## Description of the JavaCC Grammar File

This web page contains the complete syntax of Java Compiler Compiler grammar files with detailed explanations of each construct.

Tokens in the grammar files follow the same conventions as for Java. Hence identifiers, strings, characters, etc. used in the grammars are the same as Java identifiers, Java strings, Java characters, etc.

*White space* in the grammar files also follows the same conventions as for Java. This includes the syntax for comments. Most comments present in the grammar files are generated into the generated parser/lexical analyzer.

Grammar files are preprocessed for Unicode escapes just as Java files are (i.e., occurrences of strings such as \uxxxx - where xxxx is a hex value - are converted the the corresponding Unicode character before lexical analysis).

*Exceptions to the above rules:* The Java operators "<<", ">>", ">>>", "<<=", ">>=", and ">>>=" are left out of Java Compiler Compiler's input token list in order to allow convenient nested use of token specifications. Finally, the following are the additional reserved words in the Java Compiler Compiler grammar files.

| | | | |
|---|---|---|---|
| **EOF** | **IGNORE_CASE** | **JAVACODE** | **LOOKAHEAD** |
| **MORE** | **options** | **PARSER_BEGIN** | **PARSER_END** |
| **SKIP** | **SPECIAL_TOKEN** | **TOKEN** | **TOKEN_MGR_DECLS** |

Any Java entities used in the grammar rules that follow appear italicized with the prefix *java_* (*e.g.*, *java_compilation_unit*).

---

javacc_input ::= javacc_options
"PARSER_BEGIN" "(" <IDENTIFIER> ")"
*java_compilation_unit*
"PARSER_END" "(" <IDENTIFIER> ")"
( production )*
<EOF>

The grammar file starts with a list of options (which is optional). This is then followed by a Java compilation unit enclosed between "PARSER_BEGIN(name)" and "PARSER_END(name)". After this is a list of grammar productions. Options and productions are described later.

The *name* that follows "PARSER_BEGIN" and "PARSER_END" must be the same and this identifies the name of the generated parser. For example, if *name* is "MyParser", then the following files are generated:

**MyParser.java:** The generate parser.
**MyParserTokenManager.java:** The generated token manager (or scanner/lexical analyzer).
**MyParserConstants.java:** A bunch of useful constants.

Other files such as "Token.java", "ParseError.java", etc. are also generated. However, these files contain boilerplate code and are the same for any grammar and may be reused across grammars.

Between the PARSER_BEGIN and PARSER_END constructs is a regular Java compilation unit (a compilation unit in Java lingo is the entire contents of a Java file). This may be any arbitrary Java compilation unit so long as it contains a class declaration whose name is the same as the name of the generated parser ("MyParser" in the above example). Hence, in general, this part of the grammar file looks like:

```
PARSER_BEGIN(parser_name)
. . .
class parser_name . . . {
  . . .
}
. . .
PARSER_END(parser_name)
```

JavaCC does not perform detailed checks on the compilation unit, so it is possible for a grammar file to pass through JavaCC and generate Java files that produce errors when they are compiled.

If the compilation unit includes a package declaration, this is included in all the generated files. If the compilation unit includes imports declarations, this is included in the generated parser and token manager files.

The generated parser file contains everything in the compilation unit and in addition contains the generated parser code that is included at the end of the parser class. For the above example, the generated parser will look like:

```
. . .
class parser_name . . . {
  . . .
  // generated parser is inserted here.
}
. . .
```

The generated parser includes a public method declaration corresponding to each non-terminal (see javacode_production and bnf_production) in the grammar file. Parsing with respect to a non-terminal is achieved by calling the method corresponding to that non-terminal. Unlike yacc, there is no single start symbol in JavaCC - one can parse with respect to any non-terminal in the grammar.

The generated token manager provides one public method:

```
Token getNextToken() throws ParseError;
```

For more details on how this method may be used, please read the description of the Java Compiler Compiler API.

---

javacc_options ::= [ "options" "{" ( option_binding )* "}" ]

The options if present, starts with the reserved word "options" followed by a list of one or more option bindings within braces. Each option binding specifies the setting of one option. The same option may not be set multiple times.

Options may be specified either here in the grammar file, or from the command line. If the option is set from the command line, that takes precedence.

Option names are not case-sensitive.

---

```
option_binding ::= "LOOKAHEAD" "=" java_integer_literal ";"
                 | "CHOICE_AMBIGUITY_CHECK" "=" java_integer_literal ";"
                 | "OTHER_AMBIGUITY_CHECK" "=" java_integer_literal ";"
                 | "STATIC" "=" java_boolean_literal ";"
                 | "DEBUG_PARSER" "=" java_boolean_literal ";"
                 | "DEBUG_LOOKAHEAD" "=" java_boolean_literal ";"
                 | "DEBUG_TOKEN_MANAGER" "=" java_boolean_literal ";"
                 | "OPTIMIZE_TOKEN_MANAGER" "=" java_boolean_literal ";"
                 | "ERROR_REPORTING" "=" java_boolean_literal ";"
                 | "JAVA_UNICODE_ESCAPE" "=" java_boolean_literal ";"
                 | "UNICODE_INPUT" "=" java_boolean_literal ";"
                 | "IGNORE_CASE" "=" java_boolean_literal ";"
                 | "USER_TOKEN_MANAGER" "=" java_boolean_literal ";"
                 | "USER_CHAR_STREAM" "=" java_boolean_literal ";"
                 | "BUILD_PARSER" "=" java_boolean_literal ";"
                 | "BUILD_TOKEN_MANAGER" "=" java_boolean_literal ";"
                 | "SANITY_CHECK" "=" java_boolean_literal ";"
                 | "FORCE_LA_CHECK" "=" java_boolean_literal ";"
                 | "COMMON_TOKEN_ACTION" "=" java_boolean_literal ";"
                 | "CACHE_TOKENS" "=" java_boolean_literal ";"
                 | "OUTPUT_DIRECTORY" "=" java_string_literal ";"
```

- **LOOKAHEAD:** The number of tokens to look ahead before making a decision at a choice point during parsing. The default value is 1. The smaller this number, the faster the parser. This number may be overridden for specific productions within the grammar as described later. See the description of the lookahead algorithm for complete details on how lookahead works.
- **CHOICE_AMBIGUITY_CHECK:** This is an integer option whose default value is 2. This is the number of tokens considered in checking choices of the form "A | B | ..." for ambiguity. For example, if there is a common two token prefix for both A and B, but no common three token prefix, (assume this option is set to 3) then JavaCC can tell you to use a lookahead of 3 for disambiguation purposes. And if A and B have a common three token prefix, then JavaCC only tell you that you need to have a lookahead of 3 *or more*. Increasing this can give you more comprehensive ambiguity information at the cost of more processing time. For large grammars such as the Java grammar, increasing this number any further causes the checking to take too much time.
- **OTHER_AMBIGUITY_CHECK:** This is an integer option whose default value is 1. This is the number of tokens considered in checking all other kinds of choices (i.e., of the

forms "(A)*", "(A)+", and "(A)?") for ambiguity. This takes more time to do than the choice checking, and hence the default value is set to 1 rather than 2.

- **STATIC:** This is a boolean option whose default value is true. If true, all methods and class variables are specified as static in the generated parser and token manager. This allows only one parser object to be present, but it improves the performance of the parser. To perform multiple parses during one run of your Java program, you will have to call the ReInit() method to reinitialize your parser if it is static. If the parser is non-static, you may use the "new" operator to construct as many parsers as you wish. These can all be used simultaneously from different threads.
- **DEBUG_PARSER:** This is a boolean option whose default value is false. This option is used to obtain debugging information from the generated parser. Setting this option to true causes the parser to generate a trace of its actions. Tracing may be disabled by calling the method disable_tracing() in the generated parser class. Tracing may be subsequently enabled by calling the method enable_tracing() in the generated parser class.
- **DEBUG_LOOKAHEAD:** This is a boolean option whose default value is false. Setting this option to true causes the parser to generate all the tracing information it does when the option DEBUG_PARSER is true, and in addition, also causes it to generated a trace of actions performed during lookahead operation.
- **DEBUG_TOKEN_MANAGER:** This is a boolean option whose default value is false. This option is used to obtain debugging information from the generated token manager. Setting this option to true causes the token manager to generate a trace of its actions. This trace is rather large and should only be used when you have a lexical error that has been reported to you and you cannot understand why. Typically, in this situation, you can determine the problem by looking at the last few lines of this trace.
- **OPTIMIZE_TOKEN_MANAGER:** This is a boolean option whose default value is false. Setting this to true causes token manager optimizations to be performed.
- **ERROR_REPORTING:** This is a boolean option whose default value is true. Setting it to false causes errors due to parse errors to be reported in somewhat less detail. The only reason to set this option to false is to improve performance.
- **JAVA_UNICODE_ESCAPE:** This is a boolean option whose default value is false. When set to true, the generated parser uses an input stream object that processes Java Unicode escapes (\u...) before sending characters to the token manager. By default, Java Unicode escapes are not processed.
  This option is ignored if either of options USER_TOKEN_MANAGER, USER_CHAR_STREAM is set to true.
- **UNICODE_INPUT:** This is a boolean option whose default value is false. When set to true, the generated parser uses uses an input stream object that reads Unicode files. By default, ASCII files are assumed.
  This option is ignored if either of options USER_TOKEN_MANAGER, USER_CHAR_STREAM is set to true.
- **IGNORE_CASE:** This is a boolean option whose default value is false. Setting this option to true causes the generated token manager to ignore case in the token specifications and the input files. This is useful for writing grammars for languages such as HTML. It is also possible to localize the effect of IGNORE_CASE by using an alternate mechanism described later.
- **USER_TOKEN_MANAGER:** This is a boolean option whose default value is false. The default action is to generate a token manager that works on the specified grammar tokens. If this option is set to true, then the parser is generated to accept tokens from any token manager of type "TokenManager" - this interface is generated into the generated parser directory.
- **USER_CHAR_STREAM:** This is a boolean option whose default value is false. The default action is to generate a character stream reader as specified by the options JAVA_UNICODE_ESCAPE and UNICODE_INPUT. The generated token manager receives characters from this stream reader. If this option is set to true, then the token manager is generated to read characters from any character stream reader of type "CharStream.java". This file is generated into the generated parser directory.
  This option is ignored if USER_TOKEN_MANAGER is set to true.

- **BUILD_PARSER:** This is a boolean option whose default value is true. The default action is to generate the parser file ("MyParser.java" in the above example). When set to false, the parser file is not generated. Typically, this option is set to false when you wish to generate only the token manager and use it without the associated parser.
- **BUILD_TOKEN_MANAGER:** This is a boolean option whose default value is true. The default action is to generate the token manager file ("MyParserTokenManager.java" in the above example). When set to false the token manager file is not generated. The only reason to set this option to false is to save some time during parser generation when you fix problems in the parser part of the grammar file and leave the lexical specifications untouched.
- **SANITY_CHECK:** This is a boolean option whose default value is true. JavaCC performs many syntactic and semantic checks on the grammar file during parser generation. Some checks such as detection of left recursion, detection of ambiguity, and bad usage of empty expansions may be suppressed for faster parser generation by setting this option to false. Note that the presence of these errors (even if they are not detected and reported by setting this option to false) can cause unexpected behavior from the generated parser.
- **FORCE_LA_CHECK:** This is a boolean option whose default value is false. This option setting controls lookahead ambiguity checking performed by JavaCC. By default (when this option is false), lookahead ambiguity checking is performed for all choice points where the default lookahead of 1 is used. Lookahead ambiguity checking is not performed at choice points where there is an explicit lookahead specification, or if the option LOOKAHEAD is set to something other than 1. Setting this option to true performs lookahead ambiguity checking at *all* choice points regardless of the lookahead specifications in the grammar file.
- **COMMON_TOKEN_ACTION:** This is a boolean option whose default value is false. When set to true, every call to the token manager's method "getNextToken" (see the description of the Java Compiler Compiler API) will cause a call to a used defined method "CommonTokenAction" after the token has been scanned in by the token manager. The user must define this method within the TOKEN_MGR_DECLS section. The signature of this method is:
- `void CommonTokenAction(Token t)`
- **CACHE_TOKENS:** This is a boolean option whose default value is false. Setting this option to true causes the generated parser to lookahead for extra tokens ahead of time. This facilitates some performance improvements. However, in this case (when the option is true), interactive applications may not work since the parser needs to work synchronously with the availability of tokens from the input stream. In such cases, it's best to leave this option at its default value.
- **OUTPUT_DIRECTORY:** This is a string valued option whose default value is the current directory. This controls where output files are generated.

---

| | | |
|---|---|---|
| production | ::= | javacode_production |
| | \| | regular_expr_production |
| | \| | bnf_production |
| | \| | token_manager_decls |

There are four kinds of productions in JavaCC. javacode_production and bnf_production are used to define the grammar from which the parser is generated. regular_expr_production is used to define the grammar tokens - the token manager is generated from this information (as well as from inline token specifications in the parser grammar). token_manager_decls is used to introduce declarations that get inserted into the generated token manager.

---

javacode_production ::= "JAVACODE"

*java_return_type java_identifier* "(" *java_parameter_list* ")"
*java_block*

The JAVACODE production is a way to write Java code for some productions instead of the usual EBNF expansion. This is useful when there is the need to recognize something that is not context-free or for whatever reason is very difficult to write a grammar for. An example of the use of JAVACODE is shown below. In this example, the non-terminal "skip_to_matching_brace" consumes tokens in the input stream all the way upto a matching closing brace (the opening brace is assumed to have been just scanned):

```
JAVACODE
void skip_to_matching_brace() {
  Token tok;
  int nesting = 1;
  while (true) {
    tok = getToken(1);
    if (tok.kind == LBRACE) nesting++;
    if (tok.kind == RBRACE) {
      nesting--;
      if (nesting == 0) break;
    }
    tok = getNextToken();
  }
}
```

Care must be taken when using JAVACODE productions. While you can say pretty much what you want with these productions, JavaCC simply considers it a black box (that somehow performs its parsing task). This becomes a problem when JAVACODE productions appear at choice points. For example, if the above JAVACODE production was referred to from the following production:

```
void NT() :
{}
{
  skip_to_matching_brace()
|
  some_other_production()
}
```

Then JavaCC would not know how to choose between the two choices. On the other hand, if the JAVACODE production is used at a non-choice point as in the following example, there is no problem:

```
void NT() :
{}
{
  "{" skip_to_matching_brace()
|
  "(" parameter_list() ")"
}
```

When JAVACODE productions are used at choice points, JavaCC will print a warning message stating this fact. You will then have to insert some explicit LOOKAHEAD specifications to help JavaCC. See the minitutorial on LOOKAHEAD for a detailed guide on such issues.

---

bnf_production ::= *java_return_type java_identifier* "(" *java_parameter_list* ")" ":"
*java_block*
"{" expansion_choices "}"

The BNF production is the standard production used in specifying JavaCC grammars. Each BNF production has a left hand side which is a non-terminal specification. The BNF production then defines this non-terminal in terms of BNF expansions on the right hand side. The non-terminal is written exactly like a method is declared in Java. Since each non-terminal is translated into a method in the generated parser, this style of writing the non-terminal makes this association obvious. The name of the non-terminal is the name of the method, and the parameters and return value declared are the means to pass values up and down the parse tree. As will be seen later, non-terminals on the right hand sides of productions are written as method calls, so the passing of values up and down the tree are done using exactly the same paradigm as method call and return.

There are two parts on the right hand side of an BNF production. The first part is a set of arbitrary Java declarations and code (the Java block). This code is generated at the beginning of the method generated for the Java non-terminal. Hence, everytime this non-terminal is used in the parsing process, these declarations and code are executed. The declarations in this part is visible to all Java code in actions in the BNF expansions. JavaCC does not do any processing of these declarations and code, except to skip to the matching ending brace, collecting all text encountered on the way. Hence, a Java compiler can detect errors in this code that has been processed by JavaCC.

The second part of the right hand side are the BNF expansions. This is described later.

---

regular_expr_production ::= [ lexical_state_list ]
regexpr_kind [ "[" "IGNORE_CASE" "]" ] ":"
"{" regexpr_spec ( "|" regexpr_spec )* "}"

A regular expression production is used to define lexical entities that get processed by the generated token manager. A detailed description of how the token manager works is provided in this minitutorial (click here). This page describes the syntactic aspects of specifying lexical entities, while the minitutorial describes how these syntactic constructs tie in with how the token manager actually works.

A regular expression production starts with a specification of the lexical states for which it applies (the lexical state list). There is a standard lexical state called "DEFAULT". If the lexical state list is omitted, the regular expression production applies to the lexical state "DEFAULT".

Following this is a description of what kind of regular expression production this is (see below for what this means).

After this is an optional "[IGNORE_CASE]". If this is present, the regular expression production is case insensitive - it has the same effect as the IGNORE_CASE option, except that in this case it applies locally to this regular expression production.

This is then followed by a list of regular expression specifications that describe in more detail the lexical entities of this regular expression production.

---

token_manager_decls ::= "TOKEN_MGR_DECLS" ":" *java_block*

The token manager declarations starts with the reserved word "TOKEN_MGR_DECLS" followed by a ":" and then a set of Java declarations and statements (the Java block). These declarations and statements are written into the generated token manager and is accessible from within lexical actions. See the minitutorial on the token manager for more details.

There can only be one token manager declaration in a JavaCC grammar file.

---

lexical_state_list ::= "<" "*" ">"
| "<" *java_identifier* ( "," *java_identifier* )* ">"

The lexical state list describes the set of lexical states for which the corresponding regular expression production applies. If this is written as "<*>", the regular expression production applies to all lexical states. Otherwise it applies to all the lexical states in the identifier list within the angular brackets.

---

regexpr_kind ::= "TOKEN"
| "SPECIAL_TOKEN"
| "SKIP"
| "MORE"

This specifies the kind of regular expression production. There are four kinds:

- **TOKEN**: The regular expressions in this regular expression production describe *tokens* in the grammar. The token manager creates a Token object for each match of such a regular expression and returns it to the parser.
- **SPECIAL_TOKEN**: The regular expressions in this regular expression production describe *special tokens*. Special tokens are like tokens, except that they do not have significance during parsing - that is the BNF productions ignore them. Special tokens are, however, still passed on to the parser so that parser actions can access them. Special tokens are passed to the parser by linking them to neighboring real tokens using the field "specialToken" in the Token class. Special tokens are useful in the processing of lexical entities such as comments which have no significance to parsing, but still are an important part of the input file. See the minitutorial on the token manager for more details of special token handling.
- **SKIP**: Matches to regular expressions in this regular expression production are simply skipped (ignored) by the token manager.
- **MORE**: Sometimes it is useful to gradually build up a token to be passed on to the parser. Matches to this kind of regular expression are stored in a buffer until the next TOKEN or SPECIAL_TOKEN match. Then all the matches in the buffer and the final TOKEN/SPECIAL_TOKEN match are concatenated together to form one TOKEN/SPECIAL_TOKEN that is passed on to the parser. If a match to a SKIP regular expression follows a sequence of MORE matches, the contents of the buffer is discarded.

---

regexpr_spec ::= regular_expression [ *java_block* ] [ ":" *java_identifier* ]

The regular expression specification begins the actual description of the lexical entities that are part of this regular expression production. Each regular expression production may contain any number of regular expression specifications.

Each regular expression specification contains a regular expression followed by a Java block (the lexical action) which is optional. This is then followed by an identifier of a lexical state (which is also optional). Whenever this regular expression is matched, the lexical action (if any) gets executed, followed by any common token actions. Then the action depending on the regular expression production kind is taken. Finally, if a lexical state is specified, the token manager moves to that lexical state for further processing (the token manager starts initially in the state "DEFAULT").

---

expansion_choices ::= expansion ( "|" expansion )*

Expansion choices are written as a list of one or more expansions separated by "|"s. The set of legal parses allowed by an expansion choice is a legal parse of any one of the contained expansions.

---

expansion ::= ( expansion_unit )*

An expansion is written as a sequence of expansion units. A concatenation of legal parses of the expansion units is a legal parse of the expansion.

For example, the expansion "{" decls() "}" consists of three expansion units - "{", decls(), and "}". A match for the expansion is a concatenation of the matches for the individual expansion units - in this case, that would be any string that begins with a "{", ends with a "}", and contains a match for decls() in between.

---

expansion_unit ::= local_lookahead
| *java_block*
| "(" expansion_choices ")" [ "+" | "*" | "?" ]
| "[" expansion_choices "]"
| [ *java_assignment_lhs* "=" ] regular_expression
| [ *java_assignment_lhs* "=" ] *java_identifier* "(" *java_expression_list* ")"

An expansion unit can be a local LOOKAHEAD specification. This instructs the generated parser on how to make choices at choice points. For details on how LOOKAHEAD specifications work and how to write LOOKAHEAD specifications, click here to visit the minitutorial on LOOKAHEAD.

An expansion unit can be a set of Java declarations and code enclosed within braces (the Java block). These are also called *parser actions*. This is generated into the method parsing the non-terminal at the appropriate location. This block is executed whenever the parsing process crosses this point successfully. When JavaCC processes the Java block, it does not perform any detailed syntax or semantic checking. Hence it is possible that the Java compiler will find errors in your actions that have been processed by JavaCC. *Actions are not executed during lookahead evaluation.*

An expansion unit can be a parenthesized set of one or more expansion choices. In which case, a legal parse of the expansion unit is any legal parse of the nested expansion choices. The parenthesized set of expansion choices can be suffixed (optionally) by:

- **"+":** Then any legal parse of the expansion unit is one or more repetitions of a legal parse of the parenthesized set of expansion choices.
- **"*":** Then any legal parse of the expansion unit is zero or more repetitions of a legal parse of the parenthesized set of expansion choices.
- **"?":** Then a legal parse of the expansion unit is either the empty token sequence or any legal parse of the nested expansion choices. An alternate syntax for this construct is to enclose the expansion choices within brackets "[...]".

An expansion unit can be a regular expression. Then a legal parse of the expansion unit is any token that matches this regular expression. When a regular expression is matched, it creates an object of type Token. This object can be accessed by assigning it to a variable by prefixing the regular expression with "variable =". In general, you may have any valid Java assignment left-hand side to the left of the "=". *This assignment is not performed during lookahead evaluation.*

An expansion unit can be a non-terminal (the last choice in the syntax above). In which case, it takes the form of a method call with the non-terminal name used as the name of the method. A successful parse of the non-terminal causes the parameters placed in the method call to be operated on and a value returned (in case the non-terminal was not declared to be of type "void"). The return value can be assigned (optionally) to a variable by prefixing the regular expression with "variable =". In general, you may have any valid Java assignment left-hand side to the left of the "=". *This assignment is not performed during lookahead evaluation.* Non-terminals may not be used in an expansion in a manner that introduces left-recursion. JavaCC checks this for you.

---

local_lookahead ::= "LOOKAHEAD" "(" [ *java_integer_literal* ] [ "," ] [ expansion_choices ] [ "," ] [ "{" *java_expression* "}" ] ")"

A local lookahead specification is used to influence the way the generated parser makes choices at the various choice points in the grammar. A local lookahead specification starts with the reserved word "LOOKAHEAD" followed by a set of lookahead constraints within parentheses. There are three different kinds of lookahead constraints - a lookahead limit (the integer literal), a syntactic lookahead (the expansion choices), and a semantic lookahead (the expression within braces). At least one lookahead constraint must be present. If more than one lookahead constraint is present, they must be separated by commas.

For a detailed description of how lookahead works, please click here to visit the minitutorial on LOOKAHEAD. A brief description of each kind of lookahead constraint is given below:

- **Lookahead Limit:** This is the maximum number of tokens of lookahead that may be used for choice determination purposes. This overrides the default value which is specified by the LOOKAHEAD option. This lookahead limit applies only to the choice point at the location of the local lookahead specification. If the local lookahead specification is not at a choice point, the lookahead limit (if any) is ignored.
- **Syntactic Lookahead:** This is an expansion (or expansion choices) that is used for the purpose of determining whether or not the particular choice that this local lookahead specification applies to is to be taken. If this was not provided, the parser uses the expansion to be selected during lookahead determination. If the local lookahead specification is not at a choice point, the syntactic lookahead (if any) is ignored.

- **Semantic Lookahead:** This is a boolean expression that is evaluated whenever the parser crosses this point during parsing. If the expression evaluates to true, the parsing continues normally. If the expression evaluates to false and the local lookahead specification is at a choice point, the current choice is not taken and the next choice is considered. If the expression evaluates to false and the local lookahead specification is *not* at a choice point, then parsing aborts with a parse error. Unlike the other two lookahead constraints that are ignored at non-choice points, semantic lookahead is always evaluated. If fact semantic lookahead is even evaluated if it is encountered during the evaluation of some other syntactic lookahead check (for more details click here to visit the minitutorial on LOOKAHEAD).

**Default values for lookahead constraints:** If a local lookahead specification has been provided, but not all lookahead constraints have been included, then the missing ones are assigned default values as follows:

- If the lookahead limit is not provided and if the syntactic lookahead is provided, then the lookahead limit defaults to the largest integer value (2147483647). This essentially implements "infinite lookahead" - namely, look ahead as many tokens as necessary to match the syntactic lookahead that has been provided.
- If neither the lookahead limit nor the syntactic lookahead has been provided (which means the semantic lookahead is provided), the lookahead limit defaults to 0. This means that syntactic lookahead is not performed (it passes trivially), and only semantic lookahead is performed.
- If the syntactic lookahead is not provided, it defaults to the choice that the local lookahead specification applies to. If the local lookahead specification is not at a choice point, then the syntactic lookahead is ignored - hence a default value is not relevant.
- If the semantic lookahead is not provided it defaults to the boolean expression "true". That is it trivially passes.

---

regular_expression ::= *java_string_literal*
| "<" [ [ "#" ] *java_identifier* ":" ] complex_regular_expression_choices ">"
| "<" *java_identifier* ">"
| "<" "EOF" ">"

There are two places in a grammar files where regular expressions may be written:

- Within a regular expression specification (part of a regular expression production),
- As an expansion unit with an expansion. When a regular expression is used in this manner, its as if the regular expression were defined in the following manner at this location and then referred to by its label from the expansion unit:

```
      <DEFAULT> TOKEN :
      {
        regular expression
      }
```

That is, this usage of regular expression can be rewritten using the other kind of usage.

The complete details of regular expression matching by the token manager is available in the minitutorial on the token manager. The description of the syntactic constructs follows.

The first kind of regular expression is a string literal. The input being parsed matches this regular expression if the token manager is in a lexical state for which this regular expression applies and the next set of characters in the input stream is the same (possibly with case ignored) as this string literal.

A regular expression may also be a more complex regular expression using which more involved regular expression (than string literals can be defined). Such a regular expression is placed within angular brackets "<...>", and may be labeled optionally with an identifier. This label may be used to refer to this regular expression from expansion units or from within other regular expressions. If the label is preceded by a "#", then this regular expression may not be referred to from expansion units, but only from within other regular expressions. When the "#" is present, the regular expression is referred to as a "private regular expression".

A regular expression may be a reference to some other labeled regular expression in which case it is written as the label enclosed in angular brackets "<...>".

Finally a regular expression may be a reference to the predefined regular expression "<EOF>" which is matched by the end of file.

Private regular expressions are not matched as tokens by the token manager. Their purpose is solely to facilitate the definition of other more complex regular expressions.

Consider the following example defining Java floating point literals:

```
TOKEN :
{
  < FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)?
(["f","F","d","D"])?
      | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
      | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
      | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
  >
|
  < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
}
```

In this example, the token FLOATING_POINT_LITERAL is defined using the definition of another token, namely, EXPONENT. The "#" before the label EXPONENT indicates that this exists solely for the purpose of defining other tokens (FLOATING_POINT_LITERAL in this case). The definition of FLOATING_POINT_LITERAL is not affected by the presence or absence of the "#". However, the token manager's behavior is. If the "#" is omitted, the token manager will erroneously recognize a string like E123 as a legal token of kind EXPONENT (instead of IDENTIFIER in the Java grammar).

---

complex_regular_expression_choices ::= complex_regular_expression ( "|" complex_regular_expression )*

Complex regular expression choices is made up of a list of one or more complex regular expressions separated by "|"s. A match for a complex regular expression choice is a match of any of its constituent complex regular expressions.

---

complex_regular_expression ::= ( complex_regular_expression_unit )*

A complex regular expression is a sequence of complex regular expression units. A match for a complex regular expression is a concatenation of matches to the complex regular expression units.

---

complex_regular_expression_unit ::= *java_string_literal*
| "<" *java_identifier* ">"
| character_list
| "(" complex_regular_expression_choices ")" [ "+" | "*" | "?" ]

A complex regular expression unit can be a string literal. In which case there is exactly one match for this unit, namely, the string literal itself.

A complex regular expression unit can be a reference to another regular expression. The other regular expression has to be labeled so that it can be referenced. The matches of this unit are all the matches of this other regular expression. Such references in regular expressions cannot introduce loops in the dependency between tokens.

A complex regular expression unit can be a character list. A character list is a way of defining a set of characters. A match for this kind of complex regular expression unit is any character that is allowed by the character list.

An complex regular expression unit can be a parenthesized set of complex regular expression choices. In which case, a legal match of the unit is any legal match of the nested choices. The parenthesized set of choices can be suffixed (optionally) by:

- **"+":** Then any legal match of the unit is one or more repetitions of a legal match of the parenthesized set of choices.
- **"*":** Then any legal match of the unit is zero or more repetitions of a legal match of the parenthesized set of choices.
- **"?":** Then a legal match of the unit is either the empty string or any legal match of the nested choices.

Note that unlike the BNF expansions, the regular expression "[...]" is not equivalent to the regular expression "(...)?". This is because the [...] construct is used to describe character lists in regular expressions.

---

character_list ::= [ "~" ] "[" [ character_descriptor ( "," character_descriptor )* ] "]"

A character list describes a set of characters. A legal match for a character list is any character in this set. A character list is a list of character descriptors separated by commas within square brackets. Each character descriptor describes a single character or a range of characters (see character descriptor below), and this is added to the set of characters of the character list. If the character list is prefixed by the "~" symbol, the set of characters it represents is any UNICODE character not in the specified set.

---

character_descriptor ::= *java_string_literal* [ "-" *java_string_literal* ]

A character descriptor can be a single character string literal, in which case it describes a singleton set containing that character; or it is two single character string literals separated by a "-", in which case, it describes the set of all characters in the range between and including these two characters.