Verification Condition Generator

Write a Verification Condition Generator (VCG) for our simple imperative language, IMP, using the parser generator ANTLR. Use the Weakest (Liberal) Precondition Predicate Transformer semantics. This allows you to use backward substitution over if-statements. For while-loops, simply print intermediate verification conditions instead of combining them into a single verification condition using universal quantification. Use the ANTLR attribute grammar mechanism as far as possible for generating the verification conditions.

Reading Material

For information on ANTLR and the weakest precondition semantics see:

http://www.antlr.org/
http://en.wikipedia.org/wiki/Predicate_transformer_semantics

For using ANTLR, make sure you put antlr-4.8-complete.jar onto your CLASSPATH. For details, see the Quick Start and Samples instructions on the ANTLR web page or read the Makefile I provided.

Code Structure

For constructing the verification conditions, you need to build a parse tree data structure representing the verification conditions. Since expressions may be be substituted for variables, and since Boolean expressions are used in assertions, you also need to build parse trees for all expressions in the program. When traversing statements, you will then construct the verification conditions using the weakest precondition predicate transformer. This will require building the parse trees for statements as well.

For building parse trees, use the Composite Design Pattern, i.e., use a class hierarchy with an abstract superclass and subclasses corresponding to the different language constructs, e.g.,

```java
abstract class Exp { ... }
class Ident extends Exp { ... }
class IntLit extends Exp { ... }
class OpExp extends Exp { ... }
// etc.
```

For expressions, you will need methods for printing the expression and for performing a substitution. For adding statements, it would be best to group expressions and statements as two separate parts
of the class hierarchy as below or as two disjoint class hierarchies (without the common superclass Node), since you will need different methods:

```java
abstract class Node { ... }
abstract class Exp extends Node { ... }
abstract class Stmt extends Node { ... }
class Ident extends Exp { ... }
class IntLit extends Exp { ... }
class OpExp extends Exp { ... }
// etc.
class Assign extends Stmt { ... }
// etc.
```

**Grammar**

Use the following grammar for the language IMP (in ANTLR syntax) with start symbol `program` as a starting point. You may need to restructure the grammar based on how you propagate information using attributes.

```antlr
program : assertion statementlist assertion ;

statementlist : statement | statement ';' statementlist ;

statement : 'skip' | id ':=' arithexp | 'begin' statementlist 'end' | 'if' boolterm 'then' statement 'else' statement | assertion 'while' boolterm 'do' statement | 'assert' assertion ;

assertion : '{' boolean '}' ;

boolean : boolterm | boolterm '=>' boolterm | boolterm '<=>' boolterm ;

boolterm : boolterm2
```
| boolterm 'or' boolterm2
|

boolterm2
 : boolfactor
 | boolterm2 'and' boolfactor
 |

boolfactor
 : 'true'
 | 'false'
 | compexp
 | 'forall' id '.' boolexp
 | 'exists' id '.' boolexp
 | 'not' boolfactor
 | '(' boolexp ')' |
 |

compexp
 : arithexp '<' arithexp
 | arithexp '<=' arithexp
 | arithexp '=' arithexp
 | arithexp '!=' arithexp
 | arithexp '>=' arithexp
 | arithexp '>' arithexp
 |

arithexp
 : arithterm
 | arithexp '+' arithterm
 | arithexp '-' arithterm
 |

arithterm
 : arithfactor
 | arithterm '*' arithfactor
 | arithterm '/' arithfactor
 |

arithfactor
 : id
 | integer
 | '-' arithfactor
 | '(' arithexp ')'
 | id '(' arithexplist ')' |
 |

arithexplist

3
Examples

The input to the VCG is a single Hoare triple. The output is a list of verification conditions, without formatting, one VC per line. E.g., for the input

\[
\{ \text{true} \} \ x := (2 \ast 3) \ast (3 + 4) \ \{ x = 42 \}
\]

The VCG should print

\[
\text{true} \Rightarrow 2\ast3\ast(3+4)=42
\]

Surround only the operators \(\Rightarrow, \Leftrightarrow, \text{and}, \text{and} \ or \) with one space on each side, print a single space after \text{not}, \text{forall}, and \text{exists}, and print parentheses only if the parent operator has higher precedence (or for right-associative subtractions).

For loops, first print the exit condition, then the condition for the loop body, followed by the condition for the code before the loop. E.g., for the input

\[
\{ x \geq 0 \land y > 0 \} \\
q := 0; \\
r := x; \\
\{ x = q\ast y + r \land 0 \leq r \land y > 0 \} \\
\text{while } r-y \geq 0 \text{ do begin} \\
\quad q := q + 1; \\
\quad r := r - y \\
\text{end} \\
\{ x = q\ast y + r \land 0 \leq r \land r < y \}
\]
the output should be:

\[
\begin{align*}
x &= q \cdot y + r \quad 0 \leq r \quad y > 0 \quad \text{and not } r-y \geq 0 \quad \Rightarrow \quad x &= q \cdot y + r \quad 0 \leq r \quad r < y \\
x &= q \cdot y + r \quad 0 \leq r \quad y > 0 \quad r-y \geq 0 \quad \Rightarrow \quad x &= (q+1) \cdot y + r-y \\
x &\geq 0 \quad y > 0 \quad \Rightarrow \quad x &= 0 \cdot y + x \quad 0 \leq x \quad y > 0
\end{align*}
\]

**Administrative Stuff**

The skeleton code and a reference implementation are available in the `~cs7101_bau/pub/VCG` directory on the classes server and as a ZIP file on the course web page. After copying the files into your `~/prog1` directory, the code can be compiled and run with the following commands:

```bash
cd ~/prog1
make
java VCG test.vcg
```

(assuming that the CLASSPATH is set appropriately). You can run the reference implementation with the commands

```bash
cd ~/prog1
java -jar VCG.jar test.vcg
```

For submitting your project, put your files into directory `~/prog1` in your `cs7101xx` account on `classes.csc.lsu.edu`, make sure that you have a `Makefile` so that your code can be compiled, and provide a `README.txt` or `README.md` file with anything that you want me or the grader to know about your project.

For submitting the code, simply use

```bash
cd ~/prog1
make clean
p_copy 1
```

Do not put the files into a ZIP file.