ICSI 311 Assignment 8 – Interpreter Part 2

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. Brightspace will allow you to submit multiple files.

You must submit every file for every assignment.

You must submit buildable .java files for credit.

Introduction

An interpreter, generally, is conceptually simple – look at the "current node" in the program tree and "do it". Of course, "do it" is a little bit of hand waving, but most of the nodes that we have created aren't really that complex.

The interpreter structure will mirror the parser's structure. The interpreter will interpret a program. That will be done by interpreting blocks. Blocks are, of course, an optional condition and a list of statements.

Functions throw a pretty big wrench into the simplicity of our interpreter. For non-function code, we can just use the GlobalVariables hash map. But because we also need to be able to run functions, many of our methods will need to take a LocalVariables parameter to support the function's local variables. We will call these methods with null when processing a block and with the local variables hash map when we are processing a function.

We are going to build "from the bottom up" – we will build functions that handle simple, low level nodes (like OperationNode) and work our way toward handling a ProgramNode.

Details

We will write a method called "GetIDT". It will take a Node and a hash map of local variables (keep in mind this could be null). It will evaluate the node and return an IDT. Which node? Many of them:

AssignmentNode: make sure that the target is a variable (variableReferenceNode or OperationNode with type DOLLAR). Call GetIDT on the right side of the assignment. Set the target's value to the result. Return the result.

ConstantNode: return a new IDT with the value set to the constant node's value.

FunctionCallNode: create a "RunFunctionCall" method that takes the function call node and locals and returns a String. For now, it can just return "" – we will finish it later. Make a new IDT with the result of RunFunctionCall.

PatternNode – this is an error. This is a case where someone is trying to pass a pattern to a function or an assignment. Throw an exception.

TernaryNode – evaluate the boolean condition (using GetIDT), then evaluate (using GetIDT) and return either the true case or the false case

VariableReferenceNode – There are two cases – is this an array reference or not? If it is not, this is simple – just look the variable up in globals/locals and return it. If it is an array reference, you have to resolve the index then look the index up in the variable's hash map. If the variable is not an IADT, throw an exception.

OperationNode: Evaluate the left and right (if there is one) using GetIDT. Then perform the operation. For math-based operations, convert to float, do the operation then convert to string to store in a new IDT.

For compares (equal, not equal, etc), compare as floats if both sides covert to float, otherwise compare as strings.

For boolean operations (and, or, not), the rule in AWK is that any value that can be converted to float and is not 0 is true. So "1", "25.3", "-22.9" are all true and "", "true" and "0" are all false.

For match and not match, use Java's built-in "Regex" object; this is an exception to the rule above – don't use GetIDT on the right side, it must be a patternNode.

The dollar operation evaluates the left side, adds a "$", then gets that as a variable. For example, $(1+1) → Operation($, Node (ADD 1 1)).

Pre/post increment/decrement, unary +/- are similar to the math-based operations.

Concatenation is simple string concatenation.

For "IN" – check to make sure the right hand side is a variable reference and is an array (throw an exception if not). Then look up the left hand side in the array (which will be in globals or locals).

Unit testing this method is a little tedious – you really have to construct the parse tree yourself. Consider something like this:

var node = new OperationNode(ASSIGNMENT, new VariableReferenceNode("a"),

OperationNode(ADD, new ConstantNode("2"), new ConstantNode("2")).

If you create a new Interpreter and call GetIDT(node, null) you should be able to look at the global variables and see that "a" now exists with a value of 4.

Rubric	Poor	OK	Good	Great
Code Style	Few comments, bad names (0)	Some good naming, some necessary comments (3)	Mostly good naming, most necessary comments (6)	Good naming, non-trivial methods well commented, static only when necessary, private members (10)
Unit Tests	Don't exist (0)	At least one (3)	Missing tests (6)	All functionality tested (10)
AssignmentNode	Doesn't exist(0)	Attempted (5)		Evaluates both sides and assigns value (10)
ConstantNode	Doesn't exist(0)			Creates new IDT (5)
FunctionCallNode	Doesn't exist(0)			Calls stubbed out RunFunctionCall (5)
PatternNode	Doesn't exist(0)			Throws exception with good error message (5)
TernaryNode	Doesn't exist(0)			Evaluates condition, then either true or false condition (5)
VariableReferenceNode	Doesn't exist(0)	Handles scalars (2)	Handles arrays (4)	Handles both arrays and scalars (5)
Operations – Math	Doesn't exist(0)	Handles at least one (2)		Handles +,-,*,/,^,% (5)
Operations – Compare	Doesn't exist(0)	All work for strings OR numbers (5)		All work for strings and numbers (10)
Operations – Booleans	Doesn't exist(0)	At least one works (3)		And, or, not work as expected (5)
Operations – match	Doesn't exist(0)			Uses Java match (5)
Operations – dollar	Doesn't exist(0)			Evaluates dollar expression and assigns (5)
Operations – pre/post/unary	Doesn't exist(0)			Updates and returns correct value (5)
Operations – Concatenation	Doesn't exist(0)			Concatenates and returns new value(5)
Operations – In	Doesn't exist(0)			Returns 1 or 0 and throws appropriate error messages (5)