jLite

Lite version of Java

Sudhakar Perumal
skp2111@columbia.edu

Contents

1 Introduction

jLite is a light weight java compiler written in java. Though current version of jLite only compiles very small subset of java syntax. But it can be easily extended to compile variety of statements and also it gives the ability to create new language construct that can be compiled to bytecode.

1.1 Background

jLite evolved when I had the idea to create a compiler that can dynamically accept java like source code and produce bytecode, which can be loaded into the JVM and executed. This type of light weight compiler can find its use in many areas. It can used as a research compiler in experimenting with introducing new features that are not available in java. jLite compiler can extended to build static code analysers. Based on the AST produced by the parser, the walker produces derived classes, which can used to do all kinds of analyses on the source file. For example, analysing the control flow detecting infinite loops and even producing suggestions for optimizing the code. Apart from that this type of compilers can used to build expression evaluators which can used in ecommerce systems. For example, assume a software firm is building an ecommerce system that can be customized for its clients. The various cost modeling rules cannot be embedded into source code. Because these rules could change from one client to another because of their individual policies. If these rules can be kept as dynamic expressions which can be specified at run-time, not at compile-time. The system becomes much more flexible and efficient to maintain.

1.2 GOAL

The main goal of this project is create a simple light weight compiler that can accept subset of java syntax and generate bytecode.

1.2.1 Portability

Since jLite converts the source into bytecode, jLite can be used in all the platforms supported by SUN JVM.

1.2.2 Efficiency

Since the source is converted into bytecode before execution, this provides maximum efficiency.

1.3 Main language features

1.3.1 Data types

Basic data types supported by this language are int, double, string and boolean. It also supports all standard java classes that are available in the classpath..

1.3.2 Statements

The language supports relational expressions, if-then-else statements, for loops.

1.3.3 Internal Functions

There are no special internal functions, the java libraries available in the classpath can be used.

1.4 Tutorial

Here is a simple arithmetic expression that can be converted into bytecode by this compiler.

1.4.1 Arithmetic expressions

To compile invoke the compiler as follows

If source files is in a different called testcases

Java JliteCompiler –src ./testcases –o ./testcases arithexpr1.java

Or

If the source file is in current directory

Java JliteCompiler arithexpr1.java

```
class arithexpr1 {

  void main(String args[]) {
    int i = 0;
    i = ( (i + 10) + 5) * i + 1;
    i = i + 1;
```

```
    i = -i + 10;
    System.out.println(i);
    int j = 10;
    ;
    int k = 1 + 4;
    System.out.println(k);
    int x = (i + j + k);
    System.out.println(x);
    {
      int z = 0;
    }
    (i) = (20);
    System.out.println(i);
  }
}
```

The above program will produce the following bytecode that can viewed using ‘javap’ tool. (using –c option)

```
Compiled from "<jLiteCompiler>"
public class arithexpr1 extends java.lang.Object{
public arithexpr1();
  Code:
   0:   aload_0
   1:   invokespecial   #10; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   iconst_0
   1:   istore_2
   2:   iload_2
   3:   bipush  10
   5:   iadd
   6:   iconst_5
   7:   iadd
   8:   iload_2
   9:   imul
   10:  iconst_1
   11:  iadd
   12:  istore_2
   13:  iload_2
   14:  iconst_1
   15:  iadd
   16:  istore_2
```

```
17:  iload_2
18:  ineg
19:  bipush  10
21:  iadd
22:  istore_2
23:  getstatic       #20; //Field java/lang/System.out:Ljava/io/PrintStream;
26:  iload_2
27:  invokevirtual   #26; //Method java/io/PrintStream.println:(I)V
30:  bipush  10
32:  istore_3
33:  iconst_1
34:  iconst_4
35:  iadd
36:  istore  4
38:  getstatic       #20; //Field java/lang/System.out:Ljava/io/PrintStream;
41:  iload   4
43:  invokevirtual   #26; //Method java/io/PrintStream.println:(I)V
46:  iload_2
47:  iload_3
48:  iadd
49:  iload   4
51:  iadd
52:  istore  5
54:  getstatic       #20; //Field java/lang/System.out:Ljava/io/PrintStream;
57:  iload   5
59:  invokevirtual   #26; //Method java/io/PrintStream.println:(I)V
62:  iconst_0
63:  istore  6
65:  bipush  20
67:  istore_2
68:  getstatic       #20; //Field java/lang/System.out:Ljava/io/PrintStream;
71:  iload_2
72:  invokevirtual   #26; //Method java/io/PrintStream.println:(I)V
75:  return
```

1.4.2 TrigonometricDemo

The following program demonstrates the ability of this compiler to invoke java classes that are available in the classpath

```
class TrigonometricDemo {
   void main(String[] args) {
      double degrees = 45.0;
      double radians = Math.toRadians(degrees);
```

```
        TrigonometricDemo t = new TrigonometricDemo();

        t.print("The sine  ", degrees, Math.sin(radians));

        t.print("The cosine ", degrees,
                        Math.cos(radians));
        t.print("The tangent ", degrees,
                        Math.tan(radians));
        t.print("The arcsine ",
                        Math.sin(radians),
                        Math.toDegrees(Math.asin(Math.sin(radians))));
        t.print("The arccosine",
                        Math.cos(radians),
                        Math.toDegrees(Math.acos(Math.cos(radians))));
        t.print("The arctangent ",
                        Math.tan(radians),
                        Math.toDegrees(Math.atan(Math.tan(radians))));

    }

    void print(String str, double d1, double d2) {
      StringBuffer sb = new StringBuffer();
      sb.append(str);
      sb.append(" ");
      sb.append(Math.rint(d1));
      sb.append(" ");
      sb.append(Math.rint(d2));
      System.out.println(sb.toString());
    }
}
```

Output produced by the compiler (*javap –c TrigonometricDemo*)

```
Compiled from "<jLiteCompiler>"
public class TrigonometricDemo extends java.lang.Object{
public TrigonometricDemo();
  Code:
   0:   aload_0
   1:   invokespecial   #10; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   ldc2_w  #15; //double 45.0d
```

```
3:  dstore_2
4:  dload_2
5:  invokestatic    #22; //Method java/lang/Math.toRadians:(D)D
8:  dstore  4
10: new     #4; //class TrigonometricDemo
13: dup
14: invokespecial   #23; //Method "<init>":()V
17: astore  6
19: aload   6
21: ldc     #25; //String The sine
23: dload_2
24: dload   4
26: invokestatic    #28; //Method java/lang/Math.sin:(D)D
29: invokevirtual   #32; //Method print:(Ljava/lang/String;DD)V
32: aload   6
34: ldc     #34; //String The cosine
36: dload_2
37: dload   4
39: invokestatic    #37; //Method java/lang/Math.cos:(D)D
42: invokevirtual   #32; //Method print:(Ljava/lang/String;DD)V
45: aload   6
47: ldc     #39; //String The tangent
49: dload_2
50: dload   4
52: invokestatic    #42; //Method java/lang/Math.tan:(D)D
55: invokevirtual   #32; //Method print:(Ljava/lang/String;DD)V
58: aload   6
60: ldc     #44; //String The arcsine
62: dload   4
64: invokestatic    #28; //Method java/lang/Math.sin:(D)D
67: dload   4
69: invokestatic    #28; //Method java/lang/Math.sin:(D)D
72: invokestatic    #47; //Method java/lang/Math.asin:(D)D
75: invokestatic    #50; //Method java/lang/Math.toDegrees:(D)D
78: invokevirtual   #32; //Method print:(Ljava/lang/String;DD)V
81: aload   6
83: ldc     #52; //String The arccosine
85: dload   4
87: invokestatic    #37; //Method java/lang/Math.cos:(D)D
90: dload   4
92: invokestatic    #37; //Method java/lang/Math.cos:(D)D
95: invokestatic    #55; //Method java/lang/Math.acos:(D)D
98: invokestatic    #50; //Method java/lang/Math.toDegrees:(D)D
101: invokevirtual  #32; //Method print:(Ljava/lang/String;DD)V
104: aload  6
106: ldc    #57; //String The arctangent
```

```
   108: dload   4
   110: invokestatic    #42; //Method java/lang/Math.tan:(D)D
   113: dload   4
   115: invokestatic    #42; //Method java/lang/Math.tan:(D)D
   118: invokestatic    #60; //Method java/lang/Math.atan:(D)D
   121: invokestatic    #50; //Method java/lang/Math.toDegrees:(D)D
   124: invokevirtual   #32; //Method print:(Ljava/lang/String;DD)V
   127: return

public void print(java.lang.String,double,double);
  Code:
   0:   new     #66; //class StringBuffer
   3:   dup
   4:   invokespecial   #67; //Method java/lang/StringBuffer."<init>":()V
   7:   astore  6
   9:   aload   6
   11:  aload_1
   12:  invokevirtual   #71; //Method java/lang/StringBuffer.append:(Ljava/lang/
String;)Ljava/lang/StringBuffer;
   15:  aload   6
   17:  ldc     #73; //String
   19:  invokevirtual   #71; //Method java/lang/StringBuffer.append:(Ljava/lang/
String;)Ljava/lang/StringBuffer;
   22:  aload   6
   24:  dload_2
   25:  invokestatic    #76; //Method java/lang/Math.rint:(D)D
   28:  invokevirtual   #79; //Method java/lang/StringBuffer.append:(D)Ljava/lan
g/StringBuffer;
   31:  aload   6
   33:  ldc     #73; //String
   35:  invokevirtual   #71; //Method java/lang/StringBuffer.append:(Ljava/lang/
String;)Ljava/lang/StringBuffer;
   38:  aload   6
   40:  dload   4
   42:  invokestatic    #76; //Method java/lang/Math.rint:(D)D
   45:  invokevirtual   #79; //Method java/lang/StringBuffer.append:(D)Ljava/lan
g/StringBuffer;
   48:  getstatic       #85; //Field java/lang/System.out:Ljava/io/PrintStream;
   51:  aload   6
   53:  invokevirtual   #89; //Method java/lang/StringBuffer.toString:()Ljava/la
ng/String;
   56:  invokevirtual   #95; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   59:  return
}
```

2 Reference Manual

This section of the document describes the subset of java language features supported by jLite programming language. Antlr like syntax is also provided whereever applicable.

2.1 Lexical Structure

This section specifies the lexical structure of jLite programming language.

2.1.1 Line Terminators

Lines are terminated by the ASCII characters CR, or LF, or CR LF. The two characters CR immediately followed by LF are counted as one line terminator, not two.

2.1.2 Tokens

Tokens can be found in the form of identifiers, keywords, literals, separators and operators. These tokens will be separated by whitespace and line terminators.

2.1.3 White space

White space is defined as the ASCII space, horizontal tab, and form feed characters, as well as line terminators.

2.1.4 Comments

There are two kinds of comments:

/* text */ A traditional comment: all the text from the ASCII characters /* to the ASCII characters */ is ignored (as in C and C++).

// text A end-of-line comment: all the text from the ASCII characters // to the end of the line is ignored (as in C++).

2.1.5 Identifiers

An identifier is an unlimited-length sequence of letters and digits, the first of which must be a letter. An identifier cannot have the same spelling as a keyword, boolean literal or the null literal.

2.1.6 Keywords

The following character sequences, formed from ASCII letters, are reserved for use as keywords and cannot be used as identifiers

boolean	else	int
class	for	new
double	if	return

2.1.7 Literals

A literal is the source code representation of a value of a primitive type, the String type, or the null type. Following types of primitive literals are supported by this language

Integer literals	e.g. 0, 10
Double literals	e.g 10.5, 1000.500
Boolean literals	e.g. true, false

2.1.8 Separators

The following nine ASCII characters are the separators

( ) { } [ ] ; , .

2.1.9 Operators

The following 26 ASCII characters are the operators.

= > < == <= >= && ||
++ -- + - * /

2.2 Types

The jLite programming languages supports two categories of types: primitive types and reference types. Primitive types are int, double and boolean. Reference types are class types, interface types. The values of a reference type are references to objects.

2.2.1 Primitive types and values

A primitive type is predefined by the jLite programming language and named by its reserved keyword. The numeric types are the integer types and the double types. Boolean type has exactly two values: true and false. Comparison operators (==, >= and <=) and numerical operations (+, -, *, /) are allowed against integer and double types. Boolean types allows relational (==)..

Examples of primitive types

```
int i = 10;
double d = 10.0;
boolean b = true;
```

2.2.2 Reference types and values

A reference type can be of class types or, interface types. jLite supports method invocation on reference types. The reference class for the reference object should be available in the classpath. For example, any of the classes or reference types provided by the java environment can be referenced. An Object is a class instance. The reference values (often just references) are pointers to these objects.

In this version of jLite, only Method invocations are allowed on the references to objects

Example of reference type and values

```
java.util.HashMap map = new java.util.HashMap();
map.put("1", "One");
System.out.println(map.get("1"));
```

Grammar

```
type
        :       Identifier ('.' Identifier)*
        ;
```

.

2.3 Classes

Class declaration defines new reference types and describes how they are implemented. In this version of jLite nested classes, abstract classes, inner classes and interfaces are not allowed. Access modifiers on the class level are not allowed, in other words, all classes will be treated as public classes.

Example.

class test	is equivalent to	public class test

Grammar

classDeclaration
: 'class' Identifier
classBody
;

2.3.1 Constructors

This version of jLite doesn't support constructors. But jLite compiler will add a default no argument constructor into the generated bytecode, which will allow the programmer to create an instance of the class.

2.3.2 Variables

Global variable declaration is not supported in this version of jLite compiler. Since jLite support local variables the architecture can be easily extended to implement global variables.

2.3.3 Modifiers

Modifiers are not supported in this version of jLite. However, when jLite compiler encounters a main function, it will add static modifier to generated bytecode. Since it will provide an easier way to invoke the main function from java interpreter. And the compiler will treat all other functions as public virtual functions.

Example

Example 1

void main(String args[])

Generated code will contain the signature *public static void main(String args[])*

Example 2

void test()

Generated bytecode will contain *public void test()*

2.3.4 Function Definition

Function definition can have the form

<returntype> <id> (<arglist comma separated>) {
    <func body>
}

2.4 Expressions

2.4.1 Assignment

An assignment statement can be in the form of

<Variable> = <value> ;

Example : a = 10;

2.4.1 Arithmetic expression

Arithmetic operations are allowed on variables of type int and double. Java supports + to be used to concatenate string values. But Jlite doesn't support that function. Concatenation can be achieved by using StringBuffer class.

Arithmetic operation can have the form

<expr> + <expr>

2.4.2 Relational expression

Relational operations are performed on the variables of type int, double and boolean.

Relation expression can have the form

<expr> (> | < | >= | <= | ==) <expr>

where expr could be a variable or another expression.

2.4.3 Function invocation

All local function calls can be of virtual function calls. If you have main function in your program which will become static function automatically. You can only

invoke another local function as long as you are invoking thru the object instance of local class.

For example

```
Class a {
        void main (String args[]) {
                f();                            //Invalid
                a I = new a();
                I.f()                           //Valid
        }

        void f() {
                f1();                           //Valid
        }

        void f1() {
        }
}
```

2.5 Statements

2.5.1 Declaration

Declaration statement allows you to define a variable of certain type and intialize it.

Declaration can have the form of

<type> <varname> = <initialvalue.;

2.5.2 Conditional Statement

If statement is the only conditional statement available in this language.

If statement can have the form of

If <expr> stmt ; else stmt ;

2.5.3 Iterative statements

For loop is available for performing iterative tasks.
For loop can have the form of

For (declaration|initialization; conditionalcheck; expression)

```
{
        stmt
}
```

2.6 External Functions

Any method in any java class can be invoked using this compiler.

For example

System.out.println(“test”);

Integer.parseInt(“10”);

Math.sin(Math.toRadians(45));

Chapter 3

Project Plan

3.1 Development Process

Process invovled in developing this compiler was different from traditional SDLC in the sense that the requirements will created by the development team.

- In jLite, the process started with identifying the language construct to which the compiler is going to be written and since the output of this compiler is going to be bytecode, professor suggested to keep the language syntax as close to java as possible.
- Considerable amount of time has been spent in understanding bytecode representation of the java program.
- Phase III, which is also the most important phase was involved in choosing the right bytecode generator. GNU andBCEL bytecode generators have been evaluated. BCEL was chosen for this project, the major differentiating factor being the documentation provided for BCEL was detailed and made the choice easier.
- Phase IV is to finalize the requirements and create tasks that needs to be acomplished and associated timelines.
- Phase V is to indentify the tools for development and testing required for this project. Jbuilder was my primary choice to develop this project, since

it gives great flexibility in compiling and debugging. For testing, I decided to use python script will which execute the compiler by passing in all the java files from the folder which holds all the test cases. And runs those

- Phase VI is coding and unit testing. Regression testing will also be done when the change made to the code is going to be finalized.
- Final phase is to prepare documentation.

3.2 Project plan

Following is the project plan showing important milestones

Date	Task Due
June 11th 2007	Language Whitepaper
July 20th 2007	Language Reference Manual
June 25th 2007	Finalize Bytecode Generator
July 9th 2007	Front End
July 16th 2007	Reflection Library
July 27th 2007	Back End
July 30th 2007	Python regression test script
August 3rd 2007	Finish Regression testing and bug fixes
August 10th 2007	Final report

3.3 Programming style

3.3.1 Antlr Coding style

Common antlr coding style was followed,

- lexems will be in upper case.
- If the antlr rule is short then it will written in one line and ';' will be at the end of the line.
- If the rule is long, then the grammar will be split into mutliples lines with each line showing the choice that option can have. The first line of the rule will have ':' at the 9th column. At the end of the rule, ';' will appear on the 9th column.

3.3.2 Java coding style

- Since Jbuilder was used to develop this project, the default identation provided by the IDE was used. The identation was always two spaces.
- The '{' will follow one space after any statement (function declaration, for statement, if statement and try).
- The '}' will occupy an entire line.

- All the classes that belongs to the Jlite front end and Jlite types will be prefixed with ‘JLite’. Whereas the common utility and helper classes that can be detached from this project will not be prefixed with Jlite.
- Standard Javadoc style comments are followed whereever possible.

3.4 Software Development Environment

3.4.1 Operating System

This project was completed developed on windows operating system. Since java compilers are available for many platforms, compiling the source code should be an issue. Python script was used for regression testing the compiler. Also, python is available for most of the popular operating systems.

3.4.2 Java 1.4

Though Java 1.5 stable version is available during the development of this compiler. The packages used by this compiler ANTLR and BCEL was tested and stable in 1.4 version, so jdk 1.4 was used to compile the code and also run the generated bytecode.

3.4.3 Antlr

The paser and walker are generated using Antlr v 2.7.7. Antlr generates the parse and walker based on the grammar file specified by the user.

3.4.4 BCEL

Byte Code Engineering Library from apache open source community was used by the back end of Jlite compiler to generate the bytecodes.

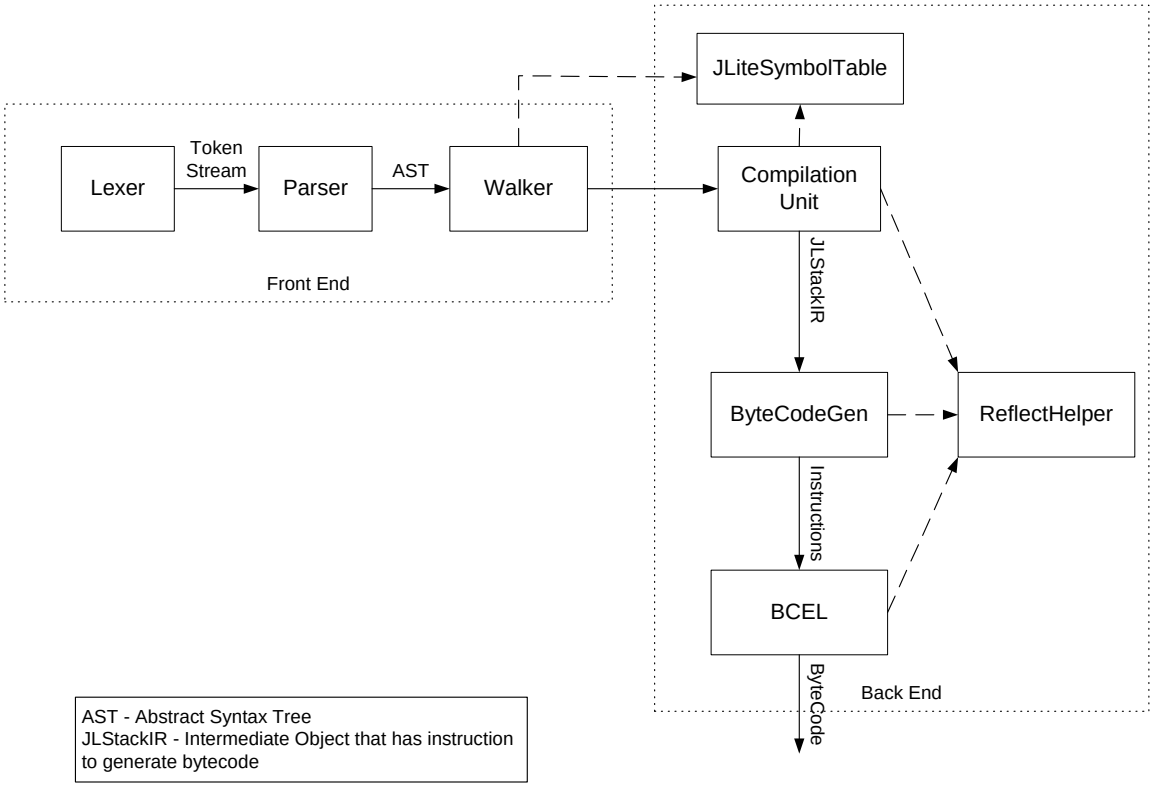
3.4.5 Python

Python script was used for regression testing the compiler. Since python provides an elegant way for writing scripts that can be executed in most of the popular operating systems. Python scripts are not only powerful but also easier maintain when compared to other scripting languages.

Chapter 4

Architecture Design

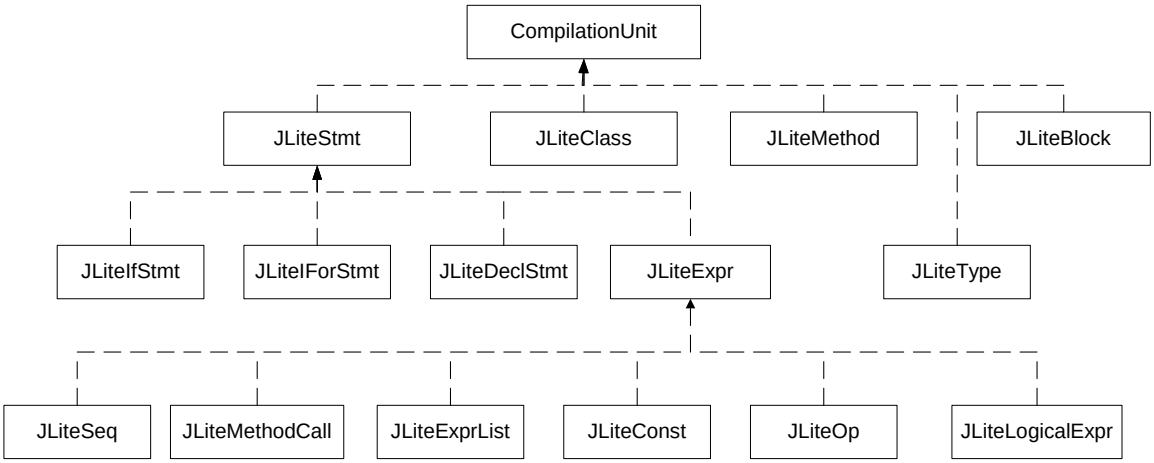
The JLite compiler component can be grouped in to two major categories: a front end and a back end. Front end consists of following components: a lexer that generates tokens from the input file, the parser which then takes these tokens and creates the Abstract Syntax Tree, the walker which recursively walks the AST and creates various compilation units. The back end consists of Compilation Units which creates the simple objects JLStackIR objects that holds the information required to generate bytecode, a BytecodeGen which creates and loads the instructions into BCEL based on JLStackIR, BCEL which then generates the actual bytecodes. The following diagram shows the interaction between these components.



Architecture Diagram

JliteCompiler class is the main program which takes in a file name and invokes the lexer, parser and the tree walker. JliteCompiler will make two passes in order to generate the bytecode. First time it will invoke the typecheck function, which will check the various semantics of the implemented language. If the sematic check succeeds it will then invoke the compile function which will produce the bytecode.

The compilationUnit is the base class which holds the objects that are common to all classes. The following diagram shows the class hierarchy used by this compiler. JliteClass is the first object that will be created and it will contain all the methods. And JliteMethod object contains blocks, which contains different statements and expressions. CompilationUnit is an abstract class with two abstract functions: typecheck and compile. All the classes which extends compilationunit must implement these two functions. And those classes may directly handle these two functions or delegates to the other objects it contains. For example, JliteClass doesn't do any typechecking it just calls typecheck function in Jlitemethod and that calls gets trickled down further.



Class Hierarchy

The typecheck process first generates the bytecode for class with just method signatures and loads the class into the repository. Repository is one of the classes provided by BCEL to load a class into the classloader. And then from that point forward that class can be introspected through reflection. The idea is to load the class (with method definition) into the repository, so whenever there is a reference to any local function or class itself made within the source program can then be resolved by looking into the repository. ReflectHelper is another class in the backend that helps to resolve the references made in the source program through reflection. For example, String will be resolved to java.lang.String class. This technique lays foundation for compiling nested classes.

During compile function call, the actual code for the statements and expression will be generated. The compilationUnit objects will create JLStackIR which is an intermediate object that contains information required to generate the bytecode. The CompilationUnit will send these JLStackIR object to ByteCodeGenImpl. ByteCodeImpl finally generates the bytecode. The reason why JLStackIR objects are created was to decouple the CompilationUnits from referring to the BCEL bytecode instructions object directly. In the future version of this compiler the ByteCodeGen then can be easily rewritten to work with any other package. And also the JLStackIR representation gives us ability to do any optimization before generating the actual bytecode.

Chapter 5

Testing

5.1 Testing Plan

Since Jlite compiler generates bytecodes directly, a robust test bed is required to cover all the statements and expressions supported by the underlying language. So test cases contains arithmetic operations, if statements, for loops and java standard library calls like System.out.println.

5.2 Unit Testing

The unit testing will be done within the jbuilder environment. A testcase will be created for the syntax that I am currently working on. Once the change made is final the testcase will be added to the regression test database and the regression testing will be done. During unit testing the code generated by the JliteCompiler will compared with the bytecode generated by java. JDK has an excellent utility to disassemble the bytecode, when javap is run with –c option will show the list of stack operations. This tool not only helped me during unit testing but also gives me reference to what kind of changes I have to make to my compiler.

5.3 Automated Regression Testing

Regression testing is very important for any project. Since the new changes we are making may inadvertanly introduce new bugs. So to run all the testcases in the database, I created a simple python script. This script will feed all the java source file in testcases directory to the compiler and then runs the compiled class using java interpreter and captures the output into .t files. The script will then compares the .t files with .golden files (which contains the expected output) and displays the miscompared files on the screen. This script has saved lot of time and provided a way test efficiently. So introducing new test case is to copy the java source files into the database directory and create a golden file for the expected output.

5.4 Sample Test cases

5.4.1 Arithexpr1.java

This test case was writen to demonstrate the various arithmetic operations that can be peformed using this language and also displays the calculated values when run through java interpreter.

```
class arithexpr1 {

  void main(String args[]) {
    int i = 0;
    i = ( (i + 10) + 5) * i + 1;
    i = i + 1;
    i = -i + 10;
    System.out.println(i);
    int j = 10;
    ;
    int k = 1 + 4;
    System.out.println(k);
    int x = (i + j + k);
    System.out.println(x);
    {
      int z = 0;
    }
    (i) = (20);
    System.out.println(i);
  }
}
```

arithexpr1.golden

```
8
5
```

23
20

5.4.2 Forloop2.java

This test case validates the functioning of for loop.

```
class forloop2 {
  void main(String args[]) {
    int isum = 0;
    double dsum = 0.0;

    for (int i=0; i<10; i++) {
      System.out.println(i);
      isum = isum + i;
    }
    System.out.println(isum);

    for (double d=0.5; d<10.5; d++) {
      System.out.println(d);
      dsum = dsum + d;
    }
    System.out.println(dsum);

  }
}
```

forloop2.golden

```
0
1
2
3
4
5
6
7
8
9
45
0.5
1.5
2.5
3.5
4.5
5.5
```

```
6.5
7.5
8.5
9.5
50.0
```

5.4.3 overload.java

This testcase validates the case where class contains two functions with same name but with different argument types and also tests the recursive invocation.

```
class overload {

 void main(String[] args) {
  {
   overload a = new overload();
   a.test(10);    //this will call the first 'test' function
   a.test(10.5);  //this will call the second 'test' function
   System.out.println(a.fac(10));
  }
 }

 int fac(int n) {
  if (n == 1)
   return n;
  else
   return n * fac(n-1);
 }

 int test(int a) {
  System.out.println("int test");
  System.out.println(a);
  return a;
 }

 int test(double a) {
  System.out.println("double test");
  System.out.println(a);
  return 10;
 }

}
```

overload1.golden

int test
10
double test
10.5
3628800

Chapter 6

Lessons Learned

- Writing a compiler is a huge undertaking. Particular if you are CVN student plan to spend lot of time on this project. Because this project has steep learning curve since it required to explore various tools you need to finish this project.

- If you have finished a task before the required deadline quickly move on to next task immediately. Because you may encounter unexpected challenges that may affect your project timelines terribly.

- Having a precise project plan will help you track the project status and gives you the ability to spot the lag.

Appendix A

A.1 Lexer and Parser

```
/**************************************************************************

                    jLite Scanner

**************************************************************************/

class JLiteLexer extends Lexer;
options { k = 2; }

WS      : (' ')+  { $setType(Token.SKIP); } ;

protected DIGITS : ('0'..'9')+ ;

NUM     : DIGITS ('.' DIGITS { $setType(REAL); } )? ;

SEMI    : ';'  ;
LPAREN  : '('  ;
ASSIGN  : '='  ;
RPAREN  : ')'  ;
LBRACE  : '{'  ;
PLUS    : '+'  ;
RBRACE  : '}'  ;
MINUS   : '-'  ;

DIV     : '/'  ;
PLUS_ASSIGN
        : "+="  ;
COMMA   : ','  ;
DOT     : '.'  ;
ESC     : '\\' ;
LBRACK  : '['  ;
RBRACK  : ']'  ;
INC     : "++"  ;
DEC     : "--"  ;
STAR    : '*'  ;
EQUAL   : "=="  ;
LAND    : "&&"  ;
LOR     : "||"  ;
LT      : '<'  ;
```

```
LTEQ    : "<="  ;
GT      : '>'   ;
GTEQ    : ">="  ;


ID      : ('_' | 'a'..'z' | 'A'..'Z') ('_' | 'a'..'z' | 'A'..'Z' | '0'..'9')* ;

STR_LITERAL
          :          ""! (ESC|~(""|'\\'|'\n'|'\r'))* ""!
          ;

NL      : ('\n' | ('\r' '\n') => '\r' '\n' | '\r')
          { $setType(Token.SKIP); newline(); }
        ;

COMMENT : ( "/*" (
                  options {greedy=false;} :
                  (NL)
                  | ~( '\n' | '\r' )
                )* "*/"
          | "//" (~( '\n' | '\r' ))* (NL)
          ) { $setType(Token.SKIP); }
        ;


/*************************************************************************

                        jLite Parser

*************************************************************************/

class JLiteParser extends Parser;
options {
        buildAST = true;
        k = 2;
}
tokens  {
         NEGATE; BOOLEAN; LOGICAL; CLASSDEF; DECL; TYPE; EXPR;
METHOD_CALL; CONSTR_CALL; ELIST; ARRAY_DECL;
         FOR_INIT; FOR_CONDITION; FOR_ITERATOR; METHOD_DEF;
PARAM_LIST; PARAM_DEF;
        }

javasource
        : (
           //("import"^ importstmt SEMI!)*
```

```
        (classdef)*
      )
    ;

importstmt
    : ID (DOT^ ID)* (DOT^ STAR)? ;

classdef
    : "class"! ID classBlock {#classdef = #(#[CLASSDEF, "CLASSDEF"],
#classdef);} ;

classBlock
    : LBRACE! (fieldOrFunc)* RBRACE! ;

fieldOrFunc!
    :
      t:type  // method
      (
        (
          id:ID  // the name of the method

          // parse the formal parameter declarations.
          LPAREN! param:paramDeclList RPAREN!

          rt:arrydecl[#t]

          ( bl:block )

          {#fieldOrFunc = #(#[METHOD_DEF,"METHOD_DEF"],
                            #(#[TYPE,"TYPE"],rt),
                            id,
                            param,
                            bl);}
        )
      )
    ;

// A list of formal parameters
paramDeclList
        : ( paramDecl ( COMMA! paramDecl )* )?
          {#paramDeclList = #(#[PARAM_LIST, "PARAM_LIST"],
#paramDeclList);}
        ;

// A formal parameter.
paramDecl!
```

```
        : t:type id:ID pd:arrydecl[#t]
          {#paramDecl = #(#[PARAM_DEF,"PARAM_DEF"], #([TYPE,"TYPE"],
pd), id);}
           ;

block   : LBRACE^ (stmt)* RBRACE! ;

basicType
        : ("int" | "char" | "boolean" | "float" | "double" | "void") ;

type    : (basicType | complexType) (LBRACK^ RBRACK!)* ;

complexType
        : ID  ( options { greedy = true; } : DOT^ ID )* ;

arrydecl[AST t]
        : {#arrydecl=getASTFactory().dupTree(t);}  (LBRACK^ RBRACK!)* ;

decl!   : t:type id:ID t2:arrydecl[#t] v:varInitializer {#decl = #(#[DECL, "DECL"],
#(#[PARAM_DEF, "PARAM_DEF"], #(#[TYPE, "TYPE"], t2), id), v); } ;

varInitializer
        :  (ASSIGN^ initializer)? ;

initializer
        :  expr | arrayInit ;

arrayInit
        : LBRACE^ initializer (COMMA! initializer)* RBRACE! ;

stmt    : (decl)=> decl SEMI!  //Check to see if this is a declaration, if not proceed
to next
        | expr SEMI!
        | block
        // If-else statement
        | "if"^ LPAREN! expr RPAREN! stmt
          (
            // CONFLICT: the old "dangling-else" problem...
            //          ANTLR generates proper code matching
            //                              as soon as possible.  Hush warning.
            options {
                  warnWhenFollowAmbig = false;
            }
          :
            "else"! stmt
          )?
```

```
        // For statement
        | "for"^
            LPAREN!
              forInit SEMI!   // initializer
              forCond SEMI!   // condition test
              forIter         // updater
            RPAREN!
              stmt       // statement to loop over
        | "return"^ (expr)? SEMI!
        | SEMI!
        ;

forInit  : (
             (decl)=> decl
             | exprList
           )?
           {#forInit = #(#[FOR_INIT,"FOR_INIT"], #forInit);}
            ;

forCond : (expr)?
          {#forCond = #(#[FOR_CONDITION,"FOR_CONDITION"],#forCond);}
           ;

forIter : (exprList)?
          {#forIter = #(#[FOR_ITERATOR,"FOR_ITERATOR"],#forIter);}
           ;

expr    : assignExpr {#expr = #(#[EXPR, "EXPR"], expr); } ;

assignExpr
        : orExpr (( ASSIGN^ | PLUS_ASSIGN^ ) assignExpr)? ;
orExpr  : andExpr (LOR^ { #orExpr.setType(LOGICAL); } andExpr)* ;
andExpr : compExpr (LAND^ { #andExpr.setType(LOGICAL); } compExpr)* ;
compExpr
        : relExpr ((EQUAL^) { #compExpr.setType(LOGICAL); } relExpr)* ;
relExpr : addExpr ((LT^ | GT^ | LTEQ^ | GTEQ^) {
#relExpr.setType(LOGICAL); } addExpr)* ;
addExpr : mulExpr ((PLUS^ | MINUS^) mulExpr)* ;
mulExpr : unary ((STAR^ | DIV^) unary)* ;
unary   : (INC^ unary | DEC^ unary | MINUS^ unary | NOT^ unary) {
#unary.setType(NEGATE); } | factor ;

factor  : (LPAREN! assignExpr RPAREN!) | constant | fieldOrMethod | newExpr
;

constant
```

```
        : NUM
        | REAL
        | STR_LITERAL
        | "true"
        | "false"
        ;

exprList
        : expr (COMMA! expr)*
        {#exprList = #(#[ELIST, "ELIST"], exprList);}
        ;

argList
        : exprList | /* nothing */ ;

/* fieldOrMethod can be of formats - a, a(), a.b(), a.b.c...() */
fieldOrMethod
        : ID
            (
              options { greedy=true; } : DOT^ ID
            )*
            (
              (
                options { greedy=true; } : ( LPAREN^ argList RPAREN! )
                {#fieldOrMethod = #(#[METHOD_CALL,"METHOD_CALL"],
fieldOrMethod);}
              )
              | (LBRACK^ expr RBRACK!)+
            )?
          (inc:INC^ { #inc.setType(NEGATE); } | dec:DEC^ {
#dec.setType(NEGATE); } )?
        ;

newExpr : "new"^ type ((LPAREN! argList RPAREN!) | newArryInit) {
#newExpr.setType(CONSTR_CALL); } ;

newArryInit
        : (LBRACK^ expr RBRACK!) ;
```

A.2 JLiteWalker

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

/*************************************************************************
******

   A tree walker that builds up an IR that can easily be converted
                              to bytecode


*************************************************************************
*****/


class JLiteWalker extends TreeParser;
{
  JLiteSymbolTable global = new JLiteSymbolTable(null);
  JLiteSymbolTable current = null;

  public static void setSrcInfo(JLiteExpr e, AST astNode) {
    SourceInfo srcInfo = new SourceInfo();
    srcInfo.linenumber = astNode.getLine();
    srcInfo.colnumber = astNode.getColumn();
    srcInfo.srctext = astNode.getText();
    e.srcInfo(srcInfo);
  }
}

classdef returns [JLiteClass c]
{
    c = null;
    JLiteMethod m = null;
}
        : #(CLASSDEF ID
        { c = new JLiteClass(#ID.getText()); }
          /* Add all the methods to JLiteClass */
        ( m = func { c.addCompilationUnit(m); } )*
        )
        ;

func returns [JLiteMethod m]
{
```

```
   m = null;
   JLiteStmt s = null;
   JLiteExprList pl = null;
   JLiteType rtype = null;
}
      : #(METHOD_DEF
         {
          current = new JLiteSymbolTable(global);
         }
         rtype=type ID pl=paramList[rtype]
         {
          m = new JLiteMethod(#ID.getText(), rtype, pl, current);
         }
         (
          s = block { m.addBlock(s); }
         )
        )
      ;

paramList[JLiteType rtype] returns [JLiteExprList e]
{
   List root = new LinkedList();
   e = new JLiteExprList(root, 0);
   JLiteId id = null;
}
      : #(PARAM_LIST (id=varDecl { root.add(id); } )*  )
      ;

varDecl returns [JLiteId id]
{
   id = null;
   JLiteType t = null;
}
      : #(PARAM_DEF t=type ID
                        {
                         JLiteId newId = current.getVariable(#ID.getText());
                         if (newId != null) {
                           System.out.println("Variable " + #ID.getText() + " already
defined.");
                         }
                         newId = new JLiteId(#ID.getText(), t,
current.getVarNum());
                         current.putVariable(newId);
                         id = newId;
                         if (id.type == JLiteType.DOUBLE)
                           current.getVarNum();
```

```
                        }
      )
      ;

block returns [JLiteStmt s]
{
   s = null;
}
      : #(LBRACE
         {
           current = new JLiteSymbolTable(current);
         }
         (s = stmts)?
         {
           current = current.getParent();
         }
       )
      ;

stmts returns [JLiteStmt s]
{
   s = null;
   JLiteStmt s1 = null;
}
      : s = stmt  (s1 = stmts { s = new JLiteSeq(s, s1); })?
      ;

stmt returns [JLiteStmt s]
{
  s = null;
  JLiteId newId = null;
  JLiteStmt s1 = null, s2 = null, s3 = null, s4 = null;
  JLiteType t = null;
  JLiteExpr e = null, a = null, b = null, c = null;
}
      : #(DECL newId=varDecl (e=expr)?     {
                                    s = new JLiteDeclStmt(newId, (JLiteSetExpr)e);
                                  }
        )
      | #(EXPR s=expr )
      | #(ELIST s=expr )
      | s=block
      | #("if"  a=expr s1=stmt (s2=stmt)?     { s = new
JLiteIfStmt((JLiteLogicalExpr)a, s1, s2); } )
      | #("for" s1=forInit s2=forCond s3=forIter s4=stmt
```

```
                                { s = new JLiteForStmt(s1, (JLiteLogicalExpr)s2,
s3, s4); } )
      | #("return" (e=expr)?              { s = new JLiteReturnStmt(e); } )
      | SEMI                                  { s = JLiteStmt.Null; }
      ;

varinit returns [JLiteExpr a]
{
   a = null;
}
      : #(ASSIGN a=expr)
      ;

type returns [JLiteType typ]
{
   typ = null;
   JLiteExpr e = null;
}
      : #(TYPE typ=type)
      | (
        "boolean"      { typ = JLiteType.BOOL;    }
      | "char"         { typ = JLiteType.CHAR;    }
      | "int"          { typ = JLiteType.INT;     }
      | "float"        { typ = JLiteType.FLOAT;   }
      | "double"       { typ = JLiteType.DOUBLE;  }
      | "void"         { typ = JLiteType.VOID;    }
        )
      | #(LBRACK typ=type
                    {
                      if (typ instanceof JLiteArrayType)
                        ((JLiteArrayType)typ).arryDim++;
                      else
                        typ = new JLiteArrayType(typ);
                    }
      )
      | (e=identifier[false, null]     { typ = new JLiteObjType(e); } )
      ;

forInit returns [JLiteStmt e]
{
   e = null;
}
      : #(FOR_INIT e=stmt) ;

forCond returns [JLiteStmt e]
{
```

```
   e = null;
}
      : #(FOR_CONDITION e=stmt) ;

forIter returns [JLiteStmt e]
{
   e = null;
}
      : #(FOR_ITERATOR (e=expr)?) ;

expr returns [JLiteExpr e]
{
   JLiteExpr a, b = null;
   JLiteType t = null;
   e = null;
   LinkedList root = null;
}
      : #(NUM                  { e = new JLiteConst(#NUM.getText(),
JLiteType.INT); } )
      | #(REAL                       { e = new JLiteConst(#REAL.getText(),
JLiteType.DOUBLE); } )
      | #("true"               { e = JLiteConst.True; } )
      | #("false"                    { e = JLiteConst.False; } )
      | #(STR_LITERAL            { e = new
JLiteConst(#STR_LITERAL.getText(), JLiteType.STRING); } )
      | #(PLUS   a=expr b=expr       { e = new JLiteArith(JLiteArith.PLUS, a, b);
} )
      | #(MINUS  a=expr b=expr       { e = new JLiteArith(JLiteArith.MINUS, a,
b); } )
      | #(STAR    a=expr b=expr      { e = new JLiteArith(JLiteArith.MUL, a, b);
} )
      | #(DIV    a=expr b=expr       { e = new JLiteArith(JLiteArith.DIV, a, b);
} )
      | #(NEGATE a=expr             { e = new JLiteUnary(#NEGATE.getText(),
a); } )
      | #(LOGICAL a=expr b=expr      { e = new
JLiteLogicalExpr(#LOGICAL.getText(), a, b); setSrcInfo(e, #LOGICAL); } )
      | #(ASSIGN  a=expr (b=expr)?   { e = new JLiteSetExpr(a, b); setSrcInfo(e,
#ASSIGN); } )
      | #(LPAREN  a=expr (b=expr)?   { e=new JLiteMethodCall(a, b);
setSrcInfo(e, #LPAREN); } )
      | #(LBRACE  e=exprlist[0])
      | #(LBRACK  e=exprlist[JLiteExprList.ARRAY_REF])
      | #(ELIST   e=exprlist[0])
      | #(METHOD_CALL  e=expr )
      | #(CONSTR_CALL  a=expr (b=expr)?
```

```
                                { e = new
JLiteMethodCall(JLConstants.CONSTRUCTOR, a, b); } )
        | #(EXPR e=expr )
        | (e=identifier[true, null])
        ;

identifier[boolean resolve, JLiteRefPtr ptr] returns [JLiteExpr e]
{
   e = null;
   JLiteRefPtr e1 = null;
}
        : #(ID                    {
                                  JLiteId id = null;
                                  if (resolve) {
                                    id = current.getVariable(#ID.getText());
                                  }
                                  if (id == null) {
                                    id = new JLiteId(#ID.getText(), null, -1);
                                    setSrcInfo(id, #ID);
                                  }
                                  if (ptr != null) {
                                     ptr.addRef(id);
                                     e = ptr;
                                  } else {
                                     e = new JLiteRefPtr(id);
                                  }
                                 }
        )
        | #(DOT    { if (ptr == null) ptr = new JLiteRefPtr(); }
(e=identifier[resolve, ptr] { resolve = false; })*  )
        ;

exprlist[int typeref] returns [JLiteExprList e]
{
   e = null;
   JLiteExpr e1 = null;
   LinkedList root = new LinkedList();
}
        : ( e1=expr { root.add(e1); } )*
          { e = new JLiteExprList(root, typeref); }
        ;
```

B.2 JLiteCompiler Source Code

B.2.1 src/ CompilationUnit

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public abstract class CompilationUnit {

  static boolean traceon = false;
  static ByteCodeGen bytecodeGen = null;
  static int UNI_NUM = 0;
  static int errors = 0;
  static String errorMsg = "Line:{0} Column:{1} {2}";

  SourceInfo srcInfo;

  public CompilationUnit() {
  }

  public abstract void compile();
  public abstract void typecheck();

  public void precompile() {}

  public void log(String msg) {
    if (traceon)
      System.out.println(msg);
  }

  public void logError(String msg) {
    errors++;
    if (srcInfo != null)
      System.out.println(java.text.MessageFormat.format(errorMsg,
          new Object[] {String.valueOf(srcInfo.linenumber),
          String.valueOf(srcInfo.colnumber), msg}));
```

```
    else
      System.out.println(java.text.MessageFormat.format(errorMsg,
          new Object[] {String.valueOf(0),
          String.valueOf(0), msg}));
    System.out.println();
  }

  public void srcInfo(SourceInfo srcInfo) {
    this.srcInfo = srcInfo;
  }

}
```

B.2.2 JLConstants

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class JLConstants {

  public static final int
      UNKNOWN = 0,
      CLASS = 1,
      METHOD = 2,
      FIELD = 3,
      CONSTRUCTOR = 4,
      PACKAGE = 5;

}
```

B.2.3 JLiteArith

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
```

```
 * @author Sudhakar Perumal
 * @version 1.0
 */


public class JLiteArith extends JLiteOp {

  public JLiteExpr expr1, expr2;

  public static final int PLUS = 1;
  public static final int MINUS = 2;
  public static final int MUL = 3;
  public static final int DIV = 4;

  public JLiteArith(int op, JLiteExpr x1, JLiteExpr x2) {
    super(String.valueOf(op), new JLiteType((String)null));
    expr1 = x1;
    expr2 = x2;
  }

  public void typecheck() {
    expr1.typecheck();
    if (expr2 != null) expr2.typecheck();

    if (JLiteType.DOUBLE.equals(expr1.type) ||
JLiteType.DOUBLE.equals(expr2.type))
      this.type = JLiteType.DOUBLE;
    else
      this.type = expr1.type;
  }

  public void compile() {
    expr1.compile();
    if (expr2.getType().equals(JLiteType.DOUBLE) &&
!expr1.getType().equals(JLiteType.DOUBLE))
      bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.I2D,
(JLiteType)null));
    if (expr2 != null) {
      expr2.compile();
      if (expr1.getType().equals(JLiteType.DOUBLE) &&
          !expr2.getType().equals(JLiteType.DOUBLE))
        bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.I2D,
(JLiteType)null));
    }
    switch (Integer.parseInt(this.token)) {
      case PLUS:
```

```
        bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.ADD,
this.type));
        break;
      case MINUS:
        bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.MINUS,
this.type));
        break;
      case MUL:
        bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.MUL,
this.type));
        break;
      case DIV:
        bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.DIV,
this.type));
        break;

    }

  }
}
```

B.2.4 JLiteArrayType

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class JLiteArrayType extends JLiteType {

  public int arryDim = 1;
  JLiteType basetype;

  public JLiteArrayType(JLiteType basetype) {
    super((String)null);
    this.basetype = basetype;
  }

  public void typecheck() {
    basetype.typecheck();
```

```
    this.clsName = basetype.getClassName();
    resolved = basetype.resolved;
  }

}
```

B.2.5 JLiteBlock

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class JLiteBlock
    extends CompilationUnit {

  private JLiteStmt s;

  public JLiteBlock(JLiteStmt s) {
    this.s = s;
  }

  public void typecheck() {
    if (s != null) s.typecheck();
  }

  public void compile() {
    if (s != null) s.compile();
  }
}
```

B.2.6 JLiteClass

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

/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
```

```
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class JLiteClass extends CompilationUnit {

  static String name;

  private LinkedList cunits = new LinkedList();

  public JLiteClass(String name) {
    this.name = name;
  }

  public void typecheck() {
    for (Iterator iter=cunits.iterator(); iter.hasNext(); ) {
         CompilationUnit cu = (CompilationUnit)iter.next();
         cu.typecheck();
    }
  }

  public void precompile() {
    bytecodeGen = new ByteCodeGenImpl(name);
    for (Iterator iter=cunits.iterator(); iter.hasNext(); ) {
      CompilationUnit cu = (CompilationUnit)iter.next();
      cu.precompile();
    }
  }

  public void compile() {
    bytecodeGen = new ByteCodeGenImpl(name);
    for (Iterator iter=cunits.iterator(); iter.hasNext(); ) {
      CompilationUnit cu = (CompilationUnit)iter.next();
      cu.compile();
    }
  }

  public String getName() {
    return this.name;
  }

  public void addCompilationUnit(CompilationUnit cu) {
    cunits.add(cu);
  }
```

```
  public void dumpByteCode(String filename) throws Exception {
    bytecodeGen.dumpByteCode(filename);
  }

}
```

B.2.7 JLiteConst

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class JLiteConst
    extends JLiteExpr {

  public JLiteConst(String tok, JLiteType p) {
    super(tok, p);
  }

  public JLiteConst(int i) {
    super("" + i, JLiteType.INT);
  }

  public static final JLiteConst
      True = new JLiteConst("true", JLiteType.BOOL),
      False = new JLiteConst("false", JLiteType.BOOL);

  public void compile() {
    bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.LOAD, this));
  }

  public void typecheck() {
  }

}
```

B.2.8 JLiteDeclStmt

```
/**
```

```
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

import java.util.Iterator;

public class JLiteDeclStmt
    extends JLiteStmt {

  private JLiteId id;
  private JLiteSetExpr expr;

  public JLiteDeclStmt(JLiteId id, JLiteSetExpr expr) {
    this.id = id;
    this.expr = expr;
  }

  public void typecheck() {
    if (expr == null) {
      logError("Variable not initialized");
      return;
    }
    id.typecheck();
    if (expr != null) expr.typecheck();
    if (!ReflectHelper.isAssignable(expr.type, id.type)) {
      logError("Invalid Assignment");
    }
  }

  public void compile() {
    JLiteArrayType arryType = null;
    if ( (id instanceof JLiteId) && (id.type instanceof JLiteArrayType)) {
      JLiteSetExpr setOp = (JLiteSetExpr) expr;
      compileArray((JLiteId) id, (JLiteExprList)setOp.id);
    }
    else {
      expr.compile();
      bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.STORE, id));
    }
  }

  private void compileArray(JLiteId l_id, JLiteExprList arryInitList) {
```

```
    int size = arryInitList.lexpr.size();
    int dimension = ((JLiteArrayType)l_id.type).arryDim;

    bytecodeGen.addInstruction(JLStackIR.initArray(size, l_id.type));
    compile(arryInitList, 0, new int[dimension+1]);
    bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.STORE, l_id));
   }

   private void printArry(JLiteExpr obj, int[] dimension) {
    bytecodeGen.addInstruction(JLStackIR.arrayIndex(dimension));
    obj.compile();
    bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.IASTORE,
(JLiteType)null));
   }

   public void compile(JLiteExprList l_list, int depth, int[] dimension) {
    Object obj = null;
    int l_depth = depth;
    for (Iterator iter = l_list.lexpr.iterator(); iter.hasNext(); ) {
     obj = iter.next();
     depth = l_depth;
     if (obj instanceof JLiteExprList) {
      compile((JLiteExprList)obj, ++depth, dimension);
     }
     else {
      printArry((JLiteExpr)obj, dimension);
     }
     ++dimension[l_depth];
    }
    dimension[l_depth] = 0;
   }


}
```

B.2.9 JLiteExpr

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */
```

```
public class JLiteExpr extends JLiteStmt {

  public String token;
  public JLiteType type;

  JLiteExpr(String token, JLiteType p) { this.token = token; type = p; }

  public void typecheck() {
  }

  public JLiteType getType() {
    return type;
  }
}
```

B.2.10 JLiteExprList

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

import java.util.*;
import java.lang.reflect.*;

/**
 * Object to hold list of expressions.
 * This could also point to an array.
 * Because an array could have list of expressions as index.
 */

public class JLiteExprList extends JLiteExpr {

  List lexpr;
  int typeref = 0;

  public static final int ELIST_REF = 1;
  public static final int ARRAY_REF = 2;

  public JLiteExprList(List lexpr, int typeref) {
```

```
    super(null, null);
    this.lexpr = lexpr;
    this.typeref = typeref;
  }

  public void compile() {
    JLiteExpr expr = null;
    for (Iterator iter=lexpr.iterator(); iter.hasNext();) {
      expr = (JLiteExpr) iter.next();
      expr.compile();
    }
    if (typeref == ARRAY_REF)
      bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.IALOAD,
(JLiteType)null));
  }

  public void typecheck() {
    JLiteExpr expr = null;
    int i = 0;
    for (Iterator iter=lexpr.iterator(); iter.hasNext();) {
      expr = (JLiteExpr) iter.next();
      expr.typecheck();
    }
    /**
     * If this is an array reference, type of this exprlist will be the type of
     * first element.
     */
    if (typeref == ARRAY_REF) {
      expr = (JLiteExpr) lexpr.get(0);
      type = expr.type;
    }
  }

}
```

B.2.11 JLiteForStmt

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */
```

```
public class JLiteForStmt extends JLiteStmt {

  JLiteStmt forInit, forIter, forbody;
  JLiteLogicalExpr forCond;

  public JLiteForStmt(JLiteStmt forInit, JLiteLogicalExpr forCond,
                      JLiteStmt forIter, JLiteStmt forbody) {
    this.forInit = forInit;
    this.forCond = forCond;
    this.forIter = forIter;
    this.forbody = forbody;
  }

  public void typecheck() {
    this.forInit.typecheck();
    this.forCond.typecheck();
    this.forIter.typecheck();
    this.forbody.typecheck();
  }

  public void compile() {
    int startNum = ++UNI_NUM;
    int endNum = ++UNI_NUM;
    forInit.compile();
    bytecodeGen.addInstruction(JLStackIR.refnum(startNum));
    if (forCond instanceof JLiteLogicalExpr)
      forCond.compile(-1, endNum);
    forbody.compile();
    forIter.compile();
    bytecodeGen.addInstruction(JLStackIR.gotoinst(startNum));
    bytecodeGen.addInstruction(JLStackIR.refnum(endNum));
  }
}
```

B.2.12 JLiteId

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */
```

```
public class JLiteId extends JLiteExpr {

  int varNum;

  public JLiteId(String id, JLiteType p, int varNum) { super(id, p);  this.varNum =
varNum; }

  public void compile() {
  }

  public void typecheck() {
    if (type != null) type.typecheck();
  }
}
```

B.2.13 JLiteIfStmt

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class JLiteIfStmt
    extends JLiteStmt {

  JLiteLogicalExpr booleanExpr;
  JLiteStmt ifpart, elsepart;
  String logicalOp;

  public JLiteIfStmt(JLiteLogicalExpr booleanExpr, JLiteStmt ifpart,
                     JLiteStmt elsepart) {
    this.booleanExpr = booleanExpr;
    this.ifpart = ifpart;
    this.elsepart = elsepart;
  }

  public void compile() {
    int startNum = ++UNI_NUM;
    int ifpartNum = ++UNI_NUM;
    int endNum = ++UNI_NUM;
    int elseNum = -1;
    if (elsepart != null)
```

```
    elseNum = ++UNI_NUM;
   int nextStmt = (elseNum != -1) ? elseNum : endNum;
   booleanExpr.compile(ifpartNum, nextStmt);
   bytecodeGen.addInstruction(JLStackIR.refnum(ifpartNum));
   this.ifpart.compile();
   /** goto end number **/
   bytecodeGen.addInstruction(JLStackIR.gotoinst(endNum));
   if (elsepart != null) {
    bytecodeGen.addInstruction(JLStackIR.refnum(elseNum));
    elsepart.compile();
   }
   /** insert end number **/
   bytecodeGen.addInstruction(JLStackIR.refnum(endNum));
  }

  public void typecheck() {
   booleanExpr.typecheck();
   ifpart.typecheck();
   if (elsepart != null) elsepart.typecheck();
  }

}
```

B.2.14 JLiteLogicalExpr

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class JLiteLogicalExpr extends JLiteExpr {

  JLiteExpr expr1, expr2;

  public JLiteLogicalExpr(String token, JLiteExpr expr1, JLiteExpr expr2) {
   super(null, null);
   this.token = token;
   this.expr1 = expr1;
   this.expr2 = expr2;
  }

  public void compile(int ifpartNum, int elsePartNum) {
```

```
    int type = -1, jumpto = -1;
    boolean shortCircuit = false;
    if ("==".equals(token)) {
      type = JLStackIR.IFCMPNE;
      jumpto = elsePartNum;
    } else if ("&&".equals(token)) {
      type = JLStackIR.IFCMPNE;
      jumpto = elsePartNum;
      shortCircuit = true;
    } else if ("||".equals(token)) {
      type = JLStackIR.IFCMPEQ;
      jumpto = ifpartNum;
      shortCircuit = true;
    } else if ("<".equals(token)) {
      type = JLStackIR.IFCMPGEQ;
      jumpto = elsePartNum;
    } else if (">".equals(token)) {
      type = JLStackIR.IFCMPLEQ;
      jumpto = elsePartNum;
    } else if ("<=".equals(token)) {
      type = JLStackIR.IFCMPGT;
      jumpto = elsePartNum;
    } else if (">=".equals(token)) {
      type = JLStackIR.IFCMPLT;
      jumpto = elsePartNum;
    }
    if (type != -1) {
      JLiteType superType = ReflectHelper.getDominable(expr1.type, expr2.type);
      if (shortCircuit) {
        ( (JLiteLogicalExpr) expr1).compile( -1, -1);
        if (JLiteType.INT.equals(expr1.type) &&
JLiteType.DOUBLE.equals(superType))
          bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.I2D,
(JLiteType)null));
        bytecodeGen.addInstruction(JLStackIR.ifcmp(type, superType, jumpto));
        ( (JLiteLogicalExpr) expr2).compile( -1, -1);
        if (JLiteType.INT.equals(expr2.type) &&
JLiteType.DOUBLE.equals(superType))
          bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.I2D,
(JLiteType)null));
        bytecodeGen.addInstruction(JLStackIR.ifcmp(type, superType, jumpto));
      } else {
        expr1.compile();
        if (JLiteType.INT.equals(expr1.type) &&
JLiteType.DOUBLE.equals(superType))
```

```
        bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.I2D,
(JLiteType)null));
       expr2.compile();
       if (JLiteType.INT.equals(expr2.type) &&
JLiteType.DOUBLE.equals(superType))
        bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.I2D,
(JLiteType)null));
       if (jumpto != -1)
        bytecodeGen.addInstruction(JLStackIR.ifcmp(type, superType, jumpto));
     }
    }
  }

  public void typecheck() {
    if (expr1 != null) expr1.typecheck();
    if (expr2 != null) expr2.typecheck();
    type = expr1.getType();
  }

}
```

B.2.15 JLiteMethod

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

import java.util.*;

public class JLiteMethod extends CompilationUnit {

  private String name;
  private JLiteSymbolTable symTable;
  JLiteType returnType;
  JLiteExprList paramList;

  private LinkedList blocks = new LinkedList();

  public JLiteMethod(String name, JLiteType rtype, JLiteExprList paramList,
JLiteSymbolTable symTable) {
    this.name = name;
```

```
    this.symTable = symTable;
    this.returnType = rtype;
    this.paramList = paramList;
  }

  public void precompile() {
    returnType.typecheck();
    paramList.typecheck();
    bytecodeGen.createMethod(this.name, this.returnType, this.paramList);
    bytecodeGen.addInstruction(JLStackIR.returnInst(returnType));
    bytecodeGen.finalizeMethod();
  }

  public void typecheck() {
    bytecodeGen.loadMethod(this.name);
    returnType.typecheck();
    paramList.typecheck();
    for (Iterator iter=blocks.iterator(); iter.hasNext(); ) {
      CompilationUnit cu = (CompilationUnit)iter.next();
      if (cu != null)
        cu.typecheck();
      }
  }

  public void compile() {
    try {
      bytecodeGen.createMethod(this.name, this.returnType, this.paramList);
      for (Iterator iter=blocks.iterator(); iter.hasNext(); ) {
        CompilationUnit cu = (CompilationUnit)iter.next();
        if (cu != null)
          cu.compile();
      }
      if (returnType == JLiteType.VOID)
        bytecodeGen.addInstruction(JLStackIR.returnInst(returnType));
      bytecodeGen.finalizeMethod();
    } catch (Exception e) {
      logError(e.toString());
    }
  }

  public void addBlock(JLiteStmt b) {
      blocks.add(b);
  }

}
```

B.2.16 JLiteMethodCall

```
import java.util.*;
import java.lang.reflect.*;

/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class JLiteMethodCall extends JLiteExpr {

  private JLiteExpr expr1, expr2;
  private int nodeType;

  private String[] argTypes;
  private List resolvedList;
  private boolean virutalFunCall;
  private boolean localFuncCall;

  public JLiteMethodCall(int nodetype, JLiteExpr a, JLiteExpr b) {
    super(null, new JLiteObjType());
    expr1 = a;
    expr2 = b;
    nodeType = nodetype;
  }

  public JLiteMethodCall(JLiteExpr a, JLiteExpr b) {
    this(JLConstants.METHOD, a, b);
  }

  public void typecheck() {
    List linearList = null;
    Object lastObj = null;
      if (expr1 instanceof JLiteRefPtr)
        linearList = ((JLiteRefPtr)expr1).getPtrList();
      /**
       * If this is a method call and list has only one element then
       * it must be a local function call
       * **/
      if ( (nodeType == JLConstants.METHOD) && (linearList.size() == 1)) {
        localFuncCall = true;
```

```
        linearList.add(0, JLiteClass.name);
      }

      if (expr2 != null)
        ((JLiteExprList)expr2).typecheck();
      resolvedList = resolve(linearList, ((JLiteExprList)expr2));
      if ((resolvedList == null) || (resolvedList.size() == 0)) {
        return;
      }
      lastObj = resolvedList.get(resolvedList.size()-1);
      if (lastObj instanceof ResolvedObj) {
        type.clsName = ((ResolvedObj)lastObj).returnType;
        if (((ResolvedObj)lastObj).type == JLConstants.METHOD) {
            argTypes = ((ResolvedObj)lastObj).argType;
            virutalFunCall = !((ResolvedObj)lastObj).isInterface;
        }
        type.resolved = true;
      }
      if (!bytecodeGen.isVirtualFunction() && localFuncCall) {
        logError("static function call not allowed in this context");
      }
    }

    public void compile() {
      if ((nodeType == JLConstants.METHOD) || (nodeType ==
JLConstants.CONSTRUCTOR)) {
        compile(nodeType, resolvedList, (JLiteExprList)expr2);
      } else {
        ;
      }
    }

    private List resolve(List tobeResolved, JLiteExprList argList) {
      Object prev = null, cObj = null;
      LinkedList finalResList = new LinkedList();
      int hinttype = -1;
      try {
        for (int i=0; i<tobeResolved.size(); i++) {
          cObj = tobeResolved.get(i);
          if (i == tobeResolved.size() - 1) { //If the item is last one in the list
            hinttype = nodeType; //Enforce hint
          }
          if (cObj instanceof String)
            prev = ReflectHelper.resolveType((String)cObj,
                                        JLConstants.CLASS,
                                        (ResolvedObj) prev, argList);
```

```
      else if (( (JLiteId) cObj).type != null) {
        prev = ((JLiteId) cObj);
      }
      else {
        if (prev instanceof JLiteId)
          prev = toResolvedObj((JLiteId)prev);
        prev = ReflectHelper.resolveType( ( (JLiteId) cObj).token,
                                  hinttype,
                                  (ResolvedObj) prev, argList);
      }
      addResolvedType(finalResList, prev);
     }
    } catch (Exception e) {
     logError((e.getMessage() != null) ? e.getMessage() : e.toString());
    }
    return finalResList;
  }

  private void addResolvedType(LinkedList resolvedList, Object resolvedObj) {
    ResolvedObj rObj = null;
    if (resolvedObj instanceof ResolvedObj) {
      rObj = (ResolvedObj)resolvedObj;
      if ( (rObj.type == JLConstants.FIELD) ||
         (rObj.type == JLConstants.CLASS) ||
         (rObj.type == JLConstants.METHOD) ||
         (rObj.type == JLConstants.CONSTRUCTOR)) {
       resolvedList.add(rObj);
      }
    } else if (resolvedObj instanceof JLiteId)
      resolvedList.add(resolvedObj);
  }

  private void compile(int l_nodeType, List l_resolvedList, JLiteExprList argList)
{
    Object cObj = null, pObj = null;
    String typename = null;
    JLiteExprList exprList = null;
    Class[] argClsType = (argList != null) ?
ReflectHelper.getTypeList(argList.lexpr) : null;

    if ((virutalFunCall) && (bytecodeGen.isVirtualFunction() && localFuncCall))
      bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.ALOAD,
(JLiteExpr)null));

    for (int i=0; i<l_resolvedList.size(); i++) {
```

```
      cObj = l_resolvedList.get(i);
      if (pObj instanceof JLiteId)
        pObj = toResolvedObj((JLiteId)pObj);
      if (cObj instanceof JLiteId)
        bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.LOAD,
(JLiteId)cObj));
      else {
        switch (((ResolvedObj)cObj).type) {
          case JLConstants.CONSTRUCTOR:
            bytecodeGen.addInstruction(JLStackIR.createNew((ResolvedObj)cObj));
            if (argList != null)
              argList.compile();
            bytecodeGen.addInstruction(JLStackIR.initObject((ResolvedObj)cObj));
            break;
          case JLConstants.FIELD:
            bytecodeGen.addInstruction(JLStackIR.fieldRef((ResolvedObj)cObj,
((ResolvedObj)pObj).returnType));
            break;
          case JLConstants.METHOD:
            if (argList != null)
              argList.compile();

bytecodeGen.addInstruction(JLStackIR.invokeMethod((ResolvedObj)cObj,
((ResolvedObj)pObj).returnType));
            break;
        }
      }
      pObj = cObj;
    }
  }

  private ResolvedObj toResolvedObj(JLiteId id) {
    ResolvedObj rObj = new ResolvedObj();
    rObj.type = JLConstants.CLASS;
    rObj.name = id.token;
    rObj.returnType = id.type.getClassName();
    return rObj;
  }

}
```

B.2.17 JLiteObjType

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

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class JLiteObjType extends JLiteType {

  JLiteExpr clsId;

  public JLiteObjType(JLiteExpr clsId) {
    super((String)null);
    this.clsId = clsId;
  }

  public JLiteObjType() {
    super((String)null);
  }

  public void typecheck() {
    if (clsId instanceof JLiteId)
      clsName = ((JLiteId)clsId).token;
    else if (clsId instanceof JLiteRefPtr)
      clsName = ((JLiteRefPtr)clsId).getFullName();
    super.typecheck();
  }

}
```

B.2.18 JLiteOp

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class JLiteOp extends JLiteExpr {
   public JLiteOp(String opcode, JLiteType p) { super(opcode, p); }
}
```

B.2.19 JLiteRefPtr

```
import org.apache.bcel.*;
import org.apache.bcel.classfile.*;
import org.apache.bcel.generic.*;
import java.util.*;

/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class JLiteRefPtr extends JLiteExpr {

  boolean store = false;
  List list = list = new LinkedList();
  JLiteId idref;
  int nodeType = 0;

  /**
   * This object is used to represent identifiers.
   * These collection of identifier may point to an id, field or method
   * For Example: a, a.b,
   */

  public JLiteRefPtr() {
    super(null, null);
  }

  public JLiteRefPtr(JLiteId id) {
    super(null, id.type);
    idref = id;
    list.add(idref);
  }

  public void addRef(JLiteId l_idref) {
    list.add(l_idref);
    nodeType = 1;
  }

  public void typecheck() {
```

```
    Object obj = null;
    if (idref != null) {
      idref.typecheck();
      type = idref.type;
    } else if (list != null) {
      for (Iterator iter = this.list.iterator(); iter.hasNext(); ) {
        obj = iter.next();
        ( (JLiteExpr) obj).typecheck();
      }
      type = ( (JLiteExpr)this.list.get(0)).type; //Take the type of first element
    }
  }

  public void compile() {
    compile(false);
  }

  public void compile(boolean store) {
    if (idref != null) {
      if (store)
        bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.STORE,
idref));
      else
        bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.LOAD, idref));
    } else {
      ; //leave it to the holder of this object to compile
    }
  }

  public List getPtrList() {
    return list;
  }

  public String getFullName() {
    StringBuffer sb = new StringBuffer();
    for (int i=0; i<list.size(); i++) {
      sb.append( ( (JLiteId) list.get(i)).token);
      if (i < (list.size() - 1))
        sb.append(".");
    }
    return sb.toString();
  }

}
```

B.2.20 JLiteReturnStmt

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class JLiteReturnStmt extends JLiteStmt {

  JLiteExpr expr;

  public JLiteReturnStmt(JLiteExpr e) {
    this.expr = e;
  }

  public void typecheck() {
    if (expr != null) expr.typecheck();
  }

  public void compile() {
    if (expr != null) {
      expr.compile();
    }
    bytecodeGen.addInstruction(JLStackIR.returnInst(expr.type));
  }
}
```

B.2.21 JLiteSeq

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class JLiteSeq extends JLiteStmt {

    JLiteStmt stmt1;
    JLiteStmt stmt2;
```

```
    public JLiteSeq(JLiteStmt s1, JLiteStmt s2) {
      stmt1 = s1; stmt2 = s2;
    }

    public String toString() {
      StringBuffer sb = new StringBuffer();
      sb.append(stmt1.toString());
      sb.append("\n");
      if (stmt2 != null)
        sb.append(stmt2.toString());
      return sb.toString();
    }

    public void typecheck() {
      if (stmt1 != null) stmt1.typecheck();
      if (stmt2 != null) stmt2.typecheck();
    }

    public void compile() {
      if (stmt1 != null) {
        stmt1.compile();
      }
      if (stmt2 != null) {
        stmt2.compile();
      }
    }
}
```

B.2.22 JLiteSetExpr

```
import java.util.*;

/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class JLiteSetExpr
    extends JLiteExpr {

  public JLiteExpr id;
```

```
  public JLiteExpr expr;

  public JLiteSetExpr(JLiteExpr i, JLiteExpr x) {
    super(null, null);
    id = i;
    expr = x;
  }

  public void typecheck() {
    if (expr != null) expr.typecheck();
    if (id instanceof JLiteRefPtr)
      ((JLiteRefPtr)id).typecheck();
    else
      id.typecheck();
    this.type = (expr != null) ? expr.type : id.type;
  }

  public void compile() {
    JLiteId l_id = null;
    if (expr != null) {
      expr.compile();
      if (ReflectHelper.getDominable(id.getType(),
          expr.type).equals(JLiteType.DOUBLE) &&
          !JLiteType.DOUBLE.equals(expr.type))
        bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.I2D,
(JLiteExpr)null));
    }
    if (id instanceof JLiteRefPtr)
      ((JLiteRefPtr)id).compile((expr != null));
    else
      id.compile();
  }

}
```

B.2.23 JLiteStmt

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */
```

```
public class JLiteStmt
   extends CompilationUnit {

  public static JLiteStmt Null = new JLiteStmt();

  public JLiteStmt() {
  }

  public void typecheck() {}

  public void compile() {
  }
}
```

B.2.24 JLiteSymbolTable

```
import java.util.*;

/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class JLiteSymbolTable {

   private Hashtable table;
   protected JLiteSymbolTable outer;
   private int varnum = 0;

   public JLiteSymbolTable(JLiteSymbolTable st) {
      table = new Hashtable();
      outer = st;
      if (outer != null)
        varnum = outer.varnum;
   }

   public void putVariable(JLiteId id) {
      table.put(id.token, id);
   }

   public JLiteId getVariable(String token) {
```

```
        for (JLiteSymbolTable tab = this ; tab != null ; tab = tab.outer) {
            JLiteId id = (JLiteId)(tab.table.get(token));
            if ( id != null ) return id;
        }
        return null;
    }

    public JLiteSymbolTable getParent() {
      return outer;
    }

    public int getVarNum() {
      return ++varnum;
    }
}
```

B.2.25 JLiteType

```
import java.lang.reflect.*;

/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class JLiteType extends CompilationUnit {

    protected String clsName;
    protected boolean resolved = false;

    public JLiteType(Class cls) {
      clsName = cls.getName();
      resolved = true;
    }

    public JLiteType(String cls) {
      clsName = cls;
    }

    public static final JLiteType
            INT   = new JLiteType(int.class),
            FLOAT = new JLiteType(float.class),
```

```
      DOUBLE = new JLiteType(double.class),
         CHAR  = new JLiteType(char.class),
         BOOL  = new JLiteType(boolean.class),
         STRING = new JLiteType(String.class),
      VOID = new JLiteType(Void.class);

   public void compile() {
   }

   public void typecheck() {
    if (resolved == false) {
     try {
      ResolvedObj c = ReflectHelper.resolveType(clsName,
JLConstants.CLASS, null, null);
       if (c != null) {
        resolved = true;
        try {
         clsName = Class.forName(c.name).getName();
        } catch (Exception e) {
         ;
         clsName = c.name;
        }
       }
     } catch (Exception e) {
      logError(e.toString());
     }
    }
   }

   public String getClassName() {
    if (!resolved) {
     logError("Unknown type ");
     return null;
    }
    return clsName;
   }

   public boolean equals(Object cmpType) {
    if (cmpType instanceof JLiteType)
     return ((JLiteType)cmpType).clsName.equals(this.clsName);
    else
     return false;
   }
}
```

B.2.26 JLiteUnary

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */


public class JLiteUnary extends JLiteOp {
   public JLiteExpr expr;
   public JLiteUnary(String tok, JLiteExpr e) {
         super(tok, null); expr = e;
   }

   public void typecheck() {
     expr.typecheck();
     this.type = expr.type;
     this.type.resolved = true;
   }

   public void compile() {
     if ("++".equals(token)) {
       bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.INC, expr));
     } else if ("--".equals(token)) {
       bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.DEC, expr));
     } else {
       expr.compile();
       bytecodeGen.addInstruction(JLStackIR.createSR(JLStackIR.UNARY,
(JLiteExpr)expr));
     }
   }
}
```

B.2.27 JLStackIR

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
```

```
 * @version 1.0
 */

public class JLStackIR {

  int opcode, size;
  int[] arrIndex;
  JLiteExpr expr;
  JLiteType type;
  String clsname;
  ResolvedObj resolvedObj;
  int instRefNum = -1, targetNum = -1;

  public static final int
     STORE = 1,
     LOAD = 2,
     PUSH = 3,
     GET_REF = 4,
     INVOKE_METHOD = 5,
     ADD = 6,
     ALOAD = 7,
     IASTORE = 8,
     IALOAD = 9,
     UNARY = 10,
     GETSTATIC = 11,
     INIT_ARRAY = 12,
     ARRAY_INDEX = 13,
     IFCMPNE = 14,
     IFCMPEQ = 15,
     IFCMPGT = 16,
     IFCMPGEQ = 17,
     IFCMPLT = 18,
     IFCMPLEQ = 19,
     GOTO = 20,
     SET_REF_NUM = 21,
     RETURN = 22,
     INC = 23,
     DEC = 24,
     INIT_OBJ = 25,
     NEW = 26,
     MINUS = 27,
     MUL = 28,
     DIV = 29,
     I2D = 30;

  private JLStackIR(int opcode) {
```

```
  this.opcode = opcode;
}

public static JLStackIR createSR(int opcode, JLiteType type) {
  JLStackIR bcn = new JLStackIR(opcode);
  bcn.type = type;
  return bcn;
}

public static JLStackIR createSR(int opcode, JLiteExpr expr) {
  JLStackIR bcn = new JLStackIR(opcode);
  bcn.expr = expr;
  return bcn;
}

public static JLStackIR fieldRef(ResolvedObj rObj, String clsName) {
  JLStackIR bcn = new JLStackIR(GETSTATIC);
  bcn.resolvedObj = rObj;
  bcn.clsname = clsName;
  return bcn;
}

public static JLStackIR invokeMethod(ResolvedObj rObj, String clsName) {
  JLStackIR bcn = new JLStackIR(INVOKE_METHOD);
  bcn.resolvedObj = rObj;
  bcn.clsname = clsName;
  return bcn;
}

public static JLStackIR initArray(int size, JLiteType type) {
  JLStackIR bcn = new JLStackIR(INIT_ARRAY);
  bcn.size = size;
  bcn.type = type;
  return bcn;
}

public static JLStackIR arrayIndex(int[] index) {
  JLStackIR bcn = new JLStackIR(ARRAY_INDEX);
  bcn.arrIndex = new int[index.length];
  System.arraycopy(index, 0, bcn.arrIndex, 0, index.length);
  return bcn;
}

public static JLStackIR refnum(int instRefNum) {
  JLStackIR bcn = new JLStackIR(SET_REF_NUM);
  bcn.instRefNum = instRefNum;
```

```
    return bcn;
  }

  public static JLStackIR gotoinst(int targetRefNum) {
    JLStackIR bcn = new JLStackIR(GOTO);
    bcn.targetNum = targetRefNum;
    return bcn;
  }

  public static JLStackIR ifcmp(int opcode, JLiteType type, int targetRefNum) {
    JLStackIR bcn = new JLStackIR(opcode);
    bcn.type = type;
    bcn.targetNum = targetRefNum;
    return bcn;
  }

  public static JLStackIR returnInst(JLiteType type) {
    JLStackIR bcn = new JLStackIR(RETURN);
    bcn.type = type;
    return bcn;
  }

  public static JLStackIR initObject(ResolvedObj rObj) {
    JLStackIR bcn = new JLStackIR(INIT_OBJ);
    bcn.resolvedObj = rObj;
    return bcn;
  }

  public static JLStackIR createNew(ResolvedObj rObj) {
    JLStackIR bcn = new JLStackIR(NEW);
    bcn.resolvedObj = rObj;
    return bcn;
  }

}
```

B.2.28 ReflectHelper

```
import org.apache.bcel.classfile.*;
import org.apache.bcel.generic.*;
import org.apache.bcel.Repository;
import java.lang.reflect.*;
import java.util.*;

/**
```

```
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class ReflectHelper {

  static HashMap classCache = null;

  static {
    classCache = new HashMap();
    classCache.put("int", int.class);
    classCache.put("long", long.class);
    classCache.put("double", double.class);
    classCache.put("boolean", boolean.class);
    classCache.put("float", float.class);
    classCache.put("char", char.class);
    classCache.put("void", void.class);
    classCache.put("String", String.class);
  }

  private static JavaClass lookupClsInRepository(String qualifiedName) {
    JavaClass javaClass = null;
    try {
      javaClass = Repository.lookupClass(qualifiedName);
      classCache.put(qualifiedName, javaClass);

    } catch (Exception e) {
      ;
    }
    return javaClass;
  }

  private static Object lookupClass(ResolvedObj rObj, String qualifiedName) {
    Object clazz = null;
    try {
      clazz = classCache.get(qualifiedName);
      if (clazz == null)
        try {
          clazz = Class.forName(qualifiedName);
        } catch (Exception e) {
          ;
        }
```

```
      if (clazz == null)
        clazz = lookupClsInRepository(qualifiedName);
      if (clazz != null) {
        rObj.type = JLConstants.CLASS;
      }
      if (clazz instanceof JavaClass) {
        rObj.name = ((JavaClass)clazz).getClassName();
      } else if (clazz instanceof Class) {
        rObj.name = ((Class)clazz).getName();
      }
      rObj.returnType = rObj.name;
    }
    catch (Exception e) {
      ;
    }
    return clazz;
  }

  private static Package lookupPackage(String pkgName) {
    Package pckage = Package.getPackage(pkgName);
    return pckage;
  }

  private static void lookupField(ResolvedObj rObj, String clsName, String fld) {
    Object clazz = null;
    clazz = lookupClass(rObj, clsName);
    if (clazz instanceof JavaClass)
      lookupFieldinJavaClass(rObj, (JavaClass)clazz, fld);
    else if (clazz instanceof Class)
      lookupFieldinClass(rObj, (Class)clazz, fld);
  }

  private static void lookupFieldinJavaClass(ResolvedObj rObj, JavaClass clazz,
String fld) {
    org.apache.bcel.classfile.Field fields[] = clazz.getFields();
    for (int i=0; i<fields.length; i++)
      if (fields[i].getName().equals(fld)) {
        rObj.type = JLConstants.FIELD;
        rObj.name = fld;
        rObj.returnType = fields[i].getType().toString();
      }
  }

  private static void lookupFieldinClass(ResolvedObj rObj, Class clazz, String fld)
{
    java.lang.reflect.Field field = null;
```

```
  java.lang.reflect.Field[] fields = clazz.getDeclaredFields();
  for (int i = 0; i < fields.length; i++)
    if (fields[i].getName().equals(fld)) {
      field = fields[i];
      rObj.type = JLConstants.FIELD;
      rObj.name = field.getName();
      rObj.returnType = Type.getType(field.getType()).toString();
    }
}

private static void lookupMethod(ResolvedObj rObj, String clsName, String fld,
                                 JLiteExprList exprList) throws Exception {
  Object clazz = null;
  clazz = lookupClass(rObj, clsName);
  if (clazz instanceof Class)
    lookupMethod(rObj, (Class) clazz, fld, exprList);
  else if (clazz instanceof JavaClass)
    lookupMethod(rObj, (JavaClass)clazz, fld, exprList);

}

private static void getMethods(List mthList, Class c, String name) {
  java.lang.reflect.Method[] allMths = c.getMethods();
  for (int i=0; i<allMths.length; i++) {
    if (allMths[i].getName().equals(name))
      mthList.add(allMths[i]);
  }
}

private static void filterMethod(List matched, int index, Class typespec) {
  for (int i=0; i<matched.size(); i++) {
    java.lang.reflect.Method mth = (java.lang.reflect.Method) matched.get(i);
    Class[] mthTypeSpec = mth.getParameterTypes();
    if (index < mthTypeSpec.length) {
      if (!mthTypeSpec[index].isAssignableFrom(typespec)) {
        matched.remove(mth);
        --i;
      }
    } else {
      matched.remove(mth);
      --i;
    }
  }
}

private static void filterMethod(List mthList, Class[] typeArry) {
```

```
    for (int i=0; i<typeArry.length; i++)
      filterMethod(mthList, i, typeArry[i]);
  }

  private static void lookupMethod(ResolvedObj rObj,Class clazz, String fld,
                                   JLiteExprList exprList) throws Exception {
    java.lang.reflect.Method mth = null;
    List mthList = null;
    Class[] typeArry = (exprList != null) ? getTypeList(exprList.lexpr) : null;
    try {
      mth = clazz.getMethod(fld, typeArry);
    } catch (Exception e) {
      ; // we are not done searching yet
    }
    if (mth == null) { // oh my god, no exact match, here we go for the lousy
routine
      mthList = new LinkedList();
      getMethods(mthList, clazz, fld);
      filterMethod(mthList, typeArry);
      if (mthList.size() > 1)
        throw new Exception("Ambiguous method call " + fld);
      else if (mthList.size() == 0) {
        throw new Exception("Method " + fld + " not found in class " +
clazz.getName());
      }
      else {
        mth = (java.lang.reflect.Method) mthList.get(0);
      }
    }
    if (mth != null) {
      rObj.type = JLConstants.METHOD;
      rObj.name = mth.getName();
      rObj.returnType = mth.getReturnType().getName();
      rObj.argType = toStringArray(Type.getTypes(mth.getParameterTypes()));
      rObj.isInterface = mth.getDeclaringClass().isInterface();
      rObj.isStatic = Modifier.isStatic(mth.getModifiers());
    }
  }

  private static void lookupMethod(ResolvedObj rObj, JavaClass clazz,
                                   String fld,
                                   JLiteExprList exprList) throws Exception {
    org.apache.bcel.classfile.Method[] methods = clazz.getMethods();
    Class[] typeArry = (exprList != null) ? getTypeList(exprList.lexpr) : null;
    for (int i = 0; i < methods.length; i++) {
      if (methods[i].getName().equals(fld)) {
```

```
        if (checkArgMatch(typeArry, methods[i].getArgumentTypes())) {
          rObj.name = methods[i].getName();
          rObj.type = JLConstants.METHOD;
          rObj.returnType = methods[i].getReturnType().toString();
          rObj.argType = toStringArray(methods[i].getArgumentTypes());
          rObj.isInterface = clazz.isInterface();
          break;
        }
      }
    }
  }

  private static boolean checkArgMatch(Class[] clsArry, Type[] mthArgTypes) {
    boolean matched = false;
    int clsArrySize = (clsArry != null) ? clsArry.length : 0;
    int mthArgSize = (mthArgTypes != null) ? mthArgTypes.length : 0;
    if ((clsArrySize == 0) && (mthArgSize == 0)) {
      matched =  true;
      return matched;
    }
    if (clsArrySize == mthArgSize) {
      matched = true;
      for (int i = 0; i < clsArry.length; i++) {
        Type bcelType = getBCELType(clsArry[i].getName());
        if (bcelType.toString().equals(mthArgTypes[i].toString()) != true) {
          matched = false;
          break;
        }
      }
    }
    return matched;
  }

  private static void lookupCtrInJavaClass(ResolvedObj rObj, JavaClass clazz,
JLiteExprList exprList) throws Exception {
    Class[] typeArry = (exprList != null) ? getTypeList(exprList.lexpr) : null;
    rObj.type = JLConstants.CONSTRUCTOR;
    rObj.name = clazz.getClassName();
    rObj.returnType = clazz.getClassName();
  }

  private static void lookupCtrInClass(ResolvedObj rObj, Class clazz,
JLiteExprList exprList) throws Exception {
    Class[] typeArry = (exprList != null) ? getTypeList(exprList.lexpr) : null;
    Constructor ctr = ((Class)clazz).getConstructor(typeArry);
    rObj.type = JLConstants.CONSTRUCTOR;
```

```
    rObj.name = ctr.getName();
    rObj.returnType = ((Class)clazz).getName();
    rObj.argType = toStringArray(ctr.getParameterTypes());
  }

  private static void lookupConstructor(ResolvedObj rObj, String clsName,
JLiteExprList exprList) throws Exception {
    Object clazz = null;
    ResolvedObj l_rObj = new ResolvedObj();
    Class[] typeArry = (exprList != null) ? getTypeList(exprList.lexpr) : null;
    clazz = lookupClass(l_rObj, clsName);
    if (clazz instanceof Class)
      lookupCtrInClass(rObj, (Class)clazz, exprList);
    else if (clazz instanceof JavaClass)
      lookupCtrInJavaClass(rObj, (JavaClass)clazz, exprList);
  }

  /** Return type could be Class, Package, Field, Method **/
  public static ResolvedObj resolveType(String nameToResolve, int hint,
                                        ResolvedObj ptrToCurrent,
                                        JLiteExprList exprList) throws
      Exception {
    String resolvedQualifier = null;
    if (ptrToCurrent != null)
       resolvedQualifier = ptrToCurrent.returnType;
    ResolvedObj rObj = new ResolvedObj();
    boolean searchClassPath = false;
    String errorMsg = null;

    if (hint == JLConstants.METHOD) {
      lookupMethod(rObj, resolvedQualifier, nameToResolve, exprList);
      if (rObj.type != JLConstants.METHOD) {
        throw new Exception("Method " + nameToResolve + " not found ");
      }
    }
    else if ( (hint == JLConstants.CLASS) || (hint ==
JLConstants.CONSTRUCTOR) ||
           (ptrToCurrent == null) || (ptrToCurrent.type ==
JLConstants.PACKAGE)) {
      if ((ptrToCurrent == null) || (ptrToCurrent.type != JLConstants.CLASS)) {
        if (resolvedQualifier != null)
          resolvedQualifier = resolvedQualifier + ".";
        else {
          searchClassPath = true;
          resolvedQualifier = "";
        }
```

```
        lookupClass(rObj, resolvedQualifier + nameToResolve);
        if (rObj.type == JLConstants.UNKNOWN)
          lookupClass(rObj, resolvedQualifier + nameToResolve);
        if ( (rObj.type == JLConstants.UNKNOWN) && searchClassPath)
          lookupClass(rObj, "java.lang." + nameToResolve);
        if (rObj.type == JLConstants.CLASS)
          resolvedQualifier = rObj.returnType;
      }
      if (hint == JLConstants.CONSTRUCTOR) {
        lookupConstructor(rObj, resolvedQualifier, exprList);
        if (rObj.type != JLConstants.CONSTRUCTOR)
          throw new Exception("Constructor not found in class " +
                              resolvedQualifier + "." +
                              nameToResolve);
      }
      Package pkg = null;
      if (rObj.type == JLConstants.UNKNOWN) {
        pkg = lookupPackage(resolvedQualifier + nameToResolve);
        rObj.type = JLConstants.PACKAGE;
        rObj.name = nameToResolve;
        rObj.returnType = resolvedQualifier + nameToResolve;
      }
    } else  {
      lookupField(rObj, resolvedQualifier, nameToResolve);
      if (rObj.type == JLConstants.UNKNOWN)
        throw new Exception("Field not found " + resolvedQualifier + "." +
nameToResolve);
    }
    return rObj;
  }

  public static Class[] getTypeList(List list) {
    LinkedList clsList = null;
    Object retObj = null;
    Object clazz = null;
    ResolvedObj rObj = new ResolvedObj();
    try {
      if (list != null) {
        clsList = new LinkedList();
        for (Iterator iter = list.iterator(); iter.hasNext(); ) {
          JLiteExpr l_expr = (JLiteExpr) iter.next();
          if (l_expr.type != null) {
            clazz = lookupClass(rObj, l_expr.type.getClassName());
            if (clazz instanceof Class)
                clsList.add(clazz);
            else
```

```
            System.out.println("Unknown class " + l_expr.type.getClassName());
          }
        }
      }
    } catch (Exception e) {
      ;
    }
    return toArray(clsList);
  }

  private static Class[] toArray(LinkedList list) {
    Class[] arry = new Class[list.size()];
    int i = 0;
    for (Iterator iter=list.iterator(); iter.hasNext();)
      arry[i++] = (Class) iter.next();
    return arry;
  }

  private static String[] toStringArray(Type[] bcelTypes) {
    String[] strArry = new String[bcelTypes.length];
    for (int i=0; i<bcelTypes.length; i++) {
      strArry[i] = bcelTypes[i].toString();
    }
    return strArry;
  }

  private static String[] toStringArray(Class[] javaClsTypes) {
    String[] strArry = new String[javaClsTypes.length];
    for (int i=0; i<javaClsTypes.length; i++) {
      strArry[i] = javaClsTypes[i].getName();
    }
    return strArry;
  }

  public static Type getBCELType(String clsName) {
    ResolvedObj rObj = new ResolvedObj();
    Object clazz = lookupClass(rObj, clsName);
    Type bcelType = null;
    if ((clazz instanceof Class) && (!(clazz instanceof String)))
      bcelType = Type.getType((Class)clazz);
    else
      bcelType = new ObjectType(clsName);
    return bcelType;
  }

  public static boolean isAssignable(JLiteType left, JLiteType right) {
```

```
    if ((left != null) && (right != null)) {
      if (JLiteType.INT.equals(left) && JLiteType.DOUBLE.equals(right))
        return false;
      else if (JLiteType.INT.equals(left) && JLiteType.DOUBLE.equals(right))
        return true;
      else
        return true;
    }
    return false;
  }

  public static JLiteType getDominable(JLiteType type1, JLiteType type2) {
    if (JLiteType.INT.equals(type1) && JLiteType.DOUBLE.equals(type2))
      return JLiteType.DOUBLE;
    else if (JLiteType.INT.equals(type2) && JLiteType.DOUBLE.equals(type1))
      return JLiteType.DOUBLE;
    else
      return type1;
  }

}


class ResolvedObj {
  int type;
  String name;
  String[] argType;
  String returnType;
  boolean isInterface;
  boolean isStatic;
}
```

B.2.29 SourceInfo

```
/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class SourceInfo {

  int linenumber, colnumber;
```

```
  String srctext;

  public SourceInfo() {
  }

}
```

B.2.30 ByteCodeGen

```
import org.apache.bcel.classfile.JavaClass;
import java.io.*;

/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public interface ByteCodeGen {

  public static final int PLUS = 1;
  public static final int MINUS = 2;
  public static final int MUL = 3;
  public static final int DIV = 4;

  public void initClass(String clsname);
  public void dump(OutputStream out) throws Exception;
  public void dumpByteCode(String filename) throws Exception;
  public void createMethod(String name, JLiteType returnType, JLiteExprList
argList);
  public void finalizeMethod();

  public void addInstruction(JLStackIR bcnotation);
  public JavaClass getJavaClass();
  public void loadMethod(String name);
  public boolean isVirtualFunction();

}
```

B.2.31 ByteCodeGenImpl

```
import org.apache.bcel.*;
import org.apache.bcel.classfile.*;
import org.apache.bcel.generic.*;
import java.io.*;
import java.util.*;

/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

public class ByteCodeGenImpl implements ByteCodeGen {

  InstructionList il = new InstructionList();

  ConstantPoolGen _cp;
  ClassGen _cg;
  InstructionFactory _factory;

  String clsname;
  MethodGen currentMethod;
  boolean virutalFunction = false;

  List listofOps = null;
  HashMap refMap = null;
  HashMap targetMap = null;

  public ByteCodeGenImpl(String name) {
    initClass(name);
  }

  public void initClass(String name) {
    this.clsname = name;
    _cg = new ClassGen(name, "java.lang.Object", "<jLiteCompiler>",
                  Constants.ACC_PUBLIC | Constants.ACC_SUPER, new String[] {
});
    _cp = _cg.getConstantPool();
    _factory = new InstructionFactory(_cg);
```

```
    addDefaultConstructor();
  }

  public void dump(java.io.OutputStream out) throws Exception {
    _cg.getJavaClass().dump(out);
  }

  /** Function looks busy needs organization **/

  public void createMethod(String name, JLiteType returnType, JLiteExprList
argList) {

    listofOps = new LinkedList();
    refMap = new HashMap();
    targetMap = new HashMap();

    Type l_returnType = (returnType == JLiteType.VOID) ? (Type) Type.VOID :
       ReflectHelper.getBCELType(returnType.getClassName());
    Type[] l_argListType = new Type[argList.lexpr.size()];
    String[] l_argListName = new String[argList.lexpr.size()];
    JLiteId l_id = null;
    for (int i=0; i<argList.lexpr.size(); i++) {
      l_id = (JLiteId)argList.lexpr.get(i);

      if (l_id.type instanceof JLiteArrayType)
        //l_argListType[i] =  new
ArrayType(ReflectHelper.getBCELType(l_id.type.getClassName()),
((JLiteArrayType)l_id.type).arryDim);
        l_argListType[i] =  new ArrayType(Type.STRING,
((JLiteArrayType)l_id.type).arryDim);
      else
        l_argListType[i] = ReflectHelper.getBCELType(l_id.type.getClassName());
      l_argListName[i] = l_id.token;
    }
    int access = Constants.ACC_PUBLIC;

    if ("main".equalsIgnoreCase(name)) {
      this.virutalFunction = false;
      access = access | Constants.ACC_STATIC;
    } else {
      this.virutalFunction = true;  //Since we don't support static functions
    }

    currentMethod = new MethodGen(access, l_returnType, l_argListType,
                                  l_argListName, name, clsname, il, _cp);
  }
```

```
/** This method will set up global variables */
public void loadMethod(String name) {
  if ("main".equalsIgnoreCase(name)) {
    this.virutalFunction = false;
  }
  else {
    this.virutalFunction = true; //Since we don't support static functions
  }
}

public boolean isVirtualFunction() {
  return this.virutalFunction;
}

public void finalizeMethod() {
  try {
    toByteCode();
    patchTargets();
    currentMethod.setMaxStack();
    currentMethod.setMaxLocals();
    _cg.addMethod(currentMethod.getMethod());
    il.dispose();
    listofOps = null;
    refMap = null;
    targetMap = null;
  } catch (Exception e) {
    ;
  }
}

public void dumpByteCode(String filename) throws Exception {
  dump(new FileOutputStream(filename));
}

public JavaClass getJavaClass() {
  return _cg.getJavaClass();
}

public void toByteCode() {
  Object inst = null;
  List refList = null;
  for (Iterator iter=listofOps.iterator(); iter.hasNext();) {
    JLStackIR bcn = (JLStackIR)iter.next();
    if (bcn.opcode == JLStackIR.SET_REF_NUM) {
      if (refList == null)
```

```
        refList = new LinkedList();
       refList.add(new Integer(bcn.instRefNum));
      } else {
       addInstruction(refList, il, bcn);
       refList = null;
      }
     }
    }

    public void addInstruction(JLStackIR bcnotation) {
     listofOps.add(bcnotation);
    }

    protected void addInstruction(List l_refList, InstructionList l_il, JLStackIR
bcnotation) {
     JLiteExpr expr = bcnotation.expr;
     InstructionHandle instHandle = null;
     BranchInstruction branchInstr = null;
     switch (bcnotation.opcode) {
      case JLStackIR.STORE:
       instHandle = store(l_il, expr);
       break;
      case JLStackIR.LOAD:
       instHandle = load(l_il, expr);
       break;
      case JLStackIR.IALOAD:
       instHandle = l_il.append(InstructionConstants.IALOAD);
       break;
      case JLStackIR.ALOAD:
         instHandle = l_il.append(new ALOAD(0));
       break;
      case JLStackIR.IASTORE:
       instHandle = l_il.append(InstructionConstants.IASTORE);
       break;
      case JLStackIR.UNARY:
       if (JLiteType.INT.equals(bcnotation.expr.type))
          instHandle = l_il.append(InstructionConstants.INEG);
       else
         instHandle = l_il.append(InstructionConstants.DNEG);
       break;
      case JLStackIR.ADD:
       if (bcnotation.type == JLiteType.INT)
         instHandle = l_il.append(InstructionConstants.IADD);
       else
         instHandle = l_il.append(InstructionConstants.DADD);
       break;
```

```
      case JLStackIR.MINUS:
        if (bcnotation.type == JLiteType.INT)
          instHandle = l_il.append(InstructionConstants.ISUB);
        else
          instHandle = l_il.append(InstructionConstants.DSUB);
        break;
      case JLStackIR.MUL:
        if (bcnotation.type == JLiteType.INT)
          instHandle = l_il.append(InstructionConstants.IMUL);
        else
          instHandle = l_il.append(InstructionConstants.DMUL);
        break;
      case JLStackIR.DIV:
        if (bcnotation.type == JLiteType.INT)
          instHandle = l_il.append(InstructionConstants.IDIV);
        else
          instHandle = l_il.append(InstructionConstants.DDIV);
        break;
      case JLStackIR.GETSTATIC:
        instHandle = createFieldAccess(l_il, bcnotation.resolvedObj,
bcnotation.clsname);
        break;
      case JLStackIR.INVOKE_METHOD:
        instHandle = createMethodAccess(l_il, bcnotation.resolvedObj,
bcnotation.clsname);
        break;
      case JLStackIR.INIT_ARRAY:
        instHandle = initArray(l_il, bcnotation.size, bcnotation.type);
        break;
      case JLStackIR.ARRAY_INDEX:
        instHandle = setArrayIndex(l_il, bcnotation.arrIndex);
        break;
      case JLStackIR.IFCMPEQ:
      case JLStackIR.IFCMPNE:
      case JLStackIR.IFCMPGT:
      case JLStackIR.IFCMPGEQ:
      case JLStackIR.IFCMPLT:
      case JLStackIR.IFCMPLEQ:
        if (bcnotation.type == JLiteType.DOUBLE)  {
          instHandle = l_il.append(InstructionConstants.DCMPG);
        }
        branchInstr = createIfInstruction(l_il, bcnotation.opcode, bcnotation.type);
        if (instHandle == null)
          instHandle = l_il.append(branchInstr);
        else
          l_il.append(branchInstr);
```

```
      break;
    case JLStackIR.GOTO:
      branchInstr = _factory.createBranchInstruction(Constants.GOTO, null);
      instHandle = l_il.append(branchInstr);
      break;
    case JLStackIR.RETURN:
      instHandle = returnInst(l_il, bcnotation.type);
      break;
    case JLStackIR.INC:
      instHandle = increment(l_il, expr);
      break;
    case JLStackIR.DEC:
      instHandle = l_il.append(new IINC(((JLiteRefPtr)expr).idref.varNum, -1));
      break;
    case JLStackIR.NEW:
      instHandle = createNew(l_il, bcnotation.resolvedObj);
      break;
    case JLStackIR.INIT_OBJ:
      instHandle = initObject(l_il, bcnotation.resolvedObj);
      break;
    case JLStackIR.I2D:
      instHandle = l_il.append(InstructionConstants.I2D);
      break;
  }
  if (l_refList != null) {
    for (Iterator iter=l_refList.iterator(); iter.hasNext();)
      refMap.put((Integer)iter.next(), instHandle);
  }
  if (bcnotation.targetNum != -1) {
    addToTarget(bcnotation.targetNum, branchInstr);
  }
}

private void patchTargets() {
  for (Iterator iter=targetMap.keySet().iterator(); iter.hasNext();) {
    Integer targetNum = (Integer) iter.next();
    List targetList = (List)targetMap.get(targetNum);
    InstructionHandle instHandle = (InstructionHandle) refMap.get(targetNum);
    for (Iterator listIter=targetList.iterator(); listIter.hasNext(); ) {
      BranchInstruction branchInst = (BranchInstruction) listIter.next();
      if (instHandle != null)
        branchInst.setTarget(instHandle);
      else {
        try {
          il.delete(branchInst);
        } catch (Exception e) {
```

```
          ;
        }
       }
      }
     }
    }

    /** Helper functions **/

    private void addToTarget(int targetNum, BranchInstruction branchInst) {
      List targetList = (List) targetMap.get(new Integer(targetNum));
      if (targetList == null) {
        targetList = new LinkedList();
        targetMap.put(new Integer(targetNum), targetList);
      }
      targetList.add(branchInst);
    }

    private JLiteType getType(JLiteExpr l_expr) {
      if (l_expr instanceof JLiteId)
        return ((JLiteId)l_expr).type;
      else
        return null;
    }

    private int getVarNum(JLiteExpr l_expr) {
      if (l_expr instanceof JLiteRefPtr)
        return ((JLiteRefPtr)l_expr).idref.varNum;
      else
        return ((JLiteId)l_expr).varNum;
    }

    private CompoundInstruction getPushInst(JLiteExpr l_expr) {
      if (l_expr.type == JLiteType.INT)
        return new PUSH(_cp, Integer.parseInt(l_expr.token));
      else if (l_expr.type == JLiteType.DOUBLE)
        return new PUSH(_cp, Double.parseDouble(l_expr.token));
      else if (l_expr.type == JLiteType.FLOAT)
        return new PUSH(_cp, Float.parseFloat(l_expr.token));
      else
        return new PUSH(_cp, l_expr.token);
    }

    private InstructionHandle initArray(InstructionList l_il, int size, JLiteType type)
    {
      InstructionHandle instHandle = l_il.append(new PUSH(_cp, size));
```

```
    l_il.append(_factory.createNewArray(Type.INT, (short) 1));
    return instHandle;
  }

  private InstructionHandle setArrayIndex(InstructionList l_il, int[] index) {
    InstructionHandle instHandle = null;
    instHandle = l_il.append(InstructionConstants.DUP);
    for (int i = 0; i < index.length; i++) {
      l_il.append(new PUSH(_cp, index[i]));
    }
    return instHandle;
  }

  private InstructionHandle createMethodAccess(InstructionList l_il, ResolvedObj
rObj,
                               String typename) {
    Type retType = ReflectHelper.getBCELType(rObj.returnType);

    String[] paramClsTypes = rObj.argType;
    Type[] paramTypes = (paramClsTypes != null) ? new
Type[paramClsTypes.length] : new Type[] {} ;
    for (int i=0; i<paramTypes.length; i++)
      paramTypes[i] = ReflectHelper.getBCELType(paramClsTypes[i]);
    boolean isInterface = rObj.isInterface;
    short invokeType = (rObj.isStatic) ? Constants.INVOKESTATIC : 0;
    if (invokeType == 0)
      invokeType = (isInterface) ? Constants.INVOKEINTERFACE :
Constants.INVOKEVIRTUAL;
    return l_il.append(_factory.createInvoke(typename, rObj.name, retType,
                               paramTypes,
                               invokeType));
  }

  private InstructionHandle createFieldAccess(InstructionList l_il, ResolvedObj
rObj, String clsname) {
    return l_il.append(_factory.createFieldAccess(clsname, rObj.name,
                           ReflectHelper.getBCELType(rObj.returnType),
                           Constants.GETSTATIC));
  }

  private InstructionHandle createNew(InstructionList l_il, ResolvedObj rObj) {
    InstructionHandle instHandle =
l_il.append(_factory.createNew(rObj.returnType));
    l_il.append(InstructionConstants.DUP);
    return instHandle;
  }
```

```
  private InstructionHandle initObject(InstructionList l_il, ResolvedObj rObj) {
    String[] paramClsTypes = rObj.argType;
    Type[] paramTypes = (paramClsTypes != null) ? new
Type[paramClsTypes.length] : new Type[] {};
    if (paramClsTypes != null)
      for (int i=0; i<paramClsTypes.length; i++)
        paramTypes[i] = ReflectHelper.getBCELType(paramClsTypes[i]);
    return l_il.append(_factory.createInvoke(rObj.returnType, "<init>",
Type.VOID,
                                    paramTypes, Constants.INVOKESPECIAL));
  }

  private InstructionHandle load(InstructionList l_il, JLiteExpr expr) {
    InstructionHandle instHandle = null;
    if (expr instanceof JLiteConst){
      if (expr == JLiteConst.True)
        instHandle = l_il.append(new PUSH(_cp, true));
      else if (expr == JLiteConst.False)
        instHandle = l_il.append(new PUSH(_cp, false));
      else
        instHandle = l_il.append(getPushInst(expr));
    }
    else if ( (getType(expr) instanceof JLiteArrayType)
            || (getType(expr) instanceof JLiteObjType))
      instHandle = l_il.append(new ALOAD(getVarNum(expr)));
    else {
      if (expr.type == JLiteType.INT)
        instHandle = l_il.append(new ILOAD(getVarNum(expr)));
      else if (expr.type == JLiteType.DOUBLE)
        instHandle = l_il.append(new DLOAD(getVarNum(expr)));
      else if (expr.type == JLiteType.FLOAT)
        instHandle = l_il.append(new FLOAD(getVarNum(expr)));
      else if (expr.type == JLiteType.BOOL)
        instHandle = l_il.append(new ILOAD(getVarNum(expr)));
    }
    return instHandle;
  }

  private InstructionHandle store(InstructionList l_il, JLiteExpr expr) {
    InstructionHandle instHandle = null;
    JLiteType l_type = null;
    if (getType(expr) instanceof JLiteArrayType)
      instHandle = l_il.append(new ASTORE(getVarNum(expr)));
    else if (getType(expr) instanceof JLiteObjType)
```

```
      instHandle = l_il.append(_factory.createStore(Type.OBJECT,
getVarNum(expr)));
    else {
      if ((expr.type == JLiteType.INT) || (expr.type == JLiteType.BOOL))
        instHandle = l_il.append(new ISTORE(getVarNum(expr)));
      else if (expr.type == JLiteType.DOUBLE)
        instHandle = l_il.append(new DSTORE(getVarNum(expr)));
      else if (expr.type == JLiteType.FLOAT)
        instHandle = l_il.append(new FSTORE(getVarNum(expr)));
    }
    return instHandle;
  }

  private InstructionHandle increment(InstructionList l_il, JLiteExpr expr) {
    InstructionHandle instHandle = null;
    if (expr instanceof JLiteRefPtr)
      if (((JLiteRefPtr)expr).type == JLiteType.INT)
        instHandle = l_il.append(new IINC(((JLiteRefPtr)expr).idref.varNum, 1));
      else if (((JLiteRefPtr)expr).type == JLiteType.DOUBLE) {
        instHandle = load(l_il, (JLiteRefPtr) expr);
        l_il.append(new PUSH(_cp, 1.0));
        l_il.append(InstructionConstants.DADD);
        store(l_il, expr);
      }
    return instHandle;
  }

  private BranchInstruction createIfInstruction(InstructionList l_il, int cmpType,
JLiteType l_type) {
    short bcCmpType = -1;
    switch (cmpType) {
      case JLStackIR.IFCMPEQ:
        if ((l_type == JLiteType.INT) || (l_type == JLiteType.BOOL))
          bcCmpType = Constants.IF_ICMPEQ;
        else
          bcCmpType = Constants.IFEQ;
        break;
      case JLStackIR.IFCMPNE:
        if ((l_type == JLiteType.INT) || (l_type == JLiteType.BOOL))
          bcCmpType = Constants.IF_ICMPNE;
        else
          bcCmpType = Constants.IFNE;
        break;
      case JLStackIR.IFCMPGT:
        if ((l_type == JLiteType.INT) || (l_type == JLiteType.BOOL))
          bcCmpType = Constants.IF_ICMPGT;
```

```
      else
        bcCmpType = Constants.IFGT;
      break;
    case JLStackIR.IFCMPGEQ:
      if ((l_type == JLiteType.INT) || (l_type == JLiteType.BOOL))
        bcCmpType = Constants.IF_ICMPGE;
      else
        bcCmpType = Constants.IFGE;
      break;
    case JLStackIR.IFCMPLT:
      if ((l_type == JLiteType.INT) || (l_type == JLiteType.BOOL))
        bcCmpType = Constants.IF_ICMPLT;
      else
        bcCmpType = Constants.IFLT;
      break;
    case JLStackIR.IFCMPLEQ:
      if ((l_type == JLiteType.INT) || (l_type == JLiteType.BOOL))
        bcCmpType = Constants.IF_ICMPLE;
      else
        bcCmpType = Constants.IFLE;
      break;
   }
   BranchInstruction bInst = _factory.createBranchInstruction(bcCmpType, null);
   return bInst;
}


private InstructionHandle returnInst(InstructionList l_il, JLiteType type) {
   InstructionHandle instHandle = null;
   if (type instanceof JLiteArrayType)
     instHandle = l_il.append(InstructionConstants.ARETURN);
   if (Void.class.getName().equals(type.getClassName()))
     instHandle = l_il.append(InstructionConstants.RETURN);
   else if (int.class.getName().equals(type.getClassName()))
     instHandle = l_il.append(InstructionConstants.IRETURN);
   else if (double.class.getName().equals(type.getClassName()))
     instHandle = l_il.append(InstructionConstants.DRETURN);
   else
     instHandle = l_il.append(InstructionConstants.ARETURN);
   return instHandle;
}


private void addDefaultConstructor() {
   InstructionList il = new InstructionList();
```

```
   MethodGen method = new MethodGen(Constants.ACC_PUBLIC, Type.VOID,
Type.NO_ARGS, new String[] {  }, "<init>", clsname, il, _cp);

   InstructionHandle ih_0 = il.append(_factory.createLoad(Type.OBJECT, 0));
   il.append(_factory.createInvoke("java.lang.Object", "<init>", Type.VOID,
Type.NO_ARGS, Constants.INVOKESPECIAL));
   InstructionHandle ih_4 = il.append(_factory.createReturn(Type.VOID));
   method.setMaxStack();
   method.setMaxLocals();
   _cg.addMethod(method.getMethod());
   il.dispose();
  }

}
```

B.2.32 JLiteCompiler

```
import java.io.*;
import java.util.HashMap;
import antlr.debug.misc.*;
import org.apache.bcel.util.*;
import org.apache.bcel.classfile.*;
import org.apache.bcel.Repository;

/**
 * <p>Title: jLite Compiler</p>
 * <p>Description: Lite Version of Java Compiler</p>
 * <p>Copyright: Copyright (c) 2007</p>
 * <p>Company: CVN</p>
 * @author Sudhakar Perumal
 * @version 1.0
 */

class JLiteCompiler {

   public static void main(String[] args) {
         try {

         HashMap params = parseParams(args);
         String outputDir = getPath((String) params.get("outdir"));
         String srcDir = getPath((String) params.get("srcdir"));
         String debug = (String) params.get("debug");
         String infile = (String) params.get("infile");

            FileInputStream filename = new FileInputStream(srcDir + infile);
```

```
            DataInputStream input = new DataInputStream(filename);
            JLiteLexer lexer = new JLiteLexer(new DataInputStream(input));
            JLiteParser parser = new JLiteParser(lexer);
          parser.setASTNodeClass("CommonInfoAST");
            parser.javasource();
            antlr.CommonAST ast = (antlr.CommonAST) parser.getAST();
          if ("true".equals(debug)) {
            ASTFrame frame = new ASTFrame("AST", ast);
            frame.setVisible(true);
          }
            JLiteWalker walker = new JLiteWalker();
            JLiteClass c = walker.classdef(ast);
          //c.traceon = true;
          c.precompile();
          if (c.errors == 0) {
            Repository.addClass(c.bytecodeGen.getJavaClass());
          } else {
            System.exit(-1);
          }
          c.typecheck();
          if (c.errors == 0) {
            c.compile();
            c.dumpByteCode(outputDir + c.getName() + ".class");
          } else {
            System.out.println();
            System.out.println(c.errors + " errors.");
          }
          } catch (Exception e) {
        System.out.println("Compiler Aborted : " + e.toString());
          }
}

private static HashMap parseParams(String[] args) {
  HashMap l_params = new HashMap();
  String arg = null;
  for (int i=0; i<args.length; i++) {
    arg = args[i];
    if ("-o".equalsIgnoreCase(arg))
      l_params.put("outdir", args[++i]);
    if ("-src".equalsIgnoreCase(arg))
      l_params.put("srcdir", args[++i]);
    else if ("-debug".equalsIgnoreCase(arg))
      l_params.put("debug", args[++i]);
    else
      l_params.put("infile", args[i]);
  }
```

```
    return l_params;
  }

  private static String getPath(String inPath) {
    return (inPath != null)  ? inPath + "//" : "";
  }

}
```