《编译技术》课程设计文档

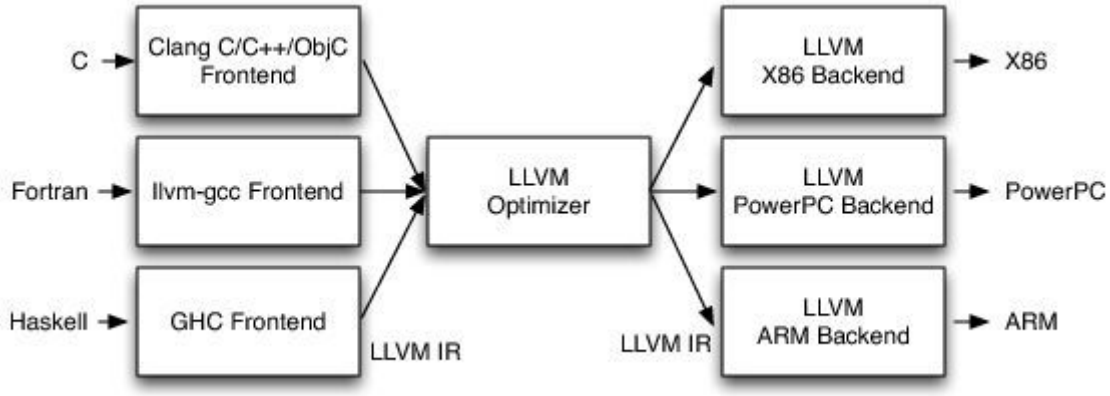
姓名：黄雨石

学号：20376156

参考编译器介绍

虽然最后由于时间原因没有实现`LLVM IR`但是在前期过程中一直都有阅读`LLVM`的相关资料以及源码，部分设计思想，比如软件包之间解耦的设计也来自于`LLVM`，参考源码来自于https://github.com/llvm/llvm-project，同时还参考了其它许多`StackOverFlow`上的帖子，不在此一一列出了，当然由于项目非常大以及本人水平有限，也只看了些皮毛

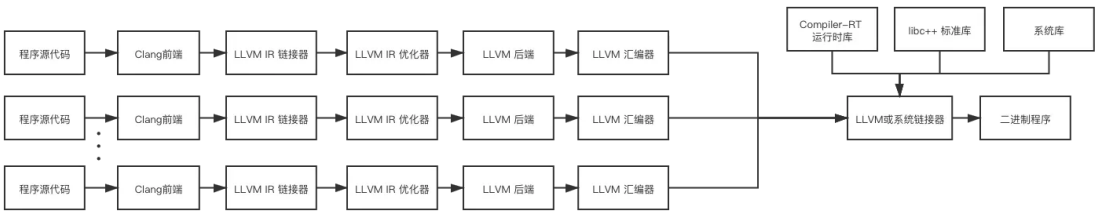
- `LLVM`介绍：传统的静态编译器分为三个阶段：前端、中端（优化）、后端。而`GCC`编译器在设计的时候没有做好层次划分，导致很多数据在前端和后端耦合在了一起，所以`GCC`支持一种新的编程语言或新的目标架构特别困难。有了`GCC`的前车之鉴，`LLVM`进行了如下图所示的三阶段设计：



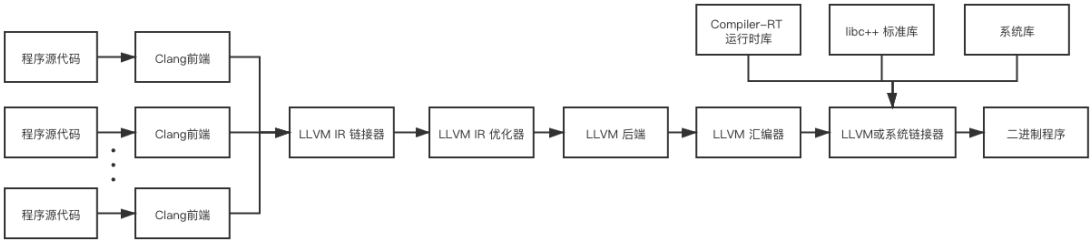
  在不同的语义环境下，`LLVM`具有以下几种不同的含义：

  - `LLVM`**基础架构**：即一个完整编译器项目的集合，包括但不限于前端、后端、优化器、汇编器、链接器、`libc++`标准库、`Compiler-RT`和`JIT`引擎
  - **基于**`LLVM`**构建的编译器**：部分或完全使用`LLVM`构建的编译器
  - `LLVM`**库**：LLVM基础架构可重用代码部分
  - `LLVM`**核心**：在`LLVM IR`上进行的优化和后端算法
  - `LLVM IR`：`LLVM`中间表示
- `LLVM`**基础架构的组成部分**：
  - **前端**：将程序源代码转换为`LLVM IR`的编译器步骤，包括词法分析器、语法分析器、语义分析器、`LLVM IR`生成器。`Clang`执行了所有与前端相关的步骤，并提供了一个插件接口和一个单独的静态分析工具来进行更深入的分析
  - **中间表示**：`LLVM IR`可以以可读文本代码和二进制代码两种形式呈现。`LLVM`库中提供了对IR进行构造、组装和拆卸的接口。`LLVM`优化器也在`IR`上进行操作，并在`IR`上进行了大部分优化。
  - **后端**：负责汇编码或机器码生成的步骤，将`LLVM IR`转换为特定机器架构的汇编代码或而二进制代码，包括寄存器分配、循环转换、窥视孔优化器、特定机器架构的优化和转换等步骤

  下面这张图展示了使用`LLVM`基础架构时各个组件之间的关系

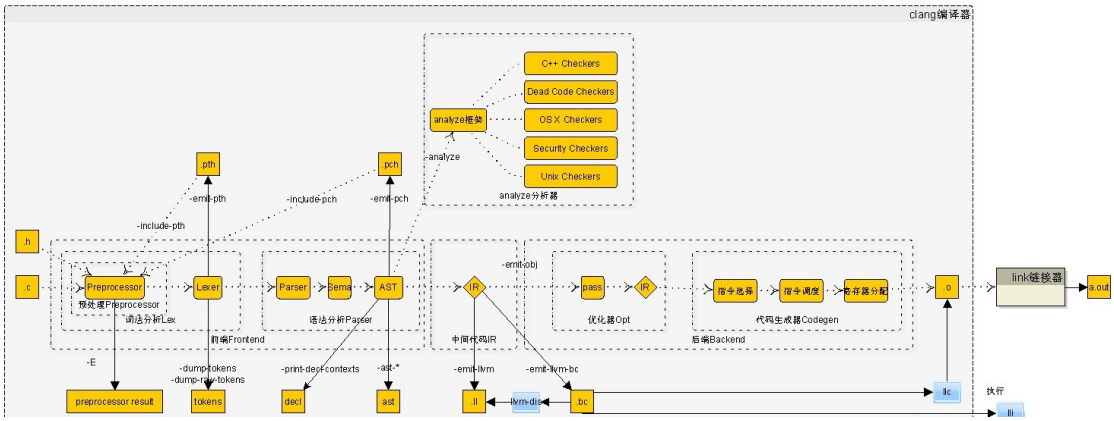


  除此之外，各个组件之间的协作关系也可以以下面这种方式组织



> 两种方式的主要区别是程序源代码内部的链接是由LLVM或系统链接器完成的还是由LLVM IR链接器完成的，前者是默认方式，后者一般在开启链接优化（Link-Time Optimization）时采用

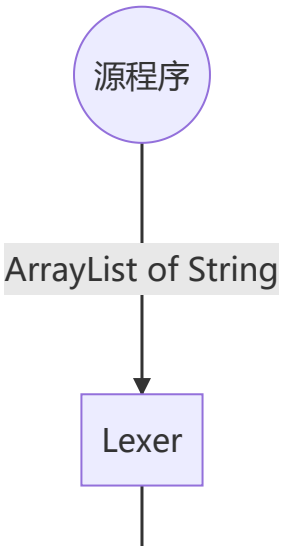
- **接口设计**：并未完整的总览过`LLVM`的前后端所有接口，~~目是在太复杂~~因此仅用一张简略的流程图表示相关模块接口间的关系以及生成物与传递物：

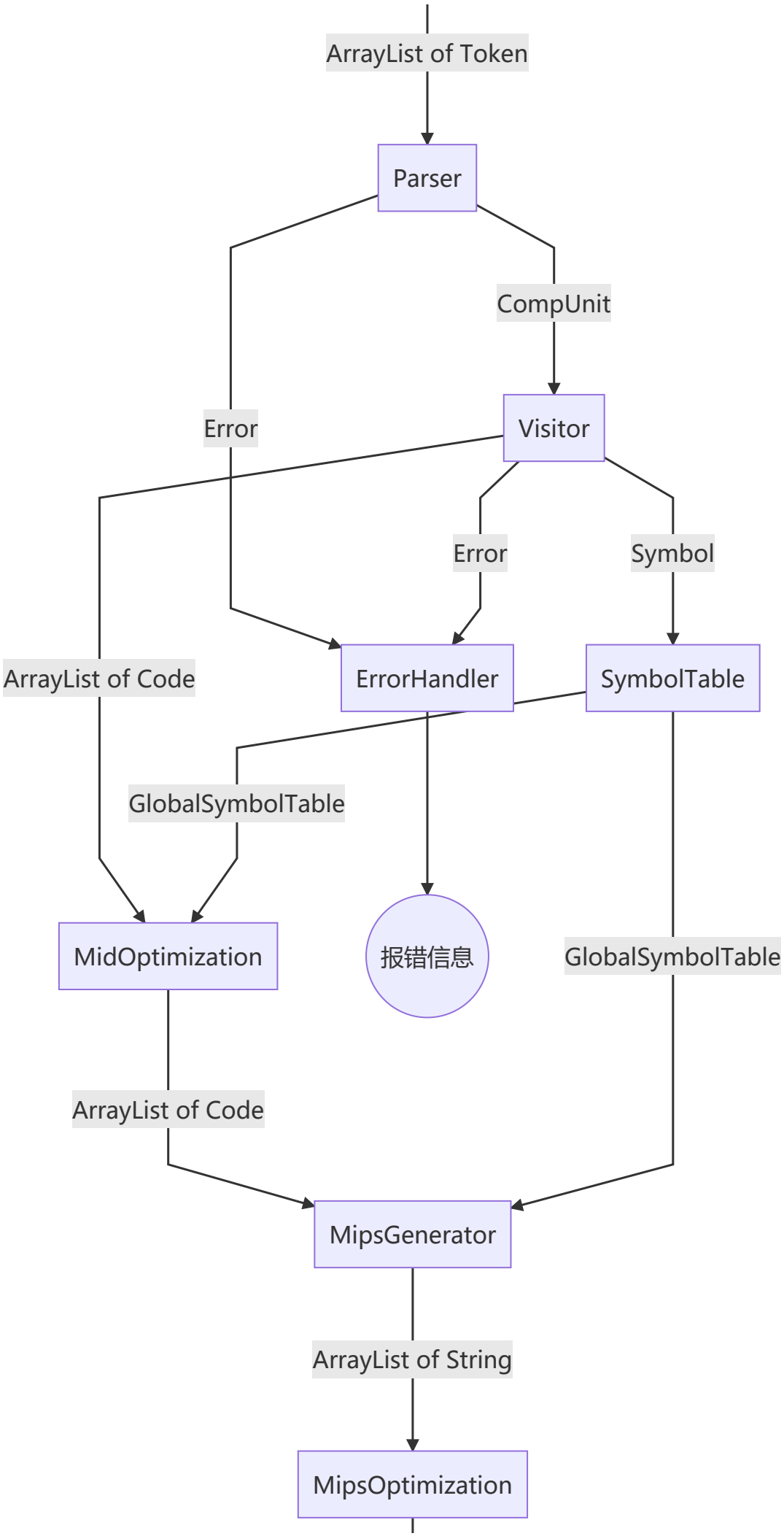


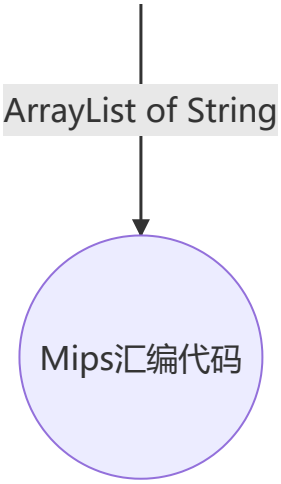
- **文件组织**：由于文件是在太多，只简单介绍一下`LLVM`的后端源码结构，关于`Clang`部分就不在此介绍~~其实这部分我搞的有点混乱，且看的时间太久已经忘了~~，参考文档https://llvm.org/docs/GettingStarted.html#getting-started：
  - `llvm/examples`：这个目录主要是一些简单例子，演示如何使用`LLVM IR`和`JIT`。还有建立一个简单的编译器的例子的代码。
  - `llvm/include`：这个目录主要包含`LLVM library`的公共头文件。
  - `lvm/lib`：这个目录包含了大部分的`LLVM`的源码。在`LLVM`中大部分的源码都是以库的形式存在的，这样不同的工具之前就很容易共用代码。
  - `llvm/projects`：这个目录包含着一些依赖`LLVM`的工程，这些工程严格来说又不算`LLVM`一部分。
  - `llvm/runtimes`：这个目录包含了一些库，这些库会编译成`LLVM`的`bitcode`，然后当`clang linking`程序的时候使用。
  - `llvm/test`：这个目录是`LLVM`的测试套件，包含了很多测试用例，这些测试用例是测试`LLVM`的所有基本功能的。
  - `llvm/tools`：这个目录理是各个工具的源码，这些工具都是建立在刚才上面的那些库的基础之上的。也是主要的用户接口。
  - `lvm/utils`：这个目录包含了一些和`LLVM`源码一起工作的应用。有些应用在`LLVM`的编译过程中是不可或缺的。

编译器总体架构

- **架构**：总体按照课本的5个基本阶段，7个逻辑部分组成，大框架为`Lexer`、`Syntax`、`Middle`、`Backend`、`Error`、`Symbol`6个软件包，其中的优化部分内嵌到`Middle`、`Backend`中，主体部分结构如下：




ArrayList of Token
Parser
CompUnit
Error
Visitor
Error
Symbol
ArrayList of Code
ErrorHandler
SymbolTable
GlobalSymbolTable
MidOptimization
报错信息
GlobalSymbolTable
ArrayList of Code
MipsGenerator
ArrayList of String
MipsOptimization



值得一提的是我在语义分析的时候是同时建立符号表、生成中间代码、进行错误处理的，由于是树形符号表因此传递的是最顶层的全局符号表

- **接口设计**：各个软件包之间**只传递上述图中所产生的信息**，较好的实现了各个软件包之间的解耦；而每个软件包内部则高度耦合，传递的接口均设计为类中的`get*`方法或者通过全局静态变量来传递信息，综上几个大的软件包之间基本上就只有一个入口和出口，非常便利与简单
- **文件组织**：

```
src
|____Compiler
|____Lexer
|       |____Lexer
|       |____Token
|       |____Type
|____Syntax
|       |____CompUnit
|       |____Parser
|       |____Decl
|       |     |____Decl
|       |     |____Def
|       |     |____InitVal
|       |           |____initArr
|       |           |____initExp
|       |           |____initVal
|       |____Func
|       |       |____Block
|       |       |____FuncDef
|       |       |____FuncFParam
|       |       |____FuncFParams
|       |       |____MainFuncDef
|       |____Stmt
|       |       |____BlockStmt
|       |       |____IfStmt
|       |       |____Stmt
|       |       |____WhileStmt
|       |       |____Simple
|       |             |____AssignStmt
|       |               |____ExpStmt
|       |               |____InputStmt
|       |               |____LoopStmt
|       |               |____OutputStmt
|       |               |____ReturnStmt
```

```
|       |               |____Simple
|       |               |____SimpleStmt
|       |_____Exp
|       |       |____Multi
|       |       |     |____AddExp
|       |       |     |____Cond
|       |       |     |____ConstExp
|       |       |     |____EqExp
|       |       |     |____Exp
|       |       |     |____LandExp
|       |       |     |____LorExp
|       |       |     |____MultiExp
|       |       |     |____RelExp
|       |       |____Unary
|       |             |____FuncRParams
|       |             |____LVal
|       |             |____Number
|       |             |____PrimaryExp
|       |             |____UnaryExp
|       |____Util
|           |____Index
|____Symbol
|       |____Func
|       |____FuncFormParam
|       |____Symbol
|       |____SymbolTable
|       |____Tmp
|       |____Val
|____Error
|       |____Error
|       |____ErrorTable
|____Middle
|       |____MidCodeList
|       |____Visitor
|       |____Util
|       |     |____Calc
|       |     |____Code
|       |____Optimization
|             |____Block
|             |____DataFlow
|             |____OptimizeLoop
|____BackEnd
    |____MipsGenerator
    |____Util
    |     |____Instruction
    |     |____RegAlloc
    |     |____ColorAlloc
    |____Optimization
          |____MulDiv
          |____PeepHole
          |____RedundantCall
```

- **工作流程**:
  - 第1遍词法分析，过程中处理词法错误；
  - 第2遍语法分析，构建 AST ，过程中处理语法错误；
  - 第3遍遍历 AST ，构建树形符号表，处理语义错误，生成中间代码。

- 第4遍至第$n-1$遍，不断扫描进行中间代码层面优化
- 第n遍，扫描中间代码翻译生成`mips`汇编代码

词法分析

编码前设计

首先定义`Token`类、`Type`枚举类、`Lexer`类，`Token`用于记录每个单词的类型、值、所在行数，`Type`则是由类别码构成的枚举类、`Lexer`提供词法分析以及存储结果的功能

在`Lexer`类中按字符读取从文件中读入的源码，由`if-else`分支语句进行类别判断，然后进入对应分支读取完整单词，得到一个`Token`实例，不断重复上述操作直到读到源码末尾

- 对于保留字采用`HashMap`进行查寻，保留字为键，类别码为值，当得到一个保留字单词就查询`HashMap`获得类别码
- 对于注释空白符跳过不识别，即直接读到注释之后的第一个字符
- 对于行号记录，读到`\n`字符行号就加一

本次不需要考虑错误处理的问题

编码后修改

与编码前基本一致，但是考虑到评测系统为`Linux`系统于是对于要跳过的空白符需要注意判断`\r`的情况

语法分析

编码前设计

封装`Parser`类，将`Lexer`类解析得到的所有`Token`以数组形式一次性传入`Parser`进行递归下降解析同时维护一个指向`Token`数组中元素的指针，然后使用自己封装的`peek()`、`retract(int step)`函数在递归下降过程中取出当前指针指向的元素或者将指针回溯。对于每一个非终结符建立相应的类作为递归下降树的非叶子结点，每一个叶子节点对应的为一个代表终结符的`Token`

其中我对于文法的具体分析处理主要如下：

- **表达式**：对于部分与`*Exp`相关的推导规则均存在左递归现象，需要改写文法如下：

```
改写前
-------
改写后

    MulExp -> UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
    ------------------------------------------------------
    MulExp -> UnaryExp { ('*' | '/' | '%') UnaryExp }

    AddExp -> MulExp | AddExp ('+' | '−') MulExp
    ---------------------------------------------
    AddExp -> MulExp { ('+' | '−') MulExp }

    RelExp -> AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
    ----------------------------------------------------------
    RelExp -> AddExp { ('<' | '>' | '<=' | '>=') AddExp }

    EqExp -> RelExp | EqExp ('==' | '!=') RelExp
    ---------------------------------------------
```

```
    EqExp -> RelExp { ('==' | '!=') RelExp }

    LAndExp -> EqExp | LAndExp '&&' EqExp
    ----------------------------------
    LAndExp -> EqExp { '&&' EqExp }

    LOrExp -> LAndExp | LOrExp '||' LAndExp
    ---------------------------------------
    LOrExp -> LAndExp { '||' LAndExp }
```

同时注意到以上列出的`*Exp`中均为`*Exp -> T {operator T}`（其中`T`可以认为是低一级的`*Exp`）的形式，于是可以建立一个基类`MultiExp`去统一这种形式，再使`*Exp`去继承它，`MultiExp`接口如下

```java
public class MultiExp<T> {
    private final String name;
    private final T first;
    private final ArrayList<Token> operators = new ArrayList<>();
    private final ArrayList<T> Ts = new ArrayList<>();
    // ...
}
```

于是可以通过上述方式将需要改写文法的`*Exp`放置于`Multi`软件包内，而其它诸如`UnaryExp`、`PrimaryExp`、`Number`、`LVal`、`FuncRParams`这种更低一级的表达式放置于`Unary`软件包中

- **语句**：由于推导规则中左侧为`Stmt`的规则非常复杂于是我们对于该规则右侧的每一个分支建立`*Stmt`类，即改写文法规则新增加`*Stmt`非终结符，具体如下：

```
    <AssignStmt> -> LVal '=' Exp
    <ExpStmt> -> Exp
    <LoopStmt> -> 'break' | 'continue'
    <ReturnStmt> -> 'return' [Exp]
    <InputStmt> -> <LVal> '=' 'getint' '(' ')'
    <OutputStmt> ->  'printf''('FormatString{','Exp}')'
    <SimpleStmt> -> [ <AssignStmt> | <ExpStmt> | <LoopStmt> |
<ReturnStmt> |
                <InputStmt> | <OutputStmt> ] ';'
    <IfStmt> -> 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
    <WhileStmt> -> 'while' '(' Cond ')' Stmt
    <BlockStmt> -> Block
    <Stmt> -> <SimpleStmt> | <IfStmt> | <WhileStmt> | <BlockStmt>
```

可以按照带分号与不带分号分为`SimpleStmt`以及`IfStmt`、`WhileStmt`、`BlockStmt`，由于`SimpleStmt`推导规则右侧涉及到较多的新建非终结符，因此可以建立`Simple`软件包统一存储。同时为了方便在`SimpleStmt`中保存，可以创建一个`Simple`接口，由`AssignStmt`、`ExpStmt`...共同实现

- **函数**：注意到其实`MainFuncDef`与`FuncDef`非常相似，于是可以将`MainFuncDef`视作`FuncDef`的特例去继承`FuncDef`类
- **变量声明**：由于不需要输出`Decl`可以直接将`ConstDecl`、`VarDecl`都视为`Decl`用一个布尔变量标识即可

对于上述分析已经我们已经可以消除所有的左递归，但是实际过程中考虑到有部分非终结符的`FIRST`是相同的，于是采取超前扫描的方法，同时定义了`pre`、`nxt`分别表示当前遍历的的数组中的`Token`的前一个和后一个以减少指针回溯，除了在处理`SimpleStmt`的过程中指针可能需要回溯一整个`LVal`的大小，其它时候至多回溯一步即可

对于`SimpleStmt`中的`AssignStmt`、`InputStmt`、`ExpStmt`可能它们的`FIRST`均为`LVal`这时侯需要先记录指针的值然后超前解析一个`LVal`判断其后的`Token`类型即可判断需要进入的分支，然后回溯到之前记录的位置进入对应分支的解析函数再次开始解析

编码后修改

- 编码前并未考虑如何输出的问题，编码时采取对于每个节点重写它的`toString`方法，最后直接调用顶层`CompUnit`的`toString`方法即可
- 由于文法中很多地方都会使用`'[' [*Exp] ']'`这种方式来表示数组变量的维度以及下标，因此，我单独建立了一个`Index`类存储这样的结构，同时完成该类的`toString`方法，具体接口如下：

```
public class Index {
    private Token lBTk;
    private Token rBTk;
    private Exp exp;
    // ...
}
```

- 编码前并未对于之后的错误处理预留接口，但是在递归下降的编码过程中可以通过判断当前`Token`非常自然的找到相关的错误并预留错误处理的空间

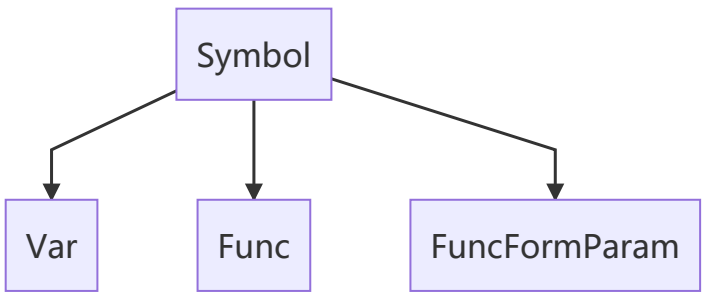
错误处理与符号表建立

编码前设计

- **符号表**：采用树形符号表，遵循一块一表，每次进入一个块时就建立一张符号表，表中存储新建的变量符号、函数符号、函数形参符号（当前为函数块），同时函数符号中存储该函数符号对应的函数块符号表以便查询函数中的变量以及形参

  其中符号一共分为三类，均实现`Symbol`接口：

  - `Var`：所有`int`和`const int`变量
  - `Func`：所有函数定义
  - `FuncFormParam`：函数形参



`Symbol`接口如下：

```
package Symbol;

public interface Symbol {
```

```
    String getName();
    int getBlockLevel();
    int getBlockNum();
    Integer setAndGetAddr(int addr); // 设置首地址并返回接下来的地址
    Integer getSize(); // 实际变量大小
    Integer getAddr(); // 首地址,在生成mips的时候计算，其实是相对与栈针或者$gp的偏移量
    boolean isConst();
    int getDim(); //维数
    SymbolTable getSymbolTable();//获取自己所在的符号表
    String getNickname();//获取自己的name加上在符号表中的层数，排名信息，方便定位变量
}
```

- **错误处理表**：建立一个`Error`类用于标识特定错误，将错误类型作为`Error`中的一个枚举类，再封装`ErrorTable`类，提供对于`Error`根据错误位置排序、打印、保存的功能
- **遍历语法树**：
  - **符号表建立**：首先维护一个当前符号表`curTable`，初始时即为全局符号表，之后对已经建立好的语法树进行自顶向下遍历，期间算法（不考虑错误处理）
    - 若遍历到`Block`或者`Func`，那么新建一个符号表，将`curTable`设置为新建的符号表，注意如果是`Func`需要将`Func`对应符号加入上一级符号表同时在其中记录新建的符号表引用`curTable`
    - 遍历到`Decl`项或者`FuncFormParm`，则将其加入当前符号表
    - 当从`Block`或者`Func`的遍历返回时将当前符号表设置为其父节点
  - **错误处理**：按错误类型分别讨论：
    - `a`：直接在词法分析中对解析到的`FormatString`判断即可
    - `i`、`j`、`k`：均属于同类错误，在语法分析递归下降的过程中判别，同时注意递归下降过程中不要偷懒，识别每个语法单元时要根据其完整的`First`来，否则可能出错
    - `b`、`c`：对于这两类错误可以通过在遍历语法树时根据`curTable`在该节点中查询是否存在名字或者在该节点以及其祖先节点中查询是否存在该名字，注意对于重复定义的函数其内部也要解析，对于函数形参其作用域为函数体本身，对于局部变量名可以和函数名相同
    - `d`、`e`：在解析函数实参时通过`curTable`找到该函数的定义，然后根据其中保存的函数的符号表中存储的函数的形参定义判断
    - `f`、`g`：除了维护一个`curTable`再维护一个当前的`curFunc`（`Symbol`），解析到`return`语句时就可以根据`curFunc`中存储的返回值类型对于`f`判断，`g`则是在解析`funcDef`的最后取出其`Block`中的最后一条语句判断是否为`return`
    - `h`：直接查找当前节点以及其祖先节点判断变量是否为`Const`类型
    - `l`：解析到`print`语句时直接遍历查找`%d`个数判断即可
    - `m`：维护一个全局变量`cycLevel`，进入循环块加1，出循环块减1，当遇到`break`或者`continue`语句时判断`cycLevel`是否大于0即可

编码后修改

并未将左值与其定义链接起来同时也并未做常量传播，因此对于错误`e`的判断比预想要难，对于函数调用的实参需要单独重新遍历每一个实参即`Exp`的第一个`PrimaryExp`（不需要检查`Exp`内中参与表达式计算的每一个算子`LVal`、`Number`的类型间是否匹配，所以相当于只用看`Exp`的第一个`PrimaryExp`的维数即可并不需要遍历完），然后查询并计算其维数再在符号表看实参与形参维数是否可以匹配，最复杂的还要在遍历的过程中判断除了维数问题还有没有未定义的问题，此时需要遍历所有的`PrimaryExp`（按我的编码结构是先判断未定义问题再检测类型是否匹配，并且无法在检查类型是否匹配时判断是否

已经检测出过未定义的问题）若有就返回，以免对于一行重复报错（`c`、`e`）。预计后续中间代码生成时会继续修改符号表定义

中间代码生成

编码前设计

初步设计是中间代码种类和形式尽可能贴近`mips`指令，方便后期的翻译。 其次是中间代码之间的耦合度要尽可能低，每条中间代码的功能要尽可能单一。最后是生成中间代码要 对源代码的语义进行一层的规范化处理，也是为了简化翻译过程。

我们可以在错误处理以及建立符号表的同时生成四元式的中间代码，以下为中间代码设计，基本格式为`op operand1 pperand2 res`，由于是类`mips`代码的设计，所以由中间代码的`op`很容易知道中间代码的用途：

```
// ALU
ASSIGN("="), // ASSIGN VAR (EMPTY) VAR
ADD("+"), // ADD VAR VAR VAR
SUB("-"), // SUB VAR VAR VAR
MUL("*"), // MUL VAR VAR VAR
DIV("/"), // DIV VAR VAR VAR
MOD("%"), // MOD VAR VAR VAR
NOT("!"), // NOT VAR (EMPTY) VAR
EQ("=="), // EQ VAR VAR VAR
NE("!="), // NE VAR VAR VAR
GT(">"), // GT VAR VAR VAR
GE(">="), // GE VAR VAR VAR
LT("<"), // LT VAR VAR VAR
LE("<="), // LE VAR VAR VAR

// IO
GET_INT("get_int"), // GET_INT (EMPTY) (EMPTY) [VAR or (EMPTY)]
PRINT_STR("print_str"), // PRINT_STR STR (EMPTY) (EMPTY)
PRINT_INT("print_int"), // PRINT_INT VAR (EMPTY) (EMPTY)

// FUNC
FUNC("func"), // FUNC VAR (EMPTY) (EMPTY)
FUNC_END("func_end"), // FUNC_END VAR (EMPTY) (EMPTY)
PREPARE_CALL("prepare_call"), // PREPARE_CALL VAR (EMPTY) (EMPTY)
CALL("call"), // FUNC VAR (EMPTY) (EMPTY)
PUSH_PAR_INT("push_para_int"), // PUSH_PAR_INT VAR (EMPTY) (EMPTY)
PUSH_PAR_ADDR("push_para_addr"), // PUSH_PAR_ADDR VAR (EMPTY) (EMPTY)
RETURN("return"), // RETURN [VAR or (EMPTY)] (EMPTY) (EMPTY)

// JUMP
JUMP("jump"), // JUMP (LABEL) (EMPTY) (EMPTY)
NEZ_JUMP("nez_jump"), // NEZ_JUMP (VAR) (EMPTY) (LABEL)
EQZ_JUMP("eqz_jump"), // EQZ_JUMP (VAR) (EMPTY) (LABEL)
LABEL("label"), // LABEL [(AUTO) or (LABEL)] (EMPTY) (EMPTY)

// DEF
DEF_VAL("var_def"), // DEF_VAL [VAR or (EMPTY)] (EMPTY) VAR
DEF_ARR("arr_def"), // DEF_ARR (EMPTY) (EMPTY) VAR
END_DEF_ARR("end_arr_def"), // END_DEF_ARR (EMPTY) (EMPTY) VAR

// BLOCK
```

```
BLOCK_BEGIN("block_begin"), // BLOCK_BEGIN VAR (EMPTY) (EMPTY)
BLOCK_END("block_end"), // BLOCK_END VAR (EMPTY) (EMPTY)

NOP("nop"); // NOP (EMPTY) (EMPTY) (EMPTY)
```

其中`(EMPTY)`表示占位符，同时可以令`VAR`为字符串`(AUTO)`表示自动生成一个新的临时变量或者一个新的标签，对于传入的`int`变量`VAR`定义为字符串`name(blockLevel,blockNum)`其中`blockLevel`代表变量在符号表树中的第几层，`blockNum`代表变量在符号表树的第`blockLevel`层的第`blockNum`个，对于数组变量`VAR`定义为字符串`name(blockLevel,blockNum)[index]`，其中`index`为数组展平为一维后的索引下标，最后令函数的返回值对应的`VAR`为`(RT)`

由此我们就可以直接通过字符串索引到变量在符号表中的具体位置，当然也会在中间代码中存储变量对应的`Symbol`，于是提供了两种方法对于中间代码中出现的变量进行查询，以下是一段符合`SYs`语言的代码及其对应生成的中间代码：

- `Sys`：

```
const int b = 9;

int fun(int a[][5]) {
    return a[3][6 * b];
}

int main() {
    int sum, a, b;
    if (a == 0) return 1;
    else return 2;
    sum = a + b
    return 0;
}
```

- 中间代码：

```
DEF_VAL 9 b(0,1)

========FUNC: fun(0,1)========
MUL 6 b(0,1) (T0)
ADD (T0) 15 (T1)
RETURN a(1,1)[(T1)]
========FUNC_END: fun(0,1)========


========FUNC: main(0,1)========
DEF_VAL sum(1,2)
DEF_VAL a(1,2)
DEF_VAL b(1,2)
DEF_VAL c(1,2)
DEF_VAL d(1,2)
DEF_VAL e(1,2)
SUB c(1,2) d(1,2) (T2)
MUL b(1,2) (T2) (T3)
SUB 0 e(1,2) (T4)
MOD (T3) (T4) (T5)
ADD a(1,2) (T5) (T6)
ASSIGN (T6) sum(1,2)
GE a(1,2) 8 (T7)
```

```
EQZ_JUMP (T7) (LABEL3)
LT b(1,2) 0 (T8)
NEZ_JUMP (T8) (LABEL2)
(LABEL3):
EQ c(1,2) 1 (T9)
EQZ_JUMP (T9) (LABEL0)
(LABEL2):
RETURN 1
JUMP (LABEL1)
(LABEL0):
EQ a(1,2) 3 (T10)
EQZ_JUMP (T10) (LABEL4)
RETURN 2
JUMP (LABEL5)
(LABEL4):
(LABEL5):
(LABEL1):
RETURN 0
========FUNC_END: main(0,1)========
```

对于控制流语句，翻译如下：

- `If-else`：

```
EQZ_JUMP Cond label_1
   ...(if的语句块)
JUMP label_2
label_1:
   ...(else的语句块)
label_2:
```

- `while-break-continue`：

```
label:
EQZ_JUMP Cond label__end
   ...(循环语句块)
JUMP label
label_end:
```

参考上述控制流语句，对于条件运算符`||`、`&&`同样采用上述方案，构造一系列跳转标签实现短路运算，我通过从上层传递一个`(LABEL)`与一个`boolean`变量，`boolean`变量为`true`表示如果该层`*Exp`正确那么跳转到`(LABEL)`反之顺序执行；`boolean`变量为`false`表示如果该层`*Exp`错误那么跳转到`(LABEL)`反之顺序执行，注意实现短路求值的地方只有`LAndExp`和`LOrExp`，其它都必须全部计算完，具体实现如下：

- `LAndExp`：

```
// 满足跳 x, ok = true
// 不满足跳 x, ok = false
private void lAndExpTravel(LAndExp lAndExp, Integer x, boolean ok) {
    EqExp first = lAndExp.getFirst();
    ArrayList<EqExp> eqExps = lAndExp.getTs();
    if (ok) {
        // ... && ... && ... y
```

```
        // 前面的不满足跳 y
        // 最后一条满足跳 x
        Integer y = ++MidCodeList.labelCounter; // 生成一个实现短路的局部标签
        if (eqExps.size() == 0) {
            eqExpTravel(first, x, true);
            return;
        }
        eqExpTravel(first, y, false);
        for (int i = 0; i < eqExps.size(); ++i) {
            if (i != eqExps.size() -1) eqExpTravel(eqExps.get(i), y, false);
            else eqExpTravel(eqExps.get(i), x, true);
        }
        MidCodeList.add(Code.Op.LABEL, "(LABEL" + y + ")", "(EMPTY)", "(EMPTY)");
    } else {
        // ... && ... && ...
        // 不满足跳 x
        eqExpTravel(first, x, false);
        for (EqExp exp : eqExps) eqExpTravel(exp, x, false);
    }
}
```

- `LOrExp`:

```
// ok = false 不满足跳 label
// ok = true 满足跳 label
private void lOrExpTravel(LOrExp lOrExp, Integer label, boolean ok) {
    LAndExp first = lOrExp.getFirst();
    if (!ok) {
        // ... || ... || ... x
        // 1. 满足跳 x 不满足顺序
        // 2. 满足跳 x 不满足顺序
        // ...
        // n. 不满足跳 label
        Integer x = ++MidCodeList.labelCounter; // 生成一个实现短路的局部标签
        // ... 1. 不满足跳 label
        // ... || ..... 1. 满足跳 x
        if (lOrExp.getTs().size() == 0) {
            lAndExpTravel(first, label, false);
            return;
        }
        lAndExpTravel(first, x, true);
        for (int i = 0; i < lOrExp.getTs().size(); ++i) {
            if (i != lOrExp.getTs().size() - 1) lAndExpTravel(lOrExp.getTs().get(i), x, true);
            else lAndExpTravel(lOrExp.getTs().get(i), label, false);
        }
        MidCodeList.add(Code.Op.LABEL, "(LABEL" + x + ")", "(EMPTY)", "(EMPTY)");
    } else {
        // ... || ... || ...
        // 满足跳 label
        lAndExpTravel(first, label, true);
```

```
28          for (LAndExp exp : lOrExp.getTs()) lAndExpTravel(exp, label,
    true);
29      }
30  }
```

中间代码的生成可以与错误处理和符号表建立同时在解析语法树的时候做，解析到对应的句子时候生成对应的中间代码

编码后修改

- 增加 `END_ARR_DEF` 便于区分赋值与初始化，同时增加 `Calc` 类可以直接在扫描语法树的过程中计算出 `ConstExp`、和全局变量的初值
- 没有对于数组变量的 `load` 或者 `save`，修改方法为遇到表达式右侧的数组变量生成一句 `ASSIGN a(xx, xx)[index] (Txx)` 中间代码，代表将内存中数组中的值加载到临时变量中，遇到左侧的数组变量那么一定会生成 `ASSIGN T(xx) a(xx, xx)[index]` 中间代码，代表直接将内存中的地址赋值，其它的中间代码中除了函数压栈可能会与数组有关外都不会出现与数组有关的操作数，于是将数组变量与其它变量区分开来，对于数组变量的操纵都是直接针对内存

目标代码生成

编码前设计

初步设计是对每条中间代码书写翻译方法，扫描中间代码完成 `MIPS` 的翻译工作。翻译之前先规划 好 `.data` 空间和 `.text` 空间如何使用，以及运行时空间应当如何管理，初步设计是要打印的字符串常量放在 `.data` 空间，代码放在 `.text` 空间，运行时栈空间大致分成两个部分，一部分用来映射寄存器用于寄存器的保存，另一部分用来映射变量，具体设计细节如下：

- **内存分配**：
  - 全局变量：全局数组变量全部在 `.data` 段初始化，全局 `int` 型变量从 `$gp` 开始处放置
  - 局部变量：如果寄存器有位置放置在寄存器中其余则均放置在栈中，且进入函数前，就会为每个函数在栈中分配空间，该空间大小为其中间代码所包含的所有临时变量和局部变量的大小且该大小以及它们相对于函数栈针的偏移量会在生成汇编代码前通过递归遍历符号表计算得出，即栈针只在函数调用前后上下移动一次，在函数执行过程中不会移动，只会通过偏移量计算出栈中变量的位置进行寻址
- **寄存器分配**：采取理论课中介绍的临时寄存器分配方法，由于没有做图着色等优化，并没有区分全局寄存器和临时寄存器，如果当前考虑的寄存器空闲那么直接分配，如果没有那么就需要将当前寄存器的值写回再建立新的寄存器与变量之间的映射，其次是按照寄存器序号循环遍历，每次分配寄存器只考虑当前所指向寄存器，分配完后指针指向下一个寄存器等待下次分配。但是有部分特定寄存器不参与分配：
  - `$zero`：0号寄存器，永远为0
  - `$at`：操作系统使用
  - `$v0`：专门用于存储函数返回值以及系统调用传参
  - `$a0`：专门用于 `print`，`getInt` 函数以及给立即数赋值
  - `$a1`：专门用于计算数组地址，处理移位用
  - `$gp`：全局 `int` 变量指针
  - `$sp`：栈指针
  - `$ra`：函数返回地址

  具体提供的函数接口如下：

```
1  public class RegAlloc {
2      // ...
3
```

```
    //强制給一个变量分配寄存器，如果有空寄存器直接用，反之写回后用，set表示是否建立寄存器与变量映射关系
    public static String mandatoryAllocOne(Symbol sym, Integer off, boolean set) {
        // ...
    }

    // 强制给一个变量分配寄存器，如果有空寄存器直接用，反之写回后用，set表示是否建立寄存器与变量映射关系
    //但是分配的寄存器不得为o，防止寄存器冲突
    public static String mandatoryAllocOne(String o, Symbol sym, Integer off, boolean set) {
        // ...
    }

    // 强制建立寄存器与变量的映射关系，不用管寄存器中原本是什么
    public static void mandatorySet(String reg, Symbol sym, Integer off) {
        // ...
    }

    // 如果有空寄存器就分配，否则不分配
    public static void allocFreeOne(Symbol sym, Integer off) {
        // ...
    }

    //寻是否有找寄存器中保存了某个变量
    public static String find(Symbol sym, Integer off) {
        // ...
    }

    //返回所有使用的寄存器，便于函数调用回写或者改变基本块时回写
    public static HashMap<String, Pair<Symbol, Integer>> getAllUsed() {
       // ...
    }

    //    清除一个寄存器的映射关系
    public static void refreshOne(String reg) {
       // ...
    }

    // 重新给一个寄存器建立映射关系
    public static void reflectOne(String reg, Pair<Symbol, Integer> p) {
        // ...
    }
}

```

- **寻址**：封装两个函数`save`、`load`处理所有的存取操作，对于`save`，如果寄存其中有该变量就存入其中否则存入内存中，对于`load`同理，注意对于地址形参，我们要将值存入该地址处而不是存入该地址在栈中的位置，而取出同样是去该地址处取而不是去栈中取出它，接口如下：

```java
/**
* @param reg: reg
* @param lVal: number, val, funcFormParam, reg
* tips: 如果lVal在寄存器中或者为寄存器，不会对reg建立和lVal(或者lVal中存储变量)的映射，对于funFormParam的数组直接从内存具体索引加载值
*/
public void loadLVal(String reg, String lVal);

/**
* @param rVal:  reg, number, Val, funcFormParam
* @param lVal: Val, funcFormParam, tmp，对于funcFormParam数组直接回写数组内存具体索引地址
*/
public static void saveLVal(String rVal, String lVal);
```

- **函数调用**：
  - **寄存器保护**：每次发生函数调用就将寄存器中的值调用`save`函数全部写回，不需要压栈，于是函数调用结束也不需要恢复，之后用到直接从内存取即可
  - **栈针变化**：再调用函数中将栈指针下移被调用函数所需空间大小，调用结束由调用函数恢复
  - **参数压栈**：此时栈指针还未移动但是由于我们已经在符号表中存储了参数的相对地址于是可以直接计算出参数地址然后压栈
  - **返回值与返回地址**：返回地址由被调用函数压栈，返回时取出，返回值不需要压栈直接存储在`$v0`中，函数返回后执行一条将返回值赋给临时变量的指令
- **四元式翻译**：对于四元式中的出现的不同的`op`，采用不同的翻译方法：
  - **基本运算**：
    - 将三个操作数中的非结果中间变量转化为变量所在的寄存器(若未分配寄存器则通过`lw`加载到临时寄存器中)，结果操作数则直接分配寄存器或找到它在的寄存器。
    - 将中间代码翻译为对应的mips指令。
  - `IO`：`getint`直接执行系统调用`li $v0, 5 - syscall`，结果赋值给变量。
    - `output`若为`STRCON`，先获取位于`.data`段中的地址，再进行系统调; 若为一个`int`，则将该变量存入`$a0`后执行系统调用。
  - **函数调用相关**：

    维护一个调用函数栈，栈顶存储当前调用函数及其实参所在变量
    - 在`PREPARE_CALL`时，记录当前调用函数，压入调用函数栈（为了处理多重调用，实际是编码过程中改的）。
    - 在`PUSH_*`时计算参数值或数组的首地，加入调用函数栈栈顶
    - `CALL`：将栈顶弹出，将其中参数压栈然后调用对应函数
  - **跳转相关**：直接翻译即可，中间代码这部分生成十分完善
  - **赋值以及初始化**：直接调用`save`函数即可，注意全局数组变量的初始化以及字符串的初始化单做如下：

```
// 数组
.data
arr_name: .space SPACE
 ...

// or

.data
arr_name: .word val1 val2 ... valn
...
```

```
11
12  // 字符串
13  .word
14  str_num: .asczii STRCON
```

编码后修改

编码前存在许多情况未考虑，具体如下：

- 函数参数压栈，如果实参是一个`int`变量和函数调用的返回值，那么按照之前的设计，将会先将`int`变量压栈，然后去计算函数调用返回值，再将函数调用的返回值压栈，而实际这样会使做为实参的函数调用在发生调用时覆盖之前压栈的`int`变量，最后传入错误的参数，所以一定要将所有实参全部都计算出来再一起压栈，比如如下·情况：

  ```
  PREPARE_CALL fun0(0,1)
  PUSH_PAR_INT a(1,2)
  PREPARE_CALL fun1(0,2)
  CALL fun1(0,2)
  ASSIGN (RT) (T1)
  PUSH_PAR_INT (T1)
  CALL fun0(0,1)
  ```

  对应于`Sys`代码就是：

  ```
  void fun0() {
      // ...
  }

  int fun1() {
      //...
      return 1;
  }

  // ...
  int main() {
      int a = 9;
      // ...
      fun0(a, fun1());
      // ...
  }
  ```

  可以看到中间代码先将`a`压栈了，然后去计算`fun1()`的返回值，由于是函数调用嵌套，同时我对于函数调用的处理是先将参数按符号表计算出的位置压栈，但是栈针不移动，全部压完栈再移动栈针，故对于嵌套的函数调用`fun1()`的栈空间就会覆盖掉压入栈的`a`，所以需要先全部计算出函数的实参值后一并压栈

- 对于全局数组地址作为形参时，由于`save`函数对于形参地址是直接写入地址处，于是当该地址在寄存器中且需要回写时就会将地址写入地址处（赋值时不会有问题，因为会将值直接写入地址处），采取的方法是永远不回写作为形参的地址，毕竟地址不会改变，这样同时还会节省一些访存类指令
- 分配寄存器时要考虑寄存器的冲突，需要考虑需要两个寄存器作为操作数的`alu`类指令，当第一个寄存器刚分配，用可能该寄存器就又被分配给了第二个操作数，要确保这两个寄存器在不是同一个变量的时候不同

- 目前还没划分基本块，但是注意由于控制流语句以及函数调用会导致函数执行流混乱且无法预测，于是需要在进入一个基本块前将所有寄存器回写防止出现函数在运行过程中明明回写了某值但是由于是循环，于是接着再次执行的时候会再次执行`sw`指令回写一个错误的值（区分生成汇编代码过程和函数设计执行过程的关系），但是只要再进入基本块前就将所有寄存器回写就可以避免问题，注意由于做了短路求值，于是每一次条件跳转以及函数调用前都需要将寄存器的值回写，编码前没有考虑过这种情况，同时为了解决这种情况还生成了一些空跳指令，比如如下情况：

```
label:  // 1
EQZ_JUMP Cond label__end
 ...(循环语句块)
JUMP label
label_end:
```

注意上面中间代码在`1`这个位置，由于`1`之前的代码可以进入`1`，而第四行的跳转也可以进入`1`，因此需要保证从两个位置处进入`1`的寄存器分配状态是一致的，即必须在进入`1`前清空寄存器分配的映射同时回写，没有划分基本块，否则可以出基本块回写，进入基本块解除寄存器映射，所以一个折中的解决方法就是遇到跳转指令就回写同时解除映射，故需要在`1`前生成一个跳转`JUMP label`，保证寄存器状态在进入`1`前已经为空

- 使用`*u`指令，使用不带`u`的`alu`类计算指令当发生溢出时会直接进入异常处理程序发生异常
- 对于寄存器中的全局变量，它已发生改变立刻回写，因为正常函数的执行顺序是乱的，而我们需要在一个函数调用结束前就将全局变量回写，但是这样做比较麻烦，因为正常的函数的局部变量在执行完后就销毁了，所以改为如果对于全局变量有赋值语句执行完后立刻回写

代码优化

代码优化部分由于是想清楚算法后就开始编码，因此并未有过编码后的修改，仅仅只有编码时候笔误或者写错的一些细节，优化架构的初步设计是对每个优化尽量做成单独的模块，并为其设置易于管理的优化开关，并且要 保持对每一个优化版本的兼容性，优先保证编译程序的正确性。初步优化思路是先对程序进行数据流 析，然后基于数据流分析进行优化，优化主要分为中间代码层面和目标代码层面，以及穿插在各个部分的小 tricks.

运算强度削弱

乘除

- 首先对于乘法，可以将赋值操作转换为移位操作，例如将`mul $s1, $s2, num`可以翻译为：

```
sll $s1, $s2, x0
sll $a1, $s2, x1
addu $s1, $a1, $s1
sll $a1, $s2, x3
addu $s1, $a1, $s1
# ...
nge $s1, $s1 # if num < 0
```

注意$|num| = 2^{x_i} + 2^{x_{i-1}} + \ldots + 2^{x_0}$，其中$x_i > x_{i-1} > \ldots > x_0 \geq 0$，该拆分可以提前计算出来，同时需要特判$num = 0$的情况，我们可以计算出这种转换所用的指令的`cycle`数，如果`cycle`数大于等于4，那么应该直接翻译为一条乘法指令，不然就是负优化

- 对于除法，可以参考下面的算法：

```
procedure CHOOSE_MULTIPLIER(uword d, int prec);
Cmt. d - Constant divisor to invert. 1 ≤ d < 2^N.
Cmt. prec - Number of bits of precision needed, 1 ≤ prec ≤ N.
Cmt. Finds m, sh_post, ℓ such that:
Cmt.      2^(ℓ-1) < d ≤ 2^ℓ.
Cmt.      0 ≤ sh_post ≤ ℓ. If sh_post > 0, then N + sh_post ≤ ℓ + prec.
Cmt.      2^(N+sh_post) < m * d ≤ 2^(N+sh_post) * (1 + 2^-prec).
Cmt. Corollary. If d ≤ 2^prec, then m < 2^(N+sh_post) * (1 + 2^(1-ℓ))/d ≤ 2^(N+sh_post-ℓ+1).
Cmt.            Hence m fits in max(prec, N - ℓ) + 1 bits (unsigned).
Cmt.
int ℓ = ⌈log2 d⌉,   sh_post = ℓ;
udword m_low = ⌊2^(N+ℓ)/d⌋,   m_high = ⌊(2^(N+ℓ) + 2^(N+ℓ-prec))/d⌋;
Cmt. To avoid numerator overflow, compute m_low as 2^N + (m_low - 2^N).
Cmt. Likewise for m_high. Compare m' in Figure 4.1.
Invariant. m_low = ⌊2^(N+sh_post)/d⌋ < m_high = ⌊2^(N+sh_post) * (1 + 2^-prec)/d⌋.
while ⌊m_low/2⌋ < ⌊m_high/2⌋ and sh_post > 0 do
    m_low = ⌊m_low/2⌋;   m_high = ⌊m_high/2⌋;   sh_post = sh_post - 1;
end while;                    /* Reduce to lowest terms. */
return (m_high, sh_post, ℓ);  /* Three outputs.          */
end CHOOSE_MULTIPLIER;
```

```
Inputs: sword d and n, with d constant and d ≠ 0.
udword m;
int ℓ, sh_post;
(m, sh_post, ℓ) = CHOOSE_MULTIPLIER(|d|, N - 1);
if |d| = 1 then
   Issue q = d;
else if |d| = 2^ℓ then
   Issue q = SRA(n + SRL(SRA(n, ℓ - 1), N - ℓ), ℓ);
else if m < 2^(N-1) then
   Issue q = SRA(MULSH(m, n), sh_post) - XSIGN(n);
else
   Issue q = SRA(n + MULSH(m - 2^N, n), sh_post)
             - XSIGN(n);
   Cmt. Caution — m - 2^N is negative.
end if

if d < 0 then
   Issue q = -q;
end if
```

当然如果对于立即数属于$\{1,-1\}$的情况特判会更快，上述算法的主要思想是将$\frac{n}{d}$转化为$\frac{n\times m}{2^{N+l}}$，即乘法加移位运算，当然还需要考虑$m$的溢出、$n$的正负性的问题

地址计算

对于访存一个数组的元素，比如`a[k]`，我们可以用移位运算代替乘法去计算偏移量

常量合并

- 对于类型为`const`或者位于全局的变量的初始值都可以在中间代码生成的过程中计算出来，于是可以写一个计算类，通过`Java`的异常机制，计算不出来那么就生成正常的中间代码
- 对于`stmt`中的一些计算语句，或者比较语句我们也可以将其算出来，但是如果遇到左值，可以用初值代替其中为`const`的变量，而非`const`变量则不行，防止变量除了初始化之外还在其它地方被赋值

对于如下代码：

```
const int a[10] = {1,2,3,4,5,6,7,8,9,10};

int main() {
    int b = a[0] + 5;
    printf("%d", a[7] + 10);
    printf("%d", b);
    return 0;
}
```

可以翻译成如下中间代码：

```
// ...
======FUNC: main(0,1)======
DEF_VAL 6 b(1,1)
PRINT_INT 18
PRINT_INT b(1,1)
RETURN 0
======FUNC_END: main(0,1)======
```

`do-while`

对于`Sys`语言中的`while(cond) {...}`可以翻译成`if (cond) {do{...} while (cond)}`，即如下：

```
# while
begin:
beqz cond end
...
j begin
end:

# do - while
beqz cond end
begin:
...
bnez cond begin
end:
```

对于一开始不满足情况那么二者均为跳转一次，但是如果多次进入循环体的内部，易得上面的`while`会执行$2n-1$次跳转，而下面的`do-while`会执行n次跳转，且不会有其他副作用，但是注意的是要翻译两次`Cond`，一次是满足要跳转`begin`，一次是不满足要跳转到`end`，这里有个问题就是如果你是解析两次`Cond`，那么如果遇到错误，你会报两次错，所以建议用一个行号为键值的`HashMap`来存储错误，这样就不会报两次同一行的错误了

数据流分析

划分基本块与流图的建立

正如在`mips`代码生成那里所说的一样，没有划分基本块，函数执行顺序信息是得不到的，只能通过一些及其不优雅的方法，例如生成空跳转来解决进入一个基本快寄存器状态不一致的问题，所以要先划分基本块并建立流图，注意这是函数内部划分，具体方法如下：

- 跳转型中间代码`JUMP`、`EQZ_JUMP`、`NEZ_JUMP`、`RETURN`的下一句作为入口点，同时除`RETURN`外，其它跳转要跳到的`LABEL`也作为入口点
- 所有入口点的前一句作为出口点，将所有代码按照入口点与出口点从小到大逐一匹配就划分完成基本块
- 将末尾为无条件跳转的基本块与其跳转到的基本块相连，注意`RETURN`算无条件跳转，且直接跳转到B_{exit}；其余则将自己与自己按代码顺序的下一个基本块相连，同时如果最后一条是条件跳转，那么也将它与要跳转的的基本块相连

注意在此过程进行前其实可以删除一些不必要的跳转，比如刚好要跳到的就是下一句话，或者是无条件跳转下方接一条跳转；还可以优化部分跳转，比如跳转到的`LABEL`下方恰好是一条无条件跳转，那么可以将自己跳转的`LABEL`直接变为找到的无条件跳转的`LABEL`，注意这个过程可以用递归循环找，但是要特判跳转成环的情况

在刚化完流图时，我们其实可以从所有基本块的入口开始`dfs`，遍历流图，然后将不会经过的基本块删除，这样在后续的到达定义分析，活跃变量分析得到的结果会更加准确

同时由于划分了基本块，我们可以改掉之前为使寄存器保持状态一直而生成空跳，并且在后续窥空也无法删掉它们的做法，新的做法如下：

- 在进入一个基本块前清空寄存器映射关系
- 在出基本块时，且最后一条语句不是跳转或者`RETURN`时执行完它立刻回写，如果是跳转则在跳转前回写，`RETURN`不需要回写

到达定义分析

注意对于数组由于较为复杂因此是不参与的，按照书本的定义$out[B]=in[B]\cup(gen[B]-kill[B])$，$in[B]=\cup_{B的前驱}out[B_{pre}]$从$B_{exit}$开始反复迭代即可，但是要注意:

- 对于一个基本块自身而言它的$kill[B]=kill[d_1]\cup kill[d_2]\cup...\cup kill[d_n]$，对于每一个$kill[d_i]$，它等于考虑的范围是这个函数的所有基本块中除了它自己之外所有中间代码
- 对于$gen[B]=gen[d_n]\cup(gen[d_{n-1}]-kill[d_n])\cup...\cup(gen[d_1]-kill[d_2]-...-kill[d_n])$，即我们可以从后往前倒着考虑每一条中间代码，如果发现该条中间代码所生成的信息已经在$gen[B]$里面了就不用添加了

活跃变量分析

基本上与到达定义分析相同，但是是从后往前分析，同时还有注意：

- 在计算$use[B]$的时候，数组元素不参与考虑，但是数组的下标如果是一个普通变量的话要算进来，同时函数调用的返回值也不必考虑
- $def[B]$和$use[B]$需要先统计每一个变量在基本块中第一次被定义和使用的时候，如果第一次使用的位置小于等于第一次被定义的位置，那么加入$use[B]$，否则加入$def[B]$

死代码删除

该优化以活跃变量分析的结果为基础：

- 首先维护一个队列q将基本块的out_{active}中的所有变量加入其中，然后倒着遍历该基本块的所有代码
- 对于当前遍历到的代码，如果它的def不为空，且为于q中或者它的def为空（def为空一定为跳转或者`printf`这种会产生副作用的代码），那么将它的use加入队列，同时将该def移除出队列；如果它的def不为空且不在队列中，那么则可以删掉改代码

注意对于含有对全局变量赋值意义的代码，我们不能删除，同时也要按照def不为空时将它的use加入队列处理；对于`getint`，即使他的def不为空且不在队列中，我们也不可以删掉，只能将它的结果由一个变量改为`(EMPTY)`，这样也省略了一条赋值语句

由于删除了很多死代码，于是也产生了很多地方可以窥孔，以至于会导致划分的基本块不再准确于是又可以重新划分基本块再优化一遍，直到优化结果不变为止

常量传播与复写传播

策略是常量可以跨基本块传播，变量只能块内传播，因为跨基本块前一定会回写内存，解除寄存器映射，如果跨基本块传播变量，那么极有可能需要读取内存，因此很有可能导致负优化

需要注意全局变量不能传播与被传播，因为全局变量的值可能在中间某次函数调用的时候发生改变，而我们的数据流分析并不会考虑函数调用因此需保守处理，同时数组同理也不处理，具体算法如下：

- 首先维护一个产生了def信息的代码的队列q，该队列中代码之间的顺序与它们在基本块中的顺序一致；
- 对于当前遍历到的代码，取出它的use信息，然后倒着遍历q，一旦发现q中有def的变量为use中的变量，那么停止遍历，
    - 同时检测该def的代码是否是常量赋值或者将变量定义为常量的代码，如果是那么传播常量；
    - 如果为变量赋值或者将变量定义为变量，那么接着从q的当前位置正向遍历，检查该变量赋值或者变量定义的代码右侧所使用的变量是否在之后被重新定义，如果没有，那么将改变量复写传播给use
- 如果倒着遍历完了整个队列q都没有可以进行的传播，且use变量没有出现在q中代码所产生的def信息中，那么可以检测该基本块的in_{arrive}中的变量，如果use信息中的变量只在其中出现过一次且满足上述常量传播的条件，那么可以进行跨基本块的常量传播

同样做了此优化会有很多窥空的空间，以至于会导致划分的基本块不再准确于是又可以重新划分基本块再优化一遍，直到优化结果不变为止

寄存器分配

图着色分配

对于局部变量和临时变量采取图着色分配策略

- 首先先计算每一条中间代码的out_{active}，然后通过分析该中间代码的操作数、结果、out_{active}之间是否有冲突关系建立冲突图
- 按照书上此步操作应该导出染色节点的序列，但是这步书上并未给出一个较好的算法，这里推荐使用MCS算法，求出该图的完美消除序列，对于实现了`SSA`的同学，这个算法是一个最优解，但是即使没有实现`SSA`，该算法也十分有效，具体可以参考https://oi-wiki.org/graph/chord/
- 当求出消除序列就可以按照理论课上的算法对原图进行染色了，对所有节点染完色后，由于先前求出的序列已经是一个"完美消除序列"，因此在此直接对每种颜色所覆盖的节点数量进行排序，先为覆盖节点数量最多的颜色建立一个寄存器映射、接着为覆盖节点第二多的节点……

按图着色分配寄存器最大的好处不仅仅是寄存器利用更加充分了，而且每次跨越基本块的时候不需要回写变量了

$CLOCK$分配

对于在图着色中未分配到寄存器的变量可以使用操作系统中学过的页面分配的改良过的$CLOCK$算法：

- 维护一个指针用于遍历当前寄存器池中的寄存器
- 当要分配寄存器时，如果当前指向的寄存器为空，那么分配，同时指针指向相邻的下一个寄存器，如果不为空指针指向下一个寄存器重复之前的操作，如果一直没有空寄存器那么就直到指针回到要为变量分配寄存器时的初始位置，然后此时再循环一轮，在此轮过程中查看寄存器的脏位是否为`true`，不为`true`就分配，反之接着直到右回到要为变量分配寄存器时的初始位置，同时将该寄存器中的变量回写内存寄存器的脏位置为`false`，指针移动

注意将一个变量写回内存时脏位置为`false`，`ASSIGN`、`ALU`类指令的目的寄存器的脏位置为`true`

引用计数

对于**未分配到图着色寄存器**的临时变量，可以直接扫描中间代码中临时变量出现的次数，统计一遍，然后在翻译为`mips`的过程中记录已经使用的次数，如果已经使用的次数与出现次数相同那么在翻译当前中间代码的过程中不需要将其回写内存，同时翻译完该条中间代码就可以将该变量与寄存器的映射关系清空

窥孔优化

对于中间代码可以在删除完四代码以及做完常量传播的时候做，比如：

- 对于`ADD`的加0，`SUB`的减0，`MUL`的乘1，`DIV`的除1，`MOD`的摸1
- 对于跳转到的刚好是下一条句子，那么可以删除；其次：

  ```
  ASSIGN 8 a
  NEZ_JUMP a LABEL
  ```

  可以优化为：

  ```
  JUMP LABEL
  ```

  还有：

  ```
  EQZ_JUMP a LABEL1
  JUMP LABEL2
  LABEL1:
  ```

  那么可以优化为：

  ```
  NEZ_JUMP a LABEL2
  LABEL1:
  ```

  上述情况其实很多时候都会出现，如果做了短路求值，以及常量合并的情况下
- 对于进行完常量传播的代码，有的可以直接算出结果，那么也可以用一条`ASSIGN`去替代之前的计算指令，有可能在之后的再次常量传播以及死代码删除的过程中就可以被优化掉

对于目标代码可以将：

- `move`到同一个寄存器的指令优化掉
- 连续的`sw`和`lw`如果是对于同一个寄存器和地址那么可以优化掉，比如：

```
sw $v0, 28($sp)
lw $v0, 28($sp)

# 或者
lw $v0, 28($sp)
sw $v0, 28($sp)
```

  可以只保留第一条代码
- 如果检测到是个叶子函数，那么进入函数与离开函数时对于`$ra`寄存器的保存与恢复都可以删掉

无用函数调用删除

对于一些没有实际效益的函数调用我们可以直接去除关于它的调用，那么什么叫没有实际效益呢：

- 被调用函数没有返回值或者说返回值没有被用到
- 被调用函数没有对全局变量进行赋值修改
- 被调用函数没有`IO`语句，没有调用其它函数
- 调用函数传递的实参中没有数组地址

以上检查可以在语义分析生成中间代码的时候做，如果满足上述条件，那么此次函数调用可以直接在中间代码中删除

指令选择

我们可以注意到如下事实：

- `div`的三操作数指令会多出判断除数是否为`0`的分支，可替换成`div + mflo`
- `mul`比`mult + mflo`要少一个指令
- 用`addi`来替代`subi`，`subu`，`addu`直接对立即数进行使用。在立即数少于等于`16`位下，即在`[-32768, 32767]`下效果显著。
- 跳转以及置位指令要用`slti`、`slt`、`sgt`、`bne`、`bnez`、`beq`、`beqz`，而其它比如`blt`等等都会在`Mars`被翻译成三条或者更多的指令

循环不变式外提

由于没有实现`SSA`，因此该优化比较难做，但是速度的提升是显著的，在完成数据流分析后首先需要找到属于循环的基本块，可以在中间代码直接标记，也可以由以下算法得到，注意在此`while`循环均已经化为`do-while`循环：

- 计算到达每个节点的必经节点集合D：

  假定有x个节点：

  - 对每一个节点B_i，初始化$D(B_i)=\{B_1,B_2,\dots,B_x\}$，除$D(B_1)=\{B_1\}$
  - 每一步计算中，若当前节点B_i的父节点集合为$\{B_r,B_s,B_t\dots\}$，那么$D(B_i)=[\cap D(B_{fa})]\cup\{B_i\}$
  - 如果在一次迭代中，x个节点中没有节点的必进节点集发生变化，就退出
- 查找回边：
  - 如果一个程序流图中存在有像边$<B_i,B_j>$并且$B_j\in D(B_i)$，那么我们就说$<B_i,B_j>$为一条回边
- 查找循环：

    - 找出回边$< B_i, B_j >$
    - 则B_i, B_j必定属于回边$< B_i, B_j >$组成的循环L中，即$B_i \in L$且$B_j \in L$
    - 若$i \neq j$且i的父节点不在L中将它加入L中，对于求出的父节点，令其为i，重复执行这步操作，直至不再有新节点加入为止

我们可以记循环的入口节点为B_{entry}，即上述算法中的B_j，对于入口节点相同的循环，我们也可以将他们直接合并（可能是在做别的优化时导致），同时我们定义循环的出口节点集为$Exit$，即$B_t \in Exit$那么存在B_t的某个后继节点不在L中，接下来要判断那些是循环不变代码，然后外提，判断循环不变代码的算法如下：

- 一次查看L中各个基本块的代码，如果他们的use集合的变量为常数或者它们的in_{arrive}中它们自己的定义点都来自于L外，那么将这句代码标记

    > 注意这里要求每句代码的到达定义分析，可以在完成数据流分析的基础上再在基本块内逐句遍历求解，复杂度为线性，无需对整个中间代码多次迭代

- 接下来依次查看L中是否还有没有被标记的代码，如果它的use集合中变量满足为常数，或者它们的in_{arrive}中定义点在L以外，或者只有一个且该点代码已经被标记，则将此代码标记，这步操作需要不断迭代，直到不再有代码需要标记为止

但是上述算法所求到的并不是最终的循环不变式集合，因为此时只考虑了中间代码的使用变量，并未考虑它的定义变量，如果该中间代码在循环内不一定被到达，那么提出循环很有可能就错了，所以需要加强约束，对于所有标记的代码s，如果它是循环不变式，那么它需要同时满足以下条件：

- s所在节点是循环L的所有出口节点的必经节点，或者s在离开循环后都不再活跃

    > 即对于所有出口节点的后继节点中不在L中的节点，它们的所有in_{active}中都不包含s代码所定义（赋值）的变量

- s在循环L的**其它基本块**中没有定值语句
- 循环L中所有对于s定义（赋值）的变量的引用，只有**s所在的基本块**中对于它的定义可以到达

    > 注意此处约束均是保证s所在基本块即可，并不需要约束到s，网上参考资料均要求约束到s（可能是本身说得不太清楚），实际上在一个基本块内代码顺序执行，所以没有必要约束到s

最后就是按照所标记循环不变式的顺序，将符合要求的循环不变式s提到循环外，注意如果s的use集合中的变量是在L中定义（赋值）的，那么只有当这些定义（赋值）语句都提到循环外面，s才能提到循环外

> 对于s的use集合中的变量是在L中定义（赋值）的，可以求s的in_{arrive}，然后查看其中对于use中变量的在L中定义（赋值）语句是否已经提出

至于具体如何提出，可以是新建一个基本块节点，将所有提出代码加入其中，然后将它指向循环入口，将之前指向循环入口的所有基本块指向该新建基本块，当然正常情况下L中的入口节点只有一个，且它的前驱中不属于L的节点只有一个，直接将提出代码放到入口的此前驱节点最后即可