CompTools 2.0 : A Compiler Generator for C and Java

Gilbert Babin

Service d'enseignement des technologies de l'information
HEC Montréal
Montréal, Québec, Canada
Gilbert.Babin@hec.ca

October 14, 2004

1 Introduction

The CompTools package (Compiler Construction Tools) is composed of three modules developed to: (1) build a lexical analyzer (LexGen), (2) build an LL(1) grammar interpreter (CompGen), and (3) draw syntax diagrams corresponding to a grammar or a lexicon (DiagGen).

In order to understand how the software works, we first need to explain some computer language concepts. A language is defined by three elements: the lexicon, the syntax, and the semantics. The lexicon is the set of all the words and symbols used by the language. The syntax describes how to arrange the lexicon elements to produce phrases. The semantics define what the phrases mean. Hence, a meaningful phrase will obey all the semantic and syntactic rules, and will only contain words available in the lexicon.

We need to analyze a phrase to determine its meaning. The analysis is performed using (1) a lexical analyzer, which will detect the words on the input stream, (2) a syntactic analyzer, which will validate the syntax of the phrase, and (3) a semantics analyzer, used to determine the meaning of the phrase and detect any invalid use of the language that might be allowed by the syntax, but have no meaning (e.g., My dog Spot is air).

We distinguish two types of words: literals and lexical constructs. Literals represent actual words. In most computer languages, literals are reserved words; e.g., in C, `for`, `while`, `==`, `!=`, etc. Hence, a literal is a series of characters that must be matched exactly in a phrase. On the other hand, lexical constructs represent classes of words or symbols that match a certain pattern; e.g., an integer or a variable identifier. We represent the pattern defining a lexical construct with a regular expression.

The literals and the lexical constructs are used when defining the syntax of the language. However, these two elements are not sufficient, because they do not allow for very complex languages. We complement the literals and the lexical constructs with categories. A category is a syntax rule, defined by a regular expression that can include literals, lexical constructs, and categories, hence allowing for recursive definition of the language.

2 The Lexical Analyzer Builder: LexGen

The goal of a lexical analyzer is to recognize literals and lexical constructs on the input stream. How does it work? The lexical analyzer is actually a deterministic finite state automaton (FSA) that reads the input stream and moves from state to state. As it moves through the FSA, it stacks up the different states it encounters until it reaches an invalid state. Then, based on a list of expected literals and lexical constructs, it determines the largest literal or lexical construct matched in the list of expected symbols in the current context.

The LexGen module builds a FSA to recongnize a lexicon, either in C or Java. In C, it builds (1) a C file containing the routines to use the FSA, i.e., the lexical analyzer routines, (2) C files containing the description of the FSA, (3) a C header file containing the declarations of the variables used to describe the FSA, and (4) a C header file

containing the information needed by other programs using the lexical analyzer. In Java, it builds (1) a Java class that implements the routines to use the FSA, i.e., the lexical analyzer routines, (2) a Java class that implements the FSA tables, specific to the current language, (3) a Java interface declaring lexical tokens common to all lexical analyzers, and (4) a Java interface declaring lexical tokens specific to the current language.

The FSA generated is usually optimized to reduce its size; the optimization step can be skipped when building the FSA. A debug option generates a trace within the lexical analyzer routine. In C, the lexical analyzer can be built to read from a file (FILE *) or from a character string (char **). In Java, this issue is resolved by using a `BufferedReader` object as the input to the lexical analyzer. Hence, any input format that can be converted to a `BufferedReader` may be used. The way LexGen is designed, the creation of the lexical analyzer routines and declarations is optional. It might occur in certain situations that the same program uses more than one language. Yet, only one set of lexical analyzer routines is needed to process both languages.

2.1 Segmenting the Lexical Analyzer Tables in C

The size of the lexical analyzer table produced may be quite large. In fact, the data structures produced might extend over the segment size limits, especially on small computers. To avoid this problem, the `$segment` operator can be used. It defines the size of a segment, in terms of variables of type `long`. Hence, a value of 8000 indicates that a segment is the size of 8000 variables of type `long`. This approach is used because the actual size of a variable of type `long` changes from machine to machine; LexGen, however, can determine the total number of variables of type `long` it needs.

2.2 The LexGen Command

To call the lexical analyzer builder, we use the following command (see Tab. 1):

For generating C code:

```
lexgen [-o] [-l] [-d] [-s] [-cl <.c file>] [-hl <.h file>]
                           [-ct <.c file>] [-ht <.h file>] <lexic file>
```

For generating Java code:

```
lexgen [-j] [-o] [-l] [-d] [-jl <.java file>] [-js <.java file>]
                           [-ji <.java file>] [-jt <.java file>] <lexic file>
```

3 The Syntax and Semantics Analyzer Builder: CompGen

The CompGen module is primarily used to create a syntax analyzer for a language defined using an LL(1) grammar. A grammar is LL(1) when at each decision point, only the current lexical element is needed to determine which direction to take in the analysis. Because of their structure, syntax analyzers for LL(1) grammars are easily programmed, using recursive functions, one for each category of the grammar. The structure of the functions reflect the syntax itself. Hence, there is a direct translation of the grammar definition into the syntax analyzer.

However, this is not the only use of CompGen. In addition to syntax analysis, the analyzer generated can perform semantics analysis. This is not automatically generated, but rather `programmed` into the grammar definition by inserting actions to be performed during the syntax analysis. These actions are C or Java statements that are placed at the exact point in the grammar where they should logically be performed.

The grammar description file contains: (1) the syntax definition, (2) the declaration of the actions to be included in the syntax analyzer, and (3) generation parameters. The grammar is made of a set of categories, each of which is defined using regular expressions composed of literals, lexical constructs, and categories. The description of the literals and lexical constructs is stored in a lexicon description file (the same used by LexGen). For example, using a Backus-Naur Form (BNF) syntax, a simple language to evaluate prefix expressions would be:

Table 1: Parameters and options for the `lexgen` command

Parameter:

`<lexic file>`	The `lexgen` command takes one parameter: the lexicon description file. The file must have a `.lex` extension. See Section 5.1 for a detailed description of the content of this file.

C generation options:

`-o`	Does not perform the optimization of the FSA.
`-l`	Does not generate the lexical analyzer routines and declaration. The user should still use the `-cl` and `-hl` options to set the name of the files containing the lexical analyzer routines (`-cl`) and declarations (`-hl`).
`-d`	Activates the debugging information in the C output file.
`-s`	Uses `char **` instead of `FILE *` as the type of the input to the lexical analyzer. It will also change the name of the lexical analyzer routines to differentiate them from the routines needed to read from a file, to avoid name conflicts.
`-cl <.c file>`	Specifies the name of the C file containing the lexical analyzer routines. By default, the name of the `.c` file will be generated based on the `.lex` file; e.g., `prefix.lex` will generate the C file named `prefix_lex.c`.
`-hl <.h file>`	Specifies the name of the header file containing the declarations of the lexical routines. By default, the name of the `.h` file will be generated based on the `.lex` file; e.g., `prefix.lex` will generate the header file named `prefix_lex.h`.
`-ct <.c file>`	Specifies the name of the C file(s) containing the FSA structures. By default, the name of the `.c` file will be generated based on the `.lex` file; e.g., `prefix.lex` will generate either a unique C file named `prefix_fsa.c` or a series of C files named `prefix_fsa1.c`, `prefix_fsa2.c`, ...
`-ht <.h file>`	Specifies the name of the header file containing the declarations of the variables describing the FSA. By default, the name of the `.h` file will be generated based on the `.lex` file; e.g., `prefix.lex` will generate the header file named `prefix_fsa.h`.

Java generation options:

`-j`	Generates Java code instead of C code.
`-o`	Does not perform the optimization of the FSA.
`-l`	Does not generate the lexical analyzer routines and declaration. The user should still use the `-jl`, `-js`, `-ji`, and `-jt` options to set the file names properly.
`-d`	Activates the debugging information in the Java output file.
`-jl <.java file>`	Specifies the name of the Java file containing the lexical analyzer routines. By default, the name of the `.java` file will be generated based on the `.lex` file; e.g., `prefix.lex` will generate the Java file named `prefixLex.java`.
`-js <.java file>`	Specifies the name of the Java file defining the generic tokens used by all lexical analyzers generated. By default, the name of the `.java` file will be generated based on the `.lex` file; e.g., `prefix.lex` will generate the Java file named `prefixLexSymbols.java`.
`-ji <.java file>`	Specifies the name of the Java file containing the FSA structures for the current language. By default, the name of the `.java` file will be generated based on the `.lex` file; e.g., `prefix.lex` will generate a Java file named `prefixInstLex.java`.
`-jt <.java file>`	Specifies the name of the Java file containing the declarations of the tokens specific to the current language. By default, the name of the `.java` file will be generated based on the `.lex` file; e.g., `prefix.lex` will generate the Java file named `prefixInstLexSymbols.java`.

```
<expressions> ::= /[ <expression> || ',' ]/ '.'  ;
<expression> ::= <atom> | <operator> <expression> <expression> ;
<atom> ::= integer ;
<operator> ::= '+' | '-' | '*' | '/' ;
```

A prefix expression is defined as either an atom (a single value) or an operator followed by two operands (which are expressions). In this particular case, an atom is simply an integer value, and the valid operators are + and -, for addition and subtraction, respectively.

If we only provide this information to CompGen, it will generate a C module that contains the routines needed to analyze the syntax of an expression. However, no other action will be performed. Therefore, we need to know how to add actions in the grammar. First, we must declare the actions. The actions are pieces of C or Java code to be inserted in the grammar. In the prefix language example, we need actions for (1) declaring local variables for the different categories (or routines), (2) assigning values to these variables, and (3) performing the operations. For a Java program, The resulting grammar will be:

```
@myInteger // declaration of inner class for passing Integer values
$begin_action
    public class myInteger {
       private Integer value ;
       public myInteger(int i) {
          value = new Integer(i);
       }
       public void setValue(int i) {
          value = new Integer(i);
       }
       public int getValue() {
          return value.intValue();
       }
       public Integer getInteger() {
          return value;
       }
    }
$end_action

@declare // declaration of local variables
$begin_action
    myInteger val2=new myInteger(0);
    myInteger operator=new myInteger(0);
$end_action

@assign_val // assigns the content of the lexical analyzer's result to val
$begin_action
    val.setValue((val.getInteger()).parseInt(inst_lexical.value));
$end_action

@assign_times // if operator is 0, then it is a multiplication
$begin_action
    operator.setValue(0);
$end_action

@assign_divide // if operator is 1, then it is a division
$begin_action
    operator.setValue(1);
$end_action

@assign_plus // if operator is 2, then it is an addition
$begin_action
    operator.setValue(2);
$end_action
```

```
@assign_minus // if operator is 3, then it is an addition
$begin_action
    operator.setValue(3);
$end_action

@do_operation // executes the actual operation
$begin_action
    if (operator.getValue() == 0)
       val.setValue(val.getValue() * val2.getValue());
       else if (operator.getValue() == 1)
       val.setValue(val.getValue() / val2.getValue());
       else if (operator.getValue() == 2)
       val.setValue(val.getValue() + val2.getValue());
       else
       val.setValue(val.getValue() - val2.getValue());
$end_action

@print_result // prints out the result
$begin_action
    System.out.println("="+(val.getInteger()).toString());
$end_action

@declare_val // declares local variable val
$begin_action
    myInteger val=new myInteger(0);
$end_action

@exit // error handling routine
$begin_action
    // empty error management routine...
$end_action

$initial <expressions> /* the initial category is <expressions> */
$global @myInteger
$error @exit
$name Prefix

<expressions> @declare_val ::= /[ <expression> ("val") @print_result || ',' ]/ '.'  ;
<expression> ("myInteger val" "val" ) @declare ::=
    <atom> ("val") |
    <operator> ("operator") <expression> ("val") <expression> ("val2") @do_operation;
<atom> ("myInteger val" "val") ::= integer @assign_val ;
<operator> ("myInteger operator" "operator") ::=
    '+' @assign_plus |
    '-' @assign_minus |
    '*' @assign_times |
    '/' @assign_divide ;
```

Here, we declared a number of actions. Notice that the `@declare` action is placed before the `::=` literal; in this case, the actions are performed prior to any other actions in the function declared for the category. The other actions are executed after the analysis of the subexpression they follow, and before the subexpression they precede, as shown in the resulting Java files (Appendix A).

Note also in the example, the use of parameters (declaration) and arguments (function call). The parameters are actual C or Java variable declarations, while the arguments are the actual C or Java expressions used when calling a function. Finally, we have defined the category `<expressions>` as the initial category of the grammar, i.e., the entry point for the analysis, by using the `$initial` operator.

3.1 Segmenting the Syntax and Semantics Analyzer Routines in C

The size of the syntax and semantics analyzer routines produced might extend over the segment size limits, especially on small computers. To avoid this problem, the `$skip` operator can be used. This operator indicates the end of a source file. In other words, if no `$skip` operator is found, only one source file will be produced. Each `$skip` operator will force the creation of a new source file. This way, the user can manually determine where to cut the source files in order for the resulting compiled files to fit within the segments.

3.2 System-defined Variables in C

There are a certain number of reserved names, used by the syntax analyzer. These variables have specific types. However, the user can change some of those types using special operators defines in the grammar definition language. The following variables can be used within each category:

`f` The input stream. The type of `f` can be modified using the `$file` operator. The default type of `f` is `FILE *`.

`token` The value of the token currently being analyzed, as returned by the lexical analyzer. The type of `token` is a pointer to a `long` (`long *`).

`value` The string matching the current token. The type of `value` can be modified using the `$value` operator. The default type of `value` is a pointer to a `char *` (`char **`).

`prev_char` The look-ahead buffer of the lexical analyzer. This buffer is used to hold the extra characters needed in determining the current token value and type. The type of `prev_char` can be modified using the `$previous` operator. The default type of `prev_char` is a pointer to a `char *` (`char **`).

`line_count` An index on the current line in the input buffer. The type of `line_count` is a pointer to a `long` (`long *`).

`char_count` An index on the current character in the input buffer. It is reset to 1 at each new line. The type of `char_count` is a pointer to a `long` (`long *`).

`context` A universal pointer used to hold user-specified data to pass to the error handling routine, when invoked. The type of context is `void *`.

`token_names` A table of external token names, as defined in the lexicon description file. This table is useful when generating error messages. The type of `token_names` is a table of strings (`char *[]`).

The code generation is made assuming that LexGen will be used to build the lexical analyzer. This is not necessarily the case, however. That is why we allow for changes in the type of some of the parameters to the lexical routine, hence to the categories. If the `-s` option of LexGen is used, the type of the input stream must be changed to `char **`. In addition to these variables, the different categories can also access the user defined parameters and local variables.

If an error is found in the course of the syntax analysis, an error handling routine is called. The actual name of this routine is dependent on the grammar itself; its name will be (1) __name_error__, where name is the string specified by the `$name` operator, if used, or (2) __initial_ error__, where initial is the name of the initial category of the grammar. The content of that routine is left to the user of CompGen. The `$error` operator let the user assign actions to that routine. The parameters accessible by the user in the error handling routine are:

`f` The input stream. The type of `f` can be modified using the `$file` operator. The default type of `f` is `FILE *`.

`token` The value of the token currently being analyzed, as returned by the lexical analyzer. The type of `token` is a pointer to a `long` (long *).

`tokens` A list of the tokens that where expected when the error routine was called. The type of `tokens` is `long []`.

`name_of_parent` The name of the category where the error occurred. The type of `name_of_parent` is `char *`.

`input_buffer` The content of the input buffer of the lexical analyzer. It is actually the value of 'prev_char' when the error occurred. It has the same type as 'prev_char' (char **, by default).

`line_count` An index on the current line in the input buffer. The type of `line_count` is a `long`.

`char_count` An index on the current character in the input buffer. It is reset to 1 at each new line. The type of `char_count` is `long`.

`context` A universal pointer used to hold user-specified data to pass to the error handling routine. The type of context is `void *`.

`token_names` A table of external token names, as defined in the lexicon description file. This table is useful when generating error messages. The type of `token_names` is a table of strings (char *[]).

Finally, the syntax analyzer is invoked using the syntax analyzer execution routine. The actual name of this routine is dependent on the grammar itself; its name will be (1) execute_ name_analyzer, where name is the string specified by the `$name` operator, if used, or (2) execute_initial_analyzer, where initial is the name of the initial category of the grammar. The syntax analyzer execution routine uses at least one parameter:

`f` The input stream. The type of `f` can be modified using the `$file` operator. The default type of `f` is `FILE *`.

The other parameters are identical to the parameters declared for the initial category.

3.3 System-defined Variables in Java

In Java, the lexical and syntactical analyzers are defined as a series of classes and interfaces, as illustrated in Figure 1.

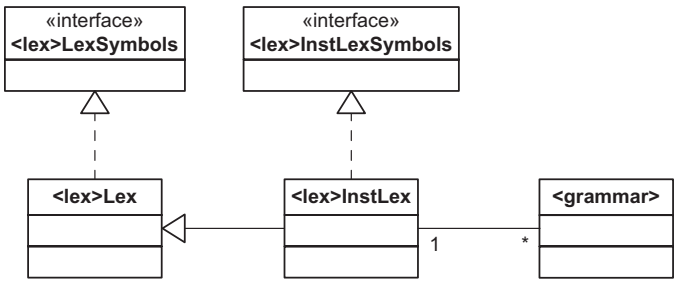


Figure 1: Classes generated by CompGen and LexGen

The `<lex>LexSymbols` interface defines generic tokens used for any lexical analyzer, where `<lex>` stands for the lexicon name. The `<lex>Lex` class implements all the lexical analyzer routines, independant of a specific language, and as such may be reused if multiple languages are defined in a single system. The `<lex>InstLexSymbols` interface defines language-specific tokens, while the `<lex>InstLex` defines a language-specific specialization of the The `<lex>Lex` class.

The syntax analyzer is implemented in the `<grammar>` class and provides a number of instance variables and methods:

`inst_lexical` This variable is a reference to the lexical analyzer. The type of this object is `<lex>InstLex`. The user-defined actions may access any of public methods of this class, including:

`assign_input(BufferedReader)` Used to modify the input stream for this lexical analyzer.

`getCharCount()` Returns the character position on the current line.

`getLineCount()` Returns the number of the current line.

`getPrevChar()` Returns the look-ahead buffer of the lexical analyzer.

`getValue()` Returns the actual value of the current token.

`context` A reference to an object used to manage the analyzer's context, such as the name of the current category being processed.

`token` The numeric value of the current token.

`token_names` A table of `String` objects which associates a name to the different numeric values that a token may take.

`execute_analyzer(BufferedReader)` Runs the syntax analyzer on a `BufferedReader` object.

`getPrevChar()` Returns the look-ahead buffer of the lexical analyzer (same as `inst_lexical.getPrevChar()`).

3.4 Exception Handling: Ambiguous Grammars

As we mentioned earlier, LL(1) grammars are such that at each branching point, only one symbol is necessary to determine what direction to take. This characteristic is called the LL(1) condition. However, this condition might not always be achieved. The CompGen program will still generate the appropriate syntax analyzer. When detecting an ambiguous category (i.e., at least one branching point does not comply with the LL(1) condition), it will declare a special local variable called `ambiguity`. At each ambiguous condition, it will add a test on the value of `ambiguity`, hence making every condition distinct. It is the task of the grammar designer to either change the grammar to avoid the ambiguity or assign the appropriate value to `ambiguity`, at the appropriate place. Whenever CompGen informs the user of an ambiguous category, the user should inspect the program generated to determine where to place the assignment to `ambiguity` and what value to assign in the different contexts.

3.5 The CompGen Command

To call the syntax and semantics analyzer builder, we use the following command (Tab. 2):

For generating C code:

```
compgen [-d] [-c <.c file>] [-h <.h file>] [-hl <.h file>]
             [-ht <.h file>] [-t <.lex file>] <.grm file>
```

For generating Java code:

```
compgen [-j] [-d] [-jc <.java file>] [-ji <.java file>] [-t <.lex file>] <.grm file>
```

4 The Syntax Diagram Builder: DiagGen

This program is only used for visualization and documentation purposes. It takes the grammar definition used by the syntax and semantics analyzer builder, or the lexicon description file, and generates Encapsulated PostScript (EPS) files illustrating the definition of categories, literals, and lexical constructs. These files can be edited by software that can read and edit EPS files (e.g., Corel Draw, Adobe Illustrator, etc.) and can be placed for printing in a variety of software.

Table 2: Parameters and options for the `compgen` command

Parameter:

`<.grm file>`	The `compgen` command takes only one parameter, which is the grammar syntax description file. The file must have a `.grm` extension. See Section 5.2 for a detailed description of the content of this file.

C generation options:

`-d`	Activates the debugging information in the C output file.
`-c <.c file>`	Specifies the name of the C file produced. By default, the name of the `.c` file will be generated based on the `.grm` file; e.g., `prefix.grm` will generate the unique C file named `prefix.c` or, if `$skip` is used, the series of C files named `prefix1.c`, `prefix2.c`...
`-h <.h file>`	Specifies the name of the header file containing the declarations needed by the syntax analyzer. By default, the name of the `.h` file will be generated based on the `.grm` file; e.g., `prefix.grm` will use the header file named `prefix.h`.
`-hl <.h file>`	Specifies the name of the header file containing the declarations lexical analyzer routines, usually generated by LexGen. By default, the name of the `.h` file will be generated based on the `.grm` file; e.g., `prefix.grm` will use the header file named `prefix_lex.h`.
`-ht <.h file>`	Specifies the name of the header file containing the FSA description, usually generated by LexGen. By default, the name of the `.h` file will be generated based on the `.grm` file; e.g., `prefix.grm` will use the header file named `prefix_fsa.h`.
`-t <.lex file>`	Specifies the name of the lexicon description file to be used. By default, the name of the `.lex` file will be generated based on the `.grm` file; e.g., `prefix.grm` will use the header file named `prefix.lex`.

Java generation options:

`-j`	Generates Java code instead of C code.
`-d`	Activates the debugging information in the Java output file.
`-jc <.java file>`	Specifies the name of the Java file produced. By default, the name of the `.java` file will be generated based on the `.grm` file; e.g., `prefix.grm` will generate the unique C file named `prefix.java`.
`-ji <.java file>`	Specifies the name of the Java file containing the FSA structures for the current language, usually generated by LexGen. By default, the name of the `.java` file will be generated based on the `.lex` file; e.g., `prefix.lex` will generate a Java file named `prefixInstLex.java`.
`-t <.lex file>`	Specifies the name of the lexicon description file to be used. By default, the name of the `.lex` file will be generated based on the `.grm` file; e.g., `prefix.grm` will use the header file named `prefix.lex`.

Table 3: Parameters and options for the `diaggen` command

Parameter:	
`<input file>`	The `diaggen` command takes only one parameter, which is a grammar syntax description file or a lexicon description file. The file must have a `.grm` or a `.lex` extension. See Sections 5.1 and 5.2 for a detailed description of the content of these files.
Options:	
`-one <name>`	Instructs DiagGen to print the category, literal, or lexical construct named ¡name¿.
`-limit <height>`	Instructs DiagGen to print the categories, literals, or lexical constructs in multiple files, each file containing the syntax diagram of many categories, literals, or lexical constructs. The image generated will not exceed ¡height¿ points. This option is the default, with ¡limit¿ set to 700 points (with 72 points/inch).
`-all`	Instructs DiagGen to print the categories, literals, or lexical constructs in multiple files, each file containing the syntax diagram of only one category, literal, or lexical construct.
`-e <.eps file>`	Specifies the file pattern used to generate the name of the EPS and PICT files generated. By default, the name of the `.eps` files will be generated based on the `.grm` file; e.g., `prefix.grm` will generate the files `prefix1.eps`, `prefix2.eps`, ...

4.1 Configuring DiagGen

The look of the diagrams produced can be modified by changing the configuration file, `diaggen.cfg`. This file contains parameters used by DiagGen to generate the diagrams. The structure of this file is defined in Section 5.3.

When installing the DiagGen module, a mapping must be made between the extended ASCII used by the local machine and the extended ASCII defined by Adobe for the PostScript language. The structure of this file is defined in the `Postscript Extended ASCII Correspondence Table File` Section 5.4.

4.2 The DiagGen Command

To call the syntax diagram builder, we use the following command (Tab. 3:

```
diaggen [-one <name>|-limit <height>|-all][-e <.eps file>] <input file>
```

5 Input files for CompTools

In this section, we describe the different file formats used by CompTools. Since most of these file formats follow LL(1) grammar syntax, we first need to define a grammar syntax to describe these file formats. We start with the following naming convention: (1) strings between `<` and `>` are categories (e.g., `<grammar>`), (2) strings between single quotes (`'`) are reserved words (e.g., `'$name'`), (3) strings containing letters and underscores (_) are lexical constructs (e.g., action_name). All other strings are operators used in the definition of the file formats. These operators are:

String	Meaning
`::=`	Start of category definition.
`;`	End of category definition.
`*[ ]*`	Zero or more repetitions of the subexpression included inside `*[` and `]*`.
`+[ ]+`	One or more repetitions of the subexpression included inside `+[` and `]+`.
`-[ ]-`	Zero or one repetition of the subexpression included inside `-[` and `]-`.
`[ ]`	A choice sub-expression. One subexpression within the list of choices should be selected.
`|`	Separation between choices within a category or within a choice subexpression.
`/[ || ]/`	One or more repetitions of the subexpression included inside `/[` and `||`. Each repetition is separated by the subexpression included inside `||` and `]/`.

5.1 The Lexicon Description File

The lexicon description file is used for the generation of (1) the lexical analyzer and (2) the syntax and semantics analyzer. In the first case, it provides the definition of the different literals and lexical constructs, while in the second case, it maps these definitions with different aliases referring to the same lexical construct that could be used within the grammar syntax definition.

The syntax of the lexicon description file is as follows:

```
<lexicon> ::=
    *[ [
        '$extension' extension |
        '$name' lexic_name |
        '$packagename' package_name |
        '$include' action_code |
        '$segment' integer
    ] ]*
    *[ <lex_unit definition> ]* ;
```

At the top of the file, we first define some parameters: (1) a suffix to be used for the lexical analyzer routines to differentiate between different usage like reading from a file or from a string (`$extension`), (2) the name of the lexicon (`$name`), used to uniquely identify the lexical analyzer tables (in C) or the classes and interfaces generated (in Java), (3) the name of the Java package containing the lexical analyzer (in Java only), (4) declarations to be inserted in the header file containing the lexical analyzer routine declarations (`$include`), and (5) the maximum number of variables of type `long` that can be placed in an individual data segment (`$segment`). Then follows the list of lexical unit definitions.

```
<lex_unit definition> ::= token_id '(' -[ super_token_id ]- ')' error_string
    '(' +[ <lexical unit alias> ]+ ')'
    regular_expression ;
```

The definition of one lexical unit is composed of the token name (`token_id`), an optional super token name (`super_token_id`), a name used to refer to a specific token in the syntax analyzer (`error_string`), a series of aliases (lexical units with the same syntax but used in different contexts in the syntax description of a language), and a regular expression defining the syntax of the token itself. Here are some examples of token definitions:

```
IDSYM () "identifier" (variable identifier) [a.z,A.Z,_]*[a.z,A.Z,_,0.9];
SCSYM () ";" (';') \;
BEGSYM (IDSYM) "begin" ('begin') begin
ENDSYM () "end" ('end') end|(a.zA.Z_0.9)
```

`IDSYM` is defined using `[a.z,A.Z,_]*[a.z,A.Z,_,0.9];`. `IDSYM` is formed of any letter (upper and lower case) or underscore, followed by any number of any letter, any digit, or underscore. Hence, `IDSYM` is a lexical construct.

`SCSYM` is defined using `\;`. `SCSYM` is the `;` character itself. `SCSYM` is a literal.

`BEGSYM` is defined using `begin`. `BEGSYM` is the string `begin`. `BEGSYM` is a literal, but is defined as one instance of the possible values of the lexical construct `IDSYM`.

ENDSYM is defined using end|(a.zA.Z_0.9). ENDSYM is the string end. To be recognized, it must be followed by any character different from a letter, a digit, or an underscore, otherwise, it would be recognized as an IDSYM.

The token name is the internal identifier by which the lexical analyzer and the syntax and semantics analyzers refer to the different literals or lexical constructs. The token names are unique, implying that no literal or lexical construct can have the same token name as another literal or lexical construct. If multiple languages are used within the same program, the user must ensure that no two tokens have the same identifier.

The super token name is used solely to reduce the size of the FSA. In certain situations, certain literals are actually one element of a set of lexical constructs. In our examples, the lexical construct IDSYM is the set of all strings beginning by a letter or an underscore and followed by a letter, underscore, or digit. It is quite obvious that begin and end both match that description. However, for the FSA to determine that end was encountered, we must specify that the literal end is the string e, n, d followed by any character but a letter, underscore or digit. The description of begin is simpler because we declare it as an element of the lexical construct IDSYM.

When generating the syntax and semantics analyzer, we build a global table called token_names, which is the name of the token for the world outside the syntax and semantics analyzer. For instance, token_names[IDSYM] yields the string identifier. This is useful when generating an error message with the error handling routine in the syntax analyzer.

```
<lexical unit alias> ::= literal | lexical_construct ;
```

The list of aliases is used to map names used in the grammar description with the token name. For readability reasons, it might be more appropriate to use the word variable in the grammar syntax than identifier to illustrate the actual usage of the identifier. Each alias is either a literal or a lexical construct.

In the lexicon description file grammar described above, we have used a number of lexical constructs. These lexical constructs have a strict syntax.

- An integer is defined as one or more digits.
- An error_string is defined as any character between double quotes ". A double quote can be included in the string by using \". The \ character itself is included by using \\.
- A literal is refered to in the grammar as a series of characters between single quotes '. A single quote can be included in the string by using \'. The \ter itself is included by using \\.
- The lexical constructs lexic_name, token_id, super_token_id, and lexical_construct are defined as a series of character different from the set :, =, |, ;, (,), +, [, -, *,], $, @, ", <, ', >, /, newline, tab, and space. Furthermore, it cannot start with a digit (0 to 9).
- The construct action_code is defined as the shortest string of character, beginning by $begin keyword, and ending by the $end keyword. It is used to enter C statements in the lexicon definition.

5.1.1 Defining a Regular Expression

The regular expression declaration starts at the first non-white character after the alias list and ends at the first newline encountered, unless the newline is preceeded by a \. It describes how to recognize a certain token (literal or lexical construct) on the input stream. The declaration of the construct regular_expression is more complex than other constructs. In fact, to show the meaning of the regular expression, we must present the grammar used to create one. A regular expression E is defined as:

```
<expression> ::= +[ <subexpression> -[ ':'  ]- ]+ -[ '|' +[ -[ ':'  ]- <subexpression> ]+ ]- ;
```

Here, the category <subexpression> is equivalent to the definition of S, below. The vertical bar is used to indicate the end of the lexical element being analysed; it means that the lexical analyzer must encounter the second sequence of subexpressions before accepting the token; the second list of subexpressions is called the anticipation. The colon is used by DiagGen to perform a line change when drawing the syntax diagram for the lexical unit.

A subexpression S is defined as any of the followings:

1. L The character L itself (except special characters).
2. L_1`.`L_2 Any character from L_1 to L_2 (L_2 included).
3. `\`L Special characters:

   `\a` Margin bell
   `\b` Backspace
   `\f` Formfeed
   `\n` Newline
   `\r` Linefeed
   `\s` Significant space
   `\t` Right tab
   `\v` Vertical tab
   \newline Continue on the next line

   Other characters: The character L itself. Used when one of the syntax special characters must be used.
4. `\xhh` The character with hexadecimal value `hh`.
5. `\Xhhhh` The character with hexadecimal value `hhhh` (for Unicode values).
6. `\ddd` The character with decimal value `ddd`.
7. `(`$S_1 S_2 \ldots S_n$`)` Any character except the ones defined by subexpressions $S_1, \ldots, S_n$. The subexpressions $S_1, \ldots, S_n$ can only be of the formats (1) through (6) defined above.
8. $S_1 S_2 \ldots S_n$ Concatenation of subexpressions $S_1, \ldots, S_n$.
9. `+`S`;` Subexpression S is repeated at least once.
10. `*`S`;` Subexpression S is repeated zero or more times.
11. `-`S`;` Subexpression S is optional (zero or one repetition).
12. `/`S_1`=`S_2`;` Subexpression S_1 is repeated at least once. Each repetition is separated by expression S_2.
13. `[`$S_1, \ldots, S_n$`]` Only one subexpression within $S_1, \ldots, S_n$ is used (a choice).
14. `#`N`#`S`;` Subexpression S is repeated exactly N times.
15. `#`N`,`M`#`S`;` Subexpression S is repeated at least N times, and at most M times.

5.2 The Grammar Syntax Description File

The grammar syntax description file contains: (1) the declaration of actions, (2) the declaration of the grammar's global parameters, and (3) the description of the grammar syntax itself. We will now define the syntax of the grammar syntax file.

The grammar syntax is defined using the grammar syntax itself as follows:

```
<grammar> ::=
    *[ <action declaration> ]*
    *[ <grammar parameters> ]*
    +[ <category> -[ '$skip' ]- ]+ ;
```

The category <grammar> is the main category of the grammar definition file syntax. When defining a grammar, we first must define all the actions to be used, if any. Then, we declare parameters to CompGen. Finally, we define the set of all categories.

```
<grammar parameters> ::=
    '$result' string |
    '$name' lexical_construct |
    '$file' string |
    '$previous' string |
    '$initial' category_name |
    '$error' +[ action_name ]+ |
    '$global' +[ action_name ]+ |
    '$include' +[ action_name ]+ ;
    '$packagename' string ;
    '$packages' +[ string ]+ ;
    '$interfaces' +[ string ]+ ;
    '$constructor' +[ action_name ]+ ;
```

The grammar parameters are the application dependent variables used by CompGen. The are (1) the type of the variable containing the current lexical symbol (`$result`), (2) the name of the grammar (`$name`), (3) the type of the input stream, e.g., `FILE *` or `char **` (`$file`) (in C only), (4) the type of the lexical analyzer buffer (`$previous`) (in C only), (5) the name of the initial category, the default being the first category in the list of category definitions (`$initial`), (6) the list of actions defining the error routine (`$error`), (7) global initialisations (`$global`), (8) global declarations (`$include`), (9) the name of the Java package containing this syntax analyzer, (10) the list of packages to import into the class generated, (11) the list of interfaces that this syntax analyzer must implement, and (12) actions to add to the constructor of the syntax analyzer.

For `$file`, we define the actual type, as it will be used in the analysis routines. In the case of `$result` and `$previous`, we define the type of the value, not the type of the pointer to that value (`char *` instead of `char **`). The operators `$global` and `$include` are both used to insert C or Java statements in the syntax and semantics analyzer produced. The distinction is only seen when generating C files, where the `$global` operator will define the C statements to include only in the first `.c` file produced, whereas the `$include` operator will define the C statements to include in the `.h` file which is included in all the `.c` file produced.

```
<action declaration> ::= action_name action_code ;
```

This category defines how to declare an action to be inserted in the generated analyzer.

```
<category> ::= <category defn> *[ action_name ]*
    '::=' <alternative>
       [ *[ ':'  <alternative> ]* | *[ '|' <alternative> ]* ]
    ';' ;
```

The category description allows for the declaration of the name and parameters of the category, as well as initialization actions; e.g., local variable declaration and initialization. The category must include at least one alternative, the alternatives being separated either by `:` (line break indicator for DiagGen) or `|` (different alternatives within the same category).

```
<category defn> ::= category_name -[ <parameters> ]- ;
```

The category definition includes the name of the category and the declaration of the parameters of the category.

```
<parameters> ::= '(' /[ declaration variable || ',' ]/ ')' ;
```

The parameters are the declarations of C or Java variables to be received by the categories in addition to the standard parameters generated by CompGen. The `declaration` is the actual declaration of the variable, while `variable` corresponds to the name of the variable being declared. The different declarations are comma-separated.

```
<alternative> ::= *[ action_name ]*
    +[ [
       <one or more> |
```

```
    <optional> |
    <zero or more> |
    <separated> |
    <choices> |
    <category call> |
    reserved_word |
    lexical_construct
] *[ action_name ]* ]+ ;
```

An alternative is one expression describing the syntax of a series of categories and subexpressions. Actions can be inserted around each subexpressions.

```
<category call> ::= category_name -[ <arguments> ]- ;
```

The category definition includes the name of the category and the arguments used in the call to the category.

```
<arguments> ::= '(' /[ expression || ',' ]/ ')' ;
```

The arguments are the actual values and/or variables passed to the categories, when called. The different arguments are comma-separated.

```
<one or more> ::= '+[' <alternative> ']+' ;
```

This type of alternative specifies that the subexpression (`<alternative>`) is to be found one or more times.

```
<optional> ::= '-[' <alternative> ']-' ;
```

This type of alternative specifies that the subexpression (`<alternative>`) is optional (zero or one only). If an action is placed after a `]-` literal, it may begin by `else`. In this case, the action would be executed only if the alternative is not used.

```
<zero or more> ::= '*[' <alternative> ']*' ;
```

This type of alternative specifies that the subexpression (`<alternative>`) is to be found zero or more times.

```
<separated> ::= '/[' <alternative> '||' <alternative> ']/' ;
```

This type of alternative specifies that the first subexpression will appear at least once. Every time it is repeated, the repetitions are separated by the second subexpression.

```
<choices> ::= '[' /[ <alternative> || '|' ]/ ']' ;
```

This type of alternative defines a list of one or more subexpressions (`<alternative>`) to choose from (one and only one is chosen within the list).

In the grammar above, we have used a number of lexical constructs. These lexical constructs have a strict syntax.

- A `string` is defined as any character between double quotes `"`. A double quote can be included in the string by using `\"`. The `\` character itself is included by using `\\`. Constructs `declaration`, `variable`, and `expresssion` are of this type as well.
- A `lexical_construct` is defined as a series of character different from the set `:`, `=`, `|`, `;`, `(`, `)`, `+`, `[`, `-`, `*`, `]`, `$`, `@`, `"`, `<`, `'`, `>`, `/`, newline, tab, and space. Furthermore, it cannot start with a digit (0 to 9).
- A `category_name` is defined as any character other than `<` and `>` between the characters `<` and `>`.
- An `action_name` is a series of character different from the set `:`, `=`, `|`, `;`, `(`, `)`, `+`, `[`, `-`, `*`, `]`, `$`, `@`, `"`, `<`, `'`, `>`, `/`, newline, tab, and space, preceded by an `@` sign.
- The construct `action_code` is defined as the shortest string of character, beginning by `$begin_action` keyword, and ending by the `$end_action` keyword. It is used to enter C statements in the grammar definition.
- A `reserved_word` (or literal) is refered to in the grammar as a series of characters between single quotes `'`. A single quote can be included in the string by using `\'`. The `\` character itself is included by using `\\`.

```
<comments> ::= *[ [ comments | eoln_comments ] ]* ;
```

Finally, comments can be placed anywhere within the grammar. Two forms of comments are accepted: comments and eoln_comments. The construct comments is defined as the shortest string of character beginning by `/*` and ending by `*/`, not including the string `/*`. The construct eoln_comments is a string starting by `//` and ending with a newline.

5.3 The Diagram Builder Configuration File

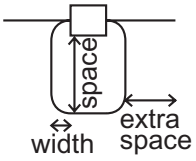
This section describe the syntax of the configuration file (`diaggen.cfg`) for the diagram builder DiagGen. The configuration file is used to change the appearance of the diagrams built by DiagGen. These parameters include the look of repetition loops, selection lists, text boxes, character intervals, anticipation marker, connection lines, category layout, character style, and the type of output expected.

```
<configuration> ::=
   /[ [
      <loop> |
      <choice> |
      <text> |
      <elipsis> |
      <separation bar> |
      <line> |
      <white box> |
      <category> |
      <style> |
      <output>
   ] || ';' ]/ '.'  ;
```

The configuration file is composed of 10 sections: loop configuration, choice configuration, text configuration, elipsis configuration, separation bar configuration, line configuration, output configuration, white box configuration, and style configuration. The sections are separated by semi-colons, the last section ending with a period. Within a section, elements are comma-separated. Note that the order of the sections is not important.

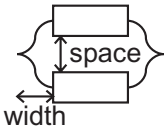
```
<loop> ::= 'loop' 'configuration' ':'
   /[ [ 'space' integer | 'width' integer | 'extra' 'space' integer ] || ',' ]/ ;
```

The loop configuration section defines the parameters to draw a repetition loop. The meaning of `space`, `width`, and `extra space` is illustrated below. Default values are 3 points for `space`, 4 points for `width`, and 2 points for `extra space`.




```
<choice> ::= 'choice' 'configuration' ':'
   /[ [ 'space' integer | 'width' integer ] || ',' ]/ ;
```

The choice configuration section defines the parameters to draw a selection list. The meaning of `space` and `width` is illustrated below. Default values are 3 points for `space` and 11 points for `width`.

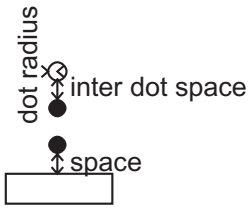



```
<text> ::= 'text' 'configuration' ':'
   /[ [ 'space' integer | 'width' integer | 'shift' integer ] || ',' ]/ ;
```

The text configuration section defines the parameters to draw a text box. The meaning of `space` and `width` is illustrated below. Default values are 5 points for `space` and 5 points for `width`.

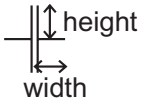
```
<elipsis> ::= 'elipsis' 'configuration' ':'
    /[ [ 'space' integer | 'dot' 'radius' integer | 'inter' 'dot' 'space' integer ] || ',' ]/ ;
```

The elipsis configuration section defines the parameters to draw an elipsis, marking an interval of possible selections. The meaning of `space`, `dot radius`, and `inter dot space` is illustrated below. Default values are 3 points for `space`, 1 points for `dot radius`, and 2 points for `inter dot space`.



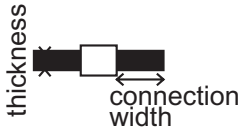

```
<separation bar> ::= 'separation' 'bar' 'configuration' ':'
    /[ [ 'width' integer | 'height' integer ] || ',' ]/ ;
```

The separation bar configuration section defines the parameters to draw a marker indicating the start of the anticipation for lexical units. The meaning of `width` and `height` is illustrated below. Default values are 3 points for `width` and 10 points for `height`.



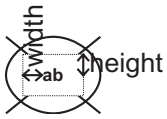

```
<line> ::= 'line' 'configuration' ':'
    /[ [ 'connection' 'width' integer | 'thickness' real ] || ',' ]/ ;
```

The line configuration section defines the parameters to draw lines. The meaning of `connection width` and `thickness` is illustrated below. Default values are 2 points for `connection width` and 0.5 points for `thickness`.




```
<white box> ::= 'white' 'box' 'configuration' ':'
    /[ [ 'width' real | 'height' real ] || ',' ]/ ;
```

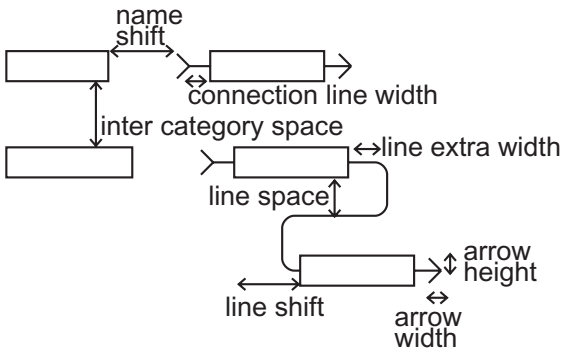
The white box configuration section defines the parameters to draw a white box. This white box is used to hide part of two lines indicating that a character is not selected. The meaning of `width` and `height` is illustrated below. Default values are 0.5 points for `width` and 2.5 points for `height`.




```
<category> ::= 'category' 'configuration' ':'
    /[ [
        'name' 'shift' integer |
        'line' 'shift' integer |
        'line' 'space' integer |
        'line' 'extra' 'width' integer |
        'inter' 'category' 'space' integer |
```

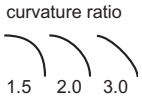
```
    'arrow' 'width' integer |
    'arrow' 'height' integer |
    'connection' 'line' 'width' integer
    ] || ',' ]/ ;
```

The category configuration section defines the parameters to draw category definitions. The meaning of `name shift`, `line shift`, `line space`, `line extra width`, `inter category space`, `arrow width`, `arrow height`, and `connection line width` is illustrated below. Default values are 3 points for `name shift`, 5 points for `line shift`, 6 points for `line space`, 2 points for `line extra width`, 20 points for `inter category space`, 2 points for `arrow width`, 2 points for `arrow height`, and 5 points for `connection line width`.




```
<style> ::= 'style' 'configuration' ':'
    /[ [
        'curvature' 'ratio' real |
        'trace' 'box' [ 'TRUE' | 'FALSE' ] |
        'uppercase' 'width' real |
        'lowercase' 'width' real |
        'category' 'typeface' 'postscript' string |
        'literal' 'typeface' 'postscript' string |
        'construct' 'typeface' 'postscript' string
        ] || ',' ]/ ;
```

The style configuration section defines miscelaneous parameters. The `curvature ratio` is used to control the curvature of curved lines in the produced diagram. The higher the `curvature ratio`, the less pronounced is the curve, as illustrated below. The default value of the `curvature ratio` is 1.9.



The `trace box` option enables or disables the drawing of characteristic boxes around categories (square box), literals (oval), and lexical constructs (rounded box). The default value of `trace box` is `TRUE`. The `uppercase width` and `lowercase width` options are used to evaluate the actual width of the characters in proportional typefaces. Default values are 2.4 for `uppercase width` and 1.75 for `lowercase width`. The typeface options (`category`, `literal`, and `construct`) are used to select the typefaces representing categories, literals, and lexical constructs.

5.4 The Postscript Extended ASCII Correspondence Table File

The ASCII character set is standard for characters from 0 to 127. However, characters from 128 to 255 are not included in the standard. The meaning of these characters change from one machine to the other. A correspondance between the local machine and the standard adopted by Adobe in PostScript must therefore be made. The file `diaggen.tbl` contains the mapping of extended ASCII characters for use by DiagGen. This file must be properly modified to match the local system.

Each line indicates: (1) the local system character matching the PostScript character and (2) the Postscript identifier corresponding to that character. The idea is to type in the actual character matching the character defined by the PostScript identifier.

For example, we have the lines for `Egrave` and `egrave`, which correspond to an uppercase `E` with a grave accent, and a lowercase `e` with a grave accent, respectively. The Adobe PostScript word are actually typographical descriptions of the characters. If no character of the local system match the character, we type in a `#` sign.

è egrave

`#` Egrave

In this example, the local (hypothetical) system has a character corresponding to `egrave`, but not to `Egrave`.

A Result from CompGen in Java

This appendix contains the files produced by CompGen, based on the grammar example of Section 3. The `Test.java` file is used as a Java test application. The other files were generated by CompGen and LexGen.

Program 1 *Test.java*

```
import java.net.*;
import java.util.*;
import java.io.*;

public class Test {
   public Prefix stringAnalyzer;
   /**
   * Constructor.
   */
   public Test() {
      stringAnalyzer = new Prefix();
   }
   public static void main(String argv[]) {
      System.out.println("enter expressions (separated by ',' and ending by '.'  followed by ctrl-D)");
      (new Test()).stringAnalyzer.execute_analyzer(new BufferedReader(new InputStreamReader(System.in)));
   }
}
```

Program 2 *Prefix.java*

```
//  Lexicon file:  prefix.lex
//  Grammar file:  prefix.grm

import java.io.*;
import java.util.*;

/**
* This class contains the syntactic/semantic analyzer.
*/

public class Prefix {
   //  make an instance of the lexical analyzer
   private PrefixInstLex inst_lexical;

   private boolean __DEBUG_SYNTACTIC__ = false;
   private int debug_level = 0;

   private Object context;
   private int token;

   //  begining action @myInteger
   public class myInteger {
```

```
    private Integer value ;
    public myInteger(int i) {
        value = new Integer(i);
    }
    public void setValue(int i) {
        value = new Integer(i);
    }

    public int getValue() {
        return value.intValue();
    }
    public Integer getInteger() {
        return value;
    }
}
//  ending action @myInteger

private String[] token_names = new String[] {
    "No symbol",
    "End-of-file",
    "integer",
    "+",
    "-",
    "*",
    "/",
    ",",
    "."
} ;

private int[] token000 = new int[] {
    inst_lexical.INT, inst_lexical.PLUS, inst_lexical.MINUS, inst_lexical.TIMES,
    inst_lexical.DIVIDE, inst_lexical.___NOSYM___
} ;
private int[] token001 = new int[] {
    inst_lexical.___EOF___, inst_lexical.___NOSYM___
} ;
private int[] token002 = new int[] {
    inst_lexical.INT, inst_lexical.PLUS, inst_lexical.MINUS, inst_lexical.TIMES,
    inst_lexical.DIVIDE, inst_lexical.COMMA, inst_lexical.PERIOD,
    inst_lexical.___NOSYM___
} ;
private int[] token003 = new int[] {
    inst_lexical.INT, inst_lexical.___NOSYM___
} ;
private int[] token004 = new int[] {
    inst_lexical.PLUS, inst_lexical.MINUS, inst_lexical.TIMES, inst_lexical.DIVIDE,
    inst_lexical.___NOSYM___
} ;

/**
* The constructor.
*/
public Prefix() {
    context = new Object();
    inst_lexical = new PrefixInstLex();
}

private void __debug__(int lev, String str) {
    for (int i=0; i<lev; i++)
        System.out.print(' ');
    System.out.println(str);
}
```

```
private void __expressions__() {
   //  begining action @declare_val
   myInteger val=new myInteger(0);
   //  ending action @declare_val

   if (__DEBUG_SYNTACTIC__) {
      __debug__(debug_level, "BEGIN of __expressions__");
      debug_level++;
   }

   __expression__(val);

   //  begining action @print_result
   System.out.println("="+(val.getInteger()).toString());
   //  ending action @print_result

   while (token == inst_lexical.COMMA) {
      token = inst_lexical.lexical(token000);
      if (token == inst_lexical.____NOSYM____)
      __error__();

      __expression__(val);

      //  begining action @print_result
      System.out.println("="+(val.getInteger()).toString());
      //  ending action @print_result
   }

   token = inst_lexical.lexical(token001);
   if (token == inst_lexical.____NOSYM____)
   __error__();

   if (token == inst_lexical.____NOSYM____)
   __error__();

   if (__DEBUG_SYNTACTIC__) {
      debug_level--;
      __debug__(debug_level, "END of __expressions__");
   }
}

private void __expression__(myInteger val) {
   //  begining action @declare
   myInteger val2=new myInteger(0);
   myInteger operator=new myInteger(0);
   //  ending action @declare


   if (__DEBUG_SYNTACTIC__) {
      __debug__(debug_level, "BEGIN of __expression__");
      debug_level++;
   }
   if (token == inst_lexical.INT) {
      __atom__(val);
   } else
   /* if ((token == inst_lexical.PLUS) || (token == inst_lexical.MINUS) || (token == inst_lexical.TIMES)
   || (token == inst_lexical.DIVIDE))
   */
   {
      __operator__(operator);

      __expression__(val);

      __expression__(val2);
```

```
      //  begining action @do_operation
      if (operator.getValue() == 0)
         val.setValue(val.getValue() * val2.getValue());
      else if (operator.getValue() == 1)
         val.setValue(val.getValue() / val2.getValue());
      else if (operator.getValue() == 2)
         val.setValue(val.getValue() + val2.getValue());
      else
         val.setValue(val.getValue() - val2.getValue());
      //  ending action @do_operation
   }

   if (__DEBUG_SYNTACTIC__) {
      debug_level--;
      __debug__(debug_level, "END of __expression__");
   }
}

private void __atom__(myInteger val) {
   if (__DEBUG_SYNTACTIC__) {
      __debug__(debug_level, "BEGIN of __atom__");
      debug_level++;
   }
   //  begining action @assign_val
   val.setValue((val.getInteger()).parseInt(inst_lexical.getValue()));
   //  ending action @assign_val

   token = inst_lexical.lexical(token002);
   if (token == inst_lexical.____NOSYM____)
   __error__();

   if (__DEBUG_SYNTACTIC__) {
      debug_level--;
      __debug__(debug_level, "END of __atom__");
   }
}

private void __operator__(myInteger operator) {
   if (__DEBUG_SYNTACTIC__) {
      __debug__(debug_level, "BEGIN of __operator__");
      debug_level++;
   }
   if (token == inst_lexical.PLUS) {
      //  begining action @assign_plus
      operator.setValue(2);
      //  ending action @assign_plus

      token = inst_lexical.lexical(token000);
      if (token == inst_lexical.____NOSYM____)
      __error__();
   } else if (token == inst_lexical.MINUS) {
      //  begining action @assign_minus
      operator.setValue(3);
      //  ending action @assign_minus

      token = inst_lexical.lexical(token000);
      if (token == inst_lexical.____NOSYM____)
      __error__();
   } else if (token == inst_lexical.TIMES) {
      //  begining action @assign_times
      operator.setValue(0);
```

```
            //  ending action @assign_times

            token = inst_lexical.lexical(token000);
            if (token == inst_lexical.____NOSYM____)
            __error__();
        } else // if (token == inst_lexical.DIVIDE) {
            //  begining action @assign_divide
            operator.setValue(1);
            //  ending action @assign_divide

            token = inst_lexical.lexical(token000);
            if (token == inst_lexical.____NOSYM____)
            __error__();
        }
        if (__DEBUG_SYNTACTIC__) {
            debug_level--;
            __debug__(debug_level, "END of __operator__");
        }
    }

    /**
    * Execute the lexical analyzer.
    */
    public void execute_analyzer(BufferedReader br) {
        inst_lexical.setLineCount(0);
        inst_lexical.setCharCount(0);
        inst_lexical.assign_input(br);
        token = inst_lexical.____NOSYM____;
        token = inst_lexical.lexical(token000);
        __expressions__();
    }

    /**
    * Get prev_char back from the lexical analyzer.
    * @return The StringBuffer prev_char.
    */
    public StringBuffer getPrevChar() {
        return inst_lexical.getPrevChar();
    }

    private void __error__() {
        //  begining action @exit
        //  ending action @exit
    }
}
```

Program 3 *PrefixLexSymbols.java*

```
//  Lexicon file:  prefix.lex

interface PrefixLexSymbols {
    int ____NOSYM____ = 0;
    int ____EOF____ = 1;
}
```

Program 4 *PrefixInstLexSymbols.java*

```
//  Lexicon file:  prefix.lex

interface PrefixInstLexSymbols {
    int INT = 2;
    int PLUS = 3;
    int MINUS = 4;
    int TIMES = 5;
    int DIVIDE = 6;
```

```
    int COMMA = 7;
    int PERIOD = 8;
}
```

Program 5 *PrefixInstLex.java*

```
//  Lexicon file:  prefix.lex

import java.util.*;
/**
* This class contains an instance of the standard
* lexical analyzer and all tables.
*/

class PrefixInstLex extends PrefixLex implements PrefixInstLexSymbols {
    //  constructor
    PrefixInstLex() {
        nb_states = 8;
        state_types = new int[] {
            6,6,6,6,6,6,6,1
        } ;
        final_symbols = new int[] {
            INT, PLUS, MINUS, TIMES, DIVIDE, COMMA, PERIOD, ___NOSYM___
        } ;
        int[] accept_NULL = {
            ___NOSYM___
        } ;
        int[] accept_000 = {
            INT,___NOSYM___
        } ;
        int[] accept_001 = {
            PLUS,___NOSYM___
        } ;
        int[] accept_002 = {
            MINUS,___NOSYM___
        } ;
        int[] accept_003 = {
            TIMES,___NOSYM___
        } ;
        int[] accept_004 = {
            DIVIDE,___NOSYM____
        } ;
        int[] accept_005 = {
            COMMA,___NOSYM___
        } ;
        int[] accept_006 = {
            PERIOD,___NOSYM____
        } ;
        accept_symbols = new int[][] {
            accept_000, accept_001, accept_002, accept_003,
            accept_004, accept_005, accept_006, accept_NULL
        } ;
        int[][] transNULL = {
            {
                FIRST_CHAR, LAST_CHAR, TRASH
            }
        } ;
        int[][] trans000 = {
            {
                48, 57, 0
            } ,
```

```
            {
               FIRST_CHAR, LAST_CHAR, TRASH
            }
         } ;
         int[][] trans007 = {
            {
               42, 42, 3
            } ,
            {
               43, 43, 1
            } ,
            {
               44, 44, 5
            } ,
            {
               45, 45, 2
            } ,
            {
               46, 46, 6
            } ,
            {
               47, 47, 4
            } ,
            {
               48, 57, 0
            } ,
            {
               FIRST_CHAR, LAST_CHAR, TRASH
            }
         } ;
         trans_sizes= new int [] {
            2, 1, 1, 1, 1, 1, 1, 8
         } ;
         trans = new int[][][] {
            trans000,transNULL,
            transNULL,transNULL,
            transNULL,transNULL,
            transNULL,trans007
         } ;

         String name_NULL = "" ;
         name_token = new String[] {
            name_NULL, name_NULL, name_NULL,
            name_NULL, name_NULL, name_NULL, name_NULL,
            name_NULL, name_NULL
         } ;
         int[] synonymNULL = {
            item ___NOSYM___
         } ;
         synonyms = new int [][] {
            synonymNULL, synonymNULL, synonymNULL,
            synonymNULL, synonymNULL, synonymNULL, synonymNULL,
            synonymNULL, synonymNULL
         } ;
      }
}
```

Program 6 *PrefixLex.java*

```
// Lexicon file: prefix.lex
```

```
import java.io.*;
import java.util.*;

/**
* This class contains the standard lexical analyzer.
*/

public class PrefixLex implements PrefixLexSymbols {
   //  some constant values
   protected final int INIT_STATE = 1;
   protected final int FINAL_STATE = 2;
   protected final int ACCEPT_STATE = 4;
   protected final int TRASH = -1;
   protected final int REG_STATE = 0;
   protected final int FIRST_CHAR = 1;
   protected final int LAST_CHAR = 65535;
   protected final int MIN_CHAR = 0;
   protected final int MAX_CHAR = 1;
   protected final int TRANS = 2;

   //  flags
   private boolean __DEBUG_LEXICAL__ = false;
   private boolean eof = false;

   //  variables to be initialized in sub class
   private int token;
   protected int[][][] trans;
   protected int[] trans_sizes;
   private int line_count;  //  the current line
   private int char_count;  //  the current character
   protected int nb_states;
   protected int[] final_symbols;
   protected int[][] accept_symbols;  //  accepted symbols by state (FSA)
   protected int[][] synonyms;
   protected int[] state_types;  //  state types (FSA)
   protected String[] name_token;

   private String value;  //  to store the result token
   private StringBuffer prev_char;
   private StringBuffer sb_value;
   private BufferedReader input;

   /**
   * The constructor.
   */
   public PrefixLex() {
      sb_value = new StringBuffer();  //  initialize global variables
      prev_char = new StringBuffer();
   }

   private boolean is_substring(String string1, String string2, int length) {
      int i;
      for (i=0; (i<length) && (string1.length() >= i) && (string2.length() >= i); i++)
         if (string1.charAt(i) != string2.charAt(i))
            return false;
      return (i == length) && (i == string2.length());
   }

   private boolean valid_token(int token, int[] tokens) {
      int i;
      for (i=0; tokens[i] != ___NOSYM___; i++)
         if (token == tokens[i])
            return true;
      return false;
   }
```

```
private int next_state(int c, int state, int[][][] trans, int[] trans_sizes) {
   int u,l,m;

   l=0;
   u=trans_sizes[state]-1;
   while (l<=u) {
      m=(l+u)/2;
      if (c >= trans[state][m][MAX_CHAR]) l=m+1;
      if (c <= trans[state][m][MAX_CHAR]) u=m-1;
   }
   if (c >= trans[state][u+1][MIN_CHAR])
      return trans[state][u+1][TRANS];
   else
      return TRASH;
}

private int find_initial(int[] state_types, int nb_states) {
   int i;
   for (i=0; i<nb_states; i++)
      if ((state_types[i] & INIT_STATE) == INIT_STATE)
         return i;
   return TRASH;
}

private int search_valid_token(Stack stack, StringBuffer str, int[] tokens) {
   int i, j=0, k, l, final_sym;
   boolean found = false;
   for (i=(stack.indexOf(stack.peek())-1); (i>=0) && (!found); i--) {
      for (j=i, final_sym = ___NOSYM___; (j>=0) && (final_sym == ___NOSYM___); ) {
         if ((state_types[((Integer)stack.elementAt(j)).intValue()] & FINAL_STATE) == FINAL_STATE)
            final_sym = final_symbols[((Integer)stack.elementAt(j)).intValue()];
         else
            j--;
      }
      for ( ; (j>=0) && (!found); ) {
         if ((state_types[((Integer)stack.elementAt(j)).intValue()] & ACCEPT_STATE) == ACCEPT_STATE) {
            for (k=0; (accept_symbols[((Integer)stack.elementAt(j)).intValue()][k] != ___NOSYM___) &&
               !found; ) {
               found = accept_symbols[((Integer)stack.elementAt(j)).intValue()][k] == final_sym;
               if (!found)
               k++;
            }
            found = false;
            for (l=0;
               (synonyms[(accept_symbols[((Integer)stack.elementAt(j)).intValue()][k])][l] !=
               ___NOSYM___) && !found; ) {
               found = valid_token(
                  synonyms[(accept_symbols[((Integer)stack.elementAt(j)).intValue()][k])][l],
                  tokens) &&
                  is_substring(new String(str),
                  name_token[synonyms[(accept_symbols[((Integer)stack.elementAt(j)).intValue()][k])][l]],
                  j);
               if (!found)
                  l++;
               else
                  token = synonyms[(accept_symbols[((Integer)stack.elementAt(j)).intValue()][k])][l];
            }
            if (!found) {
```

```
                found = valid_token(accept_symbols[((Integer)stack.elementAt(j)).intValue()][k], tokens);
                if (found)
                   token = accept_symbols[((Integer)stack.elementAt(j)).intValue()][k];
             }
             if (!found)
                j--;
          } else
             j--;
       }
    }
    return j;
}

/**
* Assign the input reader.
*/
public void assign_input(BufferedReader input) {
    this.input = input;
    eof = false;
    line_count = 1;
    char_count = 1;
}

private char read_char() {
    int ret_c = -1;
    try {
       ret_c = input.read();
    } Catch(IOException e) {
       //  Return the equivalent of EOF when an IO exception occurs
       ret_c = -1;
    }
    if (ret_c == -1)
       eof = true;
    return (char)ret_c;
}

/**
* Get the look ahead back
* @return The content of prevchar.
*/
public StringBuffer getPrevChar() {
    return prev_char;
}

/**
* Get the next token from input stream and write it to value (global)
*
* @return The kind of the found token.
*/
public int lexical(int[] tokens) {
    int first_state, state;
    int i=0, j=0, k=0;
    char c;

    Stack stack = new Stack();
    StringBuffer str = new StringBuffer();
    StringBuffer buf = new StringBuffer();

    sb_value.setLength(0);

    token = ___NOSYM___;  //  initialize the return variable
```

```
if ((state = find_initial(state_types, nb_states)) == TRASH)
   System.exit(-1);

stack.push(new Integer(state));
first_state = state;

if (prev_char.length() > j)
   c = prev_char.charAt(j++);
else
   c = read_char();

while ((!eof) &&
   ((state=next_state(c, first_state, trans, trans_sizes)) == TRASH)) {
   if (c == '\n') {
      line_count++;
      char_count = 1;
   } else
      char_count++;

   if (prev_char.length() > j)
      c = prev_char.charAt(j++);
   else
      c = read_char();
}

stack.push(new Integer(state));
while ((!eof) && (state != TRASH) && (c >= FIRST_CHAR) && (c <= LAST_CHAR)) {
   str.append(c);
   i++;

   if ((state != TRASH) && (trans_sizes[state] == 1))
      c = '\0';
   else {
      if (prev_char.length() > j)
         c = prev_char.charAt(j++);
      else
         c = read_char();
      state = next_state(c, state, trans, trans_sizes);
      stack.push(new Integer(state));
   }
}
//  makes sure TRASH state ends the stack, if the character is not
//  visible otherwise, adds the last character read in the list of
//  processed characters
if ((eof) || (c < FIRST_CHAR) || (c > LAST_CHAR))
   stack.push(new Integer(TRASH));
else {
   str.append(c);
   i++;
}
//  pops the stack until the accepting state is found
k = search_valid_token(stack, str, tokens);

if (k < 0)
   k = 0;

for (i=0; i<k; i++) {
   sb_value.append(str.charAt(i));
```

```
            if (str.charAt(i) == '\n') {
                line_count++;
                char_count = 1;
            } else
                char_count++;
        }

        //  copy the remaining not used characters from str into buf to
        //  keep them for the next run
        for ( ; i<str.length(); i++)
            buf.append(str.charAt(i));

        //  do the same with buf
        for ( ; j<prev_char.length(); j++)
            buf.append(prev_char.charAt(j));


        stack.removeAllElements();
        str.setLength(0);
        prev_char.setLength(0);

        prev_char = buf;
        value = new String(sb_value);

        if ((token == ___NOSYM___) && (eof))
            token = ___EOF___;

        //  print debugging information
        if (__DEBUG_LEXICAL__)
            System.out.println(" token=" + token +
                " eof=" + eof +
                " value=" + value +
                " prev_char=" + prev_char);

        return token;
    }

    public String getValue() {
        return value;
    }

    public int getLineCount() {
        return line_count;
    }

    public void setLineCount(int line_count) {
        this.line_count = line_count;
    }

    public int getCharCount() {
        return char_count;
    }

    public void setCharCount(int char_count) {
        this.char_count = char_count;
    }
}
```