GNU Bison 中文手册翻译完成

GNU Bison 实际上是使用最广泛的 Yacc-like 分析器生成器,使用它可以生成解释器,编译器,协议实现等多种程序. 它不但与 Yacc 兼容还具有许多 Yacc 不具备的特性.

这个手册编写十分完整,带你领略 Bison 在使用中的各个细节(注：并不是实现细节).

如果发现错误,语句不通顺,意思不明,确请立即发邮件把您的建议或者您认为正确的翻译 写信告诉我,非常需要并感谢你的帮助!

英文原件页面

这个翻译版手册也是在 GNU Free Documentation License 下发布.

提供以下格式:(会陆续增加格式)

- 原始 texinfo 格式(78,999 字节,gzip 压缩)
- HTML 格式(111,882 字节,gzip 压缩)
- HTML 格式(109,570 字节,zip 压缩)

HTML 格式使用一个由我 patch 的 texi2html 生成,如果需要的话可以跟我联系.

一些参考:

- GNU Bison Project
- The LEX & YACC Page
- GNU Flex Project
- Yacc: 另一个编译器的编译器 Stephen C. Johnson 著,寒蝉退 士译

Bison

这个手册是针对 GNU Bison (版本 2.0,22 December 2004), GNU 分析器生成器.

Chinese(zh_CN) translation:Xiao Wang

Bison 简介-Introduction

Bison 是一种通用目的的分析器生成器. 它将 LALR(1)上下文无关文法的描述转化成分析该文法的 C 程序. 一旦你精通 Bison, 你可以用它生成从简单的桌面计算器到复杂的程序设计语言等等许多语言的分析器.

Bison 向上兼容 Yacc:所有书写正确的 Yacc 语法都应该可以不加更改地与 Bison 一起工作. 熟悉 Yacc 的人能毫不费力地使用 Bison. 你应该熟练地掌握 C 程序设计语言,这样你才能使用 Bison 和理解这个手册.

我们会在这个教程的最初几章解释 BIson 的基本概念,并且展示三个详细解释的例子, 这些例子将在教程的最后被构建. 如果你对 Bison 或者 Yacc 一无所知, 你应该首先阅读这些章节. 接下来的参阅章节详细地阐述了 Bison 的各个方面.

Bison 主要由 Rovert Corbett 编写.Richard Stallman 使它与 Yacc 兼容. Carnegie Mellon 大学的 Wilfred Hansen 为 Bison 添加了多字符字符串文字(multi-character string literals)和其它一些特性.

这个版本的手则主要与 Bison2.0 相一致.

使用 Bison 的条件-Conditions for Using Bison

为了允许非自由程序(nonfree programs)使用 Bison 生成的 LALR 分析器 C 代码, 我们在 Bison 版本 1.24 的时已经修改了 yyparse 的发布条款(distribution terms for yyparse). 在这之前,这些分析器只能用于自由软件(free software)的程序.

其它 GNU 编程工具,例如 GNU C 编译器从未有过类似的要求. 它们总能用于非自由软件的发布. Bison 与它们不同的原因并不是出于特殊策略的决定, 而是由于应用通用许可证(General Public License)到所有的 Bison 源代码造成的结果.

Bison 工具的输出-也就是 Bison 分析器文件(the Bison parser file)包括大小不固定的 Bison 代码片段. 这些代码片段来源于 yyparse 函数.(你的语法的动作被 Bison 插入到这个函数的某个位置, 但是函数的其余部分并未更改).当我们应用 GPL 条款到 yyparse 的代码的时候, 它产生的效果就是 Bison 的输出只能到自由软件.

我们并未更改条款使出于对那些希望是软件私有化的人的同情. **所有的软件都应该是自由软件.**但是我们的结论是: 使 Bison 只能用于自由软件,这对鼓励人们使其它软件变为自由软件作用甚微. 所以我们决定将使用 Bison 的条件与使用其它 GNU 工具的条件相一致. (注:即和 GCC 一样可以用于非自由软件).

这个特例仅仅在 Bison 生成 LALR(1)分析器的 C 代码时适用. 否则,GPL 条款仍然正常的执行. 你可以通过查看代码,看其是否由这样的说明"As a special execption, when this file is copied by Bison into a Bison output file, you may use that output file without restriction."来分辩这个特例是否适用于你的`.c'输出文件.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if

its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
   1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.
   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.
   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.
4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

1. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies

directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.
12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND,

EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.
To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

one line to give the program's name and a brief idea of what it does.
Copyright (C) *yyyy name of author*

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19*yy name of author*

Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.

This is free software, and you are welcome to redistribute it

under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program

`Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989

Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

1. 和 Bison 相关的一些基本概念-The Concepts of Bison

这一章介绍了许多基本概念,但没有提及一些感觉不到的细节. 如果你并不了解如何使用 Bison/Yacc,我们建议你仔细阅读这一章.

1.1 语言与上下文无关文法-Languages and Context-Free Grammars

为了使 Bison 能分析语言,这种语言必须由*上下文无关文法(context-free grammar)*描述. 也就是说,你必须指出一个或者多个*语法组(syntactic groupings)*以及从语法组的部分构它的建整体的规则. 例如,在 C 语言中,有一种我们称之为`表达式(expression)'的语法组. 一个生成表达式的规则可能是"一个表达式由一个减号和另一个表达式构成". 另外一个规则可能是"一个表达式可以是一个整数". 就象你看到的一样,规则经常是递归定义的,但是必须有一个结束递归的规则.

用于表示这些规则的最普遍的系统是 *Backus-Naur 范式(Backus-Naur Form)*或者"BNF". 发明这种语言的目的是用来阐述 Algol 60. 任何用 BNF 表示的文法都是一种上下文无关文法. Bison 要求它的的输入必须用 BNF 表示.

上下文无关文法有许多重要的子集.尽管 Bison 可以处理几乎所有的上下文无关文法, 但 Bison 针对 LALR(1) 文法(LALR(1) grammars)做了优化. 简而言之,在这些文法中(注:指 LALR(1)),我们可以告之如何分析仅代有一个超前扫描记号的输入字符串

的任意部分. 严格的说,这是一个 LR(1)文法的描述. LALR(1)包括了与多难以分析的额外限制. 幸运的是,在实际中,我们很难找到一个是 LR(1)文法而不是 LALR(1)文法的例子. 参阅神秘的归约/归约冲突-Mysterious Reduce/Reduce Conflicts,以获取更多信息.

LALR(1)文法分析器具有*确定性(deterministic)*, 这就意味着应用于输入的下一个文法规则取决于之前的输入和确定的部分剩余输入(我们称之为一个*超前扫描记号(look-ahead)*. 一个上下文无关文法可能是有*歧义的(ambiguous)*,即可能可以应用多种规则来获取某些输入. 即使非歧义性文法也可能使*不确定(non-deterministic)*的, 即没有总能足以决定下一个应用的文法规则的确定的超前扫描记号。 使用孰知的 GLR 技术,Bison 的 GLR 就可以分析这些更为普通的上下文无关文法. 当任意给定字符串的可能的分析是确定的情况下,Bison 可以处理任意上下文无关文法.

在正式的语言语法规则中,每一种语法单元或组合被称之为*符号(symbol)*. 那些可以通过语法规则被分解成更小的结构的符号叫做*非终结符(nonterminal symbols)*. 那些不能被再分的符号叫做*终结符(terminal symbols)*或者*记号类型(token types)*. 我们把同终结符相对应的输入片段叫做*记号(token)*, 把同单个非终结符相对应的输入片段叫做*组(grouping)*.

我们可以使用 C 语言做为例子来解释什么是符号,以及终结符和非终结符的含义. C 语言的记号包括标识符,常量(数字或者字符串)以及各种关键字,数学操作符和标点符号. 所以 C 语言语法的终结符包括`identifier',`number',`string',加上每个关键字的符号,操作符或者标点符号. 例如:`if',`return',`const',`static',`int',`char',`plus-sign',`open-brace',`close-brace',`comma'以及更多. (这些记号可以再分为字符,但是这些是词法学而不是语法学的事情)

这是一个如何将 C 函数分解成记号的例子:

```
int           /* 关键字 `int' */
square (int x)  /* identifier, open-paren, 关键字 `int', identifier, close-paren */
{             /* open-brace */
  return x * x; /* 关键字 `return', identifier, asterisk, identifier, semicolon */
}             /* close-brace */
```

C 语言的语法组包括表达式,语句,声明和函数定义. 这些由 C 语言语法中的非终结符`expression',`statement',`declaration'和`funcation definition'表示. 完整的语法还使用了许多额外的语言结构,每种结构都有自己的非终结符来表示上述四种的含义. 上面的例子是一个函数的定义,它包括了一个声明和一个语句. 每一个`x'一个表达式,而且`x * X'也是一个表达式.

每一个非终结符必须有一个描述如何由更简单结构组成这个非终结符的语法规则. 例如,一种 C 的语句是 return 语句的非正式表达将由如下的语法规则描述:

> 一种语句可以由一个`return'关键字,一个`expression'何以个`semicolon'组成.

还有许多其它对应`statement'的规则,每一种规则对应一种 C 语句.

我们必须注意到一种特殊的非终结符,这种非终结符定义了语言的一个完整表述. 我们称之为*开始符号(start symbol)*. 在编译器中,这意味着一个完整的输入程序. 在 C 语言中,非终结符`sequence of definitions and declarations'扮演了这个角色.

例如,`1+2'是以个有效的 C 表达式--一个有效的 C 程序的部分--但它不能做为一个 C 程序的*全部(entire)*. 在 C 语言的上下文无关文法中,这遵循了`expression'不是开始符号的事实.

Bison 分析器读取一个记号序列做为它的输入并使用语法规则将记号组合. 如果输入是有效的,最终的结果是将整个的输入序列分析整理到开始符号. 如果我们使用 C 语言的语法,整个的输入必须是一个`sequence of definitions and declarations', 否则分析器会报告一个语法错误.

1.2 从正规文法转换到 Bison 的输入-From Formal Rules to Bison Input

正规文法是一种数学结构.为了定义 Bison 要分析的语言, 你必须用 Bison 语法编写一个表达该语言的文件,即一个 *Bison 语法(Bison grammar)*文件. 参阅 Bison 的语法文件-Bison Grammar Files.

就象 C 语言中的标识符一样,在 Bison 的输入中,一个正规文法的非终结符由一个标识符表示. 根据惯例,非终结符应该用小写子母表示,例如 expr,stmt 或者 declaration.

在 Bison 中,终结符也被称为*符号类型(token type)*. 符号类型也可以由类似 C 语言标识符来表示. 根据惯例,这些标识符因改用大写子母表示以区分它和非终结符. 例如,INTEGER,INDENTIFIER,IF 或者 RETURN. 一个表示某语言的特定关键字的终结符应该由紧随该关键字之后的它的大写表示来命名. 终结符 error 保留用作错误恢复之用. 参阅 符号-Symbols.

一个终结符也可以由一个像 C 中的字符常量一样的一个字符来表示. 当一个记号就是一个字符(括号,加号等等)的时候,你可以这样做: 使用同一个字符做为那个记号的终结符.

第三种表示终结符的方法是使用包含一些字符的 C 字符串常量. 获取更多这方面的信息可以参阅 符号-Symbols.

语法规则在 Bison 语法中也有相应的表示.例如,下面有一个 C 语言 return 语句的规则. 在引号中的分号,是一个字符记号,它是用来表示 C 语言部分语法的. 没有在引号中的分号和冒号是 Bison 用来表示每一条规则的标点符号.

```
stmt:   RETURN expr ';'
        ;
```

获取这方面更多信息,参阅 语法规则的语法-Syntax of Grammar Rules.

1.3 语义值-Semantic Values

正规文法仅仅靠类别来选择记号:例如,如果一个规则提到了终结符`integer constant', 这就意味着*任何*整数常量在那个位置上都是语法有效的. 常量的精确值与如何分析输入不相关:如果`x+4'符合语法,那么`x+1'或者`x+3989'也符合语法.

但是,当分析输入时,它的精确值是非常重要的,通过精确值可以了解输入的含义. 如果一个编译器不能区别程序中的 4,1 和 3989 等常量,毫无疑问,这个编译器是没有用的! 因此,每一个 Bison 语法的记号既含有一个符号类别也有一个*语义值(semantic value)*. 获取这方面的更多信息,参阅 定义语言的语义-Defining Language Semantics.

符号类型是在语法中定义的终结符,例如 INTEGER,IDENTIFIRE 或者',. 它告诉你决定记号可能有效出现的位置以及如何将它组合成其它记号的信息. 语法规则只知道符号的类型,其它的什么都不知道.

语义值包括了记号的所有剩余信息.例如整数的数值,标识符的名称. (一个如','的记号只是一个标点,并不需要语义值.)

例如,一个分类为 INTEGER 的记号包含语义值 4. 另一个也被分类为 INTEGER 的记号的语义值却是 3989. 当一个语法规则表明 INTEGER 是允许的,任意的这些记号都是可接受的,因为它们都是 INTEGER. 当一个分析器接受了记号,它会跟踪这个记号的语义值.

每一个语法组和它的非终结符也可已有语义值. 一个很典型的例子,在计算器中,一个表达式含有一个数值做为它的语义值, 在程序语言编译器中.一个典型的表达式含有一个用于描述它含义的树型结构做为语义值.

1.4 语义动作

为了更加实用,一个程序不仅仅要分析输入而且必须做的更多. 它应该可以在输入的基础上产生一些输出. 在 Bison 语法中,一个语法规则可以有一个包括多个 C 语句的*动作(action)*. 分析器每次识别一个规则的匹配,相应的动作就会被执行. 获取这方面的更多信息,参阅 动作-Actions.

大多数时候,动作的目的是从部分结构的语义值计算整个结构的语义值. 例如,加入我们有一个规则表明一个表达式可以由两个表达式相加而成. 当分析器识别了一个加法和,每一个子表达式都有一个描述其如何构建的语义值. 这个规的动做就是为了新识别的大表达式建立一个类似的语义值.

例如,这里的一个规则表明一个表达式可由两个表达式相加而成.

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

这个动作表明了如何从子表达式的语义值产生加法和表达式的语义值.

1.5 编写 GLR 分析器-Writing GLR Parsers

在一些文法中,Bison 标准的 LALR(1)分析算法, 不能针对给定的输入应用一个确定的语法规则. 这就是说,Bison 可能不能决定(在当前输入的基础上)应该使用两个可能的归约中的那一个, 或者不能决定到底应该应用一个归约还是先读取一些输入稍后再进行归约. 以上两种冲突分别被称为*归约/归约(reduce/reduce)*冲突(参阅归约/归约-Reduce/Reduce 一章)和 *移进/归约(shift/reduce)*冲突(参阅移进/归约-Shift/Reduce 一章).

有些时候, 为了使用一个很难被修改成 LALR(1)文法的文法做为 Bison 的输入, Bison 需要使用通用的分析算法. 如果你在你的文件中加入了这样的声明%glr-parser(参阅语法大纲-Grammar Outline 一章), Bison 会产通用的 LR(GLR)分析器. 这些分析器(例如,在应用了先前所述的声明之后) 在处理那些不包含未解决的冲突的文法时, 采用与 LALR(1)分析器一样的处理方式. 但是当面临未解决的移进/归约冲突和归约/归约冲突的时候, GLR 分析器权宜地同时处理这两个可能, 即有效地克隆分析器自己以便于追踪这这两种可能性. 每一个克隆出来的分析器还可以再次被克隆, 这就保证在任意给定的时间,可以处理任意个可能的分析. 分析器逐步地进行分析,即所有的分析器它们进入到下一个输入之前, 都会消耗(归约)给定的输入符号. 每一个被克隆的分析器最终只有两个可能的归宿: 或者这个分析器因进入了一个分析错误而最终被销毁, 会这它和其它的分析器合并, 因为它们把输入归约到了一个相同的符号集.

在有多个分析器并存的时刻,Bison 只记录它们的语义动作而不是执行它们. 当一个分析器消失的时候,相应的语义动作记录也消失并且永远不会被执行. 当一个规约使得两个分析器等价而合并的时候, Bison 会记录下它们两个的语义动作集. 每当最后两个分析器合并成为一个单独的分析器的时候, Bison 执行所有未完成的动作.这些动作既可能依靠语法规则的优先级被执行也可能均被 Bison 执行. 在执行完动作之后,Bison 调用指定的用户定义求值函数来产生一个独立的合并值.

1.5.1 使用 GLR 分析器分析非歧义文法

在最简单的情况下,你可以使用 GLR 算法分析那些非歧义但不是 LALR(1)文法的文法. 典型地,这些文法需要不止一个的超前扫描记号, 或者(在极少数情况下)由于 LALR(1)算法丢弃太多的信息而不属于 LALR(1)的文法(它们却属于 LR(1),参阅冲突-Mystery Conflicts).

考虑一个产生于 Pascal 语言枚举和子界类型声明中的问题,. 这里有一些例子:

```
type subrange = lo .. hi;
type enum = (a, b, c);
```

最初的语言标准只允许数字和常量标识符做为子界的范围(`lo'和`hi'). 但是扩展的 Pascal(ISO/IEC10206)以及许多以它的 Pascal 的实现还允许独立的表达式. 这导致了如下一个包含过量括号的情形.

```
type subrange = (a) .. b;
```

考虑如下这个仅含一个值的枚举类型的声明.

```
type enum = (a);
```

(在这里的这些例子是人为制造的,但它们是语法有效的. 在实际的程序中可能会出现更复杂的例子)

这两个例子看起来相同(注:指语法上)直到`..'记号的出现. 对于只含有一个超前扫描记号的普通 LALR(1)分析, 当`a'被分析的时,分析器候并不能两个形式中(注:指枚举或者子界)那一个是正确的. 因为在后一个例子中,`a'必须成为一个新的标识符来表示枚举; 而在前一个例子中必须使用`a'和它的含义来估计它可能是一个常量或者函数调用, 所以我们当然希望分析器可以对此作出正确的决定.

你可将`(a)'分析成"在括号中为说明的标识符"以便稍后进行分析. 但当括号嵌套在表达式的递归规则中的时候, 这种方式需要对语义动作和大部分语法做出相应的调整.

你可能会想到使用词法分析器通过返回已定义和未定地标识符来区别这两种形式. 但是如果这些声明在局部出现, 而`a'在外部定义的时候,这两种形式都是可能的- 既可能是局部重定义的`a'.也可能是使用外部定义的`a'的值. 所以这种方法不可行.

解决这个问题由一个简单的办法,那就是使用 GLR 算法. 当 GLR 分析器到达关键区域的时候, 它会同时沿着两个分支进行分析. 其中的一个分支迟早要进入分析错误. 如果有一个`..'记号在下一个`;'之前的化; 由于枚举类型的规则不能接受`..',所以它应用失败; 否则,子界类型规则应用失败,因为它要求一个`..'记号. 所以,一个分支无声地失败而另一个分支正常地进行, 并且执行所有的在拆分期间被推迟执行的中间动作.

如果输入不符和语法,则这两个分支的分析都会失败并且如常地报告一个语法错误.

分析器的功能似乎是在"猜测"正确的语法分支,换句话说, 它使用了比 LALR(1)算法允许使用的更多的超前扫描记号. LALR(2)有能力分析这个句子, 但是有些不符和 LALR(*k*)的例子, 即使任意多的 *k* 可以这样地处理.

一般而言,一个 GLR 分析器在最坏情况下会花费二次方或者三次方的时间复杂度, 当前的 Bison 甚至会为了某些文法而花费指数的时间. 在实际应用中,这些几乎不会发生, 并且对于许多文法来说,证明它不能发生也是可能的. 当前讨论的例子在两个规则中只有一个冲突, 并且含有冲突的类型声明不能嵌套使用. 所以在任意时刻的分支数量被限制在常量 2 以下, 这时候分析时间仍然是线性的.

这是一个同上面描述相对应的例子. 它由 Pascal 类型声明大大简化而来. 如果输入不符和语法,两个分支都会失败并且如常地报告一个语法错误.

```
%token TYPE DOTDOT ID

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

%%
```

```
type_decl : TYPE ID '=' type ';'
    ;

type : '(' id_list ')'
    | expr DOTDOT expr
    ;

id_list : ID
    | id_list ',' ID
    ;

expr : '(' expr ')'
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | ID
    ;
```

当使用平常的 LALR(1)文法的时候,Bison 会报告一个归约/归约冲突. 在冲突的时候,分析器在会众多选择中选取一个-随意地选择那个先声明的. 所以下面的正确输入不能被识别.

```
type t = (a) .. b;
```

在 Bison 输入文件中, 加入这两个声明(在第一个`%%'之前)分析器可以将分析器编成一个 GLR 分析器, 并且 Bison 不会报告一个归约/归约冲突.

```
%glr-parser
%expect-rr 1
```

并不需要对语法本身进行修改. 分析器现通过上面的限制语法后,可以认识所有有效的声明. 用户实际上并不能查觉分析器的拆分.

这就是我们使用 GLR 而几乎没有坏处的例子. 即使像这样简单的例子,至少两个潜在的问题值得我们注意. 第一,我们总应该分析 Bison 的冲突报告来确定 GLR 拆分总发生在我们想要的时候. 一个 GLR 分析器的拆分会不经意地产生比 LALR 分析器在冲突中静态的错误选择更加不明显的问题. 第二,要仔细考虑与词法分析器的互动(参阅语义记号-Sematic Tokens 一章). 由于在拆分期间的分析器消耗记号时并不产生任何动作, 词法分析器并不能通过分析动作获得信息. 一些与词法分析器的互动在使用 GLR 来消除从词法分析器到语法分析器的复杂读时可以忽略. 你必须监察其余情况下时的正确性.

在我们的例子中,因为并没有新的符号在类型声明中间被定义. 词法分析器基于记号当前在符号表的含意被返回是安全的. 即使在分析枚举常量时定义它们是可能的, 由于它们不能在相同的枚举类型声明中使用, 所以这实际上并没有什么不同.

1.5.2 使用 GLR 解决歧义-Using GLR to Resolve Ambiguities

让我们考虑由 C++语法大大简化而来的例子.

```
%{
  #include <stdio.h>
  #define YYSTYPE char const *
  int yylex (void);
  void yyerror (char const *);
%}

%token TYPENAME ID

%right '='
%left '+'

%glr-parser

%%

prog :
    | prog stmt   { printf ("\n"); }
    ;

stmt : expr ';'  %dprec 1
    | decl     %dprec 2
    ;

expr : ID               { printf ("%s ", $$); }
    | TYPENAME '(' expr ')'
                    { printf ("%s <cast> ", $1); }
    | expr '+' expr    { printf ("+ "); }
    | expr '=' expr    { printf ("= "); }
    ;

decl : TYPENAME declarator ';'
                    { printf ("%s <declare> ", $1); }
    | TYPENAME declarator '=' expr ';'
                    { printf ("%s <init-declare> ", $1); }
    ;
```

```
declarator : ID         { printf ("\"%s\" ", $1); }
      | '(' declarator ')'
      ;
```

这个例子模拟了有问题的 C++语法--某些声明和语句中的歧义.例如,

```
 T (x) = y+z;
```

既可以被分析为一个 expr 也可被分析为一个 stmt (假定`T'被识别为一个 TYPENAME 并且`x'被识别为 ID). Biosn 检测到了这个在规则 expr : ID 和规则 delarator : ID 之间的归约/归约冲突. 分析器遇到 x 的时候还不能解决冲突. 由于这是一个 GLR 分析器,所以它会把问题拆分成两个分析器, 每一个用于解决归约/归约冲突中的一个. 与上一节的例子不同(参阅简单的 GLR 分析器一章),两个分析器中的任一个都不"死亡," 因为这个语法本身就是歧义的. 其中的一个分析最终归约到 stmt : expr ';'而另一个归约到 stmt : decl, 在这之后,两个分析器进入同一个状态: 它们都已经看到`prog stmt'并且有相同的未处理剩余输入. 我们说这些分析器已经*合并(merged)*.

在这个时候,GLR 分析器需要一个关于如何在两个竞争的分析中做出选择的说明. 在上面的例子中,两个%dprec 声明说明了 Bison 给予 decl 的解释以优先级. 这就表明 x 是一个声明符(declarator). 因此分析器会打印出:

```
 "x" y z + T <init-declare>
```

%deprc 声明仅在多于一个分析幸存的情况下起作用. 考虑这个分析器的一个不同的输入字符串:

```
 T (x) + y;
```

像在上一节展示的一样(参阅简单的 GLR 分析器-Simple GLR Parsers 一章), 这是另外一个使用 GLR 分析非歧义结构的例子. 在这里并没有歧义(这个并不能被分析成一个声明). 但是在 Bison 分析器遇到 x 的时候, 它没有足够的信息解决归约/归约冲突(还是不能决定 x 是一个 expr 还是一个 declarator). 在这种情况下,没有优先级可供使用. 分析器再次被拆分成两个,一个假定 x 是一个 expr 而另一个假定 x 是一个 declarator. 两个分析器中的第二个在遇到+的时候被销毁,并且分析器打印出

```
 x T <cast> y +
```

假设你想查看所有的可能性而不是解决歧义. 你必须合并两个可能的分析器的语义动作而不是选择其中的一个. 为了达到这个目的,你需要像如下那样地改变 stmt 的声明:

```
stmt : expr ';'  %merge <stmtMerge>
     | decl      %merge <stmtMerge>
     ;
```

并且定义 stmtMerge 函数如下:

```
static YYSTYPE
stmtMerge (YYSTYPE x0, YYSTYPE x1)
{
  printf ("<OR> ");
  return "";
}
```

并且在文件的开头要伴随一个前置声明在 C 声明中:

```
%{
  #define YYSTYPE char const *
```

```
  static YYSTYPE stmtMerge (YYSTYPE x0, YYSTYPE x1);
%}
```

使用这些声明以后,产生的的分析器将第一个例子既分析成一个 expr 又分析成一个 decl, 并且打印

```
"x" y z + T <init-declare> x T <cast> y z + = <OR>
```

Bison 要求所有参加任何特定合并的产品(productions)拥有相同的`%merge'语句. 否则,歧义不能被解决而且分析器会在任何导致违反规定的合并时候报告一个错误.

1.5.3 编译 GLR 分析器时需要考虑的问题-Considerations when Compiling GLR Parsers

GLR 分析器需要支持 C89 或更新标准的编译器. 额外地,Bison 使用关键字 inline,这个关键字不是 C89 标准但却是 C99 标准,并且是许多 C99 之前编译器支持的扩展. 使用它是为了分析器的用户可以处理移植性问题. 例如,如果使用 Autoconf 和 Autoconf 宏 AC_C_INLINE,一个单单的

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

就足够用了,否则我们建议使用

```
%{
  #if __STDC_VERSION__ < 199901 && ! defined __GNUC__ && ! defined inline
   #define inline
  #endif
%}
```


1.6 位置-Locations

许多应用程序,如解释器和编译器,需要产生一些有用信息或者出错的信息. 为了达到这个目的,我们必须追踪每个语法结构的*原文位置(textual location)*或*位置(location)*. Bison 提供了追踪这些位置的机制.

每一个记号有一个语义值.类似地,每个记号也有一个位置, 对于所有记号和组来说,它们的位置的类型是相同的. 此外,输出的分析器也带有默认的存储位置的数据结构 (获取更多信息,参阅位置-Locations 一章).

像语义值一样,位置可以在动作中使用特定的一套结构来访问. 在上面个的例子中,这个组的的位置是@$,而子表达式的位置是@1 和@3.

当一个规则被匹配,一个默认的动作用于计算左侧的语义值(参阅动作-Actions 一章). 类似地,另外一个默认的动作用于计算位置. 然而,这个默认动作对于对于大多数情况已经足够永, 即经常没有必要为每个规则描述@$应该是如何形成的. 当为一个给定的组建立一个新的位置的时候, 输出的分析器的默认行为是取第一个符号的开头和最后一个符号的末尾.

1.7 Bison 的输出:分析器文件-Bison Output: the Parser File

当你运行 Bison 的时候,你需要给 Bison 一个语法文件做为其输入. Bison 的输出是一个分析这个语法文件描述的语言的 C 源代码文件. 这个文件叫做 *Bison 分析器(Bison parse)*. 我们要记住 Bison 工具和 Bison 分析器是两个明显不同的程序: Bison 工具是一个以 Bison 分析器为输出的程序. 这个 Bison 分析器应是你程序的一部分.

Bison 分析器的工作是依照语法规则组合记号--例如,将标识符和操作符构建成表达式. 在组合的过程中它还要执行相应的语法规定的动作.

记号是来源于称为 *词法分析器(lexical analyzer)*的程序. 你必须以某种形式提供词法分析器(如用 C 编写). Bison 分析器每当需要一个新的记号的时候就会调用词法分析器. Bison 分析器并不之道记号"中"有什么东西(即使它们的语义值可能反映这个). 典型的词法分析器靠分析字符来产生记号,但是 Bison 并不依靠这个. 获取更多细节,参阅 词法分析函数 yylex-The Lexical Analyzer Function yylex.

Bison 分析器文件是定义了名为 yyparse 并且实现了那个语法的函数的 C 代码. 这个函数并不能成为一个完成的 C 程序:你必须提供额外的一些函数. 其中之一是词法分析器.另外的一个是一个分析器报告错误时调用的错误报告函数. 另外,一个完整的 C 程序必须以名为 main 的函数开头; 你必须提供这个函数.并且安排它调用 yyparse. 否则分析器永远都不会运行. 参阅 分析器 C 语言接口-Parser C-Language Interface.

除了你编写的动作中的记号类型名称和符号以外 ,所有 Bison 分析器文件自己定义的符号都以`yy'或者`YY'开头. 这些符号包括了接口函数例如词法分析函数 yylex,错误报告函数 yyerror 和分析器函数 yyparse. 这些符号也包括了许多内部目的的标识符. 所以你要在 Bison 语法文件中避免使用除了本手册定义的以外的以`yy'或者`YY'开头的 C 标识符.

在一些情况下,Bison 分析器文件包含系统头文件. 在这中情况下,你的代码注意被这些文件保留的标识符. 在意些非 GNU 系统,<alloca.h>,<stddef.h>以及<stdlib.h> 被包含在内用于声明内存分配器及相关类型. 如果你定义 YYDEBUG 为非零值,其它的系统头文件也可能被包括进内. (参阅跟踪你的分析器-Tracing Your Parser 一章)

1.8 使用 Bison 的流程-Stages in Using Bison

实际使用 Bison 设计语言的流程,从语法描述到编写一个编译器或者解释器,有三个步骤:

1. 以 Bison 可识别的格式正式地描述语法.(参阅 Bison 语法文件一章) 对每一个语法规则,描述当这个规则被识别时相应的执行动作. 动作由 C 语句序列描述.
2. 编写一个词法分析器处理输入并将记号传递给语法分析器. 词法分析器既可是手工编写的 C 代码(参阅词法分析函数 yylex 一章), 也可以由 lex 产生,但是 lex 的使用并未在这个手册中讨论.
3. 编写一个调用 Bison 产生的分析器的控制函数.
4. 编写错误报告函数.

将这些源代码转换成可执行程序,你需要按以下步骤进行.

1. 按语法运行 Bison 产生分析器.
2. 同其它源代码一样编译 Bison 输出的代码.
3. 链接目标文件以产生最终的产品.

1.9 Bison 语法文件的整体布局-The Overall Layout of a Bison Grammar

Bison 工具的输入文件是以个 *Bison 语法文件(Bison grammar file)*. 通常的 Bison 语法文件格式如下:

```
%{
Prologue
%}

Bison declarations

%%
Grammar rules
%%
Epilogue
```

`%%',`%{' 和`%}'是 Bison 在每个 Bison 语法文件中用于分隔部分的标点符号.

prologue 可用来定义在动作中使用类型和变量. 你可以使用预处理器命令在那里来定义宏, 或者使用#include 包含干这些事情的头文件. 你需要声在那里与许多要在语法规则的动总中使用的全局标识符一起声明词法分析器 yylex 和错误打印程序 yyerror.

Bison declarations 声明了终结符和非终结符以及操作符的优先级和各种符号语义值的各种类型.

Grammar rules 定义了如何从每一个非终结符的部分构建其整体的语法规则.

Epilogue 可以包括任何你想使用的代码. 在 *Prologue* 中声明的函数经常定义在这里. 在简单的程序里,剩余的所有程序可以放在这里.

---

2. 实例-Examples

现在开始我们展示并解释三个使用 Bison 编写的示范程序: 一个逆波兰记号计算器,一个代数符号(中缀)计算器和一个多功能计算器. 这些程序在 BSD Unix 4.3 上测试无误;它们每一个虽然功能有限, 但确都是实用的互动桌面计算器.

这些例子虽然简单,但是用 Bison 语法编写真正的程序设计语言的道理与这相同.

2.1 逆波兰记号计算器-Reverse Polish Notation Calculator

我们的第一个例子是双精度*逆波兰记号(reverse polish notation)*计算器(一个使用后缀操作符的计算器). 这个例子是一个很好的起点,因为没有操作符优先级的问题. 第二个例子会说明如何处理运算符优先级.

这个计算器的源代码文件叫`rpcalc.y'. `.y'是 Bison 惯用的输入文件的扩展名.

2.1.1 rpclac 的声明部分-Declarations for rpcalc

这就是逆波兰记号计算器的 C 和 Bison 声明部分. 像 C 语言一样,注释部分在`/*...*/'中.

```
/* Reverse polish notation calculator.  */
/* 逆波兰记号计算器 */

%{
  #define YYSTYPE double
  #include <math.h>

  int yylex (void);
  void yyerror (char const *);
%}

%token NUM

%% /* Grammar rules and actions follow.  */
```

声明部分(参阅 *Prologue* 部分-The prologue 一章)包括了两个预处理指令和两个前置声明.

#define 指令定义了 YYSTYPE 宏, 它指明了记号和组(参阅语义值的数据类型-Data Types of Semantic Values 一章)语义值的 C 数据类型. Bison 分析器会使用任何 YYSTYPE 定义的数据类型; 如果你没有定义它,int 则是默认的类型. 因为我们指明了 double,所以每个记号和表达式已经关联了一个浮点数值.

#include 用来声明幂函数 pow.

由于 C 语言要求函数必须在使用之前声明, 所以前置声明 yylex 和 yyerror 是必须的. 这些函数将在 *epilogue* 部分定义,但是分析器会调用它们, 所以它们必须在 *prologue* 部分声明.

第二部分-*Bison declarations*,提供了有关记号类型的信息(参阅 *Bison Declarations* 部分-The Bison Declarations Section 一章). 每一个不只一个字符组成的终结符必须在这里声明.(通常没有必要声明单一字符.) 在这个例子中,所有的算术操作符都是单一字符的, 所以我们在这里仅需要声明的终结符是 NUM--数字常量的记号类型.

2.1.2 rpcalc 的语法规则-Grammar Rules for rpcalc

这就是逆波兰记号计算器的语法规则:

```
input:    /* empty */
        | input line
;

line:     '\n'
        | exp '\n'      { printf ("\t%.10g\n", $1); }
;

exp:      NUM           { $$ = $1;           }
        | exp exp '+'   { $$ = $1 + $2;      }
        | exp exp '-'   { $$ = $1 - $2;      }
        | exp exp '*'   { $$ = $1 * $2;      }
        | exp exp '/'   { $$ = $1 / $2;      }
        /* Exponentiation */
        | exp exp '^'   { $$ = pow ($1, $2); }
        /* Unary minus    */
        | exp 'n'       { $$ = -$1;          }
;
%%
```

在这里定义的 rpcalc"语言"的组是:表达式(名称是 exp), 输入行(line),整个的输入脚本(input). 这些非终结符每个都有多个可选择的规则. 这些规则由`|'连接而成. 我们可以把`|'读做"或者". 接下来的部分解释了规则的含义.

语法的语义由当组组被识别时执行的动作决定. 动作由在大括号中 C 代码组成. 参阅动作-Actions 一章.

你必须用 C 语言来指定这些动作, 但是 Bison 也提供了在规则之间传递语义值的方法. 在每个动作中,伪变量$$代表着将要构建的组的语义值. 赋值给$$是大多数动作的工作. 规则的组成部分的语义值由$1,$2 等等指定.

2.1.2.1 解释 input-Explanation of input
2.1.2.2 解释 line-Explanation of line
2.1.2.3 解释 expr-Explanation of expr

2.1.2.1 解释 input-Explanation of input

考虑 input 的定义:

```
input:    /* empty */
        | input line
;
```

这个定义可以被解释为:"一个完整的输入己可能是一个空字符串, 也可能是一个完整输入后紧跟一个输入行". 我们应该注意到"完整输入"定义在它自己的条款中. 由于 input 总是出现在符号序列的最左端, 我们称这种定义为*左递归(left recursive)*. (参阅 递归规则-Recursive Rule.)

第一个可选择的规则为空是由于在冒号和第一个`|'之间没有任何符号; 这意味着 input 可以匹配一个空字符串的输入(没有符号). 我们这样编写规则是因为在你启动计算器后,在右端输入 Ctrl-d 是合法的. 把空规则放在最开始并伴随注释`/* empty */'是使用惯例.

第二个可选的规则(input line)处理了所有非平凡的输入. 它的含义是"在读取任意个数的 line 后,如果可能的化读取更多的 line." 作递归使得这个规则进入循环. 由于第一个可选的规则与空输入匹配,循环可能被执行零次或多次.

分析器函数 yyparse 继续处理输入直到发现一个语法错误或者词法分析器表明没有更多的输入记号为止; 我们会安排后者在输入结束的时候发生.

2.1.2.2 解释 line-Explanation of line

现在我们考虑 line 的定义:

```
line:    '\n'
       | exp '\n'  { printf ("\t%.10g\n", $1); }
;
```

第一个选择是一个换行符号; 这意味着 rpcalc 接受一个空白行(并且忽略它,因为没有相关的动作). 第二个选择是一个表达式后紧跟着一个换行符. 这个选择使得 rpcalc 变得实用. $1 的值是 exp 组的语义值, 这是因为 exp 是诸多选择中的第一个符号. 这个值就是用户需要的计算结果. 相关的动作打印了这个这个值.

这个动作非同寻常,因为它并没有向$$赋值. 这样做的结果就是与 line 相关联的语义值并未被初始化(它的值是不可预期的). 如果那个值被使用的话,这会成为一个 bug,但是我们并未使用它. rpcalc 一旦已经打印了用户输入行的值,就不再需要那个值了.

2.1.2.3 解释 expr-Explanation of expr

exp 组有很多规则,每一个都是一种表达式. 第一个规则处理最简单的表达式:仅仅是数字的表达式. 第二个处理看起来像两个表达式后紧跟一个假发符号的加法表达式. 第三个处理减法等等.

```
exp:     NUM
       | exp exp '+'    { $$ = $1 + $2;    }
       | exp exp '-'    { $$ = $1 - $2;    }
```

```
        ...
        ;
```

我们已经使用`|'将 exp 的规则连接起来. 但是我们也可以将它们分开来写:

```
  exp:      NUM ;
  exp:      exp exp '+'     { $$ = $1 + $2;    } ;
  exp:      exp exp '-'     { $$ = $1 - $2;    } ;
        ...
```

大部分规则都有从其部分值来计算表达式值的动作. 例如,在加法的规则中,$1 指的是第一个部件 exp, 而$2 指的是第二个部件. 第三个部件<code>'+'并没有相关联的语义值. 但是如果它有语义值的话,你可以用$3 来代表. 当 yyparse 使用这个规则识别了一个加法和表达式, 两个自表达式值的相加而得到了整个表达是的值. 参阅 动作-Actions.

你不必为每一个规则都指定动作. 当一个规则没有动作时,Bison 会默认地将$1 的值复制给$$. 这就是在第一规则被(使用 NUM 的规则)识别时发生的事情.

在这里展示的是推荐的惯用格式, 但是 Bison 并没有要求一定要这么做. 你可以增加或者更改任意你想要的数量的空白. 例如, 这个:

```
  exp   : NUM | exp exp '+' {$$ = $1 + $2; } | ... ;
```

与这个的意义相同

```
  exp:      NUM
        | exp exp '+'    { $$ = $1 + $2; }
        | ...
  ;
```

然而,后一种写法明显更可读.

2.1.3 rpcalc 的词法分析器-The rpcalc Lexical Analyzer

词法分析器的工作是低级别的分析: 将字符或者字符序列转化成记号. Bison 分析器靠调用词法分析器来获取它的记号. 参阅 词法分析函数 yylex-The Lexical Analyzer Function yylex.

即使最简单的词法分析器也是 RPN 计算器所必须的. 这个词法分析器跳过了空白和制表符, 然后将数字读取为 double 并且以 NUM 记号返回它们. 任何不是数字的字符都是一个分隔记号. 注意到一个单个字符的记号是这个字符本身.

词法分析器的返回值是一个代表记号类型的数字码. 相同的文字(在 Biosn 规则中使用,代表这个记号类型) 也是一个代表这个类型数字码的 C 表达式. 它以两种方式工作. 如果记号类型是一个字符,那么它的数字码就是那个字符; 你可以在词法分析器中使用相同字符表达那个数字码. 如果记号类型是一个标识符, 那个标识符是一个由 Bison 定义为一个恰当的数字的宏, 因此在这个例子中,NUM 是一个给 yylex 使用的宏.

记号的语义值(如果有的话)被存储在全局变量 yylval 中. Bison 分析器会在需要语义值适当的时候找到它. (yylval 的 C 数据类是 YYSTYPE,它在语法的开始被定义; 参阅 rpcalc 的声明部分-Declarations for rpcals 一章.

符号类型码 0 在输入结束的时候被返回. (Bison 将任何不正确的值识别为输入结束)

这就是词法分析器的代码:

```
  /* The lexical analyzer returns a double floating point
```

```
   number on the stack and the token NUM, or the numeric code
   of the character read if not a number.  It skips all blanks
   and tabs, and returns 0 for end-of-input.  */
/* 词法分析起在栈上返回一个双精度浮点数(注:指 yylval)并且返回记号 NUM,
   或者返回不是数字的字符的数字码.它跳过所有的空白和制表符,
   并且返回 0 作为输入的结束. */
#include <ctype.h>

int
yylex (void)
{
  int c;

  /* Skip white space.  */
  /* 处理空白. */
  while ((c = getchar ()) == ' ' || c == '\t')
    ;
  /* Process numbers.  */
  /* 处理数字 */
  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      scanf ("%lf", &yylval);
      return NUM;
    }
  /* Return end-of-input.  */
  /* 返回输入结束 */
  if (c == EOF)
    return 0;
  /* Return a single char.  */
  /* 返回一个单一字符 */
  return c;
}
```


2.1.4 控制函数-The Controlling Function

为了保证这个例子的精巧,控制函数也保持了最小化。它仅仅要求调用 yyparse 来开始分析的处理.

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


2.1.5 错误报告的规则-The Error Reporting Routine

当 yyparse 侦测到语法错误的时候, 它会调用错误报告函数 yyerror 来打印一个错误信息(经常但不总是"syntax error"). yyparse 需要程序员来提供 yyerror(参阅**分析器 C 语言接口-Parser C-Language Interface** 一章), 所我我们使用这样的定义:

```
#include <stdio.h>

/* Called by yyparse on error.  */
void
yyerror (char const *s)
{
  fprintf (stderr, "%s\n", s);
}
```

在 yyerror 返回之后, 如果语法包括了适当的错误规则(参阅**错误恢复-Error Recovery** 一章) Bison 分析器可以从错误中恢复并且继续分析. 否则,yyparse 返回非零值. 我们在这个例子中并没有编写任何错误规则, 所以任何无效的输入会导致计算器程序退出. 这种做法显然对于这正的计算器是不够用的, 但是足够用于这个例子.

2.1.6 运行 Bison 来产生分析器-Running Bison to Make the Parser

(参阅 **Bison 语法文件的布局-The Overall Layout of a Bison Grammar** 一章). 在运行 Bison 制造分析器之前,我们要决定如何把源代码安排在一个或多个源文件中. 对于这样一个简单的例子来说, 最简单的方法就是把所有的东西放到一个文件中. yylex,yyerror 和 main 放在末尾的 *epilogue* 部分中.

对于一个大工程来说,你可能会有很多的源代码文件, 这时候你需要使用 make 安排它们的重编译.

因为所有的代码都在一个文件中, 你使用下面的命令将它转化成一个分析器文件:

```
bison file_name.y
```

在这个例子中,这个文件是`rpcalc.y'(代表 "Reverse Polish CALCulator"). Bison 产生一个名为`*file_name*.tab.c'的文件. 并且将原文件的`.y'的扩展名移去. 在输入中的额外函数(yylex,yyerror 和 main)被逐字地复制到输出文件.

2.1.7 编译分析器文件-Compiling the Parser File

这就是如何编译并运行分析器文件:

```
# 列出当前目录的文件.
$ ls

rpcalc.tab.c  rpcalc.y

# 编译 Bison 分析器.
# `-lm' 告诉编译器搜索与 pow 匹配的库.
$ cc -lm -o rpcalc rpcalc.tab.c

# 再次列文件.
$ ls
rpcalc  rpcalc.tab.c  rpcalc.y
```

文件`rpcalc'现在已经包含了可执行代码. 这就是使用 rpcalc 的会话的例子:

```
$ rpcalc
4 9 +
13
3 7 + 3 4 5 *+-
-13
3 7 + 3 4 5 * + - n            注意负号操作符, `n'
13
5 6 / 4 n +

-3.166666667
3 4 ^                          幂运算
81
^D                             文件结束标识符
$
```


2.2 中缀符号计算器:calc-Infix Notation Calculator: calc

现在我们可以通过修改 rpcalc 使其处理中缀操作符. 中缀操作符涉及到了操作符优先级的概念以及任意深度的括号嵌套. 这里是`calc.y',一个中缀桌面计算器的代码.

```
/* Infix notation calculator.  */
```

```
/* 中缀符号计算器 */

%{
  #define YYSTYPE double
  #include <math.h>

  #include <stdio.h>
  int yylex (void);
  void yyerror (char const *);
%}

/* Bison declarations.  */
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG     /* negation--unary minus */ /* 负号 */
%right '^'    /* exponentiation */ /* 幂运算 */

%% /* The grammar follows.  */ /*    下面是语法 */
input:    /* empty */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;

exp:      NUM                { $$ = $1;         }
        | exp '+' exp        { $$ = $1 + $3;    }
        | exp '-' exp        { $$ = $1 - $3;    }
        | exp '*' exp        { $$ = $1 * $3;    }
        | exp '/' exp        { $$ = $1 / $3;    }
        | '-' exp  %prec NEG { $$ = -$2;        }
        | exp '^' exp        { $$ = pow ($1, $3); }
        | '(' exp ')'        { $$ = $2;         }
;
%%
```

yylex,yyerror 和 main 可以与上一个例子一样.

这段代码展示了两个重要的新特征.

在第二部分中(*Bison declarations*), %left 声明了记号类型并且指明它们是左结合操作符. %left 和%right(右结合)的声明代替了%token. %token 是用来声明没有结合性的记号类型的. (这些记号(注:是指`'+'', `'-'', `'*'', `'/'', `'NEG'') 是原本并不用声明的单字符记号.我们声明它们的目的是指出它们的结合性.)

操作符优先级是由声明所在行的顺序决定的. 行号越大的操作符(在一页或者屏幕底端)具有越高的优先级. 因此,幂运算具有最高优先级,负号(NEG)其次, 接这是`*'和`/'等等. 参阅 操作符优先级-Operator Precedence.

另外一个重要的特征是在语法部分的负号操作符中使用了%prec. 语法中的%prec 只是简单的告诉 Bison 规则`| '-' exp'与 NEG 有相同的优先级--在前述的优先级规则中. 参阅 依赖上下文的优先级-Context-Dependent Precedence.

这就是一个运行`calc.y'的例子:

```
$ calc
4 + 4.5 - (34/(8*3+-3))
6.880952381
-56 + 2
-54
3 ^ 2
9
```

2.3 简单的错误恢复-Simple Error Recovery

直到这一章之前,这个手册并未提及*错误恢复(error recovery)*的内容-- 即在分析器侦测到错误之后如何继续分析. 所有我们需要做的事情就是编写 yyerror 函数. 我们可以回忆一下以前的例子, yyparse 在调用 yyerror 后返回. 这就意味着任一个错误的输入会导致计算机程序的退出. 现在我们就展示如何改正这个不足.

Bison 语言自己包括了可以插入到语法规则中去的保留字 error. 在这个例子中,它被添加到 line 的一个可选的规则中.

```
line:    '\n'
       | exp '\n'   { printf ("\t%.10g\n", $1); }
       | error '\n' { yyerrok;                  }
;
```

这个添加的规则允许在语法错误发生的时候有简单的错误恢复动作. 如果一个读入一个无法求值的表达式, 这个错误会被识别成 line 的第三个规则并且分析会继续执行. (yyerror 函数仍会被调用来打印它的信息). 执行这个动作的语句 yyerrok 是一个被 Bison 自动定义的宏. 它的含义是错误恢复已经完成(参阅错误恢复-Error Recovery 一章). 我们应当注意到 yyerror 和 yyerrok 的区别, 它们的印刷都没有错误.

这种形式的错误恢复用于处理语法错误. 还有很多其它形式的错误; 例如,除数为 0,这会产生一个通常致命的异常信号(an exception signal). 一个真正的计算器必须处理这种信号并且使用 longjmp 返回到 main 并且继续分析输入行; 它(注:真正的计算器)也可以丢弃剩余的输入行. 我们并不深入地讨论这个问题, 因为这与 Bison 程序无关.

2.4 带有位置追踪的计算器:ltcalc-Location Tracking Calculator: ltcalc

这个例子将中缀符号计算器扩展以使其带有位置追踪功能. 这个特征将被应用于改进错误消息的显示. 为了把保持简洁性, 这个例子是一个简单的整数计算器. 为了使用位置,大多数工作将在词法分析器中完成.

2.4.1 ltcalc 的 *Declarations*-Declarations for ltcalc

位置追踪计算器的 C 和 Bison 声明部分与中缀符号计算器的声明部分相同.

```
/* Location tracking calculator.  */
/* 位置追踪计算器 */

%{
  #define YYSTYPE int
  #include <math.h>
  int yylex (void);
  void yyerror (char const *);
%}

/* Bison declarations.  */
/* Bison 声明 */
%token NUM

%left '-' '+'
%left '*' '/'
%left NEG
%right '^'

%% /* The grammar follows.  */ /* 下面是语法 */
```

注意到并没有位置的特别声明. 并不需要定义一个存储位置的数据类型: 我们使用 Bison 提供的默认的类型(参阅位置的数据类型-Data Types of Locations 一章). 这种数据类型是带有四个整数域的结构体: first_line,first_column,last_line 和 last_column.

2.4.2 ltcalc 的语法规则-Grammar Rules for ltcalc

是否处理位置对于你的语言的语法明没有影响. 因此,这个语言的语法规则与前一个例子的非常接近; 我们仅仅修改它们以获取新的信息.

在这里,我们使用位置来报告除数为 0 并且对错误的表达式或子表达式进行定位.

```
 input   : /* empty */
         | input line
 ;

 line    : '\n'
         | exp '\n' { printf ("%d\n", $1); }
 ;

 exp     : NUM           { $$ = $1; }
         | exp '+' exp   { $$ = $1 + $3; }
         | exp '-' exp   { $$ = $1 - $3; }
         | exp '*' exp   { $$ = $1 * $3; }
         | exp '/' exp
           {
             if ($3)
               $$ = $1 / $3;
             else
               {
                 $$ = 1;
                 fprintf (stderr, "%d.%d-%d.%d: division by zero",
                          @3.first_line, @3.first_column,
                          @3.last_line, @3.last_column);
               }
           }
         | '-' exp %preg NEG     { $$ = -$2; }
         | exp '^' exp           { $$ = pow ($1, $3); }
         | '(' exp ')'           { $$ = $2; }
```

这段代码展示了如何对规则部件使用伪变量@*n* 以及对组使用伪变量@$在语义动作中确定位置.

我们不需要向@$赋值:输出分析器会自动这样作. 默认地,在执行每个动作的 C 代码之前, 对于一个具有 *n* 个部件的规则, @$被设定为从@1 的开始到@*n* 的结束. 我们可以重新定义这个动作(参阅位置的默认动作-Default Action for Locations 一章), 对于一些特殊的规则,@$可以手动计算.

2.4.3 ltcalc 的词法分析器-The ltcalc Lexical Analyzer.

直到现在,我们仍然在依靠 Bison 的默认来激活位置追踪. 下一步就我们重写词法分析器, 并且使它与语法分析器的记号位置相适合, 就像它已经为语义值做的那样.

在最后,为了避免计算的位置出错,我们必须对每个输入字符进行计数.

```
int
yylex (void)
{
  int c;

  /* Skip white space.  */
  /* 跳过空白 */
  while ((c = getchar ()) == ' ' || c == '\t')
    ++yylloc.last_column;

  /* Step.  */
  yylloc.first_line = yylloc.last_line;
  yylloc.first_column = yylloc.last_column;

  /* Process numbers.  */
  /* 处理数字 */
  if (isdigit (c))
    {
      yylval = c - '0';
      ++yylloc.last_column;
      while (isdigit (c = getchar ()))
        {
          ++yylloc.last_column;
          yylval = yylval * 10 + c - '0';
        }
```

```
      ungetc (c, stdin);
      return NUM;
    }

  /* Return end-of-input.  */
  /* 返回输入结束 */
  if (c == EOF)
    return 0;

  /* Return a single char, and update location.  */
  /* 返回一个单字符,并且更新位置 */
  if (c == '\n')
    {
      ++yylloc.last_line;
      yylloc.last_column = 0;
    }
  else
    ++yylloc.last_column;
  return c;
}
```

词法分析器基本上做出了与前一个例子相同的处理: 它跳过了空格和制表符,并且读入了许多单字符记号. 而外地,它更新了含有记号位置的全局变量(类型是 YYLTYPE)yyloc.

现在,每当这个函数返回一个记号,与语义值一样, 分析器就有了这个记号的编号和它的文字位置. 最后需要改变的就是初始化 yyloc. 例如在控制函数中:

```
int
main (void)
{
  yylloc.first_line = yylloc.last_line = 1;
  yylloc.first_column = yylloc.last_column = 0;
  return yyparse ();
}
```

要记住:计算位置并不是语法的事情. 每个字符必须关联一个位置更新, 不论它是在一个有效的输入中还是在注释中或者字符串中等等.

2.5 多功能计算器:mfcalc-Multi-Function Calculator: mfcalc

到现在为止,我们已经讨论了 Bison 的基础部分. 是时间去转移到一些高级的问题中去. 上述的计算器仅仅提供了五个功能, `+', `-', `*', `/'和`^'. 让我们的计算器提供其它数学函数如 sin,cos 等等是一个不错的想法.

一旦新的操作符只是单字符,将它们添加到中缀计算器是很简单的事情. 词法分析器 yylex 将所有非数字字符返回成记号, 所以新的语法规则对于新的操作符来说足够用. 但是我们需要一些更灵活的东西:采用这种格式的内建函数:

function_name (*argument*)

同时,我们靠建立命名变量来为计算器添加记忆功能. 我们可以在它们中存储数值,并在稍后使用它们. 这就是多功能计算器的一个会话的例子:

```
$ mfcalc
pi = 3.141592653589

3.1415926536
sin(pi)
0.0000000000
alpha = beta1 = 2.3
2.3000000000
alpha
2.3000000000
ln(alpha)
0.8329091229
exp(ln(beta1))
2.3000000000
$
```

注意到多重赋值和嵌套函数是允许使用的.

2.5.1 mfcalc 的声明-Declarations for mfcalc

这就是多功能计算器的 C 和 Bison 声明部分:

```
%{
  #include <math.h>  /* For math functions, cos(), sin(), etc.  */ /* 为了使用数学函数, cos(),
sin(), 等等 */
  #include "calc.h"  /* Contains definition of `symrec'.  */ /* 包含了 `symrec'的定义 */
  int yylex (void);
  void yyerror (char const *);
%}
```

```
%union {
  double  val;  /* For returning numbers.  */ /* 返回的数值 */
  symrec  *tptr;  /* For returning symbol-table pointers.  */ /* 返回的符号表指针 */
}
%token <val>  NUM       /* Simple double precision number.  */ /* 简单的双精度数值 */
%token <tptr> VAR FNCT   /* Variable and Function.  */ /* 变量和函数 */
%type  <val>  exp

%right '='
%left '-' '+'
%left '*' '/'
%left NEG     /* negation--unary minus */ /* 负号 */
%right '^'    /* exponentiation */ /* 幂 */
%% /* The grammar follows.  */
```

上述的语法仅仅引进了两个 Bison 语言的信特征. 这些特征允许语义值拥有多种数据类型. (参阅**多种值类型**-More Than One Value Type 一章).

%union 声明了所有可能类型清单; 这是用来取代 YYSTYPE 的. 现在允许的类型是双精度(为了 exp 和 NUM)和指向符号表目录项的指针. 参阅 **值类型集**-The Collection of Value Types.

由于语义值值现在可以有多种类型, 对每一个使用语义值的语法符号关联一个语义值类型是很必要的. 这些符号是 NUM,VAR,FNCT,和 exp. 它们的在声明的时候已经指明了语义值类型(在中括号之间).

%type 用来声明非终结符,就像%token 用来声明符号类型(注:终结符)的一样. 我们之前并没有%type 是因为非终结符经常在定义它们的规则中隐含地声明. 但是 exp 必须被明确地声明以便我们使用它的语义值类型. 参阅 **非终结符**-Nonterminal Symbols.

2.5.2 mfcalc 的语法规则-Grammar Rules for mfcalc

这就是多功能计算器的语法规则. 它们之中的大多数都是从 calc 复制过来的; 三个提到 VAR 或者 FNCT 的规则是新增的.

```
input:   /* empty */
        | input line
;

line:
         '\n'
        | exp '\n'   { printf ("\t%.10g\n", $1); }
        | error '\n' { yyerrok;                  }
;
```

```
exp:      NUM                { $$ = $1;                         }
        | VAR                { $$ = $1->value.var;              }
        | VAR '=' exp        { $$ = $3; $1->value.var = $3;     }
        | FNCT '(' exp ')'   { $$ = (*($1->value.fnctptr))($3); }
        | exp '+' exp        { $$ = $1 + $3;                    }
        | exp '-' exp        { $$ = $1 - $3;                    }
        | exp '*' exp        { $$ = $1 * $3;                    }
        | exp '/' exp        { $$ = $1 / $3;                    }
        | '-' exp  %prec NEG { $$ = -$2;                        }
        | exp '^' exp        { $$ = pow ($1, $3);               }
        | '(' exp ')'        { $$ = $2;                         }
;
/* End of grammar.  */
%%
```


2.5.3 mfcalc 的符号表-The mfcalc Symbol Table

多功能计算器需要一个符号表来追踪变量和符号的名称和意义. 这并不影响语法规则(除了动作以外)或者 Biosn 声明. 但是它要求一些额外的 C 函数支持.

符号表本身由记录的链表组成. 它的定义在头文件`calc.h'中,如下. 它提供了将函数或者变量插入符号表的功能.

```
/* Function type.  */
/* 函数类型 */
typedef double (*func_t) (double);

/* Data type for links in the chain of symbols.  */
/* 链表节点的数据类型 */
struct symrec
{
  char *name;  /* name of symbol */ /* 符号的名称 */
  int type;    /* type of symbol: either VAR or FNCT */ /* 符号的类型: VAR 或 FNCT */
  union
  {
    double var;      /* value of a VAR */ /* VAR 的值 */
    func_t fnctptr;  /* value of a FNCT */ /* FNCT 的值 */
  } value;
```

```
  struct symrec *next;  /* link field */ /* 指针域 */
};

typedef struct symrec symrec;

/* The symbol table: a chain of `struct symrec'.  */
/* 符号表: `struct symrec'的链表 */
extern symrec *sym_table;

symrec *putsym (char const *, func_t);
symrec *getsym (char const *);
```

新版本的 main 包含了一个 init_table 的调用, 这个函数用来初始化符号表. 这就是 main 和 init_table 的代码:

```
#include <stdio.h>

/* Called by yyparse on error.  */
/* 出错时被 yyparse 调用 */
void
yyerror (char const *s)
{
  printf ("%s\n", s);
}

struct init
{
  char const *fname;
  double (*fnct) (double);
};

struct init const arith_fncts[] =
{
  "sin",  sin,
  "cos",  cos,
  "atan", atan,
  "ln",   log,
  "exp",  exp,
  "sqrt", sqrt,
  0, 0
```

```
};

/* The symbol table: a chain of `struct symrec'.  */
/* 符号表: `struct symrec'链表 */
symrec *sym_table;

/* Put arithmetic functions in table.  */
/* 将数学函数放入符号表(注:保留字的实现方式) */
void
init_table (void)
{
  int i;
  symrec *ptr;
  for (i = 0; arith_fncts[i].fname != 0; i++)
    {
      ptr = putsym (arith_fncts[i].fname, FNCT);
      ptr->value.fctptr = arith_fncts[i].fnct;
    }
}

int
main (void)
{
  init_table ();
  return yyparse ();
}
```

靠简单地编辑初始化列表和添加必要的头文件, 你可以为计算器添加额外的功能.

两个重要的函数允许对符号表进行搜索和安置操作. putsym 函数需要传递要安置对象的名称和类型(VAR 或 FNCT). 对象被连接到链表的头部,一个指向这个对象的指针最后被返回. getsym 函数要传递需要查找的符号的名称. 如果找到,指向那个符号的指针回返回,否则返回 0.

```
symrec *
putsym (char const *sym_name, int sym_type)
{
  symrec *ptr;
  ptr = (symrec *) malloc (sizeof (symrec));
  ptr->name = (char *) malloc (strlen (sym_name) + 1);
```

```
  strcpy (ptr->name,sym_name);
  ptr->type = sym_type;
  ptr->value.var = 0; /* Set value to 0 even if fctn.  */ /* 置 0 即使是 fctn */
  ptr->next = (struct symrec *)sym_table;
  sym_table = ptr;
  return ptr;
}

symrec *
getsym (char const *sym_name)
{
  symrec *ptr;
  for (ptr = sym_table; ptr != (symrec *) 0;
       ptr = (symrec *)ptr->next)
    if (strcmp (ptr->name,sym_name) == 0)
      return ptr;
  return 0;
}
```

yylex 函数现在必须能识别出变量,数字值和单字符算术操作符. 开头为非字数字的字符串被识别为变量还是函数依赖于符号表对它们的描述.

字符串被传递到 getsym 用于在符号表中查找. 如果名称出现在表格中,指向它的位置的指针和它的类型会返回给 yyparse.

如果尚未出现在符号表中,它会被安置为一个 VAR 使用函数 putsym. 同样地,一个指针和它的类型(必须是 VAR)被返回给 yyparse.

yylex 中处理数字制和算术运算符的代码并不需要更改.

```
 #include <ctype.h>

int
yylex (void)
{
  int c;

  /* Ignore white space, get first nonwhite character.  */
  /* 忽略空白,获取第一个非空白的字符 */
  while ((c = getchar ()) == ' ' || c == '\t');

  if (c == EOF)
```

```
  return 0;

/* Char starts a number => parse the number.         */
/* 以数字开头 => 分析数字 */
if (c == '.' || isdigit (c))
  {
    ungetc (c, stdin);
    scanf ("%lf", &yylval.val);
    return NUM;
  }

/* Char starts an identifier => read the name.       */
/* 以标识符开头 => 读取名称 */
if (isalpha (c))
  {
    symrec *s;
    static char *symbuf = 0;
    static int length = 0;
    int i;

    /* Initially make the buffer long enough
       for a 40-character symbol name.  */
    /* 在开始的时候使缓冲区足够容纳 40 字符长的符号名称*/
    if (length == 0)
      length = 40, symbuf = (char *)malloc (length + 1);

    i = 0;
    do
      {
        /* If buffer is full, make it bigger.        */
        /* 如果缓冲区已满,使它大一点 */
        if (i == length)
          {
            length *= 2;
            symbuf = (char *) realloc (symbuf, length + 1);
          }
        /* Add this character to the buffer.         */
```

```
        /* 将这个字符加入缓冲区 */
        symbuf[i++] = c;
        /* Get another character.                    */
        /* 获取另外一个字符 */
        c = getchar ();
       }
     while (isalnum (c));

     ungetc (c, stdin);
     symbuf[i] = '\0';

     s = getsym (symbuf);
     if (s == 0)
       s = putsym (symbuf, VAR);
     yylval.tptr = s;
     return s->type;
   }

  /* Any other character is a token by itself.        */
  /* 其余的字符是自己为记号的字符 */
  return c;
}
```

这个程序既有效又灵活. 你可以轻松地加入新的函数. 而且加入于预定义数值如 pi 或者 e 也是很简单的事情.

2.6 练习-Exercises

1. 向文件 `math.h'中的初始列表添加新的函数.
2. 添加另外一个包括常量和它们数值的数组. 然活修改 init_table 将这些常量添加到符号表. 使这些常量类型为 VAR 最简单.
3. 使这个程序当用户引用一个未初始化的变量时报告一个错误.

---

3. Biosn 的语法文件-Bison Grammar Files

Bison 使用一个描述上下文无关文法的文件做为输入, 并制造一个实别改文法的正确实例的 C 语言函数.

Bison 语法输入文件通常以`.y'结尾.参阅 调用 Bison-Invoking Bison.

3.1 Bison 语法的提纲-Outline of a Bison Grammar

一个 Bison 语法文件有四个主要的部分, 就像如下所示,由恰当的分隔符分隔.

```
%{
  Prologue
%}

Bison declarations

%%
Grammar rules

%%

Epilogue
```

注释包含在`/* ... */'之中,并且可以在任意部分出现. 做为一个 GNU 扩展,`//'引进了一个直到这行末尾的注释.

3.1.1 *Prologue* 部分-The prologue

Prologue 部分包括宏定义和在语法规则动作中使用的函数和变量的声明. 这些将复制到分析器文件的开头以便先于 yyparse 的定义. 你可以使用`#include'来从头文件获取声明. 如果你不需要任何的 C 声明,你可以省略这个部分的括号分隔符`%{'和`%}'.

可以有多个与 *Bison declarations* 混合的 *Prologue* 部分. 这种做法允许你拥有相互引用的 C 和 Bison 声明. 例如,%union 可以使用定义在头文件的数据类型, 并且你希望使用带有 YYSTYPE 类型做为参数的函数. 可以通过两个 *Prologue* 块来实现这个, 一个在%union 之前,另一个在之后.

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

  #include "ptypes.h"
%}

%union {
  long int n;
  tree t;  /* tree is defined in `ptypes.h'. */
        /* tree 在`ptypes.h'中定义. */
}

%{
  static void print_token_value (FILE *, int, YYSTYPE);
  #define YYPRINT(F, N, L) print_token_value (F, N, L)
%}

...
```


3.1.2 *Bison Declarations* 部分-The Bison Declarations Section

Bison declatations 部分包含了定义终结符和非终结符的声明,优先级等等. 在一些简单的语法中,可以不需要任何声明. 参阅 *Bison Declarations* 部分-Bison Declarations.

3.1.3 语法规则部分-The Grammar Rules Section

Grammar Rules 部分包含了一个或多个 Bison 语法规则. 参阅 用来表示语法规则的语法-Syntax of Grammar Rules.

在这里至少应该有一个语法规则,并且第一个`%%'(先于语法规则的那个)绝对不能省略,解释它在文件的最开头.

3.1.4 *Epilogue* 部分-The epilogue

就像 *Prologue* 部分被复制到开头一样,*Epilogue* 部分被逐字地复制到分析器文件的结尾. 如果你想放一些代码却没必要放在 yyparse 的定义之前,这里是最方便的地方. 例如,yylex 和 yyerror 的定义就经常放在这里. 因为 C 语言要求函数在使用之前必须声明. 你经常需要在 *Prologue* 部分声明类似 yylex 和 yyerror 的函数, 即使你在 *Epilogue* 部分已经定义了它们. 参阅 分析器 C 语言接口-Parser C-Language Interface.

如果最后一部分为空,你可以省略分隔它的分隔符`%%'.

Bison 分析器自己包含了许多以`yy'和`YY'开头宏和标识符的定义. 所以在 *Epilogue* 部分避免使用这种类型的名字(出了这个文档讨论的之外)是一个好主意.

3.2 符号,终结符和非终结符-Symbols, Terminal and Nonterminal

Bison 语法中的*符号(Symbols)*代表着语言的语法类型.

一个*终结符(terminal symbol)*(也被称做*符号类型(token type)*代表了一类从构造上等价的记号. 你在语法中使用符号的意思就是一个这种类型的记号是允许的. Bison 分析器将符号表示为数字码. yylex 返回一个记号类型来指明一个被读入的记号是什么类型的. 你不需要了解那个码的数值是多少; 你使用代表它的符号就可以了.

一个*非终结符(noterminal symbol)*代表一类从构造上等价的组. 符号名称用于编写语法规则. 按照惯例,所有的非终结符都应该是小写的.

符号名称可以是字母,数字(不在开头),下划线和句点. 句点只在非终结符中有意义.

在语法中书写终结符有三种方法:

- 一个*命名符号类型(named token type)*用类似 C 语言的标识符书写. 按照惯例,它们应该是大写字母. 每一个这种名称必须由一个 Bison 声明%token 定义. 参阅 符号类型的名称-Token Type Names.
- 一个*字符记号类型(character token type*(或者*文字字符记号(literal character token)* 用如同 C 语言字符常量相同的语法书写; 例如,'+'是一个字符记号类型. 除非你要指明字符记号类型的语义值类型(参阅语义值的数据类型-Data Types of Semantic Values 一章), 结合性或优先级(参阅操作符优先级-Operator Precedence 一章), 否则没有必要声明它们.

  按照惯例,一个字符记号类型只用于表示一个由特定字符组成的记号. 因此,记号类型'+'用于将字符`+'表示为一个记号. 没有对此惯例的强制要求, 但是如果你不按照惯例做, 你的程序会使其它的读者感到困惑.

  所有常用的 C 语言的字符转义序列都可以在 Bison 中使用, 但是你不能使用一个空字符作为一个字符文字. 因为它的数字码是 0,这表示输入结束.(参阅 yylex 的调用惯例-Calling Convention for yylex 一章). 并且,不像标准 C 三字符词(trigraphs)(注:由"??"开头的九种转义,为了在缺少标准 C 标点的宿主上使用标准 C,可参考标准 C 文档) 在 Bison 中并没有特殊意义并且反斜杠换行也是不允许的.

- 一个*文字串记号(literal string token)*用类似 C 语言中的字符串常量来书写; 例如,"<="是一个文字串记号. 除非你要指明文字串记号的语义值类型(参阅**值类型**-Value Type 一章), 结合性或优先级(参阅**优先级**-Precedence 一章),你没有必要声明它们.

  你可以使用%token(参阅**符号声明**-Token Declarations 一章)将文字串记号关联一个符号名称作为别名. 如果你不这样做,词法分析器必须从 yytname 表中重新找到文字串记号的代码. (参阅**调用惯例**-Calling Convention 一章).

  警告: 文字串记号在 Yacc 中不能工作.

  按照惯例,一个文字串记号只用于表示一个由特定串构成的记号. 因此,你用该使用类型"<="表示作为记号的字符串 `<='. Bison 并没有强制要求这种管理,但是如果你偏离了惯例, 阅读你程序的人会感到困惑.

  所有常用的 C 语言的字符转义序列都可以在 Bison 中使用, 但是你不能使用一个空字符作为一个字符文字. 因为它的数字码是 0,这表示输入结束. (参阅 yylex 的调用惯例-Calling Convention for yylex.) 并且,不像标准 C, 三字符词(trigraphs)(注:由"??"开头的九种转义,为了在缺少标准 C 标点的宿主上使用标准 C,可参考标准 C 文档) 在 Bison 中并没有特殊意义并且反斜杠换行也是不允许的. 一个文字串记号必须包括两个或更多个字符; 对于进包含一个字符的记号,你应该使用字符记号(参考以上).

怎样选择终结符的写法对于它的语法意义没有影响. 它只依赖于它出现在规则的什么地方以及什么时候分器起函数返回那个符号.

yylex 的返回值通常是终结符,除非返回一个代表结束输入的 0 或者负值. 无论你采用那种方法在语法规则中书写符号类型, 你应该在 yylex 的定义中采用相同的写法. 单字符符号类型的数字码简单地就是那个字符的正数编码, 所以,即使当 char 是有符号时你需要将它转换成 unsigned char 来避免主机上符号扩展 yylex 仍可以使用相同的值来产生必要的代码. 每一个命名记号类型在分析器文件中变为一个 C 宏, 所以 yylex 可以使用名称代表那个编码. (这就是为什么句点在终结符中不起作用.) 参阅 yylex 的调用惯例-Calling Convention for yylex.

如果 yylex 是在另外的文件中定义的, 你需要安排符号类型的宏定义在那里是可见的. 在你运行 Bison 的时候使用`-d'选项以便让它将这些宏定义写入一个另外的头文件`*name*.tab.h'. 你可以将它加入其它需要它的文件. 参阅 调用 Bison-Invoking Bison.

如果你要编写一个可以移植到任何标准 C 宿主上的语法, 你必须只能从基本标准 C 字符集中选择使用非零字符记号类型. 这个字符集由 10 个数字,52 个大小写英文字母,和在下列 C 语言字符串中的字符构成的:

```
"\a\b\t\n\v\f\r !\"#%&'()*+,-./:;<=>?[\\]^_{|}~"
```

yylex 函数和 Bison 必须为字符记号使用一个一致的字符集和编码. 例如,如果你在 ASCII 环境中运行 Bison, 但是在不兼容的例如 EBCDIC 的环境中编译和运行最终的程序, 最终程序可能不会工作.因为 Bison 产生的表格将字符记号假定为 ASCII 数字值. 发布带有 Bison 在 ASCII 环境中产生的 C 源文件的软件是标准的做法, 所以在与 ASCII 不兼容的平台的安装器必须在编译它们之前重新构建这些文件.

符号 error 是一个保留用作错误恢复的终结符(参阅**错误恢复**-Error Recovery 一章); 你不应该为了其它的目的而使用它. 特别地,yylex 永远不应该返回这个值(注:error). 除非你明确地在声明中赋予一个你的记号值为 256,否则 error 记号的默认值是 256.

3.3 描述语法规则的语法-Syntax of Grammar Rules

一个 Bison 语法规则通常有如下的下形式:

```
result: components...
      ;
```

reault 所在是这个规则描述的非终结符而 *components* 是被这个规则组合在一起的多种终结符和非终结符.(参阅符号-Symbols 一章)

例如:

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

表明两组 exp 类型和一个`+'记号在中间, 可以结合成一个更大的 exp 类型组.

规则中的空白只用来分隔符号.你可以在你希望的地方添加额外的空白.

决定规则的语义的*动作*可以分散在部件中.一个动组看起来是这样:

```
{C statements}
```

通常只有一个动作跟随着部件. 参阅 动作-Actions.

result 的多种规则可以分别书写或者由垂直条`|'按如下的方法连接起来:

在这种方式下依然有我们之特考虑的特殊规则.

如果一个规则的 *components* 为空,它意味着 *result* 可以匹配空字符串. 例如,这就是一个定一个由逗号分隔的 0 个或多个 exp 组:

```
expseq:   /* empty */ /* 空 */
      | expseq1
      ;


expseq1:  exp
      | expseq1 ',' exp
      ;
```

我们通常对每个没有部件的规则加上一个`/* empty */'的注释.

3.4 递归规则-Recursive Rules

一个规则被称为*递归的(recursive*)当它的 *result* 非终结符也出现在它的右手边. 因为这种方法是唯一可以定义一个特定事物的任意数字序列的方法, 几乎所有的 Bison 语法都需要使用递归. 考虑这个逗号分隔的一个或者多个表达式的递归定义:

```
expseq1:  exp
      | expseq1 ',' exp
      ;
```

由于 expseq1 的递归使用是在有手端的最左符号, 我们称这种递归为*左递归(left recursion)*. 相反地,这里有一个相同地使用*右递归(right recursion)*的定义.

```
expseq1:  exp
        | exp ',' expseq1
        ;
```

任意序列都可以使用作递归或者右递归定义, 但是通常你应该使用左递归, 因为它可以使用空间固定的栈来分析任意个数的元素序列. 由于即使规则只应用一次,在这之前,所有元素也都必须被移进到栈中, 右递归使用的 Bison 栈空间与序列中的元素个数成正比. 参阅 Bison 分析器算法-The Bison Parser Alogorithm.获得这方面的深入解释.

*间接(Indirect)*或者*相互(mutual)*递归当规则的结果没有在它的右手端出现但是出现在其它右手端的非终结符规则的右手端时候发生.

例如:

```
expr:    primary
       | primary '+' primary
       ;

primary:  constant
       | '(' expr ')'
       ;
```

定义了两个相互递归的非终结符,因为它们互相引用.

3.5 定义语言的语义-Defining Language Semantics

语言的语法规则之决定了语言的语法. 语义值却是由与多种记号和组相关联的语义值和当各种组被识别时执行的动作决定的.

例如,正是由于对每个表达式关联了正确的数值,计算器才可以正确的计算. 因为组`*x* + *y*`的动作是将 *x* 和 *y* 相关联的值相加, 所以计算器能正确地计算加法

3.5.1 语义值的数据类型-Data Types of Semantic Values

在一个简单的程序中,对所有的语言结构的语义值使用同一个数据类型就足够用了. 在 RPN 和中缀计算器的例子中的确是这样.(参阅逆波兰记号计算器-Reverse Polish Notation Calculator 一章).

Bison 默认是对于所有语义值使用 int 类型. 如果要指明其它的类型,可以像这样将 YYSTYPE 定义成一个宏:

```
#define YYSTYPE double
```

这个宏定义比喻在语法文件的 *Prologue* 部分. (参阅 Bison 语法大纲-Outline of a Bison Grammar 一章)

3.5.2 多种值类型-More Than One Value Type

在大多数程序中,你需要对不同种类的记号和组使用不同的数据类型. 例如,一个数字常量可能需要类型 int 或 long int, 而一个字符常量可能许需要类型 char *, 并且一个标识符需要一个指向符号表项的指针做为其语义值的类型.

为了在一个分析器中使用多种语义值类型, Bison 要求你做两件事情:

- 使用 Bison 声明%union 指明全部可能的数据类型集. (参阅值类型集-The Collections of Value Types 一章).
- 从这些类型中为每个符号(终结符或者非终结符)选择一个做为其语义值类型. 要做到这些,可以对记号使用 Bison 声明%token(参阅符号类型的名称-Token Type Names 一章); 并且对组使用 Bison 声明%type(参阅非终结符-Nonterminal Symbols 一章)

3.5.3 动作-Actions

当一个规则的实例被识别的时候, 同这个规则关联的包含 C 代码的动作会被执行. 大多数动作是用来从记号的语义值或者更小组的语义值计算整个组的语义值.

一个动作由包含在大括号之内的几个 C 语句构成,很像一个 C 语句块. 一个动作可以包含任意数量的 C 语句. 然而,Bison 并不搜索三字符词(trigraphs), 所以如果你的代码中使用了三字符词,你要保证它不影响嵌套的括号或者注释的边界,字符串或单个字符.

一个动作可以安放在规则的任意部分; 它就在那个位置执行. 大多数规则仅有一个在规则所有部件最后的动作. 在规则之中的动作是富有技巧性的并且仅用于特殊的目的 (参阅在规则中间的动作-Actions in Mid-Rule 一章).

动作中的 C 代码可以使用结构$*n* 来引用匹配规则的部件的语义值. 这个结构代表了第 *n* 个部件的值. 正在被构建的组的语义值为$$. 当这些被复制到分析器文件的时候,Bison 负责将这些结构翻译成适当类型的表达式. $$被翻译成一个可以修改的左值以便可以对它赋值.

这里是一个典型的例子:

```
exp:    ...
        | exp '+' exp
          { $$ = $1 + $3; }
```

这个规则从两个由加号连接的稍小的 exp 组构建一个 exp. 在这个动作中,$1 和$3 代着两个部件 exp 组的语义值. 它们是规则右手端第一个和第三个符号. 和被存储到$$以便成为刚刚被规则识别的加法表达式的语义值. 如果`+'记号有一个有用的语义值,可以通过$2 引用它.

注意到垂直杠字符`|'的确是一个分隔符, 并且动作只被附加到一个单一的规则上. 这是一点和 Flex 工具不同的地方. 在 Flex 中,`|'既代表"或者"也能代表"与下一个规则有着相同的动作". 在下面的例子中,动作仅在`b'被发现的时候触发.

```
a-or-b: 'a'|'b'   { a_or_b_found = 1; };
```

如果你并没有指明一个规则的动作,Bison 提供了默认的动作: $$ = $1. 因此,第一个符号的值变成了整个规则的值. 当然,仅当它们的数据类型相同时,默认动作才是有效的. 对于一个空规则,并没有有意义的默认动作; 除非这个规则的值无关紧要,否则每个空规则必须有明确的动作.

在$*n* 的 *n* 中使用 0 或负值是允许的. 这使用来引用在匹配当前规则*之前*的栈中记号或组. 这是一个十分冒险的尝试. 你必须明确当前应用的规则处于的上下文才能可靠地使用它. 这里有一个可靠地使用这种方法的例子:

```
foo:      expr bar '+' expr  { ... }
        | expr bar '-' expr  { ... }
        ;

bar:      /* empty */
        { previous_expr = $0; }
        ;
```

只要 bar 仅仅以上面展示的形式使用, $0 就总是引用先前于 bar 的 foo 的定义中的 expr.

3.5.4 动作中值的数据类型-Data Types of Values in Actions

如果你为语义值只选择了一种数据类型, 那么$$和$*n* 结构总是那种类型.

如果你已经使用了%union 指定了多种数据类型, 那么你必须从这些类型中为每一个可以有语义值的终结符或非终结符声明一种. 之后当你每次使用$$或者$*n* 的时时侯, 它的数据类型由它引用的符号的类型决定. 在这个例子中:

```
exp:    ...
        | exp '+' exp
            { $$ = $1 + $3; }
```

$1 和$3 引用了 exp 的实例, 所以它们都有为非终结符 exp 声明的数据类型. 如果使用了$2,它就会拥有为终结符'+'声明数据类型, 无论它可能是什么.

另外,在你引用数值的时候, 在引用的开始`$'之后插入`<*type*>', 你还可以指定数据类型, 例如,如果你已经定义了这里展示的数据类型:

```
%union {
  int itype;
  double dtype;
}
```

那么你可以用$<itype>1 作为一个整数来引用规则的第一个子单元, 或者用$<dtype>1 作为一个双精度数来引用.

3.5.5 规则中的动作-Actions in Mid-Rule

偶尔地,将一个动作放到规则之中是很用用处的. 这些规则的写法如同在规则之后的动作一样, 但是它们却在分析器识别之后的部件之前执行.

一个规则中动作可以通过$*n* 来引用在它之前的部件, 但是不能引用接下来的部件,因为这个动作在它们被分析之前执行.

规则中的动作自己也作为这个规则的一个部件. 这与在同一个规则的另一个动作有些不同(另一个动作通常在结尾): 当在$*n* 使用序号 *n* 的时候, 你必须把则个动作和符号一样也计算在内.

规则中的动作也可以拥有语义值. 这个动作可以通过向$$复制来设置它的值, 并且规则之中的后一个动作可应使用$*n* 引用这个值. 由于没有符号可以命名这个动作, 所以没有办法为这个值事先声明一个数据类型. 每当你引用这个值的时候你必须使用`$<...>*n*'结构来指明一个数据类型.

没有办法在规则中动作中为整个规则设定一个值, 因为对$$的赋值并没有那个效果(注:看上一段). 为整个规则赋值的唯一方法是通过规则末尾的普通动作实现.

这理有一个来自假想编译器的例子,用于处理 let 语句. 格式为`let (*variable*) *statement*' 并在 *statement* 生存期创建一个名为 *variable* 的临时变量. 为了分析这个结构, 当 *statement* 被分析的时候,我们必须将 *varible* 放入符号表, 并在稍后移除它. 这就是这个规则如何工作的:

```
stmt:   LET '(' var ')'
            { $<context>$ = push_context ();
              declare_variable ($3); }
        stmt    { $$ = $6;
              pop_context ($<context>5); }
```

一旦`let (*variable*)'已经被识别,第一个动作就执行. 它使用了数据类型联合中的 context, 并保存了一个当前语义上下文(可访问变量列表)的副本作为它的语义值. 然后调用 declare_variable 来向那个列表添加新的变量. 一旦第一个动作执行完毕, 嵌入的语句 stmt 就可以被分析了. 注意到规则中动作的编号是 5,所以`stmt'的编号为 6.

在嵌入的语句 stmt 被分析之后,它的语义值边变为了整个 let-语句的值.(注:$$=$6;) 之后,前一个动作的语义值(注$<context>5)被用来存储先前的变量列表.(注:pop_context ($<context>5)) 这将临时的 let-变量从列表中删除. 这样作的目的是让它不出现在分析程序的其余部分的时候.

由于分析器为了执行动作而进行强制分析, 在一个规则没有完全被识别之前执行动作经常会导致冲突. 例如,如下的两个规则, 并没有规则中的动作,可以在分析器中共存. 因为分析器可以移进一个左大括号并且查看之后跟随的符号来确定这是否是一个声明.

```
compound: '{' declarations statements '}'
        | '{' statements '}'
        ;
```

但是当我们向下面这样添加一个规则中的动作时,规则就失效了:

```
compound: { prepare_for_local_variables (); }
          '{' declarations statements '}'
        | '{' statements '}'
        ;
```

现在,当分析器还没有读到左到括号的时候,它就被迫决定是否执行规则中的动作. 换句话说,当分析器没有足够的信息作出正确选择的时候, 它就会强制使用一个或者其它的规则. (这个时候由于分析器仍在决定怎么办, 左大括号记号被称之为*超前扫描记号(look-ahead)*. 参阅 超前扫描记号-Look-Ahead Token.) (注:这个时候两个规则的超前扫描记号都是`{')

你可能认为给这两个规则放置相同的动作可以解决这个问题,就像这样:

```
compound: { prepare_for_local_variables (); }
        '{' declarations statements '}'
      | { prepare_for_local_variables (); }
        '{' statements '}'
      ;
```

但是这种方法不可行,因为 Bison 并没有意识到两个动作是完全相同的. (Bison 永远不会尝试理解动作中的 C 代码.)

如果语法是这样的:可以依靠第一个记号(C 语言就是这样)识别出一个声明. 那么将动作放到左大括号后是一个可行的解决方法. 就像这样:

```
compound: '{' { prepare_for_local_variables (); }
        declarations statements '}'
      | '{' statements '}'
      ;
```

现在接下来的声明或者语句的第一个记号(注:超前扫描记号) 在任何情况下都会告知 Bison 该使用哪一个规则.

另外一种解决方法是将动作放入一个做为子规则的非终结符.

```
subroutine: /* empty */
        { prepare_for_local_variables (); }
      ;



compound: subroutine
        '{' declarations statements '}'
      | subroutine
        '{' statements '}'
      ;
```

现在 Bison 不用却确定最终使用的规则就可执行规则 subroutine 中的动作. 注意到:现在动作已经在规则的结尾. 任何规则中间的动作都可以由这种方法转化为一个规则结尾的动作, 并且这也是 Bison 处理规则中间的动作的方法.

3.6 追踪位置-Tracking Locations

虽然语法规则和语义值对于编写功能完善的分析器来说足够用了, 但是处理一些额外的信息特别是符号位置的信息也是非常有用的.

处理位置的方法由一个数据类型和规则被匹配时执行的动作定义的.

描述位置的数据类型.
在动作中使用位置.
定义了一个计算位置的通用方法.

3.6.1 位置的数据类型-Data Type of Locations

由于所有的记号和组总是使用相同的类型, 定义一个位置的数据类型要比定义语义值的简单的多.

位置的类型由一个名为 YYLTYPE 的宏定义. 当 YYLTYPE 没有被定义的时候, Bison 使用含有四个成员的结构体作为默认的定义:

```
typedef struct YYLTYPE
{
  int first_line;   /* 第一行   */
  int first_column;  /* 第一列   */
  int last_line;    /* 最后一行 */
  int last_column;   /* 最后一列 */
} YYLTYPE;
```


3.6.2 动作和位置-Actions and Locations

动作不仅仅是为了定义语言的语义, 它还可以使用位置描述输出的分析器的行为.

最明显的为语法组建立位置的方法与计算语义值的方法相似. 在一个给定的规则中, 多种结构可以用于访问被匹配元素的位置. 右手端第 *n* 个部件的位置是@*n* 而左边的组的位置是@$.

这是一个基本的使用位置的默认数据类型的例子:

```
exp:    ...
        | exp '/' exp
          {
            @$.first_column = @1.first_column;
            @$.first_line = @1.first_line;
            @$.last_column = @3.last_column;ip
            @$.last_line = @3.last_line;
            if ($3)
              $$ = $1 / $3;
            else
              {
                $$ = 1;
                fprintf (stderr,
                         "Division by zero, l%d,c%d-l%d,c%d",
                         @3.first_line, @3.first_column,
                         @3.last_line, @3.last_column);
              }
          }
```

就像对语义值一样,有一个每当规则被匹配时的位置的默认动作. 它将@$的开始设置为第一个符号的开始并将@$的末尾设为最后一个符号的末尾.

可以通过使用这个默认的动作来实现位置追踪的全自动. 上面的例子可以简单地这样重写:

```
exp:    ...
        | exp '/' exp
          {
            if ($3)
              $$ = $1 / $3;
            else
              {
                $$ = 1;
                fprintf (stderr,
                         "Division by zero, l%d,c%d-l%d,c%d",
                         @3.first_line, @3.first_column,
                         @3.last_line, @3.last_column);
              }
          }
```


3.6.3 位置的默认动作-Default Action for Locations

实际上,动作并不是计算位置的最好地方. 由于位置比语义值更普遍, 所以在输出的分析器里有用来重定义每个规则的默认动作(注:对位置的)的空间. YYLLOC_DEFAULT 宏每当一个规则被匹配而关联的动作尚未执行之前被调用. 当处理一个语法错误要计算错误的位置的时候,它也会被调用.

大多数时候, 这个宏足以胜过语义动作中专注于位置的代码.

YYLOC_DEFAULT 宏带有三个参数. 第一个是组(计算结果)的位置. 当匹配一个规则时, 第二个参数标识了正在匹配的规则的所有右端元素的位置, 第三个参数是规则右端元素的大小; 当处理一个语法错误的时候, 第二个参数标识了在错误处理中丢弃的符号的位置, 第三个参数是丢弃符号的数量.

YYLOC_DEFAULT 默认地被这样定义:

```
# define YYLLOC_DEFAULT(Current, Rhs, N)                        \
    do                                                          \
      if (N)                                                    \
        {                                                       \
          (Current).first_line   = YYRHSLOC(Rhs, 1).first_line;        \
          (Current).first_column = YYRHSLOC(Rhs, 1).first_column;      \
          (Current).last_line    = YYRHSLOC(Rhs, N).last_line;         \
          (Current).last_column  = YYRHSLOC(Rhs, N).last_column;       \
        }                                                       \
      else                                                      \
        {                                                       \
```

```
      (Current).first_line   = (Current).last_line   =            \
        YYRHSLOC(Rhs, 0).last_line;                               \
      (Current).first_column = (Current).last_column =            \
        YYRHSLOC(Rhs, 0).last_column;                             \
    }                                                             \
  while (0)
```

当 *k* 是正数的时候, 函数 YYRHSLOC (rhs,k)返回的是第 *k* 个符号的位置. 当 *k* 和位置 *n* 都为零的时候, YYRHSLOC(rhs,k)返回的是刚刚被归约的符号的位置.

当要自己定义宏 YYLLOC_DEFAULT 的时候,你要考虑以下几点:

- 所有的参数应该不受左端/右端的限制. 然而,只有第一个(结果)应该被 YYLOC_DEFAULT 修改.
- 出于与语义动作一致性的考虑, 右手端有效索引的范围应该是从 1 到 *n*. 当 *n* 是 0 的时候,只有 0 是有效索引并且它引用的是归约前的那个符号. 在错误处理中,*n* 总是正数.
- 因为实际的参数可能不在括号内, 如果需要的话,你应该加参数放在括号内. 同样地,当你的宏后紧跟一个分号时, 它应该被展开成可作为单独语句使用的东西.

3.7 Bison 声明-Bison Declarations

*Bison 声明(Bison declarations)*部分定义了用来描述语法的符号和语义值的数据类型. 参阅 符号-Symbols.

所有符号类型名称(但不是如'+'和'*'的单字符记号)必须被声明. 如果你要指明非终结符语义值的数据类型的话,那么它们也必须被声明 (参阅多种值类型-More Than One Value Type 一章).

默认地,文件的第一个规则也指明了开始符号. 如果你要其它的符号作为开始符号,你必须显示显式地声明它. (参阅语言与上下文无关文法-Language and Context-Free Grammars 一章)

3.7.1 符号类型名称-Token Type Names

声明符号类型(终结符)的最基本的方法如下:

```
%token
name
```

Bison 会在分析器中将这个声明转换成#define 指令以便 yylex(如果在这个文件中使用了它) 可以是用名称 *name* 代表这个记号类型码.

另外,如果你要指明结合性和优先级, 你可以使用%left,%right 或者%nonassoc 代替%token. 参阅 操作符优先级-Operator Precedence.

你可以依靠附加一个十进制活十六进制整数紧跟着记号名称来显式地指定一个符号类型的数字码.

```
%token NUM 300
%token XNUM 0x12d // a GNU extension 一个 GNU 扩
展
```

然而,我们通常最好让 Bison 选择所有记号类型的数字码. Bison 会自动地选择互不冲突或不与其它字符冲突的码.

当栈类型是一个联合体的时候, 你必须使用%token 或者其它记号声明来指明记号的语义值类型. 语义值类型由中括号分隔. (参阅多种值类型-More Than One Value Type 一章).

例如:

```
%union {            /* define stack type */ /* 定义栈类型 */
  double val;
  symrec *tptr;
}
%token <val> NUM      /* define token NUM and its type */ /* 定义一个记号 NUM 和它的数
据类型 */
```

将文字串写在%token 声明的结尾, 你可以把一个符号类型名称关联到文字串记号上. 例如:

```
%token arrow "=>"
```

例如,一个 C 语言语法可以用同等的文字串记号指明这些名称

```
%token  <operator>  OR     "||"


%token  <operator>  LE 134  "<="
%left  OR  "<="
```

一旦你使文字串和符号名称等价, 你可以在以后的声明或者语法规则中交替地使用它们. yylex 函数可以使用符号名称或者文字串来获得符号类型数字码. (参阅调用惯例-Calling Convention 一章).

3.7.2 操作符优先级-Operator Precedence

使用%left,%right 或者%nonassoc 可以一次声明一个记号并指明它的优先级和结合性. 这些被称做*优先级声明 (precedence declarations).* 获取更多这方面的信息,参阅 操作符优先级-Operator Precedence.

优先级的声明与%token 的声明相同,或者是

```
%left symbols...
```

或者是

```
%left <type> symbols...
```

这些声明都与%token 的目的相同. 但是额外地,它们还指明了 *symbols* 的结合性和相对优先级:

- 操作符的结合性决定了如何重复使用嵌套的操作符: `*x op y op z*' 是先组合 *x* 和 *y*,还是先组合 *y* 和 *z*. %left 指明左结合性(先组合 *x* 和 *y*)而 %right 指明了有结合性(先组合 *y* 和 *z*). %nonassoc 指明了无结合性,即`*x op y op z*' 被认为是一个语法错误.
- 一个操作符的优先级决定了它如何与另外的操作符嵌套使用. 在一个优先级声明中声明的所有记号有相同的优先级, 如何嵌套使用它们取决于它们的结合性. 当两个记号在不同的优先级声明中,稍晚声明的拥有更高的优先级,并且先被组合.

3.7.3 值类型集-The Collection of Value Types

%union 声明指明了语义值全部可能的数据类型集. 关键字%union 后紧跟包含了 C 语言 union 同样的东西的一对大括号. 例如:

```
%union {
  double val;
  symrec *tptr;
}
```

这说明了两个可选择的类型是 double 和 symrec*. 它们被赋予了名称 val 和 tptr; 这些名称用于在%toke 和%type 声明中为终结符或非终结符选择一个类型. (参阅**非终结符-Nonterminal Symbols** 一章).

作为一个 POSIX 扩展,一个标志被允许紧跟在 union 后. 例如:

```
%union value {
  double val;
  symrec *tptr;
}
```

指明了联合题标志 value,所以相应的 C 类型为 union value. 如果你不知名一个标志,它默认地就为 YYSTYPE.

我们应该注意到,并不像 C 语言中的 union 声明一样, 在 Bison 中,你不需要在大括号结束的时候写上分号.

3.7.4 非终结符-Nonterminal Symbols

当你使用%union 指明多种值类型的时候, 你必须为每个要使用其语义值的非终结符指明一个值类型. 通过%type 可以做到这一点,像这样:

```
%type <type> nonterminal...
```

这里的 *nonterminal* 是非终结符的名称, *type* 是在%union 中给定的名称来指定该非终结符的语义值类型. (参阅**值类型集-The Collection of Value Types** 一章). 你可以给任意多数量的非终结符以相同的数据类型, 如果它们有相同的值类型的话. 这时我们要使用空白来分隔符号名称.

你也可以声明一个终结符的值类型. 为终结符使用相同的<*type*>结构可以做到这一点. 所有种类的记号声明都允许使用 <*type*>.

3.7.5 在分析执行前执行一些动作-Performing Actions before Parsing

有些时候,你的分析器需要在分析之前执行一些初始化. 通过使用%initial-action 指令指定这种代码.

指令: **%initial-action** *{ code }*

声明了 *code* 必须在每次调用 yyparse 之前被调用. *code* 可以使用$$和@$ -- 超前扫描记号的初始值和位置 -- 和 %parse-param.

例如,如果你的位置需要使用一个文件名,你可以使用

```
%parse-param { const char *filename };
%initial-action
{
  @$.begin.filename = @$.end.filename = filename;
};
```


3.7.6 释放被丢弃的符号-Freeing Discarded Symbols

分析器可能会丢弃一些符号. 例如,在错误恢复中(参阅**错误恢复**-Error Recovery 一章), 分析器丢弃已经压入栈中为难符号, 以及来自剩余文件的为难记号直到脱离错误恢复状态。如果这些符号带有堆信息,这些内存就会出现丢失. 然而这种行为对于例如编译器一样的批分析器是可以容忍的, 但不适用于可能"没有终点"的分析器如 shells 或者通信协议的实现.

%destructor 指令允许定义当一个符号被丢弃时调用的代码.

指令: **%destructor** *{ code } symbols*

声明了 *code* 必须在每次分析器丢弃 *symbols* 时调用. *code* 应该使用$$来指明与 *symbols* 关联的语义值. 其余的分析器参数也是可用的. (参阅**分析器函数** yyparse-The Parser Funcation yyparse 一章).

警告:对于版本 1.875 来说,这个特征仍然是实验性的. 主要原因是没有足够的用户反馈. 相应的语法仍可能会改变.

例如：

```
%union
{
  char *string;
}
%token <string> STRING
%type  <string> string
%destructor { free ($$); } STRING string
```

保证了当一个 STRING 或者一个 string 将被丢弃时, 相关的内存也会被释放.

注意到在将来,Bison 会认为在动作中没有提及的右端成员也可以被销毁. 例如,在:

```
comment: "/*" STRING "*/";
```

分析器有权销毁 string 的语义值. 当然,这不适用于默认动作: 比较:

```
typeless: string;  // $$ = $1 does not apply; $1 is destroyed.
```

```
typefull: string;  // $$ = $1 applies, $1 is not destroyed.
```

被丢弃的符号(Discarded symbols) 是如下几种:

- 在第一阶段的错误恢复中栈弹出符号.
- 在第二阶段错误恢复中要到达的终结符.
- 当分析器异常终止时(或者通过显式地调用 YYABORT,或者一系列失败的错误恢复), 当前的超前扫描记号.

3.7.7 消除冲突警告-Suppressing Conflict Warnings

通常情况下,当出现任何的冲突的时候,Bison 会作出警告 (参阅移进/归约冲突-Shift/Recude Conflicts 一章), 但是大多数真正的语法含有的是可以通过预测的方法解决并且很难消除的无害的移进/归约冲突. 除非冲突的数量改变,我们渴望消除关于这些冲突的警告. 你可以使用%expect 声明做到这一点.

声明看起来是这样:

```
%expect n
```

这里的 *n* 是一个十进制整数. 这个声明表明:如果有 *n* 个移进/归约冲突并且没有归约/归约冲突, Bison 并不会作出警告. 如果有更多或更少的冲突或者有归约/归约冲突,Bison 仍会作出警告.

对于通常的 LALR(1)分析器,归约/归约冲突更加棘手并且应该完全消除. Bison 对于这些分析器总会报告归约/归约冲突. 对于 GLR 分析器来说,移进/归约冲突和归约/归约冲突都是很平常的情况(否则,就没有必要使用 GLR 分析). 因此,在 GLR 分析器中使用如下声明指定一个预期数目的归约/归约冲突也是可以的:

```
%expect-rr n
```

通常来说,使用%expect 包括了这些步骤:

- 不使用%expect 而编译你的代码. 使用`-v'选项获取冲突发生的列表. Bison 也会打印冲突的个数.
- 检查每一个冲突来确定 Bison 默认的解决方法是你真正想要的. 如果不是,重写语法并回到开始.(注:第一步)
- 添加一个%expect 声明,从 Bison 打印的列表复制一个冲突的数目.

现在,如果你不更改冲突的数目,Bison 会停止打扰你. 但是如果你改变了语法导致了更多或更少的冲突, Bison 仍会警告你.

3.7.8 开始符号-The Start-Symbol

Bison 默认地认为在语法叙述部分指明的第一个非终结符为语法的开始符号. 程序员可以使用如下的%start 声明来克服这个约束

```
%start symbol
```

3.7.9 纯(可重入)分析器-A Pure (Reentrant) Parser

一个*可重入(reentrant)*程序是在执行过程中不变更的程序; 换句话说,它全部由*纯(pure)*(只读)代码构成. 当可异步执行的时候, 可重入特性非常重要. 例如,从一个句柄调用不可重入程序可能是不安全的. 在带有多线程控制的系统中, 一个非可重入程序必须只能被互锁(interlocks)调用.

通常地,Bison 生成不可重入的分析器. 这对大多数情况足够用了,并且这种分析器提供了与 Yacc 的兼容性. (由于在 yylex,yylval 和 yyloc 通信中使用了静态分配的变量, 标准 Yacc 界面是不可重入的).

作为另外一个选择,你可以声称一个纯,可重入的分析器. Bison 声明%pure-parse 表明你要产生一个可重入的分析器. 这个声明是这样:

```
%pure-parser
```

这样做的结果是 yylval 和 yylloc 的通信变量变为一个 yyparse 中的局部变量, 并且对词法分析器函数 yylex 使用了不同的调用惯例. 参阅 **纯分析器的调用惯例**-Calling Convention for Pure Parsers.以获取更多信息. 变量 yynerrs 也变为在 yyparse 中的局部变量 (参阅**错误报告函数** yyerror-The Error Reporting Funcation yyerror 一章). yyparse 自己的调用惯例并没有改变.

分析器是否为纯分析器与语法规则毫不相关. 你可以从任何有效的语法产生一个纯分析器或者不可重入分析器.

3.7.10 Bison 声明总结-Bison Declaration Summary

这是一个用来定义语法的声明的总结:

指令: **%union**

声明了语义值可能拥有的数据类型集. (参阅**值类型集**-The Collection of Value Types 一章).

指令: **%token**

声明一个未指定优先级和结合性的终结符(符号类型名称) (参阅**符号类型名称**-Token Type Names 一章).

指令: **%right**

声明一个右结合的终结符(符号类型名称) (参阅**操作符优先级**-Operator Precedence 一章).

指令: **%left**

声明一个左结合的终结符(符号类型名称) (参阅**操作符优先级**-Operator Precedence 一章).

指令: **%nonassoc**

声明一个没有结合性的终结符(符号类型名称). (参阅**操作符优先级**-Operator Precedence 一章). 按结合性的方法使用它是一个语法错误.

指令: **%type**

声明非终结符的语义值类型. (参阅**非终结符**-Nonterminal Symbols 一章).

指令: **%start**

指明了语法的开始符号 (参阅**开始符号**-The Start-Symbol 一章).

指令: **%expect**

声明了预期的移进/归约冲突的个数. (参阅**消除冲突警告**-Suppressing Conflict Warnings 一章).

为了改变 bison 的行为,使用如下指令

指令: **%debug**

在分析器文件中,如果 YYDEBUG 未定义,将其定义为 1, 以便调式机制被编译.

参阅 **追踪你的分析器**.

指令: **%defines**

编写一个包括记号类型定义和其它声明的宏定义头文件. 如果分析器输出文件是`*name*.c', 那么这个头文件就是`*name*.h'.

除非 YYSTYPE 已经被定义成了一个宏, 否则输出头文件会声明 YYSTYPE. 因此,如果你使用了需要其它定义的%union 部件 (参阅 **多种值类型**-More Than One Value Type,) 或者你已经定义了宏 YYSTYPE (参阅**语义值的数据类型**-Data Types of Semantic Values 一章), 你需要安排这些定义使它们在所有模块的前页, 例如,把它们放入一个你的分析器和任何其它模块都包含的头文件中.

除非你的分析器是一个纯分析器, 否则输出的头文件将 yylval 声明为一个外部变量. 参阅 **一个纯(可重入)分析器**-A Pure (Reentrant) Parser.

如果你也使用了位置, 输出头文件使用声明与 YYSTYPE 和 yylval 类似的协议声明 YYLTYPE 和 yylloc. 参阅 **追踪位置**-Semantic Values of Tokens.

如果你希望将 yylex 的定义放在一个另外的源文件中的话, 这个输出的头文件是通常必须的, 因为 yylex 需要引用头文件中提供的声明和记号类型码. 参阅 **记号的语义值**-Semantic Values of Tokens.

指令: **%destructor**

指明了分析器如何回收同丢弃的符号相关联的内存. 参阅 **释放丢弃的符号**-Freeing Discarded Symbols.

指令: **%file-prefix="*prefix*"**

指定一个所有 Bison 输出文件的前缀,就好像输入文件名为`*prefix*.y'.

指令: **%locations**

产生处理位置的代码(参阅**使用动作的特殊特征**-Special Features for Use in Actions 一章). 一旦语法使用了`@*n*'记号,这种模式就会被激活. 但是如果你的语法没有用到它,使用`%locations'可以获得更精确的语法错误信息.

指令: **%name-prefix="*prefix*"**

重命名分析器使用的外部符号以便它们以 *prefix* 开始,而不是`yy'. 符号被重命名的精确列表是: yyparse, yylex, yyerror, yynerrs, yylval, yylloc, yychar, yydebug, 和可能使用的 yylloc. 例如, 如果你使用`%name-prefix="c_"', 名称就会变为 c_parse, c_lex 等等. 参阅 **同一个程序中的多个分析器**-Multiple Parsers in the Same Program.

指令: **%no-parser**

在分析器文件中不包含任何 C 代码,仅仅声称表格. 分析器文件仅仅包括#define 指令和静态变量声明.

这个选项也告诉 Bison 将语法动作的 C 代码以 switch 语句的形式写入一个名为`*filename*.act'的文件.

指令: **%no-lines**

在分析器文件中不生成任何#line 预处理指令. Bison 通常将这些指令写入分析器文件以便 C 编译器和调式器(debugger)可以将错误和目标代码与你的源文件(语法文件)关联起来. 这个指令使它们关联错误到分析器文件并将它视为一个独立的源文件.

指令: **%output="*filename*"**

将分析器文件指定为 *filename*.

指令: **%pure-parser**

请求一个纯(可重入)分析器程序(参阅**一个纯(可重入)分析器**-A Pure (Reentrant) Parser 一章).

指令: **%token-table**

在分析器文件中生成一个记号名称数组. 数组的名称是 yytname; yytname[*i*]是 Bison 内部数字码为 *i* 的记号的名称. 前三个 yytname 的元素与预定义记号"$end","error", 和"$undefined"相对应; 在这几个符号之后便是语法文件中定义的符号.

对于单字符记号和文字串记号, 表格中的名称包含单引号或者双引号字符: 例如,"'+'"是一个单字符记号而 "\"<=\""是一个文字串记号. 字符串记号的所有字符一字不差地出现在符号表中; 即使双引号字符也不跳过. 例如,如果记号包含了三个字符`*"*',在 yytname 的字符串为`"*"*"'. (在 C 语言中,那应被写成"\"*\"*\"").

当你指定了%token-table,Bison 也生成了 YYNTOKENS, YYNNTS, and YYNRULES, 和 YYNSTATES 的宏定义:

YYNTOKENS

最高记号数字加 1

YYNNTS

非终结符的数量

YYNRULES

语法规则的数量

YYNSTATES

分析器状态的数量(参阅**分析器状态**-Parser States 一章).

指令: **%verbose**

向一个额外的输出文件写入包括分析器状态和对在那个状态的每一种超前扫描记号做了些什么的详细描述. 参阅**理解你的分析器**-Understanding You Parser,以获取更多信息.

指令: **%yacc**

假定给定了`--yacc'选项,也就是模拟 Yacc,包括它的命名惯例. 参阅 Bison **选项**-Bison Options,获得更多信息.

3.8 在同一个程序中使用多个分析器-Multiple Parsers in the Same Program

大多数程序使用 Bison 仅分析一种语言,因此仅包含一个 Bison 分析器. 但是如果你要在程序中分析多种语言该怎么办? 这样的话,你需要避免在不同的 yyparse,yylval 等等不同定义的名称之间的冲突.

要做到这一点,最简单的方法就是使用`-p *prefix*'选项 (参阅**调用** Bison-Invking Bison 一章). 这个选项重命名了接口函数和 Bison 分析器变量, 使它们以 *prefix* 开头而不是`yy'. 你可以使用这个选项给予每个分析器互不冲突的独特的名称.

重命名符号的精确列表为: yyparse, yylex, yyerror, yynerrs, yylval, yylloc, yychar 和 yydebug. 例如,如果你使用了`-pc', 名称就变为 cparse,clex 等等.

所有其它与 Bison 相关的变量和宏定义并没有被重命名. 这些其它的东西并不是全局的; 所以在不同的分析器中使用相同的名称不不会产生冲突. 例如,YYSTYPE 并未被重命名, 但是在不同的分析器中以不同的方式不冲突地定义 (参阅**语义值的数据类型**-Data Types of Semantic Values 一章).

`-p'选项靠向分析器文件的开头添加宏定义的方式工作. 定义 yyparse 为 *prefix*parse,等等. 这种方式高效地在整个分析器文件中相互替代名称.

---

4. 分析器 C 语言接口-Parser C-Language Interface

Bison 分析器实际上是一个名为 yyparse 的 C 语言函数. 这里我们描述一下 yyparse 和它需要用到的函数的接口惯例.

你应该记住,分析器使用了很多以`yy'和`YY'开头的标识符. 如果你在动作或者 *epilogue* 部分使用了这样一个标识符(不在这个手册之中), 你的程序可能会遇到麻烦.

4.1 分析器函数 yyparse-The Parser Function yyparse

你通过调用函数 yyparse 开始进行分析. 这个函数读入记号,执行动作, 并且最后如果它遇到输入结束或者不能恢复的错误就会返回. 你也可以编写一个让 yyparse 立即返回不再读入的动作.

Function: int **yyparse** *(void)*

如果分析成功,yyparse 返回值为 0(当遇到输入结束的时候).

如果分析失败,返回值则为 1.(当遭遇语法错误的时候).

在动作中,你可以使用这些宏使 yyparse 立即返回:

Macro: **YYACCEPT**

立即返回 0(来报告分析成功).

Macro: **YYABORT**

立即返回 1(来报告分析失败).

如果你使用一个可重入的分析器, 你还可用可重入的方式以向它传送额外的信息. 为了做到这一点,使用%parse-param 声明:

指令: **%parse-param** *{argument-declaration}*

表明由 argument-declaration 声明的参数是一个额外的 yyparse 参数. *argument-declaration* 在声明函数或者原型时使用. *argument-declaration* 中最后一个标识符必须为参数名称.

这里有一个例子.将这些写入分析器:

```
%parse-param {int *nastiness}
%parse-param {int *randomness}
```

然后向这样调用分析器

```
{
  int nastiness, randomness;
  ...  /* Store proper data in nastiness and randomness.  */
  value = yyparse (&nastiness, &randomness);
```

```
  ...

}
```

在语法动作中,用类似这样的表达式来引用数据:

```
exp: ...    { ...; *randomness += 1; ... }
```


4.2 词法分析器函数 yylex-The Lexical Analyzer Function

yylex

*词法分析器(lexical analyzer)*函数,yylex, 从输入流中识别记号并将它们返回给分析器(注:语法分析器). Bison 并不自动生成这个函数; 你必须编写它以备 yyparse 调用. 这个函数有时候也被成为词法扫描器.

在简单的程序中,yylex 经常定义在 Bison 语法文件的末尾. 如果 yylex 定义在另外的文件中, 你需要安排符号类型宏定义在那里是可见的. 为了做到这一点,在运行 Bison 的时候使用`-d'选项以便它将这些宏定义写入到另外的名为`*name*.tab.h'的头文件中. 你可以将它包含在需要它的其它源文件中. 参阅 调用-Bison-Invoking Bison.

4.2.1 yylex 的调用惯例-Calling Convention for yylex

yylex 的返回值必须是它刚刚发现的记号类型的正值数字码; 0 或负值代表着输入的结束.

当一个记号在语法规则中由它的名称引用时, 这个名称在语法文件中是一个宏, 这个宏定义了那个记号类型的恰当的数字码. 所以 yylex 可是使用这个名称来指明那个记号类型. 参阅 符号-Symbols.

当一个记号在语法文件中由一个字符引用时, 那个字符的数字码同样也是那个记号类型的数字码. 所以 yylex 可以简单地返回那个字符码, 并且可能转换为 unsigned char 以避免符号扩展. 但空字符绝对不能这样使用, 因为它的数字吗为 0, 这意味这输入的结束.

这里是一个展示这些东西的例子:

```
int
yylex (void)
{
  ...
```

```
  if (c == EOF)    /* Detect end-of-input.  */ /* 检测到输入结束 */
    return 0;
  ...
  if (c == '+' || c == '-')
    return c;      /* Assume token type for `+' is '+'.  */ /* 认定`+'的记号类型就是'+' */
  ...

  return INT;      /* Return the type of the token.  */ /* 返回记号的类型 */
  ...
}
```

设计这种接口的目的是为了可以不加更改地使用 lex 工具的输出 yylex.

如果语法使用了文字串记号, yylex 决定记号类型马的方法有两种:

- 如果语法定义了文字串记号的符号名称别名, yylex 可以像其它符号名称一样使用这些符号名称. 这种情况下, 语法文件中使用文字串记号对 yylex 没有影响.
- yylex 可以在 yytname 表中找到多字符记号. 这个记号的索引是这个记号的类型码. 多字符记号的名称由一个双引号,记号的字符和另外一个双引号记录在 yytname 中. 无论如何,记号(注:多字符记号)的字符不能是转义的; 它一字不差地出现在表格字符串的内容里.

  这里有在 yytname 中搜索记号的代码. 这个代码假定记号的字符存储在 token_buffer 中.

```
for (i = 0; i < YYNTOKENS; i++)
  {
    if (yytname[i] != 0
        && yytname[i][0] == '"'
        && ! strncmp (yytname[i] + 1, token_buffer,
                      strlen (token_buffer))
        && yytname[i][strlen (token_buffer) + 1] == '"'
        && yytname[i][strlen (token_buffer) + 2] == 0)
      break;
  }
```

  yytname 表格只在你使用了%token-table 声明才会生成. 参阅 声明总结-Decl Summary.

4.2.2 记号的语义值-Semantic Values of Tokens

在一个普通的(不可重入)的分析器中, 记号的语义值必须被存放在全局变量 yylval 中. 当你只使用一种语义值数据类型时, yylval 就是那个类型. 因此,如果类型为 int(默认的), 你可以这样编写你的 yylex:

```
...
yylval = value;  /* Put value onto Bison stack.  */ /* 将值放入 Bison 栈中 */
return INT;      /* Return the type of the token.  */ /* 返回记号类型 */
...
```

当你使用多种数据类型时, yylval 的类型是一个由%union 声明组成的联合体. (参阅**值类型集**-The Collections of Value Types 一章). 所以,当你存储一个记号的语义值的时候, 你必须使用恰当的联合体成员. 如果%union 声明是这样的:

```
%union {
  int intval;
  double val;
  symrec *tptr;
}
```

那么 yylex 中的代码应该是这样:

```
...
yylval.intval = value; /* Put value onto Bison stack.  */ /* 将值放入 Bison 栈中. */
return INT;            /* Return the type of the token.  */ /* 返回记号的类型 */
...
```

4.2.3 记号的文字位置-Textual Locations of Tokens

如果你在动作中使用了`@*n*'-特征(参阅**追踪位置**-Tracking Locations 一章)来追踪记号和组的文字位置, 那么你必须在 yylex 中提供这些信息. yyparse 预期在全局变量 yyloc 中找到刚刚分析的记号的文字位置. 所以 yylex 必须在那个变量里存放正确的数据.

默认地,yyloc 的值是一个结构体并且你只需要初始化将被动作使用的成员. 四个成员分别是 first_line, first_column, last_line 和 last_column. 注意到:使用这个特征会使分析器的性能显著下降.

yyloc 的数据类型为 YYLTYPE.

4.2.4 纯分析器的调用惯例-Conventions for Pure Parsers

当你使用 Bison 声明%pure-parser 要求得到一个纯,可重入的分析器, 全局通信变量 yylval 和 yylloc 不能继续使用. (参阅**一个纯(可重入)分析器**-A Pure (Reentrant Parser.) 在这种分析器中,两个全局变量由传递给 yylex 的指针参数取代. 你必须如下你声明它们,并通过这些指针存储数据最后将它们传回.

```
int
yylex (YYSTYPE *lvalp, YYLTYPE *llocp)
{
```

```
  ...
  *lvalp = value;  /* Put value onto Bison stack.  */ /* 将值放入 Bison 栈 */
  return INT;      /* Return the type of the token.  */ /* 返回记号类型 */
  ...
}
```

如果语法们文件没有使用`@'结构引用文字位置, 那么类型 YYLTYPE 就不会被定义. 在这种情况下,省略第二个参数; 仅用一个参数调用 yylex.

如果你希望传递额外的数据到 yylex, 可以是用%lex-param,就像%parse-param 一样 (参阅分析器函数-Parser Function 一章).

指令: **lex-param** *{argument-declaration}*

声明 argument-declaration 是一个额外的 yylex 参数.

例如:

```
%parse-param {int *nastiness}
%lex-param   {int *nastiness}
%parse-param {int *randomness}
```

导致了如下的结果:

```
int yylex   (int *nastiness);
int yyparse (int *nastiness, int *randomness);
```

如果添加了%pure-parser:

```
int yylex   (YYSTYPE *lvalp, int *nastiness);
int yyparse (int *nastiness, int *randomness);
```

最后,如果%pure-parser 和%locations 都被使用:

```
int yylex   (YYSTYPE *lvalp, YYLTYPE *llocp, int *nastiness);
int yyparse (int *nastiness, int *randomness);
```

4.3 错误报告函数 yyerror-The Error Reporting Function

yyerror

Bison 分析器侦测到一个 *语法错误(syntax error)* 或者一个 *分析错误(parse error)* 每当它读入了一个不能满足任何规则的记号. 一个语法动作也可以使用宏 YYERROR 显式地声明一个错误 (参阅使用动作的特殊特征-Special Features for Use in Actions 一章).

Bison 分析器期望靠调用一个名为 yyerror 的错误处理报告函数报告错误. 这个函数必须由你提供. 每当 yyparse 发现一个语法错误的时候, yyparse 就会调用它. yyparse 只接受一个参数. 对于一个语法错误, 显示的字符串通常是"syntax error".

如果你在 *Bison declarations* 部分 (参阅 *Bison Declarations* 部分-The Bison Declarations Section 一章) 使用了%error-verbose 指令, 那么 Bison 会提供更加详细而明确的错误信息而不是仅有"syntax error".

分析器可以侦测到另外一种错误:栈溢出. 这在输入包含非常深层次的嵌套结构时发生. 你很难遇到这种情况, 因为 Bison 会自动将栈容量扩展到一个很大的极限. 但是如果溢出发生, yyparse 会以通常的格式调用 yyerror 并带有字符串"parser stack overflow".

下面的定义对于简单的程序足够用了:

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

当 yyerror 返回到 yyparse 后, 如果你以已经写好了恰当的错误恢复语法规则(参阅**错误恢复**-Error Recovery 一章), yyparse 会尝试进行错误恢复. 如果恢复是不可能的,yyparse 会立即返回 1.

显然,在带有错误追踪的纯分析器中, yyerror 应该会访问当前的位置. 由于历史原因,这些的确是 GLR 分析器的事情而不是 Yacc 分析器的事情. 例如,如果传递了`%locations %pure-parser',那么 yyerror 的原型是:

```
void yyerror (char const *msg);               /* Yacc parsers.  */ /* Yacc 分析器 */
void yyerror (YYLTYPE *locp, char const *msg);  /* GLR parsers.   */ /* GLR 分析器 */
```

如果使用了`%parse-param {int *nastiness}',那么原型是:

```
void yyerror (int *nastiness, char const *msg);  /* Yacc parsers.  */ /* Yacc 分析器 */
void yyerror (int *nastiness, char const *msg);  /* GLR parsers.   */ /* GLR 分析器 */
```

最终,GLR 和 Yacc 分析器对绝对的纯分析器共享相同的 yyerror 调用惯例, 例如,当 yylex *和*%pure-parse 的调用惯例是纯调用,例如:

```
/* Location tracking.  */ /* 追踪位置 */
%locations
/* Pure yylex.  */ /* 纯 yylex */
%pure-parser
%lex-param   {int *nastiness}
/* Pure yyparse.  */ /* 纯 yyparse */
%parse-param {int *nastiness}
%parse-param {int *randomness}
```

导致了如下用于所有种类分析器的原型:

```
int yylex (YYSTYPE *lvalp, YYLTYPE *llocp, int *nastiness);
int yyparse (int *nastiness, int *randomness);
void yyerror (YYLTYPE *locp,
              int *nastiness, int *randomness,
              char const *msg);
```

原型只是用来指明 Bison 产生的代码如何使用 yyerror. Bison 产生的代码通常忽略返回值, 所以 yyerror 可以返回任何类型, 包括 void. 并且 yyerror 可以是一个变参函数(variadic funcation), 这就是为什么消息总在最后传递的原因.

yyerror 在传统上返回一个经常被忽略的 int, 但这仅仅出于纯历史的原因. void 是更好的选择, 因为它更精确的反应了 yyerror 的返回类型.

变量 yynerrs 包含了到目前位置遭遇的语法错误的数量. 这个变量通常是全局的; 但是如果你要求一个纯分析器(参阅一个纯(可重入)分析器-A Pure (Reentrant) Parser 一章), 那么这个变量就是一个只能被动作访问的局部变量.

4.4 在动作中使用的特殊特征-Special Features for Use in Actions

这里是在动作中使用的 Bison 结构,变量和红的列表.

变量: **$$**

像一个变量一样工作,这个变量包含了由当前规则构成的组的语义值. 参阅 动作-Actions.

变量: **$*n***

像一个变量一样工作,这个变量包含了当前动作第 *n* 个部件的语义值. 参阅 动作-Actions.

变量: **$<*typealt*>$**

类似$$但是指明了%union 声明中的 *typealt* 选项. 参阅 动作中值的数据类型-Data Types of Values in Actions.

变量: **$<*typealt*>*n***

类似$*n* 但是指明%union 声明中的 *typealt* 选项. 参阅 动作中值的数据类型-Data Types of Values in Actions.

宏: **YYABORT;**

立即从 yyparse 返回,表明分析失败. 参阅 分析器函数 yyparse-The Parser Function yyparse.

宏: **YYACCEPT;**

立即从 yyparse 返回,表明分析成功. 参阅 分析器函数 yyparse-The Parser Function yyparse.

宏: **YYBACKUP** *(token, value);*

移出一个记号. 这个宏仅仅在一个只归约单一值的规则中使用,并且只在没有超前扫描记号的时候被允许使用. 这个宏也不允许在 GLR 分析器中使用. 这个宏建立一个超前带有记号类型 *token* 和语义值 *value* 的超前扫描记号; 然后丢弃将要被这个规则归约的值.

如果这个宏在无效的情况下使用, 例如当已经存在超前扫描记号的情况下使用, 那么它会报告一个带有消息 `cannot back up'的语法错误并且执行一个普通的错误恢复程序.

在上述任一种情况下,动作的其余部分不会被执行.

宏: **YYEMPTY**

当没有超前扫描记号的时候,值被存放在 yychar 中.

宏: **YYERROR;**

立即导致一个语法错误. 这个语句启动错误恢复就像分析器自己已经侦测到一个错误一样; 然而,它并不调用 yyerror 并且不打印任何消息. 如果你要打印一个错误消息, 在`YYERROR;'语句之前显式地调用 yyerror. 参阅 错误恢复-Error Recovery.

宏: **YYRECOVERING**

当分析器从语法错误中恢复的时候,这个表达式的值为 1. 其余时候值为 0. 参阅 错误恢复-Error Recovery.

变量: **yychar**

包含当前超前扫描记号的变量. (在一个纯分析器中,这实际上是一个 yyparse 中的局部变量). 当没有超前扫描记号的时候,这个变量存储 YYEMPTY 的值. 参阅 超前扫描记号-Look-Ahead Tokens.

宏: **yyclearin;**

丢弃当前的超前扫描记号. 它主要用于错误恢复规则. 参阅 **错误恢复**-Error Recovery.

宏: **yyerrok;**

对后来的语法错误立即恢复产生错误消息. 它主要用于错误恢复规则. 参阅 **错误恢复**-Error Recovery.

值: **@$**

像一个包含文字位置信息的结构一样工作. 这个位置是由当前规则构成的组的位置. 参阅 **追踪位置**-Tracking Locations.

值: **@*n***

像一个包含文字位置信息的结构一样工作. 这个位置是由当前规则的第 *n* 个部件的位置. 参阅 **追踪位置**-Tracking Locations.

---

5. Bison 分析器算法-The Bison Parser Algorithm

当 Bison 读取记号的时候,它将这些记号同它们的语义值一起压入栈中. 这个栈被称为*分析器栈(parser stack)*. 将一个记号压入栈在传统上被称为*移进(shifting)*.

例如,假设中缀计算器已经读取`1 + 5 *'并且将要读取`3'. 分析器栈此时有四个元素,每个元素对应一个被移进的符号.

但是这个栈并不是总含有每个被读入记号的元素. 当最后 *n* 个被移进的记号和组匹配语法规则部件时, 可以由那个规则将它们结合起来. 这叫做*归约(reduction)*. 这些栈中的记号和组被一个单一的组取代. 那个组的符号是这个规则的结果(左手端).

运行规则的动作是处理归约的一部分, 因为这就是什么在计算结果组的语义值.

例如,如果中缀计算器的分析器栈包含这个:

```
1 + 5 * 3
```

并且下一个输入是一个换行符, 那么最后三个元素可通过这个规则被归约成 15:

```
expr: expr '*' expr;
```

那么这个栈仅含有三个元素:

```
1 + 15
```

在整个时候可以进行另外一个结果为 16 的归约. 然后,换行符记号才可以被移进.

分析器通过移进和归约尝试将整个输入化为一个符号为语法开始符号的单一组. (参阅**语言与上下文无关文法**-Languages and Context-Free Grammars 一章).

这种类型的分析器被称之为*自底向上(bottom-up)*的分析器.

5.1 超前扫描记号-Look-Ahead Tokens	当分析器决定做什么的时候它查看的一个记号.
5.2 移进/归约冲突-Shift/Reduce Conflicts	冲突:移进和归约均有效.
5.3 操作符优先级-Operator Precedence	用于解决冲突的操作符优先级.
5.4 上下文依赖优先级-Context-Dependent Precedence	当一个操作符的优先级依赖上下文.
5.5 分析器状态-Parser States	分析器是一个带有栈的有限状态机.
5.6 归约/归约冲突-Reduce/Reduce Conflicts	在同一情况下可以应用两个规则.
5.7 神秘的归约/归约冲突-Mysterious Reduce/Reduce Conflicts	看起来不平等的归约/归约冲突.

5.1 超前扫描记号-Look-Ahead Tokens

Bison 分析器并*不*总是在最后 *n* 个记号和组匹配一个规则时立即进行归约. 这是由于这种策略对于处理大多数语言来说是不够的. 相反,当可以进行一个归约的时候, 分析器有时"超前扫描"下一个记号来决定该怎么做.

当读取一个记号时, 它并不是被马上移进而是首先成为不在栈中的*超前扫描记号(look-ahead token)*. 现在分析器可以对记号和组进行一个或更多的归约,而超前扫描记号仍在栈外. 这并不意味着所有可能的归约已经执行; 依赖于超前扫描记号的符号类型, 一些规则可以选择推迟它们的应用.

这里有一个需要超前扫描记号的例子. 这三个规则定义了包括二进制加法操作符和一元后缀阶称操作符(`!'), 并且允许括弧分组.

```
expr:    term '+' expr
       | term
       ;

term:    '(' expr ')'
       | term '!'
       | NUMBER
       ;
```

假定`1 + 2'已经被读取和移进; 分析器这时应该做什么? 如果接下来的记号是`)', 那么前三个记号必须被归约成一个 expr. 这是唯一有效的情况, 因为移进`)'会产生一系列的 term ')', 而没有规则允许这样.

如果接下来的符号是`!', 那么它必须马上被移进以便`2 !'可以被移进产生一个 term. 如果不这样做, 分析器将会在移进之前进行归约, `1+2'会成为一个 expr. 那是移进`!'就是不可能的, 因为这么做会使栈中产生序列 expr '!'. 没有规则允许那样的序列.

当前的超前扫描记号被存储在 yychar 中. 参阅 在动作中使用的特殊特征-Special Features for Use in Actions.

5.2 移进/归约冲突-Shift/Reduce Conflicts

假设我们正在分析一个有 if-then 和 if-then-else 语句的语言并且这个语言带有如下一对规则:

```
if_stmt:
          IF expr THEN stmt
        | IF expr THEN stmt ELSE stmt
        ;
```

这里我们假定 IF,THEN 和 ELSE 是用来指定关键字的终结符.

当 ELSE 被读入成为超前扫描记号, 栈中的内容(假定输入是有效的)刚好可以由第一个规则进行归约. 但是移进 ELSE 也是合法的, 因为这最终将会导致由第二个规则进行的归约.

这种情况,移进或者归约都是有效的,被称为*移进/归约冲突(shift/reduce conflict)*. Bison 被设计成选择**移进**来解决这些冲突, 除非有其它的操作符优先级的指导. 为了研究这样做得原因, 我们将它和另一种选择(注:选择归约)做一个对比.

由于分析器选择移进 ELSE. 这样作的结果是将 else 从句依附到最里面的 if 语句中, 并使下面两个输入在作用上等价.

```
if x then if y then win (); else lose;


if x then do; if y then win (); else lose; end;
```

但如如果分析器选择在可能的时候归约而不是移进. 这样做的结果是将 else 从句依附到最外面的 if 语句中, 并使下面两个输入在作用上等价:

```
if x then if y then win (); else lose;


if x then do; if y then win (); end; else lose;
```

冲突存在的原因是由于语法本身有歧义: 任一种简单的 if 语句嵌套的分析都是合法的. 已经建立的惯例是通过将 else 从句依附到最里面的 if 语句来解决歧义; 这就是 Bison 为什么选择移进而不是归约的原因. (在理想的情况下,最好编写一个非歧义的文法, 但是在这种情况下却很难办到.) 这种特殊的歧义在 Algol 60 的描述中首次出现并被成为"悬挂 else"歧义.

为了避免 Bison 警告那些可以预见的合法的移进/归约冲突, 我们可以使用%expect *n* 声明. 当移进/归约冲突的数目恰好是 *n* 的时候, Bison 不会做出任何警告. 参阅 消除冲突警告-Suppressing Conflict Warnings.

有人抱怨上面 if_stmt 的定义, 但是冲突的确实在没有任何额外规则的时候出现. 这又一个完整的体现这个冲突的 Bison 输入文件:

```
%token IF THEN ELSE variable
%%
stmt:     expr
        | if_stmt
        ;

if_stmt:
          IF expr THEN stmt
        | IF expr THEN stmt ELSE stmt
        ;

expr:     variable
        ;
```


5.3 操作符优先级-Operator Precedence

移进/归约冲突也可能出现在算术表达式中. 在这里,移进并不总是优先的选择; Bison 关于操作符优先级的声明允许你指定什么时候移进和什么时候归约.

5.3.1 什么时候需要优先级-When Precedence is Needed

考虑下面的歧义文法片段 (产生歧义的原因是输入`1 - 2 * 3'可以由两种方法进行分析):

```
expr:     expr '-' expr
        | expr '*' expr
        | expr '<' expr
        | '(' expr ')'
        ...
        ;
```

假设分析器已经读入了记号`1',`-'和`2'; 那么它是否应该使用减法操作符规则进行归约呢? 这依赖于下一个记号. 当然,如果下一个记号是`)',我们必须归约, 由于没有规则可以归约`- 2 )'或者以它开始的记号序列, 所以移进是无效的. 但是如果下一个符号是`*'或者`<', 我么有了一个选择: 移进和归约都可以,但是会产生不同的结果.

要决定 Bison 应该怎么做,我们必须考虑结果. 如果下一个操作符记号 *op* 被移进, 那么它必须首先被归约以便允许进行另外一个归约的机会. 结果为`1 - (2 *op* 3)'. 另一方面,如果在移进 *op* 之前归约减法, 结果为`(1 - 2) *op* 3'. 很明显,选择移进或者归约依靠操作符`-'和 *op* 的相对优先级: `*'是该首先被移进,而不是`<'.

当输入为`1 - 2 - 5'的时候会怎么样, 这应该是`(1 - 2) - 5'还是`1 - (2 - 5)'? 对于大多数操作符来说,我们选择前者. 这被成为 *左结合(left association)*. 后面一种,*右结合(right association)*, 对于赋值操作符是理想的选择. 选择左结合或者有结合是当栈包含`1 - 2'并且超前扫描记号是`-'时,分析器选择移进还是归约的问题: 移进代表着右结合.

5.3.2 指定操作符的优先级-Specifying Operator Precedence

Bison 允许你使用操作符优先级声明%left 和%right 指定这些选择. 每一个这样的声明包含了一个要声明其优先级和结合性的记号列表. %left 声明使这些操作符成为左结合的, %right 声明使这些操作符成为右结合的. 第三种选择是%nonassoc, 它声明了 "在一行中"有两个相同的操作符是一个语法错误.

不同操作符的优先级由它们声明的顺序控制. 文件中的第一个%left 或者%right 声明的优先级最低, 下一个类似声明的操作符有稍高的优先级,以此类推.

5.3.3 优先级使用的例子-Precedence Examples

在我们的例子中,我们会发现下面的声明:

```
%left '<'
%left '-'
%left '*'
```

在一个支持其它操作符的更完整的例子中, 我们会成组地声明具有相同优先级的操作符. 例如'+'和'-'一起声明.

```
%left '<' '>' '=' NE LE GE
%left '+' '-'
%left '*' '/'
```

(在这里 NE 代表着"不相等"操作符",其它以此类推. 我们假定这些记号的长度多于一个字符并且因此由它们的名字代表而不是字符.)

5.3.4 优先级如何工作-How Precedence Works

优先级声明的第一个作用是赋予声明的终结符以优先级. 第二个作用是赋予特定的规则以优先级: 每个规则从部件中最后一个提及的终结符中获取优先级. (你也可以明确的指明规则的优先级. 参阅 上下文依赖优先级-Context-Dependent Precedence.)

最终,解决中冲突的方法是比较正在考虑的规则和超前扫描记号的优先级. 如果超前扫描记号的优先级更高,那么选择移进. 如果规则的优先级更高,那么选择归约. 如果它们有相同的优先级, 那么靠那个优先级的结合性来作出选择. 由选项`-v'(参阅调用 Bison-Invoking Bison 一章)制造的冗长的输出文件(The verbose output file) 说明了每个冲突是如何解决的.

并不是所有的规则和记号都有优先级. 如果规则和超前扫描记号都没有优先级, 那么默认的动作是移进.

5.4 上下文依赖优先级-Context-Dependent Precedence

一个操作符的优先级通常依赖于上下文. 这最开始听起来很古怪,但它的确很常见. 例如,典型地,一个负号操作符有比一元操作符更高的优先级并且比二进制操作符的优先级稍低(低于乘法).

Bison 优先级声明,%left,%right 和%nonassoc 对于一个给定的操作符只能使用一次; 所以通过这种方法,一个操作符只能有一种优先级. 对于上下文依赖优先级来说,你需要使用一种额外的机制: 规则的%prec 修饰符.

%prec 靠指定用于那个规则终结符的优先级来声明特定规则的优先级. 那个符号不需要以特殊的方式出现在规则中. 修饰符的语法为:

```
%prec terminal-symbol
```

并且它写在规则的部件之后. 它的作用是赋予规则 *terminal-symbol* 的优先级而不考虑从普通方法推导出的优先级. 被改变的规则优先级会影响包含那个规则的冲突的解决方法. (参阅操作符优先级-Operator Precedence 一章).

这是%prec 如何解决负号问题的例子. 首先为一个虚构的名为 MINUS 的终结符声明优先级. 实际上没有记号是这种类型, 但是这个符号以它自己的优先级来使用.

```
...
%left '+' '-'
```

```
%left '*'
%left UMINUS
```

现在可以在规则中使用 MINUS 的优先级.

```
exp:    ...
        | exp '-' exp
        ...

        | '-' exp %prec UMINUS
```


5.5 分析器状态-Parser States

函数 yyparse 是使用有限状态机(finite-state-machine)来实现的. 压入分析器栈中的值不仅仅是符号类型码; 它们代表了整个在栈顶或者靠近栈顶的终结符和非终结符序列. 当前的状态收集了与决定下一步怎么做相关的之前输入的信息.

每次读入一个超前扫描记号, 分析器就在一个表中搜索分析装当前状态和超前扫描记号类型. 这个表项能会说"移进超前扫描记号" 在这种情况下,它在指定了一个新的分析器状态的同时将这个状态压入栈顶. 或者,它(注:指表项)也可能说"使用第 *n* 个规则进行归约." 这意味着某些个数的记号合组被移出栈,取而代之的是一个组. 用另外一种说法, 那些个数(注:*n*)的状态被弹出栈,一个新状态被压入栈.

还有另外一种选择:这个表可能会说那个超前扫描记号在当前的状态下是错误的. 这会引发错误处理.(参阅**错误恢复**-Error Recovery 一章).

5.6 归约/归约冲突-Reduce/Reduce Conflicts

一个归约/归约冲突发生在有两个或者更多规则可以被用于相同输入序列的情况下. 这通常表明了一个语法中的严重错误.

例如,这里是一个试图定义零个或者更多 word 组的错误.

```
sequence: /* empty */
            { printf ("empty sequence\n"); }
        | maybeword
        | sequence word
            { printf ("added word %s\n", $2); }
        ;

maybeword: /* empty */
            { printf ("empty maybeword\n"); }
        | word
            { printf ("single word %s\n", $1); }
```

```
        ;
```

这个错误是一个歧义:有多种方法可以将单一的 word 分析成一个 sequence. 它可以归约为一个 maybeword 然后通过第二个规则归约为一个 sequence. 另外,什么都没有可以通过第一个规则归约为一个 sequence, 可以使用 sequence 的第三个规则将它和 word 结合起来.

也有多种方法将什么都没有归约成一个 sequence. 可以直接通过第一个规则归约或者间接通过 maybeword 然后通过第二个规则归约.

你可能认为这没有什么区别, 因为不论任意的输入是否有效它没有什么变化. 但是它却影响这该执行哪一条规则. 一种分析顺序运行了第二个规则的动作, 另一个则运行了第一个和第三个规则的动作. 在这个例子中,程序的输出有所变化.

Bison 靠选择首先出现在语法中的规则解决归约/归约冲突, 但是依靠这种策略是十分冒险的事情. 必须仔细研究每一个归约/归约冲突并且通常要消灭它们. 这里有一个正确定义 sequence 的方法:

```
 sequence: /* empty */
             { printf ("empty sequence\n"); }
         | sequence word
             { printf ("added word %s\n", $2); }
         ;
```

这里是另外一个产生归约/归约冲突的普通例子:

```
 sequence: /* empty */
         | sequence words
         | sequence redirects
         ;

 words:    /* empty */
         | words word
         ;

 redirects:/* empty */
         | redirects redirect
         ;
```

这里的目的是定一个可以包含 word 或 redirect 组的序列. sequence,words 和 redirects 的定义都是没有问题的, 但是三个在一起却产生微妙的歧义: 即使一个空输入可以有无限多种分析的方式.

考虑:什么都没有可以是一个 words. 或者它可以是两个在一行的 words,或者三个,或者任意个. 它同样也可以是一个 words 后跟三个 redirects 和另外一个 words. 等等.

这有两种改正这些规则的方法. 第一,使它成为一个单层序列:

```
 sequence: /* empty */
         | sequence word
         | sequence redirect
         ;
```

第二,防止 words 或者 redirects 为空:

```
sequence: /* empty */
        | sequence words
        | sequence redirects
        ;

words:    word
        | words word
        ;

redirects:redirect
        | redirects redirect
        ;
```


5.7 神秘的归约/归约冲突-Mysterious Reduce/Reduce Conflicts

[untranslated] Sometimes reduce/reduce conflicts can occur that don't look warranted. [/untranslated] 这里有一个例子:

```
%token ID

%%
def:    param_spec return_spec ','
        ;
param_spec:
             type
        |    name_list ':' type
        ;
return_spec:
             type
        |    name ':' type
        ;
type:        ID
        ;
name:        ID
        ;
```

```
name_list:
        name
    |   name ',' name_list
    ;
```

这个文法看起来可以用一个单一的超前扫描记号分析: 当正在读入 para_spec 的时候. 如果 ID 后面紧跟一个分号,那么它是一个 name. 如果 ID 后面跟随另外一个 ID,那么它是一个 type. 换句话说,这是一个 LR(1)文法.

然而,像大多数分析器产生器一样,Bison 实际上并不能处理所有的 LR(1)文法. 在这个文法中,两个位于 param_spec 的开始,并同样地位于 return_spec 开始 ,在 ID 之后的上下文十分相似,以致于 Bison 认为它们是相同的. 它们看起来相似因为相同的规则集是活动的--归约到 name 的规则和归约到 type 的规则. Bison 在那个处理阶段没有能力决定这些规则在两个上下文中需要不同的超前扫描记号, 所以它为两种情况制造了同一个分析器状态. 结合两个上下文会在稍后引起一个冲突. 在分析器术语中, 这种情况意味着这个文法不是 LALR(1)文法.

通常来讲,最好是修补漏洞而不是将漏洞写入文档. 但是这个特殊的漏洞很难被修复; 处理 LR(1)文法的分析器生成器很难编写并且倾向于制造很大的分析器. 在实践中,Bison 显得更为实用.

当问题产生时, 你通常可以通过指明两个被混淆的分析器状态并且添加使它们看起来截然不同的额外的东西来修补它, 在上面的例子中, 如下地向 return_spec 添加一个规则会消除这个问题:

```
%token BOGUS
...
%%
...
return_spec:
        type
    |   name ':' type
    /* This rule is never used.  */ /*  这个规则永远不会被使用 */
    |   ID BOGUS
    ;
```

这样做改正这个问题, 因为在 return_spec 开始部分 ID 后的上下文中引进了一个可能的额外的活动规则. 这个规则在 param_spec 相应的上下文中并不是活动的, 所以两个上下文接受了不同的分析器状态. 只要 yylex 永远不产生记号 BOGUS, 新增的规则就不能改变分析输入的实际方法.

在这个特殊的例子中,还有另外一种解决问题的方法: 直接使用使用 ID 来替代 name 来重写 return_spec 的规则. 这样做也使两个混淆的上下文有了不同的活动集, 因为 return_spec 的活动集激活了 return_spec 的规则而不是 name 的.

```
param_spec:
        type
    |   name_list ':' type
    ;
return_spec:
        type
    |   ID ':' type
    ;
```

5.8 通用 LR (GLR)分析-Generalized LR (GLR) Parsing

Bison 产生*确定性(determinstic)*的分析器. 这种分析器基于先前输入和额外的超前扫描记号的摘要, 唯一性地选择进行归约的时机和如何进行归约. 结果,通常,Bison 处理一个上下文无关文法语言族的自己. 由于歧义文法含有可以使用多种可能的归约序列的字符串, 所以在这种情况下不能使用确定的分析器. 这种情况同样适用于需要多于一个超前扫描记号的语言, 因为分析器缺乏做出决定所需要的必要信息, 这时它必须被制作成一个移进-归约分析器. 最终,如同之前提到的(参阅神秘的冲突-Mystery Conflicts 一章), 有这样一些语言,Bison 关于如何总结输入的特殊选择目前看起来缺少必要的信息.

当你在你的语法文件中使用`%glr-parser'声明的时候, Bison 产生一个使用不同算法的分析器,这种分析器被称为通用 LR(或 GLR)分析器. 一个 Bison GLR 分析器使用同样基本的算法做为一个普通的 Bison 分析器进行分析, 但当存在一个不能被优先级规则(参阅优先级-Precedence 一章)解决的移进/归约冲突, 或者一个归约/归约冲突时却有着与普通 Bison 分析器不同的行为. 当一个 GLR 分析器遭遇这种情况的时候, 它高效地*分裂(splits)*成多个分析器, 每个对应一种可能的移进或者归约. 这些分析器如常地进行分析,使用锁步(lock-step)消耗记号. 一些栈遭遇了其它的冲突并且进一步分裂, 一个 Bison GLR 分析栈是一个取代状态序列的高效的分析树.

实际上,每个栈代表一个关于正确分析的猜想. 剩余的输入可能会表明一个猜想是错误的, 在这种情况下,不正确的栈静静地消失. 另外,每个栈中的语义动作被保存而不是立即执行. 但一个栈消失时,它存储的的语义动作永远不会被执行. 当一个归约使两个栈等价的时候, 它们的语义动作集和导致归约的状态都会被保存. 我们说两个栈是等价的当它们都代表相同的状态序列, 并且每对相应的状态代表一个产生相同输入流片段的语法符号.

每当分析器从有多个分析状态转换为一个分析状态时, 在执行了原来保存的动作后, 这个分析器将转变到通常的 LALR(1)分析算法. 在这个转换过程中,一些栈上的状态含有可能的动作集(实际上是多个集)的语义值. 分析器试图从这些动作中挑选一个被`%prec'声明指定的有最高动态优先级的动作. 否则,如果可选择的动作并未被优先级排序, 但对两个规则使用`%merge'声明了相同的合并函数, Bison 评价并解决它们之后调用合并函数求得结果. 否则它会报告一个歧义.

对 GLR 分析树使用这样一种数据结构是可能的, 这种结构可以以线性的时间(相对输入的大小)处理任意的 LALR(1)文法, 在最坏情况下以二次方的时间处理任何非歧义文法(不一定是 LALR(1)), 在最坏情况下以三次方的时间处理任何普通(可能是歧义的)上下文无关文法. 然而 Bison 当前使用一种更简单的数据结构, 这中数据结构需要与输入长度乘以输入的任意前需要缀最大栈数目成比例的时间. 因此,实际上,歧义或者不确定文法可能需要指数的时间和空间来处理. 然而,这种非常糟糕例子通常情况下很难见到. 文法中的不确定性通常是局部的--分析器一次只对很少一些记号"产生疑惑". 因此,当前的数据结构在大多数情况下足够用了. 特别地,对于文法的 LALR(1)部分,它(注:通用 GLR 分析器) 仅仅比默认的 Biosn 分析器稍慢.

想获得更详细的 GLR 分析器的说明,请参阅: Elizabeth Scott, Adrian Johnstone and Shamsa Sadaf Hussain, Tomita-Style Generalised LR Parsers, Royal Holloway, University of London, Department of Computer Science, TR-00-12, http://www.cs.rhul.ac.uk/research/languages/publications/tomita_style_1.ps, (2000-12-24).

5.9 栈溢出以及如何避免它-Stack Overflow, and How to Avoid It

如果太多的记号被移进而没有被归约, Bison 分析器栈可能会溢出. 在这种情况发生时, 分析器函数 yyparse 返回非零值, 暂停执行并调用 yyerror 来报告错误.

由于 Bison 分析器拥有生长的栈, 达到上限通常是由于使用右递归而不是左递归而产生的. 参阅 递归规则-Recursive Rules.

靠定义宏 YYMAXDEPTH,你可以控制栈溢出之前的最大深度. 我们应该用正数定义这个宏. 这个值是在溢出之前被移进(而没被归约)的记号的最大数目.

允许的栈空间不需要一次分配完毕. 如果你为 YYMAXDEPTH 指定了一个很大的数字, 分析器实际上在开始之分配了一个空间很小的栈, 随个阶段性的需求,分析器会扩大栈的容量. 增大空间的分配自动并且沉默地进行. 因此,你不需要为了不需要多少空间的普通输入节省空间而将 YYMAXDEPTH 定义的很小.

然而,我们同样不要把 YYMAXDEPTH 定义的很大以至于在计算栈容量时产生算术溢出. 并且我们也不要将 YYMAXDEPTH 定义的比 YYINITDEPTH 还小.

如果你没有定义 YYMAXDEPTH,那么它的默认值是 10000.

你可靠定义宏 YYINITDEPTH 为一个正值来控制栈初始分配的空间. 除非你使用 C99 或者其它允许变长数组的语言和编译器, 对于 C 语言 LALR(1)分析器来说, 这个值必须为编译时常量. YYINITDEPTH 的默认值为 200.

不要让 YYINITDEPTH 过大以至于当计算栈空间时发生溢出. 同样地,不要让 YYINITDEPTH 大于 YYMAXDEPTH.

由于 C 和 C++语义上的区别, 利用 C++编译器编译的用 C 语言编写的 LALR(1)分析器不能生长. (注:指栈不能生长) 在这种情况下(作为 C++来编译 C 分析器), 我们建议你增加 YYINITDEPTH 的大小. 在不久的将来, 我们会提供涉及到这个问题的 C++输出.

---

6. 错误恢复-Error Recovery

我们通常不能接受让一个程序在遇到语法错误时就终止. 例如,一个编译器应该充分的从错误中恢复以便分析输入文件的其余部分并且检查其中的错误; 一个计算器应该接受其它的表达式.

在每个输入都是一行的简单互交命令分析器中, 让 yyparse 在遇到错误时返回 1 并且使调用者忽略剩下的输入行(然后重新调用 yyparse 就足够了. 但是这对于编译器来说显然不够, 因为为它忘记了导致错误的全部构造上下文. 在编译器输入中,深入到一个函数内部的语法错误, 并不应该使编译器对待后面的行像对待源文件的开始一样.

你可以靠编写一个识别特殊记号 error 的规则来定义如何从语法错误中恢复. 它总是一个已经被定义(你不需要声明它)并且保留做错误处理使用的终结符. 每当一个语法错误发生时,Bison 分析器就产生一个 error 记号; 如果你在当前的上下文中提供了一个识别该记号的规则, 那么分析可以继续进行.

例如:

```
stmnts:  /* empty string */ /* 空字符串 */
        | stmnts '\n'
        | stmnts exp '\n'
        | stmnts error '\n'
```

这个例子的第四个规则说明了一个错误后紧跟一个换行对任何 stmnts 是有效的添加.

如果错误发生在 exp 中间的话会发生什么情况? 这个错误恢复规则,被精确地解释为应用于一个 stmnts,一个 error 和一个换行的精确序列. 如果一个错误发生在一个 exp 中间, 那么在栈中最后的 stmnts 之后很可能有一些额外的记号或者自表达式, 即有一些记号在下一个换行之前被读入. 所以这个规则并不按通常的方法应用.

但是 Bison 可以靠丢弃部分语义上下文和部分输入来强制地使这个规则(注:错误恢复规则)适用于这种情况. 首先,它从栈中丢弃状态和对象直到回到一个可以接受 error 的状态. (这意味着分析过的子表达式被丢弃,并且回到最后一个完整的 stmnts.) 这时 error 记号可以被移进. 之后,如果旧的超前扫描记号不能接受移进下一个记号, 分析器如读记号并且丢弃它们直到找到一个可以接受的记号. 在这个例子中,Bison 读入并丢弃输入直到下一个换行符以便应用第四个规则. 注意到丢弃的符号通常是内存泄露之源,参阅释放丢弃的符号-Freeing Discarded Symbols 以获取更多信息.

在语法中,对于错误恢复规则的选择就是对错误恢复策略的选择. 一个简单而使用的策略是如果检测到一个错误,跳过当前输入行的剩余部分:

```
stmnt: error ';'  /* On error, skip until ';' is read.  */ /* 当错误出现时,跳过剩余部分直到读入
';' */
```

为一个已经分析的作括号恢复匹配一个右括号也是非常实用的. 否则,右括号很可能不匹配地出现并且引发另外的更严重的错误消息:

```
primary:  '(' expr ')'
        | '(' error ')'
        ...
        ;
```

错误恢复策略是必要的猜测. 当它们猜测的时候,一个语法错误通常会导致另外一个错误. 在上面的例子中, 错误规则猜测:一个错误是由于一个 stmnt 中的错误输入引起的. 假设一个伪造的分号被插入到一个有效的 stmt 中间. 在错误恢复规则从第一错误恢复之后,分析器会立刻发现另外一个错误, 因为在伪造的分号之后的文字也是一个无效的 stmt.

为了阻止错误的倾泄而出, 分析器在第一个错误之后立即发现另一个错误时,不会输出错误消息; 仅在三个连续的数据记号被成功归约之后,分析器才会恢复输出错误消息.

注意到接受 error 记号的规则像其它任何规则一样也可以有动作.

你可以通过使用宏 yyerrok 使错误消息立即恢复. 如果你在错误恢复规则的动作中使用它, 没有任何错误消息会被抑制. 这个宏不需要任何参数; `yyerrok;'是一个有效的 C 语句.

先前的超前扫描记号在一个错误后会被立即再分析. 如果这是不可接受的, 那么宏 yyclearin 可以用于清除这个记号. 将语句 `yyclearin;'写入错误恢恢复规则的动作中.

例如,假设在遭遇一个语法错误时, 一个错误处理程序被调用用于将输入流前进到重新开始分析的地方. 词法分析器返回的下一个记号很可能是正确的. 前一个超前扫描记号应该用 `yyclearin;'丢弃.

宏 YYRECOVERING 代表一个表达式. 这个表达式在分析器从语法错误中恢复时值为 1,在其它的时候值为 0. 值为 1 指明了要抑制新的语法错误产生的错误消息.

---

7. 处理上下文依赖-Handling Context Dependencies

Bison 的分析模式是首先分析记号,之后将它们组合成更大的句法单元. 在许多语言中,一个记号的意义受到上下文的影响. 尽管这破坏了 Bison 范例, 某些技术(被称为 *kludges*)可以使你有能力为这种语言编写 Bison 分析器.

7.1 符号类型中的语义信息-Semantic Info in Token Types　　对记号的分析可能依赖于语义上下文.

7.1 符号类型中的语义信息-Semantic Info in Token Types

C 语言就有上下文依赖: 标识符使用的方法依赖于当当前的意义. 例如,考虑这个:

```
foo (x);
```

这看起来是一个函数调用语句,但如果 foo 是一个 typedef 名称, 那么这实际上是一个 x 的声明. C 语言的 Bison 分析器如何决定怎么分析这个输入呢?

GNU C 使用的办法是让它们有不同的记号类型, INDENTIFIER 和 TYPENAME. 当 yylex 发现一个标识符, 它搜索当前的标识符声明以便决定返回什么样的记号类型: 如果标识符由一个 typedef 声明的,就返回 TYPENAME,否则返回 IDENTIFIER. 这时,语法规则就可以通过对要识别的记号类型的选择来表达上下文依赖. IDENTIFIER 可以作为一个表达式被接受,但是 TYPENAME 却不能. TYPENAME 可以开始一个声明,但是 IDENTIFIER 却不可以. 在标识符的意义不明显的上下文中, 例如在可以隐藏一个 typedef 名称的声明中, TYPENAME 和 IDENTIFIER 都是可接受的-- 并没有一个针对没一种记号类型的规则.

如果在接近分析标识符的地方决定允许什么种类的标识符, 那么这个技术可以简单的应用. 但是在 C 语言中却不总是这样: C 允许重新声明之前声明的带有明确类型的 typedef 名称.

```
typedef int foo, bar, lose;
static foo (bar);       /* redeclare bar as static variable */ /* 重新声明 bar 为一个静态变量 */
static int foo (lose);  /* redeclare foo as function */ /* 重新声明 foo 为一个函数 */
```

不幸的是,这个名称被一个复杂的句法结构--"声明符"所分隔.

结果,C 语言的 Bison 分析器的某些部分要被复制,并且要改变所有非终结符的名称: 一次是为了分析可以被重定义的 typedef 声明, 一次是为了分析不能被重定义的声明. 这里是复制的部分. 为了简洁省略了动作.

```
initdcl:
          declarator maybeasm '='
          init
        | declarator maybeasm
        ;

notype_initdcl:
          notype_declarator maybeasm '='
          init
```

```
        | notype_declarator maybeasm
        ;
```

在这里 initdcl 可以重新声明一个 typedef 名称, 但是 notype_initdcl 却不能. declarator 和 notype_declarator 的区别在于同一类型的不同种类.

这种技术和词法关联技术(在下一节描述)有一些相似之处. 它们的区别是,这里的信息是全局的并且用于程序的其它目的. 一个真正的词法关联含有一个受上下文控制的特殊目的标志.

7.2 词法关联-Lexical Tie-ins

另外一种处理上下文依赖的方法是 *词法关联(lexical tie-in)*: 一个由 Bison 动作设置的标志, 它的目的是改变分析记号的方式. 例如,假设我们有一种类似 C 的语言, 但是它带有一个特殊的`hex (*hex-expr*)'结构. 在关键字 hex 之后是一个括号之中全部为十六进制整数的表达式. 特别地,在那个上下文中,记号`a1b'必须被看做是一个整数而不是一个标识符. 这里就是你如何处理它:

```
%{
  int hexflag;
  int yylex (void);
  void yyerror (char const *);
%}
%%
...
expr:   IDENTIFIER
        | constant
        | HEX '('
              { hexflag = 1; }
          expr ')'
              { hexflag = 0;
                $$ = $4; }
        | expr '+' expr
              { $$ = make_sum ($1, $3); }
        ...
        ;

constant:
          INTEGER
        | STRING
        ;
```

这里我们假设 yylex 观察 hexflag 的值; 当它的值非零时,所有的整数被分析成十六进制数, 并且带有字母的标识符也尽可能的被翻译成整数.

hexflag 出现在分析器文件的 *Prologue* 部分以便动作可以访问它 (参阅 *Prologue* 部分-The Prologue 一章). 你还必须在 yylex 中编写代码来获得这个标志.

7.3 词法关联和错误恢复-Lexical Tie-ins and Error Recovery

词法关联对你使用的任何错误恢复规则都有严格的要求. 参阅 **错误恢复**-Error Recovery.

这样的原因是错误恢复规则的目的是放弃对一个结构的分析并且恢复到某个更大的结构中去. 不例如,在类似 C 的语言中, 一个典型的错误恢复规则是跳过记号直到下一个分号, 并且开始分析一个新的语句, 像这样:

```
stmt:   expr ';'
        | IF '(' expr ')' stmt { ... }
        ...
        error ';'
                { hexflag = 0; }
        ;
```

如果在`hex (*expr*)'之中存在一个语法错误, 这个错误恢复规则就会被应用, 完整的`hex (*expr*)'的动作永远都不会执行. 所以对于其余的输入或者直到下一个关键字 hex, hexflag 仍然被置 1.这会导致标识符被错误地解释为整数.

为了避免这个错误,错误恢复规则自己要将 hexflag 清零.

也有可能存在一个与表达式一起工作的错误恢复规则. 例如,可能有一个应用于括号匹配的规则, 并且它跳跃到右括号:

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

如果这个规则在 hex 结构中执行, 它不会放弃那个结构(由于它作于在结构内部的括号(注:结构指 hex 结构)). 因此,它不应该将标志清零: hex 结构的其余部分应该在该标志仍然有效的情况下被分析.

如果有一个错误规则依靠当时的状况可能放弃 hex 结构也可能不放弃的话, 我们该怎么办? 没有办法编写一个可以决定是否放弃 hex 结构的动作. 所以,如果你使用了词法关联, 最好保证你的错误恢复规则不是这种类型. 你必须要确定每个规则总是要清零或总不要清零.

8. 调式你的分析器-Debugging Your Parser

开发分析器可能是一种挑战,特别当你不理解它的算法的时候 (参阅 Bison 分析器算法-The Bison Parser Algorithm 一章). 即使是这样,有些时候一个关于自动的详细描述可能会有所帮助 (参阅理解你的分析器- Understanding Your Parser 一章), 或者跟踪分析器的执行可以给你关于为它什么做出不正确的行为一些灵感. (参阅跟踪你的分析器- Tracing Your Parser 一章).

8.1 理解你的分析器-Understanding Your Parser

如同本文档其它部分描述的 (参阅 Bison 分析器算法-The Bison Parser Algorithm 一章), Bison 分析器是*移进/归约自动机 (shift/reduce automata)*. 在一些情况下(比你希望的要更频繁), 调整或者简单的修正一个分析器需要考虑这个自动机. Bions 提供了它(自动机)的两种表示方法,文本的或者图形的(作为一个 VCG 文件).

当指定选项`--report'或者`--verbose'时 Bison 生成文本文件, 参阅 调用 Bison-Invoking Bison. 它的名称由移除分析器输出文件名`.tab.c'或者`.c'而添加`.output'取代. 因此,如果输入文件是`foo.y', 那么默认的分析器文件为`foo.tab.c'. 结果,冗长(verbose)输出文件为`foo.output'.

下面的语法文件`calc.y'将在稍后使用:

```
%token NUM STR
%left '+' '-'
%left '*'
%%
exp: exp '+' exp
   | exp '-' exp
   | exp '*' exp
   | exp '/' exp
   | NUM
   ;
useless: STR;
%%
```

bison 报告:

```
calc.y: warning: 1 useless nonterminal and 1 useless rule
calc.y:11.1-7: warning: useless nonterminal: useless
calc.y:11.10-12: warning: useless rule: useless: STR
calc.y: conflicts: 7 shift/reduce
```

当指定`--report=state', 除了文件`calc.tab.c', 它还创建了包含如下详细信息的文件`calc.outut'. 输出和精确表述的顺序可能有所不同, 但是对此的解释是相同的.

第一个部分包括了由前面的与/或结合性解决的冲突的详细信息.

```
Conflict in state 8 between rule 2 and token '+' resolved as reduce.
Conflict in state 8 between rule 2 and token '-' resolved as reduce.
```

Conflict in state 8 between rule 2 and token '*' resolved as shift.

...

下一个部分列出了仍然有冲突的状态清单.

State 8 conflicts: 1 shift/reduce

State 9 conflicts: 1 shift/reduce

State 10 conflicts: 1 shift/reduce

State 11 conflicts: 4 shift/reduce

下一个部分报告了没有用处的记号,非终结符和规则. 没用处的非终结符和规则被移除以便产生一个更小的分析器, 但是没用记号被保留,因为它们可能被扫描器使用, (应该注意到"没用处的"和"没被使用的"之间的区别).

Useless nonterminals:

useless

Terminals which are not used:

STR

Useless rules:

#6 useless: STR;

下一个部分重新制造了 Bison 使用的精确语法:

Grammar

Number, Line, Rule

0 5 $accept -> exp $end

1 5 exp -> exp '+' exp

2 6 exp -> exp '-' exp

3 7 exp -> exp '*' exp

4 8 exp -> exp '/' exp

5 9 exp -> NUM

并且报告了使用的符号:

Terminals, with rules where they appear

$end (0) 0

'*' (42) 3

'+' (43) 1

'-' (45) 2

'/' (47) 4

error (256)

NUM (258) 5

```
Nonterminals, with rules where they appear

$accept (8)
    on left: 0
exp (9)
    on left: 1 2 3 4 5, on right: 0 1 2 3 4
```

Bison 之后进入到自己的自动机, 并且用*项目(items)*集,也被成为*指明规则(pointed rules)*,来描述每个状态. 每个都是一个产生式规则,并且带有表示输入光标的点号.

```
state 0

    $accept  ->  . exp $   (rule 0)

    NUM         shift, and go to state 1

    exp         go to state 2
```

这些有如下含义: "状态 0 相应地处于分析的开始, 在初始规则中,处于开始符号(这里是 exp)的右端. 当分析器归约了一个产生的 exp 的规则并返回这个状态之后, 控制流跳转到状态 2. 如果没有这样的非终结符转化并且超前扫描记号是 NUM, 那么这个记号被移进到分析器栈中,控制流跳转到状态 1. 任何其它的超前扫描记号都会引发一个语法错误."

即使状态 0 中的唯一活动规则看起来是规则 0, 报告将 NUM 列举为一个超前扫描记号, 这是因为 NUM 可以在任何转向 exp 的规则的开头. 默认地,Bison 报告项目集的核心(*core* or *kernel* of the item set). 但是如果你想查看更详细的信息,你可以使用选项`--report=itemset'调用 bison 来列出所有的项目,包括那些可以由此派生的.

```
state 0

    $accept  ->  . exp $   (rule 0)
    exp  ->  . exp '+' exp   (rule 1)
    exp  ->  . exp '-' exp   (rule 2)
    exp  ->  . exp '*' exp   (rule 3)
    exp  ->  . exp '/' exp   (rule 4)
    exp  ->  . NUM   (rule 5)

    NUM         shift, and go to state 1

    exp         go to state 2
```

在状态 1 中...

```
state 1
```

```
    exp  ->  NUM .   (rule 5)

    $default    reduce using rule 5 (exp)
```

规则 5,`exp: NUM;'是完整的. 无论超前扫描记号(`$default')是什么,分析器都会归约它. 如果是从状态 0 跳转过来,在归约之后会回到到状态 0,并且之后会跳转到状态 2(`exp: go to state 2').

```
  state 2

    $accept  ->  exp . $   (rule 0)
    exp  ->  exp . '+' exp   (rule 1)
    exp  ->  exp . '-' exp   (rule 2)
    exp  ->  exp . '*' exp   (rule 3)
    exp  ->  exp . '/' exp   (rule 4)

    $           shift, and go to state 3
    '+'         shift, and go to state 4
    '-'         shift, and go to state 5
    '*'         shift, and go to state 6
    '/'         shift, and go to state 7
```

在状态 2 中,自动机只能进行归约符号. 例如,根据项目`exp -> exp . '+' exp',如果超前扫描记号为`+', 它会被移进到分析器栈中,并且状态机控制会跳转到状态 4, 对应项目`exp -> exp '+' . exp'. 由于没有默认动作,任何非上述列出的记号会引起一个语法错误.

状态 3 被称为*终态(finial state)*或者*接受态(accepting state)*:

```
  state 3

    $accept  ->  exp $ .   (rule 0)

    $default    accept
```

初始规则已经完成(已经读取开始符号和输入终结), 分析成功退出.

状态 4 到 7 解释的很直接,留给读者自己分析:

```
  state 4

    exp  ->  exp '+' . exp   (rule 1)

    NUM         shift, and go to state 1

    exp         go to state 8
```

```
state 5

    exp  ->  exp '-' . exp   (rule 2)

    NUM         shift, and go to state 1

    exp         go to state 9

state 6

    exp  ->  exp '*' . exp   (rule 3)

    NUM         shift, and go to state 1

    exp         go to state 10

state 7

    exp  ->  exp '/' . exp   (rule 4)

    NUM         shift, and go to state 1

    exp         go to state 11
```

正如报告开始部分声明的, `State 8 conflicts:1 shift/reduce':

```
state 8

    exp  ->  exp . '+' exp   (rule 1)
    exp  ->  exp '+' exp .   (rule 1)
    exp  ->  exp . '-' exp   (rule 2)
    exp  ->  exp . '*' exp   (rule 3)
    exp  ->  exp . '/' exp   (rule 4)

    '*'         shift, and go to state 6
    '/'         shift, and go to state 7

    '/'         [reduce using rule 1 (exp)]
```

```
    $default    reduce using rule 1 (exp)
```

的确,有两个与超前扫描记号`/'关联的动作: 或者移进(并且转到状态 7),或者归约规则 1. 这个冲突意味着或者语法是歧义的或者分析器缺少做出正确决定的信息. 这个语法确实是歧义的,因为我们并未指明`/'的优先级, 句子`NUM + NUM / NUM'可以被分析为对应于移进`/'的`NUM + (NUM / NUM)', 也可以被分析为对应于归约规则 1 的`(NUM + NUM) / NUM'.

由于在 LALR(1)分析中只能做出一个动作, Bison 武断地选择不使用归约,参阅移进/归约冲突-Shift/Reduce Conflicts. 被丢弃的动作被报告于方括号中.

注意到先前的所有状态只有一个单一可能的动作: 或者移进下一个记号并且转到相应的状态, 或者归约一个规则. 在其它的情况下, 例如, 当移进*和*归约都是可能的或者*多个*归约都是可能的, 这是需要超前扫描记号来选择动作. 状态 8 就是这样一种状态:如果超前扫描记号是`*'或者`/' 那么多做是移进,否则动作是归约动作 1. 换句话说,前两项,对应于规则 1,当超前扫描记号是`*'的时候是不符合条件的, 因为我们指明了`*'有比`+'更高的优先级. 更普通地说, 一些项目仅在某些可能的超前扫描记号下是符合条件的. 当使用选项`--report=look-ahead',Bison 会指明这些超前扫描记号:

```
  state 8

    exp  ->  exp . '+' exp  [$, '+', '-', '/']   (rule 1)
    exp  ->  exp '+' exp .  [$, '+', '-', '/']   (rule 1)
    exp  ->  exp . '-' exp   (rule 2)
    exp  ->  exp . '*' exp   (rule 3)
    exp  ->  exp . '/' exp   (rule 4)

    '*'        shift, and go to state 6
    '/'        shift, and go to state 7

    '/'        [reduce using rule 1 (exp)]
    $default    reduce using rule 1 (exp)
```

其余的状态与之类似:

```
  state 9

    exp  ->  exp . '+' exp   (rule 1)
    exp  ->  exp . '-' exp   (rule 2)
    exp  ->  exp '-' exp .   (rule 2)
    exp  ->  exp . '*' exp   (rule 3)
    exp  ->  exp . '/' exp   (rule 4)

    '*'        shift, and go to state 6
    '/'        shift, and go to state 7

    '/'        [reduce using rule 2 (exp)]
    $default    reduce using rule 2 (exp)
```

state 10

```
    exp  ->  exp . '+' exp   (rule 1)
    exp  ->  exp . '-' exp   (rule 2)
    exp  ->  exp . '*' exp   (rule 3)
    exp  ->  exp '*' exp .   (rule 3)
    exp  ->  exp . '/' exp   (rule 4)

    '/'         shift, and go to state 7

    '/'         [reduce using rule 3 (exp)]
    $default    reduce using rule 3 (exp)
```

state 11

```
    exp  ->  exp . '+' exp   (rule 1)
    exp  ->  exp . '-' exp   (rule 2)
    exp  ->  exp . '*' exp   (rule 3)
    exp  ->  exp . '/' exp   (rule 4)
    exp  ->  exp '/' exp .   (rule 4)

    '+'         shift, and go to state 4
    '-'         shift, and go to state 5
    '*'         shift, and go to state 6
    '/'         shift, and go to state 7

    '+'         [reduce using rule 4 (exp)]
    '-'         [reduce using rule 4 (exp)]
    '*'         [reduce using rule 4 (exp)]
    '/'         [reduce using rule 4 (exp)]
    $default    reduce using rule 4 (exp)
```

注意到状态 11 包含冲突不仅仅因为缺少`/'相对于`+', `-'和`*'的优先级, 还由于并未指定`/'的结合性.

8.2 跟踪你的分析器-Tracing Your Parser

如果 Bison 语法编译正确但是在运行的时候并未达到你想要的目的, yydeug 分析器追踪特性可以帮你指明原因.

有多种方法激活追踪机制的编译:

宏 YYDEBUG

当你编译分析器的时候,将宏 YYDEBUG 定义成非零指. 这种方式与 POSIX Yacc 兼容. 你可以使用`-DYYDEBUG=1'作为一个编译器选项或者你可以将`define YYDEBUG 1' 放入语法文件的 *Prologue* 部分.(参阅 *Prologue* 部分- The Prologue 一章).

选项 `-t', `--debug'

当你运行 Bison(参阅调用 Bison-Invoking Bison 一章)时, 使用`-t'选项. 这也与 POSIX 兼容.

指令 `%debug'

加入%debug 指令(参阅 Bison 声明总结-Bison Declaration Summary 一章). 这是一个 Bison 扩展,当 Bison 为不使用预处理器的语言输出分析器的时候很实用. 除非你要考虑 POSIX 可移植性问题, 否则这是一个很好的解决方案.

我们建议你应该总是激活调式选项以便随时进行调试.

追踪机制使用 YYFPRINTF (stderr, *format*, *args*)形式的宏调用输出信息. 在这里 *format* 和 *args* 是普通的 printf 的格式和参数. 如果你定义 YYDEBUG 为一个非零值但是没有定义 YYFPRINTF, <stdio.h>自动被加入并且 YYPRINTF 被定义为 fprintf.

一旦你使用了追踪机制编译程序, 请求一个追踪的方法是在变量 yydebug 中存储一个非零值. 你可以考编写 C 代码(也许在 main 中)做到这一点, 你也可以使用 C 调试器来改变这个值.

当 yydebug 为非零的时候,分析器执行的每一步都产生一个写入 stderr 一两行的追踪信息. 追踪信息告诉你这些东西:

- 每次调用 yylex 时,读取记号的种类.
- 每次移进记号的时候,分析器栈的深度和完整的内容. (参阅分析器状态-Parser States 一章)
- 每次归约一个规则时,这个规则是哪个规则,和在归约之后状态栈的完整内容.

弄清这些信息的意思有助于查阅由 Bison 选项`-v'产生的列表文件(listing file). (参阅调用 Bison-Invoking Bison 一章). 这个文件按照各种规则的位置展示了每个状态的意义, 还展示了每个状态会怎样处理每个输入记号. 当你阅读连续的追踪信息时, 你可以看到分析器按照它在列表文件中的指示工作. 最终你会到达发生不期望事情的地方, 并且你会发现语法的哪一个部分存在问题.

分析器文件是一个 C 程序,你可以使用 C 调式器调试它, 但是我们很难解释它在做些什么. 分析器函数是一个有限状态机解释器, 除了动作以外它反复执行相同的代码. 只有变量的值才能表示它正在语法的那个地方工作.

调试信息通常给出了每个读入记号的符号类型而不是它的语义值. 你可以定一个名为 YYPRINT 的宏来打印这个值. 如果你定义 YYPRINT, 它应带有三个参数. 分析器将传递标准 I/O 流,记号类型的数字码和记号指(从 yylval 中).

这里有一个适用于多功能计算器的 YYPRINT (参阅 mfcalc 的声明部分-Declarations for mfcalc 一章):

```
%{
  static void print_token_value (FILE *, int, YYSTYPE);
  #define YYPRINT(file, type, value) print_token_value (file, type, value)
%}

... %% ... %% ...

static void
```

```
print_token_value (FILE *file, int type, YYSTYPE value)
{
  if (type == VAR)
    fprintf (file, "%s", value.tptr->name);
  else if (type == NUM)
    fprintf (file, "%d", value.val);
}
```


9. 调用 Bison-Invoking Bison

调用 Bison 的通常方法如下:

bison *infile*

这里的 *file* 是通常以`.y'结尾的语法文件名. 分析器文件名由`.tab.c'代替`.y'取得. 因此,`bison foo.y'产生`foo.tab.c', `bison hack/foo.y'产生`hack/foo.tab.c'. 如果你在你语法文件中使用 C++代码而不是 C, 把它命名为`foo.ypp'或者`foo.y++'. 那么,输出文件的扩展名类型给定的输入 (分别为`foo.tab.cpp'和`foo.tab.c++'.

例如:

bison -d *infile.yxx*

将会产生`infile.tab.cxx'和`infile.tab.hxx',并且

bison -d -o *output.c++ infile.y*

会产生`output.c++'和`outfile.h++'.

为了与 POSIX 兼容, 标准的 Bison 发行版也包含一个名为 yacc 的脚本, 该脚本使用`-y'选项调用 Bison.

9.1 Bison 选项-Bison Options

Bison 既支持传统的单字母选项也支持可记忆长选项名称. 用`--'取代`-'来指明常长项名称. Bison 允许选项名称缩写只要它们是唯一的. 当长选项带有一个参如,如`--file-prefix', 用`='连接选项名称和参数.

这里有一个 Bison 可以是用的选项清单, 按照短选项字母顺序排列. 在它之后是一个常选项的交叉键.

操作模式:

`-h'

`--help'

打印一个 Bison 命令行选项的总结并退出.

`-V'

`--version'

打印 Bison 的版本号并退出.

`-y'

`--yacc'

与`-o y.tab.c'等价; 分析器输出文件名为`y.tab.c', 并且其它输出称为`y.output'和`y.tab.h'. 这个选项的目的是模拟 Yacc 的输出文件命名惯例. 因此,如下的 shell 脚本可以替代 Yacc, 并且 Bison 发行版包含一个这种为 POSIX 兼容的脚本.

```
#! /bin/sh
bison -y "$@"
```

调整分析器:

`-S *file*'

`--skeleton=*file*'

指明要使用的骨架(skeleton). 除非你正在开发 Bison 否你很可能不需要这个选项.

`-t'

`--debug'

在分析器文件中,定义宏 YYDEBUG 为 1,如果还没有定义它, 以便调试机制被编译. 参阅 追踪你的分析器-Tracing Your Parser.

`--locations'

%locations 的伪装.参阅 声明总结-Decl Summary.

`-p *prefix*'

`--name-prefix=*prefix*'

%name-prefix="*prefix*"的伪装. 参阅 声明总结-Decl Summary.

`-l'

`--no-lines'

在分析器文件中不放入任何的#line 预处理器命令. Bison 通常将它们放入分析器文件以便 C 编译起和调试器将错误关联到你的源文件, 语法文件. 这个选项会关联错误到分析器文件,将它视为一个独立的源文件.

`-n'

`--no-parser'

%no-parser 的伪装. 参阅 声明总结-Decl Summary.

`-k'

`--token-table'

%token-table 的伪装. 参阅 声明总结-Decl Summary.

调整输出:

`-d'

`--defines'

伪装%defines,例如,向一个额外的文件写入语法中记号类型名称的宏定义和一些其它的声明. 参阅 声明总结-Decl Summary.

`--defines=*defines-file*'

与上述相同,但是保存到文件 *defines-file*.

`-b *file-prefix*'

`--file-prefix=*prefix*'

%verbose 的伪装,例如,指明所有 Bison 输出文件的前缀. 参阅 声明总结-Decl Summary.

`-r *things*'

`--report=*things*'

向一个额外的输出文件写入如下 *things* 的详细描述清,并由逗号分隔:

state

语法,冲突(解决的和未解决的)以及 LALR 自动机.

look-ahead

包含 state 并且增加每个规则的超前扫描记号集自动机的描述.

itemset

包含 state 并且增加每个状态的全部项目集的自动机而不仅仅是它核心的自动机.

例如,在下面的语法中

`-v'

`--verbose'

%verbose 的伪装,例如,向额外的输出文件写入语法和分析器的详细描述. 参阅 声明总结-Decl Summary.

`-o *filename*'

`--output=*filename*'

为分析器文件指明 *filename*.

其它输出文件的名称像`-v'和`-d'选项的描述一样由 *filename* 构成.

`-g'

输出一个由 Bison 计算的 LALR(1)语法自动机的 VCG 定义. 如果语法文件是`foo.y', VCG 输出文件将会是 `foo.vcg'.

`--graph=*graph-file*'

-graph 的行为和`-g'的行为一样. 唯一的区别在于它含有一个指明输出图形文件的可选参数.

9.2 选项交叉键-Option Cross Key

这里有一个选项列表,按照长选项的字母排序,来帮助你找到相应的所写选项.

9.3 Yacc 库-Yacc Library

Yacc 库包含 yyerror 和 main 函数的默认实现. 通常情况下,这些默认实现没有什么用处,但是 POSIX 要求它们. 要使用 Yacc 库,使用选项`-ly'链接你的程序. 注意到 Bison 实现的 Yacc 库在 GNU 通用许可证下发行. (参阅 GNU GENERAL PUBLIC LICENSE 一章).

如果你使用 Yacc 库的 yyerror 函数, 你应该如下地声明 yyerror:

Bison 忽略 yyerror 返回的 int 值. 如果你使用 Yacc 库的 main 函数, 你的 yyparse 函数应该有如下原型:

---

10. 常见问题-Frequently Asked Questions

许多关于 Bison 的问题会偶尔出现. 这里提到一些.

10.1 分析器栈溢出-Parser Stack Overflow

我的分析器返回带有`parser stack overflow'的消息.

我能做些什么?

这个问题已经在其它地方讨论过了参阅 Recursive Rules.

10.2 我如何复位分析器-How Can I Reset the Parser

下面的现象有许多征兆,导致了下面典型的问题:

我调用了 yyparse 多次,

当输入正确时,它正确地工作;

但是当发现一个分析错误的时,所有其它的调用也失败了.

我如何才能重置 yyparse 的错误标志?

或者:

我的分析器包含了一个对`#include'类似特性的支持.

当我从 yyparse 调用 yyparse 时,即使我指明我需要一个%pure-parser,

它仍然会失败.

这些典型的问题并不产生于 Bison 自己而是产生于 Lex 生成的扫描器. 出于速度的目的,这些扫描器使用容量很大的缓冲区,

它们可能不会注意到输入文件的变化. 作为一个例子,考虑下面的源文件,

`first-line.l':

%{

```
#include <stdio.h>
#include <stdlib.h>
%}
%%
.*\n    ECHO; return 1;
%%
int
yyparse (char const *file)
{
  yyin = fopen (file, "r");
  if (!yyin)
    exit (2);
  /* One token only.  */ /* 只有一个记号 */
  yylex ();
  if (fclose (yyin) != 0)
    exit (3);
  return 0;
}

int
main (void)
{
  yyparse ("input");
  yyparse ("input");
  return 0;
}
```

如果文件`input'包含

```
input:1: Hello,
input:2: World!
```

那么你并未两次取得第一行,而是:

```
  $ flex -ofirst-line.c first-line.l
  $ gcc  -ofirst-line   first-line.c -ll
  $ ./first-line
  input:1: Hello,
  input:2: World!
```

因此,无论什么时候改变 yyin, 你必须告诉 Lex 声称的扫描器丢弃当前的缓冲转换到新的缓冲中. 这依赖于你的 Lex 的实现;

可以参阅它的文档获取更多信息. 对于 Flex,在每一个 yyin 的改变后调用`YY_FLUSH_BUFFER'可以做到这一点. 如果你的

Flex 生成扫描器需要读取多个输入流来处理类似文件包含的特性, 你可以考虑使用 FLex 函数如`yy_switch_to_buffer'来操纵多个输入缓冲.

如果你的 FLex 声称扫描器使用了开始条件(参阅 *The Flex Manual* 中`Start conditions'一章(flex)Start conditions), 你还可能复位扫描器状态,例如, 使用一个 BEGIN (0)调用,退回到开始条件.

10.3 被销毁的字符串-Strings are Destroyed

我的分析器好像销毁了旧字符串,或者它可能失去了对它们的追踪.
它报告`"bar", "bar"'或者甚至`"foo\nbar", "bar"'而不是
报告`"foo", "bar"'.

这个错误可能是发送到 Bison"错误报告"列表中最频繁的一个, 但它只是一个对扫描器角色产生的误解. 考虑如下的 Lex 代码:

```
%{
#include <stdio.h>
char *yylval = NULL;
%}
%%
.*    yylval = yytext; return 1;
\n    /* IGNORE */ /* 忽略 */
%%
int
main ()
{
  /* Similar to using $1, $2 in a Bison action.  */
  /* 类似在 Bison 动作中使用的$1,$2 */
  char *fst = (yylex (), yylval);
  char *snd = (yylex (), yylval);
  printf ("\"%s\", \"%s\"\n", fst, snd);
  return 0;
}
```

如果你编译并且运行这段代码,你得到:

```
  $ flex -osplit-lines.c split-lines.l
  $ gcc  -osplit-lines   split-lines.c -ll
  $ printf 'one\ntwo\n' | ./split-lines
  "one
```

two", "two"

这是由于 yytext 是一个在动作中用于*读取*的缓冲区, 但是如果你要保留它,你必须复制它(例如,使用 strdup). 应注意到输出可能依赖于你的 Lex 实现怎么处理 yytext. 例如当指定了 Lex 兼容性选项`-l'(它引发了选项`%array'), Flex 产生了不同的行为:

```
$ flex -l -osplit-lines.c split-lines.l
$ gcc    -osplit-lines   split-lines.c -ll
$ printf 'one\ntwo\n' | ./split-lines
"two", "two"
```

[>] [<<] [上层] [>>] [顶层] [内容] [索引] [?]

10.4 C++分析器-C++ Parsers

我如何产生使用 C++代码的分析器?

我们正致力于 Bison 的 C++输出, 但是不幸的是,由于缺少时间,骨架尚未完成. 它的功能很强大,但是由于多方面的关系, 它*可能*破坏向后兼容性的额外工作. 由于 C++骨架尚未编入文档, 我们并不认为我们自己必须超这个接口努力, 尽管如此,我们会尽最大努力保证兼容性.

另一种可能是使用正规 C 分析器并使用 C++编译器进行编译. 倘若你能忍受一些简单 C++规则,例如不能在联合体中加入"真正的类"(例如, 带有构造函数的结构体),这个就可以正常工作. 因此,在%union 中应该使用指向类的指针.

[<] [>] [<<] [上层] [>>] [顶层] [内容] [索引] [?]

10.5 实现跳转/循环-Implementing Gotos/Loops

我的简单计算器支持变量,赋值和函数,

但是我如何才能实现跳转或循环?

虽然这个文档中包含的例子很有教学性, 但它模糊了分析器(它的工作是恢复文字的结构并将它转化为程序模块) 和处理这些结构的过程(如执行)之间的区别. 这在被称为直接线性程序中工作良好. 例如直接执行模式:一个接一个的执行简单指令.

如果你需要的更丰富的模式, 你可能需要分析器生一种表示它(注:分析器)已经恢复的结构的树; 这种树被通常成为*抽象语法树(abstract syntax tree)*或者简写为 *AST*. 之后,用多种方法遍历这棵树会激活对它的执行或翻译, 这最终会导致产生一个解释起或者编译器.

这个主题超出了这个手册的讨论范围, 读者可以参阅这方面的专门文献.

[<] [>] [<<] [上层] [>>] [顶层] [内容] [索引] [?]

A. Bison 符号-Bison Symbols

变量: **@$**

在动作中,规则左手端的位置参阅 **位置概述**-Locations Overview.

变量: **@*n***

在动作中,规则右端第 *n* 个符号的位置. 参阅 **位置概述**-Locations Overview.

变量: **$$**

在动作中,规则左端的语义值. 参阅 **动作**-Actions.

变量: **$*n***

在动作中,规则右端第 *n* 个符号的语义值. 参阅 **动作**-Actions.

分隔符: **%%**

用于分隔语法规则部分和 Bison 声明部分或者 *epilogue* 部分. 参阅 Bison **语法文件的布局**-The Overall Layout of a Bison Grammar.

分隔符: **%{*code*%}**

在`%{'和`%}'之间的代码不做任何解释被直接复制到输出文件. 这些代码组成了输入文件的 *Prologue* 部分. 参阅 Bison **语法的提纲**-Outline of a Bison Grammar.

结构: **/*...*** */*

注释分隔符,类似 C.

分隔符: **:**

分隔动作的结果和它的部件. 参阅 **描述语法规则的语法**-Syntax of Grammar Rules.

分隔符: **;**

结束一个规则. 参阅 **描述语法规则的语法**-Syntax of Grammar Rules.

分隔符: **|**

分隔同一个非终结符结果的不同规则. 参阅 **描述语法规则的语法**-Syntax of Grammar Rules.

符号: **$accept**

预定义非终结符, 它的唯一规则为`$accept: *start* $end', 这里的 *start* 是开始符号. 参阅 **开始符号**- The Start-Symbol. 它不能在语法中使用.

指令: **%debug**

激活分析器调试. 参阅 **声明总结**-Decl Summary.

指令: **%defines**

为扫描器创建一个头文件的 Bison 声明. 参阅 **声明总结**-Decl Summary.

指令: **%destructor**

指明分析器如何回收被丢弃符号相关的内存. 参阅 **释放丢弃的符号**- Freeing Discarded Symbols.

指令: **%dprec**

在分析的时候赋予规则一个优先级来解决归约/归约冲突的 Bison 声明. 参阅 **编写 GLR 分析器**-Writing GLR Parsers.

符号: **$end**

用来标记流结束的预定义记号,不能在语法中使用.

符号: **error**

一个保留的用于错误恢复的记号名称. 这个记号可以用在语法规则中用来允许 Bison 分析器在不终止处理的前提下识别一个错误. 实际上,一个包含错误的句子可被认为是有效的. 遇到一个语法错误时, 记号 error 成为了当前的超前扫描记号. 与 error 相应的动作被执行,并且超前扫描记号被重置为最初引起错误的记号. 参阅 **错误恢复**-Error Recovery.

指令: **%error-verbose**

请求冗长模式的 Bison 声明, 指明了当调用 yyerror 时的错误消息字符串.

指令: **%file-prefix="*prefix*"**

设置输出文件前缀的 Bison 声明.参阅 声明总结-Decl Summary.

指令: **%glr-parser**

声称 GLR 分析器的 Bison 声明. 参阅 编写 GLR 分析器-Writing GLR Parsers.

指令: **%initial-action**

在分析器前运行代码.参阅 在分析前执行动作- Performing Actions before Parsing.

指令: **%left**

为操作符指定左结合性的 Bison 声明. 参阅 操作符优先级-Operator Precedence.

指令: **%lex-param** *{argument-declaration}*

指明 yylex 的额外参数的 Bison 声明. 参阅 纯分析器的调用惯例-Calling Conventions for Pure Parsers.

指令: **%merge**

赋予规则一个合并函数的 Bison 声明. 如果有一个归约/归约冲突的规则带有相同的合并函数, 那么这个函数被应用于两个语义值来获得单一的结果. 参阅 Writing GLR Parsers-编写 GLR 分析器.

指令: **%name-prefix="*prefix*"**

重命名外部符号的 Bison 声明.参阅 声明总结-Decl Summary.

指令: **%no-lines**

在分析器文件中避免产生#line 指令的 Bison 声明. 参阅 声明总结-Decl Summary.

指令: **%nonassoc**

声明一个无结合性记号. 参阅 操作符优先级-Operator Precedence.

指令: **%output="*filename*"**

设置分析器文件的 Bison 声明 参阅 声明总结-Decl Summary.

指令: **%parse-param** *{argument-declaration}*

指定 yyparse 接受的额外参数的 Bison 声明. 参阅 分析器函数 yyparse- The Parser Function yyparse.

指令: **%prec**

给特定的规则指定优先级的 Bison 声明. 参阅 上下文依赖优先级-Context-Dependent Precedence.

指令: **%pure-parser**

请求一个纯(可重入)分析器的 Bison 声明. 参阅 一个纯(可重入)分析器-A Pure (Reentrant) Parser.

指令: **%right**

指定记号右结合性的 Bison 声明. 参阅 操作符优先级-Operator Precedence.

指令: **%start**

指定开始符号的 Bison 声明参阅 开始符号-The Start-Symbol.

指令: **%token**

声明记号但不指定优先级. 参阅 记号类型名称-Token Type Names.

指令: **%token-table**

在分析器文件中加入符号名称表的 Bison 声明. 参阅 声明总结-Decl Summary.

指令: **%type**

声明非终结符的 Bison 声明. 参阅 非终结符-Nonterminal Symbols.

符号: **$undefined**

所有 yylex 返回的未定义值被映射到这个预定义符号. 它不能在语法中使用,[untranslated]rather,use error.

指令: **%union**

指定多种可能语义值数据类型的 Bison 声明. 参阅 值类型集-The Collection of Value Types.

宏: **YYABORT**

通过使 yyparse 立即返回 1,来伪装发生一个未恢复的语法错误的宏. 并不调用错误报告函数 yyerrpr. 参阅 分析器函数 ppyarse-The Parser Function yyparse.

宏: YYACCEPT

通过使 yyparse 立即返回 0,来伪装语言的一个完整的表达已经被读取的宏. 参阅 分析器函数 ppyarse-The Parser Function yyparse.

宏: YYBACKUP

从分析器栈中丢弃一个值并伪造一个超前扫描记号的宏. 参阅 在动作中使用的特殊特征-Special Features for Use in Actions.

变量: yychar

包含当前超前扫描记号的正数值的外部整数变量. (在一个纯分析器中,它是一个在 yyparse 中的局部变量). 错误恢复规则可能要检查这个变量. 参阅 在动作中使用的特殊特征-Special Features for Use in Actions.

变量: yyclearin

在错误恢复规则中使用的宏.它清除先前的超前扫描记号. 参阅 错误恢复-Error Recovery.

宏: YYDEBUG

使分析器带有追踪代码的宏. 参阅 跟踪你的分析器-Tracing Your Parser.

变量: yydebug

默认被置 0 的外部整数变量. 如果 yydebug 被赋予一个非零值, 分析器会输入关于输入符号和分析器动作的信息. 参阅 跟踪你的分析器-Tracing Your Parser.

宏: yyerrok

使分析器在一个语法错误之后立即恢复到正常模式的宏. 参阅 错误恢复-Error Recovery.

宏: YYERROR

一个假装刚刚发现一个语法错误的宏: 调用 yyerror 然后执行通常的错误恢复如果可能的话(参阅错误恢复-Error Recovery 一章) 或者(如果恢复是不可能的)使 yyparse 返回 1. 参阅 错误恢复-Error Recovery.

函数: yyerror

用户提供的当发现错误时被 yyparse 调用的函数. 参阅 错误报告函数 yyerror-The Error Reporting Function yyerror.

宏: YYERROR_VERBOSE

一个在 *Prologue* 部分用#define 定义的陈旧的宏, 当调用 yyerror 时,它请求详细的错误消息字符串. 你把 YYERROR_VERBOSE 定义成什么东西并没有影响, 有影响的只是你是否定义了它. 使用%error-verbose 是更好的选择.

宏: YYINITDEPTH

指定分析器栈初始大小的宏. 参阅 栈溢出-Stack Overflow.

函数: yylex

用户提供的词法分析器函数, 不带有任何参数来获得下一个记号. 参阅 词法分析器函数 yylex-The Lexical Analyzer Function yylex.

宏: YYLEX_PARAM

一个用于指明 yyparse 传递到 yylex 的额外参数(或者额外参数列表)的陈旧的宏. 不赞成继续使用这个宏,它只被 Yacc 类似分析器支持. 参阅 纯分析器的调用惯例-Calling Conventions for Pure Parsers.

变量: yylloc

yylex 应该将一个记号的行列号放入这个外部变量. (在纯分析器中,它是一个 yyparse 的局部变量, 它的地址被传递到 yylex). 如果你在语法动作中不使用`@'特性,你可以忽略这个变量. 参阅 记号的文字位置-Textual Locations of Tokens.

Type: **YYLTYPE**

yyloc 的数据类型;默认地为一个带有四个成员的结构体. 参阅 位置的数据类型-Data Types of Locations.

变量: **yylval**

yylex 应该将记号相关的语义值放入这个变量. (在纯分析器中,它是一个 yyparse 中的局部变量,并且它的地址被传递到 yylex.) 参阅 记号的语义值-Semantic Values of Tokens.

宏: **YYMAXDEPTH**

指明分析器栈最大容的宏. 参阅 栈溢出-Stack Overflow.

变量: **yynerrs**

一个全局变量,每次出现语法错误时自增 1. (在纯分析器中,它是一个 yyparse 中的局部变量.) 参阅 错误报告函数 yyerror-The Error Reporting Function yyerror.

函数: **yyparse**

Bison 产生的分析器函数;调用这个函数开始分析. 参阅 分析器函数 yyparse-The Parser Function yyparse.

宏: **YYPARSE_PARAM**

指明 yyparse 应该接受的参数名成的陈旧宏. 不赞成继续使用这个宏,它只被 Yacc 类似分析器支持. 参阅 纯分析器的调用惯例-Calling Conventions for Pure Parsers.

宏: **YYRECOVERING**

一个宏,它的值指明了分析器是否正在从错误中恢复. 参阅 动作中使用的特殊特征-Special Features for Use in Actions.

宏: **YYSTACK_USE_ALLOCA**

用于控制当 C 语言 LALR(1)分析器需要扩展它的栈时,alloca 的使用. 如果定义为 0,分析器会使用 malloc 来扩展它的栈. 如果定义为 1,分析器则会使用 alloca. 除了 0 和 1 以外的值保留用于 Bison 以后的扩展. 如果没有被定义, YYSTACK_USA_ALLOCA 默认为 0.

如果你定义 YYSTACK_USE_ALLOCA 为 1, 你有责任使 alloca 是可见的, 例如,使用 GCC,或者包含<stdlib.h>. 此外,在更普通的情况下, 如果你的代码可能运行在一个有限栈容量和不可信任栈溢出检查的主机上的时候, 你应将 YYMAXDEPTH 设为当调用 alloca 时, 在一个在任何目标主机不会产生栈溢出的值. 你可以检查 Bison 生成的代码来决定适当的数值. 这需要在底层详细实现的专业技术.

类型: **YYSTYPE**

语义值的数据类型;默认为 int. 参阅 语义值的数据类型-Data Types of Semantic Values.

B. 词汇表-Glossary

Backus-Naur Form (BNF; also called "Backus Normal Form")

Backus-Naur 范式 (BNF; 也被称为 "Backus 正规范式")

由 John Backus 倡导的,最初用于描述上下文无关文法的正式方法. 在 Peter Naur 他的称为 Algol60 报告的 1960-01-02 委员会文档中得到少许改进. 参阅 语言与上下文无关文法-Languages and Context-Free Grammars.

Context-free grammars

上下文无关文法

描述可以使用而不考虑上下文的规则的文法. 因此,如果有一个规则说一个整数可做为一个表达式使用, 那么,整数 *在任何地方*都是一个允许的表达式. 参阅 语言与上下文无关文法-Languages and Context-Free Grammars.

Dynamic allocation

动态分配

在执行期间分配内存,而不是在编译期间或者进入一个函数时.

Empty string

空字符串

模拟集合论中的空集, 空字符串是长度为 0 的字符串.

Finite-state stack machine

有限状态栈机

一种含有多种离散状态的机器,每个时刻只有一种状态. 当处理输入的时候,机器按照机器逻辑的指定从一个状态转换到另一个状态. 对于分析器来说,输入就是要分析的语言, 状态对应于语法规则中的各个阶段. 参阅 Bison 分析器算法-The Bison Parser Algorithm.

Generalized LR (GLR)

通用 LR (GLR)

一种可以处理包括那些不是 LALR(1)的上下文无关文法的分析算法. 它用来解决 Bison 通常的 LALR(1)算法不能解决的冲突. 它高效地分裂成多个分析器,尝试所有可能的分析,丢弃那些在额外上下文提示下失败的分析器. 参阅 通用 LR 分析-Generalized LR Parsing.

Grouping

分组

一个(通常)在语法上可再分的语言结构; 例如 C 中的`expression'或者'declaration'. 参阅 语言和上下文无关文法-Languages and Context-Free Grammars.

Infix operator

中缀操作符

放置在操作数中间指定某些操作的算术操作符.

Input stream

输入流

在设备或程序间的连续数据流.

Language construct

语言结构

一种语言的典型应用模式. 例如,一种 C 语言的结构是 if 语句. 参阅 语言和上下文无关文法-Languages and Context-Free Grammars.

Left associativity

左结合性

拥有左结合性的操作符被从左至右地分析: `a+b+C'首先计算`a+b'然后和`c'一起计算. 参阅 操作符优先级-Operator Precedence.

Left recursion

左递归

一个结果符号同样是第一个部件符号的规则; 例如: `expseq1 : expseq1 ',' exp;'. 参阅 递归规则-Recursive Rules.

Left-to-right parsing

自左至右分析

同过自左至右分析一个个地分析记号来分析一个句子. 参阅 Bison 分析器算法-The Bison Parser Algorithm.

Lexical analyzer (scanner)

词法分析器 (扫描器)

一个读取输入流并逐个返回记号的函数. 参阅 词法分析器函数 yylex-The Lexical Analyzer Function yylex.

Lexical tie-in

词法关联

一个由语法规则动作设置的标志用来改变分析记号的方法. 参阅 词法关联-Lexical tie-in.

Literal string token

字符串文字记号

一个由两个或者更多字符组成的记号.参阅 符号-Symbols.

Look-ahead token

超前扫描记号

一个已经读取但未移进的记号. 参阅 超前扫描记号-Look-Ahead Tokens.

LALR(1)

一种 Bison(像大多数其它分析器一样)可以处理的上下文无关文法. LR(1)的子集. 参阅 令人迷惑的归约/归约冲突-Mysterious Reduce/Reduce Conflicts.

LR(1)

一种上下文无关文法, 它在大多数时侯需要一个超前扫描记号来消除任何输入片段的歧义.

Nonterminal symbol

非终结符

一个代表可以通过规则表达为更小结构的语法结构; 换句话说,一个不是记号的结构. 参阅 符号-Symbols.

Parser

分析器

一个靠分析从词法分析器传递过来的记号的语法结构来识别有效句子的函数.

Postfix operator

后缀操作符

放置在操作数后执行某些操作的算术操作符.

Reduction

归约

依照一个语法规则将非终结符和/或终结符的序列替换为非终结符. 参阅 Bison 分析器算法-The Bison Parser Algorithm.

Reentrant

可重入

一个可重入的子程序是一个可以被任意次地并行调用并且调用间相互不干扰的子程序. 参阅 一个纯(可重入)分析器-A Pure (Reentrant) Parser.

Reverse polish notation

逆波兰记号

一种所有操作符都是后缀操作符的语言.

Right recursion

右递归

一个结果记号也是它最后部件记号的规则; 例如 `expseq1: exp ',' expseq1;'. 参阅 递归规则-Recursive rules.

Semantics-语义

在计算机语言中,语义由每个语言实例的动作指明, 例如,每个语句的意义. 参阅 定义语言的语义-Defining Language Semantics.

Shift-移进

我们说一个分析器移进当它决定进从流中进一步分析输入而不是立即归约一些已经识别的规则. 参阅 Bison 分析器算法-The Bison Parser Algorithm.

Single-character literal

单字符文字

一个被识别和解释为它自己的单一字符. 参阅 从正规文法转换到 Bison 输入-From Formal Rules to Bison Input.

Start symbol

开始符号

代表要分析语言的一个完整有效的表述的非终结符. 在语言描述中,开始符号通常被列为第一个非终结符. 参阅 开始符号-The Start-Symbol.

Symbol table

符号表

用来识别和使用已经存在的符号信息的数据结构. 在进行分析器符号表存储符号名称和相关数据. 参阅 多功能计算器-Multi-function Calc.

Syntax error

语法错误

一个发生在分析无效语法的输入流时的错误. 参阅 错误恢复-Error Recovery.

Token

记号

一个基本的不可再分的语言单元. 在语法中,描述一个记号的符号是终结符. Bison 分析器的输入是来自词法分析器的记号流. 参阅 Symbols-符号.

Terminal symbol

终结符

在语法中不包含含规则因此语法上不可再分的语法符号. 它表示的输入片段是一个记号. 参阅 语言与上下文无关文法-Languages and Context-Free Grammars.

C. 复制这个手册-Copying This Manual

C.1 GNU Free Documentation License

Version 1.2, November 2002

1. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

2. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

3. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

4. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque

copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
8. Include an unaltered copy of this License.
9. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
10. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it

was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

11. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
12. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
13. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
14. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
15. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

6. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of

the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

7. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

8. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

9. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of

this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

10. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

11. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

C.1.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) *year your name*.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is included in the section entitled ``GNU
Free Documentation License''.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being *list their titles*, with
the Front-Cover Texts being *list*, and with the Back-Cover Texts
being *list*.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

索引-Index

?

?

符号(symbol)	3.2 符号,终结符和非终结符-Symbols, Terminal and Nonterminal
符号(symbols (abstract))	1.1 语言与上下文无关文法-Languages and Context-Free Grammars
符号表实例(symbol table example)	2.5.3 mfcalc 的符号表-The mfcalc Symbol Table
符号类型(token type)	3.2 符号,终结符和非终结符-Symbols, Terminal and Nonterminal
符号类型名称(token type names), 声明(declaring)	3.7.1 符号类型名称-Token Type Names

---

?

规则(rule), 没用处(useless)	8.1 理解你的分析器-Understanding Your Parser
规则(rule), 指明的(pointed)	8.1 理解你的分析器-Understanding Your Parser
规则语法(rule syntax)	3.3 描述语法规则的语法-Syntax of Grammar Rules
规则中动作(mid-rule actions)	3.5.5 规则中的动作-Actions in Mid-Rule
归约(reduction)	5. Bison 分析器算法-The Bison Parser Algorithm
归约/归约 冲突 (reduce/reduce conflicts)	1.5 编写 GLR 分析器-Writing GLR Parsers
归约/归约冲突 (reduce/reduce conflict)	5.6 归约/归约冲突-Reduce/Reduce Conflicts
归约/归约冲突 (reduce/reduce conflicts)	1.5.1 使用 GLR 分析器分析非歧义文法

?

核心(core), 项目集(item set)	8.1 理解你的分析器-Understanding Your Parser
核心(kernel), 项目集(item set)	8.1 理解你的分析器-Understanding Your Parser

---

?

计算器(calculator), 多功能(multi-function)	2.5 多功能计算器:mfcalc-Multi-Function Calculator: mfcalc
计算器(calculator), 简单(simple)	2.1 逆波兰记号计算器-Reverse Polish Notation Calculator
计算器(calculator), 位置追踪(location tracking)	2.4 带有位置追踪的计算器:ltcalc-Location Tracking Calculator: ltcalc

type)	Types of Semantic Values
默认位置类型(default location type)	3.6.1 位置的数据类型-Data Type of Locations
默认栈容量限制(default stack limit)	5.9 栈溢出以及如何避免它-Stack Overflow, and How to Avoid It
逆波兰记号(reverse polish notation)	2.1 逆波兰记号计算器-Reverse Polish Notation Calculator

?

歧义文法(ambiguous grammars)	1.1 语言与上下文无关文法-Languages and Context-Free Grammars
歧义文法(ambiguous grammars)	5.8 通用 LR (GLR)分析-Generalized LR (GLR) Parsing

?

上下文无关文法(context-free grammar)	1.1 语言与上下文无关文法-Languages and Context-Free Grammars
上下文依赖优先级(context-dependent precedence)	5.4 上下文依赖优先级-Context-Dependent Precedence
声明(declarations)	3.1.1 *Prologue* 部分-The prologue
声明(declarations), Bison	3.7 Bison 声明-Bison Declarations
声明(declarations), Bison(简介)(Bison (introduction))	3.1.2 *Bison Declarations* 部分-The Bison Declarations Section
声明部分(declarations section)	3.1.1 *Prologue* 部分-The prologue
声明操作符优先级(declaring operator precedence)	3.7.2 操作符优先级-Operator Precedence
声明符号类型名称(declaring token type names)	3.7.1 符号类型名称-Token Type Names
声明开始符号(declaring the start symbol)	3.7.8 开始符号-The Start-Symbol
声明文字串记号(declaring literal string tokens)	3.7.1 符号类型名称-Token Type Names
声明值类型(declaring value types)	3.7.3 值类型集-The Collection of Value Types
声明值类型(declaring value types), 非终结符(nonterminals)	3.7.4 非终结符-Nonterminal Symbols
声明总结(declaration summary)	3.7.10 Bison 声明总结-Bison Declaration Summary

?

实例(examples), 简单	2. 实例-Examples

(simple)

使用 Bison(using Bison)	1.8 使用 Bison 的流程-Stages in Using Bison</tr> <TR>	使用 Bison 的流程 (stages in using Bison)	1.8 使用 Bison 的流程-Stages in Using Bison
释放被丢弃的符号(freeing discarded symbols)	3.7.6 释放被丢弃的符号-Freeing Discarded Symbols		

?

通用 LR (GLR)分析 (generalized LR (GLR) parsing)	5.8 通用 LR (GLR)分析-Generalized LR (GLR) Parsing
通用 LR(GLR)分析 (generalized LR (GLR) parsing)	1.1 语言与上下文无关文法-Languages and Context-Free Grammars
通用 LR(GLR)分析 (generalized LR (GLR) parsing), 非歧义文法 (unambiguous grammars)	1.5.1 使用 GLR 分析器分析非歧义文法
通用 LR 分析(generalized LR (GLR) parsing)	1.5 编写 GLR 分析器-Writing GLR Parsers

---

?

位置(location)	1.6 位置-Locations
位置(location)	3.6 追踪位置-Tracking Locations
位置(location), 文字的 (textual)	3.6 追踪位置-Tracking Locations
位置(location), 原文的 (textual)	1.6 位置-Locations
位置的数据类型(data type of locations)	3.6.1 位置的数据类型-Data Type of Locations
位置动作(location actions)	3.6.2 动作和位置-Actions and Locations
位置追踪(location tracking calculator)	2.4 带有位置追踪的计算器: ltcalc-Location Tracking Calculator: ltcalc
文法(grammar), 上下文无关 (context-free)	1.1 语言与上下文无关文法-Languages and Context-Free Grammars
文件格式(file format)	1.9 Bison 语法文件的整体布局-The Overall Layout of a Bison Grammar
文字串记号(literal string token)	3.2 符号,终结符和非终结符-Symbols, Terminal and Nonterminal

目录