ICSI 311 Assignment 7 – A little more parsing and Interpreter Part 1

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

There are certain built-in functions in AWK that are not called the same way as user-defined functions. But some built-in functions **are** called the same way as user-defined functions.

getline, print, printf, exit, nextfile, next – all of these can be called without parenthesis.

Now, let's talk about interpreters. Remember that our AST is **read only**. It is a definition of our program in memory. But, of course, we know in computer science, our program needs two things to run: storage and input/output. The AST is the program; to build the interpreter, we will build facilities for storage of variables, input/output functionality and finally some code that will look at each Node and "execute it".

AWK is dynamically typed; it is also primarily used for string processing. We will store data as strings and convert it to float only when we need to do math or format it as a number. We will also need arrays but remember that AWK's arrays are not "really" arrays (in the Java sense) – they are really HashMaps (JavaScript does this as well). It just so happens that some of the functions in AWK create arrays that have "indices" of 1,2,3,4…

On the topic of variables in AWK – there are no local variables EXCEPT in functions. This simplifies our work quite a bit but makes tracking bugs in bigger AWK programs much harder.

Finally, remember that AWK is primarily a language for processing string input. Review the AWK documentation for understand NF, FS, and the $ variables.

Details

getline, print, printf, exit, nextfile, next – all of these can be called without parenthesis.

These need to have their own special parsing in ParseFunctionCall(). That special parsing should create FunctionCallNodes just like a "regular" function call.

With that parsing done, we are now done with the parser.

Build the two variable classes that we need: InterpreterDataType and a subclass InterpreterArrayDataType. I will abbreviate these as IDT and IADT. IDT holds a string and has two constructors – one with and one without an initial value supplied. IADT holds a HashMap<String,IDT>. Its constructor should create that hash map. We will use these as our variable storage classes as the interpreter runs.

Let's start the interpreter itself. Create a new class (Interpreter). Inside of it, we will have an inner class – LineManager. We will also need a HashMap<String,IDT> that will hold our global variables. We will also have a member HashMap<String, FunctionDefinitionNode> - this will be our source when someone calls a function.

The LineManager should take a List<String> as a parameter to the constructor and store it in a member. This will be our read-in input file. It will need a method : SplitAndAssign. This method will get the next line and split it by looking at the global variables to find "FS" – the field separator; you can use String.split for this. It will assign $0, $1, etc. It will set NF. It should return false if there is no line to split. It should update the NR and FNR variables.

Now we can create the constructor for the Interpreter. It should require a ProgramNode and a file path. If a path is provided, it should use Files.ReadAllLines() and create a LineManager member, otherwise make a LineManager with an empty list of string. It should set the FILENAME global variable. Set the default value of the FS global variable to " " (a single space), OFMT to "%.6g", OFS to " " and ORS to "\n". We won't be using the "RS" variable. Populate the function hash map from the functions in the ProgramNode and each of the BuiltIn's (described below).

Next, we will create a new class: BuiltInFunctionDefinitionNode. This will derive from FunctionDefinitionNode (so that It can be used in any place where a FunctionDefinitionNode is used). The big difference is that we will be adding a **lambda expression** to hold the functionality.

https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html

The lambda function should accept a HashMap of String, InterpreterDataType and return string. The hash map will be the parameters to the function and the string will be the return value. I named this lambda function "Execute". When we are interpreting our AWK code and come across a BuiltInFunctionNode, we will call Execute which will then do the work.

Built-ins can do one other thing that functions defined in AWK cannot do – accept "any number" of parameters. This is called **variadic**. Add a boolean to the BuiltInFunctionDefinitionNode to indicate if the function is variadic.

We will then create multiple instances of this BuiltInFunctionDefinitionNode, one for each built-in function. We won't be doing all of them (although you can!), but here are the ones we must do:

print, printf, getline, next, gsub, index, length, match, split, sprintf, sub, substr, tolower, toupper

Most of these are very short because we can use the Java built-in functionality.

print, printf: both are variadic. When you create these, they will have a single parameter, but that parameter will be a IADT. Use System.out.print or System.out.printf. The one "tricky part" is that order matters. You can assume that the IADT will have HashMap entries of "0", "1", "2", etc. You can call printf in Java by passing an array:

```
System.out.printf ("%d %d %d", myIntArray);
```

An interesting issue appears here. We are storing data in our IDTs as String. But printf could be looking for numbers (%d or %f) or strings (%s). The only way to really resolve this is to parse the format string. For now, send everything to printf as a string.

getline and next : these will call SplitAndAssign – we won't do the other forms.

gsub, match, sub: for these we can use Java's built in regular expression (Regex).

index, length, split, substr, tolower, toupper: for these we can use Java's string class.

Testing for this assignment should include testing the parser changes, the line manager and each of the Built-in functions.

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)
Parser changes	None (0)		Some are correct or all are attempted (5)	All functions completed, FunctionCallNodes are created (10)
IDT/IADT	Don't exist (0)		Attempted (2)	Exist and are correct (5)
LineManager	Don't exist (0)		Attempted (2)	Members and constructor are correct (5)
SplitAndAssign	Doesn't exist (0)		Splits string, misses variable updates (5)	Splits string, populated $0..$n & NR, uses NF (10)
Interpreter	Don't exist (0)		Attempted (2)	Members are correct (5)
Interpreter Constructor	Doesn't exist (0)		Creates LineManager, doesn't pre-populate global variables and functions (5)	Creates LineManager, pre-populates global variables and functions (10)
BuiltInFunctionDefinitionNode	Don't exist (0)		Attempted (2)	Members and constructor are correct (5)

print, printf	Don’t exist (0)	Attempted (2)	Exist but not variadic or don’t call Java function (7)	Exist, marked as variadic, call Java functions. (10)
getline and next	Don’t exist (0)	Attempted (2)		Exist and call SplitAndAssign (5)
gsub, match, sub	Don’t exist (0)	One of: Exist, call Java functions, correct return. (3)	Two of: Exist, call Java functions, correct return. (6)	Exist, call Java functions, correct return. (10)
index, length, split, substr, tolower, toupper	Don’t exist (0)	One of: Exist, call Java functions, correct return. (1)	Two of: Exist, call Java functions, correct return. (3)	Exist, call Java functions, correct return. (5)