# ICSI 311 Assignment 7 – Calling Functions in the interpreter

**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.***

## Background

Calling functions is a tricky business. There are several steps to take to make sure that everything is handled correctly.

Consider:

define add (a,b : integer; var c:integer)

begin

c:=a+b

end

Now consider some different ways that we could call this code.

add 1, 2, var c

add a, 2, var a

And some things that shouldn’t happen (and should be an error!):  
add 1,2,3

add 1,2,a

add 2.3 7.4 var a

Another “gotcha” to think about – what if we change the value of a variable that is **not** marked as var. What should happen? We could throw an error. That would require us to look at every assignment statement and judge if the assigned variable is changeable. But we would also have to look at every function call, to see if the passed variable is marked as var!

For example:

define changer(var a : integer)

begin

a:=a+1

add a,1,var a

end

And what if I call changer with:  
changer z

One approach to this is to forbid it. If you try to call a function without matching “var”, it should fail. This is a legitimate way to handle this situation, but we are doing something else – something that will let us deal with constants, too.

When a function is called:

1. Locate the function definition; this could be a built-in (like read or write) or it could be user-defined.
2. Make sure that the number of parameters matches OR that the function definition is variadic and built-in.
3. Make a collection of values (InterpreterDataType):

For every parameter in invocation:

* 1. Add the constant value or the current value of the variable in the invocation

1. Now we call the function (either the interpreter or the “execute” of the built-in function), passing it our collection.
2. Finally, we loop over that set of values – the called function might have changed some!
   1. For each value, if the called function is variadic or the called function is marked as VAR **and** the invocation is marked as VAR then
   2. Update the working variable value with the values “passed back” from the function.

## Work on the interpreter

Create a static InterpretFunction in the interpreter – it should take a FunctionNode (i.e. the function to interpret) and a collection of InterpreterDataType – the parameters to the function.

To interpret a function, we will make a hashmap of string->InterpreterDataType – this will hold our variables. Add all of our parameters to the hashmap using the names that our function expects. Next add all of the local variables to the hashmap. Remember that we stored the constants in the “Local Variable” section, so we need to set the initial values of these variables as appropriate. Finally, we will call a function called “InterpretBlock” - this function will process all of the code between “begin” and “end”; we will use it later for conditionals and loops.

InterpretBlock should take the collection of statements and a hashmap of variables. We will loop over the collection of statements. For now, the only statement type that we will handle is function calls. If the statement is a function call, implement the process described in the background section, otherwise we will ignore the statement (for now).

## Finish up

In “main” we are using the parser to parse functions. Create a hashmap in the interpreter that maps names (strings) to CallableNodes. In main, every time we encounter a function, add it to that hashmap. Also add the built-in functions that we created previously to the hashmap. We now have a complete data structure of all of our code. Finally, in main, call InterpretFunction on the function named “start”; it is an error for that function not to exist.

## Testing

You should now be able to run programs so long as they only call functions. You can use read and write to get data in and write data out. You can test random numbers and data conversions, too!

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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) |
| InterpretFunction | None (0) | One of: creates variables from the parameters, local variables and calls interpretBlock (5) | Two of: creates variables from the parameters, local variables and calls interpretBlock (10) | Creates variables from the parameters, local variables and calls interpretBlock (15) |
| Main | None (0) | One of: Creates hashmap of functions, adds built-in functions, calls start (5) | Two of: Creates hashmap of functions, adds built-in functions, calls start (10) | Creates hashmap of functions, adds built-in functions, calls start (15) |
| InterpretBlock – Create parameters | None(0) | One of: Checks parameter counts, creates values from constant values, creates values from variables (7) | Two of: Checks parameter counts, creates values from constant values, creates values from variables (13) | Checks parameter counts, creates values from constant values, creates values from variables (20) |
| InterpretBlock – call the function | None(0) |  | Calls either built-in functions OR interpretFunction (5) | Calls both built-in functions and interpretFunction (10) |
| InterpretBock – update variables | None(0) |  | Updates all variables (10) | Updates VAR variables correctly (20) |