11/14/2021

Faith Swetnam

Kennesaw State University

Faith Swetnam

Fall 2021

CS 4308/Section 03

3rd Deliverable - Interpreter

Professor Sharon Perry

98% complete and working as designed (issues noted in Conclusion)

Table of Contents

[Screenshots 2](#_Toc87815986)

[Successful Runs 2](#_Toc87815987)

[Test1.jl 2](#_Toc87815988)

[Test2.jl 3](#_Toc87815989)

[Test3.jl 4](#_Toc87815990)

[Test4.jl 5](#_Toc87815991)

[Some Error Checking 6](#_Toc87815992)

[Incorrect arithmetic/assignment operation 6](#_Toc87815993)

[Divide by 0 (operations()) 7](#_Toc87815994)

[Variable not found/invalid argument(arithmetic()) 8](#_Toc87815995)

[Incorrect value (arithmetic()) 9](#_Toc87815996)

[Variable not found/invalid argument (bool()) 10](#_Toc87815997)

[Unexpected operation (bool()) 11](#_Toc87815998)

[Source Code 11](#_Toc87815999)

[Interpreter.java 11](#_Toc87816000)

[Code Review 19](#_Toc87816001)

[Interpreter.java 19](#_Toc87816002)

[Variable 19](#_Toc87816003)

[Global variables 20](#_Toc87816004)

[Methods 20](#_Toc87816005)

[Lexical Analyzer and Parser Changes 25](#_Toc87816006)

[Conclusion 26](#_Toc87816007)

[References 26](#_Toc87816008)

# Screenshots

## Successful Runs

### A screenshot of a computer Description automatically generated with medium confidenceTest1.jl

### A screenshot of a computer Description automatically generated with medium confidenceTest2.jl

### Test3.jlA computer screen capture Description automatically generated with medium confidence

### A computer screen capture Description automatically generated with medium confidenceTest4.jl

//Test *4* in Julia  
function a()  
 x = *1* repeat  
 x += *1* until x > *100* print(x)  
end

## Some Error Checking

### A computer screen capture Description automatically generated with medium confidenceIncorrect arithmetic/assignment operation

### A computer screen capture Description automatically generated with medium confidenceDivide by 0 (operations())

### A computer screen capture Description automatically generated with medium confidenceVariable not found/invalid argument(arithmetic())

### A computer screen capture Description automatically generated with medium confidenceIncorrect value (arithmetic())

### A computer screen capture Description automatically generated with medium confidenceVariable not found/invalid argument (bool())

### A computer screen capture Description automatically generated with medium confidenceUnexpected operation (bool())

# Source Code

## Interpreter.java

/\*  
Class: CS 4308 Section 03  
Term: Fall 2021  
Name: Faith Swetnam  
Instructor: Sharon Perry  
Project: Deliverable 3 Interpreter  
 \*/  
  
import java.io.File;  
import java.util.ArrayList;  
  
public class Interpreter {  
  
 //create an object to store a variable's name and value  
 static class Variable {  
 private String name = "";  
 private String value = "";  
  
 Variable(String name, String value){ this.name = name; this.value = value; }  
 void setID (String name){ this.name = name; }  
 String getID() { return name; }  
 String getValue() { return value; }  
 }  
  
 private static Node *root*; //root node of the AST  
 private static ArrayList<Node> *nodes*; //an ArrayList of all nodes in the AST  
 private static ArrayList<Token> *tokens*; //an ArrayList to hold all Tokens  
 private static ArrayList<Error> *errors* = new ArrayList<Error>(); //an ArrayList to hold all errors generated  
 private static ArrayList<Variable> *vars* = new ArrayList<Variable>(); //an ArrayList to hold all Variables initialized  
 private static boolean *errorOccurred* = false; //boolean to determine whether error has occurred  
 private static String *output* = ""; //holds output of file  
  
 //generates an error object, adds it to errors arraylist, and sets errorOccurred to true  
 //takes in an error message, the value that threw the error, and the line it occurred on  
 private static void createError(String msg){  
 Error e = new Error(msg);  
 *errors*.add(e);  
 *errorOccurred* = true;  
 }  
  
 private static void createError(String msg, String value, int line){  
 Error e = new Error(msg, value, line);  
 *errors*.add(e);  
 *errorOccurred* = true;  
 }  
  
 //checks whether a variable has already been initialized  
 //returns the Variable if found, returns null if not  
 private static Variable varsContains(String id){  
 for(Variable var: *vars*){  
 if(var.getID().equals(id)){  
 return var;  
 }  
 }  
 return null;  
 }  
  
 //returns the index of the variable with the id passed  
 private static int getVarIndex(String id){  
 for(int i = 0; i < *vars*.size(); i++){  
 if(*vars*.get(i).getID().equals(id)){  
 return i;  
 }  
 }  
 return -1;  
 }  
  
 //This method gets all data interpreter needs to run  
 private static void getGlobals(File f){  
 *root* = Parser.*getRootNode*(f);  
 *nodes* = Parser.*getNodes*();  
 //tokens = LexicalAnalyzer.getTokenList(f);  
 }  
  
 //adds a Variable to vars  
 //creates an Error if the Variable already exists  
 private static void addVariable(Variable v, Node n){  
 if(*varsContains*(v.getID()) == null){  
 *vars*.add(v);  
 } else {  
 *createError*("variable already exists", n.getFirstToken().getLexeme(), n.getFirstToken().getLine());  
 }  
 }  
  
 //this method begins the Node processing  
 //this essentially runs the source code  
 private static void interpret(){  
 //ensure root is not null  
 if(*root* == null) {  
 *createError*("root is null");  
 //ensure root has children  
 } else if(*root*.getChildren().isEmpty()){  
 *createError*("root has 0 children");  
 } else {  
 //start interpreter  
 Node start = *root*;  
 int startChild = 0;  
 while(!start.getFirstToken().checkType(Token.TokenType.*EOF*) && !*errorOccurred*){  
 //call blocks  
 *block*(start.getFirstChild());  
 //iterate through root's children(blocks)  
 startChild++;  
 if(startChild < *root*.getChildren().size()){  
 start = *root*.getChild(startChild);  
 }  
 }  
 }  
 }  
  
 //<block> -> <statement> | <statement> <block>  
 private static void block(Node node){  
 //traverse through any block or statement nodes  
 while(node.getGrammar().equals("<block>") || node.getGrammar().equals("<statement>")){  
 node = node.getFirstChild();  
 }  
 //determine which statement method to call  
 switch(node.getGrammar()){  
 case("<assignment\_statement>"):  
 *assignI*(node);  
 break;  
 case("<print\_statement>"):  
 *printI*(node);  
 break;  
 case("<if\_statement>"):  
 *ifI*(node);  
 break;  
 case("<while\_statement>"):  
 *whileI*(node);  
 break;  
 case("<repeat\_statement>"):  
 *repeatI*(node);  
 break;  
 }  
 }  
  
 //<assignment\_statement> -> id <assignment\_operator> <arithmetic\_expression>  
 private static void assignI(Node node){  
 //get all node's children  
 Node var = node.getFirstChild();  
 Node op = node.getChild(1);  
 Node expr = node.getChild(2);  
  
 //temporary Variable object created to check if the Variable was initialized or not  
 Variable tempVar = *varsContains*(var.getFirstToken().getLexeme());  
 if(tempVar == null){  
 //variable was not initialized  
 //add new Variable with temporary value  
 tempVar = new Variable(var.getFirstToken().getLexeme(), "");  
 *addVariable*(tempVar, var);  
 }  
 //call arithmetic for expression  
 Variable arithExpr = *arithmetic*(expr);  
 //set ID of variable  
 arithExpr.setID(tempVar.getID());  
 //check if normal assignment or addition assignment  
 if(op.getFirstToken().checkType(Token.TokenType.*ASSIGN\_OP*)){  
 *vars*.set(*getVarIndex*(arithExpr.getID()), arithExpr);  
 } else if(op.getFirstToken().checkType(Token.TokenType.*AE\_OP*)){  
 int result = Integer.*parseInt*(*vars*.get(*getVarIndex*(arithExpr.getID())).getValue()) + Integer.*parseInt*(arithExpr.getValue());  
 *vars*.set(*getVarIndex*(arithExpr.getID()), new Variable(arithExpr.getID(), String.*valueOf*(result)));  
 }  
 }  
  
 //<print\_statement> -> print ( <arithmetic\_expression> )  
 private static void printI(Node node){  
 //stores the print statement's arithmetic expression node  
 Node printExpr;  
 //print depends on the number of children node has  
 if(node.getChildren().size() == 1){  
 printExpr = node.getFirstChild();  
 } else {  
 printExpr = node.getChild(2);  
 }  
 //temporary Variable for arithmetic expression  
 Variable printVar = *arithmetic*(printExpr);  
 if(printVar != null){  
 //store print output in output  
 *output* = *output* + printVar.getValue() + "\n";  
 }  
 }  
  
 //<if\_statement> -> if <boolean\_expression> <block> else <block> end  
 private static void ifI(Node node){  
 //get child nodes  
 Node boolExpr = node.getChild(1);  
 Node thenBlock = node.getChild(2);  
 Node elseBlock = node.getChild(4);  
  
 //call bool to process expression  
 boolean boolValue = *bool*(boolExpr);  
  
 if(boolValue && !*errorOccurred*){  
 *block*(thenBlock);  
 } else if(!boolValue && !*errorOccurred*){  
 *block*(elseBlock);  
 }  
 }  
  
 //<while\_statement> -> while <boolean\_expression> <block> end  
 private static void whileI(Node node){  
 Node boolExpr = node.getChild(1);  
 Node doBlock = node.getChild(2);  
  
 boolean boolValue = *bool*(boolExpr);  
 while(boolValue && !*errorOccurred*){  
 *block*(doBlock);  
 boolValue = *bool*(boolExpr);  
 }  
 }  
  
 //<repeat\_statement> -> repeat <block> until <boolean\_expression>  
 private static void repeatI(Node node){  
 //get nodes children  
 Node repeatBlock = node.getChild(1);  
 Node boolExpr = node.getChild(3);  
 //boolean to store result of boolean expression  
 boolean boolValue = false;  
 while (!boolValue && !*errorOccurred*){  
 *block*(repeatBlock);  
 boolValue = *bool*(boolExpr);  
 }  
 }  
  
 //<boolean\_expression> -> <arithmetic\_expression> <relative\_op> <arithmetic\_expression>  
 //returns the result of the boolean expression  
 private static boolean bool(Node node){  
 //get nodes children  
 Node n1 = node.getFirstChild();  
 Node op = node.getChild(1);  
 Node n2 = node.getChild(2);  
  
 //temporary Variables to hold arguments  
 Variable arg1 = *arithmetic*(n1);  
 Variable arg2 = *arithmetic*(n2);  
  
 if(arg1 == null || arg2 == null){  
 *createError*("invalid argument", "", node.getFirstToken().getLine());  
 return false;  
 }  
  
 //store variable values  
 int val1 = Integer.*parseInt*(arg1.getValue());  
 int val2 = Integer.*parseInt*(arg2.getValue());  
  
 //perform expression  
 switch(op.getFirstChild().getGrammar()){  
 case "le\_operator":  
 if(val1 <= val2)  
 return true;  
 else  
 return false;  
 case "lt\_operator":  
 if(val1 < val2)  
 return true;  
 else  
 return false;  
 case "ge\_operator":  
 if(val1 >= val2)  
 return true;  
 else  
 return false;  
 case "gt\_operator":  
 if(val1 > val2)  
 return true;  
 else  
 return false;  
 case "eq\_operator":  
 if(val1 == val2)  
 return true;  
 else  
 return false;  
 case "ne\_operator":  
 if(val1 != val2)  
 return true;  
 else  
 return false;  
 default:  
 *createError*("unexpected operation", op.getFirstToken().getLexeme(), op.getFirstToken().getLine());  
 return false;  
 }  
  
 }  
  
 //<arithmetic\_expression> -> <id> | <literal\_integer> | <arithmetic\_expression> <arithmetic\_op> <arithmetic\_expression>  
 //returns a Variable (if no errors occurred) or null (if errors occurred)  
 private static Variable arithmetic(Node node){  
 //stores result of method  
 Variable var;  
  
 //if statement chooses what to do based on the number of children node has  
 //if node has either 0-1 children the node is either a variable identifier of integer  
 //node has 0 children  
 if(node.getChildren().size() == 0){  
 //if node is an identifier  
 if(node.getGrammar().equals("id")){  
 //ensure the variable has been initialized  
 var = *varsContains*(node.getFirstToken().getLexeme());  
 if(var != null) {  
 //variable was initialized  
 return var;  
 } else {  
 //variable was not initialized  
 *createError*("variable not found", node.getFirstToken().getLexeme(), node.getFirstToken().getLine());  
 return null;  
 }  
 //if node is an integer  
 } else if(node.getGrammar().equals("literal\_integer")){  
 return new Variable("temp", node.getFirstToken().getLexeme());  
 //if node is neither identifier or integer  
 } else {  
 *createError*("incorrect value", node.getFirstToken().getLexeme(), node.getFirstToken().getLine());  
 return null;  
 }  
 //node has 1 child  
 } else if(node.getChildren().size() == 1){  
 Node nChild = node.getFirstChild();  
 if(nChild.getGrammar().equals("id")){  
 //ensure the variable has been initialized  
 var = *varsContains*(nChild.getFirstToken().getLexeme());  
 if(var != null) {  
 //variable was initialized  
 return var;  
 } else {  
 //variable was not initialized  
 *createError*("variable not found", nChild.getFirstToken().getLexeme(), nChild.getFirstToken().getLine());  
 return null;  
 }  
 //if node is an integer  
 } else if(nChild.getGrammar().equals("literal\_integer")){  
 return new Variable("temp", nChild.getFirstToken().getLexeme());  
 //if node is neither identifier or integer  
 } else {  
 *createError*("incorrect value", nChild.getFirstToken().getLexeme(), nChild.getFirstToken().getLine());  
 return null;  
 }  
 //node has 3 children  
 //expanded arithmetic expression  
 } else if(node.getChildren().size() == 3){  
 //get child nodes  
 Node n1 = node.getFirstChild();  
 Node op = node.getChild(1);  
 Node n2 = node.getChild(2);  
  
 //initialize arguments as temporary variables  
 Variable arg1 = *arithmetic*(n1);  
 Variable arg2 = *arithmetic*(n2);  
  
 if(arg1 == null || arg2 == null){  
 *createError*("invalid argument", "", node.getFirstToken().getLine());  
 return null;  
 }  
  
 return *operations*(op, arg1, arg2);  
 }  
 //return null if method gets here, an error probably occurred  
 *createError*("arithmetic expression error");  
 return null;  
 }  
  
 //performs arithmetic operations  
 //takes in node op(holds operation), and Variable arg1 and Variable arg2 (expressions arguments)  
 //returns Variable object that stores result of method  
 private static Variable operations(Node op, Variable arg1, Variable arg2){  
 //store Variable values for convenience  
 int val1 = Integer.*parseInt*(arg1.getValue());  
 int val2 = Integer.*parseInt*(arg2.getValue());  
 String result = "";  
 //perform arithmetic operation  
 switch (op.getGrammar()) {  
 case "division\_operator":  
 //cannot divide by zero  
 if(val2 != 0){  
 result = String.*valueOf*(val1 / val2);  
 break;  
 } else  
 *createError*("cannot divide by zero", arg2.getValue(), op.getFirstToken().getLine());  
 break;  
 case "multiplication\_operator":  
 result = String.*valueOf*(val1 \* val2);  
 break;  
 case "addition\_operator":  
 result = String.*valueOf*(val1 + val2);  
 break;  
 case "subtraction\_operator":  
 result = String.*valueOf*(val1 - val2);  
 break;  
 default:  
 //operation not found  
 *createError*("incorrect operation", op.getFirstToken().getLexeme(), op.getFirstToken().getLine());  
 return null;  
 }  
 //return temporary variable that holds result  
 return new Variable("temp", result);  
 }  
  
 public static void main(String args[]){  
 File f = new File("src/Julia-Files/Test3.jl");  
 *getGlobals*(f);  
 if(*nodes* != null && *root* != null) {  
 *interpret*();  
  
 if(*errorOccurred*){  
 System.*out*.println("Interpreter Errors:");  
 for(Error e: *errors*){  
 e.printError();  
 }  
 } else {  
 System.*out*.println("Lexical Analyzer Results\nSymbol Table:");  
  
 System.*out*.println("\nParser Results\nAST:");  
 Parser.*printTree*(*root*, *nodes*);  
  
 System.*out*.println("\nInterpreter Results:\nOutput of Source Code File:");  
 System.*out*.println(*output*);  
 }  
 } else if(*nodes* == null || *tokens* == null){  
 if(*nodes* == null)  
 *createError*("parser error");  
 }  
 }  
}

# Code Review

## Interpreter.java

### Variable

Variable is a nested class within Interpreter.java. Each Variable has two String variables: name and value. The String name stores the id or name that the Interpreter knows the Variable by. value stores the value associated with the Variable.

It has a singular constructor that takes in two Strings (name and value) as parameters.

It also includes three methods:

* setID(String name)
  + setID() changes the Variables name.
* getID()
  + This method returns the name of the Variable.
* getValue()
  + This returns the value associated with the Variable.

### Global variables

* root
  + This Node object stores the root Node of the AST generated by Parser.java.
* nodes
  + This is an ArrayList of Node objects. It stores all the nodes generated by Parser.java.
* tokens
  + tokens is an ArrayList that stores all Token objects generated by LexicalAnalyzer.java
* errors
  + This ArrayList stores all Error objects generated by the Interpreter. The Errors generated by LexicalAnalyzer.java and Parser.java are not included here.
* vars
  + This is an ArrayList of Variable objects which stores all Variables initialized by the Interpreter.
* errorOccurred
  + A Boolean value that serves as a check for whether an error has occurred during execution.
* output
  + output is a String that stores everything printed by the Interpreter.

### Methods

* createError(String msg) & createError(String msg, String value, int line)
  + These methods create an Error object to add to errors if an error is true. It also changes errorOccurred to true.
* varsContains(String id)
  + This method is used to check if a variable has already been created. If the Variable is found in vars, the Variable has already been created and the method returns the Variable found. If the Variable was not found in vars, the Variable has not been created and the method returns null. It takes in a String id, which is the id (or name) of the Variable that the method is looking for.
* getVarIndex(String id)
  + This method takes in String id that represents the name of the Variable (what the Variable is stored as), and searches for the Variable in vars. It returns the index of the Variable in vars or -1 otherwise.
* getGlobals(File f)
  + This method gets some of the globals for the Interpreter. It gets root and nodes from Parser.java, using getRootNode() and getNode() respectively. It also sets tokens using getTokenList() from LexicalAnalyzer.java. It takes in a File object (called f), that stores the source code file.
* addVariable(Variable v, Node n)
  + This method adds a Variable to vars if the variable doesn’t already exist. If the variable already exists it creates an Error (“variable already exists”).
* interpret()
  + This is the main driver of the Interpreter. It starts by ensuring that the root was initialized correctly. If root is null or if its children ArrayList is empty, the root was not initialized correctly and an Error is created (“root is null”, “root has 0 children”).
  + If the root was initialized correctly, the interpreter begins. A Node called start is used to hold the child Node of the root that is being interpreted and an integer, startChild, is used to store the child Node’s index from the root’s children ArrayList.
  + A while loop iterates through root’s children if start does not have a EOF TokenType and an Error has not occurred. For each loop, block() is called with start’s first child. After the block() call, startChild is increased by 1 and if startChild is less than the number of root’s children, start is changed to the next child of root.
* block(Node node)
  + block() takes in a “<block>” Node (a Node object with a grammar of “<block>”). A “<block>” Node should only have a single child Node called “<statement>”. A “<statement>” Node should only have a single child Node with grammar based on the statement type.
  + The method begins with a while loop. The loop will iterate if the passed Node (node) has a grammar of “<block>” or “<statement>”. For each loop, node is set equal to its first child (node.getFirstChild()).
  + After the loop, a switch statement begins. Using the grammar of node, the switch statement determines which statement method to call : assignI(), printI(), ifI(), whileI(), repeatI(). (I added I to the end of the methods to help with identification: these methods belong to the Interpreter.
* assignI(Node node)
  + assignI() takes in an “<assignment\_statement>” Node. It should have three child Nodes:
    - an “id” Node (left hand side of assignment statement; the variable the value is being assigned to)
    - an “operation” Node (stores the assignment operator (=) or addition assignment operator (+=))
    - an “arithmetic expression” Node (stores the arithmetic expression on the right side of the operator).
  + assignI() starts by giving each of node’s children a variable to store it in (for ease of writing). The variable is stored in var, the operation is stored in op, and the arithmetic expression is stored in expr.
  + A temporary variable, called tempVar, is created and its value is a returned Variable object from varContains(), passing var as its parameter.
  + If tempVar equals null the variable does not yet exist, so we create a new Variable and store it in tempVar. We use the lexeme of the var node as the name of the Variable, and we use an empty string for the value as a temporary place holder. Otherwise, the Variable exists and, at this point, should be stored in tempVar.
  + Then the method creates a new Variable called arithExpr that holds the return value of arithmetic(). We pass the expr Node as its parameter. After the arithmetic() call, arithExpr’s value should hold the value that will be assigned to the Variable. We set the id with setID() using the Id of tempVar. Then, based on the TokenType of the op Node, the assignment statement is performed.
    - If it is a normal assignment operator (=), we set the Variable in vars equal to arithExpr.
    - If it is an addition assignment operator (+=), we create a new integer value called result, that stores the result of the assignment operation (essentially we add the Variable’s current value to arithExpr value). Then we set the Variable in var equal to a new Variable with the same id and result as the value.
* printI(Node node)
  + printI() takes in an “<assignment\_statement>” Node. In general \*, it should have 4 child Nodes:
    - A Node to hold the “print” keyword.
    - Two Nodes to hold both parentheses.
    - A Node to hold the arithmetic expression.

\* I say ‘in general’ because there is a case where print only has one child, the arithmetic expression.

* + The first thing this method does is create a Node object to pass to arithmetic(). It should store the print statement’s arithmetic expression. It’s called printExpr. We get the arithmetic expression from the print statement Node’s children. If there is only one child, printExpr is set equal to that. Otherwise, the child with an index of 2 is the arithmetic expression.
  + We call arithmetic() with printExpr and store the returned Variable in a new Variable called printVar. printVar is a temporary variable used to store the resulting value from arithmetic. We add the value variable of printVar to the global String output (along with a new line character) and the method ends.
* ifI(Node node)
  + ifI() takes in an “<if\_statement>” Node. It should have 6 children:
    - A Node that holds the “if” keyword.
    - A Node that stores the Boolean expression for the statement.
    - Two Nodes to store the blocks of the if statement.
    - A Node to store the “else” keyword.
    - A Node to store the “end” keyword.
  + The method begins by getting all of the relevant children and setting them equal to Nodes for writability. The Boolean expression is stored in boolExpr. The first block is stored in thenBlock and the second block statement is stored in elseBlock.
  + The we call bool(), passing boolExpr as its parameter. We store the resulting Boolean value in boolValue. This stores the result of the if statement’s Boolean expression. If bool() returns true and an error has not occurred we call block() with thenBlock. If bool() returns false and an error has not occurred we call block() with elseBlock.
* whileI(Node node)
  + whileI() takes in a “<while\_statement>” Node. It should have 4 children:
    - A Node to store the “while” keyword.
    - A Node to store the Boolean expression.
    - A Node to store the block().
    - A Node that stores the “end” keyword.
  + The method begins by storing all needed Nodes in easily accessible Nodes. The Boolean expression is stored in boolExpr and the block statement is stored in doBlock. Next a Boolean variable called boolValue is created that stores the result of the bool() call. Then a while loop starts. The while loop iterates if boolValue is true and an error has not occurred. When the loop iterates block() is called with doBlock as its parameter. Then the Boolean expression is evaluated again (with boolExpr) and the result is stored in boolValue.
* repeatI(Node node)
  + repeatI() is called with a “<repeat\_statement>” Node as its parameter. It should have 4 children:
    - A Node to store the “repeat” keyword.
    - A Node to store the “until” keyword.
    - A Node to store the Boolean expression.
    - A Node that stores the block statement.
  + The method starts by storing the Boolean expression and block statement in easy to access Node objects. The Boolean expression is stored in boolExpr and the block statement is stored in repeatBlock.
  + The method creates a Boolean called boolValue that is initially set to false. Then the while statement begins. It will loop as long as boolValue is false and an error has not occurred. Each time it loops it calls block() with repeatBlock and sets boolValue equal to the result of the Boolean expression (bool(boolExpr)).
* Bool(Node node)
  + Bool() is called with a “<Boolean\_expression>” Node. Each should have 3 children:
    - Two Nodes to store the arithmetic expressions.
    - A Node to store the relative operation.
  + First, bool() stores each child in an easy to access Node object. The operation is stored in op, the first arithmetic expression is stored in n1, and the second arithmetic expression is stored in n2.
  + Then two Variables are created. They represent the results of each arithmetic expression. Variable arg1 stores the result of arithmetic() called with n1 as its parameter and Variable arg2 stores the result of arithmetic() called with n2 as its parameter. If one of these arguments is null, it means that that argument is a variable that was not initialized. For that argument, an error is created (“invalid argument”) and the method returns false.
  + Then two integers are created that store the value from each of the argument Variables, after typecasting them to integers.
  + Then a switch statement performs the operation. It chooses the operation based on the grammar of the operation. If the operation is not valid, an error is created (“unexpected operation”) and the method returns false.
  + The method should return true or false depending on the outcome of the Boolean expression.
* arithmetic(Node node)
  + arithmetic() is called with an “<arithmetic\_expression>” Node that can have 0, 1, or 3 children.
    - If the arithmetic expression has three children they are as follows:
      * Two Nodes that store arithmetic expressions.
      * A Node that stores the arithmetic operation for the expression.
    - If the arithmetic expression has 0 or 1 child it is either an identifier or literal integer.
  + This method returns a temporary Variable that stores the result of the arithmetic expression.
  + Arithmetic() is made up of nested if statements. If the passed node has either 0 or 1 child:
    - The grammar of the node (0 children) or the grammar of node’s first child (1 child) is checked.
      * If the Node stores a variable identifier, vars is checked to ensure that it was initialized using varsContains(). If the variable is not found (varsContains() returns null) an error is created (“variable not found”) and null is returned. If the variable does exist, the result of varsContains() is returned.
      * If the Node stores an integer, arithmetic() returns a temporary Variable that stores the value of the integer with a placeholder name.
  + If the passed node has 3 children:
    - Each child is stored in an easy to access Node variable. n1 stores the first arithmetic expression Node, n2 stores the second arithmetic expression Node, and the Node op stores the operation of the arithmetic expression. Arithmetic() is called for both of the arithmetic expressions (n1 and n2) and the resulting Variables are stored in arg1 and arg2. If either of those is null at this point, they are an invalid argument; an error is created (“invalid argument”) and null is returned. Otherwise, the method returns the result of operations() called with op, arg1, and arg2 as its parameters.
  + If the method reaches the end without returning, some kind of error occurred so an error is created (“arithmetic expression error”) and null in returned.
* operations(Node op, Variable arg1, Variable arg2)
  + This method performs arithmetic operations. It is called from arithmetic(). It is passed three variables:
    - A Node called op that stores the operation of the arithmetic expression.
    - Two Variable Nodes (arg1 and arg2) that store the expressions arguments.
  + It should return a temporary Variable with a name of “temp” and a value that stores the result of the arithmetic operation.
  + operations() begins by initializing three variables. val1 and val2 will store the values of arg1 and arg2 respectively. The String result will store the result of the expression.
  + A switch statement chooses the operation to perform based on the grammar of op.
* Main
  + The main method just sets up for the Interpreter. First a File object is created (called f) that holds the source code file for the Interpreter.
  + Then getGlobals() is called with f as its parameter. This call sets root, nodes, and tokens. If none of them are null, the interpreter is called. Then if errors did not occur, the main prints out the output of the Lexical Analyzer, Parser, and the result of the source code (Interpreter.ouput).
  + If errors occurred, they are printed out.

# Lexical Analyzer and Parser Changes

I had to make a few changes to my Lexical Analyzer and Parser.

First, I encountered an error when I created the Interpreter project and ported in the other files. For some reason the enumerated constants I used for my switch statements threw an error. For example, in the statement() method in Parser.java I had to change Token.TokenType.LETTER to LETTER or else the program wouldn’t run.

The other issue was the problems with the Julia files. I edited both LexicalAnalyzer.java and Parser.java to support the fixed files rather than the old ones. I essentially changed the syntax of the expression processing from prefix to infix.

Lastly, I had to add a few methods to Node.java and Parser.java so the Interpreter could get information.

To Node.java I added:

* getChild(int index)
  + Returns the child Node at the index in children.
* getFirstToken()
  + Returns the first Token object from the tokens ArrayList.
* getFirstChild()
  + Returns the first Node object stored in the children ArrayList.

To Parser.java I added:

* getRootNode(File f)
  + This method takes in the source code file and returns the root Node of an AST. This is essentially a callable main for Parser.java.
* getNodes()
  + This method returns the nodes ArrayList.

# Conclusion

This section of the report is more for future me than for the assignment. It is a collection of some thoughts I had on the project, limitations of the current version, and possible improvements I could make in the future.

One big issue I discovered while testing my Interpreter was that it can’t handle multiple line blocks. For example, if either a while, if, or repeat expression has more than one line in its block, the Interpreter will throw an error. I think this can be easily fixed with a while loop in the Parser that checks for the “end” keyword, but I didn’t think I had enough time to implement it.

Another problem is that there is (as far as I can tell) no order of operations or precedence rules for operations in place. (If there, are it was by accident.) I’m not sure yet how to do this.

I also feel like there are many redundancies in the code. Either places where I test for the same error multiple times, or places where I create an error that can never be reached. When one Error occurs, multiple Errors are printed. I just feel like, particularly for the Parser and Interpreter, I coded the same process twice. There is a part of me that want to go back through them and optimize them.

Lastly, I think I can merge some of the support files (Node and Token in particular).

# References

“Java Convert Int to String - Javatpoint.” *Www.javatpoint.com*, https://www.javatpoint.com/java-int-to-string.

“Java Convert String to Int - Javatpoint.” *Www.javatpoint.com*, https://www.javatpoint.com/java-string-to-int.