设计文档

1. 参考编译器介绍

本次实验，根据课程组老师和助教的指导，我主要参考了课程组提供的tolangc源代码，以及LLVM的官方项目。通过对这部分代码的阅读和学习，对于我后续的编译器设计有较大的帮助。

关于tolangc源代码的阅读

总体结构

代码源文件主要存在于libs文件夹中，分为四部分：tolang（用于处理词法，语法和语义分析，包含错误处理），pcode（生成pcode代码），llvm（生成llvm代码），mips（生成mips代码）

接口设计

词法分析:

```
void next(Token &token);
```

词法分析通过next函数，将源文件的字符串分析成为一组token。

语法分析:

```
std::unique_ptr<CompUnit> parse();
```

这个方法可以根据词法类lexer进行语法分析，以及进行相关的错误处理。

代码生成

在代码生成部分，采用访问者模式，访问生成的语法树，生成符号表并进行错误处理

```
void visit(const CompUnit &node);
```

在AsmPrinter类中，保留了输出的接口

```
void Print(ModulePtr module, std::ostream &out);
```

翻译成mips，保留了两个对外的接口。一个用于将module翻译成mips，另一个用于输出。

```
void translate(const ModulePtr &modulePtr);
void print(std::ostream &_out);
```

文件组织

```
文件组织结构:.
|   CMakeLists.txt
|
+---bin
|       CMakeLists.txt
|       main.cpp
```

```
|
\---libs
    |   CMakeLists.txt
    |
    +---llvm
    |   |   CMakeLists.txt
    |   |
    |   +---include
    |   |   \---llvm
    |   |       |   utils.h
    |   |       |
    |   |       +---asm
    |   |       |       AsmPrinter.h
    |   |       |       AsmWriter.h
    |   |       |
    |   |       \---ir
    |   |           |   IrForward.h
    |   |           |   Llvm.h
    |   |           |   LlvmContext.h
    |   |           |   Module.h
    |   |           |   SlotTracker.h
    |   |           |   Type.h
    |   |           |
    |   |           \---value
    |   |               |   Argument.h
    |   |               |   BasicBlock.h
    |   |               |   Constant.h
    |   |               |   ConstantData.h
    |   |               |   Function.h
    |   |               |   GlobalValue.h
    |   |               |   Use.h
    |   |               |   User.h
    |   |               |   Value.h
    |   |               |
    |   |               \---inst
    |   |                       ExtendedInstructions.h
    |   |                       Instruction.h
    |   |                       Instructions.h
    |   |                       InstructionTypes.h
    |   |
    |   \---src
    |       |   utils.cpp
    |       |
    |       +---asm
    |       |       AsmPrinter.cpp
    |       |       AsmWriter.cpp
    |       |       TypePrinter.cpp
    |       |       ValuePrinter.cpp
    |       |
    |       \---ir
    |           |   LlvmContext.cpp
    |           |   Module.cpp
    |           |   SlotTracker.cpp
    |           |   Type.cpp
    |           |
    |           \---value
```

```
|               |   Argument.cpp
|               |   BasicBlock.cpp
|               |   ConstantData.cpp
|               |   Function.cpp
|               |   Use.cpp
|               |   User.cpp
|               |   Value.cpp
|               |
|               \---inst
|                       ExtendedInstructions.cpp
|                       Instructions.cpp
|                       InstructionTypes.cpp
|
+---mips
|   |   CMakeLists.txt
|   |
|   +---include
|   |   \---mips
|   |           mips_forward.h
|   |           mips_inst.h
|   |           mips_manager.h
|   |           mips_reg.h
|   |           translator.h
|   |
|   \---src
|           mips_manager.cpp
|           mips_printer.cpp
|           translator.cpp
|
+---pcode
|   |   CMakeLists.txt
|   |
|   +---include
|   |   \---pcode
|   |       |   PcodeBlock.h
|   |       |   PcodeFunction.h
|   |       |   PcodeInstruction.h
|   |       |   PcodeInstructions.h
|   |       |   PcodeModule.h
|   |       |   PcodeSymbol.h
|   |       |   PcodeVariable.h
|   |       |
|   |       \---runtime
|   |               ActivityRecord.h
|   |               PcodeRuntime.h
|   |
|   \---src
|           PcodeRuntime.cpp
|           PcodeSymbol.cpp
|
\---tolang
    |   CMakeLists.txt
    |
    +---include
    |   \---tolang
    |           ast.h
```

```
        |           error.h
        |           lexer.h
        |           parser.h
        |           symtable.h
        |           token.h
        |           utils.h
        |           visitor.h
        |
        \---src
                ast.cpp
                lexer.cpp
                parser.cpp
                visitor.cpp
```

2. 编译器总体设计

编译器大体上分为前端，中端和后端三个部分。其中，前端包括词法分析程序，语法分析程序。中端包括符号表管理程序，语义分析程序，生成中间代码和代码优化程序；后端包括生成目标代码程序。其中每个程序都留出对应的接口，每一步程序只需调用上一步程序即可运行。

总体结构

设计的编译器主要分为五部分：

Lexer:词法分析

Parser:语法分析

Symbol:语义分析（符号表分析，并做出错误分析）

ErrorHandle:错误处理程序

LLVM:生成LLVM IR的程序

Mips:生成Mips代码的程序

接口设计

在每部分都保留了相应的接口。每一部分采取单例模式，防止冲突和干扰

主要接口用于在main函数调用处理过程，以及输出对应内容，另外就是像单词表，语法树，符号表，这些可能需要被别的程序调用，这些端口也进行了放开。

Lexer:

```
public:
    //初始化，让指针指向源代码source
    void setCharPtr(char*source);
    //完全处理所有的单词
    void handleSource();
    //获取读取的单词
    string getToken();
    //输出正确的内容
    void printRight();
    //输出错误的内容
    void printWrong();
    //获取tokenMap
    map<int,Word> getWordMap();
```

Parser

```
public:
    void ParserHandle();
    //输出至正确的文本
    void printParser();
    static Parser* getParserInstance();
    TreeNode* getPaserTree();
```

SymbolHandle

```
public:
    static SymbolHandle* getSymbolHandleInstance();
    void handleSymbol();
    void printSymbol();//输出全部的符号表
    shared_ptr<SymbolTable> getSymbolTable();//获取全局符号表
```

ErrorHandle

```
public:
    //根据错误类别码和行号，输入至文件中
    static void printfError();
    static bool isError;
    //将文件出现的错误按照顺序加入map
    static void printError(ErrorCategory error_category,int line_num);
```

LLVM

```
public:
    static GenerateIR* getInstance();
    Module* getModule();
    //生成LLVM IR的过程
    void generateLLVMIR();
    //输出LLVM代码
    void printLLVMIR();
    //遍历语法树的过程
    void traverseTreeIR(TreeNode* root);
```

Mips

```
public:
    static MipsGenerate* getInstance(Module* module);
    //生成Mips代码
    void generateMips();
    //输出Mips
    void printMips();
```

文件组织

文件组织如下（不包括优化）：

```
文件结构:.
|   CMakeLists.txt
|   config.json
|   main.cpp
|
+---ErrorHandle
|   +---include
|   |       ErrorCategory.h
|   |       ErrorPrint.h
|   |
|   \---src
|           ErrorCategory.cpp
|           ErrorPrint.cpp
|
+---Lexer
|   +---include
|   |       FileProcess.h
|   |       Lexer.h
|   |       TokenType.h
|   |
|   \---src
|           FileProcess.cpp
|           Lexer.cpp
|           TokenType.cpp
|
+---LLVM
|   +---include
|   |   |   GenerateIR.h
|   |   |   instanceof.h
|   |   |
|   |   +---generate
|   |   |       LLVMExp.h
|   |   |       LLVMGenerate.h
|   |   |
|   |   +---LLVMSymbol
|   |   |       SymbolCalculate.h
|   |   |       SymbolValue.h
|   |   |
|   |   +---optimize
|   |   |       OptimizerInit.h
|   |   |
|   |   +---type
|   |   |   |   IRName.h
```

```
|   |   |   |   IRPreName.h
|   |   |   |   IRType.h
|   |   |   |
|   |   |   \---irType
|   |   |           IRArray.h
|   |   |           IRBlock.h
|   |   |           IRBool.h
|   |   |           IRChar.h
|   |   |           IRInt.h
|   |   |           IRPointer.h
|   |   |           IRVoid.h
|   |   |
|   |   \---value
|   |       |   Value.h
|   |       |
|   |       +---architecture
|   |       |   |   BasicBlock.h
|   |       |   |   ConstString.h
|   |       |   |   ConstValue.h
|   |       |   |   Module.h
|   |       |   |   Param.h
|   |       |   |
|   |       |   +---data_structure
|   |       |   |       InitVar.h
|   |       |   |       Loop.h
|   |       |   |
|   |       |   \---user
|   |       |       |   Function.h
|   |       |       |   GlobalValue.h
|   |       |       |   GlobalVariable.h
|   |       |       |   Instruction.h
|   |       |       |
|   |       |       \---instruction
|   |       |           |   AllocaInstruction.h
|   |       |           |   BrInstruction.h
|   |       |           |   CalculateInstruction.h
|   |       |           |   CallInstruction.h
|   |       |           |   GetelementptrInstruction.h
|   |       |           |   IcmpInstruction.h
|   |       |           |   JumpInstruction.h
|   |       |           |   LoadInstruction.h
|   |       |           |   ReturnInstruction.h
|   |       |           |   StoreInstruction.h
|   |       |           |   TruncInstruction.h
|   |       |           |   ZextInstruction.h
|   |       |           |
|   |       |           \---IOInstruction
|   |       |                   IOGetChar.h
|   |       |                   IOGetInt.h
|   |       |                   IOInstruction.h
|   |       |                   IOPutCh.h
|   |       |                   IOPutInt.h
|   |       |                   IOPutStr.h
|   |       |
|   |       \---user
|   |               Use.h
```

```
|   |               User.h
|   |
|   \---src
|       |   GenerateIR.cpp
|       |
|       +---generate
|       |       LLVMExp.cpp
|       |       LLVMGenerate.cpp
|       |
|       +---LLVMSymbol
|       |       SymbolCalculate.cpp
|       |       SymbolValue.cpp
|       |
|       +---optimize
|       |       OptimizerInit.cpp
|       |
|       +---type
|       |   |   IRName.cpp
|       |   |   IRPreName.cpp
|       |   |   IRType.cpp
|       |   |
|       |   \---irType
|       |           IRArray.cpp
|       |           IRBlock.cpp
|       |           IRBool.cpp
|       |           IRChar.cpp
|       |           IRInt.cpp
|       |           IRPointer.cpp
|       |           IRVoid.cpp
|       |
|       \---value
|           |   Value.cpp
|           |
|           +---architecture
|           |   |   BasicBlock.cpp
|           |   |   ConstString.cpp
|           |   |   ConstValue.cpp
|           |   |   Module.cpp
|           |   |   Param.cpp
|           |   |
|           |   +---data_structure
|           |   |       InitVar.cpp
|           |   |       Loop.cpp
|           |   |
|           |   \---user
|           |       |   Function.cpp
|           |       |   GlobalValue.cpp
|           |       |   GlobalVariable.cpp
|           |       |   Instruction.cpp
|           |       |
|           |       \---instruction
|           |           |   AllocaInstruction.cpp
|           |           |   BrInstruction.cpp
|           |           |   CalculateInstruction.cpp
|           |           |   CallInstruction.cpp
|           |           |   GetelementptrInstruction.cpp
```

```
|           |           |   IcmpInstruction.cpp
|           |           |   JumpInstruction.cpp
|           |           |   LoadInstruction.cpp
|           |           |   ReturnInstruction.cpp
|           |           |   StoreInstruction.cpp
|           |           |   TruncInstruction.cpp
|           |           |   ZextInstruction.cpp
|           |           |
|           |           \---IOInstruction
|           |                   IOGetChar.cpp
|           |                   IOGetInt.cpp
|           |                   IOInstruction.cpp
|           |                   IOPutCh.cpp
|           |                   IOPutInt.cpp
|           |                   IOPutStr.cpp
|           |
|           \---user
|                   Use.cpp
|                   User.cpp
|
+---Mips
|   +---include
|   |   |   MipsGenerate.h
|   |   |
|   |   +---generate
|   |   |       MipsCell.h
|   |   |       MipsContent.h
|   |   |
|   |   +---Register
|   |   |       RegisterController.h
|   |   |       RegisterTool.h
|   |   |
|   |   \---structure
|   |       |   DataSegment.h
|   |       |   MipsStructure.h
|   |       |   Register.h
|   |       |   Segment.h
|   |       |   TextSegment.h
|   |       |
|   |       +---data
|   |       |       AsciizStructure.h
|   |       |       ByteStructure.h
|   |       |       DataMipsStructure.h
|   |       |       SpaceStructure.h
|   |       |       WordStructure.h
|   |       |
|   |       \---text
|   |           |   MipsBlock.h
|   |           |
|   |           +---MipsInstruction
|   |           |       Annotation.h
|   |           |       BTypeInstruction.h
|   |           |       ITypeInstruction.h
|   |           |       JalInstruction.h
|   |           |       JInstruction.h
|   |           |       LSTypeInstruction.h
```

```
|   |           |       MTypeInstruction.h
|   |           |       RTypeInstruction.h
|   |           |       SyscallInstruction.h
|   |           |
|   |           \---PseudoInstruction
|   |                   LaInstruction.h
|   |                   LiInstruction.h
|   |                   MoveInstruction.h
|   |
|   \---src
|       |   MipsGenerate.cpp
|       |
|       +---generate
|       |       MipsCell.cpp
|       |       MipsContent.cpp
|       |
|       +---Register
|       |       RegisterController.cpp
|       |       RegisterTool.cpp
|       |
|       \---structure
|           |   DataSegment.cpp
|           |   MipsStructure.cpp
|           |   Register.cpp
|           |   Segment.cpp
|           |   TextSegment.cpp
|           |
|           +---data
|           |       AsciizStructure.cpp
|           |       ByteStructure.cpp
|           |       DataMipsStructure.cpp
|           |       SpaceStructure.cpp
|           |       WordStructure.cpp
|           |
|           \---text
|               |   MipsBlock.cpp
|               |
|               +---MipsInstruction
|               |       Annotation.cpp
|               |       BTypeInstruction.cpp
|               |       ITypeInstruction.cpp
|               |       JalInstruction.cpp
|               |       JInstruction.cpp
|               |       LSTypeInstruction.cpp
|               |       MTypeInstruction.cpp
|               |       RTypeInstruction.cpp
|               |       SyscallInstruction.cpp
|               |
|               \---PseudoInstruction
|                       LaInstruction.cpp
|                       LiInstruction.cpp
|                       MoveInstruction.cpp
|
+---Parser
|   +---include
|   |       Parser.h
```

```
|   |       ParserTree.h
|   |       TokenScanner.h
|   |
|   \---src
|           Parser.cpp
|           ParserTree.cpp
|           TokenScanner.cpp
|
\---Symbol
    +---include
    |       Symbol.h
    |       SymbolHandle.h
    |
    \---src
            Symbol.cpp
            SymbolHandle.cpp
```

3. 词法分析设计

3.1 编码前的设计

分为三个功能模块：

1. 划分单词，同时提取出类别、值等信息
2. 处理注释
3. 统计行号

类设计：

- Lexer： 词法分析器类（单例模式）
- TokenType： 单词类型枚举类（单例模式，因为单词类型是固定的）
- FileProcess：程序文件的读取，将文件读成一个字符串（不是按行读入，是一次性读入）

词法分析器主要接口：

1. handleNext() 处理下一个单词
2. getToken() 获得读取的单词值
3. getTokenType() 获得读取的单词类型
4. getLexerError() 获取错误的位置和错误码

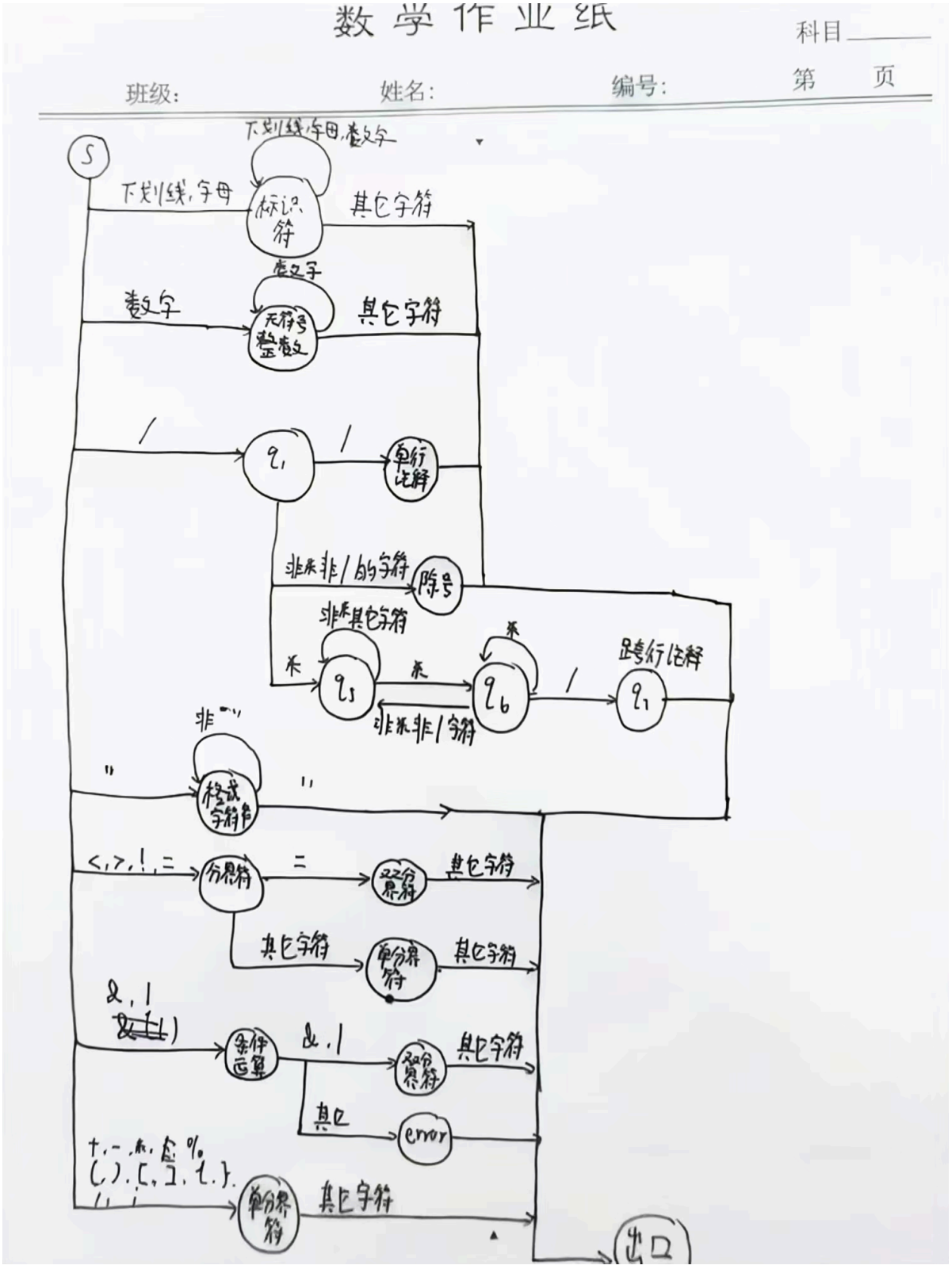
词法分析器主要数据成员：

- source：源程序字符串
- charPtr：当前字符串位置指针
- token：解析单词值
- tokenType：解析单词类型
- reserveWords：保留字表
- tokenTypeMap：单词类别码map
- lineNum：当前行号

- number：解析数值

状态图设计：

Token状态图



单词枚举类的主要接口：

1. getTokenType(const string& token)：根据单词token获取对应的类型
2. map<string, string> getTokenTypeMap()：获取单词种类存储的map

单词枚举类主要数据成员：

tokenTypeMap：记录单词类型的map

文件读入类主要接口：
string readFileIntoString(const char* filePath)：读入指定路径的文件

3.2 编码完成之后的修改

1. 首先，我发现上面的状态图缺少了**如何识别“字符常量”**，其实逻辑和字符串一样，只不过字符串先识别双引号，字符常量识别单引号。
2. 其次，在编写过程中发现一个问题：**如果字符串或者字符常量中有转义字符'或者"，那么这应当是一个字符串内容，而不是跳出“字符串状态”的符号**，所以状态机需要有对应的修改。**具体而言，跳出的状态不再只是“是否再次读到双引号"，而是读到双引号后前面是否还有转义字符**
3. 此外，在2的基础上，需要考虑这种情况：'\\'，这个字符常量**代表了\的转义字符，不能与最后的单引号相匹配。**所以在判断的时候，如果当前是单引号，上一位是\，那么如果上一位不是\，才能说明这个单引号'是字符，否则仍然是"字符/字符串的终结处"。
4. 同时，在提交评测时，评测机上却没有输出，经助教帮助查明，原因是**滥用static设定的变量和方法，导致初始化顺序出现了问题。**所以将代码不使用static方法。
5. 另外，在识别数字的过程中，负数属于“减号+常数字”，而数字范围存在很大的情况，所以不能使用int，要使用long long
6. 最后关于注释退出条件的判断：当单行注释位于最后一行时，末尾没有字符'\n'，所以不能把这个作为判断条件；考虑'\n'+'\0'判断是否结束。同时关于指针回退，如果读到了'\n'，为了方便统计行数，需要回退；但是如果读的是最后一行，即不是'\n'，就不再需要回退了，否则，回退后反而会使下一轮读到注释最后的字符，多输出一行错误结果。

3.3 最终词法分析器代码

下面是TokenType类的定义，其中在构造函数处对tokenTypeMap初始化

```
//单词类别
class TokenType {
public:
    //根据单词token获取对应的类型
    string getTokenType(const string& token);
    //获取单例模式TokenType
    static TokenType* getTokenTypeInstance();
    //进程退出时释放单实例
    static void deleteTokenTypeInstance();
    //获取map
    map<string, string> getTokenTypeMap();
protected:
    //构造函数
    TokenType();
private:
    //记录单词类型的map
    map<string, string> tokenTypeMap;
    // 唯一单实例对象指针
    static TokenType *g_pTokenType;
};
```

下面是词法分析类Lexer的部分定义，主要涉及到核心内容

```
class Lexer {
public:
```

```
    static Lexer* getLexerInstance();
    static void deleteLexerInstance();
    //初始化，让指针指向源代码source
    void setCharPtr(char*source);
    //处理下一个单词
    pair<string, string> handleNext();
    //处理的具体过程
    int handle_next();
    //完全处理所有的单词
    void handleSource();
    //获取读取的单词
    string getToken();
    //输出正确的内容
    void printRight();
    //输出错误的内容
    void printWrong();
    //错误处理
    void lexerError();
private:
    static Lexer* lexerInstance;
    //存储单词类别和单词值
    map<int,pair<string, string>> tokenMap;
    //行号+错误码
    map<int,char> lineWrong;
    string token;//当前处理的单词
    char Char = ' ';//存取当前读进的字符
    int num = 0;//存入当前读入的整型变量
    char* charPtr = nullptr;//字符指针
    int lineNum = 1;//行数
    string tokenType;
    //其它处理过程，只有handle_next需要调用，如判断当前字符是不是字母之类的，不再赘述
}
```

4. 语法分析设计

4.1 编码前的设计

语法分析按照课堂上所讲的内容，计划采用递归下降子程序法进行分析语法。
计划分为一个总的模块，下属每个语法成分都对应一个语法分析子程序，以及错误处理程序。
在词法分析中，tokenMap用来记录整个文件出现的单词（按照读入顺序），那么取单词可以直接利用，包括超前读取单词，都从这里读取即可，不再涉及以前的文件和source字符串。
同时，考虑到错误记录，需要在词法分析中记录单词的行数。所以我直接计划采用结构体的方式，定义单词的结构体，包含单词内容，类别码和行号。

4.1.1 类设计：

TokenScanner

扫描单词(tokenMap)，包括获取当前单词，回退以及提前预读单词。
包含成员：nowWord——当前单词，preRead[]——预读单词数组

ParserTree

建立语法树，主要是将给定的单词/类别整合到树上

ErrorPrint,ErrorCategory

错误输出，以及错误类型的枚举类
在本次分析中，错误只会出现i,j,k三种错误。

Parser

这个类是语法分析主要处理类，其中也包括了**每种语法类型**自己的处理程序和错误处理函数。

4.1.2 主要用到的变量

now_word: 代表当前读的单词
preWords: 代表预读的单词指针，因为可能要预读多个
root: 语法树根节点

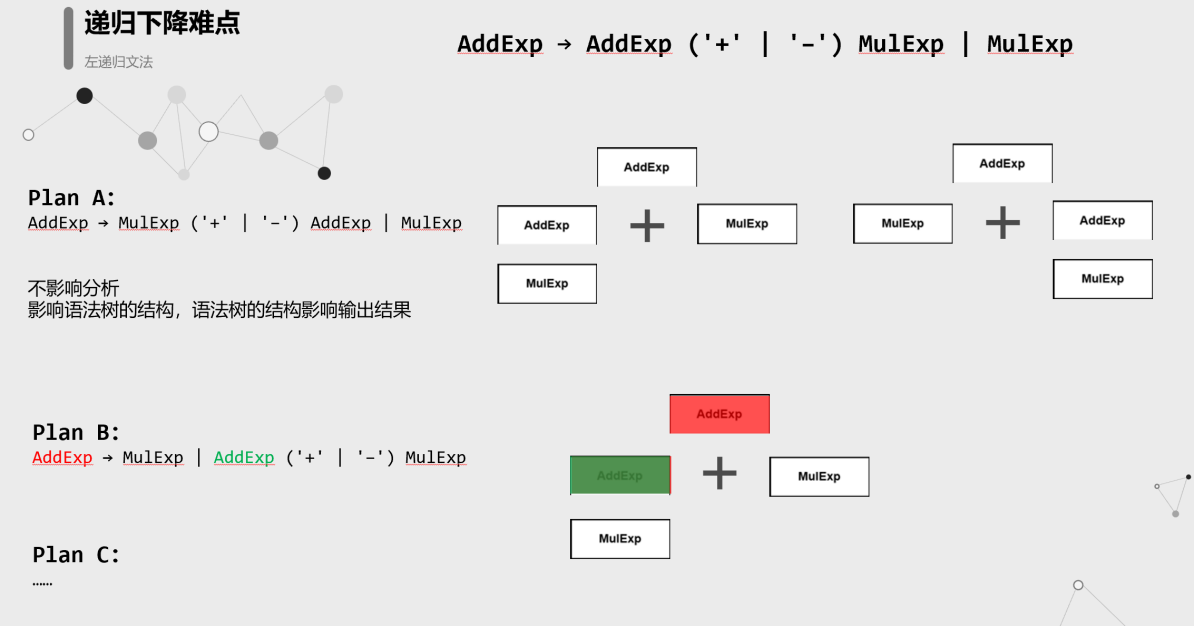
4.1.3 关于文法的处理

本次文法的语法分析采用递归下降法，用哪个程序递归取决于预读的单词属于哪个非终结符的First集。
对于左递归文法，根据课堂上所讲的转换方法，转换成Backus范式的循环即可解决。
比如，乘除模表达式 MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
可以转换成 **UnaryExp {('*' | '/' | '%') UnaryExp}。**
又如 AddExp → MulExp | AddExp ('+' | '?') MulExp，
可以转换成 **MulExp {('+' | '?') MulExp}** 等等。

4.2 编码完成后的修改

4.2.1 左递归文法的语法树问题

按照上述内容，存在一个问题，就是树的结构会少一块，原本AddExp要连接Addexp和MulExp，现在变成连接了多个MulExp，所以还需要进一步处理语法树。



语法树的调整

> 语法树的调整
> 本来，对于左递归文法，是直接转成循环处理
> 但是循环处理后，构建语法树会发现他们都作为同一类在同一层
> 比如AddExp = AddExp + MulExp
> 循环后，树AddExp下面一层有多个MulExp，每两个之间还有运算符
> 而根据题目，题目希望建的树就是AddExp + MulExp这三部分
> 所以要调整树，从右往左重新连接。

4.2.2 Stmt的识别问题

在文法Stmt的识别中，存在两组

```
Stmt → LVal '=' Exp ';'
| [Exp] ';' // i
```

这里存在一个问题，如何识别是Exp还是LVal?
观察语法，我们可以发现，Exp可以推出LVal,并且LVal也可以推出Exp
而Exp又可以推出AddExp，接下来进入左递归，很可能出现多个AddExp的情况
所以，通过"预读"的方式来判断进入哪个子程序并不合理。只能通过**回溯**的方式解决。

针对回溯，我的策略是先假设是LVal。记录当前的位置指针。处理完成后，如果预读下一位是"="，就说明假设正确；否则，选择回溯，将位置指针调整回来，然后按照Exp处理。

4.2.3 报错处理

在本次测试中，最后一个点未通过，经尝试，发现其要求错误处理按照行号的顺序输出。
但是不同错误在不同程序中识别，所以我决定用一个map存储这些错误，再根据行号顺序输出。
然而，还存在一个问题：同一行出现两个错误，单纯的map会顶替掉前一个错误，所以修正为vector<pair<int,char>>，左侧int是行号，右侧是错误码，默认情况下sort函数通过pair第一个元素排序。

4.3 最终语法分析类的相关定义代码

ErrorCategory错误类别

```
//
// Created by PengXinyang on 24-9-28.
//

#ifndef ERRORCATEGORY_H
#define ERRORCATEGORY_H
#include<map>
#include <unordered_map>
using namespace std;
//错误类别码
enum  class ErrorCategory {
    illegal_symbol,//a，非法符号
    redefine,//b,名字重定义
    undefined,//c,未定义名字
    param_num_not_match,//d,参数个数不匹配
    param_type_not_match,//e,参数类型不匹配
    return_not_match,//f,无返回值函数返回值不匹配
```

```
    return_lack,//g,缺少返回值
    const_error,//h,不能改变常量的值
    semicolon_lack,//i,缺少分号
    r_parenthesis_lack,//j,缺少右小括号
    r_bracket_lack,//k,缺少右中括号
    format_num_not_match,//l,printf格式字符个数不匹配
    break_continue_not_loop//m,非循环使用break和continue
};
inline map<ErrorCategory,char> ErrorMap={
    {ErrorCategory::illegal_symbol,'a'},
    {ErrorCategory::redefine,'b'},
    {ErrorCategory::undefined,'c'},
    {ErrorCategory::param_num_not_match,'d'},
    {ErrorCategory::param_type_not_match,'e'},
    {ErrorCategory::return_not_match,'f'},
    {ErrorCategory::return_lack,'g'},
    {ErrorCategory::const_error,'h'},
    {ErrorCategory::semicolon_lack,'i'},
    {ErrorCategory::r_parenthesis_lack,'j'},
    {ErrorCategory::r_bracket_lack,'k'},
    {ErrorCategory::format_num_not_match,'l'},
    {ErrorCategory::break_continue_not_loop,'m'}
};

#endif //ERRORCATEGORY_H
```

ErrorPrint错误打印

```
//
// Created by PengXinyang on 24-9-28.
//

#ifndef ERRORHANDLE_H
#define ERRORHANDLE_H
#include <map>
#include <vector>

#include "ErrorCategory.h"

class ErrorPrint {
public:
    //根据错误类别码和行号，输入至文件中
    static void printfError();
    static bool isError;
    //将文件出现的错误按照顺序加入map
    static void printError(ErrorCategory error_category,int line_num);
private:
    static vector<pair<int,char>> error_vector;
};

#endif //ERRORHANDLE_H
```

TokenScanner读单词类

```
//
// Created by PengXinyang on 24-9-28.
//

#ifndef SCANNER_H
#define SCANNER_H
#include "../../Lexer/include/Lexer.h"

class TokenScanner {
public:
    static TokenScanner* getTokenScannerInstance();
    //读当前单词
    Word readNowWord();
    //预读k个单词
    Word* preReadWords(int k);
    //回退一个单词，通常是错误处理
    void retractWord();
    //获得当前单词
    Word getNowWord();
    //获得预读单词
    Word* getPreWords();
    int position = 0;//记录当前所在单词位置
private:
    TokenScanner();
    static TokenScanner* instance;
    Lexer* lexer;
    Word nowWord;//当前读的单词
    Word preRead[30];//预读的单词
    //单词map
    map<int,Word> wordMap;
};

#endif //SCANNER_H
```

ParserTree语法树类

```
//
// Created by PengXinyang on 24-9-29.
//

#ifndef PARSERTREE_H
#define PARSERTREE_H
#include <cstring>
#include<string>
#include <utility>
#include <vector>

#include "../../Lexer/include/Lexer.h"
using namespace std;
struct TreeNode {
    Word word;
    int son_num = 0;//统计有几个孩子
    vector<TreeNode*> sonNode;
    TreeNode() = default;
```

```
    explicit TreeNode(Word word){this->word=std::move(word);}
};

class ParserTree {
public:
    //连接语法树
    static TreeNode* catchTree(TreeNode* root, const Word &word);
    static TreeNode* catchTree(TreeNode* root, TreeNode* son);
    //建立语法树
    static TreeNode* createTree(const Word &word);
    //后序输出语法树
    static void printTree(FILE*fp, const TreeNode* root);
    //对于左递归文法，如果转换成课堂上的循环会导致语法树缺少，这个函数用于加起来
    static TreeNode* adjustTree(TreeNode* root,const Word &word);
};

#endif //PARSERTREE_H
```

语法分析程序类Parser

```
//
// Created by PengXinyang on 24-9-28.
//

#ifndef PARSER_H
#define PARSER_H
#include "ParserTree.h"
#include "TokenScanner.h"

class Parser {
public:
    void ParserHandle();
    //输出至正确的文本
    void printParser();
    static Parser* getParserInstance();
private:
    TreeNode *ParserRoot = nullptr;
    static Parser *instance;
    TokenScanner*tokenScanner = TokenScanner::getTokenScannerInstance();
    Word now_word;
    Word*preWords = nullptr;
    //具体函数内部进行了递归下降子程序法
    TreeNode* CompUnit();//编译单元
    TreeNode* Decl();//声明
    TreeNode* ConstDecl();//常量声明
    TreeNode* Btype();//基本类型
    TreeNode* ConstDef();//常量定义
    TreeNode* ConstInitVal();//常量初值
    TreeNode* VarDecl();//变量声明
    TreeNode* VarDef();//变量定义
    TreeNode* InitVal();//变量初值
    TreeNode* FuncDef();//函数定义
    TreeNode* MainFuncDef();//主函数定义
    TreeNode* FuncType();//函数类型
```

```
    TreeNode* FuncFParams();//函数形参表
    TreeNode* FuncFParam();//函数形参
    TreeNode* Block();//语句块
    TreeNode* BlockItem();//语句块项
    TreeNode* Stmt();//语句
    TreeNode* ForStmt();//for语句
    TreeNode* Exp();//表达式
    TreeNode* Cond();//条件表达式
    TreeNode* LVal();//左值表达式
    TreeNode* PrimaryExp();//基本表达式
    TreeNode* Number();//数值
    TreeNode* Character();//字符
    TreeNode* UnaryExp();//一元表达式
    TreeNode* UnaryOp();//单目运算符
    TreeNode* FuncRParams();//函数实参表
    TreeNode* MulExp();//乘除模表达式
    TreeNode* AddExp();//加减表达式
    TreeNode* RelExp();//关系表达式
    TreeNode* EqExp();//相等性表达式
    TreeNode* LAndExp();//逻辑与表达式
    TreeNode* LOrExp();//逻辑或表达式
    TreeNode* ConstExp();//常量表达式
};

#endif //PARSER_H
```

具体实现的源码较多，超过一千行，所以在这里不再展示cpp源码。

5. 语义分析设计

5.1 编码前的设计

在语义分析中，主要做的事情是创建符号表和处理剩下的错误。
关于符号表，首先保留字，运算符这些内容不需要添加进符号表，只需要添加定义的**常量，变量和函数名**即可。
在语法分析中，已经完成了语法树的构建。通过观察我们可以发现，首先除去最外层全局符号表，每一层符号表都要依托于Block结点，只有进入Block后才会创建新的符号表（如果是函数形参则需要提前创建）。而**对于变量，一定在变量声明 VarDecl结点下面。**常量符号和函数符号的获取同理，根据对应树的节点即可。
我在语法分析时将const声明和var声明整合进了Decl节点，所以可以先找到这个节点。
另外，函数形参不属于Decl节点，属于FuncFParam节点，这里也需要考虑。
再其次，Exp节点也需要考虑，因为里面存在使用符号的情况，而符号可能未定义，需要相应的错误处理。

5.1.1 类的设计

Symbol：符号类，主要定义出现的符号单词，以及各种各样的属性。包括是不是变量，是不是数组，是不是函数，是不是常量等内容，以及单词的行号

SymbolTable：符号表类，包括指向父符号表的指针，指向子符号表的指针。以及本身要自带存符号的unordered_map（查询，存取速度更快），以及标记先后顺序的token_vector。同时，也有添加符号，设置符号的值，以及递归根据token查找符号的方法。

SymbolHandle：符号处理类，单例模式，包括了递归分析语法树，建立符号表树的方法。其中包含了一些参数，包括当前所在的函数名称token，是否在main函数，main函数是否有返回值等，当前在循环的第几层。当然，也记录了当前所在的符号表指针和全局符号表的指针。

5.1.2 主要用到的变量

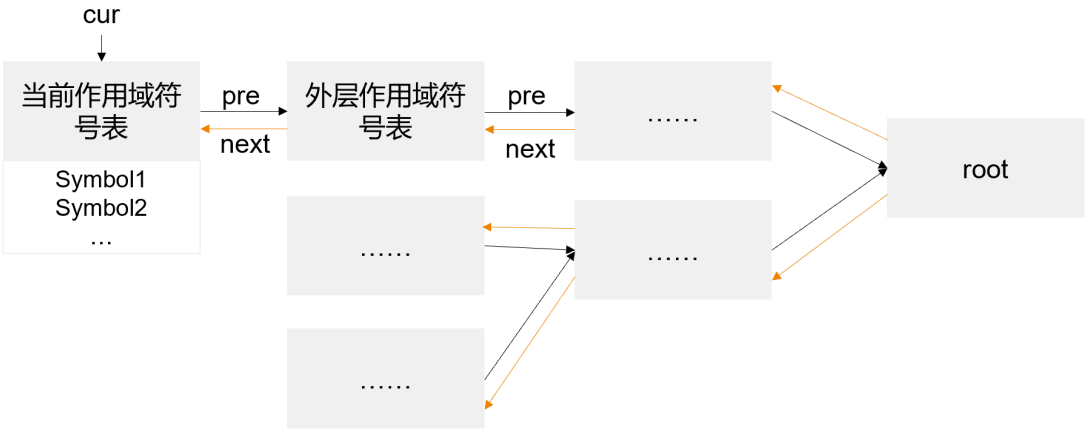
nowSymbolTable：当前所在的符号表

son_nodes：语法树某一结点的孩子节点

symbol：单词token对应的符号

5.1.3 符号表的处理

本次是针对已经创建好的语法树，还是沿用递归下降法，对于每个节点分别分析，结构上类似于语法分析。
在语法树中，遇到Ident，考虑符号；如果是在Decl下找到的Ident，说明要添加符号（可能出现重定义的错误）；否则，就是调用这个符号（可能出现未定义的错误）。除此之外，在函数部分，要创建一个新的符号表（方便形参符号的加入）；进入Block，如果上一层是函数就不建表，否则再建一个表。
除此之外，不涉及符号表的新建和符号的新建。



5.2 编码完成后的修改

本次代码书写有些杂乱，导致改动的东西不少。

首先，由于涉及到对子节点的访问，所以有可能会出现数组越界的错误；后来我才采取了直接遍历子节点的方式，通过判断子节点的类型再调用。

当然，在处理比如返回值的问题时，需要调用“倒数第二个子节点”，这时候就需要保证不发生数组越界了。

关于错误处理方面：

1. 首先，我在语法分析的k类错误——缺少右中括号 出现了错误的判断，进了死循环。经查明，是我在ConstDef分析的时候，没有预读，而是直接使用了now_word判断。已经进行了修改。
2. 其次，关于函数return的错误，需要注意的是，如果是void，那么只要return后面有返回值，就报错；如果不是void，那么**如果最后一句话不是return**，就报错；同时还要注意，**只要最后一句话是return就没有问题**，哪怕没有返回值。
   例如：

```
int f(){
    return ;
}
```

这段代码是正确的，不属于符号表的错误

```
int f(){
    if(1 == 2) return 0;
    else return 1;
}
```

这段代码就是错误的，因为函数体最后一条语句不是return

3. **关于printf错误**
   由于本次实验的格式字符只包括%d和%c，所以我们读格式字符串时，需要根据%d和%c的个数来确定后面Exp是否符合。不能只读%，比如出现%lf，这里应当被识别为常规字符串。
4. **关于break，continue出现在for循环之外的错误**

需要注意到，for循环内还可以再套多个for循环
所以，选取一个计数器，遇见for+1，出去后-1，只要计数器小于等于0，就说明在for循环之外。

5.3 最终类的设计代码

symbol 符号类

```
class Symbol {
public:
    int symbol_id{};//符号的id
    int table_id{};//符号所在的符号表的id
    string token;//当前单词对应的字符串
    int lineNum{};//当前符号所在的行
    int btype = -1;//是什么数据类型，0是int 1是char
    int type = -1;//是什么类型，0是var，1是数组，2是函数
    bool is_const = false;//是不是常量
    bool is_func = false;//是不是函数
    bool is_func_param = false;//是不是函数的参数
    bool is_var = false;//是不是单元素变量
    bool is_array = false;//是不是数组
    //函数是否有返回值，这个变量是代码运行过程中逐渐改变的，用于判断f,g类错误
    bool is_return = false;
    int dim = 0;//代表数组的元素个数，如果是数组就启用
    vector<int> array_int_values;//如果是数组，用于记录数组的值
    vector<char> array_char_values;//如果是char数组，按照字符串记录即可
    int int_var_value = 0;//如果是变量，记录变量的值
    char char_var_value = 0;//字符型变量，记录变量的值
    int reg = 1;//寄存器是哪个，代码优化的时候再用
    int func_type = -1;//是函数的时候再启用，0 int 1 char 2 void
    int param_num = 0;//如果是函数，那么后面的参数数量
    vector<int> param_type;//函数形参的类型，0是变量，1是int数组，2是char数组
    int func_value = -1;//如果函数有返回值，则设置这个
    Symbol();
};
```

SymbolTable 符号表类

```
class SymbolTable {
public:
    int table_id = 1;//符号表id，有一个全局的数字统计。初始全局符号表id是1
```

```
    int layer = 1;//现在位于第几层。父符号表层数+1，初始时全局符号表位于第1层s
    shared_ptr<SymbolTable> father_ptr = nullptr;//父符号表，默认根节点的父亲为空指针
    vector<shared_ptr<SymbolTable>> children_ptr; // 子符号表
    unordered_map<string,Symbol> symbol_table;//符号表有哪些符号
    vector<string> token_vector;//将插入的符号单词按照插入顺序计入，便于输出时按照顺序输出
    SymbolTable();
    SymbolTable(int table_id, const shared_ptr<SymbolTable> &father_ptr);
    void print_symbol_table(FILE* fp);//输出符号表，fp是文件指针
    void print_all_symbol_table(FILE* fp);//输出符号表和子符号表，递归
    bool is_in_table(const string& token);//判断单词是否在当前符号表中
    bool is_in_all_table(const string& token);//递归判断单词是否在当前或父符号表中
    Symbol* get_symbol(const string& token);//获取当前符号表token对应的符号
    Symbol* get_symbol_in_all_table(const string& token);//在所有符号表遍历，寻找符号
    Symbol* get_last_symbol();//获取最后一个符号
    bool is_in_func_table(const string& token);//判断单词是否在符号表中，且是否为函数
    bool is_in_var_table(const string& token);//判断单词是否在符号表中，且是否为变量
    bool is_in_array_table(const string& token);//判断单词是否在符号表中，且是否为数组
    bool is_const(const string& token);//判断符号是不是常量，这个是涉及多层的
    void add_symbol(const string& token, const Symbol &symbol);//将一个构造好的符号
表添加入表中
    void add_var_symbol(const Word& word,int btype,int var_value,bool
is_const);//将变量/常量符号添加入符号表中
    void add_func_symbol(const Word& word,int func_type,int param_num, const
vector<int> &param_type);//将函数名添加入符号表
    void add_func_param_symbol(const Word& word,int btype,bool is_array);//添加函
数参数
    void add_array_symbol(const Word& word,int type,int dim, bool is_const);//添加
数组符号
    void set_var_value(const string& token,int var_value);
    void set_int_array_value(const string& token, const vector<int>
&array_value);
    void set_int_array_value(const string& token, int array_value);//添加一个数字
    void set_char_array_value(const string& token, const vector<char>
&array_value);
    void set_char_array_value(const string& token, char array_value);//添加一个字符
    void set_array_dim(const string &token, int dim);//设置数组的维度
    //设置函数是否return
    void set_is_return(const string& token, bool is_return);
    void set_is_return(Symbol* func_symbol,bool is_return);
    void set_func_return(const string& token,int func_value);
    void set_func_return(Symbol* func_symbol,int func_value);
};
```

SymbolHandle 符号表处理类

```
class SymbolHandle {
public:
    static SymbolHandle* getSymbolHandleInstance();
    void handleSymbol();
    void printSymbol();//输出全部的符号表
private:
    static SymbolHandle* instance;
    TreeNode*paserTree = nullptr;//语法树根节点
    int symbol_table_id = 0;//全局符号表序号，每创建一个符号表就+1
    /*stack<SymbolTable> symbol_table_stack;//符号表栈，采用栈进行构造
```

```
    map<int, SymbolTable> symbol_table_map;//构造好符号表，弹出栈后加入这个map，用于保存。*/
    shared_ptr<SymbolTable> GlobalSymbolTable = nullptr;
    shared_ptr<SymbolTable> nowSymbolTable = nullptr;
    bool is_main = false;//当前是否在main函数，用于处理main函数没有返回值的问题
    bool is_main_return = false;// 当前在main函数，有没有返回值
    int is_in_for = 0;//0表示不在循环，其余表示在第几层循环
    string func_token;//标记当前在哪个函数域内
    SymbolHandle();
    /**
     * 创建一个符号表
     * @return
     */
    shared_ptr<SymbolTable> createSymbolTable();

    /**
     * 主编译单元，从这里进去获取整个符号表
     * @param root
     */
    void SymbolCompUnit(TreeNode*root);
    void SymbolDecl(TreeNode* root);//当遍历树的节点是Decl时进入
    void SymbolConstDecl(TreeNode* root);//当树节点是ConstDecl时进入
    /**
     * 当树节点是Btype时进入
     * @param root
     * @return 类型，0是int 1是char
     */
    int SymbolBtype(const TreeNode* root);

    /**
     * 当树节点是ConstDef时进入
     * @param root
     * @param is_const 是否是常量
     * @param btype 类型
     */
    void SymbolConstDef(TreeNode* root, bool is_const, int btype);

    /**
     * 处理ConstInitVal
     * token用于赋值时找到对应的符号
     * @param root
     * @param token
     */
    void SymbolConstInitVal(TreeNode* root, const string& token);
    void SymbolVarDecl(TreeNode* root); // 变量声明
    void SymbolVarDef(TreeNode* root, bool is_const, int btype); // 变量定义
    void SymbolInitVal(TreeNode* root, const string& token); // 变量初值
    void SymbolFuncDef(TreeNode* root); // 函数定义
    void SymbolMainFuncDef(TreeNode* root); // 主函数定义
    int SymbolFuncType(TreeNode* root); // 函数类型
    /**
     * 函数形参表
     * @param root
     * @param params_type 参数类型的vector，用于存储在函数符号中
     */
    void SymbolFuncFParams(TreeNode* root, vector<int>& params_type);
```

```
    /**
     * 函数形参
     * @param root
     * @return 形参的类型，0是变量，1是int数组，2是char数组
     */
    int SymbolFuncFParam(TreeNode* root);

    /**
     * 语句块处理，在这里负责根据“是否由函数产生”建立符号表
     * @param root
     * @param is_func
     */
    void SymbolBlock(TreeNode* root, bool is_func); // 语句块
    /**
     * Block已建立符号表，所以在这里不需要再考虑是不是函数产生了
     * @param root
     */
    void SymbolBlockItem(TreeNode* root); // 语句块项，如果不是func要创建符号表，否则符号表在函数部分创建完成，不需要再创建
    void SymbolStmt(TreeNode* root); // 语句
    void SymbolForStmt(TreeNode* root); // for语句
    /**
     * 表达式
     * @param root
     * @param token token不为空，那么就把token对应的符号的值改变；如果token为空则不处理
     */
    int SymbolExp(TreeNode* root, const string& token); //
    void SymbolCond(TreeNode* root); // 条件表达式
    /**
     * 左值表达式
     * @param root
     * @param type 这个左值表达式的Ident是不是常量，0 不是常量也有定义，1 是常量，2 未定义
     * @return 第一个是LVal表达式算出来的值，第二个是数组标号Exp;如果Exp=-1说明不是数组
     */
    pair<int,int> SymbolLVal(TreeNode* root, int* type);
    int SymbolPrimaryExp(TreeNode* root, const string& token); // 基本表达式
    int SymbolNumber(TreeNode* root); // 数值
    char SymbolCharacter(TreeNode* root); // 字符
    int SymbolUnaryExp(TreeNode* root, const string& token); // 一元表达式
    char SymbolUnaryOp(TreeNode* root); // 单目运算符
    vector<int> SymbolFuncRParams(TreeNode* root, const string& func_token); // 函数实参表
    int SymbolMulExp(TreeNode* root, const string& token); // 乘除模表达式
    int SymbolAddExp(TreeNode* root, const string& token); // 加减表达式
    int SymbolRelExp(TreeNode* root); // 关系表达式
    int SymbolEqExp(TreeNode* root); // 相等性表达式
    int SymbolLAndExp(TreeNode* root); // 逻辑与表达式
    int SymbolLOrExp(TreeNode* root); // 逻辑或表达式
    /**
     * 常量表达式
     * @param root
     * @param token
     * @return 返回表达式的值
     */
    int SymbolConstExp(TreeNode* root, const string& token);
```

```
};
```

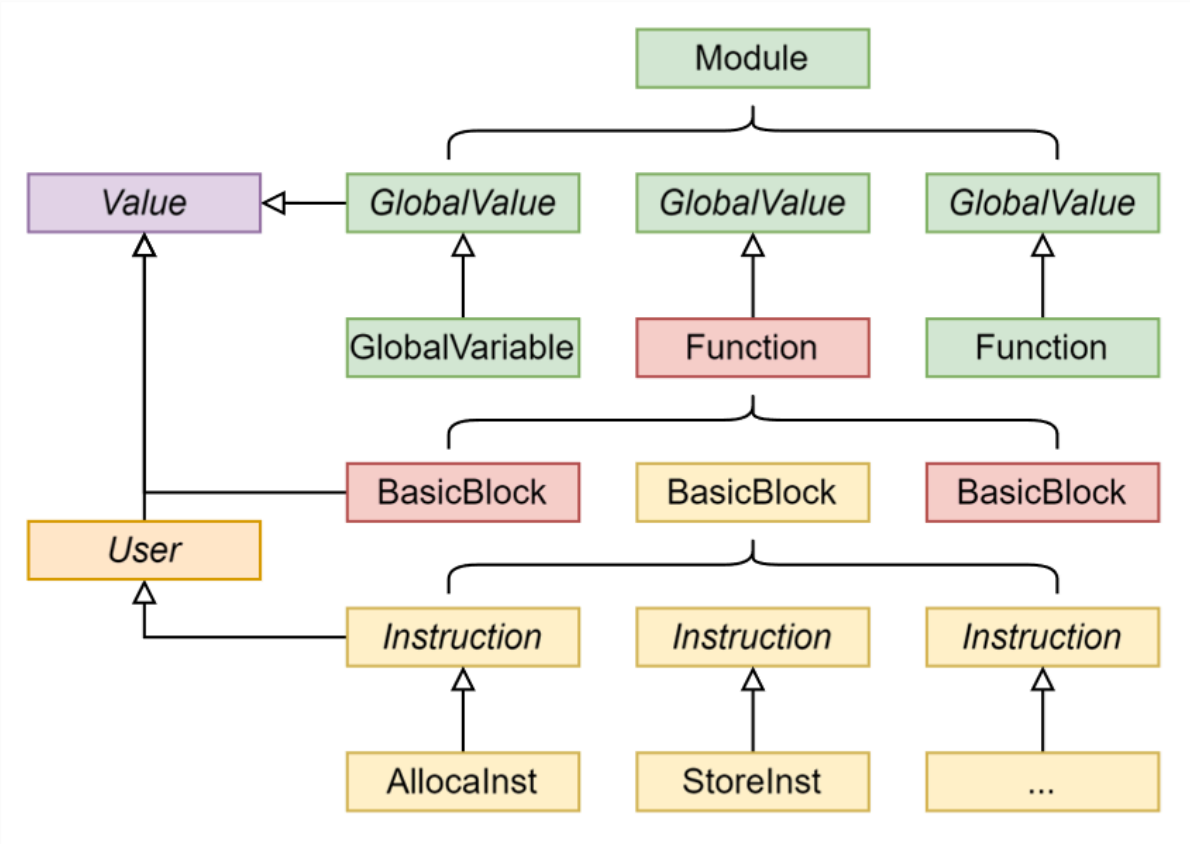
6. 代码生成设计

6.1、LLVM中间代码生成

中间代码生成主要分为**两个重要部分**，第一个是重新**遍历语法树**，结合符号表从而将各个符号**转化为** `llvm_ir` 的格式（这里我采用Value统一存储），另一个重要任务是利用 `llvm` 中**一切皆value的思想**重新建立起一棵由**不同种类的value组成的新Value树**，最后生成中间代码其实就是对新语法树的一个**后序遍历**过程。

6.1.1、构建LLVM IR的基础类

根据ppt上所讲内容，LLVM的架构图如下：



由于本次实验只涉及一个程序的编译，所以只有一个Module，自然将其选择为Value树根，在Module类中存储所有使用到的Value类（按照顺序存），这样就组成了Value树。

6.1.2、递归下降语法制导翻译

由于已经处理了语义分析，现在的程序理论上没有错误，所以配合符号表，再次进行递归下降分析，生成对应的LLVM IR

关于EXP部分，EXP应当尽量返回一个结果。所以在这里我直接获取EXP的叶子节点，叶子节点记录了各个符号和常数。对于常数和常量（查符号表），将它们的值处理成一个常数；保留变量和调用的函数，再生成相关的LLVM语句。

对于常量的处理，通过文法的定义我们可以知道，常量求值只涉及常数和其它常量。所以我们可以通过符号表定义的const，进行计算，合并常值。

对于遍历过程，结合语法树和符号表树进行。其中，根据运行的顺序，符号表的遍历相当于深度优先搜索。在符号表中可以设置一个标记，遍历过就进行标记，每次去最左侧没有遍历过的符号表。

进入符号表有两个时机：进入函数，进入新的基本块；由于函数创建时，也会有对应的新基本块，这时不能再进一次，所以如果基本块的父节点是FuncDef或MainFuncDef，就不能进入下一层基本块了。

关于强制转换

强制转换主要出现在运算处。利用继承属性，对每个语法树节点做了标记，如果是char就是char类型，如果是int就是int类型。如果父节点和某些子节点类型不同，就需要强制转换指令。

1. 比如函数定义的返回值类型为i32,实际返回为i8,需要做类型转换
2. getch(),putch()全部都是i32类型，需要进行转换
3. printf中包含格式符，也需要相应的转换
4. 左值表达式LVal部分，以及二元运算部分。其中二元运算要统一转成int类型

关于指针的问题

在本次设计中，当函数传参是数组时，实际上应当记录为指针类型。而参数属于局部变量，所以alloca的是一个指针的指针，这也就导致取值时应当先load，再getelementptr。而不应该直接对二重指针取地址，否则取出来的还是个地址。

6.1.3、输出LLVM代码

其实遍历语法树和符号表，作用是为了将代码转换成LLVM的类的组合。对于每个LLVM的类，都需要有对应的"生成LLVM代码"的方法，这样遍历后才能输出真正的LLVM IR语句。我把这个方法定义为了**toLLVM()**;每种实际的类输出的方式都有所不同，需要结合具体类的情况判断。

6.2、各种基本类类的设计代码

Value类为最主要的基类，代码如下：

```
//防止循环依赖
class Use;
class User;
//在LLVM中，一切皆Value，这个类是所有类的基类
class Value {
public:
    virtual ~Value() = default;

    IRType* value_type = nullptr;//value的类型
    string value_name;//value的名字
    vector<Use*> useChain;//Value的定义使用链
    Value() = default;
    Value(IRType* type, const string &name);
    Value(IRType* type, const string &name, const vector<Use*> &useChain);

    /**
     * 下面几个函数是get和set
     */
    [[nodiscard]] IRType* getValueType() const{
        return value_type;
    }
    void setValueType(IRType* value_type) {
        this->value_type = value_type;
    }
    [[nodiscard]] string getValueName() const {
        return value_name;
```

```
    }
    void setValueName(const string &value_name) {
        this->value_name = value_name;
    }
    [[nodiscard]] vector<Use*>& getUseChain(){
        return useChain;
    }
    void setUseChain(const vector<Use*> &use_chain) {
        useChain = use_chain;
    }
    //添加一个使用者，User使用了这个Value
    void addUser(User* user);
    //移除一个使用者
    User* removeUser(User* user);
    //使用者原本使用的Value替换为这个Value
    void replaceUser(Value* oldValue,Value* newValue,User* user);
    //获取使用这个Value的全部User
    [[nodiscard]] vector<User*> getAllUser() const;
    //将所有的使用本Value的使用者全部替换为使用新的Value
    void replaceAllUser(Value* value);
    //生成汇编的函数，具体到不同的类再具体重写
    void virtual generateMIPS(){}
    //保留toLLVM的方法，生成中间代码字符串
    string virtual toLLVM() { return ""; }
};
```

User类则是使用了其它操作数的类，这个类在优化部分非常重要，User类初步设计如下：

```
class Value;
class Use;
//根据教程“LLVM的数据结构"，可知User类继承Value类并且可以使用Value
//User使用Value的关系由Use记录
class User :public Value{
public:
    vector<Value*> opValueChain;//这个User使用的操作数
    [[nodiscard]] vector<Value *>& op_value_chain(){
        return opValueChain;
    }
    User() = default;
    User(IRType* type, const string &user_name);
    void addOpValue(Value* op_value);//添加一个操作数Value
    void replaceValue(Value* old_value,Value* new_value);//替换一个操作数
    void dropOpValue();//删除所有该User中的该操作数
};
```

其中，Use类是记录了Value和User的对应关系，便于形成定义使用链

Instruction类，指令类，负责整合所有的指令，继承于User类

```
class BasicBlock;
class Instruction :public User{
private:
    BasicBlock* blockParent = nullptr;//标记这条指令所在的基本块
protected:
    string instructionType;//指令的名字，比如是add指令还是sub指令,trunk等
```

```
public:
    string& getInstructionType(){
        return instructionType;
    }
    void setBlockParent(BasicBlock* currentBlock) {
        blockParent = currentBlock;
    }
    [[nodiscard]] BasicBlock* getBlockParent() const {
        return blockParent;
    }
    Instruction() = default;
    Instruction(IRType* ir_type,const string& name, const string
&instructionType);
    void generateMIPS() override;
};
```

BasicBlock，基本块类，里面包括了这个基本块里有什么指令，以及它属于哪个函数。在本实验中，基本块只能出现在函数中

需要考虑的是，循环块可能有多层，所以这里还需要绑定循环

```
class Function;
class Instruction;
class Loop;
//基本块类，一个函数有多个基本块，至少有一个
class BasicBlock :public Value{
    //一个基本块内有多个指令,且不能包括跳转和返回指令。
    //一个基本块内的指令，要么都执行，要么都不执行
    //先不考虑代码优化
private:
    vector<Instruction*> instructions;//多条指令按照先后顺序存入数组
    Function* functionParent = nullptr;//这个基本块属于哪个函数
    bool is_exist = true;//基本块是否还存在
    Loop* loopParent = nullptr;//这个基本块是不是某个循环下的
public:
    BasicBlock() = default;
    explicit BasicBlock(const string& name);
    //添加一条指令，如果index在当前vector范围内则插入到这个地方；否则添加在最后
    void addInstruction(Instruction* instruction,int index = -1);
    //判断指令集是否为空
    [[nodiscard]] bool isEmpty() const {
        return instructions.empty();
    }
    //获取最后一条指令
    [[nodiscard]] Instruction* getLastInstruction() const {
        return instructions.back();
    }
    //设置指令集
    void setInstructions(const vector<Instruction*>& instructions);
    //设置exist状态
    void setExist(bool exist);
    //获取exist状态
    [[nodiscard]] bool isExist() const;
    //设置属于哪个循环块
```

```
    void setLoopParent(Loop* loop);
    //获取当前属于哪个循环块
    [[nodiscard]] Loop* getLoopParent() const;
    //当前在循环第几层深度？
    [[nodiscard]] int getLoopDepth() const;
    //设置当前属于哪个函数
    void setFunctionParent(Function* function);
    //获取当前块属于哪个函数
    [[nodiscard]] Function* getFunctionParent() const;
    //覆写，基本块应当如何输出
    string toLLVM() override;
    //生成汇编，之后再补
    void generateMIPS() override;
};
```

Function类，统筹了整个函数，包括内部含有的基本块和参数

```
class Param;
class Function : public GlobalValue{
private:
    IRType* returnType = nullptr;//函数的返回类型
    vector<BasicBlock*> basicBlocks;//函数内的基本块
    vector<Param*> params;//函数的参数
    //unordered_set<Function*> callFunctions;//这个函数调用了哪些函数
    unordered_map<Value*, Register*> valueRegisterMap;//在函数内部的寄存器和Value的映
射表
public:
    Function() = default;
    Function(const string& name, IRType* returnType);
    [[nodiscard]] IRType* getReturnType() const {
        return returnType;
    }
    vector<BasicBlock *>& getBasicBlocks(){
        return basicBlocks;
    }
    vector<Param *>& getParams(){
        return params;
    }
    [[nodiscard]] unordered_map<Value *, Register *>& getValueRegisterMap() {
        return valueRegisterMap;
    }
    //添加函数使用的参数
    void addParam(Param *param);
    //添加函数包括的基本块，index表示添加基本块到哪个位置
    void addBasicBlock(BasicBlock *basicBlock,int index = -1);
    string toLLVM() override;
    void generateMIPS() override;
};
```

全局变量类，包含了全局的变量。同时根据LLVM的特性，分为了常量和变量，这两个在代码中显示的效果不一样

```
//全局变量类，继承自User
class GlobalVariable :public GlobalValue{
```

```
private:
    //InitVar作为初始化全局变量
    InitVar* initVar{};
    bool is_const_array = false;//是不是常数数组，对于常量，可以直接合并，不保留
public:
    [[nodiscard]] InitVar* getInitVar() const {
        return initVar;
    }
    GlobalVariable() = default;
    GlobalVariable(IRType* ir_type,const string& name, InitVar* init_var, bool
is_const_array = false);
    void SetInitVar(InitVar* init_var) {
        initVar = init_var;
    }
    string toLLVM() override;
    //生成mips用
    void generateMIPS() override;
};
```

到这里，整个架构的基本类已经设计完毕，其余的类是根据具体情况继承这些基本类得到的。

下面是生成LLVM的主类，LLVMGenerate主要包含了遍历时的具体实现。

```
/*
* 生成LLVM IR的主类
*/
class LLVMGenerate;
class GenerateIR {
private:
    TreeNode* root{};//语法树根节点
    static LLVMGenerate* generate;
    static Module* module;
    static GenerateIR* instance;
    GenerateIR();
public:
    static GenerateIR* getInstance();
    Module* getModule();
    //生成LLVM IR的过程
    void generateLLVMIR();
    //输出LLVM代码
    void printLLVMIR();
    //遍历语法树的过程
    void traverseTreeIR(TreeNode* root);
};
```

7. 后端代码生成设计

7.1、Mips生成

后端代码采用mips

其实生成mips的思想和生成LLVM IR的思想一样，都是根据语法树组装好的IR结构，进行翻译。不过因为两者在代码层面上不同，所以这里要做的是**把每个IR指令翻译成Mips。**这里我定义了方法**toMips()**

在优化前，采用栈式存储的方式，对应LLVM的存取部分。记录每个Value和对应的offset，每次用到这个Value的时候，就去map找offset，再根据栈帧和偏移取出对应的值。

基本上和LLVM的代码相对应，定义变量时，则找栈帧的位置，记录栈帧-4的值并压入栈中。随后记录到<Value*,offset>的map中；当需要存储值时，先从map找到偏移offset，再利用栈帧定位到这个位置，取出后就是变量定义时的地址，再将值存入这个地址

关于函数和调用，每次调用函数需要将栈帧**$sp** 和 **$ra**存入到对应的栈中；进入函数后，栈指针需要设置为新的栈底。关于函数调用，按照mips的规定，将前四个参数存入a0至a4寄存器中，返回值存入到v0中。

7.2、类的设计

我们知道，mips分为两段：data段和text段，这两个段有相似之处，但是一个负责全局的变量，另一个负责所有的指令。为了结构化，我分别设置了两个段类，以及两者都继承的一个父类

```
/**
* 数据段和代码段统一使用
*/
class Segment {
public:
    vector<MipsStructure*> MipsSegments;
    virtual ~Segment() = default;

    Segment() = default;
    void addMipsSegment(MipsStructure* newMipsSegment);
    virtual string toMips();
};
```

```
class TextSegment : public Segment{
public:
    string toMips() override;
};
```

```
class DataSegment : public Segment{
public:
    string toMips() override;
};
```

其实可以发现，两个类的区别在于生成mips的方式不一样，主要是格式问题，data段由于全是变量定义，所以紧靠左侧，有换行；而text段有换行，还有\t的因素。

承载了所有段的类，是MipsContent

```
class MipsContent {
private:
    static TextSegment* textSegment;//text段
    static DataSegment* dataSegment;//data段
public:
    MipsContent();
    static void addTextSegment(MipsStructure* mips_structure);
    static void addDataSegment(MipsStructure* mips_structure);
    string toMips();
};
```

每个指令生成后，都要将其加入到对应的段中。

MipsCell类，属于编译单元，记录了栈帧如何移动，如何记录<Value*,offset>的map问题

```
class MipsCell {
private:
    //当前栈指针偏移量
    static int stackOffset;
    //记录Value和栈指针偏移量的映射
    static std::unordered_map<Value*, int> stackOffsetMap;
public:
    //移动当前栈指针偏移量
    static void moveNowOffset(int offset);
    //每次进入一个函数，都需要重置相关属性
    static void resetFunction(Function* function);
    static int getStackOffset();
    static void addValueOffset(Value* value, int offset);
    static int getValueOffset(Value* value);
    //获取已分配的寄存器
    static vector<Register*> getRegisterDistribute();
};
```

将各种Mips的指令组好后，就可以在LLVM IR相关类中补充生成mips的指令。mips生成完毕后，也会被记录在Module中，通过如下生成mips的主类进行整合生成和输出

```
/**
* 生成MIPS代码
*/
class MipsGenerate {
private:
    Module*module = nullptr;
    MipsContent*mips_content = nullptr;
    static MipsGenerate* instance;
public:
    MipsGenerate() = default;
    explicit MipsGenerate(Module* module);
    static MipsGenerate* getInstance(Module* module);
    //生成Mips代码
    void generateMips();
    //输出Mips
    void printMips();
};
```

在这一部分，最重要的是寄存器类相关的设计

首先，寄存器类为了保证唯一性，以及容易查询性，我仿照Java的枚举类。

使用枚举类，可以便于将整数id和对应的寄存器联系起来，这样涉及到取连续寄存器或者寄存器号运算时，方便得到寄存器。

写了如下的Register类：

```
static const string regNames[] = {
    "$zero",
    "$at",
    "$v0", "$v1",
    "$a0", "$a1", "$a2", "$a3",
    "$t0", "$t1", "$t2", "$t3", "$t4", "$t5", "$t6", "$t7",
    "$s0", "$s1", "$s2", "$s3", "$s4", "$s5", "$s6", "$s7",
    "$t8", "$t9",
    "$k0", "$k1",
    "$gp",
    "$sp",
    "$fp",
    "$ra"
};
/**
* 寄存器枚举类
*/
enum class RegisterName {
    $zero,   // 0号寄存器
    $at,     // 汇编保留寄存器
    $v0, $v1, // 函数返回值寄存器
    $a0, $a1, $a2, $a3, // 函数参数寄存器
    $t0, $t1, $t2, $t3, $t4, $t5, $t6, $t7, // 临时寄存器
    $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7, // 保存寄存器
    $t8, $t9, // 额外的临时寄存器
    $k0, $k1, // 保留给操作系统使用
    $gp,      // 全局指针
    $sp,      // 堆栈指针
    $fp,      // 帧指针
    $ra       // 返回地址寄存器
};

class Register {
private:
    RegisterName reg = RegisterName::$zero;
    //采用类似于单例模式，保证每个寄存器类的指针也一样
    static unordered_map<RegisterName,Register*> registers;
    Register() = default;
    explicit Register(RegisterName reg);
public:
    [[nodiscard]] RegisterName getRegisterName() const;
    static Register* getRegister(RegisterName reg);
    string toMips();
    static RegisterName regTransform(int index) {
        return static_cast<RegisterName>(index);
    }
};
```

每个寄存器元素都是单例模式。

对应包括寄存器相关的工具类，用来处理栈式寄存器的分配和移动

首先是RegisterController，用来记录Value和寄存器之间的关系。

```
class RegisterController {
private:
    static unordered_map<Value*,Register*> valueRegisterMap;

public:
    [[nodiscard]] static unordered_map<Value *, Register *> getValueRegisterMap()
{
        return valueRegisterMap;
    }

    static void setValueRegisterMap(const unordered_map<Value *, Register *>
&value_register_map) {
        valueRegisterMap = value_register_map;
    }
    static void allocateRegister(Value* value, Register *reg);//分配寄存器
    static Register *getRegister(Value *value);
};
```

另一个是RegisterTool类，用来处理寄存器读取，移动相关

```
/**
* 工具类，用于处理寄存器的分配
*/
class RegisterTool {
public:
    /**
    * 将一个Value移动-4的偏移量，返回移动后的总偏移量
    * 并且将value和偏移添加到map中
    * 在本阶段，采用栈式存储，所有变量都存在栈里，根据value和offset的映射map来取值
    */
    static int moveValue(Value* value);

    /**
    * 申请分配的寄存器
    * 1. 首先将栈指针向下移动4个字节，用来分配寄存器
    * 2. 然后将value的偏移量和寄存器的映射添加到寄存器控制器中
    * @param value
    * @param reg
    */
    static void allocaRegister(Value* value, Register* reg);

    /**
    * 申请重新分配寄存器
    * @param value
    * @param reg
    */
    static void reAllocaRegister(Value* value, Register* reg);

    /**
```

```
    * 在内存中申请分配寄存器
    * @param value
    * @param reg
    */
    static void memoryAllocaRegister(Value* value, Register* reg);

    /**
    * 加载寄存器的值
    * @param op
    * @param reg value对应的寄存器
    * @param reg_instead 没有reg，要使用替代的寄存器
    * @return
    */
    static Register* loadRegister(Value* op, Register* reg, Register*
reg_instead);

    /**
    * 加载变量的值
    * @param op
    * @param reg
    * @param reg_instead
    * @return
    */
    static Register* loadVarValue(Value* op,Register* reg, Register*
reg_instead);

    /**
    * 加载指针的值
    * @param op
    * @param reg
    * @param reg_instead
    * @return
    */
    static Register* loadPointerValue(Value* op,Register* reg, Register*
reg_instead);

    /**
    * 加载内存偏移量，其中irType表示取的哪种类型，因为char数组是byte
    * @param op
    * @param reg
    * @param reg_instead
    * @param pointerReg
    * @param offsetReg
    * @return
    */
    static Register* loadMemoryOffset(Value* op,Register* reg, Register*
reg_instead,Register* pointerReg, Register* offsetReg, IRType* irType);

    /**
    * 在函数参数中分配寄存器
    * @param param
    * @param paramReg
    * @param currentOffset
    * @param allocatedRegs
    * @return
    */
```

```
    static Register* allocParamReg(Value* param, Register* paramReg,int
currentOffset, vector<Register*> allocatedRegs);

    /**
    * 在函数参数中分配内存
    * @param param
    * @param paramReg
    * @param currentOffset
    * @param allocatedRegs
    * @param param_num
    * @return
    */
    static Register* allocParamMem(Value* param, Register* paramReg,int
currentOffset, vector<Register*> allocatedRegs, int param_num);
};
```