Compiler Construction

To hand in:

- All relevant source files (grammars and java)
- Test examples.
- A (ca. 1-2 page) report in PDF format that documents what you did for each task: briefly describe what changes you made to the given code, design choices, and an example output of a test you performed for the different debug commands.

The Starting Point The task is based on the imperative language. Please find enclosed impl-solution.zip which is the model solution for the task . Recall in particular the following files:

- impl.g4 the grammar of the language in ANTLR
- main.java that drives the compiler and contains the translation of the parse tree into abstract syntax.
- AST.java that contains the abstract syntax including the methods
  - typecheck that can be called to check that all parts are type correct
  - eval that evaluates the program – i.e., this is *the interpreter* that executes a given program. •

  Makefile to guide the compilation of the ANTLR and java files.
- impl_input.txt and impl_additional.txt are two example programs.

The Task: Turn it into a debugger The task of this is to add debugger functionality to this interpreter. For this task you should add special Debugger commands to the impl language that allow programmers to easily check and test several things in their code. These debugging commands are ***Breakpoint***, ***Assert***, ***Monitor***, and ***Trace***; they are described in the following paragraphs.

Please note that these are in increasing order of difficulty.. Also for each of these commands there are some extra challenges that make them a bit harder to implement, and it is of course better to implement them in a basic form than not at all.

I have chosen the following concrete syntax for the debug commands:

DEBUG: <Kind of debug command> <Arguments>;

but you are of course welcome to use a different concrete syntax.

Breakpoint A breakpoint means declaring a point in the code where the execution should be paused when reaching it, possibly giving out some information, and continuing execution only after the user has pressed a key.

Example:

```
n=5;
result=1;

while(n!=0){
   result=result*n;
   output result;
   DEBUG: Breakpoint b;
   n=n-1;
}
```

Each breakpoint shall include an identifier (in the example “b”) that is being displayed when the breakpoint is reached. Thus the running the above program should produce the following output. Each point where the execution pauses and waits for a key press I have denoted by the symbol ¶ (which should of course not be part of the input/output of the debugger):

```
5.0
Breakpoint b
¶
20.0
Breakpoint b
¶
60.0
Breakpoint b
¶
120.0
Breakpoint b
¶
120.0
Breakpoint b
```

Assert An assert command stipulates a condition that should hold when the program reaches this point. The debugger should check this condition and silently continue if it is satisfied, but abort the execution with a suitable error message if the condition is violated.

Example:

```
n=5;
result=1;

while(n!=0){
   prev_result=result;
   result=result*n;
   output result;
   DEBUG: Assert prev_result<result;
   n=n-1;
}
```

This could produce the following output:

```
5.0
20.0
60.0
120.0
120.0
Debug: Assertion prev_result<result violated!
```

because in the last round of the loop, we have n==1 and thus result does not increase. Note that it can

be tricky to display the condition itself; for a simpler solution you may omit this and rather just output

Debug: Assertion violated!

For a more advanced solution, you may display the values of the involved variables,

e.g. Debug: Assertion prev_result==120<result==120 violated!

Monitor The monitor command denotes expressions that in the following should be displayed whenever we reach a breakpoint or failed assertion. In this way the programmer does not need to insert output statements into the code for debugging purposes.

Example:

```
n=5;
result=1;

DEBUG: Monitor result;

while(n!=0){
   prev_result=result;
   result=result*n;
   DEBUG: Assert prev_result<result;
   n=n-1;
}
```

This would produce the output:

```
Monitor result=120.0
Debug: Assertion prev_result<result violated!
```

A slightly simpler version does not allow for arbitrary expressions, but just for variables to be moni tored. Also for simplicity one can allow for only one variable or expression to be monitored; if allowing for more than one, note that when the monitor command is inside a loop for instance, one does not want multiple instances of the same monitor to be included.

Hint: it is easiest to extend the class Environment with a (public) member variable that holds the current monitor(s).

Trace The trace command is like having a breakpoint after every command and every condition: in trace mode, the interpreter asks the user to press a key after every step. Also it shall tell what command has just been executed.

Example:

```
n=4;
k=2;
result=1;

DEBUG: Trace;
while(n!=k && k!=0){
   result=result*n/k;
   n=n-1;
   k=k-1;
}
```

This could give the following output (where again ¶ indicates waiting for user input):

```
while n!=k&&k!=0.0==true
Assignment result=result*n/k==2.0
¶
Assignment n=n-1.0==3.0
¶
Assignment k=k-1.0==1.0
```

¶
while n!=k&&k!=0.0==true
¶
Assignment result=result*n/k==6.0
¶
Assignment n=n-1.0==2.0
¶
Assignment k=k-1.0==0.0
¶
while n!=k&&k!=0.0==false

Again, there are several simplifications one can make here, e.g., instead of displaying the actual command and value like Assignment result=result*n/k==2.0 one could simply display the assigned value:

Assignment result=2.0

and similarly for a condition only whether it is true or not.

Hint: again it could be useful to add a member variable to class Environment: a Boolean flag that tells whether the interpreter is in trace mode. Then the normal eval method for commands can check that Boolean and perform any necessary break.