# Lab 5 Frequently Asked Questions

Question 0:
My line numbers are different than yours; they are off by 1 in some places and by 2 in others, etc.  Can you tell me what is wrong?

Answer:
Your problem is caused by your definition of the single line comment.  Here's how you defined it:

```
Helpers
   cr = 0x000d;
   lf = 0x000a;
   end_of_line = (lf | cr | lf cr);
Tokens
   sl_comment = '//' [[0x0000 .. 0xfff] – [cr + lf]] *  end_of_line;
```

The problem with your definition of sl_comment is your definition of the end_of_line helper. In particular, "lf cr" will never be matched because the carriage return always comes before the line feed at the end of a line.  Thus the carriage return and line feed will be split into different tokens (cr in the sl_comment token and lf in a white space token) and will be counted twice in documents/programs that contain both to mark the end of a line (Windows text files); this will cause the line counting to be off by one for each single line comment.  This is the best explanation that I have because when the end_of_line helper is defined as follows, the problem goes away, i.e., the line numbers are correct.

```
   end_of_line = (lf | cr | cr lf);
```

You will also have the same problem for the same reason if you define end_of_line as follows:

```
   end_of_line = (lf | cr);
```

Thus you must include the sequence of carriage return followed by linefeed as an end marker of a single line comment (so the sequence is recognized as part of the same token), otherwise the count will be incorrect.

Question 1:
Lab 5 - Level 1a - Step 5 Says:
"The production rules you are looking for are those expressions with a single non-terminal on the RHS."

What about the following rules?

```
member_decl = {field} field_decl;
member_decl = {method} method_decl;
simple_stmt = {block} block;
ETC.
```

Answer:
These production rules do not define expressions, so they are not included. These rules define part of the declaration and statement syntax. Field and member declarations are handled in lab4 when the symbol table is built. A block is handled as described in a later level of lab5.

Also, constructs such as statements and declarations do not have a type attribute, i.e., statements and declarations do not create a new value; statements are executed for their side-effects and declarations create bindings whereas expressions create new values and have an associated type.

Question 2:
I have noticed a couple of issues with my SymTabBuilder.java while testing Level 1a of the Lab5. The Lab4 tests did not catch these errors, because the actual type checking is not done during Lab4. I am assuming that I will get one more chance to modify SymTabBuilder.java, as part of doing Lab 5.

Answer:
Absolutely. You can change any of the previous labs as necessary when working on lab5. The entire project will be graded based on the final version of lab5 since it combines all of the previous programming assignments.

Question 3:
The Level 1a requirements say to use method 'lookup'. Since I'm already using 'lookup' in SymTabBuilder.java, why do we have to use 'lookup' in Typechecker.java..

Answer:
Lab4 traverses the parse tree and builds the symbol table for the whole program; this symbol table must be used in Lab5, i.e., in phase 2 of the type checker. The fields and methods, declared in each scope, have to be inserted into the symbol table during Lab4 so forward references and mutual recursion can be handled when resolving identifier names in Lab5. When you traverse the parse tree with the 'TypeChecker' visitor object (Lab5), you will have to lookup the name in the symbol table (which was inserted during phase 1 of the type checker). When you find the ScopeEntry in the symbol table, you must then enter that scope. Once this is done, all the methods and fields declared in this scope can be found when resolving (looking up) the identifier names that denote these entities. Everything you did in lab4 to ensure the correct scope is entered and exited must also be done during the second traversal of the parse tree, i.e., during type checking. Because bindings for these fields and methods have already been inserted into the symbol table, they must NOT be inserted again during the second traversal, i.e., during phase 2 of the type checker. Nonetheless, during type checking the correct scope must be entered and exited so these bindings will be found in the symbol table when identifier names are resolved (see Level 1b and Level 3 and 4 for more details about how the symbol table is used).

Question 4a:
I have a question about the list of about 20 type rules you have given in the Lab5 Handout. Have some of the rules (e.g., the first few rules) already been implemented as part of levels 1(ab) and 2(ab)?

Answer:
Yes, many of the rules have already been implemented in the first two levels and a lot of the code is provided. That is, you should have implemented many of these rules while completing levels 1 and 2 using the "helper" methods provided in the type checking skeleton file.

Question 4b:
Do we need to change any of the “helper” methods to make sure they properly implement any of the type rules listed?

Answer:
No, you should not have to change any of the helper methods in the TypeChecker.java skeleton file. However, if there are any errors in these methods, I would appreciate your bringing them to my attention.

Question 5:
I’m not sure I understand what is meant by “preventing cascading error messages”.

Answer:
The main idea is that the type checker should not issue an error message if this error is caused by a previous error. To prevent this, we set the type of an expression to “error type” (if the type is unknown) or to a “default type” (if the type is known). For example, all relational expressions have type boolean. Therefore, the default type for relational expressions would be boolean. However, when the type cannot be determined because there is no default for the expression, then the type must be set to the error type. If you look at the code examples in the skeleton, you will note that error type is always treated as if it were the correct type; this prevents cascading error messages where one error propagates additional, superfluous error messages. The error type must be treated like a universal type that is valid in any expression.

Question 6:
Can method parameters be re-declared in the method body?
For example,

```
int  method1 (int i, int k)
{
   int k;
}
```

Answer:
Because the parameter ‘k’ becomes inaccessible, most programming languages do not allow this, e.g., Java does not allow this. However, you are not required to check for or prevent such re-declarations from occurring in programs. On the other hand, for extra credit, you can prevent such re-declarations if you want, but this is optional in our type checker.

Question 7:
In my current work, I define what the correct type combinations are in the ‘isAssignable()’ method. Any input that does not fall into any of these combinations is invalid. So my program does not find out exactly why the input is wrong, and thus I have no way to print out error messages like yours. Do I have to redo my entire ‘isAssignable()’ method to add this capability.

Does my code have to output error messages exactly like the ones in your test output files for Literals_E.jl or Paren_E.jl?

Answer:
My ‘isAssignable()’ method is about 15 lines of code and does not issue any error messages; it just returns true or false (since the return type is boolean). The error messages are issued where this method is called so they can be made specific.

Thus your error messages should be nearly the same. You just have to make sure that each error message is helpful to the "programmer" who has to read it. It's not good to give an error message like "something is wrong with this statement". A compiler needs to be specific and helpful to the programmer. In any case, if you are doing the right thing, you shouldn't have to completely rewrite everything you have done previously. If you do need a complete rewrite, then hopefully you will have learned something about the design of "reusable" and modular code.

If our messages do differ, then you must explain (in your readme file) why they differ. You have to make sure that every valid error message is issued and that no invalid or superfluous messages are printed; this is the main criteria for deciding whether an error message should be given. Just make sure you aren't issuing "cascading" error messages and that you aren’t missing specific error messages.

It is always possible that the error messages in the output files are not complete (so let me know if that is the case).

Question 8:
I am testing one of the files of level 1b of lab5. According to the output for this test file, the result should be "0 errors were detected", but I think there should be an error there.

In the input file IdRef_2, the class ‘OtherClass’ is not a member of class ‘Main’, i.e., ‘OtherClass’ is not in the scope of class ‘Main’; so when the type checker checks "int i = OtherClass.k", the lookup of ‘OtherClass.k’ will fail from inside Main because ‘OtherClass’ is not in scope.

Answer:
It shouldn't fail if you built the symbol table properly in phase 1 of the type checker. OtherClass should be in the global scope (i.e., it should have been inserted in Lab4 in the SymTabBuilder class). Your lookup function should look for OtherClass in class Main. Of course, as you said, it won't find it there. Then the lookup function MUST look in the surrounding scope (like it should when looking for any identifier name), i.e., it must next look in the global scope. It will find OtherClass there. If your lookup method does not do this, then you have to change it so it does. It sounds like your lookup method does NOT work properly.

Once you have found ‘OtherClass’ you then lookup k inside this class. So there should NOT be any errors in IdRef_2. This test was specifically designed to make sure your type checker works properly when referring to classes and fields defined later in a program. If you are getting errors, then you are not managing scope properly.

Question 9a:
I understand that we should override the InABlock() and OutABlock() methods in the TypeChecker program (Lab 5). Do we have to do the same in the SymTabBuilder program (Lab4)?

Answer:
Not if Lab4 is working properly and you implemented local declarations exactly as shown in the grammar of Lab3 (local declarations do not overlap with, i.e. are separate from, field declarations). Local declarations have to be handled in Phase 2 of the type checker (Lab5), not in Phase 1 (Lab4).

Question 9b:
If a new block is created in SymTabBuilder, how would we insert that block into the symbol table since it doesn't have an identifier that can be used to retrieve that block in TypeChecker?

Answer:
You don't need to insert a binding for a block into the symbol table because, as you say, there is no identifier to associate with that block entity. However, you do have to enter the scope of a block whenever a block is encountered in Phase 2 of the type checker (call method enterNewBlock of SymbolTable).

Question 9c:
If a new block is created only in the TypeChecker program, how do we insert a local declaration?

Answer:
You have to override the methods for local declarations in Phase 2 (Lab5) of the type checker (but not the methods for field declarations since that was handled in Phase 1).

That is, you do it in the same way you inserted field declarations in Lab4, except this time you insert a binding into the symbol table each time a local declaration is encountered in the program. Also, you have to enter a block scope each time a block is encountered (see part(a) above).

Question 10:
When checking procedure calls, the lab5 requirements specification suggests that we build an ArrayList (or Vector) of the types of the actual parameters (arguments) passed to a method call. I'm not sure how to do that.

Answer:
The syntax of the actual parameters in a method call is defined using recursive production rules. Recursive rules must always have a base rule that is non-recursive, otherwise the rules will not generate sentences (they will not terminate). For example,

```
(1)     args --> expr
(2)     args --> args  comma  expr
```

The first alternative is the base, non-recursive rule. This rule is non-recursive because the right-hand (rhs) side does not refer to 'args' the symbol on the left-hand side (lhs).

For the above syntax, the way to create an ArrayList of types for the symbol 'args' is as follows. Instantiate a new ArrayList and then add the type of 'expr' from the rhs of the base rule (1). This new ArrayList becomes the attribute of 'args', the symbol on the lhs of rule (1).

Then for the second alternative, the attribute of 'args' from the rhs (an ArrayList) is retrieved and the type of 'expr' from the rhs is added to this ArrayList. This ArrayList with the additional type becomes the attribute of 'args' on the lhs.

If you have defined 'args' differently, then creating the ArrayList is a little different. For example,

```
(1')    args --> expr  more_exprs *
(2')    more_exprs -->  comma  expr
```

Probably the easiest way to create the ArrayList of types for rule (1') is to first create the ArrayList and then add the type from 'expr' to this ArrayList. Next you will have to convert the list generated by 'more_exprs*' to an array (node.getMoreExprs().toArray()). Finally, loop through the array adding the type from each 'more_exprs' node to the ArrayList. You must always make sure the types of the arguments occur in the correct order; you won't be able to check the argument types unless they are in the same order as the method parameters.

The node classes for lists generated by '*' start with an 'X'; look in the node package (you will have to traverse lists of these objects). The SableCC tool generates these classes automatically. The list nodes are implemented as a TypedLinkedList (an extension of the Java LinkedList class--this is why the call to 'toArray()' works). You can look at the DepthFirstAdapter class to see how this list of nodes is converted to an array and then traversed.

Another alternative would be to define ‘more_exprs *’ as follows (this is NOT the SableCC syntax):

```
        more_exprs_star -->
        more_exprs_star --> more_exprs_star   more_exprs
```

If you are having trouble, then change the syntax to the original syntax specification, (1) and (2); traversing the nodes that represent rules (1) and (2) is a little easier for some students to understand since it is more like type checking expression nodes (however, note that if you use the ‘*’ on the final exam, then you will have to know how to traverse a list of nodes generated by ‘*’ and it’s not difficult).