概述

本课程实验为编写一个将 SysY 语言翻译为 MIPS 汇编语言的编译器，编写所用到的高级语言为 JAVA

SysY 语言是 C 语言的一个子集，支持的功能包括但不限于：

• 常量变量定义与使用

• 一维二维数组的定义与使用

• 条件与循环

• 函数调用

• 整数输入

• 字符串和整数的输出

• 短路求值

编译过程可以分为五个阶段：

• 词法分析

• 语法分析

• 语义分析和中间代码生成

• 代码优化

• 目标代码生成

此外，还有符号表管理和错误处理两部分穿插在五个阶段之中，接下来的架构设计章节将逐个介绍每一部

分。代码优化部分将中间代码转化为了 SSA (Static Single-Assignment，静态单一赋值) 形式，并进行了

相关优化，将额外开设一个章节进行介绍。

架构设计

词法分析

词法分析过程本质是根据文法所用到的终结符构建一个 DFA，识别出所有的关键字及标识符。由于实现的文法关键词较少，文法相对简单，关键词长度不超过 10，实现过程中并不需要真正构建出一个 DFA，只需模拟 DFA 进行贪心匹配即可。

具体来说，设计的关键点如下：

- 由于windows与linux环境下换行符在'\r'上存在区分，因此采用文件IO中的readline()来消除影响
- 对于源代码中存在的注释与FormatString，需要设置两个变量来标记当前状态：
    - noteSt标记注释状态，0代表无注释，1代表单行，2代表多行
    - 单行注释，从 // 开始，到换行符结束，故每读入新的一行需要去掉单行注释状态；多行注释，从 /* 开始，到 */ 结束，因此随时都有可能更新，需要预读下一个字符
    - fsSt表示是否正在读格式字符串，FormatString从 " 开始，到 " 结束；同时每读入新的一行需要清空格式字符串内容

- 空白符自动跳过即可
- 数字开头必然为常量，故直接贪心匹配即可。注意非0数字不会以0开头，因此直接可以特判
- 对于保留字，其构造词法和普通变量完全一致，只是名称特殊，故放在变量扫描中处理，在读完整个单词后直接特判即可
- 字符'/'与'*'与""需要特判，因其涉及到注释与格式字符串
- 对于剩下的字符，可以将其分为三类：
  - 双字符遵循最长匹配原则，因此需要预读下一个字符判断是什么符号

    ```
    public boolean isDouble(char c) {
            return c == '!' || c == '<' || c == '>' || c == '=';
    }
    ```

  - 另一种读入就可以直接判断：

    ```
    public boolean isSingle(char c) {
            return c == '&' || c == '|' || c == '+' || c == '-' || c == '%'
    || c == '*';
    }
    ```

  - 最后一种统一视为分隔符：

    ```
    public boolean isSep(char c) {
            return c == '[' || c == ']' || c == '{' || c == '}' || c == '('
    || c == ')' || c == ',' || c == ';';
    }
    ```

- 所有预读下一个字符都需要判断是否越界
- 保留单词所在行号以便后续功能实现
- 所有词法类统一继承Token接口，以方便输出评测

心得

- 词法做出来成分本质上是为了**语法分析程序可以去利用分析结果来分析语法成分**
- 此外，要**防止词法分析与语法分析过于耦合**，所以有**单词类和语法成分类两个分类划分**

注意：不同模块之间不要过于耦合！！

语法分析程序

产生 SysY 语言的文法是 CFG (Context Free Grammar，上下文无关文法)，后续语义分析中通过符号表以实现上下文的联系。

而语法分析本质上是对词法分析产生的"单词流"进行顺序分析，根据文法规则得到一系列语法成分。采用与面向对象第一单元类似的方法进行递归下降分析。

要点如下：

- 采用递归下降处理，每一部分的分析函数只需要根据文法实现即可，例如顶层文法CompUnit → {Decl} {FuncDef} MainFuncDef ：

  ```
  public void parseCompUnit() {
  ```

```
        while (tokens.get(pos).getName().equals("const") ||
tokens.get(pos).getName().equals("int")) {
            if (tokens.get(pos).getName().equals("const")) {
                compUnit.addConstDecl(parseConstDecl());
            } else {
                if (tokens.get(pos + 2).getName().equals("(")) {
                    break;
                }
                compUnit.addVarDecl(parseVarDecl());
            }
        }
        while (tokens.get(pos).getName().equals("void") ||
tokens.get(pos).getName().equals("int")) {
            if (tokens.get(pos).getName().equals("void")) {
                compUnit.addFuncDef(parseFuncDef());
            } else {
                if (tokens.get(pos + 1).getName().equals("main")) {
                    break;
                }
                compUnit.addFuncDef(parseFuncDef());
            }
        }
        compUnit.addMainFuncDef(parseMainFuncDef());
        prints.add("<CompUnit>");
    }
```

- 对每个语法成分都建类处理，这样可以降低耦合度并更加灵活地增加功能；而对于那些需要统一实现的功能，我们只需要增加接口使其继承即可
- 注意单词流指针位置的控制，处理每个类时，该类的处理需要读掉属于该类的全部单词，同时不能读掉该类之外的单词，防止混乱
- 需要区分可选项[]（使用if判断）以及重复0次至多次的成分{}（使用while等循环结构判断）
- 左递归文法消除：采用文法改写，例如关系表达式RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp：
    - 首先改写文法，S→a|Sb 可改写为 S→a{b}，故上述文法可改写为RelExp → AddExp ('<' | '>' | '<=' | '>=') AddExp ，直接循环处理即可
    - 改写文法后需要注意对于文法类的分析：

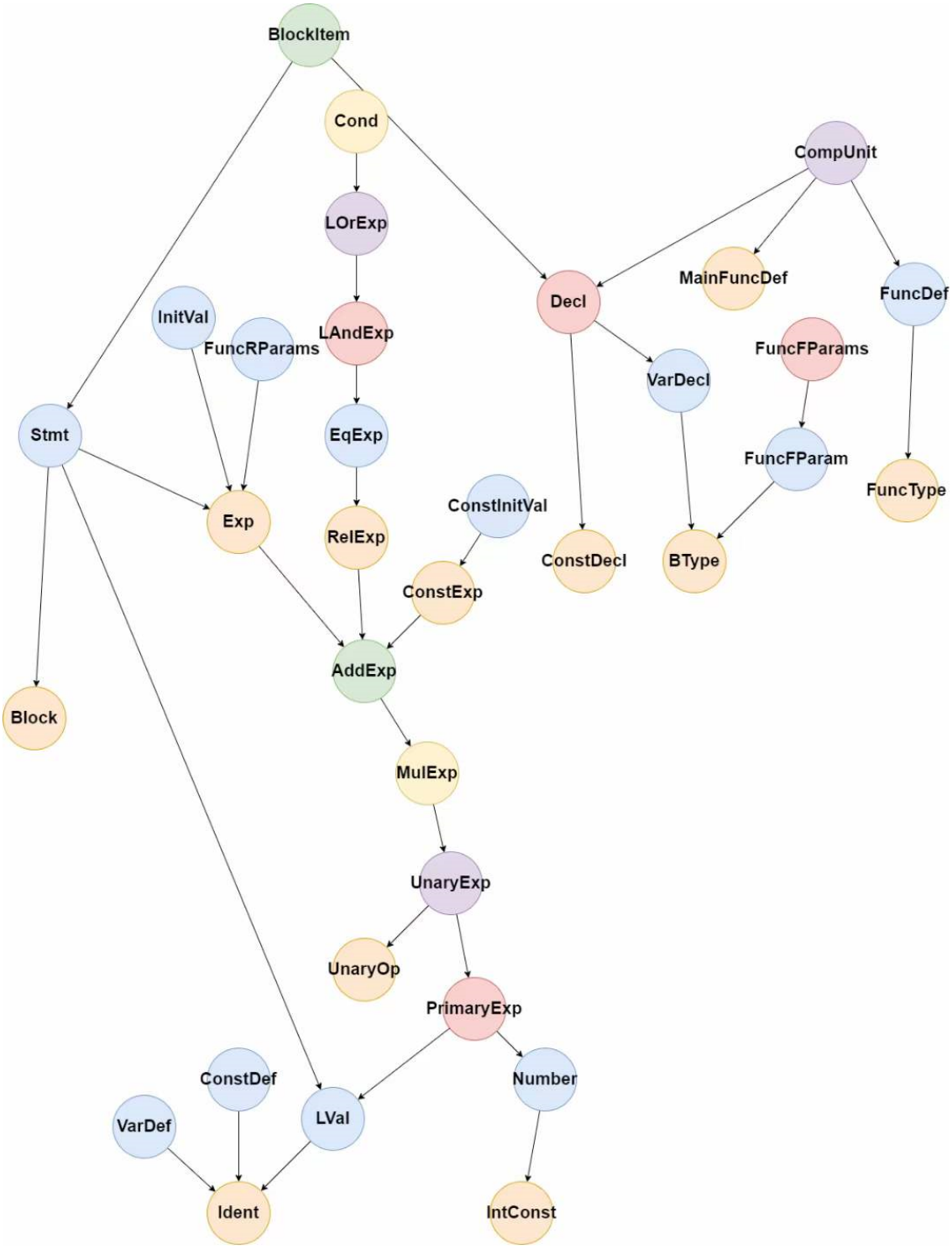
      ```
      public RelExp parseRelExp() {
              RelExp relExp = new RelExp();
              relExp.addAddExp(parseAddExp());
              prints.add("<RelExp>");
              while (syntacticalType.isRelOp(tokens.get(pos))) {
                  relExp.addOp(((Op) tokens.get(pos)));
                  addToken(1);
                  relExp.addAddExp(parseAddExp());
                  prints.add("<RelExp>");
              }
              return relExp;
      }
      ```

      其中第一处对应原文法中RelExp → AddExp；第二处对应原文法中的RelExp → RelExp ('<' | '>' | '<=' | '>=') AddExp

- 文法中存在回溯问题，因为某些非终结符规则右部存在多个选择，且first集相交。采用超前扫描处理，直到可以确定
- 语法分析中最复杂的部分是Stmt语句识别，因其可能情况较多

```
语句 Stmt → LVal '=' Exp ';'
| [Exp] ';'
| Block
| 'if' '( Cond ')' Stmt [ 'else' Stmt ]
| 'while' '(' Cond ')' Stmt
| 'break' ';' | 'continue' ';'
| 'return' [Exp] ';'
| LVal = 'getint''('')'';'
| 'printf' '('FormatString {',' Exp} ')'';'
```

  - 对于打印语句、返回语句、跳转语句、循环语句、分支语句，可以很方便地通过第一个标识符（保留字）进行识别。如果第一个符号是左括号，则判断为语句块
  - 需要区分的是LVal和Exp。注意到Exp成分中没有赋值号"="，故可以从当前位置向后找，判断先遇到分号还是先遇到赋值符号('=')。如果先遇到赋值符号，则说明是赋值语句，否则是表达式语句
- 单词流的移动统一使用addToken方法，模拟真正的词法分析过程，语法成分在完成该部分词法分析后加入


BlockItem
Cond
CompUnit
LOrExp
MainFuncDef
FuncDef
Decl
InitVal
LAndExp
FuncRParams
FuncFParams
VarDecl
Stmt
EqExp
FuncFParam
FuncType
Exp
ConstInitVal
RelExp
ConstDecl
BType
ConstExp
AddExp
Block
MulExp
UnaryExp
UnaryOp
PrimaryExp
ConstDef
Number
VarDef
LVal
Ident
IntConst

特别提醒