代码生成阶段设计文档

一、编码之前的设计

代码生成一

> 代码生成一：本次作业是为了让同学们尽快实现一个完整的编译器，测试程序中仅涉及常量说明、变量说明、读语句、写语句、赋值语句，无函数定义及调用，无数组声明及引用。

正如在本次作业说明中提到的这样，本次作业的主要目标是为了尽快实现一个完整的编译器。在涉及较少语句的基础上，让我们能够设计实现整个编译器的大致步骤，即根据文法规则及语义约定，采用自顶向下的语法制导翻译技术，进行语义分析并生成目标代码。

本次作业主要分为两个部分，分别是在语法分析过程中生成中间代码，然后根据相应的指令体系将中间代码翻译成汇编代码，在这里我选择的是mips指令。

本次作业中，难度较大的部分在于表达式的相关转化。由于<因子>成分中需要调用<表达式>子程序，所以我们在分析<表达式>成分的时候很可能会出现循环调用的情况，需要仔细考虑。

代码生成二

> 代码生成二：本次作业的测试程序覆盖所有语法成分。

在前一次的基础上，代码生成二包含了所有的语法成分，也就是说在优化之外，完成了一个完整的编译器。

本次作业的大致结构和第一次一样，都是分为生成中间代码和生成mips指令两个部分。

相较于第一次作业，本次作业主要加入了数组、函数、条件语句、循环语句、情况语句等成分。数组相关的主要是内存地址的计算，而函数、循环语句等成分主要考虑的是标签和指令之间的跳转。

二、编码之后对设计的修改

阶段一 生成中间代码

变量和函数定义

在生成中间代码阶段，我使用到的变量和函数如下。

```
// 以下为外部引用的变量和函数
extern ofstream midCodeFile;                                    // 中间代码输出文件
extern vector <struct midCode> midCodes;                        // 保存中间代码的容器
extern map<string, map<string, struct sym>> symList;            // 符号表

// 以下为内部定义的变量和函数
void insertMidCode(midOp op, string result, string left, string right);
void insertMidCode(midOp op, string result, string left, string right, string backup);
void outputMidCodes();
void outputMidCode(struct midCode m);
```

关于中间代码的格式，我使用的是课程组推荐的中缀表达式的结构，包含了结果、左操作数、操作符、右操作数和备用操作数四个成分。其中的备用操作数仅用于二位数组的相关操作。

```
struct midCode {
    string result;      // 结果
    string left;        // 左操作数
    midOp op;           // 运算符
    string right;       // 右操作数
    string backup;      // 备用操作数
};
// midOp是自己定义的包含中间代码操作符的枚举类型
```

具体实现过程

在这一阶段，我们要做的事主要采用自顶向下的语法制导翻译技术，进行语义分析并生成中间代码。

代码生成一的作业中仅涉及<常量说明>、<变量说明>、<读语句>、<写语句>、<赋值语句>五种成分，我们首先对这些元素进行一定分析：

- <常量说明>：将符号保存在符号表中，并且保存相应的常量值。
- <变量说明>：将符号保存在符号表中，特别的对于有初始值的变量的操作类似于常量，只不过还可以进行赋值等操作。
- <读语句>：根据读语句中的标识符，通过mips指令中的系统调用读取相关数据并保存到对应的内存地址。
- <写语句>：根据写语句中需要数据的字符串和标识符，通过mips指令中的系统调用输出相关数据。
- <赋值语句>：取出操作数的值，并保存到结果的内存地址即可。

除了这些成分，中间代码生成过程中难度较大的部分是关于<表达式>的中间代码生成过程。

在这里我主要使用了符号栈的思想，当读取到运算符时，弹出并检查当前在栈顶的两个元素，如果两者都是符号（常量或者变量），则生成一个新的中间临时变量作为操作结果并添加到符号表中；倘若两者中的左操作数为中间临时变量，则将运算后的结果直接保存到该中间临时变量即可。一直这样操作下去，最后将符号栈中剩余的最后一个元素，就是保存了该表达式结果的标识符。返回该标识符即可、

代码生成二的作业中则包含了所有的语法成分，我分析之后发现相较于第一次作业，主要增加了如下五种成分的语义分析：

- 数组相关：对于数组的取用，主要难度在于mips代码中对数组元素地址的计算，需要在数组地址的地址的基础上加入元素的偏移。
- 函数相关：函数相对而言是较为容易实现的，因为main本身就是一个特殊的函数，我们只需要在函数调用语句的部分加入跳转指令，并在<返回语句>中跳转回跳转前的地址即可。
- <条件语句>：首先将条件语句中的条件转化为一个跳转指令，同时对if和else分别生成两个标签`ifLabel`和`elseLabel`，前者用于条件不符合时跳转至else语句或者跳出该条件语句，后者则用于条件符合时跳出else语句。

```
    string ifLabel, elseLabel, tmpCondition;

    checkSymbolandGetSym(IFTK, '0');         // if
    checkSymbolandGetSym(LPARENT, '0');      // '('
    getCondition(tmpCondition);              // <条件>
    checkSymbolandGetSym(RPARENT, 'l');      // ')'

    ifLabel = genLabel();                    // 创建 if 标签
    insertMidCode(BZ, ifLabel, tmpCondition, "");
    getStatement();                          // <语句>

    if (symbol == ELSETK) {
        getsym();
        elseLabel = genLabel();                      // 创建 else 标签
        insertMidCode(GOTO, elseLabel, "", "");      // 无条件跳转到标签 else
        insertMidCode(LABEL, ifLabel, "", "");       // 在 else 语句开始处插入 if 标签
        getStatement();
        insertMidCode(LABEL, elseLabel, "", "");     // 在 else 语句结束处插入 else 标签
    }
    else {
        insertMidCode(LABEL, ifLabel, "", "");       // 没有 else 在语句最后插入 if 标签
    }
```

- <循环语句>：循环语句包含`while`和`for`两种情况。
  - `while`语句：在循环开始处进行条件判断是否需要跳出循环，在循环语句最后插入跳转指令，返回while语句的条件判断处。这里使用了两个标签`beginLabel`和`endLabel`，分别用于跳转回循环开始处，和条件不满足时跳出循环语句。

```
string beginLabel, endLabel, tmpCondition;

beginLabel = genLabel();
insertMidCode(LABEL, beginLabel, "", "");

checkSymbolandGetSym(LPARENT, '0');              // '('
getCondition(tmpCondition);                      // <条件>
checkSymbolandGetSym(RPARENT, 'l');              // ')'

endLabel = genLabel();
```

```
insertMidCode(BZ, endLabel, tmpCondition, "");  // 不满足条件(result==0)的话 跳转到labf

getStatement();                                 // <语句>

insertMidCode(GOTO, beginLabel, "", "");        // 执行了一次循环体 无条件回到labr
insertMidCode(LABEL, endLabel, "", "");         // 设置labf 用于结束循环
```

- `for`语句：for语句的大致结构与while语句相似，主要是在for语句开始前增加了对循环变量的初始化操作，以及循环语句的最后需要单独增加修改循环变量的语句。与while语句相同的是，这里使用了两个标签`beginLabel`和`endLabel`，分别用于跳转回循环开始处，和条件不满足时跳出循环语句。

```
int tmpExpressionType, tmpAdditionOperator, tmpStep;
string beginLabel, endLabel, tmpIdentifier, tmpExpression, tmpCondition;
string leftIdentifier, rightIdentifier;

checkSymbolandGetSym(LPARENT, '0');                 // '('

// <标识符>=<表达式>
getIdentifier(tmpIdentifier);                       // <标识符>
checkUndefinedIdentifier(tmpIdentifier);            // 检查是否有未定义的名字
checkSymbolandGetSym(ASSIGN, '0');                  // '='
tmpExpressionType = 2;
getExpression(tmpExpressionType, tmpExpression);    // <表达式>
insertMidCode(ASSIGNOP, tmpIdentifier, tmpExpression, "");

checkSymbolandGetSym(SEMICN, 'k');                  // ';'

beginLabel = genLabel();
insertMidCode(LABEL, beginLabel, "", "");           // 在条件判断前插入 beginLabel 标签
getCondition(tmpCondition);                         // <条件>
endLabel = genLabel();
insertMidCode(BZ, endLabel, tmpCondition, "");      // 不满足条件跳到 endLabel 结束循环

checkSymbolandGetSym(SEMICN, 'k');                  // ';'

// <标识符>=<标识符>(+| -)<步长>
getIdentifier(leftIdentifier);                      // <标识符>
checkUndefinedIdentifier(leftIdentifier);           // 检查是否有未定义的标识符
checkSymbolandGetSym(ASSIGN, '0');                  // '='
getIdentifier(rightIdentifier);                     // <标识符>
checkUndefinedIdentifier(rightIdentifier);          // 检查是否有未定义的标识符
getAdditionOperator(tmpAdditionOperator);           // (+|-)
getStep(tmpStep);                                   // <步长>
checkSymbolandGetSym(RPARENT, 'l');                 // ')'
getStatement();                                     // <语句>

```

```
insertMidCode(tmpAdditionOperator == PLUS ? PLUSOP : MINUOP,
leftIdentifier, rightIdentifier, int2string(tmpStep));//增加步长
insertMidCode(GOTO, beginLabel, "", "");            // 回到
beginLabel 进行条件判断
insertMidCode(LABEL, endLabel, "", "");             // 插入
endLabel 结束循环
```

- <情况语句>：对于情况语句，我采用了比较笨的办法，就是将每个case转化为了一个if语句进行分析，如果case语句中的变量和主变量不同则跳过该case语句，在每个case语句的最后增加一个跳转出整个switch语句的指令，从而完成了整个情况语句的代码构建。

```
// 将情况子语句当作if语句处理
resultExpression = genTmp();
insertSymbol(resultExpression, 1, 0);
leftExpression = tmpExpression;
rightExpression = tmpConstantType == 0 ? int2string(tmpInteger) :
tmpChar;
insertMidCode(EQLOP, resultExpression, leftExpression,
rightExpression);

// 若不相同则进行跳转
string caseLabel = genLabel();
insertMidCode(BZ, caseLabel, resultExpression, "");

checkSymbolandGetSym(COLON, '0');           // ':'
getStatement();                             // <语句>

insertMidCode(EXITCASE, "", "", "");
insertMidCode(GOTO, endLabel, "", "");      // 无条件跳转到END
insertMidCode(LABEL, caseLabel, "", "");
```

阶段二 生成mips代码

变量和函数定义

在生成mips代码阶段，我使用到的变量和函数如下。

```
// 以下为外部引用的变量和函数
extern ofstream mipsCodeFile;
extern map<string, map<string, struct sym>> symList;
extern vector <struct midCode> midCodes;
extern vector <struct mipsCode> mipsCodes;

vector<string> stringList;          //保存所有的字符串

string curFuncName = "";            // 记录当前所在函数名称
extern int string2int(string s);
extern string int2string(int t);

// 以下为内部定义的变量和函数
void generateMipsCodes();
void loadValue(string symbolName, string regName, bool gene, int& va, bool&
get);
void storeValue(string symbolName, string regName);
```

```
void insertMipsCode(mipsOp op, string result, string left, string right,
int imm);
void insertMipsCode(mipsOp op, string result, string left, string right);
void outputMipsCodes();
void outputMipsCode(struct mipsCode mc);
```

关于mips代码的格式，我采用的是类似于mips指令的结构，包含了指令、结果、左操作数、右操作数和立即数。

```
struct mipsCode {
    mipsOp op;          // 操作
    string result;      // 结果
    string left;        // 左操作数
    string right;       // 右操作数
    int imm;            // 立即数
};
// mipsOp是自己定义的包含mips指令的枚举类型
```

具体实现过程

在我的编译器设计中，代码逻辑上的跳转等已经在生成中间代码的过程中完成了设计，所以生成mips指令过程的主要工作就是为中间代码寻找对应的mips指令进行翻译即可。

从中间代码到mips代码，最大的变化就是运算的操作数从无限的变量到有限的寄存器中。因此我们在进行相应的运算之前需要将符号表或者内存地址中的值取出到寄存器中，并在运算结束后保存到内存地址中。在本次作业中，我没有对寄存器的使用进行优化，而是采用了每条语句后还原的思路，保证每条语句之间的寄存器使用互不影响。

从上述描述中，不难发现，生成mips指令阶段的主要任务就是在匹配相应的mips指令的基础上分配内存地址和寄存器。

我在这里主要分析函数调用和数组取用两个过程进行分析。

- 函数调用：函数调用的过程首先将相关变量（包括函数参数、变量和生成中间代码过程中产生的临时变量）、`$ra`和`$fp`寄存器保存到栈中，之后就可以修改栈指针并跳转至相应的标签。

```
paramSize = functionParameterCount[mc.result];
while (paramSize) {
    paramSize--;
    midCode tmpMc = pushOpStack.top();
    pushOpStack.pop();
    loadValue(tmpMc.result, "$t0", true, va, get1);
    insertMipsCode(sw, "$t0", "$sp", "", -4 * paramSize);
}
insertMipsCode(addi, "$sp", "$sp", "", -4 * symList[PROGRAM]
[mc.result].length - 8);
insertMipsCode(sw, "$ra", "$sp", "", 4);
insertMipsCode(sw, "$fp", "$sp", "", 8);
insertMipsCode(addi, "$fp", "$sp", "", 4 * symList[PROGRAM]
[mc.result].length + 8);
insertMipsCode(jal, mc.result, "", "");
insertMipsCode(lw, "$fp", "$sp", "", 8);
insertMipsCode(lw, "$ra", "$sp", "", 4);
insertMipsCode(addi, "$sp", "$sp", "", 4 * symList[PROGRAM]
[mc.result].length + 8);
```

- 数组元素：数组元素的取用主要是要进行地址的计算，这里以获取二维数组中的元素为例，先将两个下标分别保存至 `$t0` 和 `$t1` 寄存器，之后根据 `$t0 * column($t3) + $t1` 计算出该数组元素相对于数组地址的偏移，进而找出该数组元素在内存地址中位置。

```
case GETARRAY2: {
    //mc.result << " = " << mc.left << "[" << mc.right << "]" << "[" << mc.backup << "]" << endl;
    get1 = false; loadValue(mc.right, "$t0", true, va, get1);
    get2 = false; loadValue(mc.backup, "$t1", true, va2, get2);

    addr = symList[curFuncName][mc.left].addr;          // 该数组的起始点地址
    column = symList[curFuncName][mc.left].columns;     // 该数组的每一行元素数量

    insertMipsCode(addi, "$t2", "$fp", "", -4 * addr);  // 获取该数组所在地址

    // $t0 * column($t3) + $t1
    insertMipsCode(li, "$t3", "", "", column);
    insertMipsCode(mult, "$t0", "$t3", "");             // index * column
    insertMipsCode(mflo, "$t0", "", "");
    insertMipsCode(add, "$t0", "$t0", "$t1");           // + $t1
    insertMipsCode(sll, "$t0", "$t0", "", 2);

    insertMipsCode(sub, "$t2", "$t2", "$t0");           // 获取该元素所在地址
    insertMipsCode(lw, "$t1", "$t2", "", 0);
}
```