# ICSI 311 Assignment 3 – Add Function Definitions

**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.***

This assignment **must** have \_\_\_ new source files (variableDefinition.java) **as well as** your existing source files. With the “bones” of our interpreter (lexer, parser, interpreter) started, now we can start adding features. Our first feature will be function definitions.

## Lexer Changes – Words!

We start in the lexer. We will need a new state for “words”. Be careful about the transition from “number” state to “word” state and vice versa; make sure that you deal with the character accumulation correctly. It might be tempting to use a single large group of “if-then” blocks for this. I have found that it is easier to reason about by separating my logic by state. So I have something like this (pseudo-code):  
if (state is word) {

switch (currentCharcter) {

}

} else if (state is number) {

switch (currentCharacter) {

}

Else if …

There are two types of words that we need to deal with. One type is reserved words, which we know ahead of time (like “integer” and “real” and “variable”). The other type is words that we don’t know about in advance, like function names or variable names. For reserved words, we will want to output specific tokens. For other words, we will have a generic token (“identifier”). A super easy way to do this is to make a HashMap (String, TokenType) of reserved words. When your lexer completes a word, look in the HashMap; if it is in there, create a Token using that TokenType. Otherwise, use the “Identifier” token type.

Create the token types that we need:

Identifier, define, leftParen, rightParen, integer, real, begin, end, semicolon, colon, equal, comma, variables, constants

Add integer, real, begin, end, variables, constants to the HashMap with their matching token type.

Add the state(s) for “words”. When you find a word, look it up in the hashmap and make a token, as described above. Add comma, colon, equal and semicolon to your lexer as well (these are just characters like plus and minus were).

## Parser Changes

With that complete, we can look at the parser. We want to start with a subset of our language. Consider this piece of code:

|  |  |
| --- | --- |
| **Code** | **Tokens** |
| define start (t : integer; s : real) | define identifier leftParen identifier colon integer semicolon identifier colon real rightParen endOfLone |
| constants | constants endOfLine |
| pi=3.141 | identifier equal number endOfLine |
| variables | variables endOfLine |
| a,b,c : integer | identifier comma identifier comma identifier colon integer endOfLine |
| begin | begin endOfLine |
| end | end endOfLine |

How could we describe this? There is a function declaration line, then a constants section, then a variables section, then a body. The function declaration is required. The constants and variables sections may not be in every function. A body is required.

The function declaration is the word “define”, then a name, left parenthesis, then a list of variable declarations, separated by semi-colons and finally a right parenthesis.

The constants section has one (or more) name/value pairs.

The variables has one (or more) lists of names followed by a colon followed by a data type. One non-obvious element is that constants have a data type, too. It is just inferred from the value. Given this, let’s add a new ASTElement that represents both – VariableNode. The VariableNode should have a name, an “is constant”, an enum for data type (integer and real, for now) and an ASTNode for the initial value (which will be a RealNode or an IntNode, for now). Make sure that you add a ToString() method that prints all of the fields in a readable way – this helps a lot for debugging.

To build the parser, we follow the description above.

Make a “FunctionDefinition” function, including the “ToString()” method, which should print out the local variables and the parameterVariables as well as the function name. It looks for “define”. If it find that token, it starts building a functionAST node. It populates the name from the identifier, then looks for the left parenthesis. It then looks for variable declarations (see below). We then call the Constants, then Variables, then Body function from below. The functionAST should have 2 different collection classes of VariableNodes – one for parameters and one for local variables.

We make a “Constants” function. It looks for the constants token. If it finds it, it calls a “processConstants” function that looks for tokens in the format:

Identifier equals number endOfLine

It should make a VariableNode for each of these – this should be a loop until it doesn’t find an identifier anymore.

We then make a Variables function that looks for the variables token. If it finds it, it then looks for variable declarations and makes VariableNodes for each one.

A variable declaration is a list of identifiers (separated by commas) followed by a colon, then the data type (integer or real, for now) followed by endOfLine (for variables section) or a semi-colon (for function definitions). For each variable, we make a VariableNode like we did for constants.

We then make a BodyFunction that looks for begin, endOfLiine, end,endOfLine . Right now, we don’t do anything with these. We should now be able to parse function declarations!

## Testing

To test your code, remove the call to Resolve() in the Interpreter and to Expression() in the Parser from your main. Instead, call FunctionDefinition. We will use Resolve() and Expression() again soon, so don’t delete them. From your main, print out the FunctionDefinitions using the ToString() so that you can be sure that they parsed correctly.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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) |
| Create the AST classes | None (0) | Classes missing (5) | All classes present, some methods missing (10) | All classes and methods (15) |
| Lexer – string state | None (0) | Ad Hoc (5) | Has a state but some transition cases don’t work (10) | Has a new state and all transition cases work (15) |
| Lexer – recognizes keywords | None (0) | Ad Hoc (5) | Uses hash map (10) | Uses hash map and all required entries exist (15) |
| Parser – function definition | None (0) | One of: Uses MatchAndRemove, looks for all tokens, throws exceptions if required and not found (5) | Two of: Uses MatchAndRemove, looks for all tokens, throws exceptions if required and not found (7) | Uses MatchAndRemove, looks for all tokens, throws exceptions if required and not found (10) |
| Parser – variable declarations | None (0) | One of: Uses MatchAndRemove, looks for all tokens, throws exceptions if required and not found (5) | Two of: Uses MatchAndRemove, looks for all tokens, throws exceptions if required and not found (7) | Uses MatchAndRemove, looks for all tokens, throws exceptions if required and not found (10) |
| Parser – variable and constant section | None (0) | One of: Uses MatchAndRemove, looks for all tokens, throws exceptions if required and not found (5) | Two of: Uses MatchAndRemove, looks for all tokens, throws exceptions if required and not found (7) | Uses MatchAndRemove, looks for all tokens, throws exceptions if required and not found (10) |
| Parser – body section | None (0) | One of: Uses MatchAndRemove, looks for all tokens, throws exceptions if required and not found (5) | Two of: Uses MatchAndRemove, looks for all tokens, throws exceptions if required and not found (7) | Present and checks for begin and end (5) |