COVER PAGE

CS323 Programming Assignments

1. Your Name:	**PHUC LE**
2. Assignment Number:	3
3. Due Date	**Sunday, December 11, 2016**
4. Turn-In Date	**Wednesday, December 7, 2016**
5. Executable FileName:	**PhucLe_Assignment3.exe**
6. LabRoom	**CS-200, PC: 2d**
7. OS	**Windows 10**

GRADE:

COMMENTS:

1. Problem Statement:

This project is to use RDP method to implement a top-down parser as a syntax analyzer which input is a file containing simplified Rat16F source code and output is a file containing a symbol table handling and a table of assembly code generated.

2. How to Use My Program:


PhucLe_Assignment3

Simply click on "**PhucLe_Assignment3.exe**" in **Executable** folder, then click on 

to select the source code file; the program will display the result and write down the result to an output file.

In case the program doesn't start, there are three alternative methods to run my program:

- Method #1: copy folder **Executable** to desktop or anywhere else. Next, get in to Windows Command line, use ***cd*** command to change to the directory to **Executable**, and finally type "**java –jar app.jar**"
- Method #2: Open Netbeans IDE, open project "**Assignment3_PhucLe**," open file "**src\Program.java**" and click on green button  or press **Shift-F6** to run my program
- Method #3: double click on **app.jar** file.

3. Designing of My Program:

After removing left recursions and back-tracking, the productions rules are the followings:

<Rat16F> →$$ $$ <Opt Declaration List> <Statement List> $$

<Qualifier> → integer | boolean | real

<Body> → { <Statement List> }

<Opt Declaration List> → <Declaration List> | <Empty>

<Declaration List> → <Declaration> ; | <Declaration> ; <Declaration List>

<Declaration> → <Qualifier> <IDs>

<IDs> → <Identifier> | <Identifier> , <IDs>

<Statement List> → <Statement> | <Statement> <Statement List>

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

<Compound> → { <Statement List> }

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

<If> → if (<Condition>) <Statement> <If Prime>

<If Prime> → endif | else <Statement> endif

<Return> → return ; | return <Expression> ;

<Write> → print (<Expression>);

<Read> → read (<IDs>);

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

<Condition> → <Expression> <Relop> <Expression>

<Relop> → = | > | < | => | <= | !=

<Expression> → <Term> <Expression Prime>

<Expression Prime> → + <Term><Expression Prime> | - <Term><Expression Prime> | <empty>

<Term> → <Factor> <Term Prime>

<Term Prime> → * <Factor> <Term Prime> | / <Factor> <Term Prime> | <empty>

<Factor> → <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false | - <Factor>

22<Empty> → ε

4. **Limitation:** The error messages needs to be more detailed
5. **Shortcoming:** None

Source code of my program is mainly found in file

SourceCode/Assignment3_PhucLe\src\SyntaxAnalyzer.java

```
import java.util.ArrayList;
import java.util.List;

public class AssemblyGenerator {
    LexicalAnalyzer pro = new LexicalAnalyzer();
    List<Token> tokens = new ArrayList<>();
    List<Symbol> symbols = new ArrayList<>();
    List<Instruction> instructions = new ArrayList<>();
    int nextTokenPosition = 0;
    Token currentToken = null;
    int errorPosition = -1;
    String errorMessage = "";
    String qualifier_temp = "";
    boolean isRead = false;
    int jumpstack = -1;
    String acceptedType = "";


    public AssemblyGenerator() {
    }

    public List<Symbol> getSymbols() {
        return symbols;
    }

    public boolean doesSymbolExist(String identifier) {
        return this.symbols.contains(new Symbol(identifier, 0, null, 0));
    }

    public int findSymbol(String identifier) {
        for (int i = 0; i < symbols.size(); i++) {
            if (symbols.get(i).getIdentifier().equals(identifier)) {
                return i;
            }
        }
        return -1;
    }

    public int insertSymbol(Symbol symbol) {
        int index = findSymbol(symbol.getIdentifier());
        if (index >= 0) {
            if (errorMessage.equals("")) {
                errorMessage ="ERROR at line "+symbol.getLine()+": variable '"+
symbol.getIdentifier() + "' has been declared before";
            }
            return -1;
        } else {
            symbol.setMemoryLocation(6000 + symbols.size());
            this.symbols.add(symbol);
            return symbols.size() - 1;
        }
    }

    public int insertInstruction(Instruction instruction) {
        instruction.setAddress(instructions.size() + 1);
```

```
        this.instructions.add(instruction);
        return instructions.size() - 1;
    }

    public List<Instruction> getInstructions() {
        return instructions;
    }

    public void input(String source, int lineNumber) {
        tokens.addAll(pro.tokenize(source, lineNumber));
    }

    public void lexer() {
        if (tokens.size() > 0 && nextTokenPosition < tokens.size()) {
            currentToken = tokens.get(nextTokenPosition);
            nextTokenPosition++;
        } else {
            currentToken = null;
        }
    }

    public String getResult() {
        String result = "";
        for (Token token : tokens) {
            result += token.toString() + "\n";
        }
        return result;
    }

    public void func_rat16f() {
        lexer();
        if (isToken("$$")) {
            lexer();
            if (isToken("$$")) {
                func_opt_declaration_list();
                func_statement_list();
                lexer();
                if (isToken("$$")) {
                    // not necessary
                }
            }
        }
    }

    private void func_qualifiers() {
        lexer();
        if (currentToken == null) {
            nextTokenPosition--;
        }
        if (isToken("integer") || isToken("boolean") || isToken("real")) {
            qualifier_temp = currentToken.content;
        } else {
            nextTokenPosition--;
        }
    }

    private void func_opt_declaration_list() {
        int pointer = nextTokenPosition;
        lexer();
```

```
        if (!(isToken("integer") || isToken("boolean") || isToken("real"))) {
            nextTokenPosition--;
        } else {
            nextTokenPosition--;
            func_declaration_list();
        }
    }

    private void func_declaration_list() {
        func_declaration();
        lexer();
        if (isToken(";")) {
            lexer();
            if (currentToken != null) {
                if (!(currentToken.tokenType == LexicalAnalyzer.Type.IDENTIFIER ||
isToken("if") || isToken("return") || isToken("print") || isToken("read") ||
isToken("while"))) {
                    nextTokenPosition--;
                    func_declaration_list();
                } else {
                    nextTokenPosition--;
                }
            }
        }
    }

    private void func_declaration() {
        func_qualifiers();
        func_ids();
    }

    private void func_ids() {
        int pointer = nextTokenPosition;
        lexer();
        if (currentToken.tokenType == LexicalAnalyzer.Type.IDENTIFIER) {
            Token save = currentToken;
            lexer();
            if (isToken(")") || isToken(";") || isToken(":") || isToken("]")) {
                nextTokenPosition--;
                if (isRead) {
                    int index = findSymbol(save.content);
                    if (index < 0) {
                        errorMessage = "ERROR at line " + save.lineNumber + " -
variable " + save.content + " had not been declared before used";
                        return;
                    } else {
                        insertInstruction(new Instruction(0, "STDIN", ""));
                        insertInstruction(new Instruction(0, "POPM",
getAddress(save.content) + ""));
                        isRead = false;
                    }
                } else if (qualifier_temp.trim().equals("integer") ||
qualifier_temp.trim().equals("real") || qualifier_temp.trim().equals("boolean")) {
                    int insertedIndex = insertSymbol(new
Symbol(tokens.get(nextTokenPosition - 1).content, 0, qualifier_temp,
tokens.get(nextTokenPosition - 1).lineNumber));
                }
            } else if (isToken(",")) {
                  if (isRead) {
```

```
                    int index = findSymbol(save.content);
                    if (index < 0) {
                        errorMessage = "ERROR at line " + save.lineNumber + " -
variable " + save.content + " had not been declared before used";
                        return;
                    } else {
                        insertInstruction(new Instruction(0, "STDIN", ""));
                        insertInstruction(new Instruction(0, "POPM",
getAddress(save.content) + ""));
                        func_ids();
                    }
                } else if (qualifier_temp.trim().equals("integer") ||
qualifier_temp.trim().equals("real") || qualifier_temp.trim().equals("boolean")) {
                    int insertedIndex = insertSymbol(new
Symbol(tokens.get(nextTokenPosition - 2).content, 0, qualifier_temp,
tokens.get(nextTokenPosition - 2).lineNumber));
                    func_ids();
                } else {
                    nextTokenPosition--;
                }
            }
        } else {
            nextTokenPosition = pointer;
        }

    }

    private void func_statement_list() {
        int pointer = nextTokenPosition;
        func_statement();
        lexer();
        if ((isToken("}") && nextTokenPosition < (tokens.size())) || (isToken("$$") &&
nextTokenPosition == (tokens.size()))) {
            nextTokenPosition--;
        } else {
            nextTokenPosition--;
            func_statement_list();
        }
    }

    private void func_statement() {
        lexer();
        Token save = currentToken;
        nextTokenPosition--;
        if (save == null) {
            return;
        }
        if (save.content.equals("{")) {
            func_compound();
        } else if (save.tokenType == LexicalAnalyzer.Type.IDENTIFIER) {
            func_assign();
        } else if (save.content.equals("if")) {
            func_if();
        } else if (save.content.equals("print")) {
            func_write();
        } else if (save.content.equals("read")) {
            func_read();
        } else if (save.content.equals("while")) {
            System.out.println("WHILE");
```

```
            func_while();
        } else {
            return;
        }

    }

    private void func_compound() {
        lexer();
        if (isToken("{")) {
            func_statement_list();
            lexer();
            if (isToken("}")) {
                return;
            }
        }
    }

    private void func_assign() {
        int pointer = nextTokenPosition;
        lexer();
        if (currentToken.tokenType == LexicalAnalyzer.Type.IDENTIFIER) {
            Token save = currentToken;
            int index = findSymbol(save.content);
            if (index < 0) {
                nextTokenPosition = tokens.size();
                if (errorMessage.equals("")) {
                    errorMessage = "ERROR at line " + save.lineNumber + " - variable "
+ save.content + " had not been declared before used";
                }
                return;
            } else {
                acceptedType = symbols.get(index).getType();
                lexer();
                if (isToken(":=")) {
                    func_expression();
                    insertInstruction(new Instruction(0, "POPM", "" +
getAddress(save.content)));
                    getAddress(save.content);
                    lexer();
                    if (isToken(";")) {
                        return;
                    }
                }
            }
        }
        nextTokenPosition = pointer;
    }

    private void func_return() {

    }

    private void func_write() {
        int pointer = nextTokenPosition;
        lexer();
        if (currentToken.tokenType == LexicalAnalyzer.Type.KEYWORD && isToken("print"))
{
```

```
            lexer();
            if (isToken("(")) {
                func_expression();
                lexer();
                if (isToken(")")) {
                    lexer();
                    if (isToken(";")) {
                        insertInstruction(new Instruction(0, "STDOUT", ""));
                    }
                }
            }
        } else {
            nextTokenPosition = pointer;
        }
    }

    private void func_read() {
        int pointer = nextTokenPosition;
        lexer();
        if (isToken("read")) {
            lexer();
            if (isToken("(")) {
                isRead = true;
                func_ids();
                lexer();
                if (isToken(")")) {
                    lexer();
                    if (isToken(";")) {
                        return;
                    }
                }
            }
            isRead = false;
        } else {
            nextTokenPosition = pointer;
            isRead = false;
        }
    }

    private void func_while() {
        int pointer = nextTokenPosition;
        lexer();
        if (isToken("while")) {
            insertInstruction(new Instruction(0, "LABEL", ""));
            int addr = this.instructions.size();
            lexer();
            if (isToken("(")) {
                func_condition();
                lexer();
                if (isToken(")")) {
                    func_statement();
                    insertInstruction(new Instruction(0, "JUMP", "" + addr));
                    back_patch();
                } else {
                    nextTokenPosition--;
                    return;
                }
            }
        }
```

```
    }

    private void func_condition() {
        acceptedType = "";
        func_expression();
        lexer();
        Token save = currentToken;
        String token = currentToken.content;
        func_expression();

        if (token.equals("<")) {
            insertInstruction(new Instruction(0, "LES", ""));

        } else if (token.equals(">")) {
            insertInstruction(new Instruction(0, "GRT", ""));
        } else if (token.equals(">")) {
            insertInstruction(new Instruction(0, "GRT", ""));
        } else if (token.equals("=")) {
            insertInstruction(new Instruction(0, "EQU", ""));
        } else if (token.equals("=>")) {
            insertInstruction(new Instruction(0, "GET", ""));
        } else if (token.equals("<=")) {
            insertInstruction(new Instruction(0, "LET", ""));
        } else if (token.equals("!=")) {
            insertInstruction(new Instruction(0, "NEQ", ""));
        } else {

        }
        insertInstruction(new Instruction(0, "JUMPZ", "TO-BE-UPDATED"));
        push_jumpstack();
    }

    private void func_relop() {
        lexer();
        if (isToken("=")) {

        } else if (isToken(">")) {

        } else if (isToken("<")) {

        } else if (isToken("=>")) {

        } else if (isToken("<=")) {

        } else if (isToken("!=")) {

        } else {
            nextTokenPosition--;
            return;
        }

    }

    private void func_factor() {
        int pointer = nextTokenPosition;
        lexer();

        if (isToken("-")) {
```

```
            func_factor();
        } else if (isToken("(")) {
            func_expression();
            lexer();
            if (isToken(")")) {
                //return true;
            } else {
                //return false;
            }

        } else if (isToken("true") || isToken("false")) {
            if (acceptedType.equals("")) {
                acceptedType = "boolean";
            }
            if (!acceptedType.equals("boolean")) {
                errorMessage = "ERROR at line " + currentToken.lineNumber + " TOKEN: '"
+ currentToken.content + "' - Types boolean and " + acceptedType + " are mismatched";
                return;
            }

        } else if (currentToken.tokenType == LexicalAnalyzer.Type.INTEGER) {
            if (acceptedType.equals("")) {
                acceptedType = "integer";
            }
            if (!(acceptedType.equals("integer"))) {
                errorMessage = "ERROR at line " + currentToken.lineNumber + " TOKEN: '"
+ currentToken.content + "' - Types integer and " + acceptedType + " are mismatched";
                return;
            }
            insertInstruction(new Instruction(0, "PUSHI", currentToken.content + ""));
        } else if (currentToken.tokenType == LexicalAnalyzer.Type.REAL) {
            if (acceptedType.equals("")) {
                acceptedType = "real";
            }
            if (!(acceptedType.equals("real"))) {
                errorMessage = "ERROR at line " + currentToken.lineNumber + " TOKEN: '"
+ currentToken.content + "' - Types real and " + acceptedType + " are mismatched";
                return;
            }
            insertInstruction(new Instruction(0, "PUSHI", currentToken.content + ""));
        } else if (currentToken.tokenType == LexicalAnalyzer.Type.IDENTIFIER) {
            int index = findSymbol(currentToken.content);
            if (index < 0) {
                errorMessage = "ERROR at line " + currentToken.lineNumber + " -
variable '" + currentToken.content + "' had not been declared before used";
                return;
            } else {
                String type = symbols.get(index).getType();
                 if (acceptedType.equals("")) {
                acceptedType = type;
                   }
                if (acceptedType.equals("real")) {

                    if (!(type.equals("real"))) {
                        errorMessage = "ERROR at line " + currentToken.lineNumber + "
TOKEN: '" + currentToken.content + "' - Types " + type + " and " + acceptedType + " are
mismatched";
                        return;
                    }
```

```
                } else if (acceptedType.equals("integer")) {

                    if (!(type.equals("integer"))) {
                        errorMessage = "ERROR at line " + currentToken.lineNumber + "
TOKEN: '" + currentToken.content + "' - Types " + type + " and " + acceptedType + " are
mismatched";
                        return;
                    }
                } else if (!(type.equals("boolean"))) {
                    errorMessage = "ERROR at line " + currentToken.lineNumber + "
TOKEN: '" + currentToken.content + "' - Types " + type + " and " + acceptedType + " are
mismatched";
                    return;
                }
                insertInstruction(new Instruction(0, "PUSHM",
getAddress(currentToken.content) + ""));
            }
        }
    }

    private void func_if() {
        lexer();
        if (isToken("if")) {
            lexer();
            if (isToken("(")) {
                func_condition();
                lexer();
                if (isToken(")")) {
                    func_statement();
                    func_if_prime();
                    back_patch();

                } else {
                    nextTokenPosition--;
                    return;
                }
            }
        }

    }

    private void func_if_prime() {
        lexer();
        if (isToken("endif")) {
            return;
        } else if (isToken("else")) {
            func_statement();
            lexer();
            if (isToken("endif")) {
                return;
                // return true;
            } else {
                nextTokenPosition--;
                return;
            }
        }
    }

    private void func_expression() {
```

```
        func_term();
        func_expression_prime();
    }

    private void func_expression_prime() {
        lexer();
        if (isToken("+")) { //ADD
            func_term();
            insertInstruction(new Instruction(0, "ADD", ""));
            func_expression_prime();
        } else if (isToken("-")) { //SUB
            func_term();
            insertInstruction(new Instruction(0, "SUB", ""));
            func_expression_prime();
        } else {
            nextTokenPosition--;
        }
    }

    private void func_term() {
        func_factor();
        func_term_prime();
    }

    private void func_term_prime() {
        lexer();
        if (isToken("*")) {
            func_factor();
            func_term_prime();
            insertInstruction(new Instruction(0, "MUL", ""));
        } else if (isToken("/")) {
            //DIV
            func_factor();
            func_term_prime();
            insertInstruction(new Instruction(0, "DIV", ""));
        } else {
            nextTokenPosition--;
        }

    }

    private boolean isToken(String token) {
        if (currentToken != null) {
            return currentToken.content.trim().equals(token);
        }
        return false;
    }

    private int getAddress(String identifier) {
        for (Symbol s : symbols) {
            if (s.getIdentifier().equals(identifier)) {
                return s.getMemoryLocation();
            }
        }
        return -1;
    }

    private void printInstructions() {
        int i = 1;
```

```
        for (Instruction instruction : instructions) {
            System.out.println(instruction);
            i++;
        }
    }

    private void push_jumpstack() {
        jumpstack = instructions.size() - 1;
    }

    private int pop_jumstack() {
        return jumpstack;
    }

    private void back_patch() {
        instructions.get(jumpstack).setOperand(instructions.size() + 1 + "");
    }

    public String getErrorMessage() {
        return errorMessage;
    }
}
```

SAMPLE INPUT #1:

```
$$

$$
      real rate;
      real miles, kilometers;
      read (miles, kilometers);

      if(kilometers => 1600.00){
            rate:=1.6;
            print (miles - kilometers*rate);
      }
      endif
$$
```

SAMPLE OUTPUT #1:

```
rate          6000     real
miles         6001     real
kilometers    6002     real


----------------------------


1     STDIN
2     POPM     6001
3     STDIN
4     POPM     6002
5     PUSHM    6002
6     PUSHI    1600.00
7     GET
8     JUMPZ    17
9     PUSHI    1.6
10    POPM     6000
11    PUSHM    6001
12    PUSHM    6002
13    PUSHM    6000
14    MUL
15    SUB
16    STDOUT
```

SAMPLE INPUT #2:

```
$$

$$
      integer start, step;

      start :=    0;
      read(step);
      while (start != 100)
      {
            start := start * step;
            step := step + k;
            print (start+step);
            print (start+(step*2));
      }
$$
```

SAMPLE OUTPUT #2:

```
start        6000     integer
step         6001     integer
ERROR at line 11 - variable 'k' had not been declared before used
```

SAMPLE INPUT #3:

```
$$

$$
      boolean isBig;
      integer i, max, sum;
      i := 1;
      isBig:=false;
      read (max);
      while (i != max){
            sum := sum +i;
            i := i+1;
      }

      if(sum>5000){
            isBig:=true;
            sum:=5000;
      }
      endif

      print(sum +max);

$$
```

SAMPLE OUTPUT #3:

```
isBig          6000     boolean
i              6001     integer
max            6002     integer
sum            6003     integer

----------------------------

1     PUSHI    1
2     POPM     6001
3     POPM     6000
4     STDIN
5     POPM     6002
6     LABEL
7     PUSHM    6001
8     PUSHM    6002
9     NEQ
10    JUMPZ    20
11    PUSHM    6003
12    PUSHM    6001
13    ADD
14    POPM     6003
15    PUSHM    6001
16    PUSHI    1
17    ADD
18    POPM     6001
19    JUMP     6
20    PUSHM    6003
21    PUSHI    5000
22    GRT
23    JUMPZ    27
24    POPM     6000
25    PUSHI    5000
26    POPM     6003
27    PUSHM    6003
28    PUSHM    6002
29    ADD
30    STDOUT
```

SAMPLE INPUT #4:

```
$$

$$
      integer i, max, sum;

      sum := 0;
      i := 1;
      read ( max);
      while (i < max) {
            sum := sum + i;
            i := i + 1;
      }
      print (sum+max);
$$
```

SAMPLE OUTPUT #4:

```
i            6000     integer
max          6001     integer
sum          6002     integer

---------------------------

1    PUSHI    0
2    POPM     6002
3    PUSHI    1
4    POPM     6000
5    STDIN
6    POPM     6001
7    LABEL
8    PUSHM    6000
9    PUSHM    6001
10   LES
11   JUMPZ    21
12   PUSHM    6002
13   PUSHM    6000
14   ADD
15   POPM     6002
16   PUSHM    6000
17   PUSHI    1
18   ADD
19   POPM     6000
20   JUMP     7
21   PUSHM    6002
22   PUSHM    6001
23   ADD
24   STDOUT
```

SAMPLE INPUT #5:

```
$$

$$
      integer i, max, sum;
      real sum;
      sum := 0;
      i := 1;
      read ( max);
      while (i < max) {
            sum := sum + i;
            i := i + 1;
      }
      print (sum+max);
$$
```

SAMPLE OUTPUT #5:

```
i            6000     integer
max          6001     integer
sum          6002     integer
ERROR at line 5: variable 'sum' has been declared before
```

SAMPLE INPUT #6:

```
$$

$$
      integer i, max;
      real sum;

      sum := 0.0;
      i := 1;
      read ( max);
      while (i < max) {
            sum := sum + i;
            i := i + 1;
      }
      print (sum+max);
$$
```

SAMPLE OUTPUT #6:

```
i            6000     integer
max          6001     integer
sum          6002     real
ERROR at line 14 TOKEN: 'sum' - Types real and integer are mismatched
```