flexc++

flexc++.1.07.00.tar.gz

2008-2013

flexc++(1)

flexc++.1.07.00.tar.gz flexc++ scanner generator

2008-2013

NAME

flexc++ - Generate a C++ scanner class and parsing function

SYNOPSIS

flexc++ [options] *rules-file*

DESCRIPTION

Flexc++(1) was designed after **flex**(1) and **flex**++(1). Like these latter two programs **flexc**++ generates code performing pattern-matching on text, possibly executing actions when certain *regular expressions* are recognized.

Flexc++, contrary to **flex** and **flex**++, generates code that is explicitly intended for use by **C**++ programs. The well-known **flex**(1) program generates **C** source-code and **flex**++(1) merely offers a **C**++-like shell around the *yylex* function generated by **flex**(1) and hardly supports present-day ideas about **C**++ software development.

Contrary to this, **flexc**++ creates a **C**++ class offering a predefined member function **lex** matching input against regular expressions and possibly executing **C**++ code once regular expressions were matched. The code generated by **flexc**++ is pure **C**++, allowing its users to apply all of the features offered by that language.

Below, the following sections may be consulted for specific details:

- **1. QUICK START**: a quick start overview about how to use **flexc**++.
- **2. QUICK START: FLEXC++ and BISONC++**: a quick start overview about how to use **flexc**++ in combination with **bisonc**++(1)
- **3. GENERATED FILES**: files generated by **flexc**++ and their purposes
- **4. OPTIONS**: options available for **flexc**++
- **5. INTERACTIVE SCANNERS**: how to create an interactive scanner

- **6. SPECIFICATION FILE(S)**: the format and contents of **flexc**++ input files, specifying the Scanner's characteristics
- **6.1. FILE SWITCHING**: how to switch to another input specification file
- **6.2. DIRECTIVES**: directives that can be used in input specification files
- **6.3. MINI SCANNERS**: how to declare mini-scanners
- **6.4. DEFINITIONS**: how to define symbolic names for regular expressions
- **6.5. %% SEPARATOR**: the separator between the input specification sections
- **6.6. REGULAR EXPRESSIONS**: regular expressions supported by **flexc**++
- **6.7. SPECIFICATION EXAMPLE**: an example of a specification file

- **7. THE CLASS INTERFACE: SCANNER.H**: Constructors and members of the scanner class generated by **flexc**++
- **7.1. NAMING CONVENTION**: symbols defined by **flexc**++ in the scanner class.
- **7.2 CONSTRUCTORS**: constructors defined in the scanner class.
- **7.3 PUBLIC MEMBER FUNCTION**: public member declared in the scanner class.
- **7.4. PRIVATE MEMBER FUNCTIONS**: private members declared in the scanner class.
- **7.5. SCANNER CLASS HEADER EXAMPLE**: an example of a generated scanner class header

- **8.1. THE SCANNER BASE CLASS**: the scanner class is derived from a base class. The base class is described in this section
- **8.2. PUBLIC ENUMS AND -TYPES**: enums and types declared by the base class
- **8.3. PROTECTED ENUMS AND -TYPES**: enumerations and types used by the scanner and scanner base classes
- **8.4. NO PUBLIC CONSTRUCTORS**: the scanner base class does not offer public constructors.
- **8.5. PUBLIC MEMBER FUNCTIONS**: several members defined by the scanner base class have public access rights.
- **8.6. PROTECTED CONSTRUCTORS**: the base class can be constructed by a derived class. Usually this is the scanner class generated by **flexc**++.
- **8.7. PROTECTED MEMBER FUNCTIONS**: this section covers the base class member functions that can only be used by scanner class or scanner base class members
- **8.8. PROTECTED DATA MEMBERS**: this section covers the base class data members that can only be used by scanner class or scanner base class members
- **8.9. FLEX++ TO FLEXC++ MEMBERS**: a short overview of frequently used **flex**(1) members that received different names in **flexc**++.

- **9.1. THE CLASS INPUT**: the scanner's job is completely decoupled from the actual input stream. The class *Input*, nested within the scanner base class handles the communication with the input streams. The class *Input*, is described in this section.
- **9.2. CONSTRUCTORS**: the class *Input* can easily be replaced by another class. The

constructor-requirements are described in this section.
- **9.3. REQUIRED PUBLIC MEMBER FUNCTIONS**: this section covers the required public members of a self-made *Input* class

1. QUICK START

A bare-bones, no-frills scanner is generated as follows:

- Create a file *lexer* defining the regular expressions to recognize, and the tokens to return. Use token values exceeding 0xff if plain ascii character values can also be used as token values. Example (assume capitalized words are token-symbols defined in an enum defined by the scanner class):

```
%%
[ \t\n]+                            // skip white space chars.
[0-9]+                              return NUMBER;
[[:alpha:]_][[:alpha:][:digit:]_]*  return IDENTIFIER;
.                                   return matched()[0];
```

- Execute:

```
flexc++ lexer
```

This generates four files

:*Scanner.h*, *Scanner.ih*, *Scannerbase.h*, and *lex.cc*

- Edit *Scanner.h*, add the enum defining the token-symbols in (usually) the public section of the class *Scanner*. E.g.,

```
class Scanner: public ScannerBase
{
    public:
        enum Tokens
        {
            IDENTIFIER = 0x100,
            NUMBER
        };
    // ... (etc, as generated by flexc++)
```

- Create a file defining *int main*, e.g.:

```
#include <iostream>
#include "Scanner.h"

using namespace std;

int main()
{
```

```
    Scanner scanner;        // define a Scanner object

    while (int token = scanner.lex())   // get all tokens
    {
        string const &text = scanner.matched();
        switch (token)
        {
            case Scanner::IDENTIFIER:
                cout << "identifier: " << text << '\n';
            break;

            case Scanner::NUMBER:
                cout << "number: " << text << '\n';
            break;

            default:
                cout << "char. token: `" << text << "'\n";
            break;
        }
    }
}
```

- Compile all *.cc* files:

```
    g++ --std=c++11 *.cc
```

- To `tokenize' *main.cc*, execute:

```
    a.out < main.cc
```

)

QUICK START: FLEXC++ and BISONC++

To interface **flexc**++ to the **bisonc**++(1) parser generator proceed as follows:

- Specify a grammar that can be processed by **bisonc**++(1). Assuming that the scanner and parser are developed in, respectively, the sub-directories *scanner* and *parser*, then a simple grammar specification that can be used with the scanner developed in the previous section is, e.g., write the file *parser/grammar*:

```
    %scanner                ../scanner/Scanner.h
    %scanner-token-function d_scanner.lex()

    %token IDENTIFIER NUMBER CHAR

    %%

    startrule:
        startrule tokenshow
    |
        tokenshow
    ;
```

```
tokenshow:
    token
    {
        std::cout << "matched: " << d_scanner.matched() << '\n';
    }
;

token:
    IDENTIFIER
|
    NUMBER
|
    CHAR
;
```

- Write a scanner specification file. E.g.,

```
%%

[ \t\n]+                            // skip white space chars.
[0-9]+                              return Parser::NUMBER;
[[:alpha:]_][[:alpha:][:digit:]_]*  return Parser::IDENTIFIER;
.                                   return Parser::CHAR;
```

  This causes the scanner to return *Parser* tokens to the generated parser.

- Add the line

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

  to the file *scanner/Scanner.ih*

- Write a simple *main* function in the file *main.cc*. E.g.,

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

int main(int argc, char **argv)
{
    Parser parser;

    parser.parse();
}
```

- Generate a scanner in the *scanner* subdirectory:

```
flexc++ lexer
```

- Generate a parser in the *parser* subdirectory:

```
bisonc++ grammar
```

- Compile all sources:

```
g++ --std=c++0x *.cc */*.cc
```

- Execute the program, providing it some source file to be processed:

```
a.out < main.cc
```

3. GENERATED FILES

Flexc++ generates four files from a well-formed input file:

- A file containing the implementation of the *lex* member function and its support functions. By default this file is named *lex.cc*.
- A file containing the scanner's class interface. By default this file is named *Scanner.h*. The scanner class itself is generated once and is thereafter `owned' by the programmer, who may change it *ad-lib*. Newly added members (data members, function members) will survive future **flexc**++ runs as **flexc**++ will never rewrite an existing scanner class interface file, unless explicitly ordered to do so.
- A file containing the interface of the scanner class's *base class*. The scanner class is publicly derived from this base class. It is used to minimize the size of the scanner interface itself. The scanner base class is `owned' by **flexc**++ and should never be hand-modified. By default the scanner's base class is provided in the file *Scannerbase.h*. At each new **flexc**++ run this file is rewritten unless **flexc**++ is explicitly ordered *not* to do so.
- A file containing the *implementation header*. This file should contain includes and declarations that are only required when compiling the members of the scanner class. By default this file is named *Scanner.ih*. This file, like the file containing the scanner class's interface is never rewritten by **flexc**++ unless **flexc**++ is explicitly ordered to do so.

4. OPTIONS

Where available, single letter options are listed between parentheses following their associated long-option variants. Single letter options require arguments if their associated long options require arguments as well. 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 **flexc**++ cannot modify (e.g., a scanner 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.

- **--baseclass-header**=*filename* (**-b**)
  Use *filename* as the name of the file to contain the scanner class's base class. Defaults to the name of the scanner class plus *base.h*

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

- **--baseclass-skeleton**=*pathname* (**-C**)
  Use *pathname* as the path to the file containing the skeleton of the scanner class's base class. Its filename defaults to *flexc++base.h*.
- **--case-insensitive**
  Use this option to generate a scanner *case insensitively* matching regular expressions. All regular expressions specified in **flexc++**'s input file are interpreted case insensitively and the resulting scanner object will case insensitively interpret its input.

  When this option is specified the resulting scanner does not distinguish between the following rules:

```
First       // initial F is transformed to f
first
FIRST       // all capitals are transformed to lower case chars
```

  With a case-insensitive scanner only the first rule can be matched, and **flexc++** will issue warnings for the second and third rule about rules that cannot be matched.

  Input processed by a case-insensitive scanner is also handled case insensitively. The above mentioned *First* rule is matched for all of the following input words: *first First FIRST firST*.

  Although the matching process proceeds case insensitively, the matched text (as returned by the scanner's *matched()* member) always contains the original, unmodified text. So, with the above input *matched()* returns, respectively *first, First, FIRST* and *firST*, while matching the rule *First*.

- **--class-header**=*filename* (**-c**)
  Use *filename* as the name of the file to contain the scanner class. Defaults to the name of the scanner class plus the suffix *.h*
- **--class-name**=*className*
  Use *className* (rather than *Scanner*) as the name of the scanner class. Unless overridden by other options generated files will be given the (transformed to lower case) *className** name instead of *scanner**.

  A warning is issued if this option is used and an already existing scanner-class header file does not define *class `className'*

- **--class-skeleton**=*pathname* (**-C**)
  Use *pathname* as the path to the file containing the skeleton of the scanner class. Its filename defaults to *flexc++.h*.
- **--construction** (**-K**)
  Write details about the lexical scanner to the file *`rules-file'.output*. Details cover the used character ranges, information about the regexes, the raw NFA states, and the final DFAs.
- **--debug** (**-d**)

Provide *lex* and its support functions with debugging code, showing the actual parsing process on the standard output stream. When included, the debugging output is active by default, but its activity may be controlled using the *setDebug(bool on-off)* member. Note that *#ifdef DEBUG* macros are not used anymore. By rerunning **flexc++** without the **--debug** option an equivalent scanner is generated not containing the debugging code.

- **--filenames**=*genericName* (**-f**)
  Generic name of generated files (header files, not the *lex*-function source file, see the *--lex-source* option for that). By default the header file names will be equal to the name of the generated class.
- **--help** (**-h**)
  Write basic usage information to the standard output stream and terminate.
- **--implementation-header**=*filename* (**-i**)
  Use *filename* as the name of the file to contain the implementation header. Defaults to the name of the generated scanner class plus the suffix *.ih*. The implementation header should contain all directives and declarations *only* used by the implementations of the scanner's member functions. It is the only header file that is included by the source file containing **lex()**'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 scanner class in one header file.

  A warning is issued if this option is used and an already existing *'filename'* file does not include the scanner class header file.

- **--implementation-skeleton**=*pathname* (**-I**)
  Use *pathname* as the path to the file containing the skeleton of the implementation header. Its filename defaults to *flexc++.ih*.
- **--lex-skeleton**=*pathname* (**-L**)
  Use *pathname* as the path to the file containing the *lex()* member function's skeleton. Its filename defaults to *flexc++.cc*.
- **--lex-function-name**=*funname*
  Use *funname* rather than *lex* as the name of the member function performing the lexical scanning.
- **--lex-source**=*filename* (**-l**)
  Define *filename* as the name of the source file to contain the scanner member function *lex*. Defaults to *lex.cc*.
- **--matched-rules** (**-'R'**)
  The generated scanner will write the numbers of matched rules to the standard output. It is implied by the *--debug* option. Displaying the matched rules can be suppressed by calling the generated scanner's member *setDebug(false)* (or, of course, by re-generating the scanner without using specifying *--matched-rules*).
- **--max-depth**=*depth* (**-m**)
  Set the maximum inclusion depth of the lexical scanner's specification files to *depth*. By default the maximum depth is set to 10. When more than *depth* specification files are used the scanner throws a *Max stream stack size exceeded std::length_error* exception.
- **--namespace**=*identifier*
  Define the scanner class in the namespace *identifier*. By default no namespace is used. If this options is used the implementation header is provided with a commented out *using namespace* declaration for the requested namespace. In addition, the scanner and scanner 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 scanner-class header file does not define *namespace identifier*.

- **--no-baseclass-header**
  Do not write the file containing the scanner's base class interface even if it doesn't yet exist. By default the file containing the scanner's base class interface is (re)written each time **flexc++** is called.
- **--no-lines**
  Do not put **#line** preprocessor directives in the file containing the scanner's *lex* function. By default *#line* directives are entered at the beginning of the action statements in the generated *lex.cc* file, allowing the compiler and debuggers to associate errors with lines in your grammar specification file, rather than with the source file containing the *lex* function itself.
- **--no-lex-source**
  Do not write the file containing the scanner's predefined scanner member functions, even if that file doesn't yet exist. By default the file containing the scanner's *lex* member function is (re)written each time **flexc++** is called. This option should normally be avoided, as this file contains parsing tables which are altered whenever the grammar definition is modified.
- **--own-tokens** (**-T**)
  The tokens returned as well as the text matched when **flexc++** reads its input files(s) are shown when this option is used.

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

- **--print-tokens** (**-t**)
  The tokens returned as well as the text matched by the generated *lex* function are displayed on the standard output stream, just before returning the token to *lex*'s caller. Displaying tokens and matched text is suppressed again when the *lex.cc* file is generated without using this option. The function showing the tokens (*ScannerBase::print__*) is called from *Scanner::printTokens*, which is defined in-line in *Scanner.h*. Calling *ScannerBase::print__*, therefore, can also easily be controlled by an option controlled by the program using the scanner object.

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

- **--regex-calls**
  Show the function call order when parsing regular expressions (this option is normally not required. Its main purpose is to help developers understand what happens when regular expressions are parsed).
- **--show-filenames** (**-F**)
  Write the names of the files that are generated to the standard error stream.
- **--skeleton-directory**=*pathname* (**-S**)
  Defines the directory containing the skeleton files. This option can be overridden by the specific skeleton-specifying options (*-B -C*, *-H*, and *-I*).
- **--target-directory**=*pathname*
  Specifies the directory where generated files should be written. By default this is the directory where **flexc++** is called.
- **--usage** (**-h**)
  Write basic usage information to the standard output stream and terminate.
- **--verbose**(**-V**)
  The verbose option generates on the standard output stream various pieces of additional information, not covered by the *--construction* and *--show-filenames* options.
- **--version** (**-v**)

Display **flexc**++'s version number and terminate.

5. INTERACTIVE SCANNERS

An interactive scanner is characterized by the fact that scanning is postponed until an end-of-line character has been received, followed by reading all information on the line, read so far. **Flexc**++ supports the *%interactive* directive), generating an interactive scanner. Here it is assumed that *Scanner* is the name of the scanner class generated by **flexc**++.

Caveat: generating interactive and non-interactive scanners should not be mixed as their class organizations fundamentally differ, and several of the Scanner class's members are only available in the non-interactive scanner. As the *Scanner.h* file contains the Scanner class's interface, which is normally left untouched by **flexc**++, **flexc**++ cannot adapt the Scanner class when requested to change the interactivity of an existing Scanner class. Because of this support for the *--interactive* option was discontinued at **flexc**++'s 1.01.00 release.

The interactive scanner generated by **flexc**++ has the following characteristics:

- The *Scanner* class is derived privately from *std::istringstream* and (as usual) publicly from *ScannerBase*.
- The *istringstream* base class is constructed by its default constructor.
- The function *lex*'s default implementation is removed from *Scanner.h* and is implemented in the generated *lex.cc* source file. It performs the following tasks:

  - If the token returned by the scanner is not equal to 0 it is returned as then next token;

  - Otherwise the next line is retrieved from the input stream passed to the *Scanner*'s constructor (by default *std::cin*). If this fails, 0 is returned.

  - A *'\n'* character is appended to the just read line, and the scanner's *std::istringstream* base class object is re-initialized with that line;

  - The member *lex__* returns the next token.

This implementation allows code calling *Scanner::lex()* to conclude, as usual, that the input is exhausted when *lex* returns 0.

Here is an example of how such a scanner could be used:

```
// scanner generated using 'flexc++ lexer' with lexer containing
// the %interactive directive
int main()
{
    Scanner scanner;        // by default: read from std::cin

    while (true)
    {
        cout << "? ";       // prompt at each line

        while (true)        // process all the line's tokens
        {
            int token = scanner.lex();
```

```
                if (token == '\n')  // end of line: new prompt
                    break;

                if (token == 0)     // end of input: done
                    return 0;

                                    // process other tokens
                cout << scanner.matched() << '\n';
                if (scanner.matched()[0] == 'q')
                    return 0;
            }
        }
    }
```

6. SPECIFICATION FILE(S)

Flexc++ expects an input file containing directives and the regular expressions that should be recognized by objects of the scanner class generated by **flexc++**. In this man page the elements and organization of **flexc++**'s input file is described.

Flexc++'s input file consists of two sections, separated from each other by a line merely containing two consecutive percent characters:

```
%%
```

The section before this separator contains directives; the section following this separator contains regular expressions and possibly actions to perform when these regular expressions are matched by the object of the scanner class generated by **flexc++**.

White space is usually ignored, as is comment, which may be of the traditional **C** form (i.e., /*, followed by (possibly multi-line) comment text, followed by */, and it may be **C++** end-of-line comment: two consecutive slashes (*//*) start the comment, which continues up to the next newline character.

6.1. FILE SWITCHING

Flexc++'s input file may be split into multiple files. This allows for the definition of logically separate elements of the specifications in different files. Include directives must be specified on a line of their own. To switch to another specification file the following stanza is used:

```
//include file-location
```

The *//include* directive starts in the line's first column. File locations can be absolute or relative to the location of the file containing the *//include* directive. White space characters following *//include* and before the end of the line are ignored. The file specification may be surrounded by double quotes, but these double quotes are not required and are ignored (removed) if present. All remaining characters are expected to define the name of the file where **flexc++**'s rules specifications continue. Once end of file of a sub-file has been reached, processing continues at the line beyond the *//include* directive of the previously scanned file. The end-of-file of the file that was initially specified when **flexc++** was called

indicates the end of **flexc++**'s rules specification.

6.2. DIRECTIVES

The first section of **flexc++**'s input file consists of directives. In addition it may associate regular expressions with symbolic names, allowing you to use these identifiers in the rules section. Each directive is defined on a line of its own. When available, directives are overridden by **flexc++** command line options.

Some directives require arguments, which are usually provided following separating (but optional) = characters. Arguments of directives, are text, surrounded by double quotes (strings). If a string must itself contain a double quote or a backslash, then precede these characters by a backslash. The exceptions are the *%s* and *%x* directives, which are immediately followed by name lists, consisting of identifiers separated by blanks. Here is an example of the definition of a directive:

```
%class-name = "MyScanner"
```

Directives accepting a `filename' do not accept path names, i.e., they cannot contain directory separators (*/*); options 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 a directive conflicts with the contents of a file which **flexc++** cannot modify (e.g., a scanner 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"*
  Defines the name of the file to contain the scanner class's base class interface. Corresponding command-line option: *--baseclass-header*.

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

- **%case-insensitive**
  Generates a scanner *case insensitively* matching regular expressions. All regular expressions specified in **flexc++**'s input file are interpreted case insensitively and the resulting scanner object will case insensitively interpret its input.

  Corresponding command-line option: *--cases-insensitive*.

  When this directive is specified the resulting scanner does not distinguish between the following rules:

  ```
  First       // initial F is transformed to f
  first
  FIRST       // all capitals are transformed to lower case chars
  ```

  With a case-insensitive scanner only the first rule can be matched, and **flexc++** will issue

warnings for the second and third rule about rules that cannot be matched.

Input processed by a case-insensitive scanner is also handled case insensitively. The above mentioned *First* rule is matched for all of the following input words: *first First FIRST firST*.

Although the matching process proceeds case insensitively, the matched text (as returned by the scanner's *matched()* member) always contains the original, unmodified text. So, with the above input *matched()* returns, respectively *first*, *First*, *FIRST* and *firST*, while matching the rule *First*.

- **%class-header** = *"filename"*
  Defines the name of the file to contain the scanner class's interface. Corresponding command-line option: *--class-header*.
- **%class-name** = *"className"*
  Declares the name of the scanner class generated by **flexc++**. This directive corresponds to the *%name* directive used by **flex++**(1). Contrary to **flex++**'s *%name* declaration, *class-name* may appear anywhere in the first section of the grammar specification file. It may be defined only once. If no *class-name* is specified the default class name (*Scanner*) is used. Corresponding command-line option: *--class-name*.

  A warning is issued if this option is used and an already existing scanner-class header file does not define *class `className'*.

- **%debug**
  Provide *lex* and its support functions with debugging code, showing the actual parsing process on the standard output stream. When included, the debugging output is active by default, but its activity may be controlled using the *setDebug(bool on-off)* member. Note that no *#ifdef DEBUG* macros are used in the generated code.

- **%filenames** = *"basename"*
  Defines the basename of the *Scanner.h*, *Scanner.ih*, and *Scannerbase.h* files. E.g., when using the directive

```
%filenames = "scanner"
```

  the names of the generated files are, respectively, *scanner.h*, *scanner.ih*, and *scannerbase.h*. Corresponding command-line option: *--filenames*. The name of the source file (by default *lex.cc*) is controlled by the *%lex-source* directive.
- **%implementation-header** = *"filename"*
  Defines the name of the file to contain the implementation header. Corresponding command-line option: *--implementation-header*.

  A warning is issued if this option is used and an already existing *'filename'* file does not include the scanner class header file.

- **%input-implementation** = *"sourcefile"*
  Defines the pathname of the file containing the implementation of a user-defined *Input* class.
- **%input-interface** = *"interface"*
  Defines the pathname of the file containing the interface of a user-defined *Input* class. See section **9.1. THE CLASS INPUT** in the **flexc++**(1) manual page for additional information about user-defined *Input* classes.

- **%interactive**
  Generate an interactive scanner. An interactive scanner reads lines from the input stream, and then returns the tokens encountered on that line. The interactive scanner implemented by **flexc++** only predefines the *Scanner(std::istream &in, std::ostream &out)* constructor, by default assuming that input is read from *std::cin*. See also section *5. INTERACTIVE SCANNER* section in the **flexc++**(1) manual page.
- **%lex-function-name** = *"funname"*
  Defines the name of the scanner class's member to perform the lexical scanning. If this directive is omitted the default name (*lex*) is used. Corresponding command-line option: *--lex-function-name*.
- **%lex-source** = *"filename"*
  Defines the name of the file to contain the scanner member *lex*. Corresponding command-line option: *--lex-source*.
- **%no-lines**
  Do not put *#line* preprocessor directives in the file containing the scanner's *lex* function. If omitted *#line* directives are added to this file, unless overridden by the command line options *--lines* and *--no-lines*.
- **%namespace** = *"identifer"*
  Define the scanner class in the namespace *identifier*. By default no namespace is used. If this options is used the implementation header is provided with a commented out *using namespace* declaration for the requested namespace. In addition, the scanner and scanner 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 scanner-class header file does not define *namespace identifier*.

- **%print-tokens**
  This option results in the tokens as well as the matched text to be displayed on the standard output stream, just before returning the token to *lex*'s caller. Displaying is suppressed again when the *lex.cc* file is generated without using this directive. The function showing the tokens (*ScannerBase::print__*) is called from *Scanner::print()*, which is defined in-line in *Scanner.h*. Calling *ScannerBase::print__*, therefore, can also easily be controlled by an option controlled by the program using the scanner object. This option does *not* show the tokens returned and text matched by **flexc++** itself when reading its input **s**. If that is what you want, use the *--own-tokens* option.
- **%s** *namelist*
  The *%s* directive is followed by a list of one or more identifiers, separated by blanks. Each identifier is the name of an *inclusive mini scanner*.
- **%skeleton-directory** = *"pathname"*
  Use *pathname* rather than the default (e.g., */usr/share/flexc++*) path when looking for **flexc++**'s skeleton files. Corresponding command-line option: *--skeleton-directory*.
- **%target-directory** = *"pathname"*
  Pathname defines the directory where generated files should be written. By default this is the directory where **flexc++** is called. This directive is overruled by the *--target-directory* command-line option.
- **%x** *namelist*
  The *%x* directive is followed by a list of one or more identifiers, separated by blanks. Each identifier is the name of an *exclusive mini scanner*.

6.3. MINI SCANNERS

Mini scanners come in two flavors: inclusive mini scanners and exclusive mini scanners. The rules that apply to an inclusive mini scanner are the mini scanner's own rules as well as the rules which apply to no mini scanners in particular (i.e., the rules that apply to the default (or *INITIAL*) mini scanner). Exclusive mini scanners only use the rules that were defined for them.

To define an inclusive mini scanner use *%s*, followed by one or more identifiers specifying the name(s) of the mini-scanner(s). To define an exclusive mini scanner use *%x*, followed by or more identifiers specifying the name(s) of the mini-scanner(s). The following example defines the names of two mini scanners: *string* and *comment*:

```
    %x string comment
```

Following this, rules defined in the context of the *string* mini scanner (see below) will only be used when that mini scanner is active.

A **flexc++** input file may contain multiple *%s* and *%x* specifications.

6.4. DEFINITIONS

Definitions are of the form

```
identifier  regular-expression
```

Each definition must be entered on a line of its own. Definitions associate identifiers with regular expressions, allowing the use of *${identifier}* as synonym for its regular expression in the rules section of the **flexc++** input file. One defined, the identifiers representing regular expressions can also be used in subsequent definitions.

Example:

```
FIRST                   [A-Za-z_]
NAME                    {FIRST}[-A-Za-z0-9_]*
```

6.5. %% SEPARATOR

Following directives and definitions a line merely containing two consecutive % characters is expected. Following this line the rules are defined. Rules consist of regular expressions which should be recognized, possibly followed by actions to be executed once a rule's regular expression has been matched.

6.6. REGULAR EXPRESSIONS

The regular expressions defined in **flexc++**'s rules files are matched against the information passed to the scanner's *lex* function.

Regular expressions begin as the first non-blank character on a line. Comment is interpreted as comment as long as it isn't part of the regular expresssion. To define a regular expression starting with two slashes (at least) the first slash can be escaped or double quoted. (E.g., *"//".** defines **C++** comment to end-of-line).

Regular expressions end at the first blank character (to add a blank character, e.g., a space character, to a regular expression, prefix it by a backslash or put it in a double-quoted string).

Actions may be associated with regular expressions. At a match the action that is associated with the regular expression is executed, after which scanning continues when the lexical scanning function (e.g., *lex*) is called again. Actions are not required, and regular expressions can be defined without any actions at all. If such action-less regular expressions are matched then the match is performed silently, after which processing continues.

Flexc++ tries to match as many characters of the input file as possible (i.e., it uses `greedy matching'). Non-greedy matching is accomplished by a combination of a scanner and parser and/or by using the `lookahead' operator (*/*).

The following regular expression `building blocks' are available. More complex regular expressions are created by combining them:

x
: the character `x'

.
: any character (byte) except newline

[xyz]
: a character class; in this case, the pattern matches either an `x', a `y', or a `z'

[abj-oZ]
: a character class containing a range; matches an `a', a `b', any letter from `j' through `o', or a `Z'

[^A-Z]
: a negated character class, i.e., any character except for those in the class. In this example, any non-capital character.

"[xyz]\"foo"
: text between double quotes matches the literal string: *[xyz]"foo*.

\X
: if X is `a', `b', `f', `n', `r', `t', or `v', then the ANSI-C interpretation of `\x' is matched. Otherwise, a literal `X' is matched (this is used to escape operators such as `*').

\0
: a NUL character (ASCII code 0).

\123
: the character with octal value 123.

\x2a

the character with hexadecimal value 2a.

(r)
the regular expression `r'; parentheses are used to override precedence (see below)

{name}
the expansion of the `name' definition.

r*
zero or more regular expressions `r'. This also matches the empty string.

r+
one or more regular expressions `r'.

r?
zero or one regular expression `r'. This also matches the empty string.

rs
the regular expression `r' followed by the regular expression `s'; called concatenation

r{m, n}
regular expression `r' at least m, but at most n times (*1 <= m <= n*).

r{m,}
regular expression `r' m or more times (*1 <= m*).

r{m}
regular expression `r' exactly m times (*1 <= m*).

r|s
either regular expression `r' or regular expression `s'

r/s
regular expression `r' if it is followed by regular expression `s'. The text matched by `s' is included when determining whether this rule results in the longest match, but `s' is then returned to the input before the rule's action (if defined) is executed.

^r
a regular expression `r' at the beginning of a line or file.

r$
a regular expression `r', occurring at the end of a line. This pattern is identical to `r/\n'.

<s>r
a regular exprression `r' in start condition `s'

<s1,s2,s3>r
a regular exprression `r' in start conditions s1, s2, or s3.

<*>r
a regular exprression `r' in all start conditions.

<<EOF>>
an end-of-file.

<s1,s2><<EOF>>
an end-of-file when in start conditions s1 or s2

Inside a character class all regular expression operators lose their special meanings, except for the escape character (\) and the character class operators -, *]]*, and, at the beginning of the class, ^. To add a closing bracket to a character class use *[]*. To add a closing bracket to a negated character class use *[^]*. Once a character class has started, all subsequent character (ranges) are added to the set, until the final closing bracket (*]*) has been reached.

The regular expressions listed above are grouped according to precedence, from highest precedence at the top to lowest at the bottom. From lowest to highest precedence, the operators are:

- |: the or-operator at the end of a line (instead of an action) indicates that this expression's action is identical to the action of the next rule.
- */*: the look-ahead operator;
- |: the or-operator withn a regular expression;
- *CHAR*: individual elements of the regular expression: characters, strings, quoted characters, escaped characters, character sets etc. are all considered *CHAR* elements. Multiple *CHAR* elements can be combined by enclosing them in parentheses (e.g., *(abc)+* indicates sequences of *abc* characters, like *abcabcabc*);
- *, *?*, +, *{*: multipliers:
  ?: zero or one occurrence of the previous element;
  +: one or more repetitions of the previous element;
  *: zero or more repetitions of the previous element;
  {...}: interval specification: a specified number of repetitions of the previous element (see above for specific forms of the interval specification)
- *{+}*, *{-}*: set operators (*{+}* computing the union of two sets, *{-}* computing the difference of the left-hand side set minus the elements in the right-hand side set);

The lex standard defines concatenation as having a higher precedence than the interval expression. This is different from many other regular expression engines, and **flexc++** follows these latter engines, giving all `multiplication operators' equal priority.

Name expansion has the same precedence as grouping (using parentheses to influence the precedence of the other operators in the regular expression). Since the name expansion is treated as a group in **flexc++**, it is not allowed to use the lookahead operator in a name definition (a named pattern, defined in the definition section).

Character classes can also contain character class expressions. These are expressions enclosed inside *[:* and *:]* delimiters (which themselves must appear between the *[* and *]* of the character class. Other elements may occur inside the character class as well). The character class expressions are:

```
[:alnum:] [:alpha:] [:blank:]
[:cntrl:] [:digit:] [:graph:]
[:lower:] [:print:] [:punct:]
[:space:] [:upper:] [:xdigit:]
```

Character class expressions designate a set of characters equivalent to the corresponding standard **C** isXXX function. For example, *[:alnum:]* designates those characters for which *isalnum* returns true - i.e., any alphabetic or numeric character. For example, the following character classes are all equivalent:

```
[[:alnum:]]
[[:alpha:][:digit:]]
[[:alpha:][0-9]]
[a-zA-Z0-9]
```

A negated character class such as the example *[^A-Z]* above will match a newline unless *\n* (or an equivalent escape sequence) is one of the characters explicitly present in the negated character class (e.g., *[^A-Z\n]*). This differs from the way many other regular expression tools treat negated character classes, but unfortunately the inconsistency is historically entrenched. Matching newlines means that a pattern like *[^"]** can match the entire input unless there's another quote in the input.

Flexc++ allows negation of character class expressions by prepending ^ to the POSIX character class name.

```
[:^alnum:] [:^alpha:] [:^blank:]
[:^cntrl:] [:^digit:] [:^graph:]
[:^lower:] [:^print:] [:^punct:]
[:^space:] [:^upper:] [:^xdigit:]
```

The *{-}* operator computes the difference of two character classes. For example, *[a-c]{-}[b-z]* represents all the characters in the class *[a-c]* that are not in the class *[b-z]* (which in this case, is just the single character *a*). The *{-}* operator is left associative, so *[abc]{-}[b]{-}[c]* is the same as *[a]*.

The *{+}* operator computes the union of two character classes. For example, *[a-z]{+}[0-9]* is the same as *[a-z0-9]*. This operator is useful when preceded by the result of a difference operation, as in, *[[:alpha:]]{-}[[:lower:]]{+}[q]*, which is equivalent to *[A-Zq]* in the **C** locale.

A rule can have at most one instance of trailing context (the */* operator or the *$* operator). The start condition, ^, and *<<EOF>>* patterns can only occur at the beginning of a pattern, and cannot be surrounded by parentheses. The characters ^ and *$* only have their special properties at, respectively, the beginning and end of regular expressions. In all other cases they are treated as a normal characters.

6.7. SPECIFICATION EXAMPLE

```
%option debug

%x comment

NAME    [[:alpha:]][_[:alnum:]]*

%%

"//".*          // ignore

"/*"            begin(comment);
```

```
<comment>.|\n   // ignore
<comment>"*/"   begin(INITIAL);

^a              return 1;
a               return 2;
a$              return 3;
{NAME}          return 4;

.|\n            // ignore


)
```

7. THE CLASS INTERFACE: SCANNER.H

By default, **flexc++** generates a file *Scanner.h* containing the initial interface of the scanner class performing the lexical scan according to the specifications given in **flexc++**'s input file. The name of the file that is generated can easily be changed using **flexc++**'s *--class-header* option. In this man-page we'll stick to using the default name.

The file *Scanner.h* is generated only once, unless an explicit request is made to rewrite it (using **flexc++**'s *--force-class-header* option).

The provided interface is very light-weight, primarily offering a link to the scanner's base class (see this manpage's sections 8.1 through 8.8).

Many of the facilities offered by the scanner class are inherited from the *ScannerBase* base class. Additional facilities offered by the *Scanner* class. are covered below.

7.1. NAMING CONVENTION

All symbols that are required by the generated scanner class end in two consecutive underscore characters (e.g., *executeAction__*). These names should not be redefined. As they are part of the *Scanner* and *ScannerBase* class their scope is immediately clear and confusion with identically named identifiers elsewhere is unlikely.

Some member functions do not use the underscore convention. These are the scanner class's constructors, or names that are similar or equal to names that have historically been used (e.g., *length*). Also, some functions are offered offering hooks into the implementation (like *preCode*). The latter category of function also have names that don't end in underscores.

7.2 CONSTRUCTORS

- **explicit Scanner(std::istream &in = std::cin, std::ostream &out = std::cout)** This constructor by default reads information from the standard input stream and writes to the standard output stream. When the *Scanner* object goes out of scope the input and output files are closed.

  With interactive scanners input stream switching or stacking is not available; switching output streams, however, is.

- **Scanner(std::string const &infile, std::string const &outfile)** This constructor opens the input and output streams whose file names were specified. When the *Scanner* object goes out of scope the input and output files are closed. If *outfile* == "-" then the standard output stream is used as the scanner's output medium; if *outfile* == "" then the standard error stream is used as the scanner's output medium.

  This constructor is not available with interactive scanners.

7.3. PUBLIC MEMBER FUNCTIONS

- **int lex()** The *lex* function performs the lexical scanning of the input file specified at construction time (but also see section 6.1. for information about intermediate stream-switching facilities). It returns an *int* representing the *token* associated with the matched regular expression. The returned value 0 indicates end-of-file. Considering its default implementation, it could be redefined by the user. *Lex*'s default implementation merely calls *lex__*:

```
inline int Scanner::lex()
{
    return lex__();
}
```

  Caveat: with interactive scanners the *lex* function is defined in the generated *lex.cc* file. Once **flexc++** has generated the scanner class header file this scanner class header file isn't automatically rewritten by **flexc++**. If, at some later stage, an interactive scanner must be generated, then the inline *lex* implementation must be removed `by hand' from the scanner class header file. Likewise, a *lex* member implementation (like the above) must be provided `by hand' if a non-interactive scanner is required after first having generated files implementing an interactive scanner.

7.4. PRIVATE MEMBER FUNCTIONS

- **int lex__()** This function is used internally by *lex* and should not otherwise be used.
- **int executeAction__()** This function is used internally by *lex* and should not otherwise be used.
- **void preCode()** By default this function has an empty, inline implementation in *Scanner.h*. It can safely be replaced by a user-defined implementation. This function is called by *lex__*, just before it starts to match input characters against its rules: *preCode* is called by *lex__* when *lex__* is called and also after having executed the actions of a rule which did not execute a *return* statement. The outline of *lex__*'s implementation looks like this:

```
int Scanner::lex__()
{
    ...
    preCode();

    while (true)
    {
        size_t ch = get__();           // fetch next char
        ...
        switch (actionType__(range))   // determine the action
        {
            ... maybe return
```

```
            }
            ... no return, continue scanning
            preCode();
        } // while
    }
```

- **void print()** When the *--print-tokens* or *%print-tokens* directive is used this function is called to display, on the standard output stream, the tokens returned and text matched by the scanner generated by **flexc++**.

  Displaying is suppressed when the *lex.cc* file is (re)generated without using this directive. The function actually showing the tokens (*ScannerBase::print__*) is called from *print*, which is defined in-line in *Scanner.h*. Calling *ScannerBase::print__*, therefore, can also easily be controlled by an option controlled by the program using the scanner object.

7.5. SCANNER CLASS HEADER EXAMPLE

```
#ifndef Scanner_H_INCLUDED_
#define Scanner_H_INCLUDED_

// $insert baseclass_h
#include "Scannerbase.h"


class Scanner: public ScannerBase
{
    public:
        explicit Scanner(std::istream &in = std::cin,
                         std::ostream &out = std::cout);

        Scanner(std::string const &infile, std::string const &outfile);

        // $insert lexFunctionDecl
        int lex();

    private:
        int lex__();
        int executeAction__(size_t ruleNr);

        void preCode(); // re-implement this function for code to be
                        // exec'ed before the pattern matching starts
};

inline void Scanner::preCode()
{
    // optionally replace by your own code
}

inline Scanner::Scanner(std::istream &in, std::ostream &out)
:
    ScannerBase(in, out)
{}

inline Scanner::Scanner(std::string const &infile,
                                            std::string const &outfile)
:
    ScannerBase(infile, outfile)
```

```
{}

// $insert inlineLexFunction
inline int Scanner::lex()
{
    return lex__();
}

#endif // Scanner_H_INCLUDED_
```

8.1. THE SCANNER BASE CLASS

By default, **flexc++** generates a file *Scannerbase.h* containing the interface of the base class of the scanner class also generated by **flexc++**. The name of the file that is generated can easily be changed using **flexc++**'s *--baseclass-header* option. In this man-page we use the default name.

The file *Scannerbase.h* is generated at each new **flexc++** run. It contains no user-serviceable or extensible parts. Rewriting can be prevented by specifying **flexc++**'s *--no-baseclass-header* option).

8.2. PUBLIC ENUMS AND -TYPES

- **enum class StartCondition__** This strongly typed enumeration defines the names of the start conditions (i.e., mini scanners). It at least contains *INITIAL*, but when the *%s* or *%x* directives were used it also contains the identifiers of the mini scanners declared by these directives. Since *StartCondition__* is a strongly typed enum its values must be preceded by its enum name. E.g.,

```
begin(StartCondition__::INITIAL);
```

8.3. PROTECTED ENUMS AND -TYPES

- **enum class ActionType__** This strongly typed enumeration is for internal use only.
- **enum Leave__** This enumeration is for internal use only.

8.4. NO PUBLIC CONSTRUCTORS

There are no public constructors. *ScannerBase* is a base class for the *Scanner* class generated by **flexc++**. *ScannerBase* only offers protected constructors.

8.5. PUBLIC MEMBER FUNCTIONS

- **bool debug() const** returns *true* if *--debug* or *%debug* was specified, otherwise *false*.
- **bool interactiveLine()** this member is only available with interactive scanners. All remaining contents of the current interactive line buffer is discarded, and the interactive line buffer is filled with the contents of the next input line. This member can be used when a condition is encountered which invalidates the remaining contents of a line. Following a call to *interactiveLine*

the next token that is returned by the lexical scanner will be the first token on the next line. This member returns *true* if the next line is available and *false* otherwise.

- **std::string const &filename() const** returns the name of the file currently processed by the scanner object.
- **size_t length() const** returns the length of the text that was matched by *lex*. With **flex**++ this function was called *leng*.
- **size_t lineNr() const** returns the line number of the currently scanned line. This function is always available (note: **flex**++ only offered a similar function (called *lineno*) after using the *%lineno* option).
- **std::string const &matched() const** returns the text matched by *lex* (note: **flex**++ offers a similar member called *YYText*).
- **void setDebug(bool onOff)** Switches on/off debugging output by providing the argument *true* or *false*. Switching on debugging output only has visible effects if the *debug* option was specified.

- **void switchIstream(std::string const &infilename)** The currently processed input stream is closed, and processing continues at the stream whose name is specified as the function's argument. This is *not* a stack-operation: after processing *infilename* processing does not return to the original stream.

  This member is not available with interactive scanners.

- **void switchOstream(std::ostream &out)** The currently processed output stream is closed, and new output is written to *out*.

- **void switchOstream(std::string const &outfilename)**

  The current output stream is closed, and output is written to *outfilename*. If this file already exists, it is rewritten.

- **void switchStreams(std::istream &in, std::ostream &out = std::cout)** The currently processed input and output streams are closed, and processing continues at *in*, writing output to *out*. This is *not* a stack-operation: after processing *in* processing does not return to the original stream.

  This member is not available with interactive scanners.

- **void switchStreams(std::string const &infilename, std::string const &outfilename)** The currently processed input and output streams are closed, and processing continues at the stream whose name is specified as the function's first argument, writing output to the file whose name is specified as the function's second argument. This latter file is rewritten. This is *not* a stack-operation: after processing *infilename* processing does not return to the original stream. If *outfilename == "-"* then the standard output stream is used as the scanner's output medium; if *outfilename == ""* then the standard error stream is used as the scanner's output medium.

  If *outfilename == "-"* then the standard output stream is used as the scanner's output medium; if *outfilename == ""* then the standard error stream is used as the scanner's output medium.

  This member is not available with interactive scanners.

8.6. PROTECTED CONSTRUCTORS

- **ScannerBase(std::string const &infilename, std::string const &outfilename)** The scanner object opens and reads *infilename* and opens (rewrites) and writes *outfilename*. It is called from the corresponding *Scanner* constructor.

  This member is not available for interactive scanners.

- **ScannerBase(std::istream &in, std::ostream &out)** The *in* and *out* parameters are, respectively, the derived class constructor's input stream and output streams.

8.7. PROTECTED MEMBER FUNCTIONS

All member functions ending in two underscore characters are for internal use only and should not be called by user-defined members of the *Scanner* class.

The following members, however, can safely be called by members of the generated *Scanner* class:

- **void accept(size_t nChars = 0)** *accept(n)* returns all but the first `nChars' characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match. So, it matches `nChars' of the characters in the input buffer, rescanning the rest. This function effectively sets *length*'s return value to *nChars* (note: with **flex**++ this function was called *less*);

- **void begin(StartCondition__ startCondition)** activate the regular expression rules associated with *StartCondition__ startCondition*. As this enumeration is a strongly typed enum the *StartCondition__* scope must be specified as well. E.g.,

```
begin(StartCondition__::INITIAL);
```

- **void echo() const** The currently matched text (i.e., the text returned by the member *matched*) is inserted into the scanner object's output stream;

- **void leave(int retValue)** actions defined in the lexical scanner specification file may or may not return. This frequently results in complicated or overlong compound statements, blurring the readability of the specification file. By encapsulating the actions in a member function readability is enhanced. However, frequently a compound statement is still required, as in:

```
regex-to-match  {
                    if (int ret = memberFunction())
                        return ret;
                }
```

  The member *leave* removes the need for constructions like the above. The member *leave* can be called from within member functions encapsulating actions performed when a regular expression has been matched. It ends *lex*, returning *retValue* to its caller. The above rule can now be written like this:

```
regex-to-match  memberFunction();
```

and *memberFunction* could be implemented as follows:

```
void memberFunction()
{
    if (someCondition())
    {                           // any action, e.g.,
                                // switch mini-scanner
        begin(StartCondition__::INITIAL);

        leave(Parser::TOKENVALUE);    // lex returns TOKENVALUE
        // this point is never reached
    }

    pushStream(d_matched);      // switch to the next stream
                                // lex continues
}
```

The member *leave* should only (indirectly) be called (usually nested) from actions defined in the scanner's specification **s**; calling *leave* outside of this context results in undefined behavior.

- **void more()** the matched text is kept and will be prefixed to the text that is matched at the next lexical scan;
- **std::ostream &out()** returns a reference to the scanner's output stream;
- **bool popStream()** closes the currently processed input stream and continues to process the most recently stacked input stream (removing it from the stack of streams). If this switch was successfully performed *true* is returned, otherwise (e.g., when the stream stack is empty) *false* is returned;
- **void push(size_t ch)** character *ch* is pushed back onto the input stream. I.e., it will be the character that is retrieved at the next attempt to obtain a character from the input stream;
- **void push(std::string const &txt)** the characters in the string *txt* are pushed back onto the input stream. I.e., they will be the characters that are retrieved at the next attempt to obtain characters from the input stream. The characters in *txt* are retrieved from the first character to the last. So if *txt == "hello"* then the *'h'* will be the character that's retrieved next, followed by *'e'*, etc, until *'o'*;
- **void pushStream(std::istream &curStream)** this function pushes *curStream* on the stream stack;

  This member is not available with interactive scanners.
- **void pushStream(std::string const &curName)** same, but the stream *curName* is opened first, and the resulting *istream* is pushed on the stream stack;

  This member is not available with interactive scanners.
- **void redo(size_t nChars = 0)** this member acts like *accept* but its argument counts backward from the end of the matched text. All but these *nChars* characters are kept and the last *nChar* characters are rescanned. This function effectively reduces *length*'s return value by *nChars*;
- **void setFilename(std::string const &name)** this function sets the name of the stream returned by

filename to *name*;

- **void setMatched(std::string const &text)** this function stores *text* in the matched text buffer. Following a call to this function *matched* returns *text*.

- **StartCondition__ startCondition() const** returns the currently active start condition (mini scanner);

- **std::vector<StreamStruct> const &streamStack() const** returns the vector of currently stacked input streams. The vector's size equals 0 unless *pushStream* has been used. So **flexc++**'s input file is not counted here. The *StreamStruct* is a *struct* only having one accessible member: *std::string const &pushedName*, which holds the name of the pushed stream. The vector is used internally as a stack: the stream that was first pushed is found at index position 0, the most recently pushed stream is found at *streamStack().back()*.

  This member is not available with interactive scanners.

8.8. PROTECTED DATA MEMBERS

All protected data members are for internal use only, allowing *lex__* to access them. All of them end in two underscore characters.

8.9. FLEX++ TO FLEXC++ MEMBERS

Flex++ (old)	Flexc++ (new)
lineno()	*lineNr()*
YYText()	*matched()*
less()	*accept()*

9.1 THE CLASS INPUT

Flexc++ generates a file *Scannerbase.h* defining the scanner class's base class, by default named *ScannerBase* (which is the name used in this man-page). The base class *ScannerBase* contains a nested class *Input* whose interface looks like this:

```
class Input
{
    public:
        Input();
        Input(std::istream *iStream, size_t lineNr = 1);
        size_t get();
        size_t lineNr() const;
        void reRead(size_t ch);
        void reRead(std::string const &str, size_t fmIdx);
        void close();
```

```
};
```

The members of this class are all required and offer a level in between the operations of *ScannerBase* and **flexc++**'s actual input file that's being processed.

By default, **flexc++** provides an implementation for all of *Input*'s required members. Therefore, in most situations this man-page can safely be ignored.

However, users may define and extend their own *Input* class and provide **flexc++**'s base class with that *Input* class. To do so **flexc++**'s rules file must contain the following two directives:

```
%input-implementation = "sourcefile"
%input-interface = "interface"
```

Here, *interface* is the name of a file containing the class *Input*'s interface. This interface is then inserted into *ScannerBase*'s interface instead of the default class *Input*'s interface. This interface must *at least* offer the above-mentioned members and constructors (their functions are described below). The class may contain additional members if required by the user-defined implementation. The implementation itself is expected in *sourcefile*. The contents of this file are inserted in the generated *lex.cc* file instead of *Input*'s default implementation. The file *sourcefile* should probably not have a *.cc* extension to prevent its compilation by a program maintenance utility.

When the lexical scanner generated by **flexc++** switches streams using the *//include* directive (see section **6.1. FILE SWITCHING**) the input stream that's currently processed is pushed on an *Input* stack maintained by *ScannerBase*, and processing continues at the file named at the *//include* directive. Once the latter file has been processed, the previously pushed stream is popped off the stack, and processing of the popped stream continues. This implies that *Input* objects must be `stack-able'. The required interface is designed to satisfy this requirement.

9.2. CONSTRUCTORS

- **Input()** The default constructor is used by **ScannerBase** to prepare the stack for *Input* objects. It must make sure that a default (empty) *Input* object is in a valid state and can be destroyed. It serves no further purpose. *Input* objects, however, must support the default (or overloaded) assignment operator.
- **Input(std::istream *iStream, size_t lineNr = 1)** This constructor receives a pointer to a dynamically allocated *istream* object. The *Input* constructor should preserve this pointer when the *Input* object is pushed on and popped off the stack. A *shared_ptr* probably comes in handy here. The *Input* object becomes the owner of the *istream* object, albeit that its destructor is *not* supposed to destroy the *istream* object. Destruction remains the responsibility of the *ScannerBase* object, which calls the *Input::close* member (see below) when it's time to destroy (close) the stream.

  The new input stream's line counter is set to *lineNr*, by default 1.

9.3. REQUIRED PUBLIC MEMBER FUNCTIONS

- **size_t get()** returns the next character to be processed by the lexical scanner. Usually it will be the next character from the *istream* passed to the *Input* class at construction time. It is never called by

the *ScannerBase* object for *Input* objects defined using *Input*'s default constructor. It should return 0x100 once *istream*'s end-of-file has been reached.
- **size_t lineNr() const** should return the (1-based) number of the *istream* object passed to the *Input* object. At construction time the *istream* has just been opened and so at that point *lineNr* should return 1.
- **void reRead(size_t ch)** if provided with a value smaller than 0x100 *ch* should be pushed back onto the *istream*, where it becomes the character next to be returned. Physically the character doesn't have to be pushed back. The default implementation uses a *deque* onto which the character is pushed-front. Only when this *deque* is exhausted characters are retrieved from the *Input* object's *istream*.
- **void reRead(std::string const &str, size_t fmIdx)** the characters in *str* from *fmIdx* until the string's final character are pushed back onto the *istream* object so that the string's first character is retrieved first and the string's last character is retrieved last.
- **void close()** the *istream* object initially passed to the *Input* object is deleted by *close*, thereby not only freeing the stream's memory, but also closing the stream if the stream in fact was an *ifstream*. Note that the *Input*'s destructor should *not* destroy the *Input*'s *istream* object.

FILES

Flexc++'s default skeleton files are in */usr/share/flexc++*.
By default, **flexc++** generates the following files:

- *Scanner.h*: the header file containing the scanner class's interface.
- *Scannerbase.h*: the header file containing the interface of the scanner class's base class.
- *Scanner.ih*: the internal header file that is meant to be included by the scanner class's source files (e.g., it is included by *lex.cc*, see the next item's file), and that should contain all declarations required for compiling the scanner class's sources.
- *lex.cc*: the source file implementing the scanner class member function *lex* (and support functions), performing the lexical scan.

SEE ALSO

bisonc++(1)

BUGS

- The priority of interval expressions (*{...}*) equals the priority of other multiplicative operators (like *).
- All *INITIAL* rules apply to inclusive mini scanners, also those *INITIAL* rules that were explicitly associated with the *INITIAL* mini scanner.
- Generating interactive and non-interactive scanners (see section 5. INTERACTIVE SCANNERS) cannot be mixed.

ABOUT flexc++

Flexc++ was originally started as a programming project by Jean-Paul van Oosten and Richard Berendsen in the 2007-2008 academic year. After graduating, Richard left the project and moved to

Amsterdam. Jean-Paul remained in Groningen, and after on-and-off activities on the project, in close cooperation with Frank B. Brokken, Frank undertook a rewrite of the project's code around 2010. During the development of **flexc++**, the lookahead-operator handling continuously threatened the completion of the project. By now, the project has evolved to a level that we feel it's defensible to publish the program, although we still tend to consider the program in its experimental stage; it will remain that way until we decide to move its version from the 0.9x.xx series to the 1.xx.xx series.

COPYRIGHT

AUTHOR

Frank B. Brokken (**f.b.brokken@rug.nl**),
Jean-Paul van Oosten (**j.p.van.oosten@rug.nl**),
Richard Berendsen (**richardberendsen@xs4all.nl**) (until 2010).