- Table of Contents
- Next Chapter

Chapter 1: Introduction

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

Bisonc++ is highly comparable to the program bison++, written by Alain Coetmeur: all properly-written bison++ grammars ought to be convertible to **bisonc++** grammars after very little or no change. Anyone familiar with bison++ or its precursor, bison, should be able to use **bisonc++** with little trouble. You need to be fluent in using the **C++** programming in order to use **bisonc++** or to understand this manual.

This manual closely resembles **bison**(1)'s userguide. In fact, many sections of that manual were copied straight into this manual. With **bisonc++** distributions (both the full source distribution and the binary `.deb` distributions) **bison**'s orginal manual is included in both *PostScript* and (converted from the `texi` format) `HTML` format. Where necessary sections of the original manual were adapted to **bisonc++**'s characteristics. Some sections were removed, some new sections were added to the current manual. Expect upgrades of the manual to appear without further notice. Upgrades will be announced in the manual's title.

The current manual starts with tutorial chapters that explain the basic concepts of using **bisonc++** and show three explained examples, each building on its previous (where available). If you don't know **bisonc++**, bison++ or bison, start by reading these chapters. Reference chapters follow which describe specific aspects of the program **bisonc++** in detail.

Bisonc++ was designed and built by Frank B. Brokken. The program's initial release was constructed between November 2004 and May 2005.

- Table of Contents
- Next Chapter

- Table of Contents
- Previous Chapter
- Next Chapter

Chapter 2: Conditions for Using Bisonc++

Bisonc++ may be used according to the *GNU General Public License* (GPL). In short, this implies that everybody is allowed to use `b()` and its generated software in any program he/she is developing.

2.1: The `GNU General Public License' (GPL)

The text of the *GNU General Public License* (GPL) is frequently found in files named `COPYING`. On *Debian* systems the GPL may be found in the file `/usr/share/common-licenses/GPL`.

The GPL is shown below:

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The GNU General Public License is a free, copyleft license for
software and other kinds of works.

  The licenses for most software and other practical works are designed
to take away your freedom to share and change the works.  By contrast,
the GNU General Public License is intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users.  We, the Free Software Foundation, use the
GNU General Public License for most of our software; it applies also to
any other work released this way by its authors.  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
them 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 prevent others from denying you
these rights or asking you to surrender the rights.  Therefore, you have
certain responsibilities if you distribute copies of the software, or if
you modify it: responsibilities to respect the freedom of others.

  For example, if you distribute copies of such a program, whether
```

gratis or for a fee, you must pass on to the recipients the same freedoms that you received. 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.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

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

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey 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; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If 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 convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU 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 that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software

Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

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.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS 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.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

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 state 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) <year>  <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 3 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, see <http://www.gnu.org/licenses/>.

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

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

    <program>  Copyright (C) <year>  <name of author>
    This program 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, your program's commands
might be different; for a GUI interface, you would use an "about box".

  You should also get your employer (if you work as a programmer) or school,
if any, to sign a "copyright disclaimer" for the program, if necessary.
For more information on this, and how to apply and follow the GNU GPL, see
<http://www.gnu.org/licenses/>.

  The GNU 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 Lesser General
Public License instead of this License.  But first, please read
<http://www.gnu.org/philosophy/why-not-lgpl.html>.
```

- Table of Contents
- Previous Chapter
- Next Chapter

- Table of Contents
- Previous Chapter
- Next Chapter

Chapter 3: Bisonc++ concepts

This chapter introduces many of the basic concepts without which the details of **bisonc**++ will not make sense. If you do not already know how to use **bisonc**++, bison++ or bison, it is advised to start by reading this chapter carefully.

3.1: Languages and Context-Free Grammars

In order for **bisonc**++ 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 **bisonc**++ is essentially machine-readable BNF.

Not all context-free languages can be handled by **bisonc**++, 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 section 7.6 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 square(int x)   // keyword `int', identifier, open-paren,
                        // keyword `int', identifier, close-paren
    {                   // 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 **bisonc++** 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.

3.2: From Formal Rules to Bisonc++ Input

A formal grammar is a mathematical construct. To define the language for **Bisonc++**, you must write a file expressing the grammar in **bisonc++** syntax: a ***Bisonc++** grammar file*. See chapter 5.

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

The **bisonc++** 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 section 5.2, which also describes the current restrictions on the names of terminal 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 **bisonc++** syntax. For example, here is the **bisonc++** 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 **bisonc++** punctuation used in every rule.

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

See section 5.3.

3.3: 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 **bisonc++** grammar has both a token type and a *semantic value*. See section 5.6 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.

3.4: 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 **bisonc**++ 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 section 5.6.4.

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.

3.5: Bisonc++ output: the Parser class

When you run **bisonc**++, you give it a **bisonc**++ grammar file as input. The output, however, defines a **C**++ *class*, in which several *members* have already been defined. Therefore, the *output* of **bisonc**++ consists of *header files* and a **C**++ source file, defining a member (`parse()`) that parses the language described by the grammar. The class and its implementation is called a **bisonc**++ *parser class*. Keep in mind that the **Bisonc**++ utility and the **bisonc**++ parser class are two distinct pieces of software: the **bisonc**++ utility is a program whose output is the **bisonc**++ parser class that becomes part of your program.

More specifically, **bisonc**++ generates the following files from a **bisonc**++ grammar file:

- A *baseclass header*, which can be included by *lexical scanners* (see below), primarily defining the *lexical tokens* that the parser expects the lexical scanner to return;
- A *class header*, defining the **bisonc**++ parser class interface;
- An *implementation header*, which is used to declare all entities which are *only* used by **bisonc**++'s parser class *implementation* (and not required by the remaining parts of your program);
- The *parsing member*, actually performing the parsing of a provided input according to the rules of the defined **bisonc**++ grammar (that you, as **bisonc**++'s user, defined).

The job of the **bisonc**++ parsing member 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.

In **C**++ the tokens should be produced by an object called the *lexical analyzer* or *lexical scanner* that you must supply in some fashion (such as by writing it in **C**++). The **bisonc**++ parsing member requests the next token from the lexical analyzer each time it wants a new token. The parser itself 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 **bisonc**++ does not depend on this. See section 6.3.1.

The **bisonc**++ parsing function is **C**++ code defining a member function named `parse()` which implements that grammar. This parsing function nor the parser object for which it is called does make a complete **C**++ program: you must supply some additional details. One `detail' to be supplied is is the lexical analyzer. The parser class itself declares several more members which must be defined when used. One of these additional members is an error-reporting function which the parser calls to report an error. Simple default, yet sensible, implementations for these additional members may be generated by **bisonc**++. Having constructed a parser class and a lexical scanner class, *objects* of these classes must be defined in a complete **C**++ program. Usually such objects are defined in a function called `main()`; you have to provide this, and arrange for it to call the parser's `parse()` function, or the parser will never run. See chapter 6.

Note that, different from conventions used by Bison and Bison++, there is no special name convention requirement anymore imposed by **bisonc**++. In particular, there is *no* need to begin all variable and function names used in the **bisonc**++ parser with `yy' or `YY' anymore. However, some name restrictions on symbolic tokens exist. See section 5.5.24.1 for details.

3.5.1: Bisonc++: an optionally reentrant Parser

A computer program or routine is described as reentrant if it can be safely called recursively and concurrently from multiple processes. To be reentrant, a function must hold no static data, must not return a pointer to static data, must work only on the data provided to it by the caller, and must not call non-reentrant functions (Source: http://en.wikipedia.org/wiki/Reentrant).

Currently, **bisonc**++ generates a parsing member which may or may not be reentrant, depending on the specification of the --thread-safe.

The source file generated by **bisonc**++ containing the parsing member function not only contains this function, but also various tables (e.g., state transition tables) defined in the anonymous name space. When the option `--thread-safe` is provided, these tables are `const` tables: their elements are not changed by the parsing function and so the parsing function, as it only manipulates its own local data, becomes reentrant.

3.6: Stages in Using Bisonc++

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

- Formally specify the grammar in a form recognized by **bisonc**++ (see chapter 5). 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.
- Run **bisonc**++ on the grammar to produce the parser class and parsing member function.
- Write a lexical scanner to process input and pass tokens to the parser. The lexical scanner may be written by hand in **C**++ (see section 6.3.1) (it could also be produced using, e.g., **flex**(1), but the use of **flex**(1) is not discussed in this manual).
- All the parser's members (except for the member `parse()`) and its support functions must be implemented by the programmer. Of course, additional member functions should also be declared in the parser class' header. At the very least the member `int lex()` calling the lexecal scanner to produce the next available token *must* be implemented (although a standardized implementation can also be generated by **bisonc**++). The member `lex()` is called by `parse()` (support functions) to obtain the next available token. The member function `void error(char const *msg)` may also be re-implemented by the programmer, but a basic in-line implementation is provided by default. The member function `error()` is called when `parse()` detects (syntactic) errors.
- The parser can now be used in a program. A very simple example would be:

```
    int main()
    {
        Parser parser;
        return parser.parse();
    }
```

Once the software has been implemented, the following steps are required to create the final program:

- Compile the parsing function generated by **bisonc**++, as well as any other source files you have implemented.
- Link the object files to produce the final program.

3.7: The Overall Layout of a Bisonc++ Grammar File

The input file for the **bisonc**++ utility is a **bisonc**++ grammar file. Different from Bison++

and Bison grammar files, **bisonc**++ grammar file consist of only two sections. The general form of a **bisonc**++ grammar file is as follows:

```
Bisonc++ directives
%%
Grammar rules
```

Readers familiar with Bison will note that there is no *C declaration section* and no section to define *Additional C code*. With **bisonc**++ these sections are superfluous since, due to the fact that a **bisonc**++ parser is a class, all additional code required for the parser's implementation can be incorporated into the parser class itself. Also, **C**++ classes normally only require declarations that can be defined in the classes' header files, so also the `additional C code' section could be omitted from the **bisonc**++ grammar file.

The `%%' is a punctuation that appears in every **bisonc**++ grammar file to separate the two sections.

The **bisonc**++ directives section is used to 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. Furthermore, this section is also used to specify **bisonc**++ directives. These **bisonc**++ directives are used to define, e.g., the name of the generated parser class and a namespace in which the parser class will be defined.

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

One special directive is availble that may be used in the directives section and in the grammar rules section. This directive is `%include`, allowing you to split long grammar specification files into smaller, more comprehensible and accessible chunks. The `%include` directive is discussed in more detail in section 5.5.7.

- Table of Contents
- Previous Chapter
- Next Chapter

- Table of Contents
- Previous Chapter
- Next Chapter

Chapter 4: Examples

Now we show and explain three sample programs written using **bisonc++**: a reverse polish notation calculator, an algebraic (infix) notation calculator, and a multi-function calculator. All three have been tested under Linux (kernel 2.4.24 and above); each produces a usable, though limited, interactive desk-top calculator.

These examples are simple, but **bisonc++** grammars for real programming languages are written the same way. You can copy these examples from this dument into source files to try them yourself. Also, the **bisonc++** package contains the various source files ready for use.

- Reverse Polish Notation Calculator (section 4.1): A first example of a calculator not requiring any operator precedence.
- Infix Notation Calculator (section 4.2): Infix (algebraic) notation calculator, introducing. operator precedence.
- Simple Error Recovery (section 4.3): How to continue after syntactic errors.
- Multi-Function Calculator (section 4.4): Calculator having memory and trigonometrical functions. It uses multiple data-types for semantic values.
- Suggested Exercises (section 4.5): Ideas for improving the multi-function calculator.

4.1: rpn: a 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.

All sources for this calculator are found in the examples/rpn/ directory.

4.1.1: Declarations for the `rpn' calculator

Here are the **C++** and **bisonc++** directives for the reverse polish notation calculator. As in **C++**, end-of-line comments may be used.

```
%baseclass-preinclude cmath

%token NUM
%stype double
```

The directive section provides information to **bisonc++** about the token types (see section The **bisonc++** 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. Since **bisonc++** uses the type `int` as the default semantic value type, one additional directive is required to inform **bisonc++** about the fact that we will be using `double` values. The `%stype` (semantic value type) directive is used to do so.

4.1.2: Grammar rules for the `rpn' calculator

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

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

line:
        '\n'
|
        exp '\n'
        {
            std::cout << "\t" << $1 << std::endl;
        }
;

exp:
        NUM
|
        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 rpn `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 the logical *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 section 5.6.4.

You must specify these actions in **C++**, but **bisonc++** provides the means for passing semantic values between the rules. In each action, the pseudo-variable $$ represents the semantic value for the grouping that the rule is going to construct. Assigning a value to $$ is the main job of most actions. The semantic values of the components of the rule are referred to as $1 (the first component of a rule), $2 (the second component), and so on.

4.1.2.1: 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 section 5.4.

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's parsing function (parse()) 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.

4.1.2.2: Explanation of `line'

Now consider the definition of line:

```
    line:
            '\n'
    |
            exp '\n'
            {
                cout << "\t" << $1 << endl;
            }
    ;
```

The first alternative is a token which is a newline character; this means that rpn 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 rpn useful. The semantic value of the exp grouping is the value of $1 because the exp in question is the first symbol in the rule's 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 `rpn` has printed the value of the user's input line, that value is no longer needed.

4.1.2.3: 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;
            }
    ...
```

It is customary to use `|' to join all the rules for exp, but the rules could equally well have been written 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 values 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 the parser's parsing function `parse()` recognizes a sum expression using this rule, the sum of the two subexpressions' values is produced as the value of the entire expression. See section 5.6.4.

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.

4.1.3: The Lexical Scanner used by `rpn'

The lexical analyzer's job is low-level parsing: converting characters or sequences of characters into tokens. The **bisonc++** parser gets its tokens by calling the lexical analyzer `lex()`, which is a predeclared member of the parser class. See section 6.3.1.

Only a simple lexical analyzer is needed for `rpn`. 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 **bisonc++** 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 **bisonc++** as a **C++** enumeration value. In this example, therefore, `NUM` becomes an enumeration value for `lex()` to return.

The semantic value of the token (if it has one) is stored into the parser's data member `d_val` (comparable to the variable `yylval` used by, e.g., Bison). This data member has `int` as its default type, but by specifying `%stype` in the directive section this default type can be modified (to, e.g., `double`).

A token value of zero is returned once end-of-file is encountered. (**Bisonc++** recognizes any nonpositive value as indicating the end of the input).

Here is the lexical scanner's implementation:

```
#include "Parser.ih"

/*
    Lexical scanner 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.
*/

int Parser::lex()
{
    char c;
                                    // get the next non-ws character
    while (std::cin.get(c) && c == ' ' || c == '\t')
        ;

    if (!std::cin)                   // no characters were obtained
        return 0;                   // indicate End Of Input

    if (c == '.' || isdigit (c))    // if a digit char was found
    {
        std::cin.putback(c);        // return the character
        std::cin >> d_val;          // extract a number
        return NUM;                 // return the NUM token
    }

    return c;                       // otherwise return the extracted char.
}
```

4.1.4: The Controlling Function `main()'

In keeping with the spirit of this example, the controlling function `main()` is kept to the bare minimum. The only requirement is that it constructs a parser object and then calls its parsing function `parse()` to start the process of parsing:

```
/*
                              rpn.cc
*/

#include "rpn.h"

int main()
{
    Parser parser;

    parser.parse();
    return 0;
}
```

4.1.5: The error reporting member `error()'

When `parse()` detects a *syntax error*, it calls the error reporting member function `error()` to print an error message (usually but not always *parse error*). It is up to the programmer to supply an implementation, but a very bland and simple in-line implementation is provided by **bisonc++** in the class header file (see chapter 6). This default implementation is acceptable for `rpn`.

Once `error()` returns, the **bisonc++** parser may recover from the error and continue parsing if the grammar contains a suitable error rule (see chapter 8). Otherwise, the parsing function `parse()` returns nonzero. Not any error rules were included 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 for this first example.

4.1.6: Running Bisonc++ to generate the Parser

Before running **bisonc++** to produce a parser class, we need to decide how to arrange all the source code in one or more source files. Even though the example is fairly simple, all user-defined functions should be defined in source files of their own. For `rpn` this means that a source file `rpn.cc` is constructed holding `main()`, and a file `parser/lex.cc` holding the lexical scanner's implementation. Note that I've put all the parser's files in a separate directory as well (also see section 3.7).

In rpn's parser directory the file `grammar` holds the grammar specification. **Bisonc++** will construct a parser class and a parsing member function from this file after issuing the following command:

```
    b() grammar
```

From this, **bisonc++** produced the following files:

- `Parser.h`, the parser class definition;
- `Parserbase.h`, the parser's *base* class definition, defining, among other, the grammatical tokens to be used by externally defined lexical scanners;
- `Parser.ih`, the *internal header file*, to be included by all implementations of the parser class' members;
- `parse.cc`, the parsing member function.

By default, `Parserbase.h` and `parse.cc` will be *re-created* each time **bisonc++** is re-run. `Parser.h` and `Parser.ih` may safely be modified by the programmer, e.g., to add new members to to the parser class. These two files will not be overwritten by **bisonc++**, unless explicitly instructed to do so.

4.1.7: Constructing and running `rpn'

Here is how to compile and run the parser file:

```
    # List files (recursively) in the (current) examples/rpn  directory.
    % ls -R
    .:
    build  parser  rpn.cc  rpn.h

    ./parser:
    grammar  lex.cc

    # Create `rpn' using the `build' script:
    % ./build rpn

    # List files again, ./rpn is the constructed program
    % ls -R
    .:
    build  parser  rpn  rpn.cc  rpn.h

    ./parser:
    Parser.h  Parser.ih  Parserbase.h  grammar  lex.cc  parse.cc  parse.output
```

Here is an example session using rpn:

```
    % rpn
    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 +
            -3.16667
    3 4 ^                             Exponentiation
            81
    ^D                                End-of-file indicator
    %
```

4.2: `calc': an Infix Notation Calculator

We now modify rpn 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 **bisonc++** grammar specification for calc, an infix desk-top calculator:

```
%baseclass-preinclude   cmath
%stype double

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

%%

input:
        // empty
|
        input line
;

line:
        '\n'
|
        exp '\n'
        {
            std::cout << "\t" << $1 << '\n';
        }
;

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

The functions `lex()`, `error()` and `main()` can be the same as used with `rpn`.

There are two important new features shown in this code.

In the second section (**Bisonc++** directives), `%left` declares token types and says they are left-associative operators. The directives `%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 directives; the higher the line number of the directive (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 section 5.5.8.

The other important new feature is the `%prec` in the grammar section for the unary minus operator. The `%prec` simply instructs **bisonc++** that the rule `| '-' exp' has the same precedence as NEG (in this case the next-to-highest). See section 7.4.

Here is a sample run of `calc`:

```
    % calc
    4 + 4.5 - (34/(8*3+-3))
            6.88095
    -56 + 2
            -54
    3 ^ 2
            9
```

4.3: Basic Error Recovery

Up to this point, this manual has not addressed the issue of error recovery, i.e., how to continue parsing after the parser detects a syntax error. All that's been handled so far is error reporting using the `error()` member function with yyerror. Recall that by default `parse()` returns after calling `error()`. This means that an erroneous input line causes the calculator program to exit. Now we show how to rectify this deficiency.

The **bisonc++** language itself includes the reserved word `error`, which may be included in the grammar rules. In the example below it has been added as one more alternatives for line:

```
line:
        '\n'
|
        exp '\n'
        {
            std::cout << "\t" << $1 << '\n';
        }
|
        error '\n'
;
```

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 `error()` member function is still called upon to print its message as well). Different from the implementation of `error` in Bison, **bisonc++** proceeds on the assumption that whenever `error` is used in a rule it is the grammar constructor's intention to have the parser continue parsing. Therefore, a statement like `yyerrok;' seen in Bison grammars is superfluous in **bisonc++** grammars. The reserved keyword `error` itself will cause the parsing function to skip all subsequent input until a possible token following `error` is seen. In the above implementation that token would be the newline character `\n' (see chapter 8).

This form of error recovery deals with syntax errors. There are other kinds of errors; for example, divisions by zero, which raises an exception signal that is normally fatal. A real calculator program must handle this signal and use whatever it takes to discard the rest of the current line of input and resume parsing thereafter. This extensive error handling is not discussed here, as it is not specific to **bisonc++** programs.

4.4: `mfcalc': a Multi-Function Calculator

Now that the basics of **bisonc++** 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 parser's member `lex()` 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 syntaxis is as follows:

```
    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:

```
        pi = 3.141592653589
                3.14159
        sin(pi)
                7.93266e-13
        alpha = beta1 = 2.3
                2.3
        alpha
                2.3
        ln(alpha)
                0.832909
        exp(ln(beta1))
                2.3
```

Note that multiple assignment and nested function calls are permitted.

4.4.1: The Declaration Section for `mfcalc'

The grammar specification file for the `mfcalc` calculator allows us to introduce several new features. Here is the **bisonc++** directive section for the `mfcalc` multi-function calculator (line numbers were added for referential purposes, they are not part of the declaraction section as used in the actual grammar file):

```
 1   %union
 2   {
 3       double u_val;
 4       double *u_symbol;
 5       double (*u_fun)(double);
 6   }
 7
 8   %token <u_val>  NUM         // Simple double precision number
 9   %token <u_symbol> VAR       // Variable
10   %token <u_fun>  FNCT        // Function
11   %type  <u_val>  exp
12
13   %right '='
14   %left '-' '+'
15   %left '*' '/'
16   %left NEG                   // negation--unary minus
17   %right '^'                  // exponentiation
```

The above grammar introduces only two new features of the Bison language. These features allow semantic values to have various data types

The `%union` directive given in lines 1 until 6 allow semantic values to have various data types (see section 5.6.2). The `%union` directive is used instead of `%stype`, and defines the type `Parser::STYPE` as the indicated union: all semantic values will now have this `Parser::STYPE` type. As defined here the allowable types are now

- `double` (for `exp` and `NUM`);
- a *pointer* to a `double`, which will be a pointer to entries in `mfcalc`'s symbol table, used with `VAR` tokens (see section 5.5.26).
- a *pointer to a function* expecting a `double` argument and returning a `double` value, used with `FNCT` tokens.

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` (line 12) 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 also section 5.5.25.

Finally note the *right associative* operator `=', defined in line 13: by making the assignment operator right-associative we can allow *sequential assignments* of the form `a = b = c =`

expression.

4.4.2: 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'
        {
            cout << "\t" << $1 << endl;
        }
|
        error '\n'
;

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

4.4.3: The `mfcalc' Symbol- and Function Tables

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 **bisonc++** directives, but it requires some additional **C++** types for support as well as several additional data members for the parser class.

The symbol table itself will vary in size and contents once `mfcalc` is used, and if a program uses multiple parser objects (well...) each parser will require its own symbol table. Therefore it is defined as a *data member* `d_symbols` in the Parser's header file. In contrast, the *function table* has a *fixed* size and contents. Because of this, multiple parser objects (if defined) could share the same function table, and so the function table is defined as a *static* data member. Both tables profitably use the `std::map` container data type that is available in **C++**: their keys are `std::string` objects, their values, respecively, `double`s and `double (*)(double)`s. Here is the declaration of `d_symbols` and `s_functions` as used in `mfcalc`'s parser:

```
    std::map<std::string, double> d_symbols;

    static std::map<std::string, double (*)(double)> s_functions;
```

As `s_functions` is a static member, it can be initialized *compile time* from an *array of pairs*. To ease the definition of such an array a `private typedef`

```
    typedef std::pair<char const *, double (*)(double)> FunctionPair;
```

is added to the parser class, as well as a private array

```
    static FunctionPair s_funTab[];
```

These definitions allow us to initialize `s_functions` in a separate source file (`data.cc`):

```
#include "Parser.ih"

Parser::FunctionPair Parser::s_funTab[] =
{
    FunctionPair("sin",  sin),
    FunctionPair("cos",  cos),
    FunctionPair("atan", atan),
    FunctionPair("ln",   log),
    FunctionPair("exp",  exp),
    FunctionPair("sqrt", sqrt),
};

map<string, double (*)(double)> Parser::s_functions
(
    Parser::s_funTab,
    Parser::s_funTab + sizeof(Parser::s_funTab) / sizeof(Parser::FunctionPair)
);
```

By simply editing the definition of `s_funTab`, additional functions can be added to the calculator.

4.4.4: The revised `lex()' member

In `mfcalc`, the parser's member function `lex()` must now recognize variables, function names, 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 the table in which they are found. By arranging lex()'s logic such that the function table is searched first it is simple to ensure that no variable can ever have the name of a predefined function. The currently implemented approach, in which two different tables are used for the arithmetic functions and the variable symbols is appealing because it's simple to implement. However, it also has the drawback of being difficult to scale to more generic calculators, using, e.g., different data types and different types of functions. In such situations a single symbol table is more preferable, where the keys are the identifiers (variables, function names, predefined constants, etc.) while the values are objects describing their characteristics. A re-implementation of mfcalc using an integrated symbol table is suggested in one of the exercises of the upcoming section 4.5.

The parser's lex() member uses the following approach:

- All leading blanks and tabs are skipped
- If no (other) character could be obtained 0 is returned, indicating End-Of-File.
- If the first non-blank character is a dot or number, a number is extracted from the standard input. Since the semantic value data member of mfcalc's parser (d_val) is itself also a union, the numerical value can be extracted into d_val.u_val, and a NUM token can be returned.
- If the first non-blank character is not a letter, then a single-character token was received and the character's value is returned as the next token.
- Otherwise the read character is a letter. This character and all subsequent alpha-numeric characters are extracted to construct the name of an identifier. Then this identifier is searched for in the s_functions map. If found, d_val.u_fun is given the function's address, found as the value of the s_functions map element corresponding to the read identifier, and token FNCT is returned. If the symbol is not found in s_functions the address of the value ofn d_symbols associated with the received identifier is assigned to d_val.u_symbol and token VAR is returned. Note that this automatically defines newly used variables, since d_symbols[name] automatically inserts a new element in a map if d_symbol[name] wasn't already there.

Here is mfcalc's parser's lex() member function:

```
#include "Parser.ih"

/*
    Lexical scanner 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.
*/

int Parser::lex()
{
    char c;
```

```
                                    // get the next non-ws character
    while (cin.get(c) && c == ' ' || c == '\t')
        ;

    if (!cin)                   // no characters were obtained
        return 0;                   // indicate End Of Input

    if (c == '.' || isdigit (c))    // if a digit char was found
    {
        cin.putback(c);        // return the character
        cin >> d_val.u_val;    // extract a number
        return NUM;                 // return the NUM token
    }

    if (!isalpha(c))                // c doesn't start an identifier:
        return c;                   // return a single character token.

    // in all other cases, an ident is entered. Recognize a var or function

    string word;           // store the name in a string object

    while (true)                // process all alphanumerics:
    {
        word += c;              // add 'm to `word'
        if (!cin.get(c))   // no more chars? then done here
            break;

        if (!isalnum(c))        // not an alphanumeric: put it back and done.
        {
            cin.putback(c);
            break;
        }
    }
                                // Now lookup the name as a function's name
    map<string, double (*)(double)>::iterator function =
                                                s_functions.find(word);

                                // Got it, so return FPTR
    if (function != s_functions.end())
    {
        d_val.u_fun = function->second;
        return FNCT;
    }
                                // no function, so return a VAR. Set
                                // u_symbol to the symbol's address in the
                                // d_symbol map. The map will add the
                                // symbol if not yet there.
    d_val.u_symbol = &d_symbols[word];
    return VAR;
}
```

4.4.5: Constructing `mfcalc'

In order to construct mfcalc, the following steps are suggested:

- Construct a program mfcalc.cc. Actually, it is already available, since all implementations of main() used so far are identical to each other.
- Construct the parser in a subdirectory parser:
  - First, construct **bisonc++**'s input file as indicated above. Name this file grammar;
  - Run bisonc++ grammar to produce the files Parser.h, Parserbase.h, Parser.ih and parse.cc;
  - Modify Parser.h so as to include FunctionPair, s_functions, s_funTab and d_symbols;
  - Modify Parser.ih so as to include cmath and optionally `using namespace std', which is commented out by default;
  - Implement data.cc and lex.cc to initialize the static data and to contain the lexical scanner, respectively.
- Now construct mfcalc in mfcalc.cc's directory using the following command:

```
    g++ -o mfcalc *.cc parser/*.cc
```

4.5: Exercises

Here are some suggestions for you to consider to improve mfcalc's implementation and operating mode:

- Add some additional functions from `cmath' to the Parser::s_functions;
- Define a class Symbol in which the symbol type, and an appropriate value for the symbol is stored. Define only one map d_symbols in the Parser, and provide the Symbol class with means to obtain the appropriate values for the various token types.
- Remove the %union directive, and change it into %stype Symbol. Hint: use the %preinclude-header directive to make Symbol known to the parser's base class.
- Define a token CONST for numerical constants (like PI, (E)), and pre-define some numerical constants;
- Make the program report an error if the user refers to an uninitialized variable in any way except to store a value in it. Hints: use a get() and set() member pair in Symbol, and use the appropriate member in the appropriate expr rule; use ERROR() to initiate error recovery.

- Table of Contents
- Previous Chapter

- Next Chapter

- Table of Contents
- Previous Chapter
- Next Chapter

Chapter 5: Bisonc++ grammar files

Bisonc++ takes as input a context-free grammar specification and produces a **C++** class offering various predefined members, among which the member `parse()`, that recognizes correct instances of the grammar.

In this chapter the organization and specification of such a grammar file is discussed in detail.

Having read this chapter you should be able to define a grammar for which **Bisonc++** can generate a class, including a member that will recognize correctly formulated (in terms of your grammar) input, using all the features and facilities offered by **bisonc++** to specify a grammar. In principle this grammar will be in the class of **LALR(1)** grammars (see, e.g., *Aho, Sethi & Ullman*, 2003 (Addison-Wesley)).

5.1: Outline of a Bisonc++ Grammar File

The input file for the **bisonc++** utility is a **bisonc++** grammar file. Different from Bison++ and Bison grammar files, **bisonc++** grammar file consist of only two sections. The general form of a **bisonc++** grammar file is as follows:

```
    Bisonc++ directives
    %%
    Grammar rules
```

Readers familiar with Bison will note that there is no *C declaration section* and no section to define *Additional C code*. With **bisonc++** these sections are superfluous since, due to the fact that a **bisonc++** parser is a class, all additional code required for the parser's implementation can be incorporated into the parser class itself. Also, **C++** classes normally only require declarations that can be defined in the classes' header files, so also the `additional C code' section could be omitted from the **Bisonc++** grammar file.

The `%%' is a punctuation that appears in every **bisonc++** grammar file to separate the two sections.

The **bisonc++** directives section is used to 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. Furthermore, this section is also used to specify **bisonc++** directives. These **bisonc++** directives are used to define, e.g., the name of the generated parser class and a namespace in which the parser class will be defined. All **bisonc++** directives are covered in section 5.5.

The grammar rules define how to construct *nonterminal symbols* from their parts. The grammar rules section contains one or more **bisonc++** grammar rules, and nothing else. See section 5.3, covering the 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.

Bisonc++'s grammar file may be split into several files. Each file may be given a suggestive name. This allows quick identification of where a particular section or rule is found, and improves readability of the designed grammar. The %include-directive (see section 5.5.7) can be used to include a partial grammar specification file into another specification file.

5.2: Symbols, Terminal and Nonterminal Symbols

Symbols in **bisonc++** grammars represent the grammatical classifications of the language.

A *terminal symbol* (also known as a *token type*) represents a class of syntacticly 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 **Bisonc++** parser by a numeric code, and the parser's `lex()` member 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), and underscores. **Bisonc++** currently does not support periods in symbol names (Users familiar with Bison may observe that Bison *does* support periods in symbol names, but the Bison user guide remarks that `Periods make sense only in nonterminals'. Even so, it appears that periods in symbols are hardly ever used).

There are two ways to write 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 **bisonc++** directive such as `%token`. See section 5.5.24.
- A `character token type` (or `literal character 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 section 5.6.1), associativity, or precedence (see section 5.5.8).

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 **bisonc++** as well, but you must not use the `0` character as a character literal because its ASCII code, zero, is the code `lex()` must return for end-of-input (see section 6.3.1). If your program *must* be able to return 0-byte characters, define a special token (e.g., `ZERO_BYTE`) and return that token instead.

Note that *literal string tokens*, formally supported in Bison, is *not* supported by **bisonc++**. Again, such tokens are hardly ever encountered, and the dominant lexical scanner generators (like **flex**(1)) do not support them. Common practice is to define a symbolic name for a literal string token. So, a token like `EQ` may be defined in the grammar file, with the lexical scanner returning `EQ` when it matches `==`.

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 the `lex()` member 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 `lex()` can use the identical character constant to generate the requisite code. Each named token type becomes a **C++** enumeration value in the parser base-class header file, so `lex()` can use the corresponding enumeration identifiers. When using an externally (to the parser) defined lexical scanner, the lexical scanner should include the parser's base class header file, returning the required enumeration identifiers as defined in the parser class. So, if (%token NUM) is defined in the parser class `Parser`, then the externally defined lexical scanner may return `Parser::NUM`.

The symbol ``error'` is a *terminal* symbol reserved for error recovery (see chapter 8). The `error` symbol should not be used for any other purpose. In particular, the parser's member function `lex()` should never return this value. Several other identifiers should not be used as terminal symbols. See section 5.5.24.1 for a description.

5.3: Syntax of Grammar Rules

A **bisonc++** 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 section 5.2). With respect to the way rules are defined, note the following:

- The construction:

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

  means 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 section 5.6.4.

- 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.

- Alternatively, multiple rules of the same nonterminal can be defined. E.g., the previous definition of `result:` could also have been defined as:

```
    result:
        rule1-components
        ...
    ;

    result:
        rule2-components...
        ...
    ;
```

However, this is a potentially dangerous practice, since one of the two `result` rules could also have used a misspelled rule-name (e.g., the second `result`) should have been `results`. Therefore, **bisonc++** will generate a warning if the same nonterminal is used repeatedly when defining production rules.

- If *components* in a rule is *empty*, it means that *result* can match the empty string. Such a alternative is called an *empty production rule*. For example, here is how to define a comma-separated sequence of zero or more `exp` groupings:

```
    expseq:
        expseq1
    |
        // empty
    ;

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

Convention calls for a comment `` `// empty' `` in each empty production rule.

5.4: Writing recursive rules

A rule is called *recursive* when its *result* nonterminal appears also on its right hand side. Nearly all **bisonc++** 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:
        expseq1 ',' exp
|
        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 ',' expseq1
|
        exp
;
```

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 **bisonc++** 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 chapter 7 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.

5.5: Bisonc++ Directives

The **bisonc++** declarations section of a **bisonc++** grammar defines the symbols used in formulating the grammar and the data types of semantic values. See section 5.2.

All token type names (but not single-character literal tokens such as '+' and '*') must be declared. If you need to specify which data type to use for the semantic value (see section 5.6.2) of nonterminal symbols, these symbols must be declared as well.

The first rule in the file by default specifies the *start symbol*. If you want some other symbol to be the start symbol, you must use an explicit `%start` directive (see section 3.1).

In this section all **bisonc++** declarations will be discussed. Some of the declarations have already been mentioned, but several more are available. Some declarations define how the grammar will parse its input (like `%left, %right`); other declarations are available, defining, e.g., the name of the parsing function (by default `parse()`), or the name(s) of the files generated by **bisonc++**.

In particular readers familiar with Bison (or Bison++) should read this section thoroughly, since **bisonc++**'s directives are more extensive and different from the `declarations' offered by Bison, and the macros offered by Bison++.

Several directives expect file- or path-name arguments. File- or path-names must be specified on the same line as the directive itself, and they start at the first non-blank character following the directive. File- or path-names may contain escape sequences (e.g., if you must: use ``\ '` to include a blank in a filename) and continue until the first blank character thereafter. Alternatively, file- or path-names may be surrounded by double quotes (`"..."`) or pointed brackets (`<...>`). Pointed brackets surrounding file- or path-names merely function to delimit filenames. They do not refer to, e.g., **C++**'s include path. No escape sequences are required for blanks within delimited file- or path-names.

Directives accepting a `filename' do not accept path names, i.e., they cannot contain directory separators (`/`); options accepting a 'pathname' may contain directory separators.

Sometimes directives have analogous command-line options. In those cases command-line options take priority over directives.

Some directives may generate warnings. This happens when an directive conflicts with the contents of a file which **bisonc++** cannot modify (e.g., a parser class header file exists, but doesn't define a name space, but a `%namespace` directive was provided). In those cases the directive is ignored, and hand-editing may then be required to effectuate the directive.

5.5.1: %baseclass-preinclude: specifying a header included by the baseclass

Syntax: **%baseclass-preinclude** `pathname`
`Pathname` defines the path to the file preincluded in the parser's base-class header. See the description of the --baseclass-preinclude option for details about this directive. By default `filename' is surrounded by double quotes; it's OK, however, to provide them yourself. When the argument is surrounded by *pointed brackets* `#include <header>` is used.

5.5.2: %class-name: defining the name of the parser class

Syntax: **%class-name** `parser-class-name`
By default, **bisonc++** will generate a parser-class by the name of `Parser`. The default can be changed using this directive which defines the name of the **C++** class that will be generated. It may be defined only once and `parser-class-name` must be a **C++** identifier.

If you're familiar with the Bison++ program, please note:

- This directive replaces the **%name** directive previously used by Bison++.
- Contrary to Bison++'s **%name** directive, **%class-name** may appear anywhere in the directive section of the grammar specification file.

A warning is issued if this directive is used and an already existing parser-class header file does not define `class `className'` and/or if an already existing implementation header file does not define members of the class `` `className'``.

5.5.3: %debug: adding debugging code to the `parse()' member

Syntax: **%debug**
Provide `parse()` and its support functions with debugging code, showing the actual parsing process on the standard output stream. When included, the debugging output is active by default, but its activity may be controlled using the **setDebug(bool on-off)** member. Note that no `#ifdef DEBUG` macros are used anymore. Rerun **bisonc++** without the **--debug** option to generate an equivalent parser not containing the debugging code.

5.5.4: %error-verbose: dumping the parser's state stack

Syntax: **%error-verbose**
The parser's state stack is dumped to the standard error stream when an error is detected by the `parse()` member function. Following a call of the `error()` function, the stack is dumped from the top of the stack (highest offset) down to its bottom (offset 0). Each stack element is prefixed by the stack element's index.

5.5.5: %expect: suppressing conflict warnings

Syntax: **%expect** `number`
Bisonc++ normally warns if there are any conflicts in the grammar (see section 7.2), but many 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 argument `number` is a decimal integer. The declaration says there should be no warning if there are `number` 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. **Bisonc++** will also print the number of conflicts.
- Check each of the conflicts to make sure that **bisonc++**'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 of (shift-reduce) conflict printed by **bisonc++**.

Now **bisonc++** will stop annoying you about the conflicts you have checked, but it will warn you again if changes in the grammar result in another number or type of conflicts.

5.5.6: %flex: using the traditional `flex++' interface

Syntax: **%flex**
When provided, the scanner matched text function is called as `d_scanner.YYText()`, and the scanner token function is called as `d_scanner.yylex()`. This directive is only interpreted if the `%scanner` directive is also provided.

5.5.7: %include: splitting the input file

Syntax: **%include** `pathname`
This directive is used to switch to `pathname` while processing a grammar specification.

Unless pathname defines an absolute file-path, pathname is searched relative to the location of **bisonc++**'s main grammar specification file (i.e., the grammar file that was specified as **bisonc++**'s command-line option). This directive can be used to split long grammar specification files in shorter, meaningful units. After processing pathname processing continues beyond the %include pathname directive.

Bisonc++'s main grammar specification file could be:

```
    %include spec/declarations.gr
    %%
    %include spec/rules.gr
```

where spec/declarations.gr contains declarations and spec/rules.gr contains the rules. Each of the files included using %include may itself use %include directives (which are then processed relative to their locations). The default nesting limit for %include directives is 10, but the option --max-inclusion-depth can be used to change this default.

%include directives should be specified on a line of their own.

5.5.8: %left, %right, %nonassoc: defining operator precedence

Syntax:

> **%left** [<type>] terminal(s)
> **%nonassoc** [<type>] terminal(s)
> **%right** [<type>] terminal(s)

These directives are called *precedence directives* (see also section 5.5.8 for general information on operator precedence).

The %left, %right or %nonassoc directive are used to declare tokens and to specify their precedence and associativity, all at once.

- 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 not a defined operation, and could be considered an error.
- The precedence of an operator determines how it nests with other operators. All the tokens declared in a single precedence directive have equal precedence and nest together according to their associativity. When two tokens declared in different precedence directives associate, the one declared *later* has the higher precedence and is grouped

first.

The <type> specification is optional, and specifies the type of the semantic value when a token specified to the right of a <type> specification is received. The pointed arrows are part of the type specification; the type itself must be a field of a %union specification (see section 5.5.26).

When multiple tokens are listed they must be separated by whitespace or by commas. Note that the precedence directives also serve to define token names: symbolic tokens mentioned with these directives should not be defined using %token directives.

5.5.9: %locationstruct: specifying a dedicated location struct

Syntax: **%locationstruct** struct-definition
Defines the organization of the location-struct data type **LTYPE__**. This struct should be specified analogously to the way the parser's stacktype is defined using **%union** (see below). The location struct type is named **LTYPE__**. If neither **locationstruct** nor **LTYPE__** is specified, the default LTYPE__ struct is used.

5.5.10: %lsp-needed: using the default location type

Syntax: **%lsp-needed**
Defining this causes **bisonc++** to include code into the generated parser using the standard location stack. The token-location type defaults to the following struct, defined in the parser's base class when this directive is specified:

```
    struct LTYPE__
    {
        int timestamp;
        int first_line;
        int first_column;
        int last_line;
        int last_column;
        char *text;
    };
```

Note that defining this struct type does not imply that its field are also assigned. Some form of communication with the lexical scanner is probably required to initialize the fields of this struct properly.

5.5.11: %ltype: using an existing location type

%ltype typename
Specifies a user-defined token location type. If **%ltype** is used, `typename` should be the name of an alternate (predefined) type (e.g., **size_t**). It should not be used together with a %locationstruct specification. From within the parser class, this type may be used as **LTYPE__**.

Any text following `%ltype` up to the end of the line, up to the first of a series of trailing blanks or tabs or up to a comment-token (`//` or `/*`) becomes part of the type definition. Be sure *not* to end a `%ltype` definition in a semicolon.

5.5.12: %namespace: using a namespace

Syntax: **%namespace** `namespace`
Defines all of the code generated by **bisonc++** in the namespace `namespace`. By default no namespace is defined.

If this options is used the implementation header will contain a commented out `using namespace` directive for the requested namespace.

In addition, the parser and parser base class header files also use the specified namespace to define their include guard directives.

A warning is issued if this directive is used and an already existing parser-class header file and/or implementation header file does not define `namespace identifier`.

5.5.13: %negative-dollar-indices: using constructions like $-1

Syntax: **%negative-dollar-indices**
Accept (do not generate warnings) zero- or negative dollar-indices in the grammar's action blocks. Zero or negative dollar-indices are commonly used to implement inherited attributes and should normally be avoided. When used they can be specified like `$-1`, or like `$<type>-1`, where `type` is empty; an `STYPE__` tag; or a `%union` field-name. See also the sections 5.6.4 and 6.6.

In combination with the `%polymorphic` directive (see below) only the `$-i` format can be used (see also section 5.6.3).

5.5.14: %no-lines: suppressing `#line' directives

Syntax: **%no-lines**
Do not put **#line** preprocessor directives in the file containing the parser's `parse()` function. By default `#line` preprocessor directives are inserted just before action blocks in the

generated parse.cc file.

The #line directives allow compilers and debuggers to associate errors with lines in your grammar specification file, rather than with the source file containing the parse function itself.

5.5.15: %prec: overruling default precedences

Syntax: **%prec** token
The construction **%prec** token may be used in production rules to overrule the actual precendence of an operator in a particular production rule. Well known is the construction

```
    expression:
        '-' expression %prec UMINUS
        {
            ...
        }
```

Here, the default priority and precedence of the `-' token as the subtraction operator is overruled by the precedence and priority of the UMINUS token, which is frequently defined as:

```
    %right UMINUS
```

E.g., a list of arithmetic operators could consists of:

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

giving '*' '/' and '%' a higher priority than '+' and '-', ensuring at the same time that UMINUS is given both the highest priority and a right-associativity.

In the above production rule the operator order would cause the construction

```
    '-' expression
```

to be evaluated from right to left, having a higher precedence than either the multiplication or the addition operators.

5.5.16: %polymorphic: using polymorphism to define multiple semantic values

Syntax: **%polymorphic** `polymorphic-specification(s)`

The `%polymorphic` directive results in defining and using a polymorphic semantic value class, which can be used as a (preferred) alternative to the (traditional) a `union` specification.

Refer to section 5.6.3 for a detailed description of the specification, characteristics, and use of polymorphic semantic values as defined by **bisonc++**.

As a quick reference: to define multiple semantic values using a polymorphic semantic value class offering either an `int`, a `std::string` or a `std::vector<double>` specify:

```
    %polymorphic INT: int; STRING: std::string;
                 VECT: std::vector<double>
```

and use `%type` specifications (cf. section 5.5.25) to associate (non-)terminals with specific semantic values.

5.5.17: %print-tokens: displaying tokens and matched text

Syntax: **%print-tokens**

The `%print-tokens` directive provides an implementation of the Parser class's `print__` function displaying the current token value and the text matched by the lexical scanner as received by the generated `parse` function.

The `print__` function is also implemented if the `--print` command-line option is provided.

5.5.18: %required-tokens: defining the minimum number of tokens between error reports

Syntax: **%required-tokens** `ntokens`
Whenever a syntactic error is detected during the parsing process the next few tokens that are received by the parsing function may easily cause yet another (spurious) syntactic error. In this situation error recovery in fact produces an avalanche of additional errors. If this happens the recovery process may benefit from a slight modification. Rather than reporting every syntactic error encountered by the parsing function, the parsing function may wait for a series of successfully processed tokens before reporting the next error.

The directive `%required-tokens` can be used to specify this number. E.g., the specification `%required-tokens 10` requires the parsing function to process successfully a series of 10 tokens before another syntactic error is reported (and counted). If a syntactic error is encountered before processing 10 tokens then the counter counting the number of successfully processed tokens is reset to zero, no error is reported, but the error recoery procedure continues as usual. The number of required tokens can also be set using the option --required-tokens. By default the number of required tokens is initialized to 0.

5.5.19: %scanner: using a standard scanner interface

Syntax: **%scanner** `header`
Use `header` as the pathname of a file to include in the parser's class header. See the description of the --scanner option for details about this option. This directive also implies the automatic definition of a composed `Scanner d_scanner` data member into the generated parser, as well as a predefined **int lex()** member, returning `d_scanner.lex()`.

By specifying the `%flex` directive the function `d_scanner.YYText()` is called.

The specfied `header` file will be surrounded by double quotes if no delimiters were provided. If pointed brackets (`<...>`) are used, they are kept.

A warning is issued if this directive is used and an already existing parser class header file does not include `` `pathname' ``.

5.5.20: %scanner-matched-text-function: define the name of the scanner's member returning the matched texttoken

Syntax: **%scanner-matched-text-function** `function-call`

The `%scanner-matched-text-function` directive defines the scanner function returning the text matching the previously returned token. By default this is `d_scanner.matched()`.

A complete function call expression should be provided (including a scanner object, if used). This option overrules the `d_scanner.matched()` call used by default when the `%scanner` directive is specified. Example:

```
    %scanner-matched-text-function myScanner.matchedText()
```

If the function call expression contains white space then the `function-call` specification should be surrounded by double quotes ("). This directive is overruled by the **--scanner-matched-text-function** command-line option.

5.5.21: %scanner-token-function: define the name of the scanner's token function

Syntax: **%scanner-token-function** `function-call`

The scanner function returning the next token, called from the generated parser's `lex` function. A complete function call expression should be provided (including a scanner object, if used). Example:

```
    %scanner-token-function d_scanner.lex()
```

If the function call contains white space `scanner-token-function` should be surrounded by double quotes.

A warning is issued if this directive is used and the scanner token function is not called from the code in an already existing implementation header.

5.5.22: %start: defining the start rule

Syntax: **%start** `nonterminal symbol`

By default **bisonc++** uses the the LHS of the first rule in a grammar specification file as the start symbol. I.e., the parser will try to recognize that nonterminal when parsing input.

This default behavior may be overriden using the `%start` directive. The nonterminal symbol specifies a LHS that may be defined anywhere in the rules section of the grammar specification file. This LHS becomes the grammar's start symbol.

5.5.23: %stype: specifying the semantic stack type

Syntax: **%stype typename**
The type of the semantic value of tokens. The specification `typename` should be the name of an unstructured type (e.g., **size_t**). By default it is **int**. See **YYSTYPE** in **bison**. It should not be used if a **%union** specification is used. Within the parser class, this type may be used as **STYPE__**.

Any text following `%stype` up to the end of the line, up to the first of a series of trailing blanks or tabs or up to a comment-token (`//` or `/*`) becomes part of the type definition. Be sure *not* to end a `%stype` definition in a semicolon.

5.5.24: %token: defining token names

Syntax:

> **%token** `terminal token(s)`
> **%token** [<type>] `terminal token(s)`

The **%token** directive is used to define one or more symbolic terminal tokens. When multiple tokens are listed they must be separated by whitespace or by commas.

The `<type>` specification is optional, and specifies the type of the semantic value when a token specified to the right of a `<type>` specification is received. The pointed arrows are part of the type specification; the type itself must be a field of a `%union` specification (see section 5.5.26).

bisonc++ will convert the symbolic tokens (including those defined by the precedence directives (cf. section 5.5.8)) into a `Parser::Tokens` enumeration value (where `` `Parser' `` is the name of the generated parser class, see section 5.5.2). This allows the lexical scanner member function `lex()` to return these token values by name directly, and it allows externally defined lexical scanners (called by `lex()`) to return token values as `Parser::name`.

When an externally defined lexical scanner is used, it should include `Parserbase.h`, the parser's base class header file, in order to be able to use the `Parser::Tokens` enumeration type.

Although it *is* possible to specify explicitly the numeric code for a token type by appending an integer value in the field immediately following the token name (as in `%token NUM 300`) this practice is deprecated. It is generally best to let **bisonc++** choose the numeric codes for all token types. **Bisonc++** will automatically select codes that don't conflict with each other or with ASCII characters.

5.5.24.1: Improper token names

Several identifiers cannot be used as token names as their use would collide with identifiers that are defined in the parser's base class.

In particular,

- no token should end in two underscores (__).
- some identifiers are reserved and cannot be used as tokens. They are:

```
ABORT, ACCEPT, ERROR, clearin, debug, error, setDebug
```

Except for `error`, which is a predefined terminal token, these identifiers are the names of functions traditionally defined by **bisonc++**. The restriction on the above identifers could be lifted, but then the resulting generated parser would no longer be backward compatible with versions before **Bisonc++** 2.0.0. It appears that imposing minimal restrictions on the names of tokens is a small penalty to pay for keeping backward compatibility.

5.5.25: %type: associating semantic values to (non)terminals

Syntax: **%type** `<type> symbol(s)`
When `%polymorphic` is used to specify multiple semantic value types, (non-)terminals can be associated with one of the semantic value types specified with the `%polymorphic` directive.

When `%union` is used to specify multiple semantic value types, (non-)terminals can be associated with one of the `union` fields specified with the `%union` directive.

To associate (non-)terminals with specific semantic value types the **%type** directive is used.

With this directive, `symbol(s)` represents of one or more blank or comma delimited grammatical symbols (i.e., terminal and/or nonterminal symbols); `type` is either a polymorphic type-identifier or a field name defined in the `%union` specification. The specified nonterminal(s) are automatically associated with the indicate semantic type. The pointed arrows are part of the type specification.

When the semantic value type of a terminal symbol is defined the *lexical scanner* rather than the parser's actions must assign the appropriate semantic value to d_val__ just prior to returning the token. To associate terminal symbols with semantic values, terminal symbols can also be specified in a `%type` directive.

5.5.26: %union: using a 'union' to define multiple semantic values

Syntax: **%union** `union-definition body`
The `%union` directive 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 u_val;
  symrec *u_tptr;
};
```

This says that the two alternative types are `double` and `symrec *`. They are given names `u_val` and `u_tptr`; these names are used in the `%token` and `%type` directives to pick one of the types for a terminal or nonterminal symbol (see section 5.5.25).

Notes:

- The semicolon following the closing brace is *optional*.
- **C++-11** does allow class types to be used in `union` definitions, so they can also be used in `%union` directives. When a class type variant is required, all required constructors, the destructor and other members (like overloaded assignment operators) must be able to handle the actual class type data fields properly. A discussion of how to use unrestricted unions is beyon this manual's scope, but can be found, e.g., in the C++ Annotations. See also section 5.6.2.

5.5.27: %weak-tags: %polymorphic declaring 'enum Tag__'

By default, the `%polymorphic` directive declares a strongly typed enum: `enum class Tag__`, and code generated by **bisonc++** always uses the `Tag__` scope when referring to tag identifiers. It is often possible (by pre-associating tokens with tags, using `%type` directives) to avoid having to use tags in user-code.

If tags *are* explicitly used, then they must be prefixed with the `Tag__` scope. Before the arrival of the C++-11 standard strongly typed enumerations didn't exist, and explicit enum-type scope prefixes were usually omitted. This works fine, as long as there are no name-conflicts: parser-tokens, other enumerations or variables could use identifiers also used by `enum Tag__`. This results in compilation errors that can simply be prevented using strongly typed enumerations.

The `%weak-tags` directive can be specified when the `Tag__` enum should *not* be declared as a strongly typed enum. When in doubt, don't use this directive and stick to using the strongly typed `Tag__` enum. When using `%weak-tags` be prepared for compilation errors caused by name collisions.

5.5.28: Directives controlling the names of generated files

Unless otherwise specified, **bisonc++** uses the name of the parser-class to derive the names of most of the files it may generate. Below `<CLASS>` should be interpreted as the name of the parser's class, `Parser` by default, but configurable using `%class-name` (see section 5.5.2).

- The parser's base class header: `<Class>base.h`, configurable using `%baseclass-header` (see section 5.5.28.1) or `%filenames` (see section 5.5.28.3);
- The parser's class header: `<Class>.h`, configurable using `%class-header` (see section 5.5.28.2) or `%filenames` (see section 5.5.28.3);

- The parser's implementation header file: `<Class>.ih`, configurable using `%implementation-header` (see section 5.5.28.4) or `%filenames` (see section 5.5.28.3);

5.5.28.1: %baseclass-header: defining the parser's base class header

Syntax: **%baseclass-header** `filename`
`Filename` defines the name of the file to contain the parser's base class. This class defines, e.g., the parser's symbolic tokens. Defaults to the name of the parser class plus the suffix `base.h`. It is always generated, unless (re)writing is suppressed by the `--no-baseclass-header` and `--dont-rewrite-baseclass-header` options. This directive is overruled by the **--baseclass-header** (**-b**) command-line option.

A warning is issued if this directive is used and an already existing parser class header file does not contain `#include "filename"`.

5.5.28.2: %class-header: defining the parser's class header

Syntax: **%class-header** `filename`
`Filename` defines the name of the file to contain the parser class. Defaults to the name of the parser class plus the suffix `.h` This directive is overruled by the **--class-header** (**-c**) command-line option.

A warning is issued if this directive is used and an already existing parser-class header file does not define `class `className'` and/or if an already existing implementation header file does not define members of the class `` `className'``.

5.5.28.3: %filenames: specifying a generic filename

Syntax: **%filenames** `filename`
`Filename` is a generic filename that is used for all header files generated by **bisonc++**. Options defining specific filenames are also available (which then, in turn, overrule the name specified by this option). This directive is overruled by the **--filenames** (**-f**) command-line option.

5.5.28.4: %implementation-header: defining the implementation header

Syntax: **%implementation-header** `filename`
`Filename` defines the name of the file to contain the implementation header. It defaults to the name of the generated parser class plus the suffix `.ih`.
The implementation header should contain all directives and declarations *only* used by the implementations of the parser's member functions. It is the only header file that is included

by the source file containing `parse`'s implementation. User defined implementation of other class members may use the same convention, thus concentrating all directives and declarations that are required for the compilation of other source files belonging to the parser class in one header file.
This directive is overruled by the **--implementation-header** (**-i**) command-line option.

5.5.28.5: %parsefun-source: defining the parse() function's sourcefile

Syntax: **%parsefun-source** `filename`
`Filename` defines the name of the source file to contain the parser member function `parse`. Defaults to `parse.cc`. This directive is overruled by the **--parse-source** (**-p**) command-line option.

5.5.28.6: %target-directory: defining the directory where files must be written

Syntax: **%target-directory** `pathname`
`Pathname` defines the directory where generated files should be written. By default this is the directory where **bisonc++** is called. This directive is overruled by the `--target-directory` command-line option.

5.6: 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.

In this section defining the semantics of a language will be addressed. The following topics will be covered:

- Specifying one data type for all semantic values;
- Specifying several alternative data types;
- Using Polymorphism to specify several data types;
- Specifying Actions (an action is the semantic definition of a grammar rule);
- Specifying data types for actions to operate on;
- Specifying when and how to put actions in the middle of a rule (most actions go at the end of a rule. In some situations it may be desirable to put an action in in the middle of a rule).

5.6.1: 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, e.g., sections 4.1 and 4.2).

Bisonc++'s default is to use type `int` for all semantic values. To specify some other type, the directive `%stype` must be used, like this:

```
    %stype double
```

Any text following `%stype` up to the end of the line, up to the first of a series of trailing blanks or tabs or up to a comment-token (`//` or `/*`) becomes part of the type definition. Be sure *not* to end a `%stype` definition in a semicolon.

This directive must go in the directives section of the grammar file (see section 5.1). As a result of this, the parser class will define a *private type* `STYPE__` as `double`: Semantic values of language constructs always have the type `STYPE__`, and (assuming the parser class is named `Parser`) an internally used data member `d_val` that could be used by the lexical scanner to associate a semantic value with a returned token is defined as:

```
    Parser::STYPE__ d_val;
```

5.6.2: More Than One Value Type

In many programs, different kinds of data types are used in combination with different kinds of terminal and non-terminal tokens. For example, a numeric constant may need type `int` or `double`, while a string needs type `std::string`, and an identifier might need a pointer to an entry in a symbol table.

To use more than one data type for semantic values in one parser, **bisonc++** offers the following feature:

- Define polymorphic semantic values, associating (non)terminals with their proper semantic types (cf section 5.6.3), and associate (non-)terminal tokens with their appropriate semantic values;
- Specify the entire collection of possible data types, using a `%union` directive (see section 5.5.26), and associate (non-)terminal tokens with their appropriate semantic values;
- Define your own class handling the various semantic values, and associate that class with the parser's semantic value type using the `%stype` directive. The association of

(non-)terminal tokens and specific value types is handled by your own class.

The first approach (and to a lesser extent, the second approach) has the advantage that **bisonc++** is able to enforce the correct association between semantic types and rules and/or tokens, and that **bisonc++** is able to check the type-correctness of assignments to rule results.

5.6.3: Polymorphism and multiple semantic values: `%polymorphic'

Bisonc++ may generate code using polymorphic semantic values. The approach discussed here is a direct result of a suggestion originally made by Dallas A. Clement in September 2007. All sources of the example discussed in this section can be retrieved from the poly directory.

One may wonder why a `union` is still used by **bisonc++** as **C++** offers inherently superior approaches to combine multiple types in one type. The **C++** way to do so is by defining a polymorphic base class and a series of derived classes implementing the various exclusive data types. The `union` approach is still supported by **bisonc++** since it is supported by **bison**(1) and **bison++**; dropping the `union` would needlessly impede backward compatibility.

The (preferred) alternative to using a `union`, however, is using a polymorphic base class. Although it is possible to define your own polymorphic semantic value classes, **bisonc++** makes life easy by offering the `%polymorphic` directive.

The example program (cf. poly) implements a polymorphic base class, and derived classes containing either an `int` or a `std::string` semantic value. These types are asociated with tags (resp. `INT` and `TEXT` using the `%polymorphic` directive, which is discussed next.

5.6.3.1: The %polymorphic directive

The `%polymorphic` directive results in **bisonc++** generating a parser using polymorphic semantic values. Each semantic value specification consists of a *tag*, which is a **C++** identifier, and a **C++** type definition.

Tags and type definitions are separated from each other by a colon, and multiple semantic values specifications are separated by semicolons. The semicolon trailing the final semantic value specification is optional.

A grammar specification file may contain only one `%polymorphic` directive, and the `%polymorphic, %stype` and `%union` directives are mutually exclusive.

Here is an example, defining three semantic values types: an `int`, a `std::string` and a `std::vector<double>`:

```
    %polymorphic INT: int; STRING: std::string;
                 VECT: std::vector<double>
```

The identifier to the left of the colon is called the *type-identifier*, and the type definition to the right of the colon is called the *type-definition*. Types specified at the `%polymorphic` type-definitions must be built-in types or class-type declarations. Class types mentioned at the `%polymorphic` directive must offer default constructors.

If type declarations refer to types declared in header files that were not already included by the parser's base class header, then these header **s** must be inserted using the `%baseclass-preinclude` directive as these types are referred to in the generated `parserbase.h` header file.

5.6.3.2: The %polymorphic and %type: associating semantic values with (non-)terminals

The `%type` directive is used to associate (non-)terminals with semantic value types.

Semantic values may also be associated with terminal tokens. In that case it is the lexical scanner's responsibility to assign a properly typed value to the parser's `STYPE__ d_val__` data member.

Non-terminals may automatically be associated with polymorphic semantic values using `%type` directives. E.g., after:

```
    %polymorphic INT: int; TEXT: std::string
    %type <INT> expr
```

the `expr` non-terminal returns `int` semantic values. In this case, a rule like:

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

automatically associates $$, $1 and $3 with `int` values. $$ is an lvalue (representing the semantic value associated with the `expr:` rule), while $1 and $3 represent the `int` semantic value associated with the `expr` non-terminal in the production rule `'-' expr` (rvalues).

When negative dollar indices (like $-1) are used, pre-defined associations between non-terminals and semantic types are ignored. With positive indices or in combination with the production rule's return value `$$`, however, semantic value types can explicitly be specified using the common `$<type>$' or `$<type>1' syntax. (In this and following examples index number 1 represents any valid positive index; -1 represents any valid negative index).

The type-overruling syntax does not allow blanks to be used (so $<INT>$ is OK, $< INT >$ isn't).

Various combinations of type-associations and type specifications may be encountered:

- $-1: `%type` associations are ignored, and the semantic value type `STYPE__` is used instead. A warning is issued unless the `%negative-dollar-indices` directive was specified.
- $<tag>-1: *error*: `<tag>` specifications are not allowed for negative dollar indices.

$$ or $1 specifications

%type<TAG>	$<tag>	action:
absent	no <tag>	STYPE__ is used
	$<id>	tag-override
	$<>	STYPE__ is used
	$<STYPE__>	STYPE__ is used
STYPE__	no <tag>	STYPE__ is used
	$<id>	tag-override
	$<>	STYPE__ is used
	$<STYPE__>	STYPE__ is used
(existing) tag	no <tag>	auto-tag

	$<id>	tag-override
	$<>	STYPE__ is used
	$<STYPE__>	STYPE__ is used
(undefined) tag	no <tag>	tag-error
	$<id>	tag-override
	$<>	STYPE__ is used
	$<STYPE__>	STYPE__ is used

- auto-tag: $$ and $1 represent, respectively, `$$.get<tag>()` and `$1.get<tag>()`;
- tag-error: *error:* tag undefined;
- tag-override: if `id` is a defined tag, then $<tag>$ and $<tag>1 represent the tag's type. Otherwise: *error* (using undefined tag `id`).

When using `$$.' or `$1.' default tags are ignored. A warning is issued that the default tag is ignored. This syntax allows members of the semantic value type (`STYPE__`) to be called explicitly. The default tag is only ignored if there are no additional characters (e.g., blanks, closing parentheses) between the dollar-expressions and the member selector operator (e.g., no tags are used with $1.member(), but tags are used with ($1).member()). The opposite, overriding default tag associations, is accomplished using constructions like $<STYPE__>$ and $<STYPE__>1.

When negative dollar indices are used, the appropriate tag must explicitly be specified. The next example shows how this is realized in the grammar specification file itself:

```
    %polymorphic INT: int
    %type <INT> ident
    %%

    type:
        ident arg
    ;
    arg:
        {
            call($-1->get<Tag__::INT>());
        }
    ;
```

In this example `call` may define an `int` or `int &` parameter.

It is also possible to delegate specification of the semantic value to the function `call` itself, as shown next:

```
    %polymorphic INT: int
    %type <INT> ident
    %%

    type:
        ident arg
    ;

    arg:
        {
            call($-1);
        }
    ;
```

Here, the function `call` could be implemented like this:

```
    void call(STYPE__ &st)
    {
        st->get<Tag__::INT>() = 5;
    }
```

5.6.3.3: Code generated by %polymorphic

The parser using polymorphic semantic values adds several classes to the generated files. The majority of these are class templates, defined in `parserbase.h`. In addition, some of the additionally implemented code is added to the `parse.cc` source file.

To minimize namespace pollution most of the additional code is contained in a namespace of its own: `Meta__`. If the `%namespace` directive was used then `Meta__` is nested under the namespace declared by that directive. The name `Meta__` hints at the fact that much of the code implementing polymorphic semantic values uses template meta programming.

The enumeration 'enum class Tag__'

One notable exception is the enumeration `Tag__`. It is declared outside of `Meta__` (but inside the `%namespace` namespace, if provided) to simplify its use. Its identifiers are the tags declared by the `%polymorphic` directive. This is a strongly typed enumeration. The `%weak-tags` directive can be used to declare a pre C++-11 standard `enum Tag__`.

The namespace Meta__

Below, `DataType` refers to the semantic value's data type that is associated with a `Tag__` identifier; `ReturnType` equals `DataType` if `DataType` is a built-in type (like `int, double,` etc.), while it is equal to `DataType const &` otherwise (for, e.g., class-type data types).

The important elements of the namespace `Meta__` are:

- The polymorphic semantic value's base class `Base`.
  Its public interface offers the following members:
  - `Tag__ tag() const`: returns the semantic value type's tag.
  - `ReturnType get<Tag__>() const`: accesses the (non-modifiable) data element of the type matching the tag. the data element of the type matching the tag (also see below at the description of the class `SType`).
  - `DataType &get<Tag__>() const`: accesses the (modifiable) data element of the type matching the tag.

- The semantic value classes `Semantic<Tag__::ID>: public Base`.
  `Semantic<Tag__::ID>` classes are derived for each of the tag identifiers `ID` declared at the `%polymorphic` directive. `Semantic<Tag__::ID>` classes contain a `mutable DataType` data member. Their public interfaces offer the following members:
  - A default constructor;
  - A `Semantic(DataType const &)` constructor;
  - A `Semantic(DataType &&)` constructor;
  - An `operator ReturnType() const` conversion operator;
  - An `operator DataType &()t` conversion operator.

  `Semantic<Tag__::ID>` objects are usually not explicitly used. Rather, their use is implied by the actual semantic value class `SType` and by several support functions (see below).

- De semantic value class `SType: public std::shared_ptr<Base>` provides access to the various semantic value types. The `SType__` class becomes the parser's `STYPE__` type, and explicitly accessing `Semantic<Tag__::ID>` should never be necessary.
  `SType`'s public interface offers the following members:

- Constructors: default, copy and move constructors.
  Since the parser's semantic value and semantic value stack is completely controlled by the parser, and since the actual semantic data values are unknown at construction time of the semantic value (d_val__) and of the semantic value stack, no constructors expecting DataType values are provided.
- Assignment operators.
  The standard overloaded assignment operator (copy and move variants) as well as copy and move assignment operators for the types declared at the %polymorphic directive are provided. Assigning a value using operator= allocates a Semantic<Tag__::tag> object for the tag matching the right-hand side's data type, and resets the SType's shared_pointer to this new Semantic<Tag__::tag> object. Be aware that this may break the default association of the semantic value as declared by a %type directive. When breaking the default association make sure that explicit tags are used (as in $<Tag__::tag>1), overriding the default association with the currently active association. In most cases, however, the assignment is not used to break the default association but simply to assign a value to $$. By default the SType's shared pointer is zero, and the assignment initializes the semantic value to a value of the proper type. Assuming a lexical scanner may return a NR token, offering an int number() const accessor, then part of expr rule could be:

  ```
      expr:
          NR
          {
              $$ = d_scanner.number();
          }
      ...
  ```

  after which expr's semantic value has been initialized to a Semantic<Tag__::INT>.
- returns the semantic value stored inside Semantic<Tag__>. If the type-name is a built-in type a copy of the value is returned, otherwise a reference to a constant object is returned;
  This member checks for 0-pointers and for Tag__ mismatches between the requested and actual Tag__, throwing a std::logic_error if found.
- returns a reference to the (modifiable) semantic value stored inside Semantic<Tag__>.
  This member checks for 0-pointers and for Tag__ mismatches between the requested and actual Tag__, in that case replacing the current Semantic object pointed to by a new Semantic<Tag__> object of the requested Tag__.
- refers to the semantic value stored inside Semantic<Tag__>. If the type-name is a built-in type a copy of the value is returned, otherwise a reference to a constant

object is returned;
This is a (partially) *unchecking* variant of the corresponing `get` member, resulting in a *Segfault* if used when the `shared_ptr` holds a 0-pointer, and throwing a `std::bad_cast` in case of a mismatch between the requested and actual `Tag__`.

  - returns a reference to the (modifiable) semantic value stored inside `Semantic<Tag__>`.
    This is a (partially) *unchecking* variant of the corresponing `get` member, resulting in a *Segfault* if used when the `shared_ptr` holds a 0-pointer, and throwing a `std::bad_cast` in case of a mismatch between the requested and actual `Tag__`.

When an incorrect tag is specified (e.g., with `get<Tag__::tag>(), $<Tag__::tag>$`, or `$<Tag__::tag>1`), the generated code correctly copiles, but the program will throw a `std::bad_cast` exception once the offending code is executed.

Additional Headers

When using `%polymorphic` three additional header files are included by `parserbase.h`:

- `memory`, required for `std::shared_ptr`;
- `stdexcept`, required for `std::logic_error`;
- `type_traits`, required for the implementation of one of `SType`'s overloaded assignment operators.

5.6.3.4: A parser using a polymorphic semantic value type

In this section a parser is developed using polymorphic semantic values. Its `%polymorphic` directive looks like this:

```
    %polymorphic INT: int; TEXT: std::string;
```

Furthermore, the grammar declares tokens `INT` and `IDENTIFIER`, and pre-associates the `TEXT` tag with the `identifier` non-terminal, associates the `INT` tag with the `int` non-terminal, and associates `STYPE__`, the generic polymorphic value with the non=terminal `combi`:

```
    %type <TEXT>    identifier
    %type <INT>     int
    %type <STYPE__> combi
```

For this example a simple grammar was developed, expecting an optional number of input lines, formatted according to the following `rule` production rules:

```
    rule:
        identifier '(' identifier ')' '\n'
    |
        identifier '=' int '\n'
    |
        combi '\n'
    ;
```

The rules for `identifier` and `int` assign, respectively, text and an `int` value to the parser's semantic value stack:

```
    identifier:
        IDENTIFIER
        {
            $$ = d_scanner.matched();
        }
    ;
    int:
        INT
        {
            $$ = d_scanner.intValue();
        }
    ;
```

These simple assignments can be used as `int` is pre-associated with the `INT` tag and `identifier` is asociated with the `TEXT` tag.

As the `combi` rule is not associated with a specific semantic value, its semantic value could be either `INT` or `TEXT`. Irrespective of what is actually returned by `combi`, its semantic value can be passed on to a function (`process(STYPE__ const &)`, responsible for the semantic value's further processing. Here are the definition of the `combi` non-terminal and action blocks for the `rule` non-terminal:

```
    combi:
        int
    |
        identifier;
    ;

    rule:
        identifier '(' identifier ')' '\n'
        {
            cout << $1 << " " << $3 << '\n';
        }
    |
        identifier '=' int '\n'
        {
            cout << $1 << " " << $3 << '\n';
        }
    |
        combi '\n'
        {
            process($1);
        }
    ;
```

Since `identifier` has been associated with `TEXT` and `int` with `INT`, the $-references to these elements in the production rules already return, respectively, a `std::string const &` and an `int`.

For `combi` the situation is slightly more complex, as `combi` could either return an `int` (via its `int` production rule) or a `std::string const &` (via its `identifier` production rule).

Fortunately, `process` can find out by inspecting the semantic value's `Tag__`:

```
    void process(SSTYPE__ const &semVal)
    {
        if (semVal.tag() == Tag__::INT)
            cout << "Saw an int-value: " << semVal.get<Tag__::INT>() << '\n';
        else
            cout << "Saw text: " << semVal.get<Tag__::TEXT>() << '\n';
    }
```

5.6.3.5: A scanner using a polymorphic semantic value type

The scanner recognizes input patterns, and returns Parser tokens (e.g., Parser::INT) matching the recognized input.

It is easily created by **flexc++**(1) processing the following simple specification file.

```
%interactive
%filenames scanner

%%

[ \t]+                  // skip white space
```

```
[0-9]+                  return Parser::NR;

[a-zA-Z_][a-zA-Z0-9_]*  return Parser::IDENTIFIER;

.|\n                    return matched()[0];
```

The reader may refer to **flexc++**(1) documentation for details about **flexc++**(1) specification files.

5.6.4: 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 should be used only for special purposes (see section 5.6.6).

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 `$$`. (**Bisonc++** 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, **bisonc++** supplies a default: `$$ = $1`. Thus, the value of the first symbol in the rule becomes the value of the whole rule. Of course, the default rule is valid only if the two data types match. There is no meaningful default action for an empty rule; every empty rule must have an explicit action unless the rule's value does not matter. Note that the default `$$` value is assigned at the *beginning* of an action block. Any changes to `$1` are therefore *not* automatically propagated to `$$`. E.g., assuming that `$1 == 3` at the beginning of the following action block, then `$$` will still be equal to 3 after executing the statement in the action block:

```
    {                 // assume: $1 == 3
        $1 += 12;     // $1 now is 15, $$ remains 3
    }
```

If `$$` should receive the value of the modified `$1`, then `$$` must explicitly be assigned to `$$`. E.g.,

```
    {                 // assume: $1 == 3
        $1 += 12;     // $1 now is 15, $$ remains 3
        $$ = $1;      // now $$ == 15 as well.
    }
```

Using `$n` with n equal to zero or a negative value 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`. But as mentioned: it's a risky practice, which should be avoided if at all possible. See also section 6.6.

All `$`-type variables used in action blocks can be modified. All numbered `$`-variables are deleted when a production rule has been recognized. Unless an action explicitly assigns a value to `$$`, the (possibly modified) `$1` value is assigned to `$$` when a production rule has been recognized.

5.6.5: 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 a `%union` directive 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 u_int;
        double u_double;
    };
```

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

5.6.6: 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 recognizes the components that follow them.

A mid-rule action may refer to the components preceding it using `$n`, but it may not (cannot) refer to subsequent components because it is executed 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 ')'
        {
            $<u_context>$ = pushSymtab();
            temporaryVariable($3);
```

```
        }
        stmt
        {
            $$ = $6;
            popSymtab($<u_context>5);
        }
```

As soon as `let (variable)' has been recognized, the first action is executed. It saves a copy of the current symbol table as its semantic value, using alternative u_context in the data-type union. Then it uses temporaryVariable() to add the new variable (using, e.g., a name that cannot normally be used in the parsed language) to the current symbol table. Once the first action is finished, the embedded statement (stmt) can be parsed. Note that the mid-rule action is component number 5, so `stmt' is component number 6.

Once 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 former symbol table. 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:
        {
            prepareForLocalVariables();
        }
        '{' declarations statements '}'
    |
        '{' statements '}'
    ;
```

Now the parser is forced to decide whether to execute 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 section 7.1.4.

You might think that the problem can be solved by putting identical actions into the two rules, like this:

```
        {
            prepareForLocalVariables();
        }
        '{' declarations statements '}'
    |
        {
            prepareForLocalVariables();
        }
        '{' statements '}'
    ;
```

But this does not help, because **bisonc++** *never* parses the contents of actions, and so it will *not* realize that the two actions are identical.

If the grammar is such that a declaration can be distinguished from a statement by the first token (which is true in **C**, but *not* in **C++**, which allows statements and declarations to be mixed)), then one solution is to put the action after the open-brace, like this:

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

Now the next token following a recognized `'{'` token would be either the first `declarations` token or the first `statements` token, which would in any case tell **bisonc++** which rule to use, thus solving the problem.

Another (much used) solution is to bury the action inside a support non-terminal symbol which recognizes the first block-open brace and performs the required preparations:

```
    openblock:
        '{'
        {
            prepareForLocalVariables();
        }
    ;

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

Now **bisonc++** 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 **bisonc++** actually does to implement mid-rule actions.

By the way, note that in a language like **C++** the above construction is obsolete anyway, since **C++** allows mid-block variable- and object declarations. In **C++** a compound statement could be defined, e.g., as follows:

```
    stmnt_or_decl:
        declarations
    |
        pure_stmnt      // among which: compound_stmnt
    ;

    statements:
        // empty
    |
        statements stmnt_or_decl
    ;

    compound_stmnt:
        open_block statements '}'
    ;
```

Here, the `compound_stmnt` would begin with the necessary preparations for local declarations, which would then have been completed by the time they would really be needed by `declarations`.

5.7: Basic Grammatical Constructions

In the following sections several basic grammatical constructions are presented in their prototypical and generic forms. When these basic constructions are used to construct a grammar, the resulting grammar will usually be accepted by **bisonc++**. Moreover, these basic constructions are frequently encountered in programming languages. When designing your own grammar, try to stick as closely as possible to the following basic grammatical constructions.

5.7.1: Plain Alternatives

Simple alternatives can be specified using the vertical bar (|). Each alternative should begin with a unique identifying terminal token. The terminal token may actually be hidden in a non-terminal rule, in which case that nonterminal can be used as an alias for the non-terminal. In fact identical terminal tokens may be used if at some point the terminal tokens differ over different alternatives. Here are some examples:

```
    // Example 1:  plain terminal distinguishing tokens
    expr:
        ID
    |
        NUMBER
    ;

    // Example 2:  nested terminal distinguishing tokens
    expr:
        id
    |
        number
    ;

    id:
        ID
    ;

    number:
        NUMBER
    ;

    // Example 3:  eventually diverting routes
    expr:
        ID
        id
    |
        ID
        number
    ;
    id:
        ID
    ;

    number:
        NUMBER
    ;
```

5.7.2: One Or More Alternatives, No Separators

A series of elements normally use left-recursion. For example, **C++** supports *string concatenation*: series of double quote delimited `ASCII` characters define a string, and multiple white-space delimited strings are handled as one single string:

```
    "hello"         // multiple ws-delimited strings
    " "
    "world"

    "hello world"   // same thing
```

Usually a parser is responsible for concatenating the individual string-parts, receiving one or more `STRING` tokens from the lexical scanner. A `string` rule handles one or more incoming `STRING` tokens:

```
    string:
        STRING
    |
        string STRING
```

The above rule can be used as a prototype for recognizing a series of elements. The token `STRING` may of course be embedded in another rule. The generic form of this rule could be formulated as follows:

```
    series:
        unit
    |
        series unit
```

Note that the single element is *first* recognized, whereafter the left-recursive alternative may be recognized repeatedly.

5.7.3: Zero Or More Alternatives, No Separators

An *optional* series of elements also use left-recursion, but the single element alternative remains empty. For example, in **C++** a compound statement may contain statements or declarations, but it may also be empty:

```
    opt_statements:
        // empty
    |
        opt_statements statements
```

The above rule can be used as a prototype for recognizing a series of elements: the generic form of this rule could be formulated as follows:

```
    opt_series:
        // empty
    |
        opt_series unit
```

Note that the empty element is recognized *first*, even though it is empty, whereafter the left-recursive alternative may be recognized repeatedly. In practice this means that an *action block* may be defined at the empty alternative, which is then executed prior to the left-recursive alternative. Such an action block could be used to perform initializations necessary for the proper handling of the left-recursive alternative.

5.7.4: One Or More Alternatives, Using Separators

A series of elements which are separated from each other using some delimiter again normally uses left-recursion. For example, a **C++** variable definition list consists of one or more identifiers, separated by comma's. If there is only one identifier no comma is used. Here is the rule defining a list using separators:

```
    variables:
        IDENTIFIER
    |
        variables ',' IDENTIFIER
```

The above rule can be used as a prototype for recognizing a series of delimited elements. The generic form of this rule could be formulated as follows:

```
    series:
        unit
    |
        series delimiter unit
```

Note that the single element is *first* recognized, whereafter the left-recursive alternative may be recognized repeatedly. In fact, this rule is not really different from the standard rule for a series, which does not hold true for the rule to recognize an *optional* series of delimited elements, covered in the next section.

5.7.5: Zero Or More Alternatives, Using Separators

An optional series of elements, separated from each other using delimiters occurs frequently in programming languages. For example, **C++** functions have parameter lists which may or may not require arguments. Since a parameter list may be defined empty, an *empty* alternative is required. However, a simple generalization of the optional series construction (section 5.7.3) won't work, since that would imply that the *first* argument is preceded by a separator, which is clearly not the intention. So, the following construction is *wrong*:

```
    opt_parlist:
        // empty
    |
        opt_parlist ',' parameter
```

To define an optional series of delimited elements *two* rules are required: one rule handling the optional part, the other the delimited series of elements. So, the correct definition is as follows:

```
    opt_parlist:
        // empty
    |
        parlist
    ;

    parlist:
        parameter
    |
        parlist ',' parameter
    ;
```

Again, the above rule pair can be used as a prototype for recognizing an optional series of delimited elements. The generic form of these rules could be formulated as follows:

```
    opt_series:
        // empty
    |
        series
    ;

    series:
        element
    |
        series delimiter element
```

Note that the `opt_series` rules neatly distinguishes the no-element case from the case were elements are present. Usually these two cases need to be handled quite differently, and the `opt_series` rules empty alternative easily allows us to recognize the no-elements case.

5.7.6: Nested Blocks

Finally, we add the *nested* rule to our bag of rule-tricks. Again, nested rules appear frequently: parenthesized expressions and compound statements are two very well known examples. These kind of rules are characterized by the fact that the nested variant is itself an example of the element appearing in the nested variant. The definition of a statement is actually a bit more complex than the definition of an expression, since the statement appearing in the compound statement is in fact an optional series of elements. Let's first have a look at the nested expression rule. Here it is, in a basic form:

```
    expr:
        NUMBER
    |
        ID
    |
        expr '+' expr
    |
        ...
    |
        '(' expr ')'
    ;
```

This definition is simply characterized that the non-terminal `expr` appears within a set of parentheses, which is not too complex.

The definition of `opt_statements`, however, is a bit more complex. But acknowledging the fact that a `statement` contains among other elements a compound statement, and that a compound statement, in turn, contains `opt_statements` an `opt_statements` construction can be formulated accordingly:

```
    opt_statements:         // define an optional series
        // empty
    |
        opt_statements statement
    ;

    statement:              // define alternatives for `statement'
        expr_statement
    |
        if_statement
    |
        ...
    |
        compound_statement
    ;

    compound_statement:     // define the compound statement itself
        '{' opt_statements '}'
    ;
```

5.8: Multiple Parsers in the Same Program

Most programs that use **bisonc++** parse only one language and therefore contain only one **bisonc++** parser. But what if you want to parse more than one language with the same program? Since **bisonc++** constructs *class* rather than a *parsing function*, this problem can easily be solved: simply define your second (third, fourth, ...) parser class, each having its own unique class-name, using the `%class-name` directive, and construct parser objects of each of the defined classes.

- Table of Contents
- Previous Chapter
- Next Chapter

- Table of Contents
- Previous Chapter
- Next Chapter

Chapter 6: The Generated Parser Class' Members

Bisonc++ generates a **C++** *class*, rather than a *function* like Bison. **Bisonc++**'s class is a plain **C++** class and not a fairly complex macro-based class like the one generated by Bison++. The **C++** class generated by **bisonc++** does not have (need) *virtual* members. Its essential member (the member `parse()`) is generated from the grammar specification and so the software engineer will therefore hardly ever feel the need to override that function. All but a few of the remaining predefined members have very clear definitions and meanings as well, making it unlikely that they should ever require overriding.

It is likely that members like `lex()` and/or `error()` need dedicated definitions with different parsers generated by Bison++; but then again: while defining the grammar the definition of the associated support members is a natural extension of defining the grammar, and can be realized *in parallel* with defining the grammar, in practice not requiring any virtual members. By not defining (requiring) virtual members the parser's class organization is simplified, and the calling of the non-virtual members will be just a trifle faster than when these member functions would have been virtual.

In this chapter all available members and features of the generated parser class are discussed. Having read this chapter you should be able to use the generated parser class in your program (using its public members) and to use its facilities in the actions defined for the various production rules and/or use these facilities in additional class members that you might have defined yourself.

In the remainder of this chapter the class's public members are first discussed, to be followed by the class's private members. While constructing the grammar all private members are available in the action parts of the grammaticalrules. Furthermore, any member (and so not just from the action blocks) may generate errors (thus initiating error recovery procedures) and may flag the (un)successful parsing of the information given to the parser (terminating the parsing function `parse()`).

6.1: Public Members and Types

The following public members and types are available to users of the parser classes generated by **bisonc++** (parser class-name prefixes (e.g., `Parser::`) prefixes are silently implied):

- **LTYPE__**:
  The parser's location type (user-definable). Available only when either `%lsp-needed, %ltype` or `%locationstruct` has been declared.
- **STYPE__**:
  The parser's stack-type (user-definable), defaults to **int**.
- **Tokens__**:
  The enumeration type of all the symbolic tokens defined in the grammar file (i.e., **bisonc++**'s input file). The scanner should be prepared to return these symbolic tokens. Note that, since the symbolic tokens are defined in the parser's class and not in the scanner's class, the lexical scanner must prefix the parser's class name to the symbolic token names when they are returned. E.g., `return Parser::IDENT` should be used rather than `return IDENT`.
- **int parse()**:
  The parser's parsing member function. It returns 0 when parsing was successfully completed; 1 if errors were encountered while parsing the input.
- **void setDebug(bool mode)**:
  This member can be used to activate or deactivate the debug-code compiled into the parsing function. It is always defined but is only operational if the `%debug` directive or `--debug` option was specified. When debugging code has been compiled into the parsing function, it is *not* active by default. To activate the debugging code, use `setDebug(true)`.

  This member can be used to activate or deactivate the debug-code compiled into the parsing function. It is available but has no effect if no debug code has been compiled into the parsing function. When debugging code has been compiled into the parsing function, it is active by default, but debug-code is suppressed by calling `setDebug(false)`.

When the `%polymorphic` directive is used:

- **Meta__**:
  Templates and classes that are required for implementing the polymorphic semantic values are all declared in the `Meta__` namespace. The `Meta__` namespace itself is nested under the namespace that may have been declared by the `%namespace`

directive.

- **Tag__**:
  The (strongly typed) `enum class Tag__` contains all the tag-identifiers specified by the `%polymorphic` directive. It is declared outside of the Parser's class, but within the namespace that may have been declared by the `%namespace` directive.

6.2: Protected Enumerations and Types

The following enumerations and types can be used by members of parser classes generated by **bisonc++**. They are actually protected members inherited from the parser's base class.

- **Base::ErrorRecovery__**:
  This enumeration defines two values:

  ```
  DEFAULT_RECOVERY_MODE__,
  UNEXPECTED_TOKEN__
  ```

  The `DEFAULT_RECOVERY_MODE__` terminates the parsing process. The non-default recovery procedure is available once an `error` token is used in a production rule. When the parsing process throws `UNEXPECTED_TOKEN__` the recovery procedure is started (i.e., it is started whenever a syntactic error is encountered or `ERROR()` is called).

  The recovery procedure consists of (1) looking for the first state on the state-stack having an error-production, followed by (2) handling all state transitions that are possible without retrieving a terminal token. Then, in the state requiring a terminal token and starting with the initial unexpected token (3) all subsequent terminal tokens are ignored until a token is retrieved which is a continuation token in that state.

  If the error recovery procedure fails (i.e., if no acceptable token is ever encountered) error recovery falls back to the default recovery mode (i.e., the parsing process is terminated).

- **Base::Return__**:
  This enumeration defines two values:

```
        PARSE_ACCEPT = 0,
        PARSE_ABORT = 1
```

(which are of course the `parse` function's return values).

6.3: Non-public Member Functions

The following members can be used by members of parser classes generated by **bisonc++**. When prefixed by `Base::` they are actually protected members inherited from the parser's base class. Members for which the phrase ``Used internally'' is used should not be called by user-defined code.

- **Base::ParserBase()**:
  Used internally.
- **void Base::ABORT() const throw(Return__)**:
  This member can be called from any member function (called from any of the parser's action blocks) to indicate a failure while parsing thus terminating the parsing function with an error value 1. Note that this offers a marked extension and improvement of the macro `YYABORT` defined by **bison++** in that `YYABORT` could not be called from outside of the parsing member function.
- **void Base::ACCEPT() const throw(Return__)**:
  This member can be called from any member function (called from any of the parser's action blocks) to indicate successful parsing and thus terminating the parsing function. Note that this offers a marked extension and improvement of the macro `YYACCEPT` defined by **bison++** in that `YYACCEPT` could not be called from outside of the parsing member function.
- **void Base::clearin()**:
  This member replaces **bison**(++)'s macro `yyclearin` and causes **bisonc++** to request another token from its `lex+nop()()` member, even if the current token has not yet been processed. It is a useful member when the parser should be reset to its initial state, e.g., between successive calls of `parse`. In this situation the scanner will probably be reloaded with new information too.
- **bool Base::debug() const**:
  This member returns the current value of the debug variable.
- **void Base::ERROR() const throw(ErrorRecovery__)**:
  This member can be called from any member function (called from any of the parser's action blocks) to generate an error, and results in the parser executing its error recovery code. Note that this offers a marked extension and improvement of the macro `YYERROR` defined by **bison++** in that `YYERROR` could not be called from

outside of the parsing member function.

- **void error(char const *msg)**:
  By default implemented inline in the `parser.ih` internal header file, it writes a simple message to the standard error stream. It is called when a syntactic error is encountered, and its default implementation may safely be altered.
- **void errorRecovery__()**:
  Used internally.
- **void Base::errorVerbose__()**:
  Used internally.
- **void exceptionHandler__(std::exception const &exc)**:
  This member's default implementation is provided inline in the `parser.ih` internal header file. It consists of a mere `throw` statement, rethrowing a caught exception.

  The `parse` member function's body essentially consists of a `while` statement, in which the next token is obtained via the parser's `lex` member. This token is then processed according to the current state of the parsing process. This may result in executing actions over which the parsing process has no control and which may result in exceptions being thrown.

  Such exceptions do not necessarily have to terminate the parsing process: they could be thrown by code, linked to the parser, that simply checks for semantic errors (like divisions by zero) throwing exceptions if such errors are observed.

  The member `exceptionHandler__` receives and may handle such exceptions without necessarily ending the parsing process. It receives any `std::exception` thrown by the parser's actions, as though the action block itself was surrounded by a `try ... catch` statement. It is of course still possible to use an explicit `try ... catch` statement within action blocks. However, `exceptionHandler__` can be used to factor out code that is common to various action blocks.

  The next example shows an explicit implementation of `exceptionHandler__`: any `std::exception` thrown by the parser's action blocks is caught, showing the exception's message, and increasing the parser's error count. After this parsing continues as if no exception had been thrown:

```
    void Parser::exceptionHandler__(std::exception const &exc)
    {
        std::cout << exc.what() << '\n';
        ++d_nErrors__;
    }
```

Note: Parser-class header files (e.g., Parser.h) and parser-class internal header files (e.g., Parser.ih) generated with **bisonc++** < 4.02.00 require two hand-modifications when using **bisonc++** >= 4.02.00:

In Parser.h, just below the declaration

```
    void print__();
```

add:

```
    void exceptionHandler__(std::exception const &exc);
```

In Parser.ih, assuming the name of the generated class is `Parser', add the following member definition (if a namespace is used: within the namespace's scope):

```
    inline void Parser::exceptionHandler__(std::exception const &exc)
    {
        throw;  // re-implement to handle exceptions thrown by actions
    }
```

- **void executeAction(int)**:
  Used internally.
- **int lex()**:
  By default implemented inline in the `parser.ih` internal header file, it can be pre-implemented by **bisonc++** using the `scanner` option or directive (see above); alternatively it *must* be implemented by the programmer. It interfaces to the lexical scanner, and should return the next token produced by the lexical scanner, either as a plain character or as one of the symbolic tokens defined in the `Parser::Tokens__` enumeration. Zero or negative token values are interpreted as `end of input'.
- **int lookup(bool)**:
  Used internally.
- **void nextToken()**:
  Used internally.
- **void Base::pop__()**:
  Used internally.
- **void Base::popToken__()**:
  Used internally.

- **void print__()()**:
  Used internally.
- **void print()**

:
By default implemented inline in the `parser.ih` internal header file, this member calls `print__` to display the last received token and corrseponding matched text. The `print__` member is only implemented if the `--print-tokens` option or `%print-tokens` directive was used when the parsing function was generated. Calling `print__` from `print` is unconditional, but can easily be controlled by the using program, by defining, e.g., a command-line option.
- **void Base::push__()**:
Used internally.
- **void Base::pushToken__()**:
Used internally.
- **void Base::reduce__()**:
Used internally.
- **void Base::symbol__()**:
Used internally.
- **void Base::top__()**:
Used internally.)

6.3.1: `lex()': Interfacing the Lexical Analyzer

The `int lex()` private member function is called by the `parse()` member to obtain the next lexical token. By default it is not implemented, but the `%scanner` directive (see section 5.5.19) may be used to pre-implement a standard interface to a lexical analyzer.

The `lex()` member function interfaces to the lexical scanner, and it is expected to return the next token produced by the lexical scanner. This token may either be a plain character or it may be one of the symbolic tokens defined in the **Parser::Tokens** enumeration. Any zero or negative token value is interpreted as `end of input', causing `parse()` to return.

The `lex()` member function may be implemented in various ways:

- By default, if the `--scanner` option or `%scanner` directive is provided **bisonc++** assumes that it should interface to the scanner generated by **flexc++**(1). In this case, the scanner token function is called as

```
        d_scanner.lex()
```

and the scanner's matched text function is called as

```
    d_scanner.matched()
```

- `lex()` may itself implement a lexical analyzer (a *scanner*). This may actually be a useful option when the input offered to the program using **bisonc++**'s parser class is not overly complex. This approach was used when implementing the earlier examples (see sections 4.1.3 and 4.4.4).
- `lex()` may call a external function or member function of class implementing a lexical scanner, and return the information offered by this external function. When using a class, an object of that class could also be defined as additional data member of the parser (see the next alternative). This approach can be followed when generating a lexical scanner from a lexical scanner generating tool like **lex**(1) or **flex++**(1). The latter program allows its users to generate a scanner *class*.
- To interface **bisonc++** to code generated by **flex**(1), the `--flex` option or `%flex` directive can be used in combination with the `--scanner` directive or `%scanner` option. In this case the scanner token function is called as

  ```
      d_scanner.yylex()
  ```

  and the scanner's matched text function is called as

  ```
      d_scanner.YYText()
  ```

6.4: Protected Data Members

The following private members can be used by members of parser classes generated by **bisonc++**. All data members are actually protected members inherited from the parser's base class.

- **size_t d_acceptedTokens__**:
  Counts the number of accepted tokens since the start of the `parse()` function or since the last detected syntactic error. It is initialized to `d_requiredTokens__` to allow an early error to be detected as well.
- **bool d_debug__**:

When the **debug** option has been specified, this variable (**true** by default) determines whether debug information is actually displayed.

- **LTYPE__ d_loc__**:
  The location type value associated with a terminal token. It can be used by, e.g., lexical scanners to pass location information of a matched token to the parser in parallel with a returned token. It is available only when **%lsp-needed, %ltype** or **%locationstruct** has been defined.
  Lexical scanners may be offered the facility to assign a value to this variable in parallel with a returned token. In order to allow a scanner access to **d_loc__**, **d_loc__**'s address should be passed to the scanner. This can be realized, for example, by defining a member **void setLoc(STYPE__ *)** in the lexical scanner, which is then called from the parser's constructor as follows:

  ```
              d_scanner.setSLoc(&d_loc__);
  ```

  Subsequently, the lexical scanner may assign a value to the parser's **d_loc__** variable through the pointer to **d_loc__** stored inside the lexical scanner.
- **LTYPE__ d_lsp__**:
  The location stack pointer. Used internally.
- **size_t d_nErrors__**:
  The number of errors counted by `parse()`. It is initialized by the parser's base class initializer, and is updated while `parse()` executes. When `parse()` has returned it contains the total number of errors counted by `parse()`. Errors are not counted if suppressed (i.e., if `d_acceptedTokens__` is less than `d_requiredTokens__`).
- **size_t d_nextToken__**:
  A pending token. Do not modify.
- **size_t d_requiredTokens__**:
  Defines the minimum number of accepted tokens that the `parse()` function must have processed before a syntactic error can be generated.
- **int d_state__**:
  The current parsing state. Do not modify.
- **int d_token__**:
  The current token. Do not modify.
- **STYPE d_val__**:
  The semantic value of a returned token or non-terminal symbol. With non-terminal tokens it is assigned a value through the action rule's symbol **$$**. Lexical scanners may be offered the facility to assign a semantic value to this variable in parallel with a returned token. In order to allow a scanner access to **d_val__**, **d_val__**'s address should be passed to the scanner. This can be realized, for example, by defining a

member **void setSval(STYPE__ *)** in the lexical scanner, which is then called from the parser's constructor as follows:

```
        d_scanner.setSval(&d_val__);
```

Subsequently, the lexical scanner may assign a value to the parser's **d_val__** variable through the pointer to **d_val__** stored inside the lexical scanner.

Note that in some cases this approach *must* be used to make available the correct semantic value to the parser. In particular, when a grammar state defines multiple reductions, depending on the next token, the reduction's action only takes place following the retrieval of the next token, thus losing the initially matched token text. As an example, consider the following little grammar:

```
    expr:
        name
    |
        ident '(' ')'
    |
        NR
    ;

    name:
        IDENT
    ;

    ident: IDENT ;
```

Having recognized `IDENT` two reductions are possible: to `name` and to `ident`. The reduction to `ident` is appropriate when the next token is `(`, otherwise the reduction to `name` will be performed. So, the parser asks for the next token, thereby destroying the text matching `IDENT` before `ident` or `name`'s actions are able to save the text themselves. To enure the availability of the text matching `IDENT` is situations like these the *scanner* must assign the proper semantic value when it recognizes a token. Consequently the parser's `d_val__` data member must be made available to the scanner.

- **LTYPE__ d_vsp__**:
  The semantic value stack pointer. Do not modify.

6.5: Types and Variables in the Anonymous Namespace

In the file defining the `parse` function the following types and variables are defined in the anonymous namespace. These are mentioned here for the sake of completeness, and are not normally accessible to other parts of the parser.

- **char const author[]**:
  Defining the name and e-mail address of **Bisonc++**'s author.
- **ReservedTokens**:
  This enumeration defines some token values used internally by the parsing functions. They are:

  ```
      PARSE_ACCEPT   =  0,
      _UNDETERMINED_ = -2,
      _EOF_          = -1,
      _error_        = 256,
  ```

  These tokens are used by the parser to determine whether another token should be requested from the lexical scanner, and to handle error-conditions.
- **StateType**:
  This enumeration defines several moe token values used internally by the parsing functions. They are:

  ```
          NORMAL,
          ERR_ITEM,
          REQ_TOKEN,
          ERR_REQ,    // ERR_ITEM | REQ_TOKEN
          DEF_RED,    // state having default reduction
          ERR_DEF,    // ERR_ITEM | DEF_RED
          REQ_DEF,    // REQ_TOKEN | DEF_RED
          ERR_REQ_DEF // ERR_ITEM | REQ_TOKEN | DEF_RED
  ```

  These tokens are used by the parser to define the types of the various states of the analyzed grammar.
- **PI__** (Production Info):
  This `struct` provides information about production rules. It has two fields: `d_nonTerm` is the identification number of the production's non-terminal, `d_size` represents the number of elements of the productin rule.
- **static PI__ s_productionInfo**:

Used internally by the parsing function.

- **SR__** (Shift-Reduce Info):
  This `struct` provides the shift/reduce information for the various grammatic states. `SR__` values are collected in arrays, one array per grammatic state. These array, named `s_<nr>`, where tt<nr> is a state number are defined in the anonymous namespace as well. The `SR__` elements consist of two unions, defining fields that are applicable to, respectively, the first, intermediate and the last array elements.
  The first element of each array consists of (1st field) a `StateType` and (2nd field) the index of the last array element; intermediate elements consist of (1st field) a symbol value and (2nd field) (if negative) the production rule number reducing to the indicated symbol value or (if positive) the next state when the symbol given in the 1st field is the current token; the last element of each array consists of (1st field) a placeholder for the current token and (2nd field) the (negative) rule number to reduce to by default or the (positive) number of an error-state to go to when an erroneous token has been retrieved. If the 2nd field is zero, no error or default action has been defined for the state, and error-recovery is attepted.
- **STACK_EXPANSION**:
  An enumeration value specifying the number of additional elements that are added to the state- and semantic value stacks when full.
- **static SR__ s_<nr>[]**:
  Here, `<nr>` is a numerical value representing a state number. Used internally by the parsing function.
- **static SR__ *s_state[]**:
  Used internally by the parsing function.

6.6: Summary of Special Constructions for Actions

Here is an overview of special syntactic constructions that may be used inside action blocks:

- `$$`: This acts like a variable that contains the semantic value for the grouping made by the current rule. See section 5.6.4.
- `$n`: This acts like a variable that contains the semantic value for the n-th component of the current rule. See section 5.6.4.
- `$<typealt>$`: This is like `$$`, but it specifies alternative `typealt` in the union specified by the `%union` directive. See sections 5.6.1 and 5.6.2.
- `$<typealt>n`: This is like `$n` but it specifies an alternative `typealt` in the union specified by the `%union` directive. See sections 5.6.1 and 5.6.2.
- `@n`: This acts like a structure variable containing information on the line numbers and column numbers of the nth component of the current rule. The default structure

is defined like this (see section 5.5.10):

```
    struct LTYPE__
    {
        int timestamp;
        int first_line;
        int first_column;
        int last_line;
        int last_column;
        char *text;
    };
```

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

In order for the members of this structure to contain valid information, you must make sure the lexical scanner supplies this information about each token. If you need only certain fields, then the lexical scanner only has to provide those fields.

Be advised that using this or corresponding (custom-defined, see sections 5.5.11 and 5.5.9) may slow down the parsing process noticeably.

- Table of Contents
- Previous Chapter
- Next Chapter

- Table of Contents
- Previous Chapter
- Next Chapter

Chapter 7: The Bisonc++ Parser Algorithm

In this chapter the algorithm used by **bisonc++** is discussed. Generating grammar parsers of course begins with a grammar which is analyzed. The analysis consists of several phases:

- First, a set of `FIRST` tokens is determined. The `FIRST` set of a nonterminal defines all terminal tokens that can be encountered when beginning to recognize that nonterminal.
- Next, the set of `FOLLOW` tokens is determined. A `FOLLOW` set of a nonterminal defines all terminal tokens that can be encountered next, following the recognition of that nonterminal.
- Having determined the `FIRST` and `FOLLOW` sets, the grammar itself is analyzed. Starting from the start rule all possible syntactically correct derivations of the grammar are determined.
- At each state actions are defined that are eventually used by the parser to determine what to do in a particular state. These transitions are based on the next available token and may either involve a transition to another state (called a *shift*, since it involves processing a token and thus determining the next token, which involves `shifting' over the next part of the input) or it may involve a reduction (a *reduce* action), which reduces the parser's stack somewhat and optionally results in executing an *action* associated with a particular rule.
- Sometimes the analysis takes the parser generator to a state in which a choice between a *shift* or a *reduce* action must be made. This is called a *shift-reduce* conflict. Sometimes the parser generator is able to solve these conflicts itself, by looking at the *lookahead* sets for each of the continuations. If the next token is not found in the lookaheadset of either a shift or a reduce action that

particular continuation can be discarded. Likewise, two reductions may be encountered in a state (a *reduce-reduce* conflict). Here the same reasoning can be applied, causing one of the reductions to be discarded. In all other cases a true conflict is observed which somehow should be solved by the grammar designer. By default a *shift* is preferred over a *reduce* action. With a *reduce-reduce* conflict the default action is to reduce according to the reduction by the rule listed first in the grammar.

- In cases where input is not according to the rules of the grammar, a *syntactic error* is observed. Error recovery may be attempted, to allow the parser to continue parsing. This is a desirable characteristic since it provides the user of a program with a full syntactic error report, rather than one error at a time.
- Following the analysis of the grammar, code is generated and the parsing algorithm (implemented in the parser's `parse()` function) will process input according to the tables generated by the parser generator.

All the above phases will be illustrated and discussed in the next sections. Additional details of the parsing process can be found in various books about compiler construction, e.g., in Aho, Sethi and Ullman's (2003) book **Compilers** (Addison-Wesley).

In the sections below, the following grammar will be used to illustrate the various phases:

```
%token NR

%left '+'

%%

start:
    start expr
|
    // empty
;

expr:
    NR
|
    expr '+' expr
;
```

The grammar is interesting since it has a rule containing an empty alternative and since it formally suffers from a shift-reduce conflict. The shift-reduce conflict is solved by explictly assigning a priority and association to the `'+'` token.

The analysis starts by defining an additional rule, which is recognized at end of input. This rule and the rules specified in the grammar together define what is known as the *augmented grammar*. In the coming sections the symbol `$` is used to indicate `end of input'. From the above grammar the following augmented grammar is derived:

```
    1.  start:      start expr
    2.  start:      // empty
    3.  expr:       NR
    4.  expr:       expr '+' expr
    5.  start_$:    start   (input ends here)
```

bisonc++ itself will produce an analysis of any grammar it is offered when the option `--construction` is provided.

7.1: Analyzing A Grammar

7.1.1: The FIRST Sets

The `FIRST` set defines all terminal tokens that can be encountered when beginning to recognize a grammatical symbol. For each grammatical symbol (terminal and nonterminal) a `FIRST` set can be determined as follows:

- The `FIRST` set of a terminal symbol is the symbol itself.
- The `FIRST` set of an empty alternative is the empty set. The empty set is indicated by `e` and is considered an actual element of the `FIRST` set (So, a `FIRST` set could contain two elements: `'+'` *and* `e`).
- If X has a production rule `X: X1 X2 X3..., Xi, ...Xn`, then initialize `FIRST(X)` to empty (i.e., not even holding `e`). Then, for each Xi (1..n):
    - add `FIRST(Xi)` to `FIRST(X)`
    - stop when `FIRST(Xi)` does not contain `e`

If `FIRST(Xn)` does not contain `e` remove `e` from `FIRST(X)` (unless analyzing another production rule) `e` is already part of `FIRST(X)`.

When starting this algorithm, only the nonterminals need to be considered. Also, required `FIRST` sets may not yet be available. Therefore the above algorithm iterates over all nonterminals until no changes were observed. In the algorithm `$` is not considered.

Applying the above algorithm to the rules of our grammar we get:

nonterminal	rule	FIRST set
`start_$`	`start`	not yet available
`start`	`start expr`	not yet available
`start`	`// empty`	`e`
`expr`	`NR`	`NR`
`expr`	`expr '+' expr`	`NR`
changes in the next cycle:		
`start`	`start expr`	`NR e`
`start`	`// empty`	`NR e`
changes in the next cycle:		
`start_$`	`start`	`NR e`
no further changes		

7.1.2: The FOLLOW Sets

The `FOLLOW` set defines all terminal tokens that can be encountered when beginning to recognize a grammatical symbol. For each grammatical symbol (terminal and nonterminal) a `FOLLOW` set can be determined as follows (remember that `EOF` is indicated by `$`):

- `$` is put in the `FOLLOW` set of the start rule.
- For each non-empty production rule, visit all its symbols from the end of the

production rule back to to its beginning.
    - Initialize `firstSet' to FIRST(lastSymbol), where lastSymbol is the production rule's last element.
    - Otherwise, for elements E before the rule's last element:
        - If E is a nonterminal, compute FOLLOW(E) += firstset
        - If FIRST(E) contains e, compute firstset += FIRST(E)
        - If FIRST(E) does not contain e, compute firstset = FIRST(E)
- Repeatedly iterate over all production rules until until no FOLLOW set has changed. At each production rule:
    - Determine the production rule's left hand side (L).
    - If the production rule is empty, stop processing it.
    - Otherwise visit all its elements from the last back to the first element. At each element E:
        - Stop processing this production rule if E is a terminal;
        - If E is not equal to L, compute FOLLOW(E) += FOLLOW(L);
        - If FIRST(E) does not contain e, stop processing this production tule.

Applying the above algorithm to the example grammar we get:

- Part 1: FOLLOW(start_$) = $.

- Part 2:
    - Rule: start_$: start
      There is only one element, so nothing interesting is done here.
    - Rule: start: start expr
        - firstSet = FIRST(expr) = {NR}
        - FIRST(start) contains e, so FOLLOW(start) += firstSet; so FOLLOW(start) = {NR}
    - Rule: expr: NR
      There is only one element, so nothing interesting is done here.
    - Rule: expr: expr '+' expr
        - firstSet = FIRST(expr) = {NR}
        - FIRST('+') doesn not contain e, so firstSet = '+'
        - FOLLOW(expr) += firstSet, so FOLLOW(expr) = {'+'}
- At this point we have:

```
        FOLLOW(start_$) = { $ }
        FOLLOW(start)   = { NR }
        FOLLOW(expr)    = { '+' }
```

- Part 3:
    - Rule: `start_$: start`
        - `LHS: start_$`
        - `FOLLOW(start)` += `FOLLOW(start_$)`, so `FOLLOW(start) = { NR $ }`
    - Rule: `start: start expr`
        - `LSH: start`
        - `FOLLOW(expr)` += `FOLLOW(start)`, so `FOLLOW(expr) = { '+' NR $ }`
    - Remaining rules are ignored (as they are empty, only contain terminal symbols or their own `LHS`).
    - At the next cycle, no further additions are made to any of the `FOLLOW` sets, so the eventual sets become:

```
            start_$:    { $ }
            start:      { NR $ }
            expr:       { '+' NR $ }
```

7.1.3: The States

Having determined the `FIRST` and `FOLLOW` sets, **bisonc++** determines the *states* of the grammar. The analysis starts at the augmented grammar rule and proceeds until all possible states have been determined. For this analysis the concept of the *dot* symbol is used. The *dot* shows the position we are when analyzing production rules defined by a grammar. Using the provided example grammar the analysis proceeds as follows:

- State 0: `start_$ -> . start`
  At this point we haven't seen anything yet. The *dot* is before the grammar's start symbol. The above is called an *item* and the initial set of states is called the set of *kernel items*. In this state there's only one kernel item. Items are indexed, so this item receives index 0. Beyond, item indices are shown together with the items themselves. To the kernel set all production rules of non terminals immediately following the dot are added (repeatedly, so if new

rules again show nonterminals to the right of the dot, then their production rules are added too). These added rules define the non-kernel set of items. Except for the augmented grammar rule's state kernel items don't have dots at their first positions.

From the above kernel item the following non-kernel items are derived:

  - item 1: `start -> . start expr`
  - item 2: `start -> .`

From the items new states are derived: each *transition* moves the dot one postition to the right. Once the dot has reached the end of the rule, a *reduction* may take place. Following a reduction a transition based on the `LHS` of the reduced production rule is performed. This process is discussed in more detail in section 7.1.6.

Looking at the state's items, two actions are detected:

  - On `start`, to a state in which `start` has been seen (state 1)
  - By default, reduce by the rule `start -> .`

- State 1: kernel items:
  - item 0: `start_$ -> start .`
  - item 1: `start -> start . expr`

  Non-kernel items:
  - item 2: `expr -> . NR`
  - item 3: `expr -> . expr '+' expr`

  This state becomes the *accepting* state: if in this state EOF is reached, the `start_$` rule has been recognized, and so the input could be recognized by the grammar. Other transitions are possible to, though:
  - On `expr` to state 2
  - On NR to state 3

- State 2: kernel items:
  - item 0: `start -> start expr .`
  - item 1: `expr -> expr . '+' expr`

  No non-kernel items need to be added to this state. It has the following transitions:
  - On `'+'` to state 4
  - Or reduce to `start` according to its first item (removing two elements

from the parser's stack).

- State 3: kernel items:
    - item 0: `expr -> NR .`

  In this state only one action is possible: a reduction to `expr` (removing one element from the parser's stack).

- State 4: kernel item:
    - item 0: `expr -> expr '+' . expr`

  Two non-kernel items are added:
    - item 1: `expr -> . NR`
    - item 2: `expr -> . expr '+' expr`

  This state has the following transitions:
    - On `expr` to state 5
    - On `NR` to state 3

  The last action is interesting in that it returns to a previously defined state. That's OK, but it requires that a state is reached having exactly the matching number of kernel items. So, if state 3 would have had yet anoher state, a new state would have been constructed to where this state would transit on encountering a `NR` token.

- State 5: kernel items:
    - item 0: `expr -> expr '+' expr .`
    - item 1: `expr -> expr . '+' expr`

  In this state two actions are possible:
    - On `'+'` to state 4
    - Or reduce to `expr` according to its first item (removing three elements from the parser's stack).

  With the current grammar it turns out (and the reason why this is so will be discussed in the next section) that the first action will never take place: in this state there will always be a reduction.

7.1.4: The Lookahead Sets

7.1.4.1: Preamble

The **bisonc++** 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 parser's private data member `d_token`. However, this data member is not normally modified by member functions not generated by **bisonc++**. See section 6.6.

In the previous section it was stated that although state 5 has two possible actions, in fact only one is used. This is a direct consequence of the `%left '+'` specification, as will be discussed in this section.

When analyzing a grammar all states that can be reached from the augmented start rule are determined. In state 5 **bisonc++** is confronted with a choice: either a shift on `'+'` or a reduction according to the item `` `expr -> expr '+' expr .' ``. What choice will **bisonc++** make?

At this point the fact that **bisonc++** implements a parser for a *Look Ahead Left to Right (1)* (LALR(1)) grammar becomes relevant. **Bisonc++** will use *computed lookahead sets* to determine which alternative to select, when confronted with a choice. The lookahead set can be used to favor one transition over the other when eventually generating the tables for the parsing function.

Sometimes the lookahead sets allow **bisonc++** simply to remove one action from the set of possible actions. When **bisonc++** is called to process the example grammar while specifying the `--construction` option state 5 will *only* show the reduction and *not* the shifting action: it has removed that alternative from the action set. This is a direct consequence of a hidden shift-reduce conflict in the grammar: in state 5 the choice is between shifting or reducing when encountering a `'+'` token. As we'll see in this section, `'+'` is in the lookahead set of the reduce-item, and thus **bisonc++** is faced with a conflict: what to do on `'+'`?

In this case the grammar designer has provided **bisonc++** with a way out: the `%left` directive tells **bisonc++** to favor a reduction over a shift, and so it removed `expr -> expr . '+' expr` from its set of actions in state 5.

In this section we'll have a look at the way **bisonc++** determines lookahead (LA) sets.

- For each item in each state, **bisonc++** determines the items whose LA sets depend on that particular item.

To determine which items have LA set that depend on a particular item the symbol following the item's dot position is inspected. If it's a nonterminal, then all items whose LHSs are equal to that nonterminal depend on the item being considered.

Inspecting the states of our example grammar, using offsets (0-based) to indicate their items, the following dependencies are observed:

  - State 0: items 1 and 2 depend on items 0 and 1.
  - State 1: items 2 and 3 depend on items 1 and 3.
  - State 2: no dependencies
  - State 3: no dependencies
  - State 4: items 1 and 2 depend on items 0 and 2.
  - State 5: no dependencies

- Then, for each item it is determined what the next state will be if that item is actually used in a transition. Items representing a reduction are not considered here. This results in the following:
  - State 0: items 0 and 1: transition to state 1
  - State 1: items 1 and 3: transition to state 2; item 2: transition to state 3
  - State 2: item 1: transition to state 4
  - State 3: no transitions
  - State 4: items 1 and 2: transition to state 5; item 1: transition to state 3
  - State 5: item 1: transition to state 4

- Next, LA propagation takes place:
  - The LA set of the augmented start rule is initialized to `$`. All other LA sets are initiaized to an empty set.

  - For each state all items are considered. Dependent items receive their LA sets from the items on which they depend (called their `parent item').

    The LA sets of the dependent items are equal to the `FIRST` set of the subrule of their parent items, starting at the symbol following their parent item's dot positions.

  - If a subrule's `FIRST` set contains `e`, then that item's LA set is added to the subrule's LA set, removing the e.

- If items depend on multiple parents, then the LA sets of those items are the union of the LA sets as determined for each of their parents.
- Once the LA sets of items in a state have been determined, then the LA sets of items transiting to other states are added to the LA sets of the corresponding kernel items in these other states (each item in an originating state only modifies one kernel item in a destination state).
- Since transitions may return to earlier states, determining LA sets is implemented as an iterative process, terminating when all LA sets have stabilized.

Applying the above algorithm to the example grammar we get:

- State 0:
    - item 0: `start_$ -> . start` LA: `{$}`
      Add the LA set (`{$}`) to the items resulting from the `start` productions
      Next state from here: 1
    - item 1: `start -> . start expr` LA: `{$}`
      Next state from here: 1
    - item 2: `start -> .` LA: `{$}`
    - From item 1: once again consider the `start` rules, adding `FIRST(expr) = {NR}` to the LA sets of those rules:
        - item 1: `start -> . start expr` LA: `{$ NR}`
        - item 2: `start -> .` LA: `{$ NR}`
- State 1:
    - item 0: `start_$ -> start .` inherits LA: `{$}` from item 0, state 0.
    - item 1: `start -> start . expr` inherits LA: `{NR $}` from item 1, state 0.
      Add the LA set (`{NR $}`) to the items resulting from the `expr` productions
      Next state from here: 2

    - item 2: `expr -> . NR` LA: `{NR $}`
      Next state from here: 3

    - item 3: `expr -> . expr '+' expr` LA: `{NR $}`
      Next state from here: 2

    - From item 2: once again consider the `expr` rules, adding `FIRST('+') = {'+'}` to the LA sets of those rules:
        - item 2: `expr -> . NR` LA: `{+ NR $}`
        - item 3: `expr -> . expr '+' expr` LA: `{+ NR $}`

  - State 2:
    - item 0: `start -> start expr .` inherits LA: `{NR $}` from item 1, state 1.
    - item 1: `expr -> expr . '+' expr` inherits LA: `+ NR $` from item 3, state 1.
      Next state from here: 4

  - State 3:
    - item 0: `expr -> NR .` inherits LA: `{+ NR $}` from item 2, state 1.

  - State 4:
    - item 0: `expr -> expr '+' . expr` inherits LA: `{+ NR $}` from item 2, state 2.
      Add the LA set (`{+ NR $}`) to the items resulting from the `expr` productions
      Next state from here: 5
    - item 1: `expr -> . NR` LA: `{+ NR $}`
      Next state from here: 3. Since item 0 in state 3 already has a LA set containing all elements of the current LA set, no further modifications need to be propagated.

    - item 2: `expr -> . expr '+' expr` LA: `+ NR $`
      Next state from here: 5

    - From item 2 all `expr` production rules need to be considered again. This time no LA sets change, so the LA sets of all items in this state have been determined.

- State 5:
    - item 0: `expr -> expr '+' expr .` inherits LA: `+ NR $` from item 0, state 4
    - item 1: `expr -> expr . '+' expr` inherits LA: `+ NR $` from item 2, state 4.
      Next state: 4. Since item 0 in state 4 already has a LA set containing all elements of the current LA set, no further modifications need to be propagated.

Once again, look at state 5. In this state, item 0 calls for a reduction on tokens `'+', NR` or `EOF`. However, according to item 1 a *shift* must be performed when the next token is a `'+'`. This choice represents a shift-reduce conflict which is reported by **bisonc++** unless special actions are taken. One of the actions is to tell **bisonc++** what to do. A `%left` directive tells **bisonc++** to prefer a reduction over a shift when encountering a shift-reduce conflict for the token(s) mentioned with the `%left` directive. Analogously, a `%right` tells **bisonc++** to perform a shift rather than a reduction.

Since a `%left '+'` was specified, **bisonc++** drops the shift alternative, and a listing of the grammar's construction process (using the option `--construction`) shows for state 5:

```
State 5:
0: [P4 3] expr -> expr '+' expr  .   { NR '+' <EOF> }  1, () -1
1: [P4 1] expr -> expr  . '+' expr   { NR '+' <EOF> }  0, () 0
  0:
  Reduce item(s): 0
```

The shift action (implied by item 1) is not reported.

7.1.5: The Final Transition Tables

7.1.5.1: Preamble

The member function `parse()` 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 current (not yet processed) token are looked up in a table. This table entry can say *Shift the token*. This also specifies a new parser state, which is then 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 that the rule's grouping becomes the `next token' to be considered. That `next token' is then used in combination with the state then at the stack's top, to determine the next state to consider. This (next) state is then again pushed on the stack, and a new token is requested from the lexical scanner, and the process repeats itself.

There are two special situations the parsing algorithm must consider:

- First, the lexical scanner may reach *end-of-input*. If the current state on top of the parser's stack is the start-state, then the reduction (which is called for in this situation) is in fact the (successful) end of the parsing process, and `parse()` returns the value 0, indicating a successful parsing.
- There is one other alternative: the table can say that the token is erroneous in the current state. This causes error processing to begin (see chapter 8).

Once **bisonc++** has successfully analyzed the grammar it generates the tables that are used by the parsing function to parse input according to the provided grammar. Each state results in a *state transition table*. For the example grammar used so far there are five states. Each table consists of rows having two elements. The meaning of the elements depends on their position in the table.

- For the *first* row,
  - the first element indicates the *type* of the state. The following types are recognized:

NORMAL	Despite its name, it's not used
ERR_ITEM	The state allows error recovery
REQ_TOKEN	The state requires a token (which may already be available)
ERR_REQ	combines ERR_ITEM and REQ_TOKEN

DEF_RED	This state has a default reduction
ERR_DEF	combines ERR_ITEM and DEF_RED
REQ_DEF	combines REQ_TOKEN and DEF_RED
ERR_REQ_DEF	combines ERR_ITEM, REQ_TOKEN and DEF_RED

  - the second element indicates the index of the table's last element.
- For the *last* row,
  - the first element stores the current token (it is not used when the option `--thread-safe` was specified)
  - the second element defines the action to perform. A positive value indicates a shift to the indicated state; a negative value a reduction according to the indicated rule number, disregarding its sign (note that it's rule *number*, rather than rule *offset*; zero indicates the input is accepted as correct according to the parser's grammar.)
  - For all intermediate remaining rows: the first element stores the value of a required token for the action specified in the second element, similar to the way an action is specified in the last row. Symbolic values (like `PARSE_ACCEPT` rather than 0) may be used as well.

Here are the tables defining the five states of the example grammar as they are generated by **bisonc++** in the file containing the parsing function:

```
SR__ s_0[] =
{
    { { DEF_RED}, {  2} },
    { {     258}, {  1} }, // start
    { {       0}, { -2} },

};

SR__ s_1[] =
{
    { { REQ_TOKEN}, {           4} },
    { {       259}, {           2} }, // expr
    { {       257}, {           3} }, // NR
    { {     _EOF_}, { PARSE_ACCEPT} },
    { {         0}, {           0} },

};
```

```
    SR__ s_2[] =
    {
        { { REQ_DEF}, {  2} },
        { {      43}, {  4} }, // '+'
        { {       0}, { -1} },

    };

    SR__ s_3[] =
    {
        { { DEF_RED}, {  1} },
        { {       0}, { -3} },

    };

    SR__ s_4[] =
    {
        { { REQ_TOKEN}, { 3} },
        { {       259}, { 5} }, // expr
        { {       257}, { 3} }, // NR
        { {         0}, { 0} },

    };

    SR__ s_5[] =
    {
        { { REQ_DEF}, {  1} },
        { {       0}, { -4} },

    };
```

7.1.6: Processing Input

Bisonc++ implements the parsing function in the member function `parse()`. This function obtains its tokens from the member `lex()` and processes all tokens until a syntactic error, a non-recoverable error, or the end of input is encountered.

The algorithm used by `parse()` is the same, irrespective of the used grammar. In fact, the `parse()` member's behavior is completely determined by the tables generated by **bisonc++**.

The parsing algorithm is known as the *shift-reduce* (S/R) algorithm, and it allows `parse()` to perform two actions while processing series of tokens:

- When a token is received in a state in which that token is required for a transition to another state (e.g., a `NR` token is observed in state 1 of the example's grammar) a transition to state 3 is performed.
- When a state is reached which calls for a (default) reduction (e.g., state 3 of the example's grammar) a reduction is performed.

The parsing function maintains two stacks, which are manipulated by the above two actions: a state stack and a value stack. These stacks are not accessible to the parser: they are private data structures defined in the parser's base class. The parsing member `parse()` may use the following member functions to manipulate these stacks:

- `push__(stateIdx)` pushes `stateIdx` on the state stack and pushes the current semantic value (i.e., `LTYPE_ d_val__`) on the value stack;
- `pop__(size_t count = 1)` removes `count` elements from the two stacks;
- `top__()` returns the state currently on top of the state stack;

Apart from the state- and semantic stacks, the S/R algorithm itself sometimes needs to push a token on a two-element stack. Rather than using a formal stack, two variables (`d_token__` and `d_nextToken__`) are used to implement this little token-stack. The member function `pushToken__()` pushes a new value on the token stack, the member `popToken__()` pops a previously pushed value from the token stack. At any time, `d_token__` contains the topmost element of the token stack.

The member `nextToken()` determines the next token to be processed. If the token stack contains a value it is returned. Otherwise, `lex()` is called to obtain the next token to be pushed on the token stack.

The member `lookup()` looks up the current token in the current state's `SR__` table. For this a simple linear search algorithm is used. If searching fails to find an action for the token an `UNEXPECTED_TOKEN__` exception is thrown, which starts the error recovery. If an action was found, it is returned.

Rules may have actions associated with them. These actions are executed when a grammatical rule has been completely recognized. This is always at the end of a rule: mid-rule actions are converted by **bisonc++** into pseudo nonterminals, replacing mid-rule action blocks by these pseudo nonterminals. The pseudo nonterminals show up in the verbose grammar output as rules having LHSs starting with #. So, once a rule has been recognized its action (if defined) is executed. For this the member function `executeAction()` is available.

Finally, the token stack can be cleared using the member `clearin()`.

Now that the relevant support functions have been introduced, the S/R algorithm itself turns out to be a fairly simple algorithm. First, the parser's stack is initialized with state 0 and the token stack is cleared. Then, in a never ending loop:

- If a state needs a token (i.e., `REQ_TOKEN` has been specified for that state), `nextToken()` is called to obtain the next token;
- From the token and the current state `lookup()` determines the next action;
- If a shifting action was called for the next state is pushed on the stack and the token is popped off the token stack.
- If a reduction was called for that rule's action block is executed followed by a reduction of the production rule (performed by `reduce__()`): the semantic and state stacks are reduced by the number of elements found in that production rule, and the production rule's LHS is pushed on the token stack
- If the state/token combination indicates that the input is accepted (normally: when `EOF` is encountered in state 1) then the parsing function terminates, returning 0.

The following table shows the S/R algorithm in action when the example grammar is given the input `3 + 4 + 5`. The first column shows the (remaining) input, the second column the current token stack (with `-` indicating an empty token stack), the third column the state stack. The fourth column provides a short description. The leftmost elements of the stacks represent the tops of the stacks. The information shown below is also (in more elaborate form) shown when the `--debug` option is provided to **Bisonc++** when generating the parsing function.

remaining input	token stack	state stack	description
`3 + 4 + 5`	`-`	`0`	initialization

`3 + 4 + 5`	`start`	`0`	reduction by rule 2
`3 + 4 + 5`	`-`	`1 0`	shift `start'
`+ 4 + 5`	`NR`	`1 0`	obtain NR token
`+ 4 + 5`	`-`	`3 1 0`	shift NR
`+ 4 + 5`	`expr`	`1 0`	reduction by rule 3
`+ 4 + 5`	`-`	`2 1 0`	shift `expr'
`4 + 5`	`+`	`2 1 0`	obtain `+' token
`4 + 5`	`-`	`4 2 1 0`	shift `+'
`+ 5`	`NR`	`4 2 1 0`	obtain NR token
`+ 5`	`-`	`3 4 2 1 0`	shift NR
`+ 5`	`expr`	`4 3 1 0`	reduction by rule 3
`+ 5`	`-`	`5 4 3 1 0`	shift `expr'
`5`	`+`	`5 4 3 1 0`	obtain `+' token
`5`	`expr +`	`1 0`	reduction by rule 4
`5`	`+`	`2 1 0`	shift `expr'
`5`	`-`	`4 2 1 0`	shift '+'
	`NR`	`4 2 1 0`	obtain NR token
	`-`	`3 4 2 1 0`	shift NR
	`expr`	`4 2 1 0`	reduction by rule 3
	`-`	`5 4 2 1 0`	shift `expr'
	`EOF`	`5 4 2 1 0`	obtain EOF
	`expr EOF`	`1 0`	reduction by rule 4
	`EOF`	`2 1 0`	shift `expr'
	`start EOF`	`2 1 0`	reduction by rule 1
	`EOF`	`1 0`	shift `start'
	`EOF`	`1 0`	ACCEPT

7.2: Shift/Reduce Conflicts

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

```
    if_stmt:
        IF '(' expr ')' stmt
    |
        IF '(' expr ')' stmt ELSE stmt
    ;
```

Here we assume that `IF` and `ELSE` are terminal symbols for specific keywords, and that `expr` and `stmnt` are defined non-terminals.

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. **Bisonc++** is designed to resolve these conflicts by *implementing* a 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) if (y) then win(); else lose();

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

But if the parser would perform a *reduction* whenever possible rather than a *shift*, the result would be to attach the *else-clause* to the outermost if-statement, making these two inputs equivalent:

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

    if (x)
    {
        if (y) win();
    }
    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 **bisonc++** accomplishes by implementing a shift rather than a reduce. This particular ambiguity was first encountered in the specifications of Algol 60 and is called the *dangling else* ambiguity.

To avoid warnings from **bisonc++** about predictable, legitimate shift/reduce conflicts, use the `%expect n` directive. There will be no warning as long as the number of shift/reduce conflicts is exactly `n`. See section 5.5.5.

The definition of `if_stmt` above is solely to blame for the conflict, but the plain `stmnt` rule, consisting of two recursive alternatives will of course never be able to match actual input, since there's no way for the grammar to eventually derive a sentence this way. Adding one non-recursive alternative is enough to convert the grammar into one that *does* derive sentences. Here is a complete **bisonc++** input file that actually manifests the conflict:

```
%token IF ELSE VAR

%%

stmt:
    VAR ';'
|
    IF '(' VAR ')' stmt
|
    IF '(' VAR ')' stmt ELSE stmt
;
```

Looking again at the dangling else problem note that there are multiple ways to handle `stmnt` productions. Depending on the particular input that is provided it could either be reduced to a `stmt` or the parser could continue to consume input by processing an ELSE token, eventually resulting in the recognition of IF '(' VAR ')' stmt ELSE stmt as a stmt.

There is little we can do but resorting to %expect to handle the dangling else problem. The default handling is what most people intuitively expect and so in this case using %expect 1 is an easy way to prevent **bisonc++** from reporting a shift/reduce conflict. But shift/reduce conflicts are most often solved by specifying disambiguating rules specifying priorities or associations, usually in the context of arithmetic expressions, as discussed in the next sections.

However, shift-reduce conflicts can also be observed in grammars where a state contains items that could be reduced to a certain non-terminal and items in which a shift is possible in an item of a production rule of a completely different non-terminal. Here is an example of such a grammar:

```
%token  ID
%left  '-'
%left  '*'
%right UNARY

%%

expr:
    expr '-' term
|
    term
;

term:
    term '*' factor
|
    factor
;

factor:
   '-' expr %prec UNARY
|
    ID
;
```

Why these grammars show shift reduce conflicts and how these are solved is discussed in the next section.

7.3: Operator Precedence

Shift/reduce conflicts are frequently encountered in grammars specifying rules of arithmetic expressions. Here shifting is not always the preferred resolution; the **bisonc++** directives for operator precedence allow you to specify when to shift and when to reduce. How and when to do so is discussed next.

7.3.1: 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 **bisonc++** 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, e.g., 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 results in right-associativity.

7.3.2: Specifying Operator Precedence

Bisonc++ allows you to specify these choices with the operator precedence directives `%left` and `%right`. Each such directive contains a list of tokens, which are operators whose precedence and associativity is being declared. The `%left` directive makes all those operators left-associative and the `%right` directive 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'. Actually, `%nonassoc` is not currently (0.98.004) punished that way by **bisonc++**. Instead, `%nonassoc` and `%left` are handled identically.

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

7.3.3: 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.)

7.3.4: How Precedence Works

The first effect of the precedence directives 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 section 7.4).

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 section 9) shows 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.

7.3.5: Rule precedence

Consider the following (somewhat peculiar) grammar:

```
    %token  ID
    %left  '-'
    %left  '*'
    %right UNARY
```

```
%%

expr:
    expr '-' term
|
    term
;

term:
    term '*' factor
|
    factor
;

factor:
   '-' expr %prec UNARY
|
    ID
;
```

Even though operator precedence and association rules are used the grammar still displays a shift/reduce conflict. One of the grammar's states consists of the following two items:

```
0: expr -> term  .
1: term -> term  . '*' factor
```

and **bisonc++** reduces to item 0, dropping item 1 rather than shifting a `'*'` and proceeding with item 0.

When considering states where shift/reduce conflicts are encountered the `shiftable' items of these states shift when encountering terminal tokens that are also in the follow sets of the reducible items of these states. In the above example item 1 shifts when `'*'` is encountered, but `'*'` is also an element of the set of lookahead tokens of item 0. **Bisonc++** must now decide what to do. In cases we've seen earlier **bisonc++** could make the decision because the reducible item itself had a well known precedence. The precedence of a reducible item is defined as the precedence of the left-hand side non-terminal of the production rule to which the reducible item belongs. Item 0 in the above example is an item of the rule `expr -> term`.

The precedence of a production rule is defined as follows:

- If `%prec` is used then the precedence of the production rule is equal to the precedence of the terminal that is specified with the `%prec` directive;
- If `%prec` is not used then the production rule's precedence is equal to the precedence of the first terminal token that is used in the production rule;
- In all other cases the production rule's precedence is set to the maximum possible precedence.

Since `expr -> term` does not contain a terminal token and does not use `%prec`, its precedence is the maximum possible precedence. Consequently in the above state the shift/reduce conflict is solved by *reducing* rather than shifting.

Some final remark as to why the above grammar is peculiar. It is peculiar as it combines precedence and association specifying directives with auxiliary non-terminals that may be useful conceptually (or when implementing an expression parser `by hand') but which are not required when defining grammars for **bisonc++**. The following grammar does not use `term` and `factor` but recognizes the same grammar as the above `peculiar' grammar without reporting any shift/reduce conflict:

```
%token  ID
%left  '-'
%left  '*'
%right UNARY

%%

expr:
    expr '-' expr
|
    expr '*' expr
|
   '-' expr %prec UNARY
```

```
    |
        ID
    ;
```

7.4: 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 **bisonc++** precedence directives, %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 section 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

7.5: 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: %stype char * %token WORD

%%

sequence: // empty { cout << "empty sequence\n"; } | maybeword | sequence WORD { cout << "added word " << $2 << endl; } ;

maybeword: // empty { cout << "empty maybeword\n"; } | WORD { cout << "single word " << $1 << endl; } ;

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.

Bisonc++ 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 ;
```

7.6: 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, **bisonc++**, 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 **bisonc++** 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. **Bisonc++** 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, **bisonc++** 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
    |
        ID BOGUS        // This rule is never used.
    ;
```

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 the parser's member function `lex()`, 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
    ;
```

- Table of Contents
- Previous Chapter
- Next Chapter

- Table of Contents
- Previous Chapter
- Next Chapter

Chapter 8: Error Recovery

Usually it is not 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. Such errors violate the grammar for which the parser was constructed and are called *syntactic errors*. Other types of errors are called *semantical errors*: here the intended *meaning* of the language is not observed. For example, a division by too small a numeric constant (e.g., 0) may be detected by the parser *compile time*. In general, what *can* be detected compile time should not left for the run-time to detect, and so the parser should flag an error when it detects a division by a very small numerical constant. **Bisonc++**'s parsers may detect both syntactic *and* semantical errors. Syntactical errors are detected automatically, while the parser performs its parsing-job, semantical errors must explicitly be defined when the grammar is constructed. The following sections cover the way **Bisonc++**'s parser may handle syntactic errors and semantical errors, respectively.

8.1: Syntactical Error Recovery

In a simple interactive command parser where each input is one line, it may be sufficient to allow `parse()` to return `PARSE_ABORT` on error and have the caller ignore the rest of the input line when that happens (and then call `parse()` again). But this is inadequate for a compiler, because it forgets all the syntactic context leading up to the error. A syntactic 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.

It is possible to specify how to recover from a syntactic error by writing rules recognizing the special token `error`. This is a terminal symbol that is always defined (it must *not* be declared) and is reserved for error handling. The **bisonc++** parser generates an `error` token whenever a syntactic error is detected; if a rule was provided recognizing this token in the current context, the parse can continue. For example:

```
    statements:
        // empty
    |
        statements '\n'
    |
        statements expression '\n'
    |
        statements error '\n'
```

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

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

bisonc++, however, can force the situation to fit the rule, by *discarding* part of the semantic context and part of the input. When a (syntactic) error occurs the parsing algorithm will try to recover from the error in the following way: First it discards states from the stack until it encounters a state in which the `error` token is acceptable (meaning that the subexpressions already parsed are discarded, back to the last complete `statements`). At this point the error token is shifted. Then, if the available look-ahead token is not acceptable to be shifted next, the parser continues to read tokens and to discard them until it finds a token which *is* acceptable. I.e., a token which *can* follow an `error` token in the current state. In this example, **bisonc++** reads and discards input until the next newline was read 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:

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

Another useful recovery strategy is 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, generating another, spurious error message:

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

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

To prevent an outpouring of error messages, the parser may be configured in such a way that no error message will be generated for another syntactic error that happens shortly after the first. E.g., only after three consecutive input tokens have been successfully shifted error messages will be generated again. This configuration is currently not available in **bisonc++**'s parsers.

Note that rules using the `error` token may have actions, just as any other rules can.

The token causing an error is re-analyzed immediately when an error occurs. If this is unacceptable, then the member function `clearin()` may be called to skip this token. The function can be called by any member function of the Parser class. 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 token ought to be discarded using `clearin()`.

8.1.1: Error Recovery

Bisonc++ implements a simple error recovery mechanism. When the `lookup()` function cannot find an action for the current token in the current state it throws an `UNEXPECTED_TOKEN__` exception.

This exception is caught by the parsing function, calling the `errorRecovery()` member function. By default, this member function will terminates the parsing process. The non-default recovery procedure is available once an `error` token is used in a production rule. When the parsing process throws **UNEXPECTED_TOKEN__** the recovery procedure is started (i.e., it is started whenever a syntactical error is encountered or `ERROR()` is called).

The recovery procedure consists of

- looking for the first state on the state-stack having an error-production, followed by:
- handling all state transitions that are possible without retrieving a terminal token.
- then, in the state requiring a terminal token and starting with the initial unexpected token (3) all subsequent terminal tokens are ignored until a token is retrieved which is a continuation token in that state.

If the error recovery procedure fails (i.e., if no acceptable token is ever encountered) error recovery falls back to the default recovery mode (i.e., the parsing process is terminated).

Not all syntactic errors are always reported: the option --required-tokens can be used to specify the minimum number of tokens that must have been successfully processed before another syntactic error will be reported (and counted).

The option --error-verbose may be specified to obtain the contents of the state stack when a syntactic error is reported.

The example grammar may be provided with an `error` production rule:

```
    %token NR

    %left '+'

    %%

    start:
        start expr
    |
        // empty
    ;

    expr:
        error
    |
        NR
    |
        expr '+' expr
    ;
```

The resulting grammar has one additional state (handling the error production) and one state in which the `ERR_ITEM` flag has been set. When and error is encountered, this state will obtain tokens until a token having a valid continuation is obtained, after which normal processing continues.

The following output from the `parse()` function, generated by **bisonc++** using the `--debug` option illustrates error recovery for the above grammar, entering the input

```
    a
    3 + a
```

The program defining the parser and calling the parsing member was:

```
#include "Parser.h"
```

```
int main()
{
    Parser parser;

    parser.parse();
}
```

For this example the following implementation of the `lex()` member was used:

```
    int Parser::lex()
    {
        std::string word;

        std::cin >> word;
        if (std::cin.eof())
            return 0;
        if (isdigit(word[0]))
            return NR;

        return word[0];
    }
```

8.1.1.1: Error recovery --debug output

```
    parse(): Parsing starts
    push(state 0)
    ==
    lookup(0, `_UNDETERMINED_'): default reduction by rule 2
    executeAction(): of rule 2 ...
    ... action of rule 2 completed
    pop(0) from stack having size 1
    pop(): next state: 0, token: `start'
    reduce(): by rule 2 to N-terminal `start'
    ==
    lookup(0, `start'): shift 1 (`start' processed)
    push(state 1)
    ==
a
Syntax error
```

```
    nextToken(): using `a' (97)
    lookup(1, `a' (97)): Not found. Start error recovery.
    errorRecovery(): 1 error(s) so far. State = 1
    errorRecovery(): state 1 is an ERROR state
    lookup(1, `_error_'): shift 3 (`_error_' processed)
    push(state 3)
    lookup(3, `a' (97)): default reduction by rule 3
    pop(1) from stack having size 3
    pop(): next state: 1, token: `expr'
    reduce(): by rule 3 to N-terminal `expr'
    errorRecovery() REDUCE by rule 3, token = `expr'
    lookup(1, `expr'): shift 2 (`expr' processed)
    push(state 2)
    errorRecovery() SHIFT state 2, continue with `a' (97)
    lookup(2, `a' (97)): default reduction by rule 1
    pop(2) from stack having size 3
    pop(): next state: 0, token: `start'
    reduce(): by rule 1 to N-terminal `start'
    errorRecovery() REDUCE by rule 1, token = `start'
    lookup(0, `start'): shift 1 (`start' processed)
    push(state 1)
    errorRecovery() SHIFT state 1, continue with `a' (97)
    lookup(1, `a' (97)): Not found. Continue error recovery.
 3+a
    nextToken(): using `NR'
    lookup(1, `NR'): shift 4 (`NR' processed)
    push(state 4)
    errorRecovery() SHIFT state 4, continue with `_UNDETERMINED_'
    errorRecovery() COMPLETED: next state 4, no token yet
    ==
    lookup(4, `_UNDETERMINED_'): default reduction by rule 4
    executeAction(): of rule 4 ...
    ... action of rule 4 completed
    pop(1) from stack having size 3
    pop(): next state: 1, token: `expr'
    reduce(): by rule 4 to N-terminal `expr'
    ==
    lookup(1, `expr'): shift 2 (`expr' processed)
    push(state 2)
    ==
 [input terminated here]
    nextToken(): using `_EOF_'
    lookup(2, `_EOF_'): default reduction by rule 1
    executeAction(): of rule 1 ...
    ... action of rule 1 completed
    pop(2) from stack having size 3
    pop(): next state: 0, token: `start'
    reduce(): by rule 1 to N-terminal `start'
    ==
    lookup(0, `start'): shift 1 (`start' processed)
    push(state 1)
    ==
    lookup(1, `_EOF_'): ACCEPT
    ACCEPT(): Parsing successful
    parse(): returns 0
```

8.2: Semantical Error Recovery

Semantical error recovery once again requires judgment on the part of the grammar-writer. For example, an assignment expression may be syntactically defined as

```
    expr '=' expr
```

The left-hand side must be a so-called *lvalue*. An *lvalue* is simply an addressable location, like a variable's identifier, a dereferenced pointer expression or some other address-expression. The right-hand side is a so-called *rvalue*: this may be any value: any expression will do.

A rule like the above leaves room for many different semantical errors:

- Since the rule states `expr` at its left-hand side, *any* expression will be accepted by the parser. E.g.,

  ```
      3 = 12
  ```

  So, the action associated with this rule should *check* whether the left-hand side is actually an lvalue. If not, a *semantical* error should be reported;
- In a typed language (like **C++**), not all assignments are possible. E.g., it is not

acceptable to assign a **std:string** value to a **double** variable. When conflicting types are used, a *semantical* error should be reported;
- In a language requiring variables to be defined or declared before they are used (like **C++**) the parser should check whether a variable is actually defined or declared when it is used in an expression. If not, a *semantical* error should be reported

A parser that should be able to detect semantic errors will normally use a counter counting the number of semantic errors, e.g., `size_t d_nSemanticErrors`. It may be possible to test this counter's value once the input has been parsed, calling `ABORT()` (see section 6.3) if the counter isn't zero anymore. When the grammar's start symbol itself has multiple alternatives, it is probably easiest to augment the grammar with an additional rule, becoming the augmented grammar's start symbol which simply calls the former start symbol. For example, if `input` was the name of the original start-symbol, augment the grammar as follows to ensure a **PARSE_ABORT** return value of the `parse()` member when either syntactic or semantical errors were detected:

```
    semantic_input:                 // new start-symbol
        input
        {
            if (d_nSemanticErrors)  // return PARSE_ABORT
                ABORT();            // on semantic errors too.
        }
```

Returning from the parser's `parse()` member the number of syntactic and semantical errors could then be printed, whereupon the program might terminate.

- Table of Contents
- Previous Chapter
- Next Chapter

- Table of Contents
- Previous Chapter

Chapter 9: Invoking Bisonc++

9.1: Bisonc++ options

Where available, single letter options are listed between parentheses beyond their associated long-option variants. Single letter options require arguments if their associated long options require arguments. Options affecting the class header or implementation header file are ignored if these files already exist. Options accepting a `filename' do not accept path names, i.e., they cannot contain directory separators (`/`); options accepting a 'pathname' may contain directory separators.

Some options may generate warnings. This happens when an option conflicts with the contents of a file which **bisonc++** cannot modify (e.g., a parser class header file exists, but doesn't define a name space, but a `--namespace` option was provided). In those cases the option is ignored, and hand-editing may then be required to effectuate the option.

- **--analyze-only** (-A)
  Only analyze the grammar. No files are (re)written. This option can be used to test the grammatic correctness of modification `in situ', without overwriting previously generated files. If the grammar contains syntactic errors only syntax analysis is performed.

- **--baseclass-header**=`filename` (**-b**)
  `Filename` defines the name of the file to contain the parser's base class. This class defines, e.g., the parser's symbolic tokens. Defaults to the name of the parser class plus the suffix `base.h`. It is generated, unless otherwise indicated (see `--no-baseclass-header` and `--dont-rewrite-baseclass-header` below).

  A warning is issued if this option is used and an already existing parser class header file does not contain `#include "filename"`.

- **--baseclass-preinclude**=`pathname` (**-H**)
  `Pathname` defines the path to the file preincluded in the parser's base-class header. This option is needed in situations where the base class header file refers to types

which might not yet be known. E.g., with polymorphic semantic values a `std::string` value type might be used. Since the `string` header file is not by default included in `parserbase.h` we somehow need to inform the compiler about this and possibly other headers. The suggested procedure is to use a pre-include header file declaring the required types. By default `` `header' `` is surrounded by double quotes: `#include "header"` is used when the option `-H header` is specified. When the argument is surrounded by pointed brackets `#include <header>` is included. In the latter case, quotes might be required to escape interpretation by the shell (e.g., using `-H '<header>'`).

- **--baseclass-skeleton**=`pathname` (**-B**)
  `Pathname` defines the path name to the file containing the skeleton of the parser's base class. It defaults to the installation-defined default path name (e.g., `/usr/share/bisonc++/` plus `bisonc++base.h`).

- **--class-header**=`filename` (**-c**)
  `Filename` defines the name of the file to contain the parser class. Defaults to the name of the parser class plus the suffix `.h`

  A warning is issued if this option is used and an already existing implementation header file does not contain `#include "filename"`.

- **--class-name** `className`
  Defines the name of the **C++** class that is generated. If neither this option, nor the `%class-name` directory is specified, then the default class name (`Parser`) is used.

  A warning is issued if this option is used and an already existing parser-class header file does not define `` class `className' `` and/or if an already existing implementation header file does not define members of the class `` `className' ``.

- **--class-skeleton**=`pathname` (**-C**)
  `Pathname` defines the path name to the file containing the skeleton of the parser class. It defaults to the installation-defined default path name (e.g., `/usr/share/bisonc++/` plus `bisonc++.h`).

- **--construction**
  Details about the construction of the parsing tables are written to the same file as written by the `--verbose` option (i.e., `<grammar>.output`, where `<grammar>` is the input file read by **bisonc++**. This information is primarily useful for developers. It augments the information written to the verbose grammar output file, generated by the `--verbose` option.

- **--debug**
  Provide `parse` and its support functions with debugging code, showing the actual parsing process on the standard output stream. When included, the debugging output is active by default, but its activity may be controlled using the `setDebug(bool on-off)` member. An `#ifdef DEBUG` macro is not supported by **bisonc++**. Rerun **bisonc++** without the `--debug` option to remove the debugging code.

- **--error-verbose**
  When a syntactic error is reported, the generated parse function dumps the parser's state stack to the standard output stream. The stack dump shows on separate lines a stack index followed by the state stored at the indicated stack element. The first stack element is the stack's top element.

- **--filenames**=`filename` (**-f**)
  `Filename` is a generic file name that is used for all header files generated by **bisonc++**. Options defining specific file names are also available (which then, in turn, overrule the name specified by this option).

- **--flex**
  Bisonc++ generates code calling `d_scanner.yylex()` to obtain the next lexical token, and calling `d_scanner.YYText()` for the matched text, unless overruled by options or directives explicitly defining these functions. By default, the interface defined by **flexc++**(1) is used. This option is only interpreted if the `--scanner` option or `%scanner` directive is also used.

- **--help** (**-h**)
  Write basic usage information to the standard output stream and terminate.

- **--implementation-header**=`filename` (**-i**)
  `Filename` defines the name of the file to contain the implementation header. It defaults to the name of the generated parser class plus the suffix `.ih`.

  The implementation header should contain all directives and declarations *only* used by the implementations of the parser's member functions. It is the only header file that is included by the source file containing `parse`'s implementation. User defined implementation of other class members may use the same convention, thus concentrating all directives and declarations that are required for the compilation of other source files belonging to the parser class in one header file.

- **--implementation-skeleton**=`pathname` (**-I**)
  `Pathname` defines the path name to the file containing the skeleton of the

implementation header. t defaults to the installation-defined default path name (e.g., `/usr/share/bisonc++/` plus `bisonc++.ih`).

- **--insert-stype**
  This option is only effective if the `debug` option (or `%debug` directive) has also been specified. When `insert-stype` has been specified the parsing function's debug output also shows selected semantic values. It should only be used if objects or variables of the semantic value type `STYPE__` can be inserted into `ostreams`.

- **--max-inclusion-depth**=`value`
  Set the maximum number of nested grammar files. Defaults to 10.

- **--namespace** `identifier`
  Define all of the code generated by **bisonc++** in the name space `identifier`. By default no name space is defined. If this options is used the implementation header is provided with a commented out `using namespace` declaration for the specified name space. In addition, the parser and parser base class header files also use the specified namespace to define their include guard directives.

  A warning is issued if this option is used and an already existing parser-class header file and/or implementation header file does not define `namespace identifier`.

- **--no-baseclass-header**
  Do not write the file containing the parser class' base class, even if that file doesn't yet exist. By default the file containing the parser's base class is (re)written each time **bisonc++** is called. Note that this option should normally be avoided, as the base class defines the symbolic terminal tokens that are returned by the lexical scanner. By suppressing the construction of this file any modification in these terminal tokens will not be communicated to the lexical scanner.

- **--no-lines**
  Do not put `#line` preprocessor directives in the file containing the parser's `parse` function. By default the file containing the parser's `parse` function also contains `#line` preprocessor directives. This option allows the compiler and debuggers to associate errors with lines in your grammar specification file, rather than with the source file containing the `parse` function itself.

- **--no-parse-member**
  Do not write the file containing the parser's predefined parser member functions, even if that file doesn't yet exist. By default the file containing the parser's `parse` member function is (re)written each time **bisonc++** is called. Note that this option

should normally be avoided, as this file contains parsing tables which are altered whenever the grammar definition is modified.

- **--own-debug**
  Extensively displays the actions performed by **bisonc++**'s parser when it processes the grammar specification **s**. This implies the `--verbose` option.

- **--own-tokens** (**-T**)
  The tokens returned as well as the text matched when **bisonc++** reads its input files(s) are shown when this option is used.

  This option does *not* result in the generated parsing function displaying returned tokens and matched text. If that is what you want, use the `--print-tokens` option.

- **--parsefun-skeleton**=`pathname` (**-P**)
  `Pathname` defines the path name of the file containing the parsing member function's skeleton. It defaults to the installation-defined default path name (e.g., `/usr/share/bisonc++/` plus `bisonc++.cc`).

- **--parsefun-source**=`filename` (**-p**)
  `Filename` defines the name of the source file to contain the parser member function `parse`. Defaults to `parse.cc`.

- **--polymorphic-skeleton**=`pathame` (**-M**)
  `Pathname` defines the path name of the file containing the skeleton of the polymorphic template classes. It defaults to the installation-defined default path name (e.g., `/usr/share/bisonc++/` plus `bisonc++polymorphic`).

- **--polymorphic-inline-skeleton**=`pathname` (**-m**)
  `Pathname` defines the path name of the file containing the skeleton of the inline implementations of the members of the polymorphic template classes. It defaults to the installation-defined default path name (e.g., `/usr/share/bisonc++/` plus `bisonc++polymorphic`).

- **--print-tokens** (**-t**)
  The generated parsing function implements a function `print__` displaying (on the standard output stream) the tokens returned by the parser's scanner as well as the corresponding matched text. This implementation is suppressed when the parsing function is generated without using this option. The member `print__`) is called from `Parser::print`, which is defined in-line in the the parser's class header. Calling `Parser::print__` can thus easily be controlled from `print`, using, e.g., a variable

that set by the program using the parser generated by **bisonc++**.

This option does *not* show the tokens returned and text matched by **bisonc++** itself when it is reading its input **s**. If that is what you want, use the `--own-tokens` option.

- **--required-tokens**=`number`
Following a syntactic error, require at least `number` successfully processed tokens before another syntactic error can be reported. By default `number` is zero.

- **--scanner**=`pathname` (**-s**)
`Pathname` defines the path name to the file defining the scanner's class interface (e.g., `"../scanner/scanner.h"`). When this option is used the parser's member `int lex()` is predefined as

```
    int Parser::lex()
    {
        return d_scanner.lex();
    }
```

and an object `Scanner d_scanner` is composed into the parser (but see also option `scanner-class-name`). The example shows the function that's called by default. When the `--flex` option (or `%flex` directive) is specified the function `d_scanner.yylex()` is called. Any other function to call can be specified using the `--scanner-token-function` option (or `%scanner-token-function` directive).

By default **bisonc++** surrounds `pathname` by double quotes (using, e.g., `#include "pathname"`). When `pathname` is surrounded by pointed brackets `#include <pathname>` is included.

A warning is issued if this option is used and an already existing parser class header file does not include `` `pathname' ``.

- **--scanner-class-name** `scannerClassName`
Defines the name of the scanner class, declared by the `pathname` header file that is specified at the `scanner` option or directive. By default the class name `Scanner` is used.

A warning is issued if this option is used and either the `scanner` option was not provided, or the parser class interface in an already existing parser class header file does not declare a scanner class `d_scanner` object.

- **--scanner-debug**
Show de scanner's matched rules and returned tokens. This offers an extensive display of the rules and tokens matched and returned by **bisonc++**'s scanner, not of just the tokens and matched text received by **bisonc++**. If that is what you want use the `--own-tokens` option.

- **--scanner-matched-text-function**=`function-call`
The scanner function returning the text that was matched at the last call of the scanner's token function. A complete function call expression should be provided (including a scanner object, if used). This option overrules the `d_scanner.matched()` call used by default when the `%scanner` directive is specified, and it overrules the `d_scanner.YYText()` call used when the `%flex` directive is provided. Example:

```
    --scanner-matched-text-function "myScanner.matchedText()"
```

- **--scanner-token-function**=`function-call`
The scanner function returning the next token, called from the parser's `lex` function. A complete function call expression should be provided (including a scanner object, if used). This option overrules the `d_scanner.lex()` call used by default when the `%scanner` directive is specified, and it overrules the `d_scanner.yylex()` call used when the `%flex` directive is provided. Example:

```
    --scanner-token-function "myScanner.nextToken()"
```

A warning is issued if this option is used and the scanner token function is not called from the code in an already existing implementation header.

- **--show-filenames**
Writes the names of the generated files to the standard error stream.

- **--skeleton-directory**=`directory` (**-S**)
Specifies the directory containing the skeleton files. This option can be overridden by the specific skeleton-specifying options (`-B -C, -H, -I, -M` and `-m`).

- **--target-directory**=`pathname`
`Pathname` defines the directory where generated files should be written. By default this is the directory where **bisonc++** is called.

- **--thread-safe**
  No static data are modified, making **bisonc++** thread-safe.

- **--usage**
  Write basic usage information to the standard output stream and terminate.

- **--verbose** (-V)
  Write a 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 conflicts detected in the grammar, both those resolved by operator precedence and those that remain unresolved. It is not created by default, but if requested the information is written on `<grammar>.output`, where `<grammar>` is the grammar specification file passed to **bisonc++**.

- **--version** (-v)
  Display **bisonc++**'s version number and terminate.

9.2: Bisonc++ usage

When **bisonc++** is called without any arguments it generates the following usage information:

```
bisonc++ by Frank B. Brokken (f.b.brokken@rug.nl)

LALR(1) Parser Generator V 4.05.00
Copyright (c) GPL 2005-2013. NO WARRANTY.
Designed after `bison++' (1.21.9-1) by Alain Coetmeur <coetmeur@icdc.fr>

Usage: bisonc++ [OPTIONS] file
Where:
  [OPTIONS] - zero or more optional arguments (int options between
              parentheses. Short options require arguments if their
              long option variants do too):
   --analyze-only (-A): only analyze the grammar; except for possibly
           the verbose grammar description file no files are written.
   --baseclass-preinclude=<header> (-H):
           preinclude header in the base-class header file.
           Use [header] to include <header>, otherwise "header"
           will be included.
   --baseclass-header=<header> (-b):
           filename holding the base class definition.
   --baseclass-skeleton=<skeleton> (-B):
```

```
            location of the baseclass header skeleton.
    --class-header=<header> (-c):
            filename holding the parser class definition.
    --class-skeleton=<skeleton> (-C):
            location of the class header skeleton.
    --construction: write details about the grammar analysis to stdout.
    --debug: generates debug output statements in the generated parse
            function's source.
    --error-verbose: the parse function will dump the parser's state
            stack to stdout when a syntactic error is reported
    --filenames=<filename> (-f):
            filename of output files (overruling the default filename).
    --help (-h): produce this information (and terminate).
    --implementation-header=<header> (-i):
            filename holding the implementation header.
    --implementation-skeleton=<skeleton> (-I):
            location of the implementation header skeleton.
    --include-only: catenate all grammar files in their order of
            processing to the standard output stream and terminate.
    --insert-stype: show selected semantic values in the output generated
            by --debug. Ignored unless --debug was specified.
    --no-lines: don't put #line directives in generated output.
    --max-inclusion-depth=<value>:
            sets the maximum number of nested grammar files (default: 10).
    --namespace=<namespace>, (-n):
            define the parser in the mentioned namespace.
    --no-baseclass-header: don't create the parser's base class header.
    --no-lines: don't put #line directives in generated output,
            overruling the %lines directive.
    --no-parse-member: don't create the member parse().
    --own-debug:
            bisonc++ displays the actions of its parser while processing
            its input file(s) (implies --verbose).
    --own-tokens (-t):
            bisonc++ displays the tokens and their corresponding
            matched text it received from its lexcial scanner.
    --parser-skeleton=<parserskel> (-P):
            location of the parse function's skeleton.
    --parsefun-source=<source> (-p):
            filename holding the parse function's source.
    --polymorphic-inline-skeleton=<skeleton> (-m):
            location of the polymorphic inline functions skeleton.
    --polymorphic-skeleton=<skeleton> (-M):
            location of the polymorphic semantic values skeleton.
    --print-tokens (-t):
            the print() member of the generated parser class displays
            the tokens and their corresponding matched text.
    --required-tokens=<value>:
            minimum number of successfully processed tokens between
            errors (default: 0).
    --scanner=<header-file> (-s):
            include `header-file' declaring the class Scanner, and call
            d_scanner.yylex() from Parser::lex().
    --scanner-class-name=<scanner class name>:
            specifies the name of the scanner class: this option is
            only interpreted if --scanner (or %scanner) is also used.
    --scanner-debug: extensive display of the actions of bisonc++'s scanner
    --scanner-token-function=<scanner token function>:
            define the function called from lex() returning the next
            token returned (by default d_scanner.yylex() when --scanner
            is used)
    --show-filenames: show the names of the used/generated files on
            the standard error stream.
    --skeleton-directory=<skeleton-directory> (-S):
            location of the skeleton directory.
    --thread-safe: no static data are modified, making bisonc++'s
            generated code thread-safe.
    --usage: produce this information (and terminate).
    --verbose (-V):
            generate verbose description of the analyzed grammar.
    --version (-v):
            display bisonc++'s version and terminate.
```

- Table of Contents
- Previous Chapter