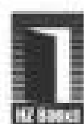
*lex & yacc*

第二版

# lex与yacc

O'REILLY®

机械工业出版社
China Machine Press



*John R. Levine, Tony Mason & Doug Brown* 著

杨作梅 张旭东 等译

UNIX Programming/UNIX Utilities

O'REILLY®

# lex与yacc

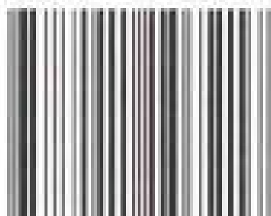
《lex与yacc》(第二版)是惟一一本专门介绍这两个重要的UNIX编程工具的书。这本新版本是完全的修订版，并以很多新的扩充示例代替了旧的示例。几个介绍性章节已经完全重写，还有一章专门介绍实现SQL语法，给出了有经验的程序员希望看到的各种细节。

本书对*lex*和*yacc*的重要主题提供了详尽的参考。对所有主要的*lex*和*yacc*的MS-DOS和UNIX版本，本书都进行了介绍，包括AT&T *lex*和*yacc*、Berkeley *yacc*、Berkeley/Gnu *flex*、Gnu bison、MKS *lex*和*yacc*、Abraxas PCYACC等等。

> "太棒了！我已经读完《lex与yacc》的第二版……总而言之，这是一件伟大的作品——这本书比第一版充实很多，详实而透彻。阅读的过程中，我经历了无数次的惊喜，心里总想'可惜了，他们已经错过了也许是细微而精妙的地方，我先做个注释'，然而随后在下一个句子或段落就看到对该问题的阐释。"

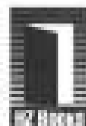
> "[John Levine] 做了大量的工作来完善这本书。我很高兴最终有一本好书推荐给人们。"

——Vern Paxson，flex的开发者



ISBN 7-111-10721-7

9 787111 107217 >



O'Reilly & Associates, Inc. 授权机械工业出版社出版

ISBN 7-111-10721-7

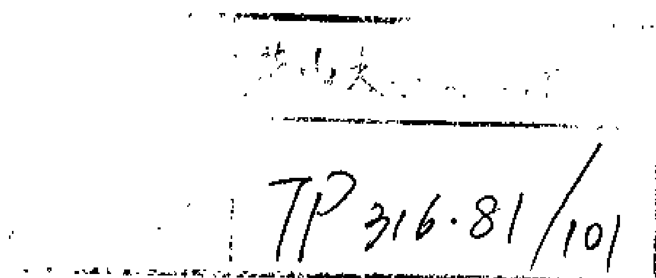
定价：45.00元

# lex与yacc

**第二版**

*John R. Levine, Tony Mason & Doug Brown* 著

**杨作梅 张旭东** 等译

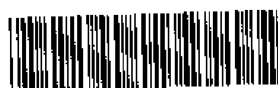



O'REILLY®

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

O'Reilly & Associates, Inc. 授权机械工业出版社出版

机械工业出版社



0354116

**图书在版编目（CIP）数据**

lex 与 yacc（第二版）/（美）莱弗恩（Levine, J. R.）等著；杨作梅等译．-北京：机械工业出版社，2003.1

书名原文：lex & yacc, Second Edition

ISBN 7-111-10721-7

I. l... II. ①莱... ②杨... III. UNIX操作系统－程序设计 IV. TP316.81

中国版本图书馆CIP数据核字（2002）第060101号

北京市版权局著作权合同登记

图字：01-2002-1829号

©1992 by O'Reilly & Associates, Inc.

Simplified Chinese Edition, jointly published by O'Reilly & Associates, Inc. and China Machine Press, 2003. Authorized translation of the English edition, 1992 O'Reilly & Associates, 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 & Associates, Inc. 出版 1992。

简体中文版由机械工业出版社出版 2003。英文原版的翻译得到 O'Reilly & Associates, 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 & Associates, 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。

书　　名／ lex 与 yacc（第二版）

书　　号／ ISBN 7-111-10721-7

责任编辑／ 刘立卿

封面设计／ Edie Freedman，张健

出版发行／ 机械工业出版社

地　　址／ 北京市西城区百万庄大街22号（邮政编码 100037）

经　　销／ 新华书店北京发行所发行

印　　刷／ 北京牛山世兴印刷厂

开　　本／ 787毫米×1092毫米　16开本　24.5印张　419千字

版　　次／ 2003年1月第1版　2003年1月第1次印刷

印　　数／ 0001-4000册

定　　价／ 45.00元

（凡购本书，如有倒页、脱页、缺页，由本社发行部调换）

# O'Reilly & Associates公司介绍

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 & Associates公司授权机械工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly & Associates公司是世界上在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二十世纪最重要的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 & Associates一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly & Associates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 & Associates公司具有深厚的计算机专业背景，这使得O'Reilly & Associates形成了一个非常不同于其他出版商的出版方针。O'Reilly & Associates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 & Associates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 & Associates依靠他们及时地推出图书。因为O'Reilly & Associates紧密地与计算机业界联系着，所以O'Reilly & Associates知道市场上真正需要什么图书。

# 目录


前言 ........ 1

第一章 lex和yacc ........ 9

最简单的lex程序 ........ 10

用lex识别单词 ........ 11

语法 ........ 23

运行lex和yacc ........ 31

lex和手写的词法分析程序 ........ 32

练习 ........ 35

第二章 使用lex ........ 36

正则表达式 ........ 37

单词计数程序 ........ 42

分析命令行 ........ 48

C源代码分析程序 ........ 55

小结 ........ 58

练习 ........ 59

## 第三章 使用 yacc ........ 60

语法 ........ 60
移进 / 归约分析 ........ 62
yacc 语法分析程序 ........ 65
词法分析程序 ........ 68
算术表达式和歧义性 ........ 69
变量和有类型的标记 ........ 74
符号表 ........ 77
函数和保留字 ........ 81
用 make 构建语法分析程序 ........ 88
小结 ........ 89
练习 ........ 89

## 第四章 菜单生成语言 ........ 91

MGL 的概述 ........ 91
开发 MGL ........ 93
构建 MGL ........ 103
屏幕处理 ........ 109
结束 ........ 112
MGL 代码示例 ........ 115
练习 ........ 120

## 第五章 分析 SQL ........ 122

SQL 的要点概述 ........ 123
语法检查程序 ........ 127
语法分析程序 ........ 133
嵌入式 SQL ........ 158
练习 ........ 163

第六章 lex 规范参考 .................................... 164

lex 规范的结构 .................................... 164

BEGIN .................................... 166

程序错误 .................................... 166

字符变换 .................................... 168

上下文相关 .................................... 169

定义（替换） .................................... 171

ECHO .................................... 172

包含操作（文件的逻辑嵌套） .................................... 172

从字符串中输入 .................................... 174

input() .................................... 176

内部表（%N 声明） .................................... 177

lex 库 .................................... 178

行号和 yylineno .................................... 179

文字块 .................................... 179

一个程序中的多个词法分析程序 .................................... 179

output() .................................... 184

lex 词法分析程序的可移植性 .................................... 185

正则表达式语法 .................................... 187

REJECT .................................... 190

从 yylex()中返回值 .................................... 190

起始状态 .................................... 191

unput() .................................... 193

yyinput()、yyoutput()、yyunput() .................................... 194

yyleng .................................... 194

yyless() .................................... 194

yylex() .................................... 195

yymore() .................................... 196

yytext .................................... 197

yywrap() .................................... 199

## 第七章 yacc 语法参考 ........ 200

yacc 语法的结构 ........ 200
动作 ........ 201
歧义和冲突 ........ 204
yacc 中的程序错误 ........ 205
结束标记 ........ 208
错误标记和错误恢复 ........ 208
%ident 声明 ........ 209
继承的属性（$0） ........ 209
词汇的反馈 ........ 211
文字块 ........ 212
文字标记 ........ 213
yacc 语法分析程序的可移植性 ........ 213
优先级、结合性和操作符声明 ........ 215
递归规则 ........ 217
规则 ........ 219
特殊字符 ........ 220
开始声明 ........ 221
符号值 ........ 222
标记 ........ 223
%type 声明 ........ 226
%union 声明 ........ 226
变体和多重语法 ........ 227
y.output 文件 ........ 232
yacc 库 ........ 233
YYABORT ........ 234
YYACCEPT ........ 235
YYBACKUP ........ 235
yyclearin ........ 235
yydebug 和 YYDEBUG ........ 236

yyerrok ........ 237
YYERROR ........ 237
yyerror() ........ 237
yyparse() ........ 238
YYRECOVERING() ........ 239

## 第八章 yacc 歧义和冲突 ........ 240

指针模型和冲突 ........ 240
冲突的普通示例 ........ 253
如何修复冲突 ........ 258
小结 ........ 266
练习 ........ 266

## 第九章 错误报告和恢复 ........ 267

错误报告 ........ 267
错误恢复 ........ 272
练习 ........ 276

## 附录一 AT&T lex ........ 279

## 附录二 AT&T yacc ........ 287

## 附录三 Berkeley yacc ........ 297

## 附录四 GNU bison ........ 303

## 附录五 flex ........ 306

## 附录六 MKS lex 和 yacc ........ 318

附录七 Abraxas lex 和 yacc ........ 320

附录八 POSIX lex 和 yacc ........ 322

附录九 MGL 编译程序代码 ........ 325

附录十 SQL 分析程序代码 ........ 341

参考文献 ........ 369

词汇表 ........ 371

# 前言

lex 和 yacc 是特意为编写编译程序和解释程序的人设计的工具，它对非编译程序编写人员所感兴趣的许多应用程序也非常有用。在输入中查找模式或者拥有输入或命令语言的任何应用程序都适于采用 lex 和 yacc。而且，它们允许快速应用程序原型设计，容易修改，而且程序的维护简单。为了激发读者的想像力，下面给出了几个用 lex 和 yacc 开发的应用：

- 台式计算器 *bc*。
- 工具 *eqn* 和 *pic*，用于数学公式和复杂图片的排版预处理器。
- *PCC* 和 *GCC*，PCC 是和许多 UNIX 系统一起使用的可移植 C 编译程序，GCC 是 GNU C 编译程序。
- 菜单编译程序。
- SQL 数据库语言语法检查程序。
- *lex* 程序本身。

## 第二版的新特点

在第二版中，我们已经做了大量的修正。主要变化包括：

- 完全重写的介绍性的第一章到第三章。
- 全新的第五章介绍了完整的SQL语法。
- 全新的详细的参考性的第六章和第七章。
- 包含了lex和yacc的所有主要的MS-DOS和UNIX版本的完整介绍，包括AT&T lex和yacc、Berkeley yacc、flex、GNU bison、MKS lex和yacc、Abraxas PCYACC。
- 包含了lex和yacc的新的POSIX 1003.2标准版本的介绍。

# 本书范围

第一章“lex和yacc”介绍了如何以及为什么使用lex和yacc来创建编译程序和解释程序，并且演示了一些小的lex和yacc应用程序。还介绍贯穿全书的基本术语。

第二章“使用lex”描述了如何使用lex。开发了几个lex应用程序，它们分别用于计算文件中单词的个数、分析程序命令开关和参数和计算C程序上的统计数据。

第三章“使用yacc”给出了使用lex和yacc开发功能完整的台式计算器的完整示例。

第四章“菜单生成语言”演示了如何使用lex和yacc开发菜单生成程序。

第五章“分析SQL”开发了针对完整的SQL关系数据库语言的分析程序。首先，我们将分析程序作为一个语法检查程序，然后将它扩展为内置于C程序中的SQL的简单预处理程序。

第六章“lex规范参考”和第七章“yacc语法参考”详细描述了lex和yacc程序设计人员可用的特性和选项。当开发新的lex和yacc应用程序时，这两章以及随后的两章为有经验的lex和yacc程序设计人员提供了丰富的技术信息。

第八章“yacc歧义和冲突”解释了yacc歧义和冲突，它们是造成yacc不能正确分析语法的原因。然后介绍了定位和纠正这些问题的方法。

第九章“错误报告和恢复”讨论了编译程序和解释程序设计人员定位、识别和报告编译程序输入中的错误的技术。

附录一“AT&t lex”描述了AT&T lex的命令行语法，报告的错误消息以及建议的解决方案。

附录二“AT&T yacc”描述了AT&T yacc的命令行语法，并且列出了由yacc报告的错误，提供了能导致这种错误的代码示例和建议的解决方案。

附录三“Berkeley yacc”描述了Berkeley yacc的命令行语法，以及随Berkeley UNIX 一起发布的、已广泛使用的免费版本的yacc，并且列出了由Berkeley yacc报告的错误及其建议的解决方案。

附录四“GNU bison”讨论了bison中的不同之处。bison是自由软件基金会实现的yacc。

附录五“flex”讨论了flex（广泛使用的免费版本的lex），列出了它与其他版本的区别，以及由flex报告的错误及其解决方案。

附录六“MKS lex和yacc”讨论了来自Mortice Kern Systems的lex和yacc的MS-DOS以及OS/2版本。

附录七“Abraxas lex和yacc”讨论了来自Abraxas Software的lex和yacc的MS-DOS以及OS/2版本PCYACC。

附录八“POSIX lex和yacc”讨论了由IEEE POSIX 1003.2标准定义的lex和yacc版本。

附录九“MGL编译程序代码”为第四章讨论的菜单生成语言编译程序提供了完整的源代码。

附录十"SQL分析程序代码"提供了第五章讨论的SQL分析程序的完整的源代码和交叉引用。

词汇表列出了关于语言和编译理论的技术术语。

参考文献列出了有关lex和yacc的其他文档,以及对编译程序设计有帮助的书籍。

我们假定读者都熟悉C语言,因为大多数示例都是用C、lex或yacc编写的,书中有些示例是用特殊目的语言开发的。

## lex和yacc的可用性

lex和yacc都是在20世纪70年代由贝尔实验室开发的。yacc的开发早于lex,它是由Stephen C.Johnson开发的。为了与yacc一起工作,Mike Lesk和Eric Schmidt开发了lex。自第7版UNIX以来,lex和yacc已经成为标准的UNIX实用程序。System V和老式的BSD版本使用原始的AT&T版本,而最新版本的BSD使用flex(见下文)和Berkeley yacc。开发者撰写的文章是lex和yacc的主要信息源。

自由软件基金会的GNU工程组发布了*bison*,即yacc的替代品;bison是由Robert Corbett和Richard Stallman编写的。Charles Donnelly和Richard Stallman编写的bison手册特别好,尤其是特殊特性参考部分。附录四论述了bison。

BSD和GNU工程组还发布了*flex*(快速词法分析发生器,*F*ast *Lex*ical Analyzer Generator),其参考页中称:"lex的重写纠正了那个工具的部分不足"。flex最初由Jef Poskanzer编写;Vern Paxson和Van Jacobson又对它进行了重大改善,而且Vern目前仍在维护它。附录五概述了有关flex的特定主题。

至少有两种版本的lex和yacc可用于MS-DOS和OS/2计算机。MKS(Mortice Kern Systems Inc.)是MKS工具包的出版商,他将lex和yacc作为一种独立的产品提出来以满足许多PC的C语言编译。MKS lex和yacc有一本非常好的手册。附录六概述了MKS lex和yacc。Abraxas Software公司发布了PCYACC,它是

另一种lex和yacc版本，该版本带有许多广泛使用的编程语言的示例分析程序。附录七概述了Abraxas的lex和yacc版本。

# 示例程序

本书的程序可以从UUNET上免费得到（指的是除了UUNET普通连接时间费用之外免费）。如果访问UUNET，就可以使用UUCP或FTP检索到源代码。对于UUCP，只需找到一种能直接访问UUNET的机器，并且键入下面的命令：

```
uucp uunet\!~/nutshell/lexyacc/progs.tar.Z yourhost\!~/yourname/
```

如果用Bourne shell（*sh*）代替C shell（*csh*），可以省略反斜线。等一段时间（一天或更长时间）后，文件就会出现在*/usr/spool/uucppublic/***yourname**目录下。如果你还没有账户，但是却想要一个账户接收电子邮件，则可以打电话703-204-8000与UUNET联系。

要使用*ftp*，就要找到一种能直接访问Internet的机器。下面是一个示例会话，黑体字表示命令。

```
% ftp ftp.oreilly.com
Connected to ftp.oreilly.com.
220 FTP server (Version 5.99 Wed May 23 14:40:19 EDT 1990) ready.
Name (ftp.oreilly.com:yourname): anonymous
331 Guest login ok, send ident as password.
Password: ambar@ora.com (use your user name and host here)
230 Guest login ok, access restrictions apply.
ftp> cd published/oreilly/nutshell/lexyacc
250 CWD command successful.
ftp> binary (you must specify binary transfer for compressed files)
200 Type set to I.
ftp> get progs.tar.Z
200 PORT command successful.
150 Opening BINARY mode data connection for progs.tar.Z.
226 Transfer complete.
ftp> quit
221 Goodbye.
%
```

该文件是一个压缩的tar档案文件。一旦得到这个档案文件就要进行解压缩，键入：

```
% zcat progs.tar.Z | tar xf -
```

System V系统需要换用下面的tar命令：

```
% zcat progs.tar.Z | tar xof -
```

# 排版约定

本书使用下列排版约定：

黑体（**Bold**）

用于语句、函数、标识符和程序名。

斜体（*Italic*）

在段落中表示文件、目录和命令名以及数据类型，当介绍新的术语和概念时也用斜体来强调。

等宽（`Constant Width`）

用在示例中表示文件的内容或命令的输出。

等宽黑体（**`Constant Bold`**）

用在示例中表示逐字键入的命令行和选项。

引号

用于标识解释性文本中的代码段。系统消息、记号和符号也用引号引起。

%

shell提示符。

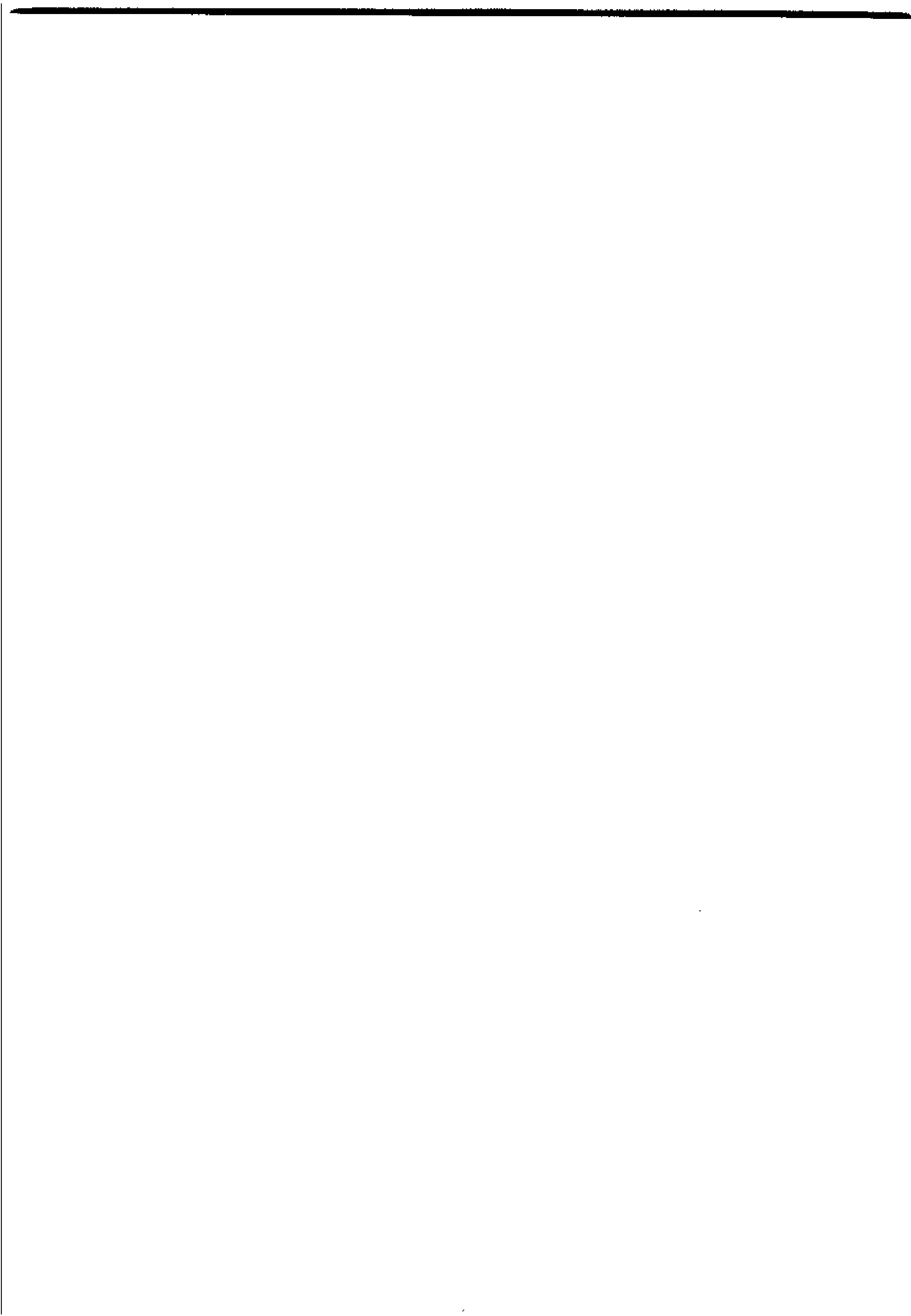
[ ]

在程序语法描述中用来表示可选的元素（不要键入括号本身）。

# 致谢

本书的第一版是基于Tony Mason的MGL和SGL编译程序编写的。Tony提供了本书的大多数素材，而一起工作的Dale Dougherty对其进行了总结。Doug Brown编写了第八章“yacc歧义和冲突”。Dale编排和修订了本书的部分内容。Tim O'Reilly为使本书成为一本更好的书，对该书进行了多次编校。感谢Butch Anton、Ed Engler和Mike Loukides对技术内容的注释。还要感谢John W.Lockhart阅读了草稿并查找文体上的问题。并且感谢Chris Reilley对本书插图所做的工作。最后，Ruth Terry辛勤地打印本书，并且光敏锐地检查每个编辑细节（虽然她努力地兼顾家庭及工作，但她还是为本书牺牲了许多照顾家人的时间）。

在第二版中，Tony重写了第一章和第二章；而Doug更新了第八章；John Levine编写了第三、五、六、七章及大多数附录，并且编辑了本书的其他部分。感谢技术评论者Bill Burke、Warren Carithers、Jon Mauney、Gary Merrill、Eugene Miya、Andy Oram和Bill Torcaso，尤其是Vern Paxson，他逐页给出了很好的、非常清晰的建议。Margaret Levine Young的蓝色铅笔（实际上是粉红色的）使文章更加紧凑，并保证了编辑的一致性。她还汇编了大部分索引。Chris Reilly再次处理了图形，而且Donna Woonteiler进行了最后的编辑，并引导本书通过生产流程。



本章内容：

- 最简单的lex程序
- 用lex识别单词
- 语法
- 运行lex和yacc
- lex和手写的词法分析程序
- 练习

# 第一章 lex和yacc

lex和yacc可以帮助你编写程序转换结构化输入。应用程序范围很广——既包括从输入文件中寻找模式的简单文本搜索程序，也包括将源程序变换为最佳的目标代码的C编译程序等。

在具有结构化输入的程序中，反复出现的两个任务是：将输入分隔成有意义的单元，然后找出这些单元之间的关系。对于文本搜索程序，这些单元可能是文本行，这些行包不包含目标匹配字符串存在着很大的区别。对于C程序，这些单元是变量名、常量、字符串、操作符、标点符号等。划分单元（通常称为标记）也称为词法分析（*lexical anslysis*，或者简写为*lexing*）。lex使用一系列对可能标记的描述，产生一个能识别那些标记的C例程（我们称为词法分析器，词法分析程序（lexer），或称为扫描程序）。赋予lex的那些描述称为*lex*规范。

lex使用的标记描述称为正则表达式，它是*grep*和*egrep*命令使用的常见模式的扩展版本。lex将这些正则表达式转变为词法分析程序能够用来极快地扫描输入文本的形式，而且速度不依赖于词法分析程序尝试匹配的表达式的数量。lex词法分析程序几乎总是比用C语言手工编写的词法分析程序快。

当将输入拆分成标记时，程序通常需要建立标记之间的关系。C编译程序需要找到程序中的表达式、语句、声明、程序块和过程。这个任务称为分析，即定义程

序能够理解的关系的规则列表，也就是语法。yacc采用简明的语法描述并产生一个能分析语法的C例程，即分析程序。yacc分析程序自动检测输入的标记序列是否匹配语法中的某条规则，并且一旦输入不匹配任一规则，它就会检测语法错误。yacc分析程序一般并不像手写的分析程序那样快，但它使编写和修改分析程序更容易，因此速度上的损失是值得的。一个程序在分析程序中用去的时间通常可以忽略不计。

当任务涉及将输入拆分成单元并且建立那些单元之间的关系时，应该考虑使用lex和yacc。(搜索程序很简单，它不需要做任何分析，所以它使用lex但不需要yacc。在第二章会再次提到，届时将只使用lex而不必用yacc来构建几个应用程序。)

至此，我们希望已经激起了读者的学习兴趣。这一章并不是lex和yacc的完整指南，我们只对lex和yacc的用法做一个初步的介绍。

# 最简单的lex程序

下面的lex程序将它的标准输入拷贝到标准输出：

```
%%
.|\n            ECHO;
%%
```

它的行为非常类似于不带参数运行的UNIX *cat* 命令。

lex自动生成实际的C程序代码，这些代码负责处理读输入文件，有时（正如上述情况下）也负责写输出。

使用lex和yacc无论是构建程序的一部分，还是构建辅助编程的工具，一旦你掌握了它们，就会发现它们的价值：处理问题时它们能简化输入的困难，提供更易维护的编码库，并且能很容易地“调整”出程序的正确语义。

# 用 lex 识别单词

让我们构建一个识别不同类型英语单词的简单程序。先标识词性（名词、动词等），然后再扩展到处理符合简单的英语语法的多个单词的句子。

先列出要识别的一组动词:

| | | | |
|---|---|---|---|
| is | am | are | were |
| was | be | being | been |
| do | does | did | will |
| would | should | can | could |
| has | have | had | go |

例 1-1 展示了识别这些动词的简单 lex 规范。

例 1-1：单词识别程序 ch1-02.l

```
%{
/*
 * 这个例子演示了（非常）简单的识别
 * 动词/非动词
 */

%}
%%

[\t ]+                  /* 忽略空白 */ ;

is |
am |
are |
were |
was |
be |
being |
been |
do |
does |
did |
will |
would |
```

```
should |
can |
could |
has |
have |
had |
go              { printf("%s: is a verb\n", yytext); }
[a-zA-Z]+       { printf("%s: is not a verb\n", yytext); }

.|\n            { ECHO; /* 通常的默认状态 */ }
%%

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

下面是编译和运行这个程序时进行的操作，键入的文字采用黑体字。

```
% example1
did I have fun?
did: is a verb
I: is not a verb
have: is a verb
fun: is not a verb
?
^D
%
```

我们从第一部分开始解释正在运行的内容：

```
%{
/*
 * 这个例子演示了非常简单的识别
 * 动词/非动词
 */

%}
```

第一部分（定义段，即定义部分）介绍了将拷贝到最终程序中的原始C程序代码。例如，如果有后来文件中的代码必须包含的头文件，那么这部分尤其重要。用特

殊的定界符“%{”和“%}”括起C代码。lex将“%{”和“%}”之间的内容直接拷贝到生成的C文件，所以在这可以编写任何有效的C代码。

在这个示例中，定义段中惟一的内容是一些C注释。你也许想知道是否能包括没有定界符的注释。在“%{”和“%}”外部，lex中的注释必须用空白缩进来正确标识它们。忘记缩进注释，lex会将它们解释为别的东西，这就会出现出人意料的错误。

%%标记这一部分结束。

下一部分是规则段（即规则部分）。每个规则都由两部分组成：模式和动作，由空白分开。当lex生成的词法分析程序识别出某个模式时，将执行相应的动作。这些模式是UNIX样式的正则表达式，即由工具（例如*grep*、*sed*和*ed*）使用的相同表达式的扩展版本。第六章描述了正规表达式的所有规则。示例中的第一条规则如下：

```
[\t ]+          /* 忽略空白 */ ;
```

方括号“[]”指示括号中的任何一个字符都与模式匹配。在我们的示例中，我们接受“\t”（一个制表符）或“ ”（一个空格）。“+”意味着模式匹配加号前面的一个或多个连续的子模式的拷贝。因此，这个模式描述了空白（制表符和空格的任意组合）。规则的第二部分（动作）只是一个分号，即一个什么也不做的C语句，它的作用是忽略输入。

接下来的规则使用“|”（垂直竖线）动作。这是一个特殊的动作，意味着下一个模式应用相同的动作，因此所有的动词都使用为最后一个动词指定的动作（注1）。

我们的第一套模式是：

```
is |
```

注1：　也可以在模式中使用垂直竖线，例如**foo|bar**是匹配字符串“foo”或“bar”的模式。在模式和垂直竖线之间留有一些空格表示后面的“竖线”是动作而不是模式的部分。

```
am |
are |
were |
was |
be |
being |
been |
do |
does |
did |
should |
can |
could |
has |
have |
had |
go { printf("%s: is a verb\n", yytext); }
```

这些模式匹配列表中的任意动词。一旦识别出一个动词，就执行这个动作，即C **printf**语句。**yytext**数组包含匹配模式的文本。这个动作将打印识别出的动词，后跟字符串“: is a verb\n”。

最后两条规则是:

```
[a-zA-Z]+     { printf("%s: is not a verb\n", yytext); }

.|\n          { ECHO; /* 通常的默认状态 */ }
```

模式“[a-zA-Z]+”是一种通用的模式：它表示至少包含一个字符的任意字母字符串。“-”字符在方括号之间使用时有一个特殊的含义：它指示从“-”的左边开始到“-”的右边结束的字符范围。当看到这些模式之一时，我们的动作是打印匹配的标记和字符串“: is not a verb\n”。

不用花很长时间就可了解到，匹配前面规则中列出的任意动词的任何单词也匹配这条规则。那么你也许想知道，当看到列表中的一个动词时为什么不执行两个动作。并且“island”以“is”开头，当它看到单词“island”时执行两个动作吗？答案是lex拥有一套简单的消除歧义的规则。使词法分析程序工作的两条规则是:

1. lex 模式只匹配输入字符或字符串一次。
2. lex 执行当前输入的最长可能匹配的动作。因为“island”是比“is”长的匹配，所以 lex 把“island”看做匹配上面那条“包括一切”的规则。

如果考虑 lex 词法分析程序如何匹配模式，就应该能够看出我们的示例只匹配列出的动词。

最后一行是默认情况语句。特殊的字符“.”（英文的句号）匹配换行符以外任意的单个字符，“\n”匹配一个换行字符。特殊动作 **ECHO** 输出匹配的模式，复制标点符号或其他字符。尽管它是一种默认行为，我们还是特别列出了这种情况。我们看过一些由于这种特征而不能正确工作的复杂的词法分析程序，这种特征在默认模式匹配出乎意料的输入字符时产生奇怪的输出。（虽然对于无法匹配的输入字符都有一个默认的动作，但是很好地编写过的词法分析程序都拥有清楚的规则来匹配所有可能的输入。）

规则段的结尾以另一个 %% 来界定。

最后的部分是用户子例程段，由任意合法的 C 代码组成。在 lex 生成代码结束之后，lex 将它复制到 C 文件。我们已经包含了一个 **main()**例程。

```
%%

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

由lex产生的词法分析程序是一个称为**yylex()**的C例程，我们可以调用它（注 2）。如果动作中不包含明确的 **return** 语句，那么 **yylex()**直到处理了完整的输入之后才会返回。

将最初的示例保存在文件 *ch1-02.1* 中（因为它是第二个例子）。为了在 UNIX 系统上创建一个可执行的程序，输入下面的命令：

注 2：　实际上，我们可以不包含主程序，因为 lex 库包含像这样的默认主程序。

```
% lex ch1-02.l
% cc lex.yy.c -o first -ll
```

lex将lex规范译成C源文件，称为*lex.yy.c*，我们对它进行编译并连接到lex库 *-ll*。然后，执行得到的程序来检查它是否像本节前面所看到的那样工作。试一试可以使你自己信服，这个简单的描述确实精确地识别出了我们想要识别的那些动词。

现在，我们已经理解了第一个示例。第二个示例（见例1-2）将词法分析程序扩展为识别不同的词性。

例1-2：具有多重词性的lex示例ch1-03.l

```
%{
/*
 * 扩展第一个示例以识别其他的词性
 */

%}
%%

[\t ]+          /* 忽略空白 */ ;
is |
am |
are |
were |
was |
be |
being |
been |
do |
does |
did |
will |
would |
should |
can |
could |
has |
have |
had |
go          { printf("%s: is a verb\n", yytext); }
```

```
very |
simply |
gently |
quietly |
calmly |
angrily     { printf("%s: is an adverb\n", yytext); }

to |
from |
behind |
above |
below |
between |
below       { printf("%s: is a preposition\n", yytext); }

if |
then |
and |
but |
or          { printf("%s: is a conjunction\n", yytext); }

their |
my |
your |
his |
her |
its         { printf("%s: is an adjective\n", yytext); }

I |
you |
he |
she |
we |
they        { printf("%s: is a pronoun\n", yytext); }

[a-zA-Z]+ {
       printf("%s: don't recognize, might be a noun\n", yytext);
       }

.|\n        { ECHO; /* 通常的默认状态 */ }

%%

main()
```

```
{
        yylex();
}
```

## 符号表

第二个示例实际上没有多少不同，只是列出了比前面更多的单词，原则上可以扩展这个示例为任意多的单词。虽然它是很方便的，但是如果在词法分析程序运行时能够构建一个单词表，那么就可以在添加新的单词时不用修改和重新编译lex程序。在下一个示例中我们就这样做，即在词法分析程序运行时从输入文件中读取声明的单词时允许动态地声明词性。声明行以词性的名字开始，后面跟着要声明的单词。例如，下面声明了4个名词和3个动词：

```
noun dog cat horse cow
verb chew eat lick
```

这个单词表是一个简单的符号表，这是lex和yacc应用程序中的公用结构。例如，C编译程序存储符号表中的变量和结构名称、标签、枚举标记和所有在程序中使用的其他名字。每个名字都连同描述名字的信息一起存储。在一个C编译程序中，信息是指符号的类型、声明的作用域、变量类型等。在当前的示例中，信息是词性。

添加符号表可以完全地改变词法分析程序。不必在词法分析程序中为每个要匹配的单词放置独立的模式，只要有一个匹配任意单词的模式，再查阅符号表就能决定所找到的词性。因为词性的名字（名词、动词等）引入了一个声明行，所以它们现在是“保留字”。对于每个保留字仍然有一个独立的lex模式。还必须添加符号表维护例程，在这种情况下，**add_word()**表示在符号表中放入一个新单词，**lookup_word()**表示查寻已经输入的单词。

在程序代码中，声明一个变量**state**，用来记录是在查找单词（状态LOOKUP）还是在声明它们（在这种情况下，**state**能记住我们正在声明的单词种类）。无论何时只要我们看到以词性名字开始的行，就可以将状态设置为声明单词的种类；每次看到\n时都切换回正常的查找状态。

例 1-3 展示了定义段。

例 1-3：带符号表的词法分析程序（3 部分中的第 1 部分）ch1-04.l

```
%{
/*
 * 带符号表的单词识别程序
 */

enum {
	LOOKUP = 0, /* 默认 —— 查找而不是定义 */
	VERB,
	ADJ,
	ADV,
	NOUN,
	PREP,
	PRON,
	CONJ
};

int state;

int add_word(int type, char *word);
int lookup_word(char *word);
%}
```

为了在表中记录单独的单词类型并声明一个变量 **state**，定义一个 *enum*。在状态变量（用来跟踪定义的内容）和符号表（用来记录每个定义的单词属于何种类型）中使用这种枚举类型。另外还声明了符号表例程。

例 1-4 展示了规则段。

例 1-4：带符号表的词法分析程序（3 部分中的第 2 部分）ch1-04.l

```
%%
\n	{ state = LOOKUP; }	/* 行结束，返回到默认状态 */

	/* 无论何时，行都以保留的词性名字开始 */
	/* 开始定义该类型的单词 */
^verb { state = VERB; }
^adj  { state = ADJ; }
^adv  { state = ADV; }
```

```
^noun { state = NOUN; }
^prep { state = PREP; }
^pron { state = PRON; }
^conj { state = CONJ; }

[a-zA-Z]+ {
                /* 一个标准的单词，定义它或查找它 */
          if(state != LOOKUP) {
                /* 定义当前的单词 */
                add_word(state, yytext);
            } else {
                switch(lookup_word(yytext)) {
                case VERB: printf("%s: verb\n", yytext); break;
                case ADJ: printf("%s: adjective\n", yytext); break;
                case ADV: printf("%s: adverb\n", yytext); break;
                case NOUN: printf("%s: noun\n", yytext); break;
                case PREP: printf("%s: preposition\n", yytext); break;
                case PRON: printf("%s: pronoun\n", yytext); break;
                case CONJ: printf("%s: conjunction\n", yytext); break;
                default:
                        printf("%s: don't recognize\n", yytext);
                        break;
                }
            }
          }

        /* 忽略其他的东西 */ ;

%%
```

为了声明单词，第一组规则将状态设置为对应于被声明的词性的类型。(模式开始处的“^”使模式只在输入行的开始处匹配。)在每行的开始处重新将状态设置为**LOOKUP**，这样在我们添加新的单词后，可以交互地测试单词表，以确定它是否正确地工作。当模式“[a-zA-Z]+”匹配时，如果状态是**LOOKUP**，使用**lookup_word()**查找单词，找到就打印出它的类型。如果是其他任意状态时，用**add_word()**定义单词。

例1-5中的用户子例程段包含同样的框架**main()**例程和两个支持函数。

例 1-5：带符号表的lexer（3个部分中的第3部分）ch1-04.l

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

/* 定义一个连接的单词和类型列表 */
struct word {
	char *word_name;
	int word_type;
	struct word *next;
};

struct word *word_list; /* first element in word list */

extern void *malloc();

int
add_word(int type, char *word)
{
	struct word *wp;

	if(lookup_word(word) != LOOKUP) {
		printf("!!! warning: word %s already defined \n", word);
		return 0;
	}

	/* 单词不在那里，分配一个新的条目并将它连接到列表上 */

	wp = (struct word *) malloc(sizeof(struct word));

	wp->next = word_list;

	/* 还必须复制单词本身 */

	wp->word_name = (char *) malloc(strlen(word)+1);
	strcpy(wp->word_name, word);
	wp->word_type = type;
	word_list = wp;
	return 1;	/* 它被处理过 */
}

int
```

```
lookup_word(char *word)
{
    struct word *wp = word_list;

    /* 向下搜索列表以寻找单词 */
    for(; wp; wp = wp->next) {
        if(strcmp(wp->word_name, word) == 0)
            return wp->word_type;
    }

    return LOOKUP;      /* 没有找到 */
}
```

最后两个函数创建并搜索单词的链表。如果有许多单词，则函数运行得很慢，因为寻找每个单词都必须搜索整个列表。在产品环境中，我们会采用一种较快的但很复杂的方案 —— 大概采用一个散列表。虽然很慢，但我们的简单示例还是采用这种方法。

下面是最后一个例子的会话示例：

```
verb is am are was were be being been do
is
is: verb
noun dog cat horse cow
verb chew eat lick
verb run stand sleep
dog run
dog: noun
run: verb
chew eat sleep cow horse
chew: verb
eat: verb
sleep: verb
cow: noun
horse: noun
verb talk
talk
talk: verb
```

强烈建议读者研究这个示例，深入理解示例直到满意为止。

# 语法

对于某些应用，我们所完成的简单的词类识别也许足够用了；而另一些应用需要识别特殊的标记序列并执行适当的动作。传统上，对这样的一套动作的描述称为语法。它似乎特别适用于我们的例子。假设我们希望识别普通的句子，下面是简单句型的列表：

*noun verb.*

*noun verb noun.*

在这一点上，引进一些描述语法的符号似乎是很方便的。使用右向箭头"→"意味着可以用一个新的符号取代一套特殊的标记（注3）。例如，

*subject → noun | pronoun*

指示一个新的符号*subject*是名词（noun）或代词（pronoun）。我们没有改变底层（underlying）符号的含义；相反，我们从已经定义的更加基础的符号中构建了新的符号。正如下面的例子那样，我们可以把一个对象定义如下：

*object → noun*

当不像英语语法那样严格要求时，可以按以下方式定义句子：

*sentence → subject verb object*

实际上，可以扩展句子的定义以适合更广的句型。然而，现在还是先构建一个yacc语法，这样能交互式地彻底检查我们的想法。为了将有用的值返回给新的分析程序，在介绍yacc语法之前，必须修改词法分析程序。

注3：　这里说的是符号而不是标记，因为我们用"标记"这个词特指从词法分析程序返回的符号，箭头左边的符号不是从词法分析程序中来的。所有的标记都是符号，但不是所有的符号都是标记。

## 词法分析程序与语法分析程序的通信

当一起使用lex扫描程序和yacc语法分析程序时，语法分析程序（parser）是较高级别的例程。当它需要来自输入的标记时，就调用词法分析程序**yylex()**。然后，词法分析程序从头到尾扫描输入识别标记。它一找到对语法分析程序有意义的标记就返回到语法分析程序，将返回标记的代码作为**yylex()**的值。

不是所有的标记都对语法分析程序有意义。例如，在多数程序设计语言中，语法分析程序不能接收注释和空白。对于忽略的标记，词法分析程序不返回，以便它继续扫描下一个标记而不“打扰”语法分析程序。

词法分析程序和语法分析程序必须对标记代码的内容达成一致。通过让yacc定义标记代码来解决这个问题。在我们的语法中，标记是词性：**NOUN**、**PRONOUN**、**VERB**、**ADVERB**、**ADJECTIVE**、**PREPOSITION**和**CONJUNCTION**。yacc使用预处理程序*#define*将它们每一个都定义为小的整数。下面是这个示例中使用的定义：

```
# define NOUN 257
# define PRONOUN 258
# define VERB 259
# define ADVERB 260
# define ADJECTIVE 261
# define PREPOSITION 262
# define CONJUNCTION 263
```

输入的逻辑结束总是返回标记代码零。yacc不为它定义符号，但是如果想定义的话你可以定义它。

yacc可以生成包含所有标记定义的C头文件。可以把这个文件（在UNIX系统上称为*y.tab.h*，在MS-DOS上称为*ytab.h*或*yytab.h*）包含在词法分析程序中，并且在词法分析程序动作代码中采用这些预处理程序符号。

## 词法分析程序中的词性

例1-6展示了新的词法分析程序的声明和规则段。

例 1-6：从语法分析程序中调用词法分析程序 ch1-05.l

```
%{
/*
 * 我们现在构建一个由高级语法分析程序使用的词法分析程序
 */

#include "y.tab.h"      /* 来自语法分析程序的标记代码 */

#define LOOKUP 0        /* 默认情况 —— 不是一个定义的单词类型 */

int state;

%}

%%

\n    { state = LOOKUP; }

\.\n {      state = LOOKUP;
            return 0; /* 句子结尾 */
       }

^verb { state = VERB; }
^adj  { state = ADJECTIVE; }
^adv  { state = ADVERB; }
^noun { state = NOUN; }
^prep { state = PREPOSITION; }
^pron { state = PRONOUN; }
^conj { state = CONJUNCTION; }

[a-zA-Z]+ {
            if(state != LOOKUP) {
             add_word(state, yytext);
            } else {
             switch(lookup_word(yytext)) {
             case VERB:
               return(VERB);
             case ADJECTIVE:
               return(ADJECTIVE);
             case ADVERB:
               return(ADVERB);
             case NOUN:
               return(NOUN);
```

```
            case PREPOSITION:
              return(PREPOSITION);
            case PRONOUN:
              return(PRONOUN);
            case CONJUNCTION:
              return(CONJUNCTION);
            default:
              printf("%s: don't recognize\n", yytext);
              /* 不返回，忽略 */
            }
            }
          }

    ;

%%
... same add_word() and lookup_word() as before ...
```

这里介绍几个重要的区别。将词法分析程序中使用的词性名字改变为与语法分析程序中的标记名字相一致。还要添加**return**语句将所识别的单词的标记代码传递给语法分析程序。词法分析程序中定义新单词的标记没有任何**return**语句，因为语法分析程序不“关心”它们。

这些返回语句表明**yylex()**操作类似于协同程序。每次语法分析程序调用它时，都在它停止的那一点进行处理。这样就允许我们渐进地检查和操作输入流。我们的第一个程序不需要利用这一点，但是当将词法分析程序作为庞大的程序的一部分时会很有用。

增加一条规则来标记句子的结尾：

```
\.\n {      state = LOOKUP;
            return 0; /* 句子结尾 */
      }
```

句号前面的反斜杠引用（quote）这个句号，所以这条规则与后跟一个换行的句号匹配。对词法分析程序所做的另一个改变是省略目前语法分析程序中提供的**main()**例程。

## yacc 语法分析程序

最后，例 1-7 介绍了 yacc 语法中的第一步。

例 1-7：简单的 yacc 句子语法分析程序 ch1-05.y

```
%{
/*
 * 用于识别英文句子基本语法的词法分析程序
 */
#include <stdio.h>
%}

%token NOUN PRONOUN VERB ADVERB ADJECTIVE PREPOSITION CONJUNCTION

%%
sentence: subject VERB object{ printf("Sentence is valid.\n"); }
        ;

subject:    NOUN
        |   PRONOUN
        ;

object:         NOUN
        ;
%%

extern FILE *yyin;

main()
{
        do
          {
              yyparse();
        }
        while(!feof(yyin));
}

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

yacc语法分析程序的结构类似于lex词法分析程序的结构（不是偶然的）。第一部分（定义段）有一个文字代码块，以“%{”和“%}”括起。在这里，我们将它用于C注释（如同lex一样，C注释属于C代码块，至少在定义段中）和单个包含文件。

然后是我们期望从词法分析程序中接收的所有标记的定义。在这个示例中，它们对应8个词性。尽管很好地选择标记名字可以告诉读者它们代表的内容，但是标记的名字对于yacc没有本质意义，虽然yacc允许为yacc符号使用任意有效的C标识符名，但是通常我们规定标记名字都用大写字母，而语法分析程序中的其他名字大部分或完全是小写字母。

第一个%%指示规则段的开始，第二个%%指示规则的结束和用户子例程段的开始。最重要的子程序是**main()**，这个子程序重复调用**yyparse()**直到词法分析程序的输入文件结束。例程**yyparse()**是由yacc生成的语法分析程序，所以我们的主程序重复尝试分析句子直到输入结束。（当词法分析程序看到行的结尾处的句号时返回零标记；它是语法分析程序的信号，指示当前分析的输入已完成。）

## 规则段

规则段将实际的语法描述为一套产生式规则或简称为规则（也有些人称它们为产生式）。每条规则都由“:”操作符左侧的一个名字、右侧的符号列表和动作代码以及指示规则结尾的分号组成。默认情况下，第一条规则是最高级别的规则，也就是说，语法分析程序试图找到一个标记序列匹配这条初始规则，或者更普通地说，从初始规则中找到的规则。规则右侧的表达式是零个或多个名字的列表。像在**object**规则中被定义为**NOUN**一样，典型的简单规则的右侧有一个符号。然后，规则左侧的符号在其他规则中能像标记一样使用。接下来我们构建复杂的语法。

我们在语法中使用特殊字符“|”，它引入和前一条规则相同的左侧规则。它通常读作“或”，举例来说，在我们的语法中，subject可以是**NOUN**或**PRONOUN**。规则的动作部分由C块组成，以“{”开始并以“}”结束。只要规则匹配，语法分析程序就在规则结尾处执行一个动作。在我们的**sentence**规则中，动作报告我

们已经成功地分析了一个句子。因为**sentence**是最高层的符号，所以整个输入必需匹配**sentence**。当词法分析程序报告输入结束时，分析程序返回到它的调用程序，在这种情况下是主程序。随后对**yyparse()**的调用重置状态并再次开始处理。如果看到输入标记的“subject VERB object”列表，那么我们的示例打印一条消息。如果看到“subject subject”或一些其他的无效的标记列表会发生什么呢？语法分析程序调用**yyerror()**(它在用户的子程序段提供)，然后识别特殊的规则**error**。可以提供错误恢复代码，尝试将分析程序返回到能继续分析的状态。如果错误恢复失败，就像这里的情况那样——没有错误恢复代码，**yyparse()**在发现错误后返回到调用程序。

第3段和最后一段（即用户子例程段）在第二个%%后开始。这一段包含任意C代码并且被逐字地复制到最终的语法分析程序。在示例中，我们为yacc生成的语法分析程序提供了一组函数：**main()**和**yyerror()**，这组函数是使用lex生成的词法分析程序进行编译时所必需的。主例程继续调用语法分析程序，直到到达**yyin**（lex输入文件）上的文件尾。惟一其他必需的例程是**yylex()**，它由词法分析程序提供。

在本章的最后一个示例（见例1-8）中，扩展前面的语法来识别一组复杂的（尽管是不完全的）句子。我们请你进一步试用这个例子——你将会看到如何用明确的方式描述复杂的英语。

例1-8：*扩展的英语语法分析程序* ch1-06.y

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

%token NOUN PRONOUN VERB ADVERB ADJECTIVE PREPOSITION CONJUNCTION

%%

sentence: simple_sentence { printf("Parsed a simple sentence.\n"); }
        | compound_sentence { printf("Parsed a compound sentence.\n"); }
        ;

simple_sentence: subject verb object
```

```
        |       subject verb object prep_phrase
        ;

compound_sentence: simple_sentence CONJUNCTION simple_sentence
        |       compound_sentence CONJUNCTION simple_sentence
        ;

subject:        NOUN
        |       PRONOUN
        |       ADJECTIVE subject
        ;

verb:           VERB
        |       ADVERB VERB
        |       verb VERB
        ;

object:                 NOUN
        |       ADJECTIVE object
        ;

prep_phrase:            PREPOSITION NOUN
        ;

%%

extern FILE *yyin;

main()
{
        do
          {
                yyparse();
        }
          while(!feof(yyin));
}

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

通过引入小学英语课中传统的语法公式，我们扩展了**sentence**规则：一个句子可以是简单句，或者是包含两个或多个独立子句的由并列连接词连接的复合句。目前的词法分析程序不能区分并列连接词（例如“and”、“but”、“or”）和从属连接词（例如“if”）。

我们还将递归（recursion）概念引入了语法。递归方式（一条规则直接或间接引用它本身）是一个描述语法的强有力的工具，并且可以在我们编写的几乎每个yacc语法中采用该技术。在这里，**compound_sentence**和**verb**规则引入了递归。前一个规则只是声明**compound_sentence**是两个或多个由连接词连接的简单句。第一个可能的匹配是：

```
simple_sentence CONJUNCTION simple_sentence
```

它定义了“两个子句”的情况，而

```
compound_sentence CONJUNCTION simple_sentence
```

则定义了“两个以上子句”的情况。在稍后的几章中将更加详细地讨论递归。

虽然这里的英语语法不是特别有用，但是用lex标识单词然后用yacc找到单词之间的关系的技术与后面几章中特定应用所采用的技术几乎相同。例如，在C语言语句中：

```
if( a == b ) break; else func(&a);
```

编译程序采用lex标识标记“**if**”、“(”、“**a**”、“**==**”等，然后使用yacc建立**if**语句的表达式部分“a == b”，**break**语句是“真”分支，而函数调用它的“假”分支。

# 运行 lex 和 yacc

本章的最后部分描述如何在系统上构建这些工具。

我们调用不同的词法分析程序*ch1-N.l*，其中的*N*对应特别的lex规范示例。同样，

调用语法分析程序*ch1-M.y*，*M*是示例的编号。那么，为了构建输出，在UNIX上使用下面的命令：

```
% lex ch1-n.l
% yacc -d ch1-m.y
% cc -c lex.yy.c y.tab.c
% cc -o example-m.n lex.yy.o y.tab.o -ll
```

第一行基于lex规范运行lex，并产生包含词法分析程序的C代码的文件*lex.yy.c*。第二行用yacc生成*y.tab.c*和*y.tab.h*（后者是由-*d*开关创建的标记定义的文件）。下一行对这两个C文件进行编译。最后一行将它们连接在一起，并使用lex库中的程序*libl.a* —— 在多数UNIX系统中通常位于*/usr/lib/libl.a*目录下。如果没有使用AT&T lex和yacc，而采用其他实现，你只需简单地替换命令名而几乎不必做其他改变(特别是，Berkeley *yacc*和*flex*仅需通过将*lex*和*yacc*命令改变为*byacc*和*flex*就可工作，并且删除-*ll*连接程序标志)。然而，我们知道读者之间有许多不同，而且这是事实。例如，如果我们使用GNU代替bison而不是yacc，它将生成两个文件，称为*ch1-M.tab.c*和*ch1-M.tab.h*。在命名有很多限制的系统上，例如MS-DOS，这些名字将会改变（通常为*ytab.c*和*ytab.h*）。参见附录一到附录八详细了解不同的lex和yacc实现。

# lex和手写的词法分析程序

有人经常告诉我们用C语言编写词法分析程序太容易了，没有必要学习lex。也许是，也许不是。例1-9展示了适用于简单命令语言的（用来处理命令、数字、字符串和换行，忽略注释和空白）用C语言编写的词法分析程序。例1-10是用lex编写的等效的词法分析程序。长度上lex版本是C词法分析程序的三分之一。我们的经验是程序中的错误数一般与它的长度成正比，我们预期词法分析程序的C版本要花三倍的时间来编写和排除错误。

例1-9：用C语言编写的词法分析程序

```
#include <stdio.h>
#include <ctype.h>
char *progname;
```

```
#define NUMBER 400
#define COMMENT 401
#define TEXT 402
#define COMMAND 403

main(argc,argv)
int argc;

char *argv[];
{
int val;
while(val = lexer()) printf("value is %d\n",val);
}

lexer()
{
    int c;

    while ((c=getchar()) == ' ' || c == '\t')
        ;
    if (c == EOF)
        return 0;
    if (c == '.' || isdigit(c)) {     /* 数字 */
        while ((c = getchar()) != EOF && isdigit(c));
    if (c == '.') while ((c = getchar()) != EOF && isdigit(c));
        ungetc(c, stdin);
        return NUMBER;
    }
    if ( c == '#' ) { /* 注释 */
        while ((c = getchar()) != EOF && c != '\n');
        ungetc(c,stdin);
        return COMMENT;
    }
    if ( c == '"' ) { /* 文本文字 */
        while ((c = getchar()) != EOF &&
      c != '"' && c != '\n');
        if(c == '\n') ungetc(c,stdin);
        return TEXT;
    }
    if ( isalpha(c)) { /* 检查是否是一个命令 */
        while ((c = getchar()) != EOF && isalnum(c));
        ungetc(c, stdin);
        return COMMAND;
    }
```

```
    return c;
}
```

例 1-10：用 lex 编写的同样的词法分析程序

```
%{
#define NUMBER 400
#define COMMENT 401
#define TEXT 402
#define COMMAND 403
%}
%%
[ \t]+                    ;
[0-9]+                    |
[0-9]+\.[0-9]+            |
\.[0-9]+                  { return NUMBER; }
#.*                       { return COMMENT; }
\"[^\"\n]*\"              { return TEXT; }
[a-zA-Z][a-zA-Z0-9]+      { return COMMAND; }
\n                        { return '\n'; }
%%
#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];
{
int val;

while(val = yylex()) printf("value is %d\n",val);
}
```

lex采用自然的方式来处理一些微妙的情况，对这些情况手工编写词法分析程序是很困难的。例如，假设正在跳过一个C语言注释，为了找到注释的结尾，需要寻找“*”，然后检查下一字符是否为“/”。如果是，就完成了；如果不是，就要继续扫描。在C词法分析程序中非常常见的一个错误是不考虑下面这种情况，即下一个字符是它本身（一个星号），并且后跟一个斜杠。实际上，这意味着一些注释失败了：

```
/** 注释 **/
```

（在示例中，我们已经看到这个错误，在一个和yacc的某个版本一起发布的手写的词法分析程序中！）

一旦你轻松地使用lex，我们预言你一定会发现，正如我们所说的那样，用lex编写是如此容易，以至你永远不会编写另一个手写的词法分析程序。

下一章我们将更深入地研究lex的使用。第三章我们研究yacc的使用。然后考虑几个描述lex和yacc的较复杂问题和特征的大型示例。

# 练习

1. 扩展英语语言语法分析程序来处理比较复杂的语法：主语中的介词短语、修饰形容词的副词等等。
2. 使语法分析程序更好地处理复合动词，例如“has seen”。你也许想为助动词添加新的单词和标记类型AUXVERB。
3. 一些单词有多个词性，例如“watch”、“fly”、“time”或“bear”。如何处理它们呢？尝试添加新的单词和标记类型NOUN_OR_VERB，并且将它作为**subject**、**verb**和**object**规则的可供选择的办法。这种工作有多好？
4. 当人们听到一个不熟悉的单词时，他们通常从上下文中猜测它的词性。词法分析程序能在运行中特征化新的单词吗？例如，以“ing”结尾的单词可能是一个动词，一个跟随“a”或“the”的词可能是名词或形容词。
5. lex和yacc是用于构建实际的英语语言语法分析程序的好工具吗？为什么不是？

# 第二章

# 使用lex

本章内容：

- 正则表达式
- 单词计数程序
- 分析命令行
- C源代码分析程序
- 小结
- 练习

上一章我们示范了如何使用lex和yacc。现在我们介绍如何单独使用lex，包括一些说明lex是一个好工具的应用示例。本章不想解释lex的每一个细节，我们将在第六章“lex规范参考”中概述。

lex是构建词法分析程序的工具。词法分析程序把随机输入流标记化（tokenize），即，将它拆分成词法标记。然后，可以进一步处理这种被标记化的输出，通常是由yacc来处理的，或者它就成为“最终产品”。在第一章中，我们介绍了如何将它作为英语语法中的中间步骤。现在，要进一步查看lex规范的细节以及如何使用lex规范；我们的示例中将lex用做最后的处理步骤，而不是作为将信息传递给基于yacc的语法分析程序的中间步骤。

当编写lex规范时，可以创建lex匹配输入所用的一套模式。每次匹配一个模式时，lex程序就调用你提供的C代码来处理被匹配的文本。采用这种方式，lex程序将输入拆分成称为标记的字符串。lex本身不产生可执行程序；相反，它把lex规范转化成包含C例程**yylex()**的文件。程序调用**yylex()**来运行词法分析程序。

使用普通的C编译程序，编译lex产生的文件以及想要的任何其他文件和库。（注意lex和C编译程序甚至不必在同一台计算机上运行。作者们经常将UNIX lex的C代码放在其他计算机上，在这些计算机上lex不可用，但C代码可用。）

# 正则表达式

在描述lex规范的结构以前，需要先描述lex使用的正则表达式。正则表达式被广泛应用于UNIX环境，并且lex可以使用丰富的正则表达式语言。

正则表达式是一种使用"元（meta）"语言的模式描述。元语言用于描述特定模式。这种元语言中使用的字符是UNIX和MS-DOS中使用的标准的ASCII字符集（有时会导致混淆）。形成正则表达式的字符是：

. 匹配除换行符（"\n"）以外的任何单个字符。

* 匹配前面表达式的零个或多个拷贝。

[] 匹配括号中的任意字符的字符类。如果第一个字符是音调符号（"^"），它的含义将改变为匹配除括号中的字符以外的任意字符。方括号中的短划线指示一个字符范围，例如"[0-9]"和"[0123456789]"的含义相同。"-"或"]"作为"["后的第一个字符时照字面意义解释，即在字符类中包括短划线或方括号。当处理非英文字母表时，POSIX引进了特殊的方括号结构。参见附录八"POSIX lex和yacc"可以得到更详细的资料。除了识别以"\"开始的C转义序列以外，其他的元字符在方括号中没有特殊的含义。

^ 作为正则表达式的第一个字符匹配行的开头。也用于方括号中的否定。

$ 作为正则表达式的最后一个字符匹配行的结尾。

{} 当括号中包含一个或2个数字时，指示前面的模式被允许匹配多少次，例如：

    A{1,3}

表示匹配字母A一次到三次。如果包含名称，则认为是以该名称替换。

\ 用于转义元字符，并且作为通常C转义序列的一部分，例如，"\n"是换行符，而"*"是星号。

+ 匹配前面的正则表达式的一次或多次出现。例如：

    [0-9]+

匹配"1"、"111"或"123456"，但不匹配空字符串（如果加号换成星号，它还可以匹配空字符串）。

? 匹配前面的正则表达式的零次或一次出现。例如：

-?[0-9]+

匹配包括一个可选的前导减号的有符号的数字。

| 匹配前面的正则表达式或随后的正则表达式。例如：

cow|pig|sheep

匹配三个单词中的任意一个。

"..." 引号中的每个字符解释为字面意义——除C转义序列外元字符会失去它们特殊的含义。

/ 只有在后面跟有指定的正则表达式时才匹配前面的正则表达式。例如：

0/1

匹配字符串"01"中的"0"，但是不匹配字符串"0"或"02"中的任何字符。由跟在斜线后的模式所匹配的内容不被"使用"，并且会被转变成随后的标记。每个模式只允许一个斜线。

() 将一系列正则表达式组成一个新的正则表达式。例如：

(01)

表示字符序列01。当用*、+和|构建复杂的模式时，圆括号很有用。

要注意这些操作符中有些只操作单个字符（例如[]），而另一些则操作正则表达式。通常，复杂的正则表达式都是由简单的正则表达式构建而成的。

## 正则表达式的示例

下面准备了一些示例。首先是适用于"数字"的正则表达式：

[0-9]

可以用这种形式构建一个整数的正则表达式:

```
[0-9]+
```

上述情况至少需要一个数字。下面的形式可以允许没有数字:

```
[0-9]*
```

添加一个可选的一元减号:

```
-?[0-9]+
```

然后，将它扩展为允许小数。首先，指定一个小数（目前，我们坚持最后一个字符总为数字）:

```
[0-9]*\.[0-9]+
```

注意句点前面的“\”使句点转义为文字含义上的句点，而不是一个通配符。这种模式匹配“0.0”、“4.5”或“.31415”。但是它不匹配“0”或“2”。下面想合并匹配它们的定义，不考虑一元减号，可以使用如下形式:

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

使用分组符号“()”确定由“|”分开的正则表达式。现在，添加一元减号:

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

通过指定浮点风格的指数，可以进一步扩展这种形式。首先，为指数编写一个正则表达式:

```
[eE][-+]?[0-9]+
```

这种形式匹配大写或小写字母E，然后是一个可选的加号或减号，接着是一个数字串。例如，这种形式匹配“e12”或“E-3”。然后，使用这种表达式构建最终的表达式，指定一个实数的最终表达式为:

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

表达式使指数部分可选。编写一个使用这种表达式的真正的词法分析程序。毫无疑问，程序要检查输入，并且每当它匹配符合正则表达式要求的数字时它都会告诉我们。

程序如例 2-1 所示。

例 2-1：小数的 lex 规范

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

-?(([0-9]+)|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) { printf("number\n"); }

.       ECHO;
%%
main()
{
        yylex();
}
```

词法分析程序忽略空白并向输出回送它认为不是数字的一部分的任意字符。例如，下面是具有一些接近于有效数字的结果：

```
.65ea12
number
eanumbe
```

我们鼓励你演示这个示例和所有的示例，直到你能理解它们是如何工作的为止。例如，试着改变这个识别一元加号和一元减号的表达式。

还有一个用于脚本和简单配置文件的常见正则表达式，它匹配以升音符号“#”开始的表达式（注 1）。我们可以如下所示构建这样的正则表达式：

```
#.*
```

这里“.”匹配除换行符以外的任意字符，而“*”匹配前面表达式的零个或多个字符。这个表达式匹配注释行上换行前的所有字符，换行表示行的结束。

注 1：　也称为散列符、磅字符、等等。

最后，下面的正则表达式匹配被引用的字符串：

```
\"[^"\n]*["\n]
```

似乎使用下面的比较简单的表达式就足够了：

```
\".*\"
```

可惜的是，如果同一个输入行上有两个被引用的字符串，就会导致lex匹配错误。例如：

```
"how" to  do
```

只匹配一个模式，因为“*”匹配尽可能多的字符串。了解这个问题后，我们可以尝试下面的表达式：

```
\"[^"]*\"
```

因为表达式“[^"]*”匹配除一个引号之外的任意字符（包括“\n”），所以如果尾部的引号没有出现，这个正则表达式就会导致lex溢出它的内部输入缓冲区。因此，如果用户错误地遗漏一个引号，这个模式就可能扫描整个文件寻找另一个引号。因为标记存储在固定大小的缓冲区中（注2），迟早总的阅读量将超过缓冲区，而词法分析程序将崩溃。例如：

```
"How", she said, "is it that I cannot find it.
```

继续匹配第二个被引用的字符串直到它发现另一个引号。可能是后面的成百上千个字符，所以我们需要添加一条新规则，即引用的字符串不能超过一行而且以前面说明的复杂的（但是比较安全）正则表达式结束。lex以一种不同的方式来处理较长的字符串。参见第六章“lex规范参考”的“yymore”节。

注2：　各版本的缓冲区的大小是不同的，有时只有100个字节，而有时达到8K个字节。要得到更多的详细资料，参见第六章的“yytext”节。

# 单词计数程序

下面来看一下lex规范的实际结构。使用基本的单词计数程序（类似于UNIX程序*wc*）。

lex规范由三部分组成：定义段、规则段和用户子例程段。第一部分（定义段）处理lex用在词法分析程序中的选项，并且一般建立词法分析程序运行的执行环境。

单词计数示例的定义段如下：

```
%{
unsigned charCount = 0, wordCount = 0, lineCount = 0;
%}

word [^ \t\n]+
eol \n
```

由“%{”和“%}”括住的部分是C代码，它们将被逐字地拷贝到词法分析程序中。这些C代码一开始即被放入输出代码中，所以这里包含的定义部分可以由规则段中的代码引用。在我们的示例中，这个代码块声明了程序中使用的三个变量，它们用来跟踪遇到的字符、单词和行的数目。

最后的两行是定义。lex提供了一种简单的替换机制，从而使定义长的或复杂的模式变得很容易。我们在这里添加两个定义。第一个定义提供了单词描述：除了空格、制表符和换行符以外的字符的非空组合。第二个定义描述行结束字符，即换行。在文件的第二部分（规则段）使用这些定义。

规则段包含指定词法分析程序的模式和动作。下面是示例中的单词计数的规则段：

```
%%
{word}      { wordCount++; charCount += yyleng; }
{eol}  { charCount++; lineCount++; }
.      charCount++;
```

规则段以“%%”开始。在模式中，lex用*substitution*（即定义段中的实际的正则表达式）代替大括号{}中的名字。在词法分析程序识别了完整的单词之后，我们的示例增加单词和字符的数目。

大括号中封闭的多个语句组成的动作生成一个C语言复合语句。lex的多数版本将模式后的所有语句当做一个动作，而另一些版本只读取行的第一条语句并且默默地忽略其他的语句。如果动作包含多条语句或多个行，为了安全起见也为了使代码更加清晰，通常使用大括号。

值得重复的是，lex总是尝试匹配可能最长的字符串。因此，词法分析程序将把字符串“well-being”作为一个单词。

示例中也使用lex的内部变量**yyleng**，它包含词法分析程序识别的字符串长度。如果匹配了well-being，**yyleng**就为10。

当词法分析程序识别一个换行时，它就增加字符数和行数。同样，如果它识别任意其他字符，它就增加字符数。对于这个词法分析程序，它识别的惟一的“其他字符”是空格或制表位；其他的字符匹配第一个正则表达式并且被当做一个单词。

词法分析程序总是尝试匹配可能最长的字符串，但是当存在两个同样长度的字符串时，词法分析程序使用lex规范中的早期规则。因此，单词“I”由**{word}**规则匹配，而不是由“.”规则匹配。理解并使用这一原则会使词法分析程序更加清晰和安全。

lex规范的第三部分和最后部分是用户子例程段。再说一次，它通过“%%”和前面的段分开。用户子例程段包含任何有效的C代码。它被逐字拷贝到生成的词法分析程序中。一般说来，这一部分包含支持例程。对于这个示例，我们的“支持”代码是主程序：

```
%%
main()
{
     yylex();
     printf("%d %d %d\n",lineCount, wordCount, charCount);
}
```

首先，它调用词法分析程序的入口点**yylex()**，然后调用**printf()**打印这次运行的结果。要注意，我们的示例不做任何想像；它不接受命令行参数、不打开任何文

件，只是使用 lex 默认地读取标准输入。我们的大部分示例程序都假设你知道如何构建做这些事情的 C 代码程序。然而，值得看一下重新连接 lex 的输入流的方式，如例 2-2 所示。

*例 2-2：单词计数程序的用户子例程 ch2-02.l*

```
main(argc,argv)
int argc;
char **argv;
{
        if (argc > 1) {
                FILE *file;

                file = fopen(argv[1], "r");
                if (!file) {
                        fprintf(stderr,"could not open %s\n",argv[1]);
                        exit(1);
                }
                yyin = file;
        }
        yylex();
        printf("%u %u %u\n",charCount, wordCount, lineCount);
        return 0;
}
```

这个示例假设调用程序的第二个参数是要处理的文件（注 3）。lex 词法分析程序从标准 I/O 文件 **yyin** 中读取输入，所以当需要时，只需要改变 **yyin**。**yyin** 的默认值是 **stdin**，因为默认输入源是标准输入。

我们在 ch2-02.l 中存储了这个示例，因为它是第二章的第二个例子，而且传统上 lex 源文件以 *.l* 结尾。运行它，获得如下结果：

```
% ch2-02 ch2-02.l
467 72 30
```

我们的单词计数示例和标准 UNIX 单词计数程序之间的显著区别是：我们的单词计数示例只处理单个文件。可以使用 lex 的文件尾处理程序来调整它。

---

注 3：　传统上，第一个名字是程序名，但是如果它处于不同的环境，就必须调整示例得到正确的结果。

当**yylex()**到达输入文件的尾端时，它调用**yywrap()**，该函数返回数值0或1。如果值为1，那么程序完成而且没有输入。换句话说，如果值为0，那么词法分析程序假设**yywrap()**已经打开了它要读取的另一个文件，而且继续读取**yyin**。默认的**yywrap()**总是返回1。通过提供自己的**yywrap()**版本，可以使程序读取命令行上命名的所有文件，一次读取一个。

处理多个文件要求对代码做很多更改。例2-3展示了最终的完整的单词计数程序。

例2-3：多文件的单词计数程序

```
%{
/*
 * ch2-03.l
 *
 * 多文件的单词计数程序示例
 *
 */

unsigned long charCount = 0, wordCount = 0, lineCount = 0;

#undef yywrap       /* 默认情况下有时是一个宏 */

%}

word [^ \t\n]+
eol  \n
%%
{word}      { wordCount++; charCount += yyleng; }
{eol} { charCount++; lineCount++; }
.     charCount++;
%%

char **fileList;
unsigned currentFile = 0;
unsigned nFiles;
unsigned long totalCC = 0;
unsigned long totalWC = 0;
unsigned long totalLC = 0;

main(argc,argv)
int argc;
```

```
char **argv;
{
        FILE *file;

        fileList = argv+1;
        nFiles = argc-1;

        if (argc == 2) {
            /*
             * 因为不需要打印摘要行，所以处理单个文件的情况与处理多个文件的情况不同
             */
            currentFile = 1;
            file = fopen(argv[1], "r");
            if (!file) {
                   fprintf(stderr,"could not open %s\n",argv[1]);
                   exit(1);
            }
            yyin = file;
        }
        if (argc > 2)
            yywrap();   /* 打开第一个文件 */

        yylex();
          /*
           * 再一次，处理零个或一个文件与处理多个文件不同
           */
        if (argc > 2) {
            printf("%8lu %8lu %8lu %s\n", lineCount, wordCount,
                   charCount, fileList[currentFile-1]);
            totalCC += charCount;
            totalWC += wordCount;
            totalLC += lineCount;
            printf("%8lu %8lu %8lu total\n",totalLC, totalWC, totalCC);
        } else
            printf("%8lu %8lu %8lu\n",lineCount, wordCount, charCount);

        return 0;
}

/*
 * 词法分析程序调用yywrap处理EOF情况（例如，在这种情况下进行操作
 * 是为了连接到一个新文件）
 */
```

```
yywrap()
{
        FILE *file = NULL;

        if ((currentFile != 0) && (nFiles > 1) && (currentFile < nFiles)) {
                /*
                 * 打印出前一个文件的统计信息
                 */
                printf("%8lu %8lu %8lu %s\n", lineCount, wordCount,
                        charCount, fileList[currentFile-1]);
                totalCC += charCount;
                totalWC += wordCount;
                totalLC += lineCount;
                charCount = wordCount = lineCount = 0;
                fclose(yyin);       /* 处理这个文件 */
        }

        while (fileList[currentFile] != (char *)0) {
               file = fopen(fileList[currentFile++], "r");
               if (file != NULL) {
                    yyin = file;
                    break;
               }
               fprintf(stderr,
                       "could not open %s\n",
               fileList[currentFile-1]);
        }
        return (file ? 0 : 1);  /* 0 表示有更多输入 */
}
```

示例使用**yywrap()**执行连续的处理。还有一些其他的方式，但是这是最简单和最方便的。每次词法分析程序调用**yywrap()**时，都尝试从命令行中打开下一个文件名并将打开的文件赋给**yyin**，如果存在另一个文件就返回0，如果没有就返回1。

示例报告独立文件的大小和一整套文件最后的累积量。如果只有一个文件，那么指定文件的数目只报告一次。

运行lex (*ch2-03.l*) 文件上的最终的单词计数程序，然后运行生成的C文件 (*ch2-03.c*) 上的单词计数程序。

```
% ch2-03.pgm ch2-03.l
```

```
    107     337    2220
% ch2-03.pgm ch2-03.l ch2-03.c
    107     337    2220 ch2-03.l
    405    1382    9356 ch2-03.c
    512    1719   11576 total
```

不同系统的运行结果也不同，因为lex的不同版本会产生不同的C代码。我们不能投入太多的时间来美化输出，它只是读者的一个练习。

# 分析命令行

现在，我们看一下采用lex分析命令输入的另一个示例。通常，lex程序会读取文件，使用预定义宏**input()**从输入中得到下一个字符，使用**unput()**将一个字符放回逻辑输入流中。词法分析程序有时需要使用**unput()**在输入流中提前取字符。例如，词法分析程序不能确定它已经找到了单词尾，除非它看见了单词尾端的标点符号，但是因为标点符号不是单词的一部分，所以它必须将标点符号放回输入流作为下一个标记。

为了扫描命令行而不是一个文件，我们必须重新编写**input()**和**unput()**。这里使用的实现只在AT&T lex中工作，因为其他版本不让你重新定义这两个程序。（例如，flex直接读取输入缓冲区而且从不使用**input()**。）如果使用另一个版本的lex，参见第六章的“从字符串中输入”一节中的介绍来了解如何完成同样的事情。

采用调用程序的命令行参数，并识别3种截然不同的参数类：**help**、**verbose**和**filename**。例2-4创建读取标准输入的词法分析程序，几乎和前面的单词计数程序示例一样。

例2-4：分析命令行输入的lex规范ch2-04.l

```
%{
unsigned verbose;
char *progName;
%}

%%
```

```
-h      |
"-?"    |
-help { printf("usage is: %s [-help | -h | -? ] [-verbose | -v] "
        "[(-file| -f) filename]\n", progName);
      }
-v      |
-verbose { printf("verbose mode is on\n"); verbose = 1; }

%%

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

定义段包括文字代码块。变量 **verbose** 和 **progName** 是用在规则段的变量。

在规则段，第一条规则识别关键字 *-help* 及其简写形式 *-h* 和 *-?*。注意规则后面的动作只是打印一个用法字符串（注 4）。第二条规则识别关键字 *-verbose* 及其简写形式 *-v*。在这种情况下，设置全局变量 **verbose**（前面定义的）的值为 1。

在用户子例程段，**main()**例程保存程序名，这个程序名用在帮助命令的用法字符串中，然后调用 **yylex()**。

当词法分析程序仍然读取标准输入而不是命令行时，这个示例不分析命令行的参数。给例 2-5 添加代码取代标准的 **input()**和 **unput()**程序。（这个例子是特定于 AT&T lex 的。参见附录五中有关 flex 的这方面的相关内容。）

例 2-5：分析命令行的 lex 规范 ch2-05.l

```
%{
#undef input
#undef unput
```

注 4：因为字符串不适合放入一行，所以在编译时使用将这个字符串拆分成两个连续的字符串的 ANSI C 技术。如果有早于 ANSI 的 C 编译程序，则必须将两个字符串粘贴在一起。

```
int input(void);
void unput(int ch);
unsigned verbose;
char *progName;
%}

%%

-h      |
"-?"    |
-help   { printf("usage is: %s [-help | -h | -? ] [-verbose | -v]"
         " [(-file| -f) filename]\n", progName);
        }
-v      |
-verbose { printf("verbose mode is on\n"); verbose = 1; }

%%
char **targv;     /* 记录参数 */
char **arglim;    /* 参数结束 */

main(int argc, char **argv)
{
     progName = *argv;
     targv = argv+1;
     arglim = argv+argc;
     yylex();
}

static unsigned offset = 0;

int
input(void)
{
      char c;

      if (targv >= arglim)
           return(0); /* EOF */
      /* 参数结束，移到下一个 */
      if ((c = targv[0][offset++]) != '\0')
            return(c);
      targv++;
      offset = 0;
      return(' ');
```

```
}

/* 只简单推回原来的备份，不允许送回不同的文本 */
void
unput(int ch)
{

        /* AT&T lex sometimes puts back the EOF ! */
        if(ch == 0)
                return;         /* 忽略，不能送回 EOF */
        if (offset) {           /* 备份当前参数 */
                offset--;
                return;
        }

        targv--; /* 返回到前一个参数 */
        offset = strlen(*targv);
}
```

在定义部分，因为在 AT&T 中 lex 默认将 **input** 和 **unput** 定义为宏，所以我们将它们解除定义（**#undef**），然后，重新定义为 C 函数。

在这个示例中不改变规则段。相反，大部分改动发生在用户子例程段。在新的部分，添加了三个变量：**targv**，跟踪当前的参数；**arglim**，标记参数的结束；**offset**，跟踪当前参数中的位置。这些变量设置在**main()**中，用于指向来自命令行的参数向量。

**input()**程序处理来自词法分析程序的获取字符的调用。当前的参数用尽时，它就移到下一个参数（如果有一个的话）并继续扫描。如果没有参数，就把它作为词法分析程序的文件尾条件并返回一个零字节。

**unput()**程序处理来自词法分析程序的将字符“推回”输入流的调用。通过颠倒指针的方向来完成这项工作，在字符串中反向移动。在这种情况下，首先假设推回的字符与位于那儿的字符相同，除非动作代码明确地推回其他的东西，否则这种情况总是真的。一般情况下，动作程序可以推回它想推回的任何东西，而且**unput()**的专用版本必需能处理这种情况。

结果示例仍然回送它不识别的输入，并为它所理解的输入打印出两条消息。例如，下面显示了运行的示例：

```
% ch2-05 -verbose foo
verbose mode is on
 foo %
```

现在输入来自命令行，而且不识别的输入被回送。词法分析程序不识别的任何文本在默认的规则处都会失败，这些文本将被回送到输出。

## 起始状态

最后，添加一个 *-file* 开关并识别文件名。使用起始状态完成这项工作，在词法分析程序中它是一种捕获上下文敏感信息的方法。用起始状态标记规则只是告诉词法分析程序在起始状态有效时识别规则。在这种情况下，为了识别 *-file* 参数后的文件名，使用起始状态标明寻找文件名的时间，如例 2-6 所示。

例 2-6：具有文件名的 lex 命令扫描程序 ch2-06.l

```
%{
#undef input
#undef unput
unsigned verbose;
unsigned fname;
char *progName;
%}

%s FNAME

%%
[ ]+            /* 忽略空白 */ ;

-h      |
"-?"    |
-help { printf("usage is: %s [-help | -h | -? ] [-verbose | -v]"
         " [(-file| -f) filename]\n", progName);
        }
-v      |
-verbose { printf("verbose mode is on\n"); verbose = 1; }
-f      |
```

```
-file { BEGIN FNAME; fname = 1; }

<FNAME>[^ ]+ { printf("use file %s\n", yytext); BEGIN 0; fname = 2;}

[^ ]+ ECHO;
%%
char **targv;    /* 记录参数 */
char **arglim;   /* 参数结束 */

main(int argc, char **argv)
{
      progName = *argv;
      targv = argv+1;
      arglim = argv+argc;
      yylex();
      if(fname < 2)
            printf("No filename given\n");
}
```

*... input()和unput()函数与例2-5 相同 ...*

在定义部分，添加行“%s FNAME”，它在词法分析程序中创建新的起始状态。在规则部分，添加以“<FNAME>”开始的规则。这些规则只在词法分析程序处于FNAME状态时被识别。没有明确状态的任何规则无论当前是什么状态都会进行匹配。*-file*参数切换为FNAME状态，激活匹配文件名的模式。一旦它匹配了文件名，就切换回正则状态。

规则段的动作中的代码改变当前状态。用BEGIN语句输入一条新状态。例如，为了改变为FNAME状态，可以使用语句“BEGIN FNAME;”；要改回到默认状态，使用“BEGIN 0”（默认地，状态零也称为INITIAL）。

除了改变lex状态以外，还增加了独立变量**fname**，所以如果参数丢失，示例程序可以识别出来。要注意，如果**fname**的值没有变成2，主程序就打印一条错误消息。

在这个示例中，其他一些改变只处理文件名参数。这里的**input()**在每个命令行的参数之后都返回一个空白。规则忽略空白，然而如果没有空白，词法分析程序中*-f ile*参数和*-file*参数的表现是一样的。

我们提到过，没有明确起始状态的规则无论活跃的起始状态是什么都会进行匹配（如例 2-7 所示）。

例 2-7：初始状态的例子 ch2-07.l

```
%s MAGIC

%%
<MAGIC>.+    { BEGIN 0; printf("Magic:"); ECHO; }
magic        BEGIN MAGIC;
%%

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

当看到关键字“magic”时切换到 MAGIC 状态；否则，只回送输入。如果处于 MAGIC 状态，就在下一个被回送的标记前插入字符串“Magic:”。创建一个具有三个单词的输入文件：magic、two 和 three，并在整个词法分析程序中运行它。

```
% ch2-07 < magic.input

Magic:two
three
```

现在，我们稍微改动一下示例，使具有起始状态的规则跟在一个没有起始状态的规则之后，如示例 2-8 所示。

例 2-8：打破起始状态的示例 ch2-08.l

```
%{
        /* 这个示例故意不工作！ */
%}

%s MAGIC

%%
magic       BEGIN MAGIC;
.+          ECHO;
```

```
<MAGIC>.+   { BEGIN 0; printf("Magic:"); ECHO; }
%%

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

虽然有同样的输入，但是却得到非常不同的结果：

```
% ch2-08 < magic.input

two
three
```

可以将没有起始状态的规则隐式地认为具有一个“通配符”起始状态，它们匹配所有的起始状态。这常常是错误的根源。flex和lex的其他新版本都有“惟一的起始状态”，可以解决通配符问题。参见第七章的“起始状态”一节可以得到更多的详细资料。

# C源代码分析程序

最后的示例检查C源文件并计算所看到的不同类型的行的数目，这些行有些包含代码，有些只包含注释或者是空白。因为单个行可以既包含注释又包含代码，所以这就是问题所在，这样我们就必须决定如何计数这样的行。

首先，描述空白行。将除换行符以外什么也没有的行当做空白行。同样，具有空白区或制表符但没有其他东西的行也是空白行。正则表达式这样描述空白行：

```
^[ \t]*\n
```

“^”操作符指示这种模式必须位于行的开始。同样，要求整个行只有一个换行符“\n”在行尾。

对代码行或注释行的描述是：任何不完全是空白的行。

```
^[ \t]*\n
\n        /* 空白行被前一条规则匹配 */
.         /* 其他东西 */
```

使用新的规则“\n”计算所看到的不是都为空白的行的数目。使用第二个新规则抛弃不感兴趣的字符。下面增加的是用于描述注释的规则：

```
^[ \t]*"/*".*"*/"[ \t]*\n
```

它描述了单个行上的单个的、自包含的注释，即在“/*”和“*/”之间有可选的文本。因为“*”和“/”是两个特殊的模式字符，它们以字面意义出现时要引起来。实际上，这种模式不很正确，因为有时为如下形式：

```
/* 注释 */ /* 注释
```

上述形式就不符合这个规律。注释可以跨越多行而且“.”操作符不包含“\n”字符。当然，如果允许用“\n”字符，那么一个长的注释可能会溢出内部的 lex 缓冲区。相反，当看到注释的开头时，添加一个起始状态 COMMENT 并进入那个状态可以防止这种问题。当看到注释的结尾时，返回到默认的起始状态。不需要对单行的注释使用起始状态。下面是识别注释的开头的规则：

```
^[ \t]*"/*"
```

动作中有一个 BEGIN 语句可以切换到 COMMENT 状态。一个 COMMENT 单独出现在一行的开头对于句子来说是很重要的，不这样做会导致计数错误，例如：

```
int counter; /* 这是
        一个奇怪的注释 */
```

因为第一行不是单独的一行。我们需要把第一行作为代码行计算，而把第二行作为注释行计算。下面是实现这种想法的规则：

```
.+"/*".*"*/".*\n
.*"/*".*"*/".+\n
```

上述两个表达式描述的字符串集合有重叠，但是它们是不同的。下面的表达式符合第一个规则但不符合第二个：

```
int counter; /* 注释 */
```

因为第二个规则要求在注释后面接有文字。同样，下面的表达式符合第二条规则，但不符合第一条规则：

```
/* 注释 */ int counter;
```

它们都符合下面的表达式：

```
/* 注释 # 1 */ int counter; /* 注释 # 2 */
```

最后，我们需要完成检测注释的正则表达式。我们决定采用起始状态，所以当处于 COMMENT 状态时，仅仅需要寻找换行字符：

```
<COMMENT>\n
```

并对它们计数。当检测到“注释尾”字符时，如果注释尾部没有其他东西，就把它当做注释行计算，否则就继续处理：

```
<COMMENT>"*/"[ \t]*\n
<COMMENT>"*/"
```

第一行算作注释行，第二行继续进行处理。当我们把这些规则放在一起时，需要进行一些粘接工作，因为需要覆盖默认起始状态和 COMMENT 起始状态中的一些情况。例 2-9 展示了最终的正则表达式的代码清单，和与它们有关的动作。

例 2-9：C 源代码分析程序 ch2-09.l

```
%{
int comments, code, whiteSpace;
%}

%x COMMENT
%%
^[ \t]*"/*" { BEGIN COMMENT; /* 进入注释处理状态 */ }
^[ \t]*"/*".*"*/"[ \t]*\n {
            comments++;      /* 自包含注释 */
            }
```

```
<COMMENT>"*/"[ \t]*\n  { BEGIN 0; comments++;}
<COMMENT>"*/"          { BEGIN 0; }
<COMMENT>\n { comments++; }
<COMMENT>.\n       { comments++; }

^[ \t]*\n { whiteSpace++; }

.+"/*".*"*/".*\n { code++; }
.*"/*".*"*/".+\n { code++; }
.+"/*".*\n       { code++; BEGIN COMMENT; }
.\n              { code++; }

.                ; /* 忽略其他东西 */
%%
main()
{
        yylex();
        printf("code: %d, comments %d, whitespace %d\n",
              code, comments, whiteSpace);
}
```

添加规则“<COMMENT>\n”和“<COMMENT>.\n”来处理注释中空行情况，以及注释中的文本。强制它们匹配行结束字符意味着如下形式不会被当成两行注释：

```
/* 这是注释的开始
和结束 */ int counter;
```

而是把它们当成一行注释和一行代码。

# 小结

本章我们概述了使用 lex 的基本原理。通常对于编写简单的应用程序（例如本章开发的单词计数程序和代码行计数实用程序），lex 自己就足够用了。

lex 用一定数量的特殊字符来描述正则表达式。当正则表达式匹配输入字符串时，执行相应的动作，就是特定的一段 C 代码。首先，它匹配最长的表达式，然后，如果两个表达式具有相同的长度，则首先匹配在 lex 应用程序中出现最早的那一

个。灵活地使用起始状态、当激活特定规则时可以进一步改进，正如我们为代码行计数实用程序所做的那样。

还讨论了由lex产生的状态机使用的特殊目的的例程，例如**yywrap()**，用于处理文件结束条件并按顺序处理多个文件。使用这个例程可以使我们的单词计数示例检查多个文件。

本章主要将lex独自用做一种处理语言。后面的文章主要集中在如何将lex和yacc集成以构建其他类型的工具。但是lex自己就能处理许多其他方面的冗长的任务，而不需要全面的yacc语法分析程序。

# 练习

1. 使单词计数程序智能化地区分单词的含义，区别字母串（也许还有连字号和省略符号）这样的真正的单词和标点符号块。这个程序不要超过10行。
2. 改进C代码分析程序：计数括号、关键字等。尝试标识函数定义和描述，它们在所有大括号外面，其后跟着一个“(”。
3. lex真的和我们所说的一样快吗？将它与*egrep*、*awk*、*sed*或你所拥有的其他模式匹配程序进行比较。编写一个lex应用程序，这个程序可以寻找包含一些字符串的行并可以打印出这些行。（为了公平比较，确认要打印整行。）比较它与其他程序花费在扫描一组文件上的时间，如果你有更多版本的lex，它们有很明显的运行速度差别吗？

# 第三章

# 使用yacc

本章内容：

- 语法
- 移进/归约分析
- yacc语法分析程序
- 词法分析程序
- 算术表达式和歧义性
- 变量和有类型的标记
- 符号表
- 函数和保留字
- 用make构建语法分析程序
- 小结
- 练习

前一章集中考虑了单独使用lex的情况。尽管使用lex可以产生词法分析程序，但本章还是要学习使用yacc。凡是在lex识别正则表达式的地方，yacc都可以识别完整的语法。lex将输入流分成块（标记），然后yacc取得这些块并将它们逻辑性地归组到一起。

本章我们创建一个台式计算器。先从简单的算术开始，然后添加内置函数、用户变量，最后是用户定义的函数。

## 语法

yacc采用你指定的语法并编写识别语法中有效"句子"的分析程序。这里，我们采用很一般的术语"句子"，在C语言语法中，句子是语法上有效的C程序（注1）。

注1．　程序可以在语法上有效但在语义上无效。例如，将字符串赋值给*int*变量的C程序。yacc只处理语法，其他的确认取决于用户。

正如我们在第一章看到的那样，语法是语法分析程序用来识别有效输入符号的一系列规则。例如，下面是本章后面用于构建计算器的语法形式。

*statement* → *NAME* = *expression*

*expression* → *NUMBER* + *NUMBER* | *NUMBER* - *NUMBER*

竖线"|"意味着同一个符号有两种可能性，例如，表达式可以是加法也可以是减法。箭头左侧的符号被认定为规则的左边，通常被缩写成LHS（left-hand side），右侧的符号是规则的右边，通常缩写成RHS（right-hand side）。有些规则可以有相同的左边，竖线是对应于这种情况的一个速记符号（short hand）。实际上，出现在输入中的和被词法分析程序返回的符号是终结符或标记，而规则的左侧出现的是非终结的符号（或非终结符）。终结符和非终结符必须是不同的，标记出现在规则左侧是错误的。

通常使用树状结构来表示所分析的句子。例如，如果用语法分析输入"fred = 12+13"，树状结构如图3-1所示。"12+13"是一个表达式，"fred = *expression*"是一条语句。实际上，yacc语法分析程序不把这种树状结构作为数据结构，尽管这件事对你来说并不难做。

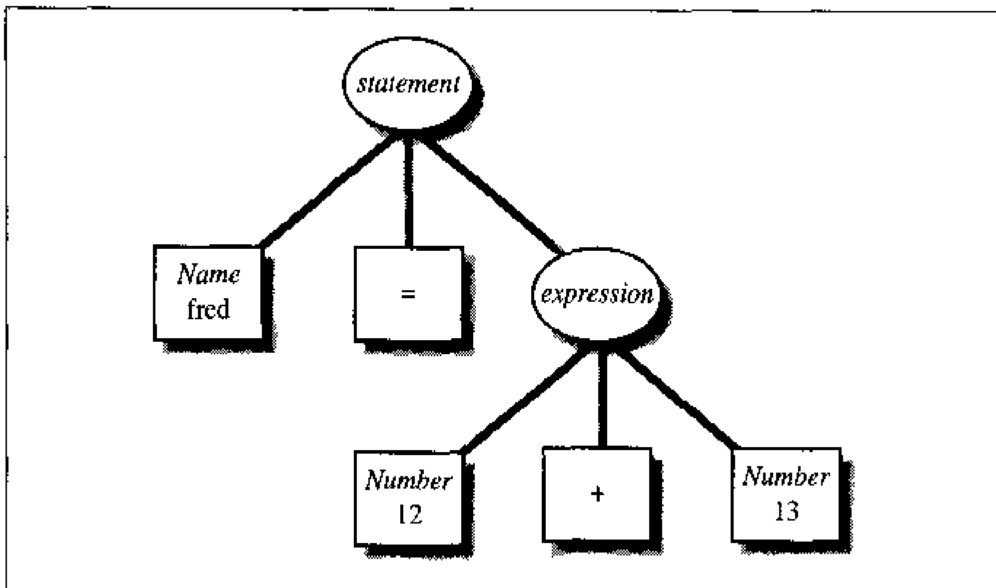



图3-1 语法分析树

每个语法都包括起始符号，这些起始符号必须位于语法分析树的根部。在这个语法中，*statement* 是起始符号。

## 递归规则

规则可以直接或间接地引用本身，这一特性使任意分析很长的输入序列成为可能。扩展语法来处理更长的算术表达式：

$$
\begin{aligned}
\textit{expression} \rightarrow\ & \textit{NUMBER} \\
\mid\ & \textit{expression} + \textit{NUMBER} \\
\mid\ & \textit{expression} - \textit{NUMBER}
\end{aligned}
$$

现在，通过重复应用表达式规则来分析一个类似“fred = 14+23−11+7”的表达式，如图 3-2 所示。yacc 对分析递归规则非常有效，所以几乎可以在使用的每个语法中看到递归规则。

# 移进 / 归约分析

yacc语法分析程序通过寻找可以匹配目前为止所看到的标记的规则来工作。yacc 处理语法分析程序时创建了一组状态，每个状态都反映一个或多个部分地被分析的规则中的一个可能的位置。当语法分析程序读取标记时，每次它读取一个没完成规则的标记，就把它压入内部堆栈中并切换到一种反映它刚刚读取的标记的新状态。这个动作称为移进（shift）。当它发现组成某条规则右侧的全部符号时，它就把右侧符号弹出堆栈，而将左侧符号压入堆栈中，并且切换到反映堆栈上新符号的新状态。这个动作称为归约（reduction），因为它通常减少堆栈上项目的数目。（但并不总是这样，因为它可能拥有右侧为空的规则。）无论yacc 什么时候归约规则，它都执行与这条规则有关的用户代码。实际上，这就是语法分析程序如何进行分析的过程。

在图 3-1 中可以看出如何使用简单的规则分析输入“fred = 12+13”。语法分析程序从将标记一次一个地移进到内部堆栈开始：

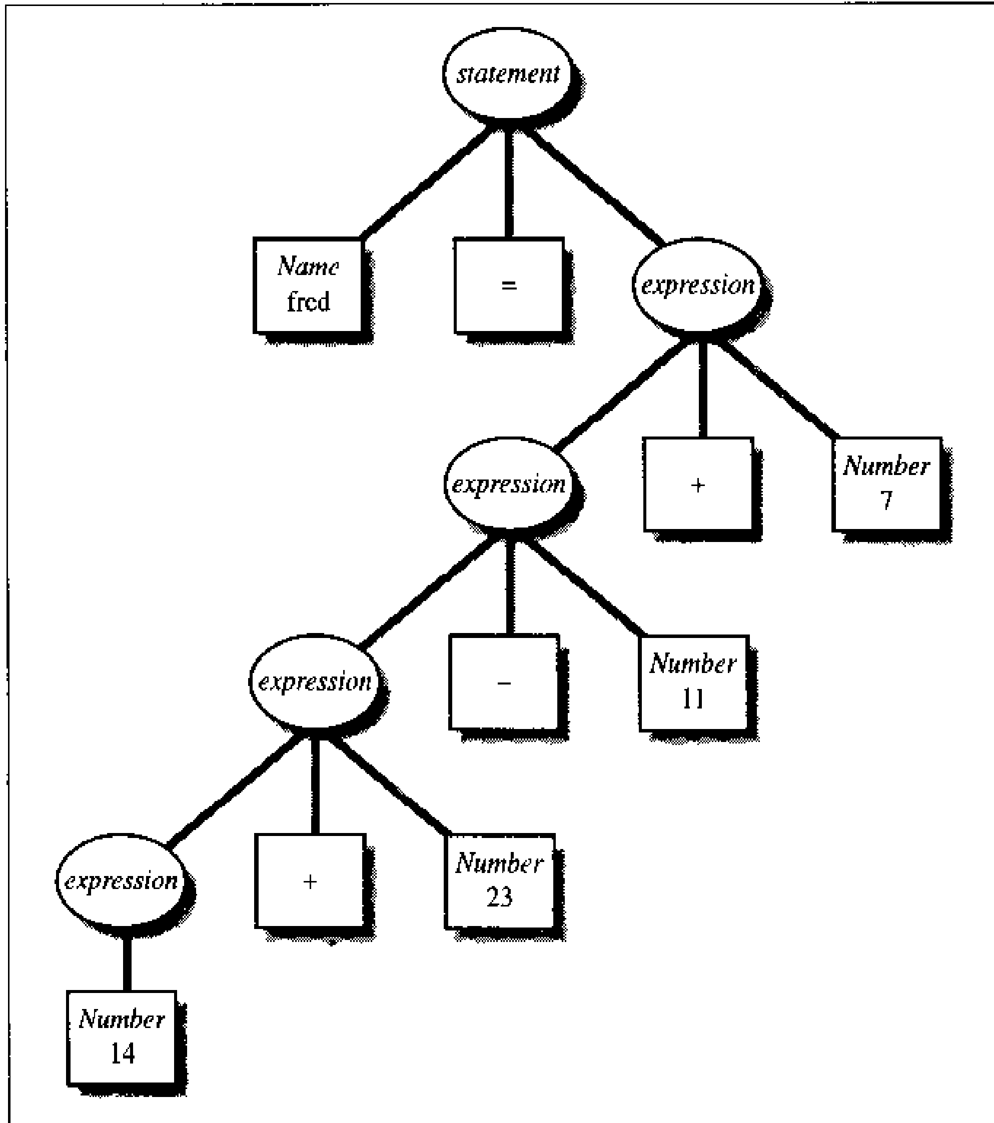



图 3-2 使用递归规则的语法分析

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

这里，它可以归约规则 “expression → NUMBER+NUMBER”，所以它从堆栈中弹出 12、加号和 13，并用下面的表达式代替它们：

```
fred = expression
```

现在，它归约了规则 “statement → NAME=expression”，所以它弹出 fred、= 和 *expression*，并用语句（statement）代替它们。当到达输入尾部时，堆栈就归约到了开始符号，所以根据这一语法这个输入是有效的。

## yacc 不能分析什么

虽然 yacc 的语法分析技术是很全面的，但也有 yacc 不能处理的语法。它不能处理歧义语法，在歧义语法中，同样的输入符合多个分析树（注 2）。它也不能处理需要向前看多于一个的标记才能确定它是否已经匹配一条规则的语法。考虑下面这个极端人为的示例：

*phrase → cart_animal AND CART*
*| work_animal AND PLOW*

*cart_animal → HORSE | GOAT*

*work_animal → HORSE | OX*

这个语法不是歧义的，因为任何有效的输入只有一个可能的语法分析树，但是 yacc 不能处理它，因为它要求向前查看两个符号。特别是，在输入 “HORSE AND CART” 中，它不能决定 HORSE 是 **cart_animal** 还是 **work_animal**，直到它看见 CART，而 yacc 不能看得那么远。

如果将第一条规则改为如下形式：

*phrase → cart_animal CART*
*| work_animal PLOW*

yacc 就不会遇到麻烦，因为它会提前看下一个标记来决定 HORSE 的输入是后跟 CART 还是后跟 PLOW。如果后跟 CART，HORSE 为 **cart_animal**，如果后跟 PLOW 则为 **work_animal**。

注 2：实际上，可以在后面看到，yacc 可以处理有限的但是却是有用的歧义语法。

实际上，这些规则不像这里看到的那样复杂和容易混淆。一个原因是yacc明确地知道能分析什么语法和不能分析什么语法。如果给出一个它不能处理的语法，它就会告诉你，所以不会出现过分复杂的分析程序悄悄失败的问题。另一个原因是yacc能处理的语法可以很好地响应人们实际上编写的程序。往往使yacc混淆的语法结构也会使人混淆，所以如果有一些语言设计方面的灵活度，就应该考虑将语言改变为使 yacc 和它的用户更加容易理解。

关于移进和归约分析的更多信息请见第八章。想要了解yacc如何才能把规范转换成可工作的C程序，请参见由Aho、Sethi和Ullman编著的经典著作《Compilers: Principles, Techniques, and Tools》，Addison-Wesley 出版社 1986 年出版，由于它的封面插图所以又称为“龙书”。

# yacc 语法分析程序

yacc 语法具有和 lex 规范一样的三部分结构（lex 结构模仿了 yacc 的结构）。第一部分（定义段）处理 yacc 生成的语法分析程序（从现在起称它为语法分析程序）的控制信息，而且通常建立语法分析程序运行的执行环境。第二部分包含语法分析程序的规则，第三部分是被逐字拷贝到生成的 C 程序中的 C 代码。

首先用最简单的语法编写语法分析程序，如图 3-1 所示，然后将它扩展为更加有用和现实的程序。

## 定义段

定义段包含语法中使用的标记的描述、分析程序堆栈中使用的值的类型和其他的东西。它还包括文字块、由 %{ %}封闭的 C 代码。下面通过声明两个符号标记来开始第一个语法分析程序。

```
%token NAME NUMBER
```

可以将单个被引起来的字符作为标记而不用声明它们，所以不需要声明“=”、“+”或“-”。

## 规则段

语法部分只是由一列语法规则组成，大部分语法规则和我们前面使用的规则具有相同的格式。因为ASCII键盘没有“→”键，所以在规则的左侧和右侧之间使用冒号，而且在每个规则的尾端都有一个分号：

```
%token NAME NUMBER
%%
statement: NAME '=' expression
        |   expression
        ;

expression: NUMBER '+' NUMBER
        |   NUMBER '-' NUMBER
        ;
```

和lex不同，yacc不关心规则部分的行边界，你会发现大量的空白使语法更容易阅读。分析程序中增加了一条新规则：语句可以是纯表达式，也可以是一个赋值。如果用户输入一个纯表达式，就可以打印出它的结果。

第一条规则左侧的符号通常是起始符号，但可以在定义部分使用**%start**声明覆盖它。

## 符号值和动作

yacc语法分析程序中的每个符号都有一个值，该值给出了符号的特定实例的额外信息。如果符号代表一个数字，那么值就是某个数字。如果它表示一个文字性的文本串，这个值可能就是一个指向该文本串拷贝的指针。如果它表示程序中的一个变量，这个值就是指向描述这个变量的符号表项的指针。有些标记没有有用的值，例如，表示闭括号的标记，因为一个闭括号和另一个闭括号是相同的。

非终结符号可以有你想要的任何值，这些值由分析程序中的代码创建。通常动作代码构建与输入相对应的语法分析树，以便后面的代码能一次处理整个语句，甚至整个程序。

在当前的语法分析程序中，*NUMBER*的值或表达式是这个数字或表达式的数值，而*NAME*的值是符号表的指针。

在真正的语法分析程序中，不同符号的值使用不同的数据类型，例如，数字符号*int*和*double*、字符串符号*char* *和较高级符号的结构指针。如果有多个值类型，就必须有语法分析程序中使用的所有值类型的列表，这样yacc可以创建包含它们的称为*YYSTYPE*的C联合类型定义。(幸运的是，yacc提供了许多方法以帮助你确保为每个符号都使用了正确的值类型。)

在计算器的第一个版本中，惟一感兴趣的值是输入数字和计算出来的表达式的数值。默认情况下，yacc使所有值类型为*int*，这对计算器的第一种版本已经足够用了。

无论语法分析程序何时归约一个规则，它都执行与规则有关的用户C代码，通常称为规则的动作。这个动作出现在规则之后且在分号或竖线之前的大括号中。动作代码可以将右侧符号的值引用为**$1**、**$2**...，而且通过设置**$$**可以设置左侧的值。在语法分析程序中，*expression*符号的值是它所代表的表达式的值。我们添加一些代码来计算和打印表达式，引出图3-2中所使用的语法。

```
%token NAME NUMBER
%%
statement: NAME '=' expression
        expression          { printf("= %d\n", $1); }
        ;

expression: expression '+' NUMBER { $$ = $1 + $3; }
        |   expression '-' NUMBER { $$ = $1 - $3; }
        |   NUMBER                { $$ = $1; }
        ;
```

构建表达式的规则计算出适当的值，而且将表达式识别为一条语句的规则打印出结果。在构建表达式的规则中，第一个和第二个数字的值分别是**$1**和**$3**。操作符的值为**$2**，尽管在这个语法中，操作符没有感兴趣的值。因为在每个归约后yacc都执行默认动作，在运行任何明确的动作代码之前，将值**$1**赋予**$$**，所以最后一条规则上的动作不是绝对必要的。

# 词法分析程序

要试验语法分析程序，需要一个词法分析程序供给标记。正如第一章所提及的那样，语法分析程序是较高级的例程，而且无论何时只要它需要输入中的标记就调用词法分析程序**yylex()**。词法分析程序一发现分析程序感兴趣的标记，它就返回给分析程序，将标记代码作为一个值返回。yacc将语法分析程序中的标记名定义为C预处理程序名字，放在*y.tab.h*中（或MS-DOS系统上的一些类似名字），所以词法分析程序可以使用它们。

下面是为语法分析程序提供标记的简单的词法分析程序：

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

%%
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
[ \t] ;              /* 忽略空白 */
\n      return 0;  /* 逻辑 EOF */
.       return yytext[0];
%%
```

数字串是数字，忽略空白，而且换行返回输入结束的标记（数字零），告诉语法分析程序没有更多的东西需要读取。词法分析程序中的最后一条规则是非常普通的截流器（catch-all），也就是将所有其他未处理的字符作为单个字符标记返回给语法分析程序。字符标记通常是标点符号，例如圆括号、分号和单字符操作符。如果语法分析程序接收到不认识的标记，就产生语法错误，所以这条规则可以使你很容易地处理所有的单字符标记，而让yacc的错误检测机制捕获和"抗议"无效的输入。

无论词法分析程序何时将标记返回给语法分析程序，如果标记有相关的值，词法分析程序在返回之前都必须在**yylval**中存储值。在这个示例中，明确地声明了**yylval**。在更复杂的语法分析程序中，yacc将**yylval**定义为一个联合（union），并将定义放置在*y.tab.h*中。

## 编译并运行简单的语法分析程序

在 UNIX 系统上，yacc 利用语法创建了 *y.tab.c*（C 语言分析程序）和包含标记号定义的头文件 *y.tab.h*。lex 创建 *lex.yy.c*（C 语言词法分析程序）。你只需要用 yacc 和 lex 库将它们编译在一起。这些库包含所有支持例程的可用的默认版本，包括用来调用语法分析程序 **yyparse()**并退出的 **main()**。

```
% yacc -d ch3-01.y    # 生成y.tab.c和"y.tab.h
% lex ch3-01.l        # 生成lex.yy.c
% cc -o ch3-01 y.tab.c lex.yy.c -ly -ll      # 编译和链接C文件
% ch3-01
99+12
= 111
% ch3-01
2 + 3-14+33
= 24
% ch3-01
100 + -50
syntax error
```

第一个版本似乎在工作。当输入某些不符合语法的东西时，第三段测试正确地报告语法错误。

# 算术表达式和歧义性

可以使算术表达式变得更加一般和实际，扩展处理乘法和除法、一元否定和带括号的表达式的 *expression* 规则：

```
expression: expression '+' expression { $$ = $1 + $3; }
        |   expression '-' expression { $$ = $1 - $3; }
        |   expression '*' expression { $$ = $1 * $3; }
        |   expression '/' expression
                {    if($3 == 0)
                          yyerror("divide by zero");
                     else
                          $$ = $1 / $3;
                }
        |   '-' expression          { $$ = -$2; }
        |   '(' expression ')'      { $$ = $2; }
```

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

除法动作检测被零除的情况，因为在多数C程序的实现中，除以零会使程序崩溃。调用 **yyerror()**(标准的 yacc 错误处理例程）报告该错误。

但是这个语法存在一个问题，它极端含糊。例如，输入 2+3*4 意味着（2+3）*4 或 2+(3*4)，输入 3-4-5-6 意味着 3-(4-(5-6))或(3-4)-(5-6)或者其他的可能。图 3-3 展示了 2+3*4 的两种可能的分析。

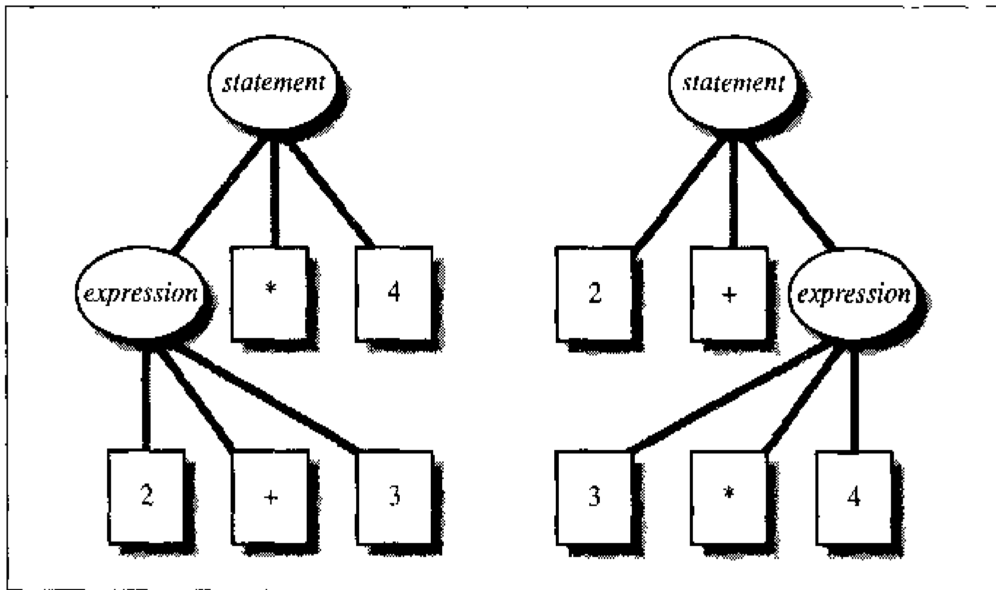



图 3-3　输入 2+3*4 时的歧义性

如果编译上面那样的语法，yacc将告知存在16个移进／归约冲突，表明它无法决定到底是先从堆栈上移进标记，还是先归约规则。

例如，当分析 “2+3*4” 时，语法分析程序要经过这些步骤（这里将表达式简写为 *E*）:

```
2                                   shift NUMBER
E                                   reduce E → NUMBER
E +                                 shift +
E + 3                               shift NUMBER
E + E                               reduce E → NUMBER
```

这时，语法分析程序查看“*”，并且可以用下面的规则将“2+3”归约为一个表达式：

```
expression:                                    expression  +  expression
```

或者移进“*”，期望稍后能归约：

```
expression:                                    expression '*  expression
```

问题是我们不能告诉yacc有关优先级和操作的组合规则。优先级控制表达式中哪个操作符要先执行。数学和编程惯例（回溯至1956年的第一个Fortan编译程序）规定乘法和除法优先于加法和减法，所以“a+b*c”意思是a+(b*c)而且d/e−f意思是(d/e)−f。在任何表达式语法中，操作符都被从低到高地归组到优先级中。级别的数目取决于语言。C语言因为有太多的优先级而出名，一共15个级别。

结合规则（associativity）控制同一优先级上的操作符组合顺序。操作符可以向左侧归组，例如在C语言中a−b−c意思是(a−b)−c；或者向右侧归组，例如a=b=c意思是a=(b=c)。在一些情况下，操作符根本不归组，例如，在Fortran中A.LE.B.LE.C是无效的。

有两种方式可以指定语法中的优先级和结合规则：隐式地和显式地。要隐式地指定它们，使用单独的非终结符号为每个优先级重新编写语法。假设使用通常的优先级和左结合，这样重新编写表达式规则如下：

```
expression: expression '+' mulexp
        |   expression '-' mulexp
        |   mulexp
        ;

mulexp:     mulexp '*' primary
        |   mulexp '/' primary
        |   primary
        ;

primary:    '(' expression ')'
```

```
        |       '-  primary
        .       NUMBER
        ;
```

这是编写一个语法的非常合理的方式，并且如果yacc没有明确的优先级规则，那么它就是惟一的方式。

但是yacc也可以让你显式地指定优先级。可以在定义部分添加这些行，结果成为例3-1中的语法。

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

这些描述定义了优先级。它们告诉yacc“+”和“-”是左结合而且处于最低的优先级，“*”和“/”是左结合而且处于较高的优先级，**UMINUS**（是代表一元减号的伪标记）没有结合规则而且处于最高的优先级。（这里没有任何右结合操作符。如果有右结合，它们将使用 **%right**。）yacc指定规则右侧最右边标记的优先级作为该规则的优先级；如果这条规则不包含具有优先级的标记，那么这条规则就没有自己的优先级。当yacc遇到歧义语法引起的移进/归约冲突时，它就参考优先级表，并且如果所有冲突的规则都包含一个出现在优先级声明中的标记时，它使用优先级来解决冲突。

在我们的语法中，所有的冲突发生在*expression OPERATOR expression*形式的规则中，所以为四个操作符设置优先级将解决所有的冲突。使用优先级的这个语法分析程序比外加隐式优先级规则的语法分析程序小而且快，因为需要归约的规则较少。

例3-1：具有表达式和优先级的计算器语法ch3-02.y

```
%token NAME NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%%
```

```
statement:  NAME '=' expression
        |   expression          { printf("= %d\n", $1); }
        ;

expression: expression '+' expression { $$ = $1 + $3; }
        |   expression '-' expression { $$ = $1 - $3; }
        |   expression '*' expression { $$ = $1 * $3; }
        |   expression '/' expression
                    {   if($3 == 0)
                            yyerror("divide by zero");
                        else
                            $$ = $1 / $3;
                    }
        |   '-' expression %prec UMINUS  { $$ = -$2; }
        |   '(' expression ')'      { $$ = $2; }
        |   NUMBER                  { $$ = $1; }
        ;
%%
```

负号规则包括“%prec UMINUS”。这条规则包括的惟一操作符是“-”，这个操作符具有很低的优先级，但是我们想让一元减号具有比乘法更高的优先级而不是较低的优先级。**%prec** 告诉 yacc 为这条规则使用 **UMINUS** 的优先级。

## 何时不使用优先规则

可以使用优先规则调整发生在语法中的任何移进/归约冲突。这通常是一个可怕的想法。在表达式语法中，冲突的原因容易理解，而且优先规则的效果清晰。在其他情况下，优先规则也可以解决移进/归约问题，但是通常很难正确地理解它们在语法中所具有的效果。

我们建议只在两种情况下使用优先级：在表达式语法中，以及为了解决 if-then-else 语言结构语法中的“虚挂的 else”冲突（参见第七章后面的例子）。或者，如果可以的话，应该调整语法来删除冲突。记住冲突意味着 yacc 不能正确地分析语法，可能因为它是歧义的，也就是说同一个输入有多个可能的分析。除了以上两种情况以外，在语言设计中，冲突就是指一个错误。如果语法对 yacc 来说是歧义的，那么它当然对人也是歧义的。参见第八章得到有关寻找和纠正冲突的更多信息。

# 变量和有类型的标记

下一步扩展计算器来处理具有单个字母名字的变量。因为只有26个字母（目前只关心小写字母），所以我们能在26个条目的数组（称它为**vbltable**）中存储变量。为了使计算器更加有用，也可以扩展它来处理多个表达式（每行一个）和使用浮点值，如例3-2和例3-3所示。

例3-2：具有变量和实值的计算器语法ch3-03.y

```
%{
double vbltable[26];
%}

%union {
        double dval;
        int vblno;
}

%token <vblno> NAME
%token <dval> NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%type <dval> expression
%%
statement_list: statement '\n'
        |    statement_list statement '\n'
        ;

statement:  NAME '=' expression     { vbltable[$1] = $3; }
        |   expression        { printf("= %g\n", $1); }
        ;

expression: expression '+' expression { $$ = $1 + $3; }
        |   expression '-' expression { $$ = $1 - $3; }
        |   expression '*' expression { $$ = $1 * $3; }
        |   expression '/' expression
                      { if($3 == 0.0)
                                yyerror("divide by zero");
                        else
                                $$ = $1 / $3;
```

```
                        }
        |       '-' expression %prec UMINUS     { $$ = -$2; }
        |       '(' expression ')'      { $$ = $2; }
        |       NUMBER
        |       NAME            { $$ = vbltable[$1]; }
        ;
%%
```

例 3-3：适用于具有变量和实值的计算器的词法分析程序 ch3-03.l

```
%{
#include "y.tab.h"
#include <math.h>
extern double vbltable[26];
%}

%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
        yylval.dval = atof(yytext); return NUMBER;
        }

[ \t] ;                 /* 忽略空白 */

[a-z]   { yylval.vblno = yytext[0] - 'a'; return NAME; }

"$"     { return 0; /* 输入结束 */ }

\n      |
.       return yytext[0];
%%
```

## 符号值和 %union

现在，我们已经有了多种符号值类型。表达式具有 *double*（双精度型）值，变量引用和 **NAME** 符号的值是对应于 **vbltable** 中的从 0 到 25 之间的整数。为了使分析程序更加简单，为什么不使词法分析程序将变量的值作为 *double* 型返回呢？问题是有两种环境可以出现变量名：一种情况是作为表达式的一部分，在这种情况下，我们要用 *double* 值；另一种情况是在等号的左边，在这种情况下需要记住它是哪个变量，这样就能更新 **vbltable**。

为了定义可能的符号类型，在定义部分，添加一个 **%union** 声明：

```
%union {
        double dval;
        int vblno;
}
```

声明的内容被逐字拷贝到输出文件，成为C的typedef语句中定义 **YYSTYPE** 类型的 **union** 声明。产生的头文件 *y.tab.h* 包括定义的拷贝，以便可以在词法分析程序中使用它。下面是从这个语法中生成的 *y.tab.h*：

```
#define NAME 257
#define NUMBER 258
#define UMINUS 259
typedef union {
        double dval;
        int vblno;
} YYSTYPE;
extern YYSTYPE yylval;
```

生成的文件还声明了变量 **yylval**，并且在语法中为符号标记定义了标记号。

现在，我们必须告诉分析程序符号使用的数值类型。通过在符号的定义部分的角括号中放置来自 union 数据类型的适当的字段名，可以完成这个任务：

```
%token <vblno> NAME
%token <dval> NUMBER

%type <dval> expression
```

新的声明 **%type** 可以为非终结符号设置类型（否则不需要声明）。也可以在 **%left**、**%right** 或 **%nonassoc** 中设置括号类型。在动作代码中，yacc 自动根据适当的字段名确定符号的引用值，例如，如果第三个符号是 **NUMBER**，引用 **$3** 的动作类似 **$3.dval**。

例 3-2 展示了新的、被扩展的分析程序。增加一个新的起始符号 **statement_list**，以便分析程序能接受一系列语句，每条语句的结尾都有一个换行符号，而不是只

有一条语句。还为设置变量的规则增加了一个动作，在尾部还增加了一条将 **NAME** 变换为 **expression** 的新规则，它获取变量的值。

必须修改一下词法分析程序（见例 3-3）。词法分析程序中的文字块不再声明 **yylval**，因为它的声明目前位于 *y.tab.h* 中。词法分析程序不能自动将类型和标记联系在一起，所以当设置 **yylval** 时必须在明确的字段引用中插入声明。从第二章以来，我们使用实数模式匹配浮点值。动作代码使用 **atof()** 读取数字，然后将值分配给 **yylval.dval**，因为分析程序期待 **dval** 字段中的数字的值。对于变量，在 **yylval.vblno** 中返回变量表中变量的下标。最后，使"\n"成为了一个普通标记，因此使用美元符号指示输入的结束。

```
% ch3-03
2/3
= 0.666667
a = 2/7
a
= 0.285714
z = a+1
z
= 1.28571
a/z
= 0.222222
$
```

# 符号表

很少有用户能满意于单个字符变量名，所以现在要增加使用较长变量名的能力。这意味着需要一个符号表——一种用来跟踪使用中的名字的结构。每次词法分析程序读取输入中的名字时，它都在符号表中查找这个名字，并且得到一个对应符号表条目的指针。在程序的其他地方，使用符号表指针而不是名字串，因为每次需要时指针比查找名字更容易更快速。

因为符号表要求词法分析程序和语法分析程序共享的数据结构，所以我们创建一个头文件 *ch3hdr.h*（见例 3-4）。这个符号表是一个结构数组，每个结构都包含变

量的名字和它的数值。我们还声明了一个例程**symlook()**，它以文本字符串形式的名字为参数，并且返回适当的符号表条目的指针，如果它不存在，就添加它。

*例3-4：具有符号表的语法分析程序的头文件ch3hdr.h*

```
#define NSYMS 20 /* maximum number of symbols */

struct symtab {
       char *name;
       double value;
} symtab[NSYMS];

struct symtab *symlook();
```

分析程序只稍做改变以使用符号表，如例3-5所示。**NAME**标记的值是指向符号表的指针而不是如前所述的下标。我们改变 **%union** 并且将指针字段称做**symp**。**NAME**的**%token**声明适当地有所改变，而且给变量赋值和读取变量的动作现在将标记值作为指针来使用，所以它们能读取或编写符号表条目的值字段。

新的程序**symlook()**在yacc规范的用户子例程部分定义，如例3-6所示。（这里没有引人注目的理由，它可以很容易地存在于lex文件中或一个单独的文件中。）它顺序地搜索字符表来寻找与作为参数来传入的名字对应的条目。如果某条目有一个 **name** 字符串并且它匹配**symlook()**正在搜索的字符串，它就返回指向该条目的指针，因为名字已经被放进了表中。如果**name**字段为空，并且已经寻找了使用中的所有表条目且没有找到这个符号，那么我们就把名字输入至今为止还是空的表条目中。

使用**strdup()**生成一个名字字符串的永久拷贝。当词法分析程序调用**symlook()**时，它传递标记缓冲区**yytext**中的名字。因为每个后来的标记都会重写**yytext**，所以这里需要生成一个拷贝（这是lex扫描程序中常见的错误根源，如果需要在扫描程序继续到下一个标记后使用**yytext**的内容，那么总要生成一个拷贝）。最后，如果当前的表条目正在使用但是没有匹配，那么**symlook()**就继续搜索下一个条目。

这个符号表程序对于这个简单的例子完全足够用了，但是越实用的符号表代码会越复杂。连续搜索对于大尺寸的符号表来说太慢了，所以要使用散列法或一些其他较快的搜索函数。实际的符号表趋向于每个条目装载相当多的信息，例如，变量的类型，无论它是一个简单的变量、数组还是结构，以及如果它是一个数组它有多少维数。

例 3-5：具有符号表的语法分析程序的规则 ch3-04.y

```
%{
#include "ch3hdr.h"
#include <string.h>
%}

%union {
      double dval;
      struct symtab *symp;
}
%token <symp> NAME
%token <dval> NUMBER
%left  -' '+'
%left  *'  /'
%nonassoc UMINUS

%type <dval> expression
%%
statement_list:  statement  \n'
      |    statement_list statement '\n'
      ;

statement:  NAME '=  expression     { $1->value = $3; }
      |     expression        { printf("= %g\n", $1); }
      ;

expression: expression '+' expression { $$ = $1 + $3; }
      |     expression  -' expression { $$ = $1 - $3; }
            expression  *' expression { $$ = $1 * $3; }
      |     expression  /' expression
                        {    if($3 == 0.0)
                                    yyerror('divide by zero');
                             else
                                    $$ = $1 / $3;
                        }
```

```
        |       '-' expression %prec UMINUS { $$ = -$2; }
        |       '(' expression ')'     { $$ = $2; }
        |       NUMBER
        |       NAME                { $$ = $1->value; }
        ;
%%
```

例 3-6：符号表程序 ch3-04.pgm

```
/* 寻找符号表条目，如果没有就添加一个 */
struct symtab *
symlook(s)
char *s;
{
        char *p;
        struct symtab *sp;
        for(sp = symtab; sp < &symtab[NSYMS]; sp++) {
                /* 它已经在这里了吗？ */
                if(sp->name && !strcmp(sp->name, s))
                        return sp;

                /* 它是空的吗？ */
                if(!sp->name) {
                        sp->name = strdup(s);
                        return sp;
                }
                /* 否则继续到下一个 */
        }
        yyerror("Too many symbols");
        exit(1); /* 不能继续 */
} /* symlook */
```

词法分析程序也只是稍微改变以适应符号表（见例3-7）。它不是直接声明符号表，而是包含*ch3hdr.h*。识别变量名的规则匹配“[A-Za-z][A-Za-z0-9]*”，任何字母和数字串都以一个字母开头。它的动作调用**symlook()**得到指向符号表条目的指针，并且将它存储在**yylval.symp**（标记的值）中。

例 3-7：具有符号表的词法分析程序 ch3-04.l

```
%{
#include "y.tab.h"
#include "ch3hdr.h"
#include <math.h>
%}
```

```
%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
            yylval.dval = atof(yytext);
            return NUMBER;
        }
[ \t] ;             /* 忽略空白 */

[A-Za-z][A-Za-z0-9]*    { /* 返回符号指针 */
            yylval.symp = symlook(yytext);
            return NAME;
        }
"$"     { return 0; }
\n      |
.       return yytext[0];
%%
```

符号表程序在一个很小的方面要优于大多数编程语言中：因为我们动态地分配字符串空间，所以对于变量名的长度没有固定的限制（注 3）：

```
% ch3-04
foo = 12
foo /5
= 2.4
thisisanextremelylongvariablenamewhichnobodywouldwanttotype = 42
3 * thisisanextremelylongvariablenamewhichnobodywouldwanttotype
= 126
$
%
```

# 函数和保留字

为计算器生成的另一个指令是添加用于平方根、指数和对数的数学函数。以如下方式处理输入：

```
s2 = sqrt(2)
s2
= 1.41421
```

注 3： 实际上，由于lex所能处理的最大标记尺寸而有所限制，但是可以使变量名非常长。见第六章 “lex 规范参考” 中的 “yytext”。

```
s2*s2
= 2
```

强制的办法是使函数名分隔标记，并且为每个函数添加单独的规则：

```
%token SQRT LOG EXP
 . . .
%%
expression: . . .
          SQRT '(' expression  )'{ $$ = sqrt($3); }
      |   LOG '(' expression  )' { $$ = log($3); }
      |   EXP  (' expression ')' { $$ = exp($3); }
```

在扫描程序中，必须为“sqrt”输入返回 **SQRT** 标记，诸如此类：

```
sqrt return SQRT;
log return LOG;
exp return EXP;

[A-Za-z][A-Za-z0-9]*   {   . . .
```

(特定模式在前，所以它们比一般的符号模式匹配得早。)

这样可以工作，但是存在问题。问题之一是必须将每个函数硬编码到语法分析程序和词法分析程序中，这样会很冗长，而且很难添加更多的函数。另一个问题是函数名是保留字，也就是说，不能将 **sqrt** 用做变量名。这是不是一个问题取决于你的意图。

## 符号表中的保留字

首先，我们从词法分析程序中拿出函数名采用的特定模式并将它们放入符号表。给每个符号表条目添加新的字段 **funcptr**，如果这个条目是函数名，那么它就是调用 C 函数的指针。

```
struct symtab {
      char *name;
      double (*funcptr)();
      double value;
} symtab[NSYMS];
```

在开始分析程序前必须将函数名放进符号表，所以编写自己的**main()**，它调用新的例程 **addfunc()**将每个函数名添加到符号表，然后调用 **yyparse()**。**addfunc()** 的代码仅仅得到名字的符号表条目并设置 **funcptr** 字段。

```
main()
{
        extern double sqrt(), exp(), log();

        addfunc("sqrt", sqrt);
        addfunc("exp", exp);
        addfunc("log", log);
        yyparse();
}

addfunc(name, func)
char *name;
double (*func)();
{
        struct symtab *sp = symlook(name);
        sp->funcptr = func;
}
```

定义**FUNC**标记代表函数名。词法分析程序看到函数名时返回**FUNC**，看到变量名时返回 **NAME**。两者的值都是符号表指针。

在语法分析程序中，用一个通用的函数规则取代每个函数的单独规则：

```
%token <symp> NAME FUNC
%%
expression: ...
        |       FUNC '(' expression ')' { $$ = ($1->funcptr)($3); }
```

当语法分析程序看到函数引用时，它可以从该函数在符号表中的条目找到真正的内部函数引用。

在词法分析程序中，针对匹配函数名的模式，如果符号表条目表明名字是函数名就改变名字的动作代码来返回 **FUNC**：

```
[A-Za-z][A-Za-z0-9]* {
```

```
        struct symtab *sp = symlook(yytext);

        yylval.symp = sp;
        if(sp->funcptr)   /* 它是函数吗？ */
            return FUNC;
        else
            return NAME;
}
```

这些改变产生和前一个程序一样工作的程序，但是函数名位于符号表中。例如，这个程序在分析过程中可以输入新的函数名。

## 可互换的函数和变量名

最后的改变是技术上的"小儿科"，但是可以有效地改变语言。函数和变量必须分开没有理由！语法分析程序根据语法可以区分函数调用和变量引用。

所以让词法分析程序返回它原有的方式，即总是为任何一种名字返回**NAME**。然后改变语法分析程序以在函数处接受**NAME**：

```
%token <symp> NAME
%%
expression: ...
    |   NAME  '(' expression ')' { ... }
```

整个程序包含在例3-8至例3-11中。正如你在例3-9中所看到的那样，必须添加错误检测以确定用户调用函数时，它是一个真正的函数。

现在，除了函数和变量名可以重叠以外，计算器如前所述进行操作。

```
% ch3-05
sqrt(3)
= 1.73205
foo(3)
foo not a function
= 0
sqrt = 5
sqrt(sqrt)
= 2.23607
```

你是否想让用户在同一程序中为两件事情使用相同的名字是有争议的。一方面它使程序变得难于理解，但是另一方面可以不再强制用户只使用不与保留字发生冲突的名字。

两者都会走向极端。COBOL 有 300 多个保留字，没有人能都记住它们，编程人员可能采用了不常用的约定，例如以数字开始每个变量名以确保它们不发生冲突。另一方面，PL/I 根本没有保留字，所以可以写：

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

**例 3-8：最终的计算器头文件 ch3hdr2.h**

```
#define NSYMS 20 /* 最大符号数 */

struct symtab {
      char *name;
      double (*funcptr)();
      double value;
} symtab[NSYMS];

struct symtab *symlook();
```

**例 3-9：最终的计算器语法分析程序的规则 ch3-05.y**

```
%{
#include "ch3hdr2.h"
#include <string.h>
#include <math.h>
%}

%union {
      double dval;
      struct symtab *symp;
}
%token <symp> NAME
%token <dval> NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%type <dval> expression
%%
```

```
statement_list:   statement '\n'
          |       statement_list statement '\n'
          ;

statement:  NAME  '='  expression    { $1->value = $3; }
          |       expression         { printf("= %g\n", $1); }
          ;

expression: expression '+' expression { $$ = $1 + $3; }
          |       expression '-' expression { $$ = $1 - $3; }
          |       expression '*' expression { $$ = $1 * $3; }
          |       expression '/' expression
                          {     if($3 == 0.0)
                                        yyerror("divide by zero");
                                else
                                        $$ = $1 / $3;
                          }
          |       '-' expression %prec UMINUS   { $$ = -$2; }
          |       '(' expression ')'        { $$ = $2; }
          |       NUMBER
          |       NAME                  { $$ = $1->value; }
          |       NAME '(' expression ')' {
                        if($1->funcptr)
                                $$ = ($1->funcptr)($3);
                        else {
                                printf("%s not a function\n", $1->name);
                                $$ = 0.0;
                        }
                  }
          ;
%%
```

例 3-10：最终计算器语法分析程序的用户子例程 ch3-05.y

```
/* 查找符号表条目，如果没有就添加一个 */
struct symtab *
symlook(s)
char *s;
{
        char *p;
        struct symtab *sp;

        for(sp = symtab; sp < &symtab[NSYMS]; sp++) {
                /* 它已经在这里吗？ */
                if(sp->name && !strcmp(sp->name, s))
```

```
                return sp;
            /* 它是空的吗? */
            if(!sp->name) {
                sp->name = strdup(s);
                return sp;
            }
            /* 否则继续到下一个 */
        }
        yyerror("Too many symbols");
        exit(1);  /* 不能继续 */
} /* symlook */

addfunc(name, func)
char *name;
double (*func)();
{
        struct symtab *sp = symlook(name);
        sp->funcptr = func;
}

main()
{
        extern double sqrt(), exp(), log();

        addfunc("sqrt", sqrt);
        addfunc("exp", exp);
        addfunc("log", log);
        yyparse();
}
```

**例 3-11: 最终的计算器词法分析程序 ch3-05.l**

```
%{
#include "y.tab.h"
#include "ch3hdr2.h"
#include <math.h>
%}

%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
                yylval.dval = atof(yytext);
                return NUMBER;
        }
```

```
[ \t] ;              /* 忽略空白 */

[A-Za-z][A-Za-z0-9]*    {   /* 返回符号指针 */
            struct symtab *sp = symlook(yytext);
            yylval.symp = sp;
            return NAME;
        }

"$" { return 0; }

\n      |
.       return yytext[0];
%%
```

# 用 make 构建语法分析程序

我们第二次重新编译这个例子，在重新编译过程中使用一些自动操作是适宜的，即使用 UNIX *make* 程序。控制这个过程的 *Makefile* 如例 3-12 所示。

例 3-12：计算器的 Makefile

```
#LEX = flex -I
#YACC = byacc

CC = cc -DYYDEBUG=1

ch3-05: y.tab.o lex.yy.o
	$(CC) -o ch3-05 y.tab.o lex.yy.o -ly -ll -lm

lex.yy.o: lex.yy.c y.tab.h

lex.yy.o y.tab.o: ch3hdr2.h

y.tab.c y.tab.h: ch3-05.y
	$(YACC) -d ch3-05.y

lex.yy.c : ch3-05.l
	$(LEX) ch3lex.l
```

在顶端是两个被注释的赋值语句，即用flex代替lex，用Berkeley yacc代替AT&T yacc。flex需要 *-I*标志以表明它产生一个交互式扫描程序，这个交互式扫描程序

不试图向前查看越过一行。CC 宏设置预处理程序符号 **YYDEBUG**，用于插入测试语法分析程序时有用的调试代码。

将所有的事情汇编到 *ch3* 的规则涉及 3 个库：yacc 库 *-ly*)、lex 库 *-ll* 和数学库 *-lm*。yacc 库提供 **yyerror()**(在计算器的早期版本中，还有 **main()**)。lex 库提供一些内部的lex扫描程序需要的支持例程(由flex产生的扫描程序不需要库，但是保留它没有害处)。数学库提供 **sqrt()**、**exp()**和 **log()**。

如果必须使用 bison（yacc的 GNU 版本)，那么就必须改变产生 *y.tab.c* 的规则，因为 bison 使用不同的默认文件名。

```
y.tab.c y.tab.h: ch3yac.y
        bison -d ch3yac.y
        mv ch3yac.tab.c y.tab.c
        mv ch3yac.tab.h y.tab.h
```

(或者将 *Makefile* 和代码的其余部分改变为使用 bison 所采用的更容易记忆的名字，另外还可以使用 *-y* 告诉 bison 使用通常的 yacc 文件名。)

要得到有关*make*的更多信息，参见Steve Talbott编写的《Managing Projects with Make》，由 O'Reilly & Associates 出版。

# 小结

本章已经看到了如何创建yacc语法规范，将它与词法分析程序结合起来生成工作的计算器，并且扩展这些计算器以处理符号变量和函数名。在下面的两章中，将讨论更庞大的而且更加实际的应用程序，即SQL数据库语言的菜单产生程序和处理程序。

# 练习

1. 给计算器添加更多的函数。例如，使用下面的规则，尝试添加两个参数函数，例如取模和反正切:

```
expression: NAME '(' expression ',' expression ')'
```

可以在符号表中为两个参数的函数放置一个独立的字段，所以可以根据一个参数还是两个参数调用适当的**atan()**版本。

2. 添加字符串数据类型，以便将字符串分配给变量并且在表达式或函数调用中使用它们。为引用文字字符串添加STRING标记。将**expression**的值变为包含值类型标签以及值的结构。另外，用**stringexp**非终结符号为具有字符串（*char**）值的字符串表达式扩展语法。

3. 如果添加一个**stringexp**非终结符号，当用户键入`42 + "grapefruit"`时会发生什么？修改语法以允许混合类型的表达式有多困难？

4. 将字符串值赋给变量时必须做什么？

5. 重载操作符有多困难（例如，如果参数是字符串，使用“+”意味着拼接）？

6. 给计算器添加命令以保存和恢复来自和去往磁盘文件的变量。

7. 向计算器添加用户定义的函数。困难的部分是以用户调用函数时能重新执行的方式存储该函数的定义。一种可能性是保存定义函数的标记流。例如：

```
statement: NAME '(' NAME ')' '=' { start_save($1, $3); }
    expression
    { end_save(); define_func($1, $3); }
```

函数**start_save()**和**end_save()**告诉词法分析程序保存表达式的所有标记的列表。在定义表达式中需要标识对哑变量**$3**的引用。

当用户调用这个函数时，重放这个标记：

```
expression: USERFUNC '(' expression ')' { start_replay($1, $3); }
    expression/* 重新执行函数 */
    { $$ = $6; }/* 使用这个值 */
```

当重放这个函数时，将参数值**$3**插入重放的表达式代替哑变量。

8. 如果继续向计算器中添加特征，最终以你的独特的程序语言结束。那是个好主意吗？为什么是或者为什么不是？

本章内容：

- MGL的概述
- 开发MGL
- 构建MGL
- 屏幕处理
- 结束
- MGL代码示例
- 练习

# 第四章
# 菜单生成语言

前一章提供了解释程序的简单例子——桌面计算器。本章重点集中在编译程序设计上：开发菜单生成语言（*menu generation language*、*MGL*）和它的编译程序。我们从对要创建的语言的描述开始。然后查看几个开发lex和yacc规范的迭代过程。最后，创建与语法有关的动作，它们实现MGL的特征。

## MGL的概述

我们将开发一种用于生成定制菜单界面的语言。它读取输入描述文件，产生能被编译的C程序，该程序能在用户终端上创建输出并使用标准的*curses*库在屏幕上绘制菜单（注1）。

在许多情况下，当应用程序要求大量冗长的、重复的代码时，设计特殊目的语言和编写将语言翻译成C或你的计算机能处理的其他语言的编译程序是比较容易和快速的。curses程序设计是冗长的，因为必须亲自定位屏幕上所有的数据。MGL自动进行大部分布局设计，这大大地减轻了工作量。

注1：要得到有关curses的更多的信息，参见由John Strang编写的《Programming with Curses》，O'Reilly & Associates出版。

菜单描述由以下部分组成:

1. 菜单屏幕的名字
2. 标题
3. 菜单项目列表，每个项目又包括以下内容:

   项目
   [命令]
   动作
   [属性]

   项目（item）是出现在菜单上的文本串；命令（command）是对菜单系统函数的记忆码，用于提供命令行访问；动作（action）是当菜单项目被选择时将执行的过程；属性（attribute）指示项目应该如何处理。括号中的项目是可选的。

4. 一个终结符

因为有用的应用程序通常有几个菜单，一个描述文件可以包含几个不同的命名菜单。

菜单描述文件的例子:

```
screen myMenu
title "My First Menu"
title "by Tony Mason"

item "List Things to Do"
command "to-do"
action execute list-things-todo
attribute command

item "Quit"
command "quit"
action quit

end myMenu
```

MGL编译程序读取这个描述文件并产生C代码，C代码本身必须被编译。当作为结果的程序被执行时，它创建了下面的菜单：

```
        My First Menu
        by Tony Mason

    1) List Things to Do
    2) Quit
```

当用户按下“1”或输入命令“to-do”时，执行过程“list-things-todo”。

这种格式的更一般的描述是：

```
screen <name>
title <string>

item <string>
[ command <string> ]
action {execute | menu | quit | ignore} [<name>]
[ attribute {visible | invisible} ]

end <name>
```

当我们开发这种语言时，从这种功能性的一个子集开始，并向这个子集添加特征直到实现完整的规范。这种方式表明了在我们改变语言时修改lex产生的词法分析程序和yacc产生的语法分析程序是多么容易。

# 开发MGL

查看生成前面的语法的设计过程。菜单为缺乏经验的用户提供了简单、清晰的界面。对于这些用户，由菜单系统提供使用的稳定性和简易性是理想的。

菜单的主要缺点是，对经验丰富的用户而言，这种方式不能直接进入想要的应用。对于这些用户，命令驱动的界面更合意。然而，几乎大部分经验丰富的用户有时也想运行菜单来访问一些不常用的函数。

很难制定合用的语言。不过，可以为它草拟词法规范：

```
ws      [ \t]+
nl      \n
%%
{ws}    ;
command         { return COMMAND; }
{nl}    { lineno++; }
.       { return yytext[0];}
```

以及相应的 yacc 语法：

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

%token COMMAND
%%
start:          COMMAND
        ;
```

词法分析程序仅仅寻找关键字并且当它识别出一个时返回适当的标记。如果语法分析程序看到标记 **COMMAND**，那么 **start** 规则将被匹配，并且 **yyparse()**将成功地返回。

菜单上的每个项目都有一个与它相关的动作。我们可以引进关键字 **action**。通过添加关键字 **ignore** 和 **execute**，一个动作可以忽略项目（忽略不能实现的或不可用的命令），另一个动作可以执行程序。

因此，使用这种修改后的词汇表的示例项目如下：

```
command action execute
```

我们必须告诉它执行什么，所以添加第一个非命令参数，即一个字符串。因为程序名可以包含标点符号，所以假定程序名是引用字符串。现在示例项目成为：

```
command action execute "/bin/sh"
```

例 4-1 表明我们可以修改 lex 规范来支持新的关键字，以及新的标记类型。

**例 4-1：MGL 词法分析程序的第一种版本**

```
ws          [ \t]+
qstring     \"[^\"\n]*[\"\n]
nl          \n
%%
{ws}        ;
{qstring}   { yylval.string = strdup(yytext+1); /* 跳过开引号 */
              if(yylval.string[yyleng-2] != '"')
                warning("Unterminated character string",(char *)0);
            else
                yylval.string[yyleng-2] = '\0'; /* 删除闭引号 */
                return QSTRING;
            }
action         { return ACTION; }
execute        { return EXECUTE; }
command        { return COMMAND; }
ignore         { return IGNORE; }
{nl}           { lineno++; }
.              { return yytext[0]; }
```

**qstring** 的复杂的定义对于阻止 lex 匹配类似下面的行是有必要的：

```
"apples' and "oranges
```

模式的 "`[^\"\n]*`" 部分表明，要匹配不是引号或者换行符的每个字符。我们不想匹配一行以外的字符，因为丢失的闭引号会导致词法分析程序费力通查文件的其他部分（这种结果当然不是用户想要的），而且也许会溢出内部 lex 缓冲区从而使程序失败。使用这种方法，我们可以通过更加"礼貌"的方式向用户报告错误条件。当我们拷贝字符串时，删除打开的和关闭的引号，因为在代码的其他部分处理没有引号的字符串更容易。

我们还需要修改 yacc 语法（见例 4-2）。

**例 4-2：MGL 语法分析程序的第一种版本**

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

```
%union {
        char *string;        /* 字符串缓冲区 */
}

%token COMMAND ACTION IGNORE EXECUTE
%token <string> QSTRING
%%
start:      COMMAND action
       ;

action:    ACTION IGNORE
         | ACTION EXECUTE QSTRING
         ;
%%
```

定义一个包括“字符串”类型和这个新标记 **QSTRING** 的 **%union**，这个标记代表一个引用字符串。

需要将单个命令和选项组合的信息归组为一个菜单项（menu item）。使用关键字 **item**，将每个新的项目都作为一个项目来介绍。如果有相关的命令，就使用关键字 **command**。向词法分析程序添加新的关键字：

```
 . . .
%%
 . . .
item       { return ITEM; }
```

虽然改变了语言的基本结构，但是对词法分析程序的改变很少。这些改变显示在 yacc 语法中，如例 4-3 所示。

例 4-3：具有项目和动作的语法

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

%union {
        char *string;        /* 字符串指针 */
}

%token COMMAND ACTION IGNORE EXECUTE ITEM
```

```
%token <string> QSTRING
%%
item:    ITEM command action
        ;

command:  /* 空值 */
         | COMMAND
         ;

action:  ACTION IGNORE
        | ACTION EXECUTE QSTRING
        ;
%%
```

因为每个菜单项不都需要相应的命令，所以**command**规则的右侧可以为空。令人惊讶的是，yacc处理这样的规则不会有麻烦，只要其余的语法使它有可能明确地表明可选的元素不在那里就行。

我们仍然没有给关键字**command**任何含义。实际上，尝试独自编写yacc语法通常是个好主意，因为它可以指示出语言设计中的“缺陷”。幸运的是，缺陷被很快地补救了。我们把命令限制为字母数字字符串。为词法分析程序的“标识符”标记添加**ID**:

```
 . . .
id        [a-zA-Z][a-zA-Z0-9]*
%%
 . . .
{id}          { yylval.string = strdup(yytext);
                return ID;
              }
```

**ID**的值是标识符名字的指针。通常，将指向**yytext**的指针作为符号值返回不是个好主意，因为一旦词法分析程序读取下一个标记，它就重写**yytext**。（没有拷贝**yytext**是常见的词法分析程序错误，通常会产生奇怪的故障，字符串和标识符看上去莫名其妙地改变，它们的名字。）使用**strdup()**复制标记并返回复本的指针。使用ID的规则必须仔细，在处理完拷贝时必须释放它。

在例4-4中向yacc语法添加**ID**标记。

例 4-4：具有命令标识符的语法

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

%union {
        char *string;          /* 字符串缓冲区 */
}

%token COMMAND ACTION IGNORE EXECUTE ITEM
%token <string> QSTRING ID
%%
item:     ITEM command action
          ;

command:   /* 空值 */
          | COMMAND ID
          ;

action:    ACTION IGNORE
         | ACTION EXECUTE QSTRING
         ;
%%
```

在一个菜单中，语法不提供多行。为支持多个 **items** 的 **items** 添加一些规则。

```
  . . .
%%
items:  /* empty */
      | items item
      ;
item:     ITEM command action
         ;
  . . .
```

与前面所有的规则不同，这些规则依赖于递归。因为yacc喜欢左递归的语法，所以编写为“items item”的形式而不是右递归的形式“item items”（参见第七章的“递归规则”一节来了解为什么左递归更好）。

**items**的一个规则有空的右侧。任何递归规则都必须有终结条件，即非终结符以非递归形式匹配的规则。如果一条规则是没有非递归替代物的完全递归式，那么

yacc的大部分版本都会由于致命的错误而停止，因为在那些情况下建立一个有效的语法分析程序是不可能的。我们在许多语法中多次使用左侧递归式。

除了能指定菜单中的项目以外，在菜单的顶部还有一个标题。下面是描述一个标题的语法规则：

```
title:          TITLE       QSTRING
          ;
```

关键字 **title** 引进了一个标题。要求标题是一个引用字符串。给lex规范添加新的标记 **TITLE**：

```
 . . .
%%
title           { return TITLE; }
 . . .
```

要想有多个标题行，附加的语法是：

```
titles:  /* empty */
       | titles title
       ;

title:      TITLE        QSTRING
       ;
```

递归定义允许多个标题行。

标题行的增加意味着必须添加一个由项目或标题组成的新的、高级的规则。标题在项目的前面，所以例4-5在语法中添加了一个新的规则 **start**。

例4-5：具有标题的语法

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

%union {
        char *string;           /* 字符串缓冲区 */
}
```

```
%token COMMAND ACTION IGNORE EXECUTE ITEM TITLE
%token <string> QSTRING ID
%%
start: titles items
       ;

titles:   /* 空值 */
        | titles title
        ;

title:      TITLE    QSTRING
        ;

items:   /* 空值 */
       | items item
       ;

item:    ITEM command action
       ;

command:   /* 空值 */
         | COMMAND ID
         ;

action:   ACTION IGNORE
        | ACTION EXECUTE QSTRING
        ;
%%
```

在少量使用MGL之后，我们发现一个菜单屏幕不够用。想要多个屏幕并能从一个屏幕中引用另一个屏幕，即允许多级菜单。

定义一条新的屏幕（screen）规则来包含具有标题和项目的完整的菜单。为了添加多个屏幕的操作，可以使用递归规则构建一个新的**screens**规则。要想允许空屏幕，添加5条新规则集：

```
screens:   /* 空值 */
         | screens screen
         ;

screen:  screen_name screen_contents screen_terminator
```

```
        | screen_name screen_terminator
        ;

screen_name:  SCREEN ID
            | SCREEN
            ;

screen_terminator:  END ID
                  | END
                  ;

screen_contents: titles lines
```

每个屏幕都有一个惟一的名字。当希望引用一个特殊的菜单屏幕时，比方说“first”，我们可以使用下面的形式：

```
item "first' command first action menu first
```

命名屏幕时，还必须指示屏幕何时结束，所以需要 **screen_terminator** 规则。因此，示例的屏幕规范也许如下所示：

```
screen main
title "Main screen"
item "fruits" command fruits action menu fruits
item 'grains' command grains action menu grains
item "quit' command quit action quit
end main

screen fruits
title 'Fruits'
item 'grape' command grape action execute '/fruit/grape'
item 'melon' command melon action execute "/fruit/melon"
item 'main" command main action menu main
end fruits

screen grains
title 'Grains'
item 'wheat' command wheat action execute "/grain/wheat"
item 'barley" command barley action execute "/grain/barley"
item "main' command main action menu main
end grains
```

规则能支持没给出名字的情况；因此，**screen_name**和**screen_terminator**具有两种情况。实际上，当编写特定规则的动作时，我们将检测名字的一致性，来检查菜单描述缓冲区中不一致的编辑错误。

进一步考虑之后，决定向菜单生成语言添加多个特征。对于每个项目，希望指示它是否可见。因为它是单个项目的一个属性，用新的关键字**attribute**来处理可见或不可见的选项。下面是描述一个属性的新语法的一部分：

```
attribute: /* empty */
         | ATTRIBUTE VISIBLE
         | ATTRIBUTE INVISIBLE
         ;
```

允许空的属性字段接受一个默认的、也许可见的属性。例4-6是可工作的语法。

**例4-6：完整的MGL语法**

```
screens:  /* 空值 */
        | screens screen
        ;

screen:   screen_name screen_contents screen_terminator
        | screen_name screen_terminator
        ;

screen_name:  SCREEN ID
            | SCREEN
            ;

screen_terminator:  END ID
                  | END
                  ;

screen_contents: titles lines
               ;

titles:  /* 空值 */
       | titles title
       ;

title: TITLE QSTRING
     ;
```

```
lines: line
      | lines line
      ;

line: ITEM QSTRING command ACTION action attribute
     ;

command:  /* 空值 */
        | COMMAND ID
        ;

action:   EXECUTE QSTRING
        | MENU ID
        | QUIT
        | IGNORE
        ;

attribute:  /* 空值 */
          | ATTRIBUTE VISIBLE
          | ATTRIBUTE INVISIBLE
          ;
```

我们已经用**screens**规则取代前面示例的**start**规则作为最高层的规则。如果在描述部分没有 %**start** 行，那么它就使用第一条规则。

# 构建 MGL

现在，有了一个基本语法，然后开始构建编译程序的工作。首先，必须完成词法分析程序的修改，以处理上一次语法修改中引进的新关键字。修改的 lex 规范如例 4-7 所示。

例 4-7：MGL lex 规范

```
ws        [ \t]+
comment   #.*
qstring   \"[^\"\n]*[\"\n]
id        [a-zA-Z][a-zA-Z0-9]*
nl        \n

%%
```

```
{ws}        ;
{comment} ;
{qstring} { yylval.string = strdup(yytext+1);
            if(yylval.string[yyleng-2] != '"')
              warning("Unterminated character string",(char *)0);
          else
              yylval.string[yyleng-2] = '\0'; /* 删除闭引号 */
            return QSTRING;
          }
screen      { return SCREEN; }
title       { return TITLE; }
item        { return ITEM; }
command     { return COMMAND; }
action      { return ACTION; }
execute     { return EXECUTE; }
menu        { return MENU; }
quit        { return QUIT; }
ignore      { return IGNORE; }
attribute { return ATTRIBUTE; }
visible     { return VISIBLE; }
invisible { return INVISIBLE; }
end         { return END; }
{id}        { yylval.string = strdup(yytext);
              return ID;
            }
{nl}        { lineno++; }
.           { return yytext[0]; }
%%
```

处理关键字的另一种方法如例4-8所示。

例4-8：另一种lex规范

```
 . . .
id          [a-zA-Z][a-zA-Z0-9]*
%%
 . . .
{id}          { if(yylval.cmd = keyword(yytext)) return yylval.cmd;
                 yylval.string = strdup(yytext);
                 return ID;
                }
%%
/*
 * keyword: 检查一个文本字符串是否是一个合法的关键字,
```

```
 * 如果是，返回关键字keyword的值，
 * 否则返回0。注意：标记值必须非零
 */

static struct keyword {
char *name;      /* 文本字符串 */
int value;       /* 标记 */
} keywords[] =
{
"screen",    SCREEN,
"title",     TITLE,
"item",      ITEM,
"command",   COMMAND,
"action",    ACTION,
"execute",   EXECUTE,
"menu",      MENU,
"quit",      QUIT,
"ignore",    IGNORE,
"attribute", ATTRIBUTE,
"visible",   VISIBLE,
"invisible", INVISIBLE,
"end",       END,
NULL, 0,
};

int keyword(string)
   char *string;
  {
   struct keyword *ptr = keywords;

   while(ptr->name != NULL)
      if(strcmp(ptr->name,string) == 0)
        {
   return ptr->value;
}
   else
       ptr++;

   return 0; /* 不匹配 */
}
```

例4-8中的实现使用静态表来标识关键字。这种形式总是比较慢，因为lex词法分析程序的速度不取决于模式的数量或复杂性。在这里介绍它只是因为它展示了一

种有用的技术，可以在需要扩展语言词汇时使用。那样的话，可以为所有的关键字应用一个查找机制，并且在必要时向表中添加新的关键字。

逻辑上，将处理 MGL 规范文件的工作分成几个部分：

**初始化** 初始化所有的内部数据表，给出生成的代码需要的任何前同步码。

**屏幕开始处理** 建立新的屏幕表条目，向名字列表添加屏幕名，并给出初始的屏幕代码。

**屏幕处理** 当遇到每个独立的项目时，处理它；当看到标题行时，向标题列表中添加它们，并且当看到新的菜单项目时，向项目列表添加它们。

**屏幕结束处理** 当看到 **end** 语句时，处理读取屏幕描述时构建的数据结构并给出屏幕的代码。

**结束** "清除"内部语句，给出任何最终的代码，并确保正常结束；如果有问题，就向用户报告问题。

## 初始化

当任何编译程序开始工作时必须执行一些工作。例如，内部数据结构必须被初始化，回忆一下例4-8中的使用关键字查找模式而不是例4-7中使用的硬编码的关键字识别模式的情况。在具有作为初始化的一部分的符号表的更复杂的应用程序中，应该像例3-10那样将关键字插入符号表中。

主函数 **main()**例程简单地这样开始：

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

还必须能通过给出一个文件名来调用编译程序。因为 lex 读取 **yyin** 并向 **yyout** 写入，而它们默认地被赋值为标准输入和标准输出，我们可以将它们重新绑定（attach）输入和输出文件以获取适当的动作。为了改变输入或输出，使用*fopen()*从标准 I/O 库中打开想要的文件并将结果赋给 **yyin** 或 **yyout**。

如果用户调用没有参数的程序，那么我们写出到一个默认文件 *screen.out*，并从标准输入 *stdin* 中读取。如果用户调用具有一个参数的程序，那么我们仍然写入 *screen.out* 并将命名的文件用做输入。如果用户调用具有两个参数的程序，那么用第一个参数作为输入文件，第二参数作为输出文件。

从 **yyparse()**返回后，执行后处理过程，然后检查以确保在没有错误的条件下结束。然后清除并退出。

例 4-9 展示最终结果 **main()**例程。

例 4-9：MGL main()例程

```
char *progname = "mgl";
int lineno = 1;

#define DEFAULT_OUTFILE "screen.out"

char *usage = "%s: usage [infile] [outfile]\n";

main(int argc, char **argv)
{
	char *outfile;
	char *infile;
	extern FILE *yyin, *yyout;

	progname = argv[0];

	if(argc > 3)
	{
		fprintf(stderr,usage, progname);
		exit(1);
	}
	if(argc > 1)
	{
		infile = argv[1];
		/* 打开后读 */
		yyin = fopen(infile,"r");
		if(yyin == NULL) /* 打开失败 */
		{
			fprintf(stderr,"%s: cannot open %s\n",
				progname, infile);
```

```
                exit(1);
            }
        }

        if(argc > 2)
        {
            outfile = argv[2];
        }
        else
        {
                outfile = DEFAULT_OUTFILE;
        }

        yyout = fopen(outfile,"w");
        if(yyout == NULL) /* 打开失败 */
        {
                fprintf(stderr,"%s: cannot open %s\n",
                progname, outfile);
            exit(1);
        }

        /* 通常与yyin和yyout的交互从这里开始 */

        yyparse();

        end_file(); /* 写出最终信息 */

        /* 现在检测EOF条件 */
        if(!screen_done) /* 位于屏幕中心 */
        {
            warning("Premature EOF",(char *)0);
            unlink(outfile); /* 删除坏文件 */
            exit(1);
        }
        exit(0); /* 没有错误 */
}

warning(char *s, char *t) /* 显示警告信息 */
{
        fprintf(stderr, "%s: %s", progname, s);
        if (t)
            fprintf(stderr, " %s", t);
        fprintf(stderr, " line %d\n", lineno);
}
```

# 屏幕处理

一旦初始化编译程序并打开文件，就进入菜单生成程序的真正工作 —— 处理菜单描述。第一个规则 **screens** 要求没有动作。**screen** 规则分解成 **screen_name**、**screen_contents** 和 **screen_terminator**。**screen-name** 首先引起我们的注意:

```
screen_name:  SCREEN ID
            | SCREEN
            ;
```

在名字副本中插入特定的名字；在没有指定名字的情况下，使用名字 “**default**”。规则如下:

```
screen_name:  SCREEN ID  { start_screen($2); }
            | SCREEN     { start_screen(strdup("default")); }
            ;
```

（我们需要调用 **strdup()**以与第一条规则相一致、第一条规则传递由词法分析程序动态分配的字符串。）**start_screen** 例程将名字输入屏幕列表并开始产生代码。

例如，如果输入文件出现 “screen first”，那么 **start_screen** 例程将产生下列代码:

```
/* screen first */
menu_first()
{
        extern struct item menu_first_items[];

        if(!init) menu_init();

        clear();
        refresh();
```

当处理菜单规范时，下一个对象是标题行:

```
title: TITLE QSTRING
      ;
```

调用 **add_title()**，它计算标题行的位置:

```
title: TITLE QSTRING { add_title($2); }
          ;
```

标题行的示例输出如下:

```
move(0,37);
addstr("First");
refresh();
```

通过定位光标并打印出指定的引用字符串(使用基本的中心对齐)来添加标题行。这个代码能重复用于遇到的每个标题行,惟一的改变是用于产生**move()**调用的行数。

为了论证这件事,生成一个具有额外标题行的菜单描述:

```
screen first
title "First"
title "Copyright 1992"

item "first" command first action ignore
      attribute visible
item "second" command second action execute "/bin/sh"
      attribute visible
end first

screen second
title "Second"
item "second" command second action menu first
      attribute visible
item "first" command first action quit
      attribute invisible
end second
```

输出的标题行如下:

```
move(0,37);
addstr("First");
refresh();
move(1,32);
addstr("Copyright 1992");
refresh();
```

一旦看到项目行列表，就获取独立的条目并构建相关动作的内部表。一直继续直到看到语句**end first**，执行后处理并结束构建屏幕。为了构建菜单项目表，向**item**规则中添加下列动作：

```
line: ITEM qstring command ACTION action attribute
        {
          item_str = $2;
          add_line($5, $6);
          $$ = ITEM;
        }
        ;
```

**command**、**action**和**attribute**的规则主要存储后面使用的标记值：

```
command: /* 空值 */ { cmd_str = strdup(""); }
       | COMMAND id  { cmd_str = $2; }
       ;
```

**command**可以为空或一个特殊的命令。在第一种情况下，保存了命令名（采用**strdup()**与下一条规则保持一致）的空字符串，而在第二种情况下，在**cmd_str**中保存命令的标识符。

**action**规则和相关的动作更复杂，部分是由于大量可能的变化：

```
action: EXECUTE qstring
          { act_str = $2;
          $$ = EXECUTE;
          }
       | MENU id
          { /* 生成 'menu_' $2 */
          act_str = malloc(strlen($2) + 6);
          strcpy(act_str,"menu_");
            strcat(act_str, $2);
          free($2);
          $$ = MENU;
        }
       | QUIT { $$ = QUIT; }
       | IGNORE { $$ = IGNORE; }
       ;
```

最后，**attribute** 规则比较简单，因为惟一的语义值是由标记呈现的：

```
attribute:  /* 空值 */            { $$ = VISIBLE; }
          | ATTRIBUTE VISIBLE     { $$ = VISIBLE; }
          | ATTRIBUTE INVISIBLE   { $$ = INVISIBLE; }
          ;
```

**action** 规则和 **attribute** 规则的返回值将传递给 **add_line** 例程；这个调用以不同静态字符串指针的内容，加上两个返回值作为参数，在内部状态表中创建一个条目。

根据看到的 **end first** 语句，我们必须进行屏幕的最终处理。从示例输出中，我们完成 **menu_first** 例程：

```
            menu_runtime(menu_first_items);
}
```

实际的菜单项目写进 **menu_first_items** 数组中：

```
/* 结束首屏 */
struct item menu_first_items[]={
      {"first","first",271,"",0,273},
      {"second","second",267,"/bin/sh",0,273},
      {(char *)0, (char *)0, 0, (char *)0, 0, 0},
};
```

运行时例程 **menu_runtime** 将显示独立的项目，它包含在生成的文件中，作为结尾的部分代码。

# 结束

处理单个屏幕的最后阶段是看到屏幕的结束。回忆 **screen** 规则：

```
screen:   screen_name screen_contents screen_terminator
        | screen_name screen_terminator
        ;
```

这个语法期待看到 **screen_terminator** 规则:

```
screen_terminator:  END ID
                  | END
                  ;
```

添加对后处理程序的调用进行屏幕结束的后处理(不是本节后面讨论的文件结束的后处理)。作为结束的规则如下:

```
screen_terminator: END id { end_screen($2); }
                 | END { end_screen(strdup("default")); }
                 ;
```

它使用屏幕名字作为参数调用 **end_screen** 例程,或者如果没有名字就用 "default"。这个例程验证屏幕名。例 4-10 展示了实现它的代码。

**例 4-10: 屏幕结束代码**

```
/*
 * end_screen:
 * 结束屏幕,打印出后同步信号
 */

end_screen(char *name)
{

      fprintf(yyout, "menu_runtime(menu_%s_items);\n",name);

      if(strcmp(current_screen,name) != 0)
      {
           warning("name mismatch at end of screen",
               current_screen);
      }
      fprintf(yyout, "}\n");
      fprintf(yyout, "/* end %s */\n",current_screen);

      process_items();

      /* 写出文件的初始化代码 */
      if(!done_end_init)
      {
```

```
                done_end_init = 1;
                dump_data(menu_init);
        }

        current_screen[0] = '\0'; /* 没有当前屏幕 */

        screen_done = 1;

        return 0;
}
```

这个例程将屏幕名不匹配作为非致命错误处理。因为错误匹配不会导致编译程序内部的问题，所以可以报告给用户而不必结束。

这个例程通过使用 **process_items()**处理独立的项目条目来处理由 **add_item()**调用产生的数据。然后，它调用**dump_data**来写出一些初始化例程，这些初始化例程是真正的构成 MGL 编译程序的字符串的静态数组。在几个不同位置调用 **dump_data()**以转储不同的代码段到输出文件。另一种可替代的途径是从包含样板（boiler-plate）代码的“框架”文件中拷贝这些代码段，正如lex和yacc的一些版本所做的那样。

后处理发生在读取和分析所有的输入之后。在成功地完成 **yyparse()**之后，通过调用 **end_file()**由 **main()**例程来进行后处理。实现为:

```
/*
 * 这个例程写出运行时的支持
 */

end_file()
{

        dump_data(menu_runtime);
}
```

这个例程包含对**dump_data()**的单个调用来写出运行时例程，它像初始化代码那样存储在编译程序中的静态数组中。处理样板代码的所有例程在设计中充分地模块化，这些模块可以重新编写为使用框架文件。

一旦这个例程被调用，**main**例程通过检查以决定输入的结尾是否位于有效点上来结束，即在屏幕的边界，并且没有产生错误消息。

# MGL 代码示例

我们已经构建了一个简单的编译程序，现在来论证一下它的基本工作。MGL的实现由三部分组成：yacc语法、lex规范和支持代码例程。yacc语法的最终形式、lex词法分析程序和支持代码展示在附录九“MGL 编译程序代码”中。

我们不是要设计真正使用中的非常健壮的示例代码，而主要集中在开发第一阶段的实现。然而，作为结果的编译程序将产生一个完全的功能性菜单编译程序。下面是示例输入文件：

```
screen first
title "First"

item "dummy line" command dummy action ignore
      attribute visible
item "run shell" command shell action execute "/bin/sh"
      attribute visible

end first

screen second

title "Second"

item "exit program" command exit action quit
      attribute invisible
item "other menu" command first action menu first
      attribute visible

end second
```

当描述文件被编译程序处理时，会得到下面的输出文件：

```
/*
 * 由MGL创建: Thu Aug 27 18:03:33 1992
 */
```

```
/* 初始化信息 */
static int init;

#include <curses.h>
#include <sys/signal.h>
#include <ctype.h>
#include "mglyac.h"

/* 用于存储菜单项目的结构 */
struct item {
	char *desc;
	char *cmd;
	int action;
	char *act_str;		/* 执行字符串 */
	int (*act_menu)();	/* 调用适当的函数 */
	int attribute;
};

/* 第一屏 */
menu_first()
{
	extern struct item menu_first_items[];
	if(!init) menu_init();

	clear();
	refresh();
	move(0,37);
	addstr("First");
	refresh();
	menu_runtime(menu_first_items);
}
/* 第一屏结束 */
struct item menu_first_items[]={
{"dummy line","dummy",269,"",0,271},
{"run shell","shell",265,"/bin/sh",0,271},
{(char *)0, (char *)0, 0, (char *)0, 0, 0},
};

menu_init()
{
	void menu_cleanup();

	signal(SIGINT, menu_cleanup);
```

```
        initscr();
        crmode();
}

menu_cleanup()
{
        mvcur(0, COLS - 1, LINES - 1, 0);
        endwin();
}

/* 第二屏 */
menu_second()
{
        extern struct item menu_second_items[];

        if(!init) menu_init();

        clear();
        refresh();
        move(0,37);
        addstr("Second");
        refresh();
        menu_runtime(menu_second_items);
}
/* 第二屏结束 */
struct item menu_second_items[]={
{"exit program","exit",268,"",0,272},
{"other menu","first",267,"",menu_first,271},
{(char *)0, (char *)0, 0, (char *)0, 0, 0},
};

/* 运行时 */
menu_runtime(items)
struct item *items;
{
        int visible = 0;
        int choice = 0;
        struct item *ptr;
        char buf[BUFSIZ];

        for(ptr = items; ptr->desc != 0; ptr++) {
            addch('\n'); /* 跳过一行 */
            if(ptr->attribute == VISIBLE) {
```

```
            visible++;
            printw("\t%d) %s",visible,ptr->desc);
        }
    }

    addstr("\n\n\t"); /* tab使输出看起来很好 */
    refresh();

    for(;;)
    {
        int i, nval;

        getstr(buf);

        /* 数字选择? */
        nval = atoi(buf);

        /* 命令选择? */
        i = 0;
        for(ptr = items; ptr->desc != 0; ptr++) {
            if(ptr->attribute != VISIBLE)
                continue;
            i++;
            if(nval == i)
                break;
            if(!casecmp(buf, ptr->cmd))
                break;
        }

        if(!ptr->desc)
            continue;  /* 没有匹配 */

        switch(ptr->action)
        {
        case QUIT:
            return 0;
        case IGNORE:
            refresh();
            break;
        case EXECUTE:
            refresh();
            system(ptr->act_str);
            break;
        case MENU:
```

```
                refresh();
                (*ptr->act_menu)();
                break;
            default:
                printw("default case, no action\n");
                refresh();
                break;
            }
            refresh();
        }
}

casecmp(char *p, char *q)
{
        int pc, qc;

        for(; *p != 0; p++, q++) {
            pc = tolower(*p);
            qc = tolower(*q);

            if(pc != qc)
                break;
        }
        return pc-qc;
}
```

依次，编译由编译程序产生的代码并写到*first.c*，使用下列命令：

```
$ cat >> first.c
main()
{
menu_second();
menu_cleanup();
}
^D
$ cc -o first first.c -lcurses -ltermcap
$
```

必须向产生的代码添加**main()**例程；在MGL的修订本中，它也许包括在主循环中，用于提供调用的例程名的命令行选项或规范选项。因为使用yacc编写语法，所以修订很容易。例如，可以修订**screens**规则为：

```
screens:  /* nothing */
        | preamble screens screen
        | screens screen
        ;

preamble: START ID
        | START DEFAULT
        ;
```

为 **START** 和 **DEFAULT** 添加适当的关键字。

运行 MGL 产生的屏幕代码，我们看到下列菜单屏幕：

```
                         Second
1) other menu
```

我们看到一个可见的菜单条目，并且当输入“1”或“first”时移到第一个菜单：

```
                         First
1) dummy line
2) run shell
```

# 练习

1. 添加命令标识主屏幕并产生一个主例程，如前所述。

2. 改善屏幕处理：用 CBREAK 模式读取字符而不是每次一行，允许更灵活的命令处理，例如，接受惟一的命令前缀而不要求整个命令，允许应用代码设置和清除不可见的属性，等等。

3. 扩展屏幕定义语言，让标题和命令名从主程序中的变量产生，例如：

```
screen sample
title $titlevar

item $label1 command $cmd1 action ignore
   attribute visible

end sample
```

其中，**titlevar** 和 **label1** 是主程序中的字符数组或指针。

4. （术语工程）设计一种指定下拉式或弹出式菜单的语言。实现几种基于同一语法分析程序的翻译程序，这样可以使用同一菜单规范创建在不同环境中运行的菜单，例如具有 curses、Motif 和 Open Look 的终端。

5. yacc 通常用于实现对于一个应用领域特定的“小语言”，并将它翻译成低级的、更一般的语言。MGL 将菜单规范转换成 C 语言，*eqn* 将方程式语言转换成原始的编辑和格式化程序。可以从小语言中获得好处的其他应用领域有哪些？用 lex 和 yacc 设计和实现几个。

# 第五章 分析SQL

本章内容：

- SQL的要点概述
- 语法检查程序
- 语法分析程序
- 嵌入式SQL
- 练习

SQL（代表“结构化查询语言”，通常发音为*sequel*）是处理关系数据库最常用的语言（注1）。首先，开发一个检测输入语法的SQL语法分析程序，但是不以它做任何事情。然后将它转换成嵌入在C程序中的SQL预处理程序。

这个语法分析程序基于C.J.Date编写的《A Guide to the SQL Standard》（第二版，Addison-Wesley出版社1989年出版）一书中的SQL定义。Date的描述采用Backus-Naur范式（BNF），它是用于编写正规语言描述的标准形式。除了标点符号以外，yacc的输入语法类似于BNF，所以在许多地方，直译BNF得到相应的yacc规则是足够的。在多数情况下，使用与Date相同的符号名，尽管在有些地方为了使语法适用于yacc不这样做。

SQL的最终定义是标准的文档——ANSI X3.135-1989（它定义SQL本身）和ANSI X3.168-1989（它定义在其他程序设计语言中嵌入SQL的方式）。

注1：　SQL是数据库的Fortran，没有人特别喜欢它，这个语言丑陋而且特别，每个数据库都支持它，而且我们都用它。

# SQL 的要点概述

SQL 是用于关系数据库的专用语言。它处理数据库表中的数据，而不是处理内存中的数据（偶尔才引用内存）。

## 关系数据库

关系数据库是表的集合，类似于文件。每个表都包含行（row）和列（column），类似于记录和字段。表中的行没有任何特定的顺序。通过给出每列的名字和类型可以创建一组表：

```
CREATE SCHEMA
AUTHORIZATION JOHNL

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
)
```

这个语法是完全格式自由的，并且同一件事通常有几种不同的语法编写方式，注意给出的PRIMARY KEY说明符的两种不同的方式。（表中的主关键字（primary key）是惟一指定一行的一个列，或一组列。）图 5-1 展示了装载数据后刚刚创建的两个表。

使用数据库时，必须告诉数据库想要从表中得到什么。数据库算出如何得到它。一组想要的数据的规范说明就是查询（query）。例如，使用图 5-1 中的两个表，要得到水果列表，可以写成：

*Foods*

| *name* | *type* | *flavor* |
|---|---|---|
| peach | fruit | sweet |
| tomato | fruit | savory |
| lemon | fruit | sour |
| lard | fat | bland |
| cheddar | fat | savory |

*Courses*

| *course* | *flavor* | *sequence* |
|---|---|---|
| salad | savory | 1 |
| main | savory | 2 |
| dessert | sweet | 3 |

图 5-1　两个关系表

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

响应是:

| *name* | *flavor* |
|---|---|
| peach | sweet |
| tomato | savory |
| lemon | sour |

也可以提出跨越多个表的问题。为了得到适合每餐的食物列表，可以写成:

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

响应是:

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

当列出列名时，如果列名是明确的就可以省去表名。

## 处理关系

SQL有丰富的表处理命令集合。用SELECT、INSERT、UPDATE和DELETE命令可以读取并写出独立的行。更常见的是，需要为一组行中的每个行做些事情。在那种情况下，SELECT的不同变体定义了游标（cursor），游标是能单步遍历一组行以便一次处理一行的一种文件指针。然后使用OPEN和CLOSE命令找到相应的行并使用FETCH、UPDATE CURRENT和DELETE CURRENT命令对这些行做一些操作。当进行事务处理时，COMMIT和ROLLBACK命令完成或异常中止一组命令。

SELECT语句有一个非常复杂的语法，可以使你寻找列中的值、比较列、做算术，并计算最小值、最大值、平均值和总和。

## 使用SQL的三种方法

在SQL的最初版本中，用户在文件中或直接在终端上键入命令并立即接收响应。有时用户仍然使用这种方式创建表并进行调试，但对大多数的应用，SQL命令来源于程序内部而且结果被返回给那些程序。

将SQL与常规语言结合在一起的第一种途径是SQL模块语言，只需要在SQL中定义少许程序就可以从常规语言中调用SQL模块语言。例5-1定义了一个游标和使用它的三个过程。

例 5-1：SQL 模块语言的示例

```
MODULE LANGUAGE C AUTHORIZATION JOHNL

DECLARE flav CURSOR FOR
      SELECT     Foods.name, Foods.type
      FROM Foods
      WHERE Foods.flavor = myflavor
      -- myflavor is defined below

PROCEDURE open_flav
      SQLCODE
      myflavor CHAR(6) ;
      OPEN flav

PROCEDURE close_flav
      SQLCODE ;
      CLOSE flav

PROCEDURE get_flav
      SQLCODE
      myname CHAR(8)
      mytype CHAR(5) ;
      FETCH flav INTO myname, mytype
```

在 C 程序中，通过编写下面的代码可以使用模块中的这些例程：

```
char flavor[6], name[8], type[5];

main()
{
      int icode;

      scanf("%s", flavor);
      open_flav(&icode, flavor);
      for(;;) {
            get_flav(&icode, name, type);
            if(icode != 0)
                  break;
            printf("%8.8s %5.5s\n", name, type);
      }
      close_flav(&icode);
}
```

这种方法可以工作，但是存在使用的瓶颈，因为所编写的每条SQL语句都必须打包到一个小例程。用户真正使用的方法是嵌入式SQL，这种嵌入式SQL允许用户将SQL程序块正确地放置在自己的程序中。每条SQL语句都由“EXEC SQL”引入并且以分号结束。对宿主语言变量的引用前面以冒号引导。例5-2展示了以嵌入式SQL编写的同样的程序。

例5-2：嵌入式SQL的示例

```
char flavor[6], name[8], type[5];
int SQLCODE;      /* 全局状态变量 */

EXEC SQL DECLARE flav CURSOR FOR
      SELECT Foods.name, Foods.type
      FROM Foods
      WHERE Foods.flavor = :flavor ;

main()
{
      scanf("%s", flavor);
      EXEC SQL OPEN flav ;
      for(;;) {
            EXEC SQL FETCH flav INTO :name, :type ;
            if(SQLCODE != 0)
                  break;
            printf("%8.8s %5.5s\n", name, type);
      }
      EXEC SQL CLOSE flav ;
}
```

为了编译这个程序，用户通过SQL预处理程序来运行它，SQL预处理程序将SQL代码转换成对C例程的调用，然后编译纯粹的C程序。本章后面将编写这样一个预处理程序的简单版本。

# 语法检查程序

用yacc编写语法检查程序很容易。试着分析一些程序，如果分析可以工作，说明语法正确。(如果语法分析程序中有错误校正，那么事情就不会这么简单。详细资料参见第九章。) 为SQL构建一个语法检查程序，这个任务几乎全部是编写yacc

语法完成的。因为SQL语法如此复杂，所以在附录十中将重新介绍整个yacc语法，及对语法中所有符号的交义引用。

## 词法分析程序

首先，对于SQL使用的标记要使用词法分析程序。语法是自由格式的，忽略空白(它仅分隔单词)。具有相当长的但却固定的保留字集合。其他的标记是很传统的：名字、字符串、数字和标点符号。注释是Ada风格的，从一对短划线开始到行尾。例5-3展示了SQL词法分析程序。

例5-3：第一个SQL词法分析程序

```
%{
#include "sql1.h"
#include <string.h>

int lineno = 1;
void yyerror(char *s);
%}

%e 1200
%%
        /* 文字关键字标记 */

ADA          { return ADA; }
ALL          { return ALL; }
AND          { return AND; }
AVG          { return AMMSC; }
MIN          { return AMMSC; }
MAX          { return AMMSC; }
SUM          { return AMMSC; }
COUNT        { return AMMSC; }
ANY          { return ANY; }
AS           { return AS; }
ASC          { return ASC; }
AUTHORIZATION      { return AUTHORIZATION; }
BETWEEN            { return BETWEEN; }
BY           { return BY; }
C            { return C; }
CHAR(ACTER)?      { return CHARACTER; }
CHECK        { return CHECK; }
```

```
CLOSE		{ return CLOSE; }
COBOL		{ return COBOL; }
COMMIT			{ return COMMIT; }
CONTINUE	{ return CONTINUE; }
CREATE			{ return CREATE; }
CURRENT			{ return CURRENT; }
CURSOR			{ return CURSOR; }
DECIMAL			{ return DECIMAL; }
DECLARE			{ return DECLARE; }
DEFAULT			{ return DEFAULT; }
DELETE			{ return DELETE; }
DESC		{ return DESC; }
DISTINCT	{ return DISTINCT; }
DOUBLE			{ return DOUBLE; }
ESCAPE			{ return ESCAPE; }
EXISTS			{ return EXISTS; }
FETCH		{ return FETCH; }
FLOAT		{ return FLOAT; }
FOR		{ return FOR; }
FOREIGN			{ return FOREIGN; }
FORTRAN			{ return FORTRAN; }
FOUND		{ return FOUND; }
FROM		{ return FROM; }
GO[ \t]*TO	{ return GOTO; }
GRANT		{ return GRANT; }
GROUP		{ return GROUP; }
HAVING			{ return HAVING; }
IN		{ return IN; }
INDICATOR	{ return INDICATOR; }
INSERT			{ return INSERT; }
INT(EGER)?	{ return INTEGER; }
INTO		{ return INTO; }
IS		{ return IS; }
KEY		{ return KEY; }
LANGUAGE	{ return LANGUAGE; }
LIKE		{ return LIKE; }
MODULE			{ return MODULE; }
NOT		{ return NOT; }
NULL		{ return NULLX; }
NUMERIC			{ return NUMERIC; }
OF		{ return OF; }
ON		{ return ON; }
OPEN		{ return OPEN; }
OPTION			{ return OPTION; }
OR		{ return OR; }
```

```
ORDER		{ return ORDER; }
PASCAL		{ return PASCAL; }
PLI		{ return PLI; }
PRECISION	{ return PRECISION; }
PRIMARY		{ return PRIMARY; }
PRIVILEGES	{ return PRIVILEGES; }
PROCEDURE	{ return PROCEDURE; }
PUBLIC		{ return PUBLIC; }
REAL		{ return REAL; }
REFERENCES	{ return REFERENCES; }
ROLLBACK	{ return ROLLBACK; }
SCHEMA		{ return SCHEMA; }
SELECT		{ return SELECT; }
SET		{ return SET; }
SMALLINT	{ return SMALLINT; }
SOME		{ return SOME; }
SQLCODE		{ return SQLCODE; }
TABLE		{ return TABLE; }
TO		{ return TO; }
UNION		{ return UNION; }
UNIQUE		{ return UNIQUE; }
UPDATE		{ return UPDATE; }
USER		{ return USER; }
VALUES		{ return VALUES; }
VIEW		{ return VIEW; }
WHENEVER	{ return WHENEVER; }
WHERE		{ return WHERE; }
WITH		{ return WITH; }
WORK		{ return WORK; }

	/* 标点符号 */
"="	|
"<>"	|
"<"	|
">"	|
"<="	|
">="		{ return COMPARISON; }

[-+*/:(),.;]		{ return yytext[0]; }

	/* 名称 */
[A-Za-z][A-Za-z0-9_]*	{ return NAME; }

	/* 数字 */
```

```
[0-9]+        |
[0-9]+"."[0-9]* |
"."[0-9]*    { return INTNUM; }

[0-9]+[eE][+-]?[0-9]+  |
[0-9]+"."[0-9]*[eE][+-]?[0-9]+ |
"."[0-9]*[eE][+-]?[0-9]+      { return APPROXNUM; }

        /* 字符串 */

'[^'\n]*'    {
            int c = input();
            unput(c);   /* 仅仅查看一下 */
            if(c != '\'')
                return STRING;
            else
                yymore();
        }

'[^'\n]*$    { yyerror("Unterminated string"); }

        /* 空白 */
\n            lineno++;

[ \t\r]+       ;      /* 空白 */

"--".*                 ;      /* 注释 */

       /* 其他 */
.           yyerror('invalid character);
%%

void
yyerror(char *s)
{
            printf("%d: %s at %s\n", lineno, s, yytext);
}

main(int ac, char **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");
} /* 主例程 */
```

词法分析程序从几个包含文件开始，特别是*sql1.h*，它是由yacc生成的标记名定义文件（从默认的*y.tab.h*中重新命名它）。所有的保留字在语法分析程序中都为独立的标记，因为这是最容易做的事情。注意CHARACTER和INTEGER可以缩写成CHAR和INT，而且GOTO可以写成一个或两个单词。保留字AVG、MIN、MAX、SUM和COUNT都转换成**AMMSC**标记；在SQL预处理程序中，使用标记值记住它们各是哪个词。

接下来是标点符号标记，包括使用同一个模式匹配所有单个字符操作符的常用技巧。名字以字母开头并且由字母、数字和下划线组成。这种模式必须跟在所有的保留字后面，以保证这些保留字模式的高优先级。

SQL定义精确的数字（它们具有小数点，但没有明确的指数），也可以定义近似数（它们有指数）。独立的模式可以区别这两种形式。

SQL字符串包围在单引号内，使用一对引号代表字符串中的单引号。第一种字符串模式匹配引号中不包含嵌入式引号的字符串。它的动作程序使用**input()**和**unput()**取得下一个字符并查看它是否为另一个引号（意思是找到两个引号，而不是字符串的结尾）。如果这样的话，它使用**yymore()**向标记附加另一个引用字符串。下一种模式捕获非终结的字符串并且在它看到时打印诊断结果。

最后几个模式跳过空白（当空白是换行符时计算行数），跳过注释，并且如果输入中出现无效的字符时会发出“抱怨”。

## 错误和主程序

**yyerror()**的这种版本报告当前的行数、当前的标记和错误消息。这个简单的程序通常用于报告有用的错误的所有内容。因为它需要引用**yytext**，并且只在词法分析程序的源文件中**yytext**才有定义，所以为了得到最好的可移植性，我们将它放

入词法分析程序中。(lex的不同版本将**yytext**定义为一个数组或一个指针，所以不能编写在任何地方都兼容的对它的引用。)

**main()**例程打开在命令行上命名的文件(若有的话)，随后调用语法分析程序。当语法分析程序返回时，返回值报告分析成功还是失败。虽然在它本身的文件中放置**main()**和声明**yyin**为一个外部“*FILE**”可以工作得同样好，但我们仍然在这里放置**main()**，因为**yyin**已经在这里定义了。

# 语法分析程序

SQL语法分析程序比到目前为止所看到的任何语法分析程序都大，但是我们可以分段理解。

## 定义

例 5-4 展示了语法分析程序的定义部分。

例 5-4：第一个 SQL 语法分析程序的定义部分

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

%token NAME
%token STRING
%token INTNUM APPROXNUM

        /* 操作符 */

%left OR
%left AND
%left NOT
%left COMPARISON /* = <> < > <= >= */
%left '+' '-'
%left '*' '/'
```

```
%nonassoc UMINUS

        /* 文字关键字标记 */

%token ALL AMMSC ANY AS ASC AUTHORIZATION BETWEEN BY
%token CHARACTER CHECK CLOSE COMMIT CONTINUE CREATE CURRENT
%token CURSOR DECIMAL DECLARE DEFAULT DELETE DESC DISTINCT DOUBLE
%token ESCAPE EXISTS FETCH FLOAT FOR FOREIGN FOUND FROM GOTO
%token GRANT GROUP HAVING IN INDICATOR INSERT INTEGER INTO
%token IS KEY LANGUAGE LIKE MODULE NULLX NUMERIC OF ON
%token OPEN OPTION ORDER PRECISION PRIMARY PRIVILEGES PROCEDURE
%token PUBLIC REAL REFERENCES ROLLBACK SCHEMA SELECT SET
%token SMALLINT SOME SQLCODE SQLERROR TABLE TO UNION
%token UNIQUE UPDATE USER VALUES VIEW WHENEVER WHERE WITH WORK
%token COBOL FORTRAN PASCAL PLI C ADA
```

首先是值union类型的定义。字段**intval**和**floatval**处理整数和浮点数。(实际上，因为SQL的精确数字包括小数，后面我们会将所有的数值都存在**floatval**中。)字符串以**strval**返回，尽管在这种版本的语法分析程序中，我们做这件事不用费力。最后，**subtok**保存代表多个输入标记的标记的子标记代码，例如，AVG、MIN、MAX、SUM和COUNT，尽管在语法检查程序中我们做这件事不用费力。

接下来是语法中使用的标记的定义。标记有NAME、STRING、INTNUM和APPROXNUM，所有这些都可以在词法分析程序中看到。%**left**声明设置所有操作符的优先级和结合规则。声明需要设置优先级的“+-*/”文字标记和只在后面的%**prec**子句中使用的伪标记UMINUS。

最后是SQL的所有保留字的标记定义。

## 最高层的规则

例5-5展示了语法分析程序的最高层的规则。

**例5-5：在第一个SQL语法分析程序中的最高层的规则**

```
sql_list:
                sql ';'
        |       sql_list sql ';'
        ;
```

```
        . . .
        /* 模式定义语言 */
        /* 说明其他 'sql:' 规则在后面的语法中出现 */
sql:    schema
        ;
        . . .
        /* 模块语言 */
sql:        module_def
        ;
        . . .
        /* 操纵语句 */

sql:        manipulative_statement
        ;
```

起始规则是**sql_list**（一列**sql**规则），每条**sql**规则都是一种语句。有三种不同的语句：定义表的**schema**语句，模块定义语句**module_def**及包括所有语句（例如实际处理数据库的OPEN、CLOSE和SELECT）的**manipulative_statement**。在每个语法部分的开始放置**sql**规则。(yacc不关心所有的具有相同左侧的规则是否一起出现在规范中，在本例中不一起出现是比较容易的，如果你这样做了，务必要包括解释你做了什么的注释。)

## 模式子语言

模式（schema）子语言定义数据库表。如例5-6所示，以CREATE SCHEMA AUTHORIZATION "user" 开始，后面跟着可选的模式元素列表。

例5-6：模式子语言，上面部分

```
schema:
            CREATE SCHEMA AUTHORIZATION user opt_schema_element_list
        ;

user:       NAME
        ;

opt_schema_element_list:
            /* 空值 */
        |   schema_element_list
        ;
```

```
schema_element_list:
                schema_element
        |       schema_element_list schema_element
        ;

schema_element:
                base_table_def
        |       view_def
        |       privilege_def
        ;
```

例5-6展示了最高层的模式定义规则。有一条说明**user**是一个NAME的规则。在语句构成上，可以在**schema**规则中直接使用NAME，但是这条独立的规则为动作代码提供了便利的位置，这个动作代码用来验证所给出的名字是可信的用户名字。

模式元素列表语法使用大量的规则，但是它们并不复杂。**opt_schema_element_list**要么是**schema_element_list**，要么什么也不是，**schema_element_list**是一列**schema_element**，而且**schema_element**是三种定义中的一种。我们尽量将这些规则简化为：

```
opt_schema_element_list:
                /* 空值 */
        |       opt_schema_element_list base_table_def
        |       opt_schema_element_list view_def
        |       opt_schema_element_list privilege_def
        ;
```

尽管在添加动作代码时，越复杂的版本越容易使用。

## 基表

例5-7展示了基表语言。

**例5-7：模式子语言，基表**

```
base_table_def:
                CREATE TABLE table '(' base_table_element_commalist ')'
        ;
```

```
table:
                NAME
        |       NAME '.' NAME
        ;

base_table_element_commalist:
                base_table_element
        |       base_table_element_commalist ',' base_table_element
        ;

base_table_element:
                column_def
        |       table_constraint_def
        ;

column_def:
                column data_type column_def_opt_list
        ;

data_type:
                CHARACTER
        |       CHARACTER '(' INTNUM ')'
        |       NUMERIC
        |       NUMERIC '(' INTNUM ')'
        |       NUMERIC '(' INTNUM ',' INTNUM ')'
        |       DECIMAL
        |       DECIMAL '(' INTNUM ')'
        |       DECIMAL '(' INTNUM ',' INTNUM ')'
        |       INTEGER
        |       SMALLINT
        |       FLOAT
        |       FLOAT '(' INTNUM ')'
        |       REAL
        |       DOUBLE PRECISION
        ;

column_def_opt_list:
                /* 空值 */
        |       column_def_opt_list column_def_opt
        ;

column_def_opt:
                NOT NULLX
        |       NOT NULLX UNIQUE
```

```
            NOT NULLX PRIMARY KEY
        |   DEFAULT literal
        |   DEFAULT NULLX
        |   DEFAULT USER
        |   CHECK '(  search_condition  )'
        |   REFERENCES table
        |   REFERENCES table '(' column_commalist ')'
        ;

table_constraint_def:
            UNIQUE  '(' column_commalist ')'
        |   PRIMARY KEY '(  column_commalist  )'
        |   FOREIGN KEY '(  column_commalist ')'
                REFERENCES table
        |   FOREIGN KEY '(' column_commalist ')'
                REFERENCES table '(' column_commalist ')
        |   CHECK '(' search_condition ')'
        ;

column_commalist:
            column
        |   column_commalist ',' column
        ;

column:         NAME
        ;

literal:
            STRING
        |   INTNUM
            APPROXNUM
        ;
```

基表同样具有大量的语法，但是一旦你读这些部分，它并不复杂。基表定义 **base_table_def** 是 “CREATE TABLE”，表名可以是简单的名字或由用户名限定的名字，而且括号中的基表元素的列表由逗号分开。每个 **base_table_element** 是一个列定义或一个表约束定义。

列定义 **column_def** 是列名、它的数据类型和一列可选的列定义选项。可能的数据类型列表和列定义选项列表很长。每列有一个数据类型，因为有一个对 **data_type** 的引用。这些标记中，一些是保留字，一些是数字，所以像

NUMERIC(5,2)这样的类型匹配第 5 个 **data_type** 规则。对于列定义选项，**column_def_opt_list**允许任意记录中零个或多个选项。这些选项规定列是否可以包含空（未定义的）值，规定值是否必须惟一，设置默认或者是设置有效性检测条件或表间的一致性（REFERENCES）条件。稍后定义**search_condition**，因为它是处理语言的语法结构部分。

用标记**NULLX**表示保留字NULL，因为yacc将所有的标记定义为C预处理程序符号，而符号**NULL**早已在C中有了含义。同理，要避免将标记命名为FILE、BUFSIZ和EOF，以及所有在标准I/O库中由符号先占用的名字。

基表元素还可以是几种形式之一的表约束。SQL语法在这里是冗余的，下面的两种形式相等：

```
thing CHAR(3) NOT NULL UNIQUE

thing CHAR(3),
      UNIQUE ( some_name )
```

第一种形式被解析为具有“NOT NULL”和“UNIQUE”的**column_def**，每个都是**column_def_opt_list**中的**column_def_opt**。第二种形式是后跟**table_constraint_def**的**column_def**，每个都是**base_table_element**而且在**base_table_element_list**中两者组合在一起。

SQL定义禁止取值惟一的列中的NULL，但对于是否必须如此明确地说明是不一致的。由动作代码来识别这两个形式是相等的，因为在语法结构上它们是不同的，而且yacc了解它们的不同。

## 视图定义

视图（view）是由查询定义的虚表，每次应用程序打开视图时，其内容都是执行查询的结果（注 2）。例如，可以在食物表中创建水果的视图：

注 2：这有点过于简单化，因为在许多情况下，可以将行写入视图并且数据库适当地更新基本表。

```
CREATE VIEW fruits (frname, frflavor)
      AS SELECT Foods.name, Foods.flavor
      FROM Foods
      WHERE Foods.type = "fruit"
```

例 5-8 展示了模式视图定义的语法。

例 5-8：模式视图定义

```
view_def:
            CREATE VIEW table opt_column_commalist
            AS query_spec opt_with_check_option
      ;

opt_with_check_option:
            /* 空值 */
      |     WITH CHECK OPTION
      ;

opt_column_commalist:
            /* 空值 */
      |     '(' column_commalist ')'
;
```

视图定义是“CREATE VIEW”、表名、可选的列名列表（默认使用基表中的名字）、关键字“AS”、一种查询规范（稍后定义）和可选的检查选项，检查选项在向视图写入新行时控制错误检查的量。

## 特权定义

例 5-9 展示了特权定义子语言，位于视图定义子语言的最后。

例 5-9：模式特权定义

```
privilege_def:
            GRANT privileges ON table TO grantee_commalist
            opt_with_grant_option
      ;

opt_with_grant_option:
            /* 空值 */
      |     WITH GRANT OPTION
```

```
        ;

privileges:
                ALL PRIVILEGES
        |       ALL
        |       operation_commalist
        ;

operation_commalist:
                operation
                operation_commalist ',' operation
        ;

operation:
                SELECT
        |       INSERT
        |       DELETE
        |       UPDATE opt_column_commalist
        |       REFERENCES opt_column_commalist
        ;

grantee_commalist:
                grantee
        |       grantee_commalist ',' grantee
        ;

grantee:
                PUBLIC
        |       user
        ;
```

表或视图的主人可以给其他用户在他们的表上执行各种操作的权力，例如:

```
GRANT SELECT, UPDATE (address, telephone)
      ON employees TO PUBLIC
GRANT ALL ON foods TO tony, dale WITH GRANT OPTION

GRANT REFERENCES (flavor) ON Foods TO PUBLIC
```

**WITH GRANT OPTION** 允许授权人向其他用户再次授予他们的权力。**REFERENCES** 是创建表所需要的权力，这个表被键入现有表的列中。另外，语法和 **GRANT** 语句的含义相当简单。

## 模块子语言

由于模块语言在实际应用中已经过时，所以这里不再详细介绍它。在附录十中SQL语法的完整列表中可以找到它的yacc定义。

### 游标定义

为了在嵌入式SQL中应用，需要模块语言中的游标定义语句。例5-10展示了游标定义的语法。

例5-10：游标定义

```
cursor_def:
		DECLARE cursor CURSOR FOR query_exp opt_order_by_clause
	;

opt_order_by_clause:
		/* 空值 */
	|	ORDER BY ordering_spec_commalist
	;

ordering_spec_commalist:	/* 定义排序次序 */
		ordering_spec
	|	ordering_spec_commalist ',' ordering_spec
	;

ordering_spec:
		INTNUM opt_asc_desc	/* 通过列号 */
	|	column_ref opt_asc_desc/* 通过列名 */
	;

opt_asc_desc:
		/* 空值 */
	|	ASC
	|	DESC
	;

cursor:		NAME
	;

column_ref:
		NAME		/* 列名 */
```

```
        |     NAME  '.' NAME       /* table.col 或 range.col */
        |     NAME  '.' NAME '.' NAME /* user.table.col */
        ;
```

典型的游标定义如下:

```
DECLARE course_cur CURSOR FOR
      SELECT     ALL
      FROM Courses
      ORDER BY sequence ASC
```

游标定义非常类似于视图定义，两者都是将名字与SELECT查询关联起来。区别是视图是驻留在数据库中的永久对象，而游标是驻留在应用程序中的临时对象。特别是，视图有自己的不同于构建它们的表的特权（这是创建视图的主要原因）。你需要一个游标以在程序中读写数据；为了读写视图，在视图上需要游标。而且，用于定义游标的查询表达式比用于定义视图的查询规范更通用。在下一节中，我们将看到 SELECT 语句与视图和游标的关系。

## 操纵子语言

SQL 的核心是操纵子语言（manipulation sublanguage）：搜索、阅读、插入、删除和更新行和表的命令。

有 11 种不同的操纵语句，如例 5-11 中的规则所示。

例 5-11：操纵子语言，上面部分

```
sql:         manipulative_statement
        ;

manipulative_statement:
             close_statement
        |    commit_statement
        |    delete_statement_positioned
        |    delete_statement_searched
        |    fetch_statement
        |    insert_statement
        |    open_statement
        |    rollback_statement
```

```
        |    select_statement
        |    update_statement_positioned
        |    update_statement_searched
        ;
```

尽管一些语句（特别是**SELECT**语句）涉及到数据库部分的大量工作，但一次只执行一条 SQL 语句。

## 简单的语句

简单的操纵语句如例 5-12 所示。

**例 5-12：简单的操纵语句**

```
open_statement:
            OPEN cursor
        ;

close_statement:
            CLOSE cursor
        ;

commit_statement:
            COMMIT WORK
        ;

rollback_statement:
            ROLLBACK WORK
        ;

delete_statement_positioned:
            DELETE FROM table WHERE CURRENT OF cursor
        ;
```

大部分操纵语句相当简单，例 5-12 全部展示了它们。OPEN 和 CLOSE 类似于打开和关闭文件。DELETE...WHERE CURRENT 删除游标标识的单个记录。

FETCH 语句是在程序中获取数据的主要方式。它的语法比前面的语句稍微复杂些，因为它要说明一列一列地放置数据的位置。

## FETCH 语句

例 5-13 展示了 FETCH 的规则。

**例 5-13：FETCH 语句**

```
fetch_statement:
        FETCH cursor INTO target_commalist
    ;

target_commalist:
        target
    |   target_commalist ',' target
    ;

target:
        parameter_ref
    ;

parameter_ref:
        parameter
        parameter parameter
        parameter INDICATOR parameter
    ;

parameter:
    |   ':' NAME /* 嵌入的参数 */
    ;
```

FETCH由于所有可能的目标而变得复杂。每个目标是一个或两个参数，可选的第二个参数作为一个指示变量用来说明存储的数据是有效的还是空的。在嵌入式SQL中，参数是前面带有冒号的宿主语言变量名。在模块语言的一个过程中，参数还是模块头中作为参数声明的名字，但是在那种情况下，词法分析程序必须区分参数名和列及区域名，否则yacc会因为词法分析程序不能表明是哪个名字而产生许多移进/归约冲突。为了使语法检查程序保持相对简单，不考虑模块语言名字。在工作的示例中，词法分析程序查找符号表中的每个名字并返回一个不同的标记，例如，模块参数名**MODPARAM**，并且添加一条规则：

```
parameter: MODPARAM ;
```

## INSERT 语句

INSERT 语句如例 5-14 所示。

*例 5-14：INSERT 语句*

```
insert_statement:
            INSERT INTO table opt_column_commalist values_or_query_spec
      ;

values_or_query_spec:
            VALUES '(' insert_atom_commalist ')'
      |     query_spec
      ;

insert_atom_commalist:
            insert_atom
      |     insert_atom_commalist ',' insert_atom
      ;

insert_atom:
            atom
      |     NULLX
      ;

atom:
            parameter_ref
      |     literal
      |     USER
      ;
```

用于向表中插入新行的INSERT语句有两个变量。在两种情况下都以表名和一个可选的列名作为参数。(我们能重新使用已经用在CREATE VIEW中的**opt_column_commalist**。)接下来是一列值或一条查询规范。这列值是“VALUE”和由逗号分隔的一列**insert_atom**。插入原子可以是NULL、参数、文字字符串或数字或者意味着当前用户ID的“USER”。被定义的查询规范**query_spec**在数据库中选择现有的数据拷贝到当前的表中。

## DELETE 语句

DELETE 语句从表中删除一行或多行。它的规则列在例 5-15 中。

例 5-15: DELETE 语句

```
delete_statement_positioned:
                DELETE FROM table WHERE CURRENT OF cursor
        ;

delete_statement_searched:
                DELETE FROM table opt_where_clause
        ;

opt_where_clause:
                /* 空值 */
        |       where_clause
        ;

where_clause:
                WHERE search_condition
        ;
```

定位情况下删除游标处的行。搜索情况下删除由可选的 WHERE 子句标识的行，或者在没有 WHERE 子句的时候删除表中所有的行。WHERE 子句使用搜索条件（下面定义）来标识要删除的行。

## UPDATE 语句

UPDATE 语句如例 5-16 所示。

例 5-16: UPDATE 语句

```
update_statement_positioned:
                UPDATE table SET assignment_commalist
                WHERE CURRENT OF cursor
        ;

assignment_commalist:
        |       assignment
        |       assignment_commalist ',' assignment
        ;

assignment:
                column COMPARISON scalar_exp
        |       column COMPARISON NULLX
        ;
```

```
update_statement_searched:
                UPDATE table SET assignment_commalist opt_where_clause
        ;
```

UPDATE语句用于重写一行或多行。有两种形式，定位的和搜索的、与DELETE的两种形式相似。在两种情况下，由逗号分隔的赋值列表给相应行的列设置新值，值可以是NULL或标量表达式。

## 标量表达式

标量表达式如例5-17所示。

例5-17：标量表达式

```
scalar_exp:
                scalar_exp '+' scalar_exp
        |       scalar_exp '-' scalar_exp
        |       scalar_exp '*' scalar_exp
        |       scalar_exp '/' scalar_exp
        |       '+' scalar_exp %prec UMINUS
        |       '-' scalar_exp %prec UMINUS
        |       atom
        |       column_ref
        |       function_ref
        |       '(' scalar_exp ')'
        ;

scalar_exp_commalist:
                scalar_exp
        |       scalar_exp_commalist ',' scalar_exp
        ;

function_ref:
                AMMSC '(' '*' ')' /* COUNT(*) */
        |       AMMSC '(' DISTINCT column_ref ')'
        |       AMMSC '(' ALL scalar_exp ')'
        |       AMMSC '(' scalar_exp ')'
        ;

scalar_exp_commalist:
                scalar_exp
        |       scalar_exp_commalist ',' scalar_exp
        ;
```

标量表达式类似于常规程序设计语言中的算术表达式。它们允许具有通用优先级的通用算术操作符。回忆一下使用 **%left** 在语法的开头设置优先级的情形，而且 **%prec** 赋予一元"+"和"-"最高的优先级。SQL 还有几个内置的函数。标记 AMMSC 是 AVG、MIN、MAX、SUM 或 COUNT 中任何一个的简写。这里的语法实际上比 SQL 允许的宽松。"COUNT(*)"计算所选择的集合中的行数，而且是允许参数"*"的惟一的地方。（动作代码必须检查 AMMSC 的标记值，并且如果它不是 COUNT 就会"抱怨"。）DISTINCT 意味着在执行函数之前删除重复值；它只允许列引用。其他函数可以采用标量引用。

为了更好地匹配允许的语法，就要使语法规则变得更加复杂，但是这种方法有两个优点：语法分析程序比较小而且快速；动作程序能发出更详细的消息（例如，"MIN 不允许 * 参数"而不是"语法错误"）。

标量函数的定义是彻底的递归式，所以这些规则可以让你编写极其复杂的表达式，例如：

```
SUM( (p.age*p.age) / COUNT( p.age ) ) - AVG( p.age ) * AVG(p.age )
```

上述表达式计算表 p 中 age 列的算术方差（也许它非常慢）。

我们还定义了 **scalar_exp_commalist** —— 由逗号分隔的一列标量表达式，后面会用到。

## SELECT 语句

SELECT 语句如例 5-18 所示。

例 5-18：SELECT 语句、查询规范和表达式

```
select_statement:
                SELECT opt_all_distinct selection
                INTO target_commalist
                table_exp
        |
                // 手工添加 (taisal)
                SELECT opt_all_distinct selection
                table_exp
        ;
```

```
opt_all_distinct:
                /* 空值 */
        |       ALL
        |       DISTINCT
        ;

selection:
                scalar_exp_commalist
        |       '*'
        ;

query_exp:
                query_term
        |       query_exp UNION query_term
        |       query_exp UNION ALL query_term
        ;

query_term:
                query_spec
        |       '(' query_exp ')'
        ;

query_spec:
                SELECT opt_all_distinct selection table_exp
        ;
```

SELECT 语句从数据库中选择一行（可能得自不同表中大量不同的行）并使用 **table_exp**（在下一节进行定义）将它移入局部变量的集合，**table_exp** 是从数据库中选择表或子表的表值（table-valued）表达式。可选的 ALL 或 DISTINCT 用于保持或放弃重复的行。

查询表达式 **query_exp** 和查询规范 **query_spec**（也在例 5-18 中）类似表值的形式。查询规范几乎和 SELECT 语句具有相同的形式，但是没有 INTO 子句，因为查询规范是较大语句的一部分。查询表达式是几个查询规范的UNION；查询的结果合并在一起。（规范必须都有相同数目和类型的列。）

## 表表达式

表表达式如例 5-19 所示。

**例 5-19：表表达式**

```
table_exp:
                from_clause
                opt_where_clause
                opt_group_by_clause
                opt_having_clause
        ;

from_clause:
                FROM table_ref_commalist
        ;

table_ref_commalist:
                table_ref
        |       table_ref_commalist ',' table_ref
        ;

table_ref:
                table
                table range_variable
        ;

range_variable:
                NAME
        ;

where_clause:
                WHERE search_condition
        ;

opt_group_by_clause:
                /* 空值 */
        |       GROUP BY column_ref_commalist
        ;

column_ref_commalist:
                column_ref
                column_ref_commalist ',' column_ref
        ;

opt_having_clause:
                /* 空值 */
        |       HAVING search_condition
        ;
```

表表达式是赋予SQL动力的东西，因为它们可以让你定义正确地检索想要的数据的任意精心设计的表达式。表表达式从强制性的FROM子句开始，后面跟着可选的WHERE、GROUP BY和HAVING子句。FROM子句命名构建表达式的表的名称。可选的范围变量可以让你用一个表达式在分隔的内容中多次使用相同的表，这个功能有时是很有用的，例如，经理人和职员都在各自的表中。WHERE子句指定一个搜索条件来控制包含在表中的行，GROUP BY子句根据普通的列值归组行，如果选择中包括类似SUM或AVG这样的函数时GROUP BY尤其有用，因为那时它们按组求和或平均。HAVING子句分组地应用搜索条件；例如，在供应者、零件名和价格表中，查询供应者供应的所有零件，这些供应者至少销售三种零件：

```
SELECT supplier
FROM p
GROUP BY supplier
HAVING COUNT(*) >= 3
```

## 搜索条件

例5-20定义了搜索条件的语法。

例5-20：搜索条件

```
search_condition:
        |       search_condition OR search_condition
        |       search_condition AND search_condition
        |       NOT search_condition
        |       '(' search_condition ')'
        |       predicate
        ;

predicate:
                comparison_predicate
        |       between_predicate
        |       like_predicate
        |       test_for_null
        |       in_predicate
        |       all_or_any_predicate
        |       existence_test
        ;
```

```
comparison_predicate:
                scalar_exp COMPARISON scalar_exp
        |       scalar_exp COMPARISON subquery
        ;

between_predicate:
                scalar_exp NOT BETWEEN scalar_exp AND scalar_exp
                scalar_exp BETWEEN scalar_exp AND scalar_exp
        ;

like_predicate:
                scalar_exp NOT LIKE atom opt_escape
        |       scalar_exp LIKE atom opt_escape
        ;

opt_escape:
                /* 空值 */
        |       ESCAPE atom
        ;

test_for_null:
                column_ref IS NOT NULLX
        |       column_ref IS NULLX
        ;

in_predicate:
                scalar_exp NOT IN '(' subquery ')'
                scalar_exp IN '(' subquery ')'
        |       scalar_exp NOT IN '(' atom_commalist ')'
                scalar_exp IN '(' atom_commalist ')'
        ;

atom_commalist:
                atom
        |       atom_commalist ',' atom
        ;

all_or_any_predicate:
                scalar_exp COMPARISON any_all_some subquery
        ;

any_all_some:
                ANY
        |       ALL
```

```
        |       SOME
        ;

existence_test:
                EXISTS subquery
        ;

subquery:
                '(' SELECT opt_all_distinct selection table_exp ')'
        ;
```

搜索条件指定想要使用的组中的行。搜索条件是与AND、OR和NOT结合起来的谓词组合。有7种不同种类的谓词，谓词是人们愿意对数据库执行的操作的聚合。

*COMPARISON*谓词比较两个标量表达式，或者一个标量表达式和一个子查询。回忆一下，COMPARISON标记是任意一种通用的比较操作符，例如“=”和“<>”。子查询（subquery）是一个递归的SELECT表达式，（在语义上而不是语法上）被限制为返回单列。

*BETWEEN*谓词仅仅是一对比较的简写。下面两个谓词是等效的，例如：

```
p.age BETWEEN 21 and 65

p.age >= 21 AND p.age <= 65
```

*LIKE*谓词执行一些字符串模式匹配、比较标量表达式与原子的操作，原子是文字字符串或字符串参数引用。与表达式进行比较的原子被看做一个简单的模式，类似UNIX命令解释程序（shell）文件名模式。可选的ESCAPE子句可以让你在文件名模式中指定类似于“\”的引号字符。

LIKE谓词的左操作符必须是一个列引用，不是一般的表达式。这里使用一个标量表达式来规避yacc局限性。LIKE谓词更常用的语法是：

```
like_predicate:
                column_ref NOT LIKE atom opt_escape
        |       column_ref LIKE atom opt_escape
        ;
```

yacc 在谓词的内容中可以看到类似下面的东西:

```
Foods.flavor NOT ...
```

当看到NOT时，它不能说明它是在NOT BETWEEN中还是在NOT LIKE中，所以它不能说明“Foods.flavor”是LIKE谓词的**column_ref**还是BETWEEN的**scalar_exp**。yacc会用移进/归约冲突对此做出反应，因为它不能说明是否归约将**column_ref**转变为**scalar_exp**的规则。解决这个问题有两种方法。调整语法接受移进/归约冲突的归约端，两种情况都允许**scalar_exp**，因为动作代码能容易地检测NOT LIKE的左操作符以确保它为列引用。(这样还为得到更好的错误消息提供了机会。)另一种可能性是词法修改。可以定义在词法分析程序中匹配两个单词的NOTLIKE标记:

```
NOT[ \t]+LIKE      { return NOTLIKE; }
```

并且在LIKE谓词中使用:

```
like_predicate:
        column_ref NOTLIKE atom opt_escape
    |   column_ref LIKE atom opt_escape
    ;
```

这样就可解决这个问题，因为yacc一旦看到了NOTLIKE标记它就能说明它在分析LIKE谓词。但是词法修改是很险恶的。(如果NOT和LIKE都在分离的行上，这种形式就会失败。如果在它们之间的空格中添加“\n”，那么在这个词法动作中就需要检测是否有任何换行，如果有，更新**lineno**，这会添加更多的混乱。)

对空值的测试只是它听起来像是空的，也就是测试特定列的内容是否为空。使用标记名NULLX在词法分析程序中避免与*stdio* NULL符号发生冲突。

IN谓词检测一个值是否为显式指定的或通过一个子查询指定的集合中的一个。显式形式等效于一组比较:

```
q.Name IN ( 'Tom', 'Dick', 'Harry' )

q.Name = 'Tom' OR q.Name = 'Dick' OR q.Name = 'Harry'
```

ANY或ALL谓词可以让你测试是表达式的任何一个还是所有的值使比较满足了子查询。它们有时是很有用的，但通常容易混淆：很难正确地书写ANY和ALL谓词。下面的例子检测表**p**的**Name**列中的名字与表**q**的**name**列中的名字匹配的所有名字：

```
p.Name =ALL (SELECT q.Name from q)
```

最后，存在测试可以测试是否有满足一些子查询的数据。

使用所有的谓词和子查询，可以创建查询和真正惊人地复杂的表表达式（该表达式执行时花费的时间也同样是惊人的）。实际上，大多数SQL SELECT表达式都很简单，而执行复杂操作的能力也可以满足需要它的人们。

## 零碎的内容

例5-21只定义了一些在嵌入式SQL程序中使用的语句。

例5-21：嵌入式SQL的条件

```
        /* 嵌入的条件 */
sql:           WHENEVER NOT FOUND when_action
        |      WHENEVER SQLERROR when_action
        ;

when_action:          GOTO NAME
        |      CONTINUE
        ;
```

它们意味着，每当选择不检索任何数据（NOT FOUND）或一些其他错误（SQLERROR）的时候，程序将要么跳到主程序中的一个特定的标签，要么忽略这个条件。

## 使用语法检查程序

例5-22展示了*Makefile*。

例 5-22：SQL 语法检查程序的 Makefile

```
LEX = flex -I
YACC = byacc -dv
CFLAGS = -DYYDEBUG=1

all: sql1

sql1: sql1.o scn1.o
	${CC} -o $@ sql1.o scn1.o

sql1.c sql1.h:	sql1.y
	${YACC} sql1.y
	mv y.tab.h sql1.h
	mv y.tab.c sql1.c
	mv y.output sql1.out

scn1.o: sql1.h
```

为了编译语法检查程序，合理地经由 lex 和 yacc 运行词法分析程序和扫描程序并与作为最终结果的 C 程序编译在一起。在这种情况下，使用 *make* 规则重新命名 lex 和 yacc 的输出来匹配输入文件。而且，使用 Berkeley yacc 和 flex（注 3），并定义自己的 **main()** 和 **yyerror()**，所以不需要使用 lex 或 yacc 库。为了测试语法检查程序，可以检测 SQL 文件的完整性，或直接键入：

```
% sql1 sqlmod
SQL parse worked
% sql1
FETCH foo INTO
:a ,
b c, -- two names are legal
d e f -- but three aren't
4: syntax error at f
SQL parse failed
```

注 3： AT&T lex 中存在的缺陷使它不能处理 SQL 词法分析程序，但是 lex 的所有的其他版本都会毫无问题地接受它。yacc 的所有版本都接受这个分析程序。

# 嵌入式SQL

本章最后讨论将SQL语法检查程序转换为简单的嵌入式SQL预处理程序。假定有一个能够解释作为文本串传递的SQL语句的SQL实现。嵌入式SQL预处理程序只需要将SQL语句转换为C过程调用，这个C过程调用将SQL语句传递给解释例程。

嵌入式SQL比它看上去要复杂一点。词法分析程序必须在两个不同的状态下运行：只传递文本通过的正常状态和缓冲传递给解释程序的SQL语句的SQL模式。还需要处理参数引用，因为在这个已编译的程序中，解释程序没有将字符串“:foo”和变量foo关联在一起的方法。在词法分析程序中提取这个参数，用“#N”代替提到的变量Nth，然后通过参数列表中的引用将所有提到的变量传递给解释程序。例如，嵌入式SQL：

```
EXEC SQL FETCH flav INTO :name, :type ;
```

应该转换成C语言：

```
exec_sql(" FETCH flav INTO #1, #2 ", &name, &type);
```

这里重点强调对词法分析程序和语法分析程序的更改。完整的代码在附录十中。

## 对词法分析程序的更改

词法分析程序实际上有庞大的更改集合。例5-23展示了修改后的定义。

例5-23：嵌入式词法分析程序中的定义

```
%{
#include "sql2.h"
#include <string.h>

int lineno = 1;
void yyerror(char *s);

        /* 保存SQL标记文本的宏 */
#define SV save_str(yytext)
```

```
        /* 保存文本并返回标记的宏 */
#define TOK(name) { SV;return name; }
%}

%s SQL
```

我们已经定义了两个C宏，定义了调用**save_str()**来保存当前标记的文本的**SV**、以及保存标记文本并给语法分析程序返回一个标记的**TOK()**。我们还添加了新的称为**SQL**的起始状态，将标准的**INITIAL**状态作为正常的非SQL的状态。

例5-24展示了修订的词法分析程序规则。

例5-24：嵌入式词法分析程序规则

```
EXEC[ \t]+SQL       { BEGIN SQL; start_save(); }

        /* 文字关键字标记 */

<SQL>ALL            TOK(ALL)
<SQL>AND            TOK(AND)
<SQL>AVG            TOK(AMMSC)

          ... 所有其他的保留字和标记 ...

        /* 名字 */
<SQL>[A-Za-z][A-Za-z0-9_]* TOK(NAME)

        /* 参数 */
<SQL>":"[A-Za-z][A-Za-z0-9_]*{
                    save_param(yytext+1);
                    return PARAMETER;
            }

        /* 数字 */

<SQL>[0-9]+ |
<SQL>[0-9]+"."[0-9]* |
<SQL>"."[0-9]*          TOK(INTNUM)

<SQL>[0-9]+[eE][+-]?[0-9]+    |
<SQL>[0-9]+"."[0-9]*[eE][+-]?[0-9]+ |
<SQL>"."[0-9]*[eE][+-]?[0-9]+TOK(APPROXNUM)
```

```
        /* 字符串 */

    <SQL>'[^'\n]*' {
                int c = input();

                unput(c); /* 仅查看一下 */
                if(c != '\'') {
                      SV;return STRING;
                } else
                      yymore();
        }

    <SQL>'[^'\n]*$    {       yyerror("Unterminated string"); }
    <SQL>\n           { save_str(" ");lineno++; }
    \n          { lineno++; ECHO; }

    <SQL>[ \t\r]+      save_str(" "); /* 空白 */

    <SQL>"--".* ;      /* 注释 */

                ECHO; /* 随机的非SQL文本 */
    %%
```

第一个新规则匹配“EXEC SQL”关键字并将扫描程序插入SQL状态。它还调用**start_save()**初始化保存SQL命令的缓冲区。然后将“<SQL>”加在所有现有的标记规则的前面，这样它们就只在SQL模式中匹配，并改变动作为使用**SV**或**TOK()**保存每个标记。因为我们需要区别对待参数和其他标记，所以为匹配冒号后跟有名字的参数添加一条新规则，并调用**save_param()**保存参数引用。匹配换行和空格的每个SQL规则都保存单个空格；因为所有的空格都是等效的，保存单个空格可以使保存的字符串较短。最后，添加两条没有<SQL>前缀的规则，它们在不处于SQL模式时匹配和回送所有的字符。在用户子程序部分，添加将词法分析程序从SQL模式切换为INITIAL模式的小例程**un_sql()**；这个例程必须在词法分析程序中，那是定义BEGIN宏的惟一的地方。

```
/* 离开SQL词法分析模式 */
un_sql()
{
      BEGIN INITIAL;
} /* un_sql */
```

## 对语法分析程序的更改

对语法分析程序的更改比对词法分析程序的更改要少得多。添加PARAMETER的 %**token** 定义。向起始规则添加动作:

```
sql_list:
                sql ';'   { end_sql(); }
        |       sql_list sql ';' { end_sql(); }
        ;
```

这些规则在每次分析完整的SQL语句时调用例程**end_sql()**来切换SQL模式外部的词法分析程序。

因为目前参数有一个特殊的标记，更改 **parameter** 规则来引用这个新标记:

```
parameter:
                PARAMETER /* :语法分析程序中的名字处理 */
        ;
```

因为嵌入式SQL不使用模块语言，所以放弃模块语言的规则，只保留游标定义的规则，并使游标定义成为最高层的 SQL 语句:

```
sql:
                cursor_def
        ;
```

完成了——语法分析程序的其他方面没有更改。

## 辅助例程

嵌入式SQL 文本支持例程的重要部分如例 5-25 所示。

例 5-25: 嵌入式SQL 文本支持例程的最重要部分

```
char save_buf[2000];      /* 用于SQL命令的缓冲区 */
char *savebp;             /* 当前缓冲区指针 */

#define NPARAM 20         /* 每个函数最大的参数 */
char *varnames[NPARAM];  /* 参数名字 */
```

```
/* 在EXEC SQL之后开始一个嵌入的命令 */
start_save(void);

/* 保存SQL标记 */
save_str(char *s);

/* 保存引用的参数 */
save_param(char *n);

/* SQL命令结束，现在输出 */
end_sql(void);
```

我们编写了一些字符串处理例程，在它们被分析时缓冲并写出SQL命令。数据结构和入口点在例5-25中，而且整个文本在附录十中。在有很大的固定尺寸的字符缓冲区**save_buf[]**中保存这些命令并使用字符指针**savebp**来跟踪缓冲区中的当前位置。作为参数使用的每个变量名都保存在**varnames[]**中。如果一个变量在同一个命令中使用两次，我们只保存一次。

当词法分析程序看到"EXEC SQL"时，程序**start_save()**初始化缓冲指针。每个标记都用**save_str()**保存，**save_str()**将它的参数附加给**save_buf**。参数引用由查找它的参数的**save_param()**处理，如果**varnames[]**中的变量名没有出现就输入它，然后保存"*#N*"形式的引用。

当语法分析程序看到整个SQL命令时，调用**end_sql()**，写出对运行时解释程序**exec_sql()**的调用。它将保存的缓冲区作为引用字符串来传递，必要时将它拆分成行，而且还传递参数表中每个变量的地址。最后，它调用词法分析程序例程**un_sql()**让词法分析程序离开SQL状态。所有的输出都进入**yyout**（默认的lex输出流），正像词法分析程序中的**ECHO**语句通过非SQL代码传递一样。

## 使用预处理程序

修改Makefile，以在辅助程序中与词法分析程序和语法分析程序相连接。因为没有更改**main**程序，只更改了它的消息（一个纯粹的装饰性更改），所以用运行语法检查程序的同样的方式运行预处理程序。

例 5-26 展示了在例 5-2 中的嵌入式 SQL 上运行预处理程序的结果。

**例 5-26：来自嵌入式 SQL 预处理程序的输出**

```
char flavor[6], name[8], type[5];
int SQLCODE;        /* 全局状态变量 */

exec_sql(" DECLARE flav CURSOR FOR SELECT Foods.name, Foods.\
type FROM Foods WHERE Foods.flavor = #1 ",
     &flavor);

main()
{
     scanf("%s", flavor);
     exec_sql(" OPEN flav ");
     for(;;) {
          exec_sql(" FETCH flav INTO #1, #2 ",
     &name,
     &type);
          if(SQLCODE != 0)
               break;
          printf("%8.8s %5.5s\n", name, type);
     }
     exec_sql(" CLOSE flav ");
}
```

# 练习

1. 在几个方面，SQL 语法分析程序接受比 SQL 本身允许的更宽松的语法。例如语法分析程序接受无效的标量表达式 “MIN(*)”，并且把任何表达式都作为 LIKE 谓词的左操作数来接受，尽管操作数必须是列引用。调整语法检查程序以诊断这些错误的输入。可以更改语法或添加动作代码来检查表达式。两种方法都尝试一下，看看哪种更容易，以及哪种方法可以给出更好的诊断。

2. 将语法分析程序转换成 SQL 交叉引用程序，读取一组 SQL 语句并产生一个报告，这个报告展示定义每个名字的地方和引用这些名字的地方。

3. （术语工程）修改嵌入式 SQL 翻译程序以接到你的系统上真正的数据库。

# 第六章

# lex 规范参考

本章内容：

- lex 规范的结构
- 以字母为顺序介绍 lex 的特征

本章讨论 lex 规范的格式并描述可用的特征和选项。本章总结了前面几章论证的能力并概述没有讨论的特征。

在讨论了 lex 程序的结构后，本章的其他几节根据特征的字母顺序进行排列。

## lex 规范的结构

lex 程序由三部分组成：定义段，规则段和用户子例程段。

```
... 定义段 ...
%%
... 规则段 ...
%%
... 用户子例程段 ...
```

这些部分由以两个百分号组成的行分隔开。尽管某一部分可以为空，但前两部分是必需的。第三部分和前面的 %% 行可以忽略。（这个结构和 yacc 使用的结构相同，其实它是从 yacc 复制过来的。）

## 定义段

定义段包括文字块(literal block)、定义(definition)、内部表声明(internal table declaration)、起始条件(start condition)和转换(translation)。(在这一参考中每个内容都有一节。)以空白开头的行被逐字拷贝到C文件中，通常，这用于包含包围在"/*"和"*/"中的注释，一般前面有空白。

## 规则段

规则段包含模式行和C代码。以空白开始的行或包围在"%{"和"%}"中的内容是C代码。以任何其他东西开始的行是模式行。

C代码被逐字拷贝到生成的C文件中。规则段开头的行靠近生成的**yylex()**函数的开头，并且应该是与模式相关的代码所使用的变量声明或扫描程序的初始化代码。任何其他地方的C代码行被拷贝到生成的C文件中不确定的地方，应该仅包含注释。(这就是如何将注释插入动作外部的规则段。)

模式行包含带有空白的模式以及输入与这个模式匹配时执行的C代码。如果C代码是多条语句或者跨越多行，它就必须包围在括号{}中(注1)。

当lex扫描程序运行时，它把输入与规则段的模式进行匹配。每次发现一个匹配(被匹配的输入称为标记(token))时就执行与那种模式相关的C代码。如果模式后面跟着一条竖线而不是C代码，那么这个模式将使用与文件中的下一个模式相同的C代码。当输入字符不匹配模式时，词法分析程序的动作就好像它匹配上了代码为"ECHO;"的模式，ECHO将标记的拷贝写到输出。

## 用户子例程段

用户子例程段的内容被lex逐字拷贝到C文件。这一部分通常包括从规则中调用

注1：缺少括号时，lex的一些版本采用行的整个其余部分，其他一些只是采用分号。为了最大化透明度和可移植性，给最无足轻重的C代码应用括号。

的例程。如果重新定义 **input()**、**unput()**、**output()**或 **yywrap()**，新的版本或支持子程序可以放在这里。

在庞大的程序中，当更改 lex 文件时，有时将支持代码插入到单独的源文件中更方便，这可以最小化重编译的量。

# BEGIN

**BEGIN** 宏在起始状态之间进行切换。通常可以在模式的动作代码中调用它，例如:

```
BEGIN statename;
```

扫描程序从状态 **0**（零）开始，也称为 **INITIAL**。所有其他状态必须用 **%s** 或 **%x** 行在定义段中指定。（参见本章后面的“起始状态”一节。）

要注意即使 **BEGIN** 是一个宏，该宏本身也不采用任何参数，而且 *statename* 不需要被包围在括号中（虽然包围在括号中是很好的风格）。

# 程序错误

像任何其他的计算机程序一样，lex的版本也有它们的程序错误（bug）。还有几个值得提及的普通的匹配特性的模式。

## 有歧义的向前查看

使用尾部上下文操作符的模式（标记的结尾可以匹配和尾部上下文的开始部分相同的文本）不会安全地工作。例如:

```
(a|ab)/ba
zx*/xy*
```

这是常用的模式匹配算法的问题，所以未必能很快解决。当这个问题使它不可能产生正确的扫描程序时，*flex* 会发出一个警告。

## AT&T lex

毫无疑问，AT&T lex 的程序错误成灾。一部分原因是由于它是第一种实现，另一部分原因是由于它是由尚未大学毕业的暑假实习生编写的。计算字符范围的重复数时有一个程序错误，所以类似下面的模式不会工作：

```
[0-9]+-[0-9]{4} [0-9
```

在规则段还有注释方面的困难。例如，第一章的例子会从 lex 中得到虚假的错误消息，除非删除两个注释行：

```
%%
\n      { state = LOOKUP; }       /* 行结束，返回默认的状态 */

        /* 每当一行以保留的词类名称开始时 */
        /* 开始定义该词类的单词 */

^verb { state = VERB; }
^adj  { state = ADJ; }
^adv  { state = ADV; }
...
```

由AT&T lex产生的复杂扫描程序还有令人懊憾的方面，即缺乏精确定点的方法。

## flex

*flex* 比 AT&T lex 更安全。自版本 2.3.7 起，我们知道的惟一的程序错误是与“I”动作有关的含糊的程序错误。这个脚本寻找使单词“lex”成为斜体的 *troff*（编辑和格式化）宏，并取消它们的斜体属性。

```
%%

^\.I\ +lex$
^\.I\ *\"lex\"$            { fputs("lex", yyout); }
```

```
%%
```

输入

```
.I lex
```

产生 lexx 而不是正确的 lex。如果写出两次这个动作，程序错误就会消失（注2）。

# 字符变换

lex的大部分版本都有由 **%T** 引入的字符变换。可惜的是，不同版本中它们所做的事情存在很大的不同。

在AT&T lex和MKS lex中，词法分析程序通常使用C编译程序使用的本机字符代码，例如，字符“A”的代码是C值“A”。偶尔使用一些其他的字符代码是很方便的，因为输入流使用不同的代码，例如baudot或EBCDIC，或者因为lex在非文本输入流中寻找模式。lex字符变换可以让你在由 **input()** 读取的字符和lex模式使用的字符之间定义明确的映像。这些变换的前后都是由 **%T** 组成的行。每个变换行包含一个数字、一些空格，然后是一个或多个字符。例如：

```
%T
1       aA
2       bB
3       cC
%T
```

在这个例子中，值为1的输入字节匹配模式中有一个“A”或“a”的任何地方，值为2的输入字节匹配有“B”或“b”的地方，而且值为3的输入字节匹配有“C”和“c”的地方。

如果不是直接来自文件，那么也许需要修改AT&T lex中的 **input()** 和 **unput()** 宏或MKS lex中的 **yygetc()** 以产生适当的值。

注2：　我们已经将这件事告诉了 *flex* 的维护人员，所以你读取它时，它已经被修正过了。

如果使用变换，那么lex模式中使用的文字字符必须出现在变换行的右侧。

*flex*有一个不同的、几乎无用的变换版本，本书不对它进行介绍。预计在*flex*的未来版本中它会被删除。使用 -*i*标志可以很容易地使用*flex*的最简单的变换——将大写和小写字母折叠在一起。

# 上下文相关

lex提供了几种使模式对左和右上下文敏感的方式，也就是对标记的前或后敏感。

## 左上下文

有三种处理左上下文的方式：行模式字符的特殊开始、起始状态和明确的代码。

模式开始处的字符"^"告诉lex只在行的开始处匹配模式。"^"不匹配任何字符，它只指定上下文。

起始状态用于要求一个标记前面是另一个标记：

```
%s MYSTATE
%%
first  { BEGIN MYSTATE; }
. . .
<MYSTATE>second    { BEGIN 0; }
```

在这个词法分析程序中，标记**second**只在标记**first**后被识别。在**first**和**second**之间也可以插入别的标记。

在一些情况下，可以通过设置一个标志（用于从一个标记的例程向另一个标记的例程传递上下文信息）来伪造左上下文相关：

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

```
a       { flag = 1; }
b       { flag = 2; }
zzz     {
              switch(flag) {
        case 1:     a_zzz_token(); break;
        case 2:     b_zzz_token(); break;
        default: plain_zzz_token(); break;
              }
              flag = 0;
        }
```

## 右上下文

有三种方法使标记识别取决于标记右侧的文本：行模式字符的特殊结尾、斜杠操作符和 **yyless()**。

模式尾部的“$”字符使标记只匹配行的尾端，即\n字符的前面。像“^”一样，“$”不匹配任何字符，它只指定上下文。它完全等效于“/\n”，也因此不能和尾部上下文一起使用。

模式中的“/”字符明确地包括尾部上下文。例如，只有当它后面立即跟有“de”时，模式“abc/de”匹配标记“abc”。“/”本身不匹配任何字符。当决定几个模式中的哪一种匹配最长时，lex 计算尾部上下文字符，但是这些字符不出现在 **yytext[]**中，而且它们不计入 **yyleng** 中。

**yyless()**函数告诉 lex“推回”刚被读取的标记部分。**yyless()**的参数是要保持的标记字符数。例如：

```
abcde { yyless(3); }
```

和“abc/de”几乎有相同的效果，因为对**yyless()**的调用保持标记的三个字符并推回其他两个。在这种情况下，惟一的区别是 **yytext[]**中的标记包含所有的 5 个字符，而且 **yyleng** 是 5 而不是 3。

# 定义（替换）

定义（或替换）允许赋予全部或部分正则表达式一个名字，并在规则段根据名字引用它。它有助于分解复杂的表达式和说明表达式的功能。定义采用下面的形式：

```
NAME expression
```

名字可以包含字母、数字和下划线，而且不能用数字开头。一些实现还允许使用连字符。

在规则段，模式可以包括用括号中的名字进行替换的引用，例如“{NAME}”。对应该名字的表达式被逐字代入模式。例如：

```
DIG [0-9]
. . .
%%
{DIG}+               process_integer();
{DIG}+\.{DIG}*       |
\.{DIG}+     process_real();
```

有一种在 lex 的版本之间改变替换处理的简单方法。在多数版本中，当对应名字的模式被代入时，它被看做被包围在括号中。虽然在另一些版本中它不是这样的，这使得在某些情况下会有些不同，例如：

```
PAT    abc
%%
{PAT}+
```

如果模式作为“(abc)+”处理，那么它匹配任意数目的“abc”的拷贝，而如果它是“abc+”，那么它匹配后面跟有任意数目的 c 的“abc”。为了最大化可移植性，将模式括入圆括号内的定义中，如下所示：

```
PAT    (abc)
```

# ECHO

在与模式相关的C代码中，宏**ECHO**将标记写到当前的输出文件*yyout*。它等效于：

```
fprintf(yyout, "%s", yytext);
```

在lex中不匹配任何模式的输入文本的默认动作是将文本写到输出，等效于**ECHO**。在flex中，命令行的标志 *-s* 使用退出作为默认动作，在扫描程序被要求包括处理所有可能的输入的模式的通用情况下是很有用的。

在某些lex版本中，可以重新定义**ECHO**对字符来做其他事情。如果重新定义**ECHO**，你还会想要重新定义**output()**，它通常向*yyout*发送单个字符。

# 包含操作（文件的逻辑嵌套）

许多输入语言都有一些包含语句，逻辑上是插入另一个文件来代替包含语句。在程序的开始，可以赋给**yyin**任何打开的*stdio*文件，以使扫描程序从那个文件中读取。可惜的是，lex中没有可移植的方法处理嵌套的输入文件，但下面是对主要实现的一些提示。

## 与yywrap()链接在一起的文件

当词法分析程序到达输入文件的结尾时，它调用**yywrap()**。可以编写自己的**yywrap()**，通过改变或重新打开**yyin**切换到新的输入文件，并继续扫描。参见本章“yywrap()”一节得到更多的详细资料。

## 文件嵌套

不同版本的lex处理嵌套文件的方法不同。我们简要描述由主要实现提供的机制。

### AT&T lex

在AT&T lex中，可以重新定义处理多个输入文件的标准的**input()**和**unput()**宏。你需要保持一个堆栈或者包含FILE指针、回推缓冲区（pushback buffer）和日录以及文件中的行数的结构链接表，而且让**input()**和**unput()**使用堆栈上最顶端的结构。在文件的结尾，结束文件，删除堆栈上的顶层结构，并继续堆栈上的下一个文件。

## flex

在flex中，不能重新定义**input()**或**unput()**，（词法分析程序不使用它们，而是从底层的数据结构中取得字符，以便提高速度。）但是能重新定义**YY_INPUT**，它是flex调用的从输入文件中读取文本的宏（参见本章“从字符串中输入”一节）。更有用的是flex缓冲区，被定义为*YY_BUFFER_STATE*类型。例程**yy_create_buffer(***FILE*,size***z0)**生成指定大小的新的flex缓冲区，通常是**YY_BUF_SIZE**（注3），用于读取stdio FILE。对**yy_switch_to_buffer(***flexbuf***)**的调用告诉扫描程序读取相应的文件，而**yy_delete_buffer(***flexbuf***)**删除flex缓冲区。当前的缓冲区是**YY_CURRENT_BUFFER**。另外一个有用的东西是特殊的标记模式“<<EOF>>”，它在调用**yywrap()**之后匹配文件的结尾。

### MKS lex

MKS lex定义例程**yySaveScan()**和**yyRestoreScan()**来保存和恢复扫描程序的当前状态。它们使用包含状态的**YY_SAVED**类型的对象。要保存状态，调用**yySaveScan(***file***)**。它返回一个**YY_SAVED**对象，并指定读取stdio流*file*。为了恢复前面保存的状态，调用**yyRestoreScan(***saved***)**，它恢复前面保存的状态。

### Abraxas pclex

虽然pclex是基于flex的，但是pclex不包括flex中可用的缓冲区切换例程。保存和恢复缓冲区状态是如此难，以至于不切实际。

---

注3：　参见本章的“yytext”得到改变flex缓冲区大小的提示。

一种途径是包括多个扫描程序并且在需要处理一个包含文件时切换扫描程序。要得到更多的信息，参见本章的“一个程序中的多个词法分析程序”一节。

### POSIX lex

POSIX.2 标准对文件包含采用简单的办法：它根本不支持文件包含。除了**yywrap()**以外，在POSIX lex中没有处理多个输入文件的标准方式。大多数实现都提供一些扩展的支持，但必须参阅特定版本的文件。

## 从字符串中输入

一般lex从文件读取，但有时想让它读取一些其他的源，例如内存中的字符串。lex的所有的版本都可以实现这一功能，但细节却有很大的不同。

### AT&T lex

AT&T lex 用**input()**宏读取所有的输入。为了改变输入源，重新定义**input()**和**unput()**宏。例如：

```
%{
extern char *mystring;

#undef input
#undef unput
#define input()  (*mystring++)
#define unput(c) (*--mystring = c)
%}
```

在输入数据的结尾，**input()**应该返回0。

### flex

虽然flex提供了一个**input()**函数，但是它采用最佳的内联代码获得字符。可以重新定义**YY_INPUT**，它是用于读取数据块的宏。按如下方式进行调用：

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

**buffer** 是字符缓冲区，**result** 是存储实际读取的字符数的变量，**max_size** 是缓冲区的大小。为了读取字符串，让你的 **YY_INPUT** 版本从字符串缓冲区中拷贝数据（见例6-1）。

例6-1：从字符串中得到 flex 输入

```
%{
#undef YY_INPUT
#define YY_INPUT(b, r, ms) (r = my_yyinput(b, ms))
%}
. . .
extern char myinput[];
extern char *myinputptr;/* myinput中的当前位置 */
extern int *myinputlim;/* 数据结尾 */

int
my_yyinput(char *buf, int max_size)
{
	int n = min(max_size, myinputlim - myinputptr);

	if(n > 0) {
		memcpy(buf, myinputptr, n);
		myinputptr += n;
	}
	return n;
}
```

## Abraxas pclex

因为 pclex 来源于 flex，所以它使用相同的输入机制。按上面所描述的重新定义 **YY_INPUT()** 来改变输入源。

## MKS lex

MKS lex 使用宏 **yygetc()** 读取所有的输入字符。为了改变输入源，重新定义 **yygetc()**。MKS 词法分析程序自动处理“推回”问题，所以不用担心它。在输入

结束时，yygetc()返回EOF。下面是一个可能的定义，稍微复杂一点的是当字符串的字符为空时返回EOF:

```
%{
extern char *mystring;
#undef yygetc
#define yygetc() (*mystring? *mystring++ : EOF)
%}
```

## POSIX lex

POSIX标准不定义任何改变输入源的方式，所以从不同于**yyin**的其他地方读取输入的程序在不同实现上是不可移植的。

# input()

**input()**程序向词法分析程序提供字符。在lex的一些版本中，例如，AT&T lex，它是一个宏；而在另一些版本中，例如flex，将它定义为函数。

当词法分析程序匹配字符时，它概念上调用**input()**提取每个字符。由于性能的原因，一些实现回避用**input()**，但效果是相同的。

最可能调用**input()**的地方是在对特殊标记后的文本进行特殊处理的一个动作例程中。例如，下面是处理C注释的一种方式:

```
"/*" {      int c1 = 0, c2 = input();
            for(;;) {
               if(c2 == EOF)
                     break;
               if(c1 == '*' && c2 == '/')
                     break;
               c1 = c2;
               c2 = input();
            }
}
```

调用**input()**处理字符，直到文件尾或字符“*/”出现。这种方法是在缺乏排它性起始状态（参见本章“起始状态”一节）时处理C样式注释的最容易的方式，并且对于处理很长的引用字符串和其他对于lex来说太长以至于不能缓冲它本身的标记来说总是最好的方式。

在lex的一些版本中，可能会重新定义**input()**以从不同于stdio文件的某处获取输入。有些版本的lex不让重新定义**input()**，但具有改变输入源的其他方式。参见本章“从字符串中输入”一节得到更多的详细资料。记住重新定义的**input()**必须能处理由**unput()**推回的字符。

# 内部表（%N声明）

虽然AT&T lex和MKS lex允许程序员显式地增加表的大小，但是它们使用对于庞大的扫描程序来说可能并不足够大的固定大小的内部表。在定义段可以用“%a”、“%e”、“%k”、“%n”、“%o”和“%p”行增加表的大小，例如：

```
%p 6000
%e 3000
```

为了查明当前统计量是多少，用*-v*标志运行lex。例如，例4-7中的MGL词法分析程序产生下面的报告：

```
133/2000 nodes(%e), 551/5000 positions(%p), 86/2500 (%n),
6182 transitions, 27/1000 packed char classes(%k),
234/5000 packed transitions(%a), 241/5000 output slots(%o)
```

显然，它通常处理比上面的形式大得多的语法来填充这个表的默认大小。

构造正则表达式有可能导致非常大的状态机，它比正常的表要大得多。一般，通过用简单的形式编写它们，将它们拆分成多个表达式或者编写C代码来处理更多的工作可以很好地简化这些表达式。

除了非常大的工程以外，增加表的大小没有必要。除非lex抱怨这些表之一已经

溢出，否则根本不需要担心它们。为了算出表的最佳大小，将溢出的表的大小增加到足够大，用 -*v* 标志运行 lex，并调整接近于词法分析程序实际需要的值。

lex 的一些老版本也接受“%r”和“%c”，“%r”使 lex 产生使用 Ratfor 语言的词法分析程序，“%c”使它产生使用 C 语言的词法分析程序。

# lex 库

大多数 lex 实现都需要有用的例程库。通过在 UNIX 系统的 *cc* 命令行的结尾（或其他系统的等价物上）给出 -*ll* 标志来链入库。库的内容根据实现的不同有所改变，但它总是包含 **main()**。

## main()

lex 的所有版本都有最小的 **main** 程序，它对于简短程序和测试都是很有用的。它非常简单，我们将它复制在下面：

```
main(int ac, char **av)
{
	return yylex();
}
```

和其有任何库例程一样，你可以提供自己的 **main()**。

## 其他的库例程

可以从 lex 扫描程序中调用的许多例程（例如 **yymore()**、**yyless()** 和 **yywrap()**）可能就在库中，与支持其他 lex 特征的例程在一起，例如 **REJECT**。

任何 lex 程序都能重新定义 **yywrap()** 以改变文件结尾所发生的事情。许多实现还让重新定义 **input()**、**unput()** 和 **output()**。参见其他章节以了解其他例程的详细资料。

# 行号和 yylineno

如果跟踪输入文件中的行号，可以在错误消息中报告它。一些 lex 版本定义 **yylineno** 来包含行号并自动更新它，但大多数版本不这样做。

跟踪行号很容易。初始化行数变量为1，并且在匹配换行字符的任意lex规则中增加该值，如下所示：

```
%{
        int     lineno = 1;
%}
%%
...
\n      { lineno++; }
```

处理嵌套的包含文件的词法分析程序必须保存并恢复与每个文件相关的行号。

# 文字块

定义部分的文字块是由行“%{”和“%}”括住的C代码。

```
%{
... C代码及声明 ...
%}
```

文字块的内容被逐字拷贝到生成的C源文件的开始处，位于**yylex()**的开头前。文字块通常包含规则段的代码使用的变量和函数的声明，以及头文件 **#include** 行。

# 一个程序中的多个词法分析程序

你也许想使同一程序中的两个部分或全部不同的标记语法都拥有词法分析程序。例如，交互调试解释程序使程序设计语言拥有一个词法分析程序并且为调试程序命令使用另一个词法分析程序。

在一个程序中处理两个词法分析程序有两种基本的途径：将它们合并为一个词法分析程序，或者在程序中放入两个完整的词法分析程序。

## 组合的词法分析程序

通过使用起始状态可以将两个词法分析程序组合成一个词法分析程序。每个词法分析程序的所有模式的前面都有一个惟一的起始状态的集合。当起动词法分析程序时，需要少量的代码将词法分析程序转成使用中特定的词法分析程序的适当初始状态，例如，下面的代码（将被复制到**yylex()**的前面）：

```
%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;
        }
%}
```

这种情况下，在调用词法分析程序之前将**first_lex**设置为词法分析程序的初始状态。通常将组合的词法分析程序和组合的yacc语法分析程序一起使用，所以通常还有一些代码让初始标记告诉语法分析程序应使用哪个语法。参见第七章的“变体和多重语法”一节。

这种方式的优点是目标代码比较小，因为只有词法分析程序代码的一个拷贝，并且不同的规则集合能共享规则。缺点是在各个地方必须小心使用正确的起始状态，不能同时有两个词法分析程序处于活动状态（例如，不能递归调用**yylex()**），并且很难为不同的词法分析程序使用不同的输入源。

## 多重词法分析程序

另一条途径是在程序中包括两个完整的词法分析程序。lex这样做并不容易，因为它产生的每个词法分析程序都有相同的入口点——**yylex()**。而且，大多数lex版本将扫描表和扫描程序缓冲区插入具有类似**yycrank**和**yysvec**这样的名字的全局变量中。如果你只是翻译两个扫描程序并且编译并链接两个结果文件的所有内容（至少重新命名其中的一个文件为不同于*lex.yy.c*的其他的名字），你仍然会得到一个包含多重定义符号的很长的列表。技巧是改变lex使用的函数和变量的名字。

### 使用p标志

lex的一些版本，特别是MKS lex，提供一个命令行开关 -*p* 来改变由lex产生的扫描程序中的名字上使用的前缀。例如，命令

```
lex -p pdq myscan.y
```

产生具有入口点**pdqlex()**的扫描程序，**pdqlex()**用于读取文件**pdqin**等等。受影响的名字有**yylex()**、**yyin**、**yyout**、**yytext**、**yylineno**、**yyleng**、**yymore()**、**yyless()**、**yywrap()**，以及所有的特定实现的变量。词法分析程序中使用的其他变量被重新命名而且也被设成静态的。还有一个 **-o** 标志用来指定产生的词法分析程序的名字，例如：

```
lex -p pdq -o pdqtab.c mygram.y
```

产生*pdqtab.c*。

### 伪造它

lex没有自动改变生成的C例程中的名字的方法，所以必须伪造它。在UNIX系统上，伪造它的最容易的方法是用字符流编辑程序*sed*。假设正在使用AT&T lex，创建包含这些*sed*命令的*yy-lsed*文件。（这里我们使用新的前缀“pdq”。）

```
s/yyback/pdqback/g
s/yybgin/pdqbgin/g
```

```
s/yycrank/pdqcrank/g
s/yyerror/pdqerror/g
s/yyestate/pdqestate/g
s/yyextra/pdqextra/g
s/yyfnd/pdqfnd/g
s/yyin/pdqin/g
s/yyinput/pdqinput/g
s/yyleng/pdqleng/g
s/yylex/pdqlex/g
s/yylineno/pdqlineno/g
s/yylook/pdqlook/g
s/yylsp/pdqlsp/g
s/yylstate/pdqlstate/g
s/yylval/pdqlval/g
s/yymatch/pdqmatch/g
s/yymorfg/pdqmorfg/g
s/yyolsp/pdqolsp/g
s/yyout/pdqout/g
s/yyoutput/pdqoutput/g
s/yyprevious/pdqprevious/g
s/yysbuf/pdqsbuf/g
s/yysptr/pdqsptr/g
s/yysvec/pdqsvec/g
s/yytchar/pdqtchar/g
s/yytext/pdqtext/g
s/yytop/pdqtop/g
s/yyunput/pdqunput/g
s/yyvstop/pdqvstop/g
s/yywrap/pdqwrap/g
```

然后，运行lex之后，这个命令编辑生成的扫描程序：

```
sed -f yy-lsed lex.yy.c > lex.pdq.c
```

你可能想在*Makefile*中插入这些规则：

```
lex.pdq.c: myscan.l
	lex -t myscan.l | sed -f yy-lsed > $@
```

如果使用MS-DOS并且不可以使用*sed*，在最坏的情况下，可以手动通查生成的C文件，改变这些名字。

在某些情况下，另一种也许比较容易的方法是在语法的开头使用C预处理程序**#define** s重新命名这些变量：

```
%{
#define yyback pdqback
#define yybgin pdqbgin
#define yycrank pdqcrank
#define yyerror pdqerror
#define yyestate pdqestate
#define yyextra pdqextra
#define yyfnd pdqfnd
#define yyin pdqin
#define yyinput pdqinput
#define yyleng pdqleng
#define yylex pdqlex
#define yylineno pdqlineno
#define yylook pdqlook
#define yylsp pdqlsp
#define yylstate pdqlstate
#define yylval pdqlval
#define yymatch pdqmatch
#define yymorfg pdqmorfg
#define yyolsp pdqolsp
#define yyout pdqout
#define yyoutput pdqoutput
#define yyprevious pdqprevious
#define yysbuf pdqsbuf
#define yysptr pdqsptr
#define yysvec pdqsvec
#define yytchar pdqtchar
#define yytext pdqtext
#define yytop pdqtop
#define yyunput pdqunput
#define yyvstop pdqvstop
#define yywrap pdqwrap
%}
```

这样可以避免使用*sed*。实际上，可能想重新命名lex和yacc，所以将有关它们两个的所有定义插入到文件中，假定为*pdqdefs.h*。无论在何地使用这些名字，首先要包括*pdqdefs.h*，例如，在lex源文件中：

```
%{
#include "pdqdefs.h"
#include "pdq.tab.h"
%}
```

在这种情况下，*pdq.tab.h*是包括标记名定义的yacc产生的头文件。因为它通常定义 **yylval**，所以它需要遵从*pdqdefs.h*。

对于 flex 词法分析程序，需要重新命名的变量是：

```
yy_create_buffer
yy_delete_buffer
yy_init_buffer
yy_load_buffer_state
yy_switch_to_buffer
yyin
yyleng
yylex
yyout
yyrestart
yytext
```

可以使用上面两种技术中的任何一种来对它们重新命名。

# output()

lex的一些版本定义函数或宏**output**(*c*)，**output**(*c*)将参数写到输出文件*yyout*。这通常等效于：

```
putc(c, yyout)
```

如果它存在，可以在动作中使用它，而且扫描程序也可以用它来实现默认的动作，即将不匹配的字符发送给输出。

如果 **output()**是一个宏，你也许想定义它对不匹配的输入字符做一些不同的事情。

很好设计的词法分析程序通常匹配所有可能的输入，在这种情况下，**output()**永远不会自动从词法分析程序内部被调用。

如果重新定义 **output()**，还要重新定义将当前 **yytext** 的内容拷贝到输出的宏 **ECHO**。

# lex 词法分析程序的可移植性

lex 词法分析程序在不同的 C 实现中具有相当大的可移植性。移植词法分析程序有两个级别：原始的 lex 规范或由 lex 生成的 C 源文件。

## 移植 lex 规范

只要能避免使用一个实现的“实现特有”的特征，通常可以编写可移植的 lex 规范。特殊的问题是：

- 如果想移植到 AT&T lex，不要使用排它性的起始状态。
- 最大的表大小不同，所以适合一个实现的词法分析程序对另一个来说也许太大了。
- 标记缓冲区 **yytext** 的大小范围可以从 100 个字节到 8K 个字节。
- 只从通常的输入文件 **yyin** 中获得输入，因为从其他地方获得输入没有标准化。参见本章的“从字符串中输入”和“包含操作”的详细资料。

## 移植生成的 C 词法分析程序

lex的大多数版本都生成可移植的C代码，并且通常可以毫不费力地将这些代码移到任何 C 编译程序中。

### 库

lex 库通常只以目标码形式来提供。对于两个标准的库例程 —— **main()**和

**yywrap()**，这没有问题，因为可以很容易地编写自己的版本。参见本章“lex库”一节。有些版本，特别是AT&T lex，将其他例程插入库中，例如**yyreject()**和**yyless()**。如果使用它们，就不能移植词法分析程序，除非你拥有库源。flex不使用库，所以它的代码通常是最具有可移植性的。

## 缓冲区大小

你也许想调整一些缓冲区的大小，flex使用两个输入缓冲区，每个缓冲区的大小都默认为8KB，这对于一些微型计算机实现来说也许太大了，参见本章“yytext”一节中有关调整缓冲区大小的详细资料。

## 字符集

最困难的移植性问题涉及字符集。由每个lex实现生成的C代码将字符代码作为词法分析程序的表中的索引。如果原始机器和目标机器都使用相同的字符代码，例如ASCII，被移植的词法分析程序就可以工作。必须处理不同的行结束约定：UNIX系统用简单的“\n”结束一行，而MS-DOS和其他系统使用“\r\n”。通常在两种情况下都可以使词法分析程序忽略“\r”并将“\n”看做行结束。

当原始机器和目标机器使用不同的字符集时，如ASCII和EBCDIC，词法分析程序根本不工作，因为作为索引使用的所有的字符代码是错误的。高级用户有时能后处理（post-process）这些表以重新为其他的字符集构建这些表，但是一般惟一合理的途径是找到在目标机器上运行的lex版本，否则重新定义词法分析程序的输入程序以将输入字符转换成原始的字符集。参见本章“从字符串中输入”一节以了解如何改变输入程序。

AT&T lex和MKS lex中的转换表为显式地指定字符代码提供了一种方法，所以如果愿意为所有的字符使用固定的数字码，那么就能编写可移植性强的词法分析程序。参见本章“字符变换”一节。

# 正则表达式语法

lex 模式是由编辑程序和实用程序（如 *grep*）使用的正则表达式的扩展版本。正则表达式由常规字符（代表它们本身）和元字符（在一种模式中具有特殊的含义）组成。不同于下面列出的所有字符都是常规字符。空白（空格和制表位）将模式与动作分隔开，因此必须将它们引起来才能包含在模式中。

## 元字符

. 匹配除换行符 "\n" 以外的任意单个字符。

[] 匹配括号中字符的任意一个。用 "-"（短划线）指示字符的范围，例如 "[0-9]" 指10个数字中的任意一个。如果开括号之后的第一个字符是短划线或闭括号，那么它就不能被解释为元字符。如果第一个字符是抑扬符号 "^"，那么它的含义就变为匹配括号内字符以外的任意字符。（例如字符类将匹配一个换行符，除非明确地拒绝它。）除了以 "\" 开始的C转义序列被识别以外，其他的元字符在方括号中没有特殊含义。POSIX lex 为国际化增加了更多的特殊方括号模式。参见下面的详细资料。

* 匹配前面正则表达式的零次或多次出现。例如，模式

```
a.*z
```

匹配以 "a" 开始并且以 "z" 结束的任意字符串，例如 "az"、"abz" 或 "alcatraz"。

\+ 匹配前面正则表达式的一次或多次出现。例如：

```
x+
```

匹配 "x"、"xxx" 或 "xxxxxx"，但是不匹配空字符串，而且

```
(ab)+
```

匹配 "ab"、"abab"、"ababab"，等等。

? 匹配前面正则表达式的零次或一次出现。例如：

```
-?[0-9]+
```

指示具有可选的前导一元减号的数字。

{} 意味着根据括号内部的不同而不同。单个数字“{n}”意味着前面的模式重复 n 次，例如：

```
[A-Z]{3}
```

表示任意 3 个大写字母。如果大括号包含由逗号分隔的两个数字“{n,m}”，那么它们是前面模式重复的最小数和最大数。例如：

```
A{1,3}
```

表示字母“A”出现 1 次到 3 次。如果第二个数字丢失，就意味着无穷大，所以“{1,}”意味着“+”，而“{0,}”意味着“*”。

如果大括号包含一个名字，它指示用那个名字来替换。

\ 如果后面的字符是小写字母，那么它就是 C 转义序列，例如制表位“\t”。一些实现允许采用如“\123”和“\x3f”这种形式的八进制和十六进制字符。否则，“\”引用后面的字符，所以“*”匹配一个星号。

() 将一系列正则表达式归组。“*”、“+”和“{}”中的每一个都直接作用于它左侧的表达式，而且“|”通常影响左侧和右侧的所有内容。圆括号能改变这种情况，例如：

```
(ab|cd)?ef
```

匹配“abef”、“cdef”或只是“ef”。

| 匹配前面的或随后的正则表达式。例如：

```
twelve|12
```

匹配“twelve”或“12”。

"..." 逐字匹配引号内的每个字符。不同于“\”的元字符会失去它的含义。例如：

```
"/*"
```

匹配两个字符“/*”。

/ 只有当有后面的表达式跟随时才匹配前面的表达式。例如：

```
0/1
```

匹配字符串“01”中的“0”，但不匹配字符串“0”或“02”中的任何字符。每个模式只允许有一个斜杠，并且模式不能同时包含斜杠和后缀“$”。

^　作为正则表达式的第一个字符，它匹配行的开始；它还在方括号中用于否定。其他方面没有特殊情况。

$　作为正则表达式的最后一个字符，它匹配行的结尾——其他方面没有特殊情况。当位于表达式的结尾时与“\n”的含义相同。

<>　位于模式开头的尖括号内的一个或一列名字使那个模式只应用于指定的起始状态。

<<EOF>>

　(只适用于 flex) 在 flex 中，这个特殊的模式 <<EOF>> 匹配文件的结尾。

## POSIX 扩展

POSIX采用可移植的和语言无关方式来定义新的正则表达式语法，以处理不同于 ASCII 的字符集和不同于英文的语言。这些都被处理正则表达式（例如 *sed* 和 *grep*，以及 lex）的所有实用程序所接受。这三个新的表达式是理序符号、等价类和字符类。

理序符号（collating symbol）是作为单个字符对待的多字符序列，例如西班牙语中的“ch”和“ll”或荷兰语中的“ij”。理序符号写在方括号和点之间，例如，“[.ch.]”。理序符号只在字符类表达式中被识别，例如“[abc[.ch.]d]”。

等价类（equivalence class）是分类在一起的字符集，一般为相同字母的加重音形式，例如“a”、“á”和“â”。类中的字符包围在方括号和等号之间。例如，“[=a=]”代表类中的字符中的任何一个，在这个例子中与“[aáâ]”是一样的。

字符类（character class）表达式代表由 *ctype* 宏处理的命名类型的任意字符，这种类型有 **alnum**、**alpha**、**blank**、**cntrl**、**digit**、**graph**、**lower**、**print**、**punct**、**space**、**upper** 和 **xdigit**。类名包围在方括号和冒号之间。例如，“[:digit:]”等价于“[0123456789]”。

本书编写时没有 lex 版本能处理 POSIX 扩展的任何内容，但 flex 将在不久的将来处理它们。

# REJECT

通常lex将输入分成非重叠的标记。但是有时想要一个标记的所有出现，即使与其他的标记重叠在一起。这个特殊的动作**REJECT**可以让你完成这项工作。如果动作执行**REJECT**，那么lex在概念上放回由模式匹配的文本并且找到下一个最佳匹配它的文本。这个示例找到文件中单词“pink”、“pin”和“ink”的所有出现，甚至当它们重叠的时候：

```
...
%%
pink    { npink++; REJECT; }
ink     { nink++; REJECT; }
pin     { npin++; REJECT; }
.       |
\n      ;       /* 丢弃其他字符 */
```

如果输入包含单词“pink”，那么所有的3个模式都匹配。没有**REJECT**语句，那么就只有“pink”匹配。

使用**REJECT**的扫描程序比不使用**REJECT**的程序大而且慢，因为它们需要相当大的额外信息来允许回溯和重新进行词法分析。

# 从yylex()中返回值

当模式匹配标记时所执行的C代码可以包括**return**语句，从**yylex()**中向调用者——通常是由yacc产生的语法分析程序——返回一个值。下一次调用**yylex()**时，扫描程序从它停止的地方继续。

当扫描程序匹配语法分析程序感兴趣的标记时（例如，关键字、变量名或操作符），它采用**return**将标记传递回语法分析程序。当它匹配语法分析程序不感兴趣的标记时（例如，空白或注释），它就不返回，并且扫描程序立即继续匹配下一个标记。

这意味着只调用**yylex()**不能重启词法分析程序。必须使用**BEGIN INITIAL**将

它重置为默认状态，通过 **unput()**丢弃任何输入文本的缓冲，并设置一些其他的东西，以便对 **input()**的下一个调用开始读取新的输入。

flex 使重新启动变得相当容易。对 **yyrestart(*file*)**的调用指定开始读取那个文件（这里，*file* 是标准的 I/O 文件指针）。

在 pclex 中可以用宏 **YY_INIT** 重置扫描程序的状态，你也许想回绕 **yyin** 或将它指定到一个新文件。

在 MKS lex 中可以使用 **YY_INIT**、**YY_INIT** 是只在扫描程序文件中工作的宏，或者调用 **yy_reset()**，它是可以从任何地方调用的例程。

# 起始状态

在定义段可以声明起始状态，也称为起始条件或起始规则。起始状态用于限制某些规则的范畴，或者改变词法分析程序处理部分文件的方式。例如，假设有下面的 C 预处理程序指令：

```
#include <somefile.h>
```

通常，尖括号和文件名作为 5 个标记来扫描，分别是 "<"、"somefile"、"."、"h" 和 ">"，但是在 "#include" 之后，它们是单个文件名标记。可以使用起始状态来规定只在某些时候应用规则集。注意，没有起始状态的那些规则能应用于任何状态（注 4）！动作中的 **BEGIN** 语句（参见 BEGIN 节）设置了当前的起始状态。例如：

```
^"#include" { BEGIN INCLMODE; }
<INCLMODE>"<"[^>\n]+">" { ... 使用名称完成某些操作 ... }
<INCLMODE>\n { BEGIN INITIAL; /* 返回到正常状态 */ }
```

用 %s 行声明起始状态。例如：

注 4：实际上，这是非常常见的 lex 程序设计错误。这个问题由排它性起始状态来调整，如本节所述。

```
%s PREPROC
```

创建起始状态 **PREPROC**。于是，在规则段，预先搁置了 **<PREPROC>** 的规则只应用在状态 **PREPROC** 中。lex 起始的标准状态是状态零，也称为 **INITIAL**。

flex 和不同于 AT&T lex 的大多数版本还有由 %x 声明的排它性起始状态。普通起始状态和排它性起始状态的区别是，没有起始状态的规则在排它性状态处于激活时不匹配。实际上，排它性状态比普通状态更有用，如果lex版本支持它们，你很可能想使用它们。

排它性起始状态使它很容易做一些事情，例如识别C语言注释：

```
%x CMNT
%%
"/*" BEGIN CMNT;              /* 注释模式的开关 */
<CMNT>.            |
<CMNT>\n ;                    /* 丢掉注释文本 */
<CMNT>"*/"BEGIN INITIAL;      /* 返回到常规模式 */
```

上面的语句使用普通起始状态就不会工作，因为所有的常规标记模式在**CMNT**状态下仍然是激活的。

在缺乏排它性起始状态的lex的版本中，通过给普通状态指定一个明确的状态并在每个表达式上放置起始状态，可以得到同样的效果。假定普通状态称为 **NORMAL**，上面的例子变为：

```
%s NORMAL CMNT
%%
%{
        BEGIN NORMAL;        /* 以NORMAL状态开始 */
%}
<NORMAL>"/*"                 BEGIN CMNT; /* 注释模式的开关 */
<CMNT>.            |
<CMNT>\n ;         /* 丢掉注释文本 */
<CMNT>"*/" BEGIN NORMAL;       /* 返回到常规模式 */
```

这个程序与上面的扫描程序并不是非常等效的，因为在每次调用例程**yylex()**时执

行**BEGIN NORMAL**，也就是在将值返回给语法分析程序的任何标记之后。如果那出现问题，那么会出现“第一次”标志，例如：

```
%s NORMAL CMNT
%{
      static int first_time = 1;
%}
. . .
%%
%{
      if(first_time) {
            BEGIN NORMAL;
            first_time = 0;
      }
%}
. . .
```

# unput()

例程**unput(*c*)**将字符*c*返回给输入流。与类似的stdio例程**unputc()**不同，可以在一行中多次调用**unput()**向输入中插入几个字符。由**unput()**“推回”的数据限制会发生变化，但它至少总是和词法分析程序识别的最长标记一样大。

一些实现可以让你重新定义**input()**和**unput()**来改变扫描程序的输入源。如果重新定义**unput()**，必须准备处理多个被推回的字符。如果扫描程序本身调用**unput()**，那么它将总是放回刚刚从**input()**中得到的同样的字符，但是没有从用户代码中调用**unput()**来完成这事的必要。

当扩展宏时，例如C的**#define**，需要插入宏的文本来代替宏调用。一种方法是调用**unput()**来推回文本，例如：

```
... in 词法分析程序 action code ...
char *p = macro_contents();
char *q = p + strlen(p);

while(q > p)
     unput(*--q);              /* 从右到左推回 */
```

# yyinput()、yyoutput()、yyunput()

lex的某些版本，特别是AT&T lex，提供函数**yyinput()**、**yyoutput()**和**yyunput()**分别作为宏**input()**、**output()**和**unput()**的包装。这些函数的存在使它们可以从其他源模块中被调用，特别是在lex库中。如果需要它们，而且lex的版本又没有定义它们，那么就要亲自在用户子例程段定义它们：

```
int yyinput(void) { return input(); }
int yyoutput(int c) { output(c); }
int yyunput(int c) { unput(c); }
```

# yyleng

只要是在扫描程序匹配标记时，标记的文本就被存储在以空字符终止的字符串**yytext**中，而且它的长度存储在**yyleng**中。**yyleng**中的长度与由**strlen(yytext)**返回的值是相同的。

# yyless()

从与规则相关的代码中调用**yyless(*n*)**，这条规则"推回"除标记开头的*n*字符以外的所有字符。当决定标记之间边界的规则不方便表示为正则表达式时，它是很有用的。例如，考虑一个匹配引用字符串的模式，但是字符串中的引号可以用反斜杠转义：

```
\"[^"]*\"     { /* 字符在闭引号\前吗?  */
               if(yytext[yyleng-2] == '\\') {
                    yyless(yyleng-1); /* 返回最后的引号 */
                    yymore();         /* 追加下一个字符串 */
              } else {
              ...  /* 处理字符串 */
      }
}
```

如果在闭引号之前用反斜杠结束引用串，那么它就使用**yyless()**推回闭引号，并

且 **yymore()**(参见该节）告诉 lex 给这个标记附加下一个标记。下一个标记是以被推回引号开始的引用串的其余部分，所以整个字符串都在 **yytext** 中结束。

对 **yyless()**的调用与调用具有被推回的字符的 **unput()**具有相同的效果，但是 **yyless()**通常更快，因为它能利用被推回的字符就是刚刚从输入中提取的字符的事实。

**yyless()**的另一个用处是使用不同起始状态的规则重新处理标记：

```
sometoken { BEGIN OTHER_STATE; yyless(0); }
```

**BEGIN** 告诉 lex 使用另一个起始状态，并且调用 **yyless()**推回标记的所有字符，所以它们可以使用新的起始状态重新读取。如果新的起始状态不能启用优先于当前模式的不同模式，那么当重复地识别并推回相同的标记时，**yyless(0)**就导致扫描程序的无限循环。

# yylex()

由 lex 创建的扫描程序有入口点 **yylex()**，调用 **yylex()**启动或重新开始扫描。如果 lex 动作执行将数值传递给调用程序的 **return**，那么对 **yylex()**的下次调用就从它停止的地方继续（参见本章“从 yylex()中返回值”一节以了解如何开始扫描一个新文件）。

## yylex()中的用户代码

规则段中的所有代码都被拷贝到 **yylex()**。以空白开始的行被假定是用户代码。“%%”后的代码直接放置在接近扫描程序的开始处，在第一条执行的语句之前。这是声明特殊规则的代码中所使用的局部变量的好地方。记住：尽管这些变量的内容从一条规则到另一条规则被保护起来，如果扫描程序返回并被再次调用，那么自动变量就不会保持相同的值。

当模式被匹配时就执行那个模式行上的代码。这些代码应该是一行上的以分号结

束的语句或者是大括号围绕的代码块。当代码不在大括号中时，一些实现就拷贝整个行，而另一些只拷贝一条语句。例如，在下面的情况中：

```
[0-9]+          yylval.ival = atoi(yytext); return NUMBER;
```

一些版本会丢弃 **return** 语句。(是的，这是很差的设计。) 作为一条规则，如果有多条语句就使用大括号：

```
[0-9]+          { yylval.ival = atoi(yytext); return NUMBER; }
```

如果行上的代码是单竖线，那么这个模式就使用和下一个模式相同的代码：

```
%%
colour      |
color { printf("Color seen\n");
```

出现在第一个模式行之后的以空白开始的代码行也被拷贝到扫描程序，但是它们最终应放到哪里尚无定论。这些行应该只包含注释：

```
rule1 { some statement; }

        /* 这个注释描述了下列规则 */

rule2 { other statement; }
```

# yymore()

可以从与规则相关的代码中调用 **yymore()**，这条规则告诉 lex 给这个标记附加下一个标记。例如：

```
%%
hyper yymore();
text printf("Token is %s\n", yytext);
```

如果输入字符串是“hypertext”，那么输出将是“Token is hypertext”。

在用正则表达式定义标记边界不方便或不实用的地方，使用**yymore()**通常是很有用的。参见“yyless()”中的示例。

# yytext

每当词法分析程序匹配标记时，标记的文本就存储在以空字符结尾的字符串**yytext**中。在lex的一些实现中，**yytext**是如下声明的字符数组。

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

在另一些实现中，它是以：

```
extern char *yytext;
```

声明的字符指针，因为C语言几乎用同样的方式看待数组和指针，编写以两种方式工作的lex程序几乎总是可能的。如果在其他的源文件中引用**yytext**，那么必须确保正确地引用它。POSIX lex包括一些新的定义行%**array**和%**pointer**，为了与其他源文件中的代码相兼容，可以用%**array**和%**pointer**来强制**yytext**被定义成一种方式或另一种方式。

每次匹配一个新标记时就替换**yytext**的内容。如果**yytext**的内容在后面还要使用（例如，由调用lex扫描程序的yacc语法分析程序中的一个动作使用），通过使用**strdup()**或者类似的例程保存字符串的拷贝，从而使字符串的拷贝位于刚刚分配的存储器中。

如果**yytext**是一个数组，那么比**yytext**长的任何标记都将溢出数组并导致以一些很难预测的方式失败。在AT&T lex中，标准大小的**yytext[]**是200个字符，而在MKS lex中，它是100个字符。即使**yytext**是一个指针，指针还是指向有限大小的I/O缓冲区，而且非常长的标记也会出现类似问题。在flex中默认的I/O缓冲区是16KB，意味着它最多处理8KB的标记，这无疑是足够用了。在pclex中，缓冲区只有256个字节。这就是为什么将多行注释当做单个的标记来识别是个坏想法的原因，因为注释可以非常长并且通常会溢出缓冲区，不管缓冲区有多大。

# 扩大 yytext

可以正常地编写词法分析程序，使单个标记不会大于你的lex版本中的默认标记缓冲区，但是有时默认缓冲区真是太小了。例如，你可能正在使用默认大小为100个字符的词法分析程序处理允许128个字符名的语言。增加缓冲区大小的技术根据版本的不同而不同。

## AT&T lex和MKS lex

这两个版本都使**yytext**成为一个静态字符数组，它的大小由编译时的符号**YYLMAX**设置。在这两种情况下，在扫描程序的定义段可以重新定义**YYLMAX**:

```
%{
#undef YYLMAX          /* 去掉默认的定义 */
#define YYLMAX 500     /* 新定义的大小 */
%}
```

## flex

因为默认标记的大小是8KB，很难想像要求它更大时的情况，但是如果内存特别紧，你也许想使缓冲区更小一些。通过称为**yy_create_buffer()**的例程运行时创建flex缓冲区，并且**yy_current_buffer**指向当前缓冲区。将创建更小的缓冲区的例程插入用户子例程段并且在开始扫描之前调用它可以创建任意大小的缓冲区。(这个例程必须在扫描程序文件中，因为它引用的一些变量被声明为静态的。)

```
%%
setupbuf(size)
int size;
{
        yy_current_buffer = yy_create_buffer( yyin, size );
}
```

## pclex

Abraxas pclex是基于flex的，所以它使用类似于flex的缓冲模式，但是它使用更小的、被声明为静态的变量。缓冲区的大小是**YY_BUF_SIZE**，它被定义为

**F_BUFSIZ**的两倍，默认为128。通过改变两者之一可以改变缓冲区的大小。最大的输入行长度**YY_MAX_LINE**默认定义为**F_BUFSIZ**，所以用下面的方法可以很容易地增加**F_BUFSIZ**：

```
%{
#undef F_BUFSIZ          /* 去掉默认的定义 */
#define F_BUFSIZ 256     /* 新定义的大小 */
%}
```

对于lex的两个MS-DOS版本，记住大多数MS-DOS C编译程序对于静态数据和堆栈的总量都限制为64KB，而非常大的标记缓冲区通常会大大地超过64KB。

# yywrap()

当词法分析程序遇到文件结尾时，它调用例程**yywrap()**来找出下一步要做什么。如果**yywrap()**返回0，那么扫描程序就继续扫描，而如果它返回1，那么扫描程序就返回报告文件结尾的零标记。

lex库中的**yywrap()**的标准版本总是返回1，但是可以用你自己的值取代它。如果**yywrap()**返回指示有更多输入的0，那么它首先需要调整指向新文件的**yyin**，可能使用**fopen()**。

自版本2.3.7以来，flex将**yywrap()**定义为宏，这使得定义自己的版本更困难，因为在定义程序或宏之前必须取消这个宏的定义。为此，将下面的形式放置在规则段的开头：

```
%{
#undef yywrap
%}
```

flex的未来版本将符合POSIX lex标准，这个标准声明**yywrap()**在库中是一个例程，在这种情况下，用户定义的版本自动具有优先权。

# 第七章

# yacc 语法参考

本章内容：

- yacc 语法的结构
- 以字母为顺序介绍 yacc 的特征

本章讨论yacc语法的格式并描述可用的各种特征和选项，总结前几章的示例讨论的特性并概述没有提到的特征。

阐述了 yacc 语法的结构后，本章其余几节将按特征的字母先后顺序进行讲述。

## yacc 语法的结构

yacc 语法包括三部分：定义段、规则段和用户子例程段。

```
... 定义段 ...
%%
... 规则段 ...
%%
... 用户子例程段 ...
%%
```

各部分由以两个百分号开头的行分开。尽管某一部分可以为空，但前两部分是必需的。第三部分和前面的百分号可以省略（lex 使用同一种结构）。

### 符号

yacc 语法由符号组成，即语法的“词”。符号是一串不以数字开头的字母、数字、

句点和下划线。符号**error**专用于错误恢复，另外yacc对任何符号都不会附加“先验”（priori）的意义。

由词法分析程序产生的符号叫做终结符号或标记。定义在规则左侧的就叫做非终结符号或非终结。标记也可能是字面上引用的字符（参见本章“字面标记”一节）。通常遵循的约定是将标记大写，非终结符号小写，本节将遵循这个约定。

## 定义段

定义段包括文字块、逐字拷贝到生成的C文件开头部分的C代码，通常包括声明和#include行。可能有%**union**、%**start**、%**token**、%**type**、%**left**、%**right**和%**nonassoc**声明（参见本章的“%union声明”、“开始声明”、“标记”、“%type声明”和“优先级、结合性和操作符声明”等有关节）。也可以包含普通的C语言风格的注释，包括在“/*”和“*/”之间。所有这些都是可选择的，所以在很简单的语法分析程序中，定义段可能完全是空的。

## 规则段

规则段由语法规则和包括C代码的动作组成，参见本章的“规则”和“动作”这两节的详细内容。

## 用户子例程段

yacc将用户子例程段的内容完全拷贝到C文件中，通常这部分包括从动作调用的例程。

在大的程序中，将支持代码放进单独的源文件中会更方便，这可以在改变yacc文件时减少重新编译的量。

# 动作

动作是yacc与在语法中规则相符时执行的C代码，动作一定是C复合语句。举例来说：

```
date: month '/' day '/' year { printf("date found"); } ;
```

通过使用后面跟有数字的美元符号，动作可以查阅在规则中与符号有关的值，举例来说，冒号后面的第一个符号是数字1:

```
date: month '/' day '/' year
            { printf("date %d-%d-%d found", $1, $3, $5); }
      ;
```

名字“$$”指的是冒号左边符号的值，符号值可以有不同的C类型。参见本章“标记”、“%type声明”和“%union声明”等节以了解详细内容。

yacc对于没有动作的规则使用默认形式:

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

## 嵌入式动作

尽管yacc的语法分析技术只允许动作在规则的末端，但yacc可以自动模拟嵌入在规则内部的动作。如果在规则内部写入一个动作，yacc就会创造一个右侧为空并且左边是自动生成的名字的规则，使得嵌入的动作进到规则的动作里去，用自动生成的名字代替最初的规则内的动作。例如，下面的句子是等价的:

```
thing:          A { printf("seen an A"); } B ;

thing:          A fakename B ;
fakename: /* 空值 */ { printf("seen an A"); } ;
```

虽然这个特征是很有用的，但它还是有一些特例的。嵌入式动作在规则内变成一个符号，所以它的值（凡是“$$”的赋值）像任何其他的符号一样对于规则末端的动作都是可用的:

```
thing:         A { $$ = 17; } B C
               { printf("%d", $2); }
        ;
```

这个例子打印17，这两种方式都可以将A值作为**$1**，规则末端的动作可将嵌入式动作的值作为**$2**，**B**和**C**的值分别是**$3**和**$4**。

嵌入式动作在其他可接受的语法中会引起移进/归约冲突。例如，下面的语法没有问题：

```
%%
thing:     abcd | abcz ;

abcd: 'A' 'B' 'C' 'D' ;
abcz: 'A' 'B' 'C' 'Z' ;
```

但是如果添加一个嵌入式动作，就会有一个移进/归约冲突：

```
%%
thing:     abcd | abcz ;

abcd: 'A' 'B' { somefunc(); } 'C' 'D' ;
abcz: 'A' 'B' 'C' 'Z' ;
```

第一种情况下，语法分析程序在看到所有4个标记之前不需要对是分析**abcd**还是分析**abcz**做出决定，看到时才会显示出结果。第二种情况下，需要在分析“B”后做决定，但是在那时还没有足够的输入决定哪条规则正在进行分析。如果嵌入式动作在“C”后就没有问题，因为yacc能够向前看一个标记来查看接下来是“Z”还是“D”。

## 嵌入式动作的符号类型

由于嵌入式动作与语法符号没有联系，所以没有办法声明由嵌入式动作返回的值的类型。如果正在使用“**%union**”和有类型的符号值，那么在引用动作值时就得将值放入尖括号中，例如将它放入嵌入式动作时的$<*type*>$，以及在规则末端动作中引用它时的$<*type*>3（使用合适的数字）（参见本章“符号值”一节）。如果像前面的例子那样，是一个全部使用*int*值的简单的语法分析程序，就不需要给出类型。

## 废弃的特征

在yacc早期版本中，在一个动作之前要求一个等号，老的语法中仍然包括它们，但在现今版本中已经不再要求，也不被认为是好的风格。

# 歧义和冲突

由于语法有歧义或包含冲突，yacc对于语法规范的翻译可能失败。一些情况下，语法确实有歧义，也就是说对于一个单独的输入字符串有两种可能的分析，而且yacc处理不了。另一些情况下，语法并无歧义，但是yacc使用的语法分析技术不足以分析这个语法。在一个并无歧义却带有冲突的语法中的问题是，语法分析程序需要向前看多于一个的标记才能决定使用两个可能的分析中的哪一个。

参见本章“优先级、结合性和操作符声明”一节和第八章“yacc歧义和冲突”以了解详细内容和如何处理这些问题的建议。

## 冲突的类型

当yacc创建语法分析程序时可能发生两种冲突：“移进/归约”和“归约/归约”。

### 移进/归约冲突

当一个输入字符串有两种可能的分析，而且其中一个分析完成一个规则（归约选项），而另一个却没有（移进选项）时，移进/归约冲突就发生了。例如，下面语法中有一个移进/归约冲突：

```
%%
e:        X'
     |    e  +' e
     ;
```

对于输入字符串“X+X+X”，有两种可能的分析：“(X+X)+X”或“X+(X+X)”（参见图3-3，该图显示了一个相似的冲突）。采用归约选项，使得语法分析程序

使用第一个分析，移进选项使用另一个。除非用户在操作声明中选定，否则yacc总是选择移进选项。详见本章“优先级、结合性和操作符声明”一节。

### 归约/归约冲突

当同样的标记可以完成两个不同的规则时就会发生归约/归约冲突。例如：

```
%%
prog: proga   progb ;

proga:       X' ;
progb:      'X' ;
```

一个“X”可能是**proga**，也可能是**progb**。大多数归约/归约冲突都不像这样明显，但是几乎在任何情况下它们在语法中都表现为错误。关于处理冲突的详细内容请见第八章“yacc歧义与冲突”

# yacc中的程序错误

尽管yacc是一个非常成熟的程序（AT&T yacc源代码本质上已经10多年没有改变了），但在已发布的版本中有一个程序错误是很常见的，并且几个古怪的地方也常会被误认为是程序错误。

## 真正的程序错误

特别是在AT&T yacc中，有一些真正的程序错误。（感谢Megatest公司的Dave Jones提供的例子。）

### 错误处理

一些AT&T yacc的老的版本对这个语法都处理不当：

```
%token a
%%
s      : oseq
```

```
        ;
oseq:   /* 空值 */
        | oseq a
          oseq error
        ;
```

有错误的版本在错误状态下做默认的归约，尤其是在状态 2 情况下，**y.output** 列表文件中有一个默认归约：

```
. reduce 1
```

正确的默认行为是检测错误标记：

```
. error
```

在 yacc 源程序中这个错误是一个 off-by-one 编码错误，任何带有 AT&T yacc 源程序的厂商都能很容易地修正它。

尽管能修正，错误恢复和yacc的默认归约之间还是有不好的影响。当计算移进和归约表时yacc不能考虑错误标记，有时即使已经有了一个向前查看标记表明有一个句法错误而且不应该归约这个规则时它仍会归约该规则。如果你打算做一些错误恢复来看一下在开始进入错误状态前什么样的规则将被归约的话，看一下 *y.output* 文件。你可能需要做比计划多许多的工作来恢复。

### 声明文字标记

AT&T yacc 不能诊断那些改变文字标记的标记号的尝试。

```
%token '9' 17
```

这将在输出文件中生成无效 C 代码。

## 无限递归

yacc语法中一个常见的错误是创建没有任何方法能结束的一个递归规则，Berkeley yacc（至少到 1.8 版本）不能诊断这个语法：

```
%%
xlist:        x_ist 'X' ;
```

其他版本能进行诊断。

## 不真实的程序错误

有一些地方 yacc 看起来是错误行为，但实际上是在做它应该做的事情。

### 交换优先级

人们有时试图使用“**%prec**”交换两个标记的优先级：

```
%token NUMBER
%left PLUS
%left MUL
%%
expr    :       expr PLUS expr %prec MUL
        |       expr MUL expr %prec PLUS
        |       NUMBER
        ;
```

这个例子看起来是让**PLUS**有比**MUL**更高的优先级，而实际上是使它们相同。优先级机制通过将被移进的标记的优先级与规则的优先级进行比较来解决移进/归约冲突。这种情况下有好几种冲突。当语法分析程序看到了“expr PLUS expr”并且下一个标记是**MUL**时出现典型的冲突。在没有“**%prec**”时，规则中的“**PLUS**”的优先级低于**MUL**的优先级，并且 yacc 对此进行移进。但是在拥有“**%prec**”时，规则和标记都具有“**MUL**”的优先级，因为**MUL**是左结合的，所以进行归约。

一种可能是引入伪标记，例如“**XPLUS**”和“**XMUL**”，利用它们本身的优先级使用“**%prec**”。另一个更好的解决方法是重新写这个语法来表现这个意图，在这种情况下，交换“**%left**”行（参见本章“优先级、结合性和操作符声明”一节）。

### 嵌入式动作

当在规则中部而不是在规则末端写入一个动作时，yacc创造一个匿名的规则来触发这个嵌入式动作。有时这个匿名规则引起不可预见的移进/归约冲突。详见本章“动作”一节。

## 结束标记

每个yacc语法都包括一个称为结束标记的伪标记，它标志输入的结束。在yacc列表中结束标记通常用**$end**来表示。

词法分析程序必须返回一个0标记来表示输入的结束。

## 错误标记和错误恢复

当yacc探测到一个语法错误时，也就是当它收到一个不能分析的输入标记时，就会试图使用以下步骤来恢复这个错误：

1. 调用**yyerror**（“语法错误”）。通常将这个错误报告给用户。
2. 会放弃任何部分被分析的规则，直到返回到能够移进这个特殊的**error**符号的状态。
3. 继续分析，从移进**error**开始。
4. 在3个标记被成功移进之前如果又出现一个错误，yacc不报告另一个错误，只是返回到步骤2。

参见第九章“错误报告和恢复”，该章对错误恢复有详细的描述。也可参见本章“YYERROR”、“YYRECOVERING()”、“yyclearin”和“yyerrok”等节了解关于控制错误恢复的详细内容。

AT&T yacc的一些版本中包含使错误恢复失败的漏洞。更多信息详见本章“yacc中的程序错误”一节。

# %ident 声明

Berkeley yacc 允许 %ident 在定义段将标识字串引入这个模块：

```
%ident "identification string
```

在生成的C代码中产生一个 #ident。C 编译程序一般将这些标识字串放在目标模块中，使用 UNIX *what* 命令可以找到它们。

目前没有其他的 yacc 版本支持 %ident，一个最简便的达到这个效果的方法是：

```
%{
#ident "identification string"
%}
```

# 继承的属性（$0）

yacc 符号值通常表现为“继承的属性”（inherited attribute）或“合成属性”（synthesized attribute）。（yacc 中称做值，而其他编译领域里通常称为属性。）属性以标记值开始，即从语法分析树的叶子开始。每次规则被归约时，信息概念性地在语法分析树中上移，并且它的动作根据规则右侧符号值合成符号（$$）的值。

有时需要将信息以另外的方式传递，即从语法分析树的根部向语法分析树的叶子方向传递。考虑下面的例子：

```
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); }
      ;
```

在动作中有可用于**namelist**的class和type是很有用的，它们可用于错误检测和进入符号表。yacc通过允许访问它的当前规则的左侧的内部堆栈上的符号（经由**$0**、**$-1**等）来使这成为可能。在这个例子中，调用**mksymbol()**中的**$0**指的是在**namelist**产生式的符号之前被堆栈的**type**的值，而且根据类型是**REAL**还是**INTEGER**来决定值为1还是2，并且**$-1**指的是class，如果是**GLOBAL**或**LOCAL**，那么它的值分别为1或者是2。

尽管继承的属性非常有用，但它们也是很难发现的错误来源，使用继承属性的动作必须考虑语法中使用规则的每个地方。在这个例子中，如果将语法改变为使用其他什么地方的名字表，那么必须确信在出现名字表的新的地方的前面要有适当的符号，以便**$0**和**$-1**得到正确的值：

```
declaration:        STRING namelist ; /* 无法工作! */
```

继承的属性有时会非常有用，特别是对于像C语言变量声明那样的句法上非常复杂的结构。但是在大多数情况下，使用合成属性更安全，而且同样简单。在上面的例子中，名字表规则能创建一个链接的将被声明的名字的引用列表，并返回那个列表的指针作为它的值。声明的动作可以采用类、类型和名字表值而且同时将类和类型指派给名字表中的名字。

## 继承属性的符号类型

使用继承属性的值时，普通的值声明技术（如%**type**）不起作用。由于对应值的符号在规则中不出现，所以yacc不能够确定什么类型是正确的。在动作代码中必须采用明确的类型提供类型名。在前面的例子中，如果**class**和**type**的类型是**cval**和**tval**，那么最后两行实际上可以理解为：

```
namelist:   NAME            { mksymbol($<tval>0, $<cval>-1, $1); }
        |   namelist NAME   { mksymbol($<tval>0, $<cval>-1, $2); }
```

参见本章“符号值”一节以了解其他的信息。

# 词汇的反馈

语法分析程序有时向词法分析程序反馈一些信息以处理其他困难的情况。例如，思考下面这样一个输入句法：

```
message (any characters)
```

在这个特殊的上下文关系中，圆括号的作用像字符串引用一样。不能看到一个开圆括号就决定分析一个字符串，因为可能在其他的语法中有不同的用法。处理这种情况的直接方式是将内容从语法分析程序中反馈到词法分析程序中，举例来说，当希望一个上下文相关的结构出现时，在语法分析程序中设置一个标记：

```
/* 语法分析程序 */
%{
int parenstring = 0;
}%
. . .
%%
statement: MESSAGE { parenstring = 1; }  '(' STRING ')' ;
```

并且在词法分析程序中寻找它：

```
%{
extern int parenstring;
%}
%s PSTRING
%%
. . .
"message"      return MESSAGE;
"(" {          if(parenstring)
                     BEGIN PSTRING;
               return '(';
    }
<PSTRING>[^)]*      {
               yylval.svalue = strdup(yytext); /* 传递字符串到语法分析程序 */
               BEGIN INITIAL;
               return STRING;
               }
```

这段代码不是安全的（bullet-proof），因为如果有以**MESSAGE**开头的一些其他规则，yacc也许必须向前查看一个标记，在这种情况下，嵌入的动作直到扫描到开圆括号之后才能执行。大多数情况下是没有问题的，因为语法相对简单。

在此例中，也能通过在词法分析程序中设置**parenstring**来处理词法分析程序中的特殊情况，例如：

```
"message"  { parenstring = 1; return MESSAGE; }
```

然而，如果标记**MESSAGE**用在语法中其他的位置不是总跟着圆括号串时就会产生问题。通常可以选择在词法分析程序中完全地执行词汇的反馈或在语法分析程序中部分地执行反馈，最好的选择取决于语法的复杂程度。如果语法简单并且标记没有出现在多重环境中，就可以在词法分析程序中做所有的词法的hackery，如果语法比较复杂，在语法分析程序中标识特殊情况会容易些。

这个途径可能走向极端——一个作者用yacc编写了一个完整的Fortran 77语法分析程序（不是用lex，由于对Fortran进行标记化太奇怪了），并且语法分析程序需要反馈一打特殊环境状态给词法分析程序，这是很杂乱的，但是也比用C语言编写整个语法分析程序和词法分析程序容易得多。

# 文字块

定义段的文字块由行“%{”和“}%”括起来。

```
%{
... 代码和声明 ...
%}
```

文字块内容被逐字拷贝到生成的C源文件中靠近开头的部分，并在**yyparse()**开始之前。文字块通常包括各种说明和规则段代码使用的变量和函数的声明，以及任意必要的头文件的“**#include**”行。

# 文字标记

yacc 将单引号中的字符作为一个标记，在下例中：

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

开括号和闭括号是文字标记。文字标记的标记号是本地字符集中的数字值，通常是 ASCII 码，这和引用符号的 C 语言值是一样的。

词法分析程序通常从输入的相应的单个字符生成这些标记，但是如同其他标记一样，输入字符和生成标记之间的一致性完全适于词法分析程序。一般的技术是使词法分析程序将其他所有不识别的符号看做是文字标记。例如，在 lex 扫描程序中：

```
        return yytext[0];
```

这覆盖了语言中所有的单字符操作符，并让 yacc 捕捉输入中所有不识别的符号。

有些 yacc 版本允许多字符文字标记，例如“<=”，但使用它们的想法并不好，因为不同的 yacc 版本采用不同的、不相容的方法处理它们。如果一个标记的输入表达式超过一个字符，给出名字是比较好的方式：

```
%token LE
```

在扫描程序中：

```
'<=' return LE;
```

# yacc 语法分析程序的可移植性

yacc 语法分析程序在 C 语言实现之间一般是可移植的。移植语法分析程序有两个等级：最初的 yacc 语法，或生成的 C 语言源文件。

## 移植yacc语法

不同的yacc版本很大部分上都是兼容的，每一种都有一些独特的特征，但是通常写语法只使用共同的特征。(例如，使用bison的可重入分析特征的语法分析程序只能和bison一起工作。)

不同的yacc版本处理错误稍有不同。特别是当语法分析程序收到一个错误标记时，语法分析程序可能归约或不归约以上一个标记结束的规则，这取决于生成语法分析程序的yacc版本。**YYERROR()**的精确行为不同。还有，一些版本完成当前规则的归约并且在开始错误恢复之前将RHS标记从分析程序堆栈中移走，有些版本就不这样。

yacc的不同版本有不同的移进限制：最普通的问题就是符号标记的最大数问题，在AT&T yacc中是127。通常可以使用文字字符作为标记来避开这个限制，参见本章“字符集”一节。

## 移植生成的C语言语法分析程序

大多数yacc版本生成可移植的C代码，通常可以毫无困难地将这些代码移到任一个C编译程序中。

### 库

yacc库里仅有的程序是**main()**和**yyerror()**。大多数语法分析程序使用这两个程序的自己的版本，所以库通常是不必要的。

### 字符代码

在两个使用不同字符代码的机器之间移动生成的语法分析程序是棘手的。实际上，必须避免像“0”这样的文字标记，因为语法分析程序将字符代码作为深入内部表的索引，所以在代码“0”为48的ASCII机器上生成的语法分析程序在代码为240的EBCDIC机器上会失败。

yacc将自己的数值分配给符号标记，所以只使用符号标记的语法分析程序能够成功地移植。

# 优先级、结合性和操作符声明

通常，所有的yacc语法都必须是无歧义的。也就是说，在语法中使用规则分析合法输入只有一个可能的方法。

有时，有歧义的语法是很容易使用的。有歧义的语法会引起冲突，也就是说有两个可能的分析，因此yacc有两个不同的方法处理一个标记。当yacc处理一个有歧义语法时，使用默认规则决定哪一种方法来分析一个有歧义的序列。这些规则通常不会产生想要的结果，所以yacc包括操作符声明，让用户可以改变由有歧义语法产生的移进/归约冲突的处理方法（参见本章“歧义和冲突”一节）。

## 优先级和组合规则

大多数程序设计语言有复杂的规则来控制算术表达式的解释。例如，下面的C语言表达式：

```
a = b = c + d / e / f
```

可以被看做：

```
a = (b = (c + ((d / e) / f)))
```

用于决定哪些操作数与哪些操作符组合的规则称为“优先级和结合性”。

### 优先级

优先级给每个操作符分配了一个先后次序“级别”。最高级别的操作符绑定的更紧密。例如，如果“*”比“+”的优先级高，那么“A+B*C”就看做“A+（B*C）”，而“D*E+F”则看做“（D*E）+F”。

### 结合性

当表达式中使用相同操作符或具有相同优先级的不同操作符时，结合性控制语法如何结合表达式，它们或者从左边结合、或者右边结合、或者根本不结合。如果“-”是左结合，表达式“A-B-C”意味着“(A-B)-C”；如果是右结合，该表达式则意味着“A-(B-C)”。

有些操作符，例如Fortran中的**.GE.**，就没有任何结合，例如“A.GE.B.GE.C”不是有效的表达式。

## 操作符声明

操作符声明出现在定义段，可能的声明是“**%left**”、“**%right**”、“**%nonassoc**”(在很老的语法中，有完全与之等价的废弃的“%<”、“%>”、“**%2** 或 **%binary**”)。“**%left**”和“**%right**”声明分别表达左结合或右结合，用“**%nonassoc**”声明非结合操作。

操作符按照优先级增加的顺序进行声明。同一行上声明的所有操作符具有同样的优先级。例如，Fortran 语法中包括：

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

最低优先级的操作符是“+”和“-”，中间优先级的操作符是“*”和“/”，最高优先级的操作符是“POW”，表明的是幂操作符“**”。

## 使用优先级和结合性解决冲突

语法中的每一个标记都可以通过操作符声明赋予优先级和结合性，每条规则都有优先级和结合性，也可以通过规则中的“**%prec**”子句获得优先级和结合性，如果不行的话，规则最右边的标记有一个被赋予的优先级。

只要有移进/归约冲突，yacc就比较需要移进的标记和需要归约的规则的优先级，

按照规则的优先级进行移进和归约。如果具有相同的优先级，yacc就查看结合性，如果是左结合就归约，是右结合就移进，如果没有结合yacc就生成一个错误提示。

## 优先级的典型用法

尽管在理论上使用优先级可以解决各种移进/归约冲突，但很少能比重写语法解决冲突来得干净。优先级声明是为处理具有大量如下规则的表达式语法而设计的：

```
expr OP expr
```

表达式语法几乎总是使用优先级来写。

惟一的其他通用的用法是if-then-else，用优先级解决“悬摆”（dangling）问题比重写语法来解决问题更容易。

详见第八章“yacc歧义和冲突”。也可以参见本章“yacc中的程序错误”一节来了解使用“**%prec**”的通常缺陷。

# 递归规则

要分析一列未确定长度的项目列表，可以按照其本身来定义一个递归规则。例如下面的例子，分析一个可能空的数字列表：

```
numberlist: /* 空值 */
        | numberlist NUMBER
        ;
```

递归规则的细节随被分析的句法的不同而不同。下一个例子分析了一个被逗号分隔的非空的表达式列表，并且符号 **expr** 在语法中的其他地方被定义了：

```
exprlist: expr
        | exprlist ',' expr
        ;
```

任何一个递归规则都必须至少有一个非递归的替代规则(不是指它本身)。否则没有办法中断与之匹配的字符串，这是一个错误（Berkeley yacc 不能诊断这个问题)。

## 左、右递归

写递归规则时，可以将递归引用放在左端尾部，也可以将它放在规则右边的右端尾部，例如:

```
exprlist: exprlist ',' expr ;  /* 左递归 */

exprlist: expr ',' exprlist ;  /* 右递归 */
```

大多数情况下，可以用任意一种方式写语法。yacc处理左递归比右递归效率更高，这是因为内部堆栈跟踪迄今为止在所有部分分析的规则中所看到的所有符号。如果使用“**exprlist**”的右递归版本并且具有内部带有10个表达式的表达式，到10个表达式读完时，堆栈上会有20个条目、一个“**expr**”和每个表达式一个的逗号。当列表结束时，从右向左开始，所有嵌套的**exprlist**将被归约。另一方面，如果使用左递归版本，**exprlist**规则在每一个**expr**之后归约，所以清单在内部堆栈上永远也不会超过3个条目。

一个10个元素的表达式列表在语法分析程序中不会引起问题，但是语法经常分析有数百个项目的列表，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: . . .
```

在这种情况下，一个400语句的程序被分析为有400个语句列表元素，并且400个元素的右递归列表对于大多数yacc分析程序来说都太大了。

右递归语法对于项目列表较短并且希望生成链接的数值列表非常有用:

```
thinglist: THING { $$ = $1; }
        | THING thinglist { $1->next = $2; $$ = $1; }
        ;
```

使用左递归语法，要么以倒序结束链接的列表，或者为了向尾端添加下一件事，在每个阶段都需要增加搜索列表的尾端的额外的代码。折中的方法是用“错误”次序创建列表，然后当整个列表创建后，执行并反转它。

# 规则

yacc语法包括一组规则，每一规则以一个非终结符和一个冒号开始，并且跟着一个可能是空的符号、文字标记和动作的列表。按照惯例，规则以一个分号结束，尽管大多数 yacc 版本中分号是可选择的，例如:

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

表明日期是“月/日/年”组成（符号日、月、年必须是在语法中的其他地方定义过）。最初的符号和冒号称做规则的左端，规则的其余部分称为右端，右端可以为空。

语法中如果好几个连贯的规则具有相同的左端，第二个和随后的规则可以用一个竖线代替而不必再用名字和冒号开头。下面这两段的意义是相同的:

```
declaration:        EXTERNAL name ;
declaration:        ARRAY name '(' size ')' ;

declaration:        EXTERNAL name
        |       ARRAY name '(' size ')' ;
```

用竖线的形式比较好，竖线之前的分号必须省略。

动作就是当语法分析程序到达语法中动作出现的点时执行的C语言复合语句:

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

```
                        { printf("Date recognized.\n"); }
    ;
```

动作中C代码可以有以yacc中特殊处理的“$”开始的专门结构（详见本章“动作”一节）。除了在规则尾端的动作外，出现在别处任何地方的动作都被特殊处理。（详见本章“嵌入式动作”一节。）

规则尾端有明确的优先级：

```
expr: expr '*' expr
    | expr '-' expr
    | '-' expr %prec UMINUS ;
```

优先级只用于解决其他的有歧义的分析。细节参见本章“优先级、组合规则和操作符声明”一节。

# 特殊字符

由于yacc处理符号标记而不是文本，它的输入字符集比起lex的来就简单得多。下面列出了yacc所使用的特殊符号的列表：

% 具有两个%号标记的行将yacc语法分成了几部分（参见本章“yacc语法的结构”一节）。定义段的所有声明都以%开始，包括“%{%}”、“%**start**”、“%**token**”、“%**type**”、“%**left**”、“%**right**”、“%**nonassoc**”和“%**union**”。参见本章“文字块”、“开始声明”、“%type声明”、“优先级、结合性和操作符”和“%union声明”等节。

\ 反斜线符号是废弃的百分号的同义词。在动作中，在C语言字符串中有其通常的作用。

$ 在动作中，美元符号引入一个值引用，举例来说，**$3**表示规则右端第3个符号的值。参见本章“符号值”一节。

' 文字标记由一个单引号结束，例如‘**Z**’。参见本章“文字标记”一节。

"　　有些yacc版本在文字标记中将单引号与双引号同等对待，这样使用根本不方便。

<>　　在一个动作的值引用中，可以不考虑由尖括号包围起来的值的默认类型，例如**$<xtype>3**。**$<**和**$>**分别是“%**left**”和“%**right**”的同义词。参见本章“优先级、结合性和操作符声明”一节。

{}　　动作中C代码在中括号里（参见本章“动作”一节）。C代码在文字块声明部分由“%{”和“%}”括起来，参见本章“文字块”一节。

;　　除了后面紧接着是以竖线开头的规则外，规则部分的每一个规则都以分号结束。在许多yacc版本中，分号都是可选择的，这样做确实是个好主意。参见本章“规则”一节。

|　　当两个连续的规则具有相同的左端时，第二个规则可用一个竖线代替符号和冒号，参见本章“规则”一节。

:　　在每一条规则里，左端的每个符号后都跟一个冒号。参见本章“规则”一节。

_　　符号可以包括和字母、数字及句点在一起的下划线。

.　　符号可以包括与字母、数字和下划线一起的句点。由于C语言标识符不包括句点，所以这会引起问题。特别是不要使用名字里带有句点的标记，因为标记名字都被C语言预处理程序符号的“**#define**”所定义。

=　　yacc早期版本中在动作之前需要等号标记，并且大多数版本仍然接受它们，但现在不要求也不推荐使用了。见“动作”。

# 开始声明

通常，起始规则，即语法分析程序开始进行分析的规则，它是在第一个规则里命名的，如果想要以其他的规则开始，在声明部分可以这样写：

```
%start somename
```

这样就会以规则“*somename*”开始。

在大多数情况下，表现语法的最直接方式是从上到下，从第一个规则开始，所以不需要“**%start**”。

# 符号值

每一个在yacc语法分析程序中的符号，包括标记和非终结符，都有一个与它们相关联的值。如果标记是“**NUMBER**”，值可能就是特别的数字；如果是“**STRING**”，值可能就是指向字符串拷贝的一个指针；如果是“**SYMBOL**”，值可能就是指向符号表中描述这个符号的条目的一个指针。每一种值符合不同的C语言类型，“**int**”或“**double**”对应数字，“**char ***”对应字符串，指针对应于符号的结构。yacc使得给符号分配类型变得容易，所以能自动给符号使用正确的类型。

## 声明符号类型

yacc在内部将每个值作为包括所有类型的C联合类型来声明。在%**union**声明中列出所有的类型（参见本章“%union声明”一节），yacc将其转变为称为**YYSTYPE**的联合类型的类型定义。对于值已在动作代码中设置和使用的符号，必须声明它的类型。为非终结符使用“**%type**”；为标记使用“**%token**”、“**%left**”、“**%right**”或“**%nonassoc**”，对于联合字段给予与其类型相符的名字。

只要你使用“**$$**”、“**$1**”等引用值，yacc就会自动使用联合的适当字段。

## 计算器例子

这有一个虽然不现实但却简单的计算器，它能够添加数字和比较字符串，所有结果都是数字。

```
%union {
     double dval;
     char *sval;
}
. . .
```

```
%token <dval> REAL
%token <sval> STRING
%type <dval> expr
%%
calc: expr { printf("%g\n", $1); }

expr: expr '+' expr { $$ = $1 + $3; }
      | REAL { $$ = $1; }
      | STRING '=' STRING { $$ = strcmp($1, $3) ? 0.0: 1.0; }
```

有两个值类型: "**dval**"和"**sval**"。它们分别是双精度型和字符指针。标记"**REAL**"和非终结符 "**expr**" 自动使用联合成员 "**dval**", 而标记 "**STRING**" 使用联合成员 "**sval**"。

yacc并不理解C语言, 所以键入的错误符号, 例如使用联合类型中没有的类型名或以C语言不允许的方式使用字段, 都会在生成的C程序里引起错误。

## 明确的符号类型

通过在美元标志和符号数字之间或者在两个美元符号之间的尖括号中插入类型名, 例如 "**$<xxx>3**" 或 "**$<zzz>$**", yacc允许为符号值引用声明一个明确的类型。

这个特征很少用, 因为在几乎所有的事情中, 声明符号更容易、更可读。最似是而非的用法是在引用继承属性和当设置并引用由嵌入式动作返回的数值的时候, 详见本章的 "继承的属性" 和 "动作"。

# 标记

标记或终结符号都是词法分析程序传递给语法分析程序的符号。只要yacc语法分析程序需要另一个标记, 它就调用**yylex()**从输入中返回下一个标记。在输入的尾端, **yylex()**返回零。

标记可以是由 "**%token**" 定义, 也可以是在单引号里的单个字符 (参见本章 "文字标记" 一节)。作为标记使用的所有符号都必须在定义部分清楚地定义过, 例如:

```
%token UP DOWN LEFT RIGHT
```

标记可以由“**%left**”、“**%right**”或“**%nonassoc**”声明进行声明，它们每一个都与“**%token**”具有完全相同的语法选项。参见本章“优先级、结合性和操作符声明”一节。

## 标记号

在词法分析程序和语法分析程序中，标记是用小的整数标识的。文字标记的标记号是本地字符集中的数值，通常是ASCII码，与引用字符的C语言值相同。

符号标记通常是由yacc赋予值，赋予它们比任何可能的字符码更高的数字，所以与任何文字标记都不会产生冲突。也可以自己按照标记名字和“**%token**”中的数字赋予标记号：

```
%token UP 50 DOWN 60 LEFT 17 RIGHT 25
```

赋予两个标记相同的数字是很严重的错误，但是大多数yacc版本不会注意到刚刚生成了不正确的语法分析程序。大多数情况下让词法分析程序选择自己的标记号是既容易又可靠的。

词法分析程序需要知道标记号以便将适当的值返回到语法分析程序。对于文字标记，使用相应的C字符常数；对于符号标记，可以告诉yacc使用带有“*-d*”命令行的标志创建一个包括所有标记号定义的C语言头文件。如果词法分析程序中有头文件就可以使用这些符号，例如，UP、DOWN、LEFT和RIGHT。头文件通常称为“*y.tab.h.*”。在MS-DOS系统内MKS yacc称之为“*ytab.h*”，而pcyacc称之为“*yytab.h*”。bison、POSIX yacc和两个MS-DOS版本都有更改生成的头文件的名字的命令行选项。

## 标记值

yacc语法分析程序中每一个符号都有一个相关的值（参见本章“符号值”一节）。由于标记可以有值，所以当词法分析程序将标记返回给语法分析程序时需要设置

这些值。标记值通常存储在 **yylval** 变量中。在最简单的语法分析程序中，**yylval** 是一个单纯的 *int* 变量，在词法分析程序扫描程序中可以这样来设置：

```
[0-9]+         { yylval = atoi(yytext); return NUMBER; }
```

尽管大多数情况下，不同的符号有不同的值类型，参见本章“%union 声明”、“符号值”和“%type 声明”等节。

在语法分析程序中，一定要声明所有具有值的标记的值类型。将适当的联合标签的名字放入 %**token** 语句的尖括号内或优先级声明中，可以像如下这样定义值的类型：

```
%union {
        enum optype opval;
        double dval;
        char *sval;
}
...
%token <dval> REAL
%token <sval> STRING
%nonassoc <opval> RELOP
```

（这种情况下，**RELOP** 可能是像“= =”或“>”一样的关系操作符，标记值表明了是哪个操作符。）

返回标记时，设置适当的 **yylval** 字段。这种情况下，最好用 **lex**：

```
%{
#include "y.tab.h"
%}
...
[0-9]+\.[0-9]*      { yylval.dval = atof(yytext); return REAL; }
\"[^"]*\"     { yylval.sval = strdup(yytext); return STRING; }
"=="          { yylval.opval = OPEQUAL; return RELOP; }
```

**REAL**的值是双精度型，所以进入**yylval.dval**，而**STRING**的值是字符型(*char**)，所以进入 **yylval.sval**。

# %type 声明

用 %**type** 声明非终结符的类型。每个声明都有如下格式：

```
%type <type> name,name,...
```

"*type*" 名字必须由 "%**union**" 定义（参见本章 "%union 声明" 一节）。每个 *name* 都是一个非终结符号的名字。

使用 %**type** 声明非终结符。为了声明标记，也可使用 %**token**、%**left**、%**right** 和 %**nonassoc**。详见本章的 "标记" 和 "优先级、结合性和操作符声明"。

# %union 声明

%**union** 声明标识符号值的所有可能的C类型（参见本章 "符号值" 一节）。声明采用这种格式：

```
%union {
  ... 域声明 ...
}
```

域声明被逐字拷贝到输出文件中 **YYSTYPE** 类型的C联合类型声明中，yacc不检查 %**union** 的内容是否是有效的 C。

在没有 %**union** 声明时，yacc 将 **YYSTYPE** 定义为 **int**，所以所有的符号值都是整数。

使用 %**type** 声明将特殊符号与用 %**union** 声明的类型关联起来。

yacc将生成的C联合类型声明放入生成的C文件和选择生成的头文件（通常称为 *y.tab.h*）中，所以就可以在其他包括生成的头文件的源文件中使用 **YYSTYPE**。反之，也可将自己的**YYSTYPE**声明放入在定义部分用**#include**引用的包含文件中。在这种情况下，必须至少有一个%**type**提示yacc你正在使用明确的符号类型。

# 变体和多重语法

可能想要语法分析程序在同样的程序里对两个部分不同或完全不同的语法进行分析。例如一个交互式调试解释程序可能有一个用于程序设计语言的语法分析程序和一个用于调试程序命令的语法分析程序，一个单通道C编译器可能需要一个语法分析程序用于预处理程序句法而另一个用于C语言本身的语法分析程序。

在一个程序里有两种方式处理两个语法：将它们组合放入一个语法分析程序，或将两个完整的语法分析程序放入程序。

## 组合的语法分析程序

通过添加一个新的起始规则可以将几个语法组合成一个，这依赖于第一个被读取的标记。例如：

```
%token CSTART PPSTART
%%
combined:       CSTART cgrammar
              | PPSTART ppgrammar
        ;

cgrammar: . . .

ppgrammar: . . .
```

在这种情况下，如果第一个标记是**CSTART**，就分析起始规则是**cgrammar**的语法，如果第一个标记是**PPSTART**，就分析起始规则是**ppgrammar**的语法。

需要将代码放入返回合适的特殊标记的词法分析程序中，即语法分析程序第一次向词法分析程序请求一个标记：

```
%%
%{
        extern first_tok;

        if(first_tok) {
                int holdtok = first_tok;
```

```
                first_tok=0;
                return holdtok;
            }
%}
...<词法分析程序的其余部分>
```

在这种情况下，在调用 **yyparse()**之前，将 **first-tok** 设置为适当的标记。

这种方法的一个优点是程序比带有多个语序分析程序要小，这是因为只有一个分析代码的拷贝。另一个优点是如果分析相关的语法，例如C预处理程序表达式和C本身，可能就能共享语法的某些部分。缺点是如果一个语法分析程序正在执行时就不能访问另一个语法分析程序（参见本章后面的“递归分析”一节），并且除了故意共享的规则以外，需要在两个语法中使用不同的符号。

实际上，当想要分析存在微小不同的单个语言版本时这种方法是有用的。例如，一个经过编译的丰富的语言和一个在调试程序中解释的交互式子集。

## 多重语法分析程序

另一种方法是在一个单个的程序中包括两个完整的语法分析程序。yacc这样做不是很容易，因为生成的每一个语法分析程序都有同样的入口点**yyparse()**，并且调用同样的词法分析程序**yylex()**，使用同样的标记值变量**yylval**。而且，大多数yacc版本将分析表和分析程序堆叠放入带有 **yyact** 和 **yyv** 名字的全局变量中。如果你刚刚翻译了两个语法并且编译和链接着两个结果文件（至少将其中一个命名为不是*y.tab.c*的其他格式），就会得到一个包括多重定义符号的很长列表。技巧就是改变 yacc 用于函数和变量的名字。

### 使用 -p 标志

yacc的现代版本（包括bison、MKS yacc和任何POSIX兼容的实现）提供了一个命令行开关 *-p* 来改变yacc生成的语法分析程序中的名字前缀。例如下面这个命令：

```
yacc -p pdq mygram.y
```

生成了一个带有入口点**pdqparse()**的语法分析程序，调用词法分析程序**pdqlex()**等等。特别受到影响的名字是**yyparse()**、**yylex()**、**yyerror()**、**yylval**、**yychar**和**yydebug**。（变量**yychar**保存最近被读取的标记，当打印错误消息时有时有用。）其他用在语法分析程序中的变量可以重命名，或者将其设置为静态的或自动的；在任何事件中，它们必须保证相互不冲突。还有一个 *-b* 标志可以指定生成的C文件的前缀；例如：

```
yacc -p pdq -b pref mygram.y
```

假定标准的名字是*y.tab.c*，那么执行上述命令后会产生*pref.tab.c*。

必须提供**yyerror()**和**yylex()**的正确命名的版本。

## 伪造它

在yacc的老版本中，在生成的C例程中没有自动更改名字的方式，所以得伪造一个。在UNIX系统中，伪造的最简单的方法是使用字符流编辑程序*sed*。假设正在使用AT&T yacc，创建文件*yy-sed*包括这26个*sed*命令。（这种情况下，新的前缀是“pdq”。）

```
s/yyact/pdqact/g
s/yychar/pdqchar/g
s/yychk/pdqchk/g
s/yydebug/pdqdebug/g
s/yydef/pdqdef/g
s/yyerrflag/pdqerrflag/g
s/yyerror/pdqerror/g
s/yyexca/pdqexca/g
s/yylex/pdqlex/g
s/yylval/pdqlval/g
s/yynerrs/pdqnerrs/g
s/yypact/pdqpact/g
s/yyparse/pdqparse/g
s/yypgo/pdqpgo/g
s/yyps/pdqps/g
s/yypv/pdqpv/g
s/yyr1/pdqr1/g
s/yyr2/pdqr2/g
```

```
s/yyreds/pdqreds/g
s/yys/pdqs/g
s/yystate/pdqstate/g
s/yytmp/pdqtmp/g
s/yytoks/pdqtoks/g
s/yyv/pdqv/g
s/yyval/pdqval/g
```

运行 yacc 后，这些命令编辑生成的语法分析程序:

```
sed -f yy-sed y.tab.c > pdq.tab.c
sed -f yy-sed y.tab.h > pdq.tab.h
```

你可能要将这些规则放进一个 *Makefile* 文件里:

```
pdq.tab.h pdq.tab.c: yyexamp.y
    yacc -vd yyexamp.y
    sed -f yy-sed y.tab.c > pdq.tab.c
    sed -f yy-sed y.tab.h > pdq.tab.h
```

另一种方法是在语法的开始部分使用 C 预处理程序 **#define** 重命名变量:

```
%{
#define yyact pdqact
#define yychar pdqchar
#define yychk pdqchk
#define yydebug pdqdebug
#define yydef pdqdef
#define yyerrflag pdqerrflag
#define yyerror pdqerror
#define yyexca pdqexca
#define yylex pdqlex
#define yylval pdqlval
#define yynerrs pdqnerrs
#define yypact pdqpact
#define yyparse pdqparse
#define yypgo pdqpgo
#define yyps pdqps
#define yypv pdqpv
#define yyr1 pdqr1
#define yyr2 pdqr2
#define yyreds pdqreds
#define yys pdqs
```

```
#define yystate pdqstate
#define yytmp pdqtmp
#define yytoks pdqtoks
#define yyv pdqv
#define yyval pdqval
%}
```

这样就避免了使用 *sed*，但是缺点是定义不能出现在生成的头文件中，只能在生成的 C 文件中。为了处理这个问题，将所有的定义放进一个文件里，称为 *pdqdefs.h*，并且在语法分析程序中输入以下内容：

```
%{
#include "pdqdefs.h"
%}
```

在使用头文件的文件中，首先包括 *pdqdefs.h*，例如，在词法分析程序中：

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

## 递归分析

稍有不同的问题是递归分析，即第二次调用 **yyparse()** 时，前一次调用还在执行。有组合语法分析程序时这是一个问题。如果有一个组合的C语言和C语言预处理语法分析程序，要在 C 语言模式下调用 **yyparse()** 一次来分析整个程序，并且当看到一个 **#if** 时递归地调用它来分析预处理程序表达式。

可惜的是，yacc 大多数版本不能提供简单的方法来处理对语法分析程序的递归调用。如果真的需要递归调用，就得对生成的 C 文件做一些不平常的编辑。例如，在一个 AT&T yacc 语法分析程序中，需要使变量 **yyv**、**yypv**、**yys** 和 **yyps** 成为语法分析程序本地化的自动变量，并且在对 **yyparse()** 的递归调用循环内保存和恢复 **ystate**、**yytmp**、**yynerrs**、**yyerrflag** 和 **yychar** 的值。

给出 %**pure_parser** 声明时，bison 是支持递归分析的一个版本。这条声明使语法分析程序可再次进入和改变对 **yylex()** 的调用顺序，传递当前 **yylval** 和 **yylloc** 拷

贝的指针作为参数。(后面这条是可选的bison专有特征的一部分，即追踪每一个标记的正确的源位置，允许更精确的错误报告。)

## 多重语法分析程序的词法分析程序

如果与多重语法分析程序一起使用lex词法分析程序，需要调节词法分析程序以符合语法分析程序的改变(参见第六章“多重词法分析程序”一节)。通常要以组合的语法分析程序方式使用组合的语法分析程序，和以多重语法分析程序方式使用多重词法分析程序。

如果使用多重语法分析程序和词法分析程序，而且yacc和lex版本不提供自动重命名，那么可能就要将sed和包含文件结合，用重命名lex变量的技术来重命名yacc变量，因为这两个技术是相同的，并且有一些相同的名字，例如**yylex()**和**yylval**，它们在两个地方都需要改。

# y.output文件

yacc的每个版本都有创建日志文件的能力，在UNIX系统下命名为*y.output*，在MS-DOS系统下命名为*y.out*或*yy.lrt*。这些文件显示语法分析程序内的所有状态和一个状态到另一个状态的转变。使用-*v*标志生成日志文件。

日志文件的准确形式对于yacc的每个版本都不同，但是选自表达式语法的如下格式是较典型的:

```
state 1
	e : ID .  (2)

	. reduce 2

state 2
	e : '(' . e ')'  (3)

	ID shift 1
	'(' shift 2
	. error
```

```
        e goto 5
```

当到达那个状态时，每个状态中的圆点显示语法分析程序对规则分析的程度。例如当语法分析程序处于状态 2 时，如果语法分析程序看到 **ID** 地址，就将 **ID** 移进堆栈并切换到状态 1。如果看到一个开放的圆括号就将它移进堆栈并切换回状态 2，并且任何其他的标记都是错误的。在状态 1 下，总是归约编号为2的规则（规则在输入文件中按出现的先后顺序编号）。

归约之后 ID 在语法分析程序堆栈上被一个 **e** 取代，语法分析程序弹回到状态 2，在这点上，**e** 将它转到状态 5。

当有冲突时，有冲突的状态就显示了冲突移进和归约动作。

```
9: shift/reduce conflict (shift 7, reduce 4) on '+'
state 9
        e : e . '+' e (1)
        e : e '+' e . (4)

        '+' shift 7
        ';' reduce 4
        ')' reduce 4
```

在这种情况下，当 yacc 看到加号时，有移进/归约冲突。可以通过重写和添加加号的操作符声明来修正。参见本章“优先级、结合性和操作符声明”一节。

# yacc 库

每个实现都需要有用的例程库。在 UNIX 系统下可以通过在 cc 命令行尾端给出 *-ly* 标志（或通过其他系统下的等价物）来包含库。库的内容在不同的实现之间是不同的，但总是包括 **main()** 和 **yyerror()**。

## main()

yacc的所有的版本都带有最小的主程序，该程序对于简短程序和测试有时是很有用的。它非常简单，我们可以复制在下面：

```
main(ac, av)
{
      yyparse();
      return 0;
}
```

和具有任何库例程一样，你可以提供自己的**main()**。在几乎所有有用的应用软件中都可以提供**main()**用来接受命令行参数和标志、打开文件和检查错误。

## yyerror()

所有yacc版本也都提供简单的错误报告例程，它也是非常简单：

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

这有时能满足要求，但是一个好的错误报告例程至少能报告行号和最近的标记（如果词法分析程序是用lex写的，会在**yytext**中），这将会使语法分析程序更加有用。

# YYABORT

特殊语句

```
YYABORT;
```

在动作中使得语法分析例程**yyparse()**以一个非零值立即返回，显示失败。

当动作例程探测到错误非常严重以致于没有继续分析的点时是很有用的。

由于语法分析程序可以向前查看一个标记，所以在语法分析程序读到另一个标记前，包括**YYABORT**的规则动作可以不被归约。

# YYACCEPT

特殊语句

```
YYACCEPT;
```

在动作中使得语法分析例程 **yyparse()**以一个零值立即返回，显示成功。

在词法分析程序不能告知输入数据何时结束而语法分析程序能告知的情况下是很有用的。

由于语法分析程序可以向前查看一个标记，所以在语法分析程序读到另一个标记前，包括 **YYACCEPT** 的规则动作可以不被归约。

# YYBACKUP

yacc的一些版本，包括最原始的AT&T yacc，有一个很少提及的宏**YYBACKUP**，该宏移出（unshift）当前标记并用其他东西来代替。语法是：

```
sym:    TOKEN  {YYBACKUP(newtok, newval); }
```

它放弃本应通过归约已经被替代的符号sym，假装词法分析程序刚刚读过带有值**newval**的标记**newtok**。如果需要向前查看标记或在右端有超过一个符号的规则，规则对 **yyerror()**的调用失败。

正确使用**YYBACKUP()**非常难，而且根本不方便，所以建议不要使用它。（在这里介绍它是以防你在使用它的现有的语法中碰到它。）

# yyclearin

如果预先查看的标记被读取，那么动作中的宏**yyclearin**就将其丢弃，在一个错误之后将语法分析程序放入已知状态，这在交互式语法分析程序中的错误恢复中是常用的：

```
stmtlist: stmt | stmt_list stmt ;

stmt: error { reset_input(); yyclearin; } ;
```

出现错误之后，这会调用用户例程 **reset_input()**(假设该用户例程将输入放入已知状态)，然后使用 **yyclearin** 准备重新开始读取标记。

更多的内容参见本章“yyerrok”和“YYRECOVERING()”部分。

# yydebug 和 YYDEBUG

大多数yacc版本能选择性地编译跟踪代码来报告语法分析程序所做的每件事，这些报告非常冗长，但却是断定语法分析程序问题原因的惟一方式。

## YYDEBUG

由于跟踪代码又大又慢，所以不能自动编译到目标程序中。为了包含跟踪代码，使用 yacc 命令行上的 *-t* 标志；或者要么在C编译程序命令行上，要么通过在定义段包含类似下面的句式，来定义 C 预处理程序符号 **YYDEBUG** 为非零：

```
%{
#define YYDEBUG 1
%}
```

## yydebug

整型变量**yydebug**在运行的语法分析程序中控制语法分析程序是否真正产生调试输出，如果值不为零，语法分析程序产生调试报告；而如果为零，就不产生调试输出。可以在任何想要的方式下设置 **yydebug** 为非零，例如，以程序命令行上的标志的形式设置，或使用调试程序在运行时修补它。

# yyerrok

yacc 探测到一个语法错误后，通常避免报告另一个错误，直到移进了 3 个连贯的标记而没有另外的错误。这稍微减轻了当语法分析程序获得再次同步时由单个失误导致的多重错误消息的问题。

如果知道何时语法分析程序退回到同步，就返回到报告所有错误的正常状态。宏 **yyerrok** 告知语法分析程序返回到正常状态。

例如，假设有一个所有命令都在单独的行上的命令解释程序，无论用户修补命令的程度多么不好，你都会知道下一行是一个新的命令：

```
cmdlist: cmd | cmdlist cmd ;

cmd: error '\n' { yyerrok; } ;
```

带有**error**的规则越过一个错误后的输入到达一个新行，**yyerrok**告知语法分析程序错误恢复已经完成。

参见本章“YYRECOVERING()”和“yyclearin”部分。

# YYERROR

有时动作代码能探测到上下文相关的而语法分析程序本身却不能检测到的语法错误。如果代码探测到一个语法错误，就可以调用宏**YYERROR**，生成的效果与语法分析程序读到一个被语法禁止的标记时完全相同。一调用**YYERROR**，语法分析程序就访问**yyerror()**，进入错误恢复模式，寻找可以移进 **error** 标记的状态。参见本章“错误标记和错误恢复”一节。

# yyerror()

只要 yacc 语法分析程序探测到语法错误，就调用 **yyerror()**将错误报告给用户，传递单个参数，即描述这个错误的字符串（通常你得到的就是“语法错误”）。yacc

库中 **yyerror** 的默认版本仅仅是在标准输出文件中打印参数。下面是一个返回更多信息的版本：

```
yyerror(const char *msg)
{
     printf("%d: %s at '%s'\n", yylineno, msg, yytext);
}
```

我们假定 **yylineno** 是当前的行号（参见第六章“行号和 yylineno”），并且 **yytext** 是包含当前标记的 lex 标记缓冲器，由于不同的 lex 版本声明 **yytext** 的方式不同，有些版本声明为数组，而有些版本则声明为指针，为了有最好的可移植性，最好是将这个程序放入词法分析程序文件的用户子例程段，因为这是惟一的能自动定义 **yytext** 的地方。

由于 yacc 顽固地试图恢复错误和分析全部输入，无论混淆的多么厉害，都可以让 **yyerror()** 计算调用的次数，10 次错误以后就可以退出——理论上讲已经报告的错误可能乱得让语法分析程序绝望了。

当动作程序探测到其他种类的错误时，也可以并且应该自己调用 **yyerror()**。

# yyparse()

yacc 生成的语法分析程序的入口点是 **yyparse()**。当程序调用 **yyparse()** 时，语法分析程序试图分析输入流。如果分析成功，语法分析程序就返回一个零值；反之，则返回一个非零值。

每次调用 **yyparse()** 时，语法分析程序就重新开始进行分析、而忘记上次返回的状态。这与 lex 生成的扫描程序十分不同，它能够获得调用过的每一次状态。

参见本章的“YYACCEPT”和“YYABORT”部分。

# YYRECOVERING()

yacc探测到语法错误时，通常进入恢复模式，即直到移进3个连贯的标记而没有另外的错误之前禁止报告另外的错误，这稍微能减轻语法分析程序再次获得同步时由单个失误造成的多重错误消息的问题。

如果语法分析程序当前处于错误恢复状态，宏**YYRECOVERING()**返回一个非零值；反之，则返回一个零值。有时用测试**YYRECOVERING()**来决定是否报告动作程序中发现的错误是方便的。

参见本章“yyclearin”和“yyerrok”部分。

# 第八章

# yacc歧义和冲突

本章内容：

- 指针模型和冲突
- 冲突的普通示例
- 如何修复冲突
- 小结
- 练习

本章重点是寻找和纠正yacc语法中的冲突。当yacc报告移进/归约和归约/归约错误时就出现冲突。找到它们是很具挑战性的，因为yacc是在*y.output*（本章将描述*y.output*）（注1）中而不是在yacc语法文件中指向它们。阅读本章之前，应该理解yacc语法分析程序工作的一般方法，这些方法在第三章“使用yacc”中进行了描述。

## 指针模型和冲突

为了根据yacc语法描述冲突是什么，引进了yacc的操作模型。在这个模型中，读取每个独立的标记时指针移过yacc语法。开始时，在起始规则的开头有一个指针（这里以向上箭头↑表示）：

```
%token A B C
%%
start: ↑ A B C;
```

注1：yacc的MS-DOS版本调用列表文件*y.out*或*yy.lrt*，并且它们中的信息格式是不同的。yacc的所有版本使用相同的分析策略并得到相同的冲突，所以列表文件包含相同的信息。

当yacc语法分析程序读取标记时，指针移动。假定它读取A和B：

```
%token A B C
%%
start: A B ↑ C;
```

有时，由于yacc语法中的其他选择而存在更多的指针。例如，假设使用下面的语法读取A和B：

```
%token A B C D E F
%%
start:      x
       |    y;
x:     A B ↑ C D;
y:     A B ↑ E F;
```

（对于本章例子的其他部分，省略%token和%%。）使指针不出现的方法有两种。一种是一个标记消除一个或多个指针，因为只有一个指针仍然匹配输入。如果yacc读取的下一个标记是C，那么第二个指针将消失，并且第一个指针往后移：

```
start:      x
       |    y;
x:     A B C ↑ D;
y:     A B E F;
```

指针消失的另一种方法是将它们并入一个普通的子规则中。在下面的例子中，z出现在x和y中：

```
start:      x
       |    y;
x:     A B z R;
y:     A B z S;
z:C D
```

读取A之后，有两个指针：

```
start:      x
       |    y;
x:     A ↑ B z R;
```

```
y:      A ↑ B z S;
z:      C D
```

读取 **ABC** 后，只有一个指针：

```
start:       x
      |      y;
x:      A B z R;
y:      A B z S;
z:      C ↑ D;
```

读取 **ABCD** 后，会再次出现两个指针：

```
start:       x
             y;
x:      A B z ↑ R;
y:      A B z ↑ S;
z:      C D;
```

当指针到达规则的尾端时，规则被归约。yacc 读取 **D** 之后，当指针到达规则 **z** 的尾端时规则 **z** 被归约。然后这个指针返回到调用被归约的规则的规则处，或者在上面的情况下，这个指针分裂到多个规则，这些规则调用了被归约的规则。

当有多个指针时，如果规则被归约就会有一个冲突。下面是只有一个指针的归约例子：

```
start:      x
            y;
x:      A ↑ ;
y:      B ;
```

在 **A** 之后，只有一个指针 —— 在规则 x 中，并且规则 x 被归约。同样，在 **B** 之后，只有一个指针 —— 在规则 y 中，并且规则 y 被归约。

下面是一个冲突示例：

```
start:      x
      |     y;
x:      A ↑ ;
y:      A ↑ ;
```

在**A**之后，有两个指针，在规则**x**和**y**的尾端，它们都想归约，所以是一个归约/归约冲突。

如果只有一个指针就不会有冲突，即使它是将多个指针并入一个普通的子规则的结果，以及即使归约将导致多个指针。

```
start:      x
            y;
x:      z R ;
y:      z S ;
z:      A B ↑ ;
```

在**AB**之后，有一个指针，在规则**z**的尾端，并且那条规则被归约，导致两个指针：

```
start:      x
            y;
x:      z ↑ R;
y:      z ↑ S;
z:      A B;
```

但是在归约的时候，只有一个指针，所以它不是冲突。

## 冲突的类型

有两种冲突：归约/归约冲突和移进/归约冲突。当一个指针在归约时，冲突根据另一个指针发生的情况被分类。如果另一条规则也正在归约，它就是归约/归约冲突。下面的例子表明在规则**x**和**y**中有一个归约/归约冲突：

```
start:      x
        |   y;
x:      A ↑ ;
y:      A ↑ ;
```

如果另一个指针没在归约，那么它就在移进，并且这个冲突是移进/归约冲突。下面的示例表明在**x**和**y**中有一个移进/归约冲突：

```
start:    x
     |    y R;
x:   A ↑ R;
y:   A ↑ ;
```

在yacc读取A之后，规则y需要归约到规则**start**，然后在那里**R**可以被接受，而规则**x**能立即接受**R**。

如果在归约时有两个以上的指针，yacc就成对地列出冲突。下面的示例表明在规则**x**和**y**中有一个归约/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 ↑ ;
```

让我们定义清楚在涉及到向前查看标记和指针消失的情况下何时会归约。这样可以保持正确的简单冲突定义。下面是一个归约/归约冲突：

```
start:    x B
     |    y B;
x:   A ↑ ;
y:   A ↑ ;
```

但是这里没有冲突：

```
start:    x B
     |    y C;
x:   A ↑ ;
y:   A ↑ ;
```

第二个示例没有冲突的原因是yacc能向前查看A以外的另一个标记。如果它看到了**B**，那么规则**y**中的指针在规则**x**被归约之前消失。同样，如果它看到了**C**，那么规则**x**中的指针在规则**y**归约之前消失。

yacc只能向前查看一个标记。下面的例子对能向前查看两个标记的编译程序不会有冲突，但是在 yacc 中，它是归约/归约冲突：

```
start:     x B C
      |    y B D;
x:    A ↑ ;
y:    A ↑ ;
```

## 分析状态

yacc 不是表明冲突在 yacc 语法中的位置，而是表明它们在 *y.output* 中的位置。*y.output* 是它生成的状态机的描述。我们要讨论状态是什么、描述 *y.output* 的内容，然后讨论如何寻找yacc语法中的问题，这些问题是*y.output*中描述的冲突所导致的。可以通过用 -*v*（冗长的）选项运行 yacc 来生成 *y.output*。

每个状态对应 yacc 语法中所有可能的指针组合中惟一一种。每个非空的 yacc 语法至少都有两个互不相同的可能状态：当没有接受输入时位于开始状态，当接受完整有效的输入时在结束状态。下面的简单示例中有两个以上的状态：

```
start:       A <one here> B <another here> C;
```

在将来的例子中，为了能标识清楚，我们给状态编号。虽然yacc可以给这些状态编号，但是编号的顺序不是很重要：

```
start:       A <state 1> B <state 2> C;
```

当给定的输入标记流对应多个可能的指针位置时，那么给定标记流的所有指针对应一个状态：

```
start:       a
      |      b;
a:    X <state 1> Y <state 2> Z;
b:    X <state 1> Y <state 2> Q;
```

当不同的输入流对应同一指针时，它们能对应同一状态：

```
start:      threeAs;
threeAs: /* 空值 */
         threeAs A <state 1> A <state2>
                   A <state 3>;
```

上面的语法接受某些3的倍数个A。状态1对应1，4，7，...个A；状态2对应2，5，8，...个A；状态3对应3，6，9，...个A。尽管不是很好的设计，但是为了解释下一点我们可以用右递归重写上面的语句：

```
start:      threeAs;
threeAs: /* 空值 */
       | A A A threeAs;
```

（如果使用右递归下面的示例就会有冲突。）规则中的位置不必只对应一种状态。一条规则中指定的指针对应另一条规则中不同的指针，生成几个状态：

```
start:      threeAs X
            twoAs Y;
threeAs: /* 空值 */
       | A A A threeAs;
twoAs: /* 空值 */
     | A A twoAs;
```

上面的语法接受2的倍数个**A**或3的倍数个A，3的倍数后面有**X**，或者2的倍数后面跟有**Y**（没有**X**或**Y**，这个语法就会有冲突，比如不知6的倍数个**A**是应该满足**threeAs**（三个**A**）还是满足**twoAs**（两个**A**））。如果按以下方式编号状态：

```
state 1: 1, 7, ... A's accepted
state 2: 2, 8, ... A's accepted
...
state 6: 6, 12, ... A's accepted
```

那么对应的指针位置如下：

```
start:      threeAs X
     |      twoAs Y;
threeAs: /* 空值 */
       | A <1,4> A <2,5> A <3,6> threeAs;
twoAs: /* 空值 */
     | A <1,3,5> A <2,4,6> twoAs;
```

也就是，在**threeAs**中的第一个**A**之后，yacc能接受6i+1或6i+4个**A**，这里i为0，1，等等。同样，在**twoAs**中的第一个**A**之后，yacc可以接受6i+1、6i+3或者6i+5个**A**。

## y.output的内容

目前，我们已经定义了状态，接下来查看*y.output*中描述的冲突。文件的格式根据yacc版本的不同而不同，但是它总是包括所有分析状态的清单。对于每一种状态，它列出了对应状态的规则和位置，当语法分析程序读取那个状态中的不同标记时语法分析程序将执行的移进和归约，以及在归约处于那个状态产生非终结符之后它将切换到的状态。下面的清单来自yacc的不同版本，所以可以看出差别很小。我们要展示一些有歧义的语法和识别这些有歧义的语法的*y.output*报告。

### 归约/归约冲突

考虑下面的有歧义的语法：

```
start:      a Y
      |     b Y ;
a:    X ;
b:    X ;
```

通过Berkeley yacc运行它时，典型的状态描述如下：

```
state 3
      start : a . Y (1)

      Y shift 5
      . error
```

在这个状态中，语法分析程序已经归约了一个**a**。如果它看到**Y**，那么它就移进**Y**并移至状态5。别的东西（用点表示）都是错误。歧义性在状态1中产生归约/归约冲突：

```
1: reduce/reduce conflict (reduce 3, reduce 4) on Y
state 1
```

```
        a : X .  (3)
        b : Y .  (4)

        . reduce 3
```

第一行表示在读取Y标记时状态1在规则3和规则4之间有一个归约/归约冲突。在这个状态中，它读入了一个也许是a也许是b的X。第三行和第四行显示可以被归约的两条规则。点（注2）表示在接受下一个标记时你所在的规则中的位置。这个位置对应yacc语法中的指针。对归约冲突，指针总是位于规则的结尾。最后一行显示yacc选择归约规则3，因为它通过归约先出现在语法中的规则来解决归约/归约冲突。

这些规则里也许有标记或规则名。下面的有歧义的语法：

```
start: a Z
       b Z;
a: X y;
b: X y;
y: Y;
```

产生具有如下状态的语法分析程序：

```
6: reduce/reduce conflict (reduce 3, reduce 4) on Z
state 6
        a : X y .  (3)
        b : X y .  (4)

        . reduce 3
```

在这个状态中，语法分析程序已经将一个Y归约为一个y，但是y能完成一个a或一个b。非终结符就像标记一样能导致归约/归约冲突。如果使用大写标记名，就很容易说明它们的不同，正如我们上面所采用的一样。

冲突的规则不必是同样的。语法如下：

注2：yacc用点表示你在规则中的位置，如果有使用点的规则就会产生混淆。yacc的一些版本使用下划线而不是点，如果有使用下划线的规则同样会产生混淆。

```
start:      A B x Z
      |     y Z;
x:    C;
y:    A B C;
```

当用 AT&T yacc 处理时产生包含如下状态的语法:

```
7: reduce/reduce conflict (red'ns 3 and 4 ) on Z
state 7
        x : C_    (3)
        y : A B C_    (4)

        . reduce 3
```

在状态 7 中，yacc 已经接受了 **ABC**。规则 **x** 中只有 **C**，因为在调用 **x** 的 **start** 规则中，**AB** 在调用 **x** 之前被接受。**C** 能完成一个 **x** 或一个 **y**。通过归约语法中前面的规则，yacc 可以再次解决这个冲突。

## 移进 / 归约冲突

标识移进 / 归约冲突有点难。为了标识该冲突，按以下步骤操作:

- 在 *y.output* 中找到移进 / 归约错误。
- 挑选归约规则。
- 挑选相关移进规则。
- 注意归约规则归约到的位置。
- 归约产生冲突的标记流。

下面的语法包含移进 / 归约冲突:

```
start:      x
      |     y R;
x:    A R;
y:    A;
```

AT&T yacc 产生如下抱怨:

```
4: shift/reduce conflict (shift 6, red'n 4) on R
state 4
      x : A_R
      y : A_    (4)

      R shift 6
      . error
```

在状态4读取**R**时，它在移进标记**R**（并移至状态6）和归约规则4之间有一个移进/归约冲突。规则4是规则**y**，如下所示：

```
y : A_        (4)
```

在移进/归约冲突中可以找到归约规则，同样在归约/归约冲突中可以找到两条规则。归约号在右边的圆括号中。在上面的情况下，具有移进冲突的规则是状态中惟一保留的规则。

```
x:        A R;
```

yacc在规则**x**中，已经接受了**A**并且将要接受**R**。移进冲突规则在这种情况下容易寻找，因为它是惟一保留的规则，并且它显示下一个标记是**R**。yacc解决移进/归约冲突时倾向于移进，所以在这种情况下，如果它接收**R**，那么它移进到状态6。

下面展示的是规则而不是标记：

```
start:    x1
     |    x2
     |    y R;
x1:   A R;
x2:   A z;
y:    A;
z:    R;
```

Berkeley yacc报告几个冲突，包括：

```
1: shift/reduce conflict (shift 6, reduce 6) on R
state 1
      x1 : A . R  (4)
```

```
x2 : A . z  (5)
y : A .  (6)

R shift 6

  goto 7
```

在上面的例子中，归约规则如下：

```
y : A .  (6)
```

所以移进冲突有两个选择：

```
x1 : A . R  (4)
x2 : A . z  (5)
```

规则 **x1** 使用下一个标记 R，所以你知道它是移进冲突的一部分，但是规则 **x2** 显示下一个非终结符（不是标记）。你必须寻找 **z** 规则以查明它是否从 **R** 开始。在这种情况下它是这样的，所以后面跟有 **R** 的 **A** 存在三种冲突：它可以是 **x1**，或者是包含一个 **z** 的 **x2**，或者是后面跟有 **R** 的 **y**。

在冲突状态下有多条规则，并且它们都不接受 **R**，考虑这个语法的扩展版本：

```
start:     x1
           x2
           x3
     |     y R;
x1:    A R;
x2:    A z1;
x3:    A z2;
y:     A;
z1:    R;
z2:    S;
```

MKS yacc 产生具有下面状态的清单：

```
State 1

         x1:       A.R
         x2:       A.z1
         x3:       A.z2
   (8)   y:        A.   [ R ]
```

```
Shift/reduce conflict ( 9,8) on R
    R      shift 10
    S      shift 7
    .      error

    z2       goto 8
    z1       goto 9
```

（括号中的 **R** 意味着在语法中，这个上下文中的 **y** 必须后面跟有 **R**。）这个冲突位于移进到状态10和归约规则8之间。归约问题（规则8）是 **y** 的规则。**x1** 的规则是移进问题，因为它表明点后的下一个标记是 **R**。它不能立即显示在 **x2** 或 **x3** 周围，因为它们表明 **z1** 和 **z2** 跟在点后面。当查看规则 **z1** 和 **z2** 时，发现 **z1** 包含邻近的 **R** 并且 **z2** 包含邻近的 **S**，所以使用 **z1** 的 **x2** 是移进冲突的一部分，而 **x3** 不是。

在最后两个移进/归约冲突示例的每一个中，还能看到归约/归约冲突吗？运行 yacc 并且搜索 *y.output* 来核对你的答案。

## y.output 中冲突的概述

下面概述指针模型、冲突和 *y.output* 之间的关系。首先，是归约/归约冲突：

```
start:      A B x Z
     |      y Z;
x:     C;
y:     A B C;
```

AT&T yacc 清单包含：

```
7: reduce/reduce conflict (red'ns 3 and 4 ) on Z
state 7
x :    C_   (3)
y :    A B C_   (4)
```

因为如果下一个标记是 **Z**，存在一个冲突，那么 yacc 想要归约规则3和4，这些规则应用于 **x** 和 **y**。或者使用指针模型，有两个指针并且都在归约：

```
start:       A B x Z
       |     y Z;
x:     C ↑ ;
```

```
y:    A B C ↑ ;
```

下面是移进/归约示例:

```
start:     x
     |     y R;
x:    A R;
y:    A;
```

Berkeley yacc报告这个冲突:

```
1: shift/reduce conflict (shift 5, reduce 4) on R
state 1
      x : A . R  (3)
      y : A .  (4)

      R shift 5
```

因为如果下一个标记是R，存在一个冲突，那么yacc想要归约y的规则并且移进x的规则中的R。或者有两个指针并且一个在归约:

```
start:     x
     |     y R;
x:    A ↑ R;
y:    A ↑ ;
```

# 冲突的普通示例

产生移进/归约冲突的3种最普通的情况是表达式语法、IF-THEN-ELSE和嵌套的项目列表。明白如何识别这3种情况以后，就可以寻找摆脱这些冲突的方法。

## 表达式语法

第一个示例来自最初的UNIX yacc手册。为了使示例完整我们添加了一个终结符:

```
expr: TERMINAL
    | expr '-' expr ;
```

有冲突的状态如下:

```
4: shift/reduce conflict (shift 3, red'n 2) on -
      state 4
      expr : expr_- expr
      expr : expr - expr_ (2)
```

当得到减号标记时，yacc 告诉我们有一个移进/归约冲突。添加指针:

```
expr: expr ↑ - expr ;
expr: expr - expr ↑ ;
```

它们是同一条规则，甚至不是相同的名字下的两条可选规则。这表明该状态下同一规则中两指针的位置不同。这是因为语法是递归的。(事实上，本节所有的示例都是递归的。我们已经发现多数棘手的 yacc 问题都是递归的。)

在接受两个 **expr** 和“-”之后，指针位于规则 **expr** 的尾部，如上面的指针示例的第二行所示。但是“expr -expr”也是一个 **expr**，所以指针也可以恰好位于第一个 **expr** 后，如上面的示例的第一行所示。如果下一个标记不是“-”，那么第一行中的指针消失，因为它想让“-”作为下一个标记，所以返回一个指针。但是如果下一个标记是“-”，那么第二行想要归约，而第一行想要移进。

为了解决这个冲突，查看 *y.output*（如上所示）找到冲突的根源。摆脱状态中的不相关规则(这里没有任何不相关规则)，并且得到刚刚讨论的那两个指针。显然问题如下:

```
expr - expr - expr
```

中间的 **expr** 可能是“expr-expr”中的第二个 **expr**，在这种情况下，输入被解释为:

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

它是左结合的。或者也可能是第一个 **expr**，在这种情况下输入被解释为:

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

它是右结合的。读取“expr - expr”之后，如果采用左结合，语法分析程序进行归约，或者分析程序采用右结合进行移进。如果不指示选择一个或另一个，这种歧义性就导致移进/归约冲突，yacc 通过选择移进来解决这种情况。图 8-1 展示了两种可能的分析。

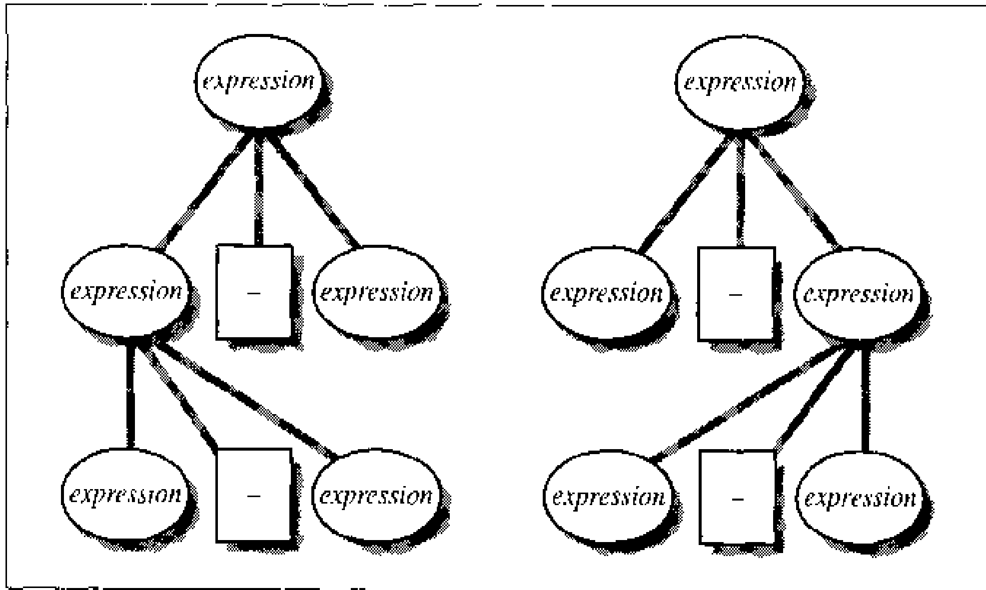



图 8-1　有歧义的输入 expr - expr - expr

在本章后面，我们要讨论这种冲突会导致什么情况。

## IF-THEN-ELSE

下一个例子也出自 UNIX yacc 手册。同样为了完整性添加一个终结符：

```
stmt: IF '(' cond ')' stmt
    | IF '(' cond ')' stmt ELSE stmt
    | TERMINAL;
cond: TERMINAL;
```

AT&T yacc 抱怨：

```
8: shift/reduce conflict (shift 9, red'n 1) on ELSE
state 8
      stmt :  IF ( cond ) stmt_       (1)
      stmt :  IF ( cond ) stmt_ELSE stmt
```

根据指针，应该为：

```
stmt: IF ( cond ) stmt ↑ ;
stmt: IF ( cond ) stmt ↑ ELSE stmt ;
```

第一行是冲突的归约部分，第二行是移进部分。这次它们是具有相同左侧的不同规则。为了断定将要发生的错误，领会第一行要归约到的位置。它必须是对**stmt**的调用，后面跟有**ELSE**。只有一个它能发生的地方：

```
stmt: IF ( cond ) stmt <return to here> ELSE stmt ;
```

在归约之后，指针返回到进行冲突移进部分的精确地点。事实上，它与上一个例子中“expr-expr-expr”发生的情况相同。并且采用类似的逻辑，将“IF (cond) stmt”归约成“stmt”并在这结束：

```
stmt: IF ( cond ) stmt <here> ELSE stmt ;
```

必须有这个标记流：

```
IF ( cond ) IF ( cond ) stmt ELSE
```

同样，像下面这样对它进行归组：

```
IF ( cond ) { IF ( cond ) stmt } ELSE stmt
```

或者像下面这样：

```
IF ( cond ) { IF ( cond ) stmt ELSE stmt }
```

下一节解释这种冲突所发生的事情。

## 嵌套的列表语法

最后的示例是一个我们数次帮人跟踪的问题的简单版。新的yacc程序员经常碰到它：

```
start:          outerList Z ;
outerList: /* 空值 */
      |    outerList outerListItem ;

outerListItem:   innerList ;

innerList:  /* 空值 */
      |    innerList innerListItem ;

innerListItem:   I ;
```

AT&T yacc 报告这种冲突:

```
2: shift/reduce conflict (shift 3, red'n 5) on Z
state 2
      start :  outerList_Z
      outerList :  outerList_outerListItem
      innerList : _   (5)
```

我们一步一步地检查，归约规则是**innerList**的空的替换物。这样可以为移进问题提供两种选择。**start**规则是其中之一，因为它显式地将Z作为下一个标记。如果它采用下一个Z，**outerList**的非空替换物也许是一种选择。我们看到**outerList**包括**outerListItem**，它是一个**innerList**。这个**innerList**不能包括**innerListItem**，因为它包含I，并且这种冲突只有在下一个标记为Z时才发生。但是**innerList**可以为空，所以**outerListItem**不包括标记，因为当冲突报告的第一行告诉我们**outerList**后面可以跟有Z时，实际上我们同样可能在**outerList**的结尾。

所有的都可以归结到这种状态：我们已经结束**innerList**(可能为空)或者**outerList**(可能为空)。它怎么不知道刚刚结束的列表呢？查看两个列表表达式。它们都为空，并且内部的表达式位于外部表达式之内，没有任何标记说明正在开始或结束内部循环。假设输入流只由一个Z组成。它是空的**outerList**(或者它也许是有一个项目的**outerList**)或一个空的**innerList**吗？那是有歧义的。

这个语法是冗余的。循环的内部有一个循环，没有任何东西隔离它们。因为这个语法实际上接受后面跟有一个Z的I的列表(可能为空)，所以可以使用惟一的递归规则编写它：

```
start:            outerList Z ;
outerList: /* 空值 */
       |     outerList outerListItem ;
outerListItem:    Z ;
```

或者可能忘记了 **outerListItem** 中区分外部循环和内部循环的一些标记。

# 如何修复冲突

一旦你已经理解了冲突是什么，那么本章的其余部分开始描述如何处理冲突。讨论如何修复为yacc用户带来麻烦的各种冲突。我们欢迎读者反馈自己所遇到的特殊问题，并在未来的版本中将它们添加在本节。第二版中有两个示例就是由读者Minneapolis 提供的。

当尝试解决冲突时，要考虑改变你正在分析的语言。有时你所使用的语言已经被定义，但是如果没有，通常采用对语言进行较小的调整来简化大量的 yacc 描述。事实上，语言中关键字的位置对yacc描述有重要影响 —— 它影响着描述是实用的、不实用的，甚至是不可能的。yacc分析有困难的语言通过人的头脑分析也是很困难的；一旦改变语言来删除这些冲突，就会得到更好的语言设计。

## IF-THEN-ELSE（移进/归约）

这是本章前面出现的例子之一，你曾经遇到过它，现在我们来描述如何处理移进/归约冲突。yacc解析这个特殊冲突的默认方式通常是你想让它去做的事情。如何知道它正在做你想让它做的事情呢？你的选择是：(1)仔细阅读 yacc 描述，(2)努力解码 *y.output* 清单，(3)测试生成的代码致死的原因。一旦确认得到了想要得到的东西，就应该使yacc停止抱怨。冲突警告可以混淆或打扰尝试维护代码的任何人，并且使你很容易就失去一条重要的警告。

可以按以下方式重写这个语法以避免这样的冲突：

```
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: /* 用于其他类型语句的规则 */ ...
```

非终结符**other_stmt**表示语言中所有其他可能的语句。虽然它可以工作，但它增加了语法的复杂性。

可以设置明确的优先级来阻止 yacc 发布警告。

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%%

stmt:       IF expr stmt            %prec LOWER_THAN_ELSE ;
          | IF expr stmt ELSE stmt;
```

如果语言使用 THEN 关键字（类似 Pascal 所用的），你可以这么做：

```
%nonassoc THEN
%nonassoc ELSE

%%

stmt:       IF expr THEN stmt
        |   IF expr stmt ELSE stmt
        ;
```

移进/归约冲突是移进标记（上面例子中的ELSE）和归约规则（**stmt**）之间的冲突。需要给标记（在例子中是 **%nonassoc ELSE**）和规则（**%nonassoc THEN** 或 **%nonassoc LOWER_THAN_ELSE** 和 **%prec LOWER_THAN_ELSE**）分配优先级。移进标记的优先级必须高于归约规则的优先级，所以 **%nonassoc ELSE** 必须出现在 **%nonassoc THEN** 或 **%nonassoc LOWER_THAN_ELSE** 之后。如果采用 **%nonassoc**、**%left** 或 **%right**，这个应用也一样。

这里的目的是隐藏你知道并理解的冲突，而不隐藏任何其他的冲突。当尝试屏蔽有关其他的移进/归约冲突的yacc警告时，从上面的例子中得到的东西越多，你应该越小心。其他的移进/归约冲突可以经得起yacc描述中的简单改变的考验。另外，正如上面所提到的，任何冲突都可以通过改变语言来修复。例如，通过坚持使用**stmt**周围的BEGIN-END或大括号来消除IF-THEN-ELSE冲突。

如果交换要移进的标记和要归约的规则的优先级会发生什么呢？正常的IF-ELSE处理视下面两个为等价：

```
if expr if expr stmt else stmt
if expr { if expr stmt else stmt }
```

交换优先级使下面两个等价似乎最合理，对吗？

```
if expr if expr stmt else stmt
if expr { if expr stmt } else stmt
```

错误！那不是它所做的事情。移进（正常的IF-ELSE）有较高的优先级使它总是移进ELSE。交换优先级使它永远不会移进ELSE，所以IF-ELSE不会再有else部分。

正常的IF-ELSE处理使ELSE与最近的IF联接在一起。假设想让它有一些其他方式。一种可能性是IF序列只允许一个ELSE，并且这个ELSE与第一个IF有关。这就要求有两个级别的语句定义，如下所示：

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%%

stmt:       IF expr stmt2 %prec LOWER_THAN_ELSE
      |     IF expr stmt2 ELSE stmt;

stmt2: IF expr stmt2;
```

我们不鼓励这样做，这样的语言是违反直觉的。

## 循环中的循环（移进/归约）

```
start:          outerList Z ;
outerList:  /* 空值 */
          | outerList outerListItem ;

outerListItem:   innerList ;
innerList:   /* 空值 */
          |  innerList innerListItem ;

innerListItem: I ;
```

解决办法取决于是想将整个循环看做一个外部循环和许多内部循环对待，还是看做许多外部循环各有一个内部循环对待。区别是每次重复时执行一次与 **outerListItem** 相关的代码，还是每组重复执行一次。如果都一样，任意选择一个。如果想要许多外部循环，删除那个内部循环：

```
start:          outerList Z ;

outerList:  /* 空值 */
         |  outerList innerListItem ;

innerListItem:     I ;
```

如果想要许多内部循环，就删除那个外部循环：

```
start:          innerList Z ;

innerList:  /* 空值 */
         |  innerList innerListItem ;

innerListItem: I ;
```

## 表达式优先级（移进/归约）

```
expr:     expr '+' expr
       |  expr '-' expr
       |  expr '*' expr
       |  ...
       ;
```

如果使用上面的技术描述表达式语法，但是忘记用 **%left** 和 **%right** 定义优先级，那么会得到大量的移进/归约冲突。为所有的操作符分配优先级应该可以解决冲突。记住如果用其他方式使用任何一个操作符，例如，使用“-”指定数值范围，那么优先级还能屏蔽其他上下文中的冲突。

## 有限的向前查看（移进/归约或归约/归约）

移进/归约冲突的类别取决于yacc的有限的向前查看。也就是，能够看得更远的分析不会有冲突。例如：

```
rule: command optional_keyword '(  identifier_list ')'
        ;

optional_keyword: /* 空 */
        |       '(' keyword ')'
        ;
```

这个例子描述以要求的命令开始、以圆括号中要求的标识符结束的命令行，而且圆括号中有可选的关键字。当yacc得到输入流中第一个圆括号时，它就得到一个移进/归约冲突，它不知道是可选的关键字还是标识符列表。在第一种情况下，yacc 将移进 **optional_keyword** 规则中的圆括号，并且在第二种情况下，它归约空的 **optional_keyword** 并移至标识符列表。如果 yacc 能看得更远，它会发现两种情况的不同。但是它不能。

yacc 的默认情况是选择移进，意味着它总是假定在那里有可选的关键字。（在那种情况下，你实际上不能称它为可选的。）如果应用优先级，你能得到有利于归约的冲突解决办法，这意味着永远不能有可选的关键字。

不管如何伪造优先级，yacc都不能分析上面描述的示例的命令行，因为yacc缺乏要求的向前查看深度。如果我们不改变命令语言的语法，我们的选择只能是展开（flatten）描述：

```
rule:     command  (' keyword  )'  (' identifier_list ')'
        | command '(' identifier_list ')'
        ;
```

通过展开列表，我们允许语法分析程序用多个可能的指针向前扫描，直到它看见一个关键字或标识符，同时它能告诉所使用的规则。

“展开”是这个例子中实用的解决方案，但是当涉及更多的规则时，由于yacc描述的指数扩展使它很快就变得不切实际了。你可以遇到许多因有限的向前查看造成的移进/归约冲突，对它们来说，惟一实用的解决方案是改变语言，或者不使用yacc。

也可能由于有限的向前查看而得到归约/归约冲突。一种方式是同等表示(alternative)的重叠:

```
rule:       command_type_1 ':' [  ...
      |     command_type_2 ':' (  ...

command_type_1:   CMD_1 | CMD_2 | CMD_COMMON ;

command_type_2: CMD_A | CMD_B | CMD_COMMON ;
```

这种情况的解决方案是展开，正如我们前面提到的那样，或者拆开同等表示，正如下一节要描述的那样。

可以从有限的向前查看中得到归约/归约冲突，因为规则中间的动作实际上是必须被归约的匿名规则:

```
rule: command_list { <action for '[' form> } ':' '[' ...
| command_list { <action for '(' form> } ':' '(' ...
```

这里已经被展开了，所以在yacc中没办法使它工作。它只需要向前查看两个标记，而yacc不能。除非在语法分析程序和词法分析程序之间进行一些额外的通信，否则你只能让这个动作后移:

```
rule:     command_list ':' '[' { <action for '[' form> } ...
      |   command_list ':' '(' { <action for '(' form> } ...
```

## 同等表示的重叠（归约/归约）

在这种情况下，同一个LHS有两个可选的规则，并且它们接受的输入有部分重叠。最好的"赌注"是拆开两个输入集合。例如：

```
rule:       girls
        |   boys
        ;

girls:          ALICE
        |   BETTY
        |   CHRIS
        |   DARRYL
        ;

boys:       ALLEN
        |   BOB
        |   CHRIS
        |   DARRYL
        ;
```

因为yacc不能说明它们是**girls**还是**boys**，所以在**CHRIS**和**DARRYL**上将得到归约/归约冲突。解决这个冲突有几个方案。一个是：

```
rule:       girls | boys | either;

girls:          ALICE
        |   BETTY
        ;

boys:       ALLEN
        |   BOB
        ;

either:         CHRIS
        |   DARRYL
        ;
```

但是如果这些列表很长，或者是复杂的规则而不止是关键字的列表怎么办，所以想要最小化复制，而且**girls**和**boys**在yacc规范的许多地方都被引用？下面展示了另一种可能的方案：

```
rule:       just_girls
        |   just_boys
        |   either
        ;

girls:          just_girls
        |   either
        ;

boys:       just_boys
            either
        ;

just_girls: ALICE
        |   BETTY
        ;

just_boys:  ALLEN
        |   BOB
        ;

either:         CHRIS
            DARRYL
        ;
```

对“boys | girls”的所有引用都必须被修改。要么修改“boys | girls”的引用、要么修改列表，这没有办法避免。

但是如果无法拆开同等表示怎么办？如果只是不能找出打破重叠的好的方法，那么必须留下归约/归约冲突。yacc 为归约/归约使用默认的消除歧义的规则，就是选择 yacc 描述中的第一个定义。所以在上面的第一个“girls | boys”例子中，**CHRIS** 和 **DARRYL** 总是 **girls**。交换 **boys** 和 **girls** 列表的位置，那么 **CHRIS** 和 **DARRYL** 总是 **boys**。仍然会得到归约/归约警告，并且 yacc 将拆开同等表示，恰恰是你想尝试避免的情况。必须重写这个语法。

# 小结

yacc语法中的歧义和冲突只是编码错误的一种类型，寻找和纠正这种错误是成问题的。本章提出了一些纠正这些错误的技术。下一章，我们主要查看其他的错误源。

本章的目的是，希望读者在解决问题的层面上来理解问题。

总结如下几条：

- 在 *y.output* 中寻找移进/归约错误。
- 挑选归约规则。
- 挑选有关的移进规则。
- 了解归约规则将归约到的位置。
- 利用这方面的更多信息，应该能推导出导致冲突的标记流。

了解归约规则所归约到的位置通常和我们已说明的一样容易。有时，语法太复杂了，所以“搜索周围（hunt-around）”的方法不实用，并且为了找到所归约的状态，需要了解状态机的详细操作。

# 练习

1. 所有的归约/归约冲突和许多移进/归约冲突是因为有歧义的语法导致的。除了 yacc 与它们不同以外，有歧义的语法为什么是一个坏主意？
2. 找到类似 C、C++ 或 Fortran 这样的真正的编程语言的语法，并在 yacc 中运行它。这个语法有冲突吗？（它们几乎都有。）遍历 *y.output* 清单并决定导致冲突的原因。修复它们有多难？
3. 进行前面的练习之后，推论为什么语言通常用有歧义的语法定义和实现。

本章内容：
- 错误报告
- 错误恢复
- 练习

# 第九章 错误报告和恢复

前面两章讨论了在yacc语法中寻找错误的技术。本章将注意力转到错误校正和检测方面，即语法分析程序和词法分析程序如何检测错误。本章描述了如何在语法中结合错误检测和报告的技术。为了用一个完整的例子为这些讨论打下基础，引用第四章“菜单生成语言”中定义的菜单生成语言。

yacc提供了**error**标记和**yyerror**例程，这对于早期的工具版本是足够了。然而，当任何程序开始变得成熟时，特别是程序设计工具，为了更好地进行错误恢复，允许检测文件后面部分的错误及更好地报告错误是很重要的。

## 错误报告

错误报告应该对可能的错误给予尽可能多的详细资料。默认的yacc错误只声明语法错误存在，并且阻止正在进行的分析。在示例中，我们一般添加一种报告行号的机制。这样能提供错误的位置，但不能报告文件中的任何其他错误或指定行中错误发生的地方。

最好对可能的错误进行分类，或者定义一组错误类型并定义符号常量来标识错误。例如，在MGL中，错误可能是没有终止字符串。另一个错误也许是使用了错误

的字符串类型（引用字符串代替了标识符，反之亦然）。最低限度，MGL应该报告：

- 一般的语法错误（例如，没有意义的行）。
- 非终结的字符串。
- 字符串的错误类型（引用代替非引用，反之亦然）。
- 过早的文件尾。
- 使用的名字重复。

报告带行号的语法错误是一种好的机制；如果不能标识错误，就把这个机制作为一种退路。我们会在识别出可能发生错误的地方放置其他更特殊的错误报告。一般，这种机制对于指出输入文件中不正确的行已经足够用了，而对决定错误的性质也足够用了。

然而，错误校正不仅仅是yacc自己的责任。许多基本错误由lex检测更好。例如，通常引用字符串的匹配模式是：

```
\"[^\"\n]*\"
```

我们可能要检测非终结的引用字符串。一种潜在的解决方案是增加一条新规则来捕获非终结字符串，就像我们在第五章的SQL语法分析程序中所做的那样。如果引用字符串一直到没有闭引号的行尾，就会打印一个错误：

```
\"[^\"\n]*\"      {
                     yylval.string = yytext;
                     return QSTRING;
                  }
\"[^\"\n]*$       {
                     warning("Unterminated string");
                     yylval.string = yytext;
                     return QSTRING;
                  }
```

接受非法的输入然后用一个错误或警告报告它，这是一种能用于改善编译程序的错误报告的强大的技术。如果没有添加这条规则，编译程序就使用一般的“语法

错误”消息；通过报告特殊的错误，可以明确地告诉用户要调整什么。本章后面，我们将描述在这样的错误之后进行再同步和尝试继续操作的方法。

通过测试在应该使用标识符的地方使用了不正确的引用字符串或反过来的情况，可以显示yacc接受错误输入的能力。例如，下面的MGL规范片断应该产生这样的错误：

```
screen 'flavors'
```

而不是：

```
screen flavors
```

告诉用户字符串错误的类型而不只是说“语法错误”会更有用；这是初学用户所犯的错误类型。为了处理字符串错误类型，修改yacc语法来识别错误条件并报告它。因此，可以引入一个非终结标记来取代目前使用的标记**QSTRING**和**ID**。目前，MGL有下面的规则：

```
screen_name: SCREEN ID { start_screen($2); }
           | SCREEN { start_screen(strdup("default")); }
           ;
screen_terminator: END ID { end_screen($2); }
                 | END { end_screen(strdup("default")); }
                 ;
screen_contents: titles lines
               ;
titles: /* 空值 */
      | titles title
      ;
title: TITLE qstring { add_title($2); }
     ;
```

相反，使用下面的规则取代**QSTRING**和**ID**标记：

```
id:             ID { $$ = $1; }
     |    QSTRING { warning("String literal inappropriate", $1);
                $$ = $1;  /* 可在任何地方使用 */
          }
     ;
```

```
qstring:    QSTRING { $$ = $1; }
        |   ID { warning("Non-string literal inappropriate", $1);
                 $$ = $1; /* 可在任何地方使用 */
               }
        ;
```

下面，当yacc语法检测字符串文字或标识符时，它能查明错误的类型。不论怎样，我们继续使用不正确的文字；产生的C代码也许是错误的，但是可以让语法分析程序继续，并寻找更多的错误。有时出错恢复是不可能的；通常它合乎要求地发出警告，但实际上不做任何出错恢复。例如，*pcc*可移植的C编译程序，当它在输入流中看到一个违法的字符时就异常中断。编译程序的编写人确定了不可能再同步和继续的一个断点。然而，*pcc*报告可疑的分配并进行恢复，正如下面的C程序段那样：

```
int i = "oops";
```

在这种情况下，它发现一个错误消息，但是处理继续进行。

下一个示例检测重用的名字。这个示例阐明了出现在编译程序代码，而不是词法分析程序或分析程序中的错误检测类型；实际上，它在语法或词法分析程序中不会实现，因为它要求以前看见的标记内存（memory）。使用MGL的途径很直接。在这种情况下，重名在句法上没有问题，但在MGL产生的C代码中会导致重复，所以当看到新名字时，就把它"注册"在使用的名字的列表中。注册前，会搜寻整个列表，如果它在列表中，就报告一个重名错误。在附录九"MGL编译程序代码"中展示了整个代码。

## 更好的lex错误报告

一些简单的lex hackery（修改）可以让你产生比相当愚笨的lex的默认更好的错误报告。在SQL语法分析程序中使用一种非常简单的技术报告行号和当前的标记。跟踪每个\n字符上的行号，并且当前的标记在**yytext**中总是可用。

```
\n          lineno++;

%%
```

```
void yyerror(char *s)
{
        printf("%d: %s at %s\n", lineno, s, yytext);
}
```

每次保存一行输入的一个更复杂的窍门:

```
%{
char linebuf[500];
%}
%%
\n.* { strcpy(linebuf, yytext+1); /* 保存下一行 */
        lineno++;
        yyless(1);      /* 返回，只有 \n 重新扫描 */
     }
%%

void yyerror(char *s)
{
     printf("%d: %s at %s in this line:\n%s\n",
            lineno, s, yytext, linebuf);
}
```

模式 "\n.*" 匹配换行字符和整个下一行。动作代码保存这一行，然后用 **yyless()** 将它返回给扫描程序。

为了查明输入文件中错误标记的精确的位置，一行中保留一个记录当前位置的变量，在每个 "\n.*" 标记上将它设置为零，并通过 **yyleng** 在每个标记上增加它。假设行位置在 **tokenpos** 中，你就可以像这样报告错误位置:

```
void yyerror(char *s)
{
     printf("%d: %s:\n%s\n",
            lineno, s, linebuf);
     printf("%*s\n", 1+tokenpos, "^");
}
```

第二个 **printf** 在 **tokenpos** 位置处打印一个插入记号（^），如下所示:

```
3: syntax error:
CREATE TABLE sample ( color CHAR(10) NOT DEFAULT 'plaid' )
                                                 ^
```

# 错误恢复

前一节我们集中讲述了错误报告，本节要讨论错误校正问题。当检测到错误时，yacc语法分析程序保留在一个模糊的位置。不对现有的语法分析程序堆栈做一些调整就继续有意义的处理是不可能的。

没有理由说明错误恢复是必要的。一旦检测到一个错误，许多程序都不会尝试继续进行。对于编译程序，这个功能通常是不受欢迎的，因为运行编译程序本身代价很高。例如，C编译程序通常由几个阶段组成：预处理程序、语法分析程序、数据流分析程序和代码生成程序。在语法分析程序阶段报告一个错误并中止操作，要求修复这个问题并再次启动这个过程 —— 而预处理程序的工作必须重做。相反，可能恢复这个错误并继续检查文件中的其他错误，在调用下一阶段之前停止编译程序。这种技术通过缩短编辑－编译－测试周期改善了程序员的工作效率，因为可以在每次重复循环时修复几个错误。

一般，随着编译程序越来越复杂，错误恢复也越来越有价值。然而，错误恢复中涉及到的问题可以用简单的编译程序（例如MGL）来说明。

## yacc错误恢复

通过使用**error**标记，yacc可以做一些错误恢复的准备。本质上讲，错误标记用于在处理很可能会继续的语法中寻找同步点(注意我们说的是很可能)。有时，恢复尝试删除的错误状态不足以保证继续进行，并且错误消息还会级联(cascade)。语法分析程序将到达处理能够继续的点或者整个语法分析程序中止的点。

在报告一个语法错误以后，yacc分析程序放弃任何局部被分析的规则，直到它找到一个能用来移进**error**标记的规则。然后它阅读并放弃输入标记直到它找到在语法中能追随**error**标记的标记。后面的过程称为再同步。

在MGL中，将屏幕用做同步点。例如，看到一个错误标记后，它就放弃整个屏幕记录并在下一个屏幕重新开始。在第四章“菜单生成语言”中，屏幕的一条规则如下：

```
screens: /* 什么都没有 */
        | preamble screens screen
        | screens screen
        ;

screen:   screen_name screen_contents screen_terminator
        | screen_name screen_terminator
        ;
```

可以参考这一点用**screen**规则同步:

```
screen:   screen_name screen_contents screen_terminator
      | screen_name screen_terminator
      | screen_name error screen_terminator
          { warning("Skipping to next screen",(char *)0); }
      ;
```

错误恢复的这个基本"技巧"是尝试在输入流中向前移动足够远以使新的输入流不会受到旧的输入的不利影响。

错误恢复可以用适当的语言设计来增强。现代的程序设计语言使用语句终结器,充当便利的同步点。例如,当分析C语法时,合理的同步字符是分号,错误恢复能引入其他的问题,例如,如果语法分析程序略过声明寻找分号就会丢失声明,但是这些问题也包含在全面的错误恢复计划中。

由失去的状态(例如,放弃的变量声明)导致的级联错误能潜在地阻碍丢弃大部分输入流的策略。消除级联错误问题的一种机制是计算所报告的错误消息的数目,并在计数超过某些任意的数字时中止编译过程。例如,许多C编译程序在报告了一个文件中的10错误之后就中止了。

同任何其他的yacc规则一样,包含**error**的规则可以跟有动作代码。在那里典型的操作是错误后的清除,数据状态的重新初始化和其他"清理"工作,以便在恢复之后可以继续处理。例如,来自MGL的上一个错误恢复片段可以表示为:

```
screen:  screen_name screen_contents screen_terminator
      | screen_name screen_terminator
      | screen_name error
```

```
              { recover(); }
              screen_terminator
                    { warning("Skipping to next screen",(char *)0); }
          ;
```

可惜的是，这意味着在恢复状态机以前整个输入必须一直被分析到**screen_terminator**。这意味着如果没有发现屏幕终结符，会导致重大的语法错误。(回忆这个例子中没有屏幕规则以上级别的错误恢复。)通常，语法分析程序制止产生更多的错误消息，直到它成功地移进了3个标记而没有引入语法错误，这时它决定再同步并返回到它的正常状态。如果我们希望强制立即再同步，可以使用特殊的yacc动作**yyerrok**。告知语法分析程序恢复完成并且重新将语法分析程序设置为它的正常状态。这样上一个示例变为：

```
screen:   screen_name screen_contents screen_terminator
        | screen_name screen_terminator
        | screen_name error
              { yyerrok; recover(); }
              screen_terminator
                    { warning("Skipping to next screen",(char *)0); }
          ;
```

**recover()**程序应该确保下一个被读入的标记是**END**，它是**screen_terminator**需要的，否则会立即得到另一个语法错误。

使用**yyerrok**的最普通的位置是在交互式分析程序中。如果正在从用户中读取命令，每条命令都位于一个新行：

```
commands:     /* 空值 */
         |    commands command
         ;

command:    ...
           error {
                 yyclearin; /* 放弃向前查看 */
                 yyerrok;
                 printf("Enter another command\n");
           }
```

宏**yyclearin**放弃任何向前查看标记，并且**yyerrok**表明分析程序重新开始正常的分析，所以它以用户键入的下一个命令重新开始。

如果代码报告它的错误，那么错误程序应该使用yacc宏**YYRECOVERING()**测试语法分析程序是否正在尝试再同步，在这种情况下，不应该显示更多的错误，例如：

```
warning(char *err1, char *err2)
{
        if(YYRECOVERING())
              return; /* 这时不报告 */
        ...
}
```

## 错误标记放在哪里

在语法中适当放置错误标记是一种"魔术"，它们有两个相互冲突的目标。因为想尽可能确信再同步取得成功，所以想要将错误标记放置在语法的最高级规则中，也许甚至是起始规则，这样就总会有一条分析程序能恢复的规则。另一方面，因为想在恢复之前尽可能少地放弃输入，所以将错误标记放置在最低级别的规则以最小化部分被匹配的规则的数目，这些规则是语法分析程序在恢复期间必须放弃的。

如果标点符号分隔了列表中的元素，那么在错误规则中使用标点符号有助于找到同步点。例如，在C编译程序中，可以写成：

```
stmt:       ...
      |     RETURN expr ';'
      |     '{' opt_decls stmt_list '}'
            error ';'
      |     error '}'
      ;
```

因为C语句若是简单的语句就以";"结束，若是复合语句就以"}"结束，所以这两条**error**规则可以告诉语法分析程序在";"或"}"之后应该开始寻找下一条语句。

还可以在较低级别放置错误规则，例如作为表达式的规则，但是以我们的经验，除非这个语言提供了易于告诉表达式结束位置的标点符号或关键字，否则语法分析程序在这样的低级别上很少进行恢复。

## 编译程序错误恢复

上一节，描述了yacc为错误恢复提供的机制。本节要讨论由程序员提供的外部恢复机制。

错误恢复所固有的困难是它通常依靠语法的语义知识而不是语法知识。这就使在语法本身中恢复错误变得更复杂。可以使用我们以前建议的用户自制的机制重新安排编译程序的内部数据结构；另外，可能还会用恢复例程自己去扫描输入，并且采用启发式规则执行适当的错误恢复。例如，C编译程序编写人员也许决定最好通过跳过整个代码块而不是继续报告另外的错误来恢复在代码块的描述部分遇到的错误，也许决定在代码块的代码部分遇到错误时只需跳到下一个分号。编译程序或解释程序的真正有雄心的作者也许希望报告错误并尝试描述潜在的正确解决方案。

一旦编译程序执行了这样的错误恢复，那么它就清除了包含错误状态标记的yacc向前查看缓冲区，使用 **yyclearin**（而且也许可能使用 **yyerrok**），以便编译程序立即报告所找到的任何其他错误。(如果你对正确恢复没有信心，那么你可能不会调用 **yyerrok**。)

通常，复杂的错误校正使用yacc错误恢复处理基本的语法错误，使用用户提供的程序处理语义错误和数据结构恢复（例如、如果恢复跳到程序的结尾、就放弃局部变量的数据、嵌套BEGIN块和循环。）最终版本的MGL（在附录九“MGL编译程序代码”中）包括其中的一些错误恢复技术。

# 练习

1. 向第五章和附录十中的SQL语法添加错误恢复。最低限度，应该在SQL语

句之间的“;”处再同步。故意创建一些错误的SQL并把它放入语法分析程序中。它恢复的有多好？通常它会进行几种尝试得到有效恢复的错误规则。

2. （术语工程）yacc的错误恢复通过放弃输入标记来工作，直到它遇到同步正确的一些东西。另一种方式是插入而不是放弃标记，因为在许多情况下，容易预知标记必须跟随什么。例如，在C程序中，每个**break**和**continue**必须后跟分号，并且每个**case**的前面都必须有分号或闭大括号。在输入错误的情况下能够提示出适当的插入标记，这样增强yacc语法分析程序有多难？

对于这个练习，如果需要了解有关yacc的内部的更多的资料，参见建议阅读的参考书目。

# 附录一

# AT&T lex

AT&T lex是UNIX系统上的最普通的版本。如果不能确定你拥有哪种版本的lex，可以通过带*-v*标志尝试运行词法分析程序。如果产生简要的像下面这样的两行摘要，那么它就是 AT&T lex。

```
5/2000 nodes(%e), 16/5000 positions(%p), 5/2500 (%n),
4 transitions, 0/1000 packed char classes(%k), 0/5000
packed transitions(%a), 123/5000 output slots(%o)
```

如果产生的统计页第一行是 lex 的版本号，那么它就是 flex。

lex处理一个规范文件并产生词法分析程序的源代码。根据惯例，规范文件有*.l*扩展名。lex 产生的文件命名为 *lex.yy.c*。

AT&T lex 命令的语法是：

**lex** *[options] file*

其中，*options*（选项）可以是：

*-c* 用C（默认的）语言编写词法分析程序。在许多版本中不会出现这个将要被废弃的标志。

*-n* 不要打印含有表大小的摘要行。这是默认的，除非定义部分改变了lex的内部表之一的大小。

*-r* 用RATFOR（FORTRAN的方言）编写的动作，在大多数lex版本中，这个选项不再使用，并且在许多版本中已经不出现了。

*-t* 源代码被发送到代替默认文件*lex.yy.c*的标准输出。在*Makefiles*和将lex的输出送往命名文件的shell脚本中是很有用的。

*-v* 产生有限状态机的一行的统计摘要。当在lex规范的定义段指定任意表大小时就包含这个选项。

*-f* 不压缩生成的表而进行快速的移进（只适用于较小的词法分析程序）。只出现在lex的BSD派生的版本中。

必需在文件之前的命令行上指定选项。可以指定一个或多个文件，但它们都被看做一个规范文件。如果没指定文件就使用标准的输入。

lex库*libl.a*包含**yyreject**（它是使用**REJECT**的任意词法分析程序所必需的内部例程）和**main()**及**yywrap()**的默认版本。

参见第六章“lex规范参考”得到有关lex规范的更多的信息。

# 错误消息

本节讨论校正问题和由AT&T lex报告的错误。按字母顺序列出这些错误消息，可作为参考使用。

**Action does not terminate**（动作不终止）

当处理动作时，在动作终止之前遇到了文件尾。这通常意味着动作的闭大括号丢失了。

解决方案：添加丢失的大括号。

**bad state %d %o**（错误状态 %d%o）

这是内部的lex错误。

解决方案：向系统的软件维护人员报告这个问题。

**bad transition %d %d**（错误的转换 %d%d）

这是内部的 lex 错误。

解决方案：向系统的软件维护人员报告这个问题。

**Can't open %s**（不能打开 %s）

lex 不能打开输出文件 *lex.yy.c*。这通常是因为没有写目录的权限或者文件存在并且不可写。

解决方案：删除这个文件；改变目录权限；改变目录。

**Can't read input file %s**（不能读输入文件 %s）

lex 不能打开命令行上指定的文件。

解决方案：用有效的文件名调用 lex。

**ch table needs redeclaration**（ch 表需要重新声明）

当读取 lex 文件中的 %T 声明时，定义的字符数超过了 lex 分配给字符表的空间总量。

解决方案：删除转换表的字符，或者如果有 lex 的源文件，重新构建 lex 以维护更大的转换表。

**character '%c' used twice**（字符 '%c' 使用两次）

**character %o used twice**（字符 %o 使用两次）

当处理新的转换表时，字符被重新声明。

解决方案：删除无关的声明。

**character value %d out of range**（字符值 %d 超出范围）

当处理新的转换表时，lex 看到无效的字符值。有效的值在 1~256 的范围内。

解决方案：纠正无效的字符值。

**Definition %s not found**（没找到定义 %s）

看到 **{definition}** 之后，lex 不能在被声明的替换列表中找到它。

解决方案：取代该替换；在定义段定义替换。

**Definitions too long**（定义太长）

lex 对定义的尺寸有限制。该定义的长度太大了。

解决方案：使定义变短（也许可以拆分成两个）；重新构建 lex 以允许更长的定义。

**EOF inside comment**（注释内部的EOF）

当处理一个注释时，lex遇到文件尾。这通常是因为有一个未终结注释所导致的。

解决方案：添加丢失的“*/”。

**Executable statements should occur right after %**（执行语句应该就在%后发生）

当处理规则段时，lex看到没有相关模式的动作。将可执行代码直接放在规则分隔符的后面是合法的（这个代码在每次调用**yylex()**时执行）。这样的代码不能出现在规则段中的其他地方。

解决方案：调整与动作相关的模式或者将代码移到规则段的开头。

**Extra slash removed**（删除额外的斜杠）

无效的“/”会被忽略。这可能意味着模式中的文字“/”没有被引用。

解决方案：引用“/”，或者修复错误。

**Invalid request %s**（无效的请求%s）

当处理定义段时，可以看到lex声明（以“%”开始），但声明是无效的。有效的请求是以“%{”开始的文字块，或者“%”的后面跟着字母。参见第六章的“内部表”和“文字块”部分。

**Iteration range must be positive**（迭代范围必须是正的）

**Can't have negative iteration**（不能有负的迭代）

迭代范围（使用{**count,count**}）使用了负值或者第二个数使用了零值。

**No space for char table reverse**（char表没有反转空间）

内部的lex错误。

解决方案：向系统的软件维护人员报告这个问题。

**No translation given - null string assumed**（没有指定的转换－假定为空字符串）

当处理定义段时，lex看到没有替换文本的替换字符串。lex使用空字符串。这只是警告消息。

**Non-portable character class**（不可移植字符类）

当扫描规则时，不可移植转义序列被指定。只要在字符类中使用八进制常数，这种情况就会发生。

解决方案：容忍不可移植，或者在那里不使用八进制常数。

**Non-terminated string**（未终结字符串）

**Non-terminated string or character constant**（未终结字符串或字符常量）

**EOF in string or character constant**（字符串或字符常量中的EOF）

当读取规则或处理动作代码中的字符串时，lex在行结束前遇到没有终结的字符串。

解决方案：如果字符串被假定为继续到下一行，那么就添加连续标记符"\"；如果没有，那么就添加缺少的"."。

**OOPS – calloc returns a 0**（OOPS calloc返回一个0）

内部错误，或者系统虚拟内存不够。

解决方案：向软件的维护人员报告这个问题。

**output table overflow**（输出表溢出）

内部错误。

解决方案：向软件的维护人员报告这个问题。

**Parse tree too big %s**（分析树太大 %s）

lex已经用尽了分析树空间。

解决方案：简化lex规范；在定义段用 %e 声明增加分析树空间。

**Premature eof**（过早的eof）

当处理定义段时，看见了"%{"，但是没看见"%}"。

解决方案：添加丢失的"%}"。

**Start conditions too long**（起始条件太长）

起始状态（也称为起始条件）的名字的总长度超过了内部表的大小。

解决方案：缩短起始条件的名字。

**String too long**（字符串太长）

当读取规则时，lex遇到的字符串太长以至于不能存储在它的内部（静态的）缓冲区中。

解决方案：缩短字符串；使用更加紧密结合的形式重写字符串表达式；重新构建lex从而允许更大的字符串。

**Substitution strings may not begin with digits**（替换字符串不能以数字开始）

当处理定义段时，lex看到以数字开始的替换字符串名。

解决方案：用不是以数字开始的字符串取代这个替换字符串。

**syntax error**（语法错误）

lex 看到一行不正确的语法。

解决方案：修复这个错误。

**Too late for language specifier**（语言说明符出现的太晚了）

当处理定义段时，lex 在已经开始编写输出文件之后才看到 **%c** 或 **%r**（C 或 RATFOR 的语言选择）。

解决方案：早点声明这个语言。

**Too little core for final packing**（用于最终压缩的核心不够）

**Too little core for parse tree**（用于分析树的核心不够）

**Too little core for state generation**（用于状态生成的核心不够）

**Too little core to begin**（开始的核心不够）

内部错误，或者系统虚拟内存不够。

解决方案：向软件的维护人员报告这个问题。

**Too many characters pushed**（被压入的字符太多）

lex 已经用尽了输入标记可用的堆栈空间。

解决方案：缩短标记的大小；重新构建 lex 来接受很大的标记。

**Too many definitions**（太多的定义）

当分析输入文件时，为了存储定义 lex 用尽了它的静态空间。

解决方案：删除一些定义；重新构建 lex 来使用更大的定义表。

**Too many large character classes**（太多的大型字符类）

lex 因为大型字符类已经用尽了内部存储器。大型字符类用于描述括号（[]）内部出现的范围。

解决方案：缩短不同的大型字符类的数目；重新构建 lex 以允许更大的字符类。

**Too many packed character classes**（太多的压缩字符类）

解决方案：使用 **%k** 声明。

**Too many position %s**（太多的位置 %s）

lex 已经用尽了位置的空间。

解决方案：使用 **%p** 声明。

**Too many positions for one state-acompute**(一个状态有太多的位置 -acompute)

lex 最多可以为一个状态应用 300 多个位置，它是一个内部 lex 限制。这个错误表明一个非常复杂的状态。

解决方案：简化 lex 规范；重新构建 lex 从而使每个状态允许更多的位置。

**Too many right contexts**（太多的右端上下文）

lex 已经用尽了右端上下文的空间，它是“/”图形字符后的模式文本。

解决方案：减少所使用的右端上下文的数目；重新构建 lex 以允许更多的右端上下文。

**Too many start conditions**（太多的起始条件）

当处理定义段时，起始条件的数目超过了 lex 的静态内部表的大小。

解决方案：使用更少的起始条件；用更多的起始条件重新编译 lex。

**Too many start conditions used**（使用的起始条件太多）

为 lex 处理的一个特殊的规则指定了太多的起始条件。

解决方案：减少起始条件的数目；重新构建 lex 以允许每个规则有更多的起始条件。

**Too many states %s**（太多的状态 %s）

解决方案：使用 **%n** 声明。

**Too many transitions %s**（太多的转移%s）

解决方案：使用 **%a** 声明。

**Undefined start conditions %s**（未定义的起始条件 %s）

在一个模式中采用了 **<start state>**，但是 lex 在声明的起始状态的列表中未能找到它。

解决方案：声明这个起始状态，或者，如果它的名字拼错了就纠正。

**Unknown option %c**（未知选项 %c）

用未知开关调用 lex。有效的开关已经在前面列出了。

**yacc stack overflow**（yacc 堆栈溢出）

采用 yacc 语法来编写 lex。yacc 产生的语法用尽了它的堆栈空间。（如果你看到这种情况会令我们印象很深！）

解决方案：缩短或重新整理 lex 规范中的表达式；用更大的 yacc 堆栈区域重新构建 lex。

# 附录二

# AT&T yacc

## 选项

AT&T yacc 与 UNIX 的大多数版本一起分发，除了最近的 Berkeley UNIX 版本以外，这个版本包含 Berkeley yacc。如果不能确信你拥有 yacc 的哪个版本，可以尝试不用参数运行它。如果它出现：

```
fatal error: cannot open input file, line 1
```

那么它就是 AT&T yacc。如果出现命令语法的摘要，那么它就是 Berkeley yacc。

yacc 处理包含语法的文件并产生分析程序的源代码。根据惯例，语法文件拥有 *.y* 扩展名。yacc 产生的这个文件命名为 *y.tab.c*。

yacc 命令的语法是：

**yacc** *[options] file*

其中，*options*（选项）可以是：

*-d* 产生包含标记名的定义的头文件 *y.tab.h*。

*-l* 在产生的代码中省略 **#line** 结构。

-*t* 当编译 *y.tab.c* 时包括运行时调试代码。

-*v* 产生文件 *y.output*，它包含产生的语法分析程序中所有的状态清单和其他有用的信息。

为了编译由 yacc 产生的语法分析程序，必须供给一个 **main** 程序和一个支持例程 **yyerror**。UNIX 库 *liby.a* 包含这些程序的默认版本。

参见第七章“yacc 语法的参考”得到有关 yacc 规范的更多信息。

# 错误消息

本节讨论修正由 yacc 报告的问题和错误，除了第八章“yacc 歧义和冲突”所讨论的移进 / 归约和归约 / 归约错误。按字母顺序识别错误消息。

**%d rules never reduced**（永不被归约的 %d 规则）

语法中的一些规则永远不会被归约，因为它们永远不会出现在其他规则的右端，或者因为它们只在归约/归约冲突中涉及。yacc 报告没有归约的规则数。

解决方案：消除冲突或检查拼写错误。

**'000' is illegal**（'000' 是非法的）

八进制转义码指定了空字符，它是 AT&T yacc 为了内部使用而保留的。

解决方案：删除非法的转义码。

**action does not terminate**（动作不终止）

输入中的动作超出了文件的结束，可能是因为额外的 '{' 或者丢失了 '}' 引起的。

解决方案：修复错误的动作。

**action table overflow**（动作表溢出）

**no space in action table**（在动作表中没有空间）

当分析输入文件（或者处理输入）时，yacc 静态动作表被填满。

解决方案：简化动作；用更大的动作表重新编译 yacc；使用 bison 或 Berkeley yacc。

**bad %start construction**（错误的 %start 结构）

%start 指令不包含非终结的名字。

解决方案：改变 **%start** 使它有一个参数。

**bad syntax in %type**（%type 中的错误语法）

**%type** 指令的类型参数无效。发生这种情况的原因是指令没有参数。

解决方案：删除 **%type** 或赋予它一个参数。

**bad syntax on $ <ident> clause**（$<ident> 子句上的错误语法）

当读取一个动作时，出现无效的值类型。

解决方案：通过删除非法的说明或修复类型说明来纠正无效的类型说明。

**bad syntax on first rule**（第一条规则上的错误语法）

第一条规则在语句构成上不正确。例如，yacc 永远不会在第一条规则的后面找到冒号。

解决方案：修复第一条规则。

**bad tempfile**（错误的临时文件）

内部错误，或者用完磁盘空间的系统。

解决方案：返回 yacc；向软件维护人员报告这个问题。

**cannot open input file**（不能打开输入文件）

yacc 不能打开命令行上指定的输入文件，或者没有名字出现。

解决方案：纠正文件名。

**cannot open temp file**（不能打开临时文件）

yacc 尝试打开 *yacc.tmp* 临时文件，但是失败。可能是因为当前的目录不可写或者是因为不可写的 *yacc.tmp* 文件已经存在。

解决方案：删除 *yacc.tmp* 或改变目录权限。

**cannot open y.output**（不能打开 y.output）

**cannot open y.tab.c**（不能打开 y.tab.c）

**cannot open y.tab.h**（不能打开 y.tab.h）

yacc 尝试打开一个输出文件失败。可能是因为当前目录不可写或者是因为不可写的文件版本已经存在。

解决方案：删除这个文件或改变目录权限。

**cannot place goto %d**（不能放置 goto%d）

内部错误。

解决方案：向系统的软件维护人员报告这个问题。

**cannot reopen action tempfile**（不能重新打开动作临时文件）

yacc在称为*yacc.acts*的临时文件中保持所有的动作。这个文件已经消失；它可能在 yacc 运行时被删除了。

解决方案：在运行 yacc 时不删除 yacc 的临时文件。

**clobber of a array, pos'n %d, by%d**（数组的乱码，pos'n%d,by%d）

内部错误。

解决方案：向系统的软件维护人员报告这个问题。

**default action causes potential type clash**（默认的动作导致潜在的类型冲突）

没有动作的规则使用默认的“$$ = $1”，但是$1 的类型不同于规则的$$.的类型。

```
%union{
      int integer;
      char *string;
}

%TOKEN <integer> int
%type <string> s
%%
      ...
int: s ;
```

解决方案：添加明确的动作或纠正这个类型。最后一行可以纠正为：

```
int: s { $$ = atoi($1); } ;
```

**eof before %**（eof 在 % 之前）

当读取输入文件时，yacc 寻找规则段失败，可能是因为省略了“%%”。

解决方案：添加“%%”。

**EOF encountered while processing %union**（当处理 %union 时遇到 EOF）

文件在 **%union** 指令的中间结束，可以是因为丢失了“}”。

解决方案：添加丢失的大括号。

**EOF in string or character constant**（EOF 在字符串或字符常量中）

**EOF inside comment** （EOF 位于注释内）

文件在字符串、字符常量或注释中结束。

解决方案：添加闭引号或“*/”。

**Error; failure to place state %d**（错误；放置状态 %d 失败）

内部错误。

解决方案：向系统的软件维护人员报告这个问题。

**illegal %prec syntax**（非法的 %prec 语法）

没有符号名跟着 **%prec** 指令。

解决方案：添加一个。

**illegal comment**（非法的注释）

动作外部的规则段中的“/”后面没有“*”。

解决方案：删除斜杠或添加一个星号。

**illegal \nnn construction**（非法的 \nnn 结构）

八进制字符转义码包含不同于八进制数字的一些东西，例如：

```
%left  '\2z'
```

解决方案：纠正八进制字符转义码。

**illegal option: %c**（非法的选项：%c）

用不同于前面列出的有效的选项来运行 yacc。

**illegal or missing 'or"**（非法的或丢失 ' 或 "）

当读取字符串文字或字符文字时，yacc 寻找闭单引号或双引号失败。

解决方案：补充闭单引号或双引号。

**illegal rule:missing semicolon or |?**（非法规则：丢失分号或 |? ）

yacc 看见无效的字符，例如规则中的“%”。

解决方案：修正这条规则。

**internal yacc error:pyield %d**（内部 yacc 错误：pyield %d）

内部错误。

解决方案：向系统的软件维护人员报告这个问题。

**invalid escape**（无效的转义码）

"\" 后的字符是无效的转义字符。

解决方案：纠正或删除这个转义码。

**illegal reserved word: %s**（非法的保留字：%s）

跟在"%"后面的指令不是yacc可以理解的指令。

解决方案：如果可能就修复这条指令。而且，查看一下这个指令是否是bison指令。参见附录四"GNU bison"。

**item too big**（项目太大）

在构建输出字符串的过程中，yacc遇到了一个太大的项目，它不适合内部缓冲区。

解决方案：使用较短的名字（当项目的名字很大时这个错误就会发生；在我们使用的这个实现中，限制范围是370个字符。）

**more than %d rules**（超过%d条规则）

当从特定的语法中读入规则时，yacc会溢出为规则分配的静态空间。

解决方案：简化这个语法；用更大的状态表重新编译yacc；使用bison或Berkeley yacc。

**must return a value,since LHS has a type**（必须返回一个值，因为LHS有一个类型）

具有类型的左端的规则没有设置"$$"。

解决方案：通过指派一个适当的值将返回值添加给"$$"。

**must specify type for %s**（必须为%s指定类型）

%token指令没有被指定类型。

解决方案：添加一种类型。

**must specify type of $%d**（必须指定$%d的类型）

在一个动作中，yacc已经找到了必须有类型的值引用用法。

解决方案：在定义段声明符号的类型。

**newline in string or char.const.**（字符串或字符常量中的换行）

字符串或字符常量的运行超过了行尾。

解决方案：添加闭单引号或双引号。

**nonterminal %s illegal after %prec**（%prec 之后的非终结的 % 非法）

%prec 指令后面跟着非终结字符。

解决方案：纠正错误的 %prec。

**nonterminal %s never derives any token string**（非终结的 %s 永远不会派生出任何标记字符串）

递归规则无限循环，因为在左端没有非递归的替换。例如：

```
x_list: X' x list
```

没有用于 **x_list** 的其他规则。

解决方案：删除这条规则或添加非递归替换。这个例子可以重写为：

```
x_list:'X' x_list   'X' ;
```

**nonterminal %s not defined!**（非终结的 %s 没有定义！）

非终结符号永远不会出现在规则的左侧。yacc报告使用了未定义的非终结符号的行。

解决方案：定义这个符号或修改拼写错误。

**optimizer cannot open tempfile**（优化程序不能打开临时文件）

yacc 使用的临时文件不能被打开。

解决方案：在 yacc 运行时不要删除 yacc 临时文件。

**out of space**（空间用尽）

当运行优化程序时，yacc 用尽了它的静态的内部工作空间。

**out of space in optimizer a array**（优化程序中的空间用尽）

**a array overflow**（a 数组溢出）

**out of state space**（状态空间用尽）

yacc 的一个内部表用光了空间。

解决方案：简化语法；在 “a” 数组中用更多的空间重新构建yacc；使用bison 或 Berkeley yacc。

**Ratfor yacc is dead:sorry.**（Ratfor yacc 死掉了：对不起。）

-*r* 标志用于产生 RATFOR 语法分析程序。

解决方案：忠于C语言。

**redeclaration of precedence of %s**（%s 的优先级重复声明）

特定的标记在多个 **%left**、**%right** 或 **%nonassoc** 指令中有自己被声明的优先级。

```
%left PLUS MINUS
%left TIMES DIVIDE
%left PLUS
```

解决方案：删除所有额外的声明。

**Rule not reduced:%s**（规则没有被归约：%s）

规则永远不会被归约，因为它永远不会出现在其他规则的右侧，或者是因为它只包含在归约/归约冲突中。错误消息在 *y.output* 中报告。

解决方案：检查规则并重写，以便它可以归约。

**syntax error**（语法错误）

yacc 不理解这条语句。

解决方案：调整这条语句。

**token illegal on LHS of grammar rule**（语法规则的 LHS 上的标记非法）

标记在指定行的左侧被找到。而标记只能出现在右侧。

```
%token FOO
%%
FOO: ;
```

解决方案：纠正这条规则。

**too many characters in ids and literals**（在标识符和文字中有太多的字符）

当处理输入文件时，因为标识符和文字，yacc 用尽了内部静态的存储器。

解决方案：简化语法；用更大的静态表重新构建yacc；使用bison或Berkeley yacc。

**too many lookahead sets**（太多的向前查看符集合）

内部缓冲区被充满。

解决方案：简化语法或用更多的向前查看符集合空间重新构建 yacc。

**too many nonterminals, limit %d**（太多的非终结符、限制%d）

yacc遇到了比适合它的表更多的非终结符。

解决方案：简化这个语法；用更大的内部表重新构建yacc；使用bison或Berkeley yacc。

**too many states**（太多的状态）

内部表被充满。

解决方案：简化这个语法（这样、可以采用更少的状态）；通过重新编译yacc从而增加被允许的状态的数目；使用bison或Berkeley yacc。

**too many terminals, limit %d**（太多的终结符、限制%d）

这条语法找到了比适合yacc的静态地被定义的缓冲空间更多的标记。限制低到127个标记。

解决方案：简化语法；用更多的内部表重新构建yacc；使用bison或Berkeley yacc。

**type redeclaration of nonterminal %s**（非终结符号%s的类型重复声明）

**type redeclaration of token %s**（标记%s的类型重复声明）

非终结标记的数值类型被多次声明。例如：

```
%union{
int integer;
char *string;
}

%type <string> foo
%type <integer> foo
```

解决方案：删除不适用的那个**%type**指令。

**unexpected EOF before %**（%之前出现意外的EOF）

指定给yacc的文件是空的。

解决方案：在文件中插入内容（最好是yacc语法）。

**unterminated <...> clause**（未终结的<...>子句）

类型名（在尖括号中）超出了文件尾。

解决方案：插入闭括号。

**working set overflow**（工作集溢出）

内部表溢出了。

解决方案：简化语法或者用更多的工作集空间重新构建 yacc。

**yacc state/nolook error**（yacc 状态 /nolook 的错误）

内部错误。

解决方案：向系统的软件维护人员报告这个问题。

# 附录三

# Berkeley yacc

Berkeley yacc与AT&T yacc非常类似，只是多了几个额外的特征。

## 选项

Berkeley yacc的选项和AT&T yacc的选项相同，但还增加了以下几个选项：

-*b* pref　用*pref*代替y作为生成文件的前缀。

-*r*　为代码和表产生独立的文件。代码文件命名为*y.code.c*，表文件命名为*y.tab.c*。

Berkeley yacc没有库；必须向它提供你自己的**main()**和**yyerror()**的版本。

## 错误消息

本节讨论纠正由Berkeley yacc报告的问题和错误，除了第八章“yacc歧义和冲突”讨论的移进/归约和归约/归约错误以外。以字母**f**开头的每个错误消息都代表致命的错误，以**e**开头的消息代表错误，以**w**开头的消息代表警告。只要yacc

看到一个错误或致命的错误，它就会停止。大多数错误消息还包含输入文件名和行号，这里我们忽略它。

## 致命错误

**f - cannot open** *file*（不能打开文件）

yacc不能打开一个文件。如果它是你指定的名字，确信这个文件存在并且可读。如果它是yacc的临时文件或输出文件之一，确信这个目录可读并且指定的文件不是只读形式的。

**f - out of space**（空间用尽）

**f - too many gotos**（太多的goto）

**f - too many states**（太多的状态）

**f - maximum table size exceeded**（超出最大的表大小）

一个内部表被充满或者没有充足的虚拟内存可用。除非使用了惊人的具有几万个标记和规则的语法，否则这可能表示yacc中存在程序错误。

解决方案：向系统的软件维护人员报告这个问题。

## 常规错误

**e - unexpected end-of-file**（意外的文件尾）

输入文件在句法上不可能的地方结束。

解决方案：检查并调整输入。

**e - syntax error**（语法错误）

yacc不能找到强制的语法元素，例如，在“%”之后，它没有找到任何可能的允许在那里的单词。

解决方案：检查并调整输入。

**e - unmatched/***（不匹配的/*）

文件在注释的中间结束，可能是因为闭注释丢失或错误的键入。

解决方案：检查并调整输入。

**e - unterminated string**（无终结字符串）

字符串超过行尾，可能是因为闭引号丢失。

解决方案：添加丢失的引号。

**e - unmatched%{**（不匹配的%{）

文件以文字块结束，可能是因为“%}”丢失。

解决方案：添加丢失的“%}”。

**e - unterminated %union declaration**（未终结的%union声明）

文件以%**union**声明结束，可能是因为闭大括号丢失。

解决方案：添加丢失的“}”。

**e - too many %union declarations**（太多的%union声明）

有多个%**union**声明。yacc只允许有一个。

解决方案：删除额外的，或合并它们。

**e - illegal tag**（非法的标签）

数值类型标签必须是有效的C标识符，例如：

```
%token <ab&z> foo
```

标签ab&z是非法的。

解决方案：更改这个标签名。

**e - illegal character**（非法的字符）

八进制或十六进制的转义序列表示一个数值太大以至不适合*char*变量。

解决方案：使用0到255之间的字符值。

**e - illegal use of reserved symbol %s**（非法使用保留符号%s）

符号名$accept、$end、任何$$*N*形式（N是数字）的名字和由单个句点组成的名字都是为yacc的内部使用而保留的。

解决方案：挑选另一个名字。

**e - the start symbol %s cannot be declared to be a token**（起始符号%s不能被声明为一个标记）

标记出现在%**start**声明中。

解决方案：不要这样做。

**e - the start symbol %s is a token**（起始符号 %s 是一个标记）

起始符号出现在 %**token** 声明中。

解决方案：不要这样做。

**e - no grammar has been specified**（没有语法被指定）

语法的规则段不包含规则，可能是因为丢失了或附加了 %% 行。

解决方案：纠正这个错误。

**e - a token appears on the lhs of a production**（标记出现在产生式的左侧）

每条规则的左侧都必须是非终结符，而不是一个标记。

解决方案：纠正这个错误。

**e - unterminated action**（未终结的动作）

语法文件在动作的中间结束，可能是因为丢失了闭大括号。

解决方案：添加丢失的闭大括号。

**e - illegal $-name**（非法的 $- 名）

具有明确的标签的值引用是无效的形式，例如：$<foo>bar。

解决方案：纠正这个错误。

**e - $$ is untyped**（$$ 未被定义类型）

动作包含对 $$ 的引用，但是符号的左侧没有值类型设置。

解决方案：删除对 $$ 的引用，或者给符号分配一个类型。

**e - $%d(%s) is untyped**（$%d(%s)未被定义类型）

动作包含对 $*N* 的引用，但是相应的右侧符号没有值类型设置。

解决方案：删除对 $*N* 的引用，或者给符号分配一个类型。

**e - $%d is untyped**（$%d 未被定义类型）

超出数值引用的范围，例如，$0，需要一个明确的类型。

解决方案：使用一个明确的类型，例如，“$<sym>0”。

**e - the start symbol %s is undefined**（起始符号 %s 未被定义）

它的左侧没有带起始符号的规则。

解决方案：添加一个，或者纠正拼写错误。

## 警告类错误

**w - the type of %s has been redeclared**（%s 的类型已经被重复声明）

符号的值类型被不一致地多次设置。

解决方案：一个符号类型只声明一次。

**w - the precedence of %s has been redeclared**（%s 的优先权被重复声明）

标记在多个 **%left**、**%right** 或 **%nonassoc** 声明中出现。

解决方案：一个符号的优先权只设置一次。

**w - the value of %s has been redeclared**（%s 的值被重复声明）

标记的标记号被多次声明。

解决方案：一个标记号只声明一次。更好的方法是，让 yacc 为非文字标记选择它自己的标记号。

**w - the start symbol has been redeclared**（起始符号被重复声明）

语法中包含多个不一致的 **%start** 声明。

解决方案：删除其余的，只保留一个。

**w - conflicting %prec specifiers**（冲突的 %prec 说明符）

规则中包含多个不一致的 **%prec** 说明符。每条规则最多只能使用一个说明符。

解决方案：删除额外的优先级说明符。

**w - $%d references beyond the end of the current rule**（$%d 引用超出了当前规则的尾端）

这个动作包含对不存在的右侧符号的引用，例如，当右侧只包含 8 个符号时使用 $9。

解决方案：纠正这个错误。

**w - the default action assigns an undefined value to $$**（默认的动作向 $$ 分配一个未定义的值）

在没有明确的动作的规则中，$$ 和 $1 没有相同的值类型。例如：

```
%union{
        int integer;
        char *string;
```

```
}
%TOKEN <integer> int
%type <string> s
%%
        ...
int: s ;
```

解决方案：改变这种类型，或者添加恰当的动作代码，例如：

```
int: s { $$ = atoi($1); } ;
```

**w - the symbol %s is undefined**（符号 %s 未被定义）

指定非终结符从未出现在规则的左部。

解决方案：添加一个，或者纠正拼写错误。

## 信息类消息

**%s:%d rules never reduced**（%s:%d 规则永远不被归约）

一些规则永远不会被使用，因为它们不能在这个语法中使用，或者因为它们位于移进/归约或归约/归约冲突的失败的结尾上。

解决方案：改变语法来使用规则或删除这些规则。

**%d shift/reduce conflicts,%d reduce/reduce conflicts**（%d 移进/归约冲突，%d 归约/归约冲突）

这个语法包含冲突，如果不希望有这些冲突，应该修正它们。参见第八章"yacc 歧义和冲突"的详细资料。

# 附录四

# GNU bison

GNU 工程的 yacc 替换称为 *bison*。简要地说，GNU（Gnu's Not UNIX）是自由软件基金会的工程，它尝试创建开放源代码的类UNIX操作系统（尽管GNU不是公众域，但它可以免费使用并且有一个特意保证它免费可用的许可证）。因此，每个人都可以使用bison。要得到有关如何获得bison、GNU或者自由软件基金会的更多的信息，请通过以下地址联系：

Free Software Foundation, Inc.
675 Massachusetts Avenue
Cambridge, MA 02139
(617) 876-3296

能访问 Internet 的用户可以从 prep.ai.mit.edu 的 */pub/gnu* 目录下 FTP bison 和所有其他的 GNU 软件。

由bison的一些版本生成的语法分析程序受GNU "copyleft" 软件许可证的约束，它设置了有关GNU和GNU派生的软件的发布条件。如果计划使用bison来开发发布给其他用户的程序，务必要检查bison发布中包含的文件 *COPYING*，看看你是否同意这些条款。

本附录的描述反映 bison 的 1.18 版本，这个版本于 1992 年 5 月发布。

# 区别

一般，bison与yacc是兼容的，尽管有少量不能在bison中正确工作的yacc语法。bison派生于Berkeley yacc的早期版本，但是几年来由于各自的独立开发现在已经有了许多小的差别。尽管如此，当尝试处理与yacc相关的一些问题时，bison仍然是很方便的，特别是yacc的内部静态缓冲区的使用。

bison使用动态内存而不是静态内存，所以它通常接受AT&T yacc不接受的yacc语法。

而且，bison做了一些改进，这些改进被证明是有价值的：

- 定义段中的 **%expect** 告诉bison预期一定数量的移进/归约冲突。如果明确了这个数字，bison就会不报告这个数量的移进/归约冲突。
- 定义段中的 **%pure_parser** 告诉bison生成可重入的语法分析程序（没有全局变量）。这样可以在多线程的环境中使用这个语法分析程序，并且允许语法分析程序循环调用本身。在可重入的语法分析程序中，到 **yylex()** 的接口稍微有些不同，并且在动作和支持程序中的代码也必须是可重入的。
- **%semantic_parser** 和 **%guard** 在语义分析程序中使用，在标记的含义（或内容）而不是标记的类型的基础上尝试更复杂的错误校正。这样的分析程序更复杂，但是可以提供更多的功能。bison内部有两个模型分析程序，一个称为*bison.simple*、另一个称为*bison.hairy*。后面的这个用于语义分析程序。"防护装置"（guard）控制分析程序的动作，处理归约和错误。这个特征很少使用并且没有归档到在线bison手册中。
- 动作中的@N维护当前规则中标记的源文件行号和列号的信息，它在错误消息中是很有用的。这条信息必须由词法分析程序提供。参见bison手册的更详细的解释。
- bison不写到名为*y.tab.c*的文件。相反它将文件*filename.y*写到名为*filename.tab.c*的文件。命令行的标志可以让你指定其他的文件名，或者使用传统的yacc名字。

- bison有改变从默认的“yy”中生成的语法分析程序中符号的前缀的命令行选项。这样可以在同一个程序中包括更多的语法分析程序。

我们注意到bison和yacc有许多不同的地方，但是bison有100页的在线文档，它完整地解释了bison和yacc之间的区别。

# 附录五

# flex

lex的免费可用版本是*flex*。这是和4.4BSD一起分发的lex的版本，并且是由GNU工程发布的。Internet用户也能从ftp.ee.lbl.gov上FTP它。flex的最大优点是它比AT&T lex更可靠，可以生成更快的词法分析程序，并且没有lex对表大小的限制。除了必须复制作者的版权声明，flex对重新发布没有任何限制，而且在flex扫描程序上根本没有分发限制。

flex与lex高度一致。一些AT&T lex扫描程序需要修改才能和flex一起工作，下面进行详细介绍。

本附录的描述反映flex的2.3.7版本，该版本1991年3月发布。

## flex的区别

全书中我们已经提到flex和其他版本的lex之间的区别。下面是重要区别的小结：

- flex不需要外部库。（AT&T lex扫描程序必须通过命令行上的 *-ll* 与lex库连接在一起。）然而，用户必须提供**main**函数或调用**yylex**的其他函数。为了与POSIX兼容，flex 2.4将默认的**yywrap()**从宏改成库例程，这样，不定义自己**yywrap()**的扫描程序就需要与库连接在一起。

- flex有一个不同的、几乎无用的、lex的转换表的版本(lex规范文件中的%t和%T声明)。
- flex扩展了模式定义，与lex稍有不同。只要它扩展模式，它就要放置圆括号“()”括起扩展部分。例如，flex文档列出以下内容:

  ```
  NAME [A-Z][A-Z0-9]*
  %%
  foo{NAME}?    printf( "Found it\n" );
  %%
  ```

  在这个例子中，lex不匹配字符串“foo”，但flex匹配。不用分组，扩展的最后参数是引号操作符的目标。使用分组，整个扩展是“? ”操作符的目标。
- flex不支持未归档的内部lex变量**yylineno**。
- flex不让重新定义宏**input**和**output**。参见第六章中的“从字符串中输入”。在lex中时，通过修改flex文件指针**yyout**，**ECHO**输出可以被重定向。同样，通过修改flex文件指针**yyin**可以重定向输入。
- flex让扫描程序读取多重嵌套输入文件。参见第六章的“包含操作”。

flex提供下面的额外特征:

- flex提供了排它性起始条件(exclusive start condition)，也就是，当处于这种状态时不执行所有其他条件的条件。
- 特殊的模式“<<EOF>>”匹配文件的结尾。
- flex动态地分配表，所以表指令不是必需的，并且如果存在，可以忽略。
- 扫描程序的名字和参数从宏YY_DECL中取得。可以重新定义宏并给予扫描程序一个不同于**yylex**的名字或者使它采用参数，或者返回一个不同于*int*的值。
- flex允许不用大括号“{}”的情况下在同一动作行上编写多重语句(这样做是可怕的风格)。
- flex允许动作中有“%{”和“%}”。当它在动作中看到“%{”时，它将直到“%}”的所有内容都拷贝到生成的C文件中，而不是尝试匹配大括号。

- flex 扫描程序能用 C++ 和 C 语言编译，尽管它们没有利用 C++ 的面向对象的特征。

# 选项

flex 的选项比 AT&T lex 更多。

*-b* 在要求回溯法的规则的 *lex.backtrack* 中生成一个报告。回溯规则是很慢的，并且通常可以调整规则避免回溯。在线flex 文档更深入地讨论了这个选项的用法。

*-d* 在生成的扫描程序中生成调试代码。

*-f* 生成更快速但却更大的未压缩的“完全的”表。

*-i* 生成不区分大小写的扫描程序，它匹配规则中的大写或小写字符而不管模式中的字母的情况。

*-p* 产生扫描程序中使用的对性能有影响的特征报告。

*-s* 抑制回送未匹配输入的默认规则，所以生成的扫描程序在未匹配输入的错误上异常中止。

*-v* 产生扫描程序统计量的汇总表。

*-C*x 控制表压缩的程度。*x* 的可能的值是 efmF。更多细节参见 flex 文档。

*-F* 生成比完全的表更快或者更小的“快速”表。

*-I* 生成交互式扫描程序，这个程序在读取每个输入行上直接匹配标记，而不是向前寻找一个字符。

*-L* 在生成的 C 代码中不放置 *#lines*。

*-S*x 使用指定的词法分析程序框架而不是默认的。主要对调试 flex 本身有用。

*-T* 以跟踪方式运行，主要对调试 flex 本身有用。

*-8* 生成 8 位字符的扫描程序，即使本地默认为 7 位的字符。

# 错误消息

本节讨论如何纠正由 flex 报告的问题和错误。

**unrecognized'%'directive**（未被识别的'%'指令）

在定义段，% 的后面必须跟有“{”或“}”括起C代码，“s”或“x”字母之一用来声明起始状态，“anpek”之一用于表大小声明（被忽略的），或者用废弃的“otcu”中的一个。

解决方案：删除或纠正这个指令。

**illegal character**（非法字符）

在定义段中出现非法字符。

解决方案：删除或纠正这个字符。

**incomplete name definition**（不完整的名字定义）

名字定义（替换）不包含模式。

解决方案：添加一个。

**unrecognized %used/%unused construct**（未被识别的%used或%unused结构）

定义段包含废弃的 *%used* 或 *%unused* 声明的无效形式。

解决方案：删除它。

**bad row in translation table**（转换表中的错误行）

转换表中的每一行必须以数字开头。

解决方案：删除或纠正该行。

**undefined {name}**（未定义的命名模式名称）

大括号中对命名模式的引用（替换）涉及未被定义的名字。

解决方案：改变这个引用或定义这个名字。

**bad start condition name**（错误的起始条件名）

<> 中的起始条件前缀有一个无效的名字。名字必须是有效的C标识符。

解决方案：纠正这个名字。

**missing quote**（丢失引号）

引用的模式越过了行尾。

解决方案：添加这个丢失的引号。

**bad character inside {}'s**（{}内部出现的错误字符）

在模式中重复计数必须只由数字组成，也许由逗号分隔。

解决方案：纠正这个计数。

**missing }**（丢失}）

重复计数运行到行尾，大概因为闭大括号丢失。

解决方案：添加丢失的大括号。

**bad name in {}'s**（{}中出现的错误名字）

模式名（替换）必须由字母、数字、下划线和连字符组成。

解决方案：纠正这个名字。

**missing }**（丢失}）

大括号中的模式名运行到行尾，大概因为闭大括号丢失。

解决方案：添加丢失的大括号。

**EOF encountered inside an action**（在动作内部遇到EOF）

动作运行到文件尾，大概因为闭大括号丢失。

解决方案：添加丢失的大括号。

**warning - %used/%unused have been deprecated**（警告 —— 不支持%used或%unused声明）

这些废弃的声明不再起作用。

解决方案：删除它们。

**fatal parse error**（致命的分析错误）

分析输入的yacc语法分析程序发现一个不可恢复的语法错误。

解决方案：纠正这个错误。

**multiple <<EOF>> rules for start condition %s**（起始条件%s的多个<<EOF>>规则）

每个起始条件只能有一个EOF。

解决方案：删除其余的，只保留一个。

**warning - all start conditions already have <<EOF>> rules**（警告——所有的起始条件都已经拥有 <<EOF>> 规则）

如果所有的起始状态都已经拥有 EOF 规则，那么没有起始状态的 EOF 规则永远不会匹配。

解决方案：删除这条规则，或者纠正起始状态。

**start condition %s declared twice**（起始条件 %s 被声明两次）

每个起始状态只能被声明一次。

解决方案：删除这个重复的声明。

**undeclared start state %s**（未声明的起始状态 %s）

在 <> 中的起始状态前缀引用了未知状态。

解决方案：声明这个状态或者纠正拼写。

**scanner requires -8 flag**（扫描程序要求 -8 标志）

词法分析程序规范包含 8 位的字符，但是本地默认的为 7 位。

解决方案：删除 8 位的字符或使用 *-8* 标志。

**REJECT cannot be used with -f or -F**（REJECT 不能和 -f 或 -F 一起使用）

*-f* 和 *-F* 标志生成不能处理 REJECT 要求的回溯法的词法分析程序。

解决方案：不使用 REJECT 或者不使用那些标志。

**could not create lex.backtrack**（不能创建 lex.backtrack）

文件不能被创建，可能是因为目录或文件的以前的版本是只读的。

解决方案：删除文件的任何以前的版本，更改目录权限或者改为另一个目录。

**read() in flex scanner failed**（flex 扫描程序中的 read()失败）

输入文件上的 I/O 错误。

解决方案：磁盘被破坏或者 flex 中存在错误。

**-C flag must be given separately**（-C 标志必须独立给出）

不能在同一参数中将 *-C* 标志与其他任何东西合并。

解决方案：不要这样做。

**full table and -Cm don't make sense together**（全表和-Cm在一起没有意义）

**full table and -I are（currently） incompatible**（全表和-I（目前）是不可兼容的）

**full table and -F are mutually exclusive**（全表和-F互相排斥）

有不一致的表压缩选项。

解决方案：指定一个或另一个。

**-S flag must be given separately**（-S标志必须独立给出）

不能在同一参数中将-S标志与其他任何东西合并。

解决方案：使用分隔的参数。

**fatal error-scanner input buffer overflow**（致命错误——扫描程序输入缓冲区溢出）

**fatal flex scanner internal error–end of buffer missed**（致命的flex扫描程序内部错误——缓冲区尾丢失）

**fatal flex scanner internal error–no action found**（致命的flex扫描程序内部错误——没有找到动作）

**flex scanner jammed**（flex扫描程序堵塞）

**flex scanner push-back overflow**（flex扫描程序推回溢出）

**out of dynamic memory in yy_create_buffer()**（yy_create_buffer()中的动态内存用尽）

**unexpected last match in input()**（input()中意外的最后匹配）

在flex本身使用的扫描程序中有重大的内部错误。

所有的情况都表示一些分类的内部错误。

解决方案：向系统的软件维护人员报告问题。

**attempt to increase array size by less than 1 byte**（试图按照小于1个字节的尺寸来增加数组大小）

**attempt to increase array size failed**（试图增加数组大小失败）

**bad state type in mark_beginning_as_normal()**（mark_beginning_as_normal()中错误的状态类型）

**bad transition character detected in sympartition()**（sympartition()中找到的错误的转换字符）

**consistency check failed in epsclosure()**(epsclosure()中失败的一致性检测)

**consistency check failed in symfollowset**（symfollowset 中失败的一致性检测）

**could not create unique end-of-buffer state**（不能创建惟一的缓冲区尾状态）

**could not re-open temporary action file**（不能重新打开临时的动作文件）

**dynamic memory failure building %t table**（构建 %t 表的动态内存失败）

**dynamic memory failure in copy_string()**(在 copy_string()中的动态内存失败)

**dynamic memory failure in copy_unsigned_string()**(在 copy_unsigned_string()中的动态内存失败)

**dynamic memory failure in snstods()**(在 snstods()中的动态内存失败)

**empty machine in dupmachine()**(在 dupmachine()中的空机器)

**error occurred when closing backtracking file**（当关闭回溯文件时发生错误）

**error occurred when closing output file**（当关闭输出文件时发生错误）

**error occurred when closing skeleton file**（当关闭框架文件时发生错误）

**error occurred when closing temporary action file**（当关闭临时动作文件时发生错误）

**error occurred when deleting output file** （当删除输出文件时发生错误）

**error occurred when deleting temporary action file**(当删除临时动作文件时发生错误)

**error occurred when writing backtracking file**（当编写回溯文件时发生错误）

**error occurred when writing output file**（当编写输出文件时发生错误）

**error occurred when writing skeleton file**（当编写框架文件时发生错误）

**error occurred when writing temporary action file**（当编写临时动作文件时发生错误）

**found too many transition in mkxtion()**(在 mkxtion()中发现太多的转换)

**memory allocation failed in allocate_array()**(在 allocate_array()中的内存分配失败)

**request for <1 byte in allocate_array()**(在 allocate_array()中的请求小于 1 字节)

**symbol table memory allocation failed**（符号表内存分配失败）

**too many %t classes!**(太多的 %t 类!)

这些都表示 flex 中的内部错误。文件错误有时意味着磁盘空间不足。

解决方案：释放一些磁盘空间，或者向系统的软件维护人员报告问题。

# 词法分析程序示例的 flex 版本

第二章使用的两个 lex 例子代码指定 AT&T lex 从字符串而不是从文件中取得输入。flex 使用不同的方法来改变输入源。例 E-1 和例 E-2 是为 flex 编写的同样的例子。

例 E-1：分析命令行的 flex 规范 ape-05.l

```
%{
unsigned verbose;
char *progName;

int myinput(char *buf, int max);
#undef YY_INPUT
#define YY_INPUT(buf,result,max) (result = myinput(buf,max))
%}

%%

-h     |
"-?"   |
-help { printf("usage is: %s [-help | -h | -? ] [-verbose | -v]"
        " [(-file| -f) filename]\n", progName);
      }
-v     |
-verbose { printf("verbose mode is on\n"); verbose = 1; }

%%
char **targv;    /* 记录参数 */
char **arglim;   /* 参数结束 */

main(int argc, char **argv)
{
progName = *argv;
targv = argv+1;
arglim = argv+argc;
yylex();
}

static unsigned offset = 0;

/* 向 flex 提供原料块 */
/* 程序自己处理输入，所以我们每次传递一个参数 */
```

```
int
myinput(char *buf, int max)
{
	int len, copylen;

	if (targv >= arglim)
		return 0; /* EOF */
	len = strlen(*targv)-offset; /* 当前的参数总量 */
	if(len >= max)
		copylen = max-1;
	else
		copylen = len;
	if(len > 0)
		memcpy(buf, targv[0]+offset, copylen);
	if(targv[0][offset+copylen] == '\0') { /* 参数结束 */
		buf[copylen] = ' ';
		copylen++;
		offset = 0;
		targv++;
	}
	return copylen;
}
```

**例 E-2：具有文件名的 flex 命令扫描程序 ape-06.l**

```
%{
unsigned verbose;
unsigned fname;
char *progName;

int myinput(char *buf, int max);
#undef YY_INPUT
#define YY_INPUT(buf,result,max) (result = myinput(buf,max))
%}

%s FNAME

%%
[ ]+ /* 忽略空白 */ ;
<FNAME>[ ]+ /* 忽略空白 */ ;

-h	|
"-?"	|
-help { printf("usage is: %s [-help | -h | -? ] [-verbose | -v]"
```

```
            " [(-file| -f) filename]\n", progName);
          }
-v
-verbose { printf("verbose mode is on\n"); verbose = 1; }

-f
 file { BEGIN FNAME; fname = 1; }

<FNAME>[^ ]+ { printf("use file %s\n", yytext); BEGIN 0; fname = 2;}

[^ ]+ ECHO;
%%
char **targv;    /* remembers arguments */
char **arglim;   /* end of arguments */

main(int argc, char **argv)
{
        progName = *argv;
        targv = argv+1;
        arglim = argv+argc;
        yylex();
        if(fname < 2)
                printf("No filename given\n");
}

static unsigned offset = 0;

/* 向flex提供原料块 */
/* 程序自己处理输入，所以我们每次传递一个参数 */
int
myinput(char *buf, int max)
{
        int len, copylen;

        if (targv >= arglim)
                return 0; /* EOF */
        len = strlen(*targv)-offset; /* 当前参数的总量 */
        if(len >= max)
                copylen = max-1;
        else
                copylen = len;
        if(len > 0)
                memcpy(buf, targv[0]+offset, copylen);
        if(targv[0][offset+copylen] == '\0') { /* 参数结束 */
```

```
                buf[copylen] = ' ';
                copylen++;
                offset = 0;
                targv++;
            }
            else offset += copylen;
            return copylen;
        }
```

# 附录六

# MKS lex和yacc

Mortice Kern Systems有在MS-DOS和OS/2环境下运行的lex和yacc包。它包括一个非常好的有450页的手册，所以这次讨论集中在MKS和其他实现之间的区别上。它们可以从下面的地址中得到：

Mortice Kern Systems
35 King Street North
Waterloo, ON N2J2W9
Canada

电话：+1 519 884 2251
在美国为 (800) 265-2797
E-mail: inquiry@mks.com

## 区别

大多数区别是由于运行在MS-DOS或OS/2环境，而不是UNIX环境。

- 输出文件有不同的名字：*lex_yy.c*、*ytab.c*、*ytab.h*和*y.out*，而不是*lex.yy.c*、*y.tab.c*、*y.tab.h*和*y.output*。

- MKS lex 有自己的处理嵌套包含文件的方法。参见第六章的“包含操作”。
- MKS lex有自己的重新设置词法分析程序为初始状态的方法。参见第六章中的“从 yylex()中返回值”。
- MKS lex 使用宏 **yygetc()**读取输入。可以重新定义它来改变输入源。参见第六章的“从字符串中输入”。
- 标准的lex标记缓冲区只有100个字符。可以通过重新定义一些宏来扩大它。参见第六章的“yytext”。
- 产生的内部的yacc表不同。这就使错误校正稍微有些不同；一般，在注意到向前查看标记上的错误以前，MKS yacc 比 UNIX yacc 执行的归约要少。

# 新特征

- MKS lex 和 yacc 能产生 C++、Pascal 以及 C 的扫描程序和语法分析程序。
- MKS 提供yacc 追踪系统，单步的面向屏幕的语法调试程序在它工作时可以插入断点并检查语法分析程序。
- MKS lex 和 yacc 都可以让用户改变生成的 C 代码中默认为“yy”的符号前缀，所以在一个程序中可以包含多个词法分析程序和语法分析程序。
- MKS yacc 可以动态地分配它的堆栈，允许递归，并可重入语法分析程序。
- MKS yacc有选择优先权，这可以使你通过指定向前查看标记来解决归约/归约冲突（向前查看标记可以告诉它使用哪一条规则）。
- MKS lex 库包含跳过 C 样式注释的程序，并处理包括转义码的 C 字符串。
- MKS yacc 归档的内部变量比 AT&T 或 Berkeley yacc 多。这样就使错误校正程序可以访问并改变更多的语法分析程序内部的状态。
- 这个包包括C、C++、dBASE、Fortran、Hypertalk、Modula-2、Pascal、pic（*troff* 图片语言）和 SQL 的示例扫描程序和语法分析程序。

# 附录七

# Abraxas lex 和 yacc

Abraxas 软件公司提供了 pcyacc，它包含 pcyacc 和 pclex，即 yacc 和 lex 的 MS-DOS 和 OS/2 版本。它可以从下面地址得到：

Abraxas Software
7033 SW Macadam Avenue
Portland OR 97219
Phone: +1 503 244 5253

pclex 是基于 flex 的，所以我们所说的有关 flex 的大部分内容也适用于 pclex。

## 区别

- 输出文件有不同的名字：*lex_yy.c*、*yytab.c*、*yytab.h* 和 *yy.lrt*，而不是 *lex.yy.c*、*y.tab.c*、*y.tab.h* 和 *y.output*。
- 标准的lex输入缓冲区只有256个字符。可以通过重新定义一些宏来扩大它。参见第六章中的“yytext”。

# 新特征

- 有一个选项可让你检查yacc规范的语法，而不是等待它产生一个完整的语法分析程序。
- 每当它归约一条规则，语法分析程序可以用那条规则的符号编写一行插入到文件中。（Abraxas 将它作为语法分析树选项来引用。）
- 可选的扩展的错误校正库允许更完整的错误报告和恢复。
- 这个包包括 ANSI 和 K&R C、C++、Cobol、dBase III 和 IV、Fortran、Hypertalk、Modula-2、Pascal、pic（与*troff*不相关的demo语言）、Postscript、Prolog、Smalltalk、SQL 以及 yacc 和 lex 本身的示例扫描程序和语法分析程序。

# 附录八

# POSIX lex 和 yacc

IEEE POSIX P1003.2 标准定义 lex 和 yacc 的可移植的标准版本。在几乎所有的情况下，这些标准仅仅系统化了长期存在的现有习惯。POSIX lex 非常类似于 flex，但有更加特殊的特征。POSIX yacc非常类似于AT&T yacc或Berkeley yacc。输入文件名和输出文件名与 flex 及 AT&T yacc 中的文件名相同。

## 选项

POSIX lex 命令的语法是:

**lex** [*options*] [*file* . . . ]

如果指定多个输入文件，它们连起来形成一个词法分析程序规范。

*options*（选项）如下所示:

*-c* 用 C 语言编写动作（将要废弃的）。

*-n* 抑制汇总统计行。

*-t* 向标准输出而不是向默认文件 *lex.yy.c* 发送源代码。

-*v* 产生有限状态机的很短的统计汇总。当在lex规范的定义段中指定任意表大小时就隐含这个选项。

yacc 命令的语法是:

**yacc** [*options*] *file*

选项如下所示:

-*b*xx 使用“xx”而不是默认的“yy”作为生成的文件名的前缀。

-*d* 为了让词法分析程序使用，生成包含标记名定义的头文件 *y.tab.h*。

-*l* 在生成的代码中不包括 **#line** 结构。这些结构有助于用错误消息标识规范文件中的行。

-*p*xx 在生成的C代码中使用“xx”而不是默认的“yy”作为符号的前缀。

-*t* 在编译 *y.tab.c* 时包括运行时调整代码。

-*v* 产生文件 *y.output*，用于分析语法中的歧义性和冲突。这个文件包含分析表的描述。

# 区别

主要区别源于POSIX的国际化。

- 在大多数现有的版本中，POSIX没标准化各版本实现方法不兼容的特征。因此，POSIX没办法指定改变lex输入源（除了分配给 **yyin** 以外）或者改变内部缓冲区的大小。它没有lex转换表形式。POSIX兼容的实现可以提供这些特征之一作为扩展。
- 在AT&T lex中，**yywrap()**是可以重新定义的函数，而不是宏。
- 通过使用 %**array** 或者 %**pointer** 声明，可以强制 **yytext** 为一个数组或一个指针。缺乏这样的声明时，实现可以使用这两种情况。

- POSIX lex定义额外的字符正则表达式来处理扩展的非英文的字符集。参见第六章的“正则表达式语法”。
- POSIX yacc有一个库，这个库具有**main()**和**yyerror()**的标准版本。你可以（并且也许应该）编写自己的**main()**和**yyerror()**版本。

# 附录九

# MGL 编译程序代码

第四章“菜单生成语言”提出了MGL的lex和yacc语法。下面给出MGL的完整代码，包括第四章没有展示的运行时支持代码。可以对运行时代码做许多改善，例如：

- 不正确的输入后的屏幕清除。
- 更好地处理 **system()**调用，特别是保存和恢复终端方式和屏幕内容。
- **main** 程序的自动生成。目前，必须在调用程序外部定义。而且，在结束之前必须调用 **menu_cleanup** 例程。
- 更灵活的命令处理，例如，允许命令的惟一的前缀而不是要求整个命令。
- 每次在 *cbreak* 模式中键入一个字符而不是每次在当前行键入一个字符。
- 更灵活的菜单嵌套。

参见前言了解有关获取这个代码的在线拷贝的方式。

## MGL yacc 源代码

下面是文件 *mglyac.c*。

```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int screen_done = 1; /* 执行完为1，否则为0 */
char *act_str;       /* 动作的额外参数 */
char *cmd_str;       /* 命令描述的额外参数 */
char *item_str;      /* 项目描述的额外参数 */
%}
%union {
    char   *string;  /* 字符串缓存区 */
    int    cmd;      /* 命令值 */
}

%token <string> QSTRING ID COMMENT
%token <cmd> SCREEN TITLE ITEM COMMAND ACTION EXECUTE EMPTY
%token <cmd> MENU QUIT IGNORE ATTRIBUTE VISIBLE INVISIBLE END

%type <cmd> action line attribute command
%type <string> id qstring

%start screens

%%

screens:  screen
        | screens screen
        ;
screen:   screen_name screen_contents screen_terminator
        | screen_name screen_terminator
        ;

screen_name: SCREEN id { start_screen($2); }
           | SCREEN { start_screen(strdup("default")); }
           ;

screen_terminator: END id { end_screen($2); }
                 | END { end_screen(strdup("default")); }
                 ;

screen_contents: titles lines
               ;
```

```
titles: /* 空值 */
        | titles title
        ;

title: TITLE qstring { add_title($2); }
      ;

lines: line
      | lines line
      ;

line: ITEM qstring command ACTION action attribute
          { item_str = $2;
          add_line($5, $6);
            $$ = ITEM;
           }
          ;

command: /* 空值 */ { cmd_str = strdup(""); }
        | COMMAND id { cmd_str = $2; }
        ;

action: EXECUTE qstring
          { act_str = $2;
          $$ = EXECUTE;
          }
       | MENU id
          { /* 生成"menu_" $2 */
          act_str = malloc(strlen($2) + 6);
          strcpy(act_str,"menu_");
            strcat(act_str, $2);
          free($2);
          $$ = MENU;
        }
        QUIT { $$ = QUIT; }
       | IGNORE { $$ = IGNORE; }
       ;

attribute: /* 空值 */ { $$ = VISIBLE; }
          | ATTRIBUTE VISIBLE { $$ = VISIBLE; }
          | ATTRIBUTE INVISIBLE { $$ = INVISIBLE; }
          ;

id: ID
```

```
        { $$ = $1; }
    | QSTRING
        { warning("String literal inappropriate",
                (char *)0);
          $$ = $1; /* 可用于任何地方 */
        }
    ;

qstring: QSTRING { $$ = $1; }
        | ID
          { warning("Non-string literal inappropriate",
                  (char *)0);
            $$ = $1; /* 可用于任何地方 */
          }
        ;
%%

char *progname = "mgl";
int lineno = 1;

#define DEFAULT_OUTFILE "screen.out"

char *usage = "%s: usage [infile] [outfile]\n";

main(int argc, char **argv)
{
        char *outfile;
        char *infile;
        extern FILE *yyin, *yyout;
        progname = argv[0];

        if(argc > 3)
        {
              fprintf(stderr,usage, progname);
              exit(1);
        }
        if(argc > 1)
        {
              infile = argv[1];
              /* 打开后读取 */
              yyin = fopen(infile,"r");
              if(yyin == NULL) /* 打开失败 */
              {
                    fprintf(stderr,"%s: cannot open %s\n",
                          progname, infile);
```

```
                exit(1);
            }
        }

        if(argc > 2)
        {
            outfile = argv[2];
        }
        else
        {
                outfile = DEFAULT_OUTFILE;
        }

        yyout = fopen(outfile,"w");
        if(yyout == NULL) /* 打开失败 */
        {
                fprintf(stderr,"%s: cannot open %s\n",
                progname, outfile);
            exit(1);
        }

        /* 从现在开始, yyin 和 yyout 正常交互 */

        yyparse();

        end_file(); /* 写出最终的任何信息 */

        /* 现在检测 EOF 条件 */
        if(!screen_done) /* 在屏幕中间 */
        {
            warning("Premature EOF",(char *)0);
            unlink(outfile); /* 删除错误文件 */
            exit(1);
        }
        exit(0); /* 没有错误 */
}

warning(char *s, char *t) /* 显示警告消息 */
{
        fprintf(stderr, "%s: %s", progname, s);
        if (t)
            fprintf(stderr, " %s", t);
        fprintf(stderr, " line %d\n", lineno);
}
```

# MGL lex 源代码

下面是文件 *mgllex.l*:

```
%{
#include "mglyac.h"
#include <string.h>

extern int lineno;
%}

ws          [ \t]+
comment     #.*
qstring     \"[^\"\n]*[\"\n]
id          [a-zA-Z][a-zA-Z0-9]*
nl          \n

%%

{ws}        ;
{comment}   ;
{qstring}   { yylval.string = strdup(yytext+1); /* 跳过开引号 */
              if(yylval.string[yyleng-2] != '"')
                warning("Unterminated character string",(char *)0);
            else /* 删除闭引号 */
                yylval.string[yyleng-2] = '\0';
              return QSTRING;
            }
screen      { return SCREEN; }
title       { return TITLE; }
item        { return ITEM; }
command     { return COMMAND; }
action      { return ACTION; }
execute     { return EXECUTE; }
menu        { return MENU; }
quit        { return QUIT; }
ignore      { return IGNORE; }
attribute   { return ATTRIBUTE; }
visible     { return VISIBLE; }
invisible   { return INVISIBLE; }
end         { return END; }
{id}        { yylval.string = strdup(yytext);
              return ID;
```

```
            }
{nl}        { lineno++; }
            { return yytext[0]; }
%%
```

# 支持 C 代码

下面是文件 *subr.c*:

```
/* subr.c */

/*
 * 菜单生成语言 (MGL) 的支持子例程
 *
 * Tony Mason
 * 1988 年 11 月
 * 1992 年由 John Levine 完成
 */

/* 包含 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mglyac.h"
#include "mgl-code"          /* 包含将要构建的框架文件的定义 */

extern FILE *yyin, *yyout;

/* 导入 */
extern int screen_done;
extern char *cmd_str, *act_str,*item str;

/* 导出 */

/* 局部变量 */
static char current_screen[100]; /* 有原因? */
static int done_start_init;
static int done_end_init;
static int current_line;
struct item {
        char              *desc;      /* 项目描述 */
        char              *cmd;       /* 命令 */
```

```
        int             action;      /* 要采用的动作 */
        char            *act_str;    /* 动作操作 */
        int             attribute;   /* 可见 / 不可见 */
        struct item     *next;       /* 列表的下一个成员 */
} *item_list, *last_item;

/* 宏 */
#define SCREEN_SIZE 80
void cfree(char *); /* 如果不为空就释放 */

/* 代码 */

/*
 * start_screen:
 * 这个程序开始屏幕的准备。
 * 它编写导言并修改全局状态变量 screen_done
 * 来展示屏幕正在进行（因此，如果看到 EOF 时屏幕正在进行，
 * 那么就会给出适当的错误消息）。
 */

start_screen(char *name) /* 创建屏幕名 */
{
        long time(),tm = time((long *)0);
        char *ctime();

        if(!done_start_init)
        {
                fprintf(yyout,
                    "/*\n * Generated by MGL: %s */\n\n",
                    ctime(&tm));
                dump_data(screen_init);
                done_start_init = 1;
        }
        if(check_name(name) == 0)
                warning("Reuse of name",name);
        fprintf(yyout,  "/* screen %s */\n", name);
        fprintf(yyout, "menu_%s()\n{\n",name);
        fprintf(yyout,
                "\textern struct item menu_%s_items[];\n\n",
                name);
        fprintf(yyout, "\tif(!init) menu_init();\n\n");
        fprintf(yyout, "\tclear();\n\trefresh();\n");

        if(strlen(name) > sizeof current_screen)
```

```
		warning("Screen name is larger than buffer",(char *)0);
	strncpy(current_screen, name, sizeof(current_screen) - 1);

	screen_done = 0;
	current_line = 0;

	return 0;
}

/*
 * add_title:
 * 向屏幕代码添加居中的文本
 */
add_title(line)
char *line;
{
	int length = strlen(line);
	int space = (SCREEN_SIZE - length) / 2;

	fprintf(yyout, "\tmove(%d,%d);\n",current_line, space);
	current_line++;
	fprintf(yyout, "\taddstr(\"%s\");\n",line);
	fprintf(yyout, "\trefresh();\n");
}

/*
 * add_line:
 * 向动作表添加一行。在添加了
 * 所有的行以后它就会被写出。
 * 要注意一些信息在全局变量中。
 */

add_line(action, attrib)
int action, attrib;
{
	struct item *new;

	new = (struct item *)malloc(sizeof(struct item));

	if(!item_list)
	{	/* 第一个项目 */
		item_list = last_item = new;
	}
```

```
        else
        {       /* 列表上已有的项目 */
                last_item->next = new;
                last_item = new;
        }

        new->next = NULL;  /* 标记列表的结束 */

        new->desc = item_str;
        new->cmd = cmd_str;
        new->action = action;

        switch(action)
        {
        case EXECUTE:
                new->act_str = act_str;
                break;
        case MENU:
                new->act_str = act_str;
                break;
        default:
                new->act_str = 0;
                break;
        }
        new->attribute = attrib;
}

/*
 * end_screen:
 * 结束屏幕，打印出后同步码
 */

end_screen(char *name)
{

        fprintf(yyout, "\tmenu_runtime(menu_%s_items);\n",name);

        if(strcmp(current_screen,name) != 0)
        {
                warning("name mismatch at end of screen",
                    current_screen);
        }
        fprintf(yyout, "}\n");
        fprintf(yyout, "/* end %s */\n",current_screen);
```

```
        process_items();

        /* 编写文件外部的初始代码 */
        if(!done_end_init)
        {
                done_end_init = 1;
                dump_data(menu_init);
        }

        current_screen[0] = '\0'; /* 没有当前屏幕 */

        screen_done = 1;

        return 0;
}

/*
 * process_items:
 * 到达菜单项目列表并将它们写到外部的
 * 已初始化的数组，而且定义运行时的
 * 支持模块（它就在这个表的下面）
 * 使用符号常数。
 */
process_items()
{
        int cnt = 0;
        struct item *ptr;

        if(item_list == 0)
                return; /* 什么都不做 */
        fprintf(yyout, "struct item menu_%s_items[]={\n",current_screen);
        ptr = item_list;

        /* 走查列表 */
        while(ptr)
        {
                struct item *optr;

                if(ptr->action == MENU)
                        fprintf(yyout,
                          "{\"%s\",\"%s\",%d,\"\",%s,%d},\n",
                          ptr->desc,ptr->cmd, ptr->action,
                          ptr->act_str,ptr->attribute);
                else
```

```
                fprintf(yyout,
                   "{\"%s\",\"%s\",%d,\"%s\",0,%d},\n",
                   ptr->desc,ptr->cmd, ptr->action,
                   ptr->act_str ? ptr->act_str : "",
                   ptr->attribute);

            cfree(ptr->desc);
            cfree(ptr->cmd);
            cfree(ptr->act_str);
            optr = ptr;
            ptr = ptr->next;
            free(optr);
            cnt++;
        }
        fprintf(yyout,
            "{(char *)0, (char *)0, 0, (char *)0, 0, 0},\n");
        fprintf(yyout, "};\n\n");
        item_list = 0;

        /* 完成所有工作的下一个运行时的模块 */;
}

/*
 * 这个程序采用字符串的空结束列表
 * 并在标准输出上打印它们。它的生命中的惟一目的是
 * 打印出大的静态的数组，
 * 这个数组组成了生成菜单的运行时代码。
 */

dump_data(array)
char **array;
{
        while(*array)
            fprintf(yyout, "%s\n",*array++);
}

/*
 * 这个程序写出了运行时的支持
 */

end_file()
{

        dump_data(menu_runtime);
}
```

```
/*
 * 检查一个名字，看看它是否已经被使用。
 * 如果没有使用返回 1；否则返回 0。这个程序还在
 * 远处储藏了用于将来引用的名字，要注意这个程序是完全动态的。
 * 当然，只设置静态数组更容易，但静态数组缺乏灵活性。
 * easier to just set up a static array, but less flexible.
 */

check_name(name)
char *name;
{
        static char **names = 0;
        static name_count = 0;
        char **ptr,*newstr;

        if(!names)
        {
                names = (char **)malloc(sizeof(char *));
                *names = 0;
        }

        ptr = names;
        while(*ptr)
        {
                if(strcmp(name,*ptr++) == 0) return 0;
        }

        /* 没有使用 */
        name_count++;
        names = (char **)realloc(names, (name_count+1) * sizeof(char *));
        names[name_count] = 0;
        newstr = strdup(name);
        names[name_count-1] = newstr;
        return 1;
}

void
cfree(char *p)
{
        if(p)
                free(p);
}
```

下面是文件 *mgl-code*，由 MGL 拷贝到生成的 C 文件中去的支持代码：

```
/*
 * MGL 运行时支持代码
 */

char *screen_init[] = {
"/* 初始化信息 */",
"static int init;\n",
"#include <curses.h>",
"#include <sys/signal.h>",
"#include <ctype.h>",
"#include \"mglyac.h\"\n",
"/* 用于存储菜单项目的结构 */",
"struct item {",
"\tchar *desc;",
"\tchar *cmd;",
"\tint action;",
"\tchar *act_str; /* 执行字符串 */",
"\tint (*act_menu)(); /* 调用适当的函数 */",
"\tint attribute;",
"};\n",
0,
};

char *menu_init[] = {
"menu_init()",
"{",
"\tvoid menu_cleanup();\n",
"\tsignal(SIGINT, menu_cleanup);",
"\tinitscr();",
"\tcrmode();",
"}\n\n",
"menu_cleanup()",
"{",
"\tmvcur(0, COLS - 1, LINES - 1, 0);",
"\tendwin();",
"}\n",
0,
};

char *menu_runtime[] = {
"/* 运行时 */",
"",
```

```
"menu_runtime(items)",
"struct item *items;",
"{",
"\tint visible = 0;",
"\tint choice = 0;",
"\tstruct item *ptr;",
"\tchar buf[BUFSIZ];",
"",
"\tfor(ptr = items; ptr->desc != 0; ptr++) {",
"\t\taddch('\\n'); /* 跳过一行 */",
"\t\tif(ptr->attribute == VISIBLE) {",
"\t\t\tvisible++;",
"\t\t\tprintw(\"\\t%d) %s\",visible,ptr->desc);",
"\t\t}",
"\t}",
"",
"\taddstr(\"\\n\\n\\t\"); /* tab输出，使它看起来更整齐 */",
"\trefresh();",
"",
"\tfor(;;)",
"\t{",
"\t\tint i, nval;",
"",
"\t\tgetstr(buf);",
"",
"\t\t/* 数字选择？ */",
"\t\tnval = atoi(buf);",
"",
"\t\t/* 命令选择？ */",
"\t\ti = 0;",
"\t\tfor(ptr = items; ptr->desc != 0; ptr++) {",
"\t\t\tif(ptr->attribute != VISIBLE)",
"\t\t\t\tcontinue;",
"\t\t\ti++;",
"\t\t\tif(nval == i)",
"\t\t\t\tbreak;",
"\t\t\tif(!casecmp(buf, ptr->cmd))",
"\t\t\t\tbreak;",
"\t\t}",
"",
"\t\tif(!ptr->desc)",
"\t\t\tcontinue;\t/* 没有匹配 */",
"",
"\t\tswitch(ptr->action)",
```

```
"\t\t{",
"\t\tcase QUIT:",
"\t\t\treturn 0;",
"\t\tcase IGNORE:",
"\t\t\trefresh();",
"\t\t\tbreak;",
"\t\tcase EXECUTE:",
"\t\t\trefresh();",
"\t\t\tsystem(ptr->act_str);",
"\t\t\tbreak;",
"\t\tcase MENU:",
"\t\t\trefresh();",
"\t\t\t(*ptr->act_menu)();",
"\t\t\tbreak;",
"\t\tdefault:",
"\t\t\tprintw(\"default case, no action\\n\");",
"\t\t\trefresh();",
"\t\t\tbreak;",
"\t\t}",
"\t\trefresh();",
"\t}",
"}",
"",
"casecmp(char *p, char *q)",
"{",
"\tint pc, qc;",
"",
"\tfor(; *p != 0; p++, q++) {",
"\t\tpc = tolower(*p);",
"\t\tqc = tolower(*q);",
"",
"\t\tif(pc != qc)",
"\t\t\tbreak;",
"\t}",
"\treturn pc-qc;",
"}",
0
};
```

# 附录十

# SQL分析程序代码

下面列出了嵌入式SQL翻译程序的完整代码，包括词法分析程序、语法分析程序和支持C代码。因为语法分析程序很长，所以给行做了编号，并在最后通过行号对所有的符号进行了交叉引用。

**main()**和**yyerror()**例程在lex扫描程序的最后。

## yacc语法分析程序

在打印的清单中，为了适应页面有些被分成了两部分。行号与语法文件中的原始行相对应。

```
        /* 符号标记 */

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

10  %token NAME
    %token STRING
```

```
%token INTNUM APPROXNUM

    /* 操作符 */
15
%left OR
%left AND
%left NOT
%left <subtok> COMPARISON /* = <> < > <= >= */
20  %left '+' '-'
%left '*' '/'
%nonassoc UMINUS

    /* 文字关键字标记 */

25
%token ALL AMMSC ANY AS ASC AUTHORIZATION BETWEEN BY
%token CHARACTER CHECK CLOSE COMMIT CONTINUE CREATE CURRENT
%token CURSOR DECIMAL DECLARE DEFAULT DELETE DESC DISTINCT
    DOUBLE
%token ESCAPE EXISTS FETCH FLOAT FOR FOREIGN FOUND FROM GOTO
30  %token GRANT GROUP HAVING IN INDICATOR INSERT INTEGER INTO
%token IS KEY LANGUAGE LIKE NULLX NUMERIC OF ON OPEN OPTION
%token ORDER PARAMETER PRECISION PRIMARY PRIVILEGES PROCEDURE
%token PUBLIC REAL REFERENCES ROLLBACK SCHEMA SELECT SET
%token SMALLINT SOME SQLCODE SQLERROR TABLE TO UNION
35  %token UNIQUE UPDATE USER VALUES VIEW WHENEVER WHERE WITH WORK

%%

sql_list:
40          sql ';' { end_sql(); }
    |   sql_list sql ';' { end_sql(); }
    ;

45  /* 模式定义语言 */
sql:        schema
    ;

schema:
50          CREATE SCHEMA AUTHORIZATION user
    opt_schema_element_list
    ;

opt_schema_element_list:
```

```
        /* 空值 */
55    | schema_element_list
      ;

schema_element_list:
        schema_element
60    | schema_element_list schema_element
      ;

schema_element:
        base_table_def
65    | view_def
      | privilege_def
      ;

base_table_def:
70      CREATE TABLE table '(' base_table_element_commalist ')'
      ;

base_table_element_commalist:
        base_table_element
75    | base_table_element_commalist ',' base_table_element
      ;

base_table_element:
        column_def
80    | table_constraint_def
      ;

column_def:
        column data_type column_def_opt_list
85    ;

column_def_opt_list:
        /* 空值 */
      | column_def_opt_list column_def_opt
90    ;

column_def_opt:
        NOT NULLX
      | NOT NULLX UNIQUE
95    | NOT NULLX PRIMARY KEY
      | DEFAULT literal
      | DEFAULT NULLX
```

```
        |  DEFAULT USER
        |  CHECK '(' search_condition ')'
100     |  REFERENCES table
        |  REFERENCES table '(' column_commalist ')'
        ;

    table_constraint_def:
105        UNIQUE '(' column_commalist ')'
           PRIMARY KEY '(' column_commalist ')'
        |  FOREIGN KEY '(' column_commalist ')'
                REFERENCES table
        |  FOREIGN KEY '(' column_commalist ')'
110             REFERENCES table '(' column_commalist ')'
        |  CHECK '(' search_condition ')'
        ;

    column_commalist:
115        column
        |  column_commalist ',' column
        ;

    view_def:
120        CREATE VIEW table opt_column_commalist
           AS query_spec opt_with_check_option
        ;

    opt_with_check_option:
125        /* 空值 */
        |  WITH CHECK OPTION
        ;

    opt_column_commalist:
130        /* 空值 */
        |  '(' column_commalist ')'
        ;

    privilege_def:
135        GRANT privileges ON table TO grantee_commalist
           opt_with_grant_option
        ;

    opt_with_grant_option:
140        /* 空值 */
        |  WITH GRANT OPTION
```

```
        ;

    privileges:
145         ALL PRIVILEGES
        |   ALL
        |   operation_commalist
        ;

150 operation_commalist:
            operation
            operation_commalist ',' operation
        ;

155 operation:
            SELECT
            INSERT
            DELETE
            UPDATE opt_column_commalist
160     |   REFERENCES opt_column_commalist
        ;


    grantee_commalist:
165         grantee
        |   grantee_commalist ',' grantee
        ;

    grantee:
170         PUBLIC
        |   user
        ;

        /* 游标定义 */
175 sql:
            cursor_def
        ;

180 cursor_def:
            DECLARE cursor CURSOR FOR query_exp
        opt_order_by_clause
        ;

    opt_order_by_clause:
185         /* 空值 */
```

```
            |  ORDER BY ordering_spec_commalist
            ;

      ordering_spec_commalist:
190            ordering_spec
            |  ordering_spec_commalist ',' ordering_spec
            ;

      ordering_spec:
195            INTNUM opt_asc_desc
            |  column_ref opt_asc_desc
            ;

      opt_asc_desc:
200            /* 空值 */
            |  ASC
            |  DESC
            ;

205         /* 操纵语句 */

      sql:          manipulative_statement
            ;

210   manipulative_statement:
               close_statement
            |  commit_statement
            |  delete_statement_positioned
            |  delete_statement_searched
215         |  fetch_statement
            |  insert_statement
               open_statement
            |  rollback_statement
            |  select_statement
220         |  update_statement_positioned
            |  update_statement_searched
            ;

      close_statement:
225            CLOSE cursor
            ;

      commit_statement:
               COMMIT WORK
230         ;
```

```
    delete_statement_positioned:
            DELETE FROM table WHERE CURRENT OF cursor
        ;
235
    delete_statement_searched:
            DELETE FROM table opt_where_clause
        ;
240 fetch_statement:
            FETCH cursor INTO target commalist
        ;

    insert_statement:
245         INSERT INTO table opt_column_commalist
        values_or_query_spec
        ;

    values_or_query_spec:
            VALUES '(  insert_atom_commalist ')
250     |   query_spec
        ;

    insert_atom_commalist:
            insert_atom
255     |   insert_atom_commalist ',' insert_atom
        ;

    insert_atom:
            atom
260     .   NULLX
        ;

    open_statement:
            OPEN cursor
265     ;

    rollback_statement:
            ROLLBACK WORK
        ;
270
    select_statement:
            SELECT opt_all_distinct selection
            INTO target_commalist
            table_exp
275     ;
```

```
    opt_all_distinct:
            /* 空值 */
        |   ALL
280     |   DISTINCT
        ;

    update_statement_positioned:
            UPDATE table SET assignment_commalist
285         WHERE CURRENT OF cursor
        ;

    assignment_commalist:
            assignment
290         assignment_commalist ',' assignment
        ;
    assignment:
            column  =' scalar_exp
295     |   column  =' NULLX
        ;

    update statement_searched:
            UPDATE table SET assignment_commalist opt_where_clause
300     ;

    target_commalist:
            target
        |   target_commalist ',' target
305     ;

    target:
            parameter_ref
        ;
310
    opt_where_clause:
            /* 空值 */
        |   where_clause
        ;
315
        /* 查询表达式 */

    query_exp:
            query_term
320     |   query_exp UNION query_term
        |   query_exp UNION ALL query_term
```

```
        ;

    query_term:
325         query_spec
        |   '(' query_exp ')'
        ;

    query_spec:
330         SELECT opt_all_distinct selection table_exp
        ;

    selection:
            scalar_exp_commalist
335     |   '*'
        ;

    table_exp:
            from_clause
340         opt_where_clause
            opt_group_by_clause
            opt_having_clause
        ;

345 from_clause:
            FROM table_ref_commalist
        ;

    table_ref_commalist:
350         table_ref
        |   table_ref_commalist ',' table_ref
        ;

    table_ref:
355         table
        |   table range_variable
        ;

    where_clause:
360         WHERE search_condition
        ;

    opt_group_by_clause:
            /* 空值 */
365     |   GROUP BY column_ref_commalist
        ;
```

```
      column_ref_commalist:
                column_ref
370           | column_ref_commalist ',' column_ref
              ;

      opt_having_clause:
                /* 空值 */
375           | HAVING search_condition
              ;

              /* 搜索条件 */

380   search_condition:
              | search_condition OR search_condition
                search_condition AND search_condition
              | NOT search_condition
              | '(' search_condition ')
385           | predicate
              ;

      predicate:
                comparison_predicate
390           | between_predicate
              | like_predicate
              | test_for_null
              | in_predicate
              | all_or_any_predicate
395           | existence_test
              ;

      comparison_predicate:
                scalar_exp COMPARISON scalar_exp
400           | scalar_exp COMPARISON subquery
              ;

      between_predicate:
                scalar_exp NOT BETWEEN scalar_exp AND scalar_exp
405           | scalar_exp BETWEEN scalar_exp AND scalar_exp
              ;

      like_predicate:
                scalar_exp NOT LIKE atom opt_escape
410           | scalar_exp LIKE atom opt_escape
              ;
```

```
    opt_escape:
            /* 空值 */
415     |   ESCAPE atom
        ;

    test_for_null:
            column_ref IS NOT NULLX
420     |   column_ref IS NULLX
        ;

    in_predicate:
            scalar_exp NOT IN '(' subquery ')'
425     |   scalar_exp IN '(' subquery ')'
        |   scalar_exp NOT IN '(' atom_commalist ')'
        |   scalar_exp IN '(' atom_commalist ')'
        ;

430 atom_commalist:
            atom
        |   atom_commalist ',' atom
        ;

435 all_or_any_predicate:
            scalar_exp COMPARISON any_all_some subquery
        ;

    any_all_some:
440         ANY
        |   ALL
        |   SOME
        ;

445 existence_test:
            EXISTS subquery
        ;

    subquery:
450         '(' SELECT opt_all_distinct selection table_exp ')'
        ;

        /* 标量表达式 */
455 scalar_exp:
            scalar_exp '+' scalar_exp
        |   scalar_exp '-' scalar_exp
```

```
        |   scalar_exp '*' scalar_exp
        |   scalar_exp '/' scalar_exp
460     |   '+' scalar_exp %prec UMINUS
        |   '-' scalar_exp %prec UMINUS
        |   atom
        |   column_ref
        |   function_ref
465     |   '(  scalar_exp ')'
        ;

    scalar_exp_commalist:
            scalar_exp
470     |   scalar_exp_commalist ',' scalar_exp
        ;

    atom:

            parameter_ref
475     |   literal
        |   USER
        ;

    parameter_ref:
480         parameter
        |   parameter parameter
        |   parameter INDICATOR parameter
        ;

485 function_ref:
            AMMSC '(' '*  ')
        |   AMMSC  (' DISTINCT column_ref ')'
        |   AMMSC  (' ALL scalar_exp ')'
        |   AMMSC '(' scalar_exp  )'
490     ;

    literal:
            STRING
        |   INTNUM
495     |   APPROXNUM
        ;

        /* 其他 */
500 table:
            NAME
```

```
        | NAME '.' NAME
        ;

505 column_ref:
            NAME
        | NAME '.' NAME /* 需要语义解释 */
        | NAME '.' NAME '.' NAME
        ;

510
            /* 数据类型 */

    data_type:
            CHARACTER
515     | CHARACTER '(' INTNUM ')'
        | NUMERIC
        | NUMERIC '(' INTNUM ')'
        | NUMERIC '(' INTNUM ',' INTNUM ')'
        | DECIMAL
520     | DECIMAL '(' INTNUM ')'
        | DECIMAL '(' INTNUM ',' INTNUM ')'
        | INTEGER
        | SMALLINT
        | FLOAT
525     | FLOAT '(' INTNUM ')'
        | REAL
        | DOUBLE PRECISION
        ;

530     /* 可以命名的不同东西 */

    column: NAME
        ;

535 cursor: NAME
        ;

    parameter:
            PARAMETER /* :语法分析程序中处理的名字 */
540     ;

    range_variable: NAME
        ;
```

```
545 user: NAME
        ;

        /* 嵌入式条件 */
    sql:        WHENEVER NOT FOUND when_action
550     |  WHENEVER SQLERROR when_action
        ;

    when_action: GOTO NAME
        |  CONTINUE
555     ;
    %%
```

## 交叉引用

因为该语法分析程序很长，所以下面提供了通过行号对所有符号进行的交叉引用。对于每个符号，符号被定义在规则左侧的行用黑体字。

### A


ALL (**26**)
145, 146, 279, 321, 441, 488

all_or_any_predicate (**435**)
394

AMMSC (**26**)
486, 487, 488, 489

AND (**17**)
382, 404, 405

ANY (**26**)
440

any_all_some (**439**)
436

APPROXNUM (**12**)
495

AS (**26**)
121

ASC (**26**)
201

assignment (**293**)
289, 290

assignment_commalist (**288**)
284, 290, 299

atom (**473**)
259, 409, 410, 415, 431, 432, 462

atom_commalist (**430**)
426, 427, 432

AUTHORIZATION (**26**)
50

B

base_table_def (**69**)
64
base_table_element (**78**)
74, 75
base_table_element_commalist (**73**)
70, 75
BETWEEN (**26**)
404, 405
between_predicate (**403**)
390
BY (**26**)
186, 365

C

CHARACTER (**27**)
514, 515
CHECK (**27**)
99, 111, 126
CLOSE (**27**)
225
close_statement (**224**)
211
column (**532**)
84, 115, 116, 294, 295
column_commalist (**114**)
101, 105, 106, 107, 109, 110, 116, 131
column_def (**83**)
79
column_def_opt (**92**)
89
column_def_opt_list (**87**)
84, 89
column_ref (**505**)
196, 369, 370, 419, 420, 463, 487
column_ref_commalist (**368**)
365, 370
COMMIT (**27**)
229
commit_statement (**228**)
212
COMPARISON (**19**)
399, 400, 436
comparison_predicate (**398**)
389
CONTINUE (**27**)
554
CREATE (**27**)
50, 70, 120
CURRENT (**27**)
233, 285
CURSOR (**28**)
181
cursor (**535**)
181, 225, 233. 241, 264, 285

cursor_def (**180**)
176

D

data_type (**513**)
84

DECIMAL (**28**)
519, 520, 521

DECLARE (**28**)
181

DEFAULT (**28**)
96, 97, 98

DELETE (**28**)
158, 233, 237

delete_statement_positioned (**232**)
213

delete_statement_searched (**236**)
214

DESC (**28**)
202

DISTINCT (**28**)
280, 487

DOUBLE (**28**)
527

E

ESCAPE (**29**)
415

existence_test (**445**)
395

EXISTS (**29**)
446

F

FETCH (**29**)
241

fetch_statement (**240**)
215

FLOAT (**29**)
524, 525

FOR (**29**)
181

FOREIGN (**29**)
107, 109

FOUND (**29**)
549

FROM (**29**)
233, 237, 346

from_clause (**345**)
339

function_ref (**485**)
464

G

GOTO (**29**)
553

GRANT (**30**)
135, 141

grantee (**169**)
165, 166

grantee_commalist (**164**)
135, 166

GROUP (**30**)
365

H

HAVING (30)
375

I

IN (**30**)
424, 425, 426, 427

INDICATOR (**30**)
482

INSERT (**30**)
157, 245

insert_atom (**258**)
254, 255

insert_atom_commalist (**253**)
249, 255

insert_statement (**244**)
216

INTEGER (**30**)
522

INTNUM (**12**)
195, 494, 515, 517, 518, 520, 521, 525

INTO (**30**)
241, 245, 273

in_predicate (**423**)
393

IS (**31**)
419, 420

K

KEY (**31**)
95, 106, 107, 109

L

LANGUAGE (**31**)

LIKE (**31**)
409, 410

like_predicate (**408**)
391

literal (**492**)
96, 475

M

manipulative_statement (**210**)
207

N

NAME (**10**)
501,502, 506, 507, 508, 532, 535, 542, 545, 553

NOT (**18**)
93, 94, 95, 383, 404, 409, 419, 424, 426, 549

NULLX (**31**)
93, 94, 95, 97, 260, 295, 419, 420

NUMERIC (**31**)
516, 517, 518

O

OF (**31**)
233, 285

ON (**31**)
135

OPEN (**31**)
264

open_statement (**263**)
217

operation (**155**)
151, 152

operation_commalist (**150**)
147, 152

OPTION (**31**)
126, 141

opt_all_distinct (**277**)
272, 330, 450

opt_asc_desc (**199**)
195, 196

opt_column_commalist (**129**)
120, 159, 160, 245

opt_escape (**413**)
409, 410

opt_group_by_clause (**363**)
341

opt_having_clause (**373**)
342

opt_order_by_clause (**184**)
181

opt_schema_element_list (**53**)
50

opt_where_clause (**311**)
237, 299, 340

opt_with_check option (**124**)
121

opt_with_grant_option (**139**)
136

OR (**16**)
381

ORDER (**32**)
186

ordering_spec (**194**)
190, 191

ordering_spec_commalist (**189**)
186, 191

P

PARAMETER (**32**)
539

parameter (**538**)
480, 481, 482

parameter_ref (**479**)
308, 474

PRECISION (**32**)
527

predicate (**388**)
385

PRIMARY (**32**)
95, 106

PRIVILEGES (**32**)
145

privileges (**144**)
135

privilege_def (**134**)
66

PROCEDURE (**32**)

PUBLIC (**33**)
170

Q

query_exp (**318**)
181, 320, 321, 326

query_spec (**329**)
121, 250, 325

query_term (**324**)
319, 320, 321

R

range_variable (**542**)
356

REAL (**33**)
526

REFERENCES (**33**)
100, 101, 108, 110, 160

ROLLBACK (**33**)
268

rollback_statement (**267**)
218

S

scalar_exp (**455**)
294, 399, 400, 404, 405, 409, 410, 424, 425, 426, 427, 436, 456, 457, 458, 459, 460, 461, 465, 469, 470, 488, 489

scalar_exp_commalist (**468**)
334, 470

SCHEMA (**33**)
50

schema (**49**)
46

schema_element (**63**)
59, 60

schema_element_list (**58**)
55, 60

search_condition (**380**)
99, 111, 360, 375, 381, 382, 383, 384

SELECT (**33**)
156, 272, 330, 450

selection (**333**)
272, 330, 450

select_statement (**271**)
219

SET (**33**)
284, 299

SMALLINT (**34**)
523

SOME (**34**)
442

sql (**46, 175, 207, 549**)
40, 41

SQLCODE (**34**)

SQLERROR (**34**)
550

sql_list (**39**)
41

STRING (**11**)
493

subquery (**449**)
400, 424, 425, 436, 446

T

TABLE (**34**)
70

table (**500**)
70, 100, 101, 108, 110, 120, 135, 233, 237, 245, 284, 299, 355, 356

table_constraint_def (**104**)
80

table_exp (**338**)
274, 330, 450

table_ref (**354**)
350, 351

table_ref_commalist (**349**)
346, 351

target (**307**)
303, 304

target_commalist (**302**)
241, 273, 304

test_for_null (**418**)
392

TO (**34**)
135

U

UMINUS (**22**)
460, 461

UNION (**34**)
320, 321

UNIQUE (**35**)
94, 105

UPDATE (**35**)
159, 284, 299

update_statement_positioned (**283**)
220

update_statement_searched (**298**)
221

USER (**35**)
98, 476

user (**545**)
50, 171

V

VALUES (**35**)
249

values_or_query_spec (**248**)
245

VIEW (**35**)
120

view_def (**119**)
65

W

WHENEVER (**35**)
549, 550

when_action (**553**)
549, 550

WHERE (**35**)
233, 285, 360

where_clause (**359**)
313

WITH (**35**)
126, 141

WORK (**35**)
229, 268

# lex 扫描程序

下面是文件 *scn2.l*:

```
%{
#include "sql2.h"
#include <string.h>

int lineno = 1;
void yyerror(char *s);

        /* 保存SQL标记文本的宏 */
#define SV save_str(yytext)

        /* 保存文本并返回标记的宏 */
#define TOK(name) { SV;return name; }
%}
```

```
%s SQL
%%

EXEC[ \t]+SQL { BEGIN SQL; start_save(); }

        /* 文字关键字标记 */

<SQL>ALL                TOK(ALL)
<SQL>AND                TOK(AND)
<SQL>AVG                TOK(AMMSC)
<SQL>MIN                TOK(AMMSC)
<SQL>MAX                TOK(AMMSC)
<SQL>SUM                TOK(AMMSC)
<SQL>COUNT              TOK(AMMSC)
<SQL>ANY                TOK(ANY)
<SQL>AS                         TOK(AS)
<SQL>ASC                TOK(ASC)
<SQL>AUTHORIZATION              TOK(AUTHORIZATION)
<SQL>BETWEEN                    TOK(BETWEEN)
<SQL>BY                         TOK(BY)
<SQL>CHAR(ACTER)?       TOK(CHARACTER)
<SQL>CHECK              TOK(CHECK)
<SQL>CLOSE              TOK(CLOSE)
<SQL>COMMIT             TOK(COMMIT)
<SQL>CONTINUE                   TOK(CONTINUE)
<SQL>CREATE             TOK(CREATE)
<SQL>CURRENT                    TOK(CURRENT)
<SQL>CURSOR             TOK(CURSOR)
<SQL>DECIMAL                    TOK(DECIMAL)
<SQL>DECLARE                    TOK(DECLARE)
<SQL>DEFAULT                    TOK(DEFAULT)
<SQL>DELETE             TOK(DELETE)
<SQL>DESC               TOK(DESC)
<SQL>DISTINCT                   TOK(DISTINCT)
<SQL>DOUBLE             TOK(DOUBLE)
<SQL>ESCAPE             TOK(ESCAPE)
<SQL>EXISTS             TOK(EXISTS)
<SQL>FETCH              TOK(FETCH)
<SQL>FLOAT              TOK(FLOAT)
<SQL>FOR                TOK(FOR)
<SQL>FOREIGN                    TOK(FOREIGN)
<SQL>FOUND              TOK(FOUND)
<SQL>FROM               TOK(FROM)
<SQL>GO[ \t]*TO                 TOK(GOTO)
```

```
<SQL>GRANT          TOK(GRANT)
<SQL>GROUP          TOK(GROUP)
<SQL>HAVING         TOK(HAVING)
<SQL>IN                  TOK(IN)
<SQL>INDICATOR           TOK(INDICATOR)
<SQL>INSERT         TOK(INSERT)
<SQL>INT(EGER)?          TOK(INTEGER)
<SQL>INTO           TOK(INTO)
<SQL>IS                  TOK(IS)
<SQL>KEY            TOK(KEY)
<SQL>LANGUAGE            TOK(LANGUAGE)
<SQL>LIKE           TOK(LIKE)
<SQL>NOT            TOK(NOT)
<SQL>NULL           TOK(NULLX)
<SQL>NUMERIC             TOK(NUMERIC)
<SQL>OF                  TOK(OF)
<SQL>ON                  TOK(ON)
<SQL>OPEN           TOK(OPEN)
<SQL>OPTION         TOK(OPTION)
<SQL>OR                  TOK(OR)
<SQL>ORDER          TOK(ORDER)
<SQL>PRECISION           TOK(PRECISION)
<SQL>PRIMARY             TOK(PRIMARY)
<SQL>PRIVILEGES          TOK(PRIVILEGES)
<SQL>PROCEDURE           TOK(PROCEDURE)
<SQL>PUBLIC         TOK(PUBLIC)
<SQL>REAL           TOK(REAL)
<SQL>REFERENCES          TOK(REFERENCES)
<SQL>ROLLBACK            TOK(ROLLBACK)
<SQL>SCHEMA         TOK(SCHEMA)
<SQL>SELECT         TOK(SELECT)
<SQL>SET            TOK(SET)
<SQL>SMALLINT            TOK(SMALLINT)
<SQL>SOME           TOK(SOME)
<SQL>SQLCODE             TOK(SQLCODE)
<SQL>SQLERROR       TOK(SQLERROR)
<SQL>TABLE          TOK(TABLE)
<SQL>TO                  TOK(TO)
<SQL>UNION          TOK(UNION)
<SQL>UNIQUE         TOK(UNIQUE)
<SQL>UPDATE         TOK(UPDATE)
<SQL>USER           TOK(USER)
<SQL>VALUES         TOK(VALUES)
<SQL>VIEW           TOK(VIEW)
```

```
<SQL>WHENEVER            TOK(WHENEVER)
<SQL>WHERE          TOK(WHERE)
<SQL>WITH           TOK(WITH)
<SQL>WORK           TOK(WORK)

        /* 标点符号 */

<SQL>"="    |
<SQL>"<>"   |
<SQL>"<"    |
<SQL>">"    |
<SQL>"<="   |
<SQL>">="           TOK(COMPARISON)

<SQL>[-+*/(),.;]    TOK(yytext[0])


        /* 名字 */
<SQL>[A-Za-z][A-Za-z0-9_]*      TOK(NAME)

        /* 参数 */
<SQL>":"[A-Za-z][A-Za-z0-9_]*{
                    save_param(yytext+1);
                    return PARAMETER;
                }

        /* 数字 */

<SQL>[0-9]+ |
<SQL>[0-9]+"."[0-9]* |
<SQL>"."[0-9]*      TOK(INTNUM)

<SQL>[0-9]+[eE][+-]?[0-9]+ |
<SQL>[0-9]+"."[0-9]*[eE][+-]?[0-9]+ |
<SQL>"."[0-9]*[eE][+-]?[0-9]+TOK(APPROXNUM)

        /* 字符串 */

<SQL>'[^'\n]*'  {
                int c = input();

                unput(c);    /* 仅仅查看一下 */
                if(c != '\'') {
```

```
                    SV;return STRING;
                } else
                    yymore();
        }

<SQL>'[^'\n]*$      {       yyerror("Unterminated string"); }

<SQL>\n { save_str(" ");lineno++; }
\n { lineno++; ECHO; }

<SQL>[ \t\r]+   save_str(" ");   /* 空白 */
<SQL>"--".* ;          /* 注释 */

.               ECHO; /* 随机的非SQL文本 */
%%

void
yyerror(char *s)
{
        printf("%d: %s at %s\n", lineno, s, yytext);
}

main(int ac, char **av)
{
        if(ac > 1 && (yyin = fopen(av[1], "r")) == NULL) {
              perror(av[1]);
              exit(1);
        }

        if(!yyparse())
              fprintf(stderr, "Embedded SQL parse worked\n");
        else
              fprintf(stderr, "Embedded SQL parse failed\n");
} /* 主程序 */

/* 退出SQL词法分析模式 */
un_sql()
{
        BEGIN INITIAL;
} /* un_sql */
```

# 支持代码

下面是文件 *sqltext.c*:

```
/*
 * 简单的嵌入式SQL的文本处理程序
 */

#include <stdio.h>
#include <string.h>
extern FILE *yyout;         /* lex输出文件 */

char save_buf[2000];        /* SQL命令的缓冲区 */
char *savebp;               /* 当前缓冲区的指针 */

#define NPARAM 20           /* 每个函数的最多参数 */
char *varnames[NPARAM];     /* 参数名 */

/* 在EXEC SQL之后开始嵌入式命令 */
start_save(void)
{
        savebp = save_buf;
} /* start_save */

/* 保存SQL标记 */
save_str(char *s)
{

        strcpy(savebp, s);
        savebp += strlen(s);
} /* save_str */

/* 保存参数引用 */
save_param(char *n)
{
        int i;
        char pbuf[10];

        /* 在表中寻找变量名 */

        for(i = 1; i < NPARAM; i++) {
              if(!varnames[i]) {
```

```
                /* 没找到，输入它 */
                varnames[i] = strdup(n);
                break;
            }

            if(!strcmp(varnames[i],n))
                break;      /* 已经存在 */
        }

        if(i >= NPARAM) {
            yyerror("Too many parameter references");
            exit(1);
        }

        /* 通过变量数字保存 #n 引用 */
        sprintf(pbuf, " #%d", i);
        save_str(pbuf);

} /* save_param *

/* 结束 SQL 命令，现在将它写出 */
end_sql(void)
{
        int i;
        register char *cp;

        savebp--; /* 结束闭引号返回 */

        /* 调用 exec_sql 函数 */

        fprintf(yyout, "exec_sql(\"");

        /* 将保存的缓冲区作为大的 C 字符串写出
         * 需要时开始换行
         */

        for(cp = save_buf, i = 20; cp < savebp; cp++, i++) {
            if(i > 70) { /* 需要换行 */
                fprintf(yyout, "\\\n");
                i = 0;
            }
            putc(*cp, yyout);
        }
```

```
        putc( "', yyout);

        /* 传递每个被引用的变量的地址 */
        for(i = 1; i < NPARAM; i++) {
                if(!varnames[i])
                        break;
                fprintf(yyout, ",\n\t&%s", varnames[i]);
                free(varnames[i]);
                varnames[i] = 0;
        }

        fprintf(yyout, "); n");

        /* 将扫描程序返回到常规模式 */
        un_sql();

} /* end_sql */
```

# 参考文献

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

经典的编译程序教材。它包括详细的yacc和lex理论及其实现的概述。

American National Standards Institute. *Programming Language C*. X3.159-1989. ANSI, December 1989.

现代ANSI C的定义。也是著名的（美国）联邦信息处理标准（FIPS）160。

Bennett, J.P. *Introduction to Compiling Techniques —A First Course Using Ansi C, Lex and Yacc*. McGraw Hill Book Co, 1990.

Deloria. "Practical yacc: a gentle introduction to the power of this famous parser generator." *C Users Journal*. Nov 1987, Dec/Jan 1988, Mar/Apr 1988, Jun/Jul 1988, and Sep/Oct 1988.

Donnely and Stallman. *The Bison Manual*. Part of the online bison distribution.

有关bison的权威性参考。

Holub, Alan. *Compiler Design in C*. Prentice-Hall, 1990.

一本包含了yacc和lex各版本的完整的源代码以及用它们建立的C编译程序源代码的巨著。

S. C. Johnson *Yacc — Yet Another Compiler-Compiler*. Comp. Sci. Tech. Rep. No. 32. Bell Laboratories, July 1975.

yacc的最初描述。作为第7版UNIX以及BSD UNIX大多数版本的文档的部分被重印。

Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

“经典” C语言的标准参考。

M. E. Lesk *Lex — A Lexical Analyzer Generator*. Comp. Sci. Tech. Rep. No. 39. Bell Laboratories, October 1975.

lex的最初描述。作为第7版UNIX以及大多数BSD UNIX版本的文档的一部分被重印。

Schreiner, Axel T. and H. George Friedman, Jr. *Introduction to Compiler Construction with UNIX*. Prentice-Hall, 1985.

使用lex和yacc开发一个小的C子集的编译程序，使用相当理论性的方法，并有错误处理和恢复的极好介绍。在示例中小心印刷上的错误。

# 词汇表

在这本手册中使用了大量的技术术语。它们中有一些是人们所熟悉的，但有一些可能是不熟悉的。为了避免混淆，将最重要的术语列在了下面。

action（动作）

与lex模式或yacc规则有关的C代码。当模式或规则与输入序列匹配时，执行动作代码。

alphabet（字母表）

不同符号的集合。例如，ASCII字符集是128个不同符号的集合。在lex规范中，字母表是计算机的本地字符集，除非使用“%T”来定义定制的字母表。在yacc语法中，字母表是语法中使用的标记和非终结符的集合。

ambiguity（歧义）

有歧义的语法指的是具有匹配同一个输入的多条规则或规则集合的语法。在yacc语法中，有歧义的规则导致移进/归约冲突或归约/归约冲突。yacc使用的分析机制不能处理有歧义的语法，所以当创建语法分析程序时，它使用%**prec**声明以及它自己的内部规则来解决这个冲突。

lex规范可以是（并且通常是）有歧义的；当两个模式匹配同一个输入时，规范中的较前面的模式取胜。

ASCII（American Standard Code for Information Interchange，美国信息交换标准代码）

代表在美国字母表中找到的普通符号的128个符号的集合：包括小写和大写字母、数字和标点符号，另外还包括用于格式化和数据通信链路控制的其他字符。运行

yacc和lex的大多数计算机都使用ASCII，尽管一些IBM大型机系统使用不同的称为EBCDIC的256符号代码。

**BNF (Backus-Naur Form，巴克斯－诺尔范式)**

表示语法的一种方法。常用于指定程序设计语言的正式语法。yacc的输入语法是BNF的简化版本。

**BSD (Berkeley Software Distribution，伯克利软件发行中心)**

伯克利的加利福尼亚大学在Seventh Edition UNIX（UNIX第7版）的基础上发行了一系列操作系统版本；通常，BSD用特殊分配的版本号所指定，例如，BSD 2.10或BSD 4.3。

**compiler (编译程序)**

它是将某种语言的一套指令（一个程序）转换成一些其他的表示方法的程序；通常，编译程序的输出采用直接在计算机上运行本地二进制语言的形式。与“解释程序”（interpreter）进行比较。

**conflict (冲突)**

冲突是指分析同一输入标记时，yacc语法中的两个（或多个）分析动作都是可能的。有两种类型的冲突：移进/归约和归约/归约。（参见ambiguity（歧义））

**empty string (空字符串)**

具有零个符号的特殊情况下的字符串，有时写成ε。在C语言中，空字符串即仅仅由ASCII字符NUL组成的字符串。yacc规则可以匹配空字符串，但是lex模式不能。

**finite automaton (有限自动机)**

由有限数目的指令（或转换）组成的抽象机。有限自动机在建模许多普通的“发生计算机进程”方面很有用，并且有有用的数学性质。lex和yacc创建基于有限自动机的词法分析程序和语法分析程序。

**input (输入)**

由程序读取的数据流。例如，lex扫描程序的输入是字节序列，而yacc语法分析程序的输入是标记序列。

**interpreter (解释程序)**

读取语言（程序）中指令并对它们进行一次一条的解码和操作的程序。与“编译程序”（compiler）进行比较。

**language (语言)**

形式上说，是基于一些字母表的定义良好的字符串集合；非形式上讲，是描述计算机能够执行的任务的一些指令的集合。

LALR (1) (LookAhead Left Recursive，向前查看左递归)

yacc 使用的分析技术。其中的（1）表示向前查看限制为一个标记。

left-hand side (LHS，左侧)

yacc规则的左侧(LHS)是冒号前面的符号。在分析过程中、当输入匹配规则的RHS序列符号时，那个序列被归约为 LHS 符号。

lex

产生词法分析器的程序，词法分析程序根据正则表达式所定义的内容匹配字符流。

lexical analyzer (词法分析器)

将字符流转化成标记流的程序。lex 将单个标记以正则表达式的形式来描述，将字符流拆分成标记并决定标记的类型和数值。例如，它也许将字符流“a = 17;”转换成由名字“a”、操作符“=”、数字“17”和单个字符标记“;”组成的标记流。词法分析器也称为词法分析程序（lexer）或扫描程序。

lookahead (向前查看)

由语法分析程序或扫描程序读取的尚未匹配模式或者规则的输入。yacc语法分析程序向前查看单个标记，而 lex 扫描程序有不限长度地向前查看。

non-terminal (非终结符)

不出现在输入中的yacc语法中的符号，而是由规则定义。与“标记”(token)进行对比。

parser stack (分析程序堆栈)

在yacc语法分析程序中，部分匹配规则的符号存储在内部堆栈中。当语法分析程序移进时，符号被添加到堆栈中；而当它被归约时，符号从堆栈中被删除。

parsing (语法分析)

取得标记流并且在某些语言中逻辑地将它们归组到语句中的过程。

pattern (模式)

在 lex 词法分析程序中，词法分析程序匹配输入的正则表达式。

precedence (优先级)

一些特殊的操作被执行的顺序；例如，当解释数学语句时、乘法和除法被赋予了比加法和减法更高的优先级，如语句“3+4*5”的执行结果是 23 而不是 35。

production(产生式)

参见“规则”(rule)。

program (程序)

执行某种预定义任务的指令集合。

reduce（归约）

在yacc语法分析程序中，当输入匹配规则的RHS上的符号列表时，语法分析程序通过从语法分析程序堆栈中删除RHS符号并用LHS符号取代它们来简化这条规则。

reduce/reduce conflict（归约/归约冲突）

在yacc语法中，两个或多个规则匹配同一标记串的情况。yacc通过归约语法中前面出现的规则来解决这个冲突。

regular expression（正则表达式）

指定匹配字符序列的模式的语言。正则表达式由标准字符（标准字符匹配输入中相同的字符）、字符类（匹配类中的任意单个字符）和其他字符（它指定部分表达式匹配输入的方式）组成。

right-hand side（RHS，右侧）

yacc规则的右侧（RHS）是跟在冒号后面的符号列表。在分析过程中，当输入匹配规则的RHS符号序列时，序列被归约为LHS符号。

rule（规则）

在yacc中，规则是语法的抽象描述。yacc规则也称为产生式。规则是称为LHS的单个非终结符、冒号和可能为空的称为RHS的符号集。每当输入匹配规则的RHS时，语法分析程序就归约规则。

semantic meaning（语义意义）

参见“值”（value）。

shift（移进）

yacc语法分析程序将输入符号移进到分析程序堆栈上，以使符号匹配语法中的一条规则。

shift/reduce conflict（移进/归约冲突）

在yacc语法中，在符号完成一条规则的RHS的情况下，语法分析程序需要归约，并且如果这个符号是其他规则的RHS中的中间符号时，语法分析程序需要移进这个符号。出现移进/归约冲突是因为语法是有歧义的，或者是因为语法分析程序在决定是否归约符号完成的这条规则之前需要向前查看更多的标记。yacc通过移进来解决这个冲突。

specification（规范）

lex规范是匹配输入流的模式集合。lex将规范转换成词法分析程序。

start state（起始状态）

在lex规范中，模式可以用起始状态标记。

**start symbol（起始符号）**

yacc语法分析程序把有效输入流所归约到的单个符号。LHS为起始符号的规则称为起始规则。

**symbol table（符号表）**

包含了与输入中出现的名字有关的信息的表，因此对同一名字的所有引用都涉及同一对象。

**symbol（符号）**

在yacc术语中，符号可以是标记或非终结符。在语法规则中，规则右侧发现的任何名字总是一个符号。

**System V（系统V）**

在UNIX第7版（基于UNIX的BSD版本）之后，AT&T又发布了UNIX的新版本，最新发布的称为系统V；较新的版本都带有版本号，所以通常指的是系统V或系统V.4。

**token（标记）**

在yacc术语中，标记或终结符是由词法分析程序提供给语法分析程序的符号。比较在语法分析程序中定义的“非终结符”（non-terminal）。

**tokenizing（标记化）**

将字符流转变为标记流的过程称为标记化。词法分析程序标记化它的输出。

**value（值）**

yacc语法中的每个标记都有一个语法值和一个语义值；它的语义值是标记的实际数据内容。例如，某个操作的语法类型可能是INTEGER，但是它的语义值也许为3。

**yacc（Yet Another Compiler Compiler，另一种编译程序的编译程序）**

用类似BNF的格式从规则列表中生成语法分析程序的程序。