gnu:guide/bison

COLLABORATORS

	TITLE : gnu:guide/bison	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 22, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1 gnu:guide/bison **1**

1.1 gnu:guide/bison.guide . . . 1
1.2 bison.guide/Introduction . . . 4
1.3 bison.guide/Conditions . . . 4
1.4 bison.guide/Copying . . . 5
1.5 bison.guide/Concepts . . . 11
1.6 bison.guide/Language and Grammar . . . 12
1.7 bison.guide/Grammar in Bison . . . 14
1.8 bison.guide/Semantic Values . . . 14
1.9 bison.guide/Semantic Actions . . . 15
1.10 bison.guide/Bison Parser . . . 16
1.11 bison.guide/Stages . . . 17
1.12 bison.guide/Grammar Layout . . . 17
1.13 bison.guide/Examples . . . 18
1.14 bison.guide/RPN Calc . . . 18
1.15 bison.guide/Rpcalc Decls . . . 19
1.16 bison.guide/Rpcalc Rules . . . 20
1.17 bison.guide/Rpcalc Input . . . 21
1.18 bison.guide/Rpcalc Line . . . 21
1.19 bison.guide/Rpcalc Expr . . . 22
1.20 bison.guide/Rpcalc Lexer . . . 23
1.21 bison.guide/Rpcalc Main . . . 24
1.22 bison.guide/Rpcalc Error . . . 24
1.23 bison.guide/Rpcalc Gen . . . 25
1.24 bison.guide/Rpcalc Compile . . . 25
1.25 bison.guide/Infix Calc . . . 26
1.26 bison.guide/Simple Error Recovery . . . 27
1.27 bison.guide/Multi-function Calc . . . 28
1.28 bison.guide/Mfcalc Decl . . . 29
1.29 bison.guide/Mfcalc Rules . . . 30

1.30 bison.guide/Mfcalc Symtab . . . 31
1.31 bison.guide/Exercises . . . 34
1.32 bison.guide/Grammar File . . . 34
1.33 bison.guide/Grammar Outline . . . 35
1.34 bison.guide/C Declarations . . . 35
1.35 bison.guide/Bison Declarations . . . 36
1.36 bison.guide/Grammar Rules . . . 36
1.37 bison.guide/C Code . . . 36
1.38 bison.guide/Symbols . . . 37
1.39 bison.guide/Rules . . . 38
1.40 bison.guide/Recursion . . . 39
1.41 bison.guide/Semantics . . . 40
1.42 bison.guide/Value Type . . . 40
1.43 bison.guide/Multiple Types . . . 41
1.44 bison.guide/Actions . . . 41
1.45 bison.guide/Action Types . . . 43
1.46 bison.guide/Mid-Rule Actions . . . 43
1.47 bison.guide/Declarations . . . 46
1.48 bison.guide/Token Decl . . . 46
1.49 bison.guide/Precedence Decl . . . 47
1.50 bison.guide/Union Decl . . . 48
1.51 bison.guide/Type Decl . . . 48
1.52 bison.guide/Expect Decl . . . 48
1.53 bison.guide/Start Decl . . . 49
1.54 bison.guide/Pure Decl . . . 49
1.55 bison.guide/Decl Summary . . . 50
1.56 bison.guide/Multiple Parsers . . . 51
1.57 bison.guide/Interface . . . 51
1.58 bison.guide/Parser Function . . . 52
1.59 bison.guide/Lexical . . . 52
1.60 bison.guide/Calling Convention . . . 53
1.61 bison.guide/Token Values . . . 54
1.62 bison.guide/Token Positions . . . 54
1.63 bison.guide/Pure Calling . . . 55
1.64 bison.guide/Error Reporting . . . 55
1.65 bison.guide/Action Features . . . 56
1.66 bison.guide/Algorithm . . . 58
1.67 bison.guide/Look-Ahead . . . 59
1.68 bison.guide/Shift-Reduce . . . 60

1.69 bison.guide/Precedence . . . 61
1.70 bison.guide/Why Precedence . . . 61
1.71 bison.guide/Using Precedence . . . 62
1.72 bison.guide/Precedence Examples . . . 63
1.73 bison.guide/How Precedence . . . 63
1.74 bison.guide/Contextual Precedence . . . 63
1.75 bison.guide/Parser States . . . 64
1.76 bison.guide/Reduce-Reduce . . . 65
1.77 bison.guide/Mystery Conflicts . . . 67
1.78 bison.guide/Stack Overflow . . . 68
1.79 bison.guide/Error Recovery . . . 69
1.80 bison.guide/Context Dependency . . . 71
1.81 bison.guide/Semantic Tokens . . . 71
1.82 bison.guide/Lexical Tie-ins . . . 73
1.83 bison.guide/Tie-in Recovery . . . 74
1.84 bison.guide/Debugging . . . 75
1.85 bison.guide/Invocation . . . 76
1.86 bison.guide/Bison Options . . . 76
1.87 bison.guide/Option Cross Key . . . 78
1.88 bison.guide/VMS Invocation . . . 79
1.89 bison.guide/Table of Symbols . . . 79
1.90 bison.guide/Glossary . . . 82
1.91 bison.guide/Index . . . 85

Chapter 1

gnu:guide/bison

1.1 gnu:guide/bison.guide

```
   This manual documents version 1.20 of Bison.


  Introduction
  Conditions
  Copying             The GNU General Public License says
                        how you can copy and share Bison

Tutorial sections:
  Concepts            Basic concepts for understanding Bison.
  Examples            Three simple explained examples of using Bison.

Reference sections:
  Grammar File        Writing Bison declarations and rules.
  Interface           C-language interface to the parser function `yyparse'.
  Algorithm           How the Bison parser works at run-time.
  Error Recovery      Writing rules for error recovery.
  Context Dependency    What to do if your language syntax is too
                        messy for Bison to handle straightforwardly.
  Debugging           Debugging Bison parsers that parse wrong.
  Invocation          How to run Bison (to produce the parser source file).
  Table of Symbols    All the keywords of the Bison language are explained.
  Glossary            Basic concepts are explained.
  Index               Cross-references to the text.

 -- The Detailed Node Listing --

The Concepts of Bison

  Language and Grammar    Languages and context-free grammars,
                            as mathematical ideas.
  Grammar in Bison    How we represent grammars for Bison's sake.
  Semantic Values     Each token or syntactic grouping can have
                        a semantic value (the value of an integer,
                        the name of an identifier, etc.).
  Semantic Actions    Each rule can have an action containing C code.
  Bison Parser        What are Bison's input and output,
                        how is the output used?
```

```
  Stages              Stages in writing and running Bison grammars.
  Grammar Layout      Overall structure of a Bison grammar file.

Examples

  RPN Calc            Reverse polish notation calculator;
                        a first example with no operator precedence.
  Infix Calc          Infix (algebraic) notation calculator.
                        Operator precedence is introduced.
  Simple Error Recovery    Continuing after syntax errors.
  Multi-function Calc      Calculator with memory and trig functions.
                        It uses multiple data-types for semantic values.
  Exercises           Ideas for improving the multi-function calculator.

Reverse Polish Notation Calculator

  Decls                 Bison and C declarations for rpcalc.
  Rules                 Grammar Rules for rpcalc, with explanation.
  Lexer                 The lexical analyzer.
  Main                  The controlling function.
  Error                 The error reporting function.
  Gen                   Running Bison on the grammar file.
  Comp                  Run the C compiler on the output code.

Grammar Rules for 'rpcalc'

  Rpcalc Input
  Rpcalc Line
  Rpcalc Expr

Multi-Function Calculator: 'mfcalc'

  Decl                    Bison declarations for multi-function calculator.
  Rules                   Grammar rules for the calculator.
  Symtab                  Symbol table management subroutines.

Bison Grammar Files

  Grammar Outline     Overall layout of the grammar file.
  Symbols             Terminal and nonterminal symbols.
  Rules               How to write grammar rules.
  Recursion           Writing recursive rules.
  Semantics           Semantic values and actions.
  Declarations        All kinds of Bison declarations are described here.
  Multiple Parsers    Putting more than one Bison parser in one program.

Outline of a Bison Grammar

  C Declarations      Syntax and usage of the C declarations section.
  Bison Declarations    Syntax and usage of the Bison declarations section.
  Grammar Rules       Syntax and usage of the grammar rules section.
  C Code              Syntax and usage of the additional C code section.

Defining Language Semantics

  Value Type          Specifying one data type for all semantic values.
  Multiple Types      Specifying several alternative data types.
```

```
  Actions             An action is the semantic definition of a grammar rule.
  Action Types        Specifying data types for actions to operate on.
  Mid-Rule Actions    Most actions go at the end of a rule.
                      This says when, why and how to use the exceptional
                        action in the middle of a rule.

Bison Declarations

  Token Decl          Declaring terminal symbols.
  Precedence Decl     Declaring terminals with precedence and associativity.
  Union Decl          Declaring the set of all semantic value types.
  Type Decl           Declaring the choice of type for a nonterminal symbol.
  Expect Decl         Suppressing warnings about shift/reduce conflicts.
  Start Decl          Specifying the start symbol.
  Pure Decl           Requesting a reentrant parser.
  Decl Summary        Table of all Bison declarations.

Parser C-Language Interface

  Parser Function     How to call 'yyparse' and what it returns.
  Lexical             You must supply a function 'yylex'
                        which reads tokens.
  Error Reporting     You must supply a function 'yyerror'.
  Action Features     Special features for use in actions.

The Lexical Analyzer Function 'yylex'

  Calling Convention    How 'yyparse' calls 'yylex'.
  Token Values        How 'yylex' must return the semantic value
                        of the token it has read.
  Token Positions     How 'yylex' must return the text position
                        (line number, etc.) of the token, if the
                         actions want that.
  Pure Calling        How the calling convention differs
                        in a pure parser (see A Pure (Reentrant) Parser).

The Bison Parser Algorithm

  Look-Ahead          Parser looks one token ahead when deciding what to do.
  Shift-Reduce        Conflicts: when either shifting or reduction is valid.
  Precedence          Operator precedence works by resolving conflicts.
  Contextual Precedence    When an operator's precedence depends on context.
  Parser States       The parser is a finite-state-machine with stack.
  Reduce-Reduce       When two rules are applicable in the same situation.
  Mystery Conflicts    Reduce/reduce conflicts that look unjustified.
  Stack Overflow      What happens when stack gets full.  How to avoid it.

Operator Precedence

  Why Precedence      An example showing why precedence is needed.
  Using Precedence    How to specify precedence in Bison grammars.
  Precedence Examples    How these features are used in the previous example.
  How Precedence      How they work.

Handling Context Dependencies

  Semantic Tokens     Token parsing can depend on the semantic context.
```

```
  Lexical Tie-ins     Token parsing can depend on the syntactic context.
  Tie-in Recovery     Lexical tie-ins have implications for how
                        error recovery rules must be written.

Invoking Bison

  Bison Options       All the options described in detail,
      in alphabetical order by short options.
  Option Cross Key    Alphabetical list of long options.
  VMS Invocation      Bison command syntax on VMS.
```

1.2 bison.guide/Introduction

```
Introduction
************

   "Bison" is a general-purpose parser generator that converts a
grammar description for an LALR(1) context-free grammar into a C
program to parse that grammar.  Once you are proficient with Bison, you
may use it to develop a wide range of language parsers, from those used
in simple desk calculators to complex programming languages.

   Bison is upward compatible with Yacc: all properly-written Yacc
grammars ought to work with Bison with no change.  Anyone familiar with
Yacc should be able to use Bison with little trouble.  You need to be
fluent in C programming in order to use Bison or to understand this
manual.

   We begin with tutorial chapters that explain the basic concepts of
using Bison and show three explained examples, each building on the
last.  If you don't know Bison or Yacc, start by reading these
chapters.  Reference chapters follow which describe specific aspects of
Bison in detail.

   Bison was written primarily by Robert Corbett; Richard Stallman made
it Yacc-compatible.  This edition corresponds to version 1.20 of Bison.
```

1.3 bison.guide/Conditions

```
Conditions for Using Bison
**************************

   Bison grammars can be used only in programs that are free software.
This is in contrast to what happens with the GNU C compiler and the
other GNU programming tools.

   The reason Bison is special is that the output of the Bison
utility--the Bison parser file--contains a verbatim copy of a sizable
piece of Bison, which is the code for the 'yyparse' function.  (The
actions from your grammar are inserted into this function at one point,
```

but the rest of the function is not changed.)

As a result, the Bison parser file is covered by the same copying conditions that cover Bison itself and the rest of the GNU system: any program containing it has to be distributed under the standard GNU copying conditions.

Occasionally people who would like to use Bison to develop proprietary programs complain about this.

We don't particularly sympathize with their complaints. The purpose of the GNU project is to promote the right to share software and the practice of sharing software; it is a means of changing society. The people who complain are planning to be uncooperative toward the rest of the world; why should they deserve our help in doing so?

However, it's possible that a change in these conditions might encourage computer companies to use and distribute the GNU system. If so, then we might decide to change the terms on 'yyparse' as a matter of the strategy of promoting the right to share. Such a change would be irrevocable. Since we stand by the copying permissions we have announced, we cannot withdraw them once given.

We mustn't make an irrevocable change hastily. We have to wait until there is a complete GNU system and there has been time to learn how this issue affects its reception.

1.4 bison.guide/Copying

GNU GENERAL PUBLIC LICENSE
****

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA

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.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. 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.

1. 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.

2. 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:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. 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.

   c. 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.

3. 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:

   a. 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,

   b. 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,

   c. 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.

4. 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.

5. 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.

6. 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.

7. 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.

8. 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.

9. 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.

10. 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.

NO WARRANTY

11. 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.

12. 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.

END OF TERMS AND CONDITIONS

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) 19YY 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., 675 Mass Ave, Cambridge, MA 02139, 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) 19YY 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.
```

1.5 bison.guide/Concepts

```
The Concepts of Bison
*********************
```

This chapter introduces many of the basic concepts without which the details of Bison will not make sense. If you do not already know how to use Bison or Yacc, we suggest you start by reading this chapter carefully.

```
Language and Grammar    Languages and context-free grammars,
                          as mathematical ideas.
Grammar in Bison    How we represent grammars for Bison's sake.
Semantic Values     Each token or syntactic grouping can have
                      a semantic value (the value of an integer,
                      the name of an identifier, etc.).
Semantic Actions    Each rule can have an action containing C code.
Bison Parser        What are Bison's input and output,
                      how is the output used?
Stages              Stages in writing and running Bison grammars.
Grammar Layout      Overall structure of a Bison grammar file.
```

1.6 bison.guide/Language and Grammar

Languages and Context-Free Grammars
==========

In order for Bison to parse a language, it must be described by a "context-free grammar". This means that you specify one or more "syntactic groupings" and give rules for constructing them from their parts. For example, in the C language, one kind of grouping is called an 'expression'. One rule for making an expression might be, "An expression can be made of a minus sign and another expression". Another would be, "An expression can be an integer". As you can see, rules are often recursive, but there must be at least one rule which leads out of the recursion.

The most common formal system for presenting such rules for humans to read is "Backus-Naur Form" or "BNF", which was developed in order to specify the language Algol 60. Any grammar expressed in BNF is a context-free grammar. The input to Bison is essentially machine-readable BNF.

Not all context-free languages can be handled by Bison, only those that are LALR(1). In brief, this means that it must be possible to tell how to parse any portion of an input string with just a single token of look-ahead. Strictly speaking, that is a description of an LR(1) grammar, and LALR(1) involves additional restrictions that are hard to explain simply; but it is rare in actual practice to find an LR(1) grammar that fails to be LALR(1). See Mysterious Reduce/Reduce Conflicts, for more information on this.

In the formal grammatical rules for a language, each kind of syntactic unit or grouping is named by a "symbol". Those which are built by grouping smaller constructs according to grammatical rules are called "nonterminal symbols"; those which can't be subdivided are called "terminal symbols" or "token types". We call a piece of input corresponding to a single terminal symbol a "token", and a piece

corresponding to a single nonterminal symbol a "grouping".

We can use the C language as an example of what symbols, terminal and nonterminal, mean. The tokens of C are identifiers, constants (numeric and string), and the various keywords, arithmetic operators and punctuation marks. So the terminal symbols of a grammar for C include ‘identifier’, ‘number’, ‘string’, plus one symbol for each keyword, operator or punctuation mark: ‘if’, ‘return’, ‘const’, ‘static’, ‘int’, ‘char’, ‘plus-sign’, ‘open-brace’, ‘close-brace’, ‘comma’ and many more. (These tokens can be subdivided into characters, but that is a matter of lexicography, not grammar.)

Here is a simple C function subdivided into tokens:

```
int             /* keyword ‘int’ */
square (x)      /* identifier, open-paren, */
                /* identifier, close-paren */
     int x;     /* keyword ‘int’, identifier, semicolon */
{               /* open-brace */
  return x * x; /* keyword ‘return’, identifier, */
                /* asterisk, identifier, semicolon */
}               /* close-brace */
```

The syntactic groupings of C include the expression, the statement, the declaration, and the function definition. These are represented in the grammar of C by nonterminal symbols ‘expression’, ‘statement’, ‘declaration’ and ‘function definition’. The full grammar uses dozens of additional language constructs, each with its own nonterminal symbol, in order to express the meanings of these four. The example above is a function definition; it contains one declaration, and one statement. In the statement, each ‘x’ is an expression and so is ‘x * x’.

Each nonterminal symbol must have grammatical rules showing how it is made out of simpler constructs. For example, one kind of C statement is the ‘return’ statement; this would be described with a grammar rule which reads informally as follows:

> A ‘statement’ can be made of a ‘return’ keyword, an ‘expression’ and a ‘semicolon’.

There would be many other rules for ‘statement’, one for each kind of statement in C.

One nonterminal symbol must be distinguished as the special one which defines a complete utterance in the language. It is called the "start symbol". In a compiler, this means a complete input program. In the C language, the nonterminal symbol ‘sequence of definitions and declarations’ plays this role.

For example, ‘1 + 2’ is a valid C expression--a valid part of a C program--but it is not valid as an *entire* C program. In the context-free grammar of C, this follows from the fact that ‘expression’ is not the start symbol.

The Bison parser reads a sequence of tokens as its input, and groups the tokens using the grammar rules. If the input is valid, the end

result is that the entire token sequence reduces to a single grouping whose symbol is the grammar's start symbol. If we use a grammar for C, the entire input must be a 'sequence of definitions and declarations'. If not, the parser reports a syntax error.

1.7 bison.guide/Grammar in Bison

From Formal Rules to Bison Input
==========

A formal grammar is a mathematical construct. To define the language for Bison, you must write a file expressing the grammar in Bison syntax: a "Bison grammar" file. See Bison Grammar Files.

A nonterminal symbol in the formal grammar is represented in Bison input as an identifier, like an identifier in C. By convention, it should be in lower case, such as 'expr', 'stmt' or 'declaration'.

The Bison representation for a terminal symbol is also called a "token type". Token types as well can be represented as C-like identifiers. By convention, these identifiers should be upper case to distinguish them from nonterminals: for example, 'INTEGER', 'IDENTIFIER', 'IF' or 'RETURN'. A terminal symbol that stands for a particular keyword in the language should be named after that keyword converted to upper case. The terminal symbol 'error' is reserved for error recovery. See Symbols.

A terminal symbol can also be represented as a character literal, just like a C character constant. You should do this whenever a token is just a single character (parenthesis, plus-sign, etc.): use that same character in a literal as the terminal symbol for that token.

The grammar rules also have an expression in Bison syntax. For example, here is the Bison rule for a C 'return' statement. The semicolon in quotes is a literal character token, representing part of the C syntax for the statement; the naked semicolon, and the colon, are Bison punctuation used in every rule.

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

See Syntax of Grammar Rules.

1.8 bison.guide/Semantic Values

Semantic Values
==========

A formal grammar selects tokens only by their classifications: for example, if a rule mentions the terminal symbol 'integer constant', it

means that *any* integer constant is grammatically valid in that position. The precise value of the constant is irrelevant to how to parse the input: if ‘x+4’ is grammatical then ‘x+1’ or ‘x+3989’ is equally grammatical.

But the precise value is very important for what the input means once it is parsed. A compiler is useless if it fails to distinguish between 4, 1 and 3989 as constants in the program! Therefore, each token in a Bison grammar has both a token type and a "semantic value". See Defining Language Semantics, for details.

The token type is a terminal symbol defined in the grammar, such as ‘INTEGER’, ‘IDENTIFIER’ or ‘’,’’. It tells everything you need to know to decide where the token may validly appear and how to group it with other tokens. The grammar rules know nothing about tokens except their types.

The semantic value has all the rest of the information about the meaning of the token, such as the value of an integer, or the name of an identifier. (A token such as ‘’,’’ which is just punctuation doesn’t need to have any semantic value.)

For example, an input token might be classified as token type ‘INTEGER’ and have the semantic value 4. Another input token might have the same token type ‘INTEGER’ but value 3989. When a grammar rule says that ‘INTEGER’ is allowed, either of these tokens is acceptable because each is an ‘INTEGER’. When the parser accepts the token, it keeps track of the token’s semantic value.

Each grouping can also have a semantic value as well as its nonterminal symbol. For example, in a calculator, an expression typically has a semantic value that is a number. In a compiler for a programming language, an expression typically has a semantic value that is a tree structure describing the meaning of the expression.

1.9 bison.guide/Semantic Actions

Semantic Actions
==========

In order to be useful, a program must do more than parse input; it must also produce some output based on the input. In a Bison grammar, a grammar rule can have an "action" made up of C statements. Each time the parser recognizes a match for that rule, the action is executed. See Actions.

Most of the time, the purpose of an action is to compute the semantic value of the whole construct from the semantic values of its parts. For example, suppose we have a rule which says an expression can be the sum of two expressions. When the parser recognizes such a sum, each of the subexpressions has a semantic value which describes how it was built up. The action for this rule should create a similar sort of value for the newly recognized larger expression.

For example, here is a rule that says an expression can be the sum of two subexpressions:

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

The action says how to produce the semantic value of the sum expression from the values of the two subexpressions.

1.10 bison.guide/Bison Parser

Bison Output: the Parser File
==========

When you run Bison, you give it a Bison grammar file as input. The output is a C source file that parses the language described by the grammar. This file is called a "Bison parser". Keep in mind that the Bison utility and the Bison parser are two distinct programs: the Bison utility is a program whose output is the Bison parser that becomes part of your program.

The job of the Bison parser is to group tokens into groupings according to the grammar rules--for example, to build identifiers and operators into expressions. As it does this, it runs the actions for the grammar rules it uses.

The tokens come from a function called the "lexical analyzer" that you must supply in some fashion (such as by writing it in C). The Bison parser calls the lexical analyzer each time it wants a new token. It doesn't know what is "inside" the tokens (though their semantic values may reflect this). Typically the lexical analyzer makes the tokens by parsing characters of text, but Bison does not depend on this. See The Lexical Analyzer Function 'yylex'.

The Bison parser file is C code which defines a function named 'yyparse' which implements that grammar. This function does not make a complete C program: you must supply some additional functions. One is the lexical analyzer. Another is an error-reporting function which the parser calls to report an error. In addition, a complete C program must start with a function called 'main'; you have to provide this, and arrange for it to call 'yyparse' or the parser will never run. See Parser C-Language Interface.

Aside from the token type names and the symbols in the actions you write, all variable and function names used in the Bison parser file begin with 'yy' or 'YY'. This includes interface functions such as the lexical analyzer function 'yylex', the error reporting function 'yyerror' and the parser function 'yyparse' itself. This also includes numerous identifiers used for internal purposes. Therefore, you should avoid using C identifiers starting with 'yy' or 'YY' in the Bison grammar file except for the ones defined in this manual.

1.11 bison.guide/Stages

```
Stages in Using Bison
=====================

   The actual language-design process using Bison, from grammar
specification to a working compiler or interpreter, has these parts:

  1. Formally specify the grammar in a form recognized by Bison (see
     Bison Grammar Files).  For each grammatical rule in the language,
     describe the action that is to be taken when an instance of that
     rule is recognized.  The action is described by a sequence of C
     statements.

  2. Write a lexical analyzer to process input and pass tokens to the
     parser.  The lexical analyzer may be written by hand in C (see
     The Lexical Analyzer Function ‘yylex’).  It could also be produced
     using Lex, but the use of Lex is not discussed in this manual.

  3. Write a controlling function that calls the Bison-produced parser.

  4. Write error-reporting routines.

   To turn this source code as written into a runnable program, you
must follow these steps:

  1. Run Bison on the grammar to produce the parser.

  2. Compile the code output by Bison, as well as any other source
     files.

  3. Link the object files to produce the finished product.
```

1.12 bison.guide/Grammar Layout

```
The Overall Layout of a Bison Grammar
=====================================

   The input file for the Bison utility is a "Bison grammar file".  The
general form of a Bison grammar file is as follows:

     %{
     C DECLARATIONS
     %}

     BISON DECLARATIONS

     %%
     GRAMMAR RULES
     %%
     ADDITIONAL C CODE

The ‘%%’, ‘%{’ and ‘%}’ are punctuation that appears in every Bison
```

grammar file to separate the sections.

The C declarations may define types and variables used in the actions. You can also use preprocessor commands to define macros used there, and use '#include' to include header files that do any of these things.

The Bison declarations declare the names of the terminal and nonterminal symbols, and may also describe operator precedence and the data types of semantic values of various symbols.

The grammar rules define how to construct each nonterminal symbol from its parts.

The additional C code can contain any C code you want to use. Often the definition of the lexical analyzer 'yylex' goes here, plus subroutines called by the actions in the grammar rules. In a simple program, all the rest of the program can go here.

1.13 bison.guide/Examples

```
Examples
********
```

Now we show and explain three sample programs written using Bison: a reverse polish notation calculator, an algebraic (infix) notation calculator, and a multi-function calculator. All three have been tested under BSD Unix 4.3; each produces a usable, though limited, interactive desk-top calculator.

These examples are simple, but Bison grammars for real programming languages are written the same way. You can copy these examples out of the Info file and into a source file to try them.

```
  RPN Calc            Reverse polish notation calculator;
                        a first example with no operator precedence.
  Infix Calc          Infix (algebraic) notation calculator.
                        Operator precedence is introduced.
  Simple Error Recovery    Continuing after syntax errors.
  Multi-function Calc    Calculator with memory and trig functions.
                           It uses multiple data-types for semantic values.
  Exercises           Ideas for improving the multi-function calculator.
```

1.14 bison.guide/RPN Calc

```
Reverse Polish Notation Calculator
==================================
```

The first example is that of a simple double-precision "reverse

polish notation" calculator (a calculator using postfix operators). This example provides a good starting point, since operator precedence is not an issue. The second example will illustrate how operator precedence is handled.

The source code for this calculator is named 'rpcalc.y'. The '.y' extension is a convention used for Bison input files.

Decls	Bison and C declarations for rpcalc.
Rules	Grammar Rules for rpcalc, with explanation.
Lexer	The lexical analyzer.
Main	The controlling function.
Error	The error reporting function.
Gen	Running Bison on the grammar file.
Comp	Run the C compiler on the output code.

1.15 bison.guide/Rpcalc Decls

Declarations for 'rpcalc'

Here are the C and Bison declarations for the reverse polish notation calculator. As in C, comments are placed between '/*...*/'.

```
/* Reverse polish notation calculator. */

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

%token NUM

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

The C declarations section (see The C Declarations Section) contains two preprocessor directives.

The '#define' directive defines the macro 'YYSTYPE', thus specifying the C data type for semantic values of both tokens and groupings (see Data Types of Semantic Values). The Bison parser will use whatever type 'YYSTYPE' is defined as; if you don't define it, 'int' is the default. Because we specify 'double', each token and each expression has an associated value, which is a floating point number.

The '#include' directive is used to declare the exponentiation function 'pow'.

The second section, Bison declarations, provides information to Bison about the token types (see The Bison Declarations Section). Each terminal symbol that is not a single-character literal must be declared here. (Single-character literals normally don't need to be declared.) In this example, all the arithmetic operators are designated by

single-character literals, so the only terminal symbol that needs to be declared is ‘NUM’, the token type for numeric constants.

1.16 bison.guide/Rpcalc Rules

```
Grammar Rules for ‘rpcalc’
--------------------------

   Here are the grammar rules for the reverse polish notation
calculator.

     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;        }
     ;
     %%

   The groupings of the rpcalc "language" defined here are the
expression (given the name ‘exp’), the line of input (‘line’), and the
complete input transcript (‘input’).  Each of these nonterminal symbols
has several alternate rules, joined by the ‘|’ punctuator which is read
as "or".  The following sections explain what these rules mean.

   The semantics of the language is determined by the actions taken
when a grouping is recognized.  The actions are the C code that appears
inside braces.  See Actions.

   You must specify these actions in C, but Bison provides the means for
passing semantic values between the rules.  In each action, the
pseudo-variable ‘$$’ stands for the semantic value for the grouping
that the rule is going to construct.  Assigning a value to ‘$$’ is the
main job of most actions.  The semantic values of the components of the
rule are referred to as ‘$1’, ‘$2’, and so on.


  Rpcalc Input
  Rpcalc Line
  Rpcalc Expr
```

1.17 bison.guide/Rpcalc Input

```
Explanation of 'input'
......................

   Consider the definition of 'input':

     input:    /* empty */
             | input line
     ;

   This definition reads as follows: "A complete input is either an
empty string, or a complete input followed by an input line".  Notice
that "complete input" is defined in terms of itself.  This definition
is said to be "left recursive" since 'input' appears always as the
leftmost symbol in the sequence.  See Recursive Rules.

   The first alternative is empty because there are no symbols between
the colon and the first '|'; this means that 'input' can match an empty
string of input (no tokens).  We write the rules this way because it is
legitimate to type 'Ctrl-d' right after you start the calculator.  It's
conventional to put an empty alternative first and write the comment
'/* empty */' in it.

   The second alternate rule ('input line') handles all nontrivial
input.  It means, "After reading any number of lines, read one more
line if possible."  The left recursion makes this rule into a loop.
Since the first alternative matches empty input, the loop can be
executed zero or more times.

   The parser function 'yyparse' continues to process input until a
grammatical error is seen or the lexical analyzer says there are no more
input tokens; we will arrange for the latter to happen at end of file.
```

1.18 bison.guide/Rpcalc Line

```
Explanation of 'line'
.....................

   Now consider the definition of 'line':

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

   The first alternative is a token which is a newline character; this
means that rpcalc accepts a blank line (and ignores it, since there is
no action).  The second alternative is an expression followed by a
newline.  This is the alternative that makes rpcalc useful.  The
semantic value of the 'exp' grouping is the value of '$1' because the
'exp' in question is the first symbol in the alternative.  The action
prints this value, which is the result of the computation the user
asked for.
```

This action is unusual because it does not assign a value to '$$'. As a consequence, the semantic value associated with the 'line' is uninitialized (its value will be unpredictable). This would be a bug if that value were ever used, but we don't use it: once rpcalc has printed the value of the user's input line, that value is no longer needed.

1.19 bison.guide/Rpcalc Expr

Explanation of 'expr'
.....

The 'exp' grouping has several rules, one for each kind of expression. The first rule handles the simplest expressions: those that are just numbers. The second handles an addition-expression, which looks like two expressions followed by a plus-sign. The third handles subtraction, and so on.

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

We have used '|' to join all the rules for 'exp', but we could equally well have written them separately:

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

Most of the rules have actions that compute the value of the expression in terms of the value of its parts. For example, in the rule for addition, '$1' refers to the first component 'exp' and '$2' refers to the second one. The third component, ''+'', has no meaningful associated semantic value, but if it had one you could refer to it as '$3'. When 'yyparse' recognizes a sum expression using this rule, the sum of the two subexpressions' values is produced as the value of the entire expression. See Actions.

You don't have to give an action for every rule. When a rule has no action, Bison by default copies the value of '$1' into '$$'. This is what happens in the first rule (the one that uses 'NUM').

The formatting shown here is the recommended convention, but Bison does not require it. You can add or change whitespace as much as you wish. For example, this:

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

means the same thing as this:

```
exp:      NUM
```

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

The latter, however, is much more readable.

1.20 bison.guide/Rpcalc Lexer

```
The 'rpcalc' Lexical Analyzer
-----------------------------

   The lexical analyzer's job is low-level parsing: converting
characters or sequences of characters into tokens.  The Bison parser
gets its tokens by calling the lexical analyzer.  See
The Lexical Analyzer Function 'yylex'.

   Only a simple lexical analyzer is needed for the RPN calculator.
This lexical analyzer skips blanks and tabs, then reads in numbers as
'double' and returns them as 'NUM' tokens.  Any other character that
isn't part of a number is a separate token.  Note that the token-code
for such a single-character token is the character itself.

   The return value of the lexical analyzer function is a numeric code
which represents a token type.  The same text used in Bison rules to
stand for this token type is also a C expression for the numeric code
for the type.  This works in two ways.  If the token type is a
character literal, then its numeric code is the ASCII code for that
character; you can use the same character literal in the lexical
analyzer to express the number.  If the token type is an identifier,
that identifier is defined by Bison as a C macro whose definition is
the appropriate number.  In this example, therefore, 'NUM' becomes a
macro for 'yylex' to use.

   The semantic value of the token (if it has one) is stored into the
global variable 'yylval', which is where the Bison parser will look for
it.  (The C data type of 'yylval' is 'YYSTYPE', which was defined at
the beginning of the grammar; see Declarations for 'rpcalc'.)

   A token type code of zero is returned if the end-of-file is
encountered.  (Bison recognizes any nonpositive value as indicating the
end of the input.)

   Here is the code for the lexical analyzer:

     /* Lexical analyzer returns a double floating point
        number on the stack and the token NUM, or the ASCII
        character read if not a number.  Skips all blanks
        and tabs, returns 0 for EOF. */

     #include <ctype.h>

     yylex ()
     {
       int c;
```

```
  /* skip white space  */
  while ((c = getchar ()) == ' ' || c == '\t')
    ;
  /* process numbers   */
  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      scanf ("%lf", &yylval);
      return NUM;
    }
  /* return end-of-file  */
  if (c == EOF)
    return 0;
  /* return single chars */
  return c;
}
```

1.21 bison.guide/Rpcalc Main

```
The Controlling Function
------------------------
```

In keeping with the spirit of this example, the controlling function is kept to the bare minimum. The only requirement is that it call ‘yyparse’ to start the process of parsing.

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

1.22 bison.guide/Rpcalc Error

```
The Error Reporting Routine
---------------------------
```

When ‘yyparse’ detects a syntax error, it calls the error reporting function ‘yyerror’ to print an error message (usually but not always ‘"parse error"’). It is up to the programmer to supply ‘yyerror’ (see Parser C-Language Interface), so here is the definition we will use:

```
#include <stdio.h>

yyerror (s)  /* Called by yyparse on error */
     char *s;
{
  printf ("%s\n", s);
}
```

After ‘yyerror’ returns, the Bison parser may recover from the error

and continue parsing if the grammar contains a suitable error rule (see Error Recovery). Otherwise, ‘yyparse’ returns nonzero. We have not written any error rules in this example, so any invalid input will cause the calculator program to exit. This is not clean behavior for a real calculator, but it is adequate in the first example.

1.23 bison.guide/Rpcalc Gen

Running Bison to Make the Parser

Before running Bison to produce a parser, we need to decide how to arrange all the source code in one or more source files. For such a simple example, the easiest thing is to put everything in one file. The definitions of ‘yylex’, ‘yyerror’ and ‘main’ go at the end, in the "additional C code" section of the file (see The Overall Layout of a Bison Grammar).

For a large project, you would probably have several source files, and use ‘make’ to arrange to recompile them.

With all the source in a single file, you use the following command to convert it into a parser file:

```
bison FILE_NAME.y
```

In this example the file was called ‘rpcalc.y’ (for "Reverse Polish CALCulator"). Bison produces a file named ‘FILE_NAME.tab.c’, removing the ‘.y’ from the original file name. The file output by Bison contains the source code for ‘yyparse’. The additional functions in the input file (‘yylex’, ‘yyerror’ and ‘main’) are copied verbatim to the output.

1.24 bison.guide/Rpcalc Compile

Compiling the Parser File

Here is how to compile and run the parser file:

```
# List files in current directory.
% ls
rpcalc.tab.c  rpcalc.y

# Compile the Bison parser.
# '-lm' tells compiler to search math library for 'pow'.
% cc rpcalc.tab.c -lm -o rpcalc

# List files again.
% ls
rpcalc  rpcalc.tab.c  rpcalc.y
```

The file ‘rpcalc’ now contains the executable code. Here is an example session using ‘rpcalc’.

```
% rpcalc
4 9 +
13
3 7 + 3 4 5 *+-
-13
3 7 + 3 4 5 * + - n              Note the unary minus, ‘n’
13
5 6 / 4 n +
-3.166666667
3 4 ^                            Exponentiation
81
^D                               End-of-file indicator
%
```

1.25 bison.guide/Infix Calc

```
Infix Notation Calculator: ‘calc’
=================================
```

We now modify rpcalc to handle infix operators instead of postfix. Infix notation involves the concept of operator precedence and the need for parentheses nested to arbitrary depth. Here is the Bison code for ‘calc.y’, an infix desk-top calculator.

```
/* Infix notation calculator--calc */

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

/* BISON Declarations */
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG     /* negation--unary minus */
%right '^'    /* exponentiation        */

/* Grammar follows */
%%
input:    /* empty string */
        | 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;         }
     ;
     %%
```

The functions 'yylex', 'yyerror' and 'main' can be the same as before.

There are two important new features shown in this code.

In the second section (Bison declarations), '%left' declares token types and says they are left-associative operators. The declarations '%left' and '%right' (right associativity) take the place of '%token' which is used to declare a token type name without associativity. (These tokens are single-character literals, which ordinarily don't need to be declared. We declare them here to specify the associativity.)

Operator precedence is determined by the line ordering of the declarations; the higher the line number of the declaration (lower on the page or screen), the higher the precedence. Hence, exponentiation has the highest precedence, unary minus ('NEG') is next, followed by '*' and '/', and so on. See Operator Precedence.

The other important new feature is the '%prec' in the grammar section for the unary minus operator. The '%prec' simply instructs Bison that the rule '| '-' exp' has the same precedence as 'NEG'--in this case the next-to-highest. See Context-Dependent Precedence.

Here is a sample run of 'calc.y':

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

1.26 bison.guide/Simple Error Recovery

```
Simple Error Recovery
=====================
```

Up to this point, this manual has not addressed the issue of "error recovery"--how to continue parsing after the parser detects a syntax error. All we have handled is error reporting with 'yyerror'. Recall that by default 'yyparse' returns after calling 'yyerror'. This means that an erroneous input line causes the calculator program to exit. Now we show how to rectify this deficiency.

The Bison language itself includes the reserved word 'error', which may be included in the grammar rules. In the example below it has been added to one of the alternatives for 'line':

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

This addition to the grammar allows for simple error recovery in the event of a parse error. If an expression that cannot be evaluated is read, the error will be recognized by the third rule for 'line', and parsing will continue. (The 'yyerror' function is still called upon to print its message as well.) The action executes the statement 'yyerrok', a macro defined automatically by Bison; its meaning is that error recovery is complete (see Error Recovery). Note the difference between 'yyerrok' and 'yyerror'; neither one is a misprint.

This form of error recovery deals with syntax errors. There are other kinds of errors; for example, division by zero, which raises an exception signal that is normally fatal. A real calculator program must handle this signal and use 'longjmp' to return to 'main' and resume parsing input lines; it would also have to discard the rest of the current line of input. We won't discuss this issue further because it is not specific to Bison programs.

1.27 bison.guide/Multi-function Calc

```
Multi-Function Calculator: 'mfcalc'
===================================
```

Now that the basics of Bison have been discussed, it is time to move on to a more advanced problem. The above calculators provided only five functions, '+', '-', '*', '/' and '^'. It would be nice to have a calculator that provides other mathematical functions such as 'sin', 'cos', etc.

It is easy to add new operators to the infix calculator as long as they are only single-character literals. The lexical analyzer 'yylex' passes back all non-number characters as tokens, so new grammar rules suffice for adding a new operator. But we want something more flexible: built-in functions whose syntax has this form:

```
FUNCTION_NAME (ARGUMENT)
```

At the same time, we will add memory to the calculator, by allowing you to create named variables, store values in them, and use them later. Here is a sample session with the multi-function calculator:

```
% acalc
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
%
```

Note that multiple assignment and nested function calls are permitted.

Decl	Bison declarations for multi-function calculator.
Rules	Grammar rules for the calculator.
Symtab	Symbol table management subroutines.

1.28 bison.guide/Mfcalc Decl

Declarations for ‘mfcalc’

Here are the C and Bison declarations for the multi-function calculator.

```
%{
#include <math.h>  /* For math functions, cos(), sin(), etc. */
#include "calc.h"  /* Contains definition of `symrec'        */
%}
%union {
double     val;  /* For returning numbers.                   */
symrec  *tptr;   /* For returning symbol-table pointers      */
}

%token <val>  NUM        /* Simple double precision number   */
%token <tptr> VAR FNCT   /* Variable and Function            */
%type  <val>  exp

%right '='
%left '-' '+'
%left '*' '/'
%left NEG     /* Negation--unary minus */
%right '^'    /* Exponentiation        */

/* Grammar follows */

%%
```

The above grammar introduces only two new features of the Bison language. These features allow semantic values to have various data types (see More Than One Value Type).

The ‘%union’ declaration specifies the entire list of possible types;

this is instead of defining ‘YYSTYPE’. The allowable types are now double-floats (for ‘exp’ and ‘NUM’) and pointers to entries in the symbol table. See The Collection of Value Types.

Since values can now have various types, it is necessary to associate a type with each grammar symbol whose semantic value is used. These symbols are ‘NUM’, ‘VAR’, ‘FNCT’, and ‘exp’. Their declarations are augmented with information about their data type (placed between angle brackets).

The Bison construct ‘%type’ is used for declaring nonterminal symbols, just as ‘%token’ is used for declaring token types. We have not used ‘%type’ before because nonterminal symbols are normally declared implicitly by the rules that define them. But ‘exp’ must be declared explicitly so we can specify its value type. See Nonterminal Symbols.

1.29 bison.guide/Mfcalc Rules

Grammar Rules for ‘mfcalc’

Here are the grammar rules for the multi-function calculator. Most of them are copied directly from ‘calc’; three rules, those which mention ‘VAR’ or ‘FNCT’, are new.

```
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 */
%%
```

1.30 bison.guide/Mfcalc Symtab

```
The 'mfcalc' Symbol Table
-------------------------

   The multi-function calculator requires a symbol table to keep track
of the names and meanings of variables and functions.  This doesn't
affect the grammar rules (except for the actions) or the Bison
declarations, but it requires some additional C functions for support.

   The symbol table itself consists of a linked list of records.  Its
definition, which is kept in the header 'calc.h', is as follows.  It
provides for either functions or variables to be placed in the table.

     /* Data type for links in the chain of symbols.      */
     struct symrec
     {
       char *name;  /* name of symbol                     */
       int type;    /* type of symbol: either VAR or FNCT */
       union {
         double var;           /* value of a VAR          */
         double (*fnctptr)();  /* value of a FNCT         */
       } value;
       struct symrec *next;    /* link field              */
     };

     typedef struct symrec symrec;

     /* The symbol table: a chain of 'struct symrec'.     */
     extern symrec *sym_table;

     symrec *putsym ();
     symrec *getsym ();

   The new version of 'main' includes a call to 'init_table', a
function that initializes the symbol table.  Here it is, and
'init_table' as well:

     #include <stdio.h>

     main ()
     {
       init_table ();
       yyparse ();
     }

     yyerror (s)  /* Called by yyparse on error */
          char *s;
     {
       printf ("%s\n", s);
     }

     struct init
     {
       char *fname;
       double (*fnct)();
```

```
};

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

/* The symbol table: a chain of `struct symrec'.  */
symrec *sym_table = (symrec *)0;

init_table ()  /* puts arithmetic functions in table. */
{
  int i;
  symrec *ptr;
  for (i = 0; arith_fncts[i].fname != 0; i++)
    {
      ptr = putsym (arith_fncts[i].fname, FNCT);
      ptr->value.fnctptr = arith_fncts[i].fnct;
    }
}
```

By simply editing the initialization list and adding the necessary include files, you can add additional functions to the calculator.

Two important functions allow look-up and installation of symbols in the symbol table. The function 'putsym' is passed a name and the type ('VAR' or 'FNCT') of the object to be installed. The object is linked to the front of the list, and a pointer to the object is returned. The function 'getsym' is passed the name of the symbol to look up. If found, a pointer to that symbol is returned; otherwise zero is returned.

```
symrec *
putsym (sym_name,sym_type)
     char *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.  */
  ptr->next = (struct symrec *)sym_table;
  sym_table = ptr;
  return ptr;
}

symrec *
getsym (sym_name)
     char *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;
}
```

The function 'yylex' must now recognize variables, numeric values, and the single-character arithmetic operators. Strings of alphanumeric characters with a leading nondigit are recognized as either variables or functions depending on what the symbol table says about them.

The string is passed to 'getsym' for look up in the symbol table. If the name appears in the table, a pointer to its location and its type ('VAR' or 'FNCT') is returned to 'yyparse'. If it is not already in the table, then it is installed as a 'VAR' using 'putsym'. Again, a pointer and its type (which must be 'VAR') is returned to 'yyparse'.

No change is needed in the handling of numeric values and arithmetic operators in 'yylex'.

```
#include <ctype.h>
yylex ()
{
  int c;

  /* Ignore whitespace, get first nonwhite character.  */
  while ((c = getchar ()) == ' ' || c == '\t');

  if (c == EOF)
    return 0;

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

/* Char starts an identifier => read the name.       */
  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.  */
      if (length == 0)
        length = 40, symbuf = (char *)malloc (length + 1);

      i = 0;
      do
```

```
{
          /* If buffer is full, make it bigger.        */
          if (i == length)
            {
              length *= 2;
              symbuf = (char *)realloc (symbuf, length + 1);
            }
          /* Add this character to the buffer.         */
          symbuf[i++] = c;
          /* Get another character.                    */
          c = getchar ();
        }

while (c != EOF && 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.        */
  return c;
}
```

This program is both powerful and flexible. You may easily add new functions, and it is a simple job to modify this code to install predefined variables such as 'pi' or 'e' as well.

1.31 bison.guide/Exercises

```
Exercises
=========

  1. Add some new functions from 'math.h' to the initialization list.

  2. Add another array that contains constants and their values.  Then
     modify 'init_table' to add these constants to the symbol table.
     It will be easiest to give the constants type 'VAR'.

  3. Make the program report an error if the user refers to an
     uninitialized variable in any way except to store a value in it.
```

1.32 bison.guide/Grammar File

```
Bison Grammar Files
*******************

   Bison takes as input a context-free grammar specification and
produces a C-language function that recognizes correct instances of the
grammar.

   The Bison grammar input file conventionally has a name ending in
`.y'.


  Grammar Outline     Overall layout of the grammar file.
  Symbols             Terminal and nonterminal symbols.
  Rules               How to write grammar rules.
  Recursion           Writing recursive rules.
  Semantics           Semantic values and actions.
  Declarations        All kinds of Bison declarations are described here.
  Multiple Parsers    Putting more than one Bison parser in one program.
```

1.33 bison.guide/Grammar Outline

```
Outline of a Bison Grammar
==========================

   A Bison grammar file has four main sections, shown here with the
appropriate delimiters:

     %{
     C DECLARATIONS
     %}

     BISON DECLARATIONS

     %%
     GRAMMAR RULES
     %%

     ADDITIONAL C CODE

   Comments enclosed in `/* ... */' may appear in any of the sections.


  C Declarations      Syntax and usage of the C declarations section.
  Bison Declarations    Syntax and usage of the Bison declarations section.
  Grammar Rules       Syntax and usage of the grammar rules section.
  C Code              Syntax and usage of the additional C code section.
```

1.34 bison.guide/C Declarations

The C Declarations Section

The C DECLARATIONS section contains macro definitions and declarations of functions and variables that are used in the actions in the grammar rules. These are copied to the beginning of the parser file so that they precede the definition of ‘yyparse’. You can use ‘#include’ to get the declarations from a header file. If you don’t need any C declarations, you may omit the ‘%{’ and ‘%}’ delimiters that bracket this section.

1.35 bison.guide/Bison Declarations

The Bison Declarations Section

The BISON DECLARATIONS section contains declarations that define terminal and nonterminal symbols, specify precedence, and so on. In some simple grammars you may not need any declarations. See Bison Declarations.

1.36 bison.guide/Grammar Rules

The Grammar Rules Section

The "grammar rules" section contains one or more Bison grammar rules, and nothing else. See Syntax of Grammar Rules.

There must always be at least one grammar rule, and the first ‘%%’ (which precedes the grammar rules) may never be omitted even if it is the first thing in the file.

1.37 bison.guide/C Code

The Additional C Code Section

The ADDITIONAL C CODE section is copied verbatim to the end of the parser file, just as the C DECLARATIONS section is copied to the beginning. This is the most convenient place to put anything that you want to have in the parser file but which need not come before the definition of ‘yyparse’. For example, the definitions of ‘yylex’ and ‘yyerror’ often go here. See Parser C-Language Interface.

If the last section is empty, you may omit the ‘%%’ that separates it

from the grammar rules.

The Bison parser itself contains many static variables whose names start with ‘yy’ and many macros whose names start with ‘YY’. It is a good idea to avoid using any such names (except those documented in this manual) in the additional C code section of the grammar file.

1.38 bison.guide/Symbols

Symbols, Terminal and Nonterminal
==========

"Symbols" in Bison grammars represent the grammatical classifications of the language.

A "terminal symbol" (also known as a "token type") represents a class of syntactically equivalent tokens. You use the symbol in grammar rules to mean that a token in that class is allowed. The symbol is represented in the Bison parser by a numeric code, and the ‘yylex’ function returns a token type code to indicate what kind of token has been read. You don’t need to know what the code value is; you can use the symbol to stand for it.

A "nonterminal symbol" stands for a class of syntactically equivalent groupings. The symbol name is used in writing grammar rules. By convention, it should be all lower case.

Symbol names can contain letters, digits (not at the beginning), underscores and periods. Periods make sense only in nonterminals.

There are two ways of writing terminal symbols in the grammar:

* A "named token type" is written with an identifier, like an identifier in C. By convention, it should be all upper case. Each such name must be defined with a Bison declaration such as ‘%token’. See Token Type Names.

* A "character token type" (or "literal token") is written in the grammar using the same syntax used in C for character constants; for example, ‘’+’’ is a character token type. A character token type doesn’t need to be declared unless you need to specify its semantic value data type (see Data Types of Semantic Values), associativity, or precedence (see Operator Precedence).

  By convention, a character token type is used only to represent a token that consists of that particular character. Thus, the token type ‘’+’’ is used to represent the character ‘+’ as a token. Nothing enforces this convention, but if you depart from it, your program will confuse other readers.

  All the usual escape sequences used in character literals in C can be used in Bison as well, but you must not use the null character as a character literal because its ASCII code, zero, is the code ‘yylex’ returns for end-of-input (see

Calling Convention for ‘yylex’).

How you choose to write a terminal symbol has no effect on its grammatical meaning. That depends only on where it appears in rules and on when the parser function returns that symbol.

The value returned by ‘yylex’ is always one of the terminal symbols (or 0 for end-of-input). Whichever way you write the token type in the grammar rules, you write it the same way in the definition of ‘yylex’. The numeric code for a character token type is simply the ASCII code for the character, so ‘yylex’ can use the identical character constant to generate the requisite code. Each named token type becomes a C macro in the parser file, so ‘yylex’ can use the name to stand for the code. (This is why periods don’t make sense in terminal symbols.) See Calling Convention for ‘yylex’.

If ‘yylex’ is defined in a separate file, you need to arrange for the token-type macro definitions to be available there. Use the ‘-d’ option when you run Bison, so that it will write these macro definitions into a separate header file ‘NAME.tab.h’ which you can include in the other source files that need it. See Invoking Bison.

The symbol ‘error’ is a terminal symbol reserved for error recovery (see Error Recovery); you shouldn’t use it for any other purpose. In particular, ‘yylex’ should never return this value.

1.39 bison.guide/Rules

Syntax of Grammar Rules
==========

A Bison grammar rule has the following general form:

```
RESULT: COMPONENTS...
        ;
```

where RESULT is the nonterminal symbol that this rule describes and COMPONENTS are various terminal and nonterminal symbols that are put together by this rule (see Symbols).

For example,

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

says that two groupings of type ‘exp’, with a ‘+’ token in between, can be combined into a larger grouping of type ‘exp’.

Whitespace in rules is significant only to separate symbols. You can add extra whitespace as you wish.

Scattered among the components can be ACTIONS that determine the semantics of the rule. An action looks like this:

```
     {C STATEMENTS}
```

Usually there is only one action and it follows the components. See Actions.

Multiple rules for the same RESULT can be written separately or can be joined with the vertical-bar character '|' as follows:

```
     RESULT:   RULE1-COMPONENTS...
             | RULE2-COMPONENTS...
             ...
             ;
```

They are still considered distinct rules even when joined in this way.

If COMPONENTS in a rule is empty, it means that RESULT can match the empty string. For example, here is how to define a comma-separated sequence of zero or more 'exp' groupings:

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

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

It is customary to write a comment '/* empty */' in each rule with no components.

1.40 bison.guide/Recursion

Recursive Rules
==========

A rule is called "recursive" when its RESULT nonterminal appears also on its right hand side. Nearly all Bison grammars need to use recursion, because that is the only way to define a sequence of any number of somethings. Consider this recursive definition of a comma-separated sequence of one or more expressions:

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

Since the recursive use of 'expseq1' is the leftmost symbol in the right hand side, we call this "left recursion". By contrast, here the same construct is defined using "right recursion":

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

Any kind of sequence can be defined using either left recursion or

right recursion, but you should always use left recursion, because it can parse a sequence of any number of elements with bounded stack space. Right recursion uses up space on the Bison stack in proportion to the number of elements in the sequence, because all the elements must be shifted onto the stack before the rule can be applied even once. See The Bison Parser Algorithm, for further explanation of this.

"Indirect" or "mutual" recursion occurs when the result of the rule does not appear directly on its right hand side, but does appear in rules for other nonterminals which do appear on its right hand side.

For example:

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

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

defines two mutually-recursive nonterminals, since each refers to the other.

1.41 bison.guide/Semantics

```
Defining Language Semantics
===========================
```

The grammar rules for a language determine only the syntax. The semantics are determined by the semantic values associated with various tokens and groupings, and by the actions taken when various groupings are recognized.

For example, the calculator calculates properly because the value associated with each expression is the proper number; it adds properly because the action for the grouping 'X + Y' is to add the numbers associated with X and Y.

```
  Value Type          Specifying one data type for all semantic values.
  Multiple Types      Specifying several alternative data types.
  Actions             An action is the semantic definition of a grammar rule.
  Action Types        Specifying data types for actions to operate on.
  Mid-Rule Actions    Most actions go at the end of a rule.
                      This says when, why and how to use the exceptional
                        action in the middle of a rule.
```

1.42 bison.guide/Value Type

Data Types of Semantic Values

In a simple program it may be sufficient to use the same data type for the semantic values of all language constructs. This was true in the RPN and infix calculator examples (see Reverse Polish Notation Calculator).

Bison's default is to use type 'int' for all semantic values. To specify some other type, define 'YYSTYPE' as a macro, like this:

```
#define YYSTYPE double
```

This macro definition must go in the C declarations section of the grammar file (see Outline of a Bison Grammar).

1.43 bison.guide/Multiple Types

More Than One Value Type

In most programs, you will need different data types for different kinds of tokens and groupings. For example, a numeric constant may need type 'int' or 'long', while a string constant needs type 'char *', and an identifier might need a pointer to an entry in the symbol table.

To use more than one data type for semantic values in one parser, Bison requires you to do two things:

* Specify the entire collection of possible data types, with the '%union' Bison declaration (see The Collection of Value Types).

* Choose one of those types for each symbol (terminal or nonterminal) for which semantic values are used. This is done for tokens with the '%token' Bison declaration (see Token Type Names) and for groupings with the '%type' Bison declaration (see Nonterminal Symbols).

1.44 bison.guide/Actions

Actions

An action accompanies a syntactic rule and contains C code to be executed each time an instance of that rule is recognized. The task of most actions is to compute a semantic value for the grouping built by the rule from the semantic values associated with tokens or smaller groupings.

An action consists of C statements surrounded by braces, much like a compound statement in C. It can be placed at any position in the rule; it is executed at that position. Most rules have just one action at the end of the rule, following all the components. Actions in the middle of a rule are tricky and used only for special purposes (see Actions in Mid-Rule).

The C code in an action can refer to the semantic values of the components matched by the rule with the construct '$N', which stands for the value of the Nth component. The semantic value for the grouping being constructed is '$$'. (Bison translates both of these constructs into array element references when it copies the actions into the parser file.)

Here is a typical example:

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

This rule constructs an 'exp' from two smaller 'exp' groupings connected by a plus-sign token. In the action, '$1' and '$3' refer to the semantic values of the two component 'exp' groupings, which are the first and third symbols on the right hand side of the rule. The sum is stored into '$$' so that it becomes the semantic value of the addition-expression just recognized by the rule. If there were a useful semantic value associated with the '+' token, it could be referred to as '$2'.

If you don't specify an action for a rule, Bison supplies a default: '$$ = $1'. Thus, the value of the first symbol in the rule becomes the value of the whole rule. Of course, the default rule is valid only if the two data types match. There is no meaningful default action for an empty rule; every empty rule must have an explicit action unless the rule's value does not matter.

'$N' with N zero or negative is allowed for reference to tokens and groupings on the stack *before* those that match the current rule. This is a very risky practice, and to use it reliably you must be certain of the context in which the rule is applied. Here is a case in which you can use this reliably:

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

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

As long as 'bar' is used only in the fashion shown here, '$0' always refers to the 'expr' which precedes 'bar' in the definition of 'foo'.

1.45 bison.guide/Action Types

```
Data Types of Values in Actions
-------------------------------

   If you have chosen a single data type for semantic values, the '$$'
and '$N' constructs always have that data type.

   If you have used '%union' to specify a variety of data types, then
you must declare a choice among these types for each terminal or
nonterminal symbol that can have a semantic value.  Then each time you
use '$$' or '$N', its data type is determined by which symbol it refers
to in the rule.  In this example,

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

'$1' and '$3' refer to instances of 'exp', so they all have the data
type declared for the nonterminal symbol 'exp'.  If '$2' were used, it
would have the data type declared for the terminal symbol ''+'',
whatever that might be.

   Alternatively, you can specify the data type when you refer to the
value, by inserting '<TYPE>' after the '$' at the beginning of the
reference.  For example, if you have defined types as shown here:

     %union {
       int itype;
       double dtype;
     }

then you can write '$<itype>1' to refer to the first subunit of the
rule as an integer, or '$<dtype>1' to refer to it as a double.
```

1.46 bison.guide/Mid-Rule Actions

```
Actions in Mid-Rule
-------------------

   Occasionally it is useful to put an action in the middle of a rule.
These actions are written just like usual end-of-rule actions, but they
are executed before the parser even recognizes the following components.

   A mid-rule action may refer to the components preceding it using
'$N', but it may not refer to subsequent components because it is run
before they are parsed.

   The mid-rule action itself counts as one of the components of the
rule.  This makes a difference when there is another action later in
the same rule (and usually there is another at the end): you have to
count the actions along with the symbols when working out which number
N to use in '$N'.
```

The mid-rule action can also have a semantic value. The action can set its value with an assignment to '$$', and actions later in the rule can refer to the value using '$N'. Since there is no symbol to name the action, there is no way to declare a data type for the value in advance, so you must use the '$<...>' construct to specify a data type each time you refer to this value.

There is no way to set the value of the entire rule with a mid-rule action, because assignments to '$$' do not have that effect. The only way to set the value for the entire rule is with an ordinary action at the end of the rule.

Here is an example from a hypothetical compiler, handling a 'let' statement that looks like 'let (VARIABLE) STATEMENT' and serves to create a variable named VARIABLE temporarily for the duration of STATEMENT. To parse this construct, we must put VARIABLE into the symbol table while STATEMENT is parsed, then remove it afterward. Here is how it is done:

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

As soon as 'let (VARIABLE)' has been recognized, the first action is run. It saves a copy of the current semantic context (the list of accessible variables) as its semantic value, using alternative 'context' in the data-type union. Then it calls 'declare_variable' to add the new variable to that list. Once the first action is finished, the embedded statement 'stmt' can be parsed. Note that the mid-rule action is component number 5, so the 'stmt' is component number 6.

After the embedded statement is parsed, its semantic value becomes the value of the entire 'let'-statement. Then the semantic value from the earlier action is used to restore the prior list of variables. This removes the temporary 'let'-variable from the list so that it won't appear to exist while the rest of the program is parsed.

Taking action before a rule is completely recognized often leads to conflicts since the parser must commit to a parse in order to execute the action. For example, the following two rules, without mid-rule actions, can coexist in a working parser because the parser can shift the open-brace token and look at what follows before deciding whether there is a declaration or not:

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

But when we add a mid-rule action as follows, the rules become nonfunctional:

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

```
            ;
```

Now the parser is forced to decide whether to run the mid-rule action when it has read no farther than the open-brace. In other words, it must commit to using one rule or the other, without sufficient information to do it correctly. (The open-brace token is what is called the "look-ahead" token at this time, since the parser is still deciding what to do about it. See Look-Ahead Tokens.)

You might think that you could correct the problem by putting identical actions into the two rules, like this:

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

But this does not help, because Bison does not realize that the two actions are identical. (Bison never tries to understand the C code in an action.)

If the grammar is such that a declaration can be distinguished from a statement by the first token (which is true in C), then one solution which does work is to put the action after the open-brace, like this:

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

Now the first token of the following declaration or statement, which would in any case tell Bison which rule to use, can still do so.

Another solution is to bury the action inside a nonterminal symbol which serves as a subroutine:

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

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

Now Bison can execute the action in the rule for ‘subroutine’ without deciding which rule for ‘compound’ it will eventually use. Note that the action is now at the end of its rule. Any mid-rule action can be converted to an end-of-rule action in this way, and this is what Bison actually does to implement mid-rule actions.

1.47 bison.guide/Declarations

```
Bison Declarations
==================

   The "Bison declarations" section of a Bison grammar defines the
symbols used in formulating the grammar and the data types of semantic
values.  See Symbols.

   All token type names (but not single-character literal tokens such as
`'+'' and `'*'') must be declared.  Nonterminal symbols must be
declared if you need to specify which data type to use for the semantic
value (see More Than One Value Type).

   The first rule in the file also specifies the start symbol, by
default.  If you want some other symbol to be the start symbol, you
must declare it explicitly (see Languages and Context-Free Grammars).


  Token Decl          Declaring terminal symbols.
  Precedence Decl     Declaring terminals with precedence and associativity.
  Union Decl          Declaring the set of all semantic value types.
  Type Decl           Declaring the choice of type for a nonterminal symbol.
  Expect Decl         Suppressing warnings about shift/reduce conflicts.
  Start Decl          Specifying the start symbol.
  Pure Decl           Requesting a reentrant parser.
  Decl Summary        Table of all Bison declarations.
```

1.48 bison.guide/Token Decl

```
Token Type Names
----------------

   The basic way to declare a token type name (terminal symbol) is as
follows:

     %token NAME

   Bison will convert this into a `#define' directive in the parser, so
that the function `yylex' (if it is in this file) can use the name NAME
to stand for this token type's code.

   Alternatively, you can use `%left', `%right', or `%nonassoc' instead
of `%token', if you wish to specify precedence.  See
Operator Precedence.

   You can explicitly specify the numeric code for a token type by
appending an integer value in the field immediately following the token
name:

     %token NUM 300

It is generally best, however, to let Bison choose the numeric codes for
```

all token types. Bison will automatically select codes that don't conflict with each other or with ASCII characters.

In the event that the stack type is a union, you must augment the '%token' or other token declaration to include the data type alternative delimited by angle-brackets (see More Than One Value Type).

For example:

```
%union {              /* define stack type */
  double val;
  symrec *tptr;
}
%token <val> NUM      /* define token NUM and its type */
```

1.49 bison.guide/Precedence Decl

Operator Precedence

Use the '%left', '%right' or '%nonassoc' declaration to declare a token and specify its precedence and associativity, all at once. These are called "precedence declarations". See Operator Precedence, for general information on operator precedence.

The syntax of a precedence declaration is the same as that of '%token': either

```
%left SYMBOLS...
```

or

```
%left <TYPE> SYMBOLS...
```

And indeed any of these declarations serves the purposes of '%token'. But in addition, they specify the associativity and relative precedence for all the SYMBOLS:

* The associativity of an operator OP determines how repeated uses of the operator nest: whether 'X OP Y OP Z' is parsed by grouping X with Y first or by grouping Y with Z first. '%left' specifies left-associativity (grouping X with Y first) and '%right' specifies right-associativity (grouping Y with Z first). '%nonassoc' specifies no associativity, which means that 'X OP Y OP Z' is considered a syntax error.

* The precedence of an operator determines how it nests with other operators. All the tokens declared in a single precedence declaration have equal precedence and nest together according to their associativity. When two tokens declared in different precedence declarations associate, the one declared later has the higher precedence and is grouped first.

1.50 bison.guide/Union Decl

```
The Collection of Value Types
-----------------------------

   The '%union' declaration specifies the entire collection of possible
data types for semantic values.  The keyword '%union' is followed by a
pair of braces containing the same thing that goes inside a 'union' in
C.

   For example:

     %union {
       double val;
       symrec *tptr;
     }

This says that the two alternative types are 'double' and 'symrec *'.
They are given names 'val' and 'tptr'; these names are used in the
'%token' and '%type' declarations to pick one of the types for a
terminal or nonterminal symbol (see Nonterminal Symbols).

   Note that, unlike making a 'union' declaration in C, you do not write
a semicolon after the closing brace.
```

1.51 bison.guide/Type Decl

```
Nonterminal Symbols
-------------------

When you use '%union' to specify multiple value types, you must declare
the value type of each nonterminal symbol for which values are used.
This is done with a '%type' declaration, like this:

     %type <TYPE> NONTERMINAL...

Here NONTERMINAL is the name of a nonterminal symbol, and TYPE is the
name given in the '%union' to the alternative that you want (see
The Collection of Value Types).  You can give any number of nonterminal
symbols in the same '%type' declaration, if they have the same value
type.  Use spaces to separate the symbol names.
```

1.52 bison.guide/Expect Decl

```
Suppressing Conflict Warnings
-----------------------------

   Bison normally warns if there are any conflicts in the grammar (see
Shift/Reduce Conflicts), but most real grammars have harmless
shift/reduce conflicts which are resolved in a predictable way and
```

would be difficult to eliminate. It is desirable to suppress the warning about these conflicts unless the number of conflicts changes. You can do this with the '%expect' declaration.

The declaration looks like this:

```
%expect N
```

Here N is a decimal integer. The declaration says there should be no warning if there are N shift/reduce conflicts and no reduce/reduce conflicts. The usual warning is given if there are either more or fewer conflicts, or if there are any reduce/reduce conflicts.

In general, using '%expect' involves these steps:

* Compile your grammar without '%expect'. Use the '-v' option to get a verbose list of where the conflicts occur. Bison will also print the number of conflicts.

* Check each of the conflicts to make sure that Bison's default resolution is what you really want. If not, rewrite the grammar and go back to the beginning.

* Add an '%expect' declaration, copying the number N from the number which Bison printed.

Now Bison will stop annoying you about the conflicts you have checked, but it will warn you again if changes in the grammar result in additional conflicts.

1.53 bison.guide/Start Decl

The Start-Symbol

Bison assumes by default that the start symbol for the grammar is the first nonterminal specified in the grammar specification section. The programmer may override this restriction with the '%start' declaration as follows:

```
%start SYMBOL
```

1.54 bison.guide/Pure Decl

A Pure (Reentrant) Parser

A "reentrant" program is one which does not alter in the course of execution; in other words, it consists entirely of "pure" (read-only) code. Reentrancy is important whenever asynchronous execution is

possible; for example, a nonreentrant program may not be safe to call from a signal handler. In systems with multiple threads of control, a nonreentrant program must be called only within interlocks.

The Bison parser is not normally a reentrant program, because it uses statically allocated variables for communication with 'yylex'. These variables include 'yylval' and 'yylloc'.

The Bison declaration '%pure_parser' says that you want the parser to be reentrant. It looks like this:

```
%pure_parser
```

The effect is that the two communication variables become local variables in 'yyparse', and a different calling convention is used for the lexical analyzer function 'yylex'. See Calling for Pure Parsers, for the details of this. The variable 'yynerrs' also becomes local in 'yyparse' (see The Error Reporting Function 'yyerror'). The convention for calling 'yyparse' itself is unchanged.

1.55 bison.guide/Decl Summary

Bison Declaration Summary

Here is a summary of all Bison declarations:

'%union'
: Declare the collection of data types that semantic values may have (see The Collection of Value Types).

'%token'
: Declare a terminal symbol (token type name) with no precedence or associativity specified (see Token Type Names).

'%right'
: Declare a terminal symbol (token type name) that is right-associative (see Operator Precedence).

'%left'
: Declare a terminal symbol (token type name) that is left-associative (see Operator Precedence).

'%nonassoc'
: Declare a terminal symbol (token type name) that is nonassociative (using it in a way that would be associative is a syntax error) (see Operator Precedence).

'%type'
: Declare the type of semantic values for a nonterminal symbol (see Nonterminal Symbols).

'%start'
: Specify the grammar's start symbol (see The Start-Symbol).

```
'%expect'
     Declare the expected number of shift-reduce conflicts (see
     Suppressing Conflict Warnings).

'%pure_parser'
     Request a pure (reentrant) parser program (see
     A Pure (Reentrant) Parser).
```

1.56 bison.guide/Multiple Parsers

```
Multiple Parsers in the Same Program
====================================

   Most programs that use Bison parse only one language and therefore
contain only one Bison parser.  But what if you want to parse more than
one language with the same program?  Then you need to avoid a name
conflict between different definitions of 'yyparse', 'yylval', and so
on.

   The easy way to do this is to use the option '-p PREFIX' (see
Invoking Bison).  This renames the interface functions and variables of
the Bison parser to start with PREFIX instead of 'yy'.  You can use
this to give each parser distinct names that do not conflict.

   The precise list of symbols renamed is 'yyparse', 'yylex',
'yyerror', 'yylval', 'yychar' and 'yydebug'.  For example, if you use
'-p c', the names become 'cparse', 'clex', and so on.

   *All the other variables and macros associated with Bison are not
renamed.* These others are not global; there is no conflict if the same
name is used in different parsers.  For example, 'YYSTYPE' is not
renamed, but defining this in different ways in different parsers causes
no trouble (see Data Types of Semantic Values).

   The '-p' option works by adding macro definitions to the beginning
of the parser source file, defining 'yyparse' as 'PREFIXparse', and so
on.  This effectively substitutes one name for the other in the entire
parser file.
```

1.57 bison.guide/Interface

```
Parser C-Language Interface
***************************

   The Bison parser is actually a C function named 'yyparse'.  Here we
describe the interface conventions of 'yyparse' and the other functions
that it needs to use.

   Keep in mind that the parser uses many C identifiers starting with
```

'yy' and 'YY' for internal purposes. If you use such an identifier (aside from those in this manual) in an action or in additional C code in the grammar file, you are likely to run into trouble.

```
  Parser Function     How to call 'yyparse' and what it returns.
  Lexical             You must supply a function 'yylex'
                        which reads tokens.
  Error Reporting     You must supply a function 'yyerror'.
  Action Features     Special features for use in actions.
```

1.58 bison.guide/Parser Function

```
The Parser Function 'yyparse'
=============================
```

You call the function 'yyparse' to cause parsing to occur. This function reads tokens, executes actions, and ultimately returns when it encounters end-of-input or an unrecoverable syntax error. You can also write an action which directs 'yyparse' to return immediately without reading further.

The value returned by 'yyparse' is 0 if parsing was successful (return is due to end-of-input).

The value is 1 if parsing failed (return is due to a syntax error).

In an action, you can cause immediate return from 'yyparse' by using these macros:

'YYACCEPT'
: Return immediately with value 0 (to report success).

'YYABORT'
: Return immediately with value 1 (to report failure).

1.59 bison.guide/Lexical

```
The Lexical Analyzer Function 'yylex'
=====================================
```

The "lexical analyzer" function, 'yylex', recognizes tokens from the input stream and returns them to the parser. Bison does not create this function automatically; you must write it so that 'yyparse' can call it. The function is sometimes referred to as a lexical scanner.

In simple programs, 'yylex' is often defined at the end of the Bison grammar file. If 'yylex' is defined in a separate source file, you need to arrange for the token-type macro definitions to be available there. To do this, use the '-d' option when you run Bison, so that it

will write these macro definitions into a separate header file 'NAME.tab.h' which you can include in the other source files that need it. See Invoking Bison.

```
  Calling Convention    How 'yyparse' calls 'yylex'.
  Token Values        How 'yylex' must return the semantic value
                        of the token it has read.
  Token Positions     How 'yylex' must return the text position
                        (line number, etc.) of the token, if the
                        actions want that.
  Pure Calling        How the calling convention differs
                        in a pure parser (see A Pure (Reentrant) Parser).
```

1.60 bison.guide/Calling Convention

Calling Convention for 'yylex'

The value that 'yylex' returns must be the numeric code for the type of token it has just found, or 0 for end-of-input.

When a token is referred to in the grammar rules by a name, that name in the parser file becomes a C macro whose definition is the proper numeric code for that token type. So 'yylex' can use the name to indicate that type. See Symbols.

When a token is referred to in the grammar rules by a character literal, the numeric code for that character is also the code for the token type. So 'yylex' can simply return that character code. The null character must not be used this way, because its code is zero and that is what signifies end-of-input.

Here is an example showing these things:

```
yylex ()
{
  ...
  if (c == EOF)     /* Detect end of file. */
    return 0;
  ...
  if (c == '+' || c == '-')
    return c;      /* Assume token type for '+' is '+'. */
  ...
  return INT;      /* Return the type of the token. */
  ...
}
```

This interface has been designed so that the output from the 'lex' utility can be used without change as the definition of 'yylex'.

1.61 bison.guide/Token Values

```
Semantic Values of Tokens
-------------------------

   In an ordinary (nonreentrant) parser, the semantic value of the
token must be stored into the global variable 'yylval'.  When you are
using just one data type for semantic values, 'yylval' has that type.
Thus, if the type is 'int' (the default), you might write this in
'yylex':

       ...
       yylval = value;  /* Put value onto Bison stack. */
       return INT;      /* Return the type of the token. */
       ...

   When you are using multiple data types, 'yylval''s type is a union
made from the '%union' declaration (see The Collection of Value Types).
So when you store a token's value, you must use the proper member of
the union.  If the '%union' declaration looks like this:

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

then the code in 'yylex' might look like this:

       ...
       yylval.intval = value; /* Put value onto Bison stack. */
       return INT;          /* Return the type of the token. */
       ...
```

1.62 bison.guide/Token Positions

```
Textual Positions of Tokens
---------------------------

   If you are using the '@N'-feature (see
Special Features for Use in Actions) in actions to keep track of the
textual locations of tokens and groupings, then you must provide this
information in 'yylex'.  The function 'yyparse' expects to find the
textual location of a token just parsed in the global variable
'yylloc'.  So 'yylex' must store the proper data in that variable.  The
value of 'yylloc' is a structure and you need only initialize the
members that are going to be used by the actions.  The four members are
called 'first_line', 'first_column', 'last_line' and 'last_column'.
Note that the use of this feature makes the parser noticeably slower.

   The data type of 'yylloc' has the name 'YYLTYPE'.
```

1.63 bison.guide/Pure Calling

```
Calling for Pure Parsers
------------------------

   When you use the Bison declaration '%pure_parser' to request a pure,
reentrant parser, the global communication variables 'yylval' and
'yylloc' cannot be used.  (See A Pure (Reentrant) Parser.)  In such
parsers the two global variables are replaced by pointers passed as
arguments to 'yylex'.  You must declare them as shown here, and pass the
information back by storing it through those pointers.

     yylex (lvalp, llocp)
          YYSTYPE *lvalp;
          YYLTYPE *llocp;
     {
       ...
       *lvalp = value;  /* Put value onto Bison stack.  */
       return INT;      /* Return the type of the token.  */
       ...
     }

   If the grammar file does not use the '@' constructs to refer to
textual positions, then the type 'YYLTYPE' will not be defined.  In
this case, omit the second argument; 'yylex' will be called with only
one argument.
```

1.64 bison.guide/Error Reporting

```
The Error Reporting Function 'yyerror'
======================================

   The Bison parser detects a "parse error" or "syntax error" whenever
it reads a token which cannot satisfy any syntax rule.  A action in the
grammar can also explicitly proclaim an error, using the macro
'YYERROR' (see Special Features for Use in Actions).

   The Bison parser expects to report the error by calling an error
reporting function named 'yyerror', which you must supply.  It is
called by 'yyparse' whenever a syntax error is found, and it receives
one argument.  For a parse error, the string is normally
'"parse error"'.

   If you define the macro 'YYERROR_VERBOSE' in the Bison declarations
section (see The Bison Declarations Section), then Bison provides a
more verbose and specific error message string instead of just plain
'"parse error"'.  It doesn't matter what definition you use for
'YYERROR_VERBOSE', just whether you define it.

   The parser can detect one other kind of error: stack overflow.  This
happens when the input contains constructions that are very deeply
nested.  It isn't likely you will encounter this, since the Bison
parser extends its stack automatically up to a very large limit.  But
```

if overflow happens, ‘yyparse’ calls ‘yyerror’ in the usual fashion, except that the argument string is ‘"parser stack overflow"’.

The following definition suffices in simple programs:

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

After ‘yyerror’ returns to ‘yyparse’, the latter will attempt error recovery if you have written suitable error recovery grammar rules (see Error Recovery). If recovery is impossible, ‘yyparse’ will immediately return 1.

The variable ‘yynerrs’ contains the number of syntax errors encountered so far. Normally this variable is global; but if you request a pure parser (see A Pure (Reentrant) Parser) then it is a local variable which only the actions can access.

1.65 bison.guide/Action Features

Special Features for Use in Actions
==========

Here is a table of Bison constructs, variables and macros that are useful in actions.

‘$$’
: Acts like a variable that contains the semantic value for the grouping made by the current rule. See Actions.

‘$N’
: Acts like a variable that contains the semantic value for the Nth component of the current rule. See Actions.

‘$<TYPEALT>$’
: Like ‘$$’ but specifies alternative TYPEALT in the union specified by the ‘%union’ declaration. See Data Types of Values in Actions.

‘$<TYPEALT>N’
: Like ‘$N’ but specifies alternative TYPEALT in the union specified by the ‘%union’ declaration. See Data Types of Values in Actions.

‘YYABORT;’
: Return immediately from ‘yyparse’, indicating failure. See The Parser Function ‘yyparse’.

‘YYACCEPT;’
: Return immediately from ‘yyparse’, indicating success. See The Parser Function ‘yyparse’.

‘YYBACKUP (TOKEN, VALUE);’

Unshift a token. This macro is allowed only for rules that reduce a single value, and only when there is no look-ahead token. It installs a look-ahead token with token type TOKEN and semantic value VALUE; then it discards the value that was going to be reduced by this rule.

If the macro is used when it is not valid, such as when there is a look-ahead token already, then it reports a syntax error with a message 'cannot back up' and performs ordinary error recovery.

In either case, the rest of the action is not executed.

'YYEMPTY'
Value stored in 'yychar' when there is no look-ahead token.

'YYERROR;'
Cause an immediate syntax error. This statement initiates error recovery just as if the parser itself had detected an error; however, it does not call 'yyerror', and does not print any message. If you want to print an error message, call 'yyerror' explicitly before the 'YYERROR;' statement. See Error Recovery.

'YYRECOVERING'
This macro stands for an expression that has the value 1 when the parser is recovering from a syntax error, and 0 the rest of the time. See Error Recovery.

'yychar'
Variable containing the current look-ahead token. (In a pure parser, this is actually a local variable within 'yyparse'.) When there is no look-ahead token, the value 'YYEMPTY' is stored in the variable. See Look-Ahead Tokens.

'yyclearin;'
Discard the current look-ahead token. This is useful primarily in error rules. See Error Recovery.

'yyerrok;'
Resume generating error messages immediately for subsequent syntax errors. This is useful primarily in error rules. See Error Recovery.

'@N'
Acts like a structure variable containing information on the line numbers and column numbers of the Nth component of the current rule. The structure has four members, like this:

```
struct {
  int first_line, last_line;
  int first_column, last_column;
};
```

Thus, to get the starting line number of the third component, use '@3.first_line'.

In order for the members of this structure to contain valid information, you must make 'yylex' supply this information about

each token. If you need only certain members, then ‘yylex’ need only fill in those members.

The use of this feature makes the parser noticeably slower.

1.66 bison.guide/Algorithm

The Bison Parser Algorithm
****

As Bison reads tokens, it pushes them onto a stack along with their semantic values. The stack is called the "parser stack". Pushing a token is traditionally called "shifting".

For example, suppose the infix calculator has read ‘1 + 5 *’, with a ‘3’ to come. The stack will have four elements, one for each token that was shifted.

But the stack does not always have an element for each token read. When the last N tokens and groupings shifted match the components of a grammar rule, they can be combined according to that rule. This is called "reduction". Those tokens and groupings are replaced on the stack by a single grouping whose symbol is the result (left hand side) of that rule. Running the rule’s action is part of the process of reduction, because this is what computes the semantic value of the resulting grouping.

For example, if the infix calculator’s parser stack contains this:

```
1 + 5 * 3
```

and the next input token is a newline character, then the last three elements can be reduced to 15 via the rule:

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

Then the stack contains just these three elements:

```
1 + 15
```

At this point, another reduction can be made, resulting in the single value 16. Then the newline token can be shifted.

The parser tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar’s start-symbol (see Languages and Context-Free Grammars).

This kind of parser is known in the literature as a bottom-up parser.

Look-Ahead	Parser looks one token ahead when deciding what to do.
Shift-Reduce	Conflicts: when either shifting or reduction is valid.
Precedence	Operator precedence works by resolving conflicts.
Contextual Precedence	When an operator’s precedence depends on context.

```
  Parser States       The parser is a finite-state-machine with stack.
  Reduce-Reduce       When two rules are applicable in the same situation.
  Mystery Conflicts    Reduce/reduce conflicts that look unjustified.
  Stack Overflow      What happens when stack gets full.  How to avoid it.
```

1.67 bison.guide/Look-Ahead

```
Look-Ahead Tokens
=================

   The Bison parser does *not* always reduce immediately as soon as the
last N tokens and groupings match a rule.  This is because such a
simple strategy is inadequate to handle most languages.  Instead, when a
reduction is possible, the parser sometimes "looks ahead" at the next
token in order to decide what to do.

   When a token is read, it is not immediately shifted; first it
becomes the "look-ahead token", which is not on the stack.  Now the
parser can perform one or more reductions of tokens and groupings on
the stack, while the look-ahead token remains off to the side.  When no
more reductions should take place, the look-ahead token is shifted onto
the stack.  This does not mean that all possible reductions have been
done; depending on the token type of the look-ahead token, some rules
may choose to delay their application.

   Here is a simple case where look-ahead is needed.  These three rules
define expressions which contain binary addition operators and postfix
unary factorial operators (`!'), and allow parentheses for grouping.

     expr:     term '+' expr
             | term
             ;

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

   Suppose that the tokens `1 + 2' have been read and shifted; what
should be done?  If the following token is `)', then the first three
tokens must be reduced to form an `expr'.  This is the only valid
course, because shifting the `)' would produce a sequence of symbols
`term ')'', and no rule allows this.

   If the following token is `!', then it must be shifted immediately so
that `2 !' can be reduced to make a `term'.  If instead the parser were
to reduce before shifting, `1 + 2' would become an `expr'.  It would
then be impossible to shift the `!' because doing so would produce on
the stack the sequence of symbols `expr '!''.  No rule allows that
sequence.

   The current look-ahead token is stored in the variable `yychar'.
See Special Features for Use in Actions.
```

1.68 bison.guide/Shift-Reduce

```
Shift/Reduce Conflicts
======================

   Suppose we are parsing a language which has if-then and if-then-else
statements, with a pair of rules like this:

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

Here we assume that ‘IF’, ‘THEN’ and ‘ELSE’ are terminal symbols for
specific keyword tokens.

   When the ‘ELSE’ token is read and becomes the look-ahead token, the
contents of the stack (assuming the input is valid) are just right for
reduction by the first rule.  But it is also legitimate to shift the
‘ELSE’, because that would lead to eventual reduction by the second
rule.

   This situation, where either a shift or a reduction would be valid,
is called a "shift/reduce conflict".  Bison is designed to resolve
these conflicts by choosing to shift, unless otherwise directed by
operator precedence declarations.  To see the reason for this, let’s
contrast it with the other alternative.

   Since the parser prefers to shift the ‘ELSE’, the result is to attach
the else-clause to the innermost if-statement, making these two inputs
equivalent:

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

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

   But if the parser chose to reduce when possible rather than shift,
the result would be to attach the else-clause to the outermost
if-statement, making these two inputs equivalent:

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

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

   The conflict exists because the grammar as written is ambiguous:
either parsing of the simple nested if-statement is legitimate.  The
established convention is that these ambiguities are resolved by
attaching the else-clause to the innermost if-statement; this is what
Bison accomplishes by choosing to shift rather than reduce.  (It would
ideally be cleaner to write an unambiguous grammar, but that is very
hard to do in this case.) This particular ambiguity was first
encountered in the specifications of Algol 60 and is called the
"dangling ‘else’" ambiguity.
```

To avoid warnings from Bison about predictable, legitimate shift/reduce conflicts, use the '%expect N' declaration. There will be no warning as long as the number of shift/reduce conflicts is exactly N. See Suppressing Conflict Warnings.

The definition of 'if_stmt' above is solely to blame for the conflict, but the conflict does not actually appear without additional rules. Here is a complete Bison input file that actually manifests the conflict:

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

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

expr:     variable
        ;
```

1.69 bison.guide/Precedence

```
Operator Precedence
===================
```

Another situation where shift/reduce conflicts appear is in arithmetic expressions. Here shifting is not always the preferred resolution; the Bison declarations for operator precedence allow you to specify when to shift and when to reduce.

```
  Why Precedence      An example showing why precedence is needed.
  Using Precedence    How to specify precedence in Bison grammars.
  Precedence Examples    How these features are used in the previous example.
  How Precedence      How they work.
```

1.70 bison.guide/Why Precedence

```
When Precedence is Needed
-------------------------
```

Consider the following ambiguous grammar fragment (ambiguous because the input '1 - 2 * 3' can be parsed in two different ways):

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

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

Suppose the parser has seen the tokens '1', '-' and '2'; should it reduce them via the rule for the addition operator? It depends on the next token. Of course, if the next token is ')', we must reduce; shifting is invalid because no single rule can reduce the token sequence '- 2)' or anything starting with that. But if the next token is '*' or '<', we have a choice: either shifting or reduction would allow the parse to complete, but with different results.

To decide which one Bison should do, we must consider the results. If the next operator token OP is shifted, then it must be reduced first in order to permit another opportunity to reduce the sum. The result is (in effect) '1 - (2 OP 3)'. On the other hand, if the subtraction is reduced before shifting OP, the result is '(1 - 2) OP 3'. Clearly, then, the choice of shift or reduce should depend on the relative precedence of the operators '-' and OP: '*' should be shifted first, but not '<'.

What about input such as '1 - 2 - 5'; should this be '(1 - 2) - 5' or should it be '1 - (2 - 5)'? For most operators we prefer the former, which is called "left association". The latter alternative, "right association", is desirable for assignment operators. The choice of left or right association is a matter of whether the parser chooses to shift or reduce when the stack contains '1 - 2' and the look-ahead token is '-': shifting makes right-associativity.

1.71 bison.guide/Using Precedence

```
Specifying Operator Precedence
------------------------------
```

Bison allows you to specify these choices with the operator precedence declarations '%left' and '%right'. Each such declaration contains a list of tokens, which are operators whose precedence and associativity is being declared. The '%left' declaration makes all those operators left-associative and the '%right' declaration makes them right-associative. A third alternative is '%nonassoc', which declares that it is a syntax error to find the same operator twice "in a row".

The relative precedence of different operators is controlled by the order in which they are declared. The first '%left' or '%right' declaration in the file declares the operators whose precedence is lowest, the next such declaration declares the operators whose precedence is a little higher, and so on.

1.72 bison.guide/Precedence Examples

```
Precedence Examples
-------------------

   In our example, we would want the following declarations:

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

   In a more complete example, which supports other operators as well,
we would declare them in groups of equal precedence.  For example,
``+'' is declared with ``-'':

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

(Here `NE' and so on stand for the operators for "not equal" and so on.
We assume that these tokens are more than one character long and
therefore are represented by names, not character literals.)
```

1.73 bison.guide/How Precedence

```
How Precedence Works
--------------------

   The first effect of the precedence declarations is to assign
precedence levels to the terminal symbols declared.  The second effect
is to assign precedence levels to certain rules: each rule gets its
precedence from the last terminal symbol mentioned in the components.
(You can also specify explicitly the precedence of a rule.  See
Context-Dependent Precedence.)

   Finally, the resolution of conflicts works by comparing the
precedence of the rule being considered with that of the look-ahead
token.  If the token's precedence is higher, the choice is to shift.
If the rule's precedence is higher, the choice is to reduce.  If they
have equal precedence, the choice is made based on the associativity of
that precedence level.  The verbose output file made by `-v' (see
Invoking Bison) says how each conflict was resolved.

   Not all rules and not all tokens have precedence.  If either the
rule or the look-ahead token has no precedence, then the default is to
shift.
```

1.74 bison.guide/Contextual Precedence

Context-Dependent Precedence
==========

Often the precedence of an operator depends on the context. This sounds outlandish at first, but it is really very common. For example, a minus sign typically has a very high precedence as a unary operator, and a somewhat lower precedence (lower than multiplication) as a binary operator.

The Bison precedence declarations, ‘%left’, ‘%right’ and ‘%nonassoc’, can only be used once for a given token; so a token has only one precedence declared in this way. For context-dependent precedence, you need to use an additional mechanism: the ‘%prec’ modifier for rules.

The ‘%prec’ modifier declares the precedence of a particular rule by specifying a terminal symbol whose precedence should be used for that rule. It’s not necessary for that symbol to appear otherwise in the rule. The modifier’s syntax is:

```
%prec TERMINAL-SYMBOL
```

and it is written after the components of the rule. Its effect is to assign the rule the precedence of TERMINAL-SYMBOL, overriding the precedence that would be deduced for it in the ordinary way. The altered rule precedence then affects how conflicts involving that rule are resolved (see Operator Precedence).

Here is how ‘%prec’ solves the problem of unary minus. First, declare a precedence for a fictitious terminal symbol named ‘UMINUS’. There are no tokens of this type, but the symbol serves to stand for its precedence:

```
...
%left '+' '-'
%left '*'
%left UMINUS
```

Now the precedence of ‘UMINUS’ can be used in specific rules:

```
exp:    ...
        | exp '-' exp
        ...
        | '-' exp %prec UMINUS
```

1.75 bison.guide/Parser States

Parser States
==========

The function ‘yyparse’ is implemented using a finite-state machine. The values pushed on the parser stack are not simply token type codes; they represent the entire sequence of terminal and nonterminal symbols

at or near the top of the stack. The current state collects all the information about previous input which is relevant to deciding what to do next.

Each time a look-ahead token is read, the current parser state together with the type of look-ahead token are looked up in a table. This table entry can say, "Shift the look-ahead token." In this case, it also specifies the new parser state, which is pushed onto the top of the parser stack. Or it can say, "Reduce using rule number N." This means that a certain number of tokens or groupings are taken off the top of the stack, and replaced by one grouping. In other words, that number of states are popped from the stack, and one new state is pushed.

There is one other alternative: the table can say that the look-ahead token is erroneous in the current state. This causes error processing to begin (see Error Recovery).

1.76 bison.guide/Reduce-Reduce

Reduce/Reduce Conflicts
==========

A reduce/reduce conflict occurs if there are two or more rules that apply to the same sequence of input. This usually indicates a serious error in the grammar.

For example, here is an erroneous attempt to define a sequence of zero or more ‘word’ groupings.

```
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); }
        ;
```

The error is an ambiguity: there is more than one way to parse a single ‘word’ into a ‘sequence’. It could be reduced to a ‘maybeword’ and then into a ‘sequence’ via the second rule. Alternatively, nothing-at-all could be reduced into a ‘sequence’ via the first rule, and this could be combined with the ‘word’ using the third rule for ‘sequence’.

There is also more than one way to reduce nothing-at-all into a ‘sequence’. This can be done directly via the first rule, or indirectly via ‘maybeword’ and then the second rule.

You might think that this is a distinction without a difference,

because it does not change whether any particular input is valid or not. But it does affect which actions are run. One parsing order runs the second rule's action; the other runs the first rule's action and the third rule's action. In this example, the output of the program changes.

Bison resolves a reduce/reduce conflict by choosing to use the rule that appears first in the grammar, but it is very risky to rely on this. Every reduce/reduce conflict must be studied and usually eliminated. Here is the proper way to define 'sequence':

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

Here is another common error that yields a reduce/reduce conflict:

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

words:    /* empty */
        | words word
        ;

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

The intention here is to define a sequence which can contain either 'word' or 'redirect' groupings. The individual definitions of 'sequence', 'words' and 'redirects' are error-free, but the three together make a subtle ambiguity: even an empty input can be parsed in infinitely many ways!

Consider: nothing-at-all could be a 'words'. Or it could be two 'words' in a row, or three, or any number. It could equally well be a 'redirects', or two, or any number. Or it could be a 'words' followed by three 'redirects' and another 'words'. And so on.

Here are two ways to correct these rules. First, to make it a single level of sequence:

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

Second, to prevent either a 'words' or a 'redirects' from being empty:

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

```
            ;

    words:    word
            | words word
            ;

    redirects:redirect
            | redirects redirect
            ;
```

1.77 bison.guide/Mystery Conflicts

```
Mysterious Reduce/Reduce Conflicts
==================================

   Sometimes reduce/reduce conflicts can occur that don't look
warranted.  Here is an example:

     %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
             ;

   It would seem that this grammar can be parsed with only a single
token of look-ahead: when a 'param_spec' is being read, an 'ID' is a
'name' if a comma or colon follows, or a 'type' if another 'ID'
follows.  In other words, this grammar is LR(1).

   However, Bison, like most parser generators, cannot actually handle
all LR(1) grammars.  In this grammar, two contexts, that after an 'ID'
at the beginning of a 'param_spec' and likewise at the beginning of a
'return_spec', are similar enough that Bison assumes they are the same.
They appear similar because the same set of rules would be active--the
rule for reducing to a 'name' and that for reducing to a 'type'.  Bison
is unable to determine at that stage of processing that the rules would
require different look-ahead tokens in the two contexts, so it makes a
single parser state for them both.  Combining the two contexts causes a
```

conflict later. In parser terminology, this occurrence means that the grammar is not LALR(1).

In general, it is better to fix deficiencies than to document them. But this particular deficiency is intrinsically hard to fix; parser generators that can handle LR(1) grammars are hard to write and tend to produce parsers that are very large. In practice, Bison is more useful as it is now.

When the problem arises, you can often fix it by identifying the two parser states that are being confused, and adding something to make them look distinct. In the above example, adding one rule to 'return_spec' as follows makes the problem go away:

```
%token BOGUS
...
%%
...
return_spec:
             type
        |    name ':' type
        /* This rule is never used.  */
        |    ID BOGUS
        ;
```

This corrects the problem because it introduces the possibility of an additional active rule in the context after the 'ID' at the beginning of 'return_spec'. This rule is not active in the corresponding context in a 'param_spec', so the two contexts receive distinct parser states. As long as the token 'BOGUS' is never generated by 'yylex', the added rule cannot alter the way actual input is parsed.

In this particular example, there is another way to solve the problem: rewrite the rule for 'return_spec' to use 'ID' directly instead of via 'name'. This also causes the two confusing contexts to have different sets of active rules, because the one for 'return_spec' activates the altered rule for 'return_spec' rather than the one for 'name'.

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

1.78 bison.guide/Stack Overflow

```
Stack Overflow, and How to Avoid It
===================================
```

The Bison parser stack can overflow if too many tokens are shifted

and not reduced. When this happens, the parser function 'yyparse' returns a nonzero value, pausing only to call 'yyerror' to report the overflow.

By defining the macro 'YYMAXDEPTH', you can control how deep the parser stack can become before a stack overflow occurs. Define the macro with a value that is an integer. This value is the maximum number of tokens that can be shifted (and not reduced) before overflow. It must be a constant expression whose value is known at compile time.

The stack space allowed is not necessarily allocated. If you specify a large value for 'YYMAXDEPTH', the parser actually allocates a small stack at first, and then makes it bigger by stages as needed. This increasing allocation happens automatically and silently. Therefore, you do not need to make 'YYMAXDEPTH' painfully small merely to save space for ordinary inputs that do not need much stack.

The default value of 'YYMAXDEPTH', if you do not define it, is 10000.

You can control how much stack is allocated initially by defining the macro 'YYINITDEPTH'. This value too must be a compile-time constant integer. The default is 200.

1.79 bison.guide/Error Recovery

Error Recovery
****

It is not usually acceptable to have a program terminate on a parse error. For example, a compiler should recover sufficiently to parse the rest of the input file and check it for errors; a calculator should accept another expression.

In a simple interactive command parser where each input is one line, it may be sufficient to allow 'yyparse' to return 1 on error and have the caller ignore the rest of the input line when that happens (and then call 'yyparse' again). But this is inadequate for a compiler, because it forgets all the syntactic context leading up to the error. A syntax error deep within a function in the compiler input should not cause the compiler to treat the following line like the beginning of a source file.

You can define how to recover from a syntax error by writing rules to recognize the special token 'error'. This is a terminal symbol that is always defined (you need not declare it) and reserved for error handling. The Bison parser generates an 'error' token whenever a syntax error happens; if you have provided a rule to recognize this token in the current context, the parse can continue.

For example:

```
stmnts:  /* empty string */
        | stmnts '\n'
        | stmnts exp '\n'
```

```
            | stmnts error '\n'
```

The fourth rule in this example says that an error followed by a newline makes a valid addition to any 'stmnts'.

What happens if a syntax error occurs in the middle of an 'exp'? The error recovery rule, interpreted strictly, applies to the precise sequence of a 'stmnts', an 'error' and a newline. If an error occurs in the middle of an 'exp', there will probably be some additional tokens and subexpressions on the stack after the last 'stmnts', and there will be tokens to read before the next newline. So the rule is not applicable in the ordinary way.

But Bison can force the situation to fit the rule, by discarding part of the semantic context and part of the input. First it discards states and objects from the stack until it gets back to a state in which the 'error' token is acceptable. (This means that the subexpressions already parsed are discarded, back to the last complete 'stmnts'.) At this point the 'error' token can be shifted. Then, if the old look-ahead token is not acceptable to be shifted next, the parser reads tokens and discards them until it finds a token which is acceptable. In this example, Bison reads and discards input until the next newline so that the fourth rule can apply.

The choice of error rules in the grammar is a choice of strategies for error recovery. A simple and useful strategy is simply to skip the rest of the current input line or current statement if an error is detected:

```
     stmnt: error ';'  /* on error, skip until ';' is read */
```

It is also useful to recover to the matching close-delimiter of an opening-delimiter that has already been parsed. Otherwise the close-delimiter will probably appear to be unmatched, and generate another, spurious error message:

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

Error recovery strategies are necessarily guesses. When they guess wrong, one syntax error often leads to another. In the above example, the error recovery rule guesses that an error is due to bad input within one 'stmnt'. Suppose that instead a spurious semicolon is inserted in the middle of a valid 'stmnt'. After the error recovery rule recovers from the first error, another syntax error will be found straightaway, since the text following the spurious semicolon is also an invalid 'stmnt'.

To prevent an outpouring of error messages, the parser will output no error message for another syntax error that happens shortly after the first; only after three consecutive input tokens have been successfully shifted will error messages resume.

Note that rules which accept the 'error' token may have actions, just as any other rules can.

You can make error messages resume immediately by using the macro 'yyerrok' in an action. If you do this in the error rule's action, no error messages will be suppressed. This macro requires no arguments; 'yyerrok;' is a valid C statement.

The previous look-ahead token is reanalyzed immediately after an error. If this is unacceptable, then the macro 'yyclearin' may be used to clear this token. Write the statement 'yyclearin;' in the error rule's action.

For example, suppose that on a parse error, an error handling routine is called that advances the input stream to some point where parsing should once again commence. The next symbol returned by the lexical scanner is probably correct. The previous look-ahead token ought to be discarded with 'yyclearin;'.

The macro 'YYRECOVERING' stands for an expression that has the value 1 when the parser is recovering from a syntax error, and 0 the rest of the time. A value of 1 indicates that error messages are currently suppressed for new syntax errors.

1.80 bison.guide/Context Dependency

Handling Context Dependencies
****

The Bison paradigm is to parse tokens first, then group them into larger syntactic units. In many languages, the meaning of a token is affected by its context. Although this violates the Bison paradigm, certain techniques (known as "kludges") may enable you to write Bison parsers for such languages.

```
  Semantic Tokens     Token parsing can depend on the semantic context.
  Lexical Tie-ins     Token parsing can depend on the syntactic context.
  Tie-in Recovery     Lexical tie-ins have implications for how
                        error recovery rules must be written.
```

(Actually, "kludge" means any technique that gets its job done but is neither clean nor robust.)

1.81 bison.guide/Semantic Tokens

Semantic Info in Token Types
==========

The C language has a context dependency: the way an identifier is used depends on what its current meaning is. For example, consider this:

```
foo (x);
```

This looks like a function call statement, but if ‘foo’ is a typedef name, then this is actually a declaration of ‘x’. How can a Bison parser for C decide how to parse this input?

The method used in GNU C is to have two different token types, ‘IDENTIFIER’ and ‘TYPENAME’. When ‘yylex’ finds an identifier, it looks up the current declaration of the identifier in order to decide which token type to return: ‘TYPENAME’ if the identifier is declared as a typedef, ‘IDENTIFIER’ otherwise.

The grammar rules can then express the context dependency by the choice of token type to recognize. ‘IDENTIFIER’ is accepted as an expression, but ‘TYPENAME’ is not. ‘TYPENAME’ can start a declaration, but ‘IDENTIFIER’ cannot. In contexts where the meaning of the identifier is *not* significant, such as in declarations that can shadow a typedef name, either ‘TYPENAME’ or ‘IDENTIFIER’ is accepted--there is one rule for each of the two token types.

This technique is simple to use if the decision of which kinds of identifiers to allow is made at a place close to where the identifier is parsed. But in C this is not always so: C allows a declaration to redeclare a typedef name provided an explicit type has been specified earlier:

```
typedef int foo, bar, lose;
static foo (bar);        /* redeclare ‘bar’ as static variable */
static int foo (lose);   /* redeclare ‘foo’ as function */
```

Unfortunately, the name being declared is separated from the declaration construct itself by a complicated syntactic structure--the "declarator".

As a result, the part of Bison parser for C needs to be duplicated, with all the nonterminal names changed: once for parsing a declaration in which a typedef name can be redefined, and once for parsing a declaration in which that can’t be done. Here is a part of the duplication, with actions omitted for brevity:

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

notype_initdcl:
          notype_declarator maybeasm '='
          init
        | notype_declarator maybeasm
        ;
```

Here ‘initdcl’ can redeclare a typedef name, but ‘notype_initdcl’ cannot. The distinction between ‘declarator’ and ‘notype_declarator’ is the same sort of thing.

There is some similarity between this technique and a lexical tie-in (described next), in that information which alters the lexical analysis is changed during parsing by other parts of the program. The difference is here the information is global, and is used for other purposes in the program. A true lexical tie-in has a special-purpose flag controlled by the syntactic context.

1.82 bison.guide/Lexical Tie-ins

Lexical Tie-ins
==========

One way to handle context-dependency is the "lexical tie-in": a flag which is set by Bison actions, whose purpose is to alter the way tokens are parsed.

For example, suppose we have a language vaguely like C, but with a special construct 'hex (HEX-EXPR)'. After the keyword 'hex' comes an expression in parentheses in which all integers are hexadecimal. In particular, the token 'a1b' must be treated as an integer rather than as an identifier if it appears in that context. Here is how you can do it:

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

constant:
          INTEGER
        | STRING
        ;
```

Here we assume that 'yylex' looks at the value of 'hexflag'; when it is nonzero, all integers are parsed in hexadecimal, and tokens starting with letters are parsed as integers if possible.

The declaration of 'hexflag' shown in the C declarations section of the parser file is needed to make it accessible to the actions (see The C Declarations Section). You must also write the code in 'yylex' to obey the flag.

1.83 bison.guide/Tie-in Recovery

```
Lexical Tie-ins and Error Recovery
==================================

   Lexical tie-ins make strict demands on any error recovery rules you
have.  See Error Recovery.

   The reason for this is that the purpose of an error recovery rule is
to abort the parsing of one construct and resume in some larger
construct.  For example, in C-like languages, a typical error recovery
rule is to skip tokens until the next semicolon, and then start a new
statement, like this:

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

   If there is a syntax error in the middle of a 'hex (EXPR)'
construct, this error rule will apply, and then the action for the
completed 'hex (EXPR)' will never run.  So 'hexflag' would remain set
for the entire rest of the input, or until the next 'hex' keyword,
causing identifiers to be misinterpreted as integers.

   To avoid this problem the error recovery rule itself clears
'hexflag'.

   There may also be an error recovery rule that works within
expressions.  For example, there could be a rule which applies within
parentheses and skips to the close-parenthesis:

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

   If this rule acts within the 'hex' construct, it is not going to
abort that construct (since it applies to an inner level of parentheses
within the construct).  Therefore, it should not clear the flag: the
rest of the 'hex' construct should be parsed with the flag still in
effect.

   What if there is an error recovery rule which might abort out of the
'hex' construct or might not, depending on circumstances?  There is no
way you can write the action to determine whether a 'hex' construct is
being aborted or not.  So if you are using a lexical tie-in, you had
better make sure your error recovery rules are not of this kind.  Each
rule must be such that you can be sure that it always will, or always
won't, have to clear the flag.
```

1.84 bison.guide/Debugging

```
Debugging Your Parser
*********************

   If a Bison grammar compiles properly but doesn't do what you want
when it runs, the 'yydebug' parser-trace feature can help you figure
out why.

   To enable compilation of trace facilities, you must define the macro
'YYDEBUG' when you compile the parser.  You could use '-DYYDEBUG=1' as
a compiler option or you could put '#define YYDEBUG 1' in the C
declarations section of the grammar file (see
The C Declarations Section).  Alternatively, use the '-t' option when
you run Bison (see Invoking Bison).  We always define 'YYDEBUG' so that
debugging is always possible.

   The trace facility uses 'stderr', so you must add
'#include <stdio.h>' to the C declarations section unless it is already
there.

   Once you have compiled the program with trace facilities, the way to
request a trace is to store a nonzero value in the variable 'yydebug'.
You can do this by making the C code do it (in 'main', perhaps), or you
can alter the value with a C debugger.

   Each step taken by the parser when 'yydebug' is nonzero produces a
line or two of trace information, written on 'stderr'.  The trace
messages tell you these things:

   * Each time the parser calls 'yylex', what kind of token was read.

   * Each time a token is shifted, the depth and complete contents of
     the state stack (see Parser States).

   * Each time a rule is reduced, which rule it is, and the complete
     contents of the state stack afterward.

   To make sense of this information, it helps to refer to the listing
file produced by the Bison '-v' option (see Invoking Bison).  This file
shows the meaning of each state in terms of positions in various rules,
and also what each state will do with each possible input token.  As
you read the successive trace messages, you can see that the parser is
functioning according to its specification in the listing file.
Eventually you will arrive at the place where something undesirable
happens, and you will see which parts of the grammar are to blame.

   The parser file is a C program and you can use C debuggers on it,
but it's not easy to interpret what it is doing.  The parser function
is a finite-state machine interpreter, and aside from the actions it
executes the same code over and over.  Only the values of variables
show where in the grammar it is working.
```

The debugging information normally gives the token type of each token read, but not its semantic value. You can optionally define a macro named 'YYPRINT' to provide a way to print the value. If you define 'YYPRINT', it should take three arguments. The parser will pass a standard I/O stream, the numeric code for the token type, and the token value (from 'yylval').

Here is an example of 'YYPRINT' suitable for the multi-function calculator (see Declarations for 'mfcalc'):

```
#define YYPRINT(file, type, value)   yyprint (file, type, value)

static void
yyprint (file, type, 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);
}
```

1.85 bison.guide/Invocation

```
Invoking Bison
**************
```

The usual way to invoke Bison is as follows:

```
bison INFILE
```

Here INFILE is the grammar file name, which usually ends in '.y'. The parser file's name is made by replacing the '.y' with '.tab.c'. Thus, the 'bison foo.y' filename yields 'foo.tab.c', and the 'bison hack/foo.y' filename yields 'hack/foo.tab.c'.

```
  Bison Options       All the options described in detail,
      in alphabetical order by short options.
  Option Cross Key    Alphabetical list of long options.
  VMS Invocation      Bison command syntax on VMS.
```

1.86 bison.guide/Bison Options

```
Bison Options
=============
```

Bison supports both traditional single-letter options and mnemonic

long option names. Long option names are indicated with '--' instead of '-'. Abbreviations for option names are allowed as long as they are unique. When a long option takes an argument, like '--file-prefix', connect the option name and the argument with '='.

Here is a list of options that can be used with Bison, alphabetized by short option. It is followed by a cross key alphabetized by long option.

'-b FILE-PREFIX'
'--file-prefix=PREFIX'
Specify a prefix to use for all Bison output file names. The names are chosen as if the input file were named 'PREFIX.c'.

'-d'
'--defines'
Write an extra output file containing macro definitions for the token type names defined in the grammar and the semantic value type 'YYSTYPE', as well as a few 'extern' variable declarations.

If the parser output file is named 'NAME.c' then this file is named 'NAME.h'.

This output file is essential if you wish to put the definition of 'yylex' in a separate source file, because 'yylex' needs to be able to refer to token type codes and the variable 'yylval'. See Semantic Values of Tokens.

'-l'
'--no-lines'
Don't put any '#line' preprocessor commands in the parser file. Ordinarily Bison puts them in the parser file so that the C compiler and debuggers will associate errors with your source file, the grammar file. This option causes them to associate errors with the parser file, treating it an independent source file in its own right.

'-o OUTFILE'
'--output-file=OUTFILE'
Specify the name OUTFILE for the parser file.

The other output files' names are constructed from OUTFILE as described under the '-v' and '-d' switches.

'-p PREFIX'
'--name-prefix=PREFIX'
Rename the external symbols used in the parser so that they start with PREFIX instead of 'yy'. The precise list of symbols renamed is 'yyparse', 'yylex', 'yyerror', 'yylval', 'yychar' and 'yydebug'.

For example, if you use '-p c', the names become 'cparse', 'clex', and so on.

See Multiple Parsers in the Same Program.

'-t'
'--debug'

```
     Output a definition of the macro 'YYDEBUG' into the parser file,
     so that the debugging facilities are compiled.  See
     Debugging Your Parser.

'-v'
'--verbose'
     Write an extra output file containing verbose descriptions of the
     parser states and what is done for each type of look-ahead token in
     that state.

     This file also describes all the conflicts, both those resolved by
     operator precedence and the unresolved ones.

     The file's name is made by removing '.tab.c' or '.c' from the
     parser output file name, and adding '.output' instead.

     Therefore, if the input file is 'foo.y', then the parser file is
     called 'foo.tab.c' by default.  As a consequence, the verbose
     output file is called 'foo.output'.

'-V'
'--version'
     Print the version number of Bison and exit.

'-h'
'--help'
     Print a summary of the command-line options to Bison and exit.

'-y'
'--yacc'
'--fixed-output-files'
     Equivalent to '-o y.tab.c'; the parser output file is called
     'y.tab.c', and the other outputs are called 'y.output' and
     'y.tab.h'.  The purpose of this switch is to imitate Yacc's output
     file name conventions.  Thus, the following shell script can
     substitute for Yacc:

          bison -y $*
```

1.87 bison.guide/Option Cross Key

```
Option Cross Key
================

   Here is a list of options, alphabetized by long option, to help you
find the corresponding short option.

     --debug                                -t
     --defines                              -d
     --file-prefix=PREFIX                   -b FILE-PREFIX
     --fixed-output-files --yacc            -y
     --help                                 -h
     --name-prefix                          -p
     --no-lines                             -l
```

```
     --output-file=OUTFILE                 -o OUTFILE
     --verbose                             -v
     --version                             -V
```

1.88 bison.guide/VMS Invocation

```
Invoking Bison under VMS
========================

   The command line syntax for Bison on VMS is a variant of the usual
Bison command syntax--adapted to fit VMS conventions.

   To find the VMS equivalent for any Bison option, start with the long
option, and substitute a '/' for the leading '--', and substitute a '_'
for each '-' in the name of the long option.  For example, the
following invocation under VMS:

     bison /debug/name_prefix=bar foo.y

is equivalent to the following command under POSIX.

     bison --debug --name-prefix=bar foo.y

   The VMS file system does not permit filenames such as 'foo.tab.c'.
In the above example, the output file would instead be named
'foo_tab.c'.
```

1.89 bison.guide/Table of Symbols

```
Bison Symbols
*************

'error'
     A token name reserved for error recovery.  This token may be used
     in grammar rules so as to allow the Bison parser to recognize an
     error in the grammar without halting the process.  In effect, a
     sentence containing an error may be recognized as valid.  On a
     parse error, the token 'error' becomes the current look-ahead
     token.  Actions corresponding to 'error' are then executed, and
     the look-ahead token is reset to the token that originally caused
     the violation.  See Error Recovery.

'YYABORT'
     Macro to pretend that an unrecoverable syntax error has occurred,
     by making 'yyparse' return 1 immediately.  The error reporting
     function 'yyerror' is not called.  See
     The Parser Function 'yyparse'.

'YYACCEPT'
     Macro to pretend that a complete utterance of the language has been
```

read, by making ‘yyparse’ return 0 immediately. See The Parser Function ‘yyparse’.

‘YYBACKUP’
Macro to discard a value from the parser stack and fake a look-ahead token. See Special Features for Use in Actions.

‘YYERROR’
Macro to pretend that a syntax error has just been detected: call ‘yyerror’ and then perform normal error recovery if possible (see Error Recovery), or (if recovery is impossible) make ‘yyparse’ return 1. See Error Recovery.

‘YYERROR_VERBOSE’
Macro that you define with ‘#define’ in the Bison declarations section to request verbose, specific error message strings when ‘yyerror’ is called.

‘YYINITDEPTH’
Macro for specifying the initial size of the parser stack. See Stack Overflow.

‘YYLTYPE’
Macro for the data type of ‘yylloc’; a structure with four members. See Textual Positions of Tokens.

‘YYMAXDEPTH’
Macro for specifying the maximum size of the parser stack. See Stack Overflow.

‘YYRECOVERING’
Macro whose value indicates whether the parser is recovering from a syntax error. See Special Features for Use in Actions.

‘YYSTYPE’
Macro for the data type of semantic values; ‘int’ by default. See Data Types of Semantic Values.

‘yychar’
External integer variable that contains the integer value of the current look-ahead token. (In a pure parser, it is a local variable within ‘yyparse’.) Error-recovery rule actions may examine this variable. See Special Features for Use in Actions.

‘yyclearin’
Macro used in error-recovery rule actions. It clears the previous look-ahead token. See Error Recovery.

‘yydebug’
External integer variable set to zero by default. If ‘yydebug’ is given a nonzero value, the parser will output information on input symbols and parser action. See Debugging Your Parser.

‘yyerrok’
Macro to cause parser to recover immediately to its normal mode after a parse error. See Error Recovery.

```
'yyerror'
     User-supplied function to be called by 'yyparse' on error.  The
     function receives one argument, a pointer to a character string
     containing an error message.  See
     The Error Reporting Function 'yyerror'.

'yylex'
     User-supplied lexical analyzer function, called with no arguments
     to get the next token.  See The Lexical Analyzer Function 'yylex'.

'yylval'
     External variable in which 'yylex' should place the semantic value
     associated with a token.  (In a pure parser, it is a local
     variable within 'yyparse', and its address is passed to 'yylex'.)
     See Semantic Values of Tokens.

'yylloc'
     External variable in which 'yylex' should place the line and
     column numbers associated with a token.  (In a pure parser, it is a
     local variable within 'yyparse', and its address is passed to
     'yylex'.)  You can ignore this variable if you don't use the '@'
     feature in the grammar actions.  See Textual Positions of Tokens.

'yynerrs'
     Global variable which Bison increments each time there is a parse
     error.  (In a pure parser, it is a local variable within
     'yyparse'.)  See The Error Reporting Function 'yyerror'.

'yyparse'
     The parser function produced by Bison; call this function to start
     parsing.  See The Parser Function 'yyparse'.

'%left'
     Bison declaration to assign left associativity to token(s).  See
     Operator Precedence.

'%nonassoc'
     Bison declaration to assign nonassociativity to token(s).  See
     Operator Precedence.

'%prec'
     Bison declaration to assign a precedence to a specific rule.  See
     Context-Dependent Precedence.

'%pure_parser'
     Bison declaration to request a pure (reentrant) parser.  See
     A Pure (Reentrant) Parser.

'%right'
     Bison declaration to assign right associativity to token(s).  See
     Operator Precedence.

'%start'
     Bison declaration to specify the start symbol.  See
     The Start-Symbol.

'%token'
```

```
     Bison declaration to declare token(s) without specifying
     precedence.  See Token Type Names.

‘%type’
     Bison declaration to declare nonterminals.  See
     Nonterminal Symbols.

‘%union’
     Bison declaration to specify several possible data types for
     semantic values.  See The Collection of Value Types.

   These are the punctuation and delimiters used in Bison input:

‘%%’
     Delimiter used to separate the grammar rule section from the Bison
     declarations section or the additional C code section.  See
     The Overall Layout of a Bison Grammar.

‘%{ %}’
     All code listed between ‘%{’ and ‘%}’ is copied directly to the
     output file uninterpreted.  Such code forms the "C declarations"
     section of the input file.  See Outline of a Bison Grammar.

‘/*...*/’
     Comment delimiters, as in C.

‘:’
     Separates a rule’s result from its components.  See
     Syntax of Grammar Rules.

‘;’
     Terminates a rule.  See Syntax of Grammar Rules.

‘|’
     Separates alternate rules for the same result nonterminal.  See
     Syntax of Grammar Rules.
```

1.90 bison.guide/Glossary

```
Glossary
********

Backus-Naur Form (BNF)
     Formal method of specifying context-free grammars.  BNF was first
     used in the ‘ALGOL-60’ report, 1963.  See
     Languages and Context-Free Grammars.

Context-free grammars
     Grammars specified as rules that can be applied regardless of
     context.  Thus, if there is a rule which says that an integer can
     be used as an expression, integers are allowed *anywhere* an
     expression is permitted.  See Languages and Context-Free Grammars.

Dynamic allocation
```

Allocation of memory that occurs during execution, rather than at compile time or on entry to a function.

Empty string
Analogous to the empty set in set theory, the empty string is a character string of length zero.

Finite-state stack machine
A "machine" that has discrete states in which it is said to exist at each instant in time. As input to the machine is processed, the machine moves from state to state as specified by the logic of the machine. In the case of the parser, the input is the language being parsed, and the states correspond to various stages in the grammar rules. See The Bison Parser Algorithm.

Grouping
A language construct that is (in general) grammatically divisible; for example, 'expression' or 'declaration' in C. See Languages and Context-Free Grammars.

Infix operator
An arithmetic operator that is placed between the operands on which it performs some operation.

Input stream
A continuous flow of data between devices or programs.

Language construct
One of the typical usage schemas of the language. For example, one of the constructs of the C language is the 'if' statement. See Languages and Context-Free Grammars.

Left associativity
Operators having left associativity are analyzed from left to right: 'a+b+c' first computes 'a+b' and then combines with 'c'. See Operator Precedence.

Left recursion
A rule whose result symbol is also its first component symbol; for example, 'expseq1 : expseq1 ',' exp;'. See Recursive Rules.

Left-to-right parsing
Parsing a sentence of a language by analyzing it token by token from left to right. See The Bison Parser Algorithm.

Lexical analyzer (scanner)
A function that reads an input stream and returns tokens one by one. See The Lexical Analyzer Function 'yylex'.

Lexical tie-in
A flag, set by actions in the grammar rules, which alters the way tokens are parsed. See Lexical Tie-ins.

Look-ahead token
A token already read but not yet shifted. See Look-Ahead Tokens.

LALR(1)

The class of context-free grammars that Bison (like most other parser generators) can handle; a subset of LR(1). See Mysterious Reduce/Reduce Conflicts.

LR(1)
The class of context-free grammars in which at most one token of look-ahead is needed to disambiguate the parsing of any piece of input.

Nonterminal symbol
A grammar symbol standing for a grammatical construct that can be expressed through rules in terms of smaller constructs; in other words, a construct that is not a token. See Symbols.

Parse error
An error encountered during parsing of an input stream due to invalid syntax. See Error Recovery.

Parser
A function that recognizes valid sentences of a language by analyzing the syntax structure of a set of tokens passed to it from a lexical analyzer.

Postfix operator
An arithmetic operator that is placed after the operands upon which it performs some operation.

Reduction
Replacing a string of nonterminals and/or terminals with a single nonterminal, according to a grammar rule. See The Bison Parser Algorithm.

Reentrant
A reentrant subprogram is a subprogram which can be in invoked any number of times in parallel, without interference between the various invocations. See A Pure (Reentrant) Parser.

Reverse polish notation
A language in which all operators are postfix operators.

Right recursion
A rule whose result symbol is also its last component symbol; for example, ‘expseq1: exp ’,’ expseq1;’. See Recursive Rules.

Semantics
In computer languages, the semantics are specified by the actions taken for each instance of the language, i.e., the meaning of each statement. See Defining Language Semantics.

Shift
A parser is said to shift when it makes the choice of analyzing further input from the stream rather than reducing immediately some already-recognized rule. See The Bison Parser Algorithm.

Single-character literal
A single character that is recognized and interpreted as is. See From Formal Rules to Bison Input.

```
Start symbol
     The nonterminal symbol that stands for a complete valid utterance
     in the language being parsed.  The start symbol is usually listed
     as the first nonterminal symbol in a language specification.  See
     The Start-Symbol.

Symbol table
     A data structure where symbol names and associated data are stored
     during parsing to allow for recognition and use of existing
     information in repeated uses of a symbol.  See Multi-function Calc.

Token
     A basic, grammatically indivisible unit of a language.  The symbol
     that describes a token in the grammar is a terminal symbol.  The
     input of the Bison parser is a stream of tokens which comes from
     the lexical analyzer.  See Symbols.

Terminal symbol
     A grammar symbol that has no rules in the grammar and therefore is
     grammatically indivisible.  The piece of text it represents is a
     token.  See Languages and Context-Free Grammars.
```

1.91 bison.guide/Index

```
Index
*****


  $$                                    Actions
  $N                                    Actions
  %expect                               Expect Decl
  %left                                 Using Precedence
  %nonassoc                             Using Precedence
  %prec                                 Contextual Precedence
  %pure_parser                          Pure Decl
  %right                                Using Precedence
  %start                                Start Decl
  %token                                Token Decl
  %type                                 Type Decl
  %union                                Union Decl
  @N                                    Action Features
  calc                                  Infix Calc
  else, dangling                        Shift-Reduce
  mfcalc                                Multi-function Calc
  rpcalc                                RPN Calc
  action                                Actions
  action data types                     Action Types
  action features summary               Action Features
  actions in mid-rule                   Mid-Rule Actions
  actions, semantic                     Semantic Actions
  additional C code section             C Code
  algorithm of parser                   Algorithm
```

associativity	Why Precedence
Backus-Naur form	Language and Grammar
Bison declaration summary	Decl Summary
Bison declarations	Declarations
Bison declarations (introduction)	Bison Declarations
Bison grammar	Grammar in Bison
Bison invocation	Invocation
Bison parser	Bison Parser
Bison parser algorithm	Algorithm
Bison symbols, table of	Table of Symbols
Bison utility	Bison Parser
BNF	Language and Grammar
C code, section for additional	C Code
C declarations section	C Declarations
C-language interface	Interface
calculator, infix notation	Infix Calc
calculator, multi-function	Multi-function Calc
calculator, simple	RPN Calc
character token	Symbols
compiling the parser	Rpcalc Compile
conflicts	Shift-Reduce
conflicts, reduce/reduce	Reduce-Reduce
conflicts, suppressing warnings of	Expect Decl
context-dependent precedence	Contextual Precedence
context-free grammar	Language and Grammar
controlling function	Rpcalc Main
dangling else	Shift-Reduce
data types in actions	Action Types
data types of semantic values	Value Type
debugging	Debugging
declaration summary	Decl Summary
declarations, Bison	Declarations
declarations, Bison (introduction)	Bison Declarations
declarations, C	C Declarations
declaring operator precedence	Precedence Decl
declaring the start symbol	Start Decl
declaring token type names	Token Decl
declaring value types	Union Decl
declaring value types, nonterminals	Type Decl
default action	Actions
default data type	Value Type
default stack limit	Stack Overflow
default start symbol	Start Decl
defining language semantics	Semantics
error	Error Recovery
error recovery	Error Recovery
error recovery, simple	Simple Error Recovery
error reporting function	Error Reporting
error reporting routine	Rpcalc Error
examples, simple	Examples
exercises	Exercises
file format	Grammar Layout
finite-state machine	Parser States
formal grammar	Grammar in Bison
format of grammar file	Grammar Layout
glossary	Glossary
grammar file	Grammar Layout

grammar rule syntax	Rules
grammar rules section	Grammar Rules
grammar, Bison	Grammar in Bison
grammar, context-free	Language and Grammar
grouping, syntactic	Language and Grammar
infix notation calculator	Infix Calc
interface	Interface
introduction	Introduction
invoking Bison	Invocation
invoking Bison under VMS	VMS Invocation
LALR(1)	Mystery Conflicts
language semantics, defining	Semantics
layout of Bison grammar	Grammar Layout
left recursion	Recursion
lexical analyzer	Lexical
lexical analyzer, purpose	Bison Parser
lexical analyzer, writing	Rpcalc Lexer
lexical tie-in	Lexical Tie-ins
literal token	Symbols
look-ahead token	Look-Ahead
LR(1)	Mystery Conflicts
main function in simple example	Rpcalc Main
mid-rule actions	Mid-Rule Actions
multi-function calculator	Multi-function Calc
mutual recursion	Recursion
nonterminal symbol	Symbols
operator precedence	Precedence
operator precedence, declaring	Precedence Decl
options for invoking Bison	Invocation
overflow of parser stack	Stack Overflow
parse error	Error Reporting
parser	Bison Parser
parser stack	Algorithm
parser stack overflow	Stack Overflow
parser state	Parser States
polish notation calculator	RPN Calc
precedence declarations	Precedence Decl
precedence of operators	Precedence
precedence, context-dependent	Contextual Precedence
precedence, unary operator	Contextual Precedence
preventing warnings about conflicts	Expect Decl
pure parser	Pure Decl
recovery from errors	Error Recovery
recursive rule	Recursion
reduce/reduce conflict	Reduce-Reduce
reduction	Algorithm
reentrant parser	Pure Decl
reverse polish notation	RPN Calc
right recursion	Recursion
rule syntax	Rules
rules section for grammar	Grammar Rules
running Bison (introduction)	Rpcalc Gen
semantic actions	Semantic Actions
semantic value	Semantic Values
semantic value type	Value Type
shift/reduce conflicts	Shift-Reduce
shifting	Algorithm

simple examples	Examples
single-character literal	Symbols
stack overflow	Stack Overflow
stack, parser	Algorithm
stages in using Bison	Stages
start symbol	Language and Grammar
start symbol, declaring	Start Decl
state (of parser)	Parser States
summary, action features	Action Features
summary, Bison declaration	Decl Summary
suppressing conflict warnings	Expect Decl
symbol	Symbols
symbol table example	Mfcalc Symtab
symbols (abstract)	Language and Grammar
symbols in Bison, table of	Table of Symbols
syntactic grouping	Language and Grammar
syntax error	Error Reporting
syntax of grammar rules	Rules
terminal symbol	Symbols
token	Language and Grammar
token type	Symbols
token type names, declaring	Token Decl
tracing the parser	Debugging
unary operator precedence	Contextual Precedence
using Bison	Stages
value type, semantic	Value Type
value types, declaring	Union Decl
value types, nonterminals, declaring	Type Decl
value, semantic	Semantic Values
VMS	VMS Invocation
warnings, preventing	Expect Decl
writing a lexical analyzer	Rpcalc Lexer
YYABORT	Parser Function
YYACCEPT	Parser Function
YYBACKUP	Action Features
yychar	Look-Ahead
yyclearin	Error Recovery
YYDEBUG	Debugging
yydebug	Debugging
YYEMPTY	Action Features
yyerrok	Error Recovery
YYERROR	Action Features
yyerror	Error Reporting
YYERROR_VERBOSE	Error Reporting
YYINITDEPTH	Stack Overflow
yylex	Lexical
yylloc	Token Positions
YYLTYPE	Token Positions
yylval	Token Values
YYMAXDEPTH	Stack Overflow
yynerrs	Error Reporting
yyparse	Parser Function
YYPRINT	Debugging
YYRECOVERING	Error Recovery
|	Rules