bisonc++

bisonc++.4.05.00.tar.gz

2005-2013

bisonc++(1)

bisonc++.4.05.00.tar.gz bisonc++ parser generator

2005-2013

NAME

bisonc++ - Generate a C++ parser class and parsing function

SYNOPSIS

bisonc++ [OPTIONS] *grammar-file*

SECTIONS

This manual page contains the following sections:

Looking for a specific section? Search for its number + a dot.

1. DESCRIPTION

Bisonc++ derives from previous work on **bison** by Alain Coetmeur (coetmeur@icdc.fr), who created in the early '90s a **C++** class encapsulating the *yyparse* function as generated by the GNU-**bison** parser

generator.

Initial versions of **bisonc++** (up to version 0.92) wrapped Alain's program in a program offering a more modern user-interface, removing all old-style (**C**) *%define* directives from **bison++**'s input specification file (see below for an in-depth discussion of the differences between **bison++** and **bisonc++**). Starting with version 0.98, **bisonc++** represents a complete rebuilt of the parser generator, closely following descriptions given in Aho, Sethi and Ullman's *Dragon Book*. Since version 0.98 **bisonc++** is a **C++** program, rather than a **C** program generating **C++** code.

Bisonc++ expands the concepts initially implemented in **bison** and **bison++**, offering a cleaner setup of the generated parser class. The parser class is derived from a base-class, mainly containing the parser's token- and type-definitions as well as several member functions which should not be modified by the programmer.

Most of these base-class members might also be defined directly in the parser class, but were defined in the parser's base-class. This design results in a very lean parser class, declaring only members that are actually defined by the programmer or that have to be defined by **bisonc++** itself (e.g., the member function *parse* as well as some support functions requiring access to facilities that are only available in the parser class itself, rather than in the parser's base class).

This design does not require any virtual members: the members which are not involved in the actual parsing process may always be (re)implemented directly by the programmer. Thus there is no need to apply or define virtual member functions.

In fact, there are only two public members in the parser class generated by **bisonc++**: *setDebug* (see below) and *parse*. Remaining members are private, and those that can be redefined by the programmer using **bisonc++** usually receive initial, very simple default in-line implementations. The (partial) exception to this rule is the member function *lex*, producing the next lexical token. For *lex* either a standardized interface or a mere declaration is offered (requiring the programmer to provide his/her own *lex* implementation).

To enforce a primitive namespace, **bison** used a well-known naming-convention: all its public symbols started with *yy* or *YY*. **Bison++** followed **bison** in this respect, even though a class by itself offers enough protection of its identifiers. Consequently, these *yy* and *YY* conventions are now outdated, and **bisonc++** does not generate or use symbols defined in either the parser (base) class or in its member functions starting with *yy* or *YY*. Instead, following a suggestion by Lakos (2001), all data members start with *d_*, and all static data members start with *s_*. This convention was not introduced to enforce identifier protection, but to clarify the storage type of variables. Other (local) symbols lack specific prefixes. Furthermore, **bisonc++** allows its users to define the parser class in a particular namespace of their own choice.

Bisonc++ should be used as follows:

- As usual, a grammar must be defined. With **bisonc++** this is not different, and the reader is referred to **bisonc++**'s manual and other sources (like Aho, Sethi and Ullman's book) for details about how to specify and decorate grammars.
- The number and function of the various *%define* declarations as used by **bison++**, however, is greatly modified. Actually, all of **bison**'s *%define* declarations were replaced by their (former) first arguments. Furthermore, `macro-style' declarations are no longer supported or required. Finally, all directives use lower-case characters only and do not contain underscore characters (but

sometimes hyphens). E.g., *%define DEBUG* is now declared as *%debug*; *%define LSP_NEEDED* is now declared as *%lsp-needed* (note the hyphen).

- As noted, no `macro style' *%define* declarations are required anymore. Instead, the normal practice of defining class members in source files and declaring them in a class header files can be adhered to using **bisonc++**. Basically, **bisonc++** concentrates on its main tasks: defining a parser class and implementing its parsing function *int parse*, leaving all other parts of the parser class' definition to the programmer.
- Having specified the grammar and (usually) some directives **bisonc++** is able to generate files defining the parser class and to implement the member function *parse* and its support functions. See the next section for details about the various files that may be generated by **bisonc++**.
- All members (except for the member *parse* and its support functions) must be implemented by the programmer. Additional member functions should be declared in the parser class' header. At the very least the member *int lex() must* be implemented (although a standard implementation can be generated by **bisonc++**). The member *lex* is called by *parse* to obtain the next available token. The member function *void error(char const *msg)* may also be re-implemented by the programmer, and a basic in-line implementation is provided by default. The member function *error* is called when *parse* detects (syntactic) errors.
- The parser can now be used in a program. A very simple example would be:

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

2. GENERATED FILES

Bisonc++ may create the following files:

- A file containing the implementation of the member function *parse* and its support functions. The member *parse* is a public member that can be called to parse a token-sequence according to a specified LALR1 type of grammar. By default the implementations of these members are written on the file *parse.cc*. The programmer should not modify the contents of this file; it is rewritten every time **bisonc++** is called.
- A file containing an initial setup of the parser class, containing the declaration of the public member *parse* and of its (private) support members. New members may safely be declared in the parser class, as it is only created by **bisonc++** if not yet existing, using the filename *<parser-class>.h* (where *<parser-class>* is the the name of the defined parser class).
- A file containing the parser class' *base class*. This base class should not be modified by the programmer. It contains types defined by **bisonc++**, as well as several (protected) data members and member functions, which should not be redefined by the programmer. All symbolic parser terminal tokens are defined in this class, thereby escalating these definitions to a separate class (cf. Lakos, (2001)), which in turn prevents circular dependencies between the lexical scanner and the parser (here, circular dependencies may easily be encountered, as the parser needs access to the lexical scanner class when defining the lexical scanner as one of its data members, whereas the lexical scanner needs access to the parser class to know about the grammar's symbolic terminal tokens; escalation is a way out of such circular dependencies). By default this file is (re)written any time **bisonc++** is called, using the filename *<parser-class>base.h*.

- A file containing an *implementation header*. The implementation header rather than the parser's class header file should be included by the parser's source files implementing member functions declared by the programmer. The implementation header first includes the parser class's header file, and then provides default in-line implementations for its members *error* and *print* (which may be altered by the programmer). The member *lex* may also receive a standard in-line implementation. Alternatively, its implementation can be provided by the programmer (see below). Any directives and/or namespace directives required for the proper compilation of the parser's additional member functions should be declared next. The implementation header is included by the file defining *parse*. By default the implementation header is created if not yet existing, receiving the filename <*parser-class*>*.ih*.
- A verbose description of the generated parser. This file is comparable to the verbose ouput file originally generated by **bison++**. It is generated when the option *--verbose* or *-V* is provided. If so, **bisonc++** writes the file <*grammar*>*.output*, where <*grammar*> is the name of the file containing the grammar definition.

3. OPTIONS

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

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

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

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

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

- **--baseclass-preinclude**=*pathname* (**-H**)
  Pathname defines the path to the file preincluded in the parser's base-class header. This option is needed in situations where the base class header file refers to types which might not yet be known. E.g., with polymorphic semantic values a *std::string* value type might be used. Since the *string* header file is not by default included in *parserbase.h* we somehow need to inform the compiler about this and possibly other headers. The suggested procedure is to use a pre-include header file declaring the required types. By default `*header*' is surrounded by double quotes:

#include "header" is used when the option *-H header* is specified. When the argument is surrounded by pointed brackets *#include <header>* is included. In the latter case, quotes might be required to escape interpretation by the shell (e.g., using *-H '<header>'*).

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

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

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

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

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

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

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

- **--debug**
  Provide *parse* and its support functions with debugging code, showing the actual parsing process on the standard output stream. When included, the debugging output is active by default, but its activity may be controlled using the *setDebug(bool on-off)* member. An *#ifdef DEBUG* macro is not supported by **bisonc++**. Rerun **bisonc++** without the *--debug* option to remove the debugging code.

- **--error-verbose**
  When a syntactic error is reported, the generated parse function dumps the parser's state stack to the standard output stream. The stack dump shows on separate lines a stack index followed by the state stored at the indicated stack element. The first stack element is the stack's top element.

- **--filenames**=*filename* (**-f**)
  Filename is a generic file name that is used for all header files generated by **bisonc++**. Options defining specific file names are also available (which then, in turn, overrule the name specified by this option).

- **--flex**
  Bisonc++ generates code calling *d_scanner.yylex()* to obtain the next lexical token, and calling *d_scanner.YYText()* for the matched text, unless overruled by options or directives explicitly defining these functions. By default, the interface defined by **flexc++**(1) is used. This option is only interpreted if the *--scanner* option or *%scanner* directive is also used.

- **--help** (**-h**)
  Write basic usage information to the standard output stream and terminate.

- **--implementation-header**=*filename* (**-i**)
  Filename defines the name of the file to contain the implementation header. It defaults to the name of the generated parser class plus the suffix *.ih*.

  The implementation header should contain all directives and declarations *only* used by the implementations of the parser's member functions. It is the only header file that is included by the source file containing *parse*'s implementation. User defined implementation of other class members may use the same convention, thus concentrating all directives and declarations that are required for the compilation of other source files belonging to the parser class in one header file.

- **--implementation-skeleton**=*pathname* (**-I**)
  Pathname defines the path name to the file containing the skeleton of the implementation header. t defaults to the installation-defined default path name (e.g., */usr/share/bisonc++/* plus *bisonc++.ih*).

- **--insert-stype**
  This option is only effective if the *debug* option (or *%debug* directive) has also been specified. When *insert-stype* has been specified the parsing function's debug output also shows selected semantic values. It should only be used if objects or variables of the semantic value type *STYPE__* can be inserted into *ostreams*.

- **--max-inclusion-depth**=*value*
  Set the maximum number of nested grammar files. Defaults to 10.

- **--namespace** *identifier*
  Define all of the code generated by **bisonc++** in the name space *identifier*. By default no name space is defined. If this options is used the implementation header is provided with a commented out *using namespace* declaration for the specified name space. In addition, the parser and parser base class header files also use the specified namespace to define their include guard directives.

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

- **--no-baseclass-header**
  Do not write the file containing the parser class' base class, even if that file doesn't yet exist. By default the file containing the parser's base class is (re)written each time **bisonc++** is called. Note that this option should normally be avoided, as the base class defines the symbolic terminal tokens that are returned by the lexical scanner. By suppressing the construction of this file any modification in these terminal tokens will not be communicated to the lexical scanner.

- **--no-lines**
  Do not put *#line* preprocessor directives in the file containing the parser's *parse* function. By

default the file containing the parser's *parse* function also contains *#line* preprocessor directives. This option allows the compiler and debuggers to associate errors with lines in your grammar specification file, rather than with the source file containing the *parse* function itself.

- **--no-parse-member**
  Do not write the file containing the parser's predefined parser member functions, even if that file doesn't yet exist. By default the file containing the parser's *parse* member function is (re)written each time **bisonc++** is called. Note that this option should normally be avoided, as this file contains parsing tables which are altered whenever the grammar definition is modified.

- **--own-debug**
  Extensively displays the actions performed by **bisonc++**'s parser when it processes the grammar specification **s**. This implies the *--verbose* option.

- **--own-tokens** (**-T**)
  The tokens returned as well as the text matched when **bisonc++** reads its input files(s) are shown when this option is used.

  This option does *not* result in the generated parsing function displaying returned tokens and matched text. If that is what you want, use the *--print-tokens* option.

- **--parsefun-skeleton**=*pathname* (**-P**)
  Pathname defines the path name of the file containing the parsing member function's skeleton. It defaults to the installation-defined default path name (e.g., */usr/share/bisonc++/* plus *bisonc++.cc*).

- **--parsefun-source**=*filename* (**-p**)
  Filename defines the name of the source file to contain the parser member function *parse*. Defaults to *parse.cc*.

- **--polymorphic-skeleton**=*pathame* (**-M**)
  Pathname defines the path name of the file containing the skeleton of the polymorphic template classes. It defaults to the installation-defined default path name (e.g., */usr/share/bisonc++/* plus *bisonc++polymorphic*).

- **--polymorphic-inline-skeleton**=*pathname* (**-m**)
  Pathname defines the path name of the file containing the skeleton of the inline implementations of the members of the polymorphic template classes. It defaults to the installation-defined default path name (e.g., */usr/share/bisonc++/* plus *bisonc++polymorphic*).

- **--print-tokens** (**-t**)
  The generated parsing function implements a function *print__* displaying (on the standard output stream) the tokens returned by the parser's scanner as well as the corresponding matched text. This implementation is suppressed when the parsing function is generated without using this option. The member *print__*) is called from *Parser::print*, which is defined in-line in the the parser's class header. Calling *Parser::print__* can thus easily be controlled from *print*, using, e.g., a variable that set by the program using the parser generated by **bisonc++**.

  This option does *not* show the tokens returned and text matched by **bisonc++** itself when it is reading its input **s**. If that is what you want, use the *--own-tokens* option.

- **--required-tokens**=*number*
  Following a syntactic error, require at least *number* successfully processed tokens before another syntactic error can be reported. By default *number* is zero.

- **--scanner**=*pathname* (**-s**)
  Pathname defines the path name to the file defining the scanner's class interface (e.g., *"../scanner/scanner.h"*). When this option is used the parser's member *int lex()* is predefined as

```
int Parser::lex()
{
    return d_scanner.lex();
}
```

  and an object *Scanner d_scanner* is composed into the parser (but see also option *scanner-class-name*). The example shows the function that's called by default. When the *--flex* option (or *%flex* directive) is specified the function *d_scanner.yylex()* is called. Any other function to call can be specified using the *--scanner-token-function* option (or *%scanner-token-function* directive).

  By default **bisonc++** surrounds *pathname* by double quotes (using, e.g., *#include "pathname"*). When *pathname* is surrounded by pointed brackets *#include <pathname>* is included.

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

- **--scanner-class-name** *scannerClassName*
  Defines the name of the scanner class, declared by the *pathname* header file that is specified at the *scanner* option or directive. By default the class name *Scanner* is used.

  A warning is issued if this option is used and either the *scanner* option was not provided, or the parser class interface in an already existing parser class header file does not declare a scanner class *d_scanner* object.

- **--scanner-debug**
  Show de scanner's matched rules and returned tokens. This offers an extensive display of the rules and tokens matched and returned by **bisonc++**'s scanner, not of just the tokens and matched text received by **bisonc++**. If that is what you want use the *--own-tokens* option.

- **--scanner-matched-text-function**=*function-call*
  The scanner function returning the text that was matched at the last call of the scanner's token function. A complete function call expression should be provided (including a scanner object, if used). This option overrules the *d_scanner.matched()* call used by default when the *%scanner* directive is specified, and it overrules the *d_scanner.YYText()* call used when the *%flex* directive is provided. Example:

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

- **--scanner-token-function**=*function-call*
  The scanner function returning the next token, called from the parser's *lex* function. A complete function call expression should be provided (including a scanner object, if used). This option

overrules the *d_scanner.lex()* call used by default when the *%scanner* directive is specified, and it overrules the *d_scanner.yylex()* call used when the *%flex* directive is provided. Example:

```
--scanner-token-function "myScanner.nextToken()"
```

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

- **--show-filenames**
  Writes the names of the generated files to the standard error stream.

- **--skeleton-directory**=*directory* (**-S**)
  Specifies the directory containing the skeleton files. This option can be overridden by the specific skeleton-specifying options (*-B -C*, *-H*, *-I*, *-M* and *-m*).

- **--target-directory**=*pathname*
  Pathname defines the directory where generated files should be written. By default this is the directory where **bisonc++** is called.

- **--thread-safe**
  No static data are modified, making **bisonc++** thread-safe.

- **--usage**
  Write basic usage information to the standard output stream and terminate.

- **--verbose** (**-V**)
  Write a file containing verbose descriptions of the parser states and what is done for each type of look-ahead token in that state. This file also describes all conflicts detected in the grammar, both those resolved by operator precedence and those that remain unresolved. It is not created by default, but if requested the information is written on <*grammar*>.*output*, where <*grammar*> is the grammar specification file passed to **bisonc++**.

- **--version** (**-v**)
  Display **bisonc++**'s version number and terminate.

4. DIRECTIVES

The following directives can be specified in the initial section of the grammar specification file. When command-line options for directives exist, they overrule the corresponding directives given in the grammar specification file. Directives affecting the class header or implementation header file are ignored if these files already exist.

Directives accepting a `filename' do not accept path names, i.e., they cannot contain directory separators (*/*); directives accepting a 'pathname' may contain directory separators. A 'pathname' using blank characters should be surrounded by double quotes.

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

editing may then be required to effectuate the directive.

- **%baseclass-header** *filename*
  Filename defines the name of the file to contain the parser's base class. This class defines, e.g., the parser's symbolic tokens. Defaults to the name of the parser class plus the suffix *base.h*. This directive is overruled by the **--baseclass-header** (**-b**) command-line option.

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

- **%baseclass-preinclude** *pathname*
  Pathname defines the path to the file preincluded by the parser's base-class header. See the description of the *--baseclass-preinclude* option for details about this directive. By default, **bisonc++** surrounds *header* by double quotes. However, when *header* itself is surrounded by pointed brackets *#include <header>* is included.
- **%class-header** *filename*
  Filename defines the name of the file to contain the parser class. Defaults to the name of the parser class plus the suffix *.h* This directive is overruled by the **--class-header** (**-c**) command-line option.

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

- **%class-name** *parser-class-name*
  Declares the name of the parser class. It defines the name of the **C++** class that is generated. If no *%class-name* is specified the default class name *Parser* is used.

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

- **%debug**
  Provide *parse* and its support functions with debugging code, showing the actual parsing process on the standard output stream. When included, the debugging output is active by default, but its activity may be controlled using the *setDebug(bool on-off)* member. No *#ifdef DEBUG* macros are used anymore. To remove existing debugging code re-run **bisonc++** without the *--debug* option or *%debug* declaration.
- **%error-verbose**
  This directive can be specified to dump the parser's state stack to the standard output stream when the parser encounters a syntactic error. The stack dump shows on separate lines a stack index followed by the state stored at the indicated stack element. The first stack element is the stack's top element.
- **%expect** *number*
  This directive specifies the exact number of shift/reduce and reduce/reduce conflicts for which no warnings are to be generated. Details of the conflicts are reported in the verbose output file (e.g., *grammar.output*). If the number of actually encountered conflicts deviates from *`number'*, then this directive is ignored.
- **%filenames** *filename*
  Filename is a generic filename that is used for all header files generated by **bisonc++**. Options defining specific filenames are also available (which then, in turn, overrule the name specified by

this directive). This directive is overruled by the **--filenames** (**-f**) command-line option.

- **%flex**
  When provided, the scanner member returning the matched text is called as *d_scanner.YYText()*, and the scanner member returning the next lexical token is called as *d_scanner.yylex()*. This directive is only interpreted if the *%scanner* directive is also provided.
- **%implementation-header** *filename*
  Filename defines the name of the file to contain the implementation header. It defaults to the name of the generated parser class plus the suffix *.ih*.
  The implementation header should contain all directives and declarations that are *only* used by the parser's member functions. It is the only header file that is included by the source file containing *parse*'s implementation. User defined implementation of other class members may use the same convention, thus concentrating all directives and declarations that are required for the compilation of other source files belonging to the parser class in one header file.
  This directive is overruled by the **--implementation-header** (**-i**) command-line option.
- **%include** *pathname*
  This directive is used to switch to *pathname* while processing a grammar specification. Unless *pathname* defines an absolute file-path, *pathname* is searched relative to the location of **bisonc++**'s main grammar specification file (i.e., the grammar file that was specified as **bisonc++**'s command-line option). This directive can be used to split long grammar specification files in shorter, meaningful units. After processing *pathname* processing continues beyond the *%include pathname* directive.
- **%left** *terminal ...*
  Defines the names of symbolic terminal tokens that must be treated as left-associative. I.e., in case of a shift/reduce conflict, a reduction is preferred over a shift. Sequences of *%left*, *%nonassoc*, *%right* and *%token* directives may be used to define the precedence of operators. In expressions, the first used directive defines the tokens having the lowest precedence, the last used defines the tokens having the highest priority. See also *%token* below.
- **%locationstruct** *struct-definition*
  Defines the organization of the location-struct data type *LTYPE__*. This struct should be specified analogously to the way the parser's stacktype is defined using *%union* (see below). The location struct is named *LTYPE__*. By default (if neither *locationstruct* nor *LTYPE__* is specified) the standard location struct (see the next directive) is used:
- **%lsp-needed**
  This directive results in **bisonc++** generating a parser using the standard location stack. This stack's default type is:

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

Bisonc++ does *not* provide the elements of the *LTYPE__* struct with values. Action blocks of production rules may refer to the location stack element associated with a production element using @ variables, like *@1.timestamp*, *@3.text*, *@5*. The rule's location struct itself may be referred to as either *d_loc__* or *@@*.

- **%ltype typename**
  Specifies a user-defined token location type. If *%ltype* is used, *typename* should be the name of an alternate (predefined) type (e.g., *size_t*). It should not be used if a *%locationstruct* specification is defined (see below). Within the parser class, this type is available as the type `*LTYPE__*'. All text on the line following *%ltype* is used for the *typename* specification. It should therefore not contain comment or any other characters that are not part of the actual type definition.
- **%namespace** *namespace*
  Define all of the code generated by **bisonc++** in the name space *namespace*. By default no name space is defined. If this directive is used the implementation header is provided with a commented out *using namespace* declaration for the specified name space. In addition, the parser and parser base class header files also use the specified namespace to define their include guard directives.

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

- **%negative-dollar-indices**
  Do not generate warnings when zero- or negative dollar-indices are used in the grammar's action blocks. Zero or negative dollar-indices are commonly used to implement inherited attributes, and should normally be avoided. When used, they can be specified like *$-1*, or like *$<type>-1*, where *type* is empty; an *STYPE__* tag; or a field-name. However, note that in combination with the *%polymorphic* directive (see below) only the *$-i* format can be used.
- **%no-lines**
  By default *#line* preprocessor directives are inserted just before action statements in the file containing the parser's *parse* function. These directives are suppressed by the *%no-lines* directive.
- **%nonassoc** *terminal ...*
  Defines the names of symbolic terminal tokens that should be treated as non-associative. I.e., in case of a shift/reduce conflict, a reduction is preferred over a shift. Sequences of *%left*, *%nonassoc*, *%right* and *%token* directives may be used to define the precedence of operators. In expressions, the first used directive defines the tokens having the lowest precedence, the last used defines the tokens having the highest priority. See also *%token* below.
- **%parsefun-source** *filename*
  Filename defines the name of the file to contain the parser member function *parse*. Defaults to *parse.cc*. This directive is overruled by the **--parse-source** (**-p**) command-line option.
- **%polymorphic** *polymorphic-specification(s)*
  Bison's traditional way of handling multiple semantic values is to use a *%union* specification (see below). Although *%union* is supported by **bisonc++**, a polymorphic semantic value class is preferred due to its improved type safety.

  The *%polymorphic* directive defines a polymorphic semantic value class and can be used instead of a *%union* specification. Refer to section **POLYMORPHIC SEMANTIC VALUES** below or to **bisonc++**'s user manual for a detailed description of the specification, characteristics, and use of polymorphic semantic values.

- **%prec** *token*
  Overrules the defined precendence of an operator for a particular grammatical rule. A well known application of *%prec* is:

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

```
        ...
    }
```

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

```
%right UMINUS
```

(see below) following, e.g., the '*' and '/' operators.

- **%print-tokens**
  The *print* directive provides an implementation of the Parser class's *print__* function displaying the current token value and the text matched by the lexical scanner as received by the generated *parse* function.
- **%required-tokens** *number*
  Following a syntactic error, require at least *number* successfully processed tokens before another syntactic error can be reported. By default *number* is zero.
- **%right** *terminal ...*
  Defines the names of symbolic terminal tokens that should be treated as right-associative. I.e., in case of a shift/reduce conflict, a shift is preferred over a reduction. Sequences of *%left*, *%nonassoc*, *%right* and *%token* directives may be used to define the precedence of operators. In expressions, the first used directive defines the tokens having the lowest precedence, the last used defines the tokens having the highest priority. See also *%token* below.
- **%scanner** *pathname*
  Use *pathname* as the path name to the file pre-included in the parser's class header. See the description of the *--scanner* option for details about this directive. Similar to the convention adopted for this argument, *pathname* by default is surrounded by double quotes. However, when the argument is surrounded by pointed brackets *#include <pathname>* is included. This directive results in the definition of a composed *Scanner d_scanner* data member into the generated parser, and in the definition of a *int lex()* member, returning *d_scanner.lex()*.

  By specifying the *%flex* directive the function *d_scanner.yylex()* is called. Any other function to call can be specified using the *--scanner-token-function* option (or *%scanner-token-function* directive).

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

- **%scanner-class-name** *scannerClassName*
  Defines the name of the scanner class, declared by the *pathname* header file that is specified at the *scanner* option or directive. By default the class name *Scanner* is used.

  A warning is issued if this directive is used and either the *scanner* directive was not provided, or the parser class interface in an already existing parser class header file does not declare a scanner class *d_scanner* object.

- **%scanner-matched-text-function** *function-call*
  The scanner function returning the text that was matched by the lexical scanner after its token function (see below) has returned. A complete function call expression should be provided (including a scanner object, if used). Example:

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

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

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

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

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

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

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

- **%start** *non-terminal*
  The non-terminal *non-terminal* should be used as the grammar's start-symbol. If omitted, the first grammatical rule is used as the grammar's starting rule. All syntactically correct sentences must be derivable from this starting rule.
- **%stype typename**
  The type of the semantic value of non-terminal tokens. By default it is *int*. *%stype*, *%union*, and *%polymorphic* are mutually exclusive directives.

  Within the parser class, the semantic value type is available as the type `*STYPE__*'. All text on the line following *%stype* is used for the *typename* specification. It should therefore not contain comment or any other characters that are not part of the actual type definition.

- **%target-directory pathname**
  Pathname defines the directory where generated files should be written. By default this is the directory where **bisonc++** is called. This directive is overruled by the *--target-directory* command-line option.
- **%token** *terminal ...*
  Defines the names of symbolic terminal tokens. Sequences of *%left*, *%nonassoc*, *%right* and *%token* directives may be used to define the precedence of operators. In expressions, the first used directive defines the tokens having the lowest precedence, the last used defines the tokens having the highest priority. See also *%token* below.
  NOTE: Symbolic tokens are defined as *enum*-values in the parser's base class. The names of symbolic tokens may not be equal to the names of the members and types defined by **bisonc++** itself (see the next sections). This requirement is *not* enforced by **bisonc++**, but compilation errors may result if this requirement is violated.
- **%type** *<type> non-terminal ...*
  In combination with *%polymorphic* or *%union*: associate the semantic value of a non-terminal symbol with a polymorphic semantic value tag or union field defined by these directives.

- **%union** *union-definition*
  Acts identically to the identically named **bison** and **bison++** declaration. **Bisonc++** generates a union, named *STYPE__*, as its semantic type.
- **%weak-tags**
  This directive is ignored unless the *%polymorphic* directive was specified. It results in the declaration of *enum Tag__* rather than *enum class Tag__*. When in doubt, don't use this directive.

5. POLYMORPHIC SEMANTIC VALUES

The *%polymorphic* directive results in **bisonc++** generating a parser using polymorphic semantic values. The various semantic values are specified as pairs, consisting of *tags* (which are **C++** identifiers), and **C++** type names. Tags and type names are separated from each other by colons. Multiple tag and type name combinations are separated from each other by semicolons, and an optional semicolon ends the final tag/type specification.

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

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

The identifier to the left of the colon is called the *tag-identifier* (or simply *tag*), and the type name to the right of the colon is called the *type-name*. The type-names must be built-in types or must offer default constructors.

If type-names refer to types declared in header files that were not already included by the parser's base class header, then these header files must be inserted using the *%baseclass-preinclude* directive.

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

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

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

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

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

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

automatically associates $$, $1 and $3 with *int* values. $$ is an lvalue (representing the semantic value associated with the *expr:* rule), while $1 and $3 represent the *int* semantic value associated with the *expr*

non-terminal in the production rule *'-' expr* (rvalues).

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

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

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

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

$$ or $1 specifications

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

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

auto-tag: $$ and $1 represent, respectively, *$$.get<tag>()* and *$1.get<tag>()*;

tag-error: *error:* tag undefined;

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

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

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

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

type:
    ident arg
;

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

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

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

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

type:
```

```
        ident arg
    ;

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

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

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

The *%polymorphic* directive adds the following definitions and declarations to the generated base class header and parser source file (if the *%namespace* directive was used then all declared/defined elements are placed inside the name space that is specified by the *%namespace* directive):

- Three additional headers are included by the parser's base class header:

```
#include <memory>
#include <stdexcept>
#include <type_traits>
```

- All semantic value type identifiers are collected in a strongly typed `*Tag__*' enumeration. E.g.,

```
enum class Tag__
{
    INT,
    STRING,
    VECT
};
```

- The name space *Meta__* contains almost all of the code implementing polymorphic values.

The name space *Meta__* contains the following elements:

- A polymorphic base class *Base*. This class is normally not explicitly referred to by user-defined code. Refer to by **bisonc++**'s user manual for a detailed description of this class.
- For each of the tag-identifiers specified with the *%polymorphic* directive a class template *Semantic<Tag__>* is defined, containing a data element of the type-name matching the *Tag__* for which *Semantic<Tag__>* was derived.

  The *Semantic<Tag__>* classes are normally not explicitly referred to by user-defined code. Refer to by **bisonc++**'s user manual for a detailed description of these classes.
- A class *SType*, derived from *std::shared_ptr<Base>*. This class becomes the parser's semantic value type, offering the following members:

Constructors: default, copy and move constructors;

Assignment operators: copy and move assignment operators declaring *SType* or any of the *%polymorphic* type-names as their right-hand side operands;

Tag__ tag() const, returning *Semantic<Tag__>*'s *Tag__* value;

ReturnType get<Tag__>() const. ReturnType refers to the semantic value stored inside *Semantic<Tag__>*. If the type-name is a built-in type a copy of the value is returned, otherwise a reference to a constant object is returned;
This member checks for 0-pointers and for *Tag__* mismatches between the requested and actual *Tag__*, throwing a *std::logic_error* if so.

DataType &get<Tag__>() returns a reference to the (modifiable) semantic value stored inside *Semantic<Tag__>*.

This member checks for 0-pointers and for *Tag__* mismatches between the requested and actual *Tag__*, in that case replacing the current *Semantic* object pointed to by a new *Semantic<Tag__>* object of the requested *Tag__*.

ReturnType data<Tag__>() const. ReturnType refers to the semantic value stored inside *Semantic<Tag__>*. If the type-name is a built-in type a copy of the value is returned, otherwise a reference to a constant object is returned;

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

DataType &data<Tag__>() returns a reference to the (modifiable) semantic value stored inside *Semantic<Tag__>*.

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

Since **bisonc++** declares *typedef Meta__::SType STYPE__*, polymorphic semantic values can be used without referring to the name space *Meta__*.

6. PUBLIC MEMBERS AND -TYPES

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

- **LTYPE__**:
  The parser's location type (user-definable). Available only when either *%lsp-needed*, *%ltype* or *%locationstruct* has been declared.
- **STYPE__**:
  The parser's stack-type (user-definable), defaults to **int**.
- **Tokens__**:
  The enumeration type of all the symbolic tokens defined in the grammar file (i.e., **bisonc++**'s

input file). The scanner should be prepared to return these symbolic tokens. Note that, since the symbolic tokens are defined in the parser's class and not in the scanner's class, the lexical scanner must prefix the parser's class name to the symbolic token names when they are returned. E.g., *return Parser::IDENT* should be used rather than *return IDENT*.

- **int parse()**:
  The parser's parsing member function. It returns 0 when parsing was successfully completed; 1 if errors were encountered while parsing the input.
- **void setDebug(bool mode)**:
  This member can be used to activate or deactivate the debug-code compiled into the parsing function. It is always defined but is only operational if the *%debug* directive or *--debug* option was specified. When debugging code has been compiled into the parsing function, it is *not* active by default. To activate the debugging code, use *setDebug(true)*.

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

When the *%polymorphic* directive is used:

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

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

7. PRIVATE ENUMS AND -TYPES

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

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

```
DEFAULT_RECOVERY_MODE__,
UNEXPECTED_TOKEN__
```

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

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

unexpected token (3) all subsequent terminal tokens are ignored until a token is retrieved which is a continuation token in that state.

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

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

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

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

8. PRIVATE MEMBER FUNCTIONS

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

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

- **void errorRecovery__()**:
  Used internally.
- **void Base::errorVerbose__()**:
  Used internally.
- **void exceptionHandler__(std::exception const &exc)**:
  This member's default implementation is provided inline in the *parser.ih* internal header file. It consists of a mere *throw* statement, rethrowing a caught exception.

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

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

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

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

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

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

  In Parser.h, just below the declaration

```
void print__();
```

  add:

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

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

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

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

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

- **void Base::push__()**:

Used internally.

- **void Base::pushToken__()**:

Used internally.

- **void Base::reduce__()**:

Used internally.

- **void Base::symbol__()**:

Used internally.

- **void Base::top__()**:

Used internally.)

9. PRIVATE DATA MEMBERS

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

- **size_t d_acceptedTokens__**:

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

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

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

  Subsequently, the lexical scanner may assign a value to the parser's *d_loc__* variable through the pointer to *d_loc__* stored inside the lexical scanner.
- **LTYPE__ d_lsp__**:
  The location stack pointer. Do not modify.
- **size_t d_nErrors__**:
  The number of errors counted by *parse*. It is initialized by the parser's base class initializer, and is updated while *parse* executes. When *parse* has returned it contains the total number of errors counted by *parse*. Errors are not counted if suppressed (i.e., if *d_acceptedTokens__* is less than *d_requiredTokens__*).
- **size_t d_nextToken__**:
  A pending token. Do not modify.
- **size_t d_requiredTokens__**:
  Defines the minimum number of accepted tokens that the *parse* function must have processed before a syntactic error can be generated.
- **int d_state__**:
  The current parsing state. Do not modify.
- **int d_token__**:
  The current token. Do not modify.
- **STYPE__ d_val__**:
  The semantic value of a returned token or non-terminal symbol. With non-terminal tokens it is assigned a value through the action rule's symbol *$$*. Lexical scanners may be offered the facility to assign a semantic value to this variable in parallel with a returned token. In order to allow a scanner access to *d_val__*, *d_val__*'s address should be passed to the scanner. This can be realized, for example, by passing *d_val__*'s address to the lexical scanner's constructor. Subsequently, the lexical scanner may assign a value to the parser's *d_val__* variable through the pointer to *d_val__* stored in a data member of the lexical scanner. Note that in some cases this approach *must* be used to make available the correct semantic value to the parser. In particular, when a grammar state defines multiple reductions, depending on the next token, the reduction's action only takes place following the retrieval of the next token, thus losing the initially matched token text.
- **LTYPE__ d_vsp__**:

The semantic value stack pointer. Do not modify.

10. TYPES AND VARIABLES IN THE ANONYMOUS NAMESPACE

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

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

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

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

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

  These tokens are used by the parser to define the types of the various states of the analyzed grammar.
- **PI__** (Production Info):
  This *struct* provides information about production rules. It has two fields: *d_nonTerm* is the identification number of the production's non-terminal, *d_size* represents the number of elements of the productin rule.
- **static PI__ s_productionInfo**:
  Used internally by the parsing function.
- **SR__** (Shift-Reduce Info):
  This *struct* provides the shift/reduce information for the various grammatic states. *SR__* values are collected in arrays, one array per grammatic state. These array, named *s_<nr>*, where tt<nr> is a state number are defined in the anonymous namespace as well. The *SR__* elements consist of two unions, defining fields that are applicable to, respectively, the first, intermediate and the last array elements.
  The first element of each array consists of (1st field) a *StateType* and (2nd field) the index of the

last array element; intermediate elements consist of (1st field) a symbol value and (2nd field) (if negative) the production rule number reducing to the indicated symbol value or (if positive) the next state when the symbol given in the 1st field is the current token; the last element of each array consists of (1st field) a placeholder for the current token and (2nd field) the (negative) rule number to reduce to by default or the (positive) number of an error-state to go to when an erroneous token has been retrieved. If the 2nd field is zero, no error or default action has been defined for the state, and error-recovery is attepted.

- **STACK_EXPANSION**:
  An enumeration value specifying the number of additional elements that are added to the state- and semantic value stacks when full.
- **static SR__ s_<nr>[]**:
  Here, *<nr>* is a numerical value representing a state number. Used internally by the parsing function.
- **static SR__ *s_state[]**:
  Used internally by the parsing function.

11. RESTRICTIONS ON TOKEN NAMES

To avoid collisions with names defined by the parser's (base) class, the following identifiers should not be used as token names:

- Identifiers ending in two underscores;
- Any of the following identifiers: *ABORT*, *ACCEPT*, *ERROR*, *clearin*, *debug*, or *setDebug*.

12. OBSOLETE SYMBOLS

All **DECLARATIONS** and **DEFINE** symbols not listed above but defined in **bison++** are obsolete with **bisonc++**. In particular, there is no *%header{ ... %}* section anymore. Also, all **DEFINE** symbols related to member functions are now obsolete. There is no need for these symbols anymore as they can simply be declared in the class header file and defined elsewhere.

13. EXAMPLE

Using a fairly worn-out example, we'll construct a simple calculator below. The basic operators as well as parentheses can be used to specify expressions, and each expression should be terminated by a newline. The program terminates when a *q* is entered. Empty lines result in a mere prompt.

First an associated grammar is constructed. When a syntactic error is encountered all tokens are skipped until then next newline and a simple message is printed using the default *error* function. It is assumed that no semantic errors occur (in particular, no divisions by zero). The grammar is decorated with actions performed when the corresponding grammatical production rule is recognized. The grammar itself is rather standard and straightforward, but note the first part of the specification file, containing various other directives, among which the *%scanner* directive, resulting in a composed *d_scanner* object as well as an implementation of the member function *int lex*. In this example, a common *Scanner* class construction strategy was used: the class *Scanner* was derived from the class *yyFlexLexer* generated by **flex**++(1). The actual process of constructing a class using **flex**++(1) is beyond the scope of this man-page, but **flex**++(1)'s specification file is mentioned below, to further complete the example. Here is **bisonc**++'s input file:

```
%filenames parser
%scanner ../scanner/scanner.h

                                // lowest precedence
%token  NUMBER                  // integral numbers
        EOLN                    // newline

%left   '+' '-'
%left   '*' '/'
%right  UNARY
                                // highest precedence

%%

expressions:
    expressions  evaluate
|
    prompt
;

evaluate:
    alternative prompt
;

prompt:
    {
        prompt();
    }
;

alternative:
    expression EOLN
    {
        cout << $1 << endl;
    }
|
    'q' done
|
    EOLN
|
    error EOLN
;

done:
    {
        cout << "Done.\n";
        ACCEPT();
    }
;

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

```
    {
        $$ = $1 * $3;
    }
|
    expression '/' expression
    {
        $$ = $1 / $3;
    }
|
    '-' expression      %prec UNARY
    {
        $$ = -$2;
    }
|
    '+' expression      %prec UNARY
    {
        $$ = $2;
    }
|
    '(' expression ')'
    {
        $$ = $2;
    }
|
    NUMBER
    {
        $$ = stoul(d_scanner.matched());
    }
;
```

Next, **bisonc++** processes this file. In the process, **bisonc++** generates the following files from its skeletons:

- The parser's base class, which should not be modified by the programmer:

```
// Generated by Bisonc++ V4.01.02 on Wed, 06 Mar 2013 15:26:21 +0100

#ifndef ParserBase_h_included
#define ParserBase_h_included

#include <vector>
#include <iostream>


namespace // anonymous
{
    struct PI__;
}



class ParserBase
{
    public:
// $insert tokens

    // Symbolic tokens:
    enum Tokens__
    {
        NUMBER = 257,
        EOLN,
```

```
        UNARY,
    };

// $insert STYPE
typedef int STYPE__;

    private:
        int d_stackIdx__;
        std::vector<size_t>   d_stateStack__;
        std::vector<STYPE__>  d_valueStack__;

    protected:
        enum Return__
        {
            PARSE_ACCEPT__ = 0,   // values used as parse()'s return values
            PARSE_ABORT__  = 1
        };
        enum ErrorRecovery__
        {
            DEFAULT_RECOVERY_MODE__,
            UNEXPECTED_TOKEN__,
        };
        bool        d_debug__;
        size_t      d_nErrors__;
        size_t      d_requiredTokens__;
        size_t      d_acceptedTokens__;
        int         d_token__;
        int         d_nextToken__;
        size_t      d_state__;
        STYPE__    *d_vsp__;
        STYPE__     d_val__;
        STYPE__     d_nextVal__;

        ParserBase();

        void ABORT() const;
        void ACCEPT() const;
        void ERROR() const;
        void clearin();
        bool debug() const;
        void pop__(size_t count = 1);
        void push__(size_t nextState);
        void popToken__();
        void pushToken__(int token);
        void reduce__(PI__ const &productionInfo);
        void errorVerbose__();
        size_t top__() const;

    public:
        void setDebug(bool mode);
};

inline bool ParserBase::debug() const
{
    return d_debug__;
}

inline void ParserBase::setDebug(bool mode)
{
    d_debug__ = mode;
}
```

```
inline void ParserBase::ABORT() const
{
    throw PARSE_ABORT__;
}

inline void ParserBase::ACCEPT() const
{
    throw PARSE_ACCEPT__;
}

inline void ParserBase::ERROR() const
{
    throw UNEXPECTED_TOKEN__;
}


// As a convenience, when including ParserBase.h its symbols are available as
// symbols in the class Parser, too.
#define Parser ParserBase


#endif
```

- The parser class *parser.h* itself. In the grammar specification various member functions are used (e.g., *done*) and *prompt*. These functions are so small that they can very well be implemented inline. Note that *done* calls *ACCEPT* to terminate further parsing. *ACCEPT* and related members (e.g., *ABORT*) can be called from any member called by *parse*. As a consequence, action blocks could contain mere function calls, rather than several statements, thus minimizing the need to rerun **bisonc++** when an action is modified.

  Once **bisonc++** had created *parser.h* it was augmented with the required additional members, resulting in the following final version:

```
#ifndef Parser_h_included
#define Parser_h_included

// $insert baseclass
#include "parserbase.h"
// $insert scanner.h
#include "../scanner/scanner.h"


#undef Parser
class Parser: public ParserBase
{
    // $insert scannerobject
    Scanner d_scanner;

    public:
        int parse();

    private:
        void error(char const *msg);
        int lex();

        void print();
        void prompt();
        void done();
```

```
    // support functions for parse():
        void executeAction(int ruleNr);
        void errorRecovery();
        int lookup(bool recovery);
        void nextToken();
        void print__();
};

inline void Parser::prompt()
{
    std::cout << "? " << std::flush;
}

inline void Parser::done()
{
    std::cout << "Done\n";
    ACCEPT();
}

#endif
```

- To complete the example, the following lexical scanner specification was used:

```
%interactive
%filenames scanner

%%

[ \t]+                          // skip white space

\n                              return Parser::EOLN;

[0-9]+                          return Parser::NUMBER;

.                               return matched()[0];


%%
```

- Since no member functions other than *parse* were defined in separate source files, only *parse* includes *parser.ih*. Since *cerr* is used in the grammar's actions, a *using namespace std* or comparable statement is required. This was effectuated from *parser.ih* Here is the implementation header declaring the standard namespace:

```
// Generated by Bisonc++ V4.01.02 on Wed, 06 Mar 2013 15:10:36 +0100

    // Include this file in the sources of the class Parser.

// $insert class.h
#include "parser.h"


inline void Parser::error(char const *msg)
{
    std::cerr << msg << '\n';
}

// $insert lex
```

```
inline int Parser::lex()
{
    return d_scanner.lex();
}

inline void Parser::print()
{
    print__();           // displays tokens if --print was specified
}


    // Add here includes that are only required for the compilation
    // of Parser's sources.



    // UN-comment the next using-declaration if you want to use
    // int Parser's sources symbols from the namespace std without
    // specifying std::

using namespace std;
```

The implementation of the parsing member function *parse* is basically irrelevant, since it should not be modified by the programmer. It was written on the file *parse.cc*.

- Finally, here is the program offering our simple calculator:

```
#include "parser/parser.h"

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

14. USING PARSER-CLASS SYMBOLS IN LEXICAL SCANNERS

Note here that although the file *parserbase.h*, defining the parser class' base-class, rather than the header file *parser.h* defining the parser class is included, the lexical scanner may simply return tokens of the class *Parser* (e.g., *Parser::NUMBER* rather than *ParserBase::NUMBER*). In fact, using a simple *#define - #undef* pair generated by the **bisonc++** respectively at the end of the base class header the file and just before the definition of the parser class itself it is the possible to assume in the lexical scanner that all symbols defined in the the parser's base class are actually defined in the parser class itself. It the should be noted that this feature can only be used to access base class the *enum* and types. The actual parser class is not available by the time the the lexical scanner is defined, thus avoiding circular class dependencies.

15. FILES

- **bisonc++base.h**: skeleton of the parser's base class;
- **bisonc++.h**: skeleton of the parser class;
- **bisonc++.ih**: skeleton of the implementation header;
- **bisonc++.cc**: skeleton of the member *parse*;

- **bisonc++polymorphic**: skeleton of the declarations used by *%polymorphic*;
- **bisonc++polymorphic.inline**: skeleton of the inline implementations of the members declared in **bisonc++polymorphic**.

16. SEE ALSO

bison(1), **bison**++(1), **bison.info** (using texinfo), **flex**++(1)

Lakos, J. (2001) **Large Scale C++ Software Design**, Addison Wesley.
Aho, A.V., Sethi, R., Ullman, J.D. (1986) **Compilers**, Addison Wesley.

17. BUGS

Parser-class header files (e.g., Parser.h) and parser-class internal header files (e.g., Parser.ih) generated with bisonc++ < 4.02.00 require two hand-modifications when used in combination with bisonc++ >= 4.02.00. See the description of *exceptionHandler__* for details.

Discontinued options:

- **--include-only**
- **--namespace**

To avoid collisions with names defined by the parser's (base) class, the following identifiers should not be used as token nams:

- Identifiers ending in two underscores;
- Any of the following identifiers: *ABORT*, *ACCEPT*, *ERROR*, *clearin*, *debug*, *error*, or *setDebug*.

When re-using files generated by **bisonc++** before version 2.0.0, minor hand-modification might be necessary. The identifiers in the following list (defined in the parser's base class) now have two underscores affixed to them: *LTYPE*, *STYPE* and *Tokens*. When using classes derived from the generated parser class, the following identifiers are available in such derived classes: *DEFAULT_RECOVERY_MODE*, *ErrorRecovery*, *Return*, *UNEXPECTED_TOKEN*, *d_debug*, *d_loc*, *d_lsp*, *d_nErrors*, *d_nextToken*, *d_state*, *d_token*, *d_val*, and *d_vsp*. When used in derived classes, they too need two underscores affixed to them.

The member function *void lookup* (< 1.00) was replaced by *int lookup*. When regenerating parsers created by early versions of **bisonc++** (versions before version 1.00), *lookup*'s prototype should be corrected by hand, since **bisonc++** will not by itself rewrite the parser class's header file.

The *Semantic* parser, mentioned in **bison++**(1) is not implemented in **bisonc++**(1). According to **bison++**(1) the semantic parser was not available in **bison++** either. It is possible that the *Pure* parser is now available through the *--thread-safe* option.

18. ABOUT bisonc++

Bisonc++ was based on **bison++**, originally developed by Alain Coetmeur (coetmeur@icdc.fr), R&D department (RDT), Informatique-CDC, France, who based his work on **bison**, GNU version 1.21.

Bisonc++ version 0.98 and beyond is a complete rewrite of an LALR-1 parser generator, closely following the construction process as described in Aho, Sethi and Ullman's (1986) book **Compilers** (i.e., the *Dragon book*). It uses the same grammar specification as **bison** and **bison++**, and it uses practically the same options and directives as **bisonc++** versions earlier than 0.98. Variables, declarations and macros that are obsolete were removed.

AUTHOR

Frank B. Brokken (f.b.brokken@rug.nl).