编译实验申优文档

19373654 王康玉

1. 写在前面

我个人的编译器整体上结束了，从词法分析写到了现在的代码优化。中间也为怎么设计架构为难过，也曾经debug许久，不过坚持下来后，还是能够完成所有的任务。我最终的代码行数大概是在5000行，也是个人写过的最大的工程项目。编译最终的竞速优化成绩个人也相对满意。

不过所有的任务全部完成之后也深感还有一些遗憾，包括部分内容没有做到解耦合，在代码优化部分注释相对较少，存在一些冗余代码等，也希望在之后的工程项目中引以为鉴。

2. 整体内容

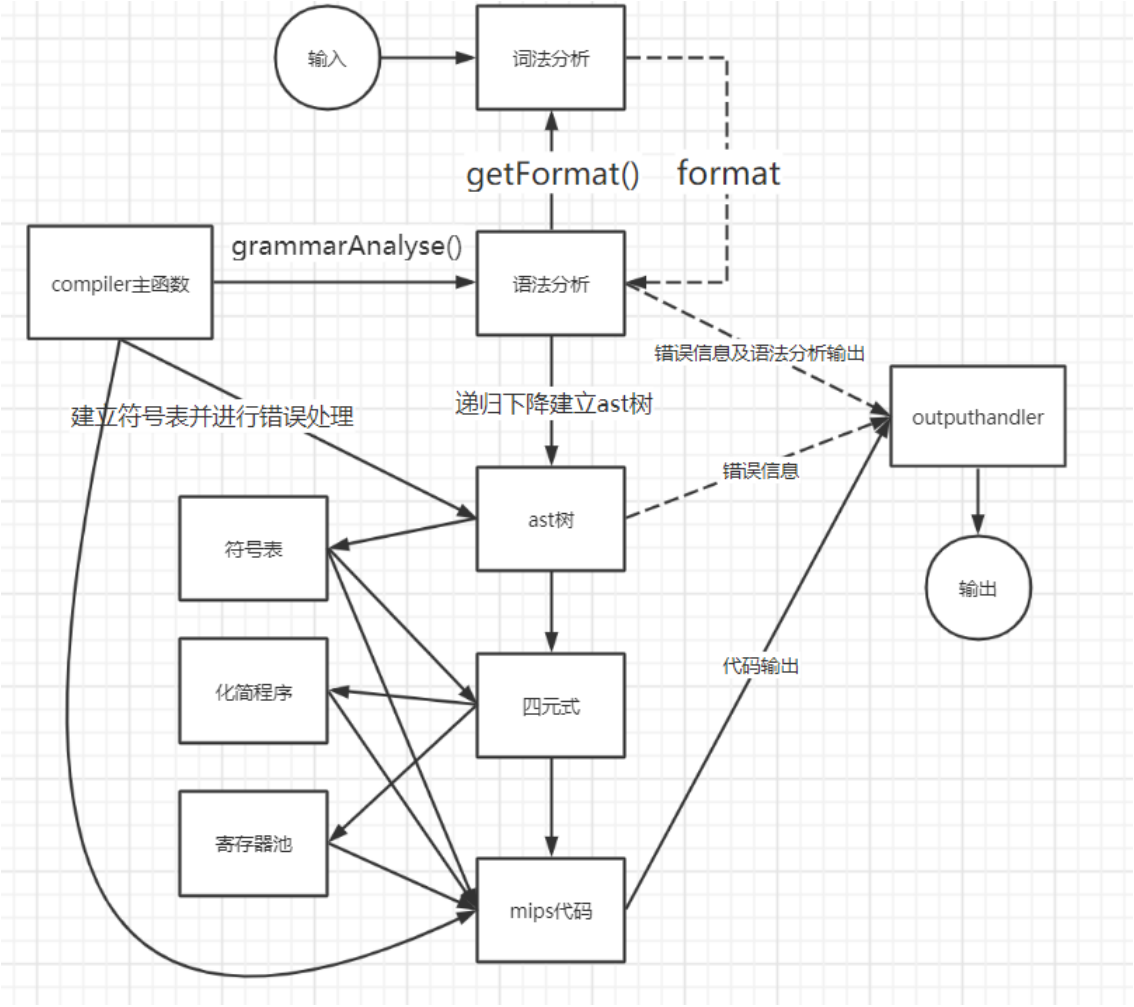
2.1.实现内容

本次编译实验实现了SysY语言编译器，具有词法分析成分输出，语法分析输出，错误处理，四元式生成，mips代码生成与优化的功能。

编译器整体上包括：

- 词法分析
- 语法分析
- ast树
- 符号表
- 四元式
- 代码生成
- 代码优化
    - 寄存器池 RegisterPool
    - 运算强度削弱 DivideSimplify
    - 叶子函数优化 LeafFunc
    - 其他

2.2.处理流程及调用关系



1. compiler主函数，主函数负责调用各个模块。
2. 各部分的结果被发送给outputHandler并由outputHandler进行输出。
3. 在词法分析部分，封装getFormat()方法供语法分析调用。读入字符并进行处理，完成后返回sym。
4. 语法分析使用递归下降的方式生成ast节点并插入到ast树适合的位置上。。
5. ast树在建立完成后，对自身进行遍历，调用符号表方法更新符号表并进行相应的错误处理。
6. ast树结合符号表直接生成四元式。
7. 四元式将会逐条被翻译为mips代码。
8. 在mips代码生成过程中，将会对四元式和mips代码进行综合优化。
9. 寄存器池提供mips代码寄存器使用的管理。

3.词法分析

3.1.词法分析实现方式

词法分析需要将目标字符或者字符串识别为某个确定的类型。

我这里将类型分为以下几部分（含文法）：

- Ident，为下划线或者字母{下划线或者字母}{数字}
- IntConst，为非零数字{数字}
- FormatString，为“{%d | \n | anscii为32,33,40-126的字符}”
- 空白字符，为空格|tab|\r|\n
- 注释语句或除号，为//|/*{*}*/
- 文件结束标志，为\0
- 其他运算符，为各种运算符

他们的起始字符不同，通过switch语句进行区分，分别为：

- ident：下划线或者字母
- IntConst：非0数字
- FormatString：双引号
- 空白字符：空格|\t|\r|\n
- 注释语句或除号：/
- 文件结束标志：\0
- 其他运算符：switch-default

在上述几部分的区分完成之后，根据需要继续读入字符直至完成该部分的读入并根据读入的内容具体判断读入内容所属类别。

3.2.实现细节与注意事项

我在实现过程中发现出现了bug，并进行了修复，需要特别注意下面这两个问题。

- 注释内部换行符仍然有效
- 注意两个字符的运算符与一个字符的运算符的区分，如&&和&

3.3.部分具体实现举例

ident部分

当获得ident字符串之后，查询是为main，int等字符串（通过HashMap）。如果是，则返回对应的类型，否则返回ident类型并记录字符串内容。

```
//初始化并完成ident字符串的读入
tepString = "";
tepString += ch;
read();
while(Character.isLetter(ch) || ch == '_' || Character.isDigit(ch)) {
    tepString += ch;
    read();
}
//查询是否为main，int等字符串。并返回对应类型
IdentAttr returnIdentAttr = stringToFormat(tepString);
if (returnIdentAttr != null) {
    return returnIdentAttr;
}
else {
    return new IdentAttr(Classification.IDENFR, tepString, lineNumber);
}
```

注释部分

首先读入两个字符判断为注释或者除号。在注释部分内部需要判断是否为换行符。

```
if (ch == '/') {
    //读第二个字符判断为注释还是除号
    read();
    if (ch == '/' || ch == '*') {
        //单行注释
        if (ch == '/') {
            read();
            while (ch != '\n') {
```

```
                if (ch == 0) {
                    return null;
                }
                read();
            }
        }
        //多行注释
        else {
            read();
            while (true) {
                if (ch == '\n') {
                    lineNumber++;
                }
                if (ch == '*') {
                    read();
                    if (ch == '/') {
                        read();
                        break;
                    }
                }
                else {
                    read();
                }
            }
        }
        return getFormat();
    }
    //除号
    else {
        return new IdentAttr(IdentAttr.valueToClassification("/"), "/",
lineNumber);
    }
}
```

4.语法分析

4.1.语法分析实现

语法分析我才用了递归下降的方式实现。

文法重写

由于递归下降的过程中不能存在左递归等文法，因此我对文法进行了重写，包括左递归消除和一些其他的调整。

这里以加减表达式为例，将左递归改为了右递归：

```
加减表达式 AddExp → MulExp | MulExp ('+' | '−') AddExp
```

FIRST的求解

我个人采用了预先求出FIRST集的方式，随后完成内容的方式实现。这种方式在现在看来是较难更改的一种方式，在修改文法之后就需要对应的修改大量FIRST集。不过也存在一定的好处，可以极可能早的发现FIRST、FOLLOW集的冲突等。

由于FIRST集的内容过多，不在这里具体贴出。

递归下降

通过上述表格中的FIRST确定递归下降调用的递归下降分程序，如果符号不在FIRST集中，则报错。

4.2. 实现细节与注意事项

存在文法

```
MainAssign → exp | Lval '=' MainVal
```

因此存在以LVal开头的语句时，可能是下面几种情况：

```
LVal = XXX;
LVal = getint();
LVal operation XXX;
```

这在递归下降中是无法判断的。如果选择修改文法，则需要修改大量内容，我最终并没有选择这种方式。而是选择直接使用LL(k)。也就是我直接预读完了整个LVal，然后根据LVal后面的符号判断这条语句到底是什么类型，然后返回调用递归下降子程序。这里要特别注意：记录开始读取的位置，不要进行除读取外的其他操作。

我具体的实现方式为超前预读设置新的flag。当该flag开启时，所有的输出、错误处理、建立ast树等功能全部被禁止，所有的读入均被视为预读操作，被存入存储ArrayList。将flag开启后调用LVal的递归下降子程序，将LVal部分读完。随后通过比较LVal之后的第一个字符来判断为exp或者是赋值语句，然后关闭flag。

```
storeFormatOn = true;        //开启flag
storeTimes = 0;              //预读符号个数
lValAnalyse();               //Lval读取
identAttr = getFormat();     //下一个sym
storeFormatOn = false;       //关闭flag
storeTimes = 0;              //重置预读数
if (identAttr.getClassification() == Classification.ASSIGN) {
    //赋值语句或getint
}
else {
    //exp语句
}
```

4.3.部分具体实现举例

预读部分

存在三个接口：

- 加载下一个format：loadFormat()
- 获得当前的一个format：getFormat()
- 向后预读第几个format（以当前位置为基准）：preGetFormat(int n)

建立一个ArrayList作为预读部分的储存内容。

loadFormat负责将arrayList的内容向前滚动一次，如果arrayList为空则重新读入一个。

getFormat直接返回arrayList的第一个format。arrayList.get(0)

preGetFormat在ArrayList中补全预读的个数并返回需要预读的内容。

常量定义constDecl（含建立ast树

程序执行的内容顺序依次为：

- 逐条识别
- const
- int
- 调用递归下降子程序constDefAnalyse()
- {逗号，调用递归下降子程序constDefAnalyse()}
- 分号

```
private ArrayList<AstDecl> constDeclAnalyse() {
    //预设ast树的属性
    ArrayList<AstDecl> astDecls = new ArrayList<>();
    //const
    IdentAttr identAttr = getFormat();
    if (identAttr.getClassification() != Classification.CONSTTK) {
        error("constDecl-const", identAttr.getLineNumber());
    }
    loadFormat();
    //int
    identAttr = getFormat();
    if (identAttr.getClassification() != Classification.INTTK) {
        error("constDecl-int", identAttr.getLineNumber());
    }
    loadFormat();
    //constDef
    astDecls.add(constDefAnalyse());
    //, constDef
    while (true) {
        identAttr = getFormat();
        if (identAttr.getClassification() != Classification.COMMA) {
            break;
        }
        else {
            loadFormat();
            astDecls.add(constDefAnalyse());    //constDef
        }
    }
    //;
    identAttr = getFormat();
    if (identAttr.getClassification() != Classification.SEMICN) {
        hasGrammarError('i');
        error("constDecl-;\n", lastLineNumber);
    }
    else {
        loadFormat();
    }
    //结束
    write("<ConstDecl>\n");
    return astDecls;
}
```

5.ast树

为了降低语法分析部分的耦合度，并且为后续四元式生成等提供更容易获得的属性，因此选择了ast树作为过渡。

5.1.ast树实现

```
//抽象类：所有类继承对象
treeNode{}

//编译单元
CompUnit {
    ArrayList<Decl>     //变量定义
    ArrayList<FuncDef>  //函数定义
}

//变量定义
Decl extends BlockItem {
    boolean isConst          //是const还是var
    Format ident             //变量名
    int dimension            //维度，0维为常数
    Exp capOne               //第一维大小
    Exp capSecond            //第二维大小
    ArrayList<Exp> initval   //初始值
}

//函数定义
FuncDef {
    isVoid                              //是void还是int
    Format ident                        //函数名
    ArrayList<FuncFParam> parametars    //参数列表
    Block block                         //block
}

//函数参数
FuncFParam {
    Format ident             //参数名
    int dimension            //维度
    Exp capSecond            //第二维大小
}

//抽象类：block项
BlockItem {}

//赋值语句
AssignStmt extends BlockItem {
    Lval lval            //左值表达式
    boolean isGetInt     //是否是getInt函数
    Exp exp              //exp值
}

//exp语句
ExpStmt extends BlockItem {
    Exp exp              //如果为空则为;
}

//block块
Block extends BlockItem {
    ArrayList<BlockItem> blockitems
```

```
}

//if语句
ifStmt extends BlockItem {
    Cond cond           //判断语句
    Block trueBlock     //判断语句正确block
    Block falseBlock    //判断语句错误block
}

//while语句
whileStmt extends BlockItem {
    Cond cond           //循环判断语句
    Block whileBlock    //循环block
}

//return语句
returnStmt extends BlockItem {
    Exp exp             //返回值，可为null
    int lineNumber
}

//print语句
printStmt extends BlockItem {
    Format formatString
    //String formatString   //格式字符串
    ArrayList<Exp>      //附加值
    int lineNumber
}

//break和continue
breakContinueStmt extends BlockItem {
    boolean isBreak     //true为break,false为continue
    int lineNumber
}

//抽象类:exp
Exp extends BlockItem {}

//运算块
Operation extends Exp {
    Operation op        //运算符
    Exp leftExp
    Exp rightExp        //单目运算则为null
}

//左值
Lval extends Exp {
    Format ident
    //String ident
    int dimension
    Exp indexOne
    Exp indexTwo
}

//常数
Number extends Exp {
    int number
}
```

```
//函数
funcExp extends Exp {
    Format ident
    ArrayList<FuncFParam> parameters
}
```

5.2.树的建立

所有ast树的节点只能通过构造函数生成。在调用构造函数时需要提供所有的属性。

因此，在语法分析递归下降过程中，需要不断的记录需要的属性，并在获得所有属性之后生成ast数的节点。

语法分析方法会返回ast节点给调用该方法的方法，成为上层方法构造树节点的一个必要属性。

例子在第4节中有提及。

6. 符号表

6.1.符号表实现

符号表主要属性为一个指向上一级表的指针，两个分别用于存储变量与常量和用于存储方法的hashMap

```
class SymbolTable {
    SymbolTable fatherTable
    HashMap<varOrConEle> varOrConMap
    HashMap<funcEle> funcMap
}
```

当产生新的符号表时，将原符号表作为父级符号表属性。这样，新符号表可以通过该指针返回。

同时，符号表支持两类操作，插入与查找

6.2.符号表插入操作

```
public boolean addConVarEle(conVarEle)  //插入变量与常量
public boolean addFuncEle(funcEle)      //插入函数
```

当使用addFuncEle()方法将一个方法插入到funcMap中时，如果该方法名不在map中，则插入成功并返回true。否则，不执行插入操作，并返回false。

同样，对于使用addConVarEle()方法插入变量或常量，有相同的操作。

6.3.符号表查找操作

```
public ConVarEle getConVarEle(String name)
public FuncEle getFuncEle(String name)
```

使用getFuncEle()方法查找一个方法是否在符号表中：

1. 直接查找该string是否在funcEle的map中，如果在，则直接返回该方法
2. 如果没有找到该方法，则调用上级符号表的符号表查找方法。
3. 如果再也没有上级符号表，则返回null，表示符号表中没有该方法。

同样，对于变量与常量有相同的操作。

7.错误处理

7.1.错误处理实现

在实现错误处理的过程中，我将错误处理分为了两类

- 分号，右小括号，右中括号缺失为第一类。在语法分析中较好处理，可以直接在原有递归下降程序中直接处理。
- 剩余部分为一类。在ast树生成之后，对ast树进行遍历，结合符号表的更新进行错误处理。例如：
    - c 未定义的名字
      在使用变量、常量或者函数时，查找符号表，如果未找到则报错。
    - f 无返回值的函数存在不匹配的return语句
      对于astBlock，在生成时会记录该block是int-fun，void-block，if，while或者普通情况下产生的block。当出现返回值不为null的return语句时，则向逐层向上寻找block类型，如果存在返回值非void的函数block则不报错。
    - g 有返回值的函数缺少return语句
      当存在返回值时会给语句本层函数block增加一条return记录，如果又出现了其他语句则会清除该记录。如果函数结束存在return记录则不报错。
    - m 在非循环块中使用break和continue语句
      向前逐层寻找while-block，如果找不到，则报错。

7.2.部分具体实现举例

e类错误：函数参数乐行不匹配。

- 调用函数的函数参数，存在四种情况：
    - void类型。函数参数位置为一个无返回值的函数时，参数类型为void
    - 常数，一维数组，二维数组。当参数位置为数组类型时，每存在一个下标，则维度减一。如fun(a[3])，如果a为一个二维数组，存在一个下标位置，则数组维度减一，为一个一维数组。如果参数位置直接为带运算的exp表达式，则直接为常数。
- 定义函数的函数参数，存在三种情况：
    - 常数
    - 一维数组
    - 二维数组

比对两者的类型，如果两者的类型相同，则不报错，否则报错。

8. 四元式

8.1. 四元式实现

四元式定义表

操作符	名称	op1	op2	op3	输出
ASSIGN	赋值	√		√	op3 = op1
SW	存值	√	√	√	op3[op2] = op1
LW	取值	√	√	√	op3 = op1[op2]
LABEL	标签	√			label op1:
SETRET	函数设置返回值	√			set return op1
GETRET	获得函数返回值			√	op3 = return
RETURN	函数返回	√			return
PUSH	输入函数参数	√		√	pushPara op1 to op3
PUSHADDR	输入函数参数地址	√	√	√	pushPara address(op1[op2]) to op3
CALL	调用函数	√			call func op1
BR	无条件跳转			√	goto op3
BEQZ	等于0跳转	√		√	if !op1 goto op3
BNEZ	不等于0跳转	√		√	if op1 goto op3
EQU	相等	√	√	√	op3 = (op1 == op2)
NEQ	不相等	√	√	√	op3 = (op1 != op2)
GRE	大于	√	√	√	op3 = (op1 > op2)
GEQ	大于等于	√	√	√	op3 = (op1 >= op2)
LES	小于	√	√	√	op3 = (op1 < op2)
LEQ	小于等于	√	√	√	op3 = (op1 <= op2)
READ	读入			√	op3 = getint
WRITESTR	写字符串	√			print op1
WRITEINT	写数字	√			print op1
ADD	加法	√	√	√	op3 = op1 + op2
SUB	减法	√	√	√	op3 = op1 - op2
MUL	乘法	√	√	√	op3 = op1 * op2
DIV	除法	√	√	√	op3 = op1 / op2
MOD	取模	√	√	√	op3 = op1 % op2
NEG	负数	√		√	op3 = -op1
NOT	取非	√		√	op3 = !op1
FUNC	函数	√			func op1

操作符	名称	op1	op2	op3	输出
CALLBEGIN	函数调用开始				funcBegin

在实现过程中，通过java的Enum类型维护

```
public enum QuaternionType {
    ASSIGN, SW, LW, LABEL, SETRET, GETRET, RETURN, PUSH, PUSHADDR,
    CALL, BR, BEQZ, BNEZ, EQU, NEQ, GRE, GEQ, LES, LEQ, NOT, READ,
    WRITESTR, WRITEINT, ADD, SUB, MUL, DIV, MOD, NEG, FUNC, CALLBEGIN
}
```

四元式生成

在第4部分生成的ast树上进行遍历。

- 对于ast的每一个节点，根据其类型，生成必要的四元式并按顺序加入到四元式集合中。
- 如果缺乏临时变量，则生成对应的临时变量。
- 如果产生的临时变量或者局部变量在后续的四元式生成中需要使用，则将该变量返回到ast树的上层节点，供新的四元式生成使用。

8.2. 操作数信息记录

变量记录与处理

- 对于所有的变量或常量（全局变量，局部变量与临时变量），全部加入ArrayList集中存储，并按照序号自增的方式对所有的常量或者变量重新命名，保证做出区分。
- 对于常量，由于前面的错误处理已经对给常量赋值的行为做了检查，因此，常量可以与变量做相同的地位处理，后面统一称为变量。
- 对于全局变量，除了记录到所有变量的集合之外，额外加入全局变量的集合。在后面的使用过程中，如果变量在该集合中，则为全局变量。
- 对于局部变量和临时变量，根据其属于的函数不同，分别加入到不同的集合中。根据变量是否在集合中和变量所在的不同集合，可以得出变量所在的函数。
- 全局变量，局部变量，临时变量，均会在属性上有所区分。

函数，跳转标签记录与处理

与变量与常量类似，函数和标签也会被分别加入到一个集合中进行存储。此外，函数和标签也会根据序号自加的方式进行重新命名。

8.3. 部分四元式实现举例

变量声明语句

- 如果变量为全局变量或者为常量，则不会出现在四元式中，只会被加入到全局变量的集合中。如果该全局变量或者常量存在初始值，则也会被一同记录。
- 如果变量为没有初值的局部变量，则不会出现在四元式中，只会被加入到局部变量的集合中。
- 如果变量为存在初值的局部变量，则会出现在四元式中。但是四元式中存在的语句仅为赋值语句，不会出现与声明有关的任何信息。同样的，该局部变量会被加入到局部变量的集合中。

与语句（或类似）

维护一个正确flag，如果可以判断flag的正确性，则直接结束并返回flag，否则，进行相应的调整。

伪代码如下：

```
leftTrueFlag = leftGen()
trueFlag = false
if leftTrueFlag = false goto end
rightTrueFlag = rightGen()
if rightTrueFlag = false goto end
trueFlag = true
end:
return trueFlag
```

判断语句

如果条件为误，跳转到else语句的block，如果正确，则继续执行直到完成if的block块（如果拥有else块，则需要跳转到else块结束位置）

如果拥有elseBlock:

```
trueFlag = condGen()
if trueFlag = false goto else
trueBlockGen()
goto end
else:
falseBlockGen()
end:
```

如果没有elseBlock:

```
trueFlag = condGen()
if trueFlag = false goto end
trueBlockGen()
end:
```

break、continue语句

对于一个循环函数，分别维护continue和break的label，并在break和continue出现的位置跳转到这两个标签。

下面为while语句的label位置

```
do {

...

continueLabel
}
while(cond)
breakLabel
```

8.4. 基本块分割

首先确定基本块的开始位置（第一条，跳转语句跳转到的位置，跳转语句的下一条语句）根据基本块的开始位置确定每个基本块对应的四元式语句。对于每一个基本块增加一个basicBlock类，通过基本块集合和四元式的属性完成基本块与编译程序的联系。

9. mips代码生成

9.1 mips实现

基本逻辑

mips以四元式为基本单元进行生成。遍历每一条四元式，根据四元式的类型进行对应mips代码的生成。

操作数

对于存在值的操作数，如果在寄存器中，则使用寄存器中的值。如果不在寄存器中，则向寄存器池申请新的寄存器（申请时做必要的回写）。

对于标签等操作数，则做相对应的处理。

同样需要注意全局变量等使用名字进行索引和函数参数使用地址进行索引的操作数。

寄存器池

寄存器池单独实现，并对外提供了接口。包括检查某个变量是否在寄存器中，为某个变量申请一个寄存器，检查某个变量是否需要回写（对于临时变量和部分局部变量等，可能不需要回写）。寄存器池中维护了s寄存器，t寄存器和a寄存器。具体的寄存器分配的选择将会在代码优化部分说明。

注释（强烈建议）

为了提高mips代码的可读性，在生成一段mips代码后，将会将四元式以#注释的形式输出到mips代码中。这样在debug中的时候可以快速定位对应位置。

9.2 mips生成实现举例

函数调用

函数调用分为多个四元式，依次为

```
CALLBEGIN
PUSH/PUSHADDR
...
CALL
```

当CALLBEGIN出现时，标志着函数调用即将开始，做好调用函数之前的准备。

PUSH四元式将函数参数push到调用函数使用空间的内存中，但是如果push的变量是前四个变量时，则不将变量push到内存中，只push到a寄存器中。

PUSHADDR的操作数中有基地址和偏移量。在这里，将会先使用mips的add语句计算出需要传入的地址，然后将该地址作为一个值push到调用函数的空间或者a寄存器中。

此外，选择在函数调用之前将sp指针移动到具体的位置，在调用函数结束只用，将sp指针调回原处。

数组操作

对数组进行操作时，有两个操作数，一个数组的基地址，一个目标变量的偏移量。

数组的基地址存在三种情况，全局变量（使用变量名称进行索引），局部变量（使用地址的数值进行索引），寄存器中的值（代表地址，如在a寄存器中）

数组的偏移值存在四种情况，全局变量，局部变量，在寄存器中，常数值。

根据上述情况的分类，计算出目标数值的值，并借此进行数组的存取操作。

```
boolean indexIsNumber;              //地址偏移是否为常数
String indexString;                 //地址偏移字符串
//op $v0
ConVarEle op = (ConVarEle) inOp;
String opReg = "$v0";
if (mGlobalVarArr.contains(op)) {
    //基地址为全局变量
    mips("la " + opReg + ", " + "var_" + op.name + "\n");
}
else {
    if (isFuncParamArr(op)) {
        //基地址为函数参数
        if ((opReg = registerPool.checkRegister(inOp)) == null) {
            opReg = "$v0";
            mips("lw " + opReg + ", " + op.address + "($sp)" +"\n");
        }
    }
    else {
        //其他情况
        mips("addiu " + opReg + ", $sp, " + op.address + "\n");
    }
}
//index $v1
if (index instanceof NumberEle) {
    //常数
    indexString = Integer.toString(((NumberEle) index).value * 4);
    indexIsNumber = true;
}
else {
    String tempRegister = null;
    if ((tempRegister = registerPool.checkRegister(index)) != null) {
        //在寄存器中
        mips("addu $v1, $0, " + tempRegister + "\n");
        mips("sll $v1, $v1, 2\n");
    }
    else if (mGlobalVarArr.contains(index)) {
        //是全局变量
        mips("lw $v1, " + "var_" + ((ConVarEle) index).name + "\n");
        mips("sll $v1, $v1, 2\n");
    }
    else {
        //局部变量
        mips("lw $v1, " + ((ConVarEle) index).address + "($sp)\n");
        mips("sll $v1, $v1, 2\n");
    }
    indexString = "$v1";
    indexIsNumber = false;
```

```
}
//合并
if (indexIsNumber) {
    return indexString + "(" + opReg + ")";
}
else {
    mips("addu $v0, " + opReg + ", $v1\n");
    return "($v0)";
}
```

10. 代码优化

代码优化应该以优化效果为主，需要在有限的时间内完成效果尽可能好的优化。我在优化的过程中，发现有些同学在没有实现寄存器分配，甚至s寄存器完全没有使用的情况下去做一些收益不是很高的优化，导致效果很好的优化没有时间完成。我认为这是得不偿失的。我主要选择了一些优化效果显著的来实现，并将会在下面说明每种优化的优化效果如何。

10.1 临时寄存器分配

opt分配

临时寄存器使用了opt的算法进行计算。

1. 对于申请寄存器的操作数，如果操作数已经在寄存器中了，则直接返回该寄存器。
2. 确定该操作数在基本块中的位置。
3. 从该操作数的位置向后扫描整个基本块，扫描的目标是在寄存器中的操作数的下次使用位置。在扫描的过程中，如果发现了某个寄存器中的操作数，则记录该位置（如果已经有更近的位置则不记录）。
4. 查看各个寄存器的最近距离，选择使用距离最远的操作数。
5. 如果被选择的操作数需要被存储则保存到内存中。

```
//初始化变量最短位置
//useNext为各个t寄存器的最近使用距离
int startDis = basicBlock.quaternions.size() + 10;
for (int i = 0; i <= 9; i++) {
    useNext.add(startDis);
}
//计算各变量最短位置
for (int i = index; i < basicBlock.quaternions.size(); i++) {
    Quaternion varQuaternion = basicBlock.quaternions.get(i);
    if (tRegisterSaved.contains(varQuaternion.op1) &&
useNext.get(getIndex(varQuaternion.op1)) == startDis) {
        useNext.set(getIndex(varQuaternion.op1), i);
    }
    if (tRegisterSaved.contains(varQuaternion.op2) &&
useNext.get(getIndex(varQuaternion.op2)) == startDis) {
        useNext.set(getIndex(varQuaternion.op2), i);
    }
    if (tRegisterSaved.contains(varQuaternion.op3) &&
useNext.get(getIndex(varQuaternion.op3)) == startDis) {
        useNext.set(getIndex(varQuaternion.op3), i);
    }
}
//计算所有的最远的寄存器
ArrayList<Integer> registerIndex = new ArrayList<>();
int maxDistance = 0;
```

```
for (int i = 0; i <= 9; i++) {
    if (useNext.get(i) > maxDistance) {
        registerIndex = new ArrayList<>();
        registerIndex.add(i);
        maxDistance = useNext.get(i);
    }
    else if (useNext.get(i) == maxDistance) {
        registerIndex.add(i);
    }
}
//registerIndex即为选择的t寄存器号
```

回写对象

应尽可能的减少回写，在下面的情况下可以不回写寄存器：

- 在进行回写时，如果参数为临时参数且不会再被使用了，则不会进行回写。
- 如果参数没有被修改过，即寄存器中的值与内存中的值相同，也不会回写。

当函数调用时，我选择将修改过的t寄存器全部回写，避免出现问题。

优化效果

在做完optl临时寄存器分配之后，读存内存的开销会大幅度减低。大约降低为原先双寄存器版本的1/5。

10.2 活跃变量分析

数据流图在课本上有详细的描述，在此不再赘述

对于每一个基本块，从前向后扫描，计算基本块的use和def。对于op1和op2，如果没有在use和def中，则将该操作数加入use。对于op3，如果没有在use和def中，则将该操作数加入def。对于某一条四元式，优先扫描op1和op2，随后扫描op3。

根据$in[B] = use \vee (out[B] - def[B])$，更新所有基本块的in，如果所有基本块的in都没有发生改变，则活跃变量分析结束。

10.2 全局寄存器分配

全局寄存器分配结合了引用计数算法和图染色算法。

引用计数

对于每一个变量，设置了一个权重属性用于记录该变量被引用的权重。

由于在本次代码实现过程中，没有对四元式做太多的优化，因此，直接在ast树部分计算了各个变量的引用数。

- 对于在非循环位置出现的变量，变量的引用数量增加1。
- 对于在循环体内部出现的变量，简单的认为每层循环将会给该位置的权重乘10。在乘完权重比例之后，将该权重加到变量的权重记录上。比如变量a在某个循环内出现了2次，该循环体嵌套了两层，则权重增加值为200。

对于全局寄存器分配时，对于权重更高的变量，需要优先考虑分配。对于权重过于小的变量，比如权重只有1，则可以考虑不分配全局寄存器。

图染色

根据活跃变量冲突可以构建冲突图，判断是否存在冲突的方式为判断某个变量在其他变量的定义处是否活跃。

对于冲突图，依次从冲突图中删除冲突边数小于s寄存器数的节点，并记录。如果不存在这样的节点，则优先删除冲突边数较多，引用计数权重较低的节点（对于计数权重远高于其他冲突点的节点，尽可能不删除该节点）。对于记录的节点顺序，反向赋s寄存器。并将最终的结果提供给寄存器池。

课本上的内容不够详细，建议参阅一部分论文作进一步的了解。由于个人最终实现的算法位置较为分散，因此不在这里详细介绍。本人参考了"基于网的寄存器分配冲突图研究"。

优化效果

全局寄存器可以大幅度降低局部变量的读取和保存次数。在竞速样例中，内存开销大概降低为不使用s寄存器的1/10 - 1/3。

10.3 叶子函数与a寄存器的使用

叶子函数是指在函数内部不分配栈空间，也不调用其它函数，也不存储非易失性寄存器，也不处理异常。在本次编译作业中，不存在运行中的异常。因此，叶子函数就是指不分配栈空间且不调用其他函数的函数。

具体实现方式：

1. 预处理与判断
    1. 在编译开始之前，将会对中间代码进行扫描，如果四元式中不含有函数，并且该函数不存在局部变量，参数个数小于等于4，则判定该函数为叶子函数，并进行记录。
    2. 此外，如果函数只是不调用其他函数，也同样进行记录，将会在后面进行一定的优化。
2. 实际处理方式
    1. 对于叶子函数，将不会给其分配空间，即sp不需要进行移动，在计算过程中只需要对a寄存器进行操作。
    2. 对于不调用其他函数的函数，将不会再调用函数时进行ra寄存器的存储。此外，在函数内部的语句中，操作数为函数参数时，将会直接使用a寄存器中存储的值或者直接更改a寄存器中的值。这样操作可以省下对函数参数的大量读取存入操作。

下面的代码是判断某个函数里面是否调用了其他的函数

```
for (Quaternion varQuaternion: mQuaternions) {
    if (varQuaternion.operation == QuaternionType.FUNC) {
        funcEle = (FuncEle) varQuaternion.op1;
        funcEle.isLeafFunc = true;
    }
    if (varQuaternion.operation == QuaternionType.CALLBEGIN
            || varQuaternion.operation == QuaternionType.WRITEINT
            || varQuaternion.operation == QuaternionType.WRITESTR) {
        if (funcEle != null) {
            funcEle.isLeafFunc = false;
        }
    }
}
```

优化效果

在部分测试样例中，读存内存和other类型指令都有大幅度的降低，相对之前有1半以上的降幅。

10.4 运算强度削弱

运算强度削弱主要是除法的运算强度削弱。

除法

除法运算强度削弱部分主要参考了论文“Division by Invariant Integers using Multiplication”。

主要思想为将除数为常数的语句改为左移、右移、乘法等语句块。由于除法的开销较大，即使更改后语句数量增加，总体时间开销仍然减少不少。

论文中使用了unsigned long等数据类型，而在java中不存在这些数据类型。

因此在实现过程中，如果类型在java中拥有，则直接使用。如果某个类型在java中不存在，则使用BigInteger等数据类型进行模拟。

论文中的思路为使用chooseMultiplier计算除法转换的算子，具体函数如下：

```
private static DivideEle chooseMultiplier(BigInteger d, int p) {
    assert d.compareTo(new BigInteger("0")) > 0;
    int l = d.subtract(new BigInteger("1")).bitLength();
    int sh = l;
    BigInteger low = ((new BigInteger("1")).shiftLeft(32 + l)).divide(d);
    BigInteger high = (new BigInteger("1")).shiftLeft(32 + l);
    high = (high.add((new BigInteger("1")).shiftLeft(32 + l - p))).divide(d);
    while ((low.shiftRight(1).compareTo(high.shiftRight(1)) < 0 && sh > 0)) {
        low = low.shiftRight(1);
        high = high.shiftRight(1);
        sh--;
    }
    return new DivideEle(high, sh, l);
}
```

在算子计算完成之后，根据各种类型的不同，分四种情况生成mips函数。

```
 public static ArrayList<String> generateSignedDivision(String reg1, int d,
String reg3) {
        ArrayList<String> outString = new ArrayList<>();
        long dAbs = d;
        dAbs = Math.abs(dAbs);
        int q;
        DivideEle divideEle = chooseMultiplier(new
BigInteger(Integer.toString(d)).abs(), 31);
        BigInteger m = divideEle.m;
        int sh = divideEle.sh;
        int l = divideEle.l;
        if (dAbs == 1) {
            outString.add("addu " + reg3 + ", $0, " + reg1 + "\n");
        }
        else if (dAbs == (1L << l)) {
            outString.add("sra " + reg3 + ", " + reg1 + ", " + (l - 1) + "\n");
            outString.add("srl " + reg3 + ", " + reg3 + ", " + (32 - l) + "\n");
            outString.add("addu " + reg3 + ", " + reg3 + ", " + reg1 + "\n");
            outString.add("sra " + reg3 + ", " + reg3 + ", " + l + "\n");
```

```
        }
        else if (m.compareTo(new BigInteger(Long.toString(1L << 31))) < 0)  {
            outString.add("mul " + reg3 + ", " + reg1 + ", " + m + "\n");
            outString.add("mfhi " + reg3 + "\n");
            outString.add("sra " + reg3 + ", " + reg3 + ", " + sh + "\n");
            outString.add("slt " + "$v1" + ", " + reg1 + ", " + "$0" + "\n");
            outString.add("add " + reg3 + ", " + reg3 + ", " + "$v1" + "\n");
        }
        else {
            outString.add("mul " + reg3 + ", " + reg1 + ", " + m.subtract(new
BigInteger("1").shiftLeft(32)) + "\n");
            outString.add("mfhi " + reg3 + "\n");
            outString.add("add " + reg3 + ", " + reg3 + ", " + reg1 + "\n");
            outString.add("sra " + reg3 + ", " + reg3 + ", " + sh + "\n");
            outString.add("slt " + "$v1" + ", " + reg1 + ", " + "$0" + "\n");
            outString.add("add " + reg3 + ", " + reg3 + ", " + "$v1" + "\n");
        }
        if (d < 0) {
            outString.add("sub " + reg3 + ", " + "$0" + ", " + reg3 + "\n");
        }
        return outString;
    }
}
```

取模

对于mod运算，采用先计算商，再通过公式$余数 = 被除数 - 除数 \times 商$的公式计算。

乘法

对乘法的运算强度削弱主要在与将一个乘数为2^n的乘法优化为左移运算。事实上，除了四元式中本身存在的复合该情况的乘法，在上面模运算的过程中，也会产生一定的乘法，通过这种方式，可以略微降低乘法的时间开销。

优化效果

竞速样例中，绝大部分除法样例的除数都为常数，因此对于部分点的效果非常好。部分测试样例点的除法条数归0，相对应的乘法和other操作数也有一定的增加，不过在开销上某些点甚至降低到了之前了几十分之一。

对于乘法的优化，由于局限性过高，导致优化掉的条数不多，不过仍然有所收益。

10.5 四元式合并

在本次优化中，对四元式的优化不多，只进行了简单的四元式合并。

合并的主要形式是删除无用的四元式。

比如将一个操作数加0赋值给另外一个操作数，显然这条语句是无效的。不过在四元式的生成过程中存在不少这样的无效生成，将类似的语句找出并删除可以有不小幅度的提升。

具体实例：

```
ADD var1, var2, var3
ADD var3, 0, var4
```

如果var3是临时变量，则可以直接修改为

```
ADD var1, var2, var4
```

但是如果var3是局部变量则不可以直接修改。

优化效果

所有测试点有较小幅度的速度优化。

10.6 常量带入与预计算

更建议实现常量传播。由于时间等原因，本次编译实验没有实现常量传播，只简单的做了优化。

将const类型的数据直接直接带入了编译程序并进行了求值与化简。

具体的做法为在ast树上进行遍历时，如果某个变量为const常量，则直接将该常量带入ast树化简。

此外，在ast树上，如果通过子节点能够计算出父节点时，也会直接对父节点进行化简。

例如：

```
const int a[2][2] = {{1,2},{3,4}};
func(int c[][a[0][0] + a[0][1]]) {
...
}
```

在上面的例子中，就可以直接计算出数组参数第二维的大小。

优化效果

由于是在ast树部分早就实现的，因此无法判断优化幅度。

10.7 do-while循环

将while修改为了do-while，能够减少跳转

```
if !cond goto end
loop:
exp
if cond goto loop
end
```

优化效果

部分使用while循环较多的点，branch开销降低为原先的一半。

10.8 其他优化

部分优化内容或许不标准，只是为了提升编译器效果而做的优化

1. 给未分配全局寄存器的局部变量分配t寄存器。（部分点有一定幅度效果提升）
2. 给不在a寄存器中的函数参数分配s寄存器。（部分使用函数参数较多的点有几倍的速度提升）
3. 给使用较多的全局变量永久性分配寄存器。（由于没做常量传播，这条优化在部分点效果明显）
4. 其他细小合并与化简，如指令的选择，部分预处理等等。（小幅度提升，较多叠加后相对可观）

10.9 优化总结

由于个人时间原因，完成的代码优化数量较少，但是大部分优化效果较好。

如果还有时间，则比较想做死代码删除，循环体的各种优化，函数内联，以及当时感觉收益不高就没有去做的dag图。

总体来说，优化效果较好，截至12.21日，竞速排名大部分样例在10-20之间。

11. 总结

在编译实验的过程中，难免会碰到迷茫与痛苦的时候，不知道架构怎么实现，不知道bug在哪里，完全没有头绪。但是当我们一路咬着牙走过来，才发现所有的一切都被顺利解决了，也获得了一个自己相对满意的结果。人们总是这样想，如果现在的我回去重新做这件事的话肯定会做的更好。不过没关系，一路坚持下来，你必定会发现，自己做的也不错！