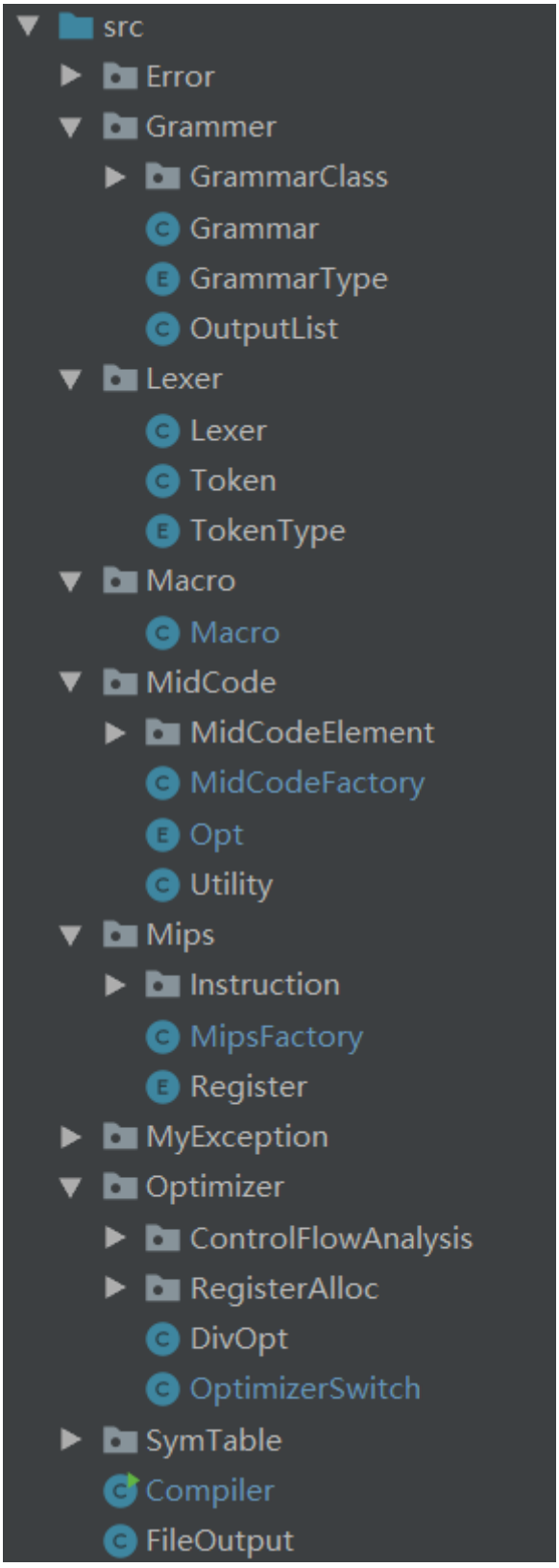
编译器设计文档

19376273 陈厚伦

零、总体架构

项目结构



设计分为**前端、中端、后端。前端**包括词法分析（含错误处理）、语法分析（含错误处理）、语义分析（含错误处理）、中间代码生成，**中端**包括大部分代码优化，**后端**为目标代码生成。

> **Lexer**

词法分析。`TokenType`定义了切词后的`Token`的类别码，`Token`类定义了一个`Token`的原始字符串、`token`类别码、行号、值（`INTCON`）、计数值（`FormatString`中`%d`个数）、有效性信息，`Lexer`是词法分析器，外部调用一个`Lexer`实例的`getToken()`方法可以读取一个`Token`.

> **Grammar**

语法分析。`OutputList`负责管理输出内容，`GrammarType`定义了语法成分名称，`Grammar`是语法分析器，接收`Token`构成的`List`并对外提供`analyze()`方法来开始递归下降分析语法成分。`GrammarClass`下是各个语法成分的类，各个语法成分继承抽象类`Node`，均实现`analyze()`方法，同时管理本语法成分通过文法推导出的语法成分和分支信息，以构建`AST`.

> **SymTable**

负责管理符号表。符号表包含简单的栈式符号表。`TableItem`为符号表项，管理标识符名称`identName`，标识符类型`identType`，数据类型`dataType`，数组的第一维长度（如果有）`array_dim_1`，数组的第二维长度（如果有）`array_dim_2`，域信息`dim`。`StackTable`实现了栈式符号表，提供了符号表的增、删、改、查操作。在后面的 MIPS 生成中，为了实现前后端的符号表解耦，增添了汇编符号表类，`MipsTableItem`，`MipsFuncTable`，`MipsGlobalTable`主要用来协助完成运行时空间管理。

> **MyException**

异常管理。定义了`EOF`异常，判断是否解析达到源程序末尾以结束程序运行。

> **Error**

错误处理。`ErrorType`定义了错误类型。`MyError`实现错误处理，一个`MyError`实例记录报错的`token`、行号`lineNo`、错误类型`errorType`，详细信息`msg`，同时`MyError`类管理一个静态的错误列表`myErrors`，提供了一系列添加不同类型错误的静态方法。

> **FileOutput**

负责文件写。

> **Macro**

定义了全局常用的一些常量。

> **MidCode**

中间代码管理，`MidCodeElement`下管理所有的中间代码类，`MidCodeFactory`是中间代码生产工厂，`Utility`定义了中间代码生成中常用的工具方法，枚举类`Opt`定义了所有中间代码的识别符。

> **Mips**

`MipsFactory`是mips翻译器，用来管理运行时空间，并将中间代码翻译成mips。`Register`中定义了所有的寄存器。

> **Optimizer**

编译器的优化中端。

> **Compiler**

编译器顶层设计，代码如下：

```
public class Compiler {
    public static void main(String[] args) throws IOException {

        FileOutput fileOutput = new FileOutput("mips.txt");
        // 词法分析
        Lexer lexer = new Lexer();
        ArrayList<Token> tokens = getTokens(lexer);
        // 语法分析
        Grammar.init(tokens);
        try {Grammar.analyze();} catch (EOF e) {}
        // 若出错停止编译
```

```
        if (MyError.myErrors.size() > 0) {
            System.exit(0);
        }
        // 中间代码生成
        ArrayList<MidCode> midCodes = MidCodeFactory.getMidCodes();
        // 代码优化
        if (OptimizerSwitch.FLOW_OPT) {
            if (OptimizerSwitch.STRONG_REG_ALLOC_OPT) {
                Flow.createFlow(midCodes);
                Flow.global2local();
                Flow.createFlow(midCodes);
                if (OptimizerSwitch.COPY_SPREAD_OPT) {
                    Flow.adjustBaseblocks();
                    Flow.copySpread();
                    Flow.updateMidCodes();
                }
                Flow.killGenAnalyze();
                Flow.arriveDefAnalyze();
                Flow.calculateAllUses();
                Flow.defUseChainAnalyze();
                Flow.defUseWebAnalyze();
                Flow.webRename();
                if (OptimizerSwitch.GLOBAL_CONST_SPREAD) {
                    Flow.globalConstSpread();
                }

                if (OptimizerSwitch.COPY_SPREAD_OPT) {
                    Flow.copySpread();
                    Flow.updateMidCodes();
                }
            }
            Flow.createFlow(midCodes);
            Flow.useDefAnalyze();
            Flow.activeAnalyze();
            Flow.conflictAnalyze();
            Flow.initRegAllocPool();
            if (OptimizerSwitch.DEAD_CODE_DELETE_OPT) {
                Flow.deadCodeDelete();
            }
            Flow.adjustBaseblocks();
            if (OptimizerSwitch.NO_SIDE_EFFECT_DELETE_OPT) {
                Flow.clearSideEffect();
                Flow.NoSideEffectDeleteOpt();
            }
        }
        // 目标代码生成
        MipsFactory.createMips(midCodes);
        fileOutput.output(MipsFactory.mipsToString());
        fileOutput.closeFile();
    }

    private static ArrayList<Token> getTokens(Lexer lexer) {
        ArrayList<Token> tokens = new ArrayList<>();
        try { while (true) { tokens.add(lexer.getToken()); } } catch (EOF e) {}
        return tokens;
    }
}
```

编译器工作流程

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`，构建栈式符号表，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`

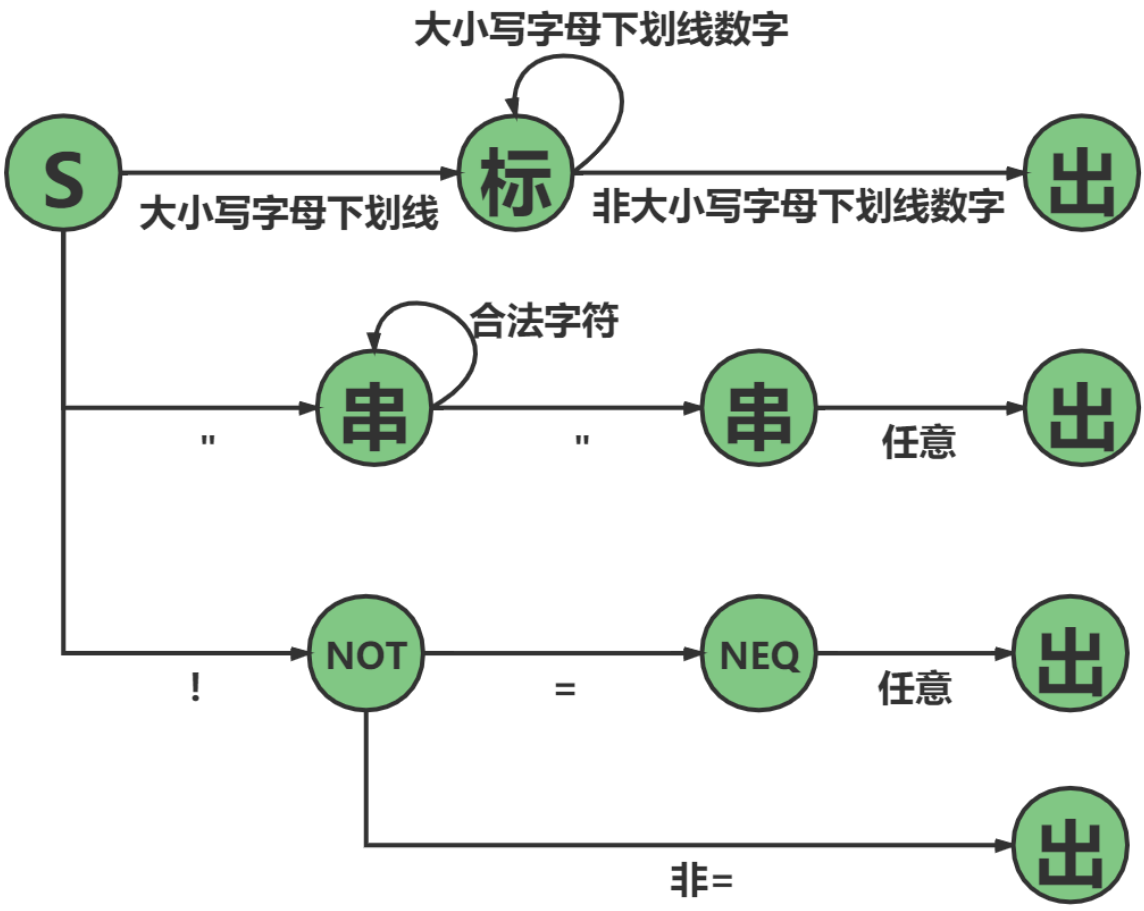
一、词法分析

（一）初步设计

词法分析一次读入一行源程序，通过一个字符流指针不断移动，读取字符并分析，读到行尾的时候读下一行，直到读到文件末尾结束。

词法分析过程与课本PL0源程序基本一致。先过滤空白符，在分析的过程中处理注释，不断读取字符，通过自动机辨识保留字、标识符、数字、一元分隔符、二元分隔符，将每个`token`的原始字符串和类别码返回，词法分析过程示例如下图所示：

通过对`getToken()`方法循环调用可实现对源程序的词法分析，最终把得到的全部`token`输出即可。



（二）实现与完善

词法分析的主要逻辑符合最初设计，初始化了`reversedTable`，`symbolTable`保存切词后的字符串到类别码的映射。将不同的子功能解耦合，实现了`_passNull()`专门处理空白符，实现了`_clearNotes`处理注释，字符合法性判断方法`_isValidChar()`，去字符方法`_getCh()`，回滚方法`_retract()`。

在`_getCh()`和`_retract()`中遇到`'\n'`要增减行号。

每次调用`getToken()`，通过调用`_getCh()`，`_retract()`处理字符流，直到读到一个`token`返回。

实现过程中，为了提高模块化程度，同时为了更好地为语法分析错误处理提供信息，对词法分析的实现进行了如下改进。

词法分析**一次性读入源文件全部内容**。最初设计中一边读文件一边词法分析，并且一次读一行的方式对行尾的处理比较复杂，而且回滚比较困难，处理行信息比较复杂。于是实现过程中改为一行一行不断读取（不读换行符），直到源程序全部读取完毕，在这个过程中在每行的行尾统一添加`'\n'`，解决不同系统中换行符不同带来的处理行号问题上的困难，读取完毕后，在整个源程序的字符流上进行词法分析，通过识别`'\n'`处理行号更新问题。

将`token`的进行了封装，词法分析得到的每个词均是一个`Token`类的实例，以尽可能多地记录该`token`在词法分析阶段能得到的信息，词法分析的`getToken()`方法会记录`token`的信息构造`Token`的实例，每次向外返回一个`token`。由此通过词法分析将源程序`char`流转化为`token`流。

```
public class Token {
    private String rawString; // token的原始字符串
    private TokenType tokenType; // token的类别码
    private int lineNo; // token的行号
    private int value; // 如果是<INTCON>记录其数值
    private int count; // 如果是<FormatString>记录其%d的个数
    private boolean valid = true; // 该token的有效性
}
```

单独处理注释，这个操作大大简化了`getToken()`中的判断逻辑。`_clearNotes`将注释处理掉，处理过程中在字符串中的`"//","/*"`要略过，并且删除注释过程中`'\n'`要保留，避免行信息丢失。

处理了词法分析阶段的错误。在`STRCON`的识别过程中调用`isValidChar()`方法进行合法性检查，处理词法分析阶段可能出现的错误。`isValidChar()`中判断`'\'`与`%`的合法性时要根据其后的字符来确定。

二、语法分析

（一）初步设计

语法分析使用递归子程序法。

首先对文法进行了处理，简化了部分文法的表达方式，并使用**扩充的`BNF`表达形式消除了左递归**，处理后的文法如下图所示，通过缩进一定程度上表达了不同成分的层次关系。

```
编译单元 CompUnit → {Decl} {FuncDef} MainFuncDef

    声明 Decl → ConstDecl | VarDecl // 不输出
        常量声明 ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';'
            基本类型 BType → 'int' // 不输出
            常数定义 ConstDef → Ident { '[' ConstExp ']' } '=' ConstInitVal
                常量初值 ConstInitVal → ConstExp | '{' [ ConstInitVal { ',' ConstInitVal } ] '}'
                    常量表达式 ConstExp → AddExp // 注：使用的Ident 必须是常量
        变量声明 VarDecl → BType VarDef { ',' VarDef } ';'
            变量定义 VarDef → Ident { '[' ConstExp ']' } ['=' InitVal]
                变量初值 InitVal → Exp | '{' [ InitVal { ',' InitVal } ] '}'

    主函数定义 MainFuncDef → 'int' 'main' '(' ')' Block
```

```
函数定义 FuncDef → FuncType Ident '(' [FuncFParams] ')' Block
    函数类型 FuncType → 'void' | 'int'
    函数形参表 FuncFParams → FuncFParam { ',' FuncFParam }
        函数形参 FuncFParam → BType Ident ['[' ']' { '[' ConstExp ']' }]
    语句块 Block → '{' { BlockItem } '}'
        语句块项 BlockItem → Decl | Stmt // 不输出
            语句 Stmt → LVal '=' Exp ';'
                      | [Exp] ';'
                      | Block
                      | 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
                      | 'while' '(' Cond ')' Stmt
                      | 'break' ';' | 'continue' ';'
                      | 'return' [Exp] ';'
                      | LVal = 'getint''('')'';'
                      | 'printf''('FormatString{,Exp}')'';'
                格式化字符串 <FormatChar> → %d
                      <NormalChar> → 十进制编码为32(' '),33('!'),40-126的ASCII字符，'\\'（编码92）出现当且仅当为'\n'
                      <Char> → <FormatChar> | <NormalChar>
                      <FormatString> → '"'{<Char>}'"' // 不输出
                左值表达式 LVal → Ident {'[' Exp ']'}
                条件表达式 Cond → LOrExp
                    逻辑或表达式 LOrExp → LAndExp { '||' LAndExp }
                        逻辑与表达式 LAndExp → EqExp { '&&' EqExp }
                            相等性表达式 EqExp → RelExp { ('==' | '!=') RelExp }
                                关系表达式 RelExp → AddExp { ('<' | '>' | '<=' | '>=') AddExp }
```

```
                表达式 Exp → AddExp // 注：SysY 表达式是int 型表达式
                    加减表达式 AddExp → MulExp { ('+' | '-') MulExp }
                        乘除模表达式 MulExp → UnaryExp { ('*' | '/' | '%') UnaryExp }
                            一元表达式 UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp
                                标识符 Ident → ('a'|...|'z'|'A'|...|'Z'|'_') {'a'|...|'z'|'A'|...|'Z'|'_'|'0'|...|'9'} // 不输出
                                基本表达式 PrimaryExp → '(' Exp ')' | LVal | Number
                                    数值 Number → IntConst
                                        数值常量 IntConst → '0' | ('1'|...|'9') {'0'|'1'|...|'9'} // 不输出
                                函数实参表 FuncRParams → Exp { ',' Exp }
                                单目运算符 UnaryOp → '+' | '-' | '!' // 注：'!'仅出现在条件表达式中

// 表达式类的左递归版，左递归保证了自左向右运算的优先级
Cond → LOrExp
Exp → AddExp

    LOrExp → LAndExp | LOrExp '||' LAndExp
        LAndExp → EqExp | LAndExp '&&' EqExp
            EqExp → RelExp | EqExp ('==' | '!=') RelExp
                RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
                    AddExp → MulExp | AddExp ('+' | '-') MulExp
                        MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
                            UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp
                                PrimaryExp → '(' Exp ')' | LVal | Number
                                    UnaryOp → '+' | '-' | '!'
```

表达式类的许多语法成分是左递归描述的，虽然文法已经改成非左递归文法，但是在输出语法成分时仍要按照左递归的描述进行输出，例如`LAndExp -> EqExp | LAndExp '&&' EqExp`改写为`LAndExp -> EqExp { '&&' EqExp }`，但是解析完每一个`EqExp`后仍要向输出符号流中追加一个`<LAndExp>`，与左递归文法保持一致。

最初的设计中，词法分析程序是语法分析程序的子程序，语法分析程序调用词法分析程序的`getToken()`方法获得一个`token`，并为每一个语法成分写一个分析方法，与课本上的方法类似，进入每个成分的分析方法中时已经预先读入了一个`token`，通过预读确定下一个语法成分，并调用其分析程序。

（二）实现与完善

在具体的实现中为了让语法成分之间的界限更加清晰，也为了避免文件尾读取出现越界异常导致最后一个语法成分无法读入，**进入每一个语法成分的分析程序时并没有提前读入一个，离开一个语法成分时也不会向后读取一个，如果有向后读取，也要回滚。**

实现的过程中**建立了**`AST`。每个语法成分均单独成为一个类，并继承了`Node`，都实现了`analyze()`方法。在每一个类的`analyze()`方法中，**识别到一个子成分时，创建一个该成分的实例，并即将其这个成分实例作为该结点的一个子节点**，将这个成分的结构信息以树的形式保存下来，并递归调用这个子节点的`analyze()`方法，对部分叶子结点`analyze()`过程中要记录下这个节点需要保存的信息，如`INTCON`的数值。每个节点同时还要记录**分支信息**，记录使用了文法的哪一条进行的推导，便于树的遍历访问。以语法成分`Stmt`为例：

```
public class Stmt extends Node {

    private Type type; // 记录分支信息
```

```
    private _Assign _assign;
    private Exp exp;
    private Block block;
    private _If _if;
    private _While _while;
    private _Break _break;
    private _Continue _continue;
    private _Return _return;
    private _Getint _getint;
    private _Printf _printf;

    private Token lValToken;

    private enum Type {
        ASSIGN,
        EXP,NULL, // [Exp] ';'
        BLOCK,
        IF,
        WHILE,
        BREAK,
        CONTINUE,
        RETURN,
        GETINT,
        PRINTF
    }

    public void analyze() throws EOF {
        Grammar.nextToken();
        if (Grammar.token.getTokenType() == TokenType.LBRACE) {
            type = Type.BLOCK;
            Grammar.retract();
            block = new Block();
            block.analyze();
        }
        else if (Grammar.token.getTokenType() == TokenType.IFTK) {
            // 同理
        }
        else if (Grammar.token.getTokenType() == TokenType.WHILETK) {
            // 同理
        }
        else if (Grammar.token.getTokenType() == TokenType.BREAKTK) {
            // 同理
        }
        else if (Grammar.token.getTokenType() == TokenType.CONTINUETK) {
            // 同理
        }
        else if (Grammar.token.getTokenType() == TokenType.RETURNTK) {
            // 同理
        }
        else if (Grammar.token.getTokenType() == TokenType.PRINTFTK) {
            // 同理
        }
        else {
            // 向后预读 判断分支
        }
        OutputList.addToList(GrammarType.Stmt);
    }
}
```

实现过程中，并没有将词法分析作为语法分析的子程序，而是将**词法分析与语法分析解耦合**，将词法分析得到的`token`**流传递到语法分析**程序中，于是只需要在语法分析成分中维护一个`token`流指针即可，`token`的读取与回滚变得非常简便，与词法分析中原理类似。

```
public static void nextToken() throws EOF {
    if (pointer >= tokens.size()) {
        throw new EOF();
    }
    token = tokens.get(pointer++);
    OutputList.addToList(token); // 维护输出流
}

public static void retract() {
    pointer -= 1;
    if (pointer-1 >= 0) {
        token = tokens.get(pointer - 1);
    }
    else { token = null; }
    while (OutputList.pop() instanceof GrammarType);// 维护输出流
}
```

在比较复杂的分支判断中，例如在`Stmt`中，为了确定下一个分支需要向后读取多个语法成分，即对应不定数量个`token`，**为了实现不定数量**`token`**的回滚，实现了**`mark()`**和**`restore()`。

```
public static void mark() {
    pointer_mark = pointer;
    token_mark = token;
}

public static void restore() {
    while (pointer != pointer_mark) {
        retract();
    }
}
```

在实现的过程中，由于有大量的读取与回滚，而且需要同时输出词法分析与语法分析的结果，对**输出流的维护也成为一个难点**。当回滚`token`的时候，含有这个`token`的语法成分也需要回滚，于是在回滚时采取了一下策略：**回滚直到回滚到一个**`token`，在这之前的全部语法成分全部也需要回滚。

完成了语法相关的错误处理。如缺少`;`，缺少`)`，缺少`]`，处理后跳过这个缺失的符号继续分析。

由于考虑了语法错误的处理，由于部分地方可能出现缺少`;`等错误，所以许多地方判断某一个语法成分之后的下一个语法成分是否存在时，必须要读这个下一个成分的**FIRST集**，而不能直接判断是否是下一个成分的下一个成分的FIRST集。

三、错误处理

（一）初步设计

由于词法和语法错误再词法和语法处理阶段已经处理，这里的错误处理是语义错误，最终输出错误的时候只需要按照行号进行排序。

初步设计是不依赖符号表，而是依赖语法树，通过不同节点之间的`get()`方法实现自下而上的属性传递。同时设计了不同错误的添加方法，以及错误列表的维护方法。

（二）实现与完善

错误类的实现基本符合设计，实现了错误类的创建，错误的添加，错误列表的维护，将错误处理与其余部分解耦合。

```
public class MyError {

    private static ArrayList<MyError> myErrors = new ArrayList<>();
    private Token token; // 发生错误的token
    private int lineNo; // 错误的行号
    private ErrorType errorType; // 错误的类别码
    private String msg; // 错误的详细信息

    public MyError();
    public MyError(Token token, ErrorType errorType);
    // 设置错误信息
    public void setToken(Token token);
    public void setLineNo(int lineNo);
    public void setErrorType(ErrorType errorType);
    public void setMsg(String msg);
    public String toString();
    // 错误列表维护方法
    public static void addErrors(MyError myError);
    public static void sort();
    public static String errors2String(boolean log);// log表示错误详细信息，方便程序
调试
    // 不同错误的添加方法
    public static void add_lack_rbrack();
    public static void add_continue_no_loop(Token token) ;
    public static void add_break_no_loop(Token token);
    public static void add_lack_semi();
    public static void add_lack_rparent();
    public static void add_printf_num_mismatch(Token printf_token,int need,int
give);
    public static void add_reDef(Token token);
    public static void add_unDef(Token token);
    public static void add_const_varied(Token token);
    public static void add_params_num_mismatch(Token token,int need,int give);
    public static void add_params_type_mismatch(Token token, DataType need,
DataType give);
    public static void add_voidFunc_return(Token token);
    public static void add_func_noReturn(Token token);
}
```

`AST`很好地保存了一个程序的静态结构，但并没有保存动态建树的信息，只使用`AST`并不是非常方便，例如对变量的引用分散在语法树的许多地方，使用`AST`并不方便。于是采取`AST`**+符号表**的策略进行错误处理，为了将语义错误与语法分析解耦合，采取**再次遍历语法树一边建立符号表一边查找错误的实现方法，查找错误既使用了`AST`，也使用了符号表。**

四、符号表管理

（一）初步设计

采取了最简单的**栈式符号表**，遍历语法树时使用全局变量`dim`记录当前的层次信息，每进入一个层次`dim++`，并不断将变量等压栈，退出一个层次时`dim--`，并把这个层次内的符号表项全部弹栈。

符号表项`TableItem`设计如下：

```
public class TableItem {
    private String identName; // 标识符名
    private IdentType identType; // 标识符类型
    private DataType dataType; // 数据类型
    private int array_dim_1; // 数组第一维度
    private int array_dim_2; // 数组第二维度
    private int dim; // 层次信息
    private ArrayList<TableItem> paras = new ArrayList<>(); // 引用函数形参的符号表
项
}
```

其中`IdentType`是标识符类型，`DataType`是数据类型，如下：

```
public enum IdentType {
    VAR,
    CONST,
    PARA,// 函数形参
    FUNC
}
public enum DataType {
    INT, // for var, const, func: int
    VOID, // for func: void
    INT_ARRAY_1, // for var, const: int[]
    INT_ARRAY_2 // for var, const: int[][x]
}
```

栈式符号表设计如下：

```
public class StackTable {
    public static Stack<TableItem> tables = new Stack<>();
    public static void push(TableItem tableItem);
    public static void pop();
    // 检查是否重定义
    public static boolean reDef(String name,boolean isFunc);
    // 找到name对应的符号表项
    public static TableItem def(String name,boolean isFunc);
    // 清除当前dim的符号表项
    public static void clear();
}
```

语义错误的处理过程是根据`AST`节点记录的分支信息辅助遍历`AST`，建立符号表。对于**重定义、未定义**等错误使用**符号表**处理非常方便，对于`return`相关错误使用语法树在顶层通过`get()`方法垂直获取`AST`下层的成分信息进行错误检查更加方便。

（二）实现与完善

在后面的实现中，为了更好地生成中间代码，通过符号表对变量进行了重命名，解决了变量的作用域覆盖问题。在`TableItem`中添加了`rename`属性。

符号表连接编译器前后端部分之一，最初的栈式符号表在`AST`遍历完毕之后所有的信息已经弹出，无法为`MIPS`翻译程序使用，而且栈式符号表查询效率低，其中的信息大部分在生成中间代码之后已经不再有用，于是在实现过程中，又设计实现了"汇编符号表"，主要为变量分配空间，维护程序中所有的字符串常量等。

汇编符号表设计了符号表项`MipsTableItem`，函数符号表`MipsFuncTable`，全局符号表`MipsGlobalTable`，由于已经对所有变量进行了重命名，这个符号表通过`HashMap<String,MipsTableItem>`的结构对其中所包含的变量进行管理，符号表项`MipsTableItem`设计如下：

```
public class MipsTableItem {
    private String itemName; // 变量名称 也可以是一个字符串的"变量名"
    private Type type; // 标记是函数还是变量还是一个字符串
    private String strMark; // 这个字符串的标记名
    private int offset; // 这个变量放在内存距离这个区域的起始的偏移
    private String constArrayMark; // 常量数组在data段的标记名
    // 数组长度信息 仅对于数组变量与数组常量 不包括数组形参
    private int len;
    public enum Type {
        FUN,
        VAR,
        PARA,
        STR,
        VAR_ARRAY,
        CONST_ARRAY,
        PARA_ARRAY,
    }
}
```

全局符号表`MipsGlobalTable`设计结构如下：

```
public class MipsGlobalTable {
    // 静态量 单例模式
    private static String strName = "String_";
    private static int autoStrName = 0; // 自动字符串命名编号

    private static HashMap<String,MipsTableItem> globalStrings = new HashMap<>
(); // 整个程序中使用到的所有字符串常量
    private static HashMap<String,ArrayList<Integer>> globalConstArrays = new
HashMap<>(); // 整个程序中所有的常数数组
    private static HashMap<String,MipsFuncTable> funs = new HashMap<>(); // 所有
的函数
    private static HashMap<String,MipsTableItem> globalVars = new HashMap<>();
// 所有的全局变量
    private static int offset = 0; // 全局变量所占用的存储空间
    private static int count = 0; // 全局变量的个数
    private static ArrayList<MidCode> midCodes = new ArrayList<>(); // 存储全局的中
间代码
    // 主要方法
    public static void addGlobalString(String str);
```

```
    public static void addFun(String funName);
    public static void addGlobalVar(String varName);
    public static void addGlobalArrayVar(String varName,int len);
    public static void addGlobalArrayConst(String constName,int len);
    public static void addMidCode(MidCode midCode);
}
```

函数符号表`MipsFuncTable`的设计如下：

```
public class MipsFuncTable {
    private HashMap<String,MipsTableItem> vars = new HashMap<>();
    private HashMap<String,MipsTableItem> paras = new HashMap<>();
    private int offset = 0; // 分配局部变量与形参的内存空间
    private int count = 0; // 记录变量与形参一共的个数
    private ArrayList<MidCode> midCodes = new ArrayList<>(); // 存储函数的中间代码
    // 主要方法
    public void addVar(String varName);
    public void addPara(String paraName);
    public void addArrayVar(String varName,int len);
    public void addArrayConst(String constName,int len);
    public void addArrayPara(String paraName);
}
```

向符号表中添加表项为了进行运行时的空间管理，这一部分将在目标代码生成阶段展开。

五、中间代码生成

（一）初步设计

对中间代码的初步设计，我的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。

（二）实现与完善

实现上，中间代码是在递归访问`AST`的同时一边建立栈式符号表一边根据符号表信息创建的。

为了对中间代码进行更好的分类管理，`MidCode`类管理所有中间代码的公共属性和公共方法，各中间代码类继承`MidCode`类。中间代码可以有不同的分类方法，我使用不同种类接口对中间代码进行分类，接口中定义了相应大类的中间代码应当实现的方法。我设计了如下的抽象接口对中间代码进行管理：

序号	接口名称	接口作用
1	**ABSTRACT_DECLARE**	管理所有声明语句
2	**ABSTRACT_DEF**	管理所有定义语句
3	**ABSTRACT_JUMP**	管理所有控制流语句
4	**ABSTRACT_LABEL**	管理所有标签语句
5	**ABSTRACT_OUTPUT**	管理所有输出语句
6	**ABSTRACT_USE**	管理所有使用语句

访问某一类中间代码时只需要按照该类的接口进行访问，顶层不需要关心具体是哪一种中间代码，底层对象对上层透明，减少了大量的判断语句，从根本上杜绝了判断语句遗漏部分条件而出现错误的现象。
我的中间代码设计如下：

序号	种类	域1	域2	域3	语义	实现接口
1	**ADD**	和	加数1	加数2	加法	DEF、USE
2	**SUB**	差	被减数	减数	减法	DEF、USE
3	**MULT**	积	乘数1	乘数2	乘法	DEF、USE
4	**DIV**	商	被除数	除数	除法	DEF、USE
5	**MOD**	余	被除数	除数	模	DEF、USE
6	**EQL**	结果	操作数1	操作数2	相等置1	DEF、USE
7	**GEQ**	结果	操作数1	操作数2	大于等于置1	DEF、USE
8	**GRE**	结果	操作数1	操作数2	大于置1	DEF、USE
9	**LEQ**	结果	操作数1	操作数2	小于等于置1	DEF、USE
10	**LSS**	结果	操作数1	操作数2	小于置1	DEF、USE
11	**NEQ**	结果	操作数1	操作数2	不等于置1	DEF、USE
12	**NEG**	结果	操作数		取相反数	DEF、USE
13	**NOT**	结果	操作数		取逻辑反	DEF、USE
14	**ASSIGN**	结果	操作数		赋值	DEF、USE
15	**ARRAY_ADDR_INIT**	数组指针			为数组指针赋地址初值	DEF
16	**CONST_ARRAY_ADDR_INIT**	数组指针			为常量数组指针赋地址初值	DEF
17	**BEGIN_WHILE**				while的开始标签	LABEL
18	**END_WHILE**				while的结束标签	LABEL
19	**IF**				if的开始标签	LABEL
20	**ELSE**				else的标签	LABEL
21	**END_IF**				if的结束标签	LABEL
22	**LABEL**				自定义标签	LABEL

序号	种类	域1	域2	域3	语义	实现接口
23	**SUPPLEMENT_LABEL**				用于基本块划分的辅助标签	LABEL
24	**BEZ**		条件	指定标签	等于0跳转到指定标签	JUMP、USE
25	**BREAK**			指定标签	break	JUMP
26	**CONTINUE**			指定标签	continue	JUMP
27	**GOTO**			指定标签	无条件跳转	JUMP
28	**RETURN**		返回值（可无）		函数返回	JUMP、USE
29	**PRINT_INT**		待打印操作数		打印操作数	OUTPUT、USE
30	**PRINT_STR**		待打印字符串		打印字符串	OUTPUT
31	**CONST**	常量名			声明常量	DECLARE
32	**CONST_ARRAY**	数组常量名	维度1	维度2	声明数组常量	DECLARE
33	**PARA_DECLARE**	形参名			声明形参	DECLARE
34	**PARA_ARRAY_DECLARE**	数组形参名	维度1	维度2	声明数组形参	DECLARE
35	**VAR**	变量名			声明变量	DECLARE
36	**VAR_ARRAY**	数组变量名	维度1	维度2	声明数组变量	DECLARE
37	**CALL_INT**		函数名		调用int函数准备工作	

序号	种类	域1	域2	域3	语义	实现接口
38	**CALL_VOID**		函数名		调用void函数准备工作	
39	**END_CALL_INT**	目标变量	函数名		调用int函数，将返回值赋给目标变量	
40	**END_CALL_VOID**		函数名		调用void函数	
41	**PUSH**		操作数		将操作数压栈	USE
42	**LOAD_ARRAY**	dst	src	index	`dst = src[index]`	DEF、USE
43	**STORE_ARRAY**	dst	index	src	`dst[index] = src`	USE
44	**FUN**		函数名		创建函数	
45	**PARA**		形参名		从栈取形参的值	DEF
46	**PARA_ARRAY**		形参名		从栈取数组形参的（地址）值	DEF
47	**GETINT**	dst			`dst = getint()`	DEF
48	**START_PROGRAM**				程序入口	
49	**END_PROGRAM**				程序出口	

每个中间代码都有高度的独立性，每个中间代码也记录了其所在的函数信息，不仅简化了每一条中间代码翻译成`mips`的复杂度，也为翻译与优化提供了足够细粒度的信息，优化阶段对中间代码的添加与删除也变得更加容易，减少了出现风险的可能，中间代码的可扩展性好，在代码优化中能够得到体现。

本部分的另一个难点是**数组**。在我的设计中，数组是按照**指针常量**的方式处理的，每声明一个数组便创建同名的指针变量，并将数组的首地址赋值给这个指针变量，这样的好处是简化了数组传参，将数组指针当做一个普通变量即可。语义分析时，数组的模板被记录下来，在生成中间代码时，已经消解了数组维度的概念，在中间代码中只留下基地址和偏移两个概念，减轻了目标代码生成的压力。

本部分的最后一个难点是**短路求值**。此部分有一定的技巧性，我采取了控制流法进行了实现，即在逻辑上将表达式拆解成一层一层的`if`，将`&&`视作`if`的嵌套，将`||`视作`if else`，在具体的实现上通过在适当的位置添加`JUMP`语句和`LABEL`实现。

六、目标代码生成

（一）初步设计

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。

（二）实现与完善

在实现上，每个中间代码均继承`MidCode`类，并实现`createMips()`方法，在`createMips()`中完成指令选择。

对于打印字符串，我的实现是通过`%d`将字符串分裂成一段一段的，分成整数打印和字符串打印两部分，将待打印的字符串统一编号放入`.data`区段，所有的代码均放入`.text`区段。

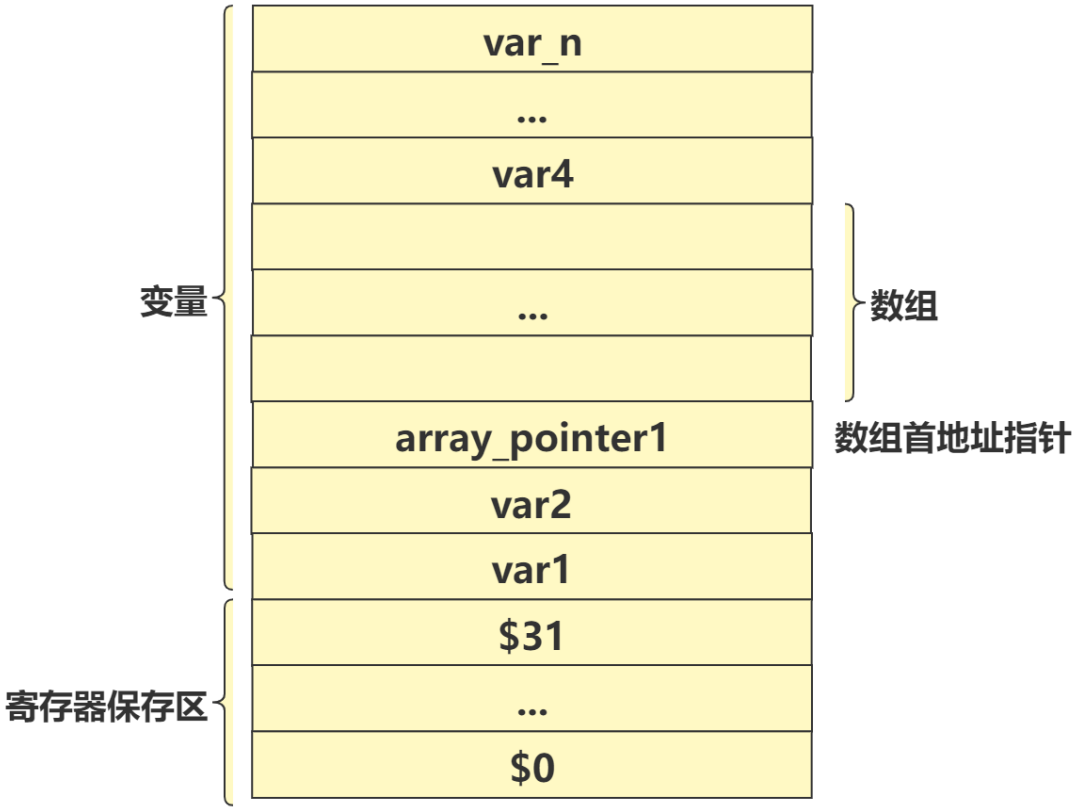
运行时空间分配与符号表相关，目标代码生成的主体逻辑如下，先通过扫描中间代码，通过中间代码中的声明语句为不同种类的变量分配空间，全局变量放在`$gp`，局部变量放在`$sp`。

```
    // 读中间代码 生成符号表
    public static void createMips(ArrayList<MidCode> midCodes) {
        for (MidCode midCode : midCodes) {
            // 先处理全局表
            if (midCode.getBelong().equals("&global")) {
                MipsGlobalTable.addMidCode(midCode);
                if (midCode instanceof VAR) {
                    MipsGlobalTable.addGlobalVar(((VAR) midCode).getVarName());
                }
                else if (midCode instanceof VAR_ARRAY) {
                    MipsGlobalTable.addGlobalArrayVar(
                            ((VAR_ARRAY) midCode).getVarName(),((VAR_ARRAY)
midCode).getLength()
                    );
                }
                else if (midCode instanceof CONST_ARRAY) {
                    MipsGlobalTable.addGlobalArrayConst(
                            ((CONST_ARRAY) midCode).getConstName(),
((CONST_ARRAY) midCode).getLength()
                    );
                }
                else if (midCode instanceof FUN) {
                    MipsGlobalTable.addFun(((FUN) midCode).getFunName());
                }
            }
        }
        // 再处理函数内部
        for (MidCode midCode : midCodes) {
            if (!midCode.getBelong().equals("&global")) {
                String belongFunc = midCode.getBelong();
                MipsFuncTable funcTable =
MipsGlobalTable.getFuncTable(belongFunc);
                // 添加这个函数管理的中间代码
                funcTable.addMidCode(midCode);
                // 形参拆分成多个变量后可以共享一个位置
                if (midCode instanceof PARA_DECLARE) {
                    funcTable.addPara(((PARA_DECLARE) midCode).getParaName());
                }
                else if (midCode instanceof VAR){
                    funcTable.addVar(((VAR) midCode).getVarName());
                }
                else if (midCode instanceof PARA_ARRAY_DECLARE) {
                    funcTable.addArrayPara(((PARA_ARRAY_DECLARE)
midCode).getParaName());
                }
                else if (midCode instanceof VAR_ARRAY) {
```

```
                    funcTable.addArrayVar(((VAR_ARRAY) midCode).getVarName(),
((VAR_ARRAY) midCode).getLength());
                }
                else if (midCode instanceof CONST_ARRAY) {
                    funcTable.addArrayConst(((CONST_ARRAY)
midCode).getConstName(),((CONST_ARRAY) midCode).getLength());
                }
                else if (midCode instanceof PRINT_STR) {
                    // 全局保存一共需要打印的字符串
                    MipsGlobalTable.addGlobalString(((PRINT_STR)
midCode).getDstString());
                }
            }
        }
        mipsCode += ".data\r\n";
        HashMap<String, MipsTableItem> globalStrings =
MipsGlobalTable.getGlobalStrings();
        for (String key : globalStrings.keySet()) {
            mipsCode += String.format("%s: .asciiz %s\r\n",
                    globalStrings.get(key).getStrMark(),
                    globalStrings.get(key).getItemName());
        }
        mipsCode += "\r\n";
        mipsCode += ".text\r\n";

        for (MidCode midCode : midCodes) {
            mipsCode += String.format("# %s",midCode); // mips中注释 中间代码
            mipsCode += midCode.createMips();
            mipsCode += "\r\n";
        }
    }
```

运行时的空间分配如下图所示，其中数组是按照指针来存储的。全局变量空间管理类似，只是不再需要寄存器保存区。



运行时空间管理

在后期的代码优化中，对指令选择进行了优化，所以对每条中间代码实现 `createMipsOpt()` 方法。

七、代码优化

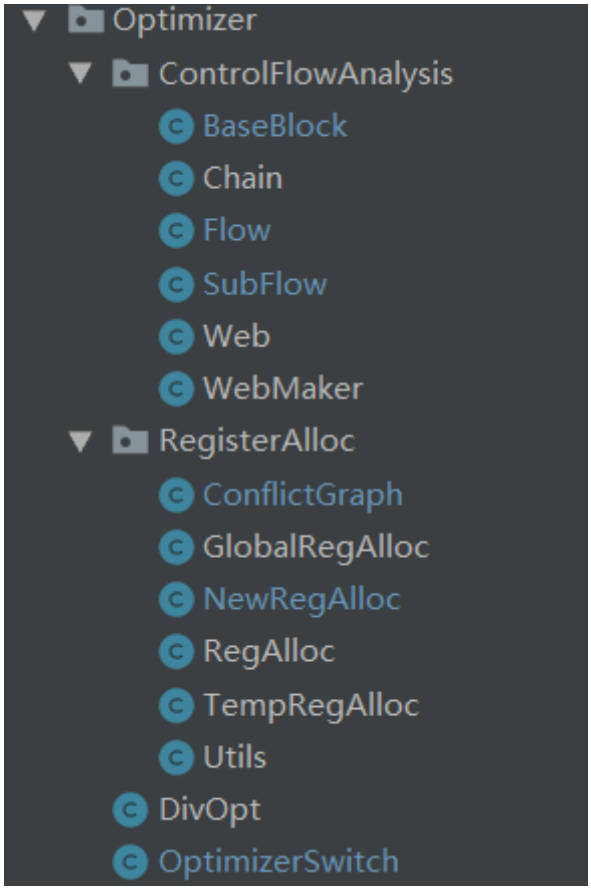
（一）初步设计

代码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分析，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.

（二）实现与完善

实现过程基本沿用了初步的设计构思，并对细节进行了补充。

> **优化部分架构设计**



优化部分单独成为编译器的中端，与其余部分解耦，保证优化部分开启与关闭不影响程序其余部分的正确性。

`ControlFlowAnalysis`中主要进行数据流分析，其中`BaseBlock`是基本块，`Chain`是使用定义链，`Web`和`WebMaker`用于构建使用定义网，`Flow`和`SubFlow`用来管理数据流，`Flow`管理全局的数据流信息并定义了全局层面的数据流分析方法，`SubFlow`管理函数的数据流信息并定义了函数中的数据流分析方法。

`RegisterAlloc`类用来管理寄存器，`Utils`中定义了寄存器管理中常用的工具方法，`ConflictGraph`用于构建冲突图并进行全局寄存器分配，`NewRegAlloc`用来管理寄存器池，并进行寄存器的申请、回收、保存、回写等工作。

> **数据流分析**

- 基本块划分与数据流图的构建

数据流分析对于全局优化有着基础性的意义，而数据流分析的第一步也是关键一步便是数据流图的构建。我的数据流图构建分为基本块划分与数据流图构建部分。

在架构设计层面，由于函数调用要保存现场，所以我认为函数调用不破坏基本块，每个函数都是一个相对独立且封闭的部分，应当拥有一个独立的数据流。所以我用两个类进行数据流的管理，`Flow`类和`SubFlow`类，`Flow`管理全局数据流并定义了全局数据流的方法，并对`SubFlow`进行管理，`SubFlow`管理函数内的数据流并定义了函数内数据流的方法，全局也作为一个函数名为`&global`的假函数。基本块的划分与数据流图也均对每个函数单独进行，所以先将中间代码根据其所处的域分配到相应函数的`SubFlow`.

基本块划分的难点是在顺序排列的中间代码中找到基本块的入口与出口，同时我的基本块划分是基于标签的划分方法，在中间代码生成的过程中生成标签，每遇到一个标签便切割一次基本块，基本块的名称即标签的名称，由于跳转语句也会使用标签，因此这样做连接基本块会变得容易。基本块的切分方法如下：

1、生成分支/跳转语句时在其后面插入一条`LABEL`，分支/跳转语句的一个`dst`已经是语句中的`LABEL`（`RETURN`设置为`"#Exit"`），如果生成的是分支语句，则将分支语句的另一个`dst`设定为其后面的`LABEL`。

2、对函数的中间代码进行扫描。如果遇到`ABSTRACT_JUMP`语句，则找到部分基本块的终点，将基本块的`dst`分别设定为`ABSTRACT_JUMP`的`dst`；如果遇到`ABSTRACT_LABEL`语句，则找到新的基本块的起点，如果前面的基本块不以`ABSTRACT_JUMP`结束，则将前面的基本块的`dst`设定为新的基本块的名称。

基本块划分过程中已经对每个基本块的`dst`信息进行了记录，所以构建数据流图只需要遍历基本块，将各个基本块通过前驱与后继关系关联起来即可。

对每个函数的数据流图中补充一个`#Exit`的结束基本块，所有`return`结尾的基本块均指向`#Exit`基本块，至此每个函数的流图均为单入口单出口。函数声明语句由于处于全局作用域中被分到了`&global`函数中，于是再将函数声明语句从`&global`中转移到各个函数的入口基本块处，至此将各个基本块顺序拼接即可得到全部的中间代码。方便后面的优化对基本块中间代码进行增删后能够将合成新的中间代码。

- 活跃变量分析

我的编译器中实现了以语句为颗粒度的活跃变量分析。按照课本上的算法可以以基本块为单位计算出在每个基本块的`in`和`out`，即基本块入口和出口处的活跃变量。接下来我对基本块内进行遍历得到了基本块内的活跃变量信息，最终得到了每条中间代码处的活跃变量，算法如下：

对于基本块`B`，初始化集合`willUse = out[B]`，**逆序遍历**基本块`B`中所有的中间代码，对于每条中间代码`m`，如过`m`是`ABSTRACT_DEF`并且`willUse`中有`DEF`语句产生的`def`，则从`willUse`中移除`def`，然后将`willUse`加入到`m.willUse`中，如果`m`还是`ABSTRACT_USE`，则将其所有的`use`加入到`willUse`中。

再初始化集合`alreadyDef = in[B]`，**顺序遍历**基本块`B`中所有的中间代码，对于每条中间代码`m`，将`alreadyDef`加入到`m.alreadyDef`中，如果`m`是`ABSTRACT_DEF`，则将`m`产生的定义`def`加入到`alreadyDef`中。**语句`m`处的活跃变量`m.active`则为`m.alreadyDef`和`m.willUse`的交集。**

在上述算法计算`willUse`时体现出了定义语句的`kill`作用，原因是因为一个变量在基本块内可能被赋值多次，**如果在基本块内实现SSA**，即在一个基本块内，一个变量至多被赋值一次，则可以简化掉定义语句的`kill`，这一点是容易做到的，需要通过下文的到达定义分析。

- 到达定义分析与du关系

我的编译器中实现了以语句为颗粒度的到达定义分析，最终得到了**du链**和**ud链**，即`ABSTRACT_DEF`语句以及所有其可达的`ABSTRACT_USE`语句，和`ABSTRACT_USE`语句以及所有可能到达其的`ABSTRACT_DEF`语句。按照课本上的算法可以得到基本块间的到达定义数据流和所有的`du`链，对基本块内的代码进行遍历对`du`链进行基本块内的**延拓**，在得到`du`链的同时记录`ud`链，并将`ud`链的信息保存在每条中间代码中。

通过`du`链可以得到网，当同一个变量的多个`du`链拥有公共的使用点时可以合并为一个网。这个问题可以**转变为求连通分量的图问题。**给定一个变量的所有`du`链，每条`du`链均视为图的节点，如果两个`du`链有公共的使用点，则认为两个`du`节点之间存在一条边，通过`BFS`可以求得无向图的所有连通分量，即可得到变量的所有网。

一个变量的不同的网可以视作不同的变量，根据这一思想，**我对每个网得到的变量进行了重命名**，如此每个基本块内实现了SSA。重命名之后可能会存在没有被重命名的变量，说明这些变量的使用没有可以到达的定义，或是变量没有赋值直接使用，或者变量所在的语句是不可达语句。对于前者，因为变量没有赋值直接使用，所以取得任何值都是合法的，对于后者，由于语句不可达，将语句直接删除即可。

重命名之后，由于产生的新的变量，所以对于新的变量要在重命名之前该变量的位置补充变量声明语句，让`MIPS`翻译器为其分配空间，如果重命名的是形参，`PARA_DECLARE`中的形参名称要与`PARA`的形参名称一致，对于形参的其他重命名，这些重命名的使用无法使用到函数调用传进来的值，这些重命名严格来讲不属于形参，所以在其使用处补充一条`VAR`声明即可。

进行了重命名之后再进行活跃变量分析，能够使得活跃变量分析的结果更加准确。

中间代码优化

- 常量替换

在生成中间代码的阶段，对非数组的常量进行替换，因为常量都是编译可求值的，所以在常量定义处将常量的值记录下来存入符号表，在生成中间代码时，访问符号表将常量用其值替换。

- 局部复制传播与常量合并

局部的复制传播与常量合并是结合在一起的，首先先进行常量合并，将尽可能多的语句转化为赋值语句。对于运算类型语句，如果参与运算的操作数均为常数，则编译期可求值，可以转化为赋值语句。如果其中有一个操作数是常数，还可以进行如下的合并操作：

```
a = b + c && c =  0  ==> a = b;
a = b - c && c =  0  ==> a = b;
a = b * c && c =  0  ==> a = 0;
a = b * c && c =  1  ==> a = b;
a = b / c && c =  1  ==> a = b;
a = b / c && b =  0  ==> a = 0;
a = b % c && c =  1  ==> a = 0;
a = b % c && c = -1  ==> a = 0;
a = b % c && b =  0  ==> a = 0;
```

局部的赋值传播算法如下，对基本块顺序遍历，如果当前语句是`a = b`，则将其后面`a`被`kill`之前所有对`a`的使用替换成`b`，遍历结束之后，再对所有的代码进行一次常量合并。

- 全局常量传播

全局的常量传播基于`ud`链，对于中间代码`m`中的使用`use`，如果只有一条`DEF`可以到达`use`，并且`DEF`是`ASSIGN`语句且赋值的右值是常数，则可以将常数替换`m`中的`use`，实现全局的常量传播。

通过实验发现，先进行一次局部复制传播与常量合并，再进行一次全局常量传播，最后再进行一次局部复制传播与常量合并，就已经可以达到非常不错的效果。

- 局部死代码删除

局部死代码删除基于活跃变量分析，以基本块为单位进行，如果一个变量在其定义之后没有使用，即变量定义之后在基本块内没有使用，而且不在基本块的`out`中，则可以删除其定义语句。进行完一遍之后，再做活跃变量分析，再进行局部死代码删除，如此循环迭代，直至中间代码不再发生变化。

对代码的删除有时并不能直接删除，因为有许多代码有副作用，如函数调用，输出输出，形参接收等，于是我对删除的代码设置删除状态，这样在翻译`MIPS`的时候如果中间代码处于删除状态时，则执行其删除后行为。

- 全局死代码删除

在局部死代码删除中，发现对于如下情况：

```
while (n < NUM) {
    d = d * d % 1000;
    n = n + 1;
}
```

当这段代码之后变量d如果没有被输出，没有参与函数传参，没有作为控制流判断信息等，`d = d * d % 1000;`是可以删除的，但是根据局部死代码删除策略，无论迭代多少次，都是无法删除的，因为语句自己使用了自己的定义。所以这里提出了**全局死代码删除**策略。

全局死代码删除基于`du`链和`ud`链条。当一个变量没有在副作用语句中使用时，则其定义点的语句可以删除，如果一个变量在副作用语句中使用时，则所有可以到达这个使用的定义语句也标记为副作用语句。

原子的副作用语句包括输入输出、函数调用、对全局变量的赋值、对数组元素的改变、控制流语句，具体展开如下：

- `ABSTRACT_DEF`且定义的变量是全局变量
- `STORE_ARRAY`
- `PRINT_INT`
- `PRINT_STR`
- `GETINT`
- `CALL_INT`
- `CALL_VOID`
- `PUSH`
- `END_CALL_INT`
- `END_CALL_VOID`
- `ABSTRACT_JUMP`

算法如下：

初始化集合`Side`为空，遍历中间代码找到所有的原子的副作用语句，将这些副作用语句加入`Side`，重复进行如下操作，直到`Side`不再变化：遍历`Side`中语句，如果语句是`ABSTRACT_USE`，将其所有的`use`根据`ud`链，找到能够到达这个使用的定义语句，加入`Side`中。

最终，将不在上述`Side`集合中的`ABSTRACT_DEF`语句删去即可。

> 目标代码优化

目标代码优化最重要的是寄存器分配。寄存器分配是影响全局的一个重要因素，寄存器分配得好有时可以将其他优化的性能更好地释放出来，达到四两拨千斤的效果。我在这一部分实现了课本上的主要方法之后，又进行了诸多**精细化处理**，尽可能将寄存器分配的效率释放出来。

整体架构上，我实现了如下接口，具体的实现将在临时寄存器一节展开。

```
// 为name申请一个读寄存器
Register applyReadReg(String name, MidCode midCode);
// 为name申请一个写寄存器
Register applyWriteReg(String name, MidCode midCode);
// 暂时借用一个寄存器
Register borrowReg(MidCode midCode);
// 归还一个暂用的寄存器
Register returnReg(Register reg, MidCode midCode);
// 清空已经借出的寄存器
Register clearBorrow(MidCode);
```

- 全局寄存器分配

全局寄存器分配使用了图着色分配算法。根据到达定义建立定义使用网，对变量重命名，再进行活跃变量分析，根据"**一个变量定义处另一个变量是否活跃**"原则建立冲突图，进行全局寄存器的图着色分配。

在实现中，我以每个函数作为进行冲突图构建与全局寄存器分配的基本单位。只有作用域跨越基本块的变量才有资格竞争全局寄存器，**作用域跨越基本块的变量是函数的所有基本块的`in`与`out`的并集**，且全局变量不能够使用全局寄存器，否则伴随函数调用可能出现值不一致的错误。

可以当做全局寄存器分配的有如下寄存器，如果全局寄存器没有被用尽，剩下的在函数内可以"借出"作为临时寄存器使用。

```
$s0,$s1,$s2,$s3,$s4,$s5,$s6,$s7,$v1,$a1,$a2,$a3,$k0,$k1
```

用冲突图分配寄存器中，我在算法的每一步执行前都将节点按照`degree`从大到小排序，在冲突图中寻找第一个连接边数小于`K`的节点可以使用二分，在标记不分配全局寄存器节点时，我**选择的是当前`degree`最大的节点**，从而移走这个节点后最大限度减少冲突，提升全局寄存器的利用率。

分配时尽可能使用最少数目的全局寄存器，具体实现上每次寻找可用寄存器时都**按照固定的顺序**找，则一定能够使用最少数目的寄存器，从而增加临时寄存器的数目。

- 临时寄存器分配

临时寄存器分配有很多门道，小细节做好可以很好地提升性能。架构设计上，我对每一个函数建立了一个寄存器分配器，由寄存器分配器管理当前函数的临时寄存器池，全局变量分配完毕后，初始每个函数的临时寄存器池，每个函数可用的临时寄存器包括如下的几个临时寄存器和当前函数没有使用到的全局寄存器。

```
$t0,$t1,$t2,$t3,$t4,$t5,$t6,$t7,$t8,$t9,$fp
```

临时寄存器部分非常难的一部分是**临时寄存器池的管理**。

临时寄存器池的管理包括很多方面，包含但不限于**临时寄存器的替换策略，寄存器的申请方式，临时寄存器的写回策略，寄存器的保存，临时寄存器池的清空**等。

我的所有寄存器申请相关的接口均传入了`MidCode`，如此可以**利用当前`MidCode`的上下文信息，更好地分配临时寄存器**。

临时寄存器是一个随用随申请的资源，因为中间代码的上下文信息是已知的，所以我使用**OPT置换策略**，每次优先替换当前`MidCode`处不活跃变量和常数，如果均活跃，则替换最晚使用的变量对应的寄存器。

寄存器的申请方面，我设计了上文所展示的5个接口，申请读写寄存器时，如果是变量已经拥有全局寄存器，则直接返回其拥有的全局寄存器，否则检查是否已经拥有临时寄存器，有则返回，否则检查是否有空闲寄存器，有则返回，否则检查是否有不活跃变量或者常数，有则返回，否则根据OPT策略找到一个寄存器。申请读寄存器中如果变量或者常数没有寄存器则需要将变量值加载进寄存器或将常数值写进寄存器，申请写寄存器时要对寄存器置脏位。当替换了一个活跃变量的寄存器时，如果是脏的，要写回内存。在寄存器申请方面，**我的一个设计亮点是提供了寄存器借用接口。**并不是所有的中间代码的变量数与寄存器数都是相等的，例如在除法优化、访存数组时，需要额外多使用1个寄存器，最简单的策略是专门为其预留一个寄存器如$t0$，但是这样的中间代码的数量是比较少的，这样做相当于牺牲掉了一个寄存器。而对于这种生命周期局限在一条中间代码的范围内的寄存器，我设置了借用的接口，使用前借用，使用后归还，借用寄存器的策略与为变量申请寄存器类似，也是优先借出空闲寄存器或者不活跃变量对应的寄存器。在寄存器申请方面，还需要注意的是，**读的寄存器之间不能冲突，写的寄存器不能与读的寄存器冲突，借出的寄存器不能与读、写和已借出的寄存器冲突。**

在函数调用处需要保存现场，我的策略是全局寄存器均保存，保存在函数调用栈的寄存器保存区。临时寄存器只保存活跃变量，如果脏，保存回变量所在位置。函数调用结束后恢复现场，从调用栈恢复全局寄存器，从变量所在位置恢复临时寄存器。

在基本块的结尾，对临时寄存器池进行清空，如果变量活跃且寄存器脏，则写回内存。

> 其他优化

- 乘法优化

当乘法指令中一个操作数是常数且是2的幂次时，可以将`mult`指令转换成`sll`.

- 除法/求模优化

除法实现了**魔数优化**，优化与论文《Division by invariant integers using multiplication》一致，取整方式为**向零取整。**值得注意的是，实现论文中计算m_{low}与m_{high}时如果使用`long`数据类型，计算的中间过程可能会出现`long`数据类型溢出从而导致一些边界数据的魔数时出现错误，解决这个问题的办法是使用`Java`中的`BigInteger`类型。

求模运算可以如下转换：

```
a % b = a - a / b * b
```

其中乘法再可以使用乘法优化。

- 全局变量转为局部变量

全局变量在数据流分析中都是作保守估计的，由于`main`函数只执行一次，所以只在`main`函数中定义和使用的全局变量可以转化为`main`中的局部变量，从而使得数据流分析的结果更加精确。

- 窥孔优化

在我的中间代码中，在表达式计算中会将右边的表达式计算出来并赋给一个临时变量，再将临时变量赋给目标变量，所以常常出现这样的中间代码：

```
#T23 = #T21 + #T22
@a = #T23
```

扫描中间代码时通过向后多看一条将其优化为

```
@a = #T21 + #T22
```