一. 参考编译器介绍

主要是在代码生成作业部分参考了各种中间代码的设计。

> 从中间代码所体现出的细节上，我们可以将中间代码分为如下三类： 1) 高层次中间代码（High-level IR或HIR）：这种中间代码体现了较高层次的细节信息，因此往往和高级语言类似，保留了不少包括数组、循环在内的源语言的特征。高层次中间代码常在编译器的前端部分使用，并在之后被转换为更低层次的中间代码。高层次中间代码常被用于进行相关性分析（ Dependence Analysis ） 和解释执行 。 我们所熟悉的 Java bytecode、Python .pyc bytecode以及目前使用得非常广泛的LLVM IR都属于高层次IR。 2) 中层次中间代码（Medium-level IR或MIR）：这个层次的中间代码在形式上介于源语言和目标语言之间，它既体现了许多高级语言的一般特性，又可以被方便地转换为低级语言的代码。正是由于MIR的这个特性，它是三种IR中最难设计的一种。在这个层次上，变量和临时变量可能已经有了区分，控制流也可能已经被简化为无条件跳转、有条件跳转、函数调用和函数返回四种操作。另外，对中层次中间代码可以进行绝大部分的优化处理，例如公共子表达式消除（Common-subexpression Elimination）、代码移动（Code Motion）、代数运算简化（Algebraic Simplification）等。 3) 低层次中间代码（Low-level IR或LIR）：低层次中间代码与目标语言非常接近，它在变量的基础上可能会加入寄存器的细节信息。事实上，LIR中的大部分代码和目标语言中的指令往往存在着一一对应的关系，即使没有对应，二者之间的转换也属于一趟指令选择就能完成的任务。

> 从表示方式来看，我们又可以将中间代码分成如下三类： 1) 图形中间代码（Graphical IR）：这种类型的中间代码将输入程序的信息嵌入到一张图中，以结点和边等元素来组织代码信息。由于要表示和处理一般的图代价会很大，人们经常会使用特殊的图，例如树或有向无环图（DAG）。一个典型的树形中间代码的例子就是抽象语法树（Abstract Syntax Tree或AST）。抽象语法树中省去了语法树里不必要的结点，将输入程序的语法信息以一种更加简洁的形式呈现出来。其它树形中间代码的例子有GCC中所使 用的GIMPLE。这类中间代码将各种操作都组织在一棵树中，在后面的实验四的指令选择部分我们会看到这种表示方式可以简化其中的某些处理。 2) 线形中间代码（Linear IR）：线形结构的代码我们见得非常多，例如我们经常使用的C语言、Java语言和汇编语言中语句和语句之间就是线性关系。你可以将这种中间代码看成是某种抽象计算机的一个简单的指令集。这种结构最大的优点是表示简单、处理高效，而缺点就是代码和代码之间的先后关系有时会模糊整段程序的逻辑，让某些优化操作变得很复杂。 3) 混合型中间代码（Hybrid IR）：顾名思义，混合型中间代码主要混合了图形和线形两种中间代码，期望结合这两种代码的优点并避免二者的缺点。例如，我们可以将中间代码组织成许多基本块，块内部采用线形表示，块与块之间采用图表示，这样既可以简化块内部的数据 流分析，又可以简化块与块之间的控制流分析。

在了解了不同形式的中间代码之后，采用遍历具体语法树中的每一个结点，当发现语法树中有特定的结构出现时，就产生出相应的中间代码的方式生成`PCODE`，这种方式简单直接，同时也是在实际应用理论课上学到的语法制导翻译：为每个语法单元在合适的位置插入翻译`PCODE`的`translate-function`。

二、编译器总体设计

总体结构 & 文件组织

```
|  Compiler.java          # 编译器入口
|  ErFunc.java            # error-func错误处理函数
```

```
|  Lexer.java            # 词法分析器
|  Parser.java           # 语法分析+错误处理+代码生成主函数
|
├─CodeGen                # 代码生成package
|       MidType.java     # 中间代码类型
|       PCode.java       # PCODE
|       PCodeExecutor.java # 中间代码执行器
|       RetInfo.java     # 函数返回信息
|       Var.java         # 变量类抽象
|
├─Components
|       Function.java
|       Item.java
|       Syntax.java      # 语法分析中间键
|       SyntaxType.java
|       Token.java       # 词法分析中间键
|       TokenMap.java
|       TokenType.java
|
├─Errors
|       ErrorLog.java    # 错误处理日志
|       MyException.java # 异常类抽象
|
├─META-INF
|       MANIFEST.MF
|
└─symbolTableEl          # symbolTableElement
         Symbol.java     # 符号
         SymbolTable.java# 符号表
```

接口设计

输入输出接口：在`Compiler.java`类中实现`readToBuffer`和`writeToLog`方法，将`testfile.txt`中内容按行读入缓冲区，供词法分析使用；将最终`PCodeExecutor`执行的结果写入`pcoderesult.txt`;在`PCodeExecutor`用`scanner`进行`getint()`函数的读入。

`ArrayList<String> stringBuf`将代码交由`Lexer`调用`lexing()`方法解析，得到`ArrayList<Token> tokens`交由`Parser`调用`parsing()`方法解析，得到`ArrayList<PCode> codes`交由`PCodeExecutor`调用`run()`方法解析，得到`printList`交由`writeToLog`打印。

一、词法分析

编码前设计：

主类：

```
enum TokenType
```

按照词法分析输出要求中的词法分析类别码定义构建枚举类`TokenType`。

```
Token implements Item
Attr:-private TokenType type;
     -private String val;
     -private int LineNo;
-String toString() {
        return type + " " + val;
    }
```

记录每个`Token`的类型、值、所在行号，提供`toString`方法在词法分析阶段输出。

```
Lexer
Attr:-private ArrayList<String> strings; // 将输入程序按行分解
     -private int lineNo; // 行号
     -private int idx; // 在本行当前解析到的位置
```

主方法：

`ArrayList<Token> lexLine(String line)`：每次将一行程序解析为若干个token并返回主程序。

对于当前读入的字符char c进行分析：

类型	类型识别	处理
多行注释(comment)	以/开头且后面的第一个字符是*	如果不是*，说明本行仍为注释行，跳过；如果是 *且后一个字符是/，说明结束注释状态
标识符/关键字	以字母或_开头	一直读到不符合identifier定义停止，将该字符传给lexToken函数识别是标识符还是关键字
数字(int const)	以数字开头	一直向后读字符直到不符合int const定义停止
! / !=/</<=/>/>=/=/==	以!/</>/=开头	查看!/</>/=后的字符，如果不是=回退idx
&& / ||	以&/|开头	如果没有出错只能识别到&&/||
+/-/*/%/;/,/(/)/[/]/{/}	以+/-/*/%/;/,/(/)/[/]/{/}开头	如果没有出错只能识别到+/-/*/%/;/,/(/)/[/]/{/}
单行注释	以/开头且后面的第一个字符是/	跳过本行解析，解析下一行

类型	类型识别	处理
除法(/)	以/开头且后面字符不符合注释的两种情况	回退idx
格式化输出(FormatString)	以 " 开头	一直向后读字符直到读到 "，整个"..."为STRCON

`public ArrayList<Token> lexing()`：对每一行使用`lexLine`方法，解析完毕将所有`tokens`返回。

编码后修改：

单独建立关键字的<单词名称，类别码>的`hashMap`，提供方法根据名称获取类别码(保留字)。

```
TokenMap
-HashMap<String, TokenType> enumMap
-public static TokenType getTokenType(String val)
```

二、语法分析

编码前设计：

为每个非终结符编写一个递归子程序。使用词法分析得到的`tokens`序列，从`CompUnit`开始，自顶向下递归下降解析每个文法左部的非终结符，最终得到一棵具体语法树。

所有节点均实现自`Node`接口，自身内部包括根据文法解析而来的子元素：

NodeCompUnit 编译单元	
ArrayList<Node> declItems	声明
ArrayList<Node> funcDefItems	函数定义
Node mainFunc	主函数

NodeConstDef 常量定义	
String name	常量名
int dimensions	维数(0/1/2)
ArrayList<Node> indexList	下标列表，用于存储各维度大小（可能是别的constSymbol定义的）
Node initVal	右部初始值，可能是Number或者Number列表等

普通变量：`x = 1` 一维数组：`x[4/2] = {1,2}`

`NodeVarDef` 变量定义 完全同 `NodeConstDef`。

NodeConstInitVal 常量初值	
Node singleVal	常表达式初值
ArrayList<Node> valList	一维数组/二维数组初值

1.常表达式初值 `1, 2`　　2.一维数组初值 `{1,2}`　　3.二维数组初值 `{{1, 2}, {3, 4}}`

NodeInitVal 变量初值	
Node singleVal	表达式初值
ArrayList<Node> valList	一维数组/二维数组初值

NodeFuncDef 函数定义	
String funcType	函数类型（返回int/void?)
String name	函数名
ArrayList<NodeFuncFParam> funcFParams	formal paras形参列表
NodeBlock block	函数块

NodeMainFuncDef extends NodeFuncDef 主函数定义

NodeFuncFParam 函数形参	
String name	形参名，需要加入符号表
int dimensions	维数(0/1/2)
Node length2	如果形参是二维数组，用来记录第二维的长度

NodeBlock 语句块	
ArrayList<Node> blockItems	声明或普通语句列表

BranchNode 分支节点(IF...Else...)	
Node condition	if分支判断条件
Node thenTodo	进入if执行的语句Stmt
Node elseTodo	进入else执行的语句Stmt

主方法`parse`:

```
ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';'
parseConstDecl()
-ArrayList<Node> constDefs = new ArrayList<>()
分别解析每个常量定义，最后生成一个常量声明节点。
```

```
VarDecl → BType VarDef { ',' VarDef } ';'
parseVarDecl()
-ArrayList<Node> varDefs = new ArrayList<>()
分别解析每个变量定义，最后生成一个变量声明节点。
```

```
ConstDef → Ident { '[' ConstExp ']' } '=' ConstInitVal
parseConstDef()
- 获取所有需要的信息，生成一个NodeConstDef节点并返回。
- 因为是定义语句，需要把定义好的常量加入符号表。
```

```
ConstInitVal → ConstExp | '{' [ ConstInitVal { ',' ConstInitVal } ] '}'
parseConstInitVal()
- 检查是常表达式还是数组，其中一个不为空就将另一个置为空。
```

```
VarDef → Ident { '[' ConstExp ']' } | Ident { '[' ConstExp ']' } '=' InitVal
parseVarDef()
- 和ConstDef基本相同，区别在于定义时可以先不赋初值。
- 获取所有需要的信息，生成一个NodeVarDef节点并返回。
- 因为是定义语句，需要把定义好的变量加入符号表。
```

```
InitVal → Exp | '{' [ InitVal { ',' InitVal } ] '}'
parseInitVal()
- 处理方法同常量初值
```

```
FuncDef → FuncType Ident '(' [FuncFParams] ')' Block
parseFuncDef()
- 判断函数返回类型；解析完函数名ident、将其填入符号表和函数表后，开辟新的作用域（符号表）；解析函数形式参数和block，解析完毕后将符号表弹栈。
- 获取函数定义的所有信息，生成一个NodeFuncDef节点并返回。
```

```
MainFuncDef → 'int' 'main' '(' ')' Block
parseMainFuncDef()
- 处理方法同FuncDef普通函数定义
```

```
FuncFParams → FuncFParam { ',' FuncFParam }
ArrayList<NodeFuncFParam> parseFParams()
- 调用子函数parseFParam()解析每个单独的形式参数，最终生成一个函数形参列表并返回。
```

```
FuncFParam → BType Ident ['[' ']' { '[' ConstExp ']' }]
NodeFuncFParam parseFParam()
- 获取一个函数形参（名字、维数、如果是二维数组则需要知道第二维的长度）的所有信息，生成一个NodeFuncFParam节点。
- 生成一个symbol并填入符号表。
```

```
Block → '{' { Decl | Stmt } '}'
NodeBlock parseBlock()
- 根据第一个字符类型的不同，确定是声明语句或普通语句，递归解析，最终生成一个NodeBlock节点并返回。
```

```
Node parseStmt()
- if语句：将if语句整体集合为一个BranchNode节点，注意在解析子语句Stmt之前需要开辟新的符号表，解析完毕后退出该符号表。
- while语句：将while语句整体集合为一个WhileNode节点。定义了全局变量cycleDepth，每当进入一个while-loop，cycleDepth自增1，可以处理循环嵌套的情况。
- return语句：返回ReturnNode节点，其中的返回值属性如果有就计算存储，如果没有就置null。
- printf语句：返回PrintNode节点，其中包括格式字符串和多个exp（如果有就存储）。
- Block语句块：进入子符号表，解析block，出子符号表。
- exp(除了以Ident开头的)
- ident开头(可能是exp或LVal，通过预读用"="与否判断)：如果是LVal，左部是Lval，右部可以生成GetIntNode或解析普通Exp。这种情况比较特殊，是赋值语句，所以将左右部集合起来生成新节点AssignNode。如果不是LVal，回退，重新当exp解析。
- ;：空语句，可以不做处理。
```

```
UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp
Node parseUnaryExp()
- 以Number/'('开头：parsePrimaryExp()
- 以'+'/'-'/'!'开头：!需要当作条件表达式处理，其余当作普通表达式处理。
- 以ident开头：如果是
```

编码后修改：

由于在最终代码生成阶段还是决定做`PCODE`，故并没有采用上述设计，而是把语法分析改回了最开始写的生成具体语法树的写法，没有对`token`进行处理，只是按照文法一步步解析。

一般需要预读的情况，我采用先记录当前位置的`index`，继续解析发现不符合文法时调用`rollBack`函数回退，但在后续错误处理时不希望同时回退`ErrorLog`，故可以直接预读2、3个字符，确定好走哪条路了再继续。(但如果是不能确定预读`token`个数的情况就无法解决了)

```
预读判断是LVal还是Exp: 通过文法中的'='判断
private boolean tryToGetLVal() {
        int index = getCurIndex();
        Token word = tokens.get(index);
        while (true) {
```

```
            if (word.getType().equals(TokenType.ASSIGN)) {
                return true;
            } else if (word.getType().equals(TokenType.SEMICN)) {
                return false;
            } else {
                word = tokens.get(++index);
            }
        }
    }
```

三、错误处理

编码前设计：

错误处理在语法分析过程中同步进行，符号表只存储变量/常量，函数存储在全局函数表中。在不同种类错误可能出现的位置进行查表等检查工作，发现错误存储到全局错误记录表`Errorlog`中。

`package: symbolTableEl`

符号类`Symbol`

Symbol	
String type	符号类型：const?var?func?
int scopeId	标识处于哪个scope
Token token	存储token节点本身
int dimensions	维数 0?1?2
String value	内容

符号表类`SymbolTable`

SymbolTable	
HashMap<String, Symbol> symbolHashMap	符号名字-符号对象本身
addSymbol(String type, int dimensions, Token token, int scopeID)	添加符号
hasSymbol(Token token / (String name))	根据名字查是否有同名符号
isConst(Token word)	检查token类型是否为常量

栈式符号表作用域控制由语法分析过程中对于scopeId/scope的控制完成。

函数类`Function`

Function	
String returnType	返回类型：int?void?
ArrayList<Integer> paras	各参数维数
int index	最终在生成代码执行栈中的位置
int parasSize	参数个数
String value	函数名

`MyException`

MyException	
int lineNo	错误行号
char exceptionType	错误类型

`ErrorLog`

ErrorLog	
static ArrayList <MyException> errors	记录错误
printErrorLog(FileWriter writer)	打印错误

具体错误处理：

非法符号	**a**
errorCheckA()	

依据文法要求，对STRCON检查：

1. `(c == '%')其后必须是 'd'`
2. `(c == '\\')其后必须是 'n'`
3. `c 不可以处在这些范围中: ((c < 32 && c != '\n') || ((c > 33) && (c < 40)) || (c > 126))`

名字重定义	b

<ConstDef>→<Ident> ...

<VarDef>→<Ident> ... |<Ident> ...

<FuncFParam> → <BType><Ident> ...

`hasSymbolInThisArea(Token token)`检查在当前作用域是否有重名`symbol`；

<FuncDef>→<FuncType><Ident> ...

检查全局函数表和全局变（常）量中是否存在重名，全局变（常）量即符号表symbols的第0层。

未定义的名字	c

`LVal → Ident {'[' Exp ']'} :`

`hasSymbol(Token word)`检查各层符号表该名字是否被定义；

`UnaryExp → Ident '(' [FuncRParams] ')':`

由于是函数调用，所以检查函数表内是否有该名字。

函数参数个数不匹配	d

函数参数类型不匹配	e

对函数参数的校验一起进行：`checkParas(int lineNo, ArrayList<Integer> paras, ArrayList<Integer> rparas)`

无返回值的函数存在不匹配的return语句	f

全局维护了`needReturn`变量表明此时解析的函数是否需要返回值，如果`(!needReturn)`且此时解析的`stmt`是return语句的话，则报错。

有返回值的函数缺少return语句	g

解析完一个`Block`后返回值`isReturn`，表示该`Block`的最后一条语句是否为`return`语句，如果`(needReturn && !isReturn)`，则报错。报错行号为函数结尾的'}'所在行号，所以在出`parseBlock`()时全局记录该行号。

不能改变常量的值	h

<Stmt>→<LVal>'=' <Exp>';' | <LVal>'=' 'getint' '(' ')' ';

调用`isConst(ident)`方法识别LVal是否类型为`const`

缺少分号	**i**
缺少右小括号')'	j
缺少右中括号']'	k

注意报错行号是*;)]本应该正常出现的位置的上一个单词所在行号*，即`(tokens.get(index-2)).getLineNo()`。(index: 待读取字符)

printf中格式字符与表达式个数不匹配	**l**

表达式个数根据','判断，格式字符根据'%d'计算。

在非循环块中使用break和continue语句	**m**

全局维护`int cycleDepth`表示当前处于第几层嵌套循环，`cycleDepth=0`时使用`break/continue`报错。

编码后修改：

一开始栈式符号表采用`hashmap`存储`<key='scope', value=当前作用域的SymbolTable>`,但由`hashmap`存储的无序性可知在使用`getSymbol`方法时可能导致取到外层同名变量，故改用`ArrayList`存储，用`index`下标代替`scope`。

```
private Symbol getSymbol(Token word) {
        Symbol symbol = null;
        for (SymbolTable s : symbols) {
            if (s.hasSymbol(word)) {
                symbol = s.getSymbol(word);
            }
        }
        return symbol;
    }
```

补充说明中提到全局变量和函数不能重名，所以需要增加函数名和符号表第0层即全局符号的比较。

四、代码生成

编码前设计：

在语法分析的过程中同步进行，即在文法的合适位置插入生成中间代码的“动作符号”，最后将中间代码交给`PCodeExecutor`在模拟运行栈上执行得到结果。

中间代码解释 MidType类

MidType	value1	value2	
VAR	scopeId+"_"+sym.getVal() [作用域id+variable名字], 保证每个名字是唯一的		变量定义/常量定义，之后紧接着就是各维数的exp，最后是DIMVAR

MidType	value1	value2	
PUSH	Integer		parseNumber()
POP	scopeId+"_"+sym.getVal() 左值名		LVal = getint() / exp
JZ	label名		Jump if zero
MAIN	main		
PARA	scopeId+"_"+sym.getVal() 形参名	该形参维数	
RPARA	dimensions	该实参维数	
CALL	funcName		调用函数
RET	1/0 是否返回了值?		从该函数返回
VALUE	var名	dimensions	获取该var的值，然后push回栈中
ADDRESS	var名	dimensions	获取该var的地址，然后push回栈中
PLACEHOLDER	var名	dimensions	变量暂时没有赋初值时的占位符
PRINT	“ ”strCon	参数个数(%d)	
DIMVAR	var名	dimensions	对于VAR多维数问题的补充说明

RetInfo类

int PC	运行完本函数后 跳转至哪条指令执行
HashMap<String, Var> varTable	保存之前运行环境的变量表
int stackPtr	栈顶指针
int paraNum	
int callArgsNum	
int nowParasNum	

Var类

int index	在栈中存放的位置
int dimension = 0	
int dim1	
int dim2	

```
函数定义的流程：
FuncDef → FuncType Ident '(' [FuncFParams] ')' Block
**PCode code = new PCode(MidType.FUNC, sym.getVal());
FuncFParams → FuncFParam { ',' FuncFParam }
FuncFParam → int Ident ['[' ']' { '[' ConstExp ']' }]
如果是数组的话，parseConstExp()->parseAddExp()->.....解析一个表达式，但在执行过程中最后会变成栈中的一个数值（因为是常量表达式是可计算的）
**PCode(MidType.PARA, scopeId + "_" + ident.getVal(), dimensions);
```

在执行过程中，先CALL FUNC，即解析实参，再跳转到被调用函数中去。调用时，调用方已经用实参在栈中占好位置了。现在跳转到被调用函数中计算，只需要形参走到占好的位置然后操控这些实参进行运算即可。

```
函数调用的流程：
一元表达式 UnaryExp → Ident '(' [FuncRParams] ')'
FuncRParams → Exp { ',' Exp }
每结束一次 parseExp()：
**codes.add(new PCode(MidType.RPARA, dimensions));
结束了对所有实参的解析后：
**codes.add(new PCode(MidType.CALL, ident.getVal()));
```

`PCodeExecutor`执行

先对所有中间代码进行一遍扫描，记录`mainAddress`[`main`函数入口地址]、`labelTable`[跳转目标`label`地址]、`funcTable`[函数表]；顺序执行，当前执行中间代码指针为`pc`：

codeType	operation
VAR	变（常）量定义，存入varTable(+globalVarTable)
PUSH	将值压栈
POP	弹出值和地址，向栈中该地址写入值
ADD/SUB...	弹出栈顶两元素计算后将结果压栈
CMPEQ...	弹出栈顶两元素比较后将结果(相等为1否则为0)压栈
AND...	弹出栈顶两元素计算后(true为1否则为0)将结果压栈
JZ	读取栈顶计算结果，如果为0去labelTable读取跳转目标的pc值，更改当前pc
FUNC	函数定义已经在第一遍扫描时处理过了，此时直接进入main函数执行
MAIN	设置当前不处于全局状态，增加一条RetInfo，设置新的局部变量表
PARA	
RPARA	实参，在rparasIndex中存储该实参地址
GETINT	scanner.nextInt()
CALL	增加一条RetInfo，设置新的局部变量表/pc，设置当前函数需要几个参数及现在读取了几个参数(目前=0)
RET	删除一条RetInfo，恢复局部变量表/pc/callArgsNum/callArgsNum(满足调用函数时参数是对其他函数的调用的情况)，如果有返回值，留下原来栈顶的元素（返回值），并将其他栈上元素（入参、局部变量）清除。
PLACEHOLDER	为该变量置0，正好满足了全局变量不初始化默认为0的要求
VALUE	获取变量值并push到栈顶
ADDRESS	获取变量地址并push到栈顶
PRINT	将要输出的内容存入list，不是%d就原样输出，是就用对应param代替
DIMVAR	弹出栈顶元素，为VAR设置维数值
EXIT	return结束全部执行

codeType	operation