**COVER PAGE**

# CS323 Programming Assignments

1. Your Name: Tri Dang Section: 02

2. Assignment Number: 3

3. Due Date: 12/11/2017

4. Turn-In Date: 12/3/17

5. Executable FileName: Compiler.jar

6. LabRoom: CS202, computer 7d

7. OS: Windows 8



GRADE:

COMMENTS:

**CS323 Assignment 3 Documentation**

**Problem Statement**

The purpose of this assignment is to generate an assembly code for the simplified version of Rat17F, as well as implement the handling of the Symbol table.

**How to use your program**

This program only requires that the user run the Compiler.jar file and have a file named “TestFile.txt” that contains the source code. Once run, this program will read the text file and output the result into another text file named “OutputFile.txt”. These files are provided in the submission folder.

If the program have successfully parse the source code, it will print out a message at the bottom of the text file indicating the result.

**Design of your program**

The program uses Java’s FileReader, FileWriter, BufferedReader, and BufferedWriter class to read from a source file and write the output to an output file. Java’s Stack data structure was used to implement the jumpStack.

The program contains several major components:

* The lexer() method, which returns a token, or Record object, when it is needed.
* The rat17f() method that starts the parsing process for the source code.
* The semantic actions that generate assembly instructions and symbols, which are stored in the Instruction Table and Symbol Table, respectively.
* Multiple recursive methods that generate all the productions that are used to parse the source code.

**Any Limitation**

The program will stop if a syntax error occurs, since the assignment does not require for it to continue. The Instruction Table contain a dummy node at the end so that instructions can have somewhere to jump out of once it reaches the end of the source code.

**Any shortcomings**

None.

**Source Code**

1 /\*\*  
 2 \* **@author** Tri Dang  
 3 \* 11/28/17  
 4 \* Assignment 3 V1  
 5 \* This creates and test a lexical analyzer class that uses finite state machines.   
 6 \* The main goal of this class is to be able to translate the Rat17F language  
 7 \* Some important features of this program:  
 8 \* -The lexer() which reads characters from a file and returns the correct tokens and lexemes based on the Rat17F language  
 9 \* -The Syntax Analyzer that parse each token from the lexer and print out all production rules used for analyzing the token  
 10 \* -The semantic actions that generates assembly instruction and symbol table handling  
 11 \*/  
 12 import java.io.FileNotFoundException;  
 13 import java.io.FileReader;  
 14 import java.io.FileWriter;  
 15 import java.io.BufferedReader;  
 16 import java.io.BufferedWriter;  
 17 import java.io.IOException;  
 18 import java.util.LinkedList;  
 19 import java.util.Stack;  
 20   
 21 public class **LexicalAnalyzer** {  
 22   
 23 //inner class Record that stores string data: token and lexeme  
 24 class **Record** {  
 25 String token, lexeme;  
 26   
 27 public **Record**(String token, String lexeme) {  
 28 this.token = token;  
 29 this.lexeme = lexeme;  
 30 }  
 31 public String **getToken**() {  
 32 return token;  
 33 }   
 34 public String **getLexeme**() {  
 35 return lexeme;  
 36 }  
 37 public String **toString**() {  
 38 return "This is a " + token + " token with lexeme: " + lexeme;  
 39 }  
 40 }  
 41 //inner class Instruction that stores instruction address, operator, operand  
 42 class **Instruction** {  
 43 int address, operand;  
 44 String operator;  
 45   
 46 public **Instruction**(int address, String operator, int operand) {  
 47 this.address = address;  
 48 this.operator = operator;  
 49 this.operand = operand;  
 50 }  
 51   
 52 public int **getAddress**() {  
 53 return address;  
 54 }  
 55 public int **getOperand**() {  
 56 return operand;  
 57 }  
 58 public String **getOperator**() {  
 59 return operator;  
 60 }  
 61 public String **toString**() {  
 62 return "Instruction Address: " + address + "**\t** Operator: " + operator + "**\t** Operand: " + operand;  
 63 }  
 64 }  
 65   
 66 //inner class Symbol that stores symbol's name, memory address, data type  
 67 class **Symbol** {  
 68 String name, type;  
 69 int memory;  
 70   
 71 public **Symbol**(String name, int memory, String type) {  
 72 this.name = name;  
 73 this.memory = memory;  
 74 this.type = type;  
 75 }  
 76   
 77 public String **getName**() {  
 78 return name;  
 79 }  
 80 public int **getMemory**() {  
 81 return memory;  
 82 }  
 83 public String **getType**() {  
 84 return type;  
 85 }  
 86 public String **toString**() {  
 87 return "Symbol: " + name + "**\t** Memory Address: " + memory + "**\t** Type: " + type;  
 88 }  
 89 }  
 90   
 91 ////////GLOBAL VARIABLES  
 92 private BufferedReader input;  
 93 private BufferedWriter writer;  
 94 private char currentChar, nextChar;  
 95 private int lastState, nextState, lineCount, statementCount;  
 96 private boolean acceptance, terminateSpaces;  
 97 private boolean printSwitch;  
 98 private boolean success;  
 99 private Record token;  
 100 private int memoryAddress, currentSymSize;  
 101 private int instructionAddress;  
 102 private Instruction[] instructionTable;  
 103 private Symbol[] symbolTable;  
 104 private String idType1, idType2;  
 105 private Record saved;  
 106 private Stack jumpStack;  
 107 private String conOp;  
 108 private int addressWhile;  
 109 private boolean whileToggle, ifToggle, readToggle;  
 110   
 111 private final String[] tokensArray = new String[] {"keyword", "identifier", "separator", "operator", "integer", "real"};  
 112 private final String[] keywordsArray = new String[]{"if", "integer", "false", "fi", "floating", "else", "return", "read", "true", "write", "while", "boolean"};  
 113 private final char[] separatorsArray = new char[] {'{', '}', '(', ')', '[', ']', '%', '@', ':', ';', ','};  
 114 private final char[] operatorsArray = new char[] {'=', '/', '>', '<', '+', '-', '\*'};  
 115 private char[] lettersArray = new char[] {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',   
 116 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'};  
 117 private char[] digitsArray = new char[] {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};  
 118   
 119 //creates a FSM table for identifier, integer + real tokens  
 120 private int[][] idStateTable = {{1, 5},   
 121 {3, 2},  
 122 {3, 5},  
 123 {1, 4},  
 124 {3, 5},  
 125 {5, 5}};   
 126   
 127 private int[][] intStateTable = {{1, 5},   
 128 {2, 3},  
 129 {2, 3},  
 130 {4, 5},  
 131 {4, 5},  
 132 {5, 5}};  
 133   
 134 ///////////MAIN CONSTRUCTOR//////////////  
 135 public **LexicalAnalyzer**(String source, String output) {  
 136 //reader   
 137 try {  
 138 input = new BufferedReader(new FileReader(source));  
 139 nextChar = (char) read();  
 140 }   
 141 catch (FileNotFoundException ex) {  
 142 System.*out*.println("Unable to open");  
 143 }  
 144 //writer  
 145 try {  
 146 writer = new BufferedWriter(new FileWriter(output));  
 147 }  
 148 catch (IOException e) {  
 149 e.printStackTrace();  
 150 }  
 151 lineCount = 1;  
 152 statementCount = 0;  
 153 success = acceptance = true;  
 154 terminateSpaces = false;  
 155 currentChar = nextChar;  
 156 lastState = 0;  
 157   
 158 //set up for symbol table  
 159 memoryAddress = 10000;  
 160 currentSymSize = 0;  
 161 symbolTable = new Symbol[100];  
 162   
 163 //set up for instruction table for semantic analyzer  
 164 instructionAddress = 1;  
 165 instructionTable = new Instruction[500];  
 166 Instruction newNode = new Instruction(0, " ", 0);  
 167 instructionTable[0] = newNode;  
 168   
 169 //set up for while conditional jumps  
 170 jumpStack = new Stack();  
 171   
 172 if (currentChar == '.' || currentChar == '#') {  
 173 nextState = 5;  
 174 }  
 175 else {  
 176 nextState = 0;  
 177 }  
 178 nextChar = read();  
 179 }  
 180 ///////////////WHILE LOOP HELPER METHODS/////////////////////  
 181 public void **backPatch**(int jumpaddr) {  
 182 int address = (int) jumpStack.pop();  
 183 instructionTable[address].operand = jumpaddr;  
 184 }  
 185 ///////////////SYMBOL TABLE HELPER METHODS////////////////////  
 186 //helper method to insert new symbol into symbol table  
 187 public void **insert**(String name, String data) {  
 188 for (int i = 0; i < currentSymSize; i++) {  
 189 if (symbolTable[i].getName().equals(name)) {  
 190 write("ERROR at line " + lineCount + "! Symbol: " + name + " has already been declared elsewhere!");  
 191 throw new RuntimeException();  
 192 }  
 193 }  
 194 Symbol newSymbol = new Symbol(name, memoryAddress, data);  
 195 symbolTable[currentSymSize] = newSymbol;  
 196 memoryAddress++; //incremented when a new identifier is declared and placed into symbol table  
 197 currentSymSize++;  
 198 }  
 199 //helper method to print symbol table  
 200 public void **printSymbolTable**() {  
 201 for (int i = 0; i < currentSymSize; i++) {  
 202 write(symbolTable[i].toString());  
 203 }  
 204 }  
 205 //helper method to check if a particular identifer is in the symbol table and return it, otherwise return null  
 206 public Symbol **lookup**(String name) {  
 207 for (int i = 0; i < currentSymSize; i++) {  
 208 if (symbolTable[i].getName().equals(name)) {  
 209 return symbolTable[i];  
 210 }  
 211 }  
 212 return null;  
 213 }  
 214   
 215 ///////////////INSTRUCTION TABLE HELPER METHODS////////////////////  
 216 //helper method to generate a new instruction in the instruction table  
 217 public void **genInstr**(String op, int operand) {  
 218 Instruction newNode = new Instruction(instructionAddress, op, operand);  
 219 instructionTable[instructionAddress] = newNode;  
 220 instructionAddress++; //incremented when a new instruction is added to table  
 221 }  
 222   
 223 public void **printInstructionTable**() {  
 224 int i = 0;  
 225 while (instructionTable[i] != null) {  
 226 write(instructionTable[i].toString());  
 227 i++;  
 228 }  
 229 }

PLEASE NOTE: Code for Lexical Analyzer (line 230-612) have been omitted to reduce paper.

613 ////////////// SYNTAX ANALYZER /////////////////  
 614 public void **rat17f**(Record currentToken) {  
 615 try {  
 616 token = currentToken;  
 617 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 618 //write("<Rat17F> -> <Optional Function Definitions> %% <Optional Declaration List> <Statement List>");  
 619 //statementCount++;  
 620 //ofd();  
 621 if (token.getLexeme().equals("%%")) {  
 622 //statementCount--;  
 623 if ((token = lexer()) == null) {  
 624 success = false;  
 625 throw new RuntimeException();  
 626 }  
 627 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 628 odl();   
 629 sl();   
 630 if (token != null) {  
 631 write("ERROR at line " + lineCount + "! Expected <Statement> at <Rat17f>. Current token: " +token.getLexeme());  
 632 write("ERROR! Parsing Unsuccessful!");  
 633 }  
 634 }  
 635 else {  
 636 write("ERROR at line " + lineCount + "! Expected token: %% at <Rat17f>. Current token: " +token.getLexeme());  
 637 throw new RuntimeException();  
 638 }  
 639 }  
 640 catch (Exception e) {  
 641 if (token == null) {   
 642 if (statementCount == 0) {  
 643 if (success == true) {  
 644 if (whileToggle == true) {  
 645 genInstr("JUMP ", addressWhile);  
 646 backPatch(instructionAddress);  
 647 }  
 648 if (ifToggle == true) {  
 649 genInstr("JUMP ", addressWhile);  
 650 backPatch(instructionAddress);  
 651 }  
 652 genInstr(" ", 0);  
 653 write("Parsing Successful!");  
 654 write("///////////SYMBOL TABLE//////////");  
 655 printSymbolTable();  
 656 write("///////////INSTRUCTION TABLE//////////");  
 657 printInstructionTable();  
 658 }  
 659 else {  
 660 write("ERROR at line " + lineCount + "! Expected complete <Statement> at <Rat17f>. Current token: " + token);  
 661 write("ERROR! Parsing Unsuccessful!");  
 662 }  
 663 }  
 664 else {  
 665 write("ERROR at line " + lineCount + "! Expected token: %% or end of <Statement> token at <Rat17f>. Current token: " + token);  
 666 write("ERROR! Parsing Unsuccessful!");  
 667 }  
 668 }  
 669 else {  
 670 try {  
 671 writer.write("ERROR! Parsing Unsuccessful!");  
 672 }  
 673 catch (IOException exception) {  
 674 e.printStackTrace();  
 675 }  
 676 }   
 677 }  
 678 }  
 679   
 680 public void **ofd**() {   
 681 write("<Optional Function Definitions> -> <Function Definitions> | <Empty>");  
 682 if (token.getLexeme().equals("@")) {  
 683 fd();  
 684 }  
 685 }  
 686   
 687 public void **fd**() {  
 688 write("<Function Definitions> -> <Function> | <Function> <Function Definitions>");  
 689 function();  
 690 if (token.getLexeme().equals("@")) {  
 691 fd();  
 692 }  
 693 }  
 694   
 695 public void **function**() {  
 696 write("<Function> -> @ Identifier (<Optional Parameter List>) <Optional Declaration List> <Body>");  
 697 if (token.getLexeme().equals("@")) {  
 698 token = lexer();  
 699 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 700 if (token.getToken().equals("Identifier")) {  
 701 token = lexer();  
 702 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 703 if (token.getLexeme().equals("(")) {   
 704 token = lexer();  
 705 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 706 opl();  
 707 if (token.getLexeme().equals(")")) {   
 708 if ((token = lexer()) == null) {  
 709 success = false;  
 710 throw new RuntimeException();  
 711 }  
 712 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 713 odl();  
 714 body();  
 715 }  
 716 else {  
 717 write("ERROR at line " + lineCount + "! Expected token: ) at <Function>. Current token: " +token.getLexeme());  
 718 throw new RuntimeException();  
 719 }  
 720 }  
 721 else {  
 722 write("ERROR at line " + lineCount + "! Expected token: ( at <Function>. Current token: " +token.getLexeme());  
 723 throw new RuntimeException();  
 724 }  
 725 }  
 726 else {  
 727 write("ERROR at line " + lineCount + "! Expected token: Identifier at <Function>. Current token: " +token.getToken());  
 728 throw new RuntimeException();  
 729 }  
 730 }  
 731 else {  
 732 write("ERROR at line " + lineCount + "! Expected token: @ at <Function>. Current token: " +token.getLexeme());  
 733 throw new RuntimeException();  
 734 }  
 735 }  
 736   
 737 public void **opl**() {  
 738 write("<Optional Parameter List> -> <Parameter List> | <Empty>");  
 739 if (token.getToken().equals("Identifier")) {  
 740 pl();  
 741 }  
 742 }  
 743   
 744 public void **pl**() {  
 745 write("<Parameter List> -> <Parameter> | <Parameter> , <Parameter List>");  
 746 statementCount++;  
 747 parameter();  
 748 statementCount--;  
 749 if (token.getLexeme().equals(",")) {  
 750 token = lexer();  
 751 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 752 pl();  
 753 }  
 754 }  
 755   
 756 public void **parameter**() {  
 757 write("<Parameter> -> <IDs> : <Qualifier>");  
 758 ids();  
 759 if (token.getLexeme().equals(":")) {   
 760 token = lexer();  
 761 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 762 qualifier();  
 763 }  
 764 else {  
 765 write("ERROR at line " + lineCount + "! Expected token: : at <Parameter>. Current token: " +token.lexeme);  
 766 throw new RuntimeException();  
 767 }  
 768 }  
 769   
 770 //terminals  
 771 public void **qualifier**() {  
 772 write("<Qualifier> -> integer | boolean");  
 773 if (token.getLexeme().equals("integer")) {  
 774 idType1 = "integer";  
 775 token = lexer();  
 776 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 777 }  
 778 else if (token.getLexeme().equals("boolean")) {   
 779 idType1 = "boolean";   
 780 token = lexer();  
 781 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 782 }  
 783 else {  
 784 write("ERROR at line " + lineCount + "! Expected token: integer | boolean at <Qualifier>. Current token: " +token.getLexeme());  
 785 throw new RuntimeException();  
 786 }  
 787 }  
 788   
 789 //terminals  
 790 public void **body**() {  
 791 write("<Body> -> {<Statement List>}");  
 792 if (token.getLexeme().equals("{")) {   
 793 statementCount++;  
 794 token = lexer();  
 795 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 796 sl();  
 797 if (token.getLexeme().equals("}")) {   
 798 statementCount--;  
 799 token = lexer();  
 800 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 801 }  
 802 else {  
 803 write("ERROR at line " + lineCount + "! Expected token: } at <Body>. Current token: " +token.getLexeme());  
 804 throw new RuntimeException();  
 805 }  
 806 }  
 807 else {  
 808 write("ERROR at line " + lineCount + "! Expected token: { at <Body>. Current token: " +token.getLexeme());  
 809 throw new RuntimeException();  
 810 }  
 811 }  
 812   
 813 public void **odl**() {  
 814 write("<Optional Declaration List> -> <Declaration List> | <Empty>");  
 815 if (token.getLexeme().equals("integer") || token.getLexeme().equals("boolean") || token.getLexeme().equals("floating")) {  
 816 dl();  
 817 }  
 818 }  
 819   
 820 public void **dl**() {  
 821 write("<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>");  
 822 statementCount++;  
 823 declaration();  
 824 if (token.getLexeme().equals(";")) {   
 825 idType1 = ""; //reset after identifier have been successfully declared  
 826 statementCount--;  
 827 token = lexer();  
 828 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 829 if (token.getLexeme().equals("integer") || token.getLexeme().equals("boolean") || token.getLexeme().equals("floating")) {  
 830 dl();  
 831 }  
 832 }  
 833 else {  
 834 write("ERROR at line " + lineCount + "! Expected token: ; at <Declaration List>. Current token: " +token.getLexeme());  
 835 throw new RuntimeException();  
 836 }  
 837 }  
 838   
 839 public void **declaration**() {  
 840 write("<Declaration> -> <Qualifier> <IDs>");  
 841 qualifier();  
 842 ids();  
 843 }  
 844   
 845 //terminals  
 846 public void **ids**() {  
 847 write("<IDs> -> Identifier | Identifier , <IDs>");  
 848 if (token.getToken().equals("Identifier")) {   
 849 if (readToggle) {  
 850 if (lookup(token.getLexeme()) == null) {  
 851 write("ERROR at line " + lineCount + "! Symbol: " + token.getLexeme() + " have not been declared!");  
 852 throw new RuntimeException();  
 853 }  
 854 int address = lookup(token.getLexeme()).getMemory();  
 855 genInstr("POPM ", address);  
 856 }  
 857 else {  
 858 insert(token.getLexeme(), idType1);  
 859 } //insert new identifier into symbol table, method contains automatic validation  
 860 token = lexer();  
 861 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 862 if (token.getLexeme().equals(",")) {  
 863 if (readToggle) {  
 864 genInstr("STDIN", 0);  
 865 }  
 866 token = lexer();  
 867 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 868 ids();  
 869 }  
 870 }  
 871 }  
 872   
 873 public void **sl**() {  
 874 write("<Statement List> -> <Statement> | <Statement> <Statement List>");  
 875 statement();   
 876 if (token.getLexeme().equals("{") || token.getLexeme().equals("if") || token.getLexeme().equals("return") || token.getLexeme().equals("write") ||   
 877 token.getLexeme().equals("read") || token.getLexeme().equals("while") || token.getToken().equals("Identifier")) {  
 878 sl();  
 879 }  
 880 }  
 881   
 882 public void **statement**() {  
 883 write("<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>");  
 884 if (token.getLexeme().equals("{")) {  
 885 compound();  
 886 }  
 887 else if (token.getToken().equals("Identifier")) {  
 888 assign();  
 889 }  
 890 else if (token.getLexeme().equals("if")) {  
 891 productionIf();  
 892 }  
 893 else if (token.getLexeme().equals("return")) {  
 894 productionReturn();  
 895 }  
 896 else if (token.getLexeme().equals("write")) {  
 897 productionWrite();  
 898 }  
 899 else if (token.getLexeme().equals("read")) {  
 900 productionRead();  
 901 }  
 902 else if (token.getLexeme().equals("while")) {  
 903 productionWhile();  
 904 }  
 905 else {  
 906 write("ERROR at line " + lineCount + "! Expected a statement at <Statement>. Current token: " +token.getLexeme());  
 907 throw new RuntimeException();  
 908 }  
 909 }  
 910   
 911 //terminals  
 912 public void **compound**() {  
 913 write("<Compound> -> { <Statement List> }");  
 914 if (token.getLexeme().equals("{")) {  
 915 statementCount++;  
 916 token = lexer();  
 917 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 918 sl();  
 919 if (token.getLexeme().equals("}")) {   
 920 statementCount--;  
 921 token = lexer();  
 922 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 923 }  
 924 else {  
 925 write("ERROR at line " + lineCount + "! Expected token: } at <Compound>. Current token: " +token.getLexeme());  
 926 throw new RuntimeException();  
 927 }  
 928 }  
 929 else {  
 930 write("ERROR at line " + lineCount + "! Expected token: { at <Compound>. Current token: " +token.getLexeme());  
 931 throw new RuntimeException();  
 932 }  
 933 }  
 934   
 935 //terminals  
 936 public void **assign**() {  
 937 write("<Assign> -> Identifier := <Expression> ;");  
 938 if (token.getToken().equals("Identifier")) {  
 939 saved = token;  
 940 statementCount++;  
 941 token = lexer();  
 942 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 943 if (token.getLexeme().equals(":=")) {   
 944 if (lookup(saved.getLexeme()) != null) {  
 945 idType1 = lookup(saved.getLexeme()).getType();  
 946 token = lexer();  
 947 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 948 expression();  
 949 if (token.getLexeme().equals(";")) {  
 950 if (idType1.equals(idType2)) {  
 951 int address = lookup(saved.getLexeme()).getMemory();  
 952 genInstr("POPM ", address);  
 953 statementCount--;  
 954 idType1 = idType2 = ""; //reset after successfully assigning identifers of matching data type  
 955 token = lexer();  
 956 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 957 }  
 958 else {  
 959 write("ERROR at line " + lineCount + "! Data type " + idType1 + " and " + idType2 + " does not match!");  
 960 throw new RuntimeException();  
 961 }  
 962 }  
 963 else {  
 964 write("ERROR at line " + lineCount + "! Expected token: ; at <Assign>. Current token: " +token.getLexeme());  
 965 throw new RuntimeException();  
 966 }  
 967 }  
 968 else {  
 969 write("ERROR at line " + lineCount + "! Symbol: " +saved.getLexeme() + " have not been declared!");  
 970 throw new RuntimeException();  
 971 }  
 972 }  
 973 else {  
 974 write("ERROR at line " + lineCount + "! Expected token: := at <Assign>. Current token: " +token.getLexeme());  
 975 throw new RuntimeException();  
 976 }  
 977 }  
 978 else {  
 979 write("ERROR at line " + lineCount + "! Expected token: Identifier at <Assign>. Current token: " +token.getToken());  
 980 throw new RuntimeException();  
 981 }  
 982 }  
 983   
 984 //terminals  
 985 public void **productionIf**() {  
 986 write("<If> -> if (<Condition>) <Statement> fi | if (<Condition>) <Statement> else <Statement> fi");  
 987 if (token.getLexeme().equals("if")) {   
 988 ifToggle = true;  
 989 statementCount++;  
 990 token = lexer();  
 991 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 992 if (token.getLexeme().equals("(")) {   
 993 token = lexer();  
 994 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 995 condition();  
 996 if (token.getLexeme().equals(")")) {   
 997 token = lexer();  
 998 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
 999 statement();  
1000 backPatch(instructionAddress);  
1001 if (token.getLexeme().equals("else")) {   
1002 token = lexer();  
1003 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1004 statement();  
1005 }  
1006 if (token.getLexeme().equals("fi")) {   
1007 ifToggle = false;  
1008 statementCount--;  
1009 token = lexer();  
1010 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1011 }  
1012 else {  
1013 write("ERROR at line " + lineCount + "! Expected token: fi at <If>. Current token: " +token.getLexeme());  
1014 throw new RuntimeException();  
1015 }  
1016 }  
1017 else {  
1018 write("ERROR at line " + lineCount + "! Expected token: ) at <If>. Current token: " +token.getLexeme());  
1019 throw new RuntimeException();  
1020 }  
1021 }  
1022 else {  
1023 write("ERROR at line " + lineCount + "! Expected token: ( at <If>. Current token: " +token.getLexeme()+"**\n**");  
1024 throw new RuntimeException();  
1025 }  
1026 }  
1027 else {  
1028 write("ERROR at line " + lineCount + "! Expected token: if at <If>. Current token: " +token.getLexeme());  
1029 throw new RuntimeException();  
1030 }  
1031 }  
1032   
1033 //terminals  
1034 public void **productionReturn**() {  
1035 write("<Return> -> return ; | return <Expression> ;");  
1036 if (token.getLexeme().equals("return")) {   
1037 statementCount++;  
1038 token = lexer();  
1039 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1040 if (token.getLexeme().equals("-") || token.getLexeme().equals("(") || token.getToken().equals("Identifier") ||   
1041 token.getToken().equals("Integer") || token.getToken().equals("Real") || token.getLexeme().equals("true") ||   
1042 token.getLexeme().equals("false")) {  
1043 expression();  
1044 if (token.getLexeme().equals(";")) {  
1045 statementCount--;  
1046 token = lexer();  
1047 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1048 }  
1049 else {  
1050 write("ERROR at line " + lineCount + "! Expected token: ; at <Return>. Current token: " +token.getLexeme());  
1051 throw new RuntimeException();  
1052 }  
1053 }  
1054 else if (token.getLexeme().equals(";")){  
1055 statementCount--;  
1056 token = lexer();  
1057 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1058 }  
1059 else {  
1060 write("ERROR at line " + lineCount + "! Expected token: ; | - | (<Expression>) | Identifier | Integer | Real | true | false at <Return>. Current token: " +token.getLexeme());  
1061 throw new RuntimeException();  
1062 }  
1063 }  
1064 else {  
1065 write("ERROR at line " + lineCount + "! Expected token: return at <Return>. Current token: " +token.getLexeme());  
1066 throw new RuntimeException();  
1067 }   
1068 }  
1069   
1070 //terminals  
1071 public void **productionWrite**() {  
1072 write("<Write> -> write (<Expression>) ;");  
1073 if (token.getLexeme().equals("write")) {   
1074 statementCount++;  
1075 token = lexer();  
1076 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1077 if (token.getLexeme().equals("(")) {   
1078 token = lexer();  
1079 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1080 expression();  
1081 if (token.getLexeme().equals(")")) {   
1082 token = lexer();  
1083 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1084 if (token.getLexeme().equals(";")) {   
1085 genInstr("STDOUT", 0);  
1086 statementCount--;  
1087 token = lexer();  
1088 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1089 }  
1090 else {  
1091 write("ERROR at line " + lineCount + "! Expected token: ; at <Write>. Current token: " +token.getLexeme());  
1092 throw new RuntimeException();  
1093 }  
1094 }  
1095 else {  
1096 write("ERROR at line " + lineCount + "! Expected token: ) at <Write>. Current token: " +token.getLexeme());  
1097 throw new RuntimeException();  
1098 }  
1099 }  
1100 else {  
1101 write("ERROR at line " + lineCount + "! Expected token: ( at <Write>. Current token: " +token.getLexeme());  
1102 throw new RuntimeException();  
1103 }  
1104 }  
1105 else {  
1106 write("ERROR at line " + lineCount + "! Expected token: write at <Write>. Current token: " +token.getLexeme());  
1107 throw new RuntimeException();  
1108 }  
1109 }  
1110   
1111 //terminals  
1112 public void **productionRead**() {  
1113 write("<Read> -> read (<IDs>) ;");  
1114 if (token.getLexeme().equals("read")) {   
1115 readToggle = true;  
1116 genInstr("STDIN", 0);  
1117 statementCount++;  
1118 token = lexer();  
1119 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1120 if (token.getLexeme().equals("(")) {   
1121 token = lexer();  
1122 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1123 ids();  
1124 if (token.getLexeme().equals(")")) {   
1125 token = lexer();  
1126 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1127 if (token.getLexeme().equals(";")) {   
1128 readToggle = false;  
1129 statementCount--;  
1130 token = lexer();  
1131 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1132 }  
1133 else {  
1134 write("ERROR at line " + lineCount + "! Expected token: ; at <Read>. Current token: " +token.getLexeme());  
1135 throw new RuntimeException();  
1136 }  
1137 }  
1138 else {  
1139 write("ERROR at line " + lineCount + "! Expected token: ) at <Read>. Current token: " +token.getLexeme());  
1140 throw new RuntimeException();  
1141 }  
1142 }  
1143 else {  
1144 write("ERROR at line " + lineCount + "! Expected token: ( at <Read>. Current token: " +token.getLexeme());  
1145 throw new RuntimeException();  
1146 }  
1147 }  
1148 else {  
1149 write("ERROR at line " + lineCount + "! Expected token: read at <Read>. Current token: " +token.getLexeme());  
1150 throw new RuntimeException();  
1151 }  
1152 }  
1153   
1154 //terminals  
1155 public void **productionWhile**() {  
1156 write("<While> -> while (<Condition>) <Statement>");  
1157 if (token.getLexeme().equals("while")) {   
1158 whileToggle = true;  
1159 addressWhile = instructionAddress;  
1160 genInstr("LABEL", 0);  
1161 token = lexer();  
1162 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1163 if (token.getLexeme().equals("(")) {   
1164 token = lexer();  
1165 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1166 condition();  
1167 if (token.getLexeme().equals(")")) {  
1168 if ((token = lexer()) == null) {  
1169 success = false;  
1170 throw new RuntimeException();  
1171 }  
1172 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1173 statement();  
1174 genInstr("JUMP ", addressWhile);  
1175 backPatch(instructionAddress);  
1176 whileToggle = false;  
1177 }  
1178 else {  
1179 write("ERROR at line " + lineCount + "! Expected token: ) at <While>. Current token: " +token.getLexeme());  
1180 throw new RuntimeException();  
1181 }  
1182 }  
1183 else {  
1184 write("ERROR at line " + lineCount + "! Expected token: ( at <While>. Current token: " +token.getLexeme());  
1185 throw new RuntimeException();  
1186 }  
1187 }  
1188 else {  
1189 write("ERROR at line " + lineCount + "! Expected token: while at <While>. Current token: " +token.getLexeme());  
1190 throw new RuntimeException();  
1191 }  
1192 }  
1193   
1194 public void **condition**() {  
1195 write("<Condition> -> <Expression> <Relop> <Expression>");  
1196 expression();  
1197 relop();  
1198 expression();  
1199 switch(conOp) {  
1200 case "<":   
1201 genInstr("LES ", 0);  
1202 jumpStack.push(instructionAddress);  
1203 genInstr("JUMPZ", 0);  
1204 break;  
1205 case ">":   
1206 genInstr("GRT ", 0);  
1207 jumpStack.push(instructionAddress);  
1208 genInstr("JUMPZ", 0);  
1209 break;  
1210 case "=":  
1211 genInstr("EQU ", 0);  
1212 jumpStack.push(instructionAddress);  
1213 genInstr("JUMPZ", 0);  
1214 break;  
1215 case "/=":  
1216 genInstr("NEQ ", 0);  
1217 jumpStack.push(instructionAddress);  
1218 genInstr("JUMPZ", 0);  
1219 break;  
1220 case "=>":  
1221 genInstr("GEQ ", 0);  
1222 jumpStack.push(instructionAddress);  
1223 genInstr("JUMPZ", 0);  
1224 break;  
1225 case "<=":  
1226 genInstr("LEQ ", 0);  
1227 jumpStack.push(instructionAddress);  
1228 genInstr("JUMPZ", 0);  
1229 break;  
1230 default:  
1231 }  
1232 }  
1233   
1234 //terminals  
1235 public void **relop**() {  
1236 write("<Relop> -> = | /= | > | < | => | <=");  
1237 if (token.getLexeme().equals("=") || token.getLexeme().equals("/=") || token.getLexeme().equals(">") || token.getLexeme().equals("<") ||  
1238 token.getLexeme().equals("=>") || token.getLexeme().equals("<=")) {  
1239 conOp = token.getLexeme();  
1240 token = lexer();  
1241 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1242 }  
1243 else {  
1244 write("ERROR at line " + lineCount + "! Expected token: = | /= | > | < | => | <= at <Relop>. Current token: " +token.getLexeme());  
1245 throw new RuntimeException();  
1246 }  
1247 }  
1248   
1249 public void **expression**() {  
1250 write("<Expression> -> <Term> <Expression'>");  
1251 term();  
1252 expressionprime();  
1253 }  
1254   
1255 //removing left-recursion  
1256 public void **expressionprime**() {  
1257 write("<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>");  
1258 if (token.getLexeme().equals("+")) {   
1259 token = lexer();  
1260 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1261 term();  
1262 genInstr("ADD ", 0);  
1263 expressionprime();  
1264 }  
1265 else if (token.getLexeme().equals("-")) {   
1266 token = lexer();  
1267 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1268 term();  
1269 genInstr("SUB ", 0);  
1270 expressionprime();  
1271 }  
1272 }  
1273   
1274 public void **term**() {  
1275 write("<Term> -> <Factor> <Term'>");  
1276 factor();  
1277 termprime();  
1278 }  
1279   
1280 //removing left-recursion  
1281 public void **termprime**() {  
1282 write("<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>");  
1283 if (token.getLexeme().equals("\*")) {   
1284 token = lexer();  
1285 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1286 factor();  
1287 genInstr("MUL ", 0);  
1288 termprime();  
1289 }  
1290 else if (token.getLexeme().equals("/")) {   
1291 token = lexer();  
1292 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1293 factor();  
1294 genInstr("DIV ", 0);  
1295 termprime();  
1296 }  
1297 }  
1298   
1299 public void **factor**() {  
1300 write("<Factor> -> - <Primary> | <Primary>");  
1301 if (token.getLexeme().equals("-")) {   
1302 token = lexer();  
1303 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1304 }  
1305 primary();  
1306 }  
1307   
1308 public void **primary**() {  
1309 write("<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false");  
1310 if (token.getToken().equals("Identifier")) {   
1311 if (lookup(token.getLexeme()) == null) {  
1312 write("ERROR at line " + lineCount + "! Symbol: " + token.getLexeme() + " have not been declared!");  
1313 throw new RuntimeException();  
1314 }  
1315 idType2 = lookup(token.getLexeme()).getType();  
1316 int address = lookup(token.getLexeme()).getMemory();   
1317 genInstr("PUSHM", address);  
1318 token = lexer();  
1319 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1320 if (token.getLexeme().equals("[")) {   
1321 token = lexer();  
1322 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1323 ids();  
1324 if (token.getLexeme().equals("]")) {   
1325 token = lexer();  
1326 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1327 }  
1328 else {  
1329 write("ERROR at line " + lineCount + "! Expected token: ] at <Primary>. Current token: " +token.getLexeme());  
1330 throw new RuntimeException();  
1331 }  
1332 }  
1333 }  
1334 else if (token.getToken().equals("Integer")) {   
1335 int value = Integer.*parseInt*(token.getLexeme());  
1336 genInstr("PUSHI", value);  
1337 idType2 = "integer";  
1338 token = lexer();  
1339 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1340 }  
1341 else if (token.getLexeme().equals("true") || token.getLexeme().equals("false")) {  
1342 idType2 = "boolean";  
1343 token = lexer();  
1344 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1345 }  
1346 else if (token.getLexeme().equals("(")) {   
1347 token = lexer();  
1348 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1349 expression();  
1350 if (token.getLexeme().equals(")")) {   
1351 token = lexer();  
1352 write("Token: "+token.getToken() + "**\t** Lexeme: " + token.getLexeme());  
1353 }  
1354 else {  
1355 write("ERROR at line " + lineCount + "! Expected token: ) at <Primary> Current token: " +token.getLexeme());  
1356 throw new RuntimeException();  
1357 }  
1358 }  
1359 else {  
1360 write("ERROR at line " + lineCount + "! Expected token: Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false at <Primary> Current token: " +token.getLexeme());  
1361 throw new RuntimeException();  
1362 }  
1363 }  
1364   
1365   
1366   
1367 public static void ***main***(String[] args) {  
1368 String fileNameIn = "TestFile.txt";  
1369 String fileNameOut = "OutputFile.txt";  
1370 LexicalAnalyzer la = new LexicalAnalyzer(fileNameIn,fileNameOut);  
1371 Record currentToken;  
1372 la.printSwitch(true);  
1373 currentToken = la.lexer();  
1374 la.rat17f(currentToken);  
1375 la.close();  
1376 }  
1377 }

**CS323 Assignment 1 Test Cases**

**Test Case 1: < 10 lines,** with declaration error, square is undeclared

**Source:**

%%

integer new;

new := 2;

while (new < 1024)

{ write(square);}

**Output:**

Token: Separator Lexeme: %%

Token: Keyword Lexeme: integer

<Optional Declaration List> -> <Declaration List> | <Empty>

<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>

<Declaration> -> <Qualifier> <IDs>

<Qualifier> -> integer | boolean

Token: Identifier Lexeme: new

<IDs> -> Identifier | Identifier , <IDs>

Token: Separator Lexeme: ;

Token: Identifier Lexeme: new

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Assign> -> Identifier := <Expression> ;

Token: Operator Lexeme: :=

Token: Integer Lexeme: 2

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Separator Lexeme: ;

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Keyword Lexeme: while

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<While> -> while (<Condition>) <Statement>

Token: Separator Lexeme: (

Token: Identifier Lexeme: new

<Condition> -> <Expression> <Relop> <Expression>

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Operator Lexeme: <

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

<Relop> -> = | /= | > | < | => | <=

Token: Integer Lexeme: 1024

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Separator Lexeme: )

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Separator Lexeme: {

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Compound> -> { <Statement List> }

Token: Keyword Lexeme: write

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Write> -> write (<Expression>) ;

Token: Separator Lexeme: (

Token: Identifier Lexeme: square

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

ERROR at line 5! Symbol: square have not been declared!

ERROR! Parsing Unsuccessful!

**Test Case 2: < 20 lines**, **s**ample program from assignment 3 document:

**Source:**

%%

integer i, max, sum;

sum := 0;

i := 1;

read(max);

while(i < max) {

sum := sum + i;

i := i + 1;

}

write(sum + max);

**Output:**

Token: Separator Lexeme: %%

Token: Keyword Lexeme: integer

<Optional Declaration List> -> <Declaration List> | <Empty>

<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>

<Declaration> -> <Qualifier> <IDs>

<Qualifier> -> integer | boolean

Token: Identifier Lexeme: i

<IDs> -> Identifier | Identifier , <IDs>

Token: Separator Lexeme: ,

Token: Identifier Lexeme: max

<IDs> -> Identifier | Identifier , <IDs>

Token: Separator Lexeme: ,

Token: Identifier Lexeme: sum

<IDs> -> Identifier | Identifier , <IDs>

Token: Separator Lexeme: ;

Token: Identifier Lexeme: sum

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Assign> -> Identifier := <Expression> ;

Token: Operator Lexeme: :=

Token: Integer Lexeme: 0

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Separator Lexeme: ;

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Identifier Lexeme: i

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Assign> -> Identifier := <Expression> ;

Token: Operator Lexeme: :=

Token: Integer Lexeme: 1

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Separator Lexeme: ;

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Keyword Lexeme: read

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Read> -> read (<IDs>) ;

Token: Separator Lexeme: (

Token: Identifier Lexeme: max

<IDs> -> Identifier | Identifier , <IDs>

Token: Separator Lexeme: )

Token: Separator Lexeme: ;

Token: Keyword Lexeme: while

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<While> -> while (<Condition>) <Statement>

Token: Separator Lexeme: (

Token: Identifier Lexeme: i

<Condition> -> <Expression> <Relop> <Expression>

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Operator Lexeme: <

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

<Relop> -> = | /= | > | < | => | <=

Token: Identifier Lexeme: max

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Separator Lexeme: )

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Separator Lexeme: {

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Compound> -> { <Statement List> }

Token: Identifier Lexeme: sum

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Assign> -> Identifier := <Expression> ;

Token: Operator Lexeme: :=

Token: Identifier Lexeme: sum

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Operator Lexeme: +

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Identifier Lexeme: i

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Separator Lexeme: ;

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Identifier Lexeme: i

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Assign> -> Identifier := <Expression> ;

Token: Operator Lexeme: :=

Token: Identifier Lexeme: i

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Operator Lexeme: +

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Integer Lexeme: 1

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Separator Lexeme: ;

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Separator Lexeme: }

Token: Keyword Lexeme: write

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Write> -> write (<Expression>) ;

Token: Separator Lexeme: (

Token: Identifier Lexeme: sum

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Operator Lexeme: +

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Identifier Lexeme: max

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Separator Lexeme: )

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Separator Lexeme: ;

Parsing Successful!

///////////SYMBOL TABLE//////////

Symbol: i Memory Address: 10000 Type: integer

Symbol: max Memory Address: 10001 Type: integer

Symbol: sum Memory Address: 10002 Type: integer

///////////INSTRUCTION TABLE//////////

Instruction Address: 0 Operator: Operand: 0

Instruction Address: 1 Operator: PUSHI Operand: 0

Instruction Address: 2 Operator: POPM Operand: 10002

Instruction Address: 3 Operator: PUSHI Operand: 1

Instruction Address: 4 Operator: POPM Operand: 10000

Instruction Address: 5 Operator: STDIN Operand: 0

Instruction Address: 6 Operator: POPM Operand: 10001

Instruction Address: 7 Operator: LABEL Operand: 0

Instruction Address: 8 Operator: PUSHM Operand: 10000

Instruction Address: 9 Operator: PUSHM Operand: 10001

Instruction Address: 10 Operator: LES Operand: 0

Instruction Address: 11 Operator: JUMPZ Operand: 21

Instruction Address: 12 Operator: PUSHM Operand: 10002

Instruction Address: 13 Operator: PUSHM Operand: 10000

Instruction Address: 14 Operator: ADD Operand: 0

Instruction Address: 15 Operator: POPM Operand: 10002

Instruction Address: 16 Operator: PUSHM Operand: 10000

Instruction Address: 17 Operator: PUSHI Operand: 1

Instruction Address: 18 Operator: ADD Operand: 0

Instruction Address: 19 Operator: POPM Operand: 10000

Instruction Address: 20 Operator: JUMP Operand: 7

Instruction Address: 21 Operator: PUSHM Operand: 10002

Instruction Address: 22 Operator: PUSHM Operand: 10001

Instruction Address: 23 Operator: ADD Operand: 0

Instruction Address: 24 Operator: STDOUT Operand: 0

Instruction Address: 25 Operator: Operand: 0

**Test Case 3: > 20 lines**

**Source:**

%%

integer low, high, step#xy;

boolean hot;

read(low,high,step#xy);

if(step#xy = high)

{

write(high);

}

else

{

write(low);

}

fi

hot := false;

while(low<high)

{

write (low);

write (high + low);

low := low + step#xy;

read(high, low);

}

**Output:**

Token: Separator Lexeme: %%

Token: Keyword Lexeme: integer

<Optional Declaration List> -> <Declaration List> | <Empty>

<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>

<Declaration> -> <Qualifier> <IDs>

<Qualifier> -> integer | boolean

Token: Identifier Lexeme: low

<IDs> -> Identifier | Identifier , <IDs>

Token: Separator Lexeme: ,

Token: Identifier Lexeme: high

<IDs> -> Identifier | Identifier , <IDs>

Token: Separator Lexeme: ,

Token: Identifier Lexeme: step#xy

<IDs> -> Identifier | Identifier , <IDs>

Token: Separator Lexeme: ;

Token: Keyword Lexeme: boolean

<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>

<Declaration> -> <Qualifier> <IDs>

<Qualifier> -> integer | boolean

Token: Identifier Lexeme: hot

<IDs> -> Identifier | Identifier , <IDs>

Token: Separator Lexeme: ;

Token: Keyword Lexeme: read

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Read> -> read (<IDs>) ;

Token: Separator Lexeme: (

Token: Identifier Lexeme: low

<IDs> -> Identifier | Identifier , <IDs>

Token: Separator Lexeme: ,

Token: Identifier Lexeme: high

<IDs> -> Identifier | Identifier , <IDs>

Token: Separator Lexeme: ,

Token: Identifier Lexeme: step#xy

<IDs> -> Identifier | Identifier , <IDs>

Token: Separator Lexeme: )

Token: Separator Lexeme: ;

Token: Keyword Lexeme: if

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<If> -> if (<Condition>) <Statement> fi | if (<Condition>) <Statement> else <Statement> fi

Token: Separator Lexeme: (

Token: Identifier Lexeme: step#xy

<Condition> -> <Expression> <Relop> <Expression>

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Operator Lexeme: =

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

<Relop> -> = | /= | > | < | => | <=

Token: Identifier Lexeme: high

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Separator Lexeme: )

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Separator Lexeme: {

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Compound> -> { <Statement List> }

Token: Keyword Lexeme: write

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Write> -> write (<Expression>) ;

Token: Separator Lexeme: (

Token: Identifier Lexeme: high

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Separator Lexeme: )

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Separator Lexeme: ;

Token: Separator Lexeme: }

Token: Keyword Lexeme: else

Token: Separator Lexeme: {

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Compound> -> { <Statement List> }

Token: Keyword Lexeme: write

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Write> -> write (<Expression>) ;

Token: Separator Lexeme: (

Token: Identifier Lexeme: low

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Separator Lexeme: )

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Separator Lexeme: ;

Token: Separator Lexeme: }

Token: Keyword Lexeme: fi

Token: Identifier Lexeme: hot

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Assign> -> Identifier := <Expression> ;

Token: Operator Lexeme: :=

Token: Keyword Lexeme: false

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Separator Lexeme: ;

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Keyword Lexeme: while

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<While> -> while (<Condition>) <Statement>

Token: Separator Lexeme: (

Token: Identifier Lexeme: low

<Condition> -> <Expression> <Relop> <Expression>

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Operator Lexeme: <

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

<Relop> -> = | /= | > | < | => | <=

Token: Identifier Lexeme: high

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Separator Lexeme: )

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Separator Lexeme: {

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Compound> -> { <Statement List> }

Token: Keyword Lexeme: write

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Write> -> write (<Expression>) ;

Token: Separator Lexeme: (

Token: Identifier Lexeme: low

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Separator Lexeme: )

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Separator Lexeme: ;

Token: Keyword Lexeme: write

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Write> -> write (<Expression>) ;

Token: Separator Lexeme: (

Token: Identifier Lexeme: high

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Operator Lexeme: +

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Identifier Lexeme: low

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Separator Lexeme: )

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Separator Lexeme: ;

Token: Identifier Lexeme: low

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Assign> -> Identifier := <Expression> ;

Token: Operator Lexeme: :=

Token: Identifier Lexeme: low

<Expression> -> <Term> <Expression'>

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Operator Lexeme: +

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Identifier Lexeme: step#xy

<Term> -> <Factor> <Term'>

<Factor> -> - <Primary> | <Primary>

<Primary> -> Identifier | Integer | Identifier [<IDs>] | (<Expression>) | true | false

Token: Separator Lexeme: ;

<Term'> -> \* <Factor> <Term'> | / <Factor> <Term'> | <Empty>

<Expression'> -> + <Term> <Expression'> | - <Term> <Expression'> | <Empty>

Token: Keyword Lexeme: read

<Statement List> -> <Statement> | <Statement> <Statement List>

<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

<Read> -> read (<IDs>) ;

Token: Separator Lexeme: (

Token: Identifier Lexeme: high

<IDs> -> Identifier | Identifier , <IDs>

Token: Separator Lexeme: ,

Token: Identifier Lexeme: low

<IDs> -> Identifier | Identifier , <IDs>

Token: Separator Lexeme: )

Token: Separator Lexeme: ;

Token: Separator Lexeme: }

Parsing Successful!

///////////SYMBOL TABLE//////////

Symbol: low Memory Address: 10000 Type: integer

Symbol: high Memory Address: 10001 Type: integer

Symbol: step#xy Memory Address: 10002 Type: integer

Symbol: hot Memory Address: 10003 Type: boolean

///////////INSTRUCTION TABLE//////////

Instruction Address: 0 Operator: Operand: 0

Instruction Address: 1 Operator: STDIN Operand: 0

Instruction Address: 2 Operator: POPM Operand: 10000

Instruction Address: 3 Operator: STDIN Operand: 0

Instruction Address: 4 Operator: POPM Operand: 10001

Instruction Address: 5 Operator: STDIN Operand: 0

Instruction Address: 6 Operator: POPM Operand: 10002

Instruction Address: 7 Operator: PUSHM Operand: 10002

Instruction Address: 8 Operator: PUSHM Operand: 10001

Instruction Address: 9 Operator: EQU Operand: 0

Instruction Address: 10 Operator: JUMPZ Operand: 13

Instruction Address: 11 Operator: PUSHM Operand: 10001

Instruction Address: 12 Operator: STDOUT Operand: 0

Instruction Address: 13 Operator: PUSHM Operand: 10000

Instruction Address: 14 Operator: STDOUT Operand: 0

Instruction Address: 15 Operator: POPM Operand: 10003

Instruction Address: 16 Operator: LABEL Operand: 0

Instruction Address: 17 Operator: PUSHM Operand: 10000

Instruction Address: 18 Operator: PUSHM Operand: 10001

Instruction Address: 19 Operator: LES Operand: 0

Instruction Address: 20 Operator: JUMPZ Operand: 36

Instruction Address: 21 Operator: PUSHM Operand: 10000

Instruction Address: 22 Operator: STDOUT Operand: 0

Instruction Address: 23 Operator: PUSHM Operand: 10001

Instruction Address: 24 Operator: PUSHM Operand: 10000

Instruction Address: 25 Operator: ADD Operand: 0

Instruction Address: 26 Operator: STDOUT Operand: 0

Instruction Address: 27 Operator: PUSHM Operand: 10000

Instruction Address: 28 Operator: PUSHM Operand: 10002

Instruction Address: 29 Operator: ADD Operand: 0

Instruction Address: 30 Operator: POPM Operand: 10000

Instruction Address: 31 Operator: STDIN Operand: 0

Instruction Address: 32 Operator: POPM Operand: 10001

Instruction Address: 33 Operator: STDIN Operand: 0

Instruction Address: 34 Operator: POPM Operand: 10000

Instruction Address: 35 Operator: JUMP Operand: 16

Instruction Address: 36 Operator: Operand: 0