Bison

El Generador de Analizadores Sintácticos compatible con YACC.
12 Febrero 1999, Bison Versión 1.27

por Charles Donnelly y Richard Stallman

Published by the Free Software Foundation
59 Temple Place, Suite 330
Boston, MA 02111-1307 USA
Printed copies are available for $15 each.
ISBN 1-882114-45-0

Cover art by Etienne Suvasa.

Diseño de cubierta por Etienne Suvasa.

Introducción

Bison es un generador de analizadores sintácticos de propósito general que convierte una descripción gramatical para una gramática independiente del contexto LALR(1) en un programa en C que analice esa gramática. Una vez que sea un experimentado en Bison, podría utilizarlo para desarollar un amplio rango de analizadores de lenguajes, desde aquellos usados en simples calculadoras de escritorio hasta complejos lenguajes de programación.

Bison es compatible hacia arriba con Yacc: todas la gramáticas escritas apropiadamente para Yacc deberían funcionar con Bison sin ningún cambio. Cualquiera que esté familiarizado con Yacc debería ser capaz de utilizar Bison con pocos problemas. Necesita ser fluente programando en C para poder utilizar Bison o para comprender este manual.

Comenzaremos con capítulos introductorios que explican los conceptos básicos del uso de Bison y muestran tres ejemplos comentados, cada uno construido sobre el anterior. Si no conoce Bison o Yacc, comience leyendo estos capítulos. A continuación se encuentran los capítulos de referencia que describen los aspectos específicos de Bison en detalle.

Bison fue escrito originalmente por Robert Corbett; Richard Stallman lo hizo compatible con Yacc. Wilfred Hansen de la Universidad de Carnegie Mellon añadió los literales de cadenas multicaracter y otras características.

Esta edición corresponde a la versión 1.27 de Bison.

Nota: las secciones tituladas "Licencia Pública General GNU", "Condiciones para el uso de Bison" y el aviso de permiso son traducciones libres de las secciones originales en inglés "GNU General Public License", "Conditions for Using Bison" y el permiso original. Ninguna de estas traducciones ha sido aprobada por la Free Software Foundation oficialmente y se han incluído solamente para facilitar su entendimiento. Si desea estar seguro de si sus actuaciones están permitidas, por favor acuda a la versión original inglesa.

La Free Software Foundation recomienda fervientemente no usar estas traducciones como los términos oficiales de distribución para sus programas; en su lugar, por favor use las versiones inglesas originales, tal y como están publicadas por la Free Software Foundation.

Conditions for Using Bison

As of Bison version 1.24, we have changed the distribution terms for `yyparse` to permit using Bison's output in non-free programs. Formerly, Bison parsers could be used only in programs that were free software.

The other GNU programming tools, such as the GNU C compiler, have never had such a requirement. They could always be used for non-free software. The reason Bison was different was not due to a special policy decision; it resulted from applying the usual General Public License to all of the Bison source code.

The output of the Bison utility—the Bison parser file—contains a verbatim copy of a sizable piece of Bison, which is the code for the `yyparse` function. (The actions from your grammar are inserted into this function at one point, but the rest of the function is not changed.) When we applied the GPL terms to the code for `yyparse`, the effect was to restrict the use of Bison output to free software.

We didn't change the terms because of sympathy for people who want to make software proprietary. **Software should be free.** But we concluded that limiting Bison's use to free software was doing little to encourage people to make other software free. So we decided to make the practical conditions for using Bison match the practical conditions for using the other GNU tools.

Condiciones para el uso de Bison

Al igual que en la versión 1.24 de Bison, hemos cambiado los términos de la distribución de `yyparse` para permitir el uso de la salida de Bison en programas no-libres. En otro tiempo, los analizadores generados por Bison solamente podían utilizarse en programas que fuesen software libre.

Las otras herramientas GNU de programación, tales como el compilador de C GNU, nunca han tenido tal tipo de requisito. Estas herramientas siempre podían utilizarse para software no-libre. La razón de que con Bison fuera diferente no fue debido a una decisión política especial; ello resultó de la aplicación de la Licencia Pública General usual a todo el código fuente de Bison.

La salida de la utilidad Bison—el archivo del analizador de Bison—contiene una copia literal de un considerable fragmento de Bison, que es el código para la función `yyparse`. (Las acciones de tu gramática se insertan dentro de esta función en un punto, pero el resto de la función no se modifica.) Cuando aplicamos los términos de la GPL al código fuente para `yyparse`, el efecto fue la restricción del uso de la salida de Bison en software libre.

No cambiamos los términos debido a simpatía con la gente que quiere hacer software propietario. **El software debería ser libre.** Pero hemos concluido que limitando el uso de Bison en software libre era hacer poco por alentar a la gente a hacer otro software libre. Así que hemos decidido hacer que concuerden las condiciones prácticas para el uso de Bison con las condiciones prácticas para usar las otras utilidades GNU.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

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

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

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

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

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

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

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

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

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

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

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

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

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

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

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

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

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

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

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

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

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

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

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

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),

EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

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

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

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

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

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

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

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

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

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
'Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
```

```
Ty Coon, President of Vice
```

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

LICENCIA PÚBLICA GENERAL GNU

Versión 2, Junio de 1991

Preámbulo

Las licencias que cubren la mayor parte del software están diseñadas para quitarle a usted la libertad de compartirlo y modificarlo. Por el contrario, la Licencia Pública General GNU pretende garantizarle la libertad de compartir y modificar software libre—para asegurar que el software es libre para todos sus usuarios. Esta Licencia Pública General se aplica a la mayor parte del software de la Free Software Foundation y a cualquier otro programa cuyos autores se comprometen a utilizarla. (Alguna parte del software de la Free Software Foundation está cubierto por la Licencia Pública General GNU para Librerías). Usted también la puede aplicar a sus programas.

Cuando hablamos de software libre, estamos refiriéndonos a la libertad, no al precio. Nuestras Licencias Públicas Generales están diseñadas para asegurarnos de que tenga la libertad de distribuir copias de software libre (y cobrar por ese servicio si quiere), que reciba el código fuente o que pueda conseguirlo si lo quiere, que pueda modificar el software o usar fragmentos de él en nuevos programas libres, y que sepa que puede hacer todas estas cosas.

Para proteger sus derechos necesitamos algunas restricciones que prohiban a cualquiera negarle a usted estos derechos o pedirle que renuncie a ellos. Estas restricciones se traducen en ciertas obligaciones que le afectan si distribuye copias del software, o si lo modifica.

Por ejemplo, si distribuye copias de uno de estos programas, sea gratuitamente, o a cambio de una contraprestación, debe dar a los receptores todos los derechos que tiene. Debe asegurarse de que ellos también reciben, o pueden conseguir, el código fuente. Y debe mostrarles estas condiciones de forma que conozcan sus derechos.

Protegemos sus derechos con la combinación de dos medidas: (1) ponemos el software bajo copyright y (2) le ofrecemos esta licencia, que le da permiso legal para copiar, distribuir y/o modificar el software.

También, para la protección de cada autor y la nuestra propia, queremos asegurarnos de que todo el mundo comprende que no se proporciona ninguna garantía para este software libre. Si el software es modificado por cualquiera y éste a su vez lo distribuye, queremos que sus receptores sepan que lo que tienen no es el original, de forma que cualquier problema introducido por otros no afecte a la reputación de los autores originales.

Por último, cualquier programa libre está constantemente amenazado por patentes sobre el software. Queremos evitar el riesgo de que los redistribuidores de un programa libre individualmente obtengan patentes, haciendo el programa propietario a todos los efectos. Para prevenir esto, hemos dejado claro que cualquier patente debe ser concedida para el uso libre de cualquiera, o no ser concedida en absoluto.

Los términos exactos y las condiciones para la copia, distribución y modificación se exponen a continuación.

TÉRMINOS Y CONDICIONES PARA LA COPIA, DISTRIBUCIÓN Y MODIFICACIÓN

0. Esta Licencia se aplica a cualquier programa u otra obra que contenga un aviso colocado por el propietario del copyright diciendo que puede ser distribuido bajo los términos de esta Licencia Pública General. En adelante, "Programa" se referirá a cualquier programa u obra de esta clase y "una obra basada en el Programa" se referirá bien al Programa o a cualquier obra derivada de este según la ley de copyright. Esto es, una obra que contenga el programa o una porción de este, bien en forma literal o con modificaciones y/o traducido en otro lenguaje. Por lo tanto, la traducción está incluida sin limitaciones en el término "modificación". Cada propietario de una licencia será tratado como "usted".

   Cualquier otra actividad que no sea la copia, distribución o modificación no está cubierta por esta Licencia, está fuera de su ámbito. El acto de ejecutar el Programa no está restringido, y los resultados del Programa están cubiertos únicamente si sus contenidos constituyen una obra basada en el Programa, independientemente de haberlo producido mediante la ejecución del programa. Que esto se cumpla, depende de lo que haga el programa.

1. Usted puede copiar y distribuir copias literales del código fuente del Programa, tal y como lo recibió, por cualquier medio, supuesto que de forma adecuada y bien visible publique en cada copia un anuncio de copyright adecuado y una renuncia de garantía, mantenga intactos todos los anuncios que se refieran a esta Licencia y a la ausencia de garantía, y proporcione a cualquier otro receptor del programa una copia de esta Licencia junto con el Programa.

   Puede cobrar un precio por el acto físico de transferir una copia, y puede a su elección ofrecer garantía a cambio de unos honorarios.

2. Usted puede modificar su copia o copias del Programa o cualquier porción de él, formando de esta manera una obra basada en el Programa, y copiar y distribuir esa modificación u obra bajo los términos del apartado 1 anterior, siempre que además cumpla las siguientes condiciones:

   a. Debe procurar que los ficheros modificados incluyan notificaciones destacadas manifestando que los ha cambiado y la fecha de cualquier cambio.

   b. Usted debe procurar que cualquier obra que distribuya o publique, que en todo o en parte contenga o sea derivada del Programa o de cualquier parte de él, sea licenciada como un todo, sin cargo alguno para terceras partes bajo los términos de esta Licencia.

   c. Si el programa modificado lee normalmente órdenes interactivamente cuando al ejecutarse, debe hacer que cuando comience su ejecución para ese uso interactivo de la forma más habitual, muestre o escriba un mensaje que incluya un anuncio de copyright y un anuncio de que no se ofrece ninguna garantía (o por el contrario que sí se ofrece garantía) y que los usuarios pueden redistribuir el programa bajo estas condiciones, e indicando al usuario cómo ver una copia de esta licencia. (Excepción: si el propio programa es interactivo pero normalmente no muestra ese anuncio, no está obligado a que su obra basada en el Programa muestre ningún anuncio).

   Estos requisitos se aplican a la obra modificada como un todo. Si algunas secciones claramente identificables de esa obra no están derivadas del Programa, y pueden razonablemente ser consideradas como obras independientes y separados por sí mismas, entonces esta Licencia y sus términos no se aplican a esas partes cuando sean distribuidas como trabajos separados. Pero cuando distribuya esas mismas secciones como partes de un todo que es una obra basada en el Programa, la distribución de ese todo debe cumplir los términos de esta Licencia, cuyos permisos para otros licenciatarios se extienden al todo completo, y por lo tanto a todas y cada una de sus partes, con independencia de quién la escribió.

Por lo tanto, no es intención de este apartado reclamar derechos u oponerse a sus derechos sobre obras escritas enteramente por usted; sino que la intención es ejercer el derecho de controlar la distribución de obras derivadas o colectivas basadas en el Programa.

Además, el simple hecho de reunir otro trabajo no basado en el Programa con el Programa (o con un trabajo basado en el Programa) en un medio de almacenamiento o en un medio de distribución no hace que dicho trabajo entre dentro del ámbito cubierto por esta Licencia.

3. Usted puede copiar y distribuir el Programa (o una obra basada en él, según se especifica en la Sección 2) en forma de código objeto o ejecutable bajo los términos de las Secciones 1 y 2 anteriores mientras cumpla además una de las siguientes condiciones:
   a. Acompañarlo con el código fuente completo correspondiente en formato legible para un ordenador, que debe ser distribuido bajo los términos de las Secciones 1 y 2 anteriores en un medio utilizado habitualmente para el intercambio de programas, o
   b. Acompañarlo con una oferta por escrito, válida durante al menos tres años, por un coste no mayor que el de realizar físicamente la distribución del fuente, de proporcionar a cualquier tercera parte una copia completa en formato legible para un ordenador del código fuente correspondiente, que será distribuido bajo las condiciones descritas en las Secciones 1 y 2 anteriores, en un medio utilizado habitualmente para el intercambio de programas, o
   c. Acompañarlo con la información que usted recibió referida al ofrecimiento de distribuir el código fuente correspondiente. (Esta opción se permite sólo para la distribución no comercial y sólo si usted recibió el programa como código objeto o en formato ejecutable con una oferta de este tipo, de acuerdo con la Sección b anterior).

   Se entiende por código fuente de un trabajo a la forma preferida de la obra para hacer modificaciones sobre este. Para una obra ejecutable, se entiende por código fuente completo todo el código fuente para todos los módulos que contiene, más cualquier fichero asociado de definición de interfaces, más los guiones utilizados para controlar la compilación e instalación del ejecutable. Como excepción especial el código fuente distribuido no necesita incluir nada que sea distribuido normalmente (ya sea en formato fuente o binario) con los componentes fundamentales (compilador, kernel y similares) del sistema operativo en el cual funciona el ejecutable, a no ser que el propio componente acompañe al ejecutable.

   Si la distribución del ejecutable o del código objeto se realiza ofreciendo acceso a una copia desde un lugar designado, entonces se considera el ofrecimiento del acceso para copiar el código fuente del mismo lugar como distribución del código fuente, incluso aunque terceras partes no estén obligadas a copiar el fuente junto al código objeto.
4. No puede copiar, modificar, sublicenciar o distribuir el Programa excepto como está expresamente permitido por esta Licencia. Cualquier intento de copiar, modificar sublicenciar o distribuir el Programa de otra forma es inválido, y hará que cesen automáticamente los derechos que le proporciona esta Licencia. En cualquier caso, las partes que hayan recibido copias o derechos bajo esta Licencia no verán sus Licencias calceladas, mientras esas partes continúen cumpliendo totalmente la Licencia.
5. No está obligado a aceptar esta licencia, ya que no la ha firmado. Sin embargo, no hay hada más que le proporcione permiso para modificar o distribuir el Programa o sus trabajos derivados. Estas acciones están prohibidas por la ley si no acepta esta Licencia. Por lo tanto, si modifica o distribuye el Programa (o cualquier trabajo basado en el Programa), está indicando que acepta esta Licencia para poder hacerlo, y todos sus términos y condiciones para copiar, distribuir o modificar el Programa o trabajos basados en él.
6. Cada vez que redistribuya el Programa (o cualquier trabajo basado en el Programa), el receptor recibe automáticamente una licencia del licenciatario original para copiar, distribuir o modificar el Programa, de forma sujeta a estos términos y condiciones. No puede imponer al receptor ninguna restricción más sobre el ejercicio de los derechos aquí garantizados. No es usted responsable de hacer cumplir esta licencia por terceras partes.
7. Si como consecuencia de una resolución judicial o de una alegación de infracción de patente o por cualquier otra razón (no limitada a asuntos relacionados con patentes) se le imponen con-

diciones (ya sea por mandato judicial, por acuerdo o por cualquier otra causa) que contradigan las condiciones de esta Licencia, ello no le exime de cumplir las condiciones de esta Licencia. Si no puede realizar distribuciones de forma que se satisfagan simultáneamente sus obligaciones bajo esta licencia y cualquier otra obligación pertinente entonces, como consecuencia, no puede distribuir el Programa de ninguna forma. Por ejemplo, si una patente no permite la redistribución libre de derechos de autor del Programa por parte de todos aquellos que reciban copias directa o indirectamente a través de usted, entonces la única forma en que podría satisfacer tanto esa condición como esta Licencia sería evitar completamente la distribución del Programa.

Si cualquier porción de este apartado se considera no válido o imposible de cumplir bajo cualquier circunstancia particular ha de cumplirse el resto y la sección por entero ha de cumplirse en cualquier otra circunstancia.

No es el propósito de este apartado inducirle a infringir ninguna patente ni ningún otro derecho de propiedad o impugnar la validez de ninguna de dichas reclamaciones. Este apartado tiene el único propósito de proteger la integridad del sistema de distribución de software libre, que se realiza mediante prácticas de licencia pública. Mucha gente ha hecho contribuciones generosas a la gran variedad de software distribuido mediante ese sistema con la confianza de que el sistema se aplicará consistentemente. Será el autor/donante quien decida si quiere distribuir software mediante cualquier otro sistema y una licencia no puede imponer esa elección.

Este apartado pretende dejar completamente claro lo que se cree que es una consecuencia del resto de esta Licencia.

8. Si la distribución y/o uso de el Programa está restringido en ciertos países, bien por patentes o por interfaces bajo copyright, el poseedor del copyright que coloca este Programa bajo esta Licencia puede añadir una limitación explícita de distribución geográfica excluyendo esos países, de forma que la distribución se permita sólo en o entre los países no excluidos de esta manera. En ese caso, esta Licencia incorporará la limitación como si estuviese escrita en el cuerpo de esta Licencia.

9. La Free Software Foundation puede publicar versiones revisadas y/o nuevas de la Licencia Pública General de tiempo en tiempo. Dichas versiones nuevas serán similares en espíritu a la presente versión, pero pueden ser diferentes en detalles para considerar nuevos problemas o situaciones.

   Cada versión recibe un número de versión que la distingue de otras. Si el Programa especifica un número de versión de esta Licencia que se aplica a ella y a "cualquier versión posterior", tiene la opción de seguir los términos y condiciones, bien de esa versión, bien de cualquier versión posterior publicada por la Free Software Foundation. Si el Programa no especifica un número de versión de esta Licencia, puede escoger cualquier versión publicada por la Free Software Foundation.

10. Si usted desea incorporar partes del Programa en otros programas libres cuyas condiciones de distribución son diferentes, escriba al autor para pedirle permiso. Si el software tiene copyright de la Free Software Foundation, escriba a la Free Software Foundation: algunas veces hacemos excepciones en estos casos. Nuestra decisión estará guiada por el doble objetivo de preservar la libertad de todos los derivados de nuestro software libre y promover el que se comparta y reutilice el software en general.

AUSENCIA DE GARANTÍA

11. YA QUE EL PROGRAMA SE LICENCIA LIBRE DE CARGAS, NO SE OFRECE NINGUNA GARANTÍA SOBRE EL PROGRAMA, HASTA LO PERMITIDO POR LAS LEYES APLICABLES. EXCEPTO CUANDO SE INDIQUE LO CONTRARIO POR ESCRITO, LOS POSEEDORES DEL COPYRIGHT Y/U OTRAS PARTES PROVEEN EL PROGRAMA "TAL Y COMO ESTÁ", SIN GARANTÍA DE NINGUNA CLASE, YA SEA EXPRESA O IMPLÍCITA, INCLUYENDO, PERO NO LIMITÁNDOSE A, LAS GARANTÍAS

IMPLÍCITAS DE COMERCIABILIDAD Y APTITUD PARA UN PROPÓSITO PARTICULAR. TODO EL RIESGO EN CUANTO A LA CALIDAD Y FUNCIONAMIENTO DEL PROGRAMA LO ASUME USTED. SI EL PROGRAMA SE COMPROBARA QUE ESTÁ DEFECTUOSO, USTED ASUME EL COSTO DE TODO SERVICIO, REPARACIÓN O CORRECCIÓN QUE SEA NECESARIO.

12. EN NINGÚN CASO, A NO SER QUE SE REQUIERA POR LAS LEYES APLICABLES O SE ACUERDE POR ESCRITO, PODRÁ NINGÚN POSEEDOR DE COPYRIGHT O CUALQUIER OTRA PARTE QUE HAYA MODIFICADO Y/O REDISTRIBUIDO EL PROGRAMA, SER RESPONSABLE ANTE USTED POR DAÑOS O PERJUICIOS, INCLUYENDO CUALQUIER DAÑO GENERAL, ESPECIAL, INCIDENTAL O CONSECUENTE DEBIDO AL USO O LA IMPOSIBILIDAD DE PODER USAR EL PROGRAMA (INCLUYENDO PERO NO LIMITÁNDOSE A LA PÉRDIDA DE DATOS O LA PRODUCCIÓN DE DATOS INCORRECTOS O PÉRDIDAS SUFRIDAS POR USTED O POR TERCERAS PARTES O LA IMPOSIBILIDAD DEL PROGRAMA DE OPERAR JUNTO A OTROS PROGRAMAS), INCLUSO SI EL POSEEDOR DEL COPYRIGHT U OTRA PARTE HA SIDO AVISADO DE LA POSIBILIDAD DE TALES DAÑOS.

FIN DE TÉRMINOS Y CONDICIONES

Cómo aplicar estos términos a sus nuevos programas.

Si usted desarrolla un nuevo Programa, y quiere que sea del mayor uso posible para el público en general, la mejor forma de conseguirlo es convirtiéndolo en software libre que cualquiera pueda redistribuir y cambiar bajo estos términos.

Para hacerlo, añada los siguientes avisos al programa. Lo más seguro es añadirlos al principio de cada fichero fuente para comunicar lo más efectivamente posible la ausencia de garantía. Además cada fichero debería tener al menos la línea de "copyright" y una indicación del lugar donde se encuentra la notificación completa.

```
una línea para indicar el nombre del programa y una rápida idea de
lo que hace.
Copyright (C) 19aa nombre del autor

Este programa es software libre; usted puede redistribuirlo y/o modificarlo
bajo los términos de la Licencia Pública General GNU tal y como está
publicada por la Free Software Foundation; ya sea la versión 2 de la
Licencia o (a su elección) cualquier versión posterior.

Este programa se distribuye con la esperanza de que sea útil, pero SIN
NINGUNA GARANTÍA; ni siquiera la garantía implícita de COMERCIABILIDAD o
APTITUD PARA UN PROPÓSITO ESPECÍFICO. Vea la Licencia Pública General
GNU para más detalles.

Usted debería haber recibido una copia de la Licencia Pública General junto
con este programa. Si no ha sido así, escriba a la Free Software
Foundation, Inc., en 675 Mass Ave, Cambridge, MA 02139, EEUU.
```

Añada también información sobre cómo contactar con usted mediante correo electrónico y postal.

Si el programa es interactivo, haga que muestre un pequeño anuncio como el siguiente, cuando comience a funcionar en modo interactivo:

```
Gnomovision versión 69, Copyright (C) 19aa nombre del autor
Gnomovision no ofrece ABSOLUTAMENTE NINGUNA GARANTÍA; para más
detalles escriba 'show w'.
Esto es software libre, y se le invita a redistribuirlo bajo ciertas
condiciones. Escriba 'show c' para más detalles.
```

Los comandos hipotéticos '`show w`' y '`show c`' deberían mostrar las partes adecuadas de la Licencia Pública General. Por supuesto, los comandos que use pueden llamarse de cualquier otra manera. Podrían incluso ser pulsaciones del ratón o elementos de un menú—lo que sea apropiado para su programa).

También debería conseguir que el empresario (si trabaja como programador) o su centro académico, si es el caso, firme una "renuncia de copyright" para el programa, si es necesario. A continuación se ofrece un ejemplo, cambie los nombres:

```
Yoyodyne, Inc. con la presente renuncia a cualquier interés de
```

```
derechos de copyright con respecto al programa 'Gnomovision' (que hace
pasadas a compiladores) escrito por Pepe Programador.
```

firma de Pepito Grillo, `20 de diciembre de 1996`
`Pepito Grillo, Presidente de Asuntillos Varios.`

Esta Licencia Pública General no permite incorporar su programa a programas propietarios. Si su programa es una librería de subrutinas, puede considerar más útil el permitir el enlazado de aplicaciones propietarias con la librería. Si este es el caso, use la Licencia Pública General GNU para Librerías en lugar de esta Licencia.

1 Los Conceptos de Bison

Este capítulo introduce muchos de los conceptos básicos sin los que no tendrán sentido los detalles de Bison. Si no conoce ya cómo utilizar Bison o Yacc, le sugerimos que comience por leer este capítulo atentamente.

1.1 Lenguajes y Gramáticas independientes del Contexto

Para que Bison analice un lenguaje, este debe ser descrito por una *gramática independiente del contexto*. Esto quiere decir que debe especificar uno o más *grupos sintácticos* y dar reglas para contruirlos desde sus partes. Por ejemplo, en el lenguaje C, un tipo de agrupación son las llamadas 'expresiones'. Una regla para hacer una expresión sería, "Una expresión puede estar compuesta de un signo menos y otra expresión". Otra regla sería, "Una expresión puede ser un entero". Como puede ver, las reglas son a menudo recursivas, pero debe haber al menos una regla que lleve fuera la recursión.

El sistema formal más común de presentar tales reglas para ser leidas por los humanos es la *Forma de Backus-Naur* o "BNF", que fue desarrollada para especificar el lenguaje Algol 60. Cualquier gramática expresada en BNF es una gramática independiente del contexto. La entrada de Bison es en esencia una BNF legible por la máquina.

No todos los lenguajes independientes del contexto pueden ser manejados por Bison, únicamente aquellos que sean LALR(1). Brevemente, esto quiere decir que debe ser posible decir cómo analizar cualquier porción de una cadena de entrada con un solo token de preanálisis. Hablando estrictamente, esto es una descripción de una gramática LR(1), y la LALR(1) implica restricciones adicionales que son difíciles de explicar de manera sencilla; pero es raro en la práctica real que se encuentre una gramática LR(1) que no sea LALR(1). Ver Seccion 5.7 [Conflictos Misteriosos de Reducción/Reducción], página 77, para más información a cerca de esto.

En las reglas gramaticales formales para un lenguaje, cada tipo de unidad sintáctica o agrupación se identifica por un *símbolo*. Aquellos que son construidos agrupando construcciones más pequeñas de acuerdo a reglas gramaticales se denominan *símbolos no terminales*; aquellos que no pueden subdividirse se denominan *símbolos terminales* o *tipos de tokens*. Denominamos *token* a un fragmento de la entrada que corresponde a un solo símbolo terminal, y *grupo* a un fragmento que corresponde a un solo símbolo no terminal.

Podemos utilizar el lenguaje C como ejemplo de qué significan los símbolos, terminales y no terminales. Los tokens de C son los identificadores, constantes (numéricas y cadenas de caracteres), y las diversas palabras reservadas, operadores aritméticos y marcas de puntuación. Luego los símbolos terminales de una gramática para C incluyen 'identificador', 'número', 'cadena de caracteres', más un símbolo para cada palabra reservada, operador o marca de puntuación: 'if', 'return', 'const', 'static', 'int', 'char', 'signo-más', 'llave-abrir', 'llave-cerrar', 'coma' y muchos más. (Estos tokens se pueden subdividir en caracteres, pero eso es una cuestión léxica, no gramatical.)

Aquí hay una función simple en C subdividida en tokens:

```
int             /* palabra reservada 'int' */
cuadrado (x)    /* identificador, paréntesis-abrir */
                /* identificador, paréntesis-cerrar */
```

```
      int x;      /* palabra reservada 'int', identificador, punto y coma */
{                 /* llave-abrir */
    return x * x; /* palabra reservada 'return', identificador, */
                  /* asterisco, identificador, punto y coma */
}                 /* llave-cerrar */
```

Las agrupaciones sintácticas de C incluyen a las expresiones, las sentencias, las declaraciones, y las definiciones de funciones. Estas se representan en la gramática de C por los símbolos no terminales 'expresión', 'sentencia', 'declaración' y 'definición de función'. La gramática completa utiliza docenas de construcciones del lenguaje adicionales, cada uno con su propio símbolo no terminal, de manera que exprese el significado de esos cuatro. El ejemplo anterios es la definición de una función; contiene una declaración, y una sentencia. En la sentencia, cada '`x`' es una expresión y también lo es '`x * x`'.

Cada símbolo no terminal debe poseer reglas gramaticales mostrando cómo está compuesto de construcciones más simples. Por ejemplo, un tipo de sentencia en C es la sentencia `return`; esta sería descrita con una regla gramatical que interpretada informalmente sería así:

> Una 'sentencia' puede estar compuesta de una parabra clave 'return', una 'expresión' y un 'punto y coma'.

Aquí existirían muchas otras reglas para 'sentencia', una para cada tipo de sentencia en C.

Se debe distinguir un símbolo no terminal como el símbolo especial que define una declaración completa en el lenguaje. Este se denomina *símbolo de arranque*. En un compilador, este representa un programa completo. En el lenguaje C, el símbolo no terminal 'secuencia de definiciones y declaraciones' juega este papel.

Por ejemplo, '`1 + 2`' es una expresión válida en C—una parte válida de un programa en C—pero no es válida como un programa en C *completo*. En la gramática independiente del contexto de C, esto se refleja en el hecho de que 'expresión' no es el símbolo de arranque.

El analizador de Bison lee una secuencia de tokens como entrada, y agrupa los tokens utilizando las reglas gramaticales. Si la entrada es válida, el resultado final es que la secuencia de tokens entera se reduce a una sola agrupación cuyo símbolo es el símbolo de arranque de la gramática. Si usamos una gramática para C, la entrada completa debe ser una 'secuencia de definiciones y declaraciones'. Si no, el analizador informa de un error de sintaxis.

1.2 De las Reglas Formales a la Entrada de Bison

Una gramática formal es una construcción matemática. Para definir el lenguaje para Bison, debe escribir un archivo expresando la gramática con la sintaxis de Bison: un archivo de *gramática de Bison*. Ver Capitulo 3 [Archivos de Gramática de Bison], página 45.

Un símbolo no terminal en la gramática formal se representa en la entrada de Bison como un identificador, similar a un identificador en C. Por convención, deberían estar en minúsculas, tales como `expr`, `stmt` o `declaracion`.

La representación en Bison para un símbolo terminal se llama también un *tipo de token*. Los tipos de tokens también se pueden representar como identificadores al estilo de C. Por convención, estos identificadores deberían estar en mayúsculas para distinguirlos de los no terminales: por ejemplo, `INTEGER`, `IDENTIFICADOR`, `IF` o `RETURN`. Un símbolo terminal que represente una palabra clave en particular en el lenguaje debería bautizarse con el nombre después de pasarlo a mayúsculas. El símbolo terminal `error` se reserva para la recuperación de errores. Ver Seccion 3.2 [Simbolos], página 46.

Un símbolo terminal puede representarse también como un caracter literal, al igual que una constante de caracter en C. Debería hacer esto siempre que un token sea simplemente un único caracter (paréntesis, signo-más, etc.): use el mismo caracter en un literal que el símbolo terminal para ese token.

Una tercera forma de representar un símbolo terminal es con una cadena de caracteres de C conteniendo varios caracteres. Ver Seccion 3.2 [Simbolos], página 46, para más información.

Las reglas gramaticales tienen también una expresión en la sintaxis de Bison. Por ejemplo, aquí está la regla en Bison para una sentencia `return` de C. El punto y coma entre comillas es un token de caracter literal, representando parte de la sintaxis de C para la sentencia; el punto y coma al descubierto, y los dos puntos, es puntuación de Bison que se usa en todas las reglas.

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

Ver Seccion 3.3 [Sintaxis de las Reglas Gramaticales], página 48.

1.3 Valores Semánticos

Una gramática formal selecciona tokens únicamente por sus clasificaciones: por ejemplo, si una regla menciona el símbolo terminal 'constante entera', quiere decir que *cualquier* constante entera es gramaticalmente válida en esa posición. El valor preciso de la constante es irrelevante en cómo se analiza la entrada: si '`x+4`' es gramatical entonces '`x+1`' o '`x+3989`' es igualmente gramatical.

Pero el valor preciso es muy importante para lo que significa la entrada una vez que es analizada. ¡Un compilador es inservible si no puede distinguir entre 4, 1 y 3989 como constantes en el programa! Por lo tanto, cada token en una gramática de Bison tiene ambos, un tipo de token y un *valor semántico*. Ver Seccion 3.5 [Definiendo la Semántica del Lenguaje], página 50, para detalles.

El tipo de token es un símbolo terminal definido en la gramática, tal como `INTEGER`, `IDENTIFICADOR` o `','`. Este dice todo lo que se necesita para saber decidir dónde podría aparecer válidamente el token y cómo agruparlo con los otros tokens. Las reglas gramaticales no saben nada acerca de los tokens excepto de sus tipos.

El valor semántico tiene todo el resto de información a cerca del significado del token, tal como el valor de un entero, o el nombre de un identificador. (Un token tal como `','` que es solo un signo de puntuación no necesita tener ningún valor semántico.)

Por ejemplo, un token de entrada podría clasificarse como un tipo de token `INTEGER` y tener el valor semántico 4. Otro token de entrada podría tener el mismo tipo de token `INTEGER` pero valor

3989. Cuando una regla gramatical dice que se admite un `INTEGER`, cualquiera de estos tokens se acepta porque cada uno es un `INTEGER`. Cuando el analizador acepta el token, este no pierde la pista del valor semántico del token.

Cada agrupación puede tener también un valor semántico al igual que su símbolo no terminal. Por ejemplo, en una calculadora, una expresión típicamente tiene un valor semántico que es un número. En un compilador para un lenguaje de programación, una expresión típicamente tiene un valor semántico que es una estructura en árbol describiendo el significado de la expresión.

1.4 Acciones Semánticas

Para que sea útil, un programa debe hacer algo más que analizar la entrada; este debe producir también alguna salida basada en la entrada. En una gramática de Bison, una regla gramatical puede tener una *acción* compuesta de sentencias en C. Cada vez que el analizador reconozca una correspondencia para esa regla, se ejecuta la acción. Ver Seccion 3.5.3 [Acciones], página 50.

La mayor parte del tiempo, el propósito de una acción es computar el valor semántico de la construcción completa a partir de los valores semánticos de sus partes. Por ejemplo, suponga que tenemos una regla que dice que una expresión puede ser la suma de dos expresiones. Cuando el analizador reconozca tal suma, cada una de las subexpresiones posee un valor semántico que describe cómo fueron elaboradas. La acción para esta regla debería crear un tipo de valor similar para la expresión mayor que se acaba de reconocer.

Por ejemplo, he aquí una regla que dice que una expresión puede ser la suma de dos subexpresiones:

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

La acción dice cómo producir el valor semántico de la expresión suma a partir de los valores de las dos subexpresiones.

1.5 La Salida de Bison: el Archivo del Analizador

Cuando ejecuta Bison, usted le da un archivo de gramática de Bison como entrada. La salida es un programa fuente en C que analiza el lenguaje descrito por la gramática. Este archivo se denomina un *analizador de Bison*. Tenga en cuenta que la utilidad Bison y el analizador de Bison son dos programas distintos: la utilidad Bison es un programa cuya salida es el analizador de Bison que forma parte de su programa.

El trabajo del analizador de Bison es juntar tokens en agrupaciones de acuerdo a las reglas gramaticales—por ejemplo, construir expresiones con identificadores y operadores. A medida que lo hace, este ejecuta las acciones de las reglas gramaticales que utiliza.

Los tokens provienen de una función llamada el *analizador léxico* que usted debe proveer de alguna manera (por ejemplo escribiéndola en C). El analizador de Bison llama al analizador léxico cada vez que quiera un nuevo token. Este no sabe qué hay "dentro" de los tokens (aunque sus valores semánticos podrían reflejarlo). Típicamente el analizador léxico construye los tokens analizando los caracteres del texto, pero Bison no depende de ello. Ver Seccion 4.2 [La Función del Analizador Léxico `yylex`], página 61.

El fichero del analizador de Bison es código C que define una función llamada `yyparse` que implementa esa gramática. Esta función no forma un programa completo en C: debe proveer algunas funciones adicionales. Una es el analizador léxico. Otra es una función de informe de errores a la que el analizador llama para informar de un error. Además, un programa completo en C debe comenzar con una función llamada `main`; debe facilitarla, y colocar en esta una llamada a `yyparse` o el analizador no será ejecutado nunca. Ver Capitulo 4 [Interfaz del Analizador en Lenguaje C], página 61.

A parte de los nombres de tipo de token y los símbolos en las acciones que escriba, todos los nombres de variable y funciones usados en el archivo del analizador de Bison comienzan con '`yy`' o '`YY`'. Esto incluye las funciones de interfaz tales como la función del analizador léxico `yylex`, la función de informe de errores `yyerror` y la propia función del analizador `yyparse`. Esto también incluye un gran número de identificadores utilizados para uso interno. Por lo tanto, debería evitar utilizar identificadores de C que comiencen con '`yy`' o '`YY`' en el archivo de la gramática de Bison excepto para aquellos definidos en este manual.

1.6 Etapas en el Uso de Bison

El proceso real de diseño de lenguajes utilizando Bison, desde la especificación de la gramática hasta llegar a un compilador o intérprete funcional, se compone de estas etapas:

1. Especificar formalmente la gramática en un formato que reconozca Bison (ver Capitulo 3 [Archivos de Gramática de Bison], página 45). Para cada regla gramatical en el lenguaje, describir la acción que se va a tomar cuando una instancia de esa regla sea reconocida. La acción se descibe por una secuencia de sentencias en C.
2. Escribir un analizador léxico para procesar la entrada y pasar tokens al analizador sintáctico. El analizador léxico podría escribirse a mano en C (ver Seccion 4.2 [La Función del Analizador Léxico `yylex`], página 61). Este puede también generarse utilizando Lex, pero el uso de Lex no se trata en este manual.
3. Escibir una función de control que llame al analizador producido por Bison.
4. Escribir las rutinas de infome de errores.

Para hacer que este código fuente escrito se convierta en un programa ejecutable, debe seguir estos pasos:

1. Ejecutar Bison sobre la gramática para producir el analizador.
2. Compilar el código de salida de Bison, al igual que cualquier otro fichero fuente.
3. Enlazar los ficheros objeto para producir el producto final.

1.7 El Formato Global de una Gramática de Bison

El fichero de entrada para la utilidad Bison es un *archivo de gramatica de Bison*. La forma general de una gramática de Bison es la siguiente:

```
%{
declaraciones en C
%}

Declaraciones de Bison

%%
Reglas gramaticales
%%
Código C adicional
```

Los '`%%`', '`%{`' y '`%}`' son signos de puntuación que aparecen en todo archivo de gramática de Bison para separar las secciones.

Las declaraciones en C podrían definir tipos y variables utilizadas en las acciones. Puede también usar comandos del preprocesador para definir macros que se utilicen ahí, y utilizar `#include` para incluir archivos de cabecera que realicen cualquiera de estas cosas.

Las declaraciones de Bison declaran los nombres de los símbolos terminales y no terminales, y también podrían describir la precedencia de operadores y los tipos de datos de los valores semánticos de varios símbolos.

Las reglas gramaticales definen cómo construir cada símbolo no terminal a partir de sus partes.

El código C adicional puede contener cualquier código C que desee utilizar. A menudo suele ir la definición del analizador léxico `yylex`, más subrutinas invocadas por las acciones en la reglas gramaticales. En un programa simple, todo el resto del programa puede ir aquí.

2 Ejemplos

Ahora presentaremos y explicaremos tres programas de ejemplo escritos utilizando Bison; una calculadora de notación polaca inversa, una calculadora de notación algebraica (infija), y una calculadora multi-función. Los tres han sido comprobados bajo BSD Unix 4.3; cada uno produce una utilizable, aunque limitada, calculadora de escritorio.

Estos ejemplos son simples, pero las gramáticas de Bison para lenguajes de programación reales se escriben de la misma manera.

2.1 Calculadora de Notación Polaca Inversa

El primer ejemplo es el de una simple calculadora de doble precisión de *notación polaca inversa* (una calculadora que utiliza operadores postfijos). Este ejemplo provee un buen punto de partida, ya que no hay problema con la precedencia de operadores. El segundo ejemplo ilustrará cómo se maneja la precendencia de operadores.

El código fuente para esta calculadora se llama '`rpcalc.y`'. La extensión '`.y`' es una convención utilizada para los archivos de entrada de Bison.

2.1.1 Declaraciones para `rpcalc`

Aqui están las declaraciones de C y Bison para la calculadora de notación polaca inversa. Como en C, los comentarios se colocan entre '`/*...*/`'.

```
/* Calculadora de notación polaca inversa. */

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

%token NUM

%% /* A continuación las reglas gramaticales y las acciones */
```

La sección de declaraciones en C (ver Seccion 3.1.1 [La Sección de Declaraciones en C], página 45) contiene dos directivas del preprocesador.

La directiva `#define` define la macro `YYSTYPE`, de este modo se especifica el tipo de dato de C para los valores semánticos de ambos, tokens y agrupaciones (ver Seccion 3.5.1 [Tipos de Datos de Valores Semánticos], página 50). El analizador de Bison utilizará cualquier tipo que se defina para `YYSTYPE`; si no lo define, por defecto es `int`. Como hemos especificado `double`, cada token y cada expresión tiene un valor asociado, que es un número en punto flotante.

La directiva `#include` se utiliza para declarar la función de exponenciación `pow`.

La segunda sección, declaraciones de Bison, provee información a Bison a cerca de los tipos de tokens (ver Seccion 3.1.2 [La Sección de Declaraciones de Bison], página 45). Cada símbolo terminal que no sea un caracter literal simple debe ser declarado aquí (Los caracteres literales simples no necesitan ser declarados.) En este ejemplo, todos los operadores aritméticos se designan por un caracter literal simple, así que el único símbolo terminal que necesita ser declarado es `NUM`, el tipo de token para las constantes numéricas.

2.1.2 Reglas Gramaticales para `rpcalc`

Aquí están las reglas gramaticales para una calculadora de notación polaca inversa.

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

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

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

Las agrupaciones del "lenguaje" de rpcalc definidas aquí son la expresión (con el nombre `exp`), la línea de entrada (`line`), y la transcripción completa de la entrada (`input`). Cada uno de estos símbolos no terminales tiene varias reglas alternativas, unidas por el puntuador '`|`' que se lee como "o". Las siguientes secciones explican lo que significan estas reglas.

La semántica del lenguaje se determina por las acciones que se toman cuando una agrupación es reconocida. Las acciones son el código C que aparecen entre llaves. Ver Seccion 3.5.3 [Acciones], página 50.

Debe especificar estas acciones en C, pero Bison facilita la forma de pasar valores semánticos entre las reglas. En cada acción, la pseudo-variable `$$` representa el valor semántico para la agrupación que la regla va a construir. El trabajo principal de la mayoría de las acciones es la asignación de un valor para `$$`. Se accede al valor semántico de los componentes de la regla con `$1`, `$2`, y así sucesivamente.

2.1.2.1 Explicación para `input`

Considere la definición de `input`:

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

Esta definición se interpreta así: “Una entrada completa es o una cadena vacía, o una entrada completa seguida por una línea de entrada”. Note que “entrada completa” se define en sus propios términos. Se dice que esta definición es *recursiva por la izquierda* ya que `input` aparece siempre como el símbolo más a la izquierda en la secuencia. Ver Seccion 3.4 [Reglas Recursivas], página 49.

La primera alternativa está vacía porque no hay símbolos entre los dos puntos y el primer ‘`|`’; esto significa que `input` puede corresponder con una cadena de entrada vacía (sin tokens). Escribimos estas reglas de esa manera porque es legítimo escribir *`Ctrl-d`* después de arrancar la calculadora. Es clásico poner una alternativa vacía al principio y escribir en esta el comentario ‘`/* vacío */`’.

La segunda alternativa de la regla (`input line`) maneja toda la entrada no trivial. Esta significa, “Después de leer cualquier número de líneas, leer una más si es posible”. La recursividad por la izquierda convierte esta regla en un bucle. Ya que la primera alternativa concuerda con la entrada vacía, el bucle se puede ejecutar cero o más veces.

La función `yyparse` del analizador continúa con el procesamiento de la entrada hasta que se encuentre con un error gramatical o el analizador diga que no hay más tokens de entrada; convendremos que esto último sucederá al final del fichero.

2.1.2.2 Explicación para `line`

Ahora considere la definición de `line`:

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

La primera alternativa es un token que es un caracter de nueva-línea; esta quiere decir que rpcalc acepta un línea en blanco (y la ignora, ya que no hay ninguna acción). La segunda alternativa es una expresión seguida de una línea nueva. Esta es la alternativa que hace que rpcalc sea útil. El valor semántico de la agrupación `exp` es el valor de `$1` porque la `exp` en cuestión es el primer símbolo en la alternativa. La acción imprime este valor, que es el resultado del cálculo que solicitó el usuario.

Esta acción es poco común porque no asigna un valor a `$$`. Como consecuencia, el valor semántico asociado con `line` está sin inicializar (su valor será impredecible). Se trataría de un error si ese valor se utilizara, pero nosotros no lo utilizaremos: una vez que rpcalc haya imprimido el valor de la línea de entrada del usuario, ese valor no se necesitará más.

2.1.2.3 Explicación para `expr`

La agrupación `exp` tiene varias reglas, una para cada tipo de expresión. La primera regla maneja la expresiones más simples: aquellas que son solamente números. La segunda maneja una expresión de adición, que tiene el aspecto de dos expresiones seguidas de un signo más. La tercera maneja la resta, y así sucesivamente.

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

Hemos utilizado '|' para unir las tres reglas de `exp`, pero igualmente podríamos haberlas escrito por separado:

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

La mayoría de las reglas tienen acciones que computan el valor de la expresión en términos del valor de sus componentes. Por ejemplo, en la regla de la adición, `$1` hace referencia al primer componenete `exp` y `$2` hace referencia al segundo. El tercer componente, `'+'`, no tiene un valor semántico asociado con significado, pero si tuviese alguno podría hacer referencia a este con `$3`. Cuando `yyparse` reconoce una expresión de suma usando esta regla, la suma de los valores de las dos subexpresiones producen el valor de toda la expresión. Ver Seccion 3.5.3 [Acciones], página 50.

Usted no tiene de dar una acción para cada regla. Cuando una regla no tenga acción, por defecto Bison copia el valor de `$1` en `$$`. Esto es lo que sucede en la primera regla (la que usa `NUM`).

El formato mostrado aquí es la convención recomendada, pero Bison no lo requiere. Puede añadir o cambiar todos los espacios en blanco que desee. Por ejemplo, esto:

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

expresa lo mismo que esto:

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

El último, sin embargo, es mucho más legible.

2.1.3 El Analizador Léxico de `rpcalc`

El trabajo del analizador léxico es el análisis a bajo nivel: la conversión de los caracteres o secuencia de caracteres en tokens. El analizador de Bison obtiene sus tokens llamando al analizador léxico. Ver Seccion 4.2 [La Función del Analizador Léxico `yylex`], página 61.

Solamente se necesita un analizador léxico sencillo para la calculadora RPN. Este analizador léxico ignora los espacios en blanco y los tabuladores, luego lee los números como `double` y los devuelve como tokens `NUM`. Cualquier otro caracter que no forme parte de un número es un token por separado. Tenga en cuenta que el código del token para un token de caracter simple es el propio caracter.

El valor de retorno de la función de análisis léxico es un código numérico que representa el tipo de token. El mismo texto que se utilizó en las reglas de Bison para representar el tipo de token también es una expresión en C con el valor numérico del tipo. Esto funciona de dos maneras. Si el tipo de token es un caracter literal, entonces su código numérico es el código ASCII de ese caracter; puede usar el mismo caracter literal en el analizador léxico para expresar el número. Si el tipo de token es un identificador, ese identificador lo define Bison como una macro en C cuya definición es un número apropiado. En este ejemplo, por lo tanto, `NUM` se convierte en una macro para que la use `yylex`.

El valor semántico del token (si tiene alguno) se almacena en la variable global `yylval`, que es donde el analizador de Bison lo buscará. (El tipo de datos de C para `yylval` es `YYSTYPE`, que se definió al principio de la gramática; ver Seccion 2.1.1 [Declaraciones para `rpcalc`], página 29.)

Se devuelve un código de tipo de token igual a cero cuando se llega al final del fichero. (Bison reconoce cualquier valor no positivo como indicador del final del fichero de entrada.)

Aquí está el código para el analizador léxico:

```
/* El analizador léxico devuelve un número en coma
   flotante (double) en la pila y el token NUM, o el
   caracter ASCII leído si no es un número.  Ignora
   todos los espacios en blanco y tabuladores,
   devuelve 0 como EOF. */

#include <ctype.h>

yylex ()
{
  int c;

  /* ignora los espacios en blanco */
  while ((c = getchar ()) == ' ' || c == '\t')
    ;
  /* procesa números   */
  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      scanf ("%lf", &yylval);
      return NUM;
    }
  /* devuelve fin-de-fichero  */
  if (c == EOF)
    return 0;
  /* devuelve caracteres sencillos */
  return c;
}
```

2.1.4 La Función de Control

Para continuar acordes a este ejemplo, la función de control se mantiene escueta al mínimo. El único requisito es que llame a `yyparse` para comenzar el proceso de análisis.

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

2.1.5 La Rutina de Informe de Errores

Cundo `yyparse` detecta un error de sintaxis, realiza una llamada a la función de informe de errores `yyerror` para que imprima un mensaje de error (normalmente pero no siempre un `"parse error"`). Es cosa del programador el proveer `yyerror` (ver Capitulo 4 [Interfaz con el Analizador en Lenguaje C], página 61), luego aquí está la definición que utilizaremos:

```
#include <stdio.h>

yyerror (s)  /* Llamada por yyparse ante un error */
     char *s;
{
  printf ("%s\n", s);
}
```

Después de que `yyerror` retorne, el analizador de Bison podría recuperarse del error y continuar analizando si la gramática contiene una regla de error apropiada (ver Capitulo 6 [Recuperacion de Errores], página 81). De otra manera, `yyparse` devolverá un valor distinto de cero. No hemos escrito ninguna regla de error en este ejemplo, así que una entrada no válida provocará que termine el programa de la calculadora. Este no es el comportamiento adecuado para una calculadora real, pero es adecuado en el primer ejemplo.

2.1.6 Ejecutando Bison para Hacer el Analizador

Antes de ejecutar Bison para producir un analizador, necesitamos decidir cómo ordenar todo en código fuente en uno o más ficheros fuente. Para un ejemplo tan sencillo, la manera más fácil es poner todo en un archivo. Las definiciones de `yylex`, `yyerror` y `main` van al final, en la sección de "código C adicional" del fichero. (ver Seccion 1.7 [El Formato Global de una gramática de Bison], página 27).

Para un proyecto más grande, probablemente tendría varios ficheros fuente, y utilizaría `make` para ordenar la recompilación de estos.

Con todo el fuente en un único archivo, utilice el siguiente comando para convertirlo en el fichero del analizador:

`bison` *nombre_archivo*`.y`

En este ejemplo el archivo se llamó '`rpcalc.y`' (de "Reverse Polish CALCulator", "Calculadora Polaca Inversa"). Bison produce un archivo llamado '*nombre_archivo*`.tab.c`', quitando el '`.y`' del nombre del fichero original. El fichero de salida de Bison contiene el código fuente para `yyparse`. Las funciones adicionales en el fichero de entrada (`yylex`, `yyerror` y `main`) se copian literalmente a la salida.

2.1.7 Compilando el Archivo del Analizador

Aquí está la forma de compilar y ejecutar el archivo del analizador:

```
# Lista los archivos en el directorio actual.
% ls
rpcalc.tab.c  rpcalc.y

# Compila el analizador de Bison.
# '-lm' le dice al compilador que busque la librería math para pow.
% cc rpcalc.tab.c -lm -o rpcalc

# Lista de nuevo los archivos.
% ls
rpcalc  rpcalc.tab.c  rpcalc.y
```

El archivo '`rpcalc`' contiene ahora el código ejecutable. He aquí una sesión de ejemplo utilizando `rpcalc`.

```
% rpcalc
4 9 +
13
3 7 + 3 4 5 *+-
-13
3 7 + 3 4 5 * + - n              Note el menos unario, 'n'
13
5 6 / 4 n +
-3.166666667
3 4 ^                            Exponenciación
81
^D                               Indicador de Fin-de-fichero
%
```

2.2 Calculadora de Notación Infija: `calc`

Ahora modificaremos rpcalc para que maneje operadores infijos en lugar de postfijos. La notación infija trae consigo el concepto de la precedencia de operadores y la necesidad de paréntesis anidados de profundidad arbitraria. Aquí está el código de Bison para '`calc.y`', una calculadora infija de escritorio.

```
/* Calculadora de notación infija--calc */

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

/* Declaraciones de BISON */
%token NUM
%left '-' '+'
```

```
%left '*' '/'
%left NEG     /* negación--menos unario */
%right '^'    /* exponenciación         */

/* A continuación la gramática */
%%
input:    /* cadena vacía */
        | input line
;

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

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

Las funciones `yylex`, `yyerror` y `main` pueden ser las mismas de antes.

Hay dos propiedades nuevas importantes presentadas en este código.

En la segunda sección (declaraciones de Bison), `%left` declara tipos de tokens y dice que son operadores asociativos por la izquierda. Las declaraciones `%left` y `%right` (asociatividad por la derecha) toma el lugar de `%token` que se utiliza para declarar un nombre de tipo de token sin asociatividad. (Estos tokens son caracteres literales simples, que de forma ordinaria no tienen que ser declarados. Los declaramos aquí para especificar la asociatividad.)

La precedencia de operadores se determina por el orden de línea de las declaraciones; cuanto más alto sea el número de línea de la declaración (esta esté más baja en la página o en la pantalla), más alta será la precedencia. Por tanto, la exponenciación tiene la precedencia más alta, el menos unario (`NEG`) es el siguiente, seguido por '`*`' y '`/`', y así sucesivamente. Ver Seccion 5.3 [Precedencia de Operadores], página 72.

La otra propiedad nueva importante es el `%prec` en la sección de la gramática para el operador menos unario. El `%prec` simplemente le dice a Bison que la regla '`| '-' exp`' tiene la misma precedencia que `NEG`—en este caso la siguiente a la más alta. Ver Seccion 5.4 [Precedencia Dependiende del Contexto], página 73.

Aquí hay un ejemplo de la ejecución de '`calc.y`':

```
% calc
4 + 4.5 - (34/(8*3+-3))
6.880952381
```

```
-56 + 2
-54
3 ^ 2
9
```

2.3 Recuperación de Errores Simple

Hasta este punto, este manual no ha tratado el tema de la *recuperación de errores*—cómo continuar analizando después de que el analizador detecte un error de sintaxis. Todo lo que hemos manejado es el informe de errores con `yyerror`. Tenga presente que por defecto `yyparse` retorna después de llamar a `yyerror`. Esto quiere decir que una línea de entrada errónea hace que el programa de la calculadora finalice. Ahora mostraremos cómo rectificar esta deficiencia.

El lenguaje de Bison por sí mismo incluye la palabra reservada `error`, que podría incluirse en las reglas de la gramática. En el siguiente ejemplo esta se ha añadido a una de las alternativas para `line`:

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

Esta ampliación a la gramática permite una recuperación de errores simple en caso de un error de análisis. Si se lee una expresión que no puede ser evaluada, el error será reconocido por la tercera regla de `line`, y el análisis continuará. (La función `yyerror` aún se sigue llamando para imprimir su mensaje también.) La acción ejecuta la sentencia `yyerrok`, una macro definida automáticamente por Bison; su significado es que la recuperación de errores ha terminado (ver Capitulo 6 [Recuperacion de Errores], página 81). Note la diferencia entre `yyerrok` y `yyerror`; no se trata de ninguna errata.

Esta forma de recuperación de errores trata con errores sintácticos. Existe otro tipo de errores; por ejemplo, la división entre cero, que conlleva una señal de excepción que normalmente es fatal. Una calculadora real debe tratar esta señal y utilizar `longjmp` para retornar a `main` y reanudar el análisis de líneas de entrada; también tendría que descartar el resto de la línea de entrada actual. No discutiremos esta cuestión más allá porque no es específica de los programas de Bison.

2.4 Calculadora Multi-Función: `mfcalc`

Ahora que se han explicado los conceptos básicos de Bison, es tiempo de movernos a problemas más avanzados. Las calculadoras anteriores ofrecían solamente cinco funciones, '`+`', '`-`', '`*`', '`/`' y '`^`'. Sería bueno tener una calculadora que dispusiera de otras funciones matemáticas tales como `sin`, `cos`, etc.

Es fácil añadir nuevos operadores a la calculadora infija siempre que estos sean únicamente caracteres literales simples. El analizador léxico `yylex` pasa todos lo caracteres no numéricos como tokens, luego basta con nuevas reglas gramaticales para añadir un nuevo operador. Pero lo que queremos es algo más flexible: funciones incorporadas cuya sintaxis tenga la siguiente forma:

nombre_función (*argumento*)

Al mismo tiempo, añadiremos memoria a la calculadora, permitiéndole crear variables con nombre, almacenar valores en ellas, y utilizarlas más tarde. Aquí hay una sesión de ejemplo con la calculadora multi-función:

```
% mfcalc
pi = 3.141592653589
3.1415926536
sin(pi)
0.0000000000
alpha = beta1 = 2.3
2.3000000000
alpha
2.3000000000
ln(alpha)
0.8329091229
exp(ln(beta1))
2.3000000000
%
```

Note que están permitidas las asignaciones múltiples y las funciones anidadas.

2.4.1 Declaraciones para `mfcalc`

Aquí están las declaraciones de C y Bison para la calculadora multi-función.

```
%{
#include <math.h>  /* Para funciones matemáticas, cos(), sin(), etc. */
#include "calc.h"  /* Contiene definición de `symrec'              */
%}
%union {
double     val;  /* Para devolver números                        */
symrec  *tptr;   /* Para devolver punteros a la tabla de símbolos    */
}

%token <val>  NUM        /* Número simple en doble precisión        */
%token <tptr> VAR FNCT   /* Variable y Función                      */
%type  <val>  exp

%right '='
%left '-' '+'
%left '*' '/'
%left NEG     /* Negación--menos unario */
%right '^'    /* Exponenciación        */

/* A continuación la gramática */

%%
```

La gramática anterior introduce únicamente dos nuevas propiedades del lenguaje de Bison. Estas propiedades permiten que los valores semánticos tengan varios tipos de datos. (ver Seccion 3.5.2 [Más de Un Tipo de Valor], página 50).

La declaración `%union` especifica la lista completa de tipos posibles; esta se encuentra en lugar de la definición de `YYSTYPE`. Los tipos permisibles son ahora double (para `exp` y `NUM`) y puntero a entrada en la tabla de símbolos. Ver Seccion 3.6.3 [La Colección de Tipos de Valores], página 57.

Ya que ahora los valores pueden tener varios tipos, es necesario asociar un tipo con cada símbolo gramatical cuyo valor semántico se utilice. Estos símbolos son `NUM`, `VAR`, `FNCT`, y `exp`. Sus declaraciones aumentan con la información a cerca de su tipo de dato (que se encuentra entre ángulos).

La construcción de Bison `%type` se utiliza para la declaración de símbolos no terminales, al igual que `%token` se utiliza para declarar tipos de tokens. No hemos usado `%type` anteriormente porque los símbolos no terminales se declaran implícitamente por las reglas que los definen. Pero `exp` debe ser declarado explícitamente para poder especificar el tipo de su valor. Ver Seccion 3.6.4 [Símbolos No Terminales], página 57.

2.4.2 Reglas Gramaticales para `mfcalc`

Aquí están las reglas gramaticales para la calculadora multi-función. La mayoría de ellas han sido copiadas directamente de `calc`; tres reglas, aquellas que mencionan a `VAR` o `FNCT`, son nuevas.

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

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

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

2.4.3 La Tabla de Símbolos de mfcalc

La calculadora multi-función requiere una tabla de símbolos para seguir la pista de los nombres y significado de las variables y funciones. Esto no afecta a las reglas gramaticales (excepto para las acciones) o las declaraciones de Bison, pero requiere algunas funciones de apoyo adicionales en C.

La tabla de símbolos de por sí contiene un lista enlazada de registros. Su definición, que está contenida en la cabecera 'calc.h', es la siguiente. Esta provee que, ya sean funciones o variables, sean colocadas en la tabla.

```
/* Tipo de datos para enlaces en la cadena de símbolos.      */
struct symrec
{
  char *name;  /* nombre del símbolo                    */
  int type;    /* tipo del símbolo: bien VAR o FNCT  */
  union {
    double var;           /* valor de una VAR        */
    double (*fnctptr)();  /* valor de una FNCT       */
  } value;
  struct symrec *next;    /* campo de enlace         */
};

typedef struct symrec symrec;

/* La tabla de símbolos: una cadena de 'struct symrec'.     */
extern symrec *sym_table;

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

La nueva versión de `main` incluye una llamada a `init_table`, una función que inicializa la tabla de símbolos. Aquí está esta, y también `init_table`:

```
#include <stdio.h>

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

yyerror (s)  /* Llamada por yyparse ante un error */
     char *s;
{
  printf ("%s\n", s);
}

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

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

/* La tabla de símbolos: una cadena de `struct symrec'.  */
symrec *sym_table = (symrec *)0;
init_table ()  /* pone las funciones aritméticas en una tabla. */
{
  int i;
  symrec *ptr;
  for (i = 0; arith_fncts[i].fname != 0; i++)
    {
      ptr = putsym (arith_fncts[i].fname, FNCT);
      ptr->value.fnctptr = arith_fncts[i].fnct;
    }
}
```

Mediante la simple edición de la lista de inicialización y añadiendo los archivos de inclusión necesarios, puede añadir funciones adicionales a la calculadora.

Dos funciones importantes permiten la localización e inserción de símbolos en la tabla de símbolos. A la función `putsym` se le pasa un nombre y el tipo (`VAR` o `FNCT`) del objeto a insertar. El objeto se enlaza por la cabeza de la lista, y devuelve un puntero al objeto. A la función `getsym` se le pasa el nombre del símbolo a localizar. Si se encuentra, se devuelve un punteo a ese símbolo; en caso contrario se devuelve un cero.

```
symrec *
putsym (sym_name,sym_type)
     char *sym_name;
     int sym_type;
{

  ptr = (symrec *) malloc (sizeof (symrec));
  ptr->name = (char *) malloc (strlen (sym_name) + 1);
  strcpy (ptr->name,sym_name);
  ptr->type = sym_type;
  ptr->value.var = 0; /* pone valor a 0 incluso si es fctn.  */
  ptr->next = (struct symrec *)sym_table;
  sym_table = ptr;
  return ptr;
}

symrec *
getsym (sym_name)
```

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

La función `yylex` debe reconocer ahora variables, valores numéricos, y los operadores aritméticos de caracter simple. Las cadenas de caracteres alfanuméricas que no comiencen con un dígito son reconocidas como variables o funciones dependiendo de lo que la tabla de símbolos diga de ellas.

La cadena de caracteres se le pasa a `getsym` para que la localice en la tabla de símbolos. Si el nombre aparece en la tabla, se devuelve a `yyparse` un puntero a su localización y su tipo (`VAR` o `FNCT`). Si no está ya en la tabla, entonces se inserta como `VAR` utilizando `putsym`. De nuevo, se devuelve a `yyparse` un puntero y su tipo (que debería ser `VAR`).

No se necesita ningún cambio en `yylex` para manejar los valores numéricos y los operadores aritméticos.

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

  /* Ignora espacios en blanco, obtiene el primer caracter */
  while ((c = getchar ()) == ' ' || c == '\t');

  if (c == EOF)
    return 0;
  /* Comienza un número => analiza el número.    */
  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      scanf ("%lf", &yylval.val);
      return NUM;
    }
  /* Comienza un identificador => lee el nombre. */
  if (isalpha (c))
    {
      symrec *s;
      static char *symbuf = 0;
      static int length = 0;
      int i;
```

```
        /* Inicialmente hace el buffer lo suficientemente
           largo para un nombre de símbolo de 40 caracteres. */
        if (length == 0)
          length = 40, symbuf = (char *)malloc (length + 1);

        i = 0;
        do
          {
            /* Si el buffer esta lleno, hacerlo mayor.        */
            if (i == length)
              {
                length *= 2;
                symbuf = (char *)realloc (symbuf, length + 1);
              }
            /* Añadir este caracter al buffer.                */
            symbuf[i++] = c;
            /* Obtiene otro caracter.                         */
            c = getchar ();
          }
        while (c != EOF && isalnum (c));

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

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

    /* Cualquier otro caracter es un token por sí mismo. */
    return c;
  }
```

Este programa es por ambos lados potente y flexible. Usted podría fácilmente añadir nuevas funciones, y es un trabajo sencillo modificar este código para introducir también variables predefinidas tales como `pi` o `e`.

2.5 Ejercicios

1. Añada algunas nuevas funciones de '`math.h`' a la lista de inicialización.
2. Añada otro array que contenga constantes y sus valores. Entonces modifique `init_table` para añadir estas constantes a la tabla de símbolos. Será mucha más fácil darle a las constantes el tipo `VAR`.
3. Hacer que el programa muestre un error si el usuario hace referencia a una variable sin inicializar de cualquier manera excepto al almacenar un valor en ella.

3 Archivos de Gramática de Bison

Bison toma como entrada la especificación de una gramática independiente del contexto y produce una función en lenguaje C que reconoce las instancias correctas de la gramática.

El archivo de entrada de la gramática de Bison tiene un nombre que finaliza por convención en '.y'.

3.1 Resumen de una Gramática de Bison

Un archivo de gramática de Bison tiene cuatro secciones principales, mostradas aquí con los delimitadores apropiados:

```
%{
Declaraciones en C
%}

Declaraciones en Bison

%%
Reglas Gramaticales
%%

Código C adicional
```

Los comentarios encerrados entre '/* ... */' pueden aparecer en cualquiera de las secciones.

3.1.1 La Sección de Declaraciones en C

La sección de *declaraciones en C* contiene definiciones de macros y declaraciones de funciones y variables que se utilizan en las acciones en las reglas de la gramática. Estas se copian al principio del archivo del analizador de manera que precedan la definición de `yyparse`. Puede utlilizar '`#include`' para obtener las declaraciones de un archivo de cabecera. Si no necesita ninguna declaración en C, puede omitir los delimitadores '`%{`' y '`%}`' que delimitan esta sección.

3.1.2 La Sección de Declaraciones de Bison

La sección de *declaraciones de Bison* contiene declaraciones que definen símbolos terminales y no terminales, especifica la precedencia, etc. En algunas gramáticas simples puede que no necesite ninguna de las declaraciones. Ver Seccion 3.6 [Declaraciones de Bison], página 55.

3.1.3 La Sección de Reglas Gramaticales

La sección de las *reglas gramaticales* contiene una o más reglas gramaticales, y nada más. Ver Seccion 3.3 [Sintaxis de las Reglas Gramaticales], página 48.

Debe haber siempre al menos una regla gramatical, y el primer '%%' (que precede a las reglas gramaticales) no puede ser omitido nunca incluso si es la primera cosa en el fichero.

3.1.4 La Sección de Código C Adicional

La sección de *código C adicional* se copia al pie de la letra a la salida del fichero del analizador, al igual que la sección de *declaraciones en C* que se copia al principio. Este es el lugar más conveniente para poner cualquier cosa que quiera tener en el archivo del analizador pero que no deba venir antes que la definición de `yyparse`. Por ejemplo, las definiciones de `yylex` e `yyerror` a menudo van ahí. Ver Capitulo 4 [Interfaz con el Analizador en Lenguaje C], página 61.

Si la última sección está vacía, puede omitir el '%%' que los separa de las reglas gramaticales.

El analizador de Bison en sí contiene muchas variables estáticas cuyos nombres comienzan con '`yy`' y muchas macros cuyos nombres comienzan con '`YY`'. Es una buena idea evitar el uso de cualquiera de estos nombres (excepto aquellos documentados en este menual) en la sección de código C adicional del archivo de la gramática.

3.2 Símbolos, Terminales y No Terminales

Los *símbolos* en las gramáticas de Bison representan las clasificaciones gramaticales del lenguaje.

Un *símbolo terminal* (también conocido como un *tipo de token*) representa una clase de tokens equivalentes sintácticamente. Usted utiliza el símbolo en las reglas de la gramática para indicar que está permitido un token en esa clase. El símbolo se representa en el analizador de Bison por un código numérico, y la función `yylex` devuelve un código de tipo de token para indicar qué tipo de token se ha leído. Usted no necesita conocer cual es el valor del código; puede utilizar el símbolo para representarlo.

Un *símbolo no terminal* representa una clase de agrupaciones sintácticamente equivalentes. El nombre del símbolo se utiliza para escribir las reglas gramaticales. Por convención, todos deberían escribirse en minúsculas.

Los nombres de los símbolos pueden contener letras, dígitos (no al principio), subrayados y puntos. Los puntos tienen sentido únicamente en no-terminales.

Hay tres maneras de escribir símbolos terminales en la gramática:

- Un *tipo de token designado* se escribe con un identifiacador, de la misma manera que un identificador en C. Por convención, debería estar todo en mayúsculas. Cada uno de estos nombres debe definirse con una declaración de Bison tal como `%token`. Ver Seccion 3.6.1 [Nombres de Tipo de Token], página 55.
- Un *tipo de token de caracter* (o *token de caracter literal*) se escribe en la gramática utilizando la misma sintaxis usada en C para las constantes de un caracter; por ejemplo, `'+'` es un tipo de token de caracter. Un tipo de token de caracter no necesita ser declarado a menos que necesite especificar el tipo de datos de su valor semántico (ver Seccion 3.5.1 [Tipo de Datos de Valores Semánticos], página 50), asociatividad, o precedencia (ver Seccion 5.3 [Precedencia de Operadores], página 72).

Por convención, un tipo de token de caracter se utiliza únicamente para representar un token que consista de ese caracter en particular. De este modo, el tipo de token `'+'` se utiliza para representar el caracter '`+`' como un token. No hay nada que obligue a seguir esta convención, pero si no lo hace, su programa será confuso para otros lectores.

Todas las secuencias usuales de escape que se utilizan en caracteres literales en C pueden ser utilizadas igualmente en Bison, pero no debe usar el caracter nulo como un caracter literal porque su codigo ASCII, el cero, es el código que `yylex` devuelve para el final de la entrada (ver Seccion 4.2.1 [Conveción de Llamada para `yylex`], página 61).

- Un *token de cadena literal* se escribe como un string constante de C; por ejemplo, `"<="` es un token de cadena literal. Un token de cadena literal no necesita ser declarado a menos que desee especificar el tipo de dato de su valor semántico (ver Seccion 3.5.1 [Tipo de Valor], página 50), asociatividad, precedencia (ver Seccion 5.3 [Precedencia], página 72).

  Puede asociar el token de cadena literal con un nombre simbólico como un alias, utilizando la declaración `%token` (ver Seccion 3.6.1 [Declaraciones de Tokens], página 55). Si no lo hace, el analizador léxico debe recuperar el número del token para el token de cadena literal desde la tabla `yytname` (ver Seccion 4.2.1 [Convenciones de Llamada], página 61).

  ADVERTENCIA: los tokens de cadena literal no funcionan en YACC.

  Por convención, un token de cadena literal se utiliza únicamente para representar un token que consiste en esa cadena en particular. Así, debería utilizar el tipo de token `"<="` para representar la cadena '`<=`' como un token. Bison no impone esta convención, pero si se aparta de ella, la gente que lea su programa se verá confusa.

  Todas las secuencias de escape utilizadas en las cadenas de literales de C pueden usarse igualmente en Bison. Un token de cadena literal debe contener dos o más caracteres; para un token que contenga un solo caracter, utilice un token de caracter (ver lo anterior).

El cómo se escoge la manera de escribir un símbolo no tiene efecto en su significado gramatical. Esto depende únicamente de dónde aparece en las reglas y cuándo la función de análisis sintáctico devuelve ese símbolo.

El valor devuelto por `yylex` es siempre uno de los símbolos terminlaes (ó 0 para el fin de la entrada). Sea cual sea la manera en la que escriba el tipo de token en las reglas gramaticales, escríbala de la misma manera en la definición de `yylex`. El código numérico para un tipo de token de caracter es simplemente el codigo ASCII para el caracter, así que `yylex` puede utilizar la constante idéntica del caracter para generar el código requerido. Cada tipo de token denominado se convierte en una macro en C en el fichero del analizador, de manera que `yylex` puede utilizar el nombre para hacer referencia al código. (Esta es la razón por la que los puntos no tienen sentido en los símbolos terminales.) Ver Seccion 4.2.1 [Convención de Llamada para `yylex`], página 61.

Si se define `yylex` en un archivo aparte, debe prepararlo para que las definiciones de las macros de los tipos de tokens estén disponibles allí. Utilice la opción '`-d`' cuando ejecute Bison, de esta forma se escribirán estas definiciones de las macros en un archivo de cabecera por separado '*nombre*`.tab.h`' que puede incluir en los otros archivos fuente que lo necesite. Ver Capitulo 9 [Invocando a Bison], página 89.

El símbolo `error` es un símbolo terminal reservado para la recuperación de errores (ver Capitulo 6 [Recuperacion de Errores], página 81); no debería utilizarlo para cualquier otro propósito. En particular, `yylex` nunca debería devolver este valor.

3.3 Sintaxis de las Reglas Gramaticales

Una regla gramatical de Bison tiene la siguiente forma general:

resultado: *componentes*...
        ;

donde *resultado* es el símbolo no terminal que describe esta regla y *componentes* son los diversos símbolos terminales y no terminales que están reunidos por esta regla (ver Seccion 3.2 [Simbolos], página 46).

Por ejemplo,

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

dice que dos agrupaciones de tipo `exp`, con un token '`+`' en medio, puede combinarse en una agrupación mayor de tipo `exp`.

Los espacios en blanco en las reglas son significativos únicamente para separar símbolos. Puede añadir tantos espacios en blanco extra como desee.

Distrubuídos en medio de los componentes pueden haber *acciones* que determinan la semántica de la regla. Una acción tiene el siguiente aspecto:

{*sentencias en C*}

Normalmente hay una única acción que sigue a los componentes. Ver Seccion 3.5.3 [Acciones], página 50.

Se pueden escribir por separado varias reglas para el mismo *resultado* o pueden unirse con el caracter de barra vertical '`|`' así:

resultado: *compoenentes-regla1*...
        | *componentes-regla2*...
        ...
        ;

Estas aún se consideran reglas distintas incluso cuando se unen de esa manera. Si los *componentes* en una regla están vacíos, significa que *resultado* puede concordar con la cadena vacía. Por ejemplo, aquí aparece cómo definir una secuencia separada por comas de cero o más agrupaciones `exp`:

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

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

Es habitual escribir el comentario '`/* vacío */`' en cada regla sin componentes.

3.4 Reglas Recursivas

Una regla se dice *recursiva* cuando su no-terminal *resultado* aparezca también en su lado derecho. Casi todas las gramáticas de Bison hacen uso de la recursión, ya que es la única manera de definir una secuencia de cualquier número de cosas. Considere esta definición recursiva de una secuencia de una o más expresiones:

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

Puesto que en el uso recursivo de `expseq1` este es el símbolo situado más a la izquierda del lado derecho, llamaremos a esto *recursión por la izquierda*. Por contraste, aquí se define la misma construcción utilizando *recusión por la derecha*:

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

Cualquier tipo de secuencia se puede definir utilizando ya sea la recursión por la izquierda o recursión por la derecha, pero debería utilizar siempre recursión por la izquierda, porque puede analizar una secuencia de elementos sin ocupar espacio de pila. La recursión por la derecha utiliza espacio en la pila de Bison en proporción al número de elementos en la secuencia, porque todos los elementos deben ser desplazados en la pila antes de que la regla pueda aplicarse incluso una única vez. Ver Capitulo 5 [El Algoritmo del Analizador de Bison], página 69, para una explicación adicional a cerca de esto.

La recursión *indirecta* o *mutua* sucede cuando el resultado de la regla no aparece directamente en su lado derecho, pero aparece en las reglas de otros no terminales que aparecen en su lado derecho.

Por ejemplo:

```
expr:     primario
        | primario '+' primario
        ;

primario:  constante
        | '(' expr ')'
        ;
```

define dos no-terminales recursivos mutuamente, ya que cada uno hace referencia al otro.

3.5 Definiendo la Semántica del Lenguaje

Las reglas gramaticales para un lenguaje determinan únicamente la sintaxis. La semántica viene determinada por los valores semánticos asociados con varios tokens y agrupaciones, y por las acciones tomadas cuando varias agrupaciones son reconocidas.

Por ejemplo, la calculadora calcula bien porque el valor asociado con cada expresión es el número apropiado; ésta suma correctamente porque la acción para la agrupación 'x + y' es sumar los números asociados con x e y.

3.5.1 Tipos de Datos para Valores Semánticos

En un programa sencillo podría ser suficiente con utilizar el mismo tipo de datos para los valores semánticos de todas las construcciones del lenguaje. Esto fue cierto en los ejemplos de calculadora RPN e infija (ver Seccion 2.1 [Calculadora de Notación Polaca Inversa], página 29).

Por defecto Bison utiliza el tipo `int` para todos los valores semánticos. Para especificar algún otro tipo, defina `YYSTYPE` como una macro, de esta manera:

```
#define YYSTYPE double
```

Esta definición de la macro debe ir en la sección de declaraciones en C del fichero de la gramática (ver Seccion 3.1 [Resumen de una Gramática de Bison], página 45).

3.5.2 Más de Un Tipo de Valor

En la mayoría de los programas, necesitará diferentes tipos de datos para diferentes clases de tokens y agrupaciones. Por ejemplo, una constante numérica podría necesitar el tipo `int` o `long`, mientras que una cadena constante necesita el tipo `char *`, y un identificador podría necesitar un puntero a la tabla de símbolos.

Para utilizar más de un tipo de datos para los valores semánticos en un analizador, Bison le pide dos cosas:

- Especificar la colección completa de tipos de datos posibles, con la declaración de Bison `%union` (ver Seccion 3.6.3 [La Colección de Tipos de Valores], página 57).
- Elegir uno de estos tipos para cada símbolo (terminal o no terminal) para los valores semánticos que se utilicen. Esto se hace para los tokens con la declaración de Bison `%token` (ver Seccion 3.6.1 [Nombres de Tipo de Token], página 55) y para las agrupaciones con la declaración de Bison `%type` (ver Seccion 3.6.4 [Símbolos No Terminales], página 57).

3.5.3 Acciones

Una acción acompaña a una regla sintáctica y contiene código C a ser ejecutado cada vez que se reconoce una instancia de esa regla. La tarea de la mayoría de las acciones es computar el valor semántico para la agrupación construida por la regla a partir de los valores semánticos asociados a los tokens o agrupaciones más pequeñas.

Una acción consiste en sentencias de C rodeadas por llaves, muy parecido a las sentencias compuestas en C. Se pueden situar en cualquier posición dentro de la regla; esta se ejecuta en esa posición. La mayoría de las reglas tienen sólo una acción al final de la regla, a continuación de todos los componentes. Las acciones en medio de una regla son difíciles y se utilizan únicamente para propósitos especiales (ver Seccion 3.5.5 [Acciones a Media Regla], página 52).

El código C en una acción puede hacer referencia a los valores semánticos de los componentes reconocidos por la regla con la construcción `$`*n*, que hace referencia al valor de la componente *n*-ésima. El valor semántico para la agrupación que se está construyendo es `$$`. (Bison traduce ambas construcciones en referencias a elementos de un array cuando copia las acciones en el fichero del analizador.)

Aquí hay un ejemplo típico:

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

Esta regla contruye una `exp` de dos agrupaciones `exp` más pequeñas conectadas por un token de signo más. En la acción, `$1` y `$3` hacen referencia a los valores semánticos de las dos agrupaciones `exp` componentes, que son el primer y tercer símbolo en el lado derecho de la regla. La suma se almacena en `$$` de manera que se convierte en el valor semántico de la expresión de adición reconocida por la regla. Si hubiese un valor semántico útil asociado con el token '`+`', debería hacerse referencia con `$2`.

Si no especifica una acción para una regla, Bison suministra una por defecto: `$$ = $1`. De este modo, el valor del primer símbolo en la regla se convierte en el valor de la regla entera. Por supuesto, la regla por defecto solo es válida si concuerdan los dos tipos de datos. No hay una regla por defecto con significado para la regla vacía; toda regla vacía debe tener una acción explícita a menos que el valor de la regla no importe.

`$`*n* con *n* cero o negativo se admite para hacer referencia a tokens o agrupaciones sobre la pila *antes de* aquellas que empareja la regla actual. Esta es una práctica muy arriesgada, y para utilizarla de forma fiable debe estar seguro del contexto en el que se aplica la regla. Aquí hay un donde puede utilizar esto de forma fiable:

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

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

Siempre que `bar` se utilice solamente de la manera mostrada aquí, `$0` siempre hace referencia a la `exp` que precede a `bar` en la definición de `foo`.

3.5.4 Tipos de Datos de Valores en Acciones

Si ha elegido un tipo de datos único para los valores semánticos, las construcciones `$$` y `$n` siempre tienen ese tipo de datos.

Si ha utilizado `%union` para especificar una variedad de tipos de datos, entonces debe declarar la elección de entre esos tipos para cada símbolo terminal y no terminal que puede tener un valor semántico. Entonces cada vez que utilice `$$` o `$n`, su tipo de datos se determina por el símbolo al que hace referencia en la regla. En este ejemplo,

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

`$1` y `$3` hacen referencia a instancias de `exp`, de manera que todos ellos tienen el tipo de datos declarado para el símbolo no terminal `exp`. Si se utilizase `$2`, tendría el tipo de datos declarado para el símbolo terminal `'+'`, cualquiera que pudiese ser.

De forma alternativa, puede especificar el tipo de datos cuando se hace referencia al valor, insertando '`<`*tipo*`>`' después del '`$`' al comienzo de la referencia. Por ejemplo, si ha definido los tipos como se muestra aquí:

```
%union {
  int tipoi;
  double tipod;
}
```

entonces puede escribir `$<tipoi>1` para hacer referencia a la primera subunidad de la regla como un entero, o `$<tipod>1` para referirse a este como un double.

3.5.5 Acciones a Media Regla

Ocasionalmente es de utilidad poner una acción en medio de una regla. Estas acciones se escriben como las acciones al final de la regla, pero se ejecutan antes de que el analizador llegue a reconocer los componentes que siguen.

Una acción en mitad de una regla puede hacer referencia a los componentes que la preceden utilizando `$n`, pero no puede hacer referencia a los componentes subsecuentes porque esta se ejecuta antes de que sean analizados.

Las acciones en mitad de una regla por sí mismas cuentan como uno de los componentes de la regla. Esto produce una diferencia cuando hay otra acción más tarde en la misma regla (y normalmente hay otra al final): debe contar las acciones junto con los símbolos cuando quiera saber qué número n debe utilizar en `$n`.

La acción en la mitad de una regla puede también tener un valor semántico. La acción puede establecer su valor con una asignación a `$$`, y las acciones posteriores en la regla pueden hacer referencia al valor utilizando `$n`. Ya que no hay un símbolo que identifique la acción, no hay manera de declarar por adelantado un tipo de datos para el valor, luego debe utilizar la construcción '`$<...>`' para especificar un tipo de datos cada vez que haga referencia a este valor.

No hay forma de establecer el valor de toda la regla con una acción en medio de la regla, porque las asignaciones a `$$` no tienen ese efecto. La única forma de establecer el valor para toda la regla es con una acción corriente al final de la regla.

Aquí hay un ejemplo tomado de un compilador hipotético, manejando una sentencia `let` de la forma '`let (`*variable*`)` *sentencia*' y sirve para crear una variable denominada *variable* temporalmente durante la duración de la *sentencia*. Para analizar esta construcción, debemos poner *variable* dentro de la tabla de símbolos mientras se analiza *sentencia*, entonces se quita después. Aquí está cómo se hace:

```
stmt:   LET '(' var ')'
                { $<contexto>$ = push_contexto ();
                  declara_variable ($3); }
        stmt    { $$ = $6;
                  pop_contexto ($<contexto>5); }
```

Tan pronto como '`let (`*variable*`)`' se haya reconocido, se ejecuta la primera acción. Esta guarda una copia del contexto semántico actual (la lista de variables accesibles) como su valor semántico, utilizando la alternativa `contexto` de la union de tipos de datos. Entonces llama a `declara_variable` para añadir una nueva variable a la lista. Una vez que finalice la primera acción, la sentencia inmersa en `stmt` puede ser analizada. Note que la acción en mitad de la regla es la componente número 5, así que '`stmt`' es la componente número 6.

Después de que la sentencia inmersa se analice, su valor semántico se convierte en el valor de toda la sentencia `let`. Entonces el valor semántico de la acción del principio se utiliza para recuperar la lista anterior de variables. Esto hace quitar la variable temporal del `let` de la lista de manera que esta no parecerá que exista mientras el resto del programa se analiza.

Tomar una acción antes de que la regla sea reconocida completamente a veces induce a conflictos ya que el analizador debe llegar a un análisis para poder ejecutar la acción. Por ejemplo, las dos reglas siguientes, sin acciones en medio de ellas, pueden coexistir en un analizador funcional porque el analizador puede desplazar el token de llave-abrir y ver qué sigue antes de decidir si hay o no una declaración:

```
compuesta: '{' declaracion sentencias '}'
         | '{' sentencias '}'
         ;
```

Pero cuando añadimos una acción en medio de una regla como a continuación, la regla se vuelve no funcional:

```
compuesta: { prepararse_para_variables_locales (); }
           '{' declaraciones sentencias '}'
         | '{' sentencias '}'
         ;
```

Ahora el analizador se ve forzado a decidir si ejecuta la acción en medio de la regla cuando no ha leído más alla de la llave-abrir. En otras palabras, debe decidir si utilia una regla u otra, sin información suficiente para hacerlo correctamente. (El token llave-abrir es lo que se llama el token *de preanálisis* en este momento, ya que el analizador está decidiendo aún qué hacer con él. Ver Seccion 5.1 [Tokens de Preanálisis], página 69.)

Podría pensar que puede corregir el problema poniendo acciones idénticas en las dos reglas, así:

```
compuesta: { prepararse_para_variables_locales (); }
             '{' declaraciones sentencias '}'
           | { prepararse_para_variables_locales (); }
             '{' sentencias '}'
          ;
```

Pero esto no ayuda, porque Bison no se da cuenta de que las dos acciones son idénticas. (Bison nunca intenta comprender el código C de una acción.)

Si la gramática es tal que una declaración puede ser distinguida de una sentencia por el primer token (lo que es cierto en C), entonces una solución que funciona es poner la acción después de la llave-abrir, así:

```
compuesta: '{' { prepararse_para_variables_locales (); }
           declaraciones sentencias '}'
         | '{' sentencias '}'
         ;
```

Ahora el primer token de la siguiente declaración o sentencia, que en cualquier caso diría a Bison la regla a utilizar, puede hacerlo aún.

Otra solución es introducir la acción dentro de un símbolo no terminal que sirva como una subrutina:

```
subrutina: /* vacío */
          { prepararse_para_variables_locales (); }
        ;

compuesta: subrutina
             '{' declaraciones sentencias '}'
           | subrutina
             '{' sentencias '}'
          ;
```

Ahora Bison puede ejecutar la acción en la regla para `subrutina` sin decidir qué regla utilzará finalmente para `compuesta`. Note que la acción está ahora al final de su regla. Cualquier acción en medio de una regla puede convertirse en una acción al final de la regla de esta manera, y esto es lo que Bison realmente hace para implementar acciones en mitad de una regla.

3.6 Declaraciones de Bison

La sección de *declaraciones de Bison* de una gramática de Bison define los símbolos utilizados en la formulación de la gramática y los tipos de datos de los valores semánticos. Ver Seccion 3.2 [Simbolos], página 46.

Todos los nombres de tipos de tokens (pero no los tokens de caracter literal simple tal como `'+'` y `'*'`) se deben declarar. Los símbolos no terminales deben ser declarados si necesita especificar el tipo de dato a utilizar para los valores semánticos (ver Seccion 3.5.2 [Más de Un Tipo de Valor], página 50).

La primera regla en el fichero también especifica el símbolo de arranque, por defecto. Si desea que otro símbolo sea el símbolo de arranque, lo debe declarar explícitamente (ver Seccion 1.1 [Lenguajes y Gramáticas Independientes del Contexto], página 23).

3.6.1 Nombres de Tipo de Token

La forma básica de declarar un nombre de tipo de token (símbolo terminal) es como sigue:

`%token` *nombre*

Bison convertirá esto es una directiva `#define` en el analizador, así que la función `yylex` (si está en este fichero) puede utilizar el nombre *nombre* para representar el código de este tipo de token.

De forma alternativa, puede utilizar `%left`, `%right`, o `%nonassoc` en lugar de `%token`, si desea especificar la precedencia. Ver Seccion 3.6.2 [Precedencia de Operadores], página 56.

Puede especificar explícitamente el código numérico para un token añadiendo un valor entero en el campo que sigue inmediatamente al nombre del token:

```
%token NUM 300
```

Es generalmente lo mejor, sin embargo, permitir a Bison elegir los códigos numéricos para todos los tipos de tokens. Bison automáticamente seleccionará los códigos que no provoquen conflictos unos con otros o con caracteres ASCII.

En el caso de que el tipo de la pila sea una union, debe aumentar `%token` u otra declaración de tokens para incluir la opción de tipo de datos delimitado por ángulos (ver Seccion 3.5.2 [Más de Un Tipo de Valor], página 50).

Por ejemplo:

```
%union {              /* define el tipo de la pila */
  double val;
  symrec *tptr;
}
%token <val> NUM       /* define el token NUM y su tipo */
```

Puede asociar un token de cadena literal con un nombre de tipo de token escribiendo la cadena literal al final de la declaración `%type` que declare el nombre. Por ejemplo:

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

Por ejemplo, una gramática para el lenguaje C podría especificar estos nombres con los tokens de cadena literal equivalente:

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

Una vez que iguale la cadena literal y el nombre del token, puede utilizarlo indistintamente en ulteriores declaraciones en reglas gramaticales. La función `yylex` puede utilizar el nombre del token o la cadena literal para obtener el número de código del tipo de token (ver Seccion 4.2.1 [Convenciones de Llamada], página 61).

3.6.2 Precedencia de Operadores

Use las declaraciones `%left`, `%right` o `%nonassoc` para declarar un token y especificar su precedencia y asociatividad, todo a la vez. Estas se llaman *declaraciones de precedencia*. Ver Seccion 5.3 [Precedencia de Operadores], página 72, para información general a cerca de la precedencia de operadores.

La sintaxis de una declaración de precedencia es la misma que la de `%token`: bien

`%left` *símbolos...*

o

`%left <`*tipo*`>` *símbolos...*

Y realmente cualquiera de estas declaraciones sirve para los mismos propósitos que `%token`. Pero además, estos especifican la asociatividad y precedencia relativa para todos los *símbolos*:

- La asociatividad de un operador *op* determina cómo se anidan los repetidos usos de un operador: si '*x op y op z*' se analiza agrupando *x* con *y* primero o agrupando *y* con *z* primero. `%left` especifica asociatividad por la izquierda (agrupando *x* con *y* primero) y `%right` especifica asociatividad por la derecha (agrupando *y* con *z* primero). `%nonassoc` especifica no asociatividad, que significa que '*x op y op z*' se considera como un error de sintaxis.
- La precedencia de un operador determina cómo se anida con otros operadores. Todos los tokens declarados en una sola declaración de precedencia tienen la misma precedencia y se anidan conjuntamente de acuerdo a su asociatividad. Cuando dos tokens declarados asocian declaraciones de diferente precedencia, la última en ser declarada tiene la mayor precedencia y es agrupada en primer lugar.

3.6.3 La Colección de Tipos de Valores

La declaración `%union` especifica la colección completa de posibles tipos de datos para los valores semánticos. La palabra clave `%union` viene seguida de un par de llaves conteniendo lo mismo que va dentro de una `union` en C.

Por ejemplo:

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

Esto dice que los dos tipos de alternativas son `double` y `symrec *`. Se les ha dado los nombres `val` y `tptr`; estos nombres se utilizan en las declaraciones de `%token` y `%type` para tomar uno de estos tipos para un símbolo terminal o no terminal (ver Seccion 3.6.4 [Símbolos No Terminales], página 57).

Note que, a diferencia de hacer una declaración de una `union` en C, no se escribe un punto y coma después de la llave que cierra.

3.6.4 Símbolos No Terminales

Cuando utilice `%union` para especificar varios tipos de valores, debe declarar el tipo de valor de cada símbolo no terminal para los valores que se utilicen. Esto se hace con una declaración `%type`, como esta:

`%type <`*tipo*`>` *noterminal*...

Aquí *noterminal* es el nombre de un símbolo no terminal, y *tipo* es el nombre dado en la `%union` a la alternativa que desee (ver Seccion 3.6.3 [La Colección de Tipos de Valor], página 57). Puede dar cualquier número de símbolos no terminales en la misma declaración `%type`, si tienen el mismo tipo de valor. Utilice espacios para separar los nombres de los símbolos.

Puede también declarar el tipo de valor de un símbolo terminal. Para hacer esto, utilice la misma construcción `<`*tipo*`>` en una declaración para el símbolo terminal. Todos las clases de declaraciones de tipos permiten `<`*tipo*`>`.

3.6.5 Suprimiendo Advertencias de Conflictos

Bison normalmente avisa si hay algún conflicto en la gramática (ver Seccion 5.2 [Conflictos Desplazamiento/Reducción], página 70), pero la mayoría de las gramáticas reales tienen conflictos desplazamiento/reducción inofensivos que se resuelven de una manera predecible y serían muy difíciles de eliminar. Es deseable suprimir los avisos a cerca de estos conflictos a menos que el número de conflictos cambie. Puede hacer esto con la declaración `%expect`.

La declaración tiene este aspecto:

```
%expect n
```

Aquí *n* es un entero decimal. La declaración dice que no deben haber avisos si hay *n* conflictos de desplazamiento/reducción y ningún conflicto reducción/reducción. Los avisos usuales se dan si hay más o menos conflictos, o si hay algún conflicto reducción/reducción.

En general, el uso de `%expect` implica estos pasos:

- Compilar su gramática sin `%expect`. Utilice la opción '`-v`' para obtener una lista amplia de dónde ocurrieron los conflictos. Bison también imprimirá el número de conflictos.
- Comprobar cada uno de los conflictos para estar seguro de que la resolución por defecto de Bison es lo que realmente quiere. Si no, reescriba la gramática y vuelva al principio.
- Añada una declaración `%expect`, copiando el número *n* a partir del número que imprime Bison.

Ahora Bison dejará de molestarle con los conflictos que ha comprobado, pero le avisará de nuevo si cambia el resultado de la gramática con conflictos adicionales.

3.6.6 El Símbolo de Arranque

Bison asume por defecto que el símbolo de arranque para la gramática es el primer no terminal que se encuentra en la sección de especificación de la gramática. El programador podría anular esta restricción con la declaración `%start` así:

```
%start símbolo
```

3.6.7 Un Analizador Puro (Reentrante)

Un programa *reentrante* es aquel que no cambia en el curso de la ejecución; en otras palabras, consiste enteramente de código *puro* (de sólo lectura). La reentrancia es importante siempre que la ejecución asíncrona sea posible; por ejemplo, un programa no reentrante podría no ser seguro al ser llamado desde un gestor de señales. En sistemas con múltiples hilos de control, un programa no reentrante debe ser llamado únicamente dentro de interbloqueos.

Normalmente, Bison genera un analizador que no es reentrante. Esto es apropiado para la mayoria de los casos, y permite la compatibilidad con YACC. (Los interfaces estandares de YACC son inherentemente no reentrantes, porque utilizan variables asignadas estáticamente para la comunicación con `yylex`, incluyendo `yylval` y `yylloc`.)

Por otra parte, puede generar un analizador puro, reentrante. La declaración de Bison `%pure_parser` dice que desea que el analizador sea reentrante. Esta aparece así:

```
%pure_parser
```

El resultado es que las variables de comunicación `yylval` y `yylloc` se convierten en variables locales en `yyparse`, y se utiliza una convención de llamada diferente para la función del analizador léxico `yylex`. Ver Seccion 4.2.4 [Convenciones de Llamada para Analizadores Puros], página 64, para los detalles a cerca de esto. La variable `yynerrs` también se convierte en local en `yyparse` (ver

Seccion 4.3 [La Función de Informe de Errores `yyerror`], página 65). La convención para llamar a `yyparse` no cambia.

Que el analizador sea o no puro no depende de las reglas gramaticales. Puede generar indistintamente un analizador puro o un analizador no reentrante a partir de cualquier gramática válida.

3.6.8 Sumario de Declaraciones de Bison

Aquí hay un sumario de todas las declaraciones de Bison:

`%union` Declara la colección de tipos de datos que los valores semánticos pueden poseer (ver Seccion 3.6.3 [La Colección de Tipos de Valores], página 57).

`%token` Declara un símbolo terminal (nombre de tipo de token) sin precedencia o asociatividad especificada (ver Seccion 3.6.1 [Nombres de Tipo de Token], página 55).

`%right` Declara un símbolo terminal (nombre de tipo de token) que es asociativo por la derecha (ver Seccion 3.6.2 [Precedencia de Operadores], página 56).

`%left` Declara un símbolo terminal (nombre de tipo de token) que es asociativo por la izquierda. (ver Seccion 3.6.2 [Precedencia de Operadores], página 56).

`%nonassoc`
Declara un símbolo terminal (nombre de tipo de token) que es no asociativo (utilizándolo de una forma que sería asociativo es un error de sintaxis) (ver Seccion 3.6.2 [Precedencia de Operadores], página 56).

`%type` Declara el tipo de valor semántico para un símbolo no terminal. (ver Seccion 3.6.4 [Símbolos No Terminales], página 57).

`%start` Especifica el símbolo de arranque de la gramática (ver Seccion 3.6.6 [El Símbolo de Arranque], página 58).

`%expect` Declara el número esperado de conflictos desplazamiento-reducción (ver Seccion 3.6.5 [Suprimiendo Advertencias de Conflictos], página 57).

`%pure_parser`
Solicita un programa de análisis puro (reentrante) (ver Seccion 3.6.7 [Un Analizador Puro (Reentrante)], página 58).

`%no_lines`
No genera ningún comando `#line` del proprocesador en el fichero del analizador. Normalmente Bison escribe estos comandos en el archivo del analizador de manera que el compilador de C y los depuradores asociarán los errores y el código objeto con su archivo fuente (el archivo de la gramática). Esta directiva provoca que asocien los errores con el archivo del analizador, tratándolo como un archivo fuente independiente por derecho propio.

`%raw` El archivo de salida '*nombre*`.h`' normalmente define los tokens con los números de token compatible con Yacc. Si se especifica esta opción, se utilizarán los números internos de Bison en su lugar. (Los números compatibles con Yacc comienzan en 257 excepto para los tokens de caracter simple; Bison asigna números de token secuencialmente para todos los tokens comenzando por 3.)

`%token_table`
Genera un array de nombres de tokens en el archivo del analizador. El nombre del array es `yytname`; `yytname[`*i*`]` es el nombre del token cuyo número de código de token interno de Bison es *i*. Los primeros tres elementos de `yytname` son siempre `"$"`,

"error", e "$illegal"; después de estos vienen los símbolos definidos en el archivo de la gramática.

Para tokens de caracter literal y tokens de cadena literal, el nombre en la tabla incluye los caracteres entre comillas simples o dobles: por ejemplo, "'+'" es un literal de caracter simple y "\"<=\"" es un token de cadena literal. Todos los caracteres del token de cadena literal aparecen textualmente en la cadena encontrada en la tabla; incluso los caracteres de comillas-dobles no son traducidos. Por ejemplo, si el token consiste de tres caracteres '*"*', su cadena en yytname contiene '"*"*"'. (En C, eso se escribiría como "\"*\"*\"").

Cuando especifique %token_table, Bison también generará definiciones para las macros YYNTOKENS, YYNNTS, y YYNRULES y YYNSTATES;

YYNTOKENS
: El número de token más alto, más uno.

YYNNTS
: El número de símbolos no terminales.

YYNRULES
: El número de reglas gramaticales,

YYNSTATES
: El número de estados del analizador (ver Seccion 5.5 [Estados del Analizador], página 74).

3.7 Múltiples Analizadores en el Mismo Programa

La mayoría de los programa que usan Bison analizan sólo un lenguaje y por lo tanto contienen sólo un analizador de Bison. Pero , ¿qué pasa si desea analizar más de un lenguaje con el mismo programa? Entonces necesita evitar un conflicto de nombres entre diferentes definiciones de yyparse, yylval, etc.

La manera más fácil de hacer esto es utilizar la opción '-p *prefijo*' (ver Capitulo 9 [Invocando a Bison], página 89). Esta renombra las funciones de interfaz y variables del analizador de Bison para comenzar con *prefijo* en lugar de 'yy'. Puede utilizarlo para darle a cada analizador nombres diferentes que no provoquen conflicto.

La lista precisa de símbolos renombrados es yyparse, yylex, yyerror, yynerrs, yylval, yychar e yydebug. Por ejemplo, si utiliza '-p c', los nombres se convierten em cparse, clex, etc.

El resto de las variables y macros asociadas con Bison no se renombran. Estas otras no son globales. Por ejemplo, YYSTYPE no se renombra, pero definirla de diferente forma en analizadores diferentes no provoca confusión (ver Seccion 3.5.1 [Tipos de Datos para Valores Semánticos], página 50).

La opción '-p' funciona añadiendo definiciones de macros al comienzo del archivo fuente del analizador, definiendo yyparse como *prefijo*parse, etc. Esto sustituye efectivamente un nombre por el otro en todo el fichero del analizador.

4 Interfaz del Analizador en Lenguaje C

El analizador de Bison es en realidad una función en C llamada `yyparse`. Aquí describimos las convenciones de interfaz de `yyparse` y las otras funciones que éste necesita usar.

Tenga en cuenta que el analizador utiliza muchos identificadores en C comenzando con '`yy`' e '`YY`' para propósito interno. Si utiliza tales identificadores (a parte de aquellos en este manual) en una acción o en codigo C adicional en el archivo de la gramática, es probable que se encuentre con problemas.

4.1 La Función del Analizador `yyparse`

Se llama a la función `yyparse` para hacer que el análisis comience. Esta función lee tokens, ejecuta acciones, y por último retorna cuando se encuentre con el final del fichero o un error de sintaxis del que no puede recuperarse. Usted puede también escribir acciones que ordenen a `yyparse` retornar inmediatamente sin leer más allá.

El valor devuelto por `yyparse` es 0 si el análisis tuvo éxito (el retorno se debe al final del fichero).

El valor es 1 si el análisis falló (el retorno es debido a un error de sintaxis).

En una acción, puede provocar el retorno inmediato de `yyparse` utilizando estas macros:

`YYACCEPT` Retorna inmediatamente con el valor 0 (para indicar éxito).

`YYABORT` Retorna inmediatamente con el valor 1 (para indicar fallo).

4.2 La Funcion del Analizador Léxico `yylex`

La función del *analizador léxico*, `yylex`, reconoce tokens desde el flujo de entrada y se los devuelve al analizador. Bison no crea esta función automáticamente; usted debe escribirla de manera que `yyparse` pueda llamarla. A veces se hace referencia a la función como el scanner léxico.

En programas simples, `yylex` se define a menudo al final del archivo de la gramática de Bison. Si `yylex` se define en un archivo fuente por separado, necesitará que las definiciones de las macros de tipos de tokens estén disponibles ahí. Para hecer esto, utilice la opción '`-d`' cuando ejecute Bison, de manera que éste escribirá esas definiciones de macros en un archivo de cabecera por separado '*nombre*`.tab.h`' que puede incluir en otros ficheros fuente que lo necesiten. Ver Capitulo 9 [Invocando a Bison], página 89.

4.2.1 Convención de Llamada para `yylex`

El valor que `yylex` devuelve debe ser un código numérico para el tipo de token que se ha encontrado, o 0 para el final de la entrada.

Cuando se hace referencia a un token en las reglas gramaticales con un nombre, ese nombre en el archivo del analizador se convierte en una macro de C cuya definición es el valor numérico apropiado para ese tipo de token. De esta manera `yylex` puede utilizar el nombre para indicar ese tipo. Ver Seccion 3.2 [Simbolos], página 46.

Cuando se hace referencia a un token en las reglas gramaticales por un caracter literal, el código numérico para ese caracter también es el código para el tipo de token. Así `yylex` puede simplemente devolver ese código de caracer. El caracter nulo no debe utilizarse de esta manera, porque su código es el cero y eso es lo que simboliza el final de la entrada.

Aquí hay un ejemplo mostrando estas cosas:

```
yylex ()
{
  ...
  if (c == EOF)     /* Detecta el fin de fichero. */
    return 0;
  ...
  if (c == '+' || c == '-')
    return c;       /* Asume que el tipo de token para '+' es '+'. */
  ...
  return INT;       /* Devuelve el tipo del token. */
  ...
}
```

Este interfaz se ha diseñado para que la salida de la utilidad `lex` pueda utilizarse sin cambios como definición de `yylex`.

Si la gramática utiliza tokens de cadena literal, hay dos maneras por las que `yylex` puede determianr los códigos de tipo de token para estos:

- Si la gramática define nombres de token simbólicos como alias para los tokens de cadena literal, `yylex` puede utilizar estos nombres simbólicos como los demás. En este caso, el uso de tokens de cadena literal en el archivo de la gramática no tiene efecto sobre `yylex`.
- `yylex` puede encontrar el token multi-caracter en la tabla `yytname`. El índice del token en la tabla es el código del tipo de token. El nombre de un token multi-caracter se almacena en `yytname` con una comilla doble, los caracteres del token, y otra comilla doble. Los caracteres del token no son traducidos de ninguna forma; ellos aparecen textualmente en el contenido de la cadena dentro de la tabla.

  Aquí está el código para localizar un token en `yytname`, asumiendo que los caracteres del token se almacenan en `token_buffer`.

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

La tabla yytname se genera sólo si se utiliza la declaración %token_table. Ver Seccion 3.6.8 [Sumario de Decls.], página 59.

4.2.2 Valores Semánticos de los Tokens

En un analizador ordinario (no reentrante), los valores semánticos del token deben almacenarse en la variable global yylval. Cuando esté usando un solo tipo de valores semánticos, yylval tiene ese tipo. Así, si el tipo es int (por defecto), podría escribir esto en yylex:

```
...
yylval = valor;  /* Pone valor en la pila de Bison. */
return INT;      /* Devuelve el tipo del token. */
...
```

Cuando esté utilizando varios tipos de datos, el tipo de yylval es una union compuesta a partir de la declaración %union (ver Seccion 3.6.3 [La Coleccion de Tipos de Valores], página 57). Así cuando almacene un valor de token, debe utilizar el miembro apropiado de la union. Si la declaración %union tiene el siguiente aspecto:

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

entonces el código en yylex podría ser así:

```
...
yylval.intval = valor; /* Pone el valor en la pila de Bison. */
return INT;            /* Devuelve el tipo del token. */
...
```

4.2.3 Posiciones en el Texto de los Tokens

Si está usando la propiedad '@n' (ver Seccion 4.4 [Propiedades Especiales para su Uso en Acciones], página 66) en acciones para seguir la pista de las posiciones en el texto de los tokens y agrupaciones, entonces debe proveer esta información en yylex. La función yyparse espera encontrar la posición en el texto de un token que se acaba de analizar en la variable global yylloc. Por ello yylex debe almacenar el dato apropiado en esa variable. El valor de yylloc es una estructura y solo tiene que inicializar los miembros que vayan a ser utilizados por las acciones. Los cuatro miembros se denominan first_line, first_column, last_line y last_column[1]. Note que el uso de estas características hacen al analizador notablemente más lento.

El tipo de dato de yylloc tiene el nombre YYLTYPE.

---

[1] primera línea, primera columna, última línea y última columna respectivamente

4.2.4 Convenciones de Llamada para Analizadores Puros

Cuando utilice la declaración `%pure_parser` para solicitar un analizador puro, reentrante, las variables globales de comunicación `yylval` y `yylloc` no pueden usarse. (Ver Seccion 3.6.7 [Un Analizador Puro (Reentrante)], página 58.) En tales analizadores las dos variables globales se reemplazan por punteros pasados como parámetros a `yylex`. Debe declararlos como se muestra aquí, y pasar la información de nuevo almacenándola a través de esos punteros.

```
yylex (lvalp, llocp)
     YYSTYPE *lvalp;
     YYLTYPE *llocp;
{
  ...
  *lvalp = valor;  /* Pone el valor en la pila de Bison.  */
  return INT;      /* Devolver el tipo del token.  */
  ...
}
```

Si el archivo de la gramática no utiliza la construcción '`@`' para hacer referencia a las posiciones del texto, entonces el tipo `YYLTYPE` no será definido. En este caso, omitir el segundo argumento; `yylex` será llamado con solo un argumento.

Si utiliza un analizador reentrante, puede opcionalmente pasar información de parámetros adicional de forma reentrante. Para hacerlo, defina la macro `YYPARSE_PARAM` como un nombre de variable. Esto modifica la función `yyparse` para que acepte un argumento, de tipo `void *`, con ese nombre.

Cuando llame a `yyparse`, pase la dirección de un objeto, haciendo una conversión de tipos de la dirección a `void *`. Las acciones gramaticales pueden hacer referencia al contenido del objeto haciendo una conversión del valor del puntero a su tipo apropiado y entonces derreferenciándolo. Aquí hay un ejemplo. Escriba esto en el analizador:

```
%{
struct parser_control
{
  int nastiness;
  int randomness;
};

#define YYPARSE_PARAM parm
%}
```

Entonces llame al analizador de esta manera:

```
struct parser_control
{
  int nastiness;
  int randomness;
};
```

```
...

{
  struct parser_control foo;
  ...  /* Almacena los datos apropiados en foo.  */
  value = yyparse ((void *) &foo);
  ...
}
```

En las acciones gramaticales, utilice expresiones como ésta para hacer referencia a los datos:

```
((struct parser_control *) parm)->randomness
```

Si desea pasar los datos de parámetros adicionales a `yylex`, defina la macro `YYLEX_PARAM` como `YYPARSE_PARAM`, tal como se muestra aquí:

```
%{
struct parser_control
{
  int nastiness;
  int randomness;
};

#define YYPARSE_PARAM parm
#define YYLEX_PARAM parm
%}
```

Debería entonces definir `yylex` para que acepte un argumento adicional—el valor de `parm`. (Este hace uno o tres argumentos en total, dependiendo de si se le pasa un argumento de tipo `YYLTYPE`.) Puede declarar el argumento como un puntero al tipo de objeto apropiado, o puede declararlo como `void *` y acceder al contenido como se mostró antes.

Puede utilizar '`%pure_parser`' para solicitar un analizador reentrante sin usar también `YYPARSE_PARAM`. Entonces debería llamar a `yyparse` sin argumentos, como es usual.

4.3 La Función de Informe de Errores `yyerror`

El analizador de Bison detecta un *error de análisis* o *error de sintaxis* siempre que lea un token que no puede satisfacer ninguna regla sintáctica. Una acción en la gramática puede también explícitamente declarar un error, utilizando la macro `YYERROR` (ver Seccion 4.4 [Propiedades Especiales para su Uso en Acciones], página 66).

El analizador de Bison espera advertir del error llamando a una función de informe de errores denominada `yyerror`, que se debe proveer. Esta es llamada por `yyparse` siempre que encuentre un error sintáctico, y ésta recibe un argumento. Para un error de análisis, la cadena normalmente es `"parse error"`.

Si define la macro `YYERROR_VERBOSE` en la sección de declaraciones de Bison (ver Seccion 3.1.2 [La Sección de Declaraciones de Bison], página 45), entonces Bison facilita una cadena de mensaje de error mas locuaz y específica que el simple "parse error". No importa qué definiciones utilice para YYERROR_VERBOSE, si ya lo define.

El analizador puede detectar otro tipo de error: desbordamiento de pila. Esto sucede cuando la entrada contiene construcciones que son profundamente anidadas. No parece que vaya a encontrarse con esto, ya que el analizador de Bison extiende su pila automáticamente hasta un límite muy largo. Pero si el desbordamiento sucede, yyparse llama a yyerror de la manera usual, excepto que la cadena del argumento es "parser stack overflow".

La siguiente definición es suficiente para programas simples:

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

Después yyerror retorna a yyparse, este último intentará la recuperación de errores si ha escrito reglas gramaticales de recuperación de errores apropiadas (ver Capitulo 6 [Recuperacion de Errores], página 81). Si la recuperación es imposible, yyparse devolverá inmediatamente un 1.

La variable yynerrs contiene el número de errores sintácticos hasta ahora. Normalmente esta variable es global; pero si solicita un analizador puro (ver Seccion 3.6.7 [Un Analizador Puro (Reentrante)], página 58) entonces es una variable local a la que sólo las acciones pueden acceder.

4.4 Propiedades Especiales para su Uso en Acciones

Aquí hay una tabla de construcciones, variables y macros que son útiles en las acciones.

'$$' Actúa como una variable que contiene el valor semántico para la agrupación construida por la regla actual. Ver Seccion 3.5.3 [Acciones], página 50.

'$*n*' Actúa como una variable que contiene el valor semántico para la componente *n*-ésima de la regla actual. Ver Seccion 3.5.3 [Acciones], página 50.

'$<*alttipo*>$'
Como $$ pero especifica la alternativa *alttipo* en la union especificada por la declaración %union. Ver Seccion 3.5.4 [Tipos de Datos de los Valores en Acciones], página 52.

'$<*alttipo*>*n*'
Como $*n* pero especifica la alternativa *alttipo* en la union especificada por la declaración %union. Ver Seccion 3.5.4 [Tipos de Datos de Valores en Acciones], página 52.

'YYABORT;'
Retorna inmediatamente desde yyparse, indicando fallo. Ver Seccion 4.1 [La Función del Analizador yyparse], página 61.

'YYACCEPT;'
Retorna inmediatamente desde yyparse, indicando éxito. Ver Seccion 4.1 [La Función del Analizador yyparse], página 61.

'YYBACKUP (*token*, *valor*);'
Deshace el desplazamiento de un token. Esta macro se permite únicamente para reglas que reducen un valor sencillo, y sólo donde no hay un token de preanálisis. Esta inserta

un token de preanálisis con un tipo de token *token* y valor semántico *valor*; entonces se descarta el valor que iba a ser reducido por esta regla.

Si la macro se utiliza cuando no es válida, tal como cuando ya hay un token de preanálisis, entonces se produce un error de sintaxis con el mensaje '`cannot back up`' y realiza la recuperación de errores ordinaria.

En cualquier caso, el resto de la acción no se ejecuta.

'`YYEMPTY`' El valor almacenado en `yychar` cuando no hay un token de preanálisis.

'`YYERROR;`'
Produce un error de sintaxis inmediatamente. Esta sentencia inicia la recuperación de errores como si el analizador hubiese detectado un error; sin embargo, no se llama a `yyerror`, y no imprime ningún mensaje. Si quiere que imprima un mensaje de error, llame a `yyerror` explícitamente antes de la sentencia '`YYERROR`'. Ver Capitulo 6 [Recuperacion de Errores], página 81.

'`YYRECOVERING`'
Esta macro representa una expresión que tiene el valor 1 cuando el analizador se está recuperando de un error de sintaxis, y 0 durante el resto del tiempo. Ver Capitulo 6 [Recuperacion de Errores], página 81.

'`yychar`' Variable que contiene el token actual de preanálisis. (En un analizador puro, esto es realmente una variable local dentro de `yyparse`.) Cuando no hay un token de preanálisis, el valor de `YYEMPTY` se almacena en la variable. Ver Seccion 5.1 [Tokens de Preanálisis], página 69.

'`yyclearin;`'
Descarta el token actual de preanálisis. Esto es útil principalmente en las reglas de error. Ver Capitulo 6 [Recuperacion de Errores], página 81.

'`yyerrok;`'
Deja de generar mensajes de error inmediatamente para los errores de sintaxis subsecuentes. Esto es útil principalmente en las reglas de error. Ver Capitulo 6 [Recuperacion de Errores], página 81.

'`@`*n*' Actúa como una variable estructurada conteniendo información de los números de línea y columna de la componente *n*-ésima de la regla actual. La estructura tiene cuatro miembros, así:

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

De esta manera, para obtener el número de línea de comienzo del tercer componente, utilizaría '`@3.first_line`'.

Para que los miembros de esta estructura contengan información válida, debe hacer que `yylex` facilite esta información para cada token. Si solo necesita de ciertos miembros, entonces `yylex` necesita únicamente rellenar esos miembros.

El uso de esta característica hace al analizador notablemente más lento.

5 El Algoritmo del Analizador de Bison

A medida que Bison lee tokens, los va insertando en una pila junto con su valor semántico. La pila se denomina *pila del analizador*. El insertar un token tradicionalmente se denomina *desplazamiento*.

Por ejemplo, suponga que la calculadora infija ha leído '`1 + 5 *`', con un '`3`' por venir. La pila contendrá cuatro elementos, uno para cada token que fue desplazado.

Pero la pila no siempre contiene un elemento para cada token leído. Cuando los últimos *n* tokens y agrupaciones desplazadas concuerden con los componentes de una regla gramatical, estos pueden combinarse de acuerdo a esa regla. A esto se le denomina *reducción*. Esos tokens y agrupaciones se reemplazan en la pila por una sola agrupación cuyo símbolo es el resultado (lado izquierdo) de esa regla. La ejecución de la acción de la regla es parte del proceso de reducción, porque ésta es la que computa el valor semántico de la agrupación resultante.

Por ejemplo, si el analizador de la calculadora infija contiene esto:

```
1 + 5 * 3
```

y el siguiente token de entrada es un caracter de nueva línea, entonces los tres últimos elementos pueden reducirse a 15 mediante la regla:

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

Entonces la pila contiene exactamente estos tres elementos:

```
1 + 15
```

En este punto, se puede realizar otra reducción, resultando en el valor 16. Entonces el token de nueva línea se puede desplazar.

El analizador intenta, mediante desplazamientos y reducciones, reducir la entrada completa a una sola agrupación cuyo símbolo es el símbolo de arranque de la gramática (ver Seccion 1.1 [Lenguajes y Gramáticas Independientes del Contexto], página 23).

Este tipo de analizador se conoce en la literatura como analizador ascendente.

5.1 Tokens de Preanálisis

El analizador de Bison *no* siempre reduce inmediatamente tan pronto como los últimos *n* tokens y agrupaciones se correspondan con una regla. Esto es debido a que es inadecuada una estrategia tan simple para manejar la mayoría de los lenguajes. En su lugar, cuando es posible una reducción, el analizador algunas veces "mira hacia delante" al próximo token para decidir qué hacer.

Cuando se lee un token, este no se desplaza inmediatamente; primero se convierte en el *token de preanálisis*, que no se pone sobre la pila. Ahora el analizador puede realizar una o más reducciones de tokens y agrupaciones sobre la pila, mientras que el token de preanálisis se mantiene fuera a un lado. Esto no significa que se han realizado todas las posibles reducciones; dependiendo del tipo de token del token de preanálisis, algunas reglas podrían escoger retrasar su aplicación.

Aquí hay un caso simple donde se necesita el token de preanálisis. Estas tres reglas definen expresiones que contienen operadores de suma binaria y operaciones factoriales unarios postfijos ('!'), y se permiten los paréntesis para agrupar.

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

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

Suponga que se ha leído el token '1 + 2' y ha sido desplazado; ¿qué debería hacerse? Si el próximo token es ')', entonces los primeros tres tokens deberían reducirse para formar una `expr`. Este es el único camino válido, porque el desplazamiento de ')' produciría una secuencia de símbolos `term ')'`, y ninguna regla lo permite.

Si el siguiente token es '!', entonces debe ser desplazado inmediatamente de manera que '2 !' se pueda reducir para hacer un `term`. Si en su lugar el analizador fuera a reducir antes de desplazar, '1 + 2' se convertiría en una `expr`. Sería entonces imposible desplazar el '!' porque haciéndolo produciría en la pila la secuencia de símbolos `expr '!'`. Ninguna regla permite esa secuencia.

El token de preanálisi actual se almacena en la variable `yychar`. Ver Seccion 4.4 [Propiedades Especiales para su Uso en Acciones], página 66.

5.2 Conflictos de Desplazamiento/Reducción

Suponga que estamos analizando un lenguaje que tiene las sentencias if-then y if-then-else, con un par de reglas como estas:

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

Aquí asumimos que `IF`, `THEN` y `ELSE` son símbolos terminales de tokens para palabras clave específicas.

Cuando se lea el token `ELSE` y se convierta en el token de preanálisis, el contenido de la pila (asumiendo que la entrada es válida) está listo para una reducción por la primera regla. Pero también es legítimo desplazar el `ELSE`, porque eso conllevaría a una redución provisional por la segunda regla.

Esta situación, donde sería válido un desplazamiento o una reducción, se denomina un *conflicto desplazamiento/reducción*. Bison está diseñado para resolver estos conflictos eligiendo el desplazamiento, a menos que se le dirija con declaraciones de precedencia de operadores. Para ver la razón de esto, vamos a contrastarlo con la otra alternativa.

Ya que el analizador prefiere desplazar el `ELSE`, el resultado sería ligar la cláusula else con la sentencia if más interior, haciendo que estas dos entradas sean equivalentes:

```
if x then if y then gana (); else pierde;

if x then do; if y then gana (); else pierde; end;
```

Pero si el analizador escoge la reducción cuando es posible en lugar de desplazar, el resultado sería ligar la cláusula else a la sentencia if más exterior, haciendo que estas dos entradas sean equivalentes:

```
if x then if y then gana (); else pierde;

if x then do; if y then gana (); end; else pierde;
```

El conflicto existe porque la gramática escrita es ambigua: en todo caso el análisis de la sentencia if simple anidada es legítima. La convención es que estas ambigüedades se resuelvan emparejando la cláusula else a la sentencia if más interior; esto es lo que Bison consigue eligiendo el desplazamiento en vez de la reducción. (Idealmente sería más adecuado escribir una gramática no ambigua, pero eso es muy duro de hacer en este caso.) Esta ambigüedad en particular se encontró en primer lugar en la especificación de Algol 60 y se denominó la ambiguedad del "balanceado del `else`".

Para evitar advertencias de Bison a cerca de los predecibles, legítimos conflictos de desplazamiento/reducción, utilice la declaración `%expect` *n*. No se producirán avisos mientras el número de conflictos de desplazamiento/reducción sea exactamente *n*. Ver Seccion 3.6.5 [Suprimiendo Advertencias de Confictos], página 57.

La definición de `if_stmt` anterior es la única que se va a quejar del conflicto, pero el conflicto no aparecerá en realidad sin reglas adicionales. Aquí hay un fichero de entrada de Bison completo que manfiesta realmente el conflicto:

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

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

expr:     variable
        ;
```

5.3 Precedencia de Operadores

Otra situación en donde pararecen los conflictos desplazamiento/reducción es en las expresiones aritméticas. Aquí el desplazamiento no siempre es la resolución preferible; las declaraciones de Bison para la precedencia de operadores le permite especificar cuándo desplazar y cuando reducir.

5.3.1 Cuándo se Necesita la Precedencia

Considere el siguiente fragmento de gramática ambigua (ambigua porque la entrada '1 - 2 * 3' puede analizarse de dos maneras):

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

Suponga que el analizador ha visto los tokens '1', '-' y '2'; ¿debería reducirlos por la regla del operador de adición? Esto depende del próximo token. Por supuesto, si el siguiente token es un ')', debemos reducir; el desplazamiento no es válido porque ninguna regla puede reducir la secuencia de tokens '- 2)' o cualquier cosa que comience con eso. Pero si el próximo token es '*' o '<', tenemos que elegir: ya sea el desplazamiento o la reducción permitiría al analizador terminar, pero con resultados diferentes.

Para decidir qué debería hacer Bison, debemos considerar los resultados. Si el siguiente token de operador *op* se desplaza, entonces este debe ser reducido primero para permitir otra oportunidad para reducir la suma. El resultado es (en efecto) '1 - (2 *op* 3)'. Por otro lado, si se reduce la resta antes del desplazamiento de *op*, el resultado es '(1 - 2) *op* 3'. Claramente, entonces, la elección de desplazar o reducir dependerá de la precedencia relativa de los operadores '-' y *op*: '*' debería desplazarse primero, pero no '<'.

¿Qué hay de una entrada tal como '1 - 2 - 5'; debería ser esta '(1 - 2) - 5' o debería ser '1 - (2 - 5)'? Para la mayoría de los operadores preferimos la primera, que se denomina *asociación por la izquierda*. La última alternativa, *asociación por la derecha*, es deseable para operadores de asignación. La elección de la asociación por la izquierda o la derecha es una cuestión de qué es lo que el analizador elige si desplazar o reducir cuando la pila contenga '1 - 2' y el token de preanálisis sea '-': el desplazamiento produce asociatividad por la derecha.

5.3.2 Especificando Precedencia de Operadores

Bison le permite especificar estas opciones con las declaraciones de precedencia de operadores `%left` y `%right`. Cada una de tales declaraciones contiene una lista de tokens, que son los operadores cuya precedencia y asociatividad se está declarando. La declaración `%left` hace que todos esos operadores sean asociativos por la izquierda y la declaración `%right` los hace asociativos por la derecha. Una tercera alternativa es `%nonassoc`, que declara que es un error de sintaxis encontrar el mismo operador dos veces "en un fila".

La precedencia relativa de operadores diferentes se controla por el orden en el que son declarados. La primera declaración `%left` o `%right` en el fichero declara los operadores cuya precedencia es la menor, la siguiente de tales declaraciones declara los operadores cuya precedencia es un poco más alta, etc.

5.3.3 Ejemplos de Precedencia

En nuestro ejemplo, queríamos las siguientes declaraciones:

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

En un ejemplo más completo, que permita otros operadores también, los declararíamos en grupos de igual precedencia. Por ejemplo, `'+'` se declara junto con `'-'`:

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

(Aquí `NE` y el resto representan los operadores para “distinto”, etc. Asumimos que estos tokens son de más de un caracter de largo y por lo tanto son representados por nombres, no por caracteres literales.)

5.3.4 Cómo Funciona la Precedencia

El primer efecto de las declaraciones de precedencia es la asignación de niveles de precedencia a los símbolos terminales declarados. El segundo efecto es la asignación de niveles de precedencia a ciertas reglas: cada regla obtiene su precedencia del último simbolo terminal mencionado en las componentes. (También puede especificar explícitamente la precedencia de una regla. Ver Seccion 5.4 [Precedencia Dependiente del Contexto], página 73.)

Finalmente, la resolución de conflictos funciona comparando la precendecia de la regla que está siendo considerada con la del token de preanálisis. Si la precedencia del token es más alta, la elección es desplazar. Si la precedencia de la regla es más alta, la elección es reducir. Si tienen la misma precedencia, la elección se hace en base a la asociatividad de ese nivel de precedencia. El archivo de salida amplia producido por ‘`-v`’ (ver Capitulo 9 [Invocando a Bison], página 89) dice cómo fue resuelto cada conflicto.

No todas las reglas y no todos los tokens tienen precedencia. Si bien la regla o el token de preanálisis no tienen precedencia, entonces por defecto de desplaza.

5.4 Precedencia Dependiente del Contexto

A menudo la precedencia de un operador depende del contexto. Esto suena raro al principio, pero realmente es muy común. Por ejemplo, un signo menos típicamente tiene una precedencia muy alta como operador unario, y una precedencia algo menor (menor que la multiplicación) como operador binario.

Las declaraciones de precedencia de Bison, `%left`, `%right` y `%nonassoc`, puede utilizarse únicamente para un token dado; de manera que un token tiene sólo una precedencia declarada de esta manera. Para la precedencia dependiente del contexto, necesita utilizar un mecanismo adicional: el modifidor `%prec` para las reglas.

El modificador `%prec` declara la precedencia de una regla en particular especificando un símbolo terminal cuya precedencia debe utilizarse para esa regla. No es necesario por otro lado que ese símbolo aparezca en la regla. La sintaxis del modificador es:

`%prec` *símbolo-terminal*

y se escribe despúes de los componentes de la regla. Su efecto es asignar a la regla la precedencia de *símbolo-terminal*, imponiéndose a la precedencia que se deduciría de forma ordinaria. La precedencia de la regla alterada afecta enconces a cómo se resuelven los conflictos relacionados con esa regla (ver Seccion 5.3 [Precedencia de Operadores], página 72).

Aquí está cómo `%prec` resuelve el problema del menos unario. Primero, declara una precedencia para un símbolo terminal ficticio llamada `UMINUS`. Aquí no hay tokens de este tipo, pero el símbolo sirve para representar su precedencia:

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

Ahora la precedencia de `UMINUS` se puede utilizar en reglas específicas:

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

5.5 Estados del Analizador

La función `yyparse` se implementa usando una máquina de estado finito. Los valores insertados sobre la pila no son únicamente códigos de tipos de tokens; estos representan toda la secuencia de símbolos terminales y no terminales en o cerca del tope de la pila. El estado actual colecciona toda esta información sobre la entrada anterior que es relevante para decidir qué hacer a continuación.

Cada vez que se lee un token de preanálisis, el estado actual del analizador junto con el tipo de token de preanálisis se localizan en una tabla. Esta entrada en la tabla puede decir, "Desplace el token de preanálisis." En este caso, también especifica el nuevo estado del analizador, que se pone sobre el tope de la pila del analizador. O puede decir, "Reduzca utilizando la regla número *n*." Esto quiere decir que un cierto número de tokens o agrupaciones se sacan de la pila, y se reemplazan por una agrupación. En otras palabras, se extrae ese número de estados de la pila, y se empuja un nuevo estado.

Hay otra alternativa: la tabla puede decir que el token de preanálisis es erróneo en el estado actual. Esto provoca que comience el procesamiento de errores (ver Capitulo 6 [Recuperacion de Errores], página 81).

5.6 Conflictos de Reducción/Reducción

Se produce un conflicto de reducción/reducción si hay dos o más reglas que pueden aplicarse a la misma secuencia de entrada. Esto suele indicar un error serio en la gramática.

Por ejemplo, aquí hay un intento fallido de definir una secuencia de cero o más agrupaciones de `palabra`.

```
secuencia: /* vacío */
                { printf ("secuencia vacía\n"); }
         | posiblepalabra
         | secuencia palabra
                { printf ("palabra añadida %s\n", $2); }
         ;
posiblepalabra: /* vacío */
                { printf ("posiblepalabra vacío\n"); }
        | palabra
                { printf ("palabra sencilla %s\n", $1); }
        ;
```

El error es una ambigüedad: hay más de una manera de analizar una `palabra` sencilla en una `secuencia`. Esta podría reducirse a una `posiblepalabra` y entonces en una `secuencia` mediante la segunda regla. Alternativamente, la cadena vacía se prodría reducir en una `secuencia` mediante la primera regla, y esto prodría combinarse con la `palabra` utilizando la tercera regla para `secuencia`.

Existe también más de una manera de reducir la cadena vacía en una `secuencia`. Esto se puede hacer directamente mediante la primera regla, o indirectamente mediante `posiblepalabra` y entonces la segunda regla.

Podría pensar que esto es una distinción sin ninguna diferencia, porque esto no cambia si una entrada particular es válida o no. Pero sí afecta en las acciones que son ejecutadas. Una ordenación del análisis ejecuta la acción de la segunda regla; la otra ejecuta la acción de la primera regla y la acción de la tercera regla. En este ejemplo, la salida del programa cambia.

Bison resuelve un conflicto reducción/reducción eligiendo el uso de la regla que aparece en primer lugar dentro de la gramática, pero es muy arriesgado contar con esto. Cada conflicto de reducción/reducción se debe estudiar y normalmente eliminar. Aquí está la manera apropiada de definir `secuencia`:

```
secuencia: /* vacío */
                { printf ("secuencia vacía\n"); }
         | secuencia palabra
                { printf ("palabra añadida %s\n", $2); }
         ;
```

Aquí hay otro error común que produce un conflicto reducción/reducción.

```
secuencia: /* vacío */
         | secuencia palabras
         | secuencia redirecciones
         ;

palabras: /* vacío */
        | palabras palabra
        ;

redirecciones: /* vacío */
             | redirecciones redireccion
             ;
```

La intención aquí es definir una secuencia que pueda contener ya sea agrupaciones `palabra` o `redireccion`. Las definiciones individuales de `secuencia`, `palabras` y `redirecciones` están libres de errores, pero las tres juntas producen una ambigüedad sutil: ¡incluso una entrada vacía puede analizarse en una infinidad de maneras diferentes!

Considere esto: la cadena vacía podría ser una `palabras`. O podrían ser dos `palabras` en una fila, o tres, o cualquier número. Podría igualmente ser una `redirecciones`, o dos, o cualquier número. O podría ser una `palabras` seguido de tres `redirecciones` y otra `palabras`. Y así sucesivamente.

Aquí hay dos maneras de corregir estas reglas. Primero, convertirlo en una secuencia de un sólo nivel:

```
secuencia: /* vacío */
         | secuencia palabra
         | secuencia redireccion
         ;
```

Segundo, prevenir bien un `palabras` o un `redireccion` de que sea vacío:

```
secuencia: /* vacío */
         | secuencia palabras
         | secuencia redirecciones
         ;

palabras: palabra
        | palabras palabra
        ;

redirecciones: redireccion
           | redirecciones redireccion
           ;
```

5.7 Conflictos Misteriosos de Reducción/Reducción

Algunas veces con los conflictos reducción/reducción puede suceder que no parezcan garantizados. Aquí hay un ejemplo:

```
%token ID

%%
def:    espc_param espc_return ','
        ;
espec_param:
             tipo
        |    lista_nombre ':' tipo
        ;
espec_return:
             tipo
        |    nombre ':' tipo
        ;
tipo:        ID
        ;
nombre:        ID
        ;
lista_nombre:
             nombre
        |    nombre ',' lista_nombre
        ;
```

Parecería que esta gramática puede ser analizada con sólo un único token de preanálisis: cuando se está leyendo un `espc_param`, un `ID` es un `nombre` si le sigue una coma o un punto, o un `tipo` si le sigue otro `nombre`. En otras palabras, esta gramática es LR(1).

Sin embargo, Bison, como la mayoría de los generadores de analizadores sintácticos, no pueden en realidad manejar todas las gramáticas LR(1). En esta gramática, los dos contextos, aquél después de un `ID` al principio de un `espc_param` y también al principio de un `espc_return`, son lo suficientemente similares para que Bison asuma que son el mismo. Estos parecen similares porque estarían activos el mismo conjunto de reglas—la regla de reducción a un `nombre` y aquella para la reducción de `tipo`. Bison es incapaz de determinar a ese nivel de procesamiento que las reglas requerirían diferentes tokens de preanálisis en los dos contextos, así que construye un solo estado del analizador para ambos. Combinando los dos contextos provoca un conflicto más tarde. En la terminología de los analizadores sintácticos, este suceso significa que la gramática no es LALR(1).

En general, es mejor arreglar las deficiencias que documentarlas. Pero esta deficiencia en particular es intrínsecamente difícil de arreglar; los generadores de analizadores sintácticos que pueden manejar gramáticas LR(1) son duros de escribir y tienden a producir analizadores que son muy grandes. En la práctica, Bison es más útil como es ahora.

Cuando el problema aparece, puede a veces arreglarlo identificando los dos estados del analizador que están siendo confundidos, y añadir algo para hacerlos pareceer distintos. En el ejemplo anterior, añadiendo una regla a `espc_return` como a continuación hace que el problema desaparezca:

```
%token BOGUS
...
%%
...
espc_return:
             tipo
        |    nombre ':' tipo
        /* Esta regla nunca se usa.  */
        |    ID BOGUS
        ;
```

Esto corrige el problema porque introduce la posibilidad de una regla activa adicional en el contexto después de `ID` al principio de un `espc_param`, así que los dos contextos reciben estados distinto del analizador. Siempre que el token `BOGUS` no se genere nunca por `yylex`, la regla adicional no podrá alterar la manera en la que la entrada es analizada.

En este ejemplo en particular, hay otra forma de resolver este problema: reescribir la regla de `espc_return` para usar `ID` directamente en lugar de hacerlo a través de `nombre`. Esto también provoca que los dos contextos confusos tengan conjuntos de reglas activas distintas, porque la de `espc_return` activa la regla alterada para `espc_return` en vez que la de `nombre`.

```
espc_param:
             tipo
        |    lista_nombre ':' tipo
        ;
espc_return:
             tipo
        |    ID ':' tipo
        ;
```

5.8 Desbordamiento de Pila, y Cómo Evitarlo

La pila del analizador de Bison puede desbordarse si se desplazan demasiados tokens y no son reducidos. Cuando esto sucede, la función del analizador `yyparse` devuelve un valor distinto de cero, haciendo pausas solamente para llamar a `yyerror` para informar del desbordamiento.

Definiendo la macro `YYMAXDEPTH`, puede controlar cuán profundo puede llegar a ser la pila del analizador antes de que suceda un desbordamiento de la pila. Defina esta macro con un valor que sea un entero. Este valor es el número máximo de tokens que pueden ser desplazados (y sin ser reducidos) antes de un desbordamiento. Debe ser una expresión constante cuyo valor se conozca en tiempo de compilación.

El espacio de la pila permitido no es asignado necesariamente. Si especifica un valor grande para `YYMAXDEPTH`, el analizador realmente asigna un pequeña pila al principio, y entonces la hace mayor por etapas a medida que se necesite. Esta asignación incremental ocurre automáticamente y silenciosamente. Por lo tanto, no necesita hacer a `YYMAXDEPTH` angustiosamente pequeño meramente para ahorrar espacio para entradas ordinarias que no necesitan mucha pila.

El valor por defecto de `YYMAXDEPTH`, si no lo define, es 10000.

Usted puede controlar cuánta pila se asigna inicialmente definiendo la macro `YYINITDEPTH`. Este valor también debe ser una constante entera en tiempo de compilación. El valor por defecto es 200.

6 Recuperación de Errores

Normalmente no es aceptable que un programa termine ante un error de análisis. Por ejemplo, un compilador debería recuperarse lo suficiente como para que analice el resto del archivo de entrada y comprobar sus errores; una calculadora debería aceptar otra expresión.

En un analizador de comandos interactivo sencillo donde cada entrada es una línea, podría ser suficiente con permitir a `yyparse` devolver un 1 ante un error y hacer que la función invocadora ignore el resto de la línea de entrada cuando suceda (y entonces llamar a `yyparse` de nuevo). Pero esto es inadecuado para un compilador, porque olvida todo el contexto sintáctico desde el comienzo hasta donde se encontró el error. Un error de sintaxis profundo dentro de una función del fichero de entrada del compilador no debería provocar que el compilador trate la línea siguiente como el principio de un fichero fuente.

Puede definir cómo recuperarse de un error de sintaxis escribiendo reglas para reconocer el token especial `error`. Este es un símbolo terminal que siempre se define (no necesita declararlo) y reservado para tratamiento de errores. El analizador de Bison genera un token `error` siempre que ocurra un error de sintaxis; si ha facilitado una regla que reconozca este token en el contexto actual, el analizador puede continuar.

Por ejemplo:

```
stmnts:  /* cadena vacía */
        | stmnts '\n'
        | stmnts exp '\n'
        | stmnts error '\n'
```

La cuarta regla en este ejemplo dice que un error seguido de una nueva línea forma una adición válida a cualquier `stmnts`.

¿Qué sucede si aparece un error de sintaxis en medio de una `exp`? La regla de recuperación de errores, interpretada estrictamente, se aplica a la secuencia precisa de una `stmnts`, un `error` y una nueva línea. Si aparece un error en medio de una `exp`, probablemente existan tokens adicionales y subexpresiones por leer antes de la nueva línea. De manera que la regla no es aplicable de la forma habitual.

Pero Bison puede forzar la situación para que se ajuste a la regla, descartando parte del contexto semántico y parte de la entrada. Primero descarta estados y objetos de la pila hasta que regrese a un estado en el que el token `error` sea aceptable. (Esto quiere decir que las subexpresiones ya analizadas son descartadas, retrocediendo hasta el último `stmnts` completo.) En este punto el token `error` puede desplazarse. Entonces, si el antiguo token de preanálisis no es aceptable para ser desplazado, el analizador lee tokens y los descarta hasta que encuentre un token que sea aceptable. En este ejemplo, Bison lee y descarta entrada hasta la siguiente línea de manera que la cuarta regla puede aplicarse.

La elección de reglas de error en la gramática es una elección de estrategias para la recuperación de errores. Una estrategia simple y útil es sencillamente saltarse el resto de la línea de entrada actual o de la sentencia actual si se detecta un error:

```
stmnt: error ';'  /* ante un error, saltar hasta que se lea ';' */
```

También es útil recuperar el delimitador de cierre que empareja con un un delimitador de apertura que ya ha sido analizado. De otra manera el delimitador de cierre probablemente aparecerá como sin emparejar, y generará otro, espúreo mensaje de error:

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

Las estrategias de recuperación de errores son por fuerza adivinanzas. Cuando estas adivinan mal, un error de sintaxis a menudo provoca otro. En el ejemplo anterior, la regla de recuperación de errores sospecha que el error es debido a una mala entrada en un `stmnt`. Suponga que en su lugar se inserta un punto y coma accidental en medio de un `stmt` válido. Después de recuperarse del primer error con la regla de recuperación de errores, se encontrará otro error de sintaxis directamente, ya que el texto que sigue al punto y coma accidental también es una `stmnt` inválida.

Para prevenir una cascada de mensajes de error, el analizador no mostrará mensajes de error para otro error de sintaxis que ocurra poco después del primero; solamente después de que se hayan desplazado con éxito tres tokens de entrada consecutivos se reanudarán los mensajes de error.

Note que las reglas que aceptan el token `error` podrían tener acciones, al igual que cualquiera de las otras reglas pueden tenerlas.

Puede hacer que los mensajes de error se reanuden inmediatamente usando la macro `yyerrok` en una acción. Si lo hace en la acción de la regla de error, no se suprimirá ningún mensaje de error. Esta macro no requiere ningún argumento; '`yyerrok;`' es una sentencia válida de C.

El token de preanálisis anterior se reanaliza inmediatamente después de un error. Si este no es aceptable, entonces la macro `yyclearin` podría utilizarse para limpiar ese token. Escriba la sentencia '`yyclearin;`' en la acción de la regla de error.

Por ejemplo, suponga que ante un error de análisis, se llama a una rutina de manejo de errores que avanza el flujo de entrada en algún punto donde el análisis prodría comenzar de nuevo. El próximo símbolo devuelto por el analizador léxico será probablemente correcto. El token de preanálisis anterior convendría que se descartara con '`yyclearin;`'.

La macro `YYRECOVERING` representa una expresión que tiene el valor 1 cuando el analizador se está recuperando de un error de sintaxis, y 0 durante el resto del tiempo. Un valor de 1 indica que actualmente los mensajes de error se están suprimiendo para nuevos errores de sintaxis.

7 Manejando Dependencias del Contexto

El paradigma de Bison es analizar tokens en primer lugar, entonces agruparlos en unidades sintácticas más grandes. En muchos lenguajes, el significado de un token se ve afectado por su contexto. A pesar de que esto viola el paradigma de Bison, ciertas técnicas (conocidas como *kludges*) podrían permitirle escribir analizadores de Bison para tales lenguajes.

(Realmente, "kludge" significa cualquier técnica que hace su trabajo pero no de una manera limpia o robusta.)

7.1 Información Semántica en Tipos de Tokens

El lenguaje C tiene una dependencia del contexto: la manera en la que se utiliza un identificador depende de cuál es su significado. Por ejemplo, considere esto:

```
foo (x);
```

Esto parece la sentencia de llamada a una función, pero si `foo` es un nombre de tipo definido, entonces esto realmente es una declaración de `x`. ¿Cómo puede un analizador de Bison para C decidirse cómo analizar esta entrada?

El método utilizado en GNU C es tener dos tipos diferentes de tokens, `IDENTIFIER` y `TYPENAME`. Cuando `yylex` encuentre un identificador, localiza la declaración actual del identificador para decidir que tipo de token devolver: `TYPENAME` si el identificador se declara como una definición de tipo, `IDENTIFIER` en otro caso.

Las reglas gramaticales pueden entonces expresar la dependencia del contexto eligiendo el tipo de token a reconocer. `IDENTIFIER` se acepta como una expresión, pero `TYPENAME` no lo es. `TYPENAME` puede empezar una declaración, pero `TYPENAME` no. En contextos donde el significado del identificador *no* es significativo, tal como en declaraciones que pueden ocultar nombre de definición de tipo, no se acepta ni `TYPENAME` o `IDENTIFIER`—hay una regla para cada uno de los dos tipos de tokens.

Esta técnica es sencilla de usar si la decisión de qué tipos de identificadores se permiten se hace en un lugar cercano a donde se analiza el identificador. Pero en C esto no es siempre así: C permite en una declaración redeclarar un nombre de tipo definido siempre que se haya especificado antes un tipo explícito:

```
typedef int foo, bar, lose;
static foo (bar);        /* redeclara bar como variable estática */
static int foo (lose);   /* redeclara foo como función */
```

Por desgracia, el nombre que se está declarando se encuentra separado de la construcción de la declaración por una estructura sintáctica complicada—el "declarador".

Como resultado, la parte del analizador de Bison para C necesita ser duplicada, con todos los nombres de los no terminales cambiados: una vez para el análisis de una declaración en la que se puede redefinir un nombre de declaración de tipo, y una vez para el análisis de una declaración en la que no puede hacerse. Aquí hay parte de la duplicación, con las acciones omitidas por brevedad:

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

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

Aquí `initdcl` puede redeclarar un nombre de definición de tipo, pero `notype_initdcl` no puede. La distinción entre `declarator` y `notype_declarator` es del mismo tipo.

Hay aquí alguna similitud entre esta técnica y una ligadura léxica (descrita a continuación), en que la información que altera el análisis léxico se cambia durante el análisis por otras partes del programa. La diferencia es que aquí la información es global, y se utiliza para otros propósitos en el programa. Una verdadera ligadura léxica tiene una bandera de propósito especial controlada por el contexto sintáctico.

7.2 Ligaduras Léxicas

Una manera de manejar las dependencias del contexto es la *ligadura léxica*: una bandera que se activa en acciones de Bison, cuyo propósito es alterar la manera en la que se analizan los tokens.

Por ejemplo, soponga que tenemos un lenguaje vagamente parecido a C, pero con una construcción especial '`hex (`*hex-expr*`)`'. Después de la palabra clave `hex` viene una expresión entre paréntesis en el que todos los enteros son hexadecimales. En particular, el token '`a1b`' debe tratarse como un entero en lugar de como un identificador si aparece en ese contexto. He aquí cómo puede hacerlo:

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

```
constant:
        INTEGER
        | STRING
        ;
```

Aquí asumimos que `yylex` mira el valor de `hexflag`; cuando no es cero, todos los enteros se analizan en hexadecimal, y los tokens que comiencen con letras se analizan como enteros si es posible.

La declaración de `hexflag` mostrada en la sección de declaraciones en C del archivo del analizador se necesita para hacerla accesible a las acciones (ver Seccion 3.1.1 [La Sección de Declaraciones en C], página 45). Debe también escribir el código en `yylex` para obedecer a la bandera.

7.3 Ligaduras Léxicas y Recuperación de Errores

Las ligaduras léxicas hacen demandas estrictas sobre cualquier regla de recuperación de errores que tenga. Ver Capitulo 6 [Recuperacion de Errores], página 81.

La razón de esto es que el propósito de una regla de recuperación de errores es abortar el análisis de una construcción y reanudar en una construcción mayor. Por ejemplo, en lenguajes como C, una regla típica de recuperación de errores es saltarse los tokens hasta el siguiente punto y coma, y entonces comenzar una sentencia nueva, como esta:

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

Si hay aquí un error de sintaxis en medio de una construcción '`hex (`*expr*`)`', esta regla de error se aplicará, y entonces la acción para la '`hex (`*expr*`)`' nunca se ejecutará. Así que `hexflag` continuaría activada durante el resto de la entrada, o hasta la próxima palabra clave `hex`, haciendo que los identificadores se malinterpreten como enteros.

Para evitar este problema la regla de recuperación de errores por sí misma desactiva `hexflag`.

Aquí podría existir también una regla de recuperación de errores que trabaje dentro de expresiones. Por ejemplo, podría haber una regla que se aplique dentro de los paréntesis y salte al paréntesis-cerrar:

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

Si esta regla actúa dentro de la construcción `hex`, no se va a abortar esa construcción (ya que ésta aparece a un nivel más interno de paréntesis dentro de la construcción). Por lo tanto, no debería desactivar la bandera: el resto de la construcción `hex` debería analizarse con la bandera aún activada.

¿Qué sucedería si hay una regla de recuperación de errores que pudiese abortar fuera la construcción `hex` o pudiese que no, dependiendo de las circunstancias? No hay manera de escribir la acción para determinar si una construcción `hex` está siendo abortada o no. De manera que si está utilizando una ligadura léxica, es mejor que esté seguro que sus reglas de recuperación de errores no son de este tipo. Cada regla debe ser tal que pueda estar seguro que siempre tendrá que tener que limpiar la bandera, o siempre no.

8 Depurando Su Analizador

Si una gramática de Bison compila adecuadamente pero no hace lo que quiere cuando se ejecuta, la propiedad de traza del analizador `yydebug` puede ayudarle para darse cuenta del por qué.

Para activar la compilación de las facilidades de traza, debe definir la macro `YYDEBUG` cuando compile el analizador. Podría utilizar '`-DYYDEBUG=1`' como una opción del compilador o podría poner '`#define YYDEBUG 1`' en la sección de declaraciones de C del archivo de la gramática (ver Seccion 3.1.1 [La Sección de Declaraciones en C], página 45). De forma alternativa, utilice la opción '`-t`' cuando ejecute Bison (ver Capitulo 9 [Invocando a Bison], página 89). Siempre definiremos `YYDEBUG` de manera que la depuración siempre es posible.

La facilidad de traza utiliza `stderr`, de manera que debe añadir `#include <stdio.h>` en la sección de declaraciones de C a menos que ya esté ahí.

Una vez que haya compilado el programa con las facilidades de traza, la manera de solicitar una traza es almacenar un valor distinto de cero en la variable `yydebug`. Puede hacer esto poniendo código C que lo haga (en `main`, tal vez), o puede alterar el valor con un depurador de C.

Cada paso tomado por el analizador cuando `yydebug` no es cero produce una línea o dos de información de traza, escrita sobre `stderr`. Los mensajes de traza le cuentan estas cosas:

- Cada vez que el analizador llama a `yylex`, qué tipo de token ha sido leido.
- Cada vez que un token se reduce, la profundidad y contenido completo de la pila de estados. (ver Seccion 5.5 [Estados del Analizador], página 74).
- Cada vez que se reduce una regla, qué regla es, y el contenido posterior completo de la pila de estados

Para hacerse una idea de esta información, ayuda el hacer referencia al listado del archivo producido por la opción '`-v`' de Bison (ver Capitulo 9 [Invocando a Bison], página 89). Este archivo muestra el significado de cada estado en términos de posición en varias reglas, y también qué hará cada estado con cada token de entrada posible. A medida que lea los mensajes de traza sucesivos, puede ver que el analizador está fucionando de acuerdo a su especificación en el achivo del listado. Finalmente llegará al lugar donde sucede algo indeseable, y verá qué parte de la gramática es la culpable.

El archivo del analizador es un programa en C y puede utilizar depuradores de C con éste, pero no es fácil interpretar lo que está haciendo. La función del analizador es un intérprete de una máquina de estado finito, y aparte de las acciones éste ejecuta el mismo código una y otra vez. Solamente los valores de las variables muestran sobre qué lugar de la gramática se está trabajando.

La información de depuración normalmente da el tipo de token de cada token leído, pero no su valor semántico. Puede opcionalmente definir una macro llamada `YYPRINT` para facilitar una manera de imprimir el valor. Si define `YYPRINT`, esta debería tomar tres argumentos. El analizador pasará un flujo de entrada/salida estandar, el valor numérico del tipo de token, y el valor del token (de `yylval`).

Aquí hay un ejemplo de `YYPRINT` apropiado para la calculadora multifunción (ver Seccion 2.4.1 [Declaraciones para `mfcalc`], página 38):

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

static void
yyprint (file, type, value)
     FILE *file;
     int type;
     YYSTYPE value;
{
  if (type == VAR)
    fprintf (file, " %s", value.tptr->name);
  else if (type == NUM)
    fprintf (file, " %d", value.val);
}
```

9 Invocando a Bison

La manera habitual de invocar a Bison es la siguiente:

```
bison fichero-entrada
```

Aquí *fichero-entrada* es el nombre del fichero de la gramática, que normalmente termina en '`.y`'. El nombre del archivo del analizador se construye reemplazando el '`.y`' con '`.tab.c`'. Así, el nombre de fichero '`bison foo.y`' produce '`foo.tab.c`', y el nombre de fichero '`bison hack/foo.y`' produce '`hack/foo.tab.c`'.

9.1 Opciones de Bison

Bison soporta las opciones tradicionales de una única letra y nombres de opción mnemónicos largos. Los nombres de opción largos se indican con '`--`' en lugar de '`-`'. Las abreviaciones para los nombres de opción se permiten siempre que sean únicas. Cuando una opción larga toma un argumento, como '`--file-prefix`', se conecta el nombre de la opcion con el argumento con '`=`'.

Aquí hay una lista de opciones que puede utilizar con Bison, alfabetizadas por la opción corta.

'`-b` *prefijo-fichero*'
'`--file-prefix=`*prefijo*'
: Especifica un prefijo a ser usado por todos los nombres de archivo de salida de Bison. Los nombres se eligen como si el fichero de entrada se llamase '*prefijo*`.c`'.

'`-d`'
'`--defines`'
: Escribe un archivo extra de salida conteniendo las definiciones de las macros para los nombres de tipo de tokens definidos en la gramática y el tipo de valor semántico `YYSTYPE`, además de unas cuantas declaraciones de variables `extern`.

  Si el archivo de salida del analizador se llama '*name*`.c`' entonces este archivo se llama '*name*`.h`'.

  Este archivo de salida es esencial si desea poner las definiciones de `yylex` en un archivo fuente por separado, porque `yylex` necesita ser capaz de hacer referencia a los códigos de tipo de token y las variables `yylval`. Ver Seccion 4.2.2 [Valores Semánticos de los Tokens], página 63.

'`-l`'
'`--no-lines`'
: No pone ningún comando `#line` del preprocesador en el fichero del analizador. Normalmente Bison los pone en el archivo del analizador de manera que el compilador de C y los depuradores asocien errores con su fichero fuente, el achivo de la gramática. Esta opción hace que asocien los errores con el archivo del analizador, tratándolo como un archivo fuente independiente por derecho propio.

'`-n`'
'`--no-parser`'
: No incluye ningún código C en el archivo del analizador; genera únicamente las tablas. El archivo del analizador contiene sólo directivas `#define` y declaraciones de variables estáticas.

Esta opción también le dice a Bison que escriba el código C para las acciones gramaticales en un archivo llamado '*nombrefichero*.act', en la forma de un cuerpo encerrado entre llaves para formar una sentencia `switch`.

'`-o` *fichero-salida*'
'`--output-file=`*fichero-salida*'
: Especifica el nombre *fichero-salida* para el archivo del analizador.

  El resto de los nombres de fichero de salida son construidos a partir de *fichero-salida* como se describió bajo las opciones '`-v`' y '`-d`'.

'`-p` *prefijo*'
'`--name-prefix=`*prefijo*'
: Renombra los símbolos externos utilizados en el analizador de manera que comiencen con *prefijo* en lugar de '`yy`'. La lista precisa de símbolos renombrados es `yyparse`, `yylex`, `yyerror`, `yynerrs`, `yylval`, `yychar` y `yydebug`.

  Por ejemplo, si utiliza '`-p c`', los nombres serán `cparse`, `clex`, etc.

  Ver Seccion 3.7 [Múltiples Analizadores en el Mismo Programa], página 60.

'`-r`'
'`--raw`'
: Hace que parezca que haya sido especificado `%raw`. Ver Seccion 3.6.8 [Sumario de Decls.], página 59.

'`-t`'
'`--debug`'
: Produce una definición de la macro `YYDEBUG` en el achivo del analizador, de manera que las facilidades de depuración sean compiladas. Ver Capitulo 8 [Depurando Su Analizador], página 87.

'`-v`'
'`--verbose`'
: Escribe un archivo de salida extra conteniendo descripciones amplias de los estados del analizador y qué se hace para cada tipo de token de preanálisis en ese estado.

  Este archivo también describe todos los conflictos, aquellos resueltos por la precedencia de operadores y los no resueltos.

  Este nombre de fichero se construye quitando '`.tab.c`' o '`.c`' del nombre de salida del analizador, y añadiendo '`.output`' en su lugar.

  Por lo tanto, si el archivo de entrada es '`foo.y`', entonces el archivo del analizador se llama '`foo.tab.c`' por defecto. Como consecuencia, el archivo de salida amplia se llama '`foo.output`'.

'`-V`'
'`--version`'
: Imprime el número de versión de Bison y termina.

'`-h`'
'`--help`'
: Imprime un sumario de las opciones de línea de comando de Bison y termina.

'`-y`'
'`--yacc`'
'`--fixed-output-files`'
: Equivalente a '`-o y.tab.c`'; el archivo de salida del analizador se llama '`y.tab.c`', y el resto de salida se llama '`y.output`' y '`y.tab.h`'. El propósito de esta opcion es la de imitar las convenciones de los nombres de fichero de Yacc. De este modo, el siguiente script de comandos puede ser sustituto de Yacc:

  ```
  bison -y $*
  ```

9.2 Clave Cruzada de Opciones

Aquí hay una lista de opciones, alfabetizada por la opcion larga, para ayudarle a encontrar la opción corta correspondiente.

```
--debug . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . -t
--defines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . -d
--file-prefix . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . -b
--fixed-output-files . . . . . . . . . . . . . . . . . . . . . . . . . . . -y
--help . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . -h
--name-prefix . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . -p
--no-lines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . -l
--no-parser . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . -n
--output-file . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . -o
--raw . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . -r
--token-table . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . -k
--verbose . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . -v
--version . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . -V
--yacc . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . -y
```

9.3 Invocando Bison bajo VMS

La sintaxis de la línea de comandos para Bison sobre VMS es una variante de la sintaxis del comando de Bison usual—adaptada para ajustarse a las convenciones de VMS.

Para encontrar el equivalente VMS de cualquier opción de Bison, comience con la opción larga, y sustituya con '/' el '--' inicial, y sustituya con '_' cada '-' en el nombre de opción largo. Por ejemplo, la siguiente invocación bajo VMS:

```
bison /debug/name_prefix=bar foo.y
```

es equivalente al siguiente comando bajo POSIX.

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

El sistema de archivos de VMS no permite nombre de ficheros tales como '`foo.tab.c`'. En el ejemplo anterior, el archivo de salida se llamaría '`foo_tab.c`'.

Apéndice A Símbolos de Bison

`error` Un nombre de token reservado para la recuperación de errores. Este token puede ser utilizado en reglas gramaticales para permitir al analizador de Bison reconocer un error en la gramática sin parar el proceso. En efecto, una sentencia conteniendo un error podría reconocerse como válida. Ante un error de análisis, el token `error` llega a ser el token de preanálisis actual. Las acciones correspondientes a `error` se ejecutan entonces, y el token de preanálisis se reestablace al token que originalmente provocó la violación. Ver Capitulo 6 [Recuperacion de Errores], página 81.

`YYABORT` Macro que pretende que haya ocurrido un error de sintaxis no recuperable haciendo que `yyparse` devuelva un 1 inmediatamente. La función de informe de errores `yyerror` no se llama. Ver Seccion 4.1 [La Función del Analizador `yyparse`], página 61.

`YYACCEPT` Macro que pretende que una expresión completa del lenguaje haya sido leida, haciendo que `yyparse` devuelva un 0 inmediatamente. Ver Seccion 4.1 [La Función del Analizador `yyparse`], página 61.

`YYBACKUP` Macro para descartar un valor de la pila del analizador y falsificar un token de preanálisis. Ver Seccion 4.4 [Propiedades Especiales para su Uso en Acciones], página 66.

`YYERROR` Macro que pretende que un error de sintaxis se haya acabado de detectar: llama a `yyerror` y entonces realiza una recuperación de errores normal si es posible (ver Capitulo 6 [Recuperacion de Errores], página 81), o (si la recuperación es imposible) hace que `yyparse` devuelva un 1. Ver Capitulo 6 [Recuperacion de Errores], página 81.

`YYERROR_VERBOSE`
Macro que usted define con `#define` en la sección de declaraciones de Bison para solicitar cadenas de mensajes de errores amplias, específicas cuando se llame a `yyerror`.

`YYINITDEPTH`
Macro para especificar el tamaño inicial de la pila del analizador. Ver Seccion 5.8 [Desbordamiento de Pila], página 78.

`YYLEX_PARAM`
Macro para especificar un argumento extra (o lista de argumentos extra) para que `yyparse` los pase a `yylex`. Ver Seccion 4.2.4 [Convenciones de Llamada para Analizadores Puros], página 64.

`YYLTYPE` Macro para el tipo de datos `yylloc`; una estructura con cuatro componentes. Ver Seccion 4.2.3 [Posiciones en el Texto de los Tokens], página 63.

`yyltype` Valor por defecto para YYLTYPE.

`YYMAXDEPTH`
Macro para especificar el tamaño máximo de la pila del analizador. Ver Seccion 5.8 [Desbordamiento de Pila], página 78.

`YYPARSE_PARAM`
Macro para especificar el nombre de un parámetro que `yyparse` debería aceptar. Ver Seccion 4.2.4 [Convenciones de Llamada para Analizadores Puros], página 64.

`YYRECOVERING`
Macro cuyo valor indica si el analizador se está recuperando de un error de sintaxis. Ver Seccion 4.4 [Propiedades Especiales para su Uso en Acciones], página 66.

`YYSTYPE` Macro para el tipo de datos de los valores semánticos; `int` por defecto. Ver Seccion 3.5.1 [Tipos de Datos de los Valores Semánticos], página 50.

`yychar` Variable entera externa que contiene el valor entero del token actual de preanálisis. (En un analizador puro, es una variable local dentro de `yyparse`.) Las acciones de las reglas de recuperación de errores podrían examinar esta variable. Ver Seccion 4.4 [Propiedades Especiales para su Uso en Acciones], página 66.

`yyclearin`
Macro utilizada en acciones de reglas de recuperación de errores. Esta borra el anterior token de preanálisis. Ver Capitulo 6 [Recuperacion de Errores], página 81.

`yydebug` Variable entera externa puesta a cero por defecto. Si se le da a `yydebug` un valor distinto de cero, el analizador sacará información a cerca de los símbolos de entrada y acciones del analizador. Ver Capitulo 8 [Depurando Su Analizador], página 87.

`yyerrok` Macro que provoca al analizador recuperar su modo normal inmediatamente después de un error de análisis. Ver Capitulo 6 [Recuperacion de Errores], página 81.

`yyerror` Función facilitada por el usuario para ser llamada por `yyparse` ante un error. La función recibe un argumento, un puntero a una cadena de caracteres conteniendo un mensaje de error. Ver Seccion 4.3 [La Función de Informe de Errores `yyerror`], página 65.

`yylex` Función del analizador léxico facilitada por el usuario, llamada sin argumentos para obtener el siguiente token. Ver Seccion 4.2 [La Función del Analizador Léxico `yylex`], página 61.

`yylval` Variable externa en la que `yylex` debería poner el valor semántico asociado con un token. (En un analizador puro, es una variable local dentro de `yyparse`, y su dirección se le pasa a `yylex`.) Ver Seccion 4.2.2 [Valores Semánticos de los Tokens], página 63.

`yylloc` Variable externa en la que `yylex` debería poner el número de línea y columna asociado a un token. (En un analizador puro, es una variable local dentro de `yyparse`, y su dirección se le pasa a `yylex`.) Puede ignorar esta variable si no utiliza la propiedad '`@`' en las acciones gramaticales. Ver Seccion 4.2.3 [Posiciones en el Texto de los Tokens], página 63.

`yynerrs` Variable global que Bison incrementa cada vez que hay un error de análisis. (En un analizador puro, es una variable local dentro de `yyparse`.) Ver Seccion 4.3 [La Función de Informe de Errores `yyerror`], página 65.

`yyparse` La función del analizador producida por Bison; llame a esta función para comenzar el análisis. Ver Seccion 4.1 [La Función del Analizador `yyparse`], página 61.

`%left` Declaración de Bison para asignar asociatividad por la izquierda a un(varios) token(s). Ver Seccion 3.6.2 [Precedencia de Operadores], página 56.

`%no_lines`
Declaración de Bison para evitar la generación de directivas `#line` en el fichero del analizador. Ver Seccion 3.6.8 [Sumario de Decls.], página 59.

`%nonassoc`
Declaración de Bison para asignar no-asociatividad a un(varios) token(s). Ver Seccion 3.6.2 [Precedencia de Operadores], página 56.

`%prec` Declaración de Bison para asignar precedencia a una regla específica. Ver Seccion 5.4 [Precedencia Dependiente del Contexto], página 73.

`%pure_parser`
Declaración de Bison para solicitar un analizador puro (reentrante). Ver Seccion 3.6.7 [Un Analizador Puro (Reentrante)], página 58.

`%raw` Declaración de Bison para usar los números de código de token internos a Bison en las tablas de tokens en lugar de los números de código de tokens usuales compatibles con Yacc. Ver Seccion 3.6.8 [Sumario de Decls.], página 59.

%right
: Declaración de Bison para asignar asociatividad por la derecha a un(varios) token(s). Ver Seccion 3.6.2 [Precedencia de Operadores], página 56.

%start
: Declaraciones de Bison para especificar el símbolo de arranque. Ver Seccion 3.6.6 [El Símbolo de Arranque], página 58.

%token
: Declaración de Bison para declarar un(varios) token(s) sin especificar la precedencia. Ver Seccion 3.6.1 [Nombres de Tipo de Token], página 55.

%token_table
: Declaración de Bison para incluir una tabla de nombres de tokens en el archivo del analizador. Ver Seccion 3.6.8 [Sumario de Decls.], página 59.

%type
: Declaración de Bison para declarar no-terminales. Ver Seccion 3.6.4 [Símbolos No Terminales], página 57.

%union
: Declaración de Bison para especificar varios tipos de datos posibles para los valores semánticos. Ver Seccion 3.6.3 [La Colección de Tipos de Valor], página 57.

Estos son los puntuadores y delimitadores utilizados en la entrada de Bison:

'%%'
: Delimitador utilizado para separar la sección de reglas gramaticales de la sección de declaraciones de Bison o la sección de código adicional en C. Ver Seccion 1.7 [El Formato Global de una Gramática de Bison], página 27.

'%{ %}'
: Todo el código listado entre '%{' y '%}' se copia directamente al archivo de salida sin ser interpretado. Este código forma la sección de "declaraciones en C" del archivo de entrada. Ver Seccion 3.1 [Resumen de una Gramática de Bison], página 45.

'/*...*/'
: Delimitadores de comentarios, como en C.

':'
: Separa el resultado de una regla de sus componentes. Ver Seccion 3.3 [Sintaxis de las Reglas Gramaticales], página 48.

';'
: Finaliza una regla. Ver Seccion 3.3 [Sintaxis de las Reglas Gramaticales], página 48.

'|'
: Separa reglas alternativas para el mismo no-terminal resultante. Ver Seccion 3.3 [Sintaxis de las Reglas Gramaticales], página 48.

Apéndice B Glosario

Agrupación
: Una construcción del lenguaje que es (en general) gramaticalmente divisible; por ejemplo, 'expresión' o 'declaración' en C. Ver Seccion 1.1 [Lenguajes y Gramáticas Independientes del Contexto], página 23.

Análisis de izquierda a derecha
: Análisis de una frase de un lenguaje analizándolo token a token de izquierda a derecha. Ver Capitulo 5 [El Algoritmo del Analizador de Bison], página 69.

Analizador léxico (scanner)
: Una función que lee un flujo de entrada y devuelve tokens uno por uno. Ver Seccion 4.2 [La Función del Analizador Léxico `yylex`], página 61.

Analizador sintáctico (parser)
: Una función que reconoce frases válidas de un lenguaje analizando la estructura sintáctica de un conjunto de tokens pasados desde un analizador léxico.

Asignación dinámica
: Asignación de memoria que ocurre durante la ejecución, en lugar de en tiempo de compilación, o a la entrada de una función.

Asociatividad por la izquierda
: Los operadores que tienen asociatividad por la izquierda se analizan de izquierda a derecha; '`a+b+c`' primero se computa '`a+b`' y entonces se combina con '`c`'. Ver Seccion 5.3 [Precedencia de Operadores], página 72.

Cadena vacía
: Análogo al conjunto vacío en la teoría de conjuntos, la cadena vacía es una cadena de caracteres de longitud cero.

Construcción del lenguaje
: Uno de los típicos esquemas de uso del lenguaje. Por ejemplo, una de las construcciones del lenguaje C es la sentencia `if`. Ver Seccion 1.1 [Lenguajes y Gramáticas Independientes del Contexto], página 23.

Desplazamiento
: Un analizador se dice que desplaza cuando realiza la elección de analizar la entrada que proviene del flujo en lugar de reducir inmediatamente alguna regla ya reconocida. Ver Capitulo 5 [El Algoritmo del Analizador de Bison], página 69.

Error de análisis
: Un error encontrado durante el análisis de un flujo de entrada debido a una sintaxis no válida. Ver Capitulo 6 [Recuperacion de Errores], página 81.

Flujo de entrada
: Un flujo de datos continuo entre dispositivos y programas.

Forma de Backus-Naur (BNF)
: Método formal para la especificación de gramáticas independientes del contexto. La BNF se utilizó en primer lugar en el informe de *ALGOL-60*, 1963. Ver Seccion 1.1 [Lenguajes y Gramáticas Independientes del Contexto], página 23.

Gramáticas independientes del contexto
: Gramáticas especificadas como reglas que pueden aplicarse sin considerar el contexto. Por lo tanto, si hay una regla que dice que un entero se puede utilizar como una expresión, los enteros se permiten *en cualquier lugar* donde una expresión se permita. Ver Seccion 1.1 [Lenguajes y Gramáticas Independientes del Contexto], página 23.

LALR(1)
: La clase de gramáticas independientes del contexto que Bison (como la mayoría de los otros generadores de analizadores sintácticos) pueden manejar; un subconjunto de las gramáticas LR(1). Ver Seccion 5.7 [Misteriosos Conflictos Reducción/Reducción], página 77.

Ligadura léxica
: Una bandera, activada por las acciones en las reglas gramaticales, que alteran la manera en la que se analizan los tokens. Ver Seccion 7.2 [Ligaduras Lexicas], página 84.

Literal de caracter simple
: Un caracter sencillo que se reconoce e interpreta como es. Ver Seccion 1.2 [De las Reglas Formales a la Entrada de Bison], página 24.

LR(1)
: La clase de gramáticas independientes del contexto en la que al menos se necesita un token de preanálisis para eliminar la ambigüedad del análisis de cualquier parte de la entrada.

Máquina de estado finito basada en pila
: Una "máquina" que tiene estados discretos los cuales se dice que existen en cada instante de tiempo. A medida que la máquina procesa la entrada, la máquina se mueve de estado a estado como se especifica en la lógica de la máquina. En el caso de un analizador sintáctico, la entrada es el lenguaje que está siendo analizado, y los estados corresponden a varias etapas en las reglas de la gramática. Ver Capitulo 5 [El Algoritmo del Analizador de Bison], página 69.

Notación polaca inversa
: Un lenguaje en el que todos los operadores son operadores postfijos.

Operador infijo
: Un operador aritmético que se situa entre los operandos sobre los que realiza alguna operación.

Operador postfijo
: Un operador aritmético que se coloca después de los operandos sobre los que realiza alguna operación.

Recursión por la derecha
: Una regla cuyo símbolo resultante es también su componente simbólica final; por ejemplo, '`expseq1: exp ',' expseq1;`'. Ver Seccion 3.4 [Reglas Recursivas], página 49.

Recursión por la izquierda
: Una regla cuyo símbolo resultante es también su primer símbolo componente; por ejemplo, '`expseq1 : expseq1 ',' exp;`'.Ver Seccion 3.4 [Reglas Recursivas], página 49.

Reducción
: Reemplazo de una cadena de no-terminales y/o terminales con un no-terminal simple, de acuerdo a una regla gramatical. Ver Capitulo 5 [El Algoritmo del Analizador de Bison], página 69.

Reentrante
: Un subprograma reentrante es un subprograma que puede ser invocado cualquier número de veces en paralelo, sin interferir entre las distintas invocaciones. Ver Seccion 3.6.7 [Un Analizador Puro (Reentrante)], página 58.

Semántica
: En los lenguajes de ordenador, la semántica se especifica con las acciones tomadas para cada instancia del lenguaje, es decir, el significado de cada sentencia. Ver Seccion 3.5 [Definiendo la Semántica del Lenguaje], página 50.

Símbolo de arranque
: El símbolo no terminal que representa una expresión completa del lenguaje que se está analizando. El símbolo de arranque normalmente se presenta como el primer símbolo no terminal en la especificación del lenguaje. Ver Seccion 3.6.6 [El Símbolo de Arranque], página 58.

Símbolo no terminal
: Un símbolo de la gramática que representa una contrucción gramatical que puede expresarse mediante reglas en términos de construcciones más pequeñas; en otras palabras, una construcción que no es un token. Ver Seccion 3.2 [Simbolos], página 46.

Símbolo terminal
: Un símbolo de la gramática que no tiene reglas en la gramática y por lo tanto es gramaticalmente indivisible. El trozo de texto que representa es un token. Ver Seccion 1.1 [Lenguajes y Gramáticas Independientes del Contexto], página 23.

Tabla de símbolos
: Una estructura de datos donde los nombres de los símbolos y datos relacionados se almacenan durante el análisis para permitir el reconocimiento y uso de información existente en usos repetidos del un símbolo. Ver Seccion 2.4 [Calc Multi-funcion], página 37.

Token
: Una unidad básica, gramaticalmente indivisible de un lenguaje. El símbolo que describe un token en la gramática es un símbolo terminal. La entrada del analizador de Bison es un flujo de tokens que proviene del analizador léxico. Ver Seccion 3.2 [Simbolos], página 46.

Token de cadena literal
: Un token que consiste de dos o más caracteres fijos. Ver Seccion 3.2 [Simbolos], página 46.

Token de preanálisis
: Un token que ya ha sido leído pero aún no ha sido desplazado. Ver Seccion 5.1 [Tokens de Preanálisis], página 69.

Índice

U

V

Y

Tabla de Contenido