设计文档

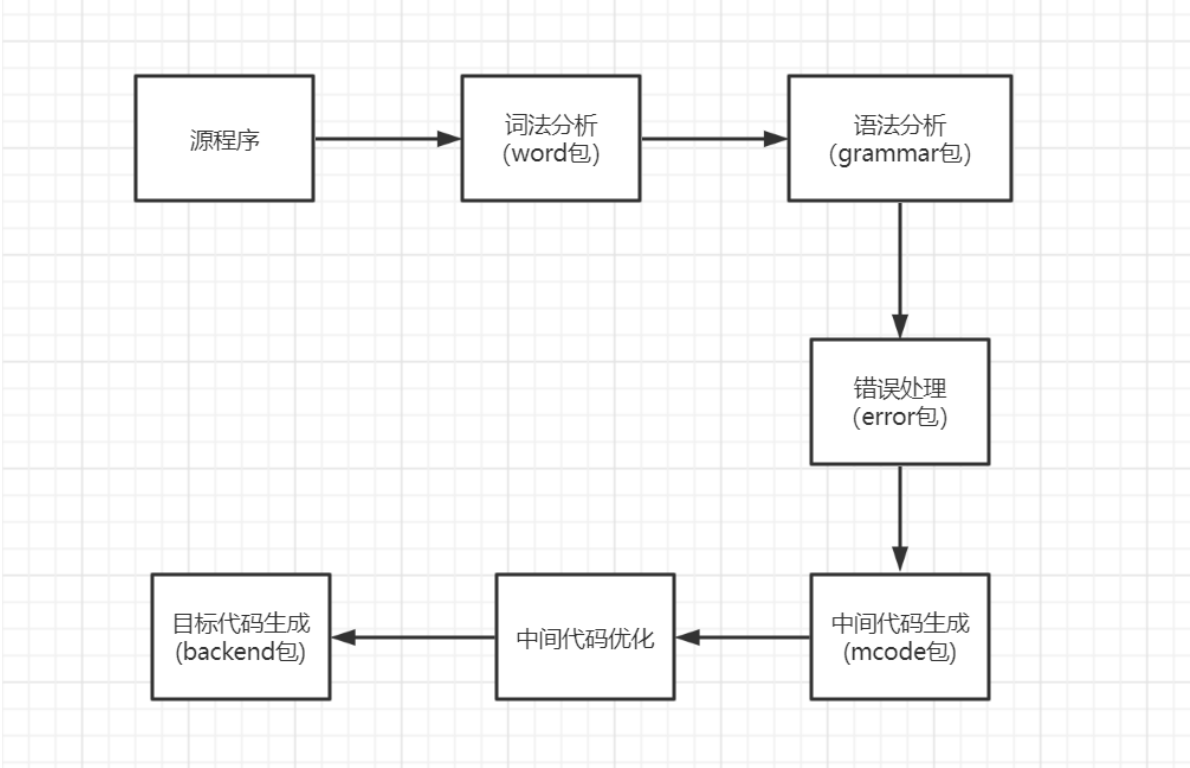
作者：19374242 高远

一、参考编译器

本编译器除代码生成部分外未参考已有的开源编译器，编译器中所使用的方法大都来自ppt和狼书。代码生成部分参考了课程组所提供的中间代码格式，也阅读了课程组所提供的教学编译器源代码，但总而言之，本编译器可以说完全是由我自行设计而成，也通过了所有的测试点，并在竞速排序中取得了令我比较满意的成绩。

二、总体架构

首先展示下我的编译器实现的主要步骤：



- 词法分析：读取源程序，分析得到字符实体串
- 语法分析：分析读取到的字符实体串，进行语法分析。
- 错误处理：在语法分析的过程中识别常见的错误并进行处理。
- 中间代码生成：因为我在语法分析时生成的语法树不够完善，因此在中间代码生成时我需要重新生成语法树，分析语法树，并在符号表的帮助下生成中间代码。
- 中间代码优化：对中间代码进行适当的优化。
- 目标代码生成：将中间代码生成相应的Mips代码

将compiler类设定为文件入口，IOTool类为输出工具类。

三、词法分析

词法分析最为简单，我定义了一个名为WordEntity的实体类对字符实体串进行存储，包含其单词名称、类别码和所在行数，按照课程组所给的词法分析类别码进行判断，如图：

单词名称	类别码	单词名称	类别码	单词名称	类别码	单词名称	类别码
Ident	IDENFR	!	NOT	*	MULT	=	ASSIGN
IntConst	INTCON	&&	AND	/	DIV	;	SEMICN
FormatString	STRCON	||	OR	%	MOD	,	COMMA
main	MAINTK	while	WHILETK	<	LSS	(	LPARENT
const	CONSTTK	getint	GETINTTK	<=	LEQ	)	RPARENT
int	INTTK	printf	PRINTFTK	>	GRE	[	LBRACK
break	BREAKTK	return	RETURNTK	>=	GEQ	]	RBRACK
continue	CONTINUETK	+	PLUS	==	EQL	{	LBRACE
if	IFTK	-	MINU	!=	NEQ	}	RBRACE
else	ELSETK	void	VOIDTK				

具体分析方法如下：

- 遇到空白字符则跳过，遇到换行符则行数增加1.
- 遇到数字则不断读取，直到下一个读到的不是数字。
- 遇到下划线或字母则不断读取，直到下一个读到的不是数字或下划线或字母，读完后，对所得到的字符串进行类别判断。
- 遇到特殊字符则直接对其进行类别判断（分类讨论是否是由多个特殊字符组成）
- 遇到注释则将其去除。

四、语法分析

语法分析较为简单，我定义了一个名为GrammarTreeEntity的实体类来存储语法树。

首先改写文法，消除左递归，采用递归下降分析法，对文法中除BlockItem, Decl, BType外的所有非终结符编写分析程序，并根据当前的词法成分，以及一定的超前扫描，判断接下来是什么语法成分，并调用相应的语法分析程序，在每个语法分析程序结束时输出该语法成分的名称即可完成题目要求的输出。具体语法规则如下图所示：

```
编译单元    CompUnit → {Decl} {FuncDef} MainFuncDef  // 1.是否存在Decl 2.是否存在FuncDef
声明  Decl → ConstDecl | VarDecl // 覆盖两种声明
常量声明    ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';' // 1.花括号内重复0次 2.花括号内重复多次
基本类型    BType → 'int' // 存在即可
常数定义    ConstDef → Ident { '[' ConstExp ']' } '=' ConstInitVal  // 包含普通变量、一维数组、二维数组共三种情况
常量初值    ConstInitVal → ConstExp
    | '{' [ ConstInitVal { ',' ConstInitVal } ] '}' // 1.常表达式初值 2.一维数组初值 3.二维数组初值
变量声明    VarDecl → BType VarDef { ',' VarDef } ';' // 1.花括号内重复0次 2.花括号内重复多次
变量定义    VarDef → Ident { '[' ConstExp ']' } // 包含普通变量、一维数组、二维数组定义
    | Ident { '[' ConstExp ']' } '=' InitVal
变量初值    InitVal → Exp | '{' [ InitVal { ',' InitVal } ] '}'// 1.表达式初值 2.一维数组初值 3.二维数组初值
函数定义    FuncDef → FuncType Ident '(' [FuncFParams] ')' Block // 1.无形参 2.有形参
主函数定义   MainFuncDef → 'int' 'main' '(' ')' Block // 存在main函数
函数类型    FuncType → 'void' | 'int' // 覆盖两种类型的函数
函数形参表   FuncFParams → FuncFParam { ',' FuncFParam } // 1.花括号内重复0次 2.花括号内重复多次
函数形参    FuncFParam → BType Ident ['[' ']' { '[' ConstExp ']' }] // 1.普通变量 2.一维数组变量 3.二维数组变量
语句块     Block → '{' { BlockItem } '}' // 1.花括号内重复0次 2.花括号内重复多次
语句块项    BlockItem → Decl | Stmt // 覆盖两种语句块项
```

```
语句  Stmt → LVal '=' Exp ';' // 每种类型的语句都要覆盖
  | [Exp] ';'  //有无Exp两种情况
  | Block
    | 'if' '(' Cond ')' Stmt [ 'else' Stmt ] // 1.有else 2.无else
    | 'while' '(' Cond ')' Stmt
    | 'break' ';' | 'continue' ';'
    | 'return' [Exp] ';' // 1.有Exp 2.无Exp
    | LVal '=' 'getint''('')'';'
    | 'printf''('FormatString{','Exp}')'';' // 1.有Exp 2.无Exp
表达式 Exp → AddExp 注：SysY 表达式是int 型表达式 // 存在即可
条件表达式   Cond → LOrExp // 存在即可
左值表达式   LVal → Ident {'[' Exp ']'} //1.普通变量 2.一维数组 3.二维数组
基本表达式   PrimaryExp → '(' Exp ')' | LVal | Number // 三种情况均需覆盖
数值  Number → IntConst // 存在即可
一元表达式   UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')' // 3种情况均需覆盖，
函数调用也需要覆盖FuncRParams的不同情况
        | UnaryOp UnaryExp // 存在即可
单目运算符   UnaryOp → '+' | '−' | '!' 注：'!'仅出现在条件表达式中 // 三种均需覆盖
函数实参表   FuncRParams → Exp { ',' Exp } // 1.花括号内重复0次 2.花括号内重复多次 3.
Exp需要覆盖数组传参和部分数组传参
乘除模表达式  MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp // 1.UnaryExp
2.* 3./ 4.% 均需覆盖
加减表达式   AddExp → MulExp | AddExp ('+' | '−') MulExp // 1.MulExp 2.+ 需覆盖 3.-
需覆盖
关系表达式   RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp // 1.AddExp
2.< 3.> 4.<= 5.>= 均需覆盖
相等性表达式  EqExp → RelExp | EqExp ('==' | '!=') RelExp // 1.RelExp 2.== 3.!= 均
需覆盖
逻辑与表达式  LAndExp → EqExp | LAndExp '&&' EqExp // 1.EqExp 2.&& 均需覆盖
逻辑或表达式  LOrExp → LAndExp | LOrExp '||' LAndExp // 1.LAndExp 2.|| 均需覆盖
常量表达式   ConstExp → AddExp 注：使用的Ident 必须是常量 // 存在即可
```

需要注意的点：

- 消除左递归：如在加减表达式 `AddExp → MulExp | AddExp ('+' | '−') MulExp`，通过消除左递归，可以变为 `AddExp → MulExp {('+' | '−') MulExp}`。
- stmt中LVal可能会遇到需要回溯的地方，当需要该类型时，先记录当前位置及输出结果 `preSize = res.size();prePos = curPos`，如果发现不是该类型，则进行回溯，如下图所示：

```
if (res.size()>preSize) {
    int nowSize=res.size();
    res.subList(preSize,nowSize).clear();
}
curPos=prePos;
```

当然，我的语法分析中的语法树构建存在着一定的缺陷，在后续的代码生成中又重新进行了构造。

五、错误处理

错误处理沿用了语法分析所生成的语法树，建立了实体类ErrorEntity来存储错误类型与行号，在进行错误处理时，始终维护一个栈式符号表，错误类型如下图所示：

错误类型	错误类别码	解释
非法符号	a	格式字符串中出现非法字符报错行号为所在行数。
名字重定义	b	函数名或者变量名在**当前作用域**下重复定义。注意，变量一定是同一级作用域下才会判定出错，不同级作用域下，内层会覆盖外层定义。报错行号为所在行数。
未定义的名字	c	使用了未定义的标识符报错行号为所在行数。
函数参数个数不匹配	d	函数调用语句中，参数个数与函数定义中的参数个数不匹配。报错行号为函数调用语句的**函数名**所在行数。
函数参数类型不匹配	e	函数调用语句中，参数类型与函数定义中对应位置的参数类型不匹配。报错行号为函数调用语句的**函数名**所在行数。
无返回值的函数存在不匹配的return语句	f	报错行号为**‘return’**所在行号。
有返回值的函数缺少return语句	g	只需要考虑函数末尾是否存在return语句，**无需考虑数据流**。报错行号为函数**结尾的**’}’**所在行号。
不能改变常量的值	h	为常量时，不能对其修改。报错行号为所在行号。
缺少分号	i	报错行号为分号**前一个非终结符**所在行号。
缺少右小括号’)’	j	报错行号为右小括号**前一个非终结符**所在行号。
缺少右中括号’]’	k	报错行号为右中括号**前一个非终结符**所在行号。
printf中格式字符与表达式个数不匹配	l	报错行号为**‘printf’**所在行号。
在非循环块中使用break和continue语句	m	报错行号为**‘break’**与‘continue’所在行号。

错误处理主要分为以下几部分来写：

- 对于i，j，k类的错误，笔者主要是在语法分析部分进行处理，因为这一部分错误如果不在语法分析进行处理，很有可能导致语法分析程序出bug甚至崩溃。在处理该类型错误时，需要在缺少的地方添加一个新的单词实体类，我的选择是缺少什么补充什么，具体使用方法如下图所示：

```
if (wordEntities.get(curPos).getType().equals("RPARENT")) subtree.addChildNodes(addFinalNodes( x: 1));
else{
    errorEntities.add(new ErrorEntity(wordEntities.get(curPos-1).getLine(), code: "j"));//错误处理
    WordEntity wordEntity=new WordEntity( type: "RPARENT", word: ")",wordEntities.get(curPos-1).getLine());
    subtree.addChildNode(new GrammarTreeEntity( type: "RPARENT", isEndNode: true,wordEntity));
}
```

- 对于a类错误，直接在语法分析遇到STRCON类型字符串时进行判断处理。这里需要注意，如果已经产生a类错误，便需要忽略掉l类型的错误。

- 对于其他类型的错误，可以在Traversal类中自顶向下遍历语法树进行分析。

错误处理结束后，对错误类按行号进行排序并输出，若存在错误，则不运行下述代码。

六、代码生成

代码生成是最最复杂的一步，笔者花费了大量的时间在这一步上。代码生成分为两个部分：中间代码生成和目标代码生成(mips)。

中间代码生成

在生成中间代码时，笔者悲剧地发现，自己语法分析时生成的语法树并没有建立各AST类，导致分析时非常麻烦。一气之下就重新扫描了一边语法（MCodeGenerator类），并生成了各AST节点类。随即对该语法树进行分析并生成了中间代码，并进行符号表的建立与维护。

我的中间代码采用了四元式结构（Quadruple类），定义op为其操作符，dst为结果，arg1和arg2为左端两个运算值，具体如下表所示：

op（操作符）	中间代码
DECL_CONST_IDENT	"const int " + this.dst + " = " + this.arg1
DECL_VAR_IDENT	"var int " + this.dst + " = " + this.arg1
DECL_ARR	"arr int " + this.dst + "[" + this.len + "]"
ASSIGNMENT_CONST_ARRAY	"const " + this.dst + "[" + this.index + "]" + " = " + this.arg1
ASSIGNMENT_VAR_ARRAY	"var " + this.dst + "[" + this.index + "]" + " = " + this.arg1
MAIN	main begin
MAIN_END	main end
BLOCK_BEGIN	block begin
BLOCK_END	block end
ASSIGNMENT	this.dst + " = " + this.arg1
ADD	this.dst + " = " + this.arg1 + " + " + this.arg2
SUB	this.dst + " = " + this.arg1 + " - " + this.arg2
MUL	this.dst + " = " + this.arg1 + " * " + this.arg2
DIV	this.dst + " = " + this.arg1 + " / " + this.arg2
MOD	this.dst + " = " + this.arg1 + " % " + this.arg2
PRINT_STR	"print string " + this.arg1
PRINT_INT	"print int " + this.arg1
GETINT	"read " + this.dst
RETURN	"ret " + this.arg1
PUSH	"push " + this.arg1
FUNC_BEGIN	func begin
FUNC_END	func end
INT_FUNC	"int " + this.arg1 + "()"
VOID_FUNC	"void " + this.arg1 + "()"
PARA	"para int " + this.arg1
CALL	"call " + this.arg1
CALL_BEGIN	"call begin " + this.arg1
GOTO	"goto " + this.arg1

op（操作符）	中间代码
`BEQ`	`"branch " + this.dst + " on " + this.arg1 + " == " + this.arg2`
`BNE`	`"branch " + this.dst + " on " + this.arg1 + " != " + this.arg2`
`EQL`	`"set " + this.dst + " on " + this.arg1 + " == " + this.arg2`
`NEQ`	`"set " + this.dst + " on " + this.arg1 + " != " + this.arg2`
`GRE`	`"set " + this.dst + " on " + this.arg1 + " > " + this.arg2`
`LSS`	`"set " + this.dst + " on " + this.arg1 + " < " + this.arg2`
`GEQ`	`"set " + this.dst + " on " + this.arg1 + " >= " + this.arg2`
`LEQ`	`"set " + this.dst + " on " + this.arg1 + " <= " + this.arg2`
`LABEL`	`"label " + this.arg1 + ":"`
`WHILE_BEGIN`	`while begin`
`WHILE_END`	`while end`

此外，在生成中间代码时，为了减少讨论次数，笔者将二维数组当成一维数组进行计算。

目标代码生成（mips）

在目标代码生成时（MipsGenerator类），不考虑全局寄存器的分配优化，只使用$t0-$t4 为中间结果寄存器，对于每一个产生的变量或中间变量，都将其存入内存和符号表中，对于每一个需要使用的变量或中间变量，都从内存中取出，并更新符号表。

此外，对所有全局变量，使用`: .word`的方式进行存储，对于全局数组元素，若已赋初值，则在`: .word`中将其一一列出（.word存储顺序和数组下标相反！），若没有赋初值，则全部赋值为0（`": .word 0:" + symbol.getDim1()`），同时，将需要输出的字符串以`.asciiz`的方式进行存储（`"str" + index + ": .asciiz \"" + str + "\""`）

下面介绍目标代码生成时几个重要函数：

- `String load(String arg)`：把变量加载到寄存器并返回该寄存器，其中，arg可为中间变量，立即数，@RET，数组变量，普通变量等等。
- `void store(String arg, String valueReg)`：与load相似，输入变量名和包含值的寄存器，将值存入变量在内存中的相应位置。
- `String loadIdentAddress(String arg)`：作用于非数组变量，返回其相对于当前sp指针的偏移量。对于非全局变量，返回sp指针的偏移量，对于全局变量，返回其名称（起始地址）
- `String loadArrayAddress(String arg, String index)`：传入数组名称和下标，返回数组的该下标元素地址。

七、代码优化

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。