10/31/2021

Faith Swetnam

Kennesaw State University

Faith Swetnam

Fall 2021

CS 4308/Section 03

1st Deliverable – Parser

Professor Sharon Perry

100% complete and working as designed

Table of Contents

[Source Code 2](#_Toc86531201)

[Parser.java 2](#_Toc86531202)

[Node.java 14](#_Toc86531203)

[Code Overview 15](#_Toc86531204)

[Parser.java 15](#_Toc86531205)

[Variables 15](#_Toc86531206)

[Methods 16](#_Toc86531207)

[Node.java 21](#_Toc86531208)

[Lexical Analyzer Changes 22](#_Toc86531209)

[Overview 22](#_Toc86531210)

[Source Code 22](#_Toc86531211)

[Lexical\_Analyzer.java 22](#_Toc86531212)

[Token.java 31](#_Toc86531213)

[Screenshots (fix bool errors) 33](#_Toc86531214)

[Successful runs 33](#_Toc86531215)

[Test1.jl 33](#_Toc86531216)

[Test2.jl 34](#_Toc86531217)

[Test3.jl 35](#_Toc86531218)

[Test4.jl 36](#_Toc86531219)

[Some Error Checking (with errors placed in source code) 37](#_Toc86531220)

[Program start errors 37](#_Toc86531221)

[assignment statement errors 38](#_Toc86531222)

[if statement errors 40](#_Toc86531223)

[while statement errors 43](#_Toc86531224)

[repeat statement errors 45](#_Toc86531225)

[print statement errors 46](#_Toc86531226)

[References 48](#_Toc86531227)

# Source Code

## Parser.java

/\*  
Class: CS 4308 Section 03  
Term: Fall 2021  
Name: Faith Swetnam  
Instructor: Sharon Perry  
Project: Deliverable 2 Parser  
 \*/  
  
import java.io.File;  
import java.util.ArrayList;  
  
public class Parser {  
 private static ArrayList<Node> *nodes* = new ArrayList<Node>(); //stores all nodes produced  
 private static ArrayList<Integer> *opcodes* = new ArrayList<Integer>(); //stores opcodes to be compared against  
 private static ArrayList<Error> *errors* = new ArrayList<Error>(); //stores Errors generated  
 private static ArrayList<Token> *tokens* = new ArrayList<Token>(); //stores tokens generated by the scanner  
 private static ArrayList<Token.TokenType> *words* = new ArrayList<Token.TokenType>(); //stores reserved words/keywords  
 private static Token *nextToken*; //stores the current token being processed  
 private static Token *prevToken*; //stores the previous token processed  
 private static Node *rootNode* = null; //stores the root node of the AST  
 private static Node *currNode* = new Node(); //stores the current AST node  
 private static Node *prevNode* = new Node(); //stores the parent of the current AST node  
 private static int *tokenCount* = 0; //stores the current index of the token arraylist  
 private static boolean *errorOccurred* = false; //stores whether an Error has been generated  
  
 //fills the opcodes arraylist with integer opcodes  
 private static void fillOpcodes(){  
 *opcodes*.add(0);*opcodes*.add(1);*opcodes*.add(2);*opcodes*.add(99);*opcodes*.add(1000);*opcodes*.add(1001);  
 *opcodes*.add(1002);*opcodes*.add(1003);*opcodes*.add(1004);*opcodes*.add(1005);*opcodes*.add(1006);*opcodes*.add(1007);  
 *opcodes*.add(1008);*opcodes*.add(1009);*opcodes*.add(2000);*opcodes*.add(2001);*opcodes*.add(2002);*opcodes*.add(2003);  
 *opcodes*.add(2004); *opcodes*.add(2005);*opcodes*.add(2006);*opcodes*.add(2007);*opcodes*.add(2008);*opcodes*.add(2009);  
 *opcodes*.add(2010);*opcodes*.add(2011);*opcodes*.add(2012);*opcodes*.add(2013);  
 }  
  
 //fills the words arraylist with TokenType reserved words and keywords  
 private static void fillWords(){  
 *words*.add(Token.TokenType.*WHILE*);*words*.add(Token.TokenType.*ELSE*);*words*.add(Token.TokenType.*DO*);  
 *words*.add(Token.TokenType.*IF*);*words*.add(Token.TokenType.*PRINT*);*words*.add(Token.TokenType.*THEN*);  
 *words*.add(Token.TokenType.*END*);*words*.add(Token.TokenType.*EOF*);*words*.add(Token.TokenType.*THEN*);  
 *words*.add(Token.TokenType.*L\_PAREN*);*words*.add(Token.TokenType.*R\_PAREN*);*words*.add(Token.TokenType.*FUNCT*);  
 }  
  
 //checks all the opcodes for validity  
 private static void checkOpcodes(){  
 for(Token t: *tokens*){  
 //if opcode is not valid  
 if(!*opcodes*.contains(t.getType().opcode)){  
 //generate an error  
 *createError*("incorrect opcode", "Opcode: " + String.*valueOf*(t.getType().opcode), t.getLine());  
 }  
 }  
 }  
  
 //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, String value, int line){  
 Error e = new Error(msg, value, line);  
 *errors*.add(e);  
 *errorOccurred* = true;  
 }  
  
 //gets the next token from tokens arraylist  
 private static void getToken(){  
 try {  
 if(*nextToken* != null){  
 *prevToken* = *nextToken*;  
 }  
 *nextToken* = *tokens*.get(*tokenCount*);  
 *tokenCount*++;  
 if(*nextToken*.getType() == Token.TokenType.*ERROR*){  
 *createError*("scanner error", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 }  
 } catch (IndexOutOfBoundsException e){  
 *createError*("index out of bounds", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 }  
 }  
  
 //creates a node object and adds it to the parent's children arraylist  
 private static void createNode(Token token, Node root, Node parent, String grammar){  
 //if the root has not been initialized  
 if(*rootNode* == null){  
 *rootNode* = new Node(token, grammar);  
 *currNode* = *rootNode*;  
 *nodes*.add(*rootNode*);  
 //find the parent to assign the node to  
 } else {  
 if (root == parent) {  
 *prevNode* = *currNode*;  
 *currNode* = new Node(token, root, grammar);  
 root.addChild(*currNode*);  
 *nodes*.add(*currNode*);  
 return;  
 } else {  
 int len = root.getChildren().size();  
 for (int i = 0; i < len; i++) {  
 *createNode*(token, root.getChildren().get(i), parent, grammar);  
 }  
 }  
 }  
 }  
  
 //prints the AST  
 //specifically prints all Node objects in nodes and each of their children  
 private static void printTree(){  
 if(*rootNode* == null) {  
 return;  
 } else {  
 for(Node node: *nodes*){  
 if(node.getChildren().isEmpty()){  
 System.*out*.println(node.getGrammar());  
 } else {  
 System.*out*.print(node.getGrammar() + " -> ");  
 for(Node child: node.getChildren()){  
 System.*out*.print(child.getGrammar() + " ");  
 }  
 }  
 System.*out*.println();  
 }  
 }  
 }  
  
 //main driver function of the parser  
 // <program> -> function id() <block> end  
 private static void parse(){  
 *getToken*(); //get first token  
 *createNode*(*nextToken*, null, null, "<program>");  
 if(*nextToken*.getType() == Token.TokenType.*FUNCT*){  
 *getToken*(); //process "function" keyword  
 if(*nextToken*.getType() != Token.TokenType.*LETTER*){  
 *createError*("function identifier expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 } else {  
 *currNode*.addToken(*nextToken*);  
 *getToken*(); //process function identifier  
 if(*nextToken*.getType() != Token.TokenType.*L\_PAREN*){  
 *createError*("( expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 } else {  
 *currNode*.addToken(*nextToken*);  
 *getToken*(); //process (  
 if(*nextToken*.getType() != Token.TokenType.*R\_PAREN*){  
 *createError*(") expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 } else {  
 *currNode*.addToken(*nextToken*);  
 *getToken*(); //process )  
 }  
 }  
 }  
 } else {  
 *createError*("function expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 }  
  
 //while the program hasn't ended, and the TokenType of nextToken isn't NULL or ERROR...  
 while(*nextToken*.getType() != Token.TokenType.*EOF* && *nextToken*.getType() != Token.TokenType.*ERROR* && *nextToken*.getType() != Token.TokenType.*NULL*){  
 *block*(*rootNode*);  
 if(*errorOccurred*){  
 break;  
 }  
 //if the program hasn't ended, and the TokenType of nextToken isn't NULL or ERROR get the next Token  
 if(*nextToken*.getType() != Token.TokenType.*EOF* && *nextToken*.getType() != Token.TokenType.*ERROR* && *nextToken*.getType() != Token.TokenType.*NULL*)  
 *getToken*();  
  
 }  
  
 //if the program has ended add a new node to the end of tree (add to rootNode)  
 if(*nextToken*.getType() == Token.TokenType.*EOF* && !*errorOccurred*){  
 *createNode*(*nextToken*, *rootNode*, *rootNode*, "end");  
 } else {  
 //if "end" keyword is not encountered  
 *createError*("EOF expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 }  
 }  
  
 //block  
 //takes in the parent Node(essentially the function that called it...)  
 // <block> -> <statement> | <statement> <block>  
 private static void block(Node parent) {  
 //if the program hasn't ended, and the TokenType of nextToken isn't NULL, ERROR, or END...  
 if (*nextToken*.getType() != Token.TokenType.*EOF* && *nextToken*.getType() != Token.TokenType.*ERROR* && *nextToken*.getType() != Token.TokenType.*NULL* && *nextToken*.getType() != Token.TokenType.*END*) {  
 //create a block Node and call statement function  
 *createNode*(*nextToken*, *rootNode*, parent, "<block>");  
 Node blockNode = *currNode*;  
 *statement*(blockNode);  
 }  
 }  
  
 //statement expression  
 //takes in its parent Node  
 // <statement> -> <if\_statement> | <assignment\_statement> | <while\_statement> | <print\_statement> | <repeat\_statement>  
 private static void statement(Node parent){  
 //create a statement Node  
 *createNode*(*nextToken*, *rootNode*, parent, "<statement>");  
 Node statementNode = *currNode*;  
 //if the program hasn't ended, and the TokenType of nextToken isn't NULL, ERROR, or END...  
 if(*nextToken*.getType() != Token.TokenType.*EOF* && *nextToken*.getType() != Token.TokenType.*ERROR* && *nextToken*.getType() != Token.TokenType.*NULL* && *nextToken*.getType() != Token.TokenType.*END*){  
 switch(*nextToken*.getType()){  
 case *LETTER*:  
 *assignState*(statementNode);  
 break;  
 case *IF*:  
 *ifState*(statementNode);  
 break;  
 case *WHILE*:  
 *whileState*(statementNode);  
 break;  
 case *PRINT*:  
 *printState*(statementNode);  
 break;  
 case *REPEAT*:  
 *repeatState*(statementNode);  
 break;  
 default:  
 *createError*("unexpected statement", *nextToken*.getLexeme(), *nextToken*.getLine());  
 break;  
 }  
 }  
 }  
  
 //statement  
 //takes in parent Node  
 // <assignment\_statement> -> id <assignment\_operator> <arithmetic\_expression>  
 private static void assignState(Node parent){  
 //create assignment\_statement Node  
 *createNode*(*nextToken*, *rootNode*, parent, "<assignment\_statement>");  
 Node assignNode = *currNode*;  
 // add id to assignment\_statement Node as a new Node  
 // if it is a character identifier (length=1)  
 if(*nextToken*.getType() == Token.TokenType.*LETTER* && *nextToken*.getLexeme().length() == 1  
 && !*words*.contains(*nextToken*.getType())) {  
 assignNode.addChild(new Node(*nextToken*, assignNode, "id"));  
 *getToken*();  
 } else {  
 *createError*("identifier expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 }  
 *arithOp*(assignNode); //calls artihOp() to process expression and operator  
 }  
  
 //if statement  
 //takes in parent Node  
 // <if\_statement> -> if <boolean\_expression> then <block> else <block> end  
 private static void ifState(Node parent){  
 //create if\_statement Node  
 *createNode*(*nextToken*, *rootNode*, parent, "<if\_statement>");  
 Node ifNode = *currNode*;  
 //add 'if' keyword as new Node to ifNode  
 ifNode.addChild(new Node(*nextToken*, ifNode, "if"));  
 *getToken*(); //process 'if'  
 *boolExpr*(ifNode); //boolExpr() call  
 *getToken*(); //process end of boolEpr() and get 'then'  
 if(*nextToken*.getType() == Token.TokenType.*THEN*){  
 //add 'then' keyword as new Node to ifNode  
 ifNode.addChild(new Node(*nextToken*, ifNode, "then"));  
 *getToken*(); //process 'then'  
 *block*(ifNode); //first block()  
 if(*nextToken*.getType() == Token.TokenType.*ELSE*){  
 //add 'else' keyword as new Node to ifNode  
 ifNode.addChild(new Node(*nextToken*, ifNode, "else"));  
 *getToken*(); //process 'else'  
 *block*(ifNode); //second block  
 if(*nextToken*.getType() == Token.TokenType.*END*){  
 //add 'end' keyword as new Node to ifNode  
 ifNode.addChild(new Node(*nextToken*, ifNode, "end"));  
 *getToken*(); //process 'end'  
 return;  
 } else {  
 *createError*("end expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 }  
 } else {  
 *createError*("else expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 }  
 } else {  
 *createError*("then expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 }  
 }  
  
 //while statement  
 //takes in parent Node  
 // <while\_statement> -> while <boolean\_expression> do <block> end  
 private static void whileState(Node parent){  
 //create while\_statement Node  
 *createNode*(*nextToken*, *rootNode*, parent, "<while\_statement>");  
 Node whileNode = *currNode*;  
 //add 'while' keyword as new Node to whileNode  
 whileNode.addChild(new Node(*nextToken*, whileNode, "while"));  
 *getToken*(); //process 'while'  
 *boolExpr*(whileNode); //boolExpr() call  
 *getToken*(); //process end of boolExpr() get 'do'  
 if(*nextToken*.getType() == Token.TokenType.*DO*){  
 //add 'do' keyword as new Node to whileNode  
 whileNode.addChild(new Node(*nextToken*, whileNode, "do"));  
 *getToken*(); //process 'do'  
 *block*(whileNode); //block() call  
 *getToken*(); // process end of block  
 if(*nextToken*.getType() == Token.TokenType.*END*){  
 //add 'end' keyword as new Node to whileNode  
 whileNode.addChild(new Node(*nextToken*, whileNode, "end"));  
 } else {  
 *createError*("end expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 }  
 } else {  
 *createError*("do expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 }  
 }  
  
 //repeat statement  
 //takes in parent Node  
 // <repeat\_statement> -> repeat <block> until <boolean\_expression>  
 private static void repeatState(Node parent){  
 //create repeat\_statement node  
 *createNode*(*nextToken*, *rootNode*, parent, "<repeat\_statement>");  
 Node repeatNode = *currNode*;  
 //add 'repeat' keyword as new Node to repeatNode  
 repeatNode.addChild(new Node(*nextToken*, repeatNode, "repeat"));  
 *getToken*(); //process 'repeat'  
 *block*(repeatNode); //block() call  
 *getToken*(); //process end of block  
 if (*nextToken*.getType() == Token.TokenType.*UNTIL*) {  
 //add 'until' keyword as new Node to repeatNode  
 repeatNode.addChild(new Node(*nextToken*, repeatNode, "until"));  
 *getToken*(); //process until  
 *boolExpr*(repeatNode); //boolExpr() call  
 return;  
 } else {  
 *createError*("until expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 }  
 }  
  
 //boolean expression  
 //takes in parent Node  
 // <boolean\_expression> -> <relative\_op> <arithmetic\_expression> <arithmetic\_expression>  
 private static void boolExpr(Node parent){  
 //create repeat Node  
 *createNode*(*nextToken*, *rootNode*, parent, "<boolean\_expression>");  
 Node boolNode = *currNode*;  
 //check next Token is operation  
 if(!*words*.contains(*nextToken*.getType())){  
 *relOp*(boolNode); //call relOp()  
 *getToken*(); //process end of relOp()  
 //check next Token is LETTER or DIGIT  
 if(*nextToken*.getType() == Token.TokenType.*LETTER* || *nextToken*.getType() == Token.TokenType.*DIGIT*){  
 *arithExpr*(boolNode); //call first arithmetic\_expression  
 *getToken*(); // process end of arithmetic\_expression  
 //check next Token is LETTER or DIGIT  
 if(*nextToken*.getType() == Token.TokenType.*LETTER* || *nextToken*.getType() == Token.TokenType.*DIGIT*){  
 *arithExpr*(boolNode); //call second arithmetic\_expression  
 } else {  
 *createError*("unexpected Token", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 }  
 } else {  
 *createError*("unexpected Token", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 }  
 } else {  
 *createError*("unexpected operation", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 }  
 }  
  
 //relative operations  
 //takes in parent Node  
 // <relative\_operation> -> le\_operator | lt\_operator | ge\_operator | gt\_operator | eq\_operator | ne\_operator  
 private static void relOp(Node parent){  
 //create relative\_operation Node  
 *createNode*(*nextToken*, *rootNode*, parent, "<relative\_op>");  
 Node relNode = *currNode*;  
 //switch statement determines which operatio nextToken is and adds it as a new child Node to relative\_operation Node  
 switch(*nextToken*.getType()){  
 case *LE\_OP*:  
 *currNode*.addChild(new Node(*nextToken*, relNode, "le\_operator"));  
 break;  
 case *LT\_OP*:  
 *currNode*.addChild(new Node(*nextToken*, relNode, "lt\_operator"));  
 break;  
 case *GE\_OP*:  
 *currNode*.addChild(new Node(*nextToken*, relNode, "ge\_operator"));  
 break;  
 case *GT\_OP*:  
 *currNode*.addChild(new Node(*nextToken*, relNode, "gt\_operator"));  
 break;  
 case *EQ\_OP*:  
 *currNode*.addChild(new Node(*nextToken*, relNode, "eq\_operator"));  
 break;  
 case *NE\_OP*:  
 *currNode*.addChild(new Node(*nextToken*, relNode, "ne\_operator"));  
 break;  
 //if nextToken is not an operation an Error is generated  
 default:  
 *createError*("unexpected operation", *nextToken*.getLexeme(), *nextToken*.getLine());  
 break;  
 }  
 }  
  
 //arithmetic\_operation  
 //takes in parent Node  
 // <arith\_op> -> add\_operator | sub\_operator | mul\_operator | div\_operator  
 // I also added the assignment\_operator and the addition\_assignment operator  
 private static void arithOp(Node parent){  
 switch(*nextToken*.getType()){  
 case *ASSIGN\_OP*:  
 parent.addChild(new Node(*nextToken*, parent, "assignment\_operator"));  
 *getToken*();  
 if(*nextToken*.getType() == Token.TokenType.*DIGIT* || *nextToken*.getType() == Token.TokenType.*LETTER*){  
 *arithExpr*(parent);  
 } else {  
 *createError*("identifier or literal\_integer expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 }  
 break;  
 case *AE\_OP*:  
 parent.addChild(new Node(*nextToken*, parent, "addition\_assignment"));  
 *getToken*();  
 if(*nextToken*.getType() == Token.TokenType.*DIGIT* || *nextToken*.getType() == Token.TokenType.*LETTER*){  
 *arithExpr*(parent);  
 } else {  
 *createError*("identifier or literal\_integer expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 }  
 break;  
 case *DIV\_OP*:  
 parent.addChild(new Node(*nextToken*, parent, "division\_operation"));  
 *getToken*();  
 if(*nextToken*.getType() == Token.TokenType.*DIGIT* || *nextToken*.getType() == Token.TokenType.*LETTER*){  
 *arithExpr*(parent);  
 *getToken*();  
 if(*nextToken*.getType() == Token.TokenType.*DIGIT* || *nextToken*.getType() == Token.TokenType.*LETTER*){  
 *arithExpr*(parent);  
 } else {  
 *createError*("identifier or literal\_integer expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 }  
 } else {  
 *createError*("identifier or literal\_integer expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 }  
 break;  
 case *MUL\_OP*:  
 parent.addChild(new Node(*nextToken*, parent, "multiplication\_operation"));  
 *getToken*();  
 if(*nextToken*.getType() == Token.TokenType.*DIGIT* || *nextToken*.getType() == Token.TokenType.*LETTER*){  
 *arithExpr*(parent);  
 *getToken*();  
 if(*nextToken*.getType() == Token.TokenType.*DIGIT* || *nextToken*.getType() == Token.TokenType.*LETTER*){  
 *arithExpr*(parent);  
 } else {  
 *createError*("identifier or literal\_integer expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 }  
 } else {  
 *createError*("identifier or literal\_integer expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 }  
 break;  
 case *ADD\_OP*:  
 parent.addChild(new Node(*nextToken*, parent, "addition\_operation"));  
 *getToken*();  
 if(*nextToken*.getType() == Token.TokenType.*DIGIT* || *nextToken*.getType() == Token.TokenType.*LETTER*){  
 *arithExpr*(parent);  
 *getToken*();  
 if(*nextToken*.getType() == Token.TokenType.*DIGIT* || *nextToken*.getType() == Token.TokenType.*LETTER*){  
 *arithExpr*(parent);  
 } else {  
 *createError*("identifier or literal\_integer expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 }  
 } else {  
 *createError*("identifier or literal\_integer expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 }  
 break;  
 case *SUB\_OP*:  
 parent.addChild(new Node(*nextToken*, parent, "subtraction\_operation"));  
 *getToken*();  
 if(*nextToken*.getType() == Token.TokenType.*DIGIT* || *nextToken*.getType() == Token.TokenType.*LETTER*){  
 *arithExpr*(parent);  
 *getToken*();  
 if(*nextToken*.getType() == Token.TokenType.*DIGIT* || *nextToken*.getType() == Token.TokenType.*LETTER*){  
 *arithExpr*(parent);  
 } else {  
 *createError*("identifier or literal\_integer expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 }  
 } else {  
 *createError*("identifier or literal\_integer expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 }  
 break;  
 default:  
 *createError*("unexpected operation", *nextToken*.getLexeme(), *nextToken*.getLine());  
 break;  
 }  
 }  
  
 //arithmetic\_expression  
 //takes in parent Node  
 // <arithmetic\_expression> -> <id> | <literal\_integer> | <arithmetic\_op> <arithmetic\_expression> <arithmetic\_expression>  
 private static void arithExpr(Node parent){  
 //create arithmetic\_expression Node  
 *createNode*(*nextToken*, *rootNode*, parent, "<arithmetic\_expression>");  
 Node arithNode = *currNode*;  
 //switch statement determines the type of nextToken  
 switch(*nextToken*.getType()){  
 case *LETTER*:  
 arithNode.addChild(new Node(*nextToken*, arithNode, "id"));  
 break;  
 case *DIGIT*:  
 arithNode.addChild(new Node(*nextToken*, arithNode, "literal\_integer"));  
 break;  
 //if nextToken type is not a LETTER or DIGIT it is an arithmetic expression  
 //essentially a recursive call  
 default:  
 //if nextToken is not a keyword/reserved word  
 //if it is a LETTER/DIGIT/or operation  
 if(*words*.contains(*nextToken*.getType())){  
 *nextToken* = *prevToken*;  
 *tokenCount*--;  
 } else if(*nextToken*.getType() == Token.TokenType.*LETTER* || *nextToken*.getType() == Token.TokenType.*DIGIT*){  
 *createError*("operation expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 } else  
 *arithOp*(arithNode);  
 break;  
 }  
 }  
  
 //print statement  
 //takes in parent Node  
 // <print\_statement> -> print ( <arithmetic\_expression> )  
 private static void printState(Node parent){  
 //create print Node  
 *createNode*(*nextToken*, *rootNode*, parent, "<print\_statement>");  
 Node printNode = *currNode*;  
 //add print keyword as new Node to printNode  
 printNode.addChild(new Node(*nextToken*, printNode, "print"));  
 *getToken*(); // process print  
 if(*nextToken*.getType() == Token.TokenType.*L\_PAREN*){  
 //add ( as new Node to printNode  
 printNode.addChild(new Node(*nextToken*, printNode, "("));  
 *getToken*(); // process (  
 *arithExpr*(printNode); //arithExpr() call  
 *getToken*(); // process end of arithExpr()  
 if(*nextToken*.getType() == Token.TokenType.*R\_PAREN*){  
 //add ) as new Node to printNode  
 printNode.addChild(new Node(*nextToken*, printNode, ")"));  
 *getToken*(); //process )  
 return;  
 } else {  
 *createError*(") expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 }  
 } else {  
 *createError*("( expected", *nextToken*.getLexeme(), *nextToken*.getLine());  
 return;  
 }  
 }  
  
 public static void main(String args[]){  
 *fillOpcodes*(); //fill opcodes ArrayList  
 *checkOpcodes*(); //ensure all Token's opcodes are valid  
 *fillWords*(); //fill words ArrayList  
 File f = new File("src/Julia-Files/Test4.jl");  
 //get tokens for file  
 *tokens* = LexicalAnalyzer.*getTokenList*(f);  
 *parse*();  
 //if errorOccurred printError otherwise printTree  
 if(*errorOccurred*){  
 *errors*.get(0).printError();  
 } else  
 *printTree*();  
 }  
  
}

## Node.java

/\*  
Class: CS 4308 Section 03  
Term: Fall 2021  
Name: Faith Swetnam  
Instructor: Sharon Perry  
Project: Deliverable 2 Parser  
 \*/  
  
import java.util.ArrayList;  
public class Node {  
 private ArrayList<Token> tokens; //stores all the Token objects associated with the Node  
 private String grammar; //stores the String grammar  
 private Node parent; //stores the Node's parent  
 private ArrayList<Node> children; //stores the Node's children  
  
 //Node constructors  
 Node(Token token, String grammar){  
 this.tokens = new ArrayList<Token>();  
 tokens.add(token);  
 this.children = new ArrayList<Node>();  
 this.parent = new Node();  
 this.grammar = grammar;  
 }  
  
 Node(Token token, Node parent, String grammar){  
 this.tokens = new ArrayList<Token>();  
 tokens.add(token);  
 this.children = new ArrayList<Node>();  
 this.parent = parent;  
 this.grammar = grammar;  
 }  
  
 Node(){  
 this.tokens = new ArrayList<Token>();  
 this.children = new ArrayList<Node>();  
 this.parent = null;  
 this.grammar = "";  
 }  
  
 //function to return the grammar of a Node  
 String getGrammar(){ return grammar; }  
 //function to return the Node's ArrayList of children  
 ArrayList<Node> getChildren(){ return children; }  
 //function to add a child to the Node  
 void addChild(Node child){ children.add(child); }  
 //function to add a Token to the Node  
 void addToken(Token token){ this.tokens.add(token); }  
  
}

# Code Overview

## Parser.java

### Variables

* nodes
  + This is an ArrayList of Node objects, which stores the Node objects produced by the parser. It essentially stores all nodes in the AST.
* opcodes
  + This is an ArrayList of Integer object, which stores the integer opcodes for reserved words/keywords and operations.
* errors
  + errors is an ArrayList of Error objects that stores any Errors generated by the parser.
* tokens
  + tokens stores all the tokens received from the lexical analyzer.
* Words
  + words is an ArrayList of TokenTypes that stores all valid TokenTypes (reserved words and keywords).
* nextToken
  + nextToken stores the current Token being processed by the parser.
* prevToken
  + prevToken stores the last Token that was processed by the parser
* rootNode
  + rootNode stores the root of the AST.
* currNode
  + currNode stores the current Node the parser is analyzing.
* prevNode
  + prevNode stores the last Node generated, or currNode’s parents.
* tokenCount
  + tokenCount is the index of nextToken in the tokens ArrayList.
* errorOccurred
  + errorOccurred stores if an error has occurred during parsing.

### Methods

* fillOpcodes(): This method fills opcodes with all possible legal opcodes.
* checkOpcodes():
  + This method checks all the Tokens in tokens to ensure they have valid opcodes.
* createError(String msg, String value, int line)
  + This method takes in a String called msg, another String called value, and one last Integer called line. Using these variables, the method creates an Error object, adds it to errors, and returns.
  + When I say ‘generate’ or ‘create’ Error objects, I mean this method.
* getToken()
  + getToken() is made up of a try-catch block. Essentially, it attempts to get the next Token object stored in tokens. If a next Token exists, prevToken is set equal to nextToken, nextToken is set equal to the next Token, and tokenCount is incremented.
  + If the TokenType of nextToken is ERROR, an Error is generated (“scanner error”). If there is not a nextToken, the try-catch block generates an Error (“index out of bounds”).
  + While I may not mention it, getToken() is used multiple times for each method between method calls to process Tokens and to get the next Token from tokens.
* createNode(Token token, Node root, Node parent, String grammar)
  + This method creates a new Node object. It is passed all the information needed to create a Node: a Token (token), the root Node(root), the parent Node, and a String grammar. This method is recursive, and the root variable is used to store the Node being tested. It is passed the rootNode first and then using a for loop it recursively calls with its child as the root variable. This will continue with each of its children, and their children’s children until root is equal to the parent Node passed.
  + If rootNode is NULL, the Node object is stored there. Otherwise, the Node is a child of one of rootNode’s children.
  + When I say “create a Node” I mean this function.
* printTree()
  + printTree() prints the AST generated depth first by printing each Node and its children on one line.
* parse()
  + The main working method of the parser.
  + It starts by getting the first Token in tokens. Then, using that Token (stored in nextToken), it creates the first Node in nodes, the “<program>” Node. This should be the rootNode. After that, a set of nested if-else statements are used to add the Tokens that store the function keyword, the function id, and the parentheses to the “<program>” Nodes tokens ArrayList. If these are not in the correct position, an Error is generated for these missing Tokens. (“function expected”, “function identifier expected”, “( expected”, “) expected”) After the nested if statement, rootNode should have been initialized and it should have 4 Tokens associated with it: the Token associated with the ‘function’ keyword, the Token associated with the function’s id, and the Tokens associated with the parentheses that come after the function id.
  + After the function Tokens are dealt with a while loop starts. It will loop as long nextToken does not have a TokenType of EOF (end of file), NULL, or ERROR. Each loop represents a call to block(). After calling block(), an if statement checks that an error has not occurred. If so, the method returns. Otherwise, if nextToken does not have a TokenType of EOF, NULL, or ERROR we call getToken() to get the next Token from tokens.
  + Finally, an if statement at the end checks if the last nextToken available has an EOF TokenType. If nextToken does not have an EOF TokenType here, the source code did not end correctly so an Error is generated. (“EOF expected”)
* block(Node parent)
  + <block> -> <statement> | <statement> <block>
  + This method takes in a Node called parent, which stores the parent Node of any Node created in the method. Most all the other methods in this program (that represent expressions or statements) takes in a parent Node for the same reason.
  + This method again checks if the nextToken has a TokenType of NULL, ERROR, EOF, or END(in this case). If it does not have one of those TokenTypes, it creates a new “<block>” Node using nextToken and calls statement() with it as its parent.
  + When I mention an expression in quotes (“<block>”), the String in the quotes is the String grammar of the Node. So, a “<block>” Node is a Node that has a grammar of “<block>”
* statement(Node parent)
  + <statement> -> <if\_statement> | <assignment\_statement> | <while\_statement> | <print\_statement> | <repeat\_statement>
  + This method takes in a parent Node. (check block())
  + Using the parent node and the Token stored in nextToken, the method creates a new “<statement>” Node, that is stored in statementNode. Then, based on the TokenType of nextToken, a switch statement calls one of the statement methods: assignState(), ifState(), whileState(), printState(), or repeatState(). If nextToken’s TokenType does not have one of the above method’s associated TokenType an Error is generated. (“unexpected statement”) For example, printState() should only be called if nextToken has a TokenType of PRINT. The method passes, with each of these methods calls, statementNode as its parent Node.
* assignState(Node parent)
  + <assignment\_statement> -> id <assignment\_operator> <arithmetic\_expression>
  + This method takes in a parent Node. (check block())
  + First, this method creates a new “<assignment\_statement>” Node with parent and nextToken, which it stores in assignNode. Then it checks for the identifier. If nextToken is does not have a TokenType of LETTER, has a length greater than one, or is part of the words ArrayList, it cannot be considered an identifier and an Error is generated. (“identifier expected”) Otherwise, it adds nextToken as an “identifier” Node to assignNode’s children array and calls getToken(). Then it calls arithOp() and passes assignNode as its parent.
* ifState(Node parent)
  + <if\_statement> -> if <boolean\_expression> then <block> else <block> end
  + This method takes in a parent Node. (check block())
  + Like all the other methods, it starts by creating a new Node with parent and nextToken, an “<if\_statement>” Node called ifNode. It adds the ‘if’ keyword as a child Node to ifNode. Then it processes the ‘if’ (getToken()) and calls boolExpr() with the ifNode as the passed parent Node.
  + After boolExpr(), nested if-else statements are used to ensure proper keyword placement. If nextToken has a TokenType of THEN, “then” is added as a child Node to ifNode and block() is called with ifNode. After that if the nextToken has a TokenType of ELSE, then an “else” Node is added to ifNode’s children and block() called with ifNode again. Lastly if nextToken has a TokenType of END, “end” is added to ifNode, and the method returns. In between these are calls of getToken() to process and get Tokens from tokens. If in any of these cases the TokenType of nextToken does not satisfy the if statement an Error is generated (“then expected”, “else expected”, “end expected”).
* whileState(Node parent)
  + <while\_statement> -> while <boolean\_expression> do <block> end
  + This method takes in a parent Node. (check block())
  + The method creates a new Node for the while statement (“<while\_statement>”) and stores it in whileNode. It adds the ‘while’ keyword to whileNode as a new child Node using nextToken as its Token and “while” as the grammar. Then it calls boolExpr with whileNode as its parent. A nested if-else statement is used to assure that ‘do’ and ‘end’ Tokens are in their proper places. Block() is called with whileNode as its parent after ‘do’ is processed. If neither ‘do’ or ‘end’ are placed properly, an Error is generated (“end expected”, “do expected”).
* repeatNode(Node parent)
  + <repeat\_statement> -> repeat <block> until <boolean\_expression>
  + This method takes in a parent Node. (check block())
  + It starts by creating a new Node and storing it in repeatNode (“<repeat\_expression>”). A new Node with the Token ‘repeat’ is created as a child to repeatNode. Then block() is called with repeatNode passed as its parent Node. An if-else statement checks that nextToken has a TokenType of UNTIL after the block() is called. If it does, a new child Node for ‘until’ is added to repeatNode and boolExpr() is called with repeatNode as its parent. Otherwise, an Error is generated (“until expected”).
* boolExpr(Node parent)
  + <boolean\_expression> -> <relative\_op> <arithmetic\_expression> <arithmetic\_expression>
  + This method takes in a parent Node. (check block())
  + First this method creates a new Node with parent, nextToken, and the grammar String “<Boolean\_expression>” and stores it in boolNode. Then if nextToken is not contained in the ArrayList words (or is an operation), relOp() is called with boolNode as its parent. Otherwise, an Error is created (“unexpected operation”). Then an if statement determines if the next two Tokens have either the LETTER or DIGIT TokenType. If so, arithExpr() is called for each Token with boolNode as their parent. If either of the Tokens fail the if statement an Error is generated (“unexpected Token”).
* relOp(Node parent)
  + <relative\_op> -> le\_operator | lt\_operator | ge\_operator | gt\_operator | eq\_operator | ne\_operator
  + This method takes in a parent Node. (check block())
  + This method starts by creating a “<relative\_op>” Node and stores it in relNode. The main section of this method is the switch statement that determines if the operation stored in nextToken is a valid operation.
  + Valid operations include:
    - <= (le\_operator)
    - >= (ge\_operator)
    - < (lt\_operator)
    - > (gt\_operator)
    - == (eq\_operator)
    - ~= (ne\_operator).
  + If the Token in nextToken does not have a TokenType equal to any of the operations above an Error is generated. (“unexpected operation”)
* arithOp(Node parent)
  + <arithmetic\_op> -> add\_operator | sub\_operator | mul\_operator | div\_operator
  + This method takes in a Node called parent. (check block())
  + I expanded <arithmetic\_op> to be larger than it is implied, I think. I added processing for = and += which are probably considered to be part of <assignment\_statement>. However, I found it easier to call arithExpr() from arithOp() rather than calling them both from assignState().
  + arithOp() is really just a big switch statement using nextToken’s TokenType. If nextToken does not contain a valid operation an Error is generated (“unexpected operation”). Valid Operations are:
    - ASSIGN\_OP (=) and AE\_OP (+=) are essentially the same.
      * To the parent passed to arithOp() we add a new child Node (“assignment\_operator” or “addition\_assignment”). Then the case ensures that nextToken has a TokenType of DIGIT or LETTER before calling arithExpr() with the passed parent Node as its parent. Otherwise, an Error is generated (“literal\_integer expected”).
    - DIV\_OP (/), MUL\_OP(\*), ADD\_OP(+), SUB\_OP(-) are also similar.
      * We create a new Node an add to the parent Node’s children (the one passed to arithOp()). (“division\_operation”, “multiplication\_operation”, “addition\_operation”, “subtraction\_operation”) Then for each arithExpr() call, it checks that the nextToken is an identifier or literal integer. If not, an Error is created (“identifier or literal\_integer expected”).
* arithExpr(Node parent)
  + <arithmetic\_expression> -> <id> | <literal\_integer> | <arithmetic\_op> <arithmetic\_expression> <arithmetic\_expression>
  + This method takes in a Node called parent. (check block())
  + First the method creates a new Node and stores it in arithNode (“<arithmetic\_expression>”). Then the switch statement begins using the TokenType of nextToken. If it has a TokenType of LETTER or DIGIT a new child Node is added to arithNode (“id” or “literal\_integer”). Otherwise, the default case runs. In the default case, an if-statement checks that nextToken’s TokenType is not contained in words. Essentially, it checks if nextToken is an operation. If it is contained in words, the method changes nextToken to prevToken and de-increments tokenCount (essentially, we go back a Token) and returns. If nextToken’s TokenType is either LETTER or DIGIT, an Error is generated (“operation expected”). Otherwise arithOp() is called with arithNode as the parent.
* printState(Node parent)
  + <print\_statement> -> print ( <arithmetic\_expression> )
  + This method takes in a Node called parent.
  + The method creates a new Node with parent and nextToken and stores it in printNode (“<print\_statement>”). Then it adds a new child Node with a grammar of “print” to printNode. Then it ensures that both parentheses are in place, otherwise an Error is generated (“) expected”, “( expected”). If the parentheses are there, they are added individually as child Nodes to printNode. In between the processing for both parentheses the function calls arithExpr() with printNode as its parent.
* Main
  + The main method first fills both opcodes and words and uses checkOpcodes(). Then we create a new File object with the location of the source code file. It calls getTokenList in Lexical\_Analyzer.java with the new File object to get the list of Token objects, which is stored in tokens. Then we call parse() and if an error has not occurred (errorOccurred = false), we call printTree() to print Nodes. Otherwise, it prints the first Error object in the errors ArrayList.

## Node.java

This class create Node objects. Each Node object has 4 characteristics or states. tokens is an ArrayList of Token objects that stores all Token objects associated with the Node. grammar is a String that stores the grammar of the Node. The next variable, parent, stores the Node’s parent Node. The last variable, children, is an ArrayList of Nodes that stores all the Node’s child Nodes.

The Node class has 4 behaviors: getGrammar(), getChildren(), addChild(), and addToken(). getGrammar() returns the Node’s grammar String. getChildren() returns the children ArrayList. addChild() adds a child Node to children and addToken() adds a new Token to tokens.

# Lexical Analyzer Changes

## Overview

The main change I made to the Lexical Analyzer was I made all the functions and static variables private. This meant that I had to add some methods to Token and Lexical\_Analyzer. Also I discovered an error if there were multiple lines of comments (the letter f became separated from function if there were multiple lines of comments), so I had to add an if-else statement to comments(). Also, whenever an Error is generated a Token with a TokenType of ERROR is added to tokens.

I added three methods to Token.java: getType(), getLexeme(), and getLine(). They are essentially just get-er methods for Token’s states. I also had to add a few new TokenTypes for the parser and a new TokenType constructor for the ERROR TokenType.

New TokenTypes:

* NULL (label: “null”, value: “”, opcode: 0)
* ERROR (label: “error”, new Error(msg: “error occurred”), opcode: 9999)
* END (label: “keyword\_end”, value: “end”, opcode: 1000)
* REPEAT (label: “keyword\_repeat”, value: “repeat”, opcode: 1008)
* UNTIL (label: “keyword\_until”, value: “until”, opcode: 1009)

The method I added to the Lexical\_Analyzer class is getTokenList(). It takes in a file object and passes it to the lexical analyzer. It returns an ArrayList of all the Tokens produced. It’s essentially the main class of the Lexical Analyzer. Also I had to add some case statements to lookup() for the new TokenTypes.

## Source Code

### Lexical\_Analyzer.java

/\*  
Class: CS 4308 Section 03  
Term: Fall 2021  
Name: Faith Swetnam  
Instructor: Sharon Perry  
Project: Deliverable 1 Scanner  
Updated: 10/30/2021  
 \*/  
  
import java.io.File;  
import java.io.FileNotFoundException;  
import java.util.ArrayList;  
import java.util.Scanner;  
  
public class LexicalAnalyzer {  
  
 //Global static variables  
 static char *nextChar*; //char to store the next character from file  
 static char[] *sourceArray* = new char[100]; //stores all characters from the source file  
 static ArrayList<Character> *lexeme* = new ArrayList<Character>(); //stores lexemes  
 static int *sourceCount* = 0; //stores position in sourceArray array  
 static Token.CharacterClass *currCharClass*; //CharacterClass of nextChar  
 static Token.CharacterClass *prevCharClass*; //CharacterClass for previous nextChar  
 static ArrayList<Token> *tokens* = new ArrayList<Token>(); //stores all tokens produced  
 static int *sourceLine* = 0; //stores the line of source the lexeme is on  
 static ArrayList<Error> *errors* = new ArrayList<Error>(); //stores errors that are found  
 static boolean *errorOccurred* = false; //stores whether an error has occurred  
 static String *validSymbols* = "=<>~+-/\*\_()"; //stores valid symbols for Julia  
 static int *tokenCount* = 0;  
  
 //ReadFile reads File f and returns a char[] of contents of the file  
 //Adds a '\n' at the end of each line and '\u001a' at the end of the file for processing reasons  
 static char[] readFile(File f) {  
 String source = "";  
 try {  
 Scanner fileReader = new Scanner(f);  
 while(fileReader.hasNextLine()) {  
 source += fileReader.nextLine() + '\n';  
 }  
 source += '\u001a';  
 *sourceArray* = source.toCharArray();  
 fileReader.close();  
 return *sourceArray*;  
 } catch (FileNotFoundException e) {  
 Error err = new Error("File could not be found");  
 *errors*.add(err);  
 *errorOccurred* = true;  
 return *sourceArray*;  
 }  
 }  
  
 //GetChar returns the next character from sourceArray  
 //It sets the currCharClass based on value of nextChar  
 static char getChar() {  
 *nextChar* = *sourceArray*[*sourceCount*];  
 if(*nextChar* != '\u001a') {  
 if(Character.*isLetter*(*nextChar*)) {  
 *currCharClass* = Token.CharacterClass.*LETTER*;  
 } else if (Character.*isDigit*(*nextChar*)) {  
 *currCharClass* = Token.CharacterClass.*DIGIT*;  
 } else {  
 *currCharClass* = Token.CharacterClass.*UNKNOWN*;  
 }  
 *sourceCount*++;  
 } else {  
 *currCharClass* = Token.CharacterClass.*EOF*;  
 }  
 return *nextChar*;  
 }  
  
 //AddChar adds the value in nextChar to lexeme  
 //if lexeme is too long it logs an error  
 static void addChar() {  
 if(*lexeme*.size() < 100) {  
 *lexeme*.add(*nextChar*);  
 } else {  
 Error err = new Error("value is too long", *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 *addToken*(Token.TokenType.*ERROR*, "value is too long", *sourceLine*);  
 }  
 }  
  
 //Function to process (skip over) white-spaces, tabs, new lines in the file  
 //Logs an error if program ends unexpectedly  
 static void processFormatting() {  
 while (Character.*isSpaceChar*(*nextChar*) || *nextChar* == '\t' || *nextChar* == '\n') {  
 if(*nextChar* == '\n'){  
 *sourceLine*++;  
 }  
 if(*nextChar* != '\u001a') {  
 *getChar*();  
 } else {  
 Error err = new Error("program ended unexpectedly", *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 *addToken*(Token.TokenType.*ERROR*, "program ended unexpectedly", *sourceLine*);  
 }  
 }  
  
 }  
  
 //Function to process single line comments  
 //Logs an error if program ends during comment  
 static void comments() {  
 if(*nextChar* == '/' && *sourceArray*[*sourceCount*] =='/') {  
 *processFormatting*();  
 while(*nextChar* != '\n') {  
 if(*nextChar* != '\u001a') {  
 *getChar*();  
 char tempChar = *nextChar*;  
 *getChar*();  
 if(*nextChar* == '/'){  
 *comments*();  
 *sourceLine*++;  
 } else {  
 *nextChar* = tempChar;  
 *sourceCount*--;  
 }  
 } else {  
 Error err = new Error("program ended unexpectedly", *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 *addToken*(Token.TokenType.*ERROR*, "program ended unexpectedly", *sourceLine*);  
 }  
 }  
 }  
 }  
  
 //String turns the lexeme into a string for processing purposes  
 //It returns the string lex when finished  
 static String string() {  
 String lex = "";  
  
 if(*lexeme*.size() > 1) {  
 StringBuilder builder = new StringBuilder(*lexeme*.size());  
 for(Character c: *lexeme*) {  
 if(c.charValue() != '\u001a')  
 builder.append(c);  
 else {  
 Error err = new Error("program ended unexpectedly", *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 *addToken*(Token.TokenType.*ERROR*, "program ended unexpectedly", *sourceLine*);  
 break;  
 }  
 }  
 lex = builder.toString();  
 } else if (*lexeme*.size() > 0){  
 if(*lexeme*.get(0) != '\u001a')  
 lex = Character.*toString*(*lexeme*.get(0));  
 else {  
 Error err = new Error("program ended unexpectedly", *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 *addToken*(Token.TokenType.*ERROR*, "program ended unexpectedly", *sourceLine*);  
 }  
 } else {  
 Error err = new Error("lexeme array is empty", *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 *addToken*(Token.TokenType.*ERROR*, "lexeme array is empty", *sourceLine*);  
 }  
  
 return lex;  
 }  
  
 //adds token to token arraylist  
 static void addToken(Token.TokenType tokenType, String lex, int sourceLine){  
 Token currToken = new Token(tokenType, lex, sourceLine);  
 *tokens*.add(currToken);  
 }  
  
 //LookUp determines the token type based on the lexeme  
 static void lookUp(String lex) {  
 Token.TokenType tokenType;  
 int line = *sourceLine* + 1;  
 //if the lexeme is a word(contains alphabetical letters) do  
 if(*prevCharClass* == Token.CharacterClass.*LETTER*) {  
 switch (lex) {  
 case "end":  
 if(*sourceCount* == *sourceArray*.length-1) {  
 tokenType = Token.TokenType.*EOF*;  
 } else {  
 tokenType = Token.TokenType.*END*;  
 }  
 *addToken*(tokenType, lex, line);  
 break;  
 case "function":  
 tokenType = Token.TokenType.*FUNCT*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "while":  
 tokenType = Token.TokenType.*WHILE*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "do":  
 tokenType = Token.TokenType.*DO*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "print":  
 tokenType = Token.TokenType.*PRINT*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "if":  
 tokenType = Token.TokenType.*IF*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "then":  
 tokenType = Token.TokenType.*THEN*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "else":  
 tokenType = Token.TokenType.*ELSE*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "repeat":  
 tokenType = Token.TokenType.*REPEAT*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "until":  
 tokenType = Token.TokenType.*UNTIL*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "null":  
 tokenType = Token.TokenType.*NULL*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "error":  
 tokenType = Token.TokenType.*ERROR*;  
 *addToken*(tokenType, lex, line);  
 break;  
 default: //lexeme is an identifier  
 tokenType = Token.TokenType.*LETTER*;  
 *addToken*(tokenType, lex, line);  
 break;  
 }  
 //else lexeme is a symbol or number  
 } else {  
 switch(lex) {  
 case "=":  
 tokenType = Token.TokenType.*ASSIGN\_OP*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "<=":  
 tokenType = Token.TokenType.*LE\_OP*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "<":  
 tokenType = Token.TokenType.*LT\_OP*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case ">=":  
 tokenType = Token.TokenType.*GE\_OP*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case ">":  
 tokenType = Token.TokenType.*GT\_OP*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "==":  
 tokenType = Token.TokenType.*EQ\_OP*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "~=":  
 tokenType = Token.TokenType.*NE\_OP*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "+=":  
 tokenType = Token.TokenType.*AE\_OP*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "+":  
 tokenType = Token.TokenType.*ADD\_OP*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "-":  
 tokenType = Token.TokenType.*SUB\_OP*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "\*":  
 tokenType = Token.TokenType.*MUL\_OP*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "/":  
 tokenType = Token.TokenType.*DIV\_OP*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case "(":  
 tokenType = Token.TokenType.*L\_PAREN*;  
 *addToken*(tokenType, lex, line);  
 break;  
 case ")":  
 tokenType = Token.TokenType.*R\_PAREN*;  
 *addToken*(tokenType, lex, line);  
 break;  
 default:  
 tokenType = Token.TokenType.*DIGIT*;  
 *addToken*(tokenType, lex, line);  
 break;  
 }  
 }  
 }  
  
 //checkValid determines if the unknown symbol is valid in the language  
 static void checkValid(){  
 String check = Character.*toString*(*nextChar*);  
 if(!Character.*isLetterOrDigit*(*nextChar*)) {  
 if (!*validSymbols*.contains(check)) {  
 Error err = new Error("unexpected symbol", check, *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 *addToken*(Token.TokenType.*ERROR*, "unexpected symbol", *sourceLine*);  
 }  
 }  
 }  
  
 //main body of the lexical\_analyzer  
 //returns a token based on the lexeme found  
 public static Token lexer() {  
 *lexeme* = new ArrayList<Character>();  
 *comments*();  
 *processFormatting*();  
 switch (*currCharClass*) {  
 case *LETTER*:  
 *addChar*();  
 *getChar*();  
 while((*currCharClass* == Token.CharacterClass.*LETTER* || *nextChar* == '\_') && !*errorOccurred*) {  
 *addChar*();  
 *getChar*();  
 }  
 *prevCharClass* = Token.CharacterClass.*LETTER*;  
 break;  
 case *DIGIT*:  
 *addChar*();  
 *getChar*();  
 while((*currCharClass* == Token.CharacterClass.*DIGIT* || *nextChar* == '.') && !*errorOccurred*) {  
 *addChar*();  
 *getChar*();  
 }  
 *prevCharClass* = Token.CharacterClass.*DIGIT*;  
 break;  
 case *UNKNOWN*:  
 *comments*();  
 *processFormatting*();  
 *checkValid*();  
 *addChar*();  
 *getChar*();  
 while(*validSymbols*.contains(Character.*toString*(*nextChar*)) && *nextChar* != '(' && *nextChar* != ')' && !*errorOccurred*){  
 *addChar*();  
 *getChar*();  
 }  
 *prevCharClass* = Token.CharacterClass.*UNKNOWN*;  
 break;  
 default:  
 *prevCharClass* = Token.CharacterClass.*EOF*;  
 break;  
 }  
 if (*prevCharClass* != Token.CharacterClass.*EOF* && !*errorOccurred*) {  
 String lex = *string*();  
 *lookUp*(lex);  
 *tokenCount*++;  
 return *tokens*.get(*tokenCount*-1);  
 } else if(*errorOccurred*){  
 return new Token(Token.TokenType.*ERROR*, *sourceLine*);  
 } else  
 return new Token(Token.TokenType.*EOF*, "end", *sourceLine*);  
 }  
  
 //checks to ensure that the program ended correctly with 'end'  
 //returns a boolean  
 static boolean endedCorrectly(){  
 if(*tokens*.size() == 0){  
 Error err = new Error("no tokens initialized", *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 *addToken*(Token.TokenType.*ERROR*, "no tokens initialized", *sourceLine*);  
 } else if(*tokens*.get(*tokens*.size()-1).getType() != Token.TokenType.*EOF*){  
 Error err = new Error("program ended unexpectedly", *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 *addToken*(Token.TokenType.*ERROR*, "program ended unexpectedly", *sourceLine*);  
 }  
 return *errorOccurred*;  
 }  
  
 //printTokenTable takes in an ArrayList of Token objects and prints out a symbol table  
 static void printTokenTable(ArrayList<Token> tokens) {  
 System.*out*.printf("%-10s\t%-20s\t%-10s\t%-10s\n", "Lexeme", "Token Type", "Opcode", "Line");  
 System.*out*.println("-----------------------------------------------------\n");  
 for(Token t: tokens) {  
 t.printToken();  
 }  
 }  
  
 //printErrorTable takes in an ArrayList of Error objects and prints them out in a table  
 static void printErrorTable(ArrayList<Error> errors) {  
 System.*out*.printf("%-50s\t%-10s\t%-10s\n", "Errors", "Value", "Line Occured");  
 System.*out*.println("-----------------------------------------------------------------------------\n");  
 for(Error e: errors) {  
 e.printError();  
 }  
 }  
  
 //ADDITION  
 //Function sets up and runs the lexical analyzer without the main and returns the ArrayList of Tokens produced  
 static ArrayList<Token> getTokenList(File f){  
 *readFile*(f);  
 *getChar*();  
 while(*currCharClass* != Token.CharacterClass.*EOF* && !*errorOccurred*) {  
 *lexer*();  
 }  
  
 *endedCorrectly*();  
  
 if(*errorOccurred*) {  
 *printErrorTable*(*errors*);  
 }  
  
 return *tokens*;  
 }  
  
  
 public static void main(String args[]) {  
 File f = new File("src/Julia-Files/Test1.jl");  
 *readFile*(f);  
 *getChar*();  
 while(*currCharClass* != Token.CharacterClass.*EOF* && !*errorOccurred*) {  
 *lexer*();  
 }  
  
 *endedCorrectly*();  
  
 if(*errorOccurred*)  
 *printErrorTable*(*errors*);  
 else  
 *printTokenTable*(*tokens*);  
  
 }  
  
}

### Token.java

/\*  
Class: CS 4308 Section 03  
Term: Fall 2021  
Name: Faith Swetnam  
Instructor: Sharon Perry  
Project: Deliverable 1 Scanner  
Updated: 10/28/2021  
 \*/  
  
//Token object holds a tokens TokenType type, lexeme (value), and the line its on  
public class Token {  
 private TokenType type;  
 private String lexeme;  
 private int line;  
  
 //Token constructor  
 Token(TokenType type, String lexeme, int line){  
 this.type = type;  
 this.lexeme = lexeme;  
 this.line = line;  
 }  
  
 Token(TokenType type, int line){  
 this.type = type;  
 this.lexeme = "ERROR";  
 this.line = line;  
 }  
  
 //Prints out token values  
 void printToken() {  
 System.*out*.printf("%-10s\t%-20s\t%-10d\t%-10d\n", lexeme, type.label, type.opcode, line);  
 }  
  
 //Returns lexeme  
 String getLexeme(){ return lexeme; }  
  
 //Returns line  
 int getLine(){ return line; }  
  
 //Returns TokenType  
 TokenType getType(){  
 return type;  
 }  
 //Enumerated type to hold the type of character in nextChar  
 //Each CharacterClass has a description for printing  
 enum CharacterClass {  
 *LETTER* ("identifier"),  
 *DIGIT* ("number"),  
 *UNKNOWN* ("unknown symbol"),  
 *EOF*("end of program");  
  
 final String description;  
  
 private CharacterClass(String description) {  
 this.description = description;  
 }  
 }  
  
 //Enumerated type to hold legal keywords and symbols (essentially the legal token types allowed by language)  
 //Each TokenType has an associated label and opcode. It should contain the String value that holds the lexeme  
 enum TokenType {  
 //ADDITIONS: ERROR, NULL, END, REPEAT  
 *NULL*("null", "", 0),  
 *LETTER*("identifier", "", 1),  
 *DIGIT*("number", "", 2),  
 *EOF*("end of file", "end", 99),  
 *END*("keyword\_end", "end", 1000),  
 *FUNCT*("keyword\_function", "function", 1001),  
 *WHILE*("keyword\_while", "while", 1002),  
 *DO*("keyword\_do", "do", 1003),  
 *PRINT*("keyword\_print", "print", 1004),  
 *IF*("keyword\_if", "if", 1005),  
 *THEN*("keyword\_then", "then", 1006),  
 *ELSE*("keyword\_else", "else", 1007),  
 *REPEAT*("keyword\_repeat", "repeat", 1008),  
 *UNTIL*("keyword\_until", "until", 1009),  
 *ASSIGN\_OP*("assignment\_operator", "=", 2000),  
 *LE\_OP*("less\_equal", "<=", 2001),  
 *LT\_OP*("less", "<", 2002),  
 *GE\_OP*("greater\_equal", ">=", 2003),  
 *GT\_OP*("greater", ">", 2004),  
 *EQ\_OP*("equal", "==", 2005),  
 *NE\_OP*("not\_equal", "~=", 2006),  
 *AE\_OP*("addition\_assignment", "+=", 2007),  
 *ADD\_OP*("addition\_operator", "+", 2008),  
 *SUB\_OP*("subtraction\_operator", "-", 2009),  
 *MUL\_OP*("multiplication\_operator", "\*", 2010),  
 *DIV\_OP*("division\_operator", "/", 2011),  
 *L\_PAREN*("left\_parenthesis", "(", 2012),  
 *R\_PAREN*("right\_parenthesis", ")", 2013),  
 *ERROR*("error", new Error("error occurred"), 9999);  
  
 final String label;  
 final int opcode;  
 Error e = new Error("error occurred");  
 String value;  
  
 //TokenType constructor  
 private TokenType(String label, String value, int opcode) {  
 this.label = label;  
 this.value = value;  
 this.opcode = opcode;  
 }  
  
 private TokenType(String label, Error e, int opcode){  
 this.label = label;  
 this.e = e;  
 this.opcode = opcode;  
 }  
 }  
}

# Screenshots (fix bool errors)

## Successful runs

### A computer screen capture Description automatically generated with medium confidenceTest1.jl

### Text Description automatically generatedTest2.jl

### A computer screen capture Description automatically generated with medium confidenceTest3.jl

### Test4.jl

I made a new test function to test TokenType.REPEAT and TokenType.UNTIL for one of the previous test functions. I’m not entirely sure its correct, but I thought it would be better to have a version of it then not.

A screenshot of a computer

Description automatically generated with medium confidence//test *4* in julia  
function a()  
 x = *1* repeat  
 x += *1* until >x *100* print(x)  
end

## Some Error Checking (with errors placed in source code)

### Program start errors

#### A computer screen capture Description automatically generated with medium confidence‘function’ keyword missing/misspelled

#### A computer screen capture Description automatically generated with medium confidencefunction id missing

#### A computer screen capture Description automatically generated with medium confidencefunction parentheses missing

### assignment statement errors

#### A computer screen capture Description automatically generated with medium confidenceassignment id missing

#### A computer screen capture Description automatically generated with medium confidenceassignment operation missing

#### A computer screen capture Description automatically generated with medium confidenceassignment operation invalid

#### assignment expression missing

A computer screen capture

Description automatically generated with medium confidence

### if statement errors

#### A computer screen capture Description automatically generated with medium confidence‘then’ keyword missing/misspelled

#### A computer screen capture Description automatically generated with medium confidence‘else’ keyword missing/misspelled

#### ‘end’ keyword missing/misspelled

A computer screen capture

Description automatically generated with medium confidence

#### Error in boolExpr()

A computer screen capture

Description automatically generated with medium confidence

#### A computer screen capture Description automatically generated with medium confidenceError in block()

### while statement errors

#### ‘do’ keyword missing/misspelled

A computer screen capture

Description automatically generated with medium confidence

#### A computer screen capture Description automatically generated with medium confidence‘end’ keyword missing/misspelled

#### A computer screen capture Description automatically generated with medium confidenceError in boolExpr()

#### A computer screen capture Description automatically generated with medium confidenceError in block()

### repeat statement errors

#### A computer screen capture Description automatically generated with medium confidence‘until’ keyword missing/misspelled

#### A computer screen capture Description automatically generated with medium confidenceError in boolExpr()

#### A computer screen capture Description automatically generated with medium confidenceError in block()

### print statement errors

#### A computer screen capture Description automatically generated with medium confidence‘(‘ missing

#### A computer screen capture Description automatically generated with medium confidence‘)’ missing

#### A computer screen capture Description automatically generated with medium confidenceError in arithExpr()

# References

Ali, Shahnawaz. “DFS Traversal of a Tree Using Recursion.” *GeeksforGeeks*, GeeksforGeeks, 30 Sept. 2021, https://www.geeksforgeeks.org/dfs-traversal-of-a-tree-using-recursion/.

Belwariar, Rachit. “Number of Ways to Traverse an n-Ary Tree.” *GeeksforGeeks*, GeeksforGeeks, 2 July 2021, https://www.geeksforgeeks.org/number-of-ways-to-traverse-an-n-ary-tree/.

Blesson, Chris. “Insertion in n-Ary Tree in given Order and Level Order Traversal.” *GeeksforGeeks*, GeeksforGeeks, 18 Aug. 2021, https://www.geeksforgeeks.org/insertion-in-n-ary-tree-in-given-order-and-level-order-traversal/.

Chauhan, Aashish. “Print All Root to Leaf Paths of an n-Ary Tree.” *GeeksforGeeks*, GeeksforGeeks, 14 Aug. 2021, https://www.geeksforgeeks.org/print-all-root-to-leaf-paths-of-an-n-ary-tree/.

code\_freak. “Inorder Traversal of an n-Ary Tree.” *GeeksforGeeks*, GeeksforGeeks, 12 Oct. 2021, https://www.geeksforgeeks.org/inorder-traversal-of-an-n-ary-tree/.

Gupta, Shubham. “Depth of an n-Ary Tree.” *GeeksforGeeks*, GeeksforGeeks, 28 June 2021, https://www.geeksforgeeks.org/depth-n-ary-tree/.

Ilic, Branko. “Java: Check If Array Contains Value or Element.” *Stack Abuse*, Stack Abuse, 19 Nov. 2020, https://stackabuse.com/java-check-if-array-contains-value-or-element/.

*Java Arraylist*, Refsnes Data, https://www.w3schools.com/java/java\_arraylist.asp.

Kvasnovsky, Ondrej. “Solution.” *N-Ary Tree Preorder Traversal*, Github, 2020, https://ondrej-kvasnovsky-2.gitbook.io/algorithms/data-structures/n-ary-tree/n-ary-tree-preorder-traversal.

“N-Ary Tree Data Structure.” *Studytonight.com*, Studytonight Technologies Pvt. Ltd., https://www.studytonight.com/advanced-data-structures/nary-tree.

Techopedia. “What Is a Java Object? - Definition from Techopedia.” *Techopedia.com*, Techopedia, 9 Sept. 2021, https://www.techopedia.com/definition/24339/java-object.