# ICSI 311 Assignment 4 – Comments and Assignments

**This assignment is extremely important – (nearly) every assignment after this one uses this one!**

**If you have bugs or missing features in this, you will need to fix them before you can continue on to new assignments. This is very typical in software development outside of school.**

**You must submit .java files. Any other file type will be ignored. Especially “.class” files.**

**You must not zip or otherwise compress your assignment. Blackboard will allow you to submit multiple files.**

***You must submit buildable .java files for credit.***

In this assignment, we will add comments and assignment statements. Comments, as you know, are ignored by compilers. But that ignoring has to be specifically coded. It doesn’t just happen. Assignment statements are of the form:

variableName := expression

Of course, you have already written (most of) expression. As promised, we are going to use that code now. By the end of this assignment, your interpreter will be using variables and running

## Comments

We need to add a new state to the lexer for comments. When the lexer sees (\* it will go into the comment state. In the comment state, all characters are ignored until it encounters \*). Note that we already process parenthesis, so you will have to add a (single character) lookahead to determine if this is a comment or just a parenthesis. When the comment ends, the compiler should go to the start state. For the input:

abc(\* comment \*)def

The lexer should generate two different identifiers “abc” and “def”.

Other than the changes to the lexer, nothing else needs to be done to support (ignore) comments.

## Assignment statements

We need to add := to the lexer. Create a new token for it and add it to the lexer. Notice that in this language there is no assignment from functions like in Java:  
char c = getACharacter();

Instead, in Shank, you would say:

getAChar(var c : character)

This makes assignments simpler. We just need to make the parser expect assignments and then (later) make the interpreter process them. Create two new ASTNodes called AssignmentNode and VariableReferenceNode. VariableReferenceNode should contain the name of the variable being referenced. AssignmentNode should have a VariableReferenceNode (for the variable being assigned) and an ASTNode for the expression that is being assigned. The tradition in compilers is to use “lhs” and “rhs” (left-hand side and right-hand side) for these members. This is not a good tradition. I used “target” and “expression”.

Create a new ASTNode called “statementNode”. Make assignmentNode derive from it (ASTNode🡪 statementNode🡪 assignmentNode). Add a collection of statementNode to the functionDefinition. Now our functions can have statements.

Make a parser method that creates assignments or fails (returns null). Assignment() succeeds if it finds:

Identifier assignment expression endOfLine

Hint: I added a “peek” function that does a look-ahead in the token stream. I used that to find the assignment token. If that is there, I know that this is an assignment. Therefore, if identifier is not there or expression fails, I throw exceptions.

Eventually, we will have many types of statements, not just assignment. Create a statement() function that just returns the result of our assignment() function. The caller of the statement() function now has a simple way to process the next statement. Next we need to process a number of statements – a block can have more than one. How do you process a block of statements? We will follow the same pattern that we have been using – delegation. “statements()” calls “statement()” until statement() doesn’t succeed, just like we did with functionDefinitions. A block consists of

begin endOfLine

{statements}

end endOfLine

Statements() should return a collection of statementNode. Do note that we will reuse this statement processing function later, so make sure that you keep the “target” (functionDefinition in this case) flexible.

You might notice a pattern – we make ASTNodes to hold the data and in parallel we make methods that look at the token stream to see if it matches the expected tokens.

## Testing

Add some assignment statements between the begin and end statements from your previous assignment. Add some comments, too. Make sure that the comments are ignored and the assignment statements are printed as part of your function.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Rubric | Poor | OK | Good | Great |
| Comments | None/Excessive (0) | “What” not “Why”, few (5) | Some “what” comments or missing some (7) | Anything not obvious has reasoning (10) |
| Variable/Function naming | Single letters everywhere (0) | Lots of abbreviations (5) | Full words most of the time (8) | Full words, descriptive (10) |
| Comment lexing | None(0) | Attempted (5) | Misses some cases of entering/leaving comment state (10) | Ignores comments but no other tokens (15) |
| ASTNodes – Assignment | None (0) |  | Class present, some methods/members missing (3) | All methods/members (5) |
| ASTNodes – Statement | None (0) |  | Class present, some methods/members missing (3) | All methods/members (5) |
| ASTNodes – function changes | None (0) |  | Class present, some methods/members missing (3) | All methods/members (5) |
| ASTNodes – variableReference | None (0) |  | Class present, some methods/members missing (3) | All methods/members (5) |
| Parser – assignment | None (0) |  | Parses valid assignments (5) | Parses valid assignments and throws exception if name or expression is not found (10) |
| Parser – statement | None (0) |  |  | Calls assignment and returns value from it (5) |
| Parser – statements | None (0) | Attempted (5) |  | Expects begin and end and calls statement until it fails (10) |
| Parser – function integration | None (0) | Attempted (5) |  | functionDefinition calls statements and integrates results (10) |
| Testing – printing whole functions | None (0) | Attempted (5) |  | Complete functions printed including definition and all statements (10) |