# ICSI 311 Assignment 6 – More on Functions

**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 finish up the lexer and parser (for now) and start on the work to get the interpreter completed. We will start by adding the code to lex/parse function calls. Then we will implement the built-in functions.

## Adding Function Calls

We start by adding “var” to the lexer tokens and the hash map of known key words. Remember that “var” is the key word that allows us to alter passed in variables – passing by reference rather than by value.

We need to add an AST node for function calls. This is a new StatementNode. A function call has a name (of the function) and a list of parameters. A parameter needs to be its own ASTNode. Remember that a parameter can be a variable (VariableReferenceNode) or a constant value (an ASTNode). It can have “var” or not have var. Remember to add a “ToString()” for these nodes for debugging.

Next we have to add function calls to the parser. A function call is a name followed by a comma separated list of 0 or more cases of one of three cases:

var name

name

value (which could be int or float)

Finally, we need to add this to the Statement() function in the parser so that function calls become a valid statement type.

## Built-In: AST Nodes

Next, we will need an ASTNode for our built-in functions. There is a significant difference between functions that are user-defined in Shank (which we have done already) and functions that are pre-defined (like read and write). The user-defined functions have statements, for example, while the pre-defined functions have Java code to implement them. So it makes sense to have two different classes with a common base type. Make a common base type of “CallableNode”, which derives from ASTNode. It will have a function name and a list of VariableNodes for the parameters – you can move this from the existing functionNode. This class should be abstract. Then we need to build BuiltInFunctionNode and FunctionNode.

BuiltInFunctions can do something that user-defined functions cannot (so far) – accept any number of parameters of any type (like read and write do). This is called **variadic**. C and Java both do this. Make a boolean in BuiltInFunctions to indicate if this built-in is variadic.

FunctionNode needs to now inherit from CallableNode and to use the inherited Parameter variables.

## Built-In – Parameters

There are two aspects to parameters, and this can get a little confusing. When we make a function, we declare the parameters to the function:  
define someFunction ( a,b:integer)

When we call that function, the function call has parameters:  
someFunction 3,4

These are all stored in the AST tree. But there is a third aspect that we need to think about – the data storage at interpreter time. The AST is different – we don’t want to change that while we are running the code. We need a data structure that the interpreter can use to hold data without changing the values in the AST.

Create a new set of classes: InterpreterDataType, IntDataType, FloatDataType. The first is an abstract base class. It declares a ToString and a FromString:

public abstract String ToString();

public abstract void FromString(String input); // sets the value of the data type by parsing the string

The int and float versions have a Value (of the appropriate type) and should implement FromString() and ToString() – we will use these in our read and write functions.

Finally, we are going to go make one quick addition to BuiltInFunctionNode – add an abstract method (making the whole class abstract) called Execute. Execute will take a collection of InterpreterDataType objects. Why? Well, when the interpreter finds a call to “read”, for example, it has to be able to call your Java code.

Now subclass BuiltInFunctionNode for each of the functions that we can implement so far:  
read

write

squareRoot

getRandom

integerToReal

realToInteger

Implement the Execute function for each of these. Use the collection of InterpreterDataType to get parameters and to output to variable parameters. Make sure that you check the data types of the InterpreterDataTypes. Throw exceptions if the functions are called incorrectly.

## Testing

Create some function calls in your code. Ensure that the AST nodes print. We will test our built-in code next assignment.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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) |
| Add var to the lexer | None (0) |  |  | Token and in the hash map (5) |
| FunctionCall AST node | None (0) |  | Attempted(3) | Implemented(5) |
| Parameter AST node | None (0) |  |  | Fields and methods correct (10) |
| Function Call parsing | None (0) |  |  | Correctly handles all function calls and makes appropriate AST nodes (10) |
| BuiltInFunction AST | None (0) |  | Attempted(3) | Implemented(5) |
| InterpreterDataType | None (0) |  | Attempted(3) | Implemented(5) |
| IntDataType | None (0) |  | Attempted(3) | Implemented(5) |
| FloatDataType | None (0) |  | Attempted(3) | Implemented(5) |
| Read | None (0) |  | Attempted(3) | Implemented(5) |
| Write | None (0) |  | Attempted(3) | Implemented(5) |
| Square Root | None (0) |  | Attempted(3) | Implemented(5) |
| GetRandom | None (0) |  | Attempted(3) | Implemented(5) |
| IntegerToReal | None (0) |  | Attempted(3) | Implemented(5) |
| RealToInteger | None (0) |  | Attempted(3) | Implemented(5) |