编译器设计文档

20231164 张岳霖

一、参考编译器介绍

首先是阅读了老师下发的pascal和pl0编译器，在了解编译器的大致结构和功能后，我又在github上上查找了开源的编译器来进行参考和阅读。下面分析一下github上的一个使用Java语言编写的SysY编译器，github地址https://github.com/Chenrt-ggx/MipsCompiler。

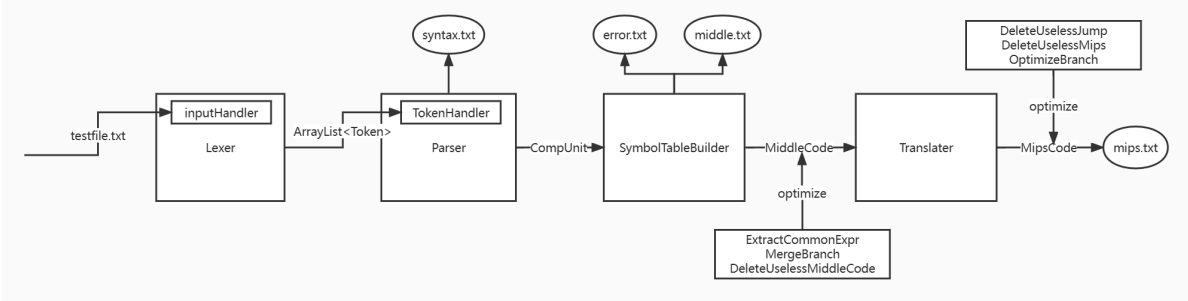
总体结构：参考编译器按照Java包可以分为词法分析部分(WordAnalyse)，语法分析部分(GrammaAnalyse)、语法树模块(SyntaxTree)、中间代码模块(IntermediateCode)、目标代码模块(MipsObjectCode)和全局配置包(Global)。整体架构和流程和课上讲的流程差不多一致。首先读取程序信息，将其以字符串的形式输入到词法分析程序中，词法分析程序分析得到单词，将单词以单词数组的形式传递到语法分析程序中。参考编译器建立了一个独立的包用于保存语法树中所有语法节点，语法分析程序的分析结果保存在语法树中，接着通过递归下降算法遍历语法树进行中间代码和目标代码的生成。

文件组织：参考编译器的文件分为六个包，文件树如下：

```
- Global
    一些全局配置文件
- GrammarAnalyse
    - xxxParser.java  // 为每个语法成分都设计了一个Parser
- IntermediateCode
    - InterElement
        定义中间代码中与内存相关的语法成分
    - Operands
        定义中间代码中与运算相关的语法成分
    - BasicBlock.java  // 基本块
    - BrachOptimizeTemplate.java  // 优化相关
    - ConflictGraph.java  // 冲突图
    - FuncBlock.java  // 中间代码中的函数
    - IntermediateBuilder.java  // 生成中间代码
    - Intermediate.java  // 中间代码类
    - VirtualRunner.java  // 中间代码运行虚拟机
- MipsObjectiveCode
    - instructionSet
        定义指令mips集合
    - MipsAssembly.java  // 汇编代码的主体类
- SyntaxTree
    定义语法树的节点
- WordAnalyse
    - Word
        定义各个词法类
    WordAnalyzer.java  // 词法分析主程序
```

二、编译器总体设计

本编译器由Java语言编写，能将SysY语言翻译成mips汇编语言。主线上对输入程序扫描四遍，流程图如下：



第一遍：词法分析

功能：读入字符串格式的输入程序，进行词法分析，将分析结果保存到ArrayList<Token>中

该部分文件在Frontend.Lexer包中，代码逻辑比较简单，包含的文件有：

```
- InputHandler.java: 输入串指针。封装输入串，维护一个输入串指针，向外提供多种指针移动和字符串读取方法；
- TokenType.java: 单词枚举类，用于保存单词类别码和对应的正则表达式；
- Lexer.java: 词法分析器，提供一个静态函数lex，进行正则匹配和保存结果。
```

词法分析结束后，得到单词数组，传入到下一阶段进一步处理。

第二遍：语法分析

功能：读入单词数组，进行语法分析，建立语法树。

该部分文件在Frontend.Parser包中，此处的关键是恰当改写文法为LL(1)，消除左递归和避免回溯，保证递归下降分析法的可行性，文件结构如下：

```
- Parser
    - decl
        - types
            decltypes
        - DeclParser.java
    - expr
        - types
            exprtypes
        - ExprParser.java
    - func
        - types
            functypes
        - FuncParser.java
    - stmt
        - types
            stmttypes
        - StmtParser.java
    - CompUnit.java
    - Parser.java
    - TokenHandler.java
```

这里将文法分为四个大类，decl、expr、func、stmt，type包下的每个文件包含一个对文法非终结符建模的类，每一类中都有一个独立的parser，采用递归下降的方法对本类中的文法进行语法分析。最外层有一个总的parser，对四个文法类进行汇总，最终结果会保存在CompUnit类中，传递给下一遍进行后续处理。另外，TokenHandler和第一遍中的InputHandler的意义相似，封装输入序列并维护一个指针，向外提供多种指针移动和单词读取方法。

第三遍：语义分析、中间代码生成（符号表建立、错误处理）

功能：进行语义分析，和中间代码生成。建立符号表和进行错误处理也在该遍中完成。

该遍任务较多，功能复杂，任务之间耦合性相对较大，代码书写比较困难，是编译器构建过程中最关键的一部分，文件结构如下：

```
- Exceptions
    allExceptions
- Frontend
    - Symbol
        - Errors.java
        - Symbol.java
        - SymbolTable.java
        - SymbolType.java
    - Util
        - ConstExpCalculator.java
    - SymbolTableBuilder.java
- Middle
    - optimizer
        - DefUseCalcUtil.java
        - DeleteUselessMiddleCode.java
        - ExtractCommonExpr.java
        - MergeBranch.java
    - type
        allMiddleTypes
    - MiddleCode.java
```

文件树中的allExceptions、allMiddleTypes分别是对错误和中间代码中元素进行抽象建模的类，为了简化文件树没有一一列出。该部分的主程序在SymbolTable.java文件中，仍然采用递归下降的方法检查和翻译各个语法结构，最终的翻译结果会保存到MiddleCode.java中。ConstExpCalculator.java的作用是进行常量表达式的计算，主要是计算常量的初始值，全局数组的初始值以及数组维数等，MiddleCode依次经过optimizer中的三步针对中间代码的优化，优化结果传递到下一遍中进一步处理。

第四遍：目标代码生成

功能：翻译中间代码，掌管和分配寄存器。

相较于第三遍，该遍任务单一，目的性明确，完成代码的编写相对容易。但要想生成更高质量的汇编代码，需要进一步花心思思考恰当的寄存器分配策略和代码合并简化策略，这部分花费的时间是最多的。这部分的文件结构如下：

```
- BackEnd
    - instructions
        allmipsinstrustions
    - optimizer
        - ConflictGraph.java
        - ConflictGraphNode.java
        - DeleteUselessJump.java
        - DeleteUselessMips.java
        - OptimizeBranch.java
    - MipsCode.java
    - Register.java
    - Translator.java
```

文件树中的allmipsinstructions代表对mips汇编指令建模描绘的类，为了简化文件树没有详细列出。翻译中间代码为mips指令的主程序在Translator.java中，接收上一遍传入的MiddleCode作为输入，在Register.java中临时寄存器分配策略（OPT）和ConflictGraph.java中全局寄存器分配策略（图着色）的指导下进行寄存器分配和目标代码生成。最终得到的汇编代码MipsCode依次经过optimizer中的三步针对目标代码的优化，输出到mips.txt中

三、词法分析设计

词法分析程序在Frontend.Lexer包中。写此部分的代码时，理论课上讲了一种基于自动机的词法分析方法，通过学长的博客又了解到了通过正则匹配进行词法分析这种方法，权衡实现难度，我最终选择了使用正则表达式进行单词匹配。该部分逻辑简单，任务和意图明确，编码前后的设计方案没有做修改，具体设计细节如下：

为了便于确定符号所处的行数，输入符号串被按行存储保存在InputHandler中，它同时维护了一个字符串指针，提供了以下的方法，为后续的正则匹配带来了极大的便利。

```
public boolean moveForward(int step) {}  // 字符指针后移step位
public String getForwardWord(int step) {}  // 返回后面step个字符
public String getCurrentLine() {}  // 返回当前行剩下的字符
public boolean reachEnd() {}  // 到达末尾
public boolean skipBlanks() {}  // 跳过空格
public boolean skipComments() {}  // 跳过注释
public int getLine() {}  // 返回当前行数
```

TokenType中定义了每个单词和相应的正则表达式，这里需要注意的是，对于保留字，需要正好匹配，否则需要识别成标识符IDENFR，这里我们需要使用正则表达式中的**负向先行断言**。正则匹配的顺序采用保留字-->标识符-->运算符分隔符等。

完成以上的准备工作，词法分析程序就水到渠成了，Lex主函数的实现如下：

```
public class Lexer {
    public static TokenList lex(InputHandler inputHandler) {
        TokenList tokenList = new TokenList();
        while (!inputHandler.reachEnd()) {
            // skip blanks
            if (inputHandler.skipBlanks()) break;
            // skip comments
            if (inputHandler.skipComments()) continue;
            // traverse all patterns and make regex match
            for (TokenType tokenType : TokenType.values()) {
                Matcher matcher =
Pattern.compile(tokenType.getPattern()).matcher(inputHandler.getCurrentLine());
                if (matcher.find()) {
                    tokenList.add(new Token(tokenType, matcher.group(0),
inputHandler.getLineNumber()));
                    inputHandler.moveForward(matcher.group(0).length());
                    break;
                }
            }
        }
        return tokenList;
    }
```

```
}
```

四、语法分析设计

```
### 编码前的设计
```

语法分析部分，关键在于改写文法，消除左递归和解决回溯问题。由于写该作业时，理论课上刚刚讲完这部分，因此文法改写的相对比较顺利。改完后的文法如下：

```
// declare
// semicn is stored in Decl
声明 Decl → ConstDecl | VarDecl
变量声明 VarDecl → 'int' VarDef { ',' VarDef } ';'
常量声明 ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';'
常数定义 ConstDef → Var '=' ConstInitVal
变量定义 VarDef → Var | Var '=' InitVal
常量变量 Var -> Ident { '[' ConstExp ']' }
常量初值 ConstInitVal → ConstExp | '{' [ ConstInitVal { ',' ConstInitVal } ] '}'
变量初值 InitVal → Exp | '{' [ InitVal { ',' InitVal } ] '}'

// expressions
左值表达式 LVal → Ident {'[' Exp ']'}
数值 Number → IntConst
一元表达式 UnaryExp → PrimaryExp | FuncExp | UnaryOp UnaryExp
单目运算符 UnaryOp → '+' | '−' | '!'
带括号的表达式 BraceExp -> '(' Exp ')'
基本表达式 PrimaryExp → BraceExp | LVal | Number
函数调用 FuncExp --> Ident '(' [FuncRParams] ')'
函数实参表 FuncRParams → Exp { ',' Exp }
表达式 Exp --> AddExp
常量表达式 const Exp --> AddExp
条件表达式 Cond → LOrExp
乘除模表达式 MulExp → UnaryExp {('*' | '/' | '%') UnaryExp}
加减表达式 AddExp → MulExp {('+' | '−') MulExp}
关系表达式 RelExp → AddExp {('<' | '>' | '<=' | '>=') AddExp}
相等性表达式 EqExp → RelExp {('==' | '!=') RelExp}
逻辑与表达式 LAndExp → EqExp {'&&' EqExp}
逻辑或表达式 LOrExp → LAndExp {'||' LAndExp}

// functions
函数定义 FuncDef → FuncType Ident '(' [FuncFParams] ')' Block
主函数定义 MainFuncDef → 'int' 'main' '(' ')' Block
函数类型 FuncType → 'void' | 'int'
函数形参表 FuncFParams → FuncFParam { ',' FuncFParam }
函数形参 FuncFParam → BType Ident ['[' ']' { '[' ConstExp ']' }]

// statement
// semicn is stored in Stmt
语句块 Block → '{' { BlockItem } '}'
语句块项 BlockItem → Decl | Stmt
语句 Stmt --> StmtInterface ';'
StmtInterface --> 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
```

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

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

在编码前的设计阶段，我考虑将每个非终结符都建立对应的类，在类中模拟每个非终结符包含的元素。非终结符之间的相互包含关系正好可以组成一颗语法树，每个类中的get方法可以作为语法树中连接各节点的边，以此种格式来保存语法结构。在解析完成后，得到一颗语法树，将该语法树的根节点返回给下一个阶段。

编码时的改动

在具体编码的过程中，我采用了一种逐层下降解析的思路。该部分的程序在Parser包中，我将语法成分共分为四大部分`decl, expr, func, stmt`，并根据文法为四种语法成分分别构建非终结符的类，比如对于AddExp，建立如下的类：

```
public class AddExp {
    // AddExp → MulExp {('+' | '−') MulExp}
    private final MulExp firstExp;
    private final ArrayList<MulExp> exps;
    private final ArrayList<Token> seps;
}
```

为FuncDef建立如下所示的类：

```
public class FuncDef {
    // FuncDef → FuncType Ident '(' [FuncFParams] ')' Block
    private final Token returnType;
    private final Token ident;
    private final Token left;
    private final Token right;
    private final ArrayList<FuncFParam> funcFParams;
    private final ArrayList<Token> seps;
    private final BlockStmt blockStmt;
}
```

非终结符的类是对文法右端信息的完全匹配和保存。此外，我将词法分析过程得到的符号保存在TokenHandler类中，并同时维护了一个指针，在解析过程中，逻辑是首先调用tokenHandler.getForwardToken()，向前得到一个单词，检查其与哪个文法匹配，接着调用解析该文法的函数进行递归下降的文法分析。

该部分的文件结构如下，decl、expr、func、stmt都有一个独立的Parser，每个都分别用来解析属于当前语法成分包含的文法形式，顶层有一个总的Parser，可以将四个部分串联起来，最终解析得到一个CompUnit，是抽象语法树的根节点。

```
- Parser
```

```
    - decl
        - types
            decltypes
        - DeclParser.java
    - expr
        - types
            exprtypes
        - ExprParser.java
    - func
        - types
            functypes
        - FuncParser.java
    - stmt
        - types
            stmttypes
        - StmtParser.java
    - CompUnit.java
    - Parser.java
    - TokenHandler.java
```

在顶层解析函数中，需要得到接下来的1、2、3个Token，记为fisrtToken、secondToken、thirdToken，当firstToken是const、secondToken是int时，代表当前解析的时全局常量，调用DeclParser进行解析；当firstToken是int、secondToken是标识符时，需要检查thirdToken，如果thirdToken不是左括号，则表示当前正在解析全局变量，调用DeclParser进行解析，如果是左括号，则进一步检查标识符，如果标识符不是main，调用FuncParser进行解析，如果标识符是main，意味着已经读到了最后一个函数，调用主函数解析程序，解析完成后退出解析过程。

五、错误处理设计

错误处理部分需要我们检查a-m八种错误。

编码前的设计

在编码前，我考虑将所有的错误分成两大类，一种是如i（缺少分号），j（缺少右小括号），k（缺少右方括号）等**语法错误**，他们是在程序中处于独立的状态，不需要建立符号表，就能够完全判断出他们的出错地点，因此考虑可以在语法分析部分直接检测出错误；另一种错误如b（名字重定义），c（未定义的名字），d（函数参数个数不匹配）等**语义错误**，其涉及到对整个程序的把握，即需要建立符号表，需要在遍历语法树、创建符号表的基础之上做错误处理检查。

同时，语义错误的产生意味着语法分析过程不一定能得到期望的结果，有的产生式还需要根据first集合做检查和判断，需要耐心的考虑到因为语法错误而导致的所有可能的文法组合情况。

编码时的改动

在具体实现过程中，我首先仍然是为每种错误建立对应的类来进行模拟，类中保存错误的异常码和错误所在的行数。对于检查和报告错误的时间，我没有采用编码前规划的，在语法分析过程检查和报告语法错误，在后续建立符号表的基础上，检查和报告语义错误。因为我觉得这样的错误检查比较零散，错误会在整个解析过程的很多阶段中产生，不利于代码的维护。我考虑在语法分析时先不报告错误，如果输入程序具有缺少括号分号等不满足文法要求的错误，则将其赋成null，并继续进行语法分析。之后我将错误处理整个的放在一个单独的类中，在此类中使用递归下降算法遍历整颗语法树，实现对所有语法和语义错误的检查。这种架构方式降低了代码之间的耦合程度，各个任务之间相互解耦，维护和调试相对比较容易。

在编码时的另一个重要的问题是如何规划和设计自己的**符号表**。符号表需要是树形结构，每个符号表都应该有自己的外层符号表（顶层符号表除外）。新建符号时，需要检查本层符号表中是否有重复出现的；当查找符号时，需要依次向上递归检查符号表中是否含有当前寻找的符号，找到符号时即可退出递归，因为内层符号可以覆盖掉外层符号。符号表中记录的信息如下：

```
public class SymbolTable {
    private final SymbolTable parent;
    private final HashSet<SymbolTable> childSymbolTables = new HashSet<>();
    private final HashMap<String, Symbol> symbols = new HashMap<>();
    private final SymbolTable funcSymbolTable;  // 当前符号表所属的funcSymbolTable
    private int stackSize = 0;
}

public class Symbol implements LeafNode, Operand, Cloneable {
    public enum Scope {
        GLOBAL,
        LOCAL,
        TEMP,
        PARAM,  // 只记录函数形参中的array类型变量
    }

    // for all
    private String name;
    private final SymbolType symbolType;
    private Token ident;
    private final boolean isConst;
    private Scope scope;
    private int size;
    private int address;
    private boolean hasAddress = false;

    // int
    private int constInitInt;
    private boolean hasConstInitInt = false;

    // 数组
    private final ArrayList<Integer> dimSize;
    private ArrayList<Integer> constInitArray;
    private boolean hasConstInitArray = false;
    private final int dimCount;

    // 函数
    private final ArrayList<Symbol> params;
    private final SymbolType returnType;

    // 是否是内联变量
    private boolean inlineVariable = false;
    private FuncBlock outerFunc = null;
    private FuncBlock inlineFunc = null;
    private int funcInliningIndex = 0;
}
```

在错误处理的过程中，需要边进行错误处理，边填充和检查符号表。错误处理各个错误的检查逻辑如下

- a 非法符号：按照文法定义文档对合法符号的限制和要求对FormatString做检查即可；
- b 名字重定义：在定义符号时，查询当前符号表中是否有该符号，如果有，则报重定义错误；
- c 未定义的名字：在使用符号时，查询当前符号表并递归查找当前符号表的父符号表中是否存在该符号，如果递归检查到顶层符号表仍然未找到该符号，则报未定义名字错误；
- d 函数参数个数不匹配：检查到FuncExp时，首先会在符号表中得到函数定义信息，之后检查调用函数的实参和函数定义时的形参个数是否相等，如果个数不相等，则报函数参数数量不相等错误；
- e 函数参数类型不匹配：检查到FuncExp时，首先会在符号表中得到函数定义信息，之后检查调用函数的实参和函数定义时的形参的类型是否匹配。需要注意的是，对于数组维数，必然是可以在编译期计算出来的，因此需要建立一个Calculator计算类，用于计算形参和实参的数组各维数是否相等；
- f 无返回值的函数存在不匹配的return语句：函数类型和函数末尾是否存在返回值信息都会保存在符号表中，因此可以在检查到FuncDef中做此类检查；
- g 有返回值的函数缺少return语句：函数类型和函数末尾是否存在返回值信息都会保存在符号表中，因此可以在检查到FuncDef中做此类检查；
- h 不能改变常量的值：符号是否是常量类型这一信息被存储在符号表中，在检查到AssignStmt和GetintStmt时会检查左端的符号是否是常量；
- i 缺少分号：按照语法定义，在可能缺少分号的语法元素处做相应的检查；
- j 缺少右小括号：按照语法定义，在可能缺少右小括号的语法元素处做相应的检查；
- k 缺少右中括号：按照语法定义，在可能缺少右中括号的语法元素处做相应的检查；
- l printf中格式字符与表达式个数不匹配：在检查到printf语句时，检查printf表达式中%d的个数和后面表达式的个数是否匹配；
- m 在非循环块中使用break和continue语句：在全局设置两个栈，分别保存进入循环过程后，break语句和continue语句出现时应该跳转到的基本块，这既能够解决代码生成部分的问题，帮助确定break和continue语句应该到达的位置，也能够在错误处理部分用来做该项错误检查。

六、代码生成设计

中间代码生成部分

这部分的实现是在遍历语法树的同时生成中间代码，关键之处在于中间代码的设计和控制流的翻译流程。一般来说，程序的语言结构主要分为顺序结构，选择结构和控制结构。对于顺序结构，我选择用**四元式**进行表达，对于控制结构和循环结构，我选择**跳转语句**和**跳转标签**进行翻译。中间代码结构如下：

```
// 顺序结构：四元式表达
a = b + c
ADD B C A

// 条件控制结构：跳转+标签
if(a || b){
    print(1);
}

IF_BEGIN:
    BNEZ a 成立
    BNEZ b 成立
    J 不成立
```

```
成立：
    PRINT 1
    J 不成立
不成立：

// 循环控制结构：跳转+标签
while(a != 5){
    print(1);
}

WHILE_CHECK:
    BNE a 5 WHILE_END
    j WHILE_BODY
WHILE_BODY:
    PRINT 1
    J WHILE_CHECK
WHILE_END:
```

中间代码的翻译的一个非常关键的部分是**对于基本块的建立和管理**。基本块的入口包括跳转语句跳转到的位置，跳转语句的下一条语句等。在分割和建立基本块以及翻译涉及到跳转的If和While语句时，我采用了如下的翻译方式：

```
 public void checkIfStmt(IfStmt ifStmt) {
        // check cond Stmt
        BasicBlock ifBody = new BasicBlock("IF_BODY_" + blockCount++);
        BasicBlock ifElse = new BasicBlock("IF_ELSE_" + blockCount++);
        BasicBlock ifEnd = new BasicBlock("IF_END_" + blockCount++);
        checkCond(ifStmt.getCond(), ifBody, ifStmt.hasElse() ? ifElse : ifEnd);

        currSymbolTable = new SymbolTable(currSymbolTable,
currFunc.getFuncSymbolTable());  // 符号表下降一层

         currBlock = ifBody;
         ifBody.setIndex(blockId++);
         checkStmt(ifStmt.getStmts().get(0));
         currBlock.addContent(new Jump(ifEnd));

        if (ifStmt.hasElse()) {
            currBlock = ifElse;
            ifElse.setIndex(blockId++);
            checkStmt(ifStmt.getStmts().get(1));
            currBlock.addContent(new Jump(ifEnd));
        }

        currBlock = ifEnd;
        ifEnd.setIndex(blockId++);
        currSymbolTable = currSymbolTable.getParent();  // 符号表上升一层
    }
```

在整个中间代码生成的过程中，维护一个全局的基本块`currBlock`用于保存当前正在翻译的基本块。在顺序语句的翻译过程中，翻译结果都会加入到这一基本块上，而对于控制语句和循环语句，则是通过改变当前基本块，为`currBlock`设定恰当的值，改变顺序语句加入到的基本块，进而实现对代码执行路径的选择。

比如如上的If语句翻译为中间代码的过程，首先建立三个基本块ifBody、ifElse和ifEnd，首先在当前基本块下翻译条件语句，由于已经确定并建立好条件成立（ifBody）和不成立（ifElse或ifEnd）应该跳转到的基本块，因此在条件语句翻译时能够确定条件成立和不成立时应该跳转到的位置。接着，将当前基本块设置为ifBody基本块，调用Stmt的翻译程序翻译Stmt，翻译后跳转到if语句结束的基本块ifEnd。如果if语句存在else语句块，则还需要将当前基本块设置为ifElse基本块，并也跳转到ifEnd基本块。当ifStmt和可能存在的elseStmt都翻译完成后，将当前基本块设置成ifEnd基本块，继续向下翻译即可。

由此翻译方式得到的基本块实质上是一个**链表**，每个基本块都有后续跳转到的一个或两个基本块。在中间代码翻译完成后，能够得到一张基本块图。在这张图上通过DFS或BFS搜索得到的基本块序列其实是不同的，对于进一步翻译基本块为mips汇编代码的难度以及需要保存到内存中的变量的多少是不同的。这里我为了得到我想要的基本块排列顺序，为每一个基本块都设置了一个id，当中间代码翻译完成得到基本块图后，我使用DFS算法遍历图得到所有基本块，在调用sort算法对基本块进行排序，便可以得到我预期的基本块排列顺序。

目标代码生成部分

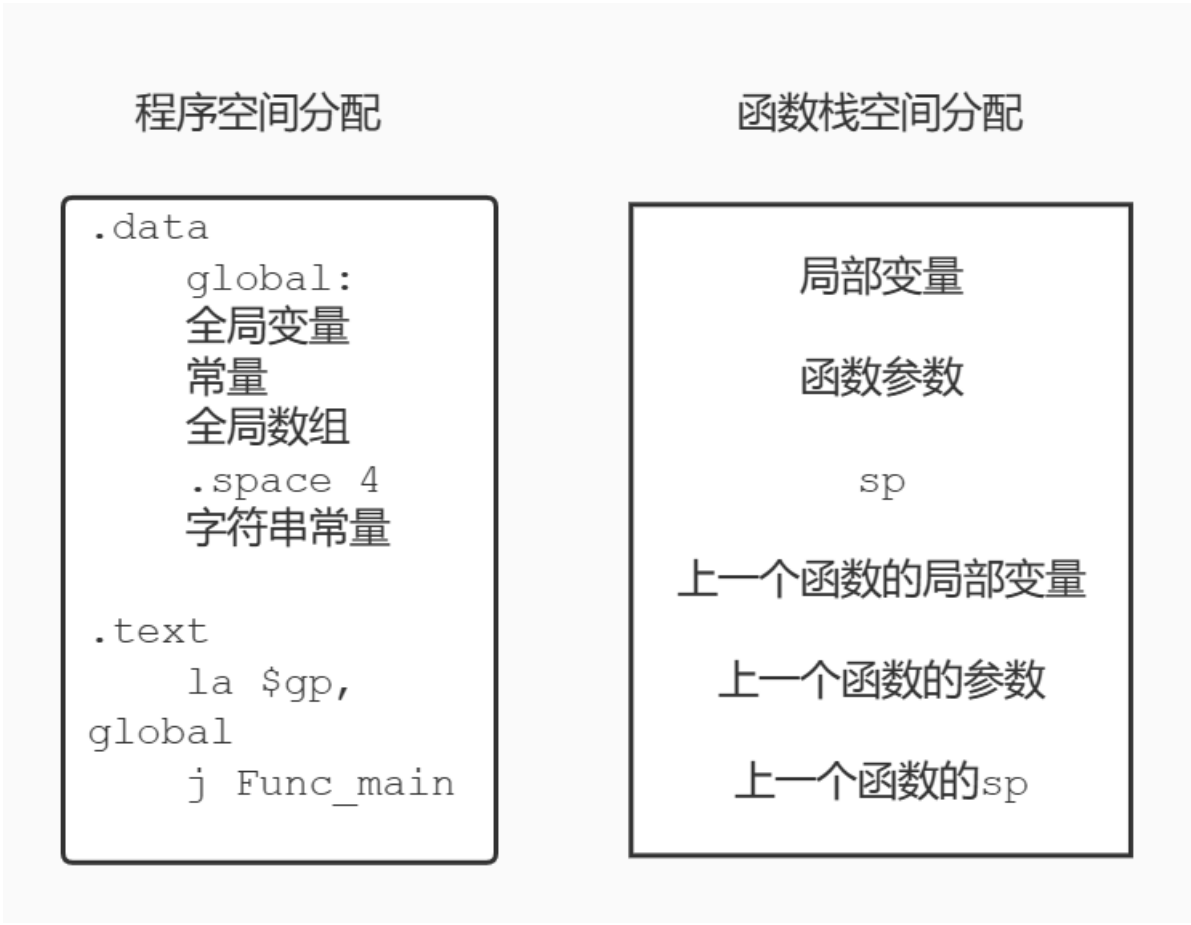
形式上将中间代码翻译成目标代码其实是难度不大的，因为中间代码中设计的四元式序列以及跳转语句在汇编代码中都有相应的代替选择。目标代码的生成程序在BackEnd.Translator.java中，仍然采用类似于递归下降的算法，逐层地翻译各个中间代码对象。

这里的困难之处在于**如何分配寄存器，以及什么时候将变量存在内存**中。BackEnd.Register.java负责寄存器的分配与指派。我最初的寄存器分配策略是LRU，在Register类中维护一个队列，队列从前到后的顺序表示寄存器最晚到最近被使用。在正常顺序翻译的基本块内，如果寄存器不足，则将队列列首的寄存器弹出，对应的变量存进内存，并将该寄存器分配给即将要使用的变量。

对于何时将寄存器的值重新存到内存中，由于我最初没有编写图着色寄存器分配算法，所有寄存器和变量均不是一直处于绑定状态，因此需要将寄存器存回内存的地方主要有三处：

1. 当寄存器被LRU时需要将寄存器存回内存
2. 当翻译到跳转语句时，需要将所有寄存器存回内存
3. 当翻译到函数调用语句时，需要将所有寄存器存回内存

在**变量内存空间**的选取方面，我将全局变量、常量、全局数组以及字符串常量存放在了data区，并将$gp寄存器赋值为全局区的基地址。其它变量存放在text区。



在**函数内存空间**的选取方面，在调用某个函数时，首先记录该函数的栈指针地址，传参的方式时将参数保存在被调用函数栈指针前几个位置，对于数组参数则将数组的地址传给调用的函数。进入函数时，会相应地从栈地址前几个位置来取调用参数。

更多的有关代码生成时采取的策略和思路将在代码优化部分介绍。

七、代码优化设计

中间代码优化

死代码删除

死代码删除需要建立在**活跃变量分析**的基础之上，在按照编译原理书上讲解的算法进行活跃变量分析后，判断每条指令的运算结果是否活跃（在该条指令的out集合中），如果指令的结果不活跃，意味着当前指令属于死代码，可以删除。需要注意的是，对于getInt指令，不能简单删除整条指令，仍需要在指令所在位置加入读取输入的代码占位，以防输入信息读取错误。

常量传播

在代码中，有两类符号可以看作是常量。其一是**本身在定义时就以`const`关键字修饰的符号**，一定是常量，可以在编译器进行计算和替换；其二是**在整个代码运行阶段都没有发生改变的变量**，也可以在编译器直接代入它们的初始值，直接进行计算。

对于寻找在整个代码运行阶段都没有发生改变的变量，采取的方法是在变量定义时将变量和对应的初始值加到哈希表中，之后在翻译到赋值语句等对左端数值做出改变的语句时，将左端符号移除哈希表。当所有代码翻译完成后，遍历每条中间代码，对中间代码中所有在哈希表中的符号替换为相应的数值。

编译期计算

编译期计算比较容易实现。在翻译到每个二元操作符如+、-、*、/时，如果其左右操作数都是常量，则可以直接进行计算，使用这个计算结果替换这两个操作数和二元操作符。

函数内联

首先需要**判断函数是否可以内联**，由于 SysY 语法中不包含函数声明语句，因此不需要生成函数调用关系图即可以判断函数是否可以内联。在生成中间代码过程中，如果发现函数定义的语句块调用了自己，证明该函数是递归函数，因此不可内联。**不包含调用自己的函数是可以内联的。**

函数内联的实现是在翻译到`FuncExp`时，如果判断得出函数可以内联，则不翻译调用函数的跳转语句，而是将内联函数的函数体嵌入到此位置。在内联函数嵌入之前，需要做两步操作，一是将函数的实参赋值给函数的形参；二是由于`return int`可以在函数中多次出现，因此需要在在内联函数外部申请一个变量用于保存内联函数的返回值，内联函数所有返回值都赋值给这个变量。

函数内联实现的**难点在于对内联函数符号表的处理**，原函数和内联函数可能会定义相同的符号，这意味着直接将函数内联替换FuncExp可能会导致出现符号重定义错误。因此，考虑对内联函数的符号做一个映射，统一映射到`INLINE_(原符号名字)_(序号)`。当开始函数内联时，开启一个全局标记，当检测到全局标记时，将所有符号名字按照上述模式做映射。

循环翻译优化

循环是程序中很重要的组成部分，且虽然语句量不一定多，但占据了程序运行中的大量时间，因此针对循环的优化可以有效提高程序的运行效率。

一个比较容易实现的优化点是将while语句翻译为do-while语句，原理如下：

```
while(<cond>) {
    <stmt>
}
==>
while_check:
    BEQ <cond> while_end
    <stmt>
    j while_check
while_end:
一共需要2n次跳转


if(<cond>){
    do {
        <stmt>
    } while(<cond>)
}
==>
j do_check
do_body:
    <stmt>
do_check:
    BNE <cond> do_body
do_end:
```

```
一共需要n+1次跳转
```

通过将while转化为do-while，可以将2n次跳转转化为n+1次，极大地降低了循环语句的跳转次数。

窥孔优化

表达式合并优化

该优化是针对于以下指令

```
ADD, t1, t2, 0  t2->t1
SUB, t1, t2, 0  t2->t1
MUL, t1, t2, 1  t2->t1
MOVE, t1, t2    t2->t1
```

对于以上的这些指令，可以使用t2来代替t1，来减少指令条数。需要注意的是，如果t1跨基本块仍旧活跃，需要在基本块结尾将t2的值重新赋给t1，这一检查可以通过活跃变量分析检查基本块结尾处的活跃变量来完成。

合并分支语句

该优化是针对于以下的指令序列

```
EQ/NEQ/LT/LE/GT/GE, a, b, 0
Branch a ? label1 : label2

==>
BEQZ/BNEZ/BLTZ/BLEZ/BGTZ/BGEZ a, label1, label2
```

对于以上的指令序列，可以将第一条的判断语句和第二条的跳转语句做合并，使用判断并跳转语句来做替换，可以减少指令条数。

目标代码优化

图着色全局寄存器分配策略

为跨基本块仍然活跃的变量分配全局寄存器，主要是函数中的局部变量。

全局寄存器采用图着色算法来管理和分配。图着色算法是理论课的重点知识，使用理论课学到的算法先得到变量冲突图，接着依次从图中拿到节点做着色操作，为着同一个颜色的变量分配同一个寄存器。图着色分配后，相当于变量与这个寄存器**处于时刻绑定状态**，对于除了非函数调用之外的跨基本块跳转语句，都可以不free全局寄存器。

OPT临时寄存器分配策略

为只在基本块内活跃的变量分配临时寄存器，主要是运算过程中的临时变量，以及整个程序可见的全局变量。

对于临时寄存器，采用理论最优的OPT全局寄存器分配策略。对于我们的指令规模，可以使用暴力扫描变量的下次使用时间，将下次使用时间距当前最长的变量保存到内存中，并释放它的寄存器。

另一个寄存器分配方面可以优化的点在于**对于基本块少而大的计算密集型函数**，可以将大部分寄存器设置为临时寄存器，仅留下少量全局寄存器来分给跨基本块仍活跃的变量。因为在操作系统课已经学过，OPT分配策略是理论最优的，因此给包含大基本块的计算密集型函数分配更多的临时寄存器用来保存计算的中间值是合理的，能最大程度地提高性能。

指令选择优化

指令选择优化看似比较简单，但是选择对了指令也能够极大地提高性能。一些指令替换方法如下：

```
sge ==> 4句基础代码
slti ==> 1句基础代码
==> 使用slti代替sge

seq $t1, $t2, 0 ==> 4句基础代码
beq $t1, $0, label1 ==> 1句基础代码
beqz $t2, label1 ==> 1句基础代码
==> 使用beqz代替seq和beq，同理sne\slt\sle\sgt\sge所在的比较后跳转指令也可以做相应的替换
```

乘除优化

乘除优化的实现参考论文写就好了（参考论文可以见Division by Invariant Integers using Multiplication），由于在竞速排序中，乘除指令被赋予了很高的权重，因此考虑将乘除指令翻译成翻译成多条移位等位运算来做等价代换。

窥孔优化

无用跳转语句的删除

当跳转到的基本块正好是接下来的基本块时，可以删除跳转语句。

此外，在做了图着色寄存器分配后，正常的跨基本块跳转只需要保存并释放临时寄存器，当跳转指令跳转到的基本块正好是下一个基本块且下一个基本块只能通过该跳转指令跳转到时，意味着这两个基本块实质上可以合并在一起，因此也就无需再保存和释放临时寄存器了，这样可以减少不必要的sw lw指令。

无用mips指令的删除

在这里，我删除的mips指令包括`addiu $t0, $t0, 0`和`move $t0, $t0`。这两种指令显然是无用的。

一个没做成功的优化--DAG图优化

DAG图优化可以消除代码段中的公共子表达式，我曾尝试了一整天想要实现这一优化，然而发现DAG图导出中间代码时的指令顺序是会影响运行结果的。此外，对于lw、sw和getint指令，需要额外定义操作符来把他们加到DAG图中。如果指令序列中出现多个getint指令，还要确保多个getint的指令相对顺序与之前一致。虽然理论课上所讲的代码原理相对简单，但是具体实现起来其实是比较困难的

结语

一学期的编译实验终于要接近尾声了，自己从编译技术这门课上收获和学到了很多，不仅是对编译课程设计的知识点有了比较深刻的把握，在编写代码，寻找程序的bug，尤其是在设计和规划一个大系统的经验上可以说是在面向对象课程基础之上的有一个重大的飞跃。当自己的编译器没有bug的目标代码，并一步步通过优化让目标代码越来越短，生成代码的质量越来越高，这个过程给了我很大的成就感。仿佛是自己一步步将一个孩子培养的越来越强壮。

这个学期中，编译实验是一个非常具有存在感的一项作业，我投入了217小时4分钟的时间，编写了一万行源代码。回顾过往，虽然有过很多的挫折和失败，但确实是一个让人成长和能给人很多收获的经历。