Gilbert Babin. *CompTools: A Compiler Generator for C and Java*. Cahier de recherche no. 04-09. HEC Montréal. Montréal, Québec, Canada. NovemberNovembre 2004.

ISSN 0846-0647

COMPTOOLS: A Compiler Generator for C and Java

Par : Gilbert Babin

Cahier de recherche no 04-09

10 novembre 2004

COMPTOOLS: 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

10 novembre 2004

Abstract

There currently exists a large number of tools to generate parsers and lexical analyzers. However, only a small number of such tools provide a complete solution for the construction of the parser and of the lexical analyzer. Furthermore, to our knowledge, none of these tools propose a syntax diagram generator to document the grammars and lexicon of the newly defined languages. We propose the COMPTOOLS suite which offers a complete set of tools to develop and document parsers and lexical analyzers. Many features of the COMPTOOLS suite make it interesting:

- it can be used for systems developed either in C or in Java,
- the parser generator checks LL(1) conditions and generates extra code when ambiguous grammars are detected, enabling the developer to decide how to manage ambiguity,
- the lexical analyzer generator combines multiple approaches to simplify the resulting lexical analyzer,
- it can generate syntax diagrams for the lexicon and the grammar in EPSF format.

The paper describes theses features and shows how it may be used to develop a new grammar.

Résumé

Un grand nombre de générateurs d'analyseur syntaxique et lexical sont présentement disponibles. Cependant, de ce nombre, peu d'outils offrent un éventail complet de solutions pour accomplir cette tâche. De plus, à notre connaissance, aucun de ces outils ne propose un générateur de diagramme syntaxique. Nous proposons ici la suite logicielle COMPTOOLS qui offre un tel éventail de solutions. Cette suite logicielle a plusieurs caractéristiques intéressantes :

– elle permet la génération d'analyseurs en C ou en Java;
– les conditions LL(1) sont vérifiée par le générateur d'analyseur lexical. Lorsque celle-ci ne sont pas vérifiée, l'analyseur généré permet au développeur de décider comment l'ambiguité peut être résolue;
– le générateur d'analyseur lexical offrent plusieurs alternatives permettant de simplifier l'analyseur généré;
– il offre la possibilité de générer des diagrammes syntaxiques en format EPS.

Cette article présente les caractéristiques de COMPTOOLS. On y illustre aussi son utilisation.

1 Introduction

In their seminal book, Aho *et al.* [1] have laid the foundations of compiler design theory. Succinctly, a computer 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. Consequently, in order to build a compiler, one would need to write modules to process these three elements.

Although the number of languages that may be created is unlimited, the ways to describe languages are limited. This has lead many researchers to develop automated tools to support the development of compilers. Table 1 presents a comparison of parser generators, while Table 2 presents a comparison of lexical analyzer generators. As we can see, there exists a large number of parser generators to address specific needs or specific problems, not resolved properly by other generators. Another reason for this large set might just be that writing a parser generator is not that complex a task, as it can only be done in so many ways. We observe however, that the number of lexical analyzer generators available is not as large, most likely because building a lexical analyzer generator is somewhat more complicated that building a parser generator.

In this paper, we describe the COMPTOOLS suite (COMPiler Construction TOOLS). This software suite is composed of three applications : (1) a parser generator for LL(1) grammars (COMPGEN), (2) a lexical analyzer generator (LEXGEN), and (3) a syntax diagrams generator (DIAGGEN). Many features of the COMPTOOLS suite make it interesting:

- it can be used for systems developed in C or in Java,
- the COMPGEN parser generator checks LL(1) conditions and generates extra code when ambiguous grammars are detected, enabling the developer to decide how to manage ambiguity,
- the LEXGEN lexical analyzer generator combines multiple approaches to simplify the resulting lexical analyzer,
- it can generate syntax diagrams for the lexicon and the grammar in EPSF format.

The remaining of the paper is organized as follows. Sections 2 to 4 describe the applications composing the COMPTOOLS suite. These applications are illustrated using a simple prefix arithmetic calculator. In Section 5, we show how the resulting parser and lexical analyzer are used. We conclude the paper in Section 6. Although the COMPTOOLS suite may be used to generate compilers in C and in Java, in this paper, we focus on the use of Java.

2 The Parser Builder: COMPGEN

The COMPGEN module is primarily used to create a syntax analyzer for a language defined using an LL(1) grammar. 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.

Table 1: Comparison of parser generators

		Languages supported				
Parser generator	*Grammar class*	*C*	*C++*	*Java*	*C#*	*Other languages*
ClearParse [6]	LL(1)	√	√			
JavaCC [19]	LL(1)			√		
LLGen [14]	LL(1)	√				
oops [21]	LL(1)			√		
RDP [18]	LL(1)	√	√			
ANTLR [27]	LL(k)		√	√	√	
CppCC [8]	LL(k)		√			
Depot4 [23]	LL(k)			√		Oberon
Grammatica [13]	LL(k)			√	√	
PRECC [4]	LL(k)	√				
SLK [30]	LL(k)	√	√	√	√	
AnaGram [2]	LALR(1)	√	√			
Bison [3]	LALR(1)	√				
BtYacc [5]	LALR(1)	√				
CUP [9]	LALR(1)			√		
GOLD Parser [12]	LALR(1)					Language-independant
iburg [11]	LALR(1)	√				
JB2CSharp [16]	LALR(1)				√	
LEMON [24]	LALR(1)	√	√			
Styx [31]	LALR(1)	√				
SYNTAX [32]	LALR(1)	√				
VLCC [7]	LALR(1)		√			
Yacc [25]	LALR(1)	√				
Yacc++ [35]	LALR(1)		√			
VisualParse++ [34]	LALR(1)	√	√	√	√	Visual Basic
YaYacc [36]	LALR(1)		√			
jay [15]	LR(1)			√		
YAY [20]	LR(2)	√				
Elkhound [26]	GLR[1]		√			
Rie [29]	ECLR[2]	√				
Toy (TPG) [33]	Prolog-like					Python
ProGrammar [28]	unspecifi ed		√			Visual Basic

[1] Generalized LR parsing
[2] Equivalence class LR parsing

Table 2: Comparison of lexical analyzer generators

	Languages supported		
Parser generator	*C*	*C++*	*Java*
Flex [10]	√		
Lex [25]	√		
JLex [17]		√	
oolex [22]		√	
YooFlex [37]			√

Table 3: EBNF Constructs Allowed by COMPGEN

Construct	*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]/.

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 either C or Java statements that are placed at the exact point in the grammar where they should logically be performed.

2.1 Defining the Syntax

In COMPGEN, grammars are defined using EBNF (Extended Backus-Naur Form) syntax. In this syntax, we distinguish between three concepts:

Categories: Non-terminal symbols which are defined using an EBNF description. Category names are specified as strings written between < and > (e.g., <grammar>).

Reserved words: Terminal symbols refering to a specific string (e.g., while). Reserved words are specified as strings between single quotes (') (e.g., 'while').

Lexical constructs: Terminal symbols refering to complex strings defined by a regular expression (e.g., identifier). Lexical constructs are specified as strings that contain letters and underscores (_) only.

In defining a category, a developer may use the different EBNF constructs illustrated in Table 3. Using this notation, a category <expression> specifying the syntax for prefix arithmetic expressions could have the following syntax:

```
<expression> ::= integer | [ '+' | '-' | '*' | '/' ] <expression> <expression> ;
```

From this, we see that an expression is either an integer value (lexical construct integer) or an arithmetic operator (either '+', '-', '*', or '/'), followed by two expressions.

When building a parser, COMPTOOLS performs a number of verifications. First, it identify parasite categories. These are categories that never lead to reserved words or lexical constructs. In other words, parasites are infinitely expanding categories. Second, COMPTOOLS lists symbols that

are never used, also called inaccessible symbols. These are reserved words or lexical constructs which may never be reached from the main category of the language. Parasites and inaccessible symbols normally generate warnings, telling the developer that some detailed analysis of the language must be performed to ensure that there is no error.

Finally, COMPTOOLS tests every category and decision points within a category to determine whether the LL(1) condition is met. 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. In this case, the grammar is said to be ambiguous. A classical example of ambiguous category is the imbricated if-else statement in most programming languages. Indeed, consider the following example:

```
if (a == b) if (a == c) equal = true; else equal = false;
```

In this example, it is not clear, from a grammar standpoint to which if statement the else clause is linked. This could be interpreted either as

```
if (a == b) {
    if (a == c) equal = true;
    else equal = false;
}
```

or

```
if (a == b) {
    if (a == c) equal = true;
}
else equal = false;
```

The problem is usually resolved by a semantic rule which states that the else is associated to the closed if statement. In this case, this would be the first interpretation. When faced with this situation, COMPGEN will still generate a 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 developer 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.

2.2 Adding Semantic Actions

Once the syntax is properly defined, we can add Java statements into the grammar to do the actual processing. The approach we have favored to enter Java statement is to declare blocks of code (called actions) and then to indicate where a given block of code is to be inserted. This way, the grammar remains readable, even when the code to insert is large. Furthermore, this allows for code reuse. An action is declared by providing a name (starting with @) and the corresponding Java statements. Here is an example of an action:

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

Another feature of COMPGEN is the possibility to pass arguments to a category. This way, results may be processed in a bottom-up or top-down approach. Once all the actions have been defined, the final grammar for our prefix arithmetic calculator is as follows:

Program 1 *prefix.grm*

```
@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.getValue()));
$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

    @exit  //  error handling routine
    $begin_action
    $end_action

    $error @exit
    $name Prefix

    <expression> ("MyInteger val" "val" ) @declare ::=
        integer @assign_val |
        ['+' @assign_plus | '-' @assign_minus | '*' @assign_times | '/' @assign_divide ]
            <expression> ("val") <expression> ("val2") @do_operation ;
```

In this example, we have declared 8 actions:

@declare: It provides the declaration of local variables for the <expression> category. Note that the class MyInteger is a user-defined class that is equivalent to the built-in Integer class but that supports value modification, hence allowing its content to be modified when passed as argument to a method.

@assign_val: It assigns the content of the lexical analyzer's result to local parameter val.

@assign_times: It sets the operation to a multiplication.

@assign_divide: It sets the operation to a division.

@assign_plus: It sets the operation to an addition.

@assign_minus: It sets the operation to a subtraction.

@do_operation: Actually performs the operation.

@exit: It defines an (empty) error handling routine.

The placement of actions is important, as the action is actually performed at the exact location in the grammar. For example, action @declare is placed before the ::= symbol, which garantees that this action will be performed before any analysis is performed in this category. Action @assign_val is placed right after lexical construct integer was encountered, and is therefore executed after the call to the lexical analyzer and before the next symbol is fetched.

When a sub-expression is optional (i.e., it is within -[and]-, the first action following that optional sub-expression may begin by a else statement. That action is always executed, but since it starts with else, the actual action will only be executed if the optional sub-expression is not present. Consider the following partial grammar:

```
    @setToTrue
    $begin_action
        flag = true;
    $end_action
```

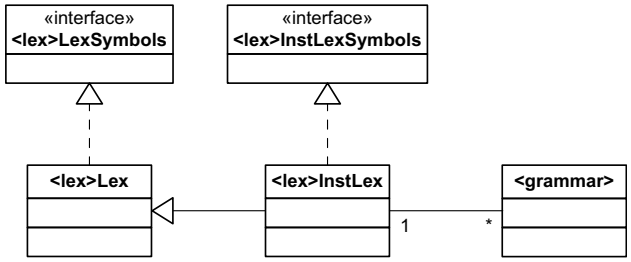

Figure 1: Classes and interfaces generated by COMPGEN and LEXGEN

```
@setToFalse
$begin_action
      else
            flag = false;
$end_action
...
-[ option @setToTrue]- @setToFalse
```

This will generate

```
if (token == inst_lexical.OPTION) {
      // begining action @setToTrue
      flag = true;
      // ending action @setToTrue

      token = inst_lexical.lexical(token001);
      if (token == inst_lexical.___NOSYM___)
            __error__();
}
// begining action @setToFalse
else
      flag = false;
// ending action @setToFalse
```

2.3 System-defined Variables in Java

The lexical analyzer and parser generated by COMPTOOLS are defined as a series of Java classes and interfaces, as illustrated in Figure 1.

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, where <grammar> stands for the grammar name, and provides a number of instance variables and methods:

inst_lexical: This variable is a reference to the lexical analyzer used for this language. The type of this object is <lex>InstLex. The user-defined actions may access any public method 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()).

If an error is found in the course of the syntax analysis, an error handling routine is called. The name of this routine is __error__. The content of that routine is left to the user of COMPGEN. The $error operator lets the user assign actions to that routine.

2.4 Generating the Compiler

In order to generate the corresponding Java files for our prefix arithmetic grammar, we used the following command (see also Tab. 4 for other options):

```
compgen -j prefix.grm
```

The resulting Java file may be found in appendix A. Note however, that this command may not be launched until the lexicon has been defined (see Sect. 3), as compgen uses the lexicon file to determine if symbols have been defined.

3 The Lexical Analyzer Generator: LEXGEN

The goal of a lexical analyzer is to recognize literals and lexical constructs on the input stream. Lexical analyzers built using LEXGEN are deterministic finite state automata (DFSA). The role of the DFSA is to reorganize the input stream into chucks of characters manipulable by the parser. We refer to these chucks as lexical tokens. The code generated by LEXGEN therefore consists of two parts: the DFSA transition tables, which is language-dependant, and the DFSA execution routines (which is language-independant). Ultimately, the lexical analyzer will return the current token found on the input stream.

Processing lexical tokens is straightforward. As it moves through the DFSA, the lexical analyzer stacks up the different states it encounters until it reaches an invalid state. Then, based on a list of expected tokens, it determines the largest token matched (the token using the most characters) in the list of expected symbols in the current context. There is therefore some insight provided by the parser to the lexical analyzer.

Table 4: Java options for the compgen command

Option	*Meaning*
-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 DFSA 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.

3.1 Defining the Lexicon

The DFSA is built automatically based on the token definition, which consists in a regular expression. Therefore, tokens may be very complex. To go from the regular expressions to a DFSA, we use the algorithms presented in [1]. When generating the DFSA, LEXGEN verifies that no two tokens may be detected in the same state. If this occurs, the DFSA may not be generated. This commonly occurs in typical programming languages, as reserved words are often using the same syntax as identifiers. This is a classical problem which LEXGEN resolves in two ways: specifying an anticipation or identifying a super-token.

Specifying an anticipation: A token may be defined with what is refered to as an *anticipation*, that is, a regular expression that must follow the token for it to be recognized. LEXGEN supports the definition of an anticipation when defining tokens. However, this approach has two drawbacks. First, the DFSA generated becomes more complex as it must incorporate the definition of the anticipation. Second, the language must be throughly inspected in order to properly specify the anticipation.

Identifying a super-token: A simpler and more efficient approach consists in simply defining one of the conflicting token as an instance of another token, which then becomes a super-token. For example, a reserved word (e.g., while) can then simply be defined as one instance of the token identifier. Therefore, token identifier becomes the super-token of token while. When matching an identifier, the lexical analyzer will determine if the value of that identifier matches the value of token while, but only if token while is expected in the current context. Consequently, this approach benefits from the reduction of the lexicon definition, as no anticipation has to be specified, and of the DFSA itself. This approach can only be used with tokens that are actual values, and not token patterns.

3.2 Linking the Lexicon to the Grammar

Normally, the lexicon definition is independant from the token context of use. This means, for instance, that token identifier will be used every time an identifier is required. However, for documentation purposes, it might be more appropriate to provide more information as to the type of identifier required. For example, in declaring a function, it might be more meaningful to say that a function_name is required instead of an identifier. Yet, function_name has the same syntax as an identifier. In fact, function_name is a subclass of identifier. To resolve this issue, LEXGEN defines aliases for a token. This leads to the following definitions:

- A token is a pattern to be matched by the lexical analyzer, and is identified by a numeric constant generated by the lexical analyzer generator, the name of the constant being provided by the developer. The developer also provides a string describing the token, which may be used in error processing.
- An alias is a name used by the parser to refer to a particular token. Hence, at least one alias must be provided for every token.

The definition of a token (category <lex_unit>) has the following syntax:

```
<lex_unit> ::= token_id '(' -[ super_token_id ]- ')' error_string '(' +[ alias ]+ ')'
        regular_expression -[ '|' anticipation ]- ;
```

Alias information is not kept once the DFSA is built. Therefore, there is no way to using that information when generating error messages. This is the reason why a single error message is provided for the token. In our prefix arithmetics example, the lexicon file is as follows:

Program 2 *prefix.lex*

```
$name Prefix
INT () "integer" (integer) +0.9;
PLUS () "+" ('+') \+
MINUS () "-" ('-') \-
TIMES () "*" ('*') \*
DIVIDE () "/" ('/') \/
```

In this case, the lexicon is quite simple and contains only 5 symbols, 4 of which are reserved words (PLUS, MINUS, TIMES, and DIVIDE), and one is a lexical construct (INT), composed of 1 or more (+ ;) digits (0.9).

3.3 Generating the Lexical Analyzer

The DFSA generated is usually optimized to reduce its size; the optimization step can be skipped when building the DFSA. Since, the DFSA execution routines are language-independant (i.e., they do not depend on the language generated), in situations where multiple languages are used in a system, in might be more appropriate to have a single copy of the execution routines, while DFSA tables are available for each language used. In LEXGEN, the creation of the lexical analyzer routines and declarations is therefore optional.

In our example, we used the following command to generate the Java files for the lexical analyzer (see also Tab. 5 for more Java options):

Table 5: Java options for the lexgen command

Option	*Meaning*
-j	Generates Java code instead of C code.
-o	Does not perform the optimization of the DFSA.
-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 DFSA 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.

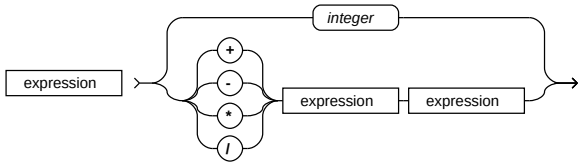

Figure 2: The <expression> Syntax Diagram

```
lexgen -j prefix.lex
```

Appendix B presents the four files generated by LEXGEN, namely PrefixLexSymbols.java, PrefixInstLexSymbols.java, PrefixInstLex.java, and PrefixLex.java.

4 The Syntax Diagram Builder: DIAGGEN

One nice feature of the COMPTOOLS suite is the possibility to automatically generate syntax diagrams for the grammar or the lexicon. This is done by the DIAGGEN application, which will generate Encapsulated PostScript (EPS) files illustrating the definition of categories, literals, and lexical constructs. These files then can be edited by software that can read and edit EPS files (e.g., Corel Draw, Adobe Illustrator, etc.) and/or can be used in a variety of word processors. Figure 2 shows the resulting syntax diagram for the <expression> category presented above. To generate this syntax diagram, we used the following command (see Tab. 6 for more options):

```
diaggen -one expression prefix.grm
```

Table 6: Options for the diaggen command

Option	*Meaning*
-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, ...

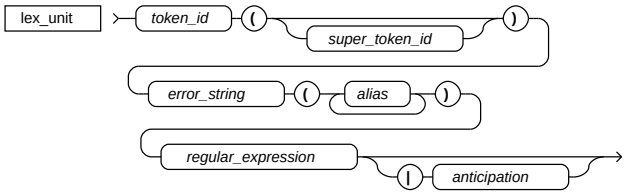

Figure 3: The <lex_unit> Syntax Diagram

It sometimes occurs that a syntax diagram is too wide. Grammar and lexicon description files support the equivalent of a line break, a special character that instructs DIAGGEN to change line. For example, consider the <lex_unit> described earlier. The corresponding diagram will be relatively wide. By placing a colon at appropriate places, it will break up the diagram on multiple lines, as shown in Figure 3.

5 Using the Parser Generated

Running COMPGEN and LEXGEN will produce a set of Java files that can then bu used to process a sentence in the new language. In order to make this possible, an instance of the parser must first be created. Then, the parser is invoked, passing the input channel and, optionally, all the arguments required by the parser. These arguments correspond to the parameters specified for the main category of the grammar.

Consider the prefix artichmetic calculator example. The parser is defined in class Prefix. We can therefore declare and instantiate a parser with the Java statement

```
Prefix stringAnalyzer = new Prefix();
```

In this particular case, the grammar contains only one category, namely <expression>. It's definition indicates that one parameter of type MyInteger[1] is required. Therefore, a call to the

[1] See Appendix C

parser's main method (execute_analyzer) will also require an argument of that type. Furthermore, the input channel expected is of type BufferedReader. It is quite easy to create an instance of that class, either from a file, a stream or a character string, as Java provide such abstractions in the java.io package. In the case where the input channel is string input, we can use the following Java statements:

```
MyInteger val = new MyInteger(0);
string Analyzer.execute_analyzer(
    new BufferedReader(
        new StringReader(input)),
    val);
```

TheTest.Java file (See Appendix C) illustrates the use of this parser. The string to parse is read using the showInputDialog method of the JOptionPane class.

6 Conclusion

This paper presented COMPTOOLS, an application suite for the development and documentation of syntax and lexical analyzers. As we pointed out in Section 1, there is a large number of such tools available. However, we believe that COMPTOOLS introduces interesting features. The COMPGEN application minimally supports LL(1) grammars. For non-LL(1) grammars, it still generates a parser where the developer may explicitly determine how ambiguous categories may be handled. The separation of code and grammar makes it easier to move from one language to the next. The LEXGEN introduces new ways to handle conflicting symbols by introducing the notion of super-token. This approach drastically reduces the size of the DFSA generated, as no extra states must be defined to consider each and every reserved word. Finally, DIAGGEN provides an easy to use tool to generate syntax diagrams for a grammar and a lexicon. Diaggen can generate the diagrams for any grammar, whether it is LL(1) or not, as long as it is written in the appropriate format, that is the grammar syntax defined for COMPGEN and LEXGEN.

The COMPTOOLS has been constructed over a period of 10 years, the first versions using hand-written parsers. The current version of the suite is actually built using the previous version. Consequently, all the grammars required by the suite are manipulated by parsers and lexical analyzers developed with COMPTOOLS.

In the current version of this application suite, there is a clear link between parser and lexical analyzer. In the future, we are considering the development of new approaches to define the lexicon and its link to the grammar. In particular, we are interested in investigating a multi-level parser-lexical analyzer structure, which would use more than the two layers currently defined (syntax and lexicon). In fact, the concept of super-token is one step in that direction. It would be interesting to have a hierarchy of super-tokens, somewhat equivalent to a class hierarchy. This would allow the creation of a highly generic token, which could then be specialized into a sub-token, which in turn could also be specialized, and so on.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[2] AnaGram. http://www.parsifalsoft.com.

[3] Bison. http://www.gnu.org/software/bison/.

[4] Peter Breuer and Jonathan Bowen. A prettier compiler-compiler: Generating higher order parsers in c. *Software — Practice and Experience*, 25(1):1263–1297, November 1995.

[5] BtYacc. http://www.siber.com/btyacc/.

[6] ClearParse. http://www.clearjump.com/products/clearparse/.

[7] Gennaro Costagliola, Genoveffa Tortora, Sergio Orefice, and Andrea De Lucia. Automatic generation of visual programming environments. *IEEE Computer*, 28(3):56–66, 1995.

[8] CppCC. http://cppcc.sourceforge.net/.

[9] CUP. http://www.cs.princeton.edu/~appel/modern/java/CUP/.

[10] Flex. http://www.gnu.org/software/flex/.

[11] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Language and Systems*, 1(3):213–226, 1992.

[12] GOLD Parser. http://www.devincook.com/goldparser/.

[13] Grammatica. http://www.nongnu.org/grammatica/.

[14] Dick Grune and Ceriel J. H. Jacobs. A programmer-friendly LL(1) parser generator. *Software — Practice and Experience*, 18(1):29–38, January 1988.

[15] jay. http://www.informatik.uni-osnabrueck.de/alumni/bernd/jay/.

[16] JB2CSharp. http://sourceforge.net/projects/jb2csharp.

[17] JLex. http://www.cs.princeton.edu/~appel/modern/java/JLex/.

[18] Adrian Johnstone and Elizabeth Scott. RDP — an iterator-based recursive descent parser generator with tree promotion operators. *SIGPLAN Notices*, 33(9):87–94, 1998.

[19] Viswanathan Kodaganallur. Incorporating language processing into Java applications: A JavaCC tutorial. *IEEE Software*, 21(4):70–77, July/August 2004.

[20] Bent Bruun Kristensen and Ole Lehrmann Madsen. Methods for computing LALR(k) lookahead. *ACM Transactions on Programming Languages and Systems*, 3(1):60–82, 1981.

[21] Bernd Kühl and Axel-Tobias Schreiner. An object-oriented ll(1) parser generator. *SIGPLAN Notices*, December 2000.

[22] Bernd Kühl and Axel-Tobias Schreiner. Objects for lexical analysis. *SIGPLAN Notices*, February 2002.

[23] Jürgen Lampe. Depot4 — a generator for dynamically extensible translators. *Software – Concepts & Tools*, 19(2):97–108, September 1998.

[24] LEMON. http://www.hwaci.com/sw/lemon/index.html.

[25] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O'Reilly & Associates, 2nd/updated edition, October 1992.

[26] Scott McPeak. Elkhound: A fast, practical GLR parser generator. Report UCB/CSD-2-1214, Computer Science Division, University of California, Berkeley, Berkeley, California, USA, December 2002.

[27] Terence Parr and Russell Quong. Antlr: A predicated-ll(k) parser generator. *Software — Practice and Experience*, 25(7):789–810, July 1995.

[28] ProGrammar. http://www.programmar.com.

[29] Masataka Sassa, Harushi Ishizuka, and Ikuo Nakata. Rie, a compiler generator based on a one-pass-type attribute grammar. *Software — Practice and Experience*, 25(3):229–250, 1995.

[30] SLK. http://home.earthlink.net/~slkpg/index.html.

[31] styx. http://speculate.de/styx/.

[32] SYNTAX. http://www-rocq.inria.fr/oscar/www/syntax/.

[33] Toy (TPG). http://christophe.delord.free.fr/en/tpg/.

[34] VisualParse++. http://www.sand-stone.com/.

[35] Yacc++. http://world.std.com/~compres/index.html.

[36] YaYacc. http://www.gradsoft.com.ua/products/yayacc_eng.html.

[37] YooLex. http://yoolex.sourceforge.net/.

A Results From COMPGEN

Program 3 *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;

    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.INT, inst_lexical.PLUS, inst_lexical.MINUS,
        inst_lexical.TIMES, inst_lexical.DIVIDE, inst_lexical.___NOSYM___
    } ;
    private int[] token002 = 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 __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) {
            //  begining action @assign_val
            val.setValue((val.getInteger()).parseInt(inst_lexical.getValue()));
```

```
        // ending action @assign_val

        token = inst_lexical.lexical(token001);
        if (token == inst_lexical.___NOSYM___)
            __error__();
    }
    else
    /* if ((token == inst_lexical.PLUS) || (token == inst_lexical.MINUS) ||
    (token == inst_lexical.TIMES) || (token == inst_lexical.DIVIDE))
    */
    {
        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___) JavaIndentBlock
            __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__();
        }

        __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 (token == inst_lexical.___NOSYM___)
                _error_();

            if (_DEBUG_SYNTACTIC__) {
                debug_level–;
                __debug__(debug_level, "END of _expression_");
            }
        }

        /**
        * Execute the lexical analyzer.
        */
        public void execute_analyzer(BufferedReader br, MyInteger val) {
            inst_lexical.setLineCount(0);
            inst_lexical.setCharCount(0);
            inst_lexical.assign_input(br);
            token = inst_lexical.___NOSYM___;
            token = inst_lexical.lexical(token000);
            __expression_(val);
        }

        /**
        * 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
        }
    }
```

B Results From LEXGEN

Program 4 *PrefixLexSymbols.java*

```
// Lexicon file: prefix.lex

interface PrefixLexSymbols {
    int ___NOSYM___ = 0;
    int ___EOF___ = 1;
}
```

Program 5 *PrefixInstLexSymbols.java*

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

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

Program 6 *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 = 6;

        state_types = new int[] {
            6,6,6,6,6,1
        } ;
        final_symbols = new int[] {
            INT, PLUS, MINUS, TIMES, DIVIDE, ___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____
        } ;
        accept_symbols = new int[][] {
            accept_000, accept_001, accept_002, accept_003,
            accept_004, accept_NULL
        } ;
        int[][] transNULL = {
            {
                FIRST_CHAR, LAST_CHAR, TRASH
```

```
            }
        } ;
        int[][] trans000 = {
            {
                48, 57, 0
            } , {
                FIRST_CHAR, LAST_CHAR, TRASH
            }
        } ;
        int[][] trans005 = {
            {
                42, 42, 3
            } , {
                43, 43, 1
            } , {
                45, 45, 2
            } , {
                47, 47, 4
            } , {
                48, 57, 0
            } , {
                FIRST_CHAR, LAST_CHAR, TRASH
            }
        } ;
        trans_sizes= new int[] {
            2, 1, 1, 1, 1, 6
        } ;
        trans = new int[][][] {
            trans000,transNULL,
            transNULL,transNULL,
            transNULL,trans005
        } ;
        String name_NULL = "" ;
        name_token = new String[] {
            name_NULL, name_NULL, name_NULL,
            name_NULL, name_NULL, name_NULL, name_NULL
        } ;
        int[] synonymNULL = {
            ___NOSYM___
        } ;
        synonyms = new int[][] {
            synonymNULL, synonymNULL, synonymNULL,
            synonymNULL, synonymNULL, synonymNULL, synonymNULL
        } ;
    }
}
```

Program 7 *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 subclass
    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 (DFSA)
    protected int[][] synonyms;
    protected int[] state_types;  // state types (DFSA)
    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;
        }
    }
```

C Other Java Files

Program 8 *Test.java*

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

    public class Test {

        public Prefix stringAnalyzer;

        /**
        * Constructor.
        */
        public Test() {
            stringAnalyzer = new Prefix();
        }
```

```
    public static void main(String argv[]) {
        String input = JOptionPane.showInputDialog("enter expressions");
        MyInteger val = new MyInteger(0);

        (new Test()).stringAnalyzer.execute_analyzer(
                new BufferedReader(new StringReader(input)),val);

        JOptionPane.showMessageDialog(null,"="+(val.getInteger()).toString());
    }
}
```

Program 9 *MyInteger.java*

```
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;
    }
}
```