代码优化的方法及遇到的困难

我在代码优化的过程中，主要使用的方法是中间代码的简化、全局寄存器的分配和计算的优化。

中间代码的简化

- 如果在中间代码中，遇到跳转指令的标签是下一条语句，则删除跳转指令，如图所示：

```
if (preMCode.getOp().equals("GOTO")&&curMCode.getOp().equals("LABEL")&&preMCode.getArg1().equals(curMCode.getArg1()))
    i--;
    GlobalVariable.quadruples.remove(i);// 删除跳转
}
```

- 我在代码生成实验中，采取保存数据的方法是：每生成一个中间数据或最终数据，都将其存入内存中，知道下次使用再将其从内存中取出。这种方法虽然简单准确，但也因为多次从内存中存取数据导致速度变慢。因此，我进行了消除表达式产生的冗余变量，如图所示：

```
// 消除表达式赋值产生的冗余变量
//$T0=i*2
//$T1=$T0
//合并为$T1=i*2
```

  通过图示中的合并，就减少了从内存中存取中间变量$T0的时间，对中间代码进行了优化。

  当然，在遇到数组时，需要重新进行讨论（赋值方式不一样）。与此同时，因为在中间表达式中我将二维数组当作一维数组进行处理，因此这里仅需讨论一维数组的情况。

```
// 消除表达式赋值产生的冗余变量(左边是数组)
//$T0=a+1
//b[0]=$T0
//合并为b[0]=a+1
```

- 早在中间代码生成时，我就有意识的优化循环方式，使其简单化，具体实现方式如下：

```
while begin
goto loop
loopBegin:
<stmt>
loop:
if cond goto loopBegin
loopEnd:
while end
```

全局寄存器的分配

全局寄存器的分配是我这次优化中花费时间最长、遇到bug最多的部分，为此，我甚至重写了mips生成函数，真是让我欲仙欲死。

在代码生成中，我只用到了四个寄存器$t0-$t4，用来存放中间变量，一旦中间变量产生了，我会立即将其存入内存中，这也势必会造成大量的内存存取，拖慢运行时间，因此我设计了一个寄存器池进行其他寄存器的分配。我们做出约定$t4-$t9为保存临时变量的寄存器，$s0-$s7为保存局部变量的寄存器。我们会在中间代码生成后，mips代码生成前，进行一次中间代码的扫描，确定可以分配的局部变量并进行标记，以便在mips代码生成时分配寄存器。**每次遇到函数时，会将寄存器池清零，重新进行寄存器的分配。**在跳转前，会将被使用的寄存器的值存起来，如图所示：

```
        // 把t寄存器中的值存入符号表
        for (int i = 4; i < registerPool.gettRegList().size(); i++) {
            if (registerPool.gettRegList().get(i) == 1) {//该寄存器被使用
//                Symbol symbolTemp = new Symbol(registerPool.gettRegMap().get(i), "VAR");
//                GlobalVariable.symbolTable.addSymbol(symbolTemp);
//                store(registerPool.gettRegMap().get(i), "$t" + i);
                Symbol symbol1 = new Symbol( name: "$TRegister" + i, type: "VAR");
                GlobalVariable.symbolTable.addSymbol(symbol1);
                store( arg: "$TRegister" + i, valueReg: "$t" + i);
                quadruple.gettRegList().add("$TRegister" + i);
            }
        }
        // 把s寄存器中的值存入符号表
        for (int i = 0; i < registerPool.getsRegList().size(); i++) {
            if (registerPool.getsRegList().get(i) == 1) {
                Symbol symbol1 = new Symbol( name: "$SRegister" + i, type: "VAR");
                GlobalVariable.symbolTable.addSymbol(symbol1);
                store( arg: "$SRegister" + i, valueReg: "$s" + i);
                quadruple.getsRegList().add("$SRegister" + i);
            }
        }
```

而在函数调用结束后，会将该值重新赋值到寄存器中，如图所示：

```
        //函数调用完毕，把s寄存器加载出来
        for (int i = quadruple.getsRegList().size() - 1; i >= 0; i--) {
            String name = quadruple.getsRegList().get(i);
            load(name, dstReg: "$s" + i);//因为寄存器没有被清除，所以改变值就行，不用重新激活寄存器（mapsReglist）
            GlobalVariable.symbolTable.removeSymbol(name);
            //获取原来的栈偏移量（去掉ra后的）
            blockOffset = GlobalVariable.symbolTable.getTableList().get(GlobalVariable.symbolTable.getLevel()).getStackOffset();
            GlobalVariable.symbolTable.getTableList().get(GlobalVariable.symbolTable.getLevel()).setStackOffset(blockOffset - 4);
        }
        //函数调用完毕，把t寄存器加载出来
        for (int i = quadruple.gettRegList().size() - 1; i >= 0; i--) {
            String name = quadruple.gettRegList().get(i);
            char[] str=name.toCharArray();
            int num=Integer.parseInt( s: str[str.length-1]+"");
            load(name, dstReg: "$t" + num);
            GlobalVariable.symbolTable.removeSymbol(name);
            //获取原来的栈偏移量（去掉ra后的）
            blockOffset = GlobalVariable.symbolTable.getTableList().get(GlobalVariable.symbolTable.getLevel()).getStackOffset();
            GlobalVariable.symbolTable.getTableList().get(GlobalVariable.symbolTable.getLevel()).setStackOffset(blockOffset - 4);
        }
```

如此，可以通过寄存器的使用，大大减少内存的访问，加快程序运行速度。

计算的优化

- 当遇到含有加减乘除模的计算式时，会直接进行计算，然后再生成计算后的中间代码，减少了mips代码的运行时间（这个真的能快很多）。
- 当遇到乘法且包含2的幂次时，进行相关的计算优化（左移），加快运算速度，如图所示：

```
    public void mulOpt(String t, String t1, String arg2) {//乘法2的幂次优化(arg2为立即数)
        int imm = Integer.parseInt(arg2);
        int absImm = Math.abs(imm);
        if ((absImm & (absImm - 1)) == 0) { //是不是2的n次
            int leftShift = (int) (Math.log(absImm) / Math.log(2));
            mips.add("sll " + t + ", " + t1 + ", " + leftShift);
            if (imm < 0) {
                mips.add("subu " + t + ", $0, " + t);
            }
        } else {
            String t2 = load(arg2);
            mips.add("mul " + t + ", " + t1 + ", " + t2);
        }
    }
```

- 当遇到除法时，也可进行除法的相关计算优化。