UPDATE insert_asgn_list { emit("DUPUPDATE %d", $4); }
;

insert_opts: /* 空 */ { $$ = 0; }
    | insert_opts LOW_PRIORITY { $$ = $1 | 01 ; }
    | insert_opts DELAYED { $$ = $1 | 02 ; }
    | insert_opts HIGH_PRIORITY { $$ = $1 | 04 ; }
    | insert_opts IGNORE { $$ = $1 | 010 ; }
;

opt_into: INTO | /* 空 */
;

opt_col_names: /* 空 */
    | '(' column_list ')' { emit("INSERTCOLS %d", $2); }
;

insert_vals_list: '(' insert_vals ')' { emit("VALUES %d", $2); $$ = 1; }
    | insert_vals_list ',' '(' insert_vals ')' { emit("VALUES %d", $4); $$ = $1 + 1; }
;

insert_vals:
    expr { $$ = 1; }
    | DEFAULT { emit("DEFAULT"); $$ = 1; }
    | insert_vals ',' expr { $$ = $1 + 1; }
    | insert_vals ',' DEFAULT { emit("DEFAULT"); $$ = $1 + 1; }
;

```

第一种格式首先指定表名和将被提供的字段列表（如果没有指定，那就是所有字段），然后指定VALUES，最后则是值列表。这个格式可以插入多条记录，因此规则`insert_vals`匹配针对一条记录的被括号括起的多个字段，而规则`insert_vals_list`匹配由逗号分隔的多个字段集合。每个字段值可以是一个表达式或者关键字DEFAULT。另外还有一些可选的关键字来控制插入的细节。

规则`opt_on_duplicate`处理ON DUPLICATE子句，它允许字段在插入记录存在重复键时进行更改。由于该语法是SET field=value而=被分析为COMPARISON操作符，所以这里我们接受COMPARISON，但是需要在我们的代码里进行检查，以保证这确实是一个等号而不是

其他什么东西^(注4)。注意ON DUPLICATE是一个记号，在词法分析器中我们把两个单词作为一个记号看待，这样可以避免与嵌套SELECT中的ON子句产生歧义。

```
INSERT INTO a SET b=1, c=2

insert_stmt: INSERT insert_opts opt_into NAME
    SET insert_asgn_list
    opt_onupdate
    { emit("INSERTASGN %d %d %s", $2, $6, $4); free($4) }
;

insert_asgn_list:
NAME COMPARISON expr
{ if ($2 != 4) { yyerror("bad insert assignment to %s", $1); YYERROR; }
  emit("ASSIGN %s", $1); free($1); $$ = 1; }
| NAME COMPARISON DEFAULT
{ if ($2 != 4) { yyerror("bad insert assignment to %s", $1); YYERROR; }
  emit("DEFAULT"); emit("ASSIGN %s", $1); free($1); $$ = 1; }
| insert_asgn_list ',' NAME COMPARISON expr
{ if ($4 != 4) { yyerror("bad insert assignment to %s", $1); YYERROR; }
  emit("ASSIGN %s", $3); free($3); $$ = $1 + 1; }
| insert_asgn_list ',' NAME COMPARISON DEFAULT
{ if ($4 != 4) { yyerror("bad insert assignment to %s", $1); YYERROR; }
  emit("DEFAULT"); emit("ASSIGN %s", $3); free($3); $$ = $1 + 1; }
;
```

第二种格式使用与ON DUPLICATE相似的赋值语句。我们也要检查COMPARISON是实际的`=`。如果不是的话，我们将通过`yyerror()`产生一条错误消息，然后我们会让语法分析器基于YYERROR开始错误恢复（这个版本的语法分析器并没有错误恢复，但是我们在第8章会看到）。这个格式在语句的末尾使用了相同的可选ON DUPLICATE子句，所以我们只要使用相同的规则就可以了。

```
INSERT into a(b,c) SELECT x,y FROM z where x < 12
insert_stmt: INSERT insert_opts opt_into NAME opt_col_names
    select_stmt
    opt_onupdate { emit("INSERTSELECT %d %s", $2, $4); free($4); }
;
```

INSERT的第三种格式使用SELECT的结果数据来创建新记录。这条语句的所有组成部分与我们前面看到的语法一致，所以我们只需要编写一条规则并重用组成部分的子规则。

替换语句

REPLACE语句的语法很像INSERT，所以它的规则也基本一致，我们只需要把INSERT改成REPLACE以及重命名顶层规则。

注4： 这可以被认为是拼凑的结果，但是对另外一种方法来说，它需要把`=`从其他比较操作符里区分出来，并且在每处比较发生的地方添加额外的规则，这将导致更多的代码。

```

/** replace就像insert */
stmt: replace_stmt { emit("STMT"); }

replace_stmt: REPLACE insert_opts opt_into NAME
    opt_col_names
    VALUES insert_vals_list
    opt_ondupupdate { emit("REPLACEVALS %d %d %s", $2, $7, $4); free($4) }
;

replace_stmt: REPLACE insert_opts opt_into NAME
    SET insert_asgn_list
    opt_ondupupdate
    { emit("REPLACEASGN %d %d %s", $2, $6, $4); free($4) }
;

replace_stmt: REPLACE insert_opts opt_into NAME opt_col_names
    select_stmt
    opt_ondupupdate { emit("REPLACESELECT %d %s", $2, $4); free($4); }
;

```

更新语句

UPDATE语句更新现有记录中的字段。同样，它的语法也允许我们重用前面语句的规则。

```

/** 更新 */
stmt: update_stmt { emit("STMT"); }

update_stmt: UPDATE update_opts table_references
    SET update_asgn_list
    opt_where
    opt_orderby
opt_limit { emit("UPDATE %d %d %d", $2, $3, $5); }

update_opts: /* nil */ { $$ = 0; }
| insert_opts LOW_PRIORITY { $$ = $1 | 01; }
| insert_opts IGNORE { $$ = $1 | 010; }
;

update_asgn_list:
NAME COMPARISON expr
{ if ($2 != 4) { yyerror("bad update assignment to %s", $1); YYERROR; }
emit("ASSIGN %s", $1); free($1); $$ = 1; }
| NAME '.' NAME COMPARISON expr
{ if ($4 != 4) { yyerror("bad update assignment to %s", $1); YYERROR; }
emit("ASSIGN %s.%s", $1, $3); free($1); free($3); $$ = 1; }
| update_asgn_list ',' NAME COMPARISON expr
{ if ($4 != 4) { yyerror("bad update assignment to %s", $3); YYERROR; }
emit("ASSIGN %s.%s", $3); free($3); $$ = $1 + 1; }
| update_asgn_list ',' NAME '.' NAME COMPARISON expr
{ if ($6 != 4) { yyerror("bad update assignment to %s.$s", $3, $5);
YYERROR; }

```

```
        emit("ASSIGN %s.%s", $3, $5); free($3); free($5); $$ = 1; }  
    ;
```

UPDATE在规则update_opts中拥有它自己的一组选项。在SET之后的一连串赋值与INSERT中的赋值是相似的，但是UPDATE允许限定过的表名，因为你可以一次更新多张表，而在INSERT中也没有默认选项，所以我们需要有一个相似但是不同的update_asgn_list。UPDATE使用与INSERT一致的opt_where和opt_orderby来限制和排序更新过的记录。

这就结束了我们的SQL子集里的数据操纵语句。MySQL有一些更多的内容在这里并没有体现，但它们的语法很容易被分析。

创建数据库

现在我们处理众多数据定义语句中的两条，它们创建和修改数据库和表的结构。

```
/** 创建数据库 **/  
stmt: create_database_stmt { emit("STMT"); }  
;  
  
create_database_stmt:  
    CREATE DATABASE opt_if_not_exists NAME  
        { emit("CREATEDATABASE %d %s", $3, $4); free($4); }  
    | CREATE SCHEMA opt_if_not_exists NAME  
        { emit("CREATEDATABASE %d %s", $3, $4); free($4); }  
    ;  
  
opt_if_not_exists: /* nil */ { $$ = 0; }  
    | IF EXISTS  
        { if(!$2) { yyerror("IF EXISTS doesn't exist"); YYERROR; }  
         $$ = $2; /* NOT EXISTS hack */ }  
    ;
```

CREATE DATABASE，或者它的等价语句CREATE SCHEMA，创建一个新的数据库，使得你可以在里面创建新表。它有一个可选子句IF NOT EXISTS，以避免在数据库存在的情况下产生错误。还记得我们通过一个特别的词法技巧在表达式中把IF NOT EXISTS和IF EXISTS看成相同的记号吗？在这个例子里，只有IF NOT EXISTS是合法的，所以我们需要在动作代码里进行测试，并且在发现它是错误的记号时通知语法分析器。

创建表

CREATE TABLE语句在它的长度和选项个数上足以与SELECT相媲美，不过它的语法相对简单许多，因为几乎所有的语法都只是声明表中每个字段的类型和属性。

我们以create_table_statement的6个不同版本开始。它们实际上是三对语法，每对语法的差异仅在于表名是NAME还是NAME.NAME。第一对是常见的版本，它在

`create_col_list`中带有显式字段列表。另外两对通过SELECT语句来创建和填充表，其中一对带有字段名列表，而另外一对则使用SELECT里的默认字段名。

```
/** 创建表 */
stmt: create_table_stmt { emit("STMT"); }

create_table_stmt: CREATE opt_temporary TABLE opt_if_not_exists NAME
  '(' create_col_list ')' { emit("CREATE %d %d %d %s", $2, $4, $7, $5); free($5); }

create_table_stmt: CREATE opt_temporary TABLE opt_if_not_exists NAME '.' NAME
  '(' create_col_list ')' { emit("CREATE %d %d %d %s.%s", $2, $4, $9, $5, $7);
    free($5); free($7); }

create_table_stmt: CREATE opt_temporary TABLE opt_if_not_exists NAME
  '(' create_col_list ')'
create_select_statement { emit("CREATESELECT %d %d %d %s", $2, $4, $7, $5); free($5); }

create_table_stmt: CREATE opt_temporary TABLE opt_if_not_exists NAME
  create_select_statement { emit("CREATESELECT %d %d 0 %s", $2, $4, $5); free($5); }

create_table_stmt: CREATE opt_temporary TABLE opt_if_not_exists NAME '.' NAME
  '(' create_col_list ')'
  create_select_statement { emit("CREATESELECT %d %d 0 %s.%s", $2, $4, $5, $7);
    free($5); free($7); }

create_table_stmt: CREATE opt_temporary TABLE opt_if_not_exists NAME '.' NAME
  create_select_statement { emit("CREATESELECT %d %d 0 %s.%s", $2, $4, $5, $7);
    free($5); free($7); }

opt_temporary: /* 空 */ { $$ = 0; }
  | TEMPORARY { $$ = 1; }
;
```

`CREATE TABLE`语句的核心在于字段列表，或者更准确地说是`create_definitions`列表，它包含字段和索引。索引可以是PRIMARY KEY，意味着它对每条记录来说都是唯一的；也可以是一个常规INDEX（也被称为KEY）；或者是一个FULLTEXT索引，它索引数据中的每个单词。这些索引都接受字段名列表，而对于字段名列表，我们再次重用了我们为SELECT定义的`column_list`规则。

```
create_col_list: create_definition { $$ = 1; }
  | create_col_list ',' create_definition { $$ = $1 + 1; }

create_definition: PRIMARY KEY '(' column_list ')' { emit("PRIKEY %d", $4); }
```

```

| KEY '(' column_list ')' { emit("KEY %d", $3); }
| INDEX '(' column_list ')' { emit("KEY %d", $3); }
| FULLTEXT INDEX '(' column_list ')' { emit("TEXTINDEX %d", $4); }
| FULLTEXT KEY '(' column_list ')' { emit("TEXTINDEX %d", $4); }
;

```

每个定义由一个逆波兰式的STARTCOL操作符括起，因为每个字段选项的集合会很长，这也帮助分隔了每个字段的选项。字段本身包括字段名、数据类型以及可选的属性，例如字段值是否可以为空、默认值是什么以及它是否是一个键值（把一个字段声明成键值也就等价于为它创建索引）。我们为每个属性产生一个ATTR操作符并且计算属性的个数。这里的代码并没有检查重复属性，但这一点也很容易实现，我们可以把column_atts的值指定为带有计数和位掩码的数据结构，然后像我们之前在SELECT选项中所做的那样来检查位掩码。

```

create_definition: { emit("STARTCOL"); } NAME data_type column_atts
                  { emit("COLUMNDEF %d %s", $3, $2); free($2); }

column_atts: /* 空 */ { $$ = 0; }
| column_atts NOT NULLX { emit("ATTR NOTNULL"); $$ = $1 + 1; }
| column_atts NULLX
| column_atts DEFAULT STRING
  { emit("ATTR DEFAULT STRING %s", $3); free($3); $$ = $1 + 1; }
| column_atts DEFAULT INTNUM
  { emit("ATTR DEFAULT NUMBER %d", $3); $$ = $1 + 1; }
| column_atts DEFAULT APPROXNUM
  { emit("ATTR DEFAULT FLOAT %g", $3); $$ = $1 + 1; }
| column_atts DEFAULT BOOL
  { emit("ATTR DEFAULT BOOL %d", $3); $$ = $1 + 1; }
| column_atts AUTO_INCREMENT
  { emit("ATTR AUTOINC"); $$ = $1 + 1; }
| column_atts UNIQUE '(' column_list ')'
  { emit("ATTR UNIQUEKEY %d", $4); $$ = $1 + 1; }
| column_atts UNIQUE KEY { emit("ATTR UNIQUEKEY"); $$ = $1 + 1; }
| column_atts PRIMARY KEY { emit("ATTR PRIKEY"); $$ = $1 + 1; }
| column_atts KEY { emit("ATTR PRIKEY"); $$ = $1 + 1; }
| column_atts COMMENT STRING
  { emit("ATTR COMMENT %s", $3); free($3); $$ = $1 + 1; }
;

```

数据类型的语法很长但并不复杂。很多类型允许指定字符或者数字的个数，所以这儿有一个opt_length，它接受一个或者两个长度值（我们通过编码把它们都放到一个数值里，使用结构类型会更优美一些）。其他选项表示数值是否是无符号数，显示时是否需要填充零，字符串是否需要作为二进制数看待，以及对于文本数据需要使用什么样的字符集和校对规则。最后两个指定为具有不同语言和校对规则的系统中的字符串，但我们这儿只需要接受任意字符串就可以。通过这些辅助规则，我们现在可以分析MySQL长长的数据类型列表。同样，我们把数据类型转为数值，而且虽然使用结构类型会更加优美，数值对于我们的逆波兰式也一样适用。

最后两个类型， ENUM和SET， 每个都接受一个字符串列表， 这些字符串命名了枚举或集合的成员， 我们把这些字符串分析为enum_val。

```
opt_length: /* 空 */ { $$ = 0; }
| '(' INTNUM ')' { $$ = $2; }
| '(' INTNUM ',' INTNUM ')' { $$ = $2 + 1000*$4; }
;

opt_binary: /* 空 */ { $$ = 0; }
| BINARY { $$ = 4000; }
;

opt_uz: /* 空 */ { $$ = 0; }
| opt_uz UNSIGNED { $$ = $1 | 1000; }
| opt_uz ZEROFILL { $$ = $1 | 2000; }
;

opt_csc: /* 空 */
| opt_csc CHAR SET STRING { emit("COLCHARSET %s", $4); free($4); }
| opt_csc COLLATE STRING { emit("COLCOLLATE %s", $3); free($3); }
;

data_type:
BIT opt_length { $$ = 10000 + $2; }
| TINYINT opt_length opt_uz { $$ = 10000 + $2; }
| SMALLINT opt_length opt_uz { $$ = 20000 + $2 + $3; }
| MEDIUMINT opt_length opt_uz { $$ = 30000 + $2 + $3; }
| INT opt_length opt_uz { $$ = 40000 + $2 + $3; }
| INTEGER opt_length opt_uz { $$ = 50000 + $2 + $3; }
| BIGINT opt_length opt_uz { $$ = 60000 + $2 + $3; }
| REAL opt_length opt_uz { $$ = 70000 + $2 + $3; }
| DOUBLE opt_length opt_uz { $$ = 80000 + $2 + $3; }
| FLOAT opt_length opt_uz { $$ = 90000 + $2 + $3; }
| DECIMAL opt_length opt_uz { $$ = 110000 + $2 + $3; }
| DATE { $$ = 100001; }
| TIME { $$ = 100002; }
| TIMESTAMP { $$ = 100003; }
| DATETIME { $$ = 100004; }
| YEAR { $$ = 100005; }
| CHAR opt_length opt_csc { $$ = 120000 + $2; }
| VARCHAR '(' INTNUM ')' opt_csc { $$ = 130000 + $3; }
| BINARY opt_length { $$ = 140000 + $2; }
| VARBINARY '(' INTNUM ')' { $$ = 150000 + $3; }
| TINYBLOB { $$ = 160001; }
| BLOB { $$ = 160002; }
| MEDIUMBLOB { $$ = 160003; }
| LONGBLOB { $$ = 160004; }
| TINYTEXT opt_binary opt_csc { $$ = 170000 + $2; }
| TEXT opt_binary opt_csc { $$ = 171000 + $2; }
| MEDIUMTEXT opt_binary opt_csc { $$ = 172000 + $2; }
| LONGTEXT opt_binary opt_csc { $$ = 173000 + $2; }
| ENUM '(' enum_list ')' opt_csc { $$ = 200000 + $3; }
| SET '(' enum_list ')' opt_csc { $$ = 210000 + $3; }
;
```

```
enum_list: STRING { emit("ENUMVAL %s", $1); free($1); $$ = 1; }
| enum_list ',' STRING { emit("ENUMVAL %s", $3); free($3); $$ = $1 + 1; }
;
```

CREATE的另外一种版本也使用SELECT语句，在SELECT语句之前还带有一些可选的关键字和可选的但并无意义的AS。

```
create_select_statement: opt_ignore_replace opt_as select_stmt { emit("CREATESELECT %d", $1) }
;

opt_ignore_replace: /* 空 */ { $$ = 0; }
| IGNORE { $$ = 1; }
| REPLACE { $$ = 2; }
;
```

用户变量

我们最后需要分析的是语句SET，它是MySQL的扩展，允许设置用户变量。它的赋值可以使用:=，我们称为ASSIGN；也可以使用简单的=符号，这里的检查是不可避免的，因为我们需要保证它不是其他的比较操作符。

```
***** 设置用户变量 *****

stmt: set_stmt { emit("STMT"); }
;

set_stmt: SET set_list ;
set_list: set_expr | set_list ',' set_expr ;
set_expr:
USERVAR COMPARISON expr { if ($2 != 4) { yyerror("bad set to @%s", $1); YYERROR; }
                           emit("SET %s", $1); free($1); }
| USERVAR ASSIGN expr { emit("SET %s", $1); free($1); }
;
```

这就结束了我们的SQL语法。MySQL还有许许多多的其他语句，但这些语句已经足以帮助我们认识到语法分析过程中所需要注意的事情。

语法分析器的例程

最后，我们需要一些辅助例程。目前的emit例程仅仅帮助打印逆波兰式。在更现实的编译器里，它将更像一个简单的汇编程序，来发出适合于逆波兰式解释器的基于字节码操作符的数据流。

你应该已经从前一章里熟悉了例程yyerror和main。这个主程序支持-d参数来打开分析时的调试信息，当我们调试这样复杂的语法时这会是一个很有用的特性。

```

%%

void
emit(char *s, ...)
{
    extern yylineno;

    va_list ap;
    va_start(ap, s);

    printf("rpn: ");
    vfprintf(stdout, s, ap);
    printf("\n");
}

void
yyerror(char *s, ...)
{
    extern yylineno;

    va_list ap;
    va_start(ap, s);

    fprintf(stderr, "%d: error: ", yylineno);
    vfprintf(stderr, s, ap);
    fprintf(stderr, "\n");
}

main(int ac, char **av)
{
    extern FILE *yyin;

    if(ac > 1 && !strcmp(av[1], "-d")) {
        yydebug = 1; ac--; av++;
    }

    if(ac > 1 && (yyin = fopen(av[1], "r")) == NULL) {
        perror(av[1]);
        exit(1);
    }

    if(!yyparse())
        printf("SQL parse worked\n");
    else
        printf("SQL parse failed\n");
} /* 主程序 */

```

SQL分析器的Makefile

Makefile通过flex和bison分别地生成词法分析器和语法分析器，然后把它们放在一起编译。Makefile的依赖性会负责生成和编译词法分析器，包括bison所生成的头文件。

```
# pmysql的Makefile
CC = cc -g
```

```
LEX = flex
YACC = bison
CFLAGS = -DYYDEBUG=1

PROGRAMS5 = pmysql

all: ${PROGRAMS5}

# chapter 5

pmysql: pmysql.tab.o pmysql.o
        ${CC} -o $@ pmysql.tab.o pmysql.o

pmysql.tab.c pmysql.tab.h: pmysql.y
        ${YACC} -vd pmysql.y

pmysql.c: pmysql.l
        ${LEX} -o $*.c $<

pmysql.o: pmysql.c pmysql.tab.h

.SUFFIXES: .pgm .l .y .c
```

练习

1. 在有些地方，SQL语法分析器允许比SQL自身限制更宽泛的语法。例如，语法分析器允许任意表达式作为LIKE谓词的左操作数，尽管事实上这个操作数必须是一个字段引用。请修改语义分析器使它能够诊断出这些错误的输入。你可以改变语义或者添加动作代码来检查表达式。两种方法你都可以试一下，看看哪一种更简单，而哪一种提供更好的诊断能力。
2. 请把这个语义分析器改成SQL的交叉引用程序，使它可以读入一组SQL语句然后输出一份报告，来表明各个名字的定义位置和引用位置。
3. （学期课题）修改这个嵌入式SQL翻译器，使它可以与你的系统中的实际数据库进行交互。

Flex规范参考

本章我们将描述flex程序的语法以及多种选项和可用的支撑函数。POSIX的lex基本上就是flex的精确子集，所以我们会注意到flex的哪些部分扩展了POSIX的需求。

在讲解完lex程序的结构之后，本章的其余部分将基于特性按字母顺序进行介绍。

Flex结构规范

flex程序由三部分构成：定义部分、规则部分和用户子例程（subroutine）。

```
... 定义部分 ...
%%
... 规则部分 ...
%%
... 用户子例程 ...
```

这三个部分通过由两个百分号组成的行来分隔。前两个部分是必需的，但它们的内容可以为空。第三部分和前面的%%行可以省略。

定义部分

定义部分包含选项、文字块、定义、开始条件和转换。在本参考中（各自都有相应的小节进行解释）以空白字符开始的行将被原样拷贝到C文件中。通常这种情况用于在/*和*/之间的注释，而且前面还需要有空白字符。

规则部分

规则部分包括模式行和C代码。以空白字符开始或者处于%{和%}之间的内容被认为是C代码，它们会被原封不动地拷贝到`yylex()`中。在规则部分开头出现的C代码也会出

现在`yylex()`的开头，它可以包含词法分析器中需要使用的变量的声明，以及每次调用`yylex()`所需要运行的代码。

C代码行将被原样拷贝到生成的C文件中。规则部分开始处的行会被放在生成的`yylex()`函数中邻近开头的地方，它应该是一些变量的声明，以便于模式所关联的代码和词法分析器初始化代码的使用。在其他地方出现的C代码行必须仅仅包含注释，因为你无法预测它们会出现在词法分析器的什么位置（这也是你在规则部分动作代码之外写注释的方法）。

以其他任何字符开始的行就是模式行。模式行包含一个模式、一些空白字符以及输入被该模式匹配时所执行的C代码。如果C代码超过一条语句或者跨过多行，它必须用括号括起（`{ }`或者`%{ %}`）。

当flex词法分析器运行时，它根据规则部分的模式进行匹配。每次它发现一个匹配（匹配的输入称为记号），它执行这个模式所关联的C代码。如果模式后面紧跟一个竖线而不是C代码的话，该模式使用与这个文件中下一模式相同的C代码。当输入字符无法匹配任何模式时，词法分析器将认为它匹配了一个动作代码为`ECHO`的模式，相应地，该记号的拷贝被输出。

用户子例程

用户子例程的内容将被flex原样拷贝到C文件。这个部分通常包括规则中需要调用的例程。如果你重定义了`yywrap()`，该例程的新版本或者相关的支撑例程会在这儿出现。

在大型程序里，把支撑代码放在另外一个单独的源文件中会更加方便，因为这样做的话，每次你修改lex文件之后需要重新编译的内容就减少了。

BEGIN

宏`BEGIN`切换起始状态。你通常在特定模式的动作代码里调用它，如下所示：

```
BEGIN statename;
```

词法分析器从状态0开始，该状态也被称为`INITIAL`。其他所有状态必须在定义部分通过`%s`或者`%x`行来命名（请参见第140页“起始状态”一节）

注意，尽管`BEGIN`是一个宏，但这个宏本身并没有任何参数，而且状态名字也不应该被括号括起，虽然那样做是一种良好的风格。

C++词法分析器

虽然flex有选项来创建C++的词法分析器，但是手册表明这个特性还处在实验阶段，而且相应的代码有很多问题，并不能够很好地工作^(注1)。不过C++程序员也不要觉得失落。你可以生成C的词法分析器，它可以被C++的编译器来编译，而且你也可以让一个C++的bison语法分析器来调用一个C的词法分析器。

上下文相关性

flex提供了多种方式来做到模式的左上下文相关和右上下文相关，也就是与记号的上文和下文相关。

左上下文相关

有三种方法可以做到左上下文相关：特殊的行首模式字符、起始状态以及显式代码。

在模式开头的字符 ^ 可以让lex只在行首匹配该模式。^本身并不匹配任何字符，它只是用来指定上下文。

起始状态可以被用来要求一个记号必须出现在另一个记号之前：

```
%s MYSTATE
%%
first { BEGIN MYSTATE; }
. .
<MYSTATE>second { BEGIN o; }
```

在这个词法分析器中，second记号只在first记号出现之后才被匹配。而且这两个记号之间也允许存在其他插入的记号。

在有些情况下，你可以通过设定标志的方法来伪造左上下文相关性，这个标志可以用来从一个记号传递上下文信息到另外一个单词。

```
%
int flag = 0;
%
a { flag = 1; }
b { flag = 2; }
zzz {
    switch(flag) {
        case 1: a_zzz_token(); break;
        case 2: b_zzz_token(); break;
```

注1： 据编写手册的人所说。

```
    default: plain_zzz_token(); break;
}
flag = 0;
}
```

右上下文相关

同样也有三种方法来使记号的识别依赖于该记号的右边文字：特殊的行尾模式字符、斜线操作符以及`yyless()`。

在模式末尾的字符\$使得该模式仅在行尾得到匹配，也就是说，再往后就是一个\n字符。和字符^一样，\$也不匹配任何字符，它仅仅指定上下文。\$等价于/\n，所以它不能被用于尾部上下文中。

模式中的字符/让你可以包含显式尾部上下文。举例来说，模式abc/de可以匹配记号abc，但仅仅在后面紧跟着de的情况下。/自身不匹配任何字符。lex在决定多个模式中哪个具有最长匹配时会计算尾部上下文字符的个数，但是这些字符不会出现在`yytext`里，也不会被计算在`yylen`中。

函数`yyless()`让lex推回刚刚读到的记号。`yyless()`的参数表明需要保留的字符个数。例如：

```
abcde { yyless(3); }
```

这个例子与abc/de的效果基本一致，因为对`yyless()`的调用保留了该记号的三个字符并且推回了另外两个字符。唯一的差别是在这个例子里，`yytext`包含所有5个字符，而且`yylen`的值是5而不是3。

定义（替换）

定义（或者说替换）允许你对全部或者部分的正则表达式进行命名，然后在规则部分通过名字来引用它们。这可以帮助我们分解复杂的表达式，以及记载你设计该表达式的目的。

定义采用如下格式：

```
NAME expression
```

名字可以包含字母、数字、连字符和下划线，且它不能以数字作为起始字符。

在规则部分，模式可能会包含通过花括号括起的基于名字的替换，例如{NAME}。这个名字所代表的表达式将被代入到模式中，并且该表达式会被认为已经用圆括号括起。例如：

```
DIG [0-9]
...
%%
{DIG}+ process_integer();
{DIG}+\.{DIG}* |
\.{DIG}+ process_real() ;
```

ECHO

在一个模式所关联的C代码中，宏ECHO用来写出记号到当前的输出文件yyout。它等价于如下语句：

```
fprintf(yyout, "%s", yytext);
```

在flex中对于没有获得任何匹配的输入文本，其默认动作就是把该文本写到输出里，也就是ECHO。你可以通过%option nodefault，或者命令行参数-s，或者--nodefault，来使得flex不产生这个默认动作，这通常来说是有用的，因为词法分析器本来就应该匹配所有可能的输入。

输入管理

flex提供了多种多样的方法来管理被分析的源文件。在你的程序的开始部分，你可以把任何打开的标准输入文件赋予yyin，这样词法分析器就会从该文件读取。如果这还不够，flex提供了其他一些不同的方法来更改输入源。

标准输入文件链

你可以通过调用yyrestart(file)来使得词法分析器读取任意标准输入文件。同样，当使用%option yywrap编译出来的词法分析器达到输入文件的末尾时，它会调用yywrap()，我们可以借此来切换到另外一个不同的输入文件。请参见第144页的“yyrestart()”和第144页“yywrap()”小节来获取更多的信息。

输入缓冲区

flex词法分析器可以从一个输入缓冲区中读取输入。输入缓冲区可以被关联到一个输入文件，这种情况下词法分析器也就是从文件读取输入，输入缓冲区也可以被关联到内存中的字符串。类型YY_BUFFER_STATE是指向flex输入缓冲区的指针。

```
YY_BUFFER_STATE bp;
FILE *f;
f = fopen(..., "r");
```

```
bp = yy_create_buffer(f,YY_BUF_SIZE ); 从f创建新的缓冲区  
yy_switch_to_buffer(bp); 使用我们刚刚创建的缓冲区  
...  
yy_flush_buffer(bp); 放弃缓冲区中的内容  
...  
void yy_delete_buffer (bp); 释放缓冲区
```

调用`yy_create_buffer`可以创建一个新的缓冲区来关联到一个打开的标准输入文件。它的第二个参数是缓冲区的大小，它必须是`YY_BUF_SIZE`。

调用`yy_switch_to_buffer`可以使得词法分析器从一个缓冲区中读取输入。你可以随时根据需要来切换缓冲区。当前缓冲区是`YY_CURRENT_BUFFER`。你可以调用`yy_flush_buffer`来放弃缓冲区中的任何内容，这在交互式词法分析器做错误恢复时会有帮助，可以让词法分析器回到一个已知的状态。调用`yy_delete_buffer`可以释放不再需要使用的缓冲区。

从字符串输入

通常flex从文件读取输入，但有些时候你可能希望从其他来源进行读取，比如内存中的字符串。

```
bp = yy_scan_bytes(char *bytes, len); 分析字节流拷贝  
bp = yy_scan_string("string"); 分析以空字符结尾的字符串拷贝  
  
bp = yy_scan_buffer (char *base, yy_size_t size); 分析长度为(size-2) 的字节流
```

例程`yy_scan_bytes`和`yy_scan_string`会创建一个缓冲区来作为被分析文本拷贝所在的位置。处理速度更快的例程是`yy_scan_buffer`，它直接分析文本，但是要求该文本的最后两个字节为空字符（\0），它们不会被分析。类型`yy_size_t`是flex的内部类型，用来代表处理对象的长度。

一旦字符串缓冲区被创建，我们就可以使用`yy_switch_to_buffer`来使得词法分析器从该缓冲区读取，然后我们可以使用`yy_delete_buffer`来释放缓冲区以及（如果需要的话）文本的拷贝。词法分析器把缓冲区的结束作为文件的结束。

文件嵌套

很多输入语言都允许输入文件包含其他文件，比如C语言里面的`#include`。flex提供了一对函数来管理输入缓冲区堆栈：

```
void yypush_buffer_state(bp); 切换到bp, 把旧的缓冲区压入堆栈  
void yypop_buffer_state(); 删除当前缓冲区, 继续使用上一个缓冲区
```

实际上，这些函数仅仅适用于最简单的输入嵌套，因为它们无法维护任何辅助的信息，比如当前文件所在的行和文件名。不过，维护你自己的输入文件堆栈也并不难。请参见例2-3的示例代码。

记号模式`<<EOF>>`对维护输入堆栈也有帮助，它可以匹配调用`yywrap()`之后的文件结尾。

input()

函数`input()`提供输入字符给词法分析器。当词法分析器匹配字符时，它理论上会调用`input()`来获取每个字符。`flex`基于性能的考虑，跳过了`input()`，但是效果是一样的。

最有可能调用`input()`的地方是动作例程，它可能会需要针对特定记号之后的文本做一些特殊处理。例如，下面是处理C语言注释的一种方法：

```
/* { int c1 = 0, c2 = input();  
    for(;;) {  
        if(c2 == EOF)  
            break;  
        if(c1 == '*' && c2 == '/')  
            break;  
        c1 = c2;  
        c2 = input();  
    }  
}
```

对于`input()`的调用会处理所有的字符，直到文件的末尾或者字符*/的出现。这种方法使得我们可以不用借助于独占的起始状态（参见第140页“起始状态”一节）来处理C语言注释。它是处理超长的引号括起的字符串或者其他记号的最佳方法，因为`flex`通常只有16K的输入缓冲。

如果你使用C++编译器的话，`input`将被改为`yyinput`以避免与C++库函数产生名字上的冲突。

YY_INPUT

`flex`词法分析器通过宏`YY_INPUT(buf,result,max_size)`来读取输入到一个缓冲区。一旦词法分析器发现需要更多的输入而缓冲区为空的时候，它将调用`YY_INPUT`，其中的`buf`和`maxsize`就是缓冲区和它的大小，而`result`就是实际读到的字符个数或者代表EOF的零（由于这是一个宏，所以使用`result`，而不是`*result`）。当缓冲区第一次被建立起来的时候，词法分析器将调用`isatty()`来确定输入源是否来自终端，如果是的话，它将每次读取一个字符而不是一大段字符。

如果输入源既不是终端也不是标准文件的话，那么重定义YY_INPUT还是比较有用的。

Flex库

flex自带有一个小型的库，里面有一些辅助例程。在Unix系统里，你可以通过cc命令末尾给定的-lfl标志来链接这个库，在其他系统里也有等效的选项。这个库包含不同版本的main()和yywrap()。

flex自带的main程序可以帮助我们快速编程和测试，而yywrap()主要作为桩函数来使用。它们非常简单，如下所示：

```
main(int ac, char **av)
{
    return yylex();
}
int yywrap() { return 1; }
```

交互模式和批处理模式的词法分析器

flex词法分析器有时需要在输入中向前查看一个字符，这样才能决定当前记号的分析是否已经结束（你可以想象一下一连串数字的情况）。事实证明如果词法分析器总是向前查看，即使并没有这个必要的时候，它的执行效率会更快，但这种做法在输入源是终端的情况下会导致一些不愉快的结果。例如，如果当前的记号是换行符\n，而词法分析器又要向前读，那么词法分析器将不得不一直等待直到你输入了另外一个换行符。

为了减少这种不愉快，词法分析器可以运行在批处理模式中，这时它总是向前查看；或者运行在交互模式中，这时它仅仅在需要的时候才向前查看（可能会慢一些）。如果你使用标准的输入例程，词法分析器会通过isatty()来检查当前输入源是否是终端，如果是，那么切换到交互模式中。你可以使用%option batch或者%option interactive来强制它使用特定模式。强制使用批处理模式只有在你知道这个词法分析器不会用于交互模式时才有意义，例如，如果你自行读取输入或者它总是从文件读取。

行号和yylineno

如果你能够始终维护输入文件的当前行号，你就可以把这个行号用在错误消息里。不过你也可以利用%option yylineno，flex会定义yylineno来包含当前行号，并且它会在每次遇到\n字符时自动地更新行号。词法分析器本身并不初始化yylineno，所以你需要在每次开始读取文件时把它设置为1。在处理嵌套包含文件的时候，如果词法分析器希望跟踪每个文件的行号时，它就必须自己来保存和恢复每个文件的当前行号。

文字块

在定义部分的文字块就是用行%{和行%}括起的C代码。

```
%{
... C代码和声明...
%}
```

每个文字块的内容都会原封不动地被拷贝到生成的C源文件中。定义部分的文字块会被拷贝到`yylex()`的开始部分之前。文字块通常包括规则部分代码所需要使用的变量和函数的声明，也包括引入头文件的`#include`。

如果文字块以`%top{`而不是`%{`开始的话，它将被拷贝到生成程序的头部附近，这种情况主要用于`#include`文件和设置`YY_BUF_SIZE`的`#define`行。

规则部分开始处的文字块将被拷贝到`yylex()`中本地变量声明之后，所以它可以包含更多的声明和设置代码。规则部分其他位置的文字块在`yylex()`中的拷贝位置并不确定，所以它只应该包含注释。

请参见第144页的“`YY_USER_ACTION`”一节。

单一程序中的多重词法分析器

你可能希望在同一个程序中使用两个部分或者完全不同的记号语法。例如，一个交互式调试解释器可能需要一个词法分析器用于编程语言，还需要另外一个词法分析器用于调试命令。

有两种基本的方法来使一个程序处理两个词法分析器：把它们合并成一个词法分析器，或者把两个完整的词法分析器放到程序中。

合并的词法分析器

你可以使用起始状态来合并两个词法分析器。每个词法分析器的模式都被加上特定的起始状态作为前缀。当词法分析器开始工作时，你需要写一小段代码来让它进入合适的初始状态，这样就可以选择特定的词法分析器，例如，下面一段代码（它会被拷贝到`yylex()`的开头）：

```
%s INITA INITB INITC
%%
%{
    extern first_tok, first_lex;
    if(first_lex) {
```

```
    BEGIN first_lex;
    first_lex = 0;
}
if(first_tok) {
    int holdtok = first_tok;
    first_tok = 0;
    return holdtok;
}
%
```

这个例子里，在你调用词法分析器之前，你会设置**first_lex**作为它的初始状态。你通常会把合并的词法分析器与合并的语法分析器结合起来使用，所以你也通常需要通过代码来产生初始记号，以便于告知语法分析器哪种语法正在使用。请参见第168页的“可变语法和多重语法”一节。

这种方法的优点是目标代码比较小，因为只会有一份词法分析器代码的拷贝，而且不同的规则集之间可以共享规则。缺点则是你在每处都需要小心使用正确的起始状态，你不能够一次激活两种词法分析器（也就是说，你不能够递归地调用**yylex()**，除非使用可重入的词法分析器选项），并且它对于不同词法分析器使用不同的输入源的情况的处理会比较麻烦。

多重词法分析器

另外一个方法是在你的程序中包含两个词法分析器。这种技巧要求改变**lex**默认使用的函数和变量名，这样两个词法分析器才可以分别生成，然后编译到同一个程序中。

flex提供了命令行选项和程序选项来改变**lex**生成的词法分析器所使用的名字的前缀。例如，下面这些选项可以让**flex**使用前缀“*foo*”而不是“*yy*”，并且生成的词法分析器源文件会是*foolex.c*。

```
%option prefix="foo"
%option outfile="foolex.c"
```

你也可以通过命令行选项来实现这一点：

```
$ flex --outfile=foolex.c --prefix=foo foo.l
```

任何一种方法生成的词法分析器都具有入口函数**foolex()**，它从标准文件*fooin*读取输入。不过，稍微有点迷惑人的是，**flex**需要在词法分析器的最前面生成一大堆#define宏，它们会把标准的“*yy*”格式的名字重定义为选定的前缀格式。这使得你可以继续使用标准名字来编写你的词法分析器，而外部可见的名字则会使用选定的前缀。

```
#define yy_create_buffer foo_create_buffer
#define yy_delete_buffer foo_delete_buffer
```

```
#define yy_flex_debug foo_flex_debug
#define yy_init_buffer foo_init_buffer
#define yy_flush_buffer foo_flush_buffer
#define yy_load_buffer_state foo_load_buffer_state
#define yy_switch_to_buffer foo_switch_to_buffer
#define yyin fooin
#define yyleng fooleng
#define yylex foolex
#define yylineno foolineno
#define yyout fooot
#define yyrestart foorestart
#define yytext footext
#define yywrap foowrap
#define yalloc foosalloc
#define yrealloc foorealloc
#define yfree foofree
```

编译词法分析器的选项

flex为编译词法分析器提供了几百个选项。大多数可以在词法分析器的开头部分写成如`%option name`的形式或者写成命令行上的`--name`的形式。如果你需要关闭一个选项的话，只需要在该选项前加`no`，比如`%option noyywrap`或者`--noyywrap`。在大多数情况下，把选项放在`%option`行是更推荐的做法，因为如果选项错误的话词法分析器通常无法正常工作。请参见flex自带的帮助文档中的“Index of Scanner Options”一节来获取所有选项的列表。

Flex词法分析器的可移植性

flex词法分析器在C语言的世界里很容易移植。你可以在两个层面移植词法分析器：最初的flex规范或者flex生成的C源文件。

移植生成的C词法分析器

flex生成可移植的C代码，你通常可以把代码用于任何一种C语言编译器而没有任何问题。请确保使用`%option noyywrap`或者使用你自己的`yywrap()`版本来避免引用flex库文件。如果需要移植到非常古老的C语言编译器，你可以使用`%option noansi-definitions`和`%option noansi-prototypes`来让flex分别产生K&R风格的过程定义和原型。

缓冲区大小

你可能希望调整一些缓冲区的大小。flex使用两个输入缓冲区，每个的默认大小是16K，这对于一些微型计算机的实现来说可能显得太大。你可以通过如下的定义方式来定义宏`YY_BUF_SIZE`：

```
%{  
#define YY_BUF_SIZE 4096  
%}
```

如果你的词法分析器使用REJECT，它也会分配一个4倍大小（在64位机器上是8倍）于YY_BUF_SIZE的备份状态缓冲区。如果空间是个问题的话，请不要使用REJECT。

字符集

最纠结的可移植性问题在于字符集。任何版本的flex所生成的C代码都使用字符编码作为词法分析器里面的表的索引。如果源机器和目标机器所使用的字符集一致，例如ASCII，移植的词法分析器可以正常工作。不过你可能需要处理不同的行结束转化：Unix系统使用单一的\n作为行结束，而Microsoft Windows和其他一些系统使用\r\n。通常你可以选择忽略\r而只把\n作为行结束，这样两种机器都可以适用。

当源机器和目标机器使用的字符集不同时，例如ASCII和EBCDIC，词法分析器根本就不能工作，因为所有通过字符编码来实现的索引都错了。有时候资深的用户还有可能基于当前的字符集来重新生成表索引，但是通常来说唯一合理的方法就是寻找在目标机器上可以运行的flex版本，或者重新定义词法分析器的输入例程，把输入字符转化为源字符集。请参见第128页的“从字符串输入”一节来了解如何更改输入例程。

可重入词法分析器

词法分析器的常规代码把它的状态信息放置在静态变量里，这样每次调用yylex()会从上次处理的位置继续开始分析，并且使用现有的输入缓冲区、输入文件、起始状态等等。在有些情况下，允许多份词法分析器的拷贝同时激活会非常有用，这种例子在多线程的程序里比较常见，它们需要处理多种互不干扰的输入源。因此，flex提供了%option reentrant或者--reentrant。

```
yyscan_t scanner;  
  
if(yylex_init(&scanner)) { printf("no scanning today\n"); abort(); }  
while((yylex(scanner))  
    ... do something ...;  
yylex_destroy(scanner);
```

在可重入词法分析器中，所有与当前词法分析相关的信息都被保存在yyscan_t变量中，它实际上是一个指针，指向了包含所有状态的结构。你通过yylex_init()来创建词法分析器，把yyscan_t的地址作为参数传入，而yylex_init()在成功时返回0，或者在它无法分配结构空间时返回1。然后你就可以把yyscan_t代入任何一次yylex()的调用中，最后通过yylex_destroy来删除yyscan_t。每次调用yylex_init都会创建一个独立

的词法分析器，多个词法分析器可以同时生效，但需要确保每次调用`yylex()`时传入相应的`yyscan_t`结构。

在可重入词法分析器中，`yylex`常用的一些变量被重定义为宏，这样你就可以像在普通的词法分析器里面一样使用它们。这些变量包括`yyin`、`yyout`、`yyextra`、`yy leng`、`yytext`、`yylineno`、`yycolumn`和`yy_flex_debug`。宏`BEGIN`、`YY_START`、`YYSTATE`、`ymore()`、`unput()`和`yyless()`也做了修改，所以你也可以像普通词法分析器那样使用它们。所有创建和维护输入缓冲区的例程都添加了一个额外的参数，`yyscan_t`，例如`yyrestart(file, scanner)`。其他接受`yyscan_t`参数的例程包括`yy_switch_to_buffer`、`yy_create_buffer`、`yy_delete_buffer`、`yy_flush_buffer`、`ypush_buffer_state`、`ypop_buffer_state`、`yy_scan_buffer`、`yy_scan_string`和`yy_scan_bytes`。

可重入词法分析器的额外数据

当你使用可重入词法分析器时，你经常会需要一些为每个词法分析器准备的数据，比如词法分析器所匹配的名字的符号表。你可以使用`yylex_init_extra`而不是`yylex_init`，来传递一个指向每个词法分析器特定数据的指针。在`yylex()`中，你的指针将可以通过`yyextra`来获得。这个额外数据的类型是`YY_EXTRA_TYPE`，它通常是`void *`，但你也可以根据需要通过`#define`来重定义它的类型。

```
yyscan_t scanner;
symbol *symp;

symp = symtabinit(); // 创建每个词法分析器的符号表

if(yylex_init_extra(symp, &scanner)) { printf("no scanning today\n"); abort(); }
while((yylex(scanner))
... 做一些操作 ...
yylex_destroy(scanner);

... 在yylex里 ...
[a-z]+ { symlookup(yyextra, yyval); }
```

访问可重入词法分析器的数据

在常规词法分析器中，`yylex()`之外的代码可以直接引用`yyin`、`yyout`和其他一些全局变量，但是在可重入词法分析器中，它们属于各个词法分析器的数据结构。`flex`提供了特别的访问函数来获取和设置这些主要变量。

```
YY_EXTRA_TYPE yyget_extra (yyscan_t yyscanner ); yyextra
void yyset_extra (YY_EXTRA_TYPE user_defined ,yyscan_t yyscanner );

FILE *yyget_in (yyscan_t yyscanner ); yyin
```

```

void yyset_in (FILE * in_str ,yyscan_t yyscanner );

FILE *yyget_out (yyscan_t yyscanner ); yyout
void yyset_out (FILE * out_str ,yyscan_t yyscanner );

int yyget_lineno (yyscan_t yyscanner );
void yyset_lineno (int line_number ,yyscan_t yyscanner );

int yyget_leng(yyscan_t yyscanner); yyleng,只读

char *yyget_text(yyscan_t yyscanner); yytext

```

这些函数也存在于非可重入词法分析器中，只是没有参数`yyscanner`，不过非可重入词法分析器不太需要使用它们，因为它们的功能与直接读取和设置对应的变量没有任何差别。

可重入词法分析器、嵌套文件和多重词法分析器

可重入词法分析器、嵌套文件和多重词法分析器都体现了同一种需求：一个程序具有多种词法分析器，但它们的实现不尽相同。

- 可重入词法分析器允许你同时拥有词法分析器的多个激活实例，它们使用相同的模式来分析不同的输入源。
- 嵌套文件（使用`yy_switch_to_buffer`和相关例程）允许你只用单一的词法分析器来读取一个文件，接着是另一个，然后有可能回到第一个文件。
- 多重词法分析器允许你对不同的输入源应用不同的模式集合。

你可以根据需要合并这三种实现方式。例如，你可以在一个可重入词法分析器中调用`yy_switch_to_buffer`来为一个特定实例改变输入源。你可以通过`%option reentrant prefix="foo"`来创建一个词法分析器，它可以被多次调用（通过调用`foolex_init`来启动），也可以与其他可重入或者非可重入词法分析器一起被链接到同一个程序中。

在Bison中使用可重入词法分析器

`bison`拥有它自己的选项来创建一个可重入的语法分析器，这种语法分析器被称为纯语法分析器（pure-parser），它可以在花费一定代价的情况下与可重入词法分析器一起使用。纯语法分析器通常通过调用`yylex`来获取记号，它会传入一个指向记号值放置位置的指针，但不会包括flex所需的`yyscan_t`参数（flex和bison的开发人员并不是始终保持联系的）。flex提供了`%option reentrant bison-bridge`，它可以把`yylex`的定义改为：

```
int yylex (YYSTYPE * yylval_param ,yyscan_t yyscanner);
```

并且会根据参数值来自动设置`yyval`的值。另外一个与普通词法分析器不同的地方在于`yyval`将变成一个指向联合类型的指针而不是联合类型，所以以前`yyval.member`的使用需要更改为`yyval->member`。请参见第213页的“纯词法分析器和纯语法分析器”一节来了解纯bison语法分析器如何调用一个可重入的词法分析器。

正则表达式语法

flex的模式是编辑器和类似于grep这样的工具所使用的扩展的正则表达式。正则表达式由普通字符组成，这些字符代表了它们自身和在模式中具有特殊意义的元字符，所有下文没有列出的字符属于常规字符。空白字符（空格和tab）被用来分隔模式和动作，所以如果它们在模式中出现的话，必须被引起。

元字符

flex表达式中的元字符如下所示：

匹配除换行符（\n）外的任意单个字符。

[]

匹配方括号中任意一个字符。字符范围通过-（破折号）来表示，例如[0-9]代表10个数字中的任意一个。如果在左方括号之后的第一个字符是破折号或者右方括号的话，它将不被解释为元字符。如果第一个字符是抑扬符号^，意义将改变为匹配除方括号内字符以外的任何字符（这样的字符集将可以匹配换行符，除非你显式地把它排除在外）。除了C语言中以\开头的转义序列（escape sequence）在方括号中也会被识别外，其他元字符就没有什么特殊意义了。POSIX为国际化添加了更多的方括号模式。请参见这个列表的最后几项来获取更多的信息。

[a-z]{-}[jv] [a-f]{+}[0-9]

设置字符类的差异或者联合。匹配的字符是在第一个字符类减去（{-}）或者加上（{+}）第二个字符类的结果里。

*

匹配零个或者多个紧接在前面的表达式。例如，模式`a.*z`匹配任何以`a`开始并以`z`结束的字符串，比如`az`、`abz`或者`alcatraz`。

+

匹配一个或者多个紧接在前面的表达式。例如，`x+`匹配`x`、`xxx`或者`xxxxxx`，但不能是空串，而`(ab)+`匹配`ab`、`abab`、`ababab`等等。

?

匹配零个或者一个紧接在前面的表达式。例如，`-?[0-9]`代表具有可选单目负号的数字。

{}

根据内容的不同会代表不同的概念。单个数字{n}意味着前面的模式重复n次，例如[A-Z]{3}匹配任意三个大写字母。如果花括号包含由逗号分隔的两个数字，{n,m}，则它们代表了前面模式可以重复的最小和最大次数，例如A{1,3}匹配一到三个字母A。如果第二个数字被省略，它将被认为是一个无限数，所以{1,}与+的意义一致，而{0,}与*的意义一致。如果花括号中是一个名字的话，则它指向这个名字所需要做的替换。

\

如果后续字符是小写字母，那么它就是C语言的转义序列，比如\t代表tab。有些实现也允许类似于\123和\x3f这样的八进制和十六进制字符。其他情况下，\引起后续的字符，所以*匹配一个星号。

()

把一系列的正则表达式组合在一起。每个*、+和?仅仅影响最靠近它的左边的表达式，而 | 则通常作用于它左边和右边的所有表达式。圆括号可以改变这一切，例如(ab|cd)?ef匹配abef、cdef或者只是ef。

/

匹配前面的正则表达式或者后续正则表达式。例如，`twelve|12`匹配*twelve*或者*12*。

"..."

匹配引号内任何字面意义上的字符。非\的元字符将失去它们的特殊意义。例如，`/*`匹配两个字符*/。

/

匹配斜线前的正则表达式，但是要求其后紧跟斜线后的表达式。例如，`0/1`匹配字符串01中的0但是不会匹配字符串0或者02。每个模式只允许一个尾部上下文操作符，而且一个模式不能既有斜线又有尾部的\$。

^

当它是正则表达式的第一个字符时，它匹配一行的开始。它也被用于方括号中代表补集。其他情况下，它没有特殊含义。

\$

当它是正则表达式的最后一个字符时，它匹配一行的结束，否则没有特殊意义。它在表达式的末尾时与\n具有相同的意义。

<>

在模式开头出现的用尖括号括起的名字或者名字列表使得该模式仅在给定的起始状态下被应用。

<<EOF>>

特殊模式<<EOF>>匹配文件末尾。

(?# comment)

Perl风格的表达式注释。

(?a:pattern)或者(?a-x:pattern)

Perl风格的模式修饰符。使用修饰符a但不包括修饰符x来解释模式。修饰符包括代表大小写无关的a，把所有字符作为单行匹配的s（这会使.可以匹配\n字符），以及忽略空白字符和C风格注释的x。这个模式可以被扩展为多行。

```
(?i:xyz) [Xx][Yy][Zz]
(?i:x(?-i:y)z) [Xx]y[Zz]
(?s:a.b) a(.|\n)b
(?x:a /* hi */ b) ab
```

flex也允许一部分POSIX字符类出现在字符类的表达式中：

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

这些字符类代表了由ctype宏来处理的名字类型的字符，包括类型alnum、alpha、blank、cntrl、digit、graph、lower、print、punct、space、upper和xdigit。该类名被封装在方括号和冒号中，同时它自己也需要在一个字符类中。例如，[[[:digit:]]等价于[0123456789]，而[x[:xdigit:]]等价于[x0123456789AaBbCcDdEeFf]。

REJECT

通常lex把输入分解为各个不重叠的记号。但有时候即使有些记号彼此重叠，你也希望分析出每个出现的记号。特殊动作REJECT就能够让你实现这一功能。当一个动作执行REJECT时，flex理论上会退回已经匹配模式的文本，然后继续寻找它的下一个最佳匹配。下面的例子展示了如何在一个文件中寻找所有出现的单词，pink、pin和ink，即使它们相互重叠：

```
...
%%
pink { npink++; REJECT; }
ink { nink++; REJECT; }
pin { npin++; REJECT; }
.
\n ; /* 抛弃其他字符 */
```

如果输入包含单词*pink*, 三个模式都会被匹配。如果没有REJECT语句的话, 则只有*pink*会被匹配。

使用REJECT的词法分析器会比那些不使用的分析器在代码量上显得更多, 速度也会变慢, 因为它们需要相当多的额外信息来允许回溯和重新分析。

从yylex()返回值

当模式匹配时执行的C代码可以包含一条返回语句, 它将从yylex()返回相应的值给调用方, 通常该调用方也就是yacc所生成的语法分析器。下一次yylex()被调用时, 词法分析器将从上次离开的点继续分析。当词法分析器匹配到语法分析器感兴趣的记号(例如, 关键字、变量名或者操作符)时, 它将把该记号返回给语法分析器。当对于它匹配的记号语法分析器并不感兴趣的话, 它就不返回, 这样词法分析器就可以立即开始匹配下一个记号。

这意味着你不可能通过仅仅调用yylex()来重启一个词法分析器。你需要使用BEGIN INITIAL来把它重置到默认状态, 并且你还需要重置输入状态, 以便于input()的下一个调用将从新的输入读取。你可以使用yyrestart(file)——这儿的file是一个标准I/O文件指针, 来使得词法分析器从该文件读取。

起始状态

你可以在定义部分定义起始状态, 它也被称为起始条件(*start condition*)或者起始规则(*start rule*)。起始状态被用来限制特定规则的作用范围, 或者针对文件的部分内容来改变词法分析器的工作方式。起始状态有两种模式, 用%s声明的共享模式和用%<x>声明的独占模式。例如, 假定我们要分析如下的C预处理器指令:

```
#include <somefile.h>
```

通常, 尖括号和文件名将被分析为5个记号<、*somefile*、.、*h*和>, 但是如果它们出现在#include之后, 它们将只是单个文件名记号。你可以使用一个起始状态来使得一组规则仅在特定时间被应用。请注意, 那些并不具备起始状态的规则可以应用在任何共享的状

态中！动作中的语句**BEGIN**（参见第124页的“**BEGIN**”一节）设置当前的起始状态。例如：

```
%s INCLMODE
%%
^"#include" { BEGIN INCLMODE; }
<INCLMODE><"[^>\n]+>" { ... 针对名字做一些操作 ... }
<INCLMODE>\n { BEGIN INITIAL; /* 返回正常状态 */ }
```

你可以通过带有%**s**和%**x**的行来声明起始状态。例如：

```
%s PREPROC
```

这行代码创建了一个共享的起始状态**PREPROC**。接着在规则部分，以**<PREPROC>**作为前缀的规则将只在状态**PREPROC**中被应用。**lex**开始时的标准状态是状态零，也被称为**INITIAL**。当前起始状态可以通过**YY_START**（也被称为**YYSTATE**，以便于和老版本的**lex**兼容）获得，这一点很有用，特别是当你在一个特定状态中想回到前一个状态时，例如：

```
%s A B C X
int savevar;
%%
<A,B,C>start { save = YY_START; BEGIN X; }
<X>end { BEGIN save; }
```

flex也可以用%**x**声明独占的起始状态。共享的起始状态和独占的起始状态的差异在于当独占的状态被激活时，没有任何起始状态修饰的规则将不会被匹配。实际上，独占的状态比共享的状态更有用，而你可能会希望两者都用到。

独占的起始状态使得像识别C语言注释这样的事情变得简单：

```
%x COMMENT
%%
"/**" { BEGIN COMMENT; /* 切换到注释模式 */ }
<COMMENT>. |
<COMMENT>\n ; /* 抛弃注释文字 */
<COMMENT>"/" { BEGIN INITIAL; /* 回到正常状态 */ }
```

如果使用常规起始状态（共享的）的话，这段模式将无法工作，因为所有常规的记号模式在**COMMENT**状态中依然有效。

unput()

宏**unput(c)**返回字符c给输入流。与类似的标准输入输出例程**unputc()**不同的是，你可以在一行里多次调用**unput()**来推回多个字符给输入。宏**unput()**所允许推回的字符个数在不同情况下有所差别，但它至少大于词法分析器所能识别的最长记号。

例如，当扩展类似于C语言的#define这样的宏时，你需要在该宏被调用的地方插入这个宏所代表的文本。一种实现方法是调用unput()来推回相应的文本，例如：

```
... 在词法分析器的动作代码中...
char *p = macro_contents ();
char *q = p + strlen(p);

while(q > p)
    unput(--q); /* 从右向左推回 */
```

yyinput() yyunput()

在C++的词法分析器中，宏input()和unput()被更改为yyinput()和yyunput()，以避免和C++库函数名产生冲突。

yyleng

当词法分析器匹配一个记号时，记号的文本被存放在以空字符结束的字符串yytext中，它的长度是在yyleng中。yyleng中的长度与strlen(yytext)的返回值一致。

yyless()

你可以在规则的关联代码中调用yyless(n)来推回记号的前n个字符。这在两个记号之间的边界很难用正则表达式描述的时候会显得很有用。例如，考虑一下用来匹配被引起字符串的模式，这个字符串里也会存在通过反斜线来转义的引号：

```
\\"[^"]*\\" { /* 右边引号之前的字符是否是 \ ? */
    if(yytext[yyleng-2] == '\\') {
        yyless(yyleng-1); /* 返回最后的引号 */
        yymore(); /* 添加下一个字符串 */
    } else {
        ... /* 处理字符串 */
    }
}
```

如果被引起的字符串在结束引号之前带有一个反斜线，这段代码将使用yyless()来推回最后的引号，然后用yymore()来告诉lex把下一个记号添加到这个字符串（参见第143页的“yymore()”一节）。下一个记号将以这个回退的引号作为开始的被引起字符串的剩余部分，这样整个字符串都会在yytext中。

对`yyless()`的调用与调用`unput()`有相同的效果，特定的字符将被推回，但`yyless()`通常会更快，因为它可以利用这样一个事实，就是推回的字符通常也是刚刚从输入中获取的字符。

`yyless()`的另一个用处是使用不同起始状态下的规则来重新处理一个记号：

```
sometoken { BEGIN OTHER_STATE; yyless(0); }
```

`BEGIN`让lex使用另外一个起始状态，然后`yyless()`调用来推回当前匹配记号的所有字符，以便于它们可以在新的起始状态下被重新读入。如果新的起始状态不能够使得一些不同的模式先于这个模式启用的话，`yyless(0)`将导致词法分析器陷入死循环中，因为相同的记号被重复地识别和推回（这种方法与`REJECT`函数类似，但是相比较而言更快）。与`REJECT`不同的是，它将循环匹配相同模式，直到你使用`BEGIN`来更改当前激活的模式。

yylex()和YY_DECL

通常来说，`yylex`的调用并没有参数，它主要通过全局变量与程序的其他部分交互。宏`YY_DECL`定义了它的调用顺序，你可以重定义它来添加你想要的参数。

```
%{  
#define YY_DECL int yylex(int *fruitp)  
%}  
  
%%  
  
apple|orange { (*fruitp)++; }
```

注意在`YY_DECL`里并没有分号，因为它在`yylex()`函数体开头的左花括号之前被展开。

当使用可重入或者bison桥语法分析器时，你依然可以重定义`YY_DECL`，但是你必须要确保包含词法分析器中可重入代码期望的参数。在bison桥语法分析器中，通常的定义如下所示：

```
#define YY_DECL int yylex (YYSTYPE * yylval_param , yyscan_t yyscanner)
```

在简单的可重入词法分析器中可以没有`yylval_param`。

yymore()

你可以在规则的关联代码中调用`yymore()`来使得lex把下一个记号也添加到当前记号中。例如：

```
%%
hyper yymore ();
text printf("Token is %s\n", yytext);
```

如果输入字符串是`hypertext`，它的输出将是“Token is `hypertext`”。

当正则表达式不方便或者不可能定义记号之间的界限时，使用`yymore()`就非常有用。参见第142页的“`yyless()`”一节中的例子。

yyrestart()

你可以调用`yyrestart(f)`来使得你的词法分析器从打开的标准输入输出文件`f`来读取输入。请参见第127页的“标准输入文件链”一节。

yy_scan_string和yy_scan_buffer

这些函数被用来更改词法分析器的输入源到一个字符串或者内存中的缓冲区。请参见第128页的“从字符串输入”一节。

YY_USER_ACTION

这个宏将在每个词法分析器动作的代码前被展开，在`yytext`和`yylen`被设置之后。这经常被用来设定bison记号的位置。参见第8章。

yywrap()

当词法分析器到达文件的末尾时，它可以选择性地调用例程`yywrap()`来了解下一步操作。如果`yywrap()`返回0，词法分析器将继续分析；如果它返回1，词法分析器将返回一个零记号来表明文件结束。如果你的词法分析器不使用`yywrap()`来切换文件，选项`%option noyywrap`可以用来移除对`yywrap()`的调用。特殊记号`<<EOF>>`通常是更好的处理文件结束情况的方式。

flex库文件中的`yywrap()`的标准版本总是返回1，所以如果你希望使用`yywrap()`的话，你需要把它替换为自己的版本。当你在`yywrap()`中返回0来表明还有更多的输入时，你必须在此之前首先调整`yyin`使它指向一个新的文件，通常可以使用`fopen()`来打开它。

Bison规范参考

本章我们将描述bison程序的语法以及各种各样的选项和可用的支撑函数。POSIX的yacc基本上就是bison的精确子集，所以我们会注意到bison的部分功能是超过POSIX需求之外的扩展。

在讲解完bison程序的结构之后，本章的其余部分将基于特性按字母顺序进行介绍。

Bison语法结构

bison程序由三部分构成：定义部分、规则部分和用户子例程。

```
... 定义部分 ...
%%
... 规则部分 ...
%%
... 用户子例程 ...
```

这三个部分通过由两个百分号组成的行来分隔。前两个部分是必需的，但它们的内容可以为空。第三部分以及它前面的`%%`行可以省略。

符号

bison语法由符号组成，它们是语法的“单词”。符号由一连串字母、数字、点号和下划线组成，但是首字符不能是数字。符号`error`代表错误恢复；对于其他符号，bison没有预先的设定。（由于bison为每个记号都定义一个C的预处理器符号，所以你也需要确保记号名字不会跟C语言的保留字以及bison自身的符号如`yyparser`产生冲突，否则会发生一些奇怪的错误。）

由词法分析器产生的符号被称为终结符（*terminal symbol*）或者记号（*token*）。在规则左部定义的符号被称为非终结符（*nonterminal symbol*）。记号也可以是直接用引号引起

的文字字符（参见第155页的“文字记号”一节）。广泛遵循的惯例是记号名字都为大写的而非终结符则是小写的。在本书中我们也遵守这个惯例。

定义部分

定义部分可以包含文字块，它们是被原样拷贝到生成的C文件的开始部分的C代码，文字块存在于%{和%}或者%code中，通常包括声明和#include行。声明包括%union、%start、%token、%type、%left、%right和%nonassoc。（参见第167页的“%union声明”，第164页的“%start声明”，第165页的“记号”、第165页的“%type声明”、第158页的“优先级和结合性声明”这几节。）定义部分可以包含C语言格式的注释，通过/*和*/来表示。所有这些都是可选的，所以在很简单的语法分析器里，定义部分可以完全为空。

规则部分

规则部分包含语法规则和语义动作的C代码。参见下面的“动作”和第161页的“规则”两节来获取更详细的信息。

用户子例程部分

用户子例程的内容将被bison原样拷贝到C文件。这个部分通常包括语义动作中需要调用的例程。

在大型程序里，把支撑代码放在另外一个单独的源文件中会更加方便，因为这样的话，每次你修改bison文件之后需要重新编译的内容就减少了。

动作

动作（action）是bison匹配语法中一条规则时执行的C代码。动作必须是C语言的复合语句，例如：

```
date: month '/' day '/' year { printf("date found"); } ;
```

动作可以通过一个美元符号加上一个数字来使用规则中语法符号所关联的值，冒号后第一个语法符号的数字是1，例如：

```
date: month '/' day '/' year { printf("date %d-%d-%d found", $1, $3, $5); } ;
```

名字\$\$指向左部符号——也就是冒号左边的符号——的值。符号值（语义值）可以有不

同的C类型。参见第165页的“记号”、第167页的“%type声明”和第167页的“%union声明”几节来获取更多的信息。

对于没有语义动作的规则，bison使用如下的默认动作：

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

如果你有个规则没有右部符号，而左部符号又声明了类型，你必须为它编写一个语义动作来赋值。

嵌入动作

尽管bison的分析技术只允许语义动作出现在规则的末尾，bison也可以支持嵌入在规则中间的语义动作。如果你在规则中间写了一个动作，bison将创造一条新的规则，这条规则的右部为空，而左部则是一个伪造的名字，然后把嵌入动作作为该规则的语义动作，并且使用那个伪造的名字来替换原有规则中的嵌入动作。例如，以下两种规则就是等价的：

```
thing:      A { printf("seen an A"); } B ;  
thing:      A fakename B ;  
fakename:  /* 空 */ { printf("seen an A"); } ;
```

虽然这个特性很有用，但它也会造成一些令人惊讶的后果。嵌入动作被转化为规则中的符号，所以规则末尾的语义动作也可以像使用其他符号一样使用它的值（也就是\$\$被赋予的值）：

```
thing:      A { $$ = 17; } B C  
              { printf("%d", $2); }  
;
```

这个例子会打印出“17”。两个动作都可以用\$1来表示A的值，而规则末尾的语义动作则可以使用\$2来表示嵌入动作的值以及用\$3和\$4来表示B和C的值。

嵌入动作在一些本来可以接受的语法中会导致移进/归约或者归约/归约冲突。例如，这个语法不会有任何问题：

```
%%  
thing: abcd | abcZ ;  
  
abcd: 'A' 'B' 'C' 'D' ;  
abcZ: 'A' 'B' 'C' 'Z' ;
```

但是如果你加入一个嵌入动作，它就存在移进/归约冲突：

```
%%
```

```
thing: abcd | abcZ ;  
abcd: 'A' 'B' { somefunc(); } 'C' 'D' ;  
abcZ: 'A' 'B' 'C' 'Z' ;
```

在第一个例子中，语法分析器并不需要在看到所有4个单词之前就来确定它正在分析 `abcd` 还是 `abcZ`。在第二个例子中当它分析到 `B` 时它就需要确定这一点，但是此刻它还没有足够的输入来知道它正在分析哪一条规则。如果嵌入动作在 `C` 之后，就没有问题了，因为 `bison` 可以使用它的向前查看一个记号的方法来知道下一个是 `D` 还是 `Z`。

嵌入动作的符号类型

由于嵌入动作并不关联任何语法符号，所以我们没有任何方法来定义嵌入动作返回值的类型。如果你要使用 `%union` 和类型化的符号值，你必须把值放到尖括号里来使用嵌入动作的值，例如，当你在嵌入动作中设置值时需要使用 `$<type>$`，而你在规则末尾的语义动作中引用它的值时需要使用 `$<type>3`（使用相应的数字）。参见第164页的“符号值”一节。如果你的语法分析器足够简单，全部都使用 `int` 值，就像前面的例子一样，你就不需要为它设定类型。

二义性和冲突

`bison` 在把语法翻译成语法分析器时有可能报告冲突。一些情况下，语法确实有歧义，也就是说，对于一个输入字符串，存在两种可能的语法分析器，而 `bison` 无法处理这种情况。其他情况下，语法本身并没有歧义，但是 `bison` 所使用的标准分析技术还不够强大到分析该语法。无歧义语法存在冲突的原因是语法分析器需要向前查看超过一个记号来决定哪种语法分析器被使用。通常你可以重写这个语法使得向前查看一个记号就足够了，但 `bison` 也提供了一个更强大的技术，GLR，它允许无限制地向前查看。

参见第158页的“优先级和结合性声明”一节和第7章来获取更多如何解决这些问题的细节和建议，参见第234页的“GLR分析”一节来获取GLR分析的细节。

冲突类型

当 `bison` 试图创建语法分析器时存在两种可能的冲突：移进/归约和归约/归约。

移进/归约冲突

移进/归约 (*shift/reduce*) 冲突是指一个输入字符串存在两种可能的语法分析器，并且其中一个分析器结束一条规则（选择归约），而另外一个并不结束（选择移进）。例如，下面这个语法就有移进/归约冲突：

```
%%
e:      'X'
|  e '+' e
;
```

对于输入字符串X+X+X，有两种可能的语法分析器：(X+X)+X或者X+(X+X)。选择归约将使语法分析器使用第一条规则，而选择移进则使语法分析器使用第二条规则。除非用户使用操作符的优先级声明，否则bison选择移进。参见第158页的“优先级和结合性声明”一节来获取更多细节。

归约/归约冲突

归约/归约 (*reduce/reduce*) 发生在同一个记号可以结束两条不同规则的时候。例如：

```
%%
prog: proga | progb ;
proga:      'X' ;
progb:      'X' ;
```

X可以是proga也可以是progb。大多数归约/归约冲突比这个的歧义要小，但它们通常意味着语法存在错误。参见第8章来了解更多解决冲突的细节。

%expect

有时你的语法可能仅有少量的冲突，你确信bison会按照你的想法来解决这些冲突，而且选择重写语法来消除冲突会引起太多的争执（If/then/else形式的冲突是最常见的例子）。你可以借助于定义%expect N，它会告诉bison你的语法分析器会有N个移进/归约冲突，这样bison在发现这个数字不同时会报告一个编译期错误。

你也可以使用%expect-rr N来告诉bison预期会有多少个归约/归约冲突。但是除非你使用GLR语法分析器，否则不要使用这个特性，因为归约/归约冲突总是意味着语法存在错误。

GLR语法分析器

有时候一个语法对于bison常见的LALR分析算法来说就是很难处理。在这种情况下，你可以通过包含%glr-parser声明来让bison创建通用LR (Generalized LR, GLR) 语法分析器。当GLR语法分析器遇到冲突时，理论上来说它会分裂出并行的两种可能的分析，每种分析会消耗其对应的记号。如果有更多的冲突，它可以创建一棵部分语法分析的树，在每次冲突时进一步分裂。在分析结束时，要么仅存活一种分析，其他分析由于不能匹配剩余的输入而被放弃；要么在语法确实有歧义的情况下存活多种分析，而这时就需要你来决定如何处理它们。

参见第234页的“GLR分析”一节来获得更多的例子和细节。

Bison程序的问题

bison本身很健壮，但确实有一些常见的编程错误，它们会导致你的语法分析器产生严重的问题。

无限递归

bison语法中一个常见错误是创建了一个递归规则，但是并没有任何方法来结束这个递归。bison对这类语法的诊断结果便是有些神秘的“起始符号xlist没有产生任何语句”。

```
%%
xlist:      xlist 'X' ;
```

互换优先级

人们有时试图使用%prec来交换两个记号的优先级：

```
%token NUMBER
%left PLUS
%left MUL
%%
expr   :   expr PLUS expr %prec MUL
        |   expr MUL expr %prec PLUS
        |   NUMBER
;
```

这个例子看起来使得PLUS的优先级高于MUL，但事实上它使两者变得一致。优先级机制通过比较移进的记号与规则的优先级来解决移进/归约冲突。这个例子存在若干个冲突。比较典型的冲突发生在语法分析器看到“expr PLUS expr”而下一个记号是MUL的时候。如果没有%prec，这个规则将使用PLUS的优先级，而它比MUL的优先级要低，bison将选择移进。但是由于%prec，规则和记号都具有MUL的优先级，所以它将选择归约，因为MUL是左结合的。

一种可能的解决方法是引入伪记号，例如XPLUS和XMUL，让它们自己的优先级去使用%prec。更好的方法是重写这个语法来表达它的真正意图，这个例子中需要交换%left行（参见第158页的“优先级和结合性声明”一节）。

嵌入动作

当你在规则中间而不是末尾来添加语义动作时，bison将创建一个匿名规则来触发该嵌入

动作。有时匿名规则会导致没有预料到的移进/归约冲突。参见第146页的“动作”一节来获取更多的信息。

C++语法分析器

bison可以产生C++语法分析器。如果你的bison文件是comp.yxx，它将会产生对应的C++源文件comp.tab.cxx和头文件comp.tab.hxx（你可以通过-o标志来更改输出文件名，比如-o comp.c++，它也会把头文件改为comp.h++）。bison还会创建头文件stack.hh、location.hh和position.hh，它们定义了语法分析器中需要使用的三个类。这三个文件的内容总是一致的，除非你使用-p或者%name -prefix来把这个语法分析器的命名空间从“yy”改为其他一些名字。

bison定义了一个叫做yy::parser（除非你更改了名字）的类，它有一个主要的parse例程和一些用于错误报告与调试目的的例程。参见第239页的“C++语法分析器”一节来获取更多的信息。

%code块

bison在定义部分总是可以接受形如%{...%}的C代码。有时这些代码必须放置在所生成程序中的特定位置，通常是标准分析器框架代码特定部分的前或者后。指令%code就可以实现这一点。

```
%code [place] {  
    ... 代码放在这里 ...  
}
```

选项`place`，bison手册称之为限定符（*qualifier*），表明代码在生成程序中的放置位置。目前C语言程序的位置包括`top`、`provides`和`requires`。对应的位置也就是文件的顶部、在YYSTYPE与YYLTYPE的定义之前和定义之后。这个特性还处在实验期，所以这些位置也可能发生变化，因此请参考当前的bison手册来了解当前这些选项的具体内容。不带有`place`选项的%code依然保留，而且人们打算用它来替代%{和%}。

结束标记

每个bison语法都包含一个被称为结束标记（*end marker*）的伪记号，它标记输入的结束。在bison列表中，这个结束标记通常用\$end来表示。

词法分析器必须返回一个零记号来表明输入的结束。

错误记号和错误恢复

bison语法分析器总是会尽可能早地检测语法错误，也就是说，一旦它们发现一个记号没有合适的分析时，它们就会报错。当bison检测到语法错误时，或者说，当它无法分析接收到的输入时，它将尝试基于下面的步骤来从错误中恢复：

1. 它调用`yyerror("syntax error")`。这通常报告错误给用户。
2. 它抛弃任何部分分析的规则，直到它回到一个能够移进特殊符号`error`的状态。
3. 它重新开始分析，首先会移进一个`error`。
4. 如果在成功移进三个符号之前另一错误发生，bison不会报告该错误，直接回到步骤2。

参见第8章来获得更多关于错误恢复的信息。你也可以参见第172页的“`yyerror()`”、第175页的“`YYRECOVERING()`”、第173页的“`yyclearin`”和第174页的“`yyerrok`”这几节来获取关于帮助控制错误恢复的一些特性的信息。

%destructor

当bison试图从分析错误中恢复时，它从分析器堆栈抛弃符号和符号值。如果该值是指向动态分配内存的指针或者在抛弃时需要特殊处理，你可以借助于`%destructor`在特定符号或者具有特定类型值的符号被删除时来获取控制权。它也可以用来处理在成功分析后的起始符号的值。参见第8章来获取更多的信息。

继承属性 (\$0)

bison的符号值可以作为继承属性 (*inherited attribute*) 或者综合属性 (*synthesized attribute*) 来使用（当bison提到“值”时，通常指编译器语境下的“属性”）。常见的继承属性是记号值，它们是语法分析树的叶节点。在理论上每当一条规则被归约，信息就会在分析树中向上移动，而动作根据规则右部的符号值来综合获得结果符号 (\$\$) 值。

有时候你会希望通过不同的方式来传递信息，比如从语法分析树的根节点到叶节点。考虑一下这个例子：

```
declaration: class type namelist ;  
class:      GLOBAL { $$ = 1; }  
          | LOCAL { $$ = 2; }  
          ;
```

```

type:      REAL { $$ = 1; }
|   INTEGER { $$ = 2; }
;

namelist:  NAME { mksymbol($0, $-1, $1); }
|   namelist NAME { mksymbol($0, $-1, $2); }
;

```

考虑到错误检查和输入符号表的需要，如果能够在namelist的动作中就可以获得类和类型会比较有用。bison通过允许访问它内部堆栈中的符号来实现这一可能，这些在当前规则左部符号之前的值分别表示为\$0、\$-1等等。本例中，`mksymbol()`调用中的\$0指向type的值，它在堆栈中出现在namelist产生式的符号之前，它根据type是REAL还是INTEGER会有值1或者2。\$-1指向class的值，它将基于class是GLOBAL还是LOCAL而具有值1或者2。

虽然继承属性有它的用处，但它也可能导致一些难以发现的错误。使用继承属性的动作需要考虑该规则在语法中出现的每个地方。这个例子中，如果你更改语法使得在其他地方也使用了namelist，那么你就必须确保namelist出现的地方，相应的符号也在它之前出现，这样\$0和\$-1才可能得到正确的值：

```
declaration:      STRING namelist ; /* won't work! */
```

继承属性有时候非常有用，特别是那些复杂的语法，比如C语言的变量声明。但通常来说更安全和更简单的方法是为那些需要从继承属性里获取的值使用全局变量。在前例中，规则namelist可以创建一个被声明名字的引用链表，然后返回指向该链表的指针作为规则的值。规则declaration的语义动作可以获得类、类型和namelist的值，并且把类和类型赋予namelist中相关的名字。

继承属性的符号类型

当你使用一个继承属性的值时，常见的值声明技术（例如%type）并不管用。因为针对该值的符号并不出现在规则中，bison无法判定什么是正确的类型。你必须在动作代码中使用显式类型来提供类型名字。前例中，如果class和type的类型分别是cval和tval，最后两行可能需要如下编写：

```

namelist:  NAME    { mksymbol($<tval>0, $<cval>-1, $1); }
|  namelist NAME  { mksymbol($<tval>0, $<cval>-1, $2); }
;
```

参见第164页的“符号值”一节来获取更多的信息。

%initial-action

如果你在语法分析器启动的时候需要初始化一些内容，你可以使用%initial-action

{some-code}来让bison拷贝some-code到yparse的开始部分。具体位置将是标准初始化代码之后，所以你无法把变量声明也放在该代码中（它们会被允许，但是你的动作代码无法访问它们）。如果你需要定义自己的分析时间变量，你必须使用静态全局变量，或者通过%parse-param把它们作为参数传递。

词法反馈

有时，语法分析器可以反馈一些信息给词法分析器以便于处理一些困难的情况。例如，考虑下面这个输入语法：

```
message (any characters)
```

在这个特定上下文中，圆括号里的内容都被看作引起的字符串。你无法每次看到左圆括号都决定开始分析字符串，因为左圆括号在该语法的其他地方可能有不同的解释。

处理这种情况的简单做法是从语法分析器反馈上下文信息给词法分析器，例如，当特定规则是上下文相关时就在语法分析器中设置一个标志：

```
/* 语法分析器 */
%{
int parenstring = 0;
}%
. .
%%
statement: MESSAGE { parenstring = 1; } '(' STRING ')';
```

然后在词法分析器中判定该标记：

```
%{
extern int parenstring;
}
%s PSTRING
%%
. .
"message"  return MESSAGE;
 "(" {      if(parenstring)
            BEGIN PSTRING;
            return '(';
        }
    <PSTRING>[^)* {
            yyval.svalue = strdup(yytext); /* 传递字符串给语法分析器 */
            BEGIN INITIAL;
            return STRING;
        }
}
```

这段代码并不是没有瑕疵的，因为如果其他一些规则也以MESSAGE作为起始符号，bison可能必须向前查看一个记号，这种情况下内嵌的动作需要等到左圆括号被分析后才能够

执行。在大多数实际的例子中，这不会是个问题，因为语法会更简单。如果语法分析器执行错误恢复，相关的代码需要重置parenstring。

这个例子中，你也可以通过在词法分析器里设置parenstring来处理这个特别的情况，例如：

```
"message(" { parenstring = 1; return MESSAGE; }
```

但它也会导致问题，如果记号MESSAGE在该语法的其他地方被使用而且并不总是伴随着圆括号括起的字符串的话。你通常需要选择完全在词法分析器中实现词法反馈还是部分地在语法分析器中实现，最佳的方案依赖于语法的复杂度。如果语法很简单而且记号不会出现在多个上下文中，你可以在词法分析器里处理完必要的转换，而如果语法足够复杂，在语法分析器里识别这种特殊情况会更加容易。

这种方法可以被发挥到极致。我通过yacc, bison的前身（但不是lex，因为记号化的Fortran太古怪了），编写了完整的Fortran77语法分析器，而这个语法分析器需要反馈十几个特别的上下文状态给词法分析器。这确实是一团糟，但是它比起用C来实现全部的语法分析器和词法分析器要简单的多。

文字块

定义部分中的文字块通过行%{和%}括起。

```
%{  
... C代码与声明 ...  
%}
```

文字块的内容会被原封不动地拷贝到生成的C源文件中靠近开头的位置，在yyparse()开头之前。文字块通常包括规则部分代码所需要使用的变量和函数的声明，以及用于任何必需的头文件的#include行。

bison也提供了实验性质的%code POS { ... }，这里的POS指定了该代码在生成文件中的位置。参见第151页的“%code块”一节来获取当前详细信息。

文字记号

bison把单引号引起的字符也作为一个记号看待。例如：

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

左圆括号和右圆括号都是文字记号（literal token）。文字记号的记号编号也就是它们在本地字符集（通常是ASCII）中对应的数值，与引起的字符在C语言中的值也一致。

词法分析器通常从输入中对应的单个字符来产生这些记号，但是如同其他记号一样，输入字符和生成的记号之间的对应关系是完全由词法分析器决定的。一种常见的技术是让词法分析器把所有不能识别的字符作为文字记号看待。例如，在flex词法分析器中：

```
return yytext[0];
```

这包括了语言中的所有单字符操作符，而让bison来捕获那些输入中存在不能识别的字符然后报告错误。

bison也允许你为字符串定义一个别名来方便识别记号，例如：

```
%token NE "!="  
%%  
...  
exp: exp "!=" exp ;
```

它定义了记号NE，使得你可以在语法分析器中任意地使用NE或者!=。词法分析器读到这个单词时，必须依然返回NE的内部记号编号，而不是一个字符串。

位置

为了辅助错误报告，bison提供了位置信息，它可以跟踪语法分析器里每个符号的行与列范围。位置信息可以通过%locations来显式地激活或者在动作代码里隐式地使用位置信息。词法分析器必须在返回记号前记录当前行与列信息，并且为该记号在yyloc中设置位置范围（flex词法分析器自动记录行号，但是你需要自行记录列数）。语法分析器在每次规则被归约时会执行一个默认规则来设置左部符号的位置范围，该范围为第一个右部符号的开始行与列到最后一个右部符号的结束行与列。

在动作代码中，每个符号的位置可以使用@\$来代表左部符号，第一个右部符号使用@1，依次类推。每个位置信息实际上是一个结构，你可以使用类似于@3.first_column这样的表达式来引用该结构的域。

对于多数或者说大多数语法分析器来说，位置信息已经远远超过错误报告的需求。分析错误依然只需要报告分析错误所在的单个记号，所以只有当动作代码报告位置范围的时候，用户才会看到。最有可能使用它们的地方是集成开发环境，它可以通过位置信息来高亮显示源代码以指出具体错误。请参见第8章的一些例子。

%parse-param

通常你调用yyparse()而不需要任何参数。如果语法分析器需要从周边程序导入一些信息，它可以使用全局变量，或者你也可以为其定义添加参数：

```
%parse-param {char *modulename}  
%parse-param {int intensity}
```

这允许你调用`yyparse("mymodule", 42)`，然后在语法分析器的动作代码中使用`modulename`和`intensity`。注意这里并没有使用分号和结束标点，因为参数就被放置在`yyparse`定义中的两个圆括号之间。

通常的语法分析器很少有机会使用分析参数，但是如果你需要产生一个可能被递归地或者在多线程中调用多次的纯语法分析器，参数定义是为每个语法分析器的实例提供参数的最简单的方法。

Bison语法分析器的可移植性

你可以在两个层面移植一个语法分析器：原始bison语法或者生成的C源文件。

移植bison语法

大多数情况下，你可以编写bison语法分析器然后依赖任何人的bison版本去处理它，因为bison的核心特性在过去15年里的变化很少。如果你确定你的语法分析器需要最近几年添加的特性，你可以要求能够编译该语法分析器的bison的最低版本：

```
%require "2.4"
```

移植生成的C语法分析器

通常bison语法分析器在现代的C和C++实现（C89或者更后，或者ANSI/ISO C++）里很容易移植。

库文件

在bison库文件中的例程通常是`main()`和`yyerror()`。任何较大的语法分析器都会有它自己的上述两个例程的版本，所以库文件通常并不需要。

字符编码

在两个使用不同字符编码的机器之间移植生成的语法分析器时需要特别注意。尤其是，你需要避免类似于'`o`'这样的文字记号，因为语法分析器使用字符编码来作为内部表中的索引，所以在一台ASCII（'`o`'的编码是48）机器上生成的语法分析器无法在一台EBCDIC机器上正常工作（'`o`'的编码是240）。

bison将自己的数值赋值给符号记号，所以如果一个语法分析器只使用符号记号的话，它

可以被成功地移植。

优先级和结合性声明

通常，所有bison语法都必须没有歧义。也就是说，只存在一种方式来根据语法中的规则分析任意合法的输入。

有时，有歧义的语法更容易使用。但二义性语法会导致冲突，这样就存在两种可能的分析，导致bison对一个记号可以选择两种不同的处理方法。当bison处理二义性语法时，它使用默认规则来决定如何分析具有二义性的序列。但这种规则通常并不会产生我们期望的结果，所以bison使用操作符声明来使得你可以改变它处理移进/归约冲突的方式（参见第148页的“二义性和冲突”一节）。

大多数编程语言拥有复杂的规则来控制算术表达式的解释方式。例如，如下的C表达式：

```
a = b = c + d / e / f
```

将被认为是如下的表达式：

```
a = (b = (c + ((d / e) / f)))
```

决定操作符和操作数关系的规则被称为优先级 (*precedence*) 和结合性 (*associativity*)。

优先级

优先级为每个操作符赋予一个优先“级别”。高优先级的操作符的绑定更紧密，例如，如果 $*$ 比 $+$ 有更高优先级的话， $A+B*C$ 将被认为是 $A+(B*C)$ ，而 $D*E+F$ 则是 $(D*E)+F$ 。

结合性

结合性决定语法在使用相同操作符或者具有相同优先级的操作符时如何对表达式分组。它们可以从左、从右分组，或者根本不分组。如果 $-$ 是左结合，表达式 $A-B-C$ 将意味着 $(A-B)-C$ ，而如果它是右结合的话，那就是 $A-(B-C)$ 。

有些操作符，比如Fortran的`.GE.`，没有任何结合性，也就是说， $A .GE. B .GE. C$ 不是一个合法的表达式。

优先级声明

优先级声明出现在定义部分。可能的声明包括`%left`、`%right`和`%nonassoc`。`%left`和`%right`声明分别使得一个操作符左结合或者右结合。你可以使用`%nonassoc`来声明没有结合性的操作符。

操作符按照优先级的升序被声明。在同一行的所有操作符具有相同的优先级。例如，Fortran语法可能包括如下声明：

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

优先级最低的操作符是`+`和`-`，优先级中等的操作符是`*`和`/`，优先级最高的是`POW`，它表示`**`幂操作符。

使用优先级和结合性解决冲突

语法中的每个记号都可以通过优先级声明来获得相应的优先级和结合性。每条规则也可以有各自的优先级和结合性，它可以通过该规则中的`%prec`子句来声明，如果没有`%prec`子句的话，该规则的优先级由最右记号决定。

当存在移进/归约冲突时，bison会比较可能移进的记号和可能归约的规则的优先级。如果记号的优先级更高，那么就移进；如果规则的优先级更高，那么就归约。如果两者具有相同的优先级，bison将检查结合性。如果它们是左结合，那么就归约；如果它们是右结合，那么就移进；如果它们没有结合性，那么bison就报告错误。

优先级的典型使用

尽管在理论上你可以使用优先级来解决任何的移进/归约冲突，但你必须仅仅在充分了解的情况下使用优先级，否则最好重写语法。优先级声明被设计用来处理表达式语法，带有大量的规则如下所示：

```
expr OP expr
```

表达式语法几乎总是使用优先级来编写。

另一种优先级的常见用法是`if/then/else`，你可以通过优先级更容易地解决“dangling else”问题而不需要重写语法。

参见第7章来获取更多的信息。也可以参见第150页的“Bison程序的问题”来了解使用`%prec`的一个常见问题。

递归规则

为了分析不定长的项目列表，你需要使用递归规则，也就是用自身来定义自己。例如，下面这个例子分析一个可能为空的数字列表：

```
numberlist: /* 空 */
    | numberlist NUMBER
    ;
```

递归规则的实现完全依赖于具体需要分析的语法。下面这个例子分析一个通过逗号分隔的不为空的表达式列表，其中的expr在语法的其他地方已经被定义：

```
exprlist:     expr
    |     exprlist ',' expr
    ;
```

也可能存在交互的递归规则，它们彼此引用对方：

```
exp:         term
    |     term '+' term
    ;
term:        '(' exp ')'
    |     VARIABLE
    ;
```

任何递归规则或者交互递归规则组里的每个规则都必须至少有一条非递归的分支（不指向自身）；否则，将没有任何途径来终止它所匹配的字符串，这是一个错误。

左递归和右递归

当你编写一个递归规则时，你可以把递归的引用放在规则右部的左端或者右端，例如：

```
exprlist: exprlist ',' expr ; /* left recursion */
exprlist: expr ',' exprlist ; /* right recursion */
```

大多数情况下，你可以选择任意一种方式来编写语法。bison处理左递归要比处理右递归更有效率。这是因为它的内部堆栈需要追踪到目前为止所有还处在分析中的规则的全部符号。如果你使用右递归版本的exprlist，而且有个表达式包含了10个子表达式，当读取第10个表达式的时候，堆栈中会有20个元素：10个表达式各自有个expr和逗号。当列表结束时，所有嵌套的exprlist都需要按照从右向左的顺序来归约。而另一方面，如果你使用左递归的版本，exprlist将在每个expr之后进行归约，这样在内部堆栈中列表将永远不会有超过三个元素。

具有10个元素的表达式列表不会对语法分析器造成什么问题，但是我们的语法经常需要

分析拥有成千上万个元素的列表，尤其是当程序被定义为语句的列表时：

```
%start program
%%
program: statementlist ;

statementlist : statement
    | statementlist ';' statement
    ;
statement: . . .
```

这个例子中，假定有一个5 000条语句的程序需要分析，那么列表中将有包括语句和分号的10 000个元素，而右递归版本中的10 000个元素的列表对绝大多数语法分析器来说都显得太大了。

当你确定列表中的元素个数很少而且你需要把它们存到一个链表中时，右递归语法就比较有用：

```
thinglist: THING { $$ = $1; }
    | THING thinglist { $1->next = $2; $$ = $1; }
    ;
```

如果你对这个例子使用左递归语法的话，你就必须在最后把链表中倒序的内容做一次翻转，或者你需要在每次添加时找到链表的末端。

你可以定义YYINITDEPTH来控制语法分析器堆栈的长度，它表明堆栈的初始大小，通常为200，也可以定义YYMAXDEPTH来设置堆栈长度的最大值，通常为1 000。例如：

```
%{
#define YYMAXDEPTH 50000
%}
```

每个堆栈元素的大小是语义值的大小（%union元素的最大长度）加上2个字节的记号编号，如果你使用位置信息，还要加上16个字节。在具有千兆字节虚拟内存的工作站上，100 000个元素的堆栈大致需要2到3兆字节，但在更小的嵌入式系统中，你可能会希望重写你的语法来减少堆栈大小。

规则

bison语法由一组规则组成。每个规则由一个非终结符开始，然后是冒号和可能为空的符号、文字记号和动作的列表。为了方便起见，规则用分号结束，虽然分号在技术上也是可以省略的。例如：

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

该例表明date由month、斜线、day、斜线和year组成（符号month、day和year必须在语法的其他地方加以定义）。起始符号和冒号被称为规则的左部（left-hand side, LHS），而规则剩余的部分则是右部（right-hand side, RHS）。右部可以为空。

如果语法中连续几个规则拥有相同的左部，第二个和后续的规则可以用竖线而不是名字和冒号开始。下面两段是等价的：

```
declaration: EXTERNAL name ;
declaration: ARRAY name '(' size ')';

declaration: EXTERNAL name
| ARRAY name '(' size ')';
```

使用竖线是更好的风格。在竖线之前的分号必须被省略。具有相同左部的多个规则并不用放在一起。可以参见第4章的SQL语法，那里面有多条贯穿整个语法的规则定义了术语sql。

动作由C语言的复合语句组成，它在语法分析器到达语法中动作出现的点时就会执行。

```
date: month '/' day '/' year
      { printf("Date recognized.\n"); }
;
```

动作中的C代码可以有一些以\$或者@起始的特别格式，bison会做特殊处理（参见第146页的“动作”和第156页的“位置”两节来获取更多信息）。出现在规则中间而不是末尾的动作将被特殊处理（参见第147页的“嵌入动作”来获取更多信息）。

规则可以在末尾显式声明优先级：

```
expr: expr '*' expr
| expr '-' expr
| '-' expr %prec UMINUS ;
```

优先级仅用来处理那些有二义性的语法。参见第158页的“优先级和结合性声明”一节来获取更多信息。在一个GLR语法分析器里，规则可以使用%prec优先级声明来解决二义性分析。

特殊字符

由于bison处理符号记号而不是字面文本，它的输入字符集比词法分析器要来得简单。下面是bison所使用的特殊字符列表：

%

具有两个百分号的行用来分隔bison语法的各个部分（参见第145页的“Bison语法结构”一节）。所有定义部分的声明都以%开始，包括%{ %}、%start、%token、%type、%left、%right、%nonassoc和%union。参见第155页的“文字块”、第164页的“%start声明”、第167页的“%type声明”、第158页的“优先级和结合性声明”和第167页的“%union声明”这几节。

\$

在语义动作中，美元符号引入一个值引用，例如，\$3代表规则右部第三个符号的值。参见第164页的“符号值”一节。

@

在语义动作中，@符号引入一个位置引用，比如@2代表规则右部第二个符号的位置。

,

文字记号用单引号引起，例如'Z'。参见第155页的“文字记号”一节。

"

bison允许你把双引号引起的字符串定义为记号的别名。参见第155页的“文字记号”一节。

<>

在语义动作中的值引用里，你可以通过括在尖括号里的类型名来覆盖值的默认类型，例如\$<xtype>3。参见第164页的“声明符号类型”一节。

{}

语义动作中的C代码使用花括号括起（参见第144页的“动作”一节）。在文字块声明部分中的C代码通过%{和%}括起。参见第155页的“文字块”一节。

;

规则部分的每个规则都必须使用分号结尾，后面又紧跟以竖线开始的另一规则的规则可以除外。分号是可以省略的，但是使用它总是好的。参见第161页的“规则”一节。

/

当两个连续的规则具有相同的左部时，第二个规则可以把左部的符号和冒号替换为竖线。参见第161页的“规则”一节。

:

在每条规则中，冒号出现在规则左部的符号之后。参见第161页的“规则”一节。

-

符号可以包含下划线、字母、数字和点号。

符号可以包含点号、字母、数字和下划线。这有可能会引起问题，因为C标识符不允许点号。特别是，不允许使用名字中包含点号的记号，因为记号名字都会通过`#define`来作为C预处理器符号。

%start声明

通常起始规则，也就是语法分析器首先开始分析的规则，是第一个出现的规则。如果你希望以其他规则作为起始规则，你可以在声明部分中如下编写：

```
%start somename
```

来以规则`somename`作为起始规则。

在大多数情况下，最清楚的表达语法的方式是自上而下，起始规则放在第一个，这样`%start`就不需要了。

符号值

bison语法分析器中的每个符号，包括记号和非终结符，都可以有关联的值。如果记号是NUMBER，那么它的值也就是特定的数值；如果它是STRING，它的值可能是指向字符串拷贝的指针；而如果它是SYMBOL，它的值可能是指向符号表中描述该符号的条目的指针。每个不同类型的值都对应了不同的C类型：表示数值的int或者double，表示字符串的char *，以及指向表示符号的结构的指针。bison可以方便地为不同的符号分配不同的类型，所以它能够自动为每个符号选择正确的类型。

声明符号类型

在内部，bison通过C语言的联合类型来声明符号值，使得它可以包含所有类型。你在`%union`声明中列出所有可能的类型。bison会把它们转化为联合类型的`typedef`，该类型被称为YYSTYPE。对于每个在动作代码中需要被使用或者设置值的符号，你必须声明它的类型。你可以使用`%type`来声明非终结符。你还可以使用`%token`、`%left`、`%right`、`%nonassoc`来声明记号，以便于可以在联合类型的域定义中使用记号的名字和对应的类型。

接着，当你通过\$\$、\$1等等来使用符号值时，bison将自动使用联合类型中的恰当域。

bison并不分析任何的C代码，所以任何你所犯下的符号拼写上的错误，例如使用了一个并不在联合类型中的类型名字或者使用了一个C语言不允许的域，都将导致生成的C程序

的错误。

显式符号类型

bison允许你在使用符号值的地方声明该符号的显式类型，这主要是通过用尖括号把类型名括起，然后放在美元符号和符号编号之间或者是两个美元符号之间达成，例如\$<xxx>3或者\$<zzz>\$。

这个特性很少被使用，因为对于几乎所有的情况而言，声明符号类型会更简单易懂。最有可能使用显式符号类型的地方是在我们使用继承属性或者设置和使用嵌入动作返回的值的时候。参见第152页的“继承属性（\$0）”和第146页的“动作”两节来获取更多的信息。

记号

记号，或者说终结符，是词法分析器传递给语法分析器的符号。当bison语法分析器需要新的记号时，它调用yylex()，这将从输入中返回下一个记号。在输入结束时，yylex()返回零。

记号可以是通过%token定义的符号或者是单引号中的各个字符（参见第155页的“文字记号”一节）。所有被用来作为记号的符号必须在定义部分显式声明，例如：

```
%token UP DOWN LEFT RIGHT
```

记号也可以通过%left、%right、%nonassoc来声明，它们都具有和%token一样的语法选项。参见第158页的“优先级和结合性声明”。

记号编号

在词法分析器和语法分析器中，记号通过小型整数来唯一标识。一个文字记号的记号编号就是它在本地字符集（通常是ASCII）中的数值，而且也与被引起字符的C语言的值一致。

符号记号通常由bison来负责编号，该编号要大于任何可能的字符编码，所以它们不会和文字记号发生冲突。你也可以在%token的记号名字后直接加上要赋予的编号：

```
%token UP 50 DOWN 60 LEFT 17 RIGHT 25
```

给两个记号赋予相同的编号是错误的行为。在大多数情况下，让bison来选择每个记号的编号最简单易懂。

词法分析器需要知道记号编号，以便于能够返回合适的数值给语法分析器。对于文字记号而言，它使用对应的C字符常量。对于符号记号，你可以通过-d命令行标志来让bison创建一个C的头文件，里面包含所有记号编号的定义。如果你在你的词法分析器中#include这个头文件，你就可以直接在C代码中使用符号记号，例如，UP、DOWN、LEFT和RIGHT。如果你的源文件是xxx.y的话，你的头文件通常会生成为xxx.tab.h，你也可以通过%defines声明或者--defines=filename命令行选项来更改它。

```
%defines "xxxsyms.h"
```

记号值

bison中的每个符号都可以有关联值（参见第164页的“符号值”一节）。由于记号可以有值，你需要在词法分析器返回记号给语法分析器时来设置值。记号值总是保存在变量yylval中。在最简单的语法分析器里，yylval就是简单的int变量，你可以在flex词法分析器中做如下设置：

```
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
```

不过，在大多数情况下，不同的符号会有不同的值类型。参见第167页的“%union声明”和“%type声明”以及第167页的“符号值”这几节。

在语法分析器中，你必须定义所有拥有值的记号的值类型。你只需要把相应的联合类型中的标记名字用尖括号括起，然后放到%token或者优先级声明中。你可以如下定义你的值类型：

```
%union {
    enum optype opval;
    double dval;
}

%nonassoc <opval> RELOP
%token <dval> REAL

%union { char *sval; }
...
%token <sval> STRING
```

在这个例子中，RELOP是一个关系操作符，比如==或者>，而记号值表明具体的操作符。

当你返回记号时，你需要设置yylval中相应的域。本例中，你可以在词法分析器中如下进行设置：

```
%
#include "parser.tab.h"
%
```

```
.[0-9]+.[0-9]* { yyval.dval = atof(yytext); return REAL; }
\"[^"]*\"
{ yyval.sval = strdup(yytext); return STRING; }
"=="           { yyval.opval = OPEQUAL; return RELOP; }
```

REAL的值是一个double类型，所以它被放在yyval.dval中，而STRING的值是char *类型，所以它被放在yyval.sval中。

%type声明

你可以使用%type来声明非终结符的类型。该定义具有如下格式：

```
%type <type> name, name, . . .
```

每个type的名字必须以%union定义过（参见下面的%union声明一节）。而每个name就是非终结符的名字。参见第164页的“声明符号类型”一节来获取更多的信息和例子。

%type被用来声明非终结符。对于记号而言，你需要使用%token、%left、%right和%nonassoc。参见第165页的“记号”和第158页的“优先级和结合性声明”两节。

%union声明

%union声明标识出了符号值可能拥有的所有C类型（参见第164页的“符号值”一节）。声明格式如下：

```
%union {
    ... 域声明 ...
}
```

域声明将被原封不动地拷贝到输出文件中类型为YYSTYPE的C的union声明里。bison并不检查%union中的内容是否是有效的C代码。如果你有多个%union声明，它们的内容将会合并成一个C或者C++的联合声明。

如果不存在%union声明，bison将把YYSTYPE定义为int，这样所有的符号值都为整型。

你可以通过%type来把%union中声明的类型与特定的符号关联起来。

bison把生成的C的联合声明放在生成的C文件和可选的头文件（除非你另行指定，否则被命名为name.tab.h）中，所以你可以在其他源文件中通过包含该头文件来使用YYSTYPE。反过来说，你也可以把你自己声明的YYSTYPE放在一个头文件中，然后在定义部分通过#include来引用它。在这种情况下，你必须至少有一个%type或者其他符号类型声明来告知bison你在使用显式声明的符号类型。

可变语法和多重语法

你可能希望你的语法分析器可以处理两个部分或者完全不同的语法。例如，一个交互式调试解释器需要有一个语法分析器来处理编程语言，还需要另一个语法分析器来处理调试命令。一个只处理一次的（one-pass）C编译器需要一个语法分析器来处理预处理器语法，还需要另一个语法分析器来处理C语言本身。

有两种方法可以处理一个程序中的两种语法：把它们合并为单一的语法分析器，或者在程序中放入两个完整的语法分析器。

合并的语法分析器

如果你有几个相似的语法，你可以通过添加一个新的起始规则来把它们合并为一个语法，新的起始规则需要依赖于第一个读到的记号。例如：

```
%token CSTART PPSTART
%%
combined:  CSTART cgrammar
          |  PPSTART ppgrammar
          ;
cgrammar: ...
ppgrammar: ...
```

这个例子中，如果第一个记号是CSTART，它将分析起始规则为cgrammar的语法；如果第一个记号是PPSTART，那么它将分析起始规则为ppgrammar的语法。

你需要在词法分析器中放入相应的代码，使得在语法分析器第一次询问一个记号时，词法分析器可以返回合适的特殊记号：

```
%%
%{
extern first_tok;

if(first_tok) {
    int holdtok = first_tok;

    first_tok = 0;
    return holdtok;
}
%}
... <词法分析器的剩余部分>
```

这个例子中，你在调用`yyparse()`前需要为`first_tok`设置恰当的记号。

这种方法的一个优点是使用该方法的程序比处理多个语法分析器的程序要小，因为只需

要一份语法分析器代码。另外一个优点是如果你分析有关联的语法，例如C预处理器表达式和C语言本身，你可以共享部分语法。这个方法的缺点则是你无法在一个语法分析器激活的情况下调用另外一个语法分析器，除非你创建一个纯语法分析器，对于所有非共享的语法使用完全不同的语法符号，但这样你很有可能由于在两个语法中使用了相同的符号而导致难以查找的错误。

事实上，这种方法在你分析同一语言的具有微小差异的多个版本时十分有用，例如一个可以被编译的完整的语言和在调试器里可以被解释执行的语言的交互子集。如果一个语言实际上是另一语言的子集，较好的方法是为它们使用一个语法分析器，然后在非子集规则里的语义动作中判断哪个版本正在被分析，如果被分析的是子集的话，可以报告错误给用户。

多重语法分析器

另一种方法是在一个程序中包含两个完整的语法分析器。每个bison语法分析器都具有相同的人口函数`yyparse()`，并且调用相同的词法分析器`yylex()`，而词法分析器又使用相同的记号值变量`yyval`。其次，分析表和语法分析器堆栈都使用类似于`yyact`和`yyv`的全局变量。如果你仅仅只是转化两种语法然后把生成的语法分析器文件直接编译和链接，你将得到很长一串符号多重定义的错误。实现多重语法分析器的技巧便在于重新命名bison所使用的函数名和变量名。

使用%name-prefix或者-p标志

你可以在bison源代码中使用特定的声明来更改bison生成的语法分析器所使用的名字前缀。

```
%name-prefix "pdq"
```

这将产生一个具有入口函数为`pdqparse()`的语法分析器，而它将调用`pdqlex()`，诸如此类。

具体来说，被影响的名字包括`yyparse()`、`yylex()`、`yyerror()`、`yyval`、`yychar`和`yydebug`（变量`yychar`保留最近读到的记号，它在打印错误消息时很有用）。其他语法分析器使用的变量也会被重命名或者变成`static`和`auto`类型，在任何情况下，它们都被保证不会存在冲突。另外我们也可以通过命令行上的`-p`标志而不是源文件中的定义来实现这一点，还有一个`-b`标志可以指定生成的C文件的前缀，例如，

```
bison -d -p pdq -b pref mygram.y
```

将产生pref.tab.c和pref.tab.h的语法分析器，该语法分析器的入口函数是pdqparse。你同时也需要提供yyerror()和yylex()的恰当命名的版本。

多重语法分析器的词法分析器

如果你为你的多重语法分析器使用flex词法分析器，你也需要调整该词法分析器来适应语法分析器的改变（参见第131页的“单一程序中的多重词法分析器”一节）。你通常会希望为合并的语法分析器使用合并的词法分析器，而为多重语法分析器使用多重词法分析器。

纯语法分析器

另外还存在一个些许不同的问题，那就是递归分析，也就是在原有的yyparse()调用正在进行的时候再次调用yyparse()。当你使用合并的语法分析器时这会是一个问题。如果你使用一个合并的C语言和C预处理器的语法分析器，你会在C语言模式中只调用一次yyparse()来分析整个程序，但遇到#if时你会需要递归地调用yyparse()来分析预处理器表达式。

纯语法分析器在多线程程序中也非常有用，多线程程序的每个线程可以同时分析不同来源的输入。参见第9章中第213页的“纯词法分析器和纯语法分析器”一节来获取更多纯语法分析器的信息。

y.output文件

bison可以创建日志文件，以前通常命名为y.output，现在更多的是命名为name.output，它展示了语法分析器的所有状态和状态的变迁。使用标志--report=all可以生成相应的日志文件。

下面是第1章中bison程序的部分日志：

```
state 3
    10 term: NUMBER .
$default reduce using rule 10 (term)

state 4
    11 term: ABS . term

    NUMBER shift, and go to state 3
    ABS shift, and go to state 4
```

```
term go to state 9  
  
state 5  
  
2 calclist: calclist calc . EOL  
  
EOL shift, and go to state 10
```

每个状态中的点号表明了当语法分析器进入该状态时的规则分析情况。例如，当语法分析器在状态4时，如果语法分析器看到一个NUMBER记号，它将把NUMBER移进堆栈然后切换到状态3。如果它看到ABS，它将移进该记号并切换到状态4，对于其他记号则报告错误。如果随后的归约回到该状态并且堆栈顶部是term，它将切换到状态9。在状态3，它总是使用规则10进行归约（规则将基于它们在输入文件中的出现顺序来编号）。在归约之后，语法分析器堆栈中的NUMBER将被替换为term，而语法分析器回到状态4，这时term会使它变到状态9。

当存在冲突时，冲突的状态会显示有冲突的移进和归约动作。

```
State 19 conflicts: 3 shift/reduce  
  
state 19  
  
5 exp: exp . ADD exp  
5 | exp ADD exp .  
6 | exp . SUB factor  
7 | exp . ABS factor  
  
ADD shift, and go to state 12  
SUB shift, and go to state 13  
ABS shift, and go to state 14  
  
ADD [reduce using rule 5 (exp)]  
SUB [reduce using rule 5 (exp)]  
ABS [reduce using rule 5 (exp)]  
$default reduce using rule 5 (exp)
```

这个例子中，当bison看到一个加号时会存在移进/归约冲突。你可以重写该语法或者为加号添加一个操作符声明来解决冲突。参见第158页的“优先级和结合性声明”。

Bison库文件

bison从它的前身yacc继承了带有辅助例程的库文件。你可以通过在cc命令行末尾的-ly标志来包含库文件。这个库文件包括例程main()和yyerror()。

main()

bison库文件有一个最小的主程序，适合于快速编程和测试。它非常简单，我们可以直接写出它来：

```
main(ac, av)
{
    yyparse();
    return 0;
}
```

和其他库例程一样，你也可以提供自己的**main()**。几乎在所有有用的应用程序中你都会希望提供自己的**main()**来接受命令行参数、打开文件和检查错误。

yyerror()

bison也提供一个简单的错误报告例程。它也足够简单到我们可以把它完整地列出：

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

有时候这就已经足够，但是较好的错误例程可以至少报告行号和最近的记号（如果你的词法分析器是用lex实现的话就存在于yytext中），这将使得你的语法分析器更有用。

YYABORT

语义动作中的特殊语句

```
YYABORT;
```

使得语法分析器的分析例程yyparse()立即返回一个非零值，表明存在错误。

这在语义动作检测到一个非常严重的错误以至于无法继续分析时非常有用。

由于语法分析器可能会向前查看一个记号，包含YYABORT的规则动作将会等到语法分析器已经读到另一个记号时才会归约。

YYACCEPT

语义动作中的特殊语句

```
YYACCEPT
```

使得语法分析器的分析例程`yyparse()`立即返回一个零值，表明成功。

这在词法分析器无法判定输入结束而语法分析器可以的时候会比较有用。

由于语法分析器可能会向前查看一个记号，包含`YYACCEPT`的规则动作将会等到语法分析器已经读到另一个记号时才会归约。

YYBACKUP

宏`YYBACKUP`使得你可以移出当前记号并把它替换为另外一个记号。该语法如下：

```
sym: TOKEN { YYBACKUP(newtok, newval); }
```

它将放弃已经被归约的符号`sym`，并且假装词法分析器刚刚读到记号`newtok`，其值为`newval`。如果此时存在一个向前查看的记号或者该规则在右部的符号超过一个的话，这个规则将通过调用`yyerror()`来宣告失败。

正确使用`YYBACKUP()`非常困难，所以你最好就是不用它（这里记载它是因为你有可能会碰到一个使用它的现存的语法）。

yyclearin

语义动作中的宏`yyclearin`可以放弃一个被预先读到的记号。它在交互式语法分析器的错误恢复中非常有用，可以帮助语法分析器在错误发生后进入一个已知状态。

```
stmtlist: stmt | stmtlist stmt ;  
stmt: error { reset_input(); yyclearin; } ;
```

这个例子在错误发生后调用用户例程`reset_input()`，该例程用来把输入返回到已知状态，接着使用`yyclearin`来准备重新读取记号。

参见第175页的“YYRECOVERING()”和第174页的“yyerrok”两节来获取更多的信息。

yydebug和YYDEBUG

bison允许编译跟踪代码（trace code）来报告语法分析器所做的每件事情。这些报告异常冗长，但是它在很多情况下也是读懂复杂的语法分析器的唯一方法。

YYDEBUG

由于跟踪代码又大又慢，它并不会自动编译进目标程序。如果你需要包含跟踪代码的

话，你可以在bison的命令行上使用-t标志，或者把C预处理器符号YYDEBUG定义为非零值，这一点你可以通过C编译器的命令行或者在定义部分添加如下的内容来实现：

```
%{  
#define YYDEBUG 1  
%}
```

yydebug

在运行时的语法分析器中，整型变量yydebug控制语法分析器是否确实产生调试输出。如果它是非零值，语法分析器将产生调试报告，而零值则不会产生。你可以通过任何方法来设置yydebug为非零值，比如，针对程序的命令行参数的响应或者在运行时通过调试器来实现。

yyerrok

在bison检测到语法错误后，它通常会抑制其他错误的报告，直到在没有错误的情况下移进了三个连续的记号。这在一定程度上减轻了语法分析器再次同步过程中单一错误导致多重错误消息的问题。

如果你知道语法分析器何时能够同步，你就可以回到正常状态来继续报告所有错误。宏yyerrok用来让语法分析器回到正常状态。

例如，假定你有一个命令解释器，其中每个命令都在不同的行里。无论用户输入的命令有多么糟糕，你知道下一行总会是一条新的命令。

```
cmdlist: cmd | cmdlist cmd ;  
cmd: error '\n' { yyerrok; } ;
```

带有error的规则跳过所有错误发生后的输入，直到遇到换行符，而yyerrok则告知语法分析器错误恢复已经结束。

参见第175页的“YYRECOVERING”和第173页的“yyclearin”两节。

YYERROR

有时你的动作代码可以检测出上下文相关的语法错误而语法分析器本身却不能。如果你的代码检测到语法错误，你可以调用宏YYERROR来产生一个错误，该错误的效果与语法分析器读到一个该语法不支持的记号的错误一致。一旦你调用YYERROR，语法分析器将进入错误恢复模式来寻找一个它可以移进error记号的状态，参见第152页的“错误记号

和错误恢复”一节来获取更多信息。如果你希望产生一个错误消息，你就必须自己调用 `yyerror`。

yyerror()

当**bison**语法分析器检测到语法错误时，它调用`yyerror()`来报告错误给用户，并且会传给该例程一个参数：描述错误的字符串（除非你在你自己的代码中调用了`yyerror()`，否则通常你会得到的唯一错误就是“语法错误”）。**bison**库文件中的`yyerror`默认版本基本上就是把它的输入参数打印到标准输出。这儿是一个有更多信息的版本：

```
yyerror(const char *msg)
{
    printf("%d: %s at '%s'\n", yylineno, msg, yytext);
}
```

我们假定`yylineno`是当前行号（参见第130页的“行号和`yylineno`”一节）。`yytext`是flex存放当前记号的记号缓冲区。

由于无论情况多么糟糕，**bison**总是试图从错误中进行恢复来分析它的所有输入，你可能会希望对`yyerror()`的发生次数进行计数，然后在10次之后退出，因为理论上语法分析器可能已经被前面报告的错误完全迷惑住了。

在你的动作例程检测其他类型的错误时，你可以并且可能应该自行调用`yyerror()`。

yyparse()

bison生成的语法分析器的人口函数就是`yyparse()`。当你的程序调用`yyparse()`，语法分析器将试图分析输入流。如果分析成功，该分析函数返回零值，否则返回非零值。该函数通常不带有任何参数，但请参见第156页的“%parse-param”一节来获取更多信息。

每次你调用`yyparse()`，语法分析器会忘记上次分析可能拥有的任何状态而重新开始分析。这不像lex产生的词法分析器的`yylex()`，它在你每次调用它时都从上次离开的地方继续分析。

参见第172页的“YYACCEPT”和“YYABORT”两节。

YYRECOVERING()

在**bison**检测到语法错误后，它通常会抑制其他错误的报告，直到在没有错误的情况下移

进了三个连续的记号。这在一定程度上减轻了语法分析器再次同步过程中单一错误导致多重错误消息的问题。

如果语法分析器正处在错误恢复模式中，宏YYRECOVERING()返回非零值，否则返回零。有时候检测YYRECOVERING()的值可以帮助动作例程确定是否需要报告发现的错误。

参见第173页的“yyclearin”和第174页的“yyerrok”两节。



二义性和冲突

本章着重于如何在bison语法中寻找和纠正冲突。冲突发生在bison报告移进/归约或者归约/归约错误的时候。bison把所有的错误都罗列在列表文件*name.output*中，我们会在本章讨论该文件，不过基于该文件查找语法中的错误和进行修正对我们来说是一个挑战。在阅读本章之前，你需要理解bison语法分析器的工作方式，该方式在第3章有详细的描述。

指针模型和冲突

为了准确地描述bison语法中的冲突，我们引入一种bison的操作模型。在这个模型中，一个指针将会在每次读到一个记号（token）时在bison语法中移动。开始时，这个指针（这儿使用一个向上的箭头表示，↑）在起始规则的开始位置：

```
%token A B C  
%%  
start: ↑ A B C;
```

当bison语法分析器读取记号时，这个指针进行移动。假定语法分析器读取了A和B：

```
%token A B C  
%%  
start: A B ↑ C;
```

有时候，可能存在超过一个指针，因为你的bison语法中存在多种选择。例如，假定下述语法读取了A和B：

```
%token A B C D E F  
%%  
start:      x  
          |      y;  
x:  A B ↑ C D;  
y:  A B ↑ E F;
```

(对于本章中后续的例子，所有的大写字母都代表记号，这样我们就可以省去`%token`和`%%`。) 有两种方法让指针消失。一种发生在后续记号无法匹配当前已经部分匹配的规则时。如果下一个语法分析器读到的记号是C的话，第二个指针将消失，而第一个指针可以向前移动：

```
start:      x
          |
          y;
x:  A B C ↑ D;
y:  A B E F;
```

另外一种让指针消失的方法是把它们合并成一个公共的子规则。下面这个例子中，z在x和y中都有出现：

```
start:      x
          |
          y;
x:  A B z R;
y:  A B z S;
z:  C D
```

在读到一个A后，存在两个指针：

```
start:      x
          |
          y;
x:  A ↑ B z R;
y:  A ↑ B z S;
z:  C D
```

在A B C后，将只有一个指针存在于规则z中：

```
start:      x
          |
          y;
x:  A B z R;
y:  A B z S;
z:  C ↑ D;
```

而在A B C D后，这个语法分析器结束规则z，再次出现两个指针：

```
start:      x
          |
          y;
x:  A B z ↑ R;
y:  A B z ↑ S;
z:  C D;
```

当指针达到规则结束的位置时，该规则将被归约。在语法分析器读到D后，指针到达规则z的末尾，这样规则z就被归约。接着指针会回到归约的规则被调用的原规则中，或者像前面那个例子一样，指针会分裂到调用该归约的规则的多个规则中。

在有多个指针的情况下归约一个规则会存在冲突。下面是仅有一个指针时的归约例子：

```
start:      x
          |
          y;
x:  A ↑ ;
y:  B ;
```

在读到A后，只存在一个指针——在规则x中，而规则x将被归约。同样地，在读到B后，只有一个指针——在规则y中，而规则y将被归约。

下面是冲突的例子：

```
start:      x
          |
          y;
x:  A ↑ ;
y:  A ↑ ;
```

在读到A后，存在两个指针，分别在规则x和规则y的末尾。它们都希望被归约，所以这是一个归约/归约冲突。

如果仅存在一个指针将不会有冲突，即使它是把多个指针合并为一个共同子规则的结果，即使归约将导致更多的指针：

```
start:      x
          |
          y;
x:  z R ;
y:  z S ;
z:  A B ↑ ;
```

在A B之后，只存在一个指针，在规则z的末尾，而该规则被归约后，产生两个指针：

```
start:      x
          |
          y;
x:  z ↑ R;
y:  z ↑ S;
z:  A B;
```

但在归约时，只有一个指针，所以不存在冲突。

冲突类型

冲突有两种类型，归约/归约冲突和移进/归约冲突。冲突的分类基于一个指针归约时另外一个指针的情况。如果另外一个指针也在归约，那么就是归约/归约冲突。下面这个例子在规则x和规则y中存在一个归约/归约冲突：

```
start:      x
          |
          y;
x:  A ↑ ;
y:  A ↑ ;
```

如果另外一个指针没有归约而正在移进，那么该冲突就是移进/归约冲突。下面这个例子在规则x和规则y中存在一个移进/归约冲突：

```
start:      x
          |
          y R;
x:  A ↑ R;
y:  A ↑ ;
```

在语法分析器读到A后，规则y需要归约规则start，这样R就可以被接纳，而规则x也能够接受R。

如果在归约时有超过两个指针，bison将列出所有的冲突。下面这个例子在规则x和规则y里有个归约/归约冲突，并且在规则x和规则z中有另一个归约/归约冲突：

```
start:      x
          |
          y
          |
          z;
x:  A ↑ ;
y:  A ↑ ;
z:  A ↑ ;
```

让我们根据预读的记号和即将消失的指针来确切地定义归约发生的时刻，这样我们就可以保证简单的冲突定义是准确的。

下面是一个归约/归约冲突：

```
start:      x B
          |
          y B;
x:  A ↑ ;
y:  A ↑ ;
```

但下面这个例子就不存在冲突：

```
start:      x B
          |
          y C;
x:  A ↑ ;
y:  A ↑ ;
```

第二个例子没有冲突的原因是在于bison语法分析器可以在A之后预读一个记号。如果它看到的是B，规则y中的指针将在规则x被归约之前消失。相似地，如果它看到的是C，规则x中的指针将在规则y被归约之前消失。

bison语法分析器只能预读一个记号。下面这个例子在能够预读两个记号的语法分析器中将不存在冲突，但是在bison语法分析器中，它将存在归约/归约冲突：

```
start:      x B C
          |
          y B D;
x:  A ↑ ;
```

```
y: A ↑ ;
```

如果不太可能重写语法来解决冲突的话，你可以选择GLR语法分析器来解决这种冲突。参见第9章。

语法分析器状态

bison可以通过*name.output*来告诉你语法存在的冲突，该文件是bison产生的所有状态机的描述。我们将讨论这些具体的状态，描述*name.output*的内容，然后讨论如何基于*name.output*给出的冲突描述来定位你的bison语法的问题。你可以通过执行带有-v(verbose模式) 选项的bison来生成*name.output*。

每个状态都对应着你的bison语法中的可能指针的唯一组合。每个非空的bison语法至少有三个可能的状态：一个在最开始还没有接受任何输入时，另一个则是所有有效的输入都已经被接受时，第三个在记号\$end被接受之后。下面的简单例子还有两个额外的状态：

```
start:      A <一个在这> B <另一个在这> C;
```

对于以后的例子，我们将为状态编号以便于清楚标识每个状态。bison会为每个状态编号，不过具体的编号并没有特殊意义。不同的bison版本可能对状态的编号也会不同。

```
start:      A <状态1> B <状态2> C;
```

当一个给定的输入记号流可以匹配多个可能的指针位置时，这些指针都对应于同一状态：

```
start:          a  
    |          b;  
a:  X <状态1> Y <状态2> Z;  
b:  X <状态1> Y <状态2> Q;
```

当不同的输入流对应的指针相同时，它们对应的状态也相同：

```
start:      threeAs;  
threeAs: /* 空 */  
    | threeAs A <状态1> A <状态2> A <状态3>;
```

前面这个语法接受成倍增长的三个A。状态1对应于1、4、7……个A，状态2对应于2、5、8……个A，而状态3对应于3、6、9……个A。我们把它重写为一个右递归的语法来阐述下面一个要点。

```
start:      threeAs;  
threeAs: /* 空 */  
    | A A A threeAs;
```

规则中的位置并不需要仅仅对应一个状态。一条规则中给定的指针可以对应另一规则中的不同指针，从而产生多个状态：

```
start:      threeAs X
           |
           twoAs Y;
threeAs: /* 空 */
| A A A threeAs;
twoAs: /* 空 */
| A A twoAs;
```

上述语法接受成倍增长的两个或者三个A，三个A后面伴随的是X，两个A后面伴随的是Y。如果没有X和Y，这个语法将会有冲突，因为不知道6个A的倍数情况是符合threeAs还是twoAs。如果我们使用左递归的话也会有冲突，因为它需要在看到最后的X或者Y之前先归约twoAs和threeAs。如果我们把状态做如下编号：

```
state 1: 1, 7, ... A's accepted
state 2: 2, 8, ... A's accepted
...
state 6: 6, 12, ... A's accepted
```

那么相应的指针位置如下：

```
start:      threeAs X
           |
           twoAs Y;
threeAs: /* 空 */
| A <1,4> A <2,5> A <3,6> threeAs;
twoAs: /* 空 */
| A <1,3,5> A <2,4,6> twoAs;
```

也就是说，在threeAs中的第一个A之后，语法分析器可能已经接受了 $6i+1$ 或者 $6i+4$ 个A，其中*i*为0, 1, 依次类推。同样地，在twoAs中的第一个A之后，语法分析器可能已经接受了 $6i+1$ 、 $6i+3$ 或者 $6i+5$ 个A。

name.output的内容

现在我们已经定义好了状态，可以看一下*name.output*所描述的冲突。该文件的格式在不同版本的**bison**中会有变化，但它总是会包括一个语法中所有规则的列表和所有的语法分析器状态。它通常在开头部分会有一个冲突和错误的总结，以及没有被使用的规则，通常是由冲突的原因。对于每个状态，它列出该状态对应的规则和位置，在该状态下语法分析器针对不同记号所进行的移进和归约操作，以及该状态归约产生一个非终结符后迁移到什么状态。我们会展示一些二义性语法和*name.output*是如何标识这些语法的二义性的。**bison**所产生的文件把箭头表示为一个点号，这里我们依然会使用向上的箭头（↑），这样可以更容易阅读，并且和前面的例子保持一致。

归约/归约冲突

考虑下面的二义性语法：

```
start:      a Y
          |
          b Y ;
a:   X ;
b:   X ;
```

当我们通过bison来运行它时，常见的状态描述如下：

```
state 3

1 start: a ↑ Y

Y shift, and go to state 6
```

在这个状态中，语法分析器已经归约了一个a。如果它看到Y，它会移进Y然后跳到状态6。其他所有的情况都将是错误。这个语法的二义性会在状态1上产生一个归约/归约冲突：

```
state 1

3 a: X ↑
4 b: X ↑

Y reduce using rule 3 (a)
Y [reduce using rule 4 (b)]
$default reduce using rule 3 (a)
```

第4和第5行显示了在读到记号Y时规则3和规则4之间的冲突。在这个状态中，它读到一个X，可能为a，也可能为b。这表明有两个规则可以被归约。点号指明了在接受下一个记号之前你在规则中所处的位置。这对应于bison语法中的指针。对于归约冲突，指针总是处在规则的末尾。在冲突发生时，没有被选择的规则显示在方括号中。在这个例子中，bison选择归约规则3，因为它通过归约更早出现在语法中的规则来解决归约/归约冲突。

规则中可能会有记号或者非终结符。看一下下面这个二义性语法：

```
start:      a Z
          |
          b Z;
a:   X y;
b:   X y;
y:   Y;
```

产生的语法分析器存在如下状态：

```
state 6
```

```

3 a: X y .
4 b: X y .

Z reduce using rule 3 (a)
Z [reduce using rule 4 (b)]
$default reduce using rule 3 (a)

```

在这个状态中，语法分析器已经把一个Y归约为y，但y可能属于a，也可能属于b。非终结符的变化与记号一样也可能导致归约/归约冲突。如果你使用全大写的记号名字，就像现在这样，你会很容易区分它们。

冲突的规则并不需要是相同的规则。看下面这个语法：

```

start:      A B x Z
           |
           y Z;
x:        C;
y:        A B C;

```

当该语法被bison处理时，会包含如下状态：

```

state 7

3 x: C .
4 y: A B C .

Z      reduce using rule 3 (x)
Z      [reduce using rule 4 (y)]
$default reduce using rule 3 (x)

```

在状态7中，语法分析器已经接受A B C。规则x中只有C，因为在调用x的规则start中，A B在到达x前就已经被接受。C可以结束x或者y。bison将继续选择归约在语法中更早出现的规则来解决冲突，这个例子中就是规则3。

移进/归约冲突

确定移进/归约冲突相对来说就困难一些。为了确定移进/归约冲突，我们将做如下事情：

- 寻找在*name.output*中的移进/归约错误。
- 确定归约规则。
- 确定相关的移进归约。
- 理解归约规则在归约后的状态。
- 归约会产生冲突的记号流。

下面这个语法包含一个移进/归约冲突：

```
start:      x
          |
          y R;
x:   A R;
y:   A;
```

bison会产生如下的警告：

```
state 1

3 x: A . R
4 y: A .

R shift, and go to state 5

R [reduce using rule 4 (y)]
```

状态1存在一个移进/归约冲突，如果移进记号R，它将会跳转到状态5，它也可以在读到R之后应用规则4来归约。规则4就是规则y，如下所示：

```
4 y: A .
```

你可以使用在归约/归约冲突中寻找冲突规则的相同方法来寻找移进/归约冲突中的归约规则。归约编号会在reduce using行中列出。在前例中，有移进冲突的规则是状态中唯一剩下的规则：

```
3 x: A . R
```

当语法分析器读完A，准备接受R时，它处在规则x中。这个例子的移进冲突规则很容易被找到，因为它是当时仅有的规则，而且它表明下一个记号是R。bison总是选择移进来解决移进/归约冲突，所以在这个例子中，如果它读到一个R，它将移进到状态5。

下面这个例子会更多地讨论规则而不是记号：

```
start:      x1
          |
          x2
          |
          y R;
x1:  A R;
x2:  A z;
y:   A;
z:   R;
```

bison将报告多个冲突，其中一个如下所示：

```
state 1

4 x1: A ↑ R
5 x2: A ↑ z
6 y: A ↑
```

```
R shift, and go to state 6  
R [reduce using rule 6 (y)]  
z go to state 7
```

在前例中，归约规则如下：

```
6 y: A ↑
```

这就产生了移进冲突：

```
4 x1: A ↑ R  
5 x2: A ↑ z
```

规则x1使用下一个记号R，所以你可以知道R属于移进冲突的组成部分，但是规则x2的下一个符号并不是记号。这样你就必须查看规则z来了解它是否可能以R作为开始。这个例子中规则z可以以R开始，所以对于A后紧跟R的这种输入存在冲突：它可能是x1，或者是包含z的x2，又或者就是y后跟随一个R。

在冲突状态中可能存在着更多的规则，而它们并不一定都可以接受R。考虑一下这个扩展的语法：

```
start:      x1  
|          x2  
|          x3  
|          y R;  
x1:  A R;  
x2:  A z1;  
x3:  A z2  
y:   A;  
z1:  R;  
z2:  S;
```

bison将产生如下状态的列表：

```
state 1  
  
5 x1: A ↑ R  
6 x2: A ↑ z1  
7 x3: A ↑ z2  
8 y: A ↑  
  
R shift, and go to state 7  
S shift, and go to state 8  
  
R [reduce using rule 8 (y)]  
  
z1 go to state 9  
z2 go to state 10
```

冲突存在于移进至状态7和归约规则8之间。规则8的归约问题在于规则y。规则x1有一个移进的问题，因为在点号后的下一个记号是R。我们并不能够一下子就确定是x2还是x3导致了冲突，因为这两个规则的点号之后是z1和z2。不过当你查看规则z1和z2时，你就发现z1包含的下一个记号是R而z2包含的则是S，所以使用z1的规则x2也是引起冲突的原因而x3则不是。

在我们最后的两个移进/归约冲突例子里，你是否能够看出一个归约/归约冲突呢？请运行bison，然后查看*name.output*来验证你的答案。

复习*name.output*中的冲突

我们将回顾我们的指针模型、冲突和*name.output*之间的关系。首先是个归约/归约冲突：

```
start:      A B x Z
           |
           y Z;
x:   C;
y:   A B C;
```

bison会列出如下内容：

```
state 7

3 x: C ↑
4 y: A B C ↑

Z reduce using rule 3 (x)
Z [reduce using rule 4 (y)]
$default reduce using rule 3 (x)
```

这里存在一个冲突，因为下一个记号是Z，bison既可以归约规则3，也可以归约规则4，它们分别对应于规则x和规则y。我们也可以使用我们的指针模型来分析这个冲突，这里存在两个指针，而且它们都要进行归约：

```
start:      A B x z
           |
           y Z;
x:   c ↑ ;
y:   A B C ↑ ;
```

下面是一个移进/归约冲突的例子：

```
start:      x
           |
           y R;
x:   A R;
y:   A;
```

bison报告如下冲突：

```

state 1

3 x: A ↑ R
4 y: A ↑

R shift, and go to state 5

R [reduce using rule 4 (y)]

```

如果下一个记号是R的话，bison既想归约规则y，也想在规则x中移进R，这就导致了冲突。或者说这里存在两个指针，其中一个希望归约：

```

start:      x
           |
           y R;
x:   A ↑ R;
y:   A ↑ ;

```

常见的冲突例子

有三种常见的情况会产生移进/归约冲突，它们分别是表达式语法、if/then/else和嵌套项目列表。在我们知道如何判断这三种情况之后，我们会来了解一下如何消除冲突。

表达式语法

我们第一个例子改编于最初的1975年的Unix yacc手册。

```

expr: TERMINAL
    | expr '-' expr ;

```

冲突状态如下：

```

state 5

2 expr: expr ↑ '-' expr
2 | expr '-' expr ↑

'-' shift, and go to state 4

'-' [reduce using rule 2 (expr)]
$default reduce using rule 2 (expr)

```

bison告诉我们当它读到减号时存在一个移进/归约冲突。我们添加一些指针来获得如下的表示：

```

expr: expr ↑ - expr ;
expr: expr - expr ↑ ;

```

它们存在于同一个规则中，甚至不是同一左部的不同规则。确实有可能在某些状态下，

你的指针处在同一规则的不同位置，因为这个语法是递归的。（事实上，本节的所有例子都是递归的。大部分蹊跷的bison问题都与递归规则有关系。）

在接受两个expr和-后，指针处在规则expr的末尾，如前面的指针例子的第二行所示。但是expr - expr也是一个expr，所以指针也可以处在第一个expr之后，如前面的例子的第一行所示。如果下一个记号不是-，那么第一行中的指针将消失，因为它需要下一个也是-，这样你就回到一个指针的状态。但是如果下一个记号是-，那么第二行就希望归约，而第一行希望移进，导致了冲突。

为了解决这个冲突，我们可以看一下前面展示的*name.output*，来寻找冲突的来源。我们可以去掉这个状态中不相关的规则（这个极小的例子中没有），然后你就会看到我们刚刚讨论的两个指针。这就变得很清楚：

```
expr - expr - expr
```

中间的expr可以是expr - expr的第二个expr，这种情况下输入被解释为：

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

这是左结合的情况。或者它也可以是第一个expr，这种情况下输入被解释为：

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

这是右结合的情况。在读到expr - expr之后，语法分析器如果使用左结合性就会归约，使用右结合性就会移进。如果没有任何指令来指定哪一种结合性的话，这种二义性就导致移进/归约冲突，bison将选择移进来解决冲突。图7-1显示了两种可能的分析。

本章稍后部分我们会讨论处理这种冲突的方法。

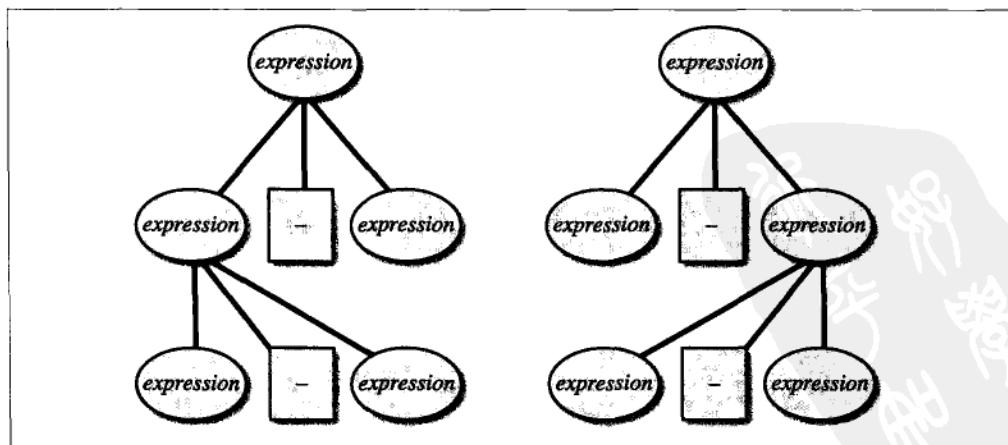


图7-1: `expr - expr - expr`的两种分析

IF/THEN/ELSE

我们的下一个例子也来自于Unix yacc手册。同样，为了完整性我们添加了一个终结符：

```
stmt: IF '(' cond ')' stmt
      | IF '(' cond ')' stmt ELSE stmt
      | TERMINAL;
cond: TERMINAL;
```

bison的警告：

```
State 9 conflicts: 1 shift/reduce
...
state 9

1 stmt: IF '(' cond ')' stmt ↑
2 | IF '(' cond ')' stmt ↑ ELSE stmt

ELSE shift, and go to state 10

ELSE [reduce using rule 1 (stmt)]
$default reduce using rule 1 (stmt)
```

指针的表示如下：

```
stmt: IF ( cond ) stmt ↑ ;
stmt: IF ( cond ) stmt ↑ ELSE stmt ;
```

第一行是冲突的归约部分，而第二行是冲突的移进部分。这次它们是拥有相同左部的不同规则。为了找出错误原因，我们需要来了解第一行归约后的位置所在。它必定在一个调用stmt的规则中，后面紧跟一个ELSE。这样就只有一处地方满足我们的条件：

```
stmt: IF ( cond ) stmt <return to here> ELSE stmt ;
```

在归约之后，指针回到了与冲突的另一方移进动作相同的位置。这个问题与前例中的expr - expr - expr很相似。使用类似的逻辑，为了归约IF (cond) stmt到stmt，从而得到如下的状态：

```
stmt: IF ( cond ) stmt <here> ELSE stmt ;
```

你就必须要有这样的记号流：

```
IF ( cond ) IF ( cond ) stmt ELSE
```

那么，你希望使用如下的组合吗？

```
IF ( cond ) { IF ( cond ) stmt } ELSE stmt
```

还是这样的组合呢？

```
IF ( cond ) { IF ( cond ) stmt ELSE stmt }
```

下面一节会解释如何处理这种冲突。

嵌套列表语法

我们的最后一个例子是bison编程新手经常会遇到的问题的简化版本：

```
start:          outerList Z ;
outerList: /* 空 */
|   outerList outerListItem ;

outerListItem:    innerList ;

innerList: /* 空 */
|   innerList innerListItem ;

innerListItem: I ;
```

bison报告如下冲突：

```
state 2

1 start: outerList ↑ Z
3 outerList: outerList ↑ outerListItem

Z shift, and go to state 4

Z  [reduce using rule 5 (innerList)]
$default reduce using rule 5 (innerList)

outerListItem go to state 5
innerList go to state 6
```

我们可以再次一步步地分析这个问题。归约的规则是`innerList`的空规则。该规则使得移进存在两个选择。规则`start`是其中的一个选择，因为它显式地接受`Z`作为下一个记号。`outerList`的非空规则可能是一个选择，如果它也接受`Z`作为下一个记号的话。我们可以看到`outerList`包含一个`outerListItem`，它又是一个`innerList`。在我们的情况中，`innerList`不会包括`innerListItem`，因为`innerListItem`的下一个记号是`I`，而我们的冲突仅仅发生在下一个记号是`Z`的情况下。不过`innerList`可以为空，这样`outerListItem`就不需要任何记号，所以我们实际上也可能处在`outerList`的末尾，因为冲突报告的第一行告诉我们，`outerList`后可以跟随`Z`。

所有这些都归结为这样一个状态：我们刚刚结束了一个可能为空的`innerList`，也可能刚刚结束的是一个可能为空的`outerList`。我们如何知道哪个列表是刚刚处理结束的呢？

让我们来看一下这两个列表表达式。它们都可以为空，`innerList`存在于`outerList`中，而且不需要任何记号来表明`innerList`循环的开始和结束。假定输入流仅仅只包含一个`Z`。它是一个空的`outerList`，还是一个带有一个子项的`outerList`，又或者是一个空的`innerList`? 这是有歧义的。

这个语法的问题在于它是多余的。它不得不在一个循环中使用另外一个循环，但是又没有任何记号来分隔这两层循环。由于这个语法事实上接受一个可能为空的`I`的列表，其后跟随一个`Z`，它可以很容易被重写为只带有一个递归规则的语法：

```
start:          outerList Z ;
outerList: /* 空 */
|      outerList outerListItem ;
outerListItem:   I ;
```

但是这样重写语法很少是正确的做法。更多的可能是确实需要两个嵌套列表，只是你忘记在`outerListItem`中使用一些标点符号来分隔内层循环和外层循环。我们会在本章的稍后部分提供一些建议。

你如何解决冲突?

本章的剩余部分将描述如何处理你所找到的冲突。我们会讨论如何解决各种各样的经常给bison用户带来问题的冲突。

当我们开始尝试解决冲突时，我们应该首先从陷入最多冲突的规则着手。通常来说，当你解决了主要的冲突，很多次要的冲突也就自动消失了。

如果你正在为一个你所发明的语言编写语法分析器，语法分析器中的冲突通常暗示着语言定义本身的二义性和不一致性。你需要首先找出冲突的起源，然后再来确定语言本身正确与否，如果语言本身没有问题，那么你只需要调整语法来正确地描述语言，否则你就需要同时修改语言和语法，使得它们都没有歧义。如果一个语言的bison分析有问题，那么人们通过头脑来分析也很困难，所以如果你消除了语言的冲突，你通常就会有更好的语言设计。

IF/THEN/ELSE (移进/归约冲突)

我们在本章的前面部分看到过这种冲突。我们这里将要讨论的是一旦你确定了这种移进/归约冲突，你如何解决它。通常来说bison处理这种特定冲突的默认方法也就是你所希望的。但是你如何知道它确实如你所愿呢？你的选择只有如下几种：(1) 深入理解bison的描述；(2) 像受虐狂那样分析`name.output`的所有输出；或者(3) 花费毕生时间来测试生成的代码。一旦你验证了你所得到的代码就是你希望的，你就应该让bison再对这种

冲突报警。冲突警告会让其他试图维护你的代码的人感到困惑和苦恼，而且如果这个语法中还存在着其他表明真正错误的冲突时，你会很难把这种真正的错误从虚假的警报中区分出来。

解决这种冲突的标准方法是为匹配到的和没有匹配到的IF/THEN语句编写独立的规则，并且如下重写整个语法：

```
stmt:      matched
          |
          unmatched
          ;
matched:   other_stmt
          |
          IF expr THEN matched ELSE matched
          ;
unmatched: IF expr THEN stmt
           |
           IF expr THEN matched ELSE unmatched
           ;
other_stmt: /* 其他类型语句的规则 */ ...
```

非终结符other_stmt代表了这个语言中其他所有可能的语句。

我们也可以使用显式的优先级声明来使得bison知道如何解决这种冲突，这样就不会产生警告信息。如果你的语言使用了一个THEN关键字（就像Pascal做的那样），你可以如下实现：

```
%nonassoc THEN
%nonassoc ELSE

%%
stmt:      IF expr THEN stmt
          |
          IF expr stmt ELSE stmt
          ;
```

如果你的语言中并没有THEN关键字，你可以使用一个伪记号和%prec来实现相同的结果：

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%%
stmt:  IF expr stmt      %prec LOWER_THAN_ELSE ;
       |  IF expr stmt ELSE stmt;
```

这里的移进/归约冲突存在于移进ELSE记号和归约stmt规则之间。因此，你需要为这个记号(%nonassoc ELSE)和规则(设定%nonassoc THEN或者%nonassoc LOWER_THAN_ELSE和%prec LOWER_THAN_ELSE)分别赋予一个优先级。移进记号的优先级必须比要归约的规则的优先级要高，所以%nonassoc ELSE必须在%nonassoc THEN或者%nonassoc

`LOWER_THAN_ELSE`之后。对于具体应用而言，使用`%nonassoc`、`%left`或者`%right`并没有什么差别，因为这种冲突中的归约规则并不会包含移进的记号。

这儿的目的是隐藏你所了解的冲突，显示其他任何冲突。如果你正尝试让**bison**对一些移进/归约冲突保持缄默，那么你从前面例子里学到的越多，你就会越小心。当你使用`%nonassoc`时，即使你不小心添加了导致冲突的其他规则，**bison**依然可以报告出来。其他移进/归约冲突请根据**bison**的描述来进行修改。我们前面也说过，任何冲突都可以通过修改语言本身来解决。例如，如果我们要求`stmt`前后始终要有`BEGIN-END`或者成对的花括号，那么`IF/THEN/ELSE`冲突自然就解决了。

如果你交换移进记号和归约规则的优先级会发生什么样的事情？常见的`IF-ELSE`处理可以使得下面两种表示等价：

```
if expr if expr stmt else stmt  
if expr { if expr stmt else stmt }
```

看起来很清楚，交换优先级将会使得下面两种表示等价，对不对呢？

```
if expr if expr stmt else stmt  
if expr { if expr stmt } else stmt
```

答案是否。事实并不如此。让移进（常见的`IF-ELSE`）拥有更高的优先级将使语法分析器总是移进`ELSE`。交换优先级会使得语法分析器永远都不移进`ELSE`，这样你的`IF-ELSE`就无法支持`ELSE`。

常见的`IF-ELSE`处理把`ELSE`与最近的`IF`相关联。如果你希望它按照其他方式来实现，比如，对于一连串的`IF`你只允许一个`ELSE`，而这个`ELSE`与第一个`IF`相关联，那么它将需要如下的两层语句定义：

```
%nonassoc LOWER_THAN_ELSE  
%nonassoc ELSE  
  
%%  
  
stmt: IF expr stmt2 %prec LOWER_THAN_ELSE  
      | IF expr stmt2 ELSE stmt;  
stmt2: IF expr stmt2;
```

但请不要这么做，因为这种语言完全违背了大家的直觉。

嵌套循环（移进/归约冲突）

存在这种冲突的语法有两个嵌套的创建列表的循环，并且没有任何标点符号来表明外层列表的边界。

```
start:          outerList Z ;
outerList: /* 空 */
|       outerList outerListItem ;

outerListItem: innerList ;
innerList: /* 空 */
|       innerList innerListItem ;

innerListItem: I ;
```

假定这就是你所要的语法，那么解决这种冲突的方法取决于你如何定义这些循环，它们是一个外层循环和很多内层循环，还是很多外层循环且每个外层循环只有一个内存循环。这两者的差异在于outerListItem所关联的代码在前一种情况下对每个循环都会执行一遍，而在后一种情况下则是对一组循环才执行一遍。如果这两者对你来说都没有差别，那么你可以随意选择一种实现。如果你希望要很多的外层循环，那么就去掉内层循环：

```
start:          outerList Z ;
outerList: /* 空 */
|       outerList innerListItem ;
innerListItem: I ;
```

如何你希望要很多内层循环，那么就去掉外层循环：

```
start:          innerList Z ;
innerList: /* 空 */
|       innerList innerListItem ;

innerListItem: I ;
```

事实上，很少存在语法有两个嵌套列表而没有使用标点符号的情况。它不仅让bison感到困惑，也会让我们人类感到困惑。如果外层列表类似于编程语言中的语句列表，你可以修改这个语言来使得每个outerListItem后加入一个分号，那么冲突就会消失：

```
start:          outerList Z ;
outerList: /* 空 */
|       outerList outerListItem ';' ;

outerListItem: innerList ;
innerList: /* 空 */
|       innerList innerListItem ;
innerListItem: I ;
```

表达式优先级（移进/归约冲突）

```
expr:      expr '+' expr
```

```
|   expr '-' expr  
|   expr '*' expr  
...  
;
```

如果你描述一种表达式语法而又忘记定义优先级，你将会得到一卡车的移进/归约冲突。为每个操作符赋予优先级可以解决这些冲突。记住如果这些操作符被用于其他目的，例如使用-来表示值的范围，优先级也会掩盖相应上下文中的冲突。

有限的向前查看（移进/归约冲突或者归约/归约冲突）

大多数移进/归约冲突缘自于bison有限的向前查看。也就是说，如果一个语法分析器可以向前看更多的记号时，这种冲突就不会发生。例如：

```
statement: command optional_keyword '(' identifier_list ')'  
;  
optional_keyword: /* 空 */  
| '(' keyword ')'  
;
```

这个例子描述了一个命令行，它以必需的命令开始，以必需的以圆括号括起的标识符列表结束，中间是可选的以圆括号括起的关键字。bison在读到输入流中的第一个圆括号时就会报告一个移进/归约冲突，因为它无法识别这个圆括号是可选关键字的一部分还是标识符列表的一部分。如果是前者，语法分析器将在optional_keyword规则中移进圆括号；如果是后者，它将归约一个空的optional_keyword规则，然后转移到标识符列表上。如果bison语法分析器能够向前查看更多的记号时，它就能够辨认出这两者的差异。但是使用常规bison分析算法的语法分析器无法做到这一点。

bison的默认动作将是选择移进，也就是说它总是认为存在可选的关键字（这种情况下你无法说这个关键字是可选的）。如果你使用优先级的话，你可以让语法分析器选择归约来消除冲突，但这就意味着你不可能拥有这个可选的关键字。

我们可以扁平化这个描述，然后扩展statement中的optional_keyword规则：

```
statement:  command '(' keyword ')' '(' identifier_list ')'  
| command '(' identifier_list ')'  
;
```

通过这种修改，我们使得语法分析器能够向前查看多个可能的记号直到它发现一个关键字或者一个标识符，那时它便可以判定需要使用哪条规则。

扁平化可以适用于这个例子，但是当有更多的规则被加入时，这种方法很快就无法使用，因为bison的描述会呈现指数级的增长。这样你就会陷入因为有限的向前查看而导致的移进/归约冲突，这种情况下可行的解决方法就是改变语言本身或者选择GLR语法分析器。

有限的向前查看也可能导致归约/归约冲突，比如有重叠的规则时：

```
statement: command_type_1 ':' '[' ...
          | command_type_2 ':' '(' ...
          command_type_1: CMD_1 | CMD_2 | CMD_COMMON ;
          command_type_2: CMD_A | CMD_B | CMD_COMMON ;
```

如果输入包含CMD_COMMON，那么语法分析器在看到方括号和圆括号之前将无法辨别它分析的是command_type_1还是command_type_2，但是要想看到方括号和圆括号需要向前查看两个记号。解决它的方法同样是扁平化，像我们前面做的那样，或者使得规则不会重叠，我们会在下一节讨论这种方法。

还有一些情况是规则中间的动作成为必须被归约的匿名规则：

```
statement: command_list { <action for '[' form>}:' '[' ...
          | command_list { <action for '(' form>}:' '(' ...
```

这个语法已经被扁平化，所以如果不使用GLR语法分析器的话你将无能为力。它只是需要向前查看两个记号，而LALR(1)语法分析器并不具备这个能力。除非你确实需要在语法分析器和词法分析器之间实现一些特别的交互，否则你应该把动作置后。

```
statement: command_list ':' '[' { <action for '[' form>} ...
          | command_list ':' '(' { <action for '(' form>} ...
```

规则重叠（归约/归约冲突）

这种情况下，相同的左部存在两条可选的规则，而这两条规则可以接受的输入又有部分的重叠。在常规bison语法分析器中解决这种冲突的最简单方法就是消除这两个输入集合的重叠部分。例如：

```
person:      girls
           |      boys
           ;
girls:       ALICE
           |       BETTY
           |       CHRIS
           |       DARRYL
           ;
boys:        ALLEN
           |        BOB
           |        CHRIS
           |        DARRYL
           ;
```

这个语法在CHRIS和DARRYL上存在一个归约/归约冲突，因为bison无法确定它们需要被归约为girls还是boys。有几种解决这类冲突的方法。下面是其中的一种：

```
person:      girls | boys | either;  
  
girls:       ALICE  
|          BETTY  
;  
  
boys:        ALLEN  
|          BOB  
;  
  
either:      CHRIS  
|          DARRYL  
;
```

但是如果这个列表很长，又或者非常复杂，并不是关键字列表那么简单，你应该怎么做呢？如果你希望缩小重复部分，但是girls和boys在bison描述中的很多地方被引用的话，你又该怎么做呢？下面是一种可能的解决方法：

```
person:      just_girls  
|          just_boys  
|          either  
;  
  
girls:       just_girls  
|          either  
;  
  
boys:        just_boys  
|          either  
;  
  
just_girls:  ALICE  
|          BETTY  
;  
  
just_boys:   ALLEN  
|          BOB  
;  
  
either:      CHRIS  
|          DARRYL  
;
```

所有使用boys | girls的地方都必须修改。在这儿GLR也帮不了什么忙，因为原来的语法就存在二义性，你必须解决它。

但是如果无法消除重叠部分呢？如果你无法找到一种彻底的方法来分解重叠部分，那么

你将不得不保留这个归约/归约冲突，然后你可以使用GLR语法分析器以及第9章会讲到的技术来显式地处理这个二义性。

如果你不使用GLR语法分析器，bison将使用它默认的解决二义性的方法来处理归约/归约冲突，也就是选择在bison描述中第一个出现的规则。所以在前面第一个*girls* | *boys*的例子中，CHRIS和DARRYL将总是*girls*。如果交换*boys*和*girls*列表的位置的话，那么CHRIS和DARRYL将总是*boys*。你依然会看到归约/归约冲突的警告信息，而bison会为你消除重叠部分，但这却是你希望避免的。

总结

bison语法中的二义性和冲突仅仅只是代码错误的一种，它需要我们寻找和解决。本章提供了一些解决这种错误的技术。在随后的章节中，我们将看到其他类型的错误。

本章的目的是使得你能够在足够高的层面来理解这种错误和修正它。你可以通过如下步骤来进行复习：

- 在*name.output*中寻找移进/归约错误。
- 确定归约规则。
- 确定相关的移进规则。
- 找出归约规则在归约之后的去向。
- 通过这些信息，你应该有能力找出导致冲突的记号流。

找出归约规则在归约之后的去向通常是很简单的，正如我们所展示的那样。不过有时语法可能很复杂以至于无法使用我们的“搜索附近”的方法，在那种情况下你需要了解状态机的具体操作来找到归约后的状态。

练习

1. 所有归约/归约冲突和很多移进/归约冲突是由二义性语法所导致的。除了bison并不喜欢它们这个事实之外，还有什么原因使得二义性语法通常是有问题的？
2. 找到一个实际编程语言比如C、C++或者Fortran的语法，然后通过bison来执行它。这个语法有冲突吗？（基本上它们都会有。）请浏览*name.output*输出，然后确定导致冲突的原因。解决它们有多难？
3. 在做完前面这个练习后，想一下为什么语言总是使用二义性语法来定义和实现。

第8章

错误报告和恢复

前一章讨论了在**bison**语法中查找错误的技术。本章我们将把我们的注意力转向错误检测上——语法分析器和词法分析器是如何检测错误的。本章提供了一些技术来把错误检测和报告集成到语法分析器中。我们将修改第4章的SQL语法分析器来演示这些技术。

bison提供了**error**记号和**yyerror()**例程，它们对于一个程序的早期版本来说已经足够。但是，当这个程序渐渐完善起来时，特别是对于编程工具而言，提供更好的错误恢复机制就非常重要，它使得语法分析器可以在错误发生之后继续检测文件的后续内容，以便于提供更好的错误报告。

错误报告

错误报告应该尽可能详细地给出错误信息。默认的**bison**错误报告仅仅表明它发现一个语法错误，然后就停止分析。在我们的例子中，我们使用**yylineno**来报告行号。这可以提供错误的位置，但是无法报告这个文件中的其他错误，也无法指出错误在该行中的具体位置。**bison**的位置特性，本章稍后部分会讲到，是一种较为简单的方法来精确定位错误，能够指出特定的行号和列号。在我们的例子中，我们将只打印出位置信息，不过精确的位置信息可以允许一个可视化界面来高亮显示相关的文本。

分类各种可能的错误非常有用，我们可以构造一个错误类型的数组以及定义符号常量来标识错误。例如，在许多语言中，一种常见的错误是没有结束的字符串，另一种错误可能是使用了不正确的字符串类型（引号引起的字符串而不是一个标识符，或者反过来的情况）。一个语法分析器需要能够检测出下面这些错误：

- 常见的语法错误（比如，没有意义的行）
- 没有结束的字符串
- 错误的字符串类型（使用引号引起或者没有引起的情况）

- 注释中出现文件结尾或者其他一些需要终结符的项目
- 具有多重定义的名字，或者使用没有定义过的名字

错误检测的职责并不仅仅存在于**bison**中，事实上，很多基础性的错误更容易被词法分析器检测到。例如，常见的引号引起的字符串的匹配模式如下：

```
\\"[^\\\"\\n]*\\"
```

我们希望能够检测到一个没有结束引号的字符串。一种潜在的方法是添加一条新规则来捕获没有终结的字符串，如同我们在第4章的SQL语法分析器中所做的那样。如果引号引起的字符串直到文件结束都没有给出结束的引号，我们就打印一条错误信息：

```
\\"[^\\\"\\n]*\\"
{
    yyval.string = yytext;
    return QSTRING;
}
\\"[^\\\"\\n]*$ {
    warning("Unterminated string");
    yyval.string = yytext;
    return QSTRING;
}
```

接受不正确的输入然后把它报告为一个错误或者警告的技术很强大，它可以被用来改善编译器的错误报告。如果我们不添加这样的规则，编译器将报告一个通用的“语法错误”消息；而通过报告具体的错误，我们可以让用户精确地知道如何修改。在本章的稍后部分，我们将描述在这种错误发生后继续进行分析的方法。

bison的这种接受错误输入的能力可以通过把字符串当作标识符的错误使用测试来得到演示。例如，在MySQL中，我们很容易把一个用单引号引起的字符串'`string`'，与一个反引号引起的名字``name``混淆起来。在那些只有其中一种是有效的上下文里，你可以为另一种情况添加一条规则，然后仔细地诊断它。下面是`column_list`规则的一个版本，它被用来实现`SELECT...INTO`：

```
column_list: NAME { emit("COLUMN %s", $1); free($1); $$ = 1; }
| STRING      { yyerror("Column name %s cannot be a string", $1);
                 emit("COLUMN %s", $1); free($1); $$ = 1; }
| column_list ',' NAME   { emit("COLUMN %s", $3); free($3); $$ = $1 + 1; }
| column_list ',' STRING { yyerror("Column name %s cannot be a string", $3);
                           emit("COLUMN %s", $3); free($3); $$ = $1 + 1; }
;
;
```

如果用户输入了一个字符串而不是名字，它将调用`yyerror()`来报告具体的错误，然后它能够继续分析，就如同那个字符串就是名字一样。

一些简单的flex技术可以让你产生更好的错误报告而不是单调的默认信息。我们在SQL

语法分析器中使用的一种简单技术就可以报告行号和当前记号。选项`yylineno`可以在遇到每个`\n`字符时自动增加行号，而当前的记号总是可以从`yytext`中获得，所以一个简单但是有用的错误例程可以如下实现：

```
void yyerror(char *s)
{
    printf("%d: %s at %s\n", yylineno, s, yytext);
}
```

稍微复杂一些的技巧可以帮助我们每次保存一行输入：

```
%code {
char linebuf[500];
%
%*
\n.* { strncpy(linebuf, yytext+1, sizeof(linebuf)); /* 保存下一行 */
       yyless(1); /* 只读取\n然后重新分析 */
}
%%

void yyerror(char *s)
{
    printf("%d: %s at %s in this line:\n%s\n",
           lineno, s, yytext, linebuf);
}
```

模式`\n.*`匹配一个换行符和下一行内容。相应的语义动作保存该行，然后它通过`yyless()`把该行退回给词法分析器。

为了精确地定位错误记号在输入行中的位置，我们需要借助于位置信息。

位置

`bison`的位置特性可以把由行号和列号确定的一小节输入文件与语法分析器中的每个符号关联起来。位置信息被存储在`YYLTYPE`结构中，该结构的默认声明如下：

```
typedef struct YYLTYPE
{
    int first_line;
    int first_column;
    int last_line;
    int last_column;
} YYLTYPE;
```

我们稍后会看到如何在一些特定的情况下重写这个结构，比如，如果你希望为每个位置添加文件名或者其他额外的信息。

词法分析器会在它返回时为每个记号设置位置信息，而每次语法分析器归约一条规则

时，它自动地把最新需要执行的规则左部符号的位置信息设定为从第一个右部符号的开始到最后一个右部符号的结束。在语法分析器的语义动作中，你可以通过@\$来使用左部符号的位置信息，以及通过@1、@2等等来使用右部符号的位置信息。词法分析器必须把每个记号的位置信息放到yyloc中，语法分析器会在每次词法分析器返回记号时定义对应的yyloc。幸运的是，我们不需要为词法分析器的任何语义动作添加代码就可以实现这一点。

为语法分析器添加位置信息

bison在看到语义动作中使用@N的位置信息时就会自动地添加位置代码到语法分析器中，或者你也可以在程序的声明部分置入%locations声明。

我们还需要修改错误例程来使用位置信息。在我们的SQL例子中，我们有一个yyerror()，它使用yyloc中的当前位置信息，以及一个新的lyyerror()，它接受一个额外的表示错误位置的参数。这两个例程都会在错误报告之前打印出位置信息（如果有的话）。

```
/*在语法分析器末尾的代码部分 */
void
yyerror(char *s, ...)
{
    va_list ap;
    va_start(ap, s);

    if(yyloc.first_line)
        fprintf(stderr, "%d.%d-%d.%d: error: ", yyloc.first_line, yyloc.first_column,
                yyloc.last_line, yyloc.last_column);
    vfprintf(stderr, s, ap);
    fprintf(stderr, "\n");

}

void
lyyerror(YYLTYPE t, char *s, ...)
{
    va_list ap;
    va_start(ap, s);

    if(t.first_line)
        fprintf(stderr, "%d.%d-%d.%d: error: ", t.first_line, t.first_column,
                t.last_line, t.last_column);
    vfprintf(stderr, s, ap);
    fprintf(stderr, "\n");
}
```

注意这里对非零值first_line的保护性检查，对于右部为空的规则，它使用语法分析器堆栈的前一项的位置信息。

在这个语法分析器里，我们修改所有`yyerror()`调用为`lyyerror()`，这样就可以报告相应的记号。例如：

```
column_list: NAME      { emit("COLUMN %s", $1); free($1); $$ = 1; }
| STRING                { lyyerror(@1, "string %s found where name required", $1);
                           emit("COLUMN %s", $1); free($1); $$ = 1; }
...
select_opts: { $$ = 0; }
| select_opts ALL     { if($$ & 01) lyyerror(@2,"duplicate ALL option"); $$ = $1 | 01; }
...
insert_asgn_list:
NAME COMPARISON expr { if ($2 != 4) {
                           lyyerror(@2,"bad insert assignment to %s", $1); YYERROR;
                         }
                           emit("ASSIGN %s", $1); free($1); $$ = 1;
                         }
```

这就是我们对语法分析器需要做的所有事情，因为默认的位置更新代码已经为我们提供了正确的位置信息。

为词法分析器添加位置信息

由于我们的位置信息需要有能力报告出错误的行与列范围，因此词法分析器需要在每次它分析到一个记号时记录下当前的行号和列号，然后把信息返回到语法分析器的`yyloc`中。幸运的是，一个很少被人知道的特性`YY_USER_ACTION`使得实现变得简单。如果你的词法分析器的第一部分定义了宏`YY_USER_ACTION`，那么它会在`yylex`识别出每个记号时被调用，就在语义动作代码执行之前。我们可以定义一个新的变量，`yycolumn`，来记住当前的列号，同时我们将在词法分析器的定义部分如下定义`YY_USER_ACTION`：

```
%code {
/* 处理位置信息 */
int yycolumn = 1;

#define YY_USER_ACTION yyloc.first_line = yyloc.last_line = yylineno; \
    yyloc.first_column = yycolumn; yyloc.last_column = yycolumn+yyleng-1; \
    yycolumn += yyleng;
{}
```

由于`yyleng`，记号的长度，已经被设置过，所以我们可以直接使用它来计算`yyloc`和`yycolumn`的值。在有些情况下（注释和空白字符），识别到的记号并不被返回给语法分析器，而词法分析器会继续分析，但这并不影响我们填写`yyloc`。这种方法可以实现绝大多数的位置记录需求。

我们需要做的最后一件事情是在发现换行符时重置`yycolumn`为1（flex已经为我们处理了`yylineno`）。

```

NOT[ \t]+EXISTS { yyval.subtok = 1; return EXISTS; }

ON[ \t]+DUPLICATE { return ONDUPLICATE; } /* 由于有限的向前查看而不得不采取的方法 */

<COMMENT>\n { yycolumn = 1; }
<COMMENT><<EOF>> { yyerror("unclosed comment"); }

[ \t] /* 空白字符 */
\n { yycolumn = 1; }

```

在每个能够匹配换行符的模式里，我们把\n的情况分解到一个新的独立的模式中，然后在里面重置yycolumn为1。我们也简化了NOT EXISTS和ON DUPLICATE的模式，这样它们中间就不允许出现换行符。或者你也可以选择重新分析当前的记号来检查其中是否存在换行符，然后把yycolumn设置为换行符后的字符数。

这就足以使得我们可以在报告错误时给出具体的行号和列号。你可以看到它很容易被实现，所以我不觉得有什么理由不在你的bison语法分析器中使用位置信息，即使你并不需要每个记号和规则的列号信息。

更现实的带有文件名的位置信息

大多数我们所写的语法分析器能够处理多个输入文件。在位置数据中包含文件名会有多困难呢？我们很快就会看到，并不怎么困难。我们必须定义我们自己的YYLTYPE来包含一个指向文件名的指针。我们重定义了语法分析器的宏YYLLOC_DEFAULT，它可以在语法分析器归约一条规则时绑定相应的位置信息，我们还需要修改词法分析器中YY_USER_ACTION的代码来放置每个记号对应的文件名到yyloc中，此外还需要一些其他小修改来记住语法分析器正在读取的文件名。我们在语法分析器的定义部分加入下面这段代码：

```

%code requires {

char *filename; /* 给词法分析器使用的当前文件名 */

typedef struct YYLTYPE {
    int first_line;
    int first_column;
    int last_line;
    int last_column;
    char *filename;
} YYLTYPE;
#define YYLTYPE_IS_DEFINED 1 /* 告诉语法分析器我们使用自己的定义 */

#define YYLLOC_DEFAULT(Current, Rhs, N)
do
    if (N)
    {
        (Current).first_line = YYRHSLOC (Rhs, 1).first_line;

```

```

(�Current).first_column = YYRHSLOC (Rhs, 1).first_column; \
(�Current).last_line = YYRHSLOC (Rhs, N).last_line; \
(�Current).last_column = YYRHSLOC (Rhs, N).last_column; \
(�Current).filename = YYRHSLOC (Rhs, 1).filename; /* 新代码 */ \
}
else \
{ /* 空的规则右部 */ \
(�Current).first_line = (�Current).last_line = \
YYRHSLOC (Rhs, 0).last_line; \
(�Current).first_column = (�Current).last_column = \
YYRHSLOC (Rhs, 0).last_column; \
(�Current).filename = NULL; /* 新代码 */ \
}
while (o)
}

```

我们并没有选择从头开始编写结构和宏，相反，我们查看了第一版带有位置信息的语法分析器所生成的C代码，拷贝了YYLTYPE和YYLOC_DEFAULT的定义，然后做了小小的修改。默认的YYLTYPE定义由#*if !YYLTYPE_IS_DECLARED*包围，而默认的YYLOC_DEFAULT定义则由#*ifndef YYLOC_DEFAULT*包围，所以我们的新版本必须定义这两个条件编译宏来关闭默认版本。

这段代码实现在一个%code require { }块中。通常的%code { %}块把相应的代码放置在YYLTYPE的定义之后，这在生成的C程序中显得太晚了，而且该代码并不会被拷贝到生成的头文件中。这里的require可以让bison把代码放置在默认版本的前面，并且代码也会被拷贝到头文件中。

这个版本的YYLTYPE包含四个标准的域以及一个指向文件名的指针。然后是YYLTYPE_IS_DECLARED定义来防止产生标准版本的YYLTYPE。

宏YYLLOC_DEFAULT的代码比较多，它用来把一条规则右部符号的位置信息拷贝到新的左部符号中。这个宏的三个参数分别为：Current，左部的位置信息；Rhs，第一个右部符号位置结构的地址；N，右部符号个数。内部的宏YYRHSLOC返回特定右部符号的位置结构。如果N是非零值，也就是说，至少存在一个右部符号，它将从第一个到第N个符号中拷贝相关的信息。do...while(o)是C语言的习惯用法，它可以使得该宏的展开式能够被正确分析，即使在调用宏的地方加了分号（记住代码块结束处的{后并没有分号，比如这儿的else子句）。只有两行标记了“新代码”的语句才是新代码，其他都是从默认的YYLLOC_DEFAULT中拷贝过来的。

既然我们已经把文件名添加到YYSTYPE里，我们可以继续添加一些少量的代码到yyerror和l yyerror中报告文件名，同时我们还在main()中做了修改，从而在开始执行语法分析器之前把filename设置为具体的文件名或者字符串(stdin)。

```

void
yyerror(char *s, ...)
{
    va_list ap;
    va_start(ap, s);

    if(yyloc.first_line)
        fprintf(stderr, "%s:%d.%d-%d.%d: error: ", yyloc.filename, yyloc.first_line,
                yyloc.first_column, yyloc.last_line, yyloc.last_column);
    vfprintf(stderr, s, ap);
    fprintf(stderr, "\n");
}

void
lyyerror(YYLTTYPE t, char *s, ...)
{
    va_list ap;
    va_start(ap, s);
    if(t.first_line)
        fprintf(stderr, "%s:%d.%d-%d.%d: error: ", t.filename, t.first_line,
                t.first_column, t.last_line, t.last_column);
    vfprintf(stderr, s, ap);
    fprintf(stderr, "\n");
}

main(int ac, char **av)
{
    extern FILE *yyin;

    if(ac > 1 && !strcmp(av[1], "-d")) {
        yydebug = 1; ac--; av++;
    }

    if(ac > 1) {
        if((yyin = fopen(av[1], "r")) == NULL) {
            perror(av[1]);
            exit(1);
        }
        filename = av[1];
    } else
        filename = "(stdin)";

    if(!yyparse())
        printf("SQL parse worked\n");
    else
        printf("SQL parse failed\n");
} /* 主函数 */

```

对于词法分析器来说，我们只要添加一行代码——在宏YY_USER_ACTION中拷贝filename到yyloc.filename中：

```
#define YY_USER_ACTION yyloc.filename = filename; \
    yyloc.first_line = yyloc.last_line = yylineno; \
```

```
yyloc.first_column = yycolumn; yyloc.last_column = yycolumn+yleng-1; \
yycolumn += yleng;
```

现在我们的编译器就可以报告文件名、行号和列号。在带有支持输入文件切换的 `include` 语句的编译器中，基于这种技术的报告并不完全准确，因为如果错误本身跨越了两个输入文件的话，就只有前一个文件名会被报告出来，但是显而易见的是，我们可以添加代码使得 `YYLTYPE` 支持两个文件名。

错误恢复

我们在前一节着重讲解了错误报告，而在本节中，我们将讨论错误恢复。当一个错误被检测到时，`bison` 语法分析器将停留在一个有歧义的位置中。毫无置疑的是如果不对现有的语法分析器堆栈做一些调整，我们无法继续进行任何有意义的处理。

由于你可能在不同的环境下使用语法分析器，如果该环境下纠正错误和返回语法分析器的处理很简单，你就并不一定需要错误恢复。在其他一些环境中，比如编译器，是有可能需要从错误中彻底恢复过来，然后继续分析和寻找更多的错误，在语法分析器的结束阶段停止编译器。这种技术可以通过缩短编辑 - 编译 - 测试的循环次数来提高程序员的生产力，因为一次循环就可以修正几个错误。

Bison 错误恢复

`bison` 针对错误恢复有一些预先设定的措施，我们可以通过具有特殊目的的记号 `error` 来使用它。从本质上来说，记号 `error` 被用来在语法中寻找一个同步点 (*synchronization point*)，在该点之后分析很有希望可以继续进行。但只是有希望，并不确定。有时候有些恢复并没有移去足够多的错误状态，这样就无法继续分析，而错误消息会蜂拥而出。语法分析器需要到达一个分析可以继续的点，否则整个语法分析器将不得不停止分析。

在报告一个语法错误后，`bison` 语法分析器将丢弃语法分析器堆栈中的符号，直到它找到一个能够移进记号 `error` 的状态。接着它开始不断地读取和放弃输入记号，直到它发现一个在语法中紧跟记号 `error` 之后的记号。后面的这部分处理被称为再次同步 (*resynchronizing*)。然后它开始在一个恢复 (*recovering*) 状态中重新分析，这个恢复状态不产生任何后续的分析错误。一旦它成功地移进三个记号之后，它将认为恢复已经结束，它会离开这个恢复状态，重新开始正常的分析。

这就是 `bison` 错误恢复的基本技巧——尝试在输入流中向前移进足够远，这样新的输入就不怎么会被旧的输入所影响。

通过恰当的语言设计，错误恢复会比较容易实现。现代编程语言使用语句终止符，它可

以被当作一个很方便的同步点。比如，当我们分析一个C语言语法时，一个逻辑上的同步字符就是分号。错误恢复会导致其他问题的出现，比如在语法分析器跳过声明部分来寻找分号时会出现缺少声明的错误，但这些也可以被包括在整个错误恢复机制中。

在SQL语法分析器里，最简单的同步位置就是每条SQL语句末尾的分号。下面这些规则被添加到语法分析器，以便于在终止每条语句的分号处可以重新同步。

```
stmt_list: error ';' error in the first statement
| stmt_list error ';' error in a subsequent statement
;
```

状态缺失（比如，变量声明被丢弃）可能会导致大量的错误，这使得放弃大部分输入流的做法不太有效。一种抵消大量错误问题的方法是对报告的错误消息进行计数，然后在该数值达到一个特定值时停止编译过程。例如，一些C编译器在某个文件的错误超过10个时就停止编译了。

像其他bison规则一样，包含error的规则可以带有语义动作代码。我们可以在错误之后做一些清理工作，重新初始化数据状态，或者再次恢复到一个可以继续分析的点。例如，前面的错误恢复可以如下实现：

```
stmt_list: error ';' { yyerror("First statement discarded, try again"); }
| stmt_list error ';' { yyerror("Current statement discarded, try again"); }
;
```

释放被丢弃的符号

bison的错误恢复包括从内部堆栈弹出符号的过程。每个符号可以有一个语义值，而如果这些语义值带有指向分配内存或者数据结构的指针时，就会发生内存泄漏和数据被破坏的问题。`%destructor`声明可以让bison知道在它弹出一个具有语义值的符号时应该怎么做。该声明的语法如下：

```
%destructor { ... code ... } symbols or <types>
```

这样语法分析器在它每次弹出一个命名的符号或者一个给定类型的值的符号时，就执行相应的代码。我们可以为不同的丢弃处理分别声明不同的`%destructor`。类型`<*>`可以用来匹配没有出现在其他`%destructor`声明中的具有给定类型的符号的类型。

在我们的SQL语法分析器中，唯一需要特殊处理的符号是具有`<strval>`值的符号，它们就是一些需要被释放的字符串：

```
/* free discarded tokens */
%destructor { printf ("free at %d %s\n",@$first_line, $$); free($$); } <strval>
```

这段代码会打印出它正要做的事情，包括位置引用，这对于调试来说十分有用，但对于一个产品级的语法分析器来说就可能太夸张了。

交互式语法分析器中的错误恢复

当bison语法分析器被设计用来直接从终端读取输入时，有一些技巧可以帮助改善错误恢复。下面是一个典型的读取一连串命令的语法分析器：

```
commands:          /* 空 */
|      commands command
;

command:   . . .
|  error {
    yyclearin /* 放弃预读的记号 */
    yyerrok;
    printf("Enter another command\n");
}
;
```

宏`yyclearin`用来放弃任何预读的记号，而`yyerrok`则让语法分析器重新开始正常的分析，所以该程序将以用户输入的下一条命令作为开始。

如果你的代码需要报告自己的错误，那么你的错误例程可以使用bison的宏`YYRECOVERING()`来测试该语法分析器是否正在尝试再次同步，这种情况下你不应该打印任何错误信息，例如：

```
warning(char *err1, char *err2)
{
    if (YYRECOVERING() )
        return; /* 这种情况下不打印错误信息 */
    . .
}
```

如何放置error记号

在一个语法中正确地放置error记号是一种两难的抉择。你希望使得再次同步很有可能成功，因此你会把error记号放置在语法的最高层规则中，甚至是起始规则中，这样语法分析器总是可以找到一条作为恢复起点的规则。但另一方面，你会希望在恢复之前尽可能少地放弃输入，所以你会把error记号放置在语法的最低层规则中，来减少恢复过程中不得不放弃的那些部分匹配的规则。在实践中最经常使用的恢复点是列表中用来分隔元素的标点符号。

如果你的顶层规则匹配一个列表（比如，SQL语法分析器中的语句列表）或者C编译器

中的一连串声明和定义，你可以让这个列表的某一条规则包含error，正如前面命令行的例子和SQL的例子。这也适用于相对高层的列表，比如C函数中的语句列表。

例如，由于C语句由分号和花括号强调，在C编译器中，你可以如下编写：

```
stmt:      ...
|   RETURN expr ';'
|   '{' opt_decls stmt_list '}'
|   error ';'
|   error '}'
;
```

两条包含error的规则告诉语法分析器它必须在;或者}之后寻找下一条语句。

你也可以把error规则放在较低层次的位置，例如放在表达式规则中，但是除非语言本身提供了特定的标点符号或者关键字来使得它能够很容易辨别表达式的结束位置，否则语法分析器不太可能从那么低的层次进行恢复。

编译器错误恢复

在前一节我们讨论了bison所提供的错误恢复机制。在本节中我们将讨论程序员如何来扩展恢复机制。

错误恢复依赖于语法的语义知识而不仅仅是语法知识。这在很大程度上更加复杂化了语法的错误恢复。

对于错误恢复例程而言，它可以建立在启发式的性能适中的错误恢复上来分析输入。例如，C编译器程序员可以决定在一个代码块的声明部分发现的错误应该通过放弃整段代码块来实现恢复，而不是继续报告额外的错误。她也可以决定在代码块发生的错误应该直接跳到下一条语句。而对于那些雄心勃勃的程序员来说，他们在实现编译器或者解释器的时候会希望报告错误的同时能够给出可能的解决方法。一些恢复机制已经在尝试往输入流中插入一些在错误发生点语法分析器可能接受的新记号。它们会做一些试探性的分析，然后判断其建议的修正方法是否可以让语法分析器继续读取输入。

此前已经有很多关于错误修正和恢复的著作，其中的大部分发表于批量计算的年代，那时再次运行程序所需要等待的时间要用小时或者天来计算，编译器开发者们不得不尝试猜测程序的错误原因以及如何解决它。根据我的经验，可以说他们的猜测并不怎么样，那些错误并不是一成不变或者微不足道的，也因此阻碍了相应的解决机制。对于今天的计算机来说，你在几秒钟内就可以再次运行程序，所以编译器并不需要猜测程序员的意图，也不需要在发生严重错误后继续分析，更加合理的做法是尽可能快地恢复到一个状态，使得程序员可以修改输入然后再次执行编译过程。

练习

1. 为第3章的计算器添加错误恢复。最有可能恢复的点是每条语句末尾的EOL记号。请不要忘记释放抽象语法树、符号和符号列表的析构过程。
2. (学期课题) `bison`的错误恢复会放弃输入记号，直到它发现一些输入在语法上是正确的。另外一种方法则是选择插入而不是放弃记号，因为很多情况下你很容易判断出下一个记号应该是什么。例如，在一个C程序中，每个`break`和`continue`之后必须带有一个分号，而每个`case`之前必须有个分号或者右花括号。如果我们增强`bison`语法分析器，使得它在输入错误时能够建议一个需要插入的合适的记号，这会有多困难呢？你需要了解更多的`bison`内在的知识来完成这个练习。`bison`语法分析器的框架具有一些未被记载的代码，它会尝试建议一些有效的记号，你可以从这些代码入手。

Flex和Bison进阶

早期的**bison**是**yacc**的一个版本，而**yacc**则是最初的Unix语法分析器生成器，它可以生成C语言版的LALR语法分析器。在最近这些年里，**bison**增加了很多新特性。我们将在这里讨论其中最有用的一些。

纯词法分析器和纯语法分析器

使用通常方法生成的flex词法分析器和bison语法分析器并不支持重入，它们只能一次分析一个输入流。这是因为这两者都使用了静态数据结构来记录分析过程，它们之间以及它们与调用程序之间的通信也依赖于静态数据结构。**flex**和**bison**可以创建“纯”可重入的代码，这种代码把所有静态数据结构替换为参数传递给词法分析器和语法分析器中的例程。这就使得对于词法分析器和语法分析器的递归调用成为可能，而递归调用有时会很有用，同时它还允许词法分析器和语法分析器被用在多线程程序中，在那里不同线程中的各种分析会同步进行。

作为演示，我们修改第3章中的计算器，使得它使用纯词法分析器和纯语法分析器。我们并不会让语法分析器立即执行每一行代码，相反，我们会让它返回抽象语法树给调用方。与大多数可重入程序相同的是，调用例程会分配一个空间结构来存储每个实例的数据，并且在每次调用时把这个结构传递给词法分析器和语法分析器。

不幸的是，在这本书将要出版的时候（2009年中），flex纯词法分析器和bison纯语法分析器的代码简直是一团糟。对于一个纯的yylex()来说，bison的调用方式与flex完全不同，而它们处理每个实例数据的方法也不同。不过我们还是有可能弥补现有代码中的问题，让纯词法分析器和纯语法分析器可以协同工作，而这正是我们在本章要做的事情，但在做这些之前，我们需要查看一下最新的flex和bison的说明文档。大多数不兼容性可以通过修改用来创建词法分析器和语法分析器的代码框架加以解决，而很有可能有些人已经实现了这种修改，这样你就不再需要再做一遍。

Flex中的纯词法分析器

单一的分析工作可能需要调用若干次`yylex()`，因为`yylex()`需要返回记号给调用程序。由于词法分析器的状态必须在多次调用之间得以保存，你就不得不自行管理每个词法分析器的数据。`flex`提供了一些例程来创建和销毁一个词法分析器的上下文数据，以及其他一些例程用来访问词法分析器的值，这些值以前被实现为类似于`yyin`和`yytext`这样的静态变量，而现在你可以通过这些例程在`yylex()`之外设置和获取这些值。

```
yyscan_t scaninfo;    指向词法分析器每个实例数据的指针
int yylex_init(&scaninfo);    创建一个词法分析器
int yylex_init_extra(userstuff, &scaninfo);    或者创建一个词法分析器，其带有指向用户
                                                数据的指针

yyset_in(stdin, scaninfo);    设置输入文件和其他参数

while( ... ) {
    tok = yylex(scaninfo);    一直调用直到分析结束
}

yylex_destroy(scaninfo);    释放词法分析器的数据
```

结构`yyscan_t`包含每个词法分析器的状态，比如输入和输出文件，以及用来记住输入缓冲区位置以便于下次继续分析的指针。它还包含一个指向结构`YY_BUFFER_STAT`的指针堆栈，它用来记录所有处于激活状态的输入缓冲区。（正如我们在第2章看到的那样，内在的缓冲区堆栈不是太有用，因为你通常需要记住额外的一些与每个缓冲区有关的信息。）

`yylex_init_extra`中增加的`userstuff`使得你可以为词法分析器提供每个实例的数据，比如符号表的地址。这个变量的类型是`YY_EXTRA_TYPE`，默认情况下被定义为`void *`，但你可以很容易通过`%option extra-type`来重定义它。每个实例的数据被固定存储在一个数据结构中，所以`userstuff`会是指向该结构的指针。如我们先前所见，你可以在词法分析器中通过一行代码来获取它，然后把它放在一个具有合适名字和类型的指针变量中。

在词法分析器中，`flex`为`yytext`、`yylen`和其他一些指向实例数据的域定义了宏。`yylineno`和`yycolumn`并没有出现在非可重入词法分析器中，它们的值被存储在当前缓冲区结构里，这样就使得词法分析器更容易在多个输入文件的时候记录行和列的信息，而其他部分变量则保存在`yyscan_t`结构中。`flex`会像对待非可重入词法分析器一样来维护`yylineno`的值，对于`yycolumn`而言，`flex`唯一自动做的事情就是在它发现`\n`字符时把`yycolumn`清零，所以你依然需要使用第8章的技术来自行记录列号。

例9-1展示了第2章的字数统计程序，它被修改为使用一个纯词法分析器。

例9-1：字数统计程序的纯词法分析器版

```
/*字数统计程序的纯词法分析器版*/
%option noyywrap nodefault reentrant
%{
struct pwc { our per-scanner data
    int chars;
    int words;
    int lines;
};

%}
%option extra-type="struct pwc *"

%%
%{
    struct pwc *pp = yyextra;          这段代码出现在yylex之上
                                         yyextra是flex定义的宏
%}

[a-zA-Z]+ { pp->words++; pp->chars += strlen(yytext); }
\n { pp->chars++; pp->lines++; }
. { pp->chars++; }

%%
```

三个用来计算字符数、单词数和行数的变量现在被放在一个结构中，该结构在词法分析器的每个实例中都有一份拷贝，并且通过`%option extra-type`使得flex词法分析器中的`userstuff`成为指向该结构的指针。

规则部分的第一行代码被flex放在`yylex`之上，也就是在flex变量定义之后和任何可执行代码之前。这行代码使我们可以声明一个指向我们自己的实例数据的指针并把它初始化为`yyextra`，`yyextra`是flex提供的宏，它指向词法分析器当前实例数据中的额外数据域。在我们的这些规则里，原本指向静态数据的地方现在改为指向我们的实例数据。

```
main(argc, argv)
int argc;
char **argv;
{
    struct pwc mypwc = { 0, 0, 0 }; /* 我的实例数据 */
    yyscan_t scanner; /* flex的实例数据 */

    if(yylex_init_extra(&mypwc, &scanner)) {
        perror("init alloc failed");
        return 1;
    }

    if(argc > 1) {
        FILE *f;

        if(!f = fopen(argv[1], "r")) {
            perror(argv[1]);
            return (1);
        }
    }
}
```

```

    yyset_in(f, scanner);
} else
    yyset_in(stdin, scanner);

yylex(scanner);
printf("%d%d%d\n", mypwc.lines, mypwc.words, mypwc.chars);

if(argc > 1)
    fclose(yyget_in(scanner));

yylex_destroy(scanner);
}

```

主函数会声明和初始化`mypwc`、我们的实例数据并且声明`scanner`——它将成为flex的实例数据。对于`yylex_init_extra`的调用可以传递一个指向`scanner`的指针，所以我们使用一个指向最新分配实例的指针，该调用在返回0时代表成功而返回1时代表失败（如果实例数据的`malloc`操作失败的话）。

如果存在传入的文件，我们就打开这个文件，并且使用`yyset_in`把该文件句柄存储在词法分析器数据中的`yyin`相关的域。接着我们调用`yylex`，把flex的实例数据传递给它，打印出词法分析器存储在我们的实例数据中的结果，然后我们释放和销毁词法分析器。

可以看到，它比通常的非纯词法分析器需要更多的工作，但这些主要是针对机制部分的修改：把静态数据转移到一个结构中，把对静态数据的使用转为对结构的使用，以及添加创建和销毁词法分析器实例的代码。

Bison中的纯语法分析器

纯语法分析器的创建比纯词法分析器要简单一些，因为整个**bison**的分析发生在一次`yyparse`的调用中。因此，语法分析器可以在开始分析时创建它的实例数据，进行分析，然后释放它，而不需要其他程序方面的指导。语法分析器确实需要一些应用层面的实例数据来作为参数传递给`yyparse`。被用来与词法分析器通信的静态变量——`yyval`和（如果语法分析器使用位置信息的话）`yyloc`，将变成必须传递给`yylex`的实例数据，它们可以与应用层面的实例数据放在一起。

`bison`在看到`%define api.pure`（以前是`%pure_parser`）声明时会创建一个纯语法分析器。该声明使得语法分析器是可重入的。为了启动一个纯语法分析器，你需要传递给它一个指向应用层面的每个实例数据的指针。`%parse-param`声明的内容被放置在`yyparse()`定义的圆括号内，所以你可以声明任意多的参数，不过通常来说你只需要一个指向实例数据的指针：

```

#define api.pure
%parse-param { struct pureparse *pp }

```

纯语法分析器也改变了`yylex()`的调用方式，需要传递一个指向当前`yyval`拷贝的指针，如果使用位置信息的话，还需要传递一个指向`yyloc`拷贝的指针。

```
/* 语法分析器中生成的调用 */
token = yylex(YYSTYPE *yylvalp);           不带有位置信息
token = yylex(YYSTYPE *yylvalp, YYLTYPE *yylocp); 带有位置信息
```

如果你希望传递应用数据，你可以通过`%lex-param{ }`或者`#define YYLEX_PARAM`来声明它。不过一些轻率的过度优化可能会导致`%lex-param`的参数只剩下花括号内的最后一个记号，所以最好使用`YYLEX_PARAM`。（说明文档表明该实现假定你会提供一个声明，就如同在`parse-param`中的做法一致，但是由于针对flex词法分析器的参数必须是词法分析器的`yyscan_t`，所以我总是从每个实例的应用数据中获取它。）

```
%code {
#define YYLEX_PARAM pp->scaninfo
%
%%
/* 语法分析器中生成的调用 */
token = yylex(YYSTYPE *yylvalp, pp->scaninfo);           不带有位置信息
token = yylex(YYSTYPE *yylvalp, YYLTYPE *yylocp, pp->scaninfo); 带有位置信息
```

当生成的语法分析器遇到一个语法错误时，它将调用`yyerror()`，如果语法分析器使用位置信息，它会传递一个指向当前位置的指针，以及除了通常的错误消息字符串之外的语法分析器参数^(注1)：

```
yyerror(struct pureparse *pp, "Syntax error"); 不带有位置信息
yyerror(YYLTYPE &yylocp, struct pureparse *pp, "Syntax error"); 带有位置信息
```

如果你使用一个手写的词法分析器，这些就是你需要为纯语法分析器实现的所有钩子代码（hook）。当你在词法分析器和语法分析器中调用内部例程时，你会希望同时传递实例数据，我们会在本章稍后的可重入计算器中看到这一点。但是由于我们使用flex词法分析器，所以首先我们需要解决flex和bison的不兼容调用方法。

使纯词法分析器和纯语法分析器协同工作

如果你比较一下纯词法分析器和纯语法分析器对于`yylex`的调用方法，你就会注意到它们并不兼容。flex希望第一个参数是词法分析器的实例数据，而bison则把第一个参数作为指向`yyval`的指针。虽然我们可以使用一些没有记载的C预处理器符号来把这个问题蒙混过去，flex的维护人员还是出于对程序员的怜悯而添加了选项`%bison-bridge`，使得

注1： 如果你需要传递多个参数给语法分析器，每个参数都必须声明在独立的`%parse-param`中。
如果你把两个参数放在同一个`%parse-param`中，bison不会报告任何错误，但是第二个参数会消失。

它的纯词法分析器的调用方法可以与bison的兼容。如果你使用`%option bison-bridge`, `yylex`的声明将变为:

```
int yylex(YYSTYPE* lvalp, yyscan_t scaninfo);
```

如果你使用`%option bison-bridge bison-locations`, 声明将变为:

```
int yylex (YYSTYPE* lvalp, YYLTYPE* llocp, yyscan_t scaninfo);
```

flex定义了宏`yyval`和（可选的）`yyloc`，它们是参数的拷贝，但由于它们分别是指向bison值的联合结构和位置结构的指针，所以`yyval.field`和`yyloc.first_line`必须更改为`yyval->field`和`yyloc->first_line`。这实际上是个错误，但是它被记录在flex和bison的手册中，所以它不太可能被修改。

多重词法分析器和语法分析器的多个实例

本章中关于纯分析的讨论包含了你使用单一词法分析器和语法分析器的多个拷贝的情况，它们都使用相同的规则。我们也有可能在单一程序中使用不同词法分析器或者语法分析器的多个实例。第131页的“单一程序中的多重词法分析器”和第168页的“可变语法和多重语法”两节描述了如何把多个不同的词法分析器和语法分析器放到单一程序中，我们只需要把符号的前缀从“yy”修改为其他字符串。你可以以相同的方法来重命名纯词法分析器和纯语法分析器，然后在需要的时候调用它们。这种复杂程度对调试来说是一种挑战，但你可以基于你的需要来决定是否使用这种方法。

可重入计算器

让第3章的计算器支持可重入的修改大多是机制上的，也就是把静态数据放到每个实例结构中。与在语法分析器内直接执行分析好的抽象语法树不同的是，新版本的计算器每次只分析一个表达式或者一个函数定义，然后它会返回抽象语法树给调用者，由调用者来决定是立即运行还是先保存起来。在另一方面，词法分析器被作为单一会话来管理，所有`yyparse`的调用都需要经过这个会话，这样每次语法分析器重启时词法分析器就不会丢失任何缓冲的输入。这就意味着程序会在启动时创建词法分析器的上下文，然后它每次传递相同的上下文给语法分析器。例9-2展示了计算器程序修改后的头文件。

例9-2：可重入计算器头文件`purecalc.h`

```
/*
 * 计算器声明，纯模式版本
 */
```

```

/* 分析数据 */
struct padata {
    yyscan_t scaninfo;           /* 词法分析器上下文 */
    struct symbol *syntab;       /* 本次分析的符号 */
    struct ast *ast;             /* 最近分析的抽象语法树 */
};

```

新的数据结构padata包含了语法分析器的应用上下文。它包括一个指向符号表的指针，允许不同的分析带有不同的命名空间；一个词法分析器上下文，语法分析器会传递它给yylex；一个返回结果，语法分析器会把分析完的抽象语法树放在其中。（记住yparse的值为1或者0代表了分析成功与否，所以它不能简单返回抽象语法树。）

头文件中剩余部分的修改为所有函数添加了一个初始上下文参数，包括那些计算器所特有的函数和yyerror。

```

/* 符号表 */
struct symbol {                  /* 变量名 */
    char *name;
    double value;
    struct ast *func;            /* 函数的抽象语法树 */
    struct symlist *syms;        /* 虚拟参数列表 */
};

/* 固定大小的简单符号表 */
#define NHASH 9997
struct symbol syntab[NHASH];

struct symbol *lookup(struct padata *, char*);

/* 针对参数列表的符号列表 */
struct symlist {
    struct symbol *sym;
    struct symlist *next;
};

struct symlist *newsymlist(struct padata *, struct symbol *sym, struct symlist *next);
void symlistfree(struct padata *, struct symlist *sl);

/* 节点类型
 * + - * / ]
 * 0-7 为比较操作符，位编码：04 等于，02 小于，01大于
 * M 单目负号
 * L 语句列表
 * I IF语句
 * W WHILE语句
 * N 符号引用
 * = 赋值
 * S 符号列表
 * F 内置函数调用
 * C 用户函数调用
 */

enum bifs { /* 内置函数 */

```

```

B_sqrt = 1,
B_exp,
B_log,
B_print
};

/* 抽象语法树中的节点 */
/* 所有节点都有共同的初始nodetype */

... 所有节点都与初始版本一致 ...

/* 构造一棵抽象语法树 */
struct ast *newast(struct pcdata *, int nodetype, struct ast *l, struct ast *r);
struct ast *newcmp(struct pcdata *, int cmptype, struct ast *l, struct ast *r);
struct ast *newfunc(struct pcdata *, int functype, struct ast *l);
struct ast *newcall(struct pcdata *, struct symbol *s, struct ast *l);
struct ast *newref(struct pcdata *, struct symbol *s);
struct ast *newasgn(struct pcdata *, struct symbol *s, struct ast *v);
struct ast *newnum(struct pcdata *, double d);
struct ast *newflow(struct pcdata *, int nodetype, struct ast *cond, struct ast *tl,
struct ast *tr);

/* 定义函数 */
void dodef(struct pcdata *, struct symbol *name, struct symlist *syms, struct ast *stmts);

/* 计算抽象语法树 */
double eval(struct pcdata *, struct ast *);

/* 删除和释放抽象语法树 */
void treefree(struct pcdata *, struct ast *);

/* 与词法分析器的接口 */
void yyerror(struct pcdata *pp, char *s, ...);

```

词法分析器带有选项`reentrant`和`bison-bridge`，这使它成为一个可重入的与`bison`兼容的词法分析器。同时我们第一次让`flex`创建了一个与`bison`类似的头文件。这个文件包含各种各样的函数定义，它们被用来在词法分析器的上下文中获取和设置变量，该文件还包含语法分析器所需的`yytoken_t`的定义。

千万不要在你的词法分析器中包含该头文件！

如果你创建了词法分析器的头文件，本例中为`purecalc.lex.h`，请确保不要在你的词法分析器中直接或者间接地包含它。由于某些原因，该头文件带有若干词法分析器内部宏的`#undef`，这会导致词法分析器的编译错误，产生一些模糊的变量未定义的错误消息。（猜猜我是怎么发现的。）

如果你需要在一个公共的头文件中包含这个词法分析器的头文件，你可以通过`#ifndef/#define`代码行来保护`#include`语句，或者你可以像这个例子所做的一样，只在需要该头文件的文件中直接包含它。

例9-3：可重入计算器的词法分析器purecalc.l

```
/* 识别计算器的记号 */
/* 纯词法分析器和纯语法分析器版本 */
/* $Header: /usr/home/johnl/flnb/RCS/ch09.tr,v 1.4 2009/05/19 18:28:27 johnl Exp $ */
%option noyywrap nodefault yylineno reentrant bison-bridge

%option header-file="purecalc.lex.h"

%{
#include "purecalc.tab.h"
#include "purecalc.h"
%}
/* 浮点数指数部分 */
EXP ([Ee][-+]?[0-9]+)

%%
%{
    struct padata *pp = yyextra;
%}
/* 单字符操作符 */
"+"
"-"
"*"
"/"
"="
"|"
","
";"
 "("
 ")" { return yytext[0]; }

/* 比较操作符 */
">" { yylval->fn = 1; return CMP; }
"<" { yylval->fn = 2; return CMP; }
"<>" { yylval->fn = 3; return CMP; }
"==" { yylval->fn = 4; return CMP; }
">>=" { yylval->fn = 5; return CMP; }
"<=" { yylval->fn = 6; return CMP; }

/* 关键字 */
"if" { return IF; }
"then" { return THEN; }
"else" { return ELSE; }
"while" { return WHILE; }
"do" { return DO; }
"let" { return LET; }

/* 内置函数 */
"sqrt" { yylval->fn = B_sqrt; return FUNC; }
"exp" { yylval->fn = B_exp; return FUNC; }
"log" { yylval->fn = B_log; return FUNC; }
"print" { yylval->fn = B_print; return FUNC; }

/* 名字 */
```

```

[a-zA-Z][a-zA-Z0-9]* { yyval->s = lookup(pp, yytext); return NAME; }

[0-9]+."[0-9]*{EXP}? |
".? [0-9]+{EXP}? { yyval->d = atof(yytext); return NUMBER; }

/*.*
[ \t] /* 忽略空白字符 */
\\n printf("c> "); /* 忽略续行符 */
\\n" { return EOL; }

. { yyerror(pp, "Mystery character %c\\n", *yytext); }
<<EOF>> { exit(0); }
%%

```

例9-3展示了词法分析器为了支持可重入而做的修改。规则之上的一行代码把指向应用层面实例数据的指针`yyextra`放到变量`pp`中，这使我们可以在后面访问其中的任何域。以前直接引用`yyval`的地方现在改为指针方式，而`yyerror`的调用则增加了实例数据作为参数。

在词法分析器的结束位置是`<EOF>`规则，它被用来退出词法分析器。这种退出程序的方式不太优雅，但是可以满足我们的目的。我们会在后面讨论一些其他退出程序的方式。

例9-4展示了语法分析器为了支持可重入而做的修改。它主要有两方面的修改。一方面是机制层面的修改，以便于处理显式状态；另一方面则会更改语法分析器，使它一次只处理一条语句。

例9-4：可重入计算器的语法分析器purecalc.y

```

/* 带有抽象语法树的计算器 */
#define api.pure
%parse-param { struct padata *pp }

%{
#include <stdio.h>
#include <stdlib.h>
%}

%union {
    struct ast *a;
    double d;
    struct symbol *s; /* 符号 */
    struct symlist *sl;
    int fn; /* 函数 */
}

%{
#include "purecalc.lex.h"
#include "purecalc.h"
#define YYLEX_PARAM pp->scaninfo
%}

```

```

/* 声明记号 */
%token <d> NUMBER
%token <s> NAME
%token <fn> FUNC
%token EOL

%token IF THEN ELSE WHILE DO LET

%nonassoc <fn> CMP
%right '='
%left '+' '-'
%left '*' '/'
%nonassoc '|' UMINUS

%type <a> exp stmt list explist
%type <sl> symlist

%start calc
%%

```

这个语法分析器文件定义了api.pure来生成一个可重入的语法分析器，它还使用parse-param来声明新的语法分析器所需的一个参数，指向应用状态的指针。随后的代码块包含了flex生成的头文件purecalc.lex.h，并且定义了YYLEX_PARAM来传递词法分析器状态到词法分析器中，该状态存储在实例数据中。

```

calc: /* 空 */ EOL { pp->ast = NULL; YYACCEPT; }
    | stmt EOL { pp->ast = $1; YYACCEPT; }
    | LET NAME '(' symlist ')' '=' list EOL {
        dodef(pp, $2, $4, $7);
        printf("%d: Defined %s\n", yyget_lineno(pp->scaninfo),
               $2->name);
        pp->ast = NULL; YYACCEPT; }
    ;

```

顶层规则现在变为calc，它可以处理空行、一条分析为抽象语法树的语句或者保存在本地符号表中的函数定义。通常一个bison语法分析器会不断读取记号流直到文件结束。而这个语法分析器使用YYACCEPT来结束分析。当语法分析器结束分析时，它会保持词法分析器的当前状态，这样下次语法分析器再次启动时，可以使用相同的词法分析器状态，这样它会从上次离开的点继续读取。另外一种方法是可以在用户输入换行符时让词法分析器返回一个文件结束记号，这也达到我们的目的。对于目前的情况而言，并没有什么特别的理由支持说哪一种方法更好。

语法分析器的剩余部分与非重入的版本没有差异，除了每次针对外部例程的调用现在需要传入一个指向实例数据的指针。正如我们将要看到的那样，很多例程并不处理状态变量，但显而易见，简单地为每个例程都传递实例状态比起试图记住哪些要传递而哪些又不要传递要容易的多。

```

stmt: IF exp THEN list { $$ = newflow(pp, 'I', $2, $4, NULL); }
| IF exp THEN list ELSE list { $$ = newflow(pp, 'I', $2, $4, $6); }
| WHILE exp DO list { $$ = newflow(pp, 'W', $2, $4, NULL); }
| exp
;

list: /* 空 */ { $$ = NULL; }
| stmt ';' list { if ($3 == NULL)
                  $$ = $1;
                else
                  $$ = newast(pp, 'L', $1, $3);
              }
;

exp: exp CMP exp { $$ = newcmp(pp, $2, $1, $3); }
| exp '+' exp { $$ = newast(pp, '+', $1,$3); }
| exp '-' exp { $$ = newast(pp, '-', $1,$3); }
| exp '*' exp { $$ = newast(pp, '*', $1,$3); }
| exp '/' exp { $$ = newast(pp, '/', $1,$3); }
| exp '|' exp { $$ = newast(pp, '|', $2, NULL); }
| '(' exp ')' { $$ = $2; }
| '-' exp %prec UMINUS { $$ = newast(pp, 'M', $2, NULL); }
| NUMBER { $$ = newnum(pp, $1); }
| FUNC '(' explist ')' { $$ = newfunc(pp, $1, $3); }
| NAME { $$ = newref(pp, $1); }
| NAME '=' exp { $$ = newasgn(pp, $1, $3); }
| NAME '(' explist ')' { $$ = newcall(pp, $1, $3); }
;

explist: exp
| exp ',' explist { $$ = newast(pp, 'L', $1, $3); }
;
symplist: NAME { $$ = newsymlist(pp, $1, NULL); }
| NAME ',' symplist { $$ = newsymlist(pp, $1, $3); }
;
%%
```

例9-5展示了辅助函数为可重入词法分析器和语法分析器所做的调整。现在每个实例状态都有一个符号表，所以查询符号表的例程需要从状态结构中获取符号表指针。

例9-5：辅助函数purecalcfuncs.c

```

/*
 * 纯计算器的辅助函数
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <math.h>

#include "purecalc.tab.h"
#include "purecalc.lex.h"
#include "purecalc.h"
```

```

/* 符号表 */
/* 哈希一个符号 */
static unsigned
symhash(char *sym)
{
    unsigned int hash = 0;
    unsigned c;

    while(c = *sym++) hash = hash*9 ^ c;

    return hash;
}

struct symbol *
lookup(struct pcdata *pp, char* sym)
{
    struct symbol *sp = &(pp->syntab)[symhash(sym)%NHASH];
    int scount = NHASH; /* 需要查看的个数 */

    while(--scount >= 0) {
        if(sp->name && !strcmp(sp->name, sym)) { return sp; }

        if(!sp->name) { /* 新条目 */
            sp->name = strdup(sym);
            sp->value = 0;
            sp->func = NULL;
            sp->syms = NULL;
            return sp;
        }

        if(++sp >= pp->syntab+NHASH) sp = pp->syntab; /* 尝试下一个条目 */
    }
    yyerror(pp, "symbol table overflow\n");
    abort(); /* 尝试完所有的条目，符号表已满 */
}

struct ast *
newast(struct pcdata *pp, int nodetype, struct ast *l, struct ast *r)
{
    struct ast *a = malloc(sizeof(struct ast));

    if(!a) {
        yyerror(pp, "out of space");
        exit(0);
    }
    a->nodetype = nodetype;
    a->l = l;
    a->r = r;
    return a;
}

struct ast *
newnum(struct pcdata *pp, double d)

```

```

{
    struct numval *a = malloc(sizeof(struct numval));
    if(!a) {
        yyerror(pp, "out of space");
        exit(0);
    }
    a->nodetype = 'K';
    a->number = d;
    return (struct ast *)a;
}

struct ast *
newcmp(struct pcdta *pp, int cmptype, struct ast *l, struct ast *r)
{
    struct ast *a = malloc(sizeof(struct ast));

    if(!a) {
        yyerror(pp, "out of space");
        exit(0);
    }
    a->nodetype = 'o' + cmptype;
    a->l = l;
    a->r = r;
    return a;
}

struct ast *
newfunc(struct pcdta *pp, int functype, struct ast *l)
{
    struct fncall *a = malloc(sizeof(struct fncall));

    if(!a) {
        yyerror(pp, "out of space");
        exit(0);
    }
    a->nodetype = 'F';
    a->l = l;
    a->functype = functype;
    return (struct ast *)a;
}

struct ast *
newcall(struct pcdta *pp, struct symbol *s, struct ast *l)
{
    struct ufcall *a = malloc(sizeof(struct ufcall));

    if(!a) {
        yyerror(pp, "out of space");
        exit(0);
    }
    a->nodetype = 'C';
    a->l = l;
    a->s = s;
    return (struct ast *)a;
}

```

```

struct ast *
newref(struct padata *pp, struct symbol *s)
{
    struct symref *a = malloc(sizeof(struct symref));

    if(!a) {
        yyerror(pp, "out of space");
        exit(0);
    }
    a->nodetype = 'N';
    a->s = s;
    return (struct ast *)a;
}

struct ast *
newasgn(struct padata *pp, struct symbol *s, struct ast *v)
{
    struct symasgn *a = malloc(sizeof(struct symasgn));

    if(!a) {
        yyerror(pp, "out of space");
        exit(0);
    }
    a->nodetype = '=';
    a->s = s;
    a->v = v;
    return (struct ast *)a;
}

struct ast *
newflow(struct padata *pp, int nodetype, struct ast *cond, struct ast *tl, struct ast *el)
{
    struct flow *a = malloc(sizeof(struct flow));

    if(!a) {
        yyerror(pp, "out of space");
        exit(0);
    }
    a->nodetype = nodetype;
    a->cond = cond;
    a->tl = tl;
    a->el = el;
    return (struct ast *)a;
}

struct symlist *
newsymlist(struct padata *pp, struct symbol *sym, struct symlist *next)
{
    struct symlist *sl = malloc(sizeof(struct symlist));

    if(!sl) {
        yyerror(pp, "out of space");
        exit(0);
    }
    sl->sym = sym;
}

```

```

    sl->next = next;
    return sl;
}

void
symlistfree(struct pldata *pp, struct symlist *sl)
{
    struct symlist *nsl;

    while(sl) {
        nsl = sl->next;
        free(sl);
        sl = nsl;
    }
}

/* 定义函数 */
void
dodef(struct pldata *pp, struct symbol *name, struct symlist *syms, struct ast *func)
{
    if(name->syms) symlistfree(pp, name->syms);
    if(name->func) treefree(pp, name->func);
    name->syms = syms;
    name->func = func;
}

static double callbuiltin(struct pldata *pp, struct fncall *);
static double calluser(struct pldata *pp, struct ufcall *);

double
eval(struct pldata *pp, struct ast *a)
{
    double v;

    if(!a) {
        yyerror(pp, "internal error, null eval");
        return 0.0;
    }

    switch(a->nodetype) {
        /* 常量 */
        case 'K': v = ((struct numval *)a)->number; break;

        /* 名字引用 */
        case 'N': v = ((struct symref *)a)->s->value; break;

        /* 赋值 */
        case '=': v = ((struct symasgn *)a)->s->value =
            eval(pp, ((struct symasgn *)a)->v); break;

        /* 表达式 */
        case '+': v = eval(pp, a->l) + eval(pp, a->r); break;
        case '-': v = eval(pp, a->l) - eval(pp, a->r); break;
        case '*': v = eval(pp, a->l) * eval(pp, a->r); break;
        case '/': v = eval(pp, a->l) / eval(pp, a->r); break;
    }
}

```

```

case '|': v = fabs(eval(pp, a->l)); break;
case 'M': v = -eval(pp, a->l); break;

/* 比较 */
case '1': v = (eval(pp, a->l) > eval(pp, a->r))? 1 : 0; break;
case '2': v = (eval(pp, a->l) < eval(pp, a->r))? 1 : 0; break;
case '3': v = (eval(pp, a->l) != eval(pp, a->r))? 1 : 0; break;
case '4': v = (eval(pp, a->l) == eval(pp, a->r))? 1 : 0; break;
case '5': v = (eval(pp, a->l) >= eval(pp, a->r))? 1 : 0; break;
case '6': v = (eval(pp, a->l) <= eval(pp, a->r))? 1 : 0; break;

/* 控制流 */
/* 语法中允许空的if/else/do表达式，所以需要检查这种可能性*/
case 'I':
    if( eval(pp, ((struct flow *)a)->cond) != 0) {
        if( ((struct flow *)a)->tl) {
            v = eval(pp, ((struct flow *)a)->tl);
        } else
            v = 0.0; /* 默认值 */
    } else {
        if( ((struct flow *)a)->el) {
            v = eval(pp, ((struct flow *)a)->el);
        } else
            v = 0.0; /* 默认值 */
    }
    break;

case 'W':
    v = 0.0; /* 默认值 */

    if( ((struct flow *)a)->tl) {
        while( eval(pp, ((struct flow *)a)->cond) != 0)
            v = eval(pp, ((struct flow *)a)->tl);
    }
    break; /* 最后一次的值就是需要返回的值 */

case 'L': eval(pp, a->l); v = eval(pp, a->r); break;

case 'F': v = callbuiltin(pp, (struct fncall *)a); break;

case 'C': v = calluser(pp, (struct ufcall *)a); break;

default: printf("internal error: bad node %c\n", a->nodetype);
}
return v;
}

static double
callbuiltin(struct padata *pp, struct fncall *f)
{
    enum bifs functype = f->functype;
    double v = eval(pp, f->l);

    switch(functype) {
    case B_sqrt:

```

```

        return sqrt(v);
    case B_exp:
        return exp(v);
    case B_log:
        return log(v);
    case B_print:
        printf("= %4.4g\n", v);
        return v;
    default:
        yyerror(pp, "Unknown built-in function %d", functype);
        return 0.0;
    }
}

static double
calluser(struct padata *pp, struct ufn call *f)
{
    struct symbol *fn = f->s; /* 函数名 */
    struct symlist *sl; /* 虚拟参数 */
    struct ast *args = f->l; /* 实际参数 */
    double *oldval, *newval; /* 保存的参数值 */
    double v;
    int nargs;
    int i;

    if(!fn->func) {
        yyerror(pp, "call to undefined function", fn->name);
        return 0;
    }

    /* 计算参数个数 */
    sl = fn->syms;
    for(nargs = 0; sl; sl = sl->next)
        nargs++;

    /* prepare to save them */
    oldval = (double *)malloc(nargs * sizeof(double));
    newval = (double *)malloc(nargs * sizeof(double));
    if(!oldval || !newval) {
        yyerror(pp, "Out of space in %s", fn->name); return 0.0;
    }

    /* 计算参数值 */
    for(i = 0; i < nargs; i++) {
        if(!args) {
            yyerror(pp, "too few args in call to %s", fn->name);
            free(oldval); free(newval);
            return 0;
        }

        if(args->nodetype == 'L') { /* 是否是节点列表 */
            newval[i] = eval(pp, args->l);
            args = args->r;
        } else { /* 是否是列表末尾 */
            newval[i] = eval(pp, args);
        }
    }
}

```

```

        args = NULL; *
    }
}

/* 保存虚拟参数的旧值，赋予新值 */
sl = fn->syms;
for(i = 0; i < nargs; i++) {
    struct symbol *s = sl->sym;

    oldval[i] = s->value;
    s->value = newval[i];
    sl = sl->next;
}
free(newval);

/* 计算函数 */
v = eval(pp, fn->func);

/* 恢复虚拟参数的值 */
sl = fn->syms;
for(i = 0; i < nargs; i++) {
    struct symbol *s = sl->sym;

    s->value = oldval[i];
    sl = sl->next;
}

free(oldval);
return v;
}

void
treefree(struct padata *pp, struct ast *a)
{
    switch(a->nodetype) {

        /* 两棵子树 */
        case '+':
        case '-':
        case '*':
        case '/':
        case '1': case '2': case '3': case '4': case '5': case '6':
        case 'L':
            treefree(pp, a->r);

        /* 一棵子树 */
        case '|':
        case 'M': case 'C': case 'F':
            treefree(pp, a->l);

        /* 没有子树 */
        case 'K': case 'N':
            break;

        case '=':
    }
}

```

```

    free( ((struct symasgn *)a)->v);
    break;

    case 'I': case 'W':
        free( ((struct flow *)a)->cond);
        if( ((struct flow *)a)->t1) free( ((struct flow *)a)->t1);
        if( ((struct flow *)a)->el) free( ((struct flow *)a)->el);
        break;
    default: printf("internal error: free bad node %c\n", a->nodetype);
    }
    free(a); /* 总是释放节点自身 */
}

```

当前行号以前保存在静态变量yylineno中，现在函数yyerror需要通过yyget_lineno来从词法分析器状态中获取它。

```

void
yyerror(struct pcdata *pp, char *s, ...)
{
    va_list ap;
    va_start(ap, s);

    fprintf(stderr, "%d: error: ", yyget_lineno(pp->scaninfo));
    vfprintf(stderr, s, ap);
    fprintf(stderr, "\n");
}

```

主函数负责创建实例数据，它通过yylex_init_extra把指向应用实例数据的指针放在词法分析器实例中，接着它把指向词法分析器实例的指针保存到p.scaninfo中。然后它分配一个清空的符号表，准备开始分析。

在这个简单例子中，每次主函数调用语法分析器的时候，如果语法分析器返回一个抽象语法树，主函数就立即计算该抽象语法树然后释放它。

```

int
main()
{
    struct pcdata p = { NULL, 0, NULL };

    /* 设置词法分析器 */
    if(yylex_init_extra(&p, &p.scaninfo)) {
        perror("init alloc failed");
        return 1;
    }

    /* 分配和清空符号表 */
    if(!(p.symtab = calloc(NHASH, sizeof(struct symbol)))) {
        perror("sym alloc failed");
        return 1;
    }

    for(;;) {

```

```

    printf( "> ";
    yyparse(&p);
    if(p.ast) {
        printf( "= %4.4g\n" , eval(&p, p.ast));
        treefree(&p, p.ast);
        p.ast = 0;
    }
}
}
}

```

纯应用的Makefile

这个Makefile比前几章中的Makefile要更复杂一些，因为purewc词法分析器和语法分析器都使用了对方创建的头文件。

```

CFLAGS = -g

all: purewc purecalc

purewc: purewc.lex.o
    cc -g -o $@ purewc.lex.c

purewc.lex.c: purewc.l
    flex -opurewc.lex.c purewc.l

purecalc: purecalc.lex.o purecalc.tab.o purecalcfuns.o
    cc -g -o $@ purecalc.tab.o purecalc.lex.o purecalcfuns.o -lm

purecalc.lex.o: purecalc.lex.c purecalc.tab.h purecalc.h
purecalc.tab.o: purecalc.tab.c purecalc.lex.h purecalc.h
purecalc.lex.c purecalc.lex.h: purecalc.l
    flex -opurecalc.lex.c purecalc.l

purecalc.tab.c purecalc.tab.h: purecalc.y
    bison -vd purecalc.y

```

推语法分析器和拉语法分析器

bison有一个实验性质的推分析 (*push parse*) 选项，它可以关闭内部的控制流。一个正常的拉 (*pull*) 语法分析器启动后持续地调用`yylex`来把每个记号“拉”进语法分析器。而在一个推 (*push*) 语法分析器中，你需要创建一个`yyystate`语法分析器状态，然后你为每个记号调用语法分析器，传递记号和状态给它来把每个记号“推”入语法分析器。语法分析器会对该记号做它所能做的事情，移进、归约、调用任何语义动作例程，然后在每个记号处理完后返回。推语法分析器通常也是可重入的，更多是由GUI或者相似程序中的事件函数来调用。由于它们还处在实验阶

段，实现细节很有可能发生变化，因此请参考最新的bison手册来了解当前的调用方法。

目前并没有flex的推词法分析器。对于推词法分析器而言，每次调用将需要传入一串输入文本，而它将处理该文本并返回多个记号。由于推词法分析器的控制流将不再与拉语法分析器有任何关系，因此推词法分析器并不返回，它将直接调用推语法分析器，传入当前分析的记号和值。当词法分析器处理完所有的输入，它需要记住它所在的位置，可以在自动的分析中保存当前位置或者备份至上一个识别记号的末尾，然后保存下一次所需的剩余文本并返回到调用方。

任何人都可以实现这样一种flex词法分析器。它是开源的，所以你如果感兴趣，你就可以来修改它。

GLR分析

更强的力量意味着更大的责任

语法分析器产生器，比如yacc和bison，变得越来越流行的一个重要原因是它们可以创建比手工实现更加可靠的语法分析器。如果你交给bison一个语法而它没有报告任何冲突，那么你可以确信生成的语法分析器所接受的语言与该语法所描述的语言完全一致。它不会有任何手工实现的语法分析器可能有的漏洞，特别是对于诊断错误输入来说。如果你可以保守地使用优先级声明来解决已知位置的冲突、表达式语法、if/then/else，那么你依然可以保证你的语法分析器会像你想的那样分析语言。

从另一方面来说，如果你使用GLR分析，你可以交给bison任何语法，它会创建一个能够实现一定分析的语法分析器，并在分析期来解决冲突。但当有越多冲突存在时，它越不可能像你期望的那样来解决冲突。在我们开始GLR之前，请确保你理解你的语法为什么有一些期望GLR来处理的冲突，并且你知道你会如何解决它们。否则，你就是在冒险，你的语法分析器很有可能会由于你所没有预料到的冲突而停止工作，也有可能由于不正确的解决冲突的方法而导致它所分析的语言并不是你想要的。

理论上来说GLR分析器会非常慢，因为并行的执行N个分析大致上要比单个分析慢上N倍，而一些二义性比较严重的语法可能要在每个记号上都分裂出不同的语法分析器来。有用的GLR语法通常只有很少的二义性，它们可以在较少的记号内解决，所以性能可以满足要求。

通常的bison LALR语法分析器并不需要处理移进/归约和归约/归约冲突，因为当语法分

析器被创建时，那些冲突已经通过这种或者那种方法得以解决（参见第8章）。但是当一个GLR语法分析器遇到冲突时，理论上它会分裂为多个语法分析器来分别处理多种可能，每个语法分析器会平行地分析输入记号。当存在多个冲突时，它将创建一棵部分分析的树，在每次有冲突时就分裂一次。

如果一个语法并没有二义性，它只是需要向前查看比LALR(1)提供的单一记号更多的记号时，大多数分析会发现它们无法匹配下一个记号而导致失败。`bison`会默默地抛弃掉失败的分析而继续那些还存活的分析。如果所有可能的分析都失败了，`bison`将像往常一样报告一个错误。对于这样的语法，GLR语法分析器工作起来与普通的LALR语法分析器非常相似，你只需要添加少量的代码来让`bison`使用GLR语法分析器，并且告诉它你期望有多少冲突。

另一方面，如果语法确实有歧义，语法分析器会发现相同左部符号存在两个或者更多的归约规则，它不得不决定归约哪一条规则。如果你知道它应该总是归约相同的规则，你可以在每条规则中加入标签`%dprec N`来设置它们之间的优先级。如果有歧义的归约规则都有`%dprec`，语法分析器会归约具有最高N的那条规则。你还可以选择使用`%merge`，它可以让语法分析器调用你所写的例程来检查所有规则归约的结果，然后“合并”这些结果到左部符号需要使用的值中。

当GLR语法分析器正在处理多个可能的分析时，它会记住为每种分析做的归约，但它并不调用语义动作例程。在它解决了二义性之后，应该只剩下一个成功的分析，或者它可以通过`%dprec`标签来选定一个分析，然后它会调用相应的语义动作例程。通常这种做法并没有什么问题，但是如果你的语法分析器需要返回一些信息给词法分析器，设置起始状态或者其他词法分析器需要测试的标志，你就有可能遇到一些难以调试的错误，因为这种语法分析器并不会在它的分析过程立即设置状态和标志。

SQL语法分析器的GLR版本

第4章的SQL语法分析器使用了一些词法技巧来解决LALR语法分析器的局限性。我们将去掉这些技巧而直接使用GLR语法分析器。其中的一个技巧考虑到向前查看的限制而把`ONDUPLICATE`作为一个单一记号来看待。另外一个技巧把`NOTEXISTS`作为`EXISTS`变体的一个单一记号，因为不这么做的话语法会存在二义性。而新版的语法分析器可以简单地去掉它们，从而把`EXISTS`、`ON`和`DUPLICATE`当作普通关键字来看待。

```
EXISTS { return EXISTS; }
ON { return ON; }
DUPLICATE { return DUPLICATE; }
```

在这个语法分析器中，语法将变得很简单，`ON DUPLICATE`会作为两个独立的记号，而且会有一条单独的规则来处理`NOT EXISTS`。

```

opt_ondupupdate: /* 空 */
| ON DUPLICATE KEY UPDATE insert_asgn_list { emit("DUPUPDATE %d", $5); }
;
...
expr: ...
| NOT expr { emit("NOT"); }
| EXISTS '(' select_stmt ')' { emit("EXISTS 1"); }
| NOT EXISTS '(' select_stmt ')' { emit("EXISTS 0"); }
;

```

下一步是通过bison来运行这个语法，我们使用参数-v来创建bison列表，这样可以知道它说了些什么。本例中，它表明这个语法存在2个移进/归约冲突和59个归约/归约冲突：

```

State 249 conflicts: 1 shift/reduce
State 317 conflicts: 1 shift/reduce
State 345 conflicts: 59 reduce/reduce

```

在切换到GLR语法分析器之前有一点很重要，我们需要确保这些列出的冲突是我们所期望的，所以我们会来查看一下列表文件中的这三个状态：

```

state 249

55 join_table: table_reference STRAIGHT_JOIN table_factor .
56 | table_reference STRAIGHT_JOIN table_factor . ON expr

ON shift, and go to state 316

ON [reduce using rule 55 (join_table)]
$default reduce using rule 55 (join_table)

state 317

54 join_table: table_reference opt_inner_cross JOIN table_factor . opt_join_condition

ON shift, and go to state 377
USING shift, and go to state 378

ON [reduce using rule 70 (opt_join_condition)]
$default reduce using rule 70 (opt_join_condition)

opt_join_condition go to state 379
join_condition go to state 380

state 345

263 expr: EXISTS '(' select_stmt ')'.
264 | NOT EXISTS '(' select_stmt ')'.

NAME reduce using rule 263 (expr)
NAME [reduce using rule 264 (expr)]

... 57 个更多的归约/归约冲突...

```

```
'')' reduce using rule 263 (expr)
'')' [reduce using rule 264 (expr)]
$default reduce using rule 263 (expr)
```

状态249和状态317是由于无法向前查看到ON，而状态345则是无法确定把NOT EXISTS作为一个操作符还是两个操作符来看待（存在大量的冲突是由于NOT EXISTS之后的记号可以是在表达式之后的任意有效记号）。在我们确认这些冲突是我们期望的之后，我们添加三行代码到定义部分，一行设定GLR语法分析器，而另外两行则告诉bison期望的冲突个数。如果你更改语法而导致冲突数量发生变化，那么bison就会失败，这是很好的防范措施，这样你就可以重新来确定新冲突集是否依然是你期望的。

```
%gllr-parser
%expect 2
%expect-rr 59
```

通过这些额外的代码，语法分析器可以开始重新编译了，而且它可以正常处理大部分情况。它能够正确地分析使用ON和ON DUPLICATE的语句，但我们还没有处理完表达式的歧义性：

```
insert into foo select a from b straight_join c on d;
rpn: NAME a
rpn: TABLE b
rpn: TABLE c
rpn: NAME d
rpn: JOIN 128
rpn: SELECT 0 1 1
rpn: INSERTSELECT 0 foo
rpn: STMT

insert into foo select a from b straight_join c on duplicate key update x=y;
rpn: NAME a
rpn: TABLE b
rpn: TABLE c
rpn: JOIN 128
rpn: SELECT 0 1 1
rpn: NAME y
rpn: ASSIGN x
rpn: DUPUPDATE 1
rpn: INSERTSELECT 0 foo
rpn: STMT

select not exists(select a from b);
rpn: NAME a
rpn: TABLE b
rpn: SELECT 0 1 1
Ambiguity detected.
Option 1,
expr -> <Rule 247, tokens 2 .. 9>
NOT <tokens 2 .. 2>
expr -> <Rule 263, tokens 3 .. 9>
```

```

EXISTS <tokens 3 .. 3>
'(' <tokens 4 .. 4>
select_stmt <tokens 5 .. 8>
')' <tokens 9 .. 9>

Option 2,
expr -> <Rule 264, tokens 2 .. 9>
NOT <tokens 2 .. 2>
EXISTS <tokens 3 .. 3>
'(' <tokens 4 .. 4>
select_stmt <tokens 5 .. 8>
')' <tokens 9 .. 9>

1: error: syntax is ambiguous
SQL parse failed

```

bison会在GLR语法分析器中产生非常完善的诊断代码。这里我们可以看到两种可能的分析：第一种把NOT EXISTS作为两个操作符，而第二种则把它们作为一个操作符。这个问题很容易通过%*dprec*来解决，因为单操作符版本就是我们所要的。

```

expr: ...
| NOT expr { emit("NOT"); } %dprec 1
...
| NOT EXISTS '(' select_stmt ')' { emit("EXISTS 0"); } %dprec 2

```

现在语法分析器就可以正确工作了。另外一种解决这个二义性分析的方法是提供你自己的函数来处理两条规则返回的结果，然后返回最终的归约规则的值。该函数的参数和值的类型为YYSTYPE，它与flex为%union声明所创建的联合结构的名字一致。如果存在多个归约/归约冲突，你需要为每个你希望自行解决的冲突定义单独的函数。冲突所涉及的每条规则需要有一个%*merge*标签，而特定规则所涉及的所有规则需要具有相同的标签：

```

%{
YYSTYPE exprmerge(YYSTYPE x1, YYSTYPE x2);
%}
expr: ...
| NOT expr { emit("NOT"); } %merge <exprmerge>
...
| NOT EXISTS '(' select_stmt ')' { emit("EXISTS 0"); } %merge <exprmerge>

```

使用%*merge*的应用需要精心设计，否则会极其复杂。因为合并函数在所有可能的规则被归约之后才执行，规则的值必须是抽象语法树或者其他类似实现来包含所有可能分析的必要信息。这个合并函数可以查看所有的抽象语法树，然后选择其中之一返回而放弃其他的抽象语法树，请记住在放弃之前需要释放相应的分配内存。GLR语法分析器处理的经典二义性是C++的声明，它在语法上与具有强制类型转化的赋值语句并无差异，解决该问题的规则很简单，就是认为所有可能是声明的语句就是声明语句，这样%*dprec*就足以用来处理这种情况而无需合并函数。

在很多情况下并不需要使用GLR语法分析器，你最好选择在常规的LALR语法分析器中执行你的词法分析器和语法分析器，但是如果你有一个预先定义的语言并不适用于LALR，GLR会成为你的救命稻草的。

C++语法分析器

bison可以创建使用C++的语法分析器。虽然flex看起来也能够创建C++的词法分析器文件，但它所生成的C++代码并不能正常工作^(注2)。幸运的是，flex创建的C版词法分析器可以在C++里进行编译，而把flex C版的词法分析器和bison C++版的语法分析器联合起来使用并不困难，我们会在下一个例子里来做这样的尝试。

所有bison C++版的语法分析器都是可重入的，所以bison为语法分析器创建了一个类。在使用可重入的语法分析器时，程序员可以创建他所需要数量的实例，然后传人在另一个类中保存的每个实例的应用数据。

每次你创建C++版的语法分析器时，bison会创建四个类的头文件：`location.hh`和`position.hh`用来定义位置结构，`stack.hh`定义内部语法分析器堆栈，以及一个定义语法分析器自身的头文件。前三个头文件内容并不会变化，最后一个头文件则包含该语法分析器的特定内容，与它的C版相似。语法分析器头文件会包含其他三个文件，这四个头文件可以被词法分析器和其他需要处理位置信息的模块包含。（有人可能会问为什么不把前三个文件做成标准库的包含文件。）创建语法分析器的头文件是强制性的，因为生成的C++源文件会包含它，尽管你依然需要告诉bison来创建头文件。

一个C++版的计算器

C++版的语法分析器比C版的语法分析器要更加复杂，所以为了让代码好管理，下面这个例子是基于第1章的最简单的计算器。为了让它变得有趣一些，这个计算器可以工作在从2到10的任意基数里，该基数信息被保存在每个语法分析器的应用上下文中。

应用类`cppcalc_ctx`被定义在头文件`cppcalc-ctx.hh`中，如例9-6所示。

例9-6：C++版计算器的应用上下文类`cppcalc-ctx.hh`

```
class cppcalc_ctx {
public:
    cppcalc_ctx(int r) { assert(r > 1 && r <= 10); radix = r; }

    inline int getradix(void) { return radix; }
private:
```

注2：这已经被flex的作者确认了。它最终很有可能被修正，但事实上为flex词法分析器设计一个良好的C++接口还是非常困难的。

```
    int radix;  
};
```

C++版语法分析器命名

除非另行指定，bison将在命名空间yy中创建语法分析器，其类名为parser。这个语法分析器类有个方法称为parse，你可以在创建完该类的实例后调用它。命名空间yy可以通过声明%define namespace来修改，而类名则可以通过%define parser_class_name来修改。这个例子中，我们把类名修改为cppcalc。这个类包含语法分析器的私有数据。它还包括一些可以通过预处理器符号YYDEBUG，特别是set_debug_level(N)，来启用的调试方法，其中的N需要是非零值。

一个C++版的语法分析器

例9-7中展示的C++版的语法分析器使用了一些新的声明。

例9-7：C++版计算器语法分析器cppcalc.yy

```
/* C++版的计算器 */  
  
%language "C++"  
%defines  
%locations  
  
%define parser_class_name "cppcalc"  
  
%{  
#include <iostream>  
using namespace std;  
#include "cppcalc-ctx.hh"  
%}  
  
%parse-param { cppcalc_ctx &ctx }  
%lex-param { cppcalc_ctx &ctx }  
  
%union {  
    int ival;  
};  
  
/* 声明记号 */  
%token <ival> NUMBER  
%token ADD SUB MUL DIV ABS  
%token OP CP  
%token EOL  
  
%type <ival> exp factor term  
  
%{  
extern int yylex(yy::cppcalc::semantic_type *yyval,
```

```

yy::cppcalc::location_type* yyloc,
cppcalc_ctx &ctx);

void myout(int val, int radix);
%}

%initial-action {
// 位置信息中的文件名
@$	begin.filename = @$._end.filename = new std::string("stdin");
}
%%

```

这个语法分析器使用`%language`来声明它使用C++而不是C, `%defines`用来创建头文件, `%locations`用来把处理位置信息的代码加入语法分析器, `%define`则用来调用类`cppcalc`而不是默认的语法分析器。然后是一些C++代码用来包含*iostream*库和前面提到的上下文头文件。

我们在第216页的“Bison中的纯语法分析器”一节已经遇见过`%parser-param`和`%lex-param`声明, 我们为语法分析器(语法分析器的类构造函数)和`yylex`增加了一个额外的参数。这个例子中, 我们并没有使用可重入词法分析器, 如果我们用的话, 传给词法分析器的参数必须是一个与前例一致的flex词法分析器上下文`yyScan_t`。我们可以使用相同的技巧来把它存在语法分析器的参数中, 不过现在我们只需要把这个参数设置为上下文类的一个实例。

我们声明了一个`%union`, 它与C版中的效果一致。在这个简单例子中, 它只有一个成员, 表达式的整数值。接下来则是记号和非终结符声明。

C++和`%union`

尽管C语言允许一个联合包含数据结构, C++却并不允许一个联合包含类实例, 所以你不能够直接在`%union`中使用一个类。指向类实例的指针会比较适合这种场合而且也是在C++语法分析器中常用的实现方法。由于指向的类实例需要通过`new`来动态分配, 所以在那些右部符号存在类指针值的语义动作里, 你需要找到合适的地方来存放每个指针值, 以便于它们的后续使用或者删除。如果你使用语法分析器错误恢复, 请使用`%destructor`声明来告知bison如何释放那些在错误恢复中丢弃的值, 否则这个程序就会有内存泄漏问题。

这些规则同样适用于通过`malloc`来分配的C的结构类型, 比如我们在第3章创建的抽象语法树。

由于某些原因, bison并不创建C++所要求的`yylex`的声明, 所以我们在这里需要手工声

明它。它的参数与纯C版本的例子一致：指向记号值的指针、记号位置和指向lex-param上下文的指针。在C++版语法分析器中，记号值和位置信息的类型为semantic_type和location_type，它们是语法分析器类的成员。我们也定义了myout，这是一个输出例程，可以使用特定基数（radix）来打印值。

在使用位置信息的C版语法分析器中，如果你希望输出文件名，你必须添加自己的代码来实现这一点，参见第205页的“更现实的带有文件名的位置信息”，但C++版的位置信息考虑到了这个问题，它在每个位置的开始和结束点添加了filename（文件名）字符串域。%initial-action用来设置\$@中初始位置信息中的文件名，该文件名作为位置信息的一部分也会传递给词法分析器。这就是所有我们要做的事情，因为本例中的词法分析器只读取一个文件，所以不需要更新文件名。

```
// C++版语法分析器的bison规则
calclist: /* 空 */
| calclist exp EOL { cout << "="; myout(ctx.getradix(), $2); cout << "\n> "; }
| calclist EOL { cout << "> "; } /* blank line or a comment */
;

exp: factor
| exp ADD factor { $$ = $1 + $3; }
| exp SUB factor { $$ = $1 - $3; }
| exp ABS factor { $$ = $1 | $3; }
;

factor: term
| factor MUL term { $$ = $1 * $3; }
| factor DIV term { if($3 == 0) {
                    error(@3, "zero divide");
                    YYABORT;
                }
                    $$ = $1 / $3; }
;

term: NUMBER
| ABS term { $$ = $2 >= 0? $2 : - $2; }
| OP exp CP { $$ = $2; }
;
%%
```

语法分析器中的规则与第1章中的实现没有什么差别，只是语义动作代码从C改成了C++。注意打印顶层calclist表达式的规则现在调用myout来打印，它会把从ctx结构中获得的基数信息传递给该函数。除法规则现在会检测除零的情况，它在检测到错误时调用语法分析器类的成员函数error，以前是yyerror。（C++语法分析器框架有一个错误，即使你不使用%location，它在调用error时也会带有位置信息参数。最简单的解决方法就在C++版语法分析器中总是使用位置信息。）

```
// 语法分析器的C++代码段
```

```

main()
{
    cppcalc_ctx ctx(8); // work in octal today

    cout << "> ";

    yy::cppcalc_parser parser(ctx); // make a cppcalc parser

    int v = parser.parse(); // and run it

    return v;
}

// 使用指定基数打印整数
void
myout(int radix, int val)
{
    if(val < 0) {
        cout << "-";
        val = -val;
    }
    if(val > radix) {
        myout(radix, val/radix);
        val %= radix;
    }
    cout << val;
}

int
myatoi(int radix, char *s)
{
    int v = 0;

    while(*s) {
        v = v*radix + *s++ - '0';
    }
    return v;
}

namespace yy {
    void
    cppcalc::error(location const &loc, const std::string& s) {
        std::cerr << "error at " << loc << ":" << s << std::endl;
    }
}

```

与C版的纯语法分析器不同的是，C++版的纯语法分析器要求你首先创建一个语法分析器的实例，然后才能调用它。因此，主程序首先创建一个具有合理基数的ctx结构，接着创建一个yy::parser的实例parser，该实例使用ctx作为上下文，然后调用方法parse来执行实际的分析工作。

两个辅助例程，myout和myatoi，用来进行基数和二进制的转化。最后是yy::error，

这是个与yyerror等价的错误例程。由于某种原因，bison在语法分析器类中把error定义为私有成员函数，这就意味着你不能在其他地方调用它；特别是，你不能够在词法分析器中调用它。bison手册建议让yy::error调用实际的错误例程，该错误例程可以定义在词法分析器可见的上下文中，这可能是最佳的解决手段。顺便说一下，你需要注意这个错误例程使用了C++的<<操作符来输出错误所在的位置。之所以可以这样做是因为location类定义了多种操作符，其中包括输出格式化。

连接C++版语法分析器和词法分析器

例9-8中的flex词法分析器是使用兼容C++的C来编写的。

例9-8：C++版计算器词法分析器cppcalc.l

```
/*识别C++版计算器的记号并且把它们打印出来 */

%option noyywrap
%{
#include <cstdlib>

#include "cppcalc-ctx.hh"
#include "cppcalc.tab.hh"

#define YY_DECL int yylex(yy::cppcalc::semantic_type *yylval, \
    yy::cppcalc::location_type *yylloc, cppcalc_ctx &ctx)
// 让位置信息包含当前记号
#define YY_USER_ACTION yylloc->columns (yyleng);

typedef yy::cppcalc::token token;
extern int myatoi(int radix, char *s); // 该方法定义在语法分析器中
%}
%%
```

词法分析器的定义部分包含了C语言的标准库和上下文与语法分析器类的头文件。它定义了YY_DECL来使得yylex的调用方式符合语法分析器的要求，它还定义了YY_USER_ACTION，该宏在每个记号的语义动作之前被调用，来根据记号的长度设置位置信息。这个技巧我们在第8章也用过，但这儿的代码更短，因为C++版的位置信息拥有我们需要的方法。

语法分析器的记号编号在语法分析器类的token成员中被定义，因此我们typedef了名字token，以方便设定记号值的类型。

```
// C++兼容的词法分析器的规则
%{
    // 从上次结束的记号开始
    yylloc->step ();
%}
```

```

"+ { return token::ADD; }
"- { return token::SUB; }
"* { return token::MUL; }
"/ { return token::DIV; }
"|" { return token::ABS; }
 "(" { return token::OP; }
 ")" { return token::CP; }
[0-9]+ { yyval->ival = myatoi(ctx.getradix(), yytext); return token::NUMBER; }

\n { yylloc->lines(1); return token::EOL; }

/* 跳过注释和空白字符 */
"/\/*.* |
[\t] { yylloc->step (); }

. { printf("Mystery character %c\n", *yytext); }
%%
```

规则部分开始的代码会被拷贝到yylex的开头部分。`step`方法把位置的起始值设置为与结束值相等，这样位置就指向了上一个记号的结束位置。（另一种方法我们在第8章中尝试过，把行号和列号记录在本地变量中，然后把它们拷贝到每个记号的位置信息中，现在的办法可以借助于C++版里位置的预定义方法而使得代码变短。）

语义动作代码为每个记号名字都添加了`token::`前缀，因为这些记号名字现在是语法分析器类的成员。换行符的语义动作使用方法`lines`来更新位置中的行号，注释和空白字符的语义动作直接调用`step`，因为它们并不需要从词法分析器中返回，而前一个`step`的调用只是在yylex返回后的再次调用时才会发生。

最后一个匹配所有可能的规则会打印出一条错误消息。原来C版的词法分析器调用`yyerror`来实现，但由于现在的词法分析器并不属于C++语法分析器类，所以它无法直接调用`error`例程。我们并没有试图编写一个糅合例程来使得各种程序都能够调用相同的错误报告例程，简单起见，我们直接调用了`printf`。

你是否应该用C++来实现你的语法分析器？

显而易见的是，`bison`对C++的支持在很多地方都不如对C的支持那么成熟，我们不需要感到惊讶，因为C语言版的已经有30年的历史了。我们需要一些额外的工作来解决`%union`不能包含类实例的问题，而目前C++版的`bison`和C版的`flex`之间无法做到无缝集成，这意味着我们在编程时需要加倍小心，特别是当它们需要共享一些重要的数据以便于词法分析器可以在C中访问而语法分析器可以在C++中访问时，又或者是当词法分析器需要使用C的标准库来读取输入而程序的其他部分使用C++的I/O库来读取时。一个好的对象设计方法是把应用上下文（本例中的`ctx`）、语法分析器甚至词法分析器封装在一个类中，这样可以为程序的其他部分提供一个统一的接口。

尽管如此，C++版的bison语法分析器依然可以工作，语法分析器类的设计也是合理的。如果你正在把你的语法分析器集成到一个大型的C++项目中，或者你希望使用没有相应C语言实现的C++类库，C++版的语法分析器照样可以很好地工作。

Java和其他语言支持

目前（2009年）bison正在尝试支持Java语言版的语法分析器。当你读到这本书的时候，它可能已经可以支持更多的语言了。对于Java的支持建立在它对C++的支持上，以及一些针对Java环境所做的调整，因为Java没有预处理器和联合，但它却有垃圾收集机制（garbage collection）。由于实现的细节有可能发生变化，所以请参考最新的bison手册来了解当前的Java接口。

练习

1. 修改纯模式计算器中的语法分析器，使它每次分析一条语句然后返回但不使用YYACCEPT。你可能需要更改词法分析器，使它在文件结束时返回一个零记号而不是基于EOL的判定。
2. SQL语法分析器的GLR版接受的语言是否与原版一致？请给出一个例子，它可以被一种版本支持而另一种则不支持。（提示：你可以试着在记号之间加入注释，通常记号之间只有一个空格。）



SQL语法分析器文法 和交叉引用

由于第4章中SQL语法分析器的文法太长，这里基于它们在源文件中出现的顺序重新列出了所有的规则，此外还包括每个记号和非终结符，以及使用它们的规则的交叉引用信息。该交叉引用是基于第8章的增强版本，它带有错误恢复规则。

例A-1中的列表和例子A-2中的交叉引用是从列表文件lpmysql.ouput中摘录出来的，该列表文件在bison编译SQL语法时创建。这个列表文件还包括一些语法没有使用到的记号，因为有些SQL关键字并没有在我们分析的子集里用到，另外还包括语法分析器状态和移进与归约动作的完整集合。整个列表超过10 000行，所以不适合包含在本书中，但在调试语法时它的参考作用是无价的。

例A-1：SQL语法列表

```
0 $accept: stmt_list $end

1 stmt_list: stmt ';'
2           | stmt_list stmt ';'
3           | error ';'
4           | stmt_list error ';'

5 stmt: select_stmt

6 select_stmt: SELECT select_opts select_expr_list
7           | SELECT select_opts select_expr_list FROM table_references opt_where
8           | opt_groupby opt_having opt_orderby opt_limit opt_into_list

8 opt_where: /* 空 */
9           | WHERE expr
10 opt_groupby: /* 空 */
11           | GROUP BY groupby_list opt_with_rollup
12 groupby_list: expr opt_asc_desc
13           | groupby_list ',' expr opt_asc_desc

14 opt_asc_desc: /* 空 */
```

```
15          | ASC
16          | DESC

17 opt_with_rollup: /* 空 */
18          | WITH ROLLUP

19 opt_having: /* 空 */
20          | HAVING expr

21 opt_orderby: /* 空 */
22          | ORDER BY groupby_list

23 opt_limit: /* 空 */
24          | LIMIT expr
25          | LIMIT expr ',' expr

26 opt_into_list: /* 空 */
27          | INTO column_list

28 column_list: NAME
29          | STRING
30          | column_list ',' NAME
31          | column_list ',' STRING

32 select_opts: /* 空 */
33          | select_opts ALL
34          | select_opts DISTINCT
35          | select_opts DISTINCTROW
36          | select_opts HIGH_PRIORITY
37          | select_opts STRAIGHT_JOIN
38          | select_opts SQL_SMALL_RESULT
39          | select_opts SQL_BIG_RESULT
40          | select_opts SQL_CALC_FOUND_ROWS

41 select_expr_list: select_expr
42          | select_expr_list ',' select_expr
43          | '*'

44 select_expr: expr opt_as_alias

45 table_references: table_reference
46          | table_references ',' table_reference

47 table_reference: table_factor
48          | join_table

49 table_factor: NAME opt_as_alias index_hint
50          | NAME '.' NAME opt_as_alias index_hint
51          | table_subquery opt_as NAME
52          | '(' table_references ')'

53 opt_as: AS
54          | /* 空 */

55 opt_as_alias: AS NAME
```

```

56      | NAME
57      | /* 空 */

58 join_table: table_reference opt_inner_cross JOIN table_factor opt_join_condition
59      | table_reference STRAIGHT_JOIN table_factor
60      | table_reference STRAIGHT_JOIN table_factor ON expr
61      | table_reference left_or_right opt_outer JOIN table_factor
       | join_condition
62      | table_reference NATURAL opt_left_or_right_outer JOIN table_factor

63 opt_inner_cross: /* 空 */
64      | INNER
65      | CROSS

66 opt_outer: /* 空 */
67      | OUTER

68 left_or_right: LEFT
69      | RIGHT

70 opt_left_or_right_outer: LEFT opt_outer
71      | RIGHT opt_outer
72      | /* 空 */

73 opt_join_condition: join_condition
74      | /* 空 */

75 join_condition: ON expr
76      | USING '(' column_list ')'

77 index_hint: USE KEY opt_for_join '(' index_list ')'
78      | IGNORE KEY opt_for_join '(' index_list ')'
79 | FORCE KEY opt_for_join '(' index_list ')'
80      | /* 空 */

81 opt_for_join: FOR JOIN
82      | /* 空 */

83 index_list: NAME
84      | index_list ',' NAME

85 table_subquery: '(' select_stmt ')'

86 stmt: delete_stmt

87 delete_stmt: DELETE delete_opts FROM NAME opt_where opt_orderby opt_limit

88 delete_opts: delete_opts LOW_PRIORITY
89      | delete_opts QUICK
90      | delete_opts IGNORE
91      | /* 空 */

92 delete_stmt: DELETE delete_opts delete_list FROM table_references opt_where

93 delete_list: NAME opt_dot_star

```

```
94           | delete_list ',' NAME opt_dot_star
95 opt_dot_star: /* 空 */
96           | '.' '*'
97 delete_stmt: DELETE delete_opts FROM delete_list USING table_references
              opt_where
98 stmt: insert_stmt
99 insert_stmt: INSERT insert_opts opt_into NAME opt_col_names VALUES
               insert_vals_list opt_onduupdate
100 opt_onduupdate: /* 空 */
101           | ONDUPLICATE KEY UPDATE insert_asgn_list
102 insert_opts: /* 空 */
103           | insert_opts LOW_PRIORITY
104           | insert_opts DELAYED
105           | insert_opts HIGH_PRIORITY
106           | insert_opts IGNORE
107 opt_into: INTO
108 | /* 空 */
109 opt_col_names: /* 空 */
110           | '(' column_list ')'
111 insert_vals_list: '(' insert_vals ')'
112           | insert_vals_list ',' '(' insert_vals ')'
113 insert_vals: expr
114           | DEFAULT
115           | insert_vals ',' expr
116           | insert_vals ',' DEFAULT
117 insert_stmt: INSERT insert_opts opt_into NAME SET insert_asgn_list
               opt_onduupdate
118           | INSERT insert_opts opt_into NAME opt_col_names select_stmt
               opt_onduupdate
119 insert_asgn_list: NAME COMPARISON expr
120           | NAME COMPARISON DEFAULT
121           | insert_asgn_list ',' NAME COMPARISON expr
122           | insert_asgn_list ',' NAME COMPARISON DEFAULT
123 stmt: replace_stmt
124 replace_stmt: REPLACE insert_opts opt_into NAME opt_col_names VALUES
                 insert_vals_list opt_onduupdate
125           | REPLACE insert_opts opt_into NAME SET insert_asgn_list
                 opt_onduupdate
126           | REPLACE insert_opts opt_into NAME opt_col_names select_stmt
                 opt_onduupdate
```

```

127 stmt: update_stmt

128 update_stmt: UPDATE update_opts table_references SET update_asgn_list opt_where
    opt_orderby opt_limit

129 update_opts: /* 空 */
130     | insert_opts LOW_PRIORITY
131     | insert_opts IGNORE

132 update_asgn_list: NAME COMPARISON expr
133             | NAME '.' NAME COMPARISON expr
134             | update_asgn_list ',' NAME COMPARISON expr
135             | update_asgn_list ',' NAME '.' NAME COMPARISON expr

136 stmt: create_database_stmt

137 create_database_stmt: CREATE DATABASE opt_if_not_exists NAME
138 | CREATE SCHEMA opt_if_not_exists NAME

139 opt_if_not_exists: /* 空 */
140 | IF EXISTS

141 stmt: create_table_stmt

142 create_table_stmt: CREATE opt_temporary TABLE opt_if_not_exists
    NAME '(' create_col_list ')'
143     | CREATE opt_temporary TABLE opt_if_not_exists NAME '.' NAME '(' create_col_list ')'
144     | CREATE opt_temporary TABLE opt_if_not_exists
        NAME '(' create_col_list ')' create_select_statement
145     | CREATE opt_temporary TABLE opt_if_not_exists
        NAME create_select_statement
146     | CREATE opt_temporary TABLE opt_if_not_exists NAME '.' NAME '(' create_col_list ')' create_select_statement
147     | CREATE opt_temporary TABLE opt_if_not_exists NAME '.' NAME create_select_statement

148 create_col_list: create_definition
149         | create_col_list ',' create_definition

150 $@1: /* 空 */

151 create_definition: $@1 NAME data_type column_atts
152         | PRIMARY KEY '(' column_list ')'
153         | KEY '(' column_list ')'
154         | INDEX '(' column_list ')'
155         | FULLTEXT INDEX '(' column_list ')'
156         | FULLTEXT KEY '(' column_list ')'

157 column_atts: /* 空 */
158     | column_atts NOT NULLX
159     | column_atts NULLX
160     | column_atts DEFAULT STRING
161     | column_atts DEFAULT INTNUM
162     | column_atts DEFAULT APPROXNUM

```

```
163      | column_atts DEFAULT BOOL
164      | column_atts AUTO_INCREMENT
165      | column_atts UNIQUE '(' column_list ')'
166      | column_atts UNIQUE KEY
167      | column_atts PRIMARY KEY
168      | column_atts KEY
169      | column_atts COMMENT STRING

170 opt_length: /* 空 */
171      | '(' INTNUM ')'
172      | '(' INTNUM ',' INTNUM ')'

173 opt_binary: /* 空 */
174      | BINARY

175 opt_uz: /* 空 */
176      | opt_uz UNSIGNED
177      | opt_uz ZEROFILL

178 opt_csc: /* 空 */
179      | opt_csc CHAR SET STRING
180      | opt_csc COLLATE STRING

181 data_type: BIT opt_length
182      | TINYINT opt_length opt_uz
183      | SMALLINT opt_length opt_uz
184      | MEDIUMINT opt_length opt_uz
185      | INT opt_length opt_uz
186      | INTEGER opt_length opt_uz
187      | BIGINT opt_length opt_uz
188      | REAL opt_length opt_uz
189      | DOUBLE opt_length opt_uz
190      | FLOAT opt_length opt_uz
191      | DECIMAL opt_length opt_uz
192      | DATE
193      | TIME
194      | TIMESTAMP
195      | DATETIME
196      | YEAR
197      | CHAR opt_length opt_csc
198      | VARCHAR '(' INTNUM ')' opt_csc
199      | BINARY opt_length
200      | VARBINARY '(' INTNUM ')'
201      | TINYBLOB
202      | BLOB
203      | MEDIUMBLOB
204      | LONGBLOB
205      | TINYTEXT opt_binary opt_csc
206      | TEXT opt_binary opt_csc
207      | MEDIUMTEXT opt_binary opt_csc
208      | LONGTEXT opt_binary opt_csc
209      | ENUM '(' enum_list ')' opt_csc
210      | SET '(' enum_list ')' opt_csc

211 enum_list: STRING
```

```
212           | enum_list ',' STRING
213 create_select_statement: opt_ignore_replace opt_as select_stmt
214 opt_ignore_replace: /* 空 */
215           | IGNORE
216           | REPLACE
217 opt_temporary: /* 空 */
218           | TEMPORARY
219 stmt: set_stmt
220 set_stmt: SET set_list
221 set_list: set_expr
222           | set_list ',' set_expr
223 set_expr: USERVAR COMPARISON expr
224           | USERVAR ASSIGN expr
225 expr: NAME
226           | USERVAR
227           | NAME '.' NAME
228           | STRING
229           | INTNUM
230           | APPROXNUM
231           | BOOL
232           | expr '+' expr
233           | expr '-' expr
234           | expr '*' expr
235           | expr '/' expr
236           | expr '%' expr
237           | expr MOD expr
238           | '-' expr
239           | expr ANDOP expr
240           | expr OR expr
241           | expr XOR expr
242           | expr COMPARISON expr
243           | expr COMPARISON '(' select_stmt ')'
244           | expr COMPARISON ANY '(' select_stmt ')'
245           | expr COMPARISON SOME '(' select_stmt ')'
246           | expr COMPARISON ALL '(' select_stmt ')'
247           | expr '|' expr
248           | expr '&' expr
249           | expr '^' expr
250           | expr SHIFT expr
251           | NOT expr
252           | '!' expr
253           | USERVAR ASSIGN expr
254           | expr IS NULLX
255           | expr IS NOT NULLX
256           | expr IS BOOL
257           | expr IS NOT BOOL
```

```
258     | expr BETWEEN expr AND expr
259 val_list: expr
260         | expr ',' val_list
261 opt_val_list: /* 空 */
262         | val_list
263 expr: expr IN '(' val_list ')'
264     | expr NOT IN '(' val_list ')'
265     | expr IN '(' select_stmt ')'
266     | expr NOT IN '(' select_stmt ')'
267     | EXISTS '(' select_stmt ')'
268     | NAME '(' opt_val_list ')'
269     | FCOUNT '(' '*' ')'
270     | FCOUNT '(' expr ')'
271     | FSUBSTRING '(' val_list ')'
272     | FSUBSTRING '(' expr FROM expr ')'
273     | FSUBSTRING '(' expr FROM expr FOR expr ')'
274     | FTRIM '(' val_list ')'
275     | FTRIM '(' trim_ltb expr FROM val_list ')'
276 trim_ltb: LEADING
277         | TRAILING
278         | BOTH
279 expr: FDATE_ADD '(' expr ',' interval_exp ')'
280     | FDATE_SUB '(' expr ',' interval_exp ')'
281 interval_exp: INTERVAL expr DAY_HOUR
282             | INTERVAL expr DAY_MICROSECOND
283             | INTERVAL expr DAY_MINUTE
284             | INTERVAL expr DAY_SECOND
285             | INTERVAL expr YEAR_MONTH
286             | INTERVAL expr YEAR
287             | INTERVAL expr HOUR_MICROSECOND
288             | INTERVAL expr HOUR_MINUTE
289             | INTERVAL expr HOUR_SECOND
290 expr: CASE expr case_list END
291     | CASE expr case_list ELSE expr END
292     | CASE case_list END
293     | CASE case_list ELSE expr END
294 case_list: WHEN expr THEN expr
295         | case_list WHEN expr THEN expr
296 expr: expr LIKE expr
297     | expr NOT LIKE expr
298     | expr REGEXP expr
299     | expr NOT REGEXP expr
300     | CURRENT_TIMESTAMP
301     | CURRENT_DATE
302     | CURRENT_TIME
303     | BINARY expr
```

例子A-2：SQL语法终结符交叉引用

终结符，以及使用它们的规则

```
$end (0) 0
'!' (33) 252
'%' (37) 236
'&' (38) 248
'(' (40) 52 76 77 78 79 85 110 111 112 142 143 144 146 152 153 154
    155 156 165 171 172 198 200 209 210 243 244 245 246 263 264 265
    266 267 268 269 270 271 272 273 274 275 279 280
')' (41) 52 76 77 78 79 85 110 111 112 142 143 144 146 152 153 154
    155 156 165 171 172 198 200 209 210 243 244 245 246 263 264 265
    266 267 268 269 270 271 272 273 274 275 279 280
'*' (42) 43 96 234 269
'+' (43) 232
',' (44) 13 25 30 31 42 46 84 94 112 115 116 121 122 134 135 149 172
    212 222 260 279 280
'-' (45) 233 238
'.' (46) 50 96 133 135 143 146 147 227
'/' (47) 235
';' (59) 1 2 3 4
'^' (94) 249
'||' (124) 247
error (256) 3 4
NAME (258) 28 30 49 50 51 55 56 83 84 87 93 94 99 117 118 119 120 121
    122 124 125 126 132 133 134 135 137 138 142 143 144 145 146 147
    151 225 227 268
STRING (259) 29 31 160 169 179 180 211 212 228
INTNUM (260) 161 171 172 198 200 229
BOOL (261) 163 231 256 257
APPROXNUM (262) 162 230
USERVAR (263) 223 224 226 253
ASSIGN (264) 224 253
OR (265) 240
XOR (266) 241
ANDOP (267) 239
REGEXP (268) 298 299
LIKE (269) 296 297
IS (270) 254 255 256 257
IN (271) 263 264 265 266
NOT (272) 158 251 255 257 264 266 297 299
BETWEEN (273) 258
COMPARISON (274) 119 120 121 122 132 133 134 135 223 242 243 244 245
    246
SHIFT (275) 250
MOD (276) 237
UMINUS (277)
ADD (278)
ALL (279) 33 246
ALTER (280)
ANALYZE (281)
AND (282) 258
ANY (283) 244
AS (284) 53 55
```

ASC (285) 15
AUTO_INCREMENT (286) 164
BEFORE (287)
BIGINT (288) 187
BINARY (289) 174 199 303
BIT (290) 181
BLOB (291) 202
BOTH (292) 278
BY (293) 11 22
CALL (294)
CASCADE (295)
CASE (296) 290 291 292 293
CHANGE (297)
CHAR (298) 179 197
CHECK (299)
COLLATE (300) 180
COLUMN (301)
COMMENT (302) 169
CONDITION (303)
CONSTRAINT (304)
CONTINUE (305)
CONVERT (306)
CREATE (307) 137 138 142 143 144 145 146 147
CROSS (308) 65
CURRENT_DATE (309) 301
CURRENT_TIME (310) 302
CURRENT_TIMESTAMP (311) 300
CURRENT_USER (312)
CURSOR (313)
DATABASE (314) 137
DATABASES (315)
DATE (316) 192
DATETIME (317) 195
DAY_HOUR (318) 281
DAY_MICROSECOND (319) 282
DAY_MINUTE (320) 283
DAY_SECOND (321) 284
DECIMAL (322) 191
DECLARE (323)
DEFAULT (324) 114 116 120 122 160 161 162 163
DELAYED (325) 104
DELETE (326) 87 92 97
DESC (327) 16
DESCRIBE (328)
DETERMINISTIC (329)
DISTINCT (330) 34
DISTINCTROW (331) 35
DIV (332)
DOUBLE (333) 189
DROP (334)
DUAL (335)
EACH (336)
ELSE (337) 291 293
ELSEIF (338)
ENCLOSED (339)

END (340) 290 291 292 293
ENUM (341) 209
ESCAPED (342)
EXISTS (343) 140 267
EXIT (344)
EXPLAIN (345)
FETCH (346)
FLOAT (347) 190
FOR (348) 81 273
FORCE (349) 79
FOREIGN (350)
FROM (351) 7 87 92 97 272 273 275
FULLTEXT (352) 155 156
GRANT (353)
GROUP (354) 11
HAVING (355) 20
HIGH_PRIORITY (356) 36 105
HOUR_MICROSECOND (357) 287
HOUR_MINUTE (358) 288
HOUR_SECOND (359) 289
IF (360) 140
IGNORE (361) 78 90 106 131 215
INDEX (362) 154 155
INFILE (363)
INNER (364) 64
INOUT (365)
INSENSITIVE (366)
INSERT (367) 99 117 118
INT (368) 185
INTEGER (369) 186
INTERVAL (370) 281 282 283 284 285 286 287 288 289
INTO (371) 27 107
ITERATE (372)
JOIN (373) 58 61 62 81
KEY (374) 77 78 79 101 152 153 156 166 167 168
KEYS (375)
KILL (376)
LEADING (377) 276
LEAVE (378)
LEFT (379) 68 70
LIMIT (380) 24 25
LINES (381)
LOAD (382)
LOCALTIME (383)
LOCALTIMESTAMP (384)
LOCK (385)
LONG (386)
LONGBLOB (387) 204
LONGTEXT (388) 208
LOOP (389)
LOW_PRIORITY (390) 88 103 130
MATCH (391)
MEDIUMBLOB (392) 203
MEDIUMINT (393) 184
MEDIUMTEXT (394) 207

MINUTE_MICROSECOND (395)
MINUTE_SECOND (396)
MODIFIES (397)
NATURAL (398) 62
NO_WRITE_TO_BINLOG (399)
NULLX (400) 158 159 254 255
NUMBER (401)
ON (402) 60 75
ONDUPLOCATE (403) 101
OPTIMIZE (404)
OPTION (405)
OPTIONALLY (406)
ORDER (407) 22
OUT (408)
OUTER (409) 67
OUTFILE (410)
PRECISION (411)
PRIMARY (412) 152 167
PROCEDURE (413)
PURGE (414)
QUICK (415) 89
READ (416)
READS (417)
REAL (418) 188
REFERENCES (419)
RELEASE (420)
RENAME (421)
REPEAT (422)
REPLACE (423) 124 125 126 216
REQUIRE (424)
RESTRICT (425)
RETURN (426)
REVOKE (427)
RIGHT (428) 69 71
ROLLUP (429) 18
SCHEMA (430) 138
SCHEMAS (431)
SECOND_MICROSECOND (432)
SELECT (433) 6 7
SENSITIVE (434)
SEPARATOR (435)
SET (436) 117 125 128 179 210 220
SHOW (437)
SMALLINT (438) 183
SOME (439) 245
SONAME (440)
SPATIAL (441)
SPECIFIC (442)
SQL (443)
SQLEXCEPTION (444)
SQLSTATE (445)
SQLWARNING (446)
SQL_BIG_RESULT (447) 39
SQL_CALC_FOUND_ROWS (448) 40
SQL_SMALL_RESULT (449) 38

SSL (450)
STARTING (451)
STRAIGHT_JOIN (452) 37 59 60
TABLE (453) 142 143 144 145 146 147
TEMPORARY (454) 218
TEXT (455) 206
TERMINATED (456)
THEN (457) 294 295
TIME (458) 193
TIMESTAMP (459) 194
TINYBLOB (460) 201
TINYINT (461) 182
TINYTEXT (462) 205
TO (463)
TRAILING (464) 277
TRIGGER (465)
UNDO (466)
UNION (467)
UNIQUE (468) 165 166
UNLOCK (469)
UNSIGNED (470) 176
UPDATE (471) 101 128
USAGE (472)
USE (473) 77
USING (474) 76 97
UTC_DATE (475)
UTC_TIME (476)
UTC_TIMESTAMP (477)
VALUES (478) 99 124
VARBINARY (479) 200
VARCHAR (480) 198
VARYING (481)
WHEN (482) 294 295
WHERE (483) 9
WHILE (484)
WITH (485) 18
WRITE (486)
YEAR (487) 196 286
YEAR_MONTH (488) 285
ZEROFILL (489) 177
FSUBSTRING (490) 271 272 273
FTRIM (491) 274 275
FDATE_ADD (492) 279
FDATE_SUB (493) 280
FCOUNT (494) 269 270

非终结符，以及使用它们的规则

```
$accept (254)
  on left: 0
stmt_list (255)
  on left: 1 2 3 4, on right: 0 2 4
stmt (256)
  on left: 5 86 98 123 127 136 141 219, on right: 1 2
select_stmt (257)
```

```
on left: 6 7, on right: 5 85 118 126 213 243 244 245 246 265 266
267
opt_where (258)
    on left: 8 9, on right: 7 87 92 97 128
opt_groupby (259)
    on left: 10 11, on right: 7
groupby_list (260)
    on left: 12 13, on right: 11 13 22
opt_asc_desc (261)
    on left: 14 15 16, on right: 12 13
opt_with_rollup (262)
    on left: 17 18, on right: 11
opt_having (263)
    on left: 19 20, on right: 7
opt_orderby (264)
    on left: 21 22, on right: 7 87 128
opt_limit (265)
    on left: 23 24 25, on right: 7 87 128
opt_into_list (266)
    on left: 26 27, on right: 7
column_list (267)
    on left: 28 29 30 31, on right: 27 30 31 76 110 152 153 154 155
156 165
select_opts (268)
    on left: 32 33 34 35 36 37 38 39 40, on right: 6 7 33 34 35 36
37 38 39 40
select_expr_list (269)
    on left: 41 42 43, on right: 6 7 42
select_expr (270)
    on left: 44, on right: 41 42
table_references (271)
    on left: 45 46, on right: 7 46 52 92 97 128
table_reference (272)
    on left: 47 48, on right: 45 46 58 59 60 61 62
table_factor (273)
    on left: 49 50 51 52, on right: 47 58 59 60 61 62
opt_as (274)
    on left: 53 54, on right: 51 213
opt_as_alias (275)
    on left: 55 56 57, on right: 44 49 50
join_table (276)
    on left: 58 59 60 61 62, on right: 48
opt_inner_cross (277)
    on left: 63 64 65, on right: 58
opt_outer (278)
    on left: 66 67, on right: 61 70 71
left_or_right (279)
    on left: 68 69, on right: 61
opt_left_or_right_outer (280)
    on left: 70 71 72, on right: 62
opt_join_condition (281)
    on left: 73 74, on right: 58
join_condition (282)
    on left: 75 76, on right: 61 73
index_hint (283)
```

```
    on left: 77 78 79 80, on right: 49 50
opt_for_join (284)
    on left: 81 82, on right: 77 78 79
index_list (285)
    on left: 83 84, on right: 77 78 79 84
table_subquery (286)
    on left: 85, on right: 51
delete_stmt (287)
    on left: 87 92 97, on right: 86
delete_opts (288)
    on left: 88 89 90 91, on right: 87 88 89 90 92 97
delete_list (289)
    on left: 93 94, on right: 92 94 97
opt_dot_star (290)
    on left: 95 96, on right: 93 94
insert_stmt (291)
    on left: 99 117 118, on right: 98
opt_ondupupdate (292)
    on left: 100 101, on right: 99 117 118 124 125 126
insert_opts (293)
    on left: 102 103 104 105 106, on right: 99 103 104 105 106 117
    118 124 125 126 130 131
opt_into (294)
    on left: 107 108, on right: 99 117 118 124 125 126
opt_col_names (295)
    on left: 109 110, on right: 99 118 124 126
insert_vals_list (296)
    on left: 111 112, on right: 99 112 124
insert_vals (297)
    on left: 113 114 115 116, on right: 111 112 115 116
insert_asgn_list (298)
    on left: 119 120 121 122, on right: 101 117 121 122 125
replace_stmt (299)
    on left: 124 125 126, on right: 123
update_stmt (300)
    on left: 128, on right: 127
update_opts (301)
    on left: 129 130 131, on right: 128
update_asgn_list (302)
    on left: 132 133 134 135, on right: 128 134 135
create_database_stmt (303)
    on left: 137 138, on right: 136
opt_if_not_exists (304)
    on left: 139 140, on right: 137 138 142 143 144 145 146 147
create_table_stmt (305)
    on left: 142 143 144 145 146 147, on right: 141
create_col_list (306)
    on left: 148 149, on right: 142 143 144 146 149
create_definition (307)
    on left: 151 152 153 154 155 156, on right: 148 149
$@1 (308)
    on left: 150, on right: 151
column_atts (309)
    on left: 157 158 159 160 161 162 163 164 165 166 167 168 169, on right:
    151 158 159 160 161 162 163 164 165 166 167 168 169
```

```
opt_length (310)
    on left: 170 171 172, on right: 181 182 183 184 185 186 187 188
    189 190 191 197 199
opt_binary (311)
    on left: 173 174, on right: 205 206 207 208
opt_uz (312)
    on left: 175 176 177, on right: 176 177 182 183 184 185 186 187
    188 189 190 191
opt_csc (313)
    on left: 178 179 180, on right: 179 180 197 198 205 206 207 208
    209 210
data_type (314)
    on left: 181 182 183 184 185 186 187 188 189 190 191 192 193 194
    195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210,
    on right: 151
enum_list (315)
    on left: 211 212, on right: 209 210 212
create_select_statement (316)
    on left: 213, on right: 144 145 146 147
opt_ignore_replace (317)
    on left: 214 215 216, on right: 213
opt_temporary (318)
    on left: 217 218, on right: 142 143 144 145 146 147
set_stmt (319)
    on left: 220, on right: 219
set_list (320)
    on left: 221 222, on right: 220 222
set_expr (321)
    on left: 223 224, on right: 221 222
expr (322)
    on left: 225 226 227 228 229 230 231 232 233 234 235 236 237 238
    239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254
    255 256 257 258 263 264 265 266 267 268 269 270 271 272 273 274
    275 279 280 290 291 292 293 296 297 298 299 300 301 302 303, on right:
    9 12 13 20 24 25 44 60 75 113 115 119 121 132 133 134 135 223 224
    232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247
    248 249 250 251 252 253 254 255 256 257 258 259 260 263 264 265
    266 270 272 273 275 279 280 281 282 283 284 285 286 287 288 289
    290 291 293 294 295 296 297 298 299 303
val_list (323)
    on left: 259 260, on right: 260 262 263 264 271 274 275
opt_val_list (324)
    on left: 261 262, on right: 268
trim_ltb (325)
    on left: 276 277 278, on right: 275
interval_exp (326)
    on left: 281 282 283 284 285 286 287 288 289, on right: 279 280
case_list (327)
    on left: 294 295, on right: 290.291 292 293 295
```

术语表

动作 (action)

flex的模式或者bison的规则所关联的C/C++代码。当模式或者规则能够匹配输入串时，该动作会被执行。

字母表 (alphabet)

不同符号的集合。例如，ASCII字符集合包括128个不同的符号。在flex规范中，字母表是计算机的本地字符集。在bison语法中，字母表是语法中所使用的记号和非终结符的集合。

二义性 (ambiguity)

一个二义性语法是指它有多条规则可以匹配相同输入。在bison语法中，二义性规则将导致移进/归约冲突或者归约/归约冲突。bison常用的分析机制无法处理二义性语法。在创建语法分析器时，程序员可以使用%prec声明和一些bison自己的内部规则来解决冲突，也可以使用GLR语法分析器，它能够直接处理二义性语法。

ASCII

American Standard Code for Information Interchange (美国信息互换标准码)，包含了在美国字母表中常用的128个符号：小写和大写字母、数字和标点符号，以

及额外的格式化和数据通信链路控制的字符。大多数运行flex和bison的系统都使用ASCII或者IOS-8859系列中扩展的8位编码(ASCII是其中的一个子集)。

bison

把BNF表达式翻译成LALR(1)或者GLR语法分析器的程序。

BNF

Backus-Naur范式，表达上下文无关文法的一种方法。它常用来表示程序语言的正式语法。bison的输入语法是简化版的BNF。

编译器 (compiler)

把一种语言中的指令集（程序）翻译成其他的表达方式。通常编译器的输出就是能够在计算机上直接运行的本地二进制语言。参见解释器 (*interpreter*)。

冲突 (conflict)

bison语法中的一种错误，它发生在同一输入记号可能对应两个（或者更多）分析动作时。有两种不同的冲突：移进/归约冲突和归约/归约冲突。参见二义性 (*ambiguity*)。

上下文无关文法 (context-free grammar)

该文法中每条规则的左部只有一个符号，因此，右部符号只需要匹配输入，无需关心它所匹配内容的前后分别是什么。该文法也被称为短语结构文法 (*phrase structure grammar*)。上下文相关文法的规则允许左部出现多个符号，它并不适合用来分析计算机语言。

空串 (empty)

特殊的字符串，它包含零个符号，有时它被写成希腊文的第五个字母 (ϵ)。bison规则可以匹配一个空串，但是flex模式不行。

有限自动机 (finite automation)

由有限数量的指令（或者转换）组成的抽象机器。在模型化常见的计算机程序时，有限自动机十分有效，它有很多有用的数学特性。flex和bison基于有限自动机来创建词法分析器和语法分析器。

flex

产生词法分析器的程序，也被称为扫描器 (*scanner*)，它在字符流中匹配由正则表达式定义的模式。

GLR

Generalized Left to Right (通用的自左向右)，bison可以选择性使用的非常强大的分析技术。与LALR(1)不同的是，它能够分析那些具有二义性或者需要无限向前查看的语法，因为它可以维护由输入产生的所有可能分析。

语法 (grammar)

规则的集合，它们共同定义了一种语言。

输入 (input)

程序读取的数据流。例如，flex词法分析

器的输入是字节流，而语法分析器的输入则是记号流。

解释器 (interpreter)

一种程序，它读取一种语言（或者程序）的指令，然后每次解码并执行一条指令。参见编译器 (*compiler*)。

LALR(1)

Look Ahead Left to Right (自左向右查看)，bison常用的分析技术。(1)表明只允许向前查看一个记号。

语言 (language)

比较正式的定义是基于特定字母表上的定义明确的字符串集合。非正式的定义则是一个指令集，它描述了能够被计算机执行的任务。

左部 (LHS)

bison规则的左部是指在冒号之前的符号。在分析过程中，当输入可以匹配规则右部的一系列符号时，这组符号将被归约为左部符号。

lex

产生词法分析器的程序，匹配规则由正则表达式定义。现在已经被更加可靠和强大的flex取代。

词法分析器 (lexical analyzer)

能够将字符流转换成记号流的程序。flex使用正则表达式来描述每个记号，它把字符流分解为各个记号，然后确定每个记号的类型和值。例如，它会把字符流a=17；转化为由名字a、操作符=、数字17和单个字符记号；组成的记号流。

向前查看 (lookahead)

词法分析器或者语法分析器已经读取的输

入，但是还没有被任一模式或者规则匹配。bison语法分析器支持向前查看一个记号，而flex词法分析器可以无限制地向前查看。

非终结符号 (nonterminal)

bison语法中的符号，它并不出现在输入中，而仅仅由规则来定义。参见记号化 (*tokenizing*)。

语法分析器堆栈 (parser stack)

在bison语法分析器中，得到部分匹配的规则中的符号被保存在一个内部堆栈中。当语法分析器执行移进动作时，符号被加入堆栈；而归约动作被执行时，符号从堆栈中移去。

分析 (parsing)

在逻辑上把记号流分组为特定语言中的语句。

模式 (pattern)

在flex词法分析器中，模式就是词法分析器匹配输入所使用的正则表达式。

优先级 (precedence)

操作符执行的先后次序。例如，当解释数学语句时，乘法和除法拥有比加法和减法更高的优先级。因此，语句 $3+4*5$ 等于23而不是35。

产生式 (production)

参见规则 (*rule*)。

程序 (program)

完成指定任务的指令集。

归约 (reduce)

在bison语法分析器中，当输入可以匹配一条规则右部的所有符号时，语法分析器

将归约这条规则，把右部符号从语法分析器堆栈中移出，然后压入左部符号。

归约/归约冲突 (reduce/reduce conflict)

在bison语法中，当两个或者更多的规则可以匹配相同的记号串时，就形成归约/归约冲突。bison通过选择归约语法中更早出现的规则来解决这种冲突。

正则表达式 (regular expression)

用来指定匹配字符序列模式的语言。正则表达式由如下元素组成：普通字符，用来匹配输入中的相同字符；字符集，用来匹配该集合中的任一字符；其他特殊字符，用来指定特定匹配的表达式。

右部 (right-hand side)

bison规则的右部是跟在冒号后面的符号列表。在分析过程中，当输入可以匹配规则右部的符号序列时，该序列将被归约成左部符号。

规则 (rule)

在bison中，规则是语言的抽象表述。bison规则也被称为产生式 (*production*)。一条规则由被称为左部的单一非终结符、一个冒号和被称为右部的可能为空的符号列表组成。每当输入匹配到一条规则的右部时，语法分析器将归约这条规则。

语义 (semantic meaning)

参见值 (*value*)。

移进 (shift)

当bison语法分析器认为输入符号能够匹配语法中的某条规则时，它会移进输入符号，把它放到语法分析器堆栈中。

移进/归约冲突 (shift/reduce conflict)

在bison语法中，当一个符号既可以完成某条规则的右部（语法分析器执行归约动作），也可以作为其他规则右部的中间符号存在（语法分析器执行移进动作）时，就会形成移进/归约冲突。这种冲突可能是由于语法本身具有二义性，也可能是由于语法分析器需要向前查看更多记号来决定是否归约可以完成的规则。bison通过选择移进来解决这种冲突。

规范 (specification)

flex的规范是用来匹配输入流的模式集合。flex会把规范转化成词法分析器。

起始符号 (start)

bison语法分析器需要把有效的输入流最终归约到单一的符号，也就是起始符号。左部为起始符号的规则被称为起始规则。

起始状态 (start state)

在flex规范中，模式可以使用起始状态来标记。在任何时候都会有一种激活的起始状态，用该起始状态标记的模式可以参与匹配过程。在独占的起始状态中，只有该起始状态标记的模式才可以匹配输入，而在共享的起始状态中，没有任何状态标记的模式也可以匹配输入。

符号 (symbol)

在bison的术语里，符号可以是记号也可以是非终结符。在语法的规则中，规则右部的任何一个名字都是一个符号。

在bison的术语里，终结符，或者说记号 (token)，是由词法分析器提供给语法分析器的。而非终结符只是定义在语法分析器中。

符号表 (symbol table)

一种数据结构，它包含输入中出现的所有名字信息，这样相同名字的引用可以被关联到相同的对象。

记号化 (tokenizing)

把字符流转化为记号流的过程。词法分析器记号化它的输入。

值 (value)

在bison语法中的每个记号都有一个语法 (syntactic) 值和一个语义 (semantic) 值，它的语义值是该记号的真正的数据内容。例如，一个特定操作数的语法类型可能是INTEGER，但它的语义值可能是3。

yacc

Yet Another Compiler Compiler（另一种编译器的编译器），bison的前身，一种可以通过形如BNF的格式的规则列表来生成语法分析器的程序。