# 吉林大学

# 软件学院

2016 级

《编译原理程设计》

总结报告

| 实验题目 | 2016 级《编译原理课程设计》总结报告 | | |
|---|---|---|---|
| 实验项目 | SNL 程序设计语言编译程序 | | |
| 实验地点 | 计算机楼 A108 | 机器编号 | |
| 指导教师 | 申春 | 实验时间 | 2019 年 06 月 21 日 |

| 小组成员 | 学号 | 任务分工 |
|---|---|---|
| 何宇鹏 | 55160121 | 词法分析/LL(1)方法/GUI 界面 |
| | | |
| | | |

# 一、实验目的

- 通过对 SNL 编译程序的学习和动手实践，使学生可以更加深入、全面地掌握编译程序的工作原理和实现技术；
- 培养大型软件的程序设计方法。

# 二、实验内容

1. 设计并实现 SNL 程序设计语言的编译程序；
2. 两个程序
   - 词法分析模块
   - 语法分析模块（二选一）
     - 递归下降方法
     - LL(1)方法

# 三、设计思想

**3.1 实验模块之间的关系**

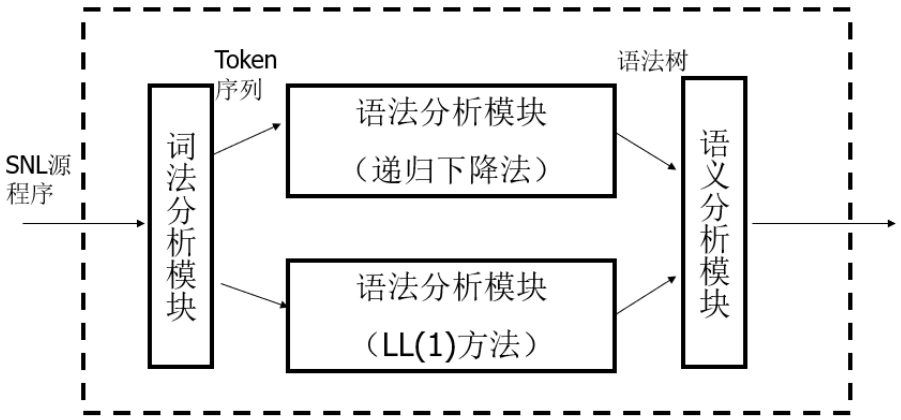


图 1-实验模块之间的关系

### 3.2 词法分析模块

词法分析是依据语言的此法规则，扫描源程序的字符序列，识别每一个单词及其种类，并将其表示成所谓的机内表示 Token 记号形式。

词法分析中，最重要的一些问题是：Token 的定义、单词的分类方法以及自动机的实现。

通过实验文档《SNL 语言介绍》可知 SNL 语言的词法，包括字符表、单词分类等等。文档中给出了 SNL 语言的字符分类：

- 保留字：program、procedure、type、if、then、integer 等；
- 标识符：变量名等；
- 常量：数字等；
- 特殊符号等：+、-、[、<等。

因此首先定义 **TokenType** 枚举类型，用于列举出所有的 Token 类型。

然后定义 **Token** 类，此类可以表示一个四元组<行，列，Token 类型，值>。例如源文件中的一个单词“program”可以表示为< 1,7, PROGRAM, program >，即在第一行第七列识别出一个词法单元 PROGRAM，它在源文件中的写法是 program。

有了词法单元的最小表示形式，那么如何识别出词法单元呢？

利用自动机的思想，对于源文件中的每一个字符，进行逐个读入，再根据当前的自动机状态来判断下一步是应该继续读入还是应该识别出词法单元。也就是说自动机对于同一个输入字符，会根据自身的状态而表现出不同的行为。

定义 **Lexer** 类，称为词法分析类，也是自动机的实现类，其内有枚举类型 **StateEnum**，用于表示当前自动机的状态。自动机最初是 NOEMAL（正常）状态，若遇到的第一个字符是字母时，会转入 INID（标识符）状态；若遇到的第一个字符是数字，则会转入 INNUM（数字）状态。进入 INID 状态的自动机，只能接受字符或数字，如果遇到其他字符，则将已接受的字符串标记为标识符，接着判断该字符串是否为保留字，从而最终确定该字符串的类型。而进入 INNUM 状态的自动机，只能接受数字，直到遇到其他字符时才终止。以上是自动机的简单设计思想。

Lexer 类中有 getResult 方法，可以将输入字符流不断地送入自动机进行匹配，最终会得到一个 Token 序列。我们用 **LexerResult** 类来封装这一结果。

如此一来，词法分析模块基本结束。

### 3.3 语法分析模块（LL(1)方法）

对于 LL(1)方法，语法分析模块需要输入 Token 序列，并给出该序列的语法树和语法检查结果。为了将语法树输出为以图 2 的表示形式，我们需要定义一种特殊的树节点。

```
ProK
      PheadK  p
      TypeK
          DecK  IntegerK  t1
      VarK
          DecK  IntegerK  v1  v2
      ProcDecK  q
          DecK value param:  IntegerK  i
          VarK
              DecK  IntegerK  a
          StmLk
```

图 2-实验要求的语法树的输出实例

定义树节点 **TreeNode**，该类存储了 3 个数据成员，分别是：

- siblings：连接兄弟姐妹节点；
- children：连接第一个孩子节点；
- value：存储节点的值。

例如，对于本实验中的上下文无关文法：

Program ::= ProgramHead DeclarePart ProgramBody .

可以连接成如下形式：

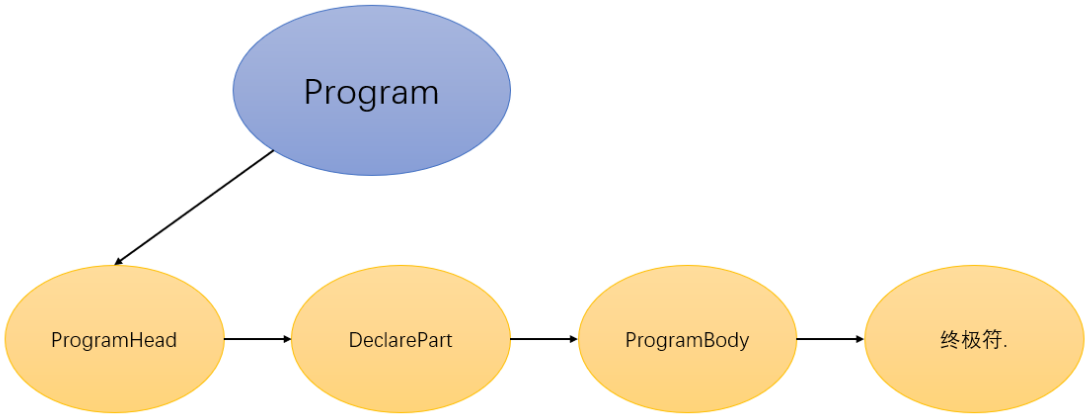



图 3-树的连接形式示意图

可以看出，Program 是整个程序的根节点，而 Program 可以推出 4 个子成分，分别是 ProgramHead、DeclarePart、ProgramBody 以及终极符.，这样的树形结构便于构造实验要求的输出样式。

为了将树形结构封装，定义 **SyntaxTree** 类，该类通过记录语法树的根节点来封装了语法树，并且可以按照实验要求将语法树进行图形化输出。

回到语法分析这里，进行语法分析时，有两种符号，分别是非终极符和终极符。在上述的树形结构中，我们没有很明显地区分二者。所以在本实验方案下，有一个抽象类 **Symbol** 是符号类，而终极符和非终极符继承自它，分别定义为 **Terminal** 和 **NonTerminal**。而一个符号即代表着一个树节点，一个树节点也代表着一个符号，同时一个终极符对应着一个单词，而一个非终极符对应着一个文法中的单词。

因此对于 Terminal，其内包含着树节点 node 以及词法单元 Token，而对于 NonTerminal，则包含树节点 node 和文法单词 value。

进行语法分析时，根据实验文档中给出的产生式，再将文法符号根据 predict 集逐步推导。为了根据当前符号以及该符号后面的一个单词来确定运用哪个产生式，我们定义枚举类型 **NON_TERMINAL**，给定特定的非终极符以及出现在其之后的第一个单词，该类型可以给出产生式右部的符号链表。

而最关键的则是如何进行 LL(1)分析。利用一个栈代表当前的符号栈，初始状态下符号栈中只有 SNL 总程序符号，即栈中只有非终极符 Program。

之后进行不断地出栈。每出栈一个符号，判断该符号是终极符还是非终极符。如果是终极符，则判断当前符号是否和输入的词法单元匹配，如果匹配则 match 成功，否则就是匹配失败；如果是非终极符，则根据 NON_TERMINAL 中的 predict 来获得产生式的右部符号的链表，将他们倒序压入符号栈中。如果查询 predict 集没有对应的产生式，则查询失败。

截止目前，已经将语法分析核心部分介绍完毕。但是语法分析必须依赖词法分析的结果，二者存在很强的关联，因此对于语法分析部分，我们划分出 **LexParse** 与 **LL1Parse** 两个部分来完成。其中 LexParse 是 LL1Parse 的父类。

# 四、实验步骤

为了让实验过程更加直观、简洁，本实验应该开发前端界面。

从前端可以输入待分析的代码片段或者选择文件打开，之后可以进行词法分析或者语法分析，并将结果输出。

综合设计思想中提到的内容，工程的结构如下所示，其中的包以粗体表示。

- **compiler：编译器包**
  - **GUI：前端界面包**
    - LineNumberHeaderView
    - ParseForGUI
    - SNLCompilerGUI
  - **lexer：词法分析包**
    - Lexer
    - LexerResult
    - Token
    - TokenType
  - **syntax：语法分析包**
    - **LL1：LL(1)分析包**
      - LexParse
      - LL1Parse
    - **symbol：文法符号包**

        - NON_TERMINAL
        - NonTerminal
        - Symbol
        - Terminal
      - **tree：树形结构包**
        - SyntaxTree
        - TreeNode
    - **utils：其他功能包**
      - FileUtils
      - PropertiesUtils
      - ToStringUtils
  - SNLC
  - SNLCforGUI

# 五、数据结构

```
public enum TokenType {
   ERROR("error"),
   ID("id"),
   PROGRAM("program"), PROCEDURE("procedure"), TYPE("type"), VAR("var"),
   IF("if"), THEN("then"), ELSE("else"), FI("fi"),
   WHILE("while"), DO("do"), ENDWH("endwh"),
   BEGIN("begin"), END("end"),
   READ("read"), WRITE("write"),
   OF("of"), RETURN("return"),
   …
}

public class Token {
   private int line;      //行
   private int column;    //列
   private TokenType type; //类型
   private String value;  //含义
   public void checkKeyWords();//根据含义精确修改类型
   …
}

public class Lexer {
   private enum StateEnum {
      NORMAL,          // 初始状态
```

```
        INID,           // 标识符状态
        INNUM,          // 数字状态
        INASSIGN,       // 赋值状态
        INCOMMENT,      // 注释状态
        INDOT,          // 点状态
        INRANGE,        // 数组下标界限状态
        INCHAR,         // 字符标志状态
        ERROR,          // 出错
    }
    private static Logger LOG = LoggerFactory.getLogger(Lexer.class);
    private int line = 1;
    private int column = 0;
    private int getMeFirst = -1;
    private boolean lf = false;
    private Reader reader;
    private List<String> errors;
    public LexerResult getResult(Reader reader) throws IOException;//获取词法分析结果
    private Token getToken() throws IOException;//自动机
    …
}

public class LexerResult {
    private List<Token> tokenList;
    …
}

public class TreeNode extends ToStringUtils {
    private TreeNode siblings;
    private TreeNode children;
    private String value;
    。。。
}

public class SyntaxTree {
    private TreeNode root;
    private PrintStream out;
    public void preOrderRecursiveCore(TreeNode node, int level, int cnt, int[] b) throws
FileNotFoundException;//递归打印树形结构
    …
}
```

```
public abstract class Symbol extends ToStringUtils {
    public abstract TreeNode getNode();
}

public class Terminal extends Symbol {
    private final TreeNode node;
    private Token token;
    …
}

public class NonTerminal extends Symbol {
    private final TreeNode node;
    private final String value;
    …
}

public enum NON_TERMINAL {
    Program(new NonTerminal("Program")) {
        @Override
        public List<Symbol> predict(Token token) {
            if (token.getType() == PROGRAM) {
                return asList(nonFactory("ProgramHead"), nonFactory("DeclarePart"),
                        nonFactory("ProgramBody"), terFactory(EOF));
            }
            return null;
        }
    },
    ProgramHead(new NonTerminal("ProgramHead")) {
        @Override
        public List<Symbol> predict(Token token) {
            if (token.getType() == PROGRAM) {
                return asList(terFactory(PROGRAM), nonFactory("ProgramName"));
            }
            return null;
        }
    },
    …
}

public abstract class LexParse {
    private static Logger LOG = LoggerFactory.getLogger(LexParse.class);
```

```
    private List<Token> tokenList;
    private int currentIndex = 0;
    public void lexParse(String inPath);//进行词法分析
    private TreeNode match(TokenType type);//进行终结符匹配
    private List<Symbol> find(NonTerminal symbol);//查找 predict 集
    …
}

public class LL1Parse extends LexParse {
    private static Logger LOG = LoggerFactory.getLogger(LL1Parse.class);
    private SyntaxTree syntaxTree;
    public SyntaxTree syntaxParse();//进行语法分析
    …
}
```

# 六、算法设计

6.1 词法分析

```
    public LexerResult getResult(Reader reader) throws IOException {
        LexerResult result = new LexerResult();
        List<Token> tokenList = new ArrayList<>();

        if (reader == null) {
            LOG.error("字符输入流为空");
            result.setTokenList(tokenList);
            return result;
        }

        this.reader = reader;
        Token token = getToken();
        while (token != null) {
            tokenList.add(token);
            token = getToken();
        }
        result.setTokenList(tokenList);
        return result;
    }
```

**6.2 语法分析**

```
    public SyntaxTree syntaxParse() {
```

```
    LOG.info("=========================语法分析开始
============================");
    Stack<Symbol> stack = new Stack<>();
    // 开始符 start： Program 并且是 root
    NonTerminal start = NonTerminal.nonFactory("Program");
    TreeNode root = start.getNode();
    // 开始符压栈
    stack.push(start);

    Symbol symbol;
    while (!stack.isEmpty()) {
      symbol = stack.pop();
      if (symbol instanceof Terminal) {
        LOG.trace("symbol 是终结符：<" + ((Terminal) symbol).getToken().getType() +
">");
        // 判断是否 symbol 和当前 token 是否匹配
        // 新建 TreeNode，通过 match 返回
        TreeNode match = match(((Terminal) symbol).getToken().getType());
        if (match != null) {
          symbol.getNode().setValue(match.getValue());
          LOG.trace("##########终结符识别完毕：<" + symbol.getNode().getValue() + ">
##########");
        } else {
          LOG.error("匹配终结符失败");
          return null;
        }

      } else if (symbol instanceof NonTerminal) {
        LOG.trace("symbol 是非终结符：<" + ((NonTerminal) symbol).getValue() + ">");
        // 查预测分析表，将 symbol 替换成右端生成式
        List<Symbol> symbols = find((NonTerminal) symbol);
        LOG.info("预测分析表返回：" + symbols);

        if (symbols != null) {
          // 右端生成式为 symbol 的 children
          int size = symbols.size();
          for (int i = 0; i < size - 1; i++) {
            symbols.get(i).getNode().setSiblings(symbols.get(i + 1).getNode());
          }
          symbol.getNode().setChildren(symbols.get(0).getNode());
```

```
                LOG.trace("构建" + ((NonTerminal) symbol).getValue() + "的子树：" +
symbol.getNode().getChildren());
                // 右端生成式逆序入栈
                for (int i = size - 1; i >= 0; i--) {
                    Symbol item = symbols.get(i);
                    if (item instanceof Terminal || !((NonTerminal) item).isBlank())
                        stack.push(item);
                }
            } else {
                LOG.error("预测分析表查询失败");
                return null;
            }
        } else {
            // error
            LOG.error("未识别的字符，出现了不应该出现的字符：[ " +
symbol.getNode().getValue() + " ]");
            return null;
        }
    }
    syntaxTree = new SyntaxTree(root);
    return syntaxTree;
  }
```

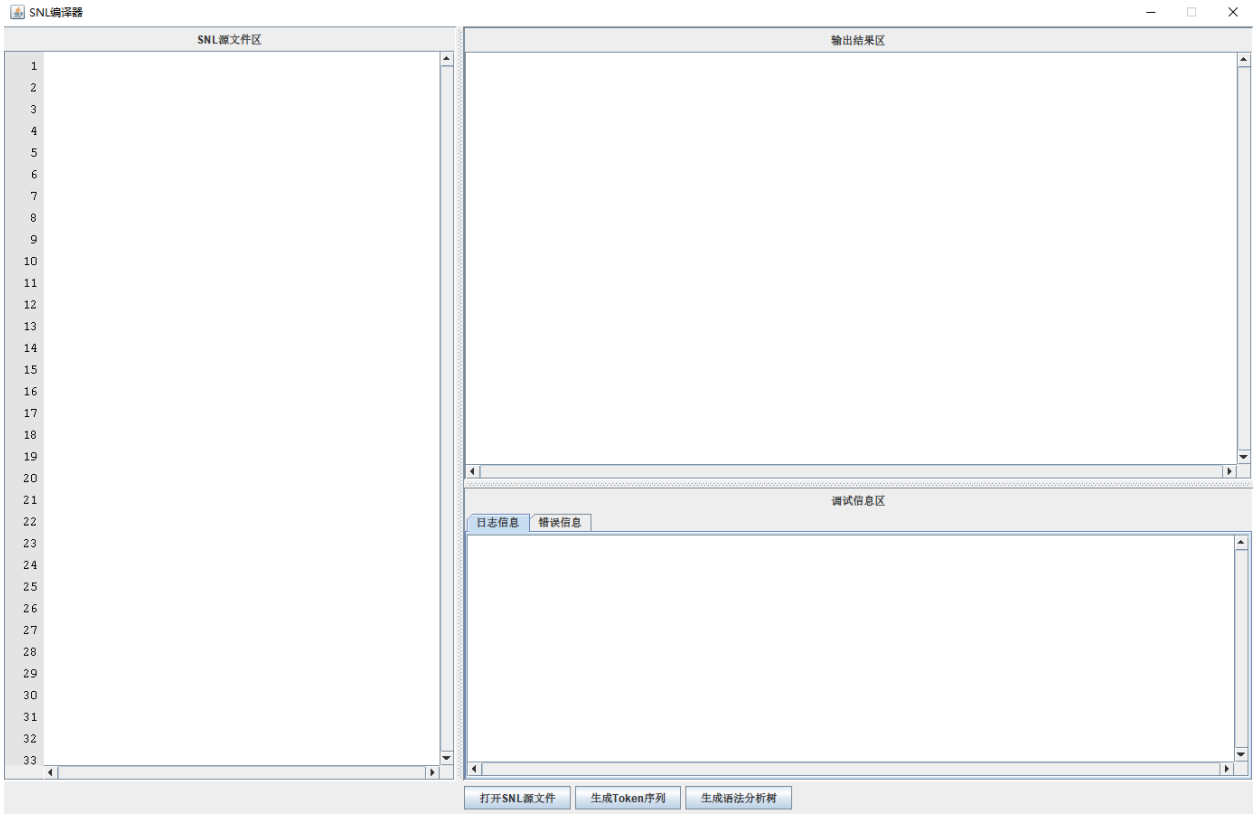
**6.3 递归打印树形结构**

```
  public void preOrderRecursiveCore(TreeNode node, int level, int cnt, int[] b) throws
FileNotFoundException {
    if (node == null) return;
    // int temp = cnt;
    for (int i = 0; i < level - 1; i++) {
        if (b[i] == 1) {
            System.out.print("|  ");
            out.print("|  ");
        } else {
            System.out.print("   ");
            out.print("   ");
        }
    }

    if (node == root) {
        System.out.println(node.getValue());
        out.println(node.getValue());
```

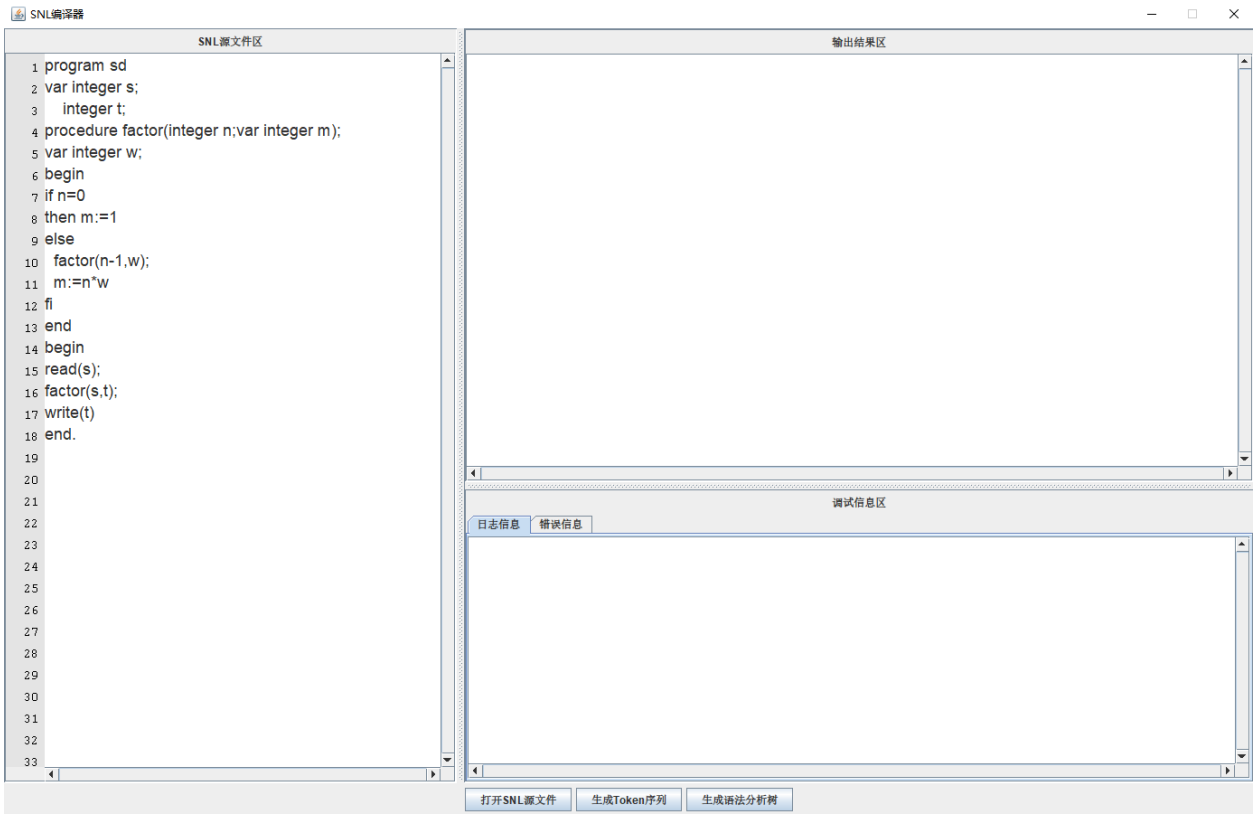
```
    } else {
        System.out.println("|__" + node.getValue());
        out.println("|__" + node.getValue());
    }
    if (node.hasChild()) {
        if (node.hasSiblings()) {
            if (level > 0)
                b[level - 1] = 1;
            preOrderRecursiveCore(node.getChildren(), level + 1, cnt + 1, b);
        } else {
            if (level > 0)
                b[level - 1] = 0;
            preOrderRecursiveCore(node.getChildren(), level + 1, cnt, b);
        }
    }

    if (node.hasSiblings()) {

        preOrderRecursiveCore(node.getSiblings(), level, cnt, b);
    }
}
```

# 七、实验结果

**7.1 运行程序（SNLCforGUI）**




## 7.2 输入一段正确的 SNL 代码片段

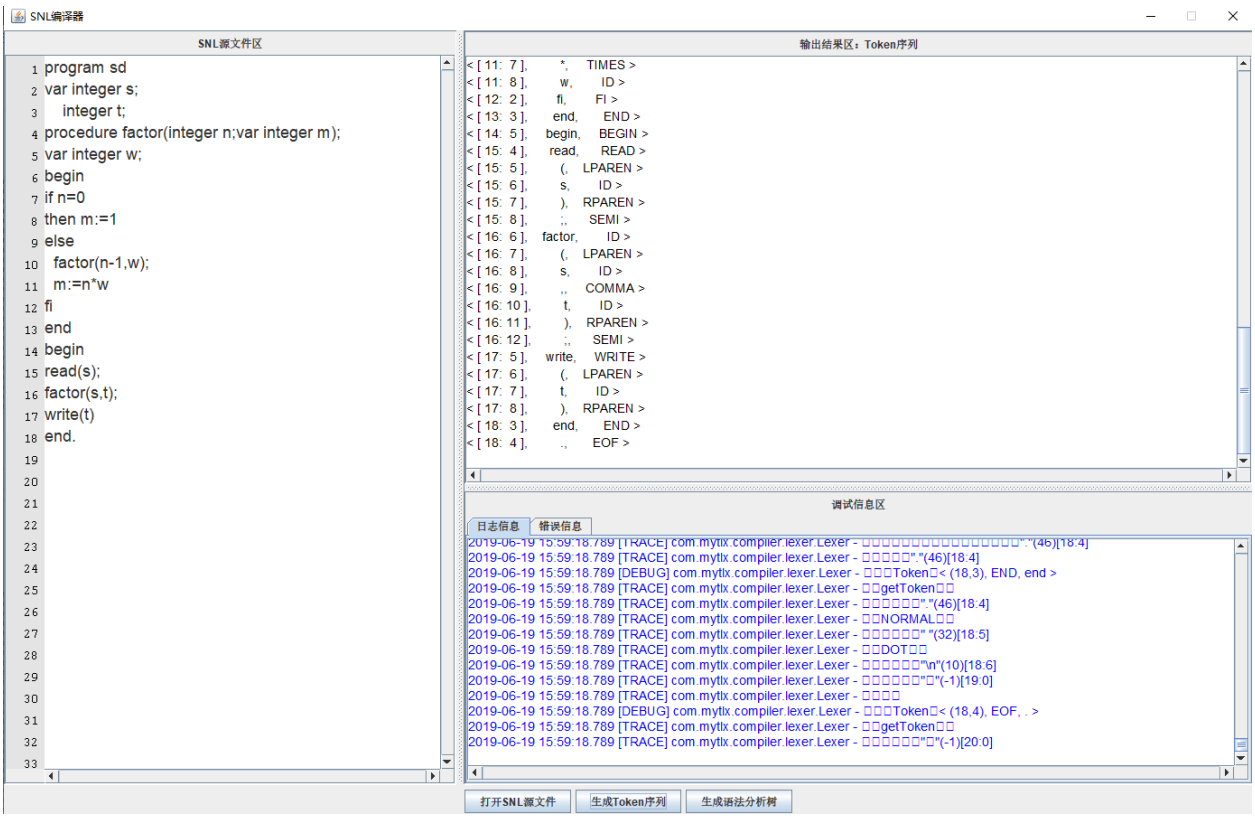



代码如下所示：

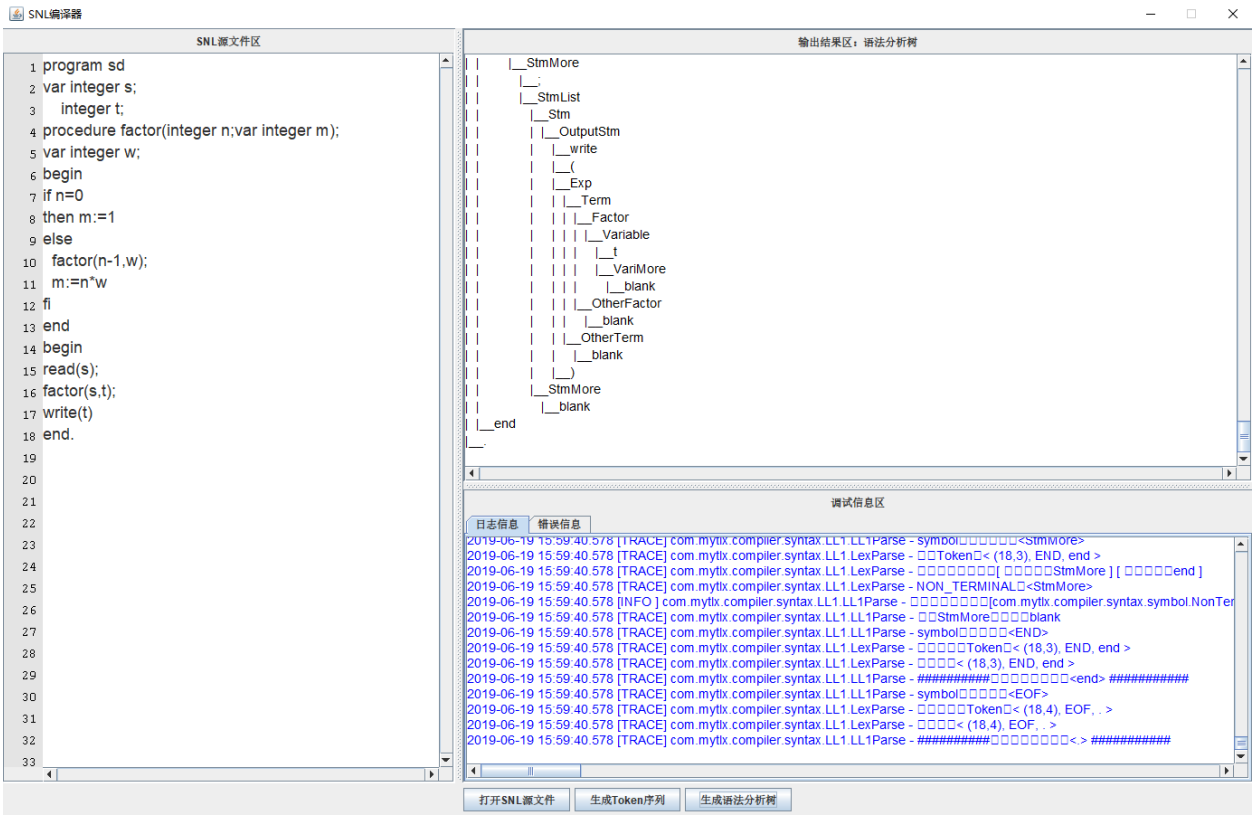
```
program sd
```

```
var integer s;
    integer t;
procedure factor(integer n;var integer m);
var integer w;
begin
if n=0
then m:=1
else
  factor(n-1,w);
  m:=n*w
fi
end
begin
read(s);
factor(s,t);
write(t)
end.
```

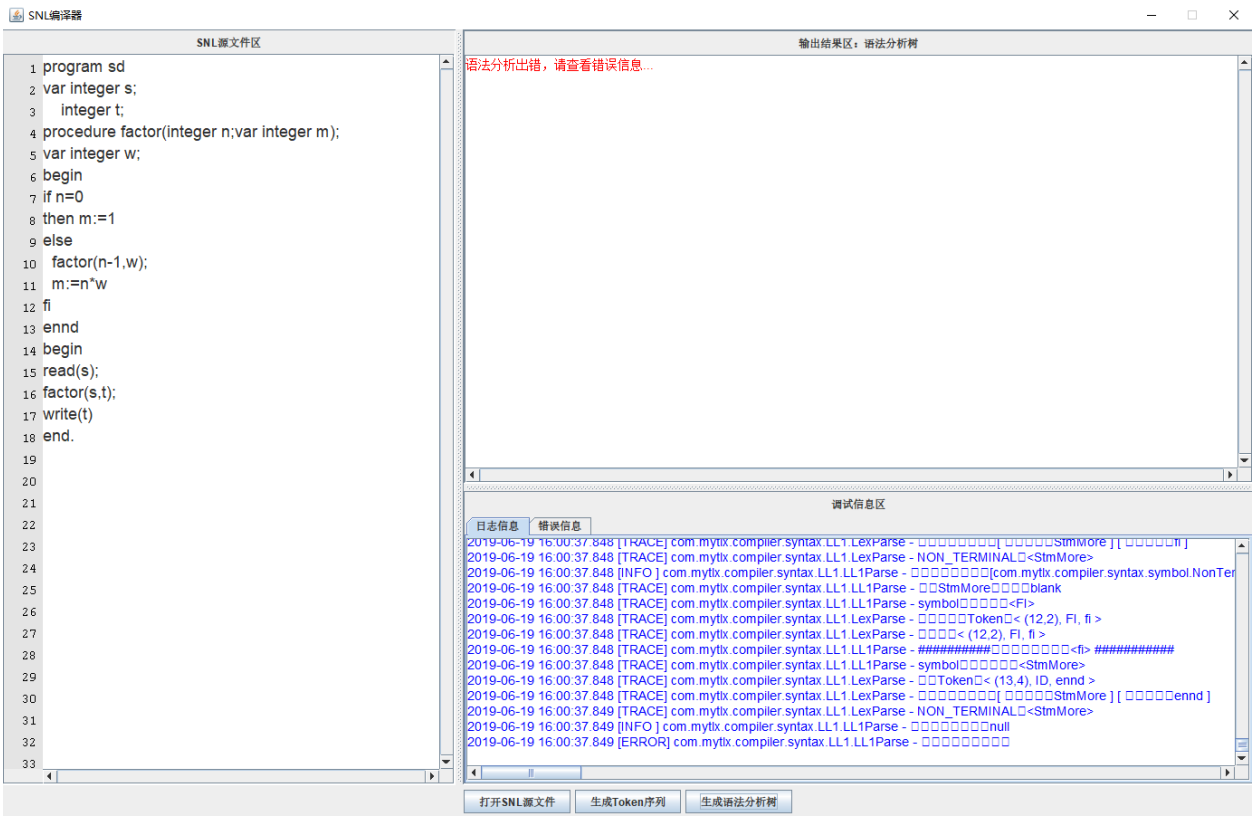
### 7.3 点击【生成 Token 序列】按钮

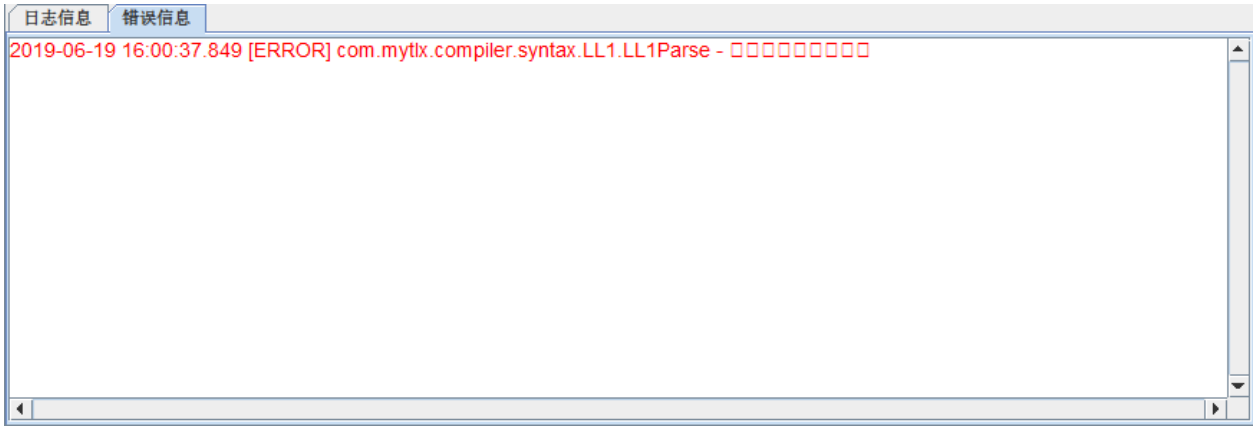



## 7.4 点击【生成语法分析树】按钮




## 7.5 将第 13 行的 end 更改为 ennd，测试错误样例






**附：上述中的中文字符都出现了乱码，而实际上本程序还额外有一个日志文件，在日志文件中可以观察到正确的字符。**

log.log - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
2019-06-19 16:00:37.810 [INFO ] com.mytlx.compiler.GUI.SNLCompilerGUI - =======开始语法分析========
2019-06-19 16:00:37.810 [INFO ] com.mytlx.compiler.GUI.SNLCompilerGUI - 源文件：./src/main/resources/tmp/snlText.t
2019-06-19 16:00:37.810 [INFO ] com.mytlx.compiler.GUI.SNLCompilerGUI - 输出文件：./src/main/resources/tmp/syntax]
2019-06-19 16:00:37.810 [INFO ] com.mytlx.compiler.GUI.SNLCompilerGUI - ==========================
2019-06-19 16:00:37.811 [TRACE] com.mytlx.compiler.lexer.Lexer - 调用getToken方法
2019-06-19 16:00:37.811 [TRACE] com.mytlx.compiler.lexer.Lexer - 当前字符是："p"(112)[1:1]
2019-06-19 16:00:37.811 [TRACE] com.mytlx.compiler.lexer.Lexer - 进入NORMAL状态
2019-06-19 16:00:37.811 [TRACE] com.mytlx.compiler.lexer.Lexer - 当前字符是："r"(114)[1:2]
2019-06-19 16:00:37.812 [TRACE] com.mytlx.compiler.lexer.Lexer - 进入ID状态
2019-06-19 16:00:37.812 [TRACE] com.mytlx.compiler.lexer.Lexer - 当前字符是："o"(111)[1:3]
2019-06-19 16:00:37.812 [TRACE] com.mytlx.compiler.lexer.Lexer - 进入ID状态
2019-06-19 16:00:37.812 [TRACE] com.mytlx.compiler.lexer.Lexer - 当前字符是："g"(103)[1:4]
2019-06-19 16:00:37.812 [TRACE] com.mytlx.compiler.lexer.Lexer - 进入ID状态
2019-06-19 16:00:37.812 [TRACE] com.mytlx.compiler.lexer.Lexer - 当前字符是："r"(114)[1:5]
2019-06-19 16:00:37.812 [TRACE] com.mytlx.compiler.lexer.Lexer - 进入ID状态
```

error.log - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
2019-06-19 16:00:37.849 [ERROR] com.mytlx.compiler.syntax.LL1.LL1Parse - 预测分析表查询失败
```

# 八、实验结论

在实际编码的过程中，才能真正体会到编译原理的本质。本次实验采用的是 Java 语言，可能对某些过程没有采用较高效的方法，可能会损失一部分效率。倘若采用 C++语言，效率一定会提升不少。但我对 C++前端编码仍然不够熟练，可能会浪费很多时间在前端调试上。

在最后组装界面时，中文乱码问题一直无法解决，调试了两三个小时也无从知晓。但同步写入日志与控制台输出都具有良好的格式。倘若以后能有机会，可能会对这部分代码进行改进。

该程序具有良好的交互性，而且能看到详细的分析过程，十分人性化。

我很感谢周围同学在此实验上给予我的帮助，他们帮助我更好地进行系统设计，使我受益匪浅。这次实验课让我深入了解到编译的奥秘，希望以后能再了解学习其他语言的编译方式。