Program 1: Verification Condition Generator

CSC 7101, Spring 2018

Due: 3 March 2018

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/
https://en.wikipedia.org/wiki/Predicate_transformer_semantics

For using ANTLR, make sure you put antlr-4.6-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 may or may not 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.,

abstract class Node { ...
  class Ident extends Node { ...
  class IntLit extends Node { ...
  class BinOp extends Node { ...
  // etc.

For expressions, you will need methods for printing the expression and for performing a substitution. If you decide to represent statements using parse tree nodes as well, it would be best to group expressions and statements as two separate parts of the class hierarchy, since you will likely need different methods, e.g.,

abstract class Node { ...
abstract class Exp extends Node { ...
abstract class Stm extends Node { ...
class Ident extends Exp { ...
class IntLit extends Exp { ...
class BinOp extends Exp { ...
// etc.
class Assign extends Stm { ...
// 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.

```
program : assertion statementlist assertion ;

statementlist : statement |
               statementlist ';' statement ;

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

assertion : '{' quantexp '}' ;

quantexp : boolexp |
          'forall' id '.' boolexp |
          'exists' id '.' boolexp ;

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

boolterm : boolterm2 |
         boolterm 'or' boolterm2 ;

boolterm2 : boolfactor |
           boolterm2 'and' boolfactor ;

boolfactor : 'true' |
            'false' |
            compexp |
            'not' quantexp |
            '{' quantexp '}' ;
```
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
     | '-' arithexp
     | '(' arithexp ')' 
     | id '(' arithexplist ')'
     ;

arithexplist
  : arithexp
     | arithexplist ',' arithexp
     ;

id
  : IDENT
     ;

integer
  : INT
     ;

IDENT
  : [A-Za-z][A-Za-z0-9_]*
     ;

INT
  : [0][1-9][0-9]*
     ;

WS
  : [ \r\n\t] -> skip
     ;
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

```
{ true } x := (2 * 3) * (3 + 4) { x = 42 }
```

The VCG should print

```
true => 2*3*(3+4)=42
```

Surround only the operators \( \Rightarrow, \leq \), and \( \lor \) with spaces, print a space after \( \neg \), and print parentheses only if the parent operator has higher precedence.

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 \text{ and } y > 0 }
q := 0;
r := x;
{ x = q*y + r \text{ and } 0 \leq r \text{ and } y > 0 }
while \ r-y \geq 0 \ do \ begin
\quad q := q + 1;
\quad r := r - y
end
{ x = q*y + r \text{ and } 0 \leq r \text{ and } r < y }
```

the output should be:

```
x=q*y+r \text{ and } 0\leq r \text{ and } y>0 \text{ and not } r-y=0 \Rightarrow x=q*y+r \text{ and } 0\leq r \text{ and } r<y
x=q*y+r \text{ and } 0\leq r \text{ and } y>0 \text{ and not } r-y=0 \Rightarrow x=(q+1)*y+r-y \text{ and } y>0 \text{ and } r-y>0
x=0 \text{ and } y>0 \Rightarrow x=0*y+x \text{ and } 0\leq x \text{ and } y>0
```

Administrative Stuff

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 run using the commands

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

and provide a file README.txt with anything that you want me or the grader to know about your project. For submitting the code, simply use

```
cd ~/prog1
make clean
~cs7101_bau/bin/p_copy 1
```

Do not put the files into a ZIP file.