CppCC user's guide*

by Alec Panovici (alecutzu@fx.ro)

Last updated on 1st February 2003.

*http://sourceforge.net/projects/cppcc

Contents

1 Introduction

1.1 What is CppCC?

CppCC is aimed as a replacement to LEX&YACC for those who wish to write pure C++ parsers and preffer an object-oriented interface to global variables and C++ wrappers over bare C. The main differences are the more human (in my opinion[1]) syntax and a real object-oriented API towards the user's code.

CppCC generates LL(k) recursive-descendant parsers and DFA-based scanners.

1.2 About this document

The master version of this document was created with LyX(http://www.lyx.org). It can be downloaded (both the original and other formats) from http://cppcc.sourceforge.net.

This document reffers to version 0.0.5 of CppCC.

2 How it works

CppCC is not very different from other similar tools in the way it works. A grammar file has to be written, and as the result of running the tool providing the grammar file as its input[2] , a set of C++ source files will be obtained (See Fig. 1). These sources will contain the scanner and the parser that accept as input the language specified in the grammar file. More precisely, three pairs of header/definition source files will be created, one for the scanner's class, one for the parser's class and one for the token class.

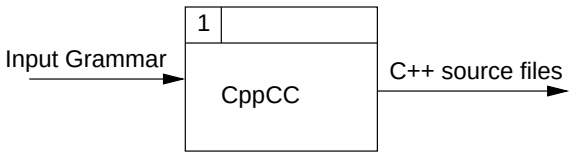

Figure 1: Data flow through CppCC.

3 The CppCC input grammar file

The input of CppCC consist of a single file containing a description of the grammar for which it will generate the scanner and the parser[3]. Hence, the grammar file will contain at least two section, one which will describe the scanner's accepted language and one describing the parser's. Additionally, one can provide a number of options that will affect the code generation process and a "token customization section". Each of these sections is introduced by a keyword at the "top level" of the grammar file. Their syntax was tried to be kept as C++-like as possible, in order to provide the user with a good approximation of what CppCC will generate out of it.

To be more precise, the grammar file BNF production is:

```
<grammar_file> -> [ <options_section> ]
                  <token_customization_section>
                  <lexical_section> <syntax_section>
```

The options section applies to the overall CppCC operation, while the other three sections refer to a specific part of the code to be generated. As an example, the token customization section whose syntax is:

```
<token_customization_section> -> [ <c_block> ] "TOKEN" <c_token_class_id>
                                 [ <inheritance> ] [ <c_block> ]
```

[1] By the time I'm writing this, I have already played with it a bit (the examples subdir of the distribution contains the results of that) and found it quite pleasant to use.

[2] See Annex C for the command line syntax.

[3] The full lexical and syntactic specifications are provided in Annex B.

lets you specify the name of the class to be generated, and also make it inherit from one or more classes, if the project into which the parser will be integrated requires so (as an example, this feature can be used to switch from a handwritten parser to a CppCC generated one in an existing project where it is very desirable to preserve the interface between the parser and the rest of the code. This can be easily achieved by making the generated class(es) inherit the old interfaces). Moreover, within each section, handwritten methods and field declarations that will be merged into the final generated source can be freely added (continuing the parser replacement example, after making the new parser inherit the old one's interface, one can then implement its methods in terms of the ones provided by the generated parser. The easiest way is to place these wrapper methods inside the parser's section itself).

Each of the token customization, lexical and syntax sections can optionally be preceded by a block of C++ code. This code will also be placed before the code that is generated from the sections' contents itself, and therefore it is a good place for adding `#include` lines for headers that will be used by the user code inside the section, typedefs, etc.

3.1 The options section

This section contains various switches that control the code generation algorithm. Options are specified as *key = value* pairs. The syntax of this section is:

```
<options_section> -> "OPTIONS" "{" ( <option_name> "=" <option_value> ) * "}"
```

The possible options are:

DEBUG_PARSER tells CppCC to add debugging code into the parser source. The default value is *"false*". When enabled, the generated parser class will contain extra methods for tracing its functioning at runtime;

DEBUG_SCANNER same as DEBUG_PARSER, but affects the scanner's class;

CASE_SENSITIVE cause CppCC to add code into the scanner's class for dealing with case-insensitive languages. The default value is *"true"*. If set to *"false"*, the generated scanner will make no difference between upper and lower case letters.

USE_EXCEPTIONS if set to *"true"*, the generated code will use the exceptions mechanism for error handling (See Sec. 3.5). If not, no exception-related code will be generated. Defaults to *"true"*.

DEFAULT_LOOKAHEAD specifies how many tokens the parser should inspect when making choices of what expansions it should chose next, when just the current token is not enough for making such choices. The default value is 1, which makes the parser be pure LL(1)[4].

TOKENS_SPAN_EOF specifies what should the scanner do when a file ends before a token was completely matched. If this option is false, a scan exception will occur, otherwise the scanner will attempt to silently switch to another input stream and continue scanning. The default value of this option is false.

COUNT_COLUMNS tells CppCC to generate extra code in the scanner in order to also keep track of the starting/ending column for each token. The default value is *"true"*.

SHARP_LINES tells CppCC whether to insert `#line` directives into the generated code. The directives will indicate the compiler/debugger the places where the user's code was inserted into the input file. The default value is *"true"*.

PROFILING_FILE specifies the name of a profile file to be used when generating the scanner (See Annex A).

STRING_CLASS specifies a class to be used instead of std::string for representing the token images (See Sec. 4.1.1)

NAMESPACE_NAME specify an altrernate name for the namespace enclosing the cppcc generated class declarations (See Sec. 4)

SRC_EXTENSION, HDR_EXTENSION allows use of alternate extensions when writing the generated source files (See Sec. 4).

3.1.0.0.1 Example: If we need a parser that will treat upper/lower case letters the same and we don't want to deal with the LOOKAHEAD issues, the OPTIONS section would be the one shown in Fig. 2.

By setting the default lookahead to a very large value (we used the machine's INT_MAX, but any fairly large value is just as good), we ensured that the generated parser will always try out a whole expansion when not sure about what to do next (not accidentally we used `INT_MAX`, as it is also the largest lookahead value the parser could use anyway).

[4]Note that this option *only* affects non-LL(1) choices, which are automatically detected by CppCC. In all the others situations, a single token is inspected, no matter how large the default lookahead was specified. Also, if a local lookahead hint is specified at non-LL(1) choice points, the value of this option is disregarded and a warning is issued.

```
OPTIONS
{
  DEFAULT_LOOKAHEAD = 2147483647; // inspect up to this many tokens when needed...
  CASE_SENSITIVE = false; //and ignore case.
}
```

Figure 2: Sample options section.

3.2 The token customization section

The token customization section allows you to specify the token class' name, inheritance and add some C++ code to be included into the generated class.

Its syntax is:

```
<token_customization_section> -> [ <c_block> ]
                                 "TOKEN" <c_token_class_id> [ <inheritance> ]
                                 <c_block>
```

As it can be noticed above, at least the token's class name followed by an (eventually empty) C block must be present.

The code block that precedes the `TOKEN` keyword, if present will be inserted verbatim (without the enclosing braces) into the scanner's class' header file. It is a good place for placing `#include` lines or other fragments of code that need to appear before any other kind of user code in the generated sources.

If the inheritance list is present, then each of the parent classes must provide a default constructor (this is needed because the token's class constructors themselves will be generated by CppCC which doesn't know what constructors should be called for each of the parent classes. Thus, the simplest solution is to just let the C++ compiler generate calls to their default parameterless constructors).

The C++ block following the token's class name can contain any extra application-specific fields and methods. Because they are inlined into the token's class the user-defined methods will have access to all the token objects' internal fields. The following fields of the token's class are accessible (See Sec. 4.1 for details):

`int length;`
: The length of the token's image.

`cppcc::Position bPos, ePos:;`
: Beginning and end location of the token relative to the beginning of the input stream;

`int id;`
: the token's id.

The string image of the token can be obtained by calling the **`std::string& image ()`** method of the token class, which returns a reference to an internal field that contains the token's image. This field can be modified in any way by the user's code.

3.2.0.0.2 Example: Suppose an application containing a parser for some language *foo* requires a way of finding out whether a token spans multiple lines. An implementation using the user-code feature of the token customization section is given in Fig.3.

3.3 The lexical section

The lexical section contains the scanner's specification. As for the token customization section, a name for the generated class is always required. The syntax is:

```
<lexical_section> -> [ <c_block> ]
                     "SCANNER" <c_scanner_class_id> [ <inheritance> ]
                     "{" <scanner_decl> * "}"
```

```
TOKEN FooToken //FooToken is the name of the generated token class
{ //this is the <c_block> for the token
  public: //remember, this code will be inserted as-is into the class' code,
          //anything that's legal inside a normal class is also ok here
  /**
   * Returns a boolean value indicating whether this token spans on multiple lines.
   */
  bool spanLines ()
  {
    return bPos.ln != ePos.ln; //bPos, ePos are fields that will be declared
                               //inside the final FooToken class
  }
}
```

Figure 3: Adding a custom method into a token section.

Just like the token customization section, a first block of code for placing `#include` or other C++ code that needs to be placed at the beginning of the generated file is provided.

If an inheritance is present in this section, each of the parent classes must provide a default constructor that will be called by the generated scanner class' constructors. Inside the body of the lexical section multiple `<scanner_decl>` items can appear. These can be either token declarations or user code:

```
<scanner_decl> -> <c_block>
                | [ <lexical_states_list> ] <token_kind> "{" <token_decl> * "}"
                | "SPECIAL" "{" <special_token_decl> * "}"
<token_kind> -> "TOKEN" | "SKIP" | "MORE" | "KEYWORD"
```

Token declarations specify the language that is accepted by the scanner. A CppCC generated scanner supports five kinds of tokens:

- regular tokens, which are created by the scanner and handed over to the parser
- *skip* tokens, which are recognized by the scanner but are ignored and not handed over to the parser
- *keyword* tokens are pretty simmilar to regular tokens, except that their image is ignored once they have been matched.
- *more* tokens are tokens that are treated as mere prefixes for longer tokens
- *special* tokens, which are treated as nonexistent as far as the scanning routine is concerned.

3.3.1 Non-special tokens

The *regular tokens* are those declared using the reserved keyword "TOKEN". Such tokens will be returned by successive calls to the scanning method, as they are recognized in the input stream.

Unlike the regular tokens, the "SKIP" tokens, although recognized as valid language constructs in the input stream, will never be returned to the parser, but simply ignored. This kind of token declaration is useful when the scanner should throw away some tokens such as comments and whitespaces. They are declared using the reserved word `SKIP`. Note that, however, any associated token actions are still executed (including the `commonTokenAction` hook, if defined)[5].

The "MORE" tokens are also never returned by the scanner's routine to the parser, but are nevertheless ignored. A token may be flagged as "MORE" when it is merely the prefix for a longer token. After it has recognized such a token, the scanner will continue to match characters until a non-MORE token is found, then return a single token having as its image the concatenation of the images of all the previously matched MORE tokens and the last token of some other kind. Of course the same result could have been achieved by using a single long regular expression that matched the

[5]The reason for this being so is that, although they are not important during the parsing process itself (and therefore they are never returned to the parser), a user may want for instance to remember the comments that appeared inside an input source (comments are the most common case where SKIP tokens are used), and then it can a user action that will store the comment token images somewhere for later use.

same concatenation, but using MORE tokens allow user code to be executed during the match (after each MORE token a user action can be placed).

The "KEYWORD" tokens are very similar to regular tokens, except that the scanner will avoid making the token's image available to the user code[6]. This can result in a performance improvement for the scanner.

The rest of the token declaration for these four kinds is the same:

```
<token_decl> -> "<#" <c_token_id> ":" <token_regexp> ">"
              | "<" <c_token_id> ":" <token_regexp> ">" [ <c_block> ]
```

All the tokens in the above category have an associated *lexical state set*, a *symbolic name*, a *regular expression* and an (optional) *user action*. Each of these will be discussed below.

Lexical states provide a simple mechanism for switching between sets of accepted input tokens while the scanner is running. Each such set of tokens is called a lexical state. While in a lexical state *x*, the scanner will only accept those tokens that had *x* inside their lexical state list. If a token is to be recognized in any lexical state, its lexical state list should only contain the special wildcard marker * which stands for "any state". If no states are specified for a token's declaration, then its state list will implicitly contain a single state called `START`. As the name implies, the `START` state is also the initial state of the scanner, and if no other states ever occur within the lexical section, it is the only lexical state of the scanner. The token declaration syntax allows a single lexical state list to apply to multiple tokens.

The *symbolic name* (or simply the name) of a token is the name by which the token will be known to the parser and user actions. It is specified by the `<c_token_id>` inside a token declaration. For each symbolic name, a unique integer constant will be created into the generated code. For each `Token` object returned by the scanning method, the `id` field (See. 4) will be set to the constant value that is associated to the token's symbolic name[7].

A special case is that of *macro tokens*, those whose symbolic names are preceded by a "#" sign, These are not real tokens, and cannot be used inside the syntactic section. Their sole purpose is to act as "regular expressions macro definitions". Such tokens' names can appear inside other regular expressions definitions of other tokens, and will be replaced by their own definition by CppCC (See Example 4).

The *regular expression* describes the class of strings that match the token. When the scanning routine will successfully match a string that conforms to a token's regular expression, it will return a new Token object with the corresponding `id`. The complete regular expression grammar can be found in Section B. The regular expression can be any of the following:

- a single character, like `'k'`; The scanner will match a `'k'` character from the input stream. The character can contain any escape sequence accepted by the C++ language, such as '\n';
- a character range, like `'k'-'y'`; The scanner will match any characters starting from `'k'` up to `'y'` (`'k'` and `'y'` included).
- a list of characters or character ranges, like `['a', 'c', 'e'-'g']`. The scanner will match an `'a'` or a `'c'` or any of `'e'`, `'f'` and `'g'`. The list must be enclosed between right brackets.
- a negated list of character or character ranges, like `~['\n', '\r']`. The scanner will match any characters that are not `'\n'` or `'\r'` (i.e. any character that is not a line terminator). The negation is indicated by the tilde sign preceding the list.
- a string, such as `"kay"`. The scanner will successively match a `'k'` then an `'a'` then an `'y'` on the input stream. Similar to characters, C++ escape sequences are allowed inside the strings, e.g `"\r\n"`.
- the name of another token (it can be a macro token), like `<digit>`. The scanner will match the regular expression that defines the `digit` token. Note that when a token symbolic name is referred, it must always be enclosed between angle braces - `"<"` and `">"`. Also, note that this expansion is done "textually", by inserting the regular expression associated with that name in the place where the name is used; any lexical states specified for the name of that token or macro token do not apply here.
- a regular expression postfixed with the `"+"` operator, like `['a'-'z', 'A'-'Z']+`. The scanner will match one or more occurrences of the regular expression to which the `"+"` operator is applied, in this case one or more lower case or upper case letters.

[6] The idea behind this is that keywords have as their image a fixed string representation which is specified by a language's grammar, and thus the image found in the input file would be redundant (an `IF` keyword can only have as its image the string `"if"`, aso).

[7] Therefore, it is correct to use the notions of a token's id and its name interchangeably, as they are biunivocally equivalent.

- a regular expression postfixed with the `"*"` operator, such as `['a'-z', 'A'-'Z']*`. The scanner will match zero or more occurrences of the regular expression that precedes the operator, in this case zero, one or more lower case or upper case letters.

- a regular expression postfixed with the `"?"` operator, such as `"maybe"?`. The scanner will match zero or one occurrence of the regular expression that precedes the operator, in this case zero or one occurrence of the string literal `"maybe"`.

- a concatenation of any number of regular expressions, such as `['a'-'z', 'A'-'Z']['a'-'z', 'A'-'Z', '_', '0'-'9']*`. The scanner will sequentially match each of the regular expression in the order that they appeared on the input stream; in this case, a C-style identifier is matched, i.e. one that starts with a letter, then continues with an undefined number of letters, digits or the `'_'` character.

- a choice between two or more regular expressions, such as `['\n', '\r'] | "\n\r" | "\r\n"`. The scanner will try to match one of the choices, in this case any flavour of line terminator.

- a group of regular expressions such as `("foo" | "bar")`. Regular expressions can be grouped using parentheses, `"("` and `")"`. Grouping is used for changing the operators' precedence, for instance the regular expression `("foo" | "bar")+` will match one or more occurrences of strings `"foo"` and `"bar"`.

3.3.1.0.3 Example: Suppose we have a simple language called *Foo* that only contains expressions. The sample lexical section given in Fig. 4 is a possible description of the scanner for such a language. It only accepts decimal numbers and identifiers as operands, and the usual arithmetic operators, and is able to recognize and skip C++-style one line comments.

The *token action* associated to a token is a block of C++ code that will get executed every time the scanner matches that token. This code has access to all the scanner class' internal data, including the currently matched token object, which can be modified by this code if so needed. The (eventually) modified token object will be returned as the result value of the current call of the scanning method. The code in a token action can use the following variables and functions:

`<TokenClass> *token;`
: The just matched token. The id, image and position fields of the token object are set by the scanner before entering the token action code and are not modified by it in any way after wards. Thus, any modifications of these fields will be preserved and seen from the parser.

`Position bPos, ePos;`
: The scanner's internal position fields. They have the same values as the token's fields with the same names.

Additionally to these fields, all the methods from the scanner's public interface are also accessible (see Sec. 4.2.1).

3.3.1.0.4 Example: Let's modify Example 3.3.1.0.3 so that each time a `DECIMAL_NUMBER` is matched on the input the scanner will also determine the value of the number. This value will be stored in a field we're going to add to the token class. The code is shown in Fig. 5.

3.3.2 Special tokens

If a particular application needs to reserve a number of tokens for user-defined purposes, then these tokens can be declared as `SPECIAL`. No regular expressions are associated to them and they are basically treated as nonexistent by the token matching routine, their creation must be done by the user code. However, such tokens can be freely used inside the syntactic section just as normal tokens. This kind of tokens was introduced to facilitate the implementation of "semantic tokens". A simple way to implement this mechanism with CppCC is to declare the various semantic tokens as `SPECIAL` and add user actions that will create them when needed. For instance, consider a language where we have to distinguish between function names and variable names. From the scanner's perspective, both of these are just identifiers, that can be matched by a regular expression like `letter (letter | digit)*`. This is what the scanning method would normally return. But we can associate a user action that, each time a `name` token is matched, will perform a symbol table lookup and find the symbol with the given name, then change the token's `id` from `name` to say `function_name` or `variable_name`. The so-modified token object will then be returned to the parser.

```
/**
 * Scanner specification for language Foo.
 */
SCANNER FooScanner
{
  TOKEN {
    // we define some some macro tokens first, top make things easier to
    // read later
    <#LOCASE_LETTER: ['a'-'z'] >
    <#UPCASE_LETTER: ['A'-'Z'] >
    <#DECIMAL_DIGIT: ['0'-'9'] >
    <#LETTER: <LOCASE_LETTER> | <UPCASE_LETTER> >
    <#LETTER_OR_DIGIT: <LETTER> | <DIGIT> >

    // and now the real tokens:

    // a decimal number is defined as a sequence of digits (at least one)
    <DECIMAL_NUMBER: <DECIMAL_DIGIT>+ > // the resulting regexp will be:
                                        // ['0'-'9']+

    // an identifier always starts with a letter, followed by any number
    // (possibly none) of letters or digits
    <IDENTIFIER: <LETTER> <LETTER_OR_DIGIT>* >

    <ADD_OP: ['+', '-'] >
    <MUL_OP: ['*', '/'] >
    //and so on....
  }

  SKIP {

    // the scanner will ignore whitespaces, end-of-lines and C++ style
    // one-line comments
    <WS: [' ', '\t'] >
    <EOL: '\n' | '\r' | "\n\r" | "\r\n" >
    <ONE_LINER: "//" ~['\n', '\r']* > //this regexp will match everything
                                      //starting from a "//" up to the
                                      //end of the current line
  }
}
```

Figure 4: A simple lexical section.

```
TOKEN FooToken
{
public:
  int intValue;
}
{ // for atoi:
#include <cstdlib>
}
SCANNER FooScanner
{
  // all definition except DECIMAL_NUMBER are the same as in Fig. 4

  // we modify the DECIMAL_NUMBER by adding a token action
  <DECIMAL_NUMBER: <DECIMAL_DIGIT>+ > { token.intValue = atoi(token->image().c_str()); }
}
```

Figure 5: Token action that computes a token's semantic value.

3.3.2.0.5 Example: Supposing the language in Example 3.3.1.0.3 needs to differentiate between function and variable identifiers. This will require some user code that will do the symbol table lookup and change the `IDENTIFIER` tokens to either `FUNCTION_IDENTIFIER` or `VARIABLE_IDENTIFIER`. A possible way of doing this is shown in 6. The example code only shows what needs to be modified in the lexical specification shown in Fig. 4. The extra code consists of a user action associated to the IDENTIFIER token that does the replacements mentioned above, and an extra field that is added into the token's class to store its "semantic information", i.e. a reference to the symbol table entry that is associated with the identifier.

3.3.3 The user code

C++ code can be freely intermixed with token declarations. Any block of C++ code, which can contain any methods or fields[8] that is found inside the lexical section will be inserted into the generated scanner class. From within this code access to scanner's internal fields and methods is permitted, as from any normal member function.

Several methods have a special meaning if defined in a code item inside the scanner's section:

`void commonTokenAction (<TokenClassName> &token);`

This method, if defined, will be called after a token has been successfully matched on the input stream its fields have been set up, but before before it is returned to the parser[9]. The argument is the matched token and has as its type a reference to an instance of the class denoted by the token's class name as specified into the token customization section.

The `commonTokenAction` is also the last point where one can cause the scanner to throw away the current token by calling the `rejectToken` method. If such a call is made, the token will never be returned and scanning will continue as if this was a SKIP token.

`bool wrap();`

This method, if defined, is called each time the scanner encounters the "end of file" condition as it reads from the current input stream. An implementation of this method can switch to another input stream by calling the `switchToStream` method of the scanner with a newly created input stream object as argument (see Sec 4.2.1 for a complete list of stream switching methods). If a new input stream was opened, it must return a `true` value. If a true value is returned, the scanner will reset the stream location pointers and continue (i.e. the next token will be matched from the new input stream, having its start position at line 1, column 1). If not, depending on the current scanning state, one of the following actions is taken:

- if a token was matched, it is returned, and successive calls to the scanning method will return the special `<eof>` token.

[8] The only "restriction" is that the block must have balanced parentheses. CppCC relies on this to find out where the block ends.

[9] Note that, if a token action was also defined for that token, it is executed *before* the `commonTokenAction` method is called. Also, this method is only called once after the token has been matched, even if the token is returned more than once to the parser (this situation may occur during the parser's lookahead).

```
TOKEN FooToken
{
public:
  // for identifier tokens, we want the parser to also indicate the definition of
  // the symbol with a given identifier.
  const Symbol *sym;
}
SCANNER FooScanner
{
  TOKEN
  {
    // token definitions are the same as in Fig. 4

    // we need to modify the IDENTIFIER by adding a user action
    <IDENTIFIER: <LETTER> <LETTER_OR_DIGIT>* >
     { // ... and this is the user action

        // assume the SymTable.lookup method returns the entry for a symbol
        // whose identifier is the string passed as argument
        token->sym = SymTable.lookup(token->image());
        if (s.getKind() == Symbol::K_VAR) { //test the symbol's kind
          // we found a variable, adjust the token's id field accordingly
          token->id = FooScanner::TOK_VARIABLE_IDENTIFIER;
        } else {
          //the symbol is a function, make the token say so
          token->id = FooScanner::TOK_FUNCTION_IDENTIFIER;
        }
     }
  }

  SPECIAL_TOKEN // we need to tell the parser and scanner about the existence of
                // two extra tokens:
  {
    VARIABLE_IDENTIFIER FUNCTION_IDENTIFIER
  }
  SKIP
  {
     // same as in Fig. 4
  }
}
```

Figure 6: A lexer specification that makes use of special tokens to represent semantic information.

- if a token was not completely matched, a scanning error is signaled.
- if no token matched was started yet, an <eof> token is returned.

bool onScanError (const ScanException &ex);
See Section 3.5.3.

3.4 The syntax section

contains the parser's specification. The syntax is:

```
<syntax_section> -> [ <c_block> ]
                    "PARSER" <c_parser_class_id> [ <inheritance> ]
                    "{" <parser_decl> * "}"
```

As for the token and lexer classes, a first block of code and an inheritance can be specified for the parser class. If present the first block of code is merged inside the generated header file of the parser's class, before the parser's class itself.

If an inheritance is given, each parent class must have a parameterless constructor that will be called by the generated parser's class constructors. Inside a lexical section, two kinds of constructs can be added: *user code* and *EBNF productions*. We will first discuss the EBNF productions, then the user code.

The CppCC productions are written in a code-like fashion: the syntax for a production resembles to that of a C++ function. It has a return type, arguments, a name and a body. The name is actually the name of the described nonterminal, the one that usually appears on the left hand side in a BNF production. The body contains, among other things, the actual right hand of the production. CppCC will generate a method inside the parser's class for each production found in the input grammar. The method will have exactly the same signature as the one specified in its production within the grammar file. CppCC parsers have no default start symbol.The parsing process starts by calling one of these generated methods from somewhere inside the rest of your project's code, and ends when the called method returns[10]. The syntax for specifying productions is (For the complete syntax see Sec B):

```
<production> -> "(" <c_type_decl> ")" <c_nonterminal_id> "(" <c_formal_args_list> ")"
                    [ <throw_clause> ]
                "{" <c_block> <or_list> "}"
```

The body of a production is a mixture of BNF expressions and blocks of C++ code. The user-supplied C++ code will be inserted into the generated methods at the appropriate places so that it is executed each time the parsing algorithm has reached the point where the code appeared inside the EBNF specification.

A special case is a production that only contains the first block of code and no expansions. For these, CppCC will assume that the production's code will somehow match the nonterminal it is intended for (such "black box productions" can be used for writing fast routines that match comments or strings by accessing the input stream directly[11], or to implement error recovery rules - See 3.5).

3.4.1 BNF expansions

The grammar of BNF expressions (or *expansions*) accepted by CppCC allows the "|", "*","+" and concatenation operators. Precedence can be changed by grouping expansions with round parentheses "(" and ")". An expansion can be any of the following:

- A token, such as <IDENTIFIER>. The parser will match an identifier on the input token stream. The match consists of a call to the scanning routine and a check that the token returned as the result of the call is indeed an IDENTIFIER token.
- A nonterminal, like FACTOR(). The parser will match tokens according to the nonterminal's expansion.
- An expression postfixed with the * operator, like <IDENTIFIER>*. The parser will match zero, one, or more occurrences of the expression, in this case zero, one or more IDENTIFIER tokens.

[10] Because the generated parser is recursive-descendant, the called method will most probably call other methods in its turn as part of the parsing process.

[11] Actually, i think string matching will be faster using a regular expression than anything one could write by hand.

- An expression postfixed with the `?` operator, such as `FACTOR()?`. The parser will match zero or one occurrence of the expression, in this case zero or one occurrence of the nonterminal `FACTOR`.
- An expression postfixed with the `+` operator, like `FACTOR()+`. The parser will match one or more occurrences of the expression, in this case one or more `FACTOR`s.
- A concatenation of two or more expression, such as `FACTOR() <MUL_OP> FACTOR()`. The parser will successively match each of the expressions, in the order that they appeared in. In this case, first a `FACTOR` nonterminal will be matched, then a `MUL_OP` token, and then another `FACTOR`.
- A choice between expressions, such as `<MUL_OP> | <DIV_OP>`. The parser will match any (but only one) of the expressions. In this case, it will match either a `MUL_OP` token or a `DIV_OP` token.
- An expression surrounded by parentheses `"("` and `")"`, as in `( <MUL_OP> FACTOR() )*`. Grouping only affects the precedence, in this case forcing the `*` operator to apply to the whole expression contained between the parentheses. The parser will match any number of occurrences of the construction composed of a `MUL_OP` token followed by a `FACTOR`.

3.4.1.0.6 Example In Example 3.3.1.0.3 we introduced a language consisting of arithmetic expressions. Fig. 7 contains the syntax section for that language. It corresponds to the lexical section given in Fig. 4. The parser accepts as its input a (possibly zero) number of expressions. Not that the `EXPRESSION`, `TERM` and `FACTOR` all return a value of type int, which is not used in any way within this example. After we will introduce the user actions later in this section, this example will be enhanced to make use of the return values.

3.4.2 User actions

The parser in Example 7 is a bit useless. It only recognizes the syntactic constructs but does not do anything with them, For that, one must add pieces of code at various points that will get executed along with the parsing process, the *user actions*. Code can be added before and/or after each BNF expansion. The code that is added before an expression will get executed before the parsing algorithm selects that expression for expansion (if the expression is not selected at all, the code will not be executed; this can happen if the expression is an alternative inside a choice expression). The code added after an expression will be executed after the expression was expanded successfully. Inside a block of user code, a variable called token is guaranteed to point to the most recently matched token, and can be used in any suitable way. Note that this pointer will change its value as soon as a new token is retrieved from the scanner, and therefore saving a token must be done by creating a copy of the token opbject the variable points to, not by just saving the pointer. Also, it is most certainly a bad idea to `delete` the token object from a user action. As an example, the EBNF construction

```
<IDENTIFIER> { cout << "Found an identifier at line " << token->bPos.ln << endl; }
```

will print something like "`Found an identifier at line 10`", if the file given as input to the parser had an identifier on line 10.

The productions' syntax allows a special block of code to be placed at the very start of its body. This block is guaranteed to be also inserted at the very start of the generated method and have the same declarative context as the method itself, thus being is a good place to add declarations of local variables that will be used later inside the user actions.

The semantic value of a production is also computed by user actions. At any point inside a user action, a value of an appropriate type (i.e. compatible with the declared return type of the production's method) can be returned with a plain C++ return statement as the semantic value of that production. Note that, because the user code is inserted textually inside the method's body, the return statement will be executed as such, causing the method to terminate and return that value with no further actions. For instance, an EBNF construction like

```
<IDENTIFIER> {return something; } <IDENTIFIER>
```

will most probably not produce the expected result, i.e. the parser will never attempt to match two `IDENTIFIER`s but only the first one, after which the return statement is taken. It is therefore advisable to place the return statement(s) only inside the user code that gets executed after the whole right hand side of the production was expanded (unless, of course, some particular reasons impose a different approach - one example could be that if at some point the user decides it is useless to continue parsing at at the current level, it can always interrupt it by adding a return statement with some failure code).

Retrieving the semantic value of a nonterminal is possible by using a special syntax inside the EBNF expression. Its form is `<variable> = NONTERMINAL(...)`, eg. `x = FACTOR()`. This indicates that the return value of the method that parses a `FACTOR` will be stored inside the variable `x`.

```
/**
 * Parser specification for language Foo, Version 1.
 */
PARSER FooParser // FooParser is the name of the generated parser class
{
  /**
   * This is our starting symbol. It doesn't matter where it appears inside
   * the syntax section, but it makes things easier to read if we write
   * the grammar in top-down fashion.
   */
  (void) INPUT_FILE ()
  {
    EXPRESSION()*
  }

  /**
   * An expression is a sequence of terms with + or - between them.
   */
  (int) EXPRESSION ()
  {
    TERM() ( <ADD_OP> TERM() )*
  }

  /**
   * A term is a sequence of factors with * or / between them
   */
  (int) TERM ()
  {
    FACTOR() ( <MUL_OP> FACTOR() )*
  }

  /**
   * A factor is either a number, a variable name or another expression enclosed between
   * parentheses.
   */
  (int) FACTOR ()
  {
    <DECIMAL_NUMBER> | ( <LPAREN> EXPRESSION() <RPAREN> )
  }
}
```

Figure 7: A simple syntax section

3.4.2.0.7 Example We will enhance the parser in Fig. 7 with user actions that will evaluate each expression and print its value to the standard output. For this, we assume the scanner stores the value of a `DECIMAL_NUMBER` inside the token as explained in Example 3.3.1.0.4. The syntax section of the enhanced parser is shown in Fig. 8.

```
/**
 * Parser specification for language Foo, Version 2.
 */
PARSER FooParser
{
  /**
   * INPUT_FILE will now print the value of each expression it finds. Note that the user
   * action is associated to EXPRESSION, not to EXPRESSION*, and therefore will be
   * correctly be executed after each expression.
   */
  () INPUT_FILE ()
  {
    { int x; } //this is the first block; add local declarations here

    x = EXPRESSION() {cout << "Expression's value is " << x << endl; } *
  }

  (int) EXPRESSION ()
  {
    { int r, tmp, sign; }
    /* Here we have to either add or substract, depending on what the ADD_OP token is.
       For this, we assign the sign variable with either 1 or -1, depending on whether
       the <ADD_OP>'s token image was "+" or "-". Note that the comparison was done
       assuming that the image field is a string (OWN_STRING option true). Otherwise,
       we should have probably used strncmp.
    */
    r = TERM()
    ( <ADD_OP> {sign = token->image() == "+" ? 1 : -1; } tmp = TERM() {r += sign * tmp} )*
    { return r; } // This is the semantic value of our EXPRESSION
  }

  (int) TERM ()
  {
    { int r, tmp; bool mul; }
    r = FACTOR()
    ( <MUL_OP> {mul = token->image() == "*"} tmp = FACTOR() {r = mul ? r*t : r/t} )*
    { return r; }
  }

  (int) FACTOR ()
  {
    { int r; }
      <DECIMAL_NUMBER> { return token->intValue; } // it's ok to return here
    | ( <LPAREN> r = EXPRESSION() <RPAREN> ) { return r; }
  }
}
```

Figure 8: A parser with user actions that evaluate expressions.

3.4.3 Local lookahead

In Section 1 we said that the parsers generated by CppCC are capable of handling LL(k) grammars. However, a LL(k > 1) parser is notably slow. Moreover, usual grammars are designed so that they are "mostly" LL(1), with only small areas being LL(k > 1). There are two common solutions to overcome such situation: left-factoring the non-LL(1) production and increasing the lookahead depth. Although left factoring has the advantage that it converts a non-LL(1) grammar to LL(1), it introduces new productions which have no semantic significance, as they are only the common prefixes of those production that were subject to factorization. This usually complicates semantic information handling (such as parse-tree building).

3.4.3.0.8 Example: Let's consider two productions that have a common prefix, such as:

```
A -> αX
B -> αY,
```

where α has a length of at least two tokens. A grammar containing these two productions is not LL(1). If we apply left factoring, we obtain something like:

```
A       -> A_OR_B X
B       -> A_OR_B Y
A_OR_B -> α,
```

where A_OR_B is a newly introduced nonterminal that expands to the common part of A and B. Now, if we need to create the parse tree based on the new grammar, we have to always add a new node for A_OR_B. The problems arise when we do not know exactly the semantics of an A_OR_B until we can decide whether it is part of an A or a B. In this case, we have to somehow store the date in a neutral way and postpone its interpretation until we can decide what it actually represents. This will most probably result in rather obscure user actions that will be difficult to read and maintain.

For such situations, where breaking down a production into two or more parts is not desirable, the other solution can be used, i.e. locally increasing the lookahead. The rest of the grammar will still be LL(1), so there will be no overall performance penalty. CppCC accepts three kinds of local lookahead hints, which can be combined:

fixed lookahead: its effect is to tell the parser to inspect up to a specified number of tokens before deciding to chose a certain expansion.

syntactic lookahead: there are cases when the number of lookahead tokens cannot be predicted. In this case, the parser can try to match a whole expansion, and chose the associated "real expansion" if the match succeeds. Note that if the expansion contains nonterminals, the user actions inside the productions of those nonterminals are never executed[12]. However, user actions followed by an exclamation mark "`!`" will be executed even during lookahead[13].

semantic lookahead: there are also cases when the decision whether to chose a certain expansion is based on some semantic information. One can specify a boolean expression that will tell the parser what to do. Note that unlike user actions, semantic lookahead (although it is also user code) is always evaluated while the parser is performing the lookahead algorithm (to be more specific, if expansion `A` has a `LOOKAHEAD(B())` and `B`'s production itself contains a semantic lookahead (actually this is also true for all kinds of lookahead hints), then this semantic lookahead is always taken into consideration while performing the `LOOKAHEAD(B())`).

A lookahead hint can precede any expansion in a CppCC production that appears at a non-LL(1) choice point[14]. The syntax is[15]:

[12] If not obvious, the reason is that when a production is expanded during lookahead evaluation, there is no "real parsing" being performed, just a "peek" at what follows on the input stream. If the lookahead would fail, those user action should have never been executed.

[13] See the sample grammar for the CppCC's input file in the CppCC distribution ofr an example on how to use it. Additional restrictions apply to such user actions: for instance, they cannot use the normal environment guaranteed for normal functioning of the parser like the current token, aso) and, because the lookahead code can return at any point when the decision whether it succeeded or not was taken they are not even guaranteed to be executed.

[14] CppCC will detect such points and signal them to the user. Also, it will detect useless lookahead hints, point them out and ignore them. The only exception here is the semantic lookahead, which is never ignore at a choice point (it is intended to act more as a supplementary user-defined condition in the parsing algorithm).

[15] See Annex B for the exact syntax.

```
LOOKAHEAD(fixed_lookahead, syntactic_lookahead, semantic_lookahead) <some_expansion>
```

The effect is that if the lookahead succeeds, the expansion that follows it will be taken[16]. Any of the `fixed_lookahead`, `syntactic_lookahead` and `semantic_lookahead` parameters can be omitted (at least one must be present, and if they are combined they must appear in this order and separated by commas). If more than one kind of lookahead parameters is given, the lookahead will succeed if all of them do succeed (their combination is the AND operation applied to them). However, the fixed lookahead always takes precedence over the syntactic one, that is, when both are present, the parser will look ahead for the expansion provided as a syntactic lookahead, but only until it reaches the number of lookahead tokens specified as the fixed lookahead.

3.4.3.0.9 Example: We will discuss several of the most common situation where each kind of lookahead is needed:

a) Suppose we have the following productions:

```
ASSIGN_STATEMENT -> [ LABEL ":" ] LVALUE ":=" EXPRESSION ";"
LABEL -> <IDENTIFIER>
LVALUE -> <IDENTIFIER>
```

The problem here is that both LABEL and LVALUE start with the same token, IDENTIFIER, and if we make the choice point obvious by re-writing the first production the ambiguity becomes more clear:

```
ASSIGN_STATEMENT ->  LVALUE ":=" EXPRESSION ";"
                   | LABEL ":" LVALUE ":=" EXPRESSION ";"
```

This new production is the same as the original one, but it emphasizes the fact the the parser must make a decision when it encounters the "|" operator. Specifically, by following the general LL(1) algorithm, it reads a token from the input stream, sees that it is an `IDENTIFIER` token and must chose to consider it as the starting of a `LVALUE ":=" EXPRESSION ";"` expansion or of a `LABEL ":" LVALUE ...` expansion. The problem is, both alternatives start with `IDENTIFIER`, so the parser does not have enough information to base a decision upon. However, we can see that if we make it look at the next token after the `IDENTIFIER`, the two alternatives differ: one needs a `":"` while the other needs a `":="`. So, a lookahead of 2 would suffice here to disambiguate the grammar. Fig. 9 shows the CppCC production for this case.

```
(void) ASSIGN_STATEMENT ()
{
  (LOOKAHEAD(2) (LABEL <TWODOTS>))? LVALUE <ASSIGN_OP> EXPRESSION
  <SEMICOLON>
}
```

Figure 9: Fixed lookahead example

In presence of he lookahead hint, the parser will examine the next two tokens, and if they form a valid prefix for the following expansion (i.e. `(LABEL <TWODOTS>)`[17] in our case), then the parser choses that expansion. If not, the whole optional part is skipped and the parser continues with the `LVALUE`.

b) Let's now consider a grammar where any fixed-depth lookahead will not be enough, thus forcing us to use a syntactic lookahead[18]:

```
DECLARATION -> CLASS_DECLARATION | INTERFACE_DECLARATION
CLASS_DECLARATION -> ("public" | "abstract" | "final")* "class" REST_1
INTERFACE_DECLARATION -> ("public" | "abstract" | "final")*
                         "interface" REST_1
```

[16] Note that the syntax is a bit misleading, in that the lookahead is associated with an expansion, but it actually affects the decision at the level within which that expansion is included as an operand. For instance `(LOOKAHEAD(3) THIS | OTHER)` means "when having to chose between `THIS` and `OTHER`, use a lookahead of up to 3 tokens", although the `LOOKAHEAD` itself is associated with the `THIS` expansion.

[17] The parentheses surrounding the `LABEL <TWODOTS>` expansion are mandatory. Their purpose is to tell CppCC that the lookahead applies to them as whole, otherwise it would only apply to the expansion following it, i.e. `LABEL`, which is not what we want.

[18] This is a real-life example taken from the Java Language Specifications.

The problem with these two productions is that they have a common prefix that can have any length, so any finite lookahead will not suffice. In this case, we can use a syntactic lookahead to make the parser look after the string of modifiers for the keyword that actually tells what declaration it is. The CppCC production is shown in Fig. 10. The choice determination algorithm will then start by first inspecting any number of `PUBLIC`, `ABSTRACT` and `FINAL` tokens, then one more token. If this last token is a CLASS token, then the lookahead has succeeded and the `CLASS_DECLARATION` choice is taken. Otherwise, the algorithm continues with the `INTERFACE_DECLARATION` choice. The same effect (but with a serious performance penalty) would have been achieved by making a whole `CLASS_DECLARATION()` be the lookahead argument by writing something like `LOOKAHEAD(CLASS_DECLARATION()) CLASS_DECLARATION() | ...`. Although the parser will work correctly, the decision would involve parsing a whole `CLASS_DECLARATION`, which can be something quite large as compared to just the part up to the `class` keyword.

```
() DECLARATION ()
{
   LOOKAHEAD((<PUBLIC> | <ABSTRACT> | <FINAL>)* <CLASS>)
   CLASS_DECLARATION()
 | INTERFACE_DECLARATION()
}
```

Figure 10: Syntactic lookahead example

c) As an example for using the semantic lookahead we will consider a grammar containing the following productions[19]:

```
EXPRESSION -> FUNCTION_CALL | INDEXED_NAME
FUNCTION_CALL -> IDENTIFIER ( "(" EXPRESSION ( "," EXPRESSION)* ")" )?
INDEXED_NAME -> IDENTIFIER "(" EXPRESSION, ("," EXPRESSION)* ")"
```

The FUNCTION_CALL and INDEXED_NAME nonterminals look syntactically identical. The only difference is actually in their semantics. The identifier, in case of a function call should obviously be one that denotes a function, while for an indexed name the identifier should denote a variable. Fig. 11 shows an implementation of this grammar. Both kinds of expressions are created inside the EXPRESSION production, the choice whether to create a function call or an indexed name expression is taken depending on the kind of identifier that is found. The productions for each of the above expressions will receive as one of the arguments the symbol found as their prefix, and will contain the appropriate code for creating either a list of actual arguments or a list of indices. For symbol table handling, we used a LocalContext class which is assumed to implement a structure of nested declarative contexts.

3.4.4 The user code

Inside the syntax section, one can add any C++ code that it is wanted to be inserted into the generated parser class (i.e. class fields and methods of any kind). User code must be placed in a block surrounded by balanced curly braces, like this: `{ public: myMethod(...) { ..... }}`[20]. Methods added here have unrestricted access to the parser class internal fields. Several methods, if added inside such a block of code, have a special meaning:

`bool onParseError (const ParseException &pex);`
The parser's common error handler, see Section 3.5.4.

3.5 Error recovery

CppCC is very flexible regarding the error-handling approach. It offers support for all the error recovery techniques offered by other similar tools. Any of them (or both combined) can be used, according to one's taste. Basically, there are two ways of implementing a customizable error handling mechanism:

- using customizable error handlers (error catching expansions are also included here).

[19] This example is adapted from the VHDL93 grammar.
[20] CppCC is smart enough to skip strings and comments inside a user code, so they can contain any number of curly braces.

```
(Expression*) EXPRESSION (const LocalCtxt *ctxt)
{
  {
    Symbol *sym;
    Expression *exp;
  }
  <IDENTIFIER> {sym = ctxt->lookupName(token->image()); }
   /* here we decide how to parse the expression list that follows,
      depending on what its semantic meaning is
   */
  (  LOOKAHEAD({sym->kind == Symbol::FUNCTION})
     exp = FUNCTION_CALL(ctxt, sym)
   | exp = INDEXED_NAME(ctxt, sym)
  )
  {return exp; }
}
(Expression*) FUNCTION_CALL (const LocalContext *ctxt,
                             const Symbol *fun)
{
  {
    list<Expression*> argumentsList;
  }
  argumentsList = FUNCTION_ARGUMENTS_LIST(ctxt, fun);
  {
    return new FunctionCallExpression(fun, argumentsList);
  }
}
(Expression*) INDEXED_NAME (const LocalContext *ctxt,
                            const Symbol *var)
{
  {
    list<Expression*> indices;
  }

  indices = INDICES_LIST(ctxt, var);

  {
    return new IndexedName(var, indices);
  }
}
```

Figure 11: Semantic lookahead example

- using the C++ exception mechanism.

The use of exceptions is particularly well suited for a descendant recursive parser as they allow the user to deal with errors at the syntactic level that is most convenient (for instance, in a C parser, a good place to catch an exception that occurred due to a parse error could be just after the call to a `FUNCTION_DEFINITION` nonterminal; after reporting the error, the parsing process can continue by skipping to the next function and resume from there). The idea is that a catch clause implicitly sets a recovery point in the parser. However, if an application requires (or the one prefers), CppCC can generate code that does not use exception, errors being dealt with by using various error handlers which will be described below. The use of exceptions in the generated code is controlled by the `USE_EXCEPTIONS` option (See Sec. 3.1). If this option is set to false, use of exceptions is disabled and the generated code will be "exceptions clean" [21]. Figure 12 shows the exceptions used by the generated code.

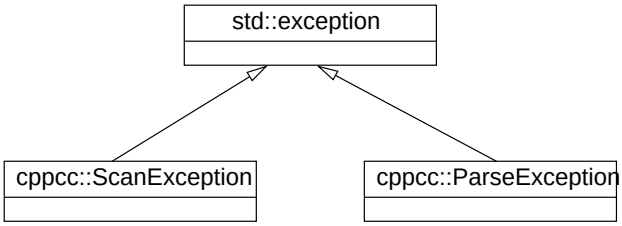

Figure 12: Exception hierarchy used inside an exception-enabled parser.

Usually, a parser has to deal with three kinds of errors: I/O errors, scanning errors, and parse errors. A fourth kind can also be (partially) handled here, that is semantic errors. Each of these will be discussed below along with how they can be dealt with in a CppCC generated front-end. Each kind of error will be discussed in two situation: when exceptions are enabled and when they are disabled.

3.5.1 I/O errors

From the CppCC's perspective, the only I/O errors that need to be dealt with are those that arise as the scanner reads an input stream. All the other I/O errors can only be caused by user code and therefore it is user's responsibility to provide a mechanism for handling them.

Two handlers are available for I/O errors. One is the scanner's default error handler, and one is user-defined. To define a custom error handler, all that's needed is to add a special method into one of the blocks of C++ code of the lexical section. The prototype of this method must be:

```
bool onIOError (ScanException &s)[22];
```

After each input operation, the scanner performs the algorithm shown in Fig. 13.

If an error has occurred, the scanner will always call a user error handler if one has been provided. This handler can either attempt to fix the problem in some way, in which case it can ask the parser to try and continue its operation by returning true as the result of the `onIOError` method. If a user handler is not available or it was unable to fix the problem, the default action is taken. What the default means depends on whether use of exceptions was enabled or not.

If exceptions were disabled, then the scanner will simply `abort()`. If the exceptions were enabled, then a `ScanException` object. With exceptions enabled, all the methods generated by CppCC that somehow will call the scanning method (including the parser's methods) have `ScanException` class into their throw clauses, thus allowing the exception to go up the stack until it reaches a user's catch clause for it (this may happen at the very top of the stack, where the parsing process was started, or inside any EBNF production, as will be explained below).

3.5.2 Example

Fig. 14 shows how to add I/O error handling to the Foo parser presented in Example 3.4.1.0.6. We will use both kinds of error handling: we write an error handler that attempts to recover from the error and a catch clause that deals with the situations in which recovery was not possible. The try-catch block surrounds the call to the start symbol's method of the parser because as said above, exceptions will propagate through the stack back to the user code if the I/O error handler returns false.

[21] Of course, this does not mean that other standard exception such as `std::bad_cast`, etc are guaranteed not to be thrown.

[22] This will change in future versions, as soon as the GNU's gcc C++ library will implement the ANSI standard I/O error handling through exceptions.

```
if (last input operation failed)
{
  create a new ScanException object ex containing information about the occurred error
  should_continue = false;
  if (onIOError method was implemented by the user)
    should_continue = onIOError(ex);
  if (!should_continue)
  { //this is the default error handler:
    if (exceptions were enabled)
      throw the ex object;
    else
      abort();
  }
  //else restart the last operation
}
```

Figure 13: I/O error handling algorithm performed by the scanning routine.

3.5.3 Lexical errors

Lexical errors occur if one of the following situations apply:

- none of the defined tokens could be matched against the input stream;
- `TOKENS_SPAN_EOF` option was set to false and the input stream ended before the current token was completely matched
- a call to `switchToState` made the scanner go into an invalid lexical state.

If no user handler is defined, the default action taken by the scanner is to wither throw a ScanException (if exception were enabled) or to simply abort. To define a user handler, a special method must be added inside a user code block inside the lexical section. The prototype of this method must be:

```
bool onScanError (ScanException &ex);
```

As it can be seen, if exceptions are disabled, the exception object (which contain information about the error that occurred) is still available because it is passed as an argument to the error handler. The algorithm performed by the lexer when an error occurs is shown in Fig. 15.

Just as for `onIOError`, the return value from `onScanError` indicates whether the scanner should attempt to resume its operation. An implementation of `onScanError` can try and fix the error situation (eg. it can simply skip the current character from the input stream if the error was cause by a character that could not be matched in the current DFA state of the scanner, print a warning that some characters have been skipped) then return true to the parser in order to make it try to resume its normal operation. IF the return value is false, the scanner assumes the error is unrecoverable and does the default error handling action, which implicitly terminates the current call of the scanning method.

3.5.3.0.10 Example: Let's enhance the parser from Example 3.5.2 by adding lexical error handling. Fig. 16 shows what needs to be added: we have another `code` block into the lexical section that contains the lexical error handler and an extra catch clause into the `main` function. The error handling routine will first attempt to fix the error in some way (it may for instance try to skip discard some characters from the input stream) and if that succeeds it tells the scanner to continue. If recovering is not possible, it just returns false and let the scanner abort operation. In this latter case, a ScanException will be thrown and it will be caught by the catch clause in the `main` functions after it propagates through the current call stack.

3.5.4 Syntactic errors

Doe to the fact that the parser is made up of many functions that call each other recursively handling syntactic errors is a more complex matter here. A generic error handler as provided for I/O and lexical errors is of limited use here, as

```
/* into the Foo grammar file: */

OPTIONS {
  USE_EXCEPTIONS = "true"; //enable exceptions into the generated code
}

SCANNER FooScanner
{
  // tokens, etc.....

  {
    bool onIOError (ScanException &ex)
    {
      // try to recover....

      if (could_recover) return true;
      else return false;
    }
  }
}

/* into a C++ source that needs to parse some Foo code: */

int main ()
{
  FooParser p(&cin); // Create a new parser object that reads from cin

  try {
    p.INPUT_FILE();
  } catch (ScanException &ex)
  { // If we got here, something bad happen to cin....
    cerr << (string) ex << endl;
    cerr << "Aborting..." << endl;
  }
}
```

Figure 14: Handling I/O errors using both an error handler and exceptions.

```
if (scanner encountered an error)
{
  create an instance ex of the class ScanException and
                                    put current status info in it;
  should_continue = false;
  if (onScanError was implemented by the user)
    should_continue = onScanError(ex);
  if (!should_continue)
  { //this is the default error handler
    if (exceptions were enabled)
      throw ex;
    else
      abort();
  }
}
```

Figure 15: Lexical error handling algorithm performed by the scanning routine.

```
/**
 * Additions to the example in Fig. 14 in order to do lexical error handling.
 */

/* The modified lexical section will contain this block */
{
  bool onScanError (const ScanException &ex)
  {
    if (can_recover) {
      cerr << "Warning: " << ex.what() << endl;
      cerr << "Input truncated, continuing..." << endl;

      // Do some recovering stuff here...

      return true; // Tell the scanner to keep going
    } else
      return false; // We are hopeless....
  }
}

int main () /* The modified main function: now we catch I/O and lexical exceptions */
{
  FooParser p(&cin); // Create a new parser object that reads from cin

  try {
    p.INPUT_FILE();
       << "Aborting." << endl;
  } catch (cppcc::ScanException &scex)
  { // If we got here, the scanner jammed really bad...
    cerr << "Unrecoverable lexical error: " << scex.what() << endl
         << "Aborting." << endl;
  }
}
```

Figure 16: Handling lexical and I/O errors using both an error handler and exceptions

each syntactic error occurs in a certain context and needs to be dealt with separately. However, such an error handler is available, and can be used to execute common actions to be taken when an error occurs, such as reporting it (or aborting the parsing process if no other error recovery method is implemented). The user error handler must be inserted into a code block inside the syntax section, and has the prototype:

```
bool onParseError (const ParseException &pex);
```

The return value has the same meaning as for the lexical and I/O error handlers, if true it means the parser should try and resume from where it left, and if false it means that further error-handling actions must be taken. If exceptions are not enabled, the only thing left to do next is to `abort()`. With exceptions enabled, the created `ParseException` object will be thrown up the calling stack. The error handling algorithm is shown in Fig. 17.

```
if (a parse exception has occurred)
{
  create an instance pex of the class ParseException and store the state data into it.
  should_continue = false;
  if (onParseError was implemented by the user)
    should_continue = onParseError(pex);
  if (!shouldContinue)
    if (exceptions were enabled)
      throw pex;
    else
      abort();
}
```

Figure 17: Syntax error handling algorithm performed by the parser.

The productions grammar of CppCC allow exceptions that were thrown while parsing according to a certain expansion to be caught and some user-provided code to be executed, in a manner very close to the C++ try-catch semantics. The general form is shown in Fig. 18.

```
( SOME_EXPANSION_THAT_MIGHT_FAIL() )
   catch (ParseException pex) { do something about it }
   catch (ScanException scex) { you can do something about
                                     that too, if you wish }
   catch (MyException mex)    { handle some exception thrown
                                       by user actions }
   // More exception types....
   catch (...)                { handle any possible exception here, if want to }
{ user action for the expansion, executed if everything was ok }
```

Figure 18: General form of a catch chain inside a production.

Generally, after each expansion, a list of catch clauses can follow. These will catch exceptions thrown at any recursion level inside that expansion (if no other catch clauses for the same exception(s) are reached first). In order for this mechanism to work, when exceptions are enabled, all the methods generated from CppCC productions will have in their throw clause at least the exception classes mentioned above. Any I/O, lexical, parsing or user-specific exception can be caught and dealt with inside any CppCC production. If such an exception is never caught inside a production, it will cause the parsing process to terminate abnormally and the exception will either be caught in the user code that invoked the parser or cause the whole program to abort by calling the standard handler.

An alternative solution for handling errors locally is to add error-handling expansions, as code-only productions (see Sec. 3.4). Suppose the parser reaches point where the current expansion is `(A() | B() | skipt_to_C()) C()`. If the current lookahead does not match neither an A or a B at the choice point, then `skip_to_C()` will be chosen. This can be a code-only production that displays an error message and consumes tokens until a point from where recovery is possible, such as the beginning of a `C()`.

3.5.5 Semantic errors

CppCC offers support for user-defined error handling such as semantic checks during parsing. Let's assume we want to write a parser for a language where the atoms of an expression can be, among other things, identifiers. A semantic check here would be to see whether a variable with that identifier was declared before being used inside an expression. If the check fails, the user code that performed it can chose to throw an exceptions, something like a SymbolNotFoundException. Because this exception is thrown from inside the user code located in a production, if the exception will be caught at another level of the grammar, the exception must appear inside the throw clause of the method that will be generated from that production. For this reason, CppCC grammars can specify custom throw clauses for each production, where an user can add his own exceptions. If exceptions are enabled, besides these custom exception, the throw clause will of course contain the standard CppCC exceptions.

3.5.5.0.11 Example Fig. 19 shows how the parser for our Foo language (See Fig 7) can be enhanced to do error recovery and semantic checks. The `FACTOR` nonterminal can now also be an `<IDENTIFIER>` when this happens, the identifier is looked up into the symbol table, and if it doesn't exist, a `SymbolNotFoundException` is thrown. The error recovery strategy implemented here consists of simply skipping the token that caused the error. The next time EXPRESSION() is called from `INPUT_FILE()`, it will attempt to continue with parsing starting with the next token after the one that was discarded.

```
(int) EXPRESSION () // No throw here, we catch everything
{
  ( TERM() ( <ADD_OP> TERM() )*
  ) catch (exception &pex) { // We catch  all exceptions, including
                             // ParseException and SymbolNotFoudException
     cerr << pex.what() << endl;
     scanner.consume(); // We simply consume the token that caused the
                        // the error and try to move on
  }
}

(int) FACTOR () throw (SymbolNotFoundException) // FACTOR might throw this
{
  <DECIMAL_NUMBER> | (<LPAREN> EXPRESSION() <RPAREN>) |
  <IDENTIFIER> { if (symTable.lookup(token.image) == NULL)
                   throw SymbolNotFoundException(token.bPos,
                                "Symbol \"" + token.image + "\" not found.")
               }
}
```

Figure 19: Syntactic and semantic error recovery.

4 Overview of the generated code

From a grammar specification, CppCC will generate a set of C++ sources that will be written in a certain target directory which can be given as an argument on the command line. Considering the (complete) grammar for the language Foo presented in Example 3.3.1.0.3 as input file, CppCC will generate the following files (their base names are identical to the class names specified into the grammar file):

foo_token.hh, foo_token.cc header and definition of the token's and helper classes used by the Foo parser and scanner;

foo_scanner.hh, foo_scanner.cc header and definition of the Foo scanner and its helper classes;

foo_parser.hh, foo_parser.cc header and definition of the Foo parser and its helper classes.

The contents of each of these files as well as what each of the classes contains will be discussed in the remainder of this section.

The `cc` and `hh` extensions can be changed to whatever the compiling platform recognizes as the C++ source and header extension, by using the `SRC_EXTENSION` and `HDR_EXTENSION` options.

All the class names in the sources generated with CppCC will be placed in a separate namespace whose default name is *cppcc*. Therefore, before using the parser or scanner, the user code must contain at least the following two lines;

```
#include "foo_parser.hh"
using namespace cppcc; // or use cppcc::FooParser, if it's the only thing used here
```

The `NAMESPACE_NAME` option can be used to specify an alternate name for the namespace[23].

4.1 The token class' sources

These sources contain two class definitions, the token's class and a helper class called Position. Both these classes are declared into the cppcc namespace. If present, the block of code that precedes the TOKEN section into the grammar file will be inserted at the beginning of the token class' header file, just before the line that declares the cppcc namespace. The header will therefore look like this:

```
... preamble user code, if any ...
namespace cppcc
{
  class Position { ... };
  class FooToken { ... };
}
```

4.1.1 The token class

The token class is the mean of communication between the scanner and the parser. Each time it is invoked, the scanning routine returns a new Token object containing the next token to be used by the parser. The token object is an instance of the token class. The token's class features are given below:

cppcc::FooToken the token class used within the generated scanner and parser

- public fields:

  `Position bPos, ePos;` the starting and ending position of the token inside the input file;

  `int id;` indicates the kind of this token; each token that is declared inside the lexical section will be assigned a unique integer to indicate its kind.

- public constructors:

  `FooToken ();` default constructor: image is initialized to the void string, `bPos` and `ePos` are initialized using the Position's default constructor (see below) and the `id` is set to 0. These values are the default values to which the fields will be set by other constructors if not otherwise mentioned.

  `FooToken (int id_, const std::string &image_,`
  `const Position &bPos_, const Position &ePos_);` creates a new token and sets the fields to the given values.

  `FooToken (int id_);` creates a new token with the given id;

  `FooToken (const std::string &image);` creates a new token with the given image;

  `FooToken (const Position &bPos, const Position &ePos = Position());` creates a new token with the position fields set to the given values; if no end position is given, a default value is provided.

- public methods:

  `std::string& image ();` returns the image of this token. The obtained reference points to an internal field of the token object so any changes made to the image will be preserved[24]. The actual return type of this method (and of the internal image field) can be controlled by the user, by means of the `STRING_CLASS` option.

[23] E.g. when several cppcc generated parsers are part of the same project, using different namespace names is needed in order to avoid confilcting definitions.

[24] Note that the length of the token is a proper field and therefore changes to the image will not be reflected in it; its value will remain set to the initial value filled in by the scanner.

4.1.1.0.12 Example Suppose in our project we have an already-implemented-and-used-all-over-the-place string class, called `MyString`. We don't want to have to convert token images from `std::string` into a `MyString`; the best way of avoiding this is to have the tokens store their images directly as `MyString` objects. Therefore, we add to the options section:

```
STRING_CLASS = "MyString"
```

Then the resulting token class will have its `image` method declared like this:

```
MyString& image ();
```

For this customization to work, the custom string class must meet certain (rather straightforward) conditions:

- it must be default constructible
- it must provide an `assign` public method with the same semantics as for `std::string`, i.e. must be declared as:

  ```
  MyString& MyString::assign(const char* s, size_type n)
  ```

  and its effect must be the assignment of the first `n` characters starting at `s` to `*this`.

Also, any user code found into the token's section of the grammar file will be merged inside the token's class body (the user is free to add more custom constructors if needed).

Additionally, for each token defined in the syntax section and for the special `<eof>` token, a symbolic constant id is generated as a public static field of the token's class.

4.1.2 The Position class

Besides the token's class, the token's header contains the declaration for the `Position` class. This class is used to encapsulate a position inside a file. Its main features are given below:

cppcc::Position a textual position inside a file

- public fields:

  `int ln;` line inside the file.

  `int col;` the column inside the file. This field is only generated if the `COUNT_COLUMNS` option is set to *true*.

- public constructors:

  `Position ();` default constructor: `ln` and `col` fields are initialized to `0`.

  `Position (int ln_, int col_);` creates a new position object indicating the given line and column;

  `Position (const Position& o);` copy constructor.

4.2 The scanner class' sources

These sources contain the definitions of the scanner's class and of the ScanException class used in lexical error handling (see Sec. 3.5.3). If present, the block of code that precedes the lexical section into the grammar file will also be included at the beginning of the header file, which will roughly look like this:

```
... preamble user code, if any ...
namespace cppcc
{
  class ScanException { ... };
  class FooScanner { ... };
}
```

4.2.1 The scanner class

The main purpose of the scanner class is to encapsulate a DFA that recognizes tokens on the input stream. Its interface must mainly provide methods for retrieving tokens in the order that they appeared into the input stream. The tokens that are recognized by the scanner class CppCC generates are those that were defined into the lexical section of the input grammar. Besides retrieving tokens one at a time, the scanner class must facilitate lookahead examination of tokens, because the parser that uses the token class must be able to inspect not only the current token, but also a certain number of tokens that follow it.

The implementation of the scanner class uses an queue of matched tokens in order to allow lookahead inspection. New tokens are inserted into the queue upon request, i.e. wen the parser requests tokens beyond the point reached by the scanner. In this case, the scanner will invoke an internal method that will scan the input stream as many times as necessary to reach the token that was requested. Tokens are kept in the queue and can be inspected several times by using the `FooScanner::la(...)` method described below. To request a token to be removed from the queue, the `FooScanner::consume()` method must be called. This method will remove the oldest token (eventually by reading it just then form the input stream, if the queue was empty at the moment of the call). Figure 20 illustrates the functioning of the lookahead queue.

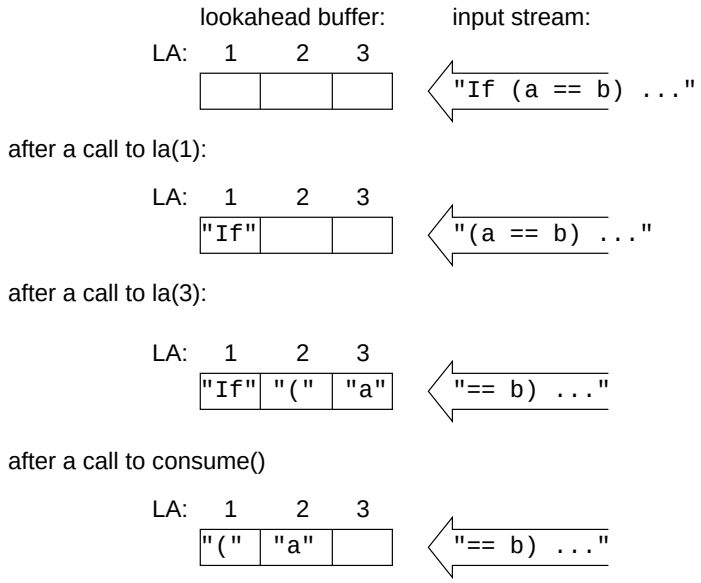

Figure 20: Functioning of the scanner's token queue.

In order to speed up character reading, the scanner class uses its own input buffer. Several methods are provided to the user that allow a buffer with its current state and associated input stream to be saved in a single opaque object that can be later restored or to simply switch over to another input stream next time the buffer needs to be refilled[25].

The main features of the scanner's class are described below:

`cppcc::FooScanner` transforms an input character stream into a stream of tokens according to the grammar

- public fields:
- public constructors:

  `FooScanner ();` creates a new scanner object with no associated input stream. In this case, an input stream must be set up before the scanner is first used.

  `FooScanner (istream *in_ = NULL);` creates a new scanner object that will read characters from the given stream; the state of the input stream is not checked until the first read attempt. This means that `NULL` can be given as argument. If so, the first call to the scanning method will also result in a call to the `wrap()` method prior to any token matching. This allows all the stream-opening related code to be grouped in a single place (i.e. in the wrap method).

[25] These are the most common situation a scanner needs to deal with, i.e. file inclusion, in which case a new input stream must be open at a certain point and once it is completely read, the scanner must be switched back to the old input stream at the point where it was left. The second case occurs when multiple files are to be processed sequentially.

- public methods:
  - input stream related methods:
    - input stream switching: these methods allow the user's code to query, set, save and restore the input state of the scanner

      `istream& getInputStream ();` returns the stream that is currently used by the scanner;

      `void switchToStream (istream *in);` causes the scanner to start reading characters from the given stream next time its input buffer underruns. The current input state will not be saved when the switching occurs. This method is intended o be called from the wrap() handler as the default action when an input stream was completely read and the user wishes to move to the next stream. Note that switching to a new stream does not involve the scanner taking any actions such as closing or deleting the old istream object. It is the user's responsibility to open, close and delete the stream objects.

      `FooScanner::StreamState* pushStream (istream *in);` causes the scanner to save the current input state (this includes the internal buffer and the pointer into it, the positions where last token was matched, the associated input stream, etc) into a newly allocated `StreamState` object and then switch over to the given input stream as soon as it will need to match the next token.

      `void popStream (FooScanner::StreamState *s);` causes the scanner to switch to a previously saved input state that is contained in the given `StreamState` object. As a pair of `pushStream`, this method also takes care of deleting the `StreamState` object that was allocated by it. The next token read will be from the restored input stream.
    - low lever character-oriented interface: these methods provide character-oriented access from the user's code to the internal input buffer of the scanner.

      `int getChar () throw(ScanException);` returns the current character that would have been be read by the scanner and advances its pointer to the next character[26].

      `void unGetChars (const char *s, int n);` puts n characters starting from s into the input buffer, cause them to be the next characters read by the scanner. Note that characters will be read *in the same order* as they are in s, not in reversed order.

      `void unGetChar (char c);` puts c into the input buffer causing it to be the next character read by the scanner.

      `void unGetChars (const string &s);` puts all the characters in s into the input buffer causing them to be the next characters read by the scanner. Note that characters will be read *in the same order* as they are in s, not in reversed order.

      `void unGetChars (char *s);` puts all the characters in s into the input buffer causing them to be the next characters read by the scanner. Unlike the version that also receives a counter argument, here s must be a null-terminated string. Note that characters will be read *in the same order* as they are in s, not in reversed order.
  - token retrieval methods:

    `FooToken* la (int k = 0) throw(ScanException);` returns the (k+1)'th lookahead token. If `k == 0`, this is exactly the current token in the input stream (the LL(1) token).
    This method may cause the scanner to read tokens from the input stream and store them into the queue (if `k == 2` for instance and there was only one token into the queue, two more tokens are read and stored into the queue which will now contain three tokens. The third will be returned as the result of the method).
    The token object is guaranteed not to be altered until a subsequent call to consume or la. The user code must *never* attempt to delete a token object, as this will cause unexpected behaviour from the parser.

    `void consume () throw(ScanException);` removes the currently LA(1) (i.e. the token that would be returned by a call to la(0)) from the tokens' queue. If the tokens queue is empty at the moment of the call, a call to la(0) is issued prior to removing the token.
  - lookahead support methods (see Sec. 3.4.3):

    `void setMarker ();` saves the current position into the input token stream so that it can be restored later. Successive calls are stacked.

[26] Note that due to internal buffering that is done by the scanner, this is almost never similar to getInputStream().get(). This would happen only when the internal buffer would be empty.

`void rewindToMarker ();` rewinds the input token stream to the most recently saved position, and pops that position from the markers stack. After the rewind operation, the next tokens read are exactly the ones that were on the input stream previous to the matching `setMarker()` call.

`void pushBack (const Token &t);` pushes the token object t at the front of the token queue, causing it to be the new LA(1) token (t is not copied, and should therefore not be deleted after the call to `pushBack`).

`bool lookingAhead ();` returns true if the scanner is working in lookahead mode (i.e. at least one marker was set).

– input stream position handling:

`const Position& getCurrentPos ();` returns the current position into the input stream. The current position that is reported is that at which the LA(1) token begins (this is actually the point that the parser has reached, the fact that the scanner is reading tokens ahead should be transparent to the external classes).

`void resetPos();` resets the current position within the input stream (can be called from within the `wrap()` user method, if a new input stream is opened). This method is automatically called when a new input stream is set up (by calling `pushStream` or `switchToStream`). The beginning position is line 1, column 1.

`void newLine();` increments the line counter of the scanner and resets the column counter to 1; A call to this method should appear as part of the token action associated to the line terminator tokens[27].

– lexical state handling (see Sec. 3.3):

`void switchToState (int s);` sets the lexical state of the scanner to s; has no effect if s is an invalid state. Note that the call only takes effect when a new token will be read from the input stream (any tokens that are already into the token buffer will be returned to the parser as they are). For this reason. state switches should only appear as part of token actions, which are always executed just after a token was matched on the input stream, and it is certain that the next token will be matched according to the new state.

`void getState ();` returns the current lexical state of the scanner.

`int pushState (int newState);` sets the lexical state of the scanner to newState, but saves the current one onto an internal stack. The old lexical state is also returned as the result value of the call.

`void popState ();` switches back to the topmost lexical state saved onto the stack. Honest, i cant remember what happens if the stack is empty, but i'm sure noone that really cares about that at four o'clock in the morning anyway.

Besides the methods and fields mentioned above, any user code (fields and methods) found into code of the lexical section will be merged into the scanner class.

The token actions will be inserted into the internal scanning method at the appropriate points so that they will be executed each time a new token is accepted by the scanner[28].

4.2.2 The ScanException class

This class is used by the scanner for signaling lexical errors. When such an error occurs while in the scanning method, a new object of this type is created and the location within the input file and the error's reason are stored inside it before it is passed away to the parser or to the user's lexical error handler. The main features of the ScanException class are:

`cppcc::ScanException` contains the description of a lexical error

- public fields:

  `Position pos;` the position relative to the beginning of the input stream where the error occurred.

- public constructors:

[27] The scanner itself does not treat characters like `'\n'` or `'\r'` in any special way, it is the user's responsibility to handle line terminators. However, the column counter is always incremented as characters are read.

[28] This means that a user action will only be executed once, just after the token has been recognized in the input stream, and never later, even if the token is returned more than once as the result of a `la(k)` call.

`ScanException (const string &reason = "Scan exception");` creates a new ScanException object with no associated position and the given description.

`ScanException (const Position &pos_, const std::string &reason_ = "");` creates a new object with pos initialized to the given Position and that will return as the given string as the result of its `what()` method;

* public methods:

  `char* what ();` reimplemented from the std::exception superclass. Returns a string containing a description of the lexical error that occurred.

  `operator string();` returns a string containing the position in the input stream associated with the exception followed by the description of the error.

4.3 The parser class' sources

These sources contain the parser's class and the ParseException class which is used by the lexical error handling (see Sec. 17). If present, the block of code that precedes the syntax section into the grammar file will also be included at the beginning of the parser's header file, which roughly looks like this:

```
... preamble user code, if any ...
namespace cppcc
{
  class ParseException { ... };
  class FooParser { ... };
}
```

4.3.1 The parser class

The parser class encapsulates the automaton that accepts inputs conforming to the grammar specified into the syntax section. As each syntactic expansion is matched against the input token stream, the associated user actions are executed, if any. Each instance of a parser will use a scanner object to filter its input stream and transform characters into tokens. The main features of the parser class are:

`cppcc::FooParser` contains a parser for the grammar specified into the syntax section

- public fields:

  `FooScanner scanner;` the scanner object used to retrieve tokens from the input stream.

  `FooToken *token;` points to the token object that was the most recently retrieved from the scanner (aka the current token).

- public constructors:

  `FooParser (istream *in = NULL);` creates a new parser object that will read from the given stream. The pointer to the initial input stream is passed on to the scanner's constructor.

- public methods:

  – parsing methods: for each production found into the syntax section of the grammar file, a public method will be generated. The formal arguments list and return type are exactly those specified into the productions' declarations. The exceptions specified into the throw clause of each production will also appear into the throw clause of the corresponding method. Additionally, if the *USE_EXCEPTIONS* option is set to true, the `ios_base::failure`, `ScanException` and `ParseException` exceptions will also be appended to the exceptions list of each method. Each of these methods can be called from the user's application code in order to start the parsing process; the called method will not return until either the parsing is completed or aborted due to an error.

4.3.2 The ParseException class

This class is used by the parser for signaling syntax errors. When such an error occurs into one of the parser's methods, a new object of this type is created and the location within the input file and the error's reason are stored inside it before it is passed to the user's syntax error handler and/or thrown down the call stack. The main features of the ParseException class are:

`cppcc::ParseException` contains the description of a syntax error

- public fields:

  `Position pos;` the position relative to the beginning of the input stream where the error occurred. The position is the one reported by the scanner's `getCurrentPos()` method.

- public constructors:

  `ParseException (const string &reason_ = "Parse exception");` creates a new ParseException object with no associated position and the given description.

  `ParseException (const Position &pos_, const string &reason_ = "Parse exception");` creates a new object with pos initialized to the given Position and that will return as the given string as the result of its `what()` method;

* public methods:

  `char* what ();` reimplemented from the `std::exception` superclass. Returns a string containing a description of the syntax error that occurred.

  `operator string ();` returns a string containing the position associated with the parse exception followed by the exception's description.

Annexes

A Using the profile based optimization feature

A.1 Rationale

Due to the hardcoded implementation of the scanner's DFAs, the structure of the generated code contains a large number of cascaded if statements, something like:

```
if (the current character has the code x) goto ...;
else if (the current character has the code y) goto ...;
```

Such a sequence of if statements is generated for each state of each DFA, and one if must be taken for each class of characters that are valid in that state. However, certain characters occur alot more often in usual grammars (e.g. in C, although the character set is the whole ASCII set, a very large percent of a C source file will only contain printable character codes). It is therefore possible to arrange these cascated ifs such that those that are most probabe to be taken are placed at the beginning of each sequence.

A.2 How to create and use the profile information.

Activating this optimization in cppcc is easy: just set the `PROFILING_FILE` option to the name of file to be used for gathering profile information as the parser runs.

Obtaining an optimized scanner is a two-step process:

First, set the `PROFILING_FILE` option and re-generate the scanner's sources. Because the profiling data file does not exist, cppcc will figure out that it has to be created, and will generate a scanner that will gather up profiling data as it runs, then dumps it into that file. With this first scanner, a dummy program must be created that just reads typical input (something like "training the scanner"). The scanner contains a special destructor that will put the gathered data on the disk.

Once this is done, the resulted profile file was created and it will be used by cppcc when re-generating the scanner. So all that's left to do is to run cppcc again on the **unmodified**[29] grammar. This time, the scanner will be generated by taking into account the statistics found in the profiling file.

A.3 Statistical results

As of yet, I didn't obtain any astonishing speedups. I have a few exaplanations for this (one might be that most processors do this sort of optimization anyway - it is essentially just another form of jump prediction). I am relying on you folks to contribute some statistical data on this - or some valid explanation of why the whole idea is worthless :-)

B Complete *CppCC* grammar

B.1 The CppCC input lexical structure

The CppCC's input grammar language is case-sensitive. C or C++ style comments cab be freely used anywhere within the file.

The reserved words used by CppCC are:

```
OPTIONS TOKEN SCANNER SPECIAL SKIP PARSER LOOKAHEAD throw catch
```

Note that these are only reserved words outside a `c_block` construct (see below).

B.2 The CppCC input syntax

```
<grammar_file> -> [ <options_section> ]
                  <token_customization_section>
                  <lexical_section> <syntax_section>

<options_section> -> "OPTIONS" "{" ( <option_name> "=" <option_value> )* "}"
```

[29] Of course, if the lexical section is modified, the profiling data becomes inaccurate, and cppcc will refuse to use it.

```
<token_customization_section> -> [ <c_block> ]
                                 "TOKEN" <c_token_class_id> [ <inheritance> ]
                                 <c_block>

<lexical_section> -> [ <c_block> ] "SCANNER" <c_scanner_class_id> [ <inheritance> ]
                     "{" <scanner_decl> * "}"

<scanner_decl> ->  <c_block>
                 | [ <lexical_states_list> ] <token_kind> "{" <token_decl> * "}"
                 | "SPECIAL" "{" <special_token_decl> * "}"

<token_kind> -> "TOKEN" | "SKIP" | "MORE" | "KEYWORD"

<lexical_state_list> -> "<" ( "*" | <c_state_id> ( "," <c_state_id> )* ) ">"

<token_decl> ->  "<#" <c_token_id> ":" <token_regexp> ">"
               | "<" <c_token_id> ":" <token_regexp> ">" [ <c_block> ]

<special_token_decl> -> <c_token_id>

<token_regexp> -> <regexp_or_list>

<regexp_or_list> -> <regexp_cat_list> ( "|" <regexp_cat_list> )*

<regexp_cat_list> -> <regexp_term> +

<regexp_term> -> <regexp_atom> [ "+" | "?" | "*" ]

<regexp_atom> ->  <c_string_literal>
                | "<" <c_terminal_id> ">"
                | <character_list>
                | "(" <regexp_or_list> ")"

<character_list> -> ["~"] "[" <character_descriptor> ( "," <character_descriptor> )* "]"

<character_descriptor> -> <c_character_literal> [ "-" <c_character_literal> ]


<syntax_section> -> [ <c_block> ] "PARSER" <c_parser_class_id> [ <inheritance> ]
                    "{" <parser_decl> * "}"

<parser_decl> ->  <c_block>
                | <production>

<production> -> ( <c_type_decl> ) <c_nonterminal_id> "(" <c_formal_args_list> ")"
                  [ <throw_clause> ]
                "{" <c_block> <or_list> "}"

<throw_clause> -> "throw" "(" [ <c_type_id> ( "," <c_type_id> )* ] ")"

<or_list> -> <cat_list> ( "|" <cat_list> )*

<cat_list> -> <expansion> +

<expansion> -> [ <lookahead> ] [ <c_block> ]["!"] <atom> [ "+" | "?" | "*" ]
               [ <catch_clauses> ][ <c_block> ]["!"]
```

```
<lookahead> -> "LOOKAHEAD" "(" [ <integer_literal> ] [","]
                                [ <or_list> ] [","]
                                [ <c_expression> ] ")"

<atom> ->  [ <c_identifier> "=" ] ( <nonterminal_call> | <c_terminal_id> )
         | "(" <or_list> ")"

<nonterminal_call> -> <c_nonterminal_id> "(" <c_actual_args_list> ")"

<catch_clauses> -> <catch_clause> +

<catch_clause> -> "catch" "(" <exception_id_specs> ")" <c_block>

<exception_id_specs> -> "..." | <c_exception_decl> ( "," <c_exception_decl> )*

<c_exception_decl> -> <c_type_id> <c_exception_id>

<c_terminal_id> -> "<" <c_token_id> ">"

<c_token_class_id> -> c++_identifier

<c_state_id> -> c++_identifier

<c_token_id> -> c++_identifier

<c_scanner_class_id> -> c++_identifier

<c_parser_class_id> -> c++_identifier

<c_nonterminal_id> -> c++_identifier

<c_exception_id> -> c++_identifier

<c_type_decl> -> c++_type_specification

<c_block> -> "{" c++_code_with_balanced_parentheses "}"

<c_expression> -> "{" c++_expression "}"

<c_character_literal> -> c++_character_literal
```

C CppCC commad line syntax

```
Usage: cppcc [-h | --help [=yes|no|true|false]] [-V | --version [=yes|no|true|false]]
            [-v | --verbose [=yes|no|true|false]] [-d | --dir <string>]
            [-g | --debug <string>]
Options:
```

[-h | --help [=yes|no|true|false]]	Prints this help message.
[-V | --version [=yes|no|true|false]]	Output version and copyright information.
[-v | --verbose [=yes|no|true|false]]	Makes CppCC be verbose about what it's doing.
[-d | --dir <string>]	Put the ouput files ino the specified dir.
[-g | --debug <string>]	Enables debug switches; each letter in the string argument enables a certain feature: 's' - enables scanner's debug features, 'p' - enables parser's debug features, 'o' - enables the option parser debug features, 'd' - causes a parse tree dump 'f' - dumps DFA intermediate data.

D GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.

- Preserve the section entitled “History”, and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- In any section entitled “Acknowledgements” or “Dedications”, preserve the section’s title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

- Delete any section entitled “Endorsements”. Such a section may not be included in the Modified Version.

- Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise combine any sections entitled “Acknowledgements”, and any sections entitled “Dedications”. You must delete all sections entitled “Endorsements.”

Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have no Invariant Sections, write “with no Invariant Sections” instead of saying which ones are invariant. If you have no Front-Cover Texts, write “no Front-Cover Texts” instead of “Front-Cover Texts being LIST”; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.