# ICSI 311 Assignment 9 – Adding more data types

**This is the end! How you do on this assignment doesn’t impact any others!**

**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

Now that we have completed the work for int and float, we are going to go back and add other data types. This is a very common pattern – adding a new feature once you have something working. Instead of “just” working in the interpreter, we will touch every piece, but just a little.

There are several data types remaining. We will leave arrays and variable ranges for next assignment.

**Boolean** – we need to add keywords “true” and “false”, allow variables of the type boolean, allow assignment and allow them to be used in places where we expect boolean expressions (like in “if” and “while”).

**Character and string** – like boolean, we need to allow them to be declared and assigned to. This requires us to add single and double quotes to the lexer. We also need to add them to the boolean expressions so that we can say something like:

if myString =”hello” then

Finally, we have some new built-in functions to implement.

My approach is to work module by module – lexer, then parser and AST, then interpreter. It would be perfectly reasonable to split these the other way and add a whole data type, then the next, etc.

## The Lexer

We need to add keywords for the new types, true, false. Then we need to add single quote and double quote modes (like we did for comments). I started by making the tokens. Note that for both single and double quotes, I don’t make individual tokens for start quote and end quote. Instead, I created a “CharContents” and “StringContents”. Note that one thing that we need to do is enforce the length limit of 1 character on CharContents.

## The Parser

Let’s start with adding the new variable types.

We implemented BooleanExpression() specifically for loops and conditions. That made sense because we knew from context that the type of expression had to be boolean (because it followed a key word like “if”). We also didn’t have a type for boolean.

We are going to merge our boolean expression into our expression() function. This might seem a little weird – 2+2 seems different from x>2. But, are they? They are both mathematical expressions that return a value.

Create a BoolNode like we created IntNode and FloatNode. Change Factor() to return a BoolNode when it finds the true and false tokens. Then merge in from your existing code the boolean operators (less than, less than or equal, etc) to ExpressionRightHandSide. Remember that we can’t chain booleans like we do addition and subtraction; that is to say 2+3+4 is allowed, but 4>x>9 is not allowed.

Likewise, create a CharNode and a StringNode for those data types. Add those to Factor() as well. This will allow expressions to serve for all of our assignment statements. One might protest that this allows all sorts of invalid things like:

2+”this is a test”\*’c’

That is true. But remember that the parser’s job is to build the AST that the user specified. We will walk through it to make sure that the types all make sense.

## Semantic Analysis

This is an additional step that is performed once parsing is complete. Diagram

Description automatically generated

In this step, we analyze the AST. In this step, we have all the data of the complete AST so we can do things like ensure (for example) that all variables are declared. In Shank, this is not an issue, since all variables are at the function scope (and declared at the top), but think about Java where a member variable could be declared after it is used in a method.

Create a new class called SemanticAnalysis. Add a method called “CheckAssignments”. We will pass our collection of functions to this method. For each function, we need to look at every assignment statement (remember that they could be in sub-blocks like inside an “if” hint - recursion). Consider the variable on the left side of the assignment as the “correct” type. Make sure that everything on the right side of the assignment is the same type. If it is not, throw an exception. Add as much information as you can to help the user debug this. Remember that string + character and character + string are allowable (both create a string). It’s OK to assume that char and string are interchangeable for this check. Also remember that operands can be variables or constants. You may need to make several functions to implement this – I did.

Call this method from “main” before you call the interpreter.

## The interpreter

Add new InterpreterDataTypes for character, string and boolean.

Implement the built-in functions for strings (refer to the Language Definition document).

Create ResolveBoolean, ResolveString and ResolveCharacter similar to how we implemented ResolveInteger and ResolveFloat. Implement the operator + for string/string, character/string and string/character. Use ResolveBoolean for if, while and repeat statements. This will allow us to use variables for conditions:

if done then

a:=a+1

else

a:=a-1

Add string, boolean and character types to function calls.

When this is complete, you should be able to use string, character and boolean anywhere in a Shank program that makes sense. Since each of your programs will vary, you will need to find any places that need additional changes.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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) |
| Lexer – new keywords | None (0) |  | At least 2 of true, false, string, character, boolean (2) | All of true, false, string, character, boolean (5) |
| Lexer – character literals | None (0) |  | Ad hoc/not state based (5) | State based and limited to 1 character (10) |
| Lexer – string literals | None (0) |  | Ad hoc/not state based (5) | State based (10) |
| Parser – New node types | None (0) |  | Class present, some methods/members missing (3) | All methods/members (5) |
| Parser – merge boolean expression | None (0) |  | Boolean expressions not usable with other expressions (5) | Boolean expressions can be used with other expressions (ex: 1+2<2) (10) |
| Semantic analysis | None (0) | One of: recursive with blocks, with MathOps and throws well worded exceptions when a type violation occurs (7) | Two of: recursive with blocks, with MathOps and throws well worded exceptions when a type violation occurs (13) | Semantic analysis is recursive with blocks, with MathOps and throws well worded exceptions when a type violation occurs (20) |
| Interpreter – boolean | None (0) |  | Boolean usage limited (5) | Booleans can be used in if, while, repeat, assignment (10) |
| Interpreter – character | None (0) |  | Characters can be used in either assignment or + (3) | Characters can be used in both assignment and with “+” (5) |
| Interperter – string | None (0) |  | string can be used in either assignment or + (3) | string can be used in both assignment and with “+” (5) |