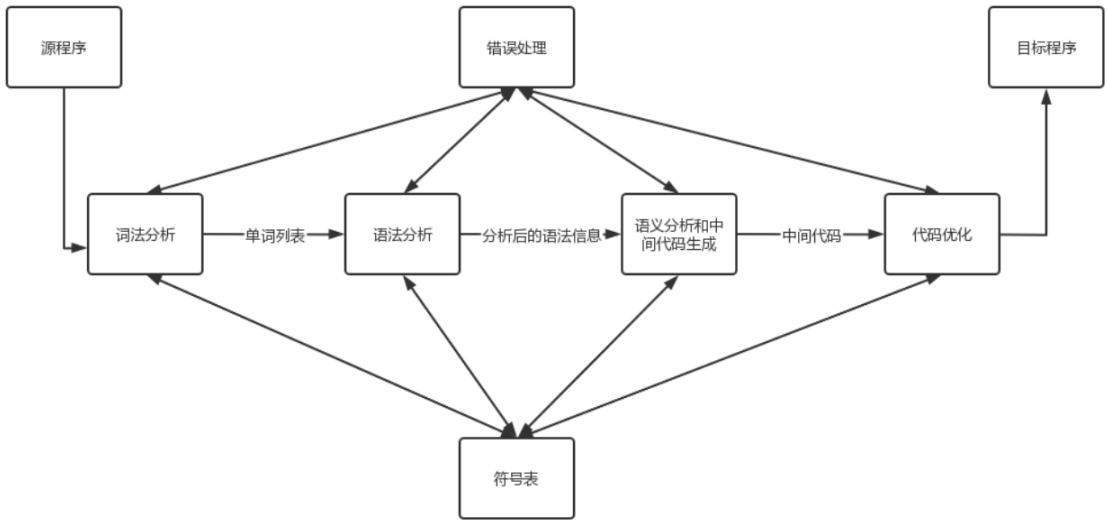
编译器构造指南

19231177 邢云鹏

本文分为两部分。第一部分讲述如何做一个基础版本的MIPS编译器出来，它不含任何优化内容，但保证功能的正确性，以及程序的可维护性，追求架构清晰。第二部分讲述如何在第一部分的基础上进行优化，追求编译器的性能。

基础版本的编译器



上图是一个编译器的主要逻辑部分和主要工作流程。下面逐模块的阐述构造细节。

词法分析

输入：testfile.txt源文件

输出：单词信息元组**(单词原文,类别码,行号)**的列表**vector<tuple<string, string, int> >**

辅助数据结构

①类别对照表

```
map<string, string> WordAnalyser::name2class = {
        {"Ident",           "IDENFR"},  (标识符)
        {"IntConst",        "INTCON"},  (整数字面量)
        {"FormatString",    "STRCON"},  (格式字符串)
        {"main",            "MAINTK"},
        {"const",           "CONSTTK"},
        {"int",             "INTTK"},
        {"break",           "BREAKTK"},
```

```
        {"continue",        "CONTINUETK"},
        {"if",              "IFTK"},
        {"else",            "ELSETK"},
        {"!",               "NOT"},
        {"&&",              "AND"},
        {"||",              "OR"},
        {"while",           "WHILETK"},
        {"getint",          "GETINTTK"},
        {"printf",          "PRINTFTK"},
        {"return",          "RETURNTK"},
        {"+",               "PLUS"},
        {"-",               "MINU"},
        {"void",            "VOIDTK"},
        {"*",               "MULT"},
        {"/",               "DIV"},
        {"%",               "MOD"},
        {"<",               "LSS"},
        {"<=",              "LEQ"},
        {">",               "GRE"},
        {">=",              "GEQ"},
        {"==",              "EQL"},
        {"!=",              "NEQ"},
        {"=",               "ASSIGN"},
        {";",               "SEMICN"},
        {",",               "COMMA"},
        {"(",               "LPARENT"},
        {")",               "RPARENT"},
        {"[",               "LBRACK"},
        {"]",               "RBRACK"},
        {"{",               "LBRACE"},
        {"}",               "RBRACE"},
};
```

②首字符启发索引表

除了格式字符串、注释、标识符之外，一旦看到某个字符串的首字符，马上就能按范围查找。

在前面的优先级高

```
map<char, vector<string> > WordAnalyser::char2words = {
        {'!', vector<string>{"!=", "!"}},
        {'&', vector<string>{"&&"}},
        {'|', vector<string>{"||"}},
        {'+', vector<string>{"+"}},
        {'-', vector<string>{"-"}},
        {'*', vector<string>{"*"}},
        {'/', vector<string>{"/*", "//", "/"}},
        {'%', vector<string>{"%"}},
        {'<', vector<string>{"<=", "<"}},
        {'>', vector<string>{">=", ">"}},
        {'=', vector<string>{"==", "="}},
        {';', vector<string>{";"}},
        {',', vector<string>{","}},
        {'(', vector<string>{"("}},
        {')', vector<string>{")"}},
```

```
        {'[', vector<string>{"["}},
        {']', vector<string>{"]"}},
        {'{', vector<string>{"{"}},
        {'}', vector<string>{"}"}},
};
```

工作流程

①读入所有字符，得到一整个字符串。

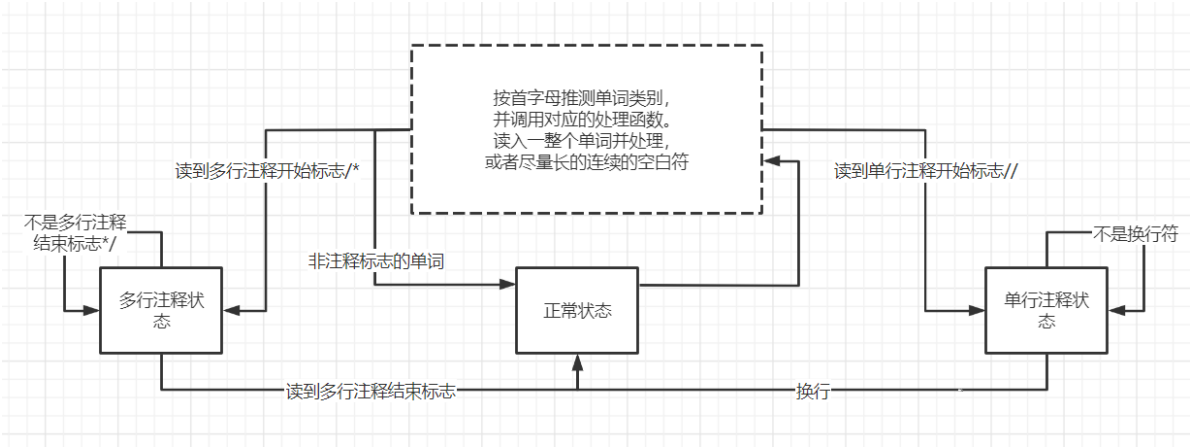
扫描输入文件中的所有字符，如果不是'\r'，就把它存入全局字符流，最后转成全局字符串。

【实现阶段的更改】去除\r的做法是编码完成后的更改。这样做是因为本程序要通过单一的\n来判断换行，去除\r避免判断换行时出错。windows中的换行符是\r+\n，但是在windows下运行的程序会把\r和\n合并处理，本身不会出错。但测评机的环境是Linux，此时如果测评样例是在windows中编辑得到的，那么\r+\n就会被拆成两个字符，而后续词法分析代码没有处理\r的能力，从而导致出错。

②状态+贪心扫描结合遍历字符串

词法分析本身可以写成一个大状态机，但是为了编程的简便和代码的可读性，采用了状态+贪心扫描结合的方式。

设置三个状态：**正常状态**，**单行注释状态**，**多行注释状态**



语法分析

理清语法结构，给错误处理和语义分析保留接口

符号表

采用面向对象的思路设计。包括顶级类SymbolTable（符号表），和各种符号类（以Symbol为公共子类）。

顶级类SymbolTable

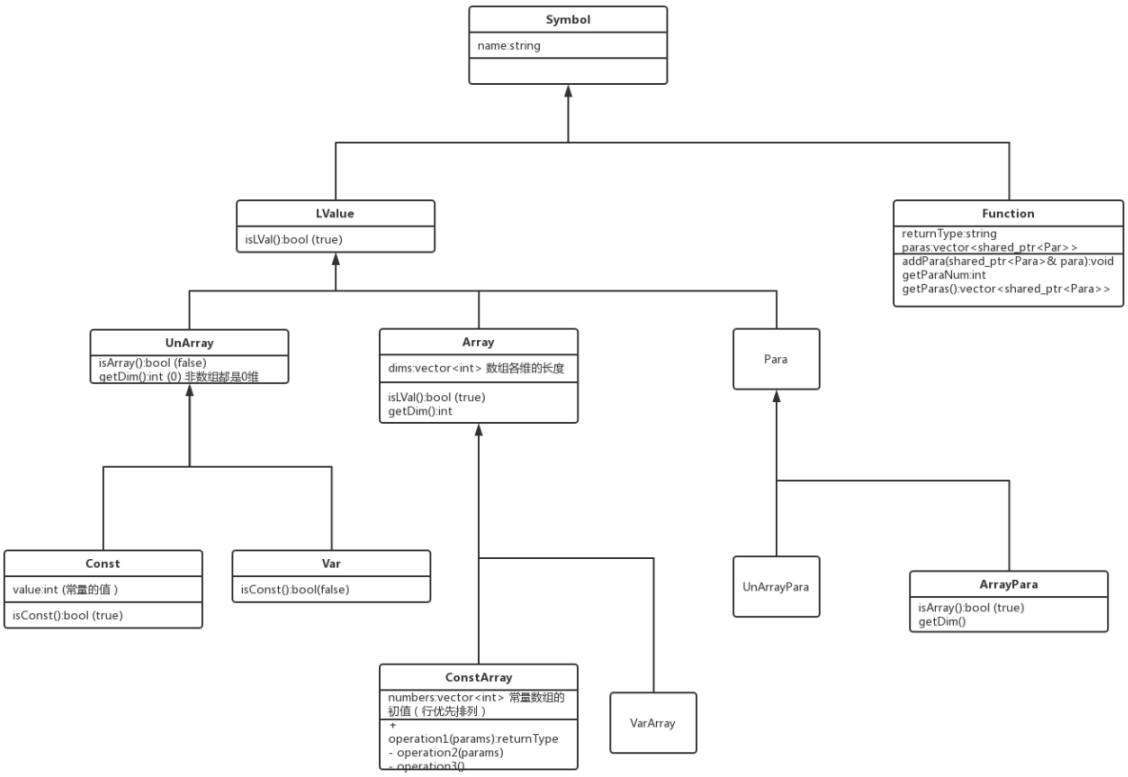
数据结构：

各层级的符号列表map<int , vector<shared_ptr<Symbol> > >

方法和功能

返回值类型	方法名	功能
bool	isRepeatLVal	检查当前作用域是否有重名左值（常量、变量、形参）
bool	isRepeatFunction	检查当前作用域是否有重名函数
bool	addConst	向符号表中添加一个常量（含数组），返回是否重复
bool	addVar	向符号表中添加一个变量（含数组），返回是否重复
bool	addFunction	向符号表中添加一个函数，返回是否重复
bool	addPara	向符号表中添加一个函数参数，返回是否重复
int	addLev	增加符号表的作用域级别，返回修改后的作用域级别
int	deleteLev	回退符号表的作用域级别，返回修改后的作用域级别
shared_ptr<LValue>	getLVal	从符号表中按名称查找一个左值，返回对应指针，查不到返回nullptr
shared_ptr<Function>	getFunc	从符号表中按名称查找一个函数，返回对应指针，查不到返回nullptr

各种符号类



文法

为方便处理，首先去除文法中的左递归，以及不需要输出的语法成分。

修改后的文法为:

```
编译单元 CompUnit → { ConstDecl | VarDecl } {FuncDef} MainFuncDef
常量声明 ConstDecl → 'const' 'int' ConstDef { ',' ConstDef } ';'
常数定义 ConstDef → Ident { '[' ConstExp ']' } '=' ConstInitVal
常量初值 ConstInitVal → ConstExp | '{' [ ConstInitVal { ',' ConstInitVal } ] '}'
变量声明 VarDecl → 'int' VarDef { ',' VarDef } ';'
变量定义 VarDef → Ident { '[' ConstExp ']' } | Ident { '[' ConstExp ']' } '='
InitVal
变量初值 InitVal → Exp | '{' [ InitVal { ',' InitVal } ] '}'
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 → 'int' Ident ['[' ']' { '[' ConstExp ']' }]
语句块 Block → '{' { ConstDecl | VarDecl | Stmt } '}'
语句 Stmt → LVal '=' Exp ';'
 | [Exp] ';'
 | Block
 | 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
 | 'while' '(' Cond ')' Stmt
 | 'break' ';' | 'continue' ';'
 | 'return' [Exp] ';' // 1.有Exp 2.无Exp
 | LVal '=' 'getint''('')'';'
 | 'printf''('FormatString{','Exp}')'';'
表达式 Exp → AddExp
条件表达式 Cond → LOrExp
左值表达式 LVal → Ident {'[' Exp ']'}
```

```
基本表达式 PrimaryExp → '(' Exp ')' | LVal | Number
数值 Number → IntConst
一元表达式 UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp
单目运算符 UnaryOp → '+' | '−' | '!'
函数实参表 FuncRParams → Exp { ',' Exp }
乘除模表达式 MulExp → UnaryExp | MulExp ('*' | '/' | '%') 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 }
常量表达式 ConstExp → AddExp
```

流程

对于每个语法成分，均有一个函数与其对应。采用非回溯的自顶向下分析方法。

为了避免回溯，需要提前读取1~3个单词，并结合First集合判断。

First	symbmols
AddExp	(, LVal,Number,Ident,'+','-'
Exp(非条件表达式)	(, LVal,Number,Ident,'+','-'
ConstExp	(, LVal,Number,Ident,'+','-'
UnaryExp	(, LVal,Number,Ident,'+','-','!'
PrimaryExp	(, LVal,Number
LOrExp	(, LVal,Number,Ident,'+','-','!'
LAndExp	(, LVal,Number,Ident,'+','-','!'

各部分的逻辑举例：

CompUnit

首先读取尽量多的常量或变量定义：首单词为const，或者首单词为int，但2个单词之后不是左括号

其次读取尽量多的函数。为了兼容main函数重复的错误处理情况，如果是普通函数就调用FuncDef，如果是主函数就调用MainFuncDef

符合以下所有条件才是主函数：

首个单词是int

1个单词之后是main

3个单词之后是右括号或者左花括号（为了兼容缺少右括号的错误处理）

ConstDecl

读入const和int后调用ConstDef，然后每遇到一个逗号就调用一次ConstDef。最后处理分号。

ConstDef

首先读入标识符，然后每遇到一个左中括号就调用一次ConstExp并处理右括号。读入等号，并调用ConstInitVal。

ConstInitVal

如果开头的是左花括号，读入它，再读一个ConstInitVal，然后每遇到一个逗号就调用一次ConstInitVal。然后处理右括号。

如果开头的不是左花括号，直接调用ConstExp。

VarDecl

读一个"int",调用VarDef。每遇到一个逗号就调用一次VarDef。

VarDef

读入标识符，根据[的个数调用ConstExp，获取维度信息。如果后面有等号，读入等号，调用InitVal。

错误处理

错误处理部分的实现主要着眼于三个方面：

①能够正确识别并报告错误。

②某一行代码的错误不影响编译过程继续进行。

③便于拓展到新的错误类型。

以下是各种错误的处理方案：

错误类别	类型	检测错误的方案
a	非法符号	对双引号之间的FormatString进行格式检验
b	名字重定义	查符号表并判断是否已经有过这个名字
c	未定义的名字	查符号表并判断是否是未定义的变量
d	函数参数个数不匹配	查符号表得到函数形参个数，与函数实参个数进行比对
e	函数参数类型不匹配	当d不发生时，查符号表得到函数形参的类型，与函数实参的类型进行比对
f	无返回值的函数存在不匹配的return	在符号表中记录函数是否有返回值，当遇到return时，查询符号表并判断是否报错
g	有返回值的函数缺少返回值	编译到某个函数末尾时，如果没有return语句，则查询符号表，判断是否是void函数
h	不能改变常量的值	当给某个左值赋值时，查询符号表判断它是否是常量或常量数组
i	缺少分号	每条语句结束时判断是否有分号
j	缺少右小括号	每次遇到需要有右小括号的文法成分时，判断是否有。
k	缺少右中括号	每次遇到需要有右中括号的文法成分时，判断是否有右中括号。
l	printf中格式字符与表达式个数不匹配	遇到FormatString后，统计其中的格式字符个数。与接下来遇到的表达式个数进行比对。
m	在非循环块中使用break和continue语句	每当进入一层循环语句时，作出记录。遇到break和continue时，查询当前是否在循环块里。

当遇到错误时，进行错误信息的报告，同时跳过当前的错误位置继续编译，防止后面的编译过程受到影响。

为了便于拓展新的错误，把报告错误的功能集成到一个类里：`ErrorManager`

以下面这一行代码为例：

```
errorManager.assert(symbolTable.addFunction(functionName,functionType),getPreLine(),"b");
```

若`symbolTable.addFunction(functionName,functionType)`的返回值是true，则任何事情都不会发生。

若`symbolTable.addFunction(functionName,functionType)`的返回值是false, 则错误管理器里会插入一条类型为b，对应行号为getPreLine()的错误，用于最后统一输出。

代码生成

对于语义分析模块给出的中间代码进行分析，并继续生成MIPS汇编代码。在简单的代码生成部分，不进行数据流分析或基本块划分等任务，函数调用传参也仅仅通过内存而非寄存器来进行。目的是尽可能生成一个不追求性能而保证正确性的编译器。在优化阶段，添加了数据流分析等环节，函数调用也用寄存器传递参数和返回值。

编译时给每个变量赋予唯一一个编号，由于已经经过了语义分析的错误处理过程，所以保证不同函数中定义的变量不同，而且在程序任意位置对任意变量的使用一定不超过变量的作用域。

所有的常量都改成字面量，对应MIPS中的立即数操作。

寄存器管理

在基本的编译器构造过程中，所有的寄存器都认为是临时寄存器。MIPS架构下，任何运算都只能发生在寄存器上，所以需要进行寄存器分配和管理。比如加法运算： @C = @A + @B。需要把变量A和B分别加载到寄存器中，然后利用add指令求和，最后把结果寄存器的值保存在变量C对应的内存地址里。

实现上述功能需要两个辅助函数：

函数**allocateRegister()**分配一个寄存器，并标记寄存器为繁忙。

函数**freeRegister(int register_)** 释放一个寄存器（标记寄存器为空闲）。

因此 @C = @A + @B的代码生成过程可以表示为：

①分配一个空闲寄存器a，并加载变量A到寄存器a中。

②分配一个空闲寄存器b，并加载变量B到寄存器b中。

③分配一个空闲寄存器c，并利用add 指令把a+b赋值给寄存器c。

④释放寄存器a,b。

⑤把寄存器c的值存储到变量C对应的内存地址。

⑥释放寄存器c。

基本的运算语句

流程为：

①从内存中加载需要的数据到寄存器。

②利用寄存器进行运算。

③把寄存器中的结果保存到内存

④释放寄存器，标记为空闲。

静态内存管理

所有的全局变量和main函数中的变量（由于main函数只会执行一次）都放在全局静态数据区，空间向高地址增长。基地址存储在$gp中。每个变量的地址偏移可以在编译阶段获知。对于数组，由于数组的大小可以在编译时确定，所以把数组看做若干个变量，也累积到全局数据区。

编译时若遇到某个访问全局变量（或main函数中的变量）时，首先算出它的地址偏移，与$gp求和，然后用lw访问。

若想要对某个全局变量（或main函数中的变量赋值）,首先算出它的地址偏移，与$gp求和，然后用sw设置它的值。

动态内存管理

由于函数被调用的次数和被调用的顺序难以在编译时完全确定，所以函数中的变量应当存储到动态数据区。这包括：函数参数、函数中的局部变量、函数中的局部变量数组、函数中的局部常量数组。

一个活动记录由三部分构成：DISPLAY，参数区，局部数据区。然而由于本次实验不要求嵌套函数，而且最外层的main函数里面的变量以及全局变量存储在静态数据区，所以不需要DISPLAY。$fp指向当前函数活动记录基地址，每次调用其他函数，就维护这个寄存器的值。

活动记录的布局
局部变量...
函数参数...
函数返回地址
函数返回值

随着编译的进行，会逐步获知当前函数定义的局部变量的信息，所以当前函数的活动记录不断增大。

数组相关操作

与数组相关的操作包括：**数组存取**，**数组地址传递**

如果一个数组是全局数组，那么它里面的各个值依次排列在全局数据区；如果数组是函数中定义的局部数组，则数组中的值依次排列在对应函数的活动记录中；如果数组是参数数组，那么活动记录中记录的是数组的基地址。不特别区分一维数组与二维数组：它们的区别仅仅在于访问数组元素时的偏移计算方法有差异。

当从某个数组中获取值或者向某个数组中存储值时，首先判断这个数组是全局数组还是局部数组，然后获得其相对于静态数据区（或者当前活动记录基地址）的偏移，与$gp（或$fp）求和，获得数组基地址。然后根据访问数组的位置，求出相对于数组基地址的偏移。最后用lw或者sw访存。

数组地址传递：仅出现在函数调用时。

当调用某个函数时，如果这个函数需要一个数组作为参数，则需要把某个全局数组或者局部数组或参数数组的地址传入。若需要传入一个非参数数组的地址，那么可以在编译时确定数组相对于静态数据区或当前活动记录的偏移，从而与$gp或$fp求和，获得数组地址并传入。如果是参数数组，那么从活动记录中获取到的数组值已经是数组地址，直接传入即可。如果需要传入二维数组中的某个一维数组的地址，那么还需要加上偏移（动态计算）

函数

函数调用和传参

①把需要传递的参数从静态数据区或者当前活动记录中加载到寄存器中。

②把寄存器中的值存储到下一个活动记录的参数区。

③释放寄存器，标记为空闲。

④保存当前$fp（活动记录基地址）到下一活动记录的prev abp区。

⑤增加$fp的值，使其指向的位置移动到活动记录的顶部。

⑥利用jal指令进行跳转。（每个函数开头会把$31的值保存到活动记录的返回值区域）

函数返回和传递返回值

①把需要返回的变量从静态数据区或者当前活动记录中加载到寄存器中。

②把寄存器的值存储到活动记录中的函数返回值部分。

③释放寄存器，标记为空闲。

④从prev abp区加载上一活动记录基地址，赋值给$fp。

⑤从活动记录的函数返回地址区域加载函数返回地址到$ra。

⑥jr $ra

编译器的优化

上面阐述了一个基本的编译器构造过程，下面阐述如何进行优化， 以便使编译器的性能有所提高。

基本块的划分

为了便于进行后续的代码优化，首先划分基本块：**跳转语句之后的语句是一个基本块的开头。跳转的目标语句是一个基本块的开头。函数定义语句是基本块的开头，函数调用语句的下一条语句是基本块的开头。**

划分基本块之后，利用跳转信息判断当前基本块的下一个基本块可能是哪些基本块，并存储到基本块的信息里。

根据当前基本块的最后一条语句的类型，设置不同的“下一基本块”

- 函数调用语句：下一基本块包括对应函数的基本块和顺序执行的下一基本块
- 条件跳转语句：下一基本块包括跳转标签所在基本块和顺序执行的下一基本块
- 直接跳转语句：下一基本块为跳转标签所在的基本块
- 函数返回语句：下一基本块包括所有调用这个函数的语句的下一语句所在基本块
- 其他语句：下一基本块为顺序执行的基本块

到达定义分析

①首先求出每个基本块的gen和kill集合。然后利用迭代算法求出每个基本块的in和out集合，从而得到了到达定义的数据流信息。利用到达定义的信息，可以分析出：**任意一条语句中使用的任意一个变量的值是在哪里定义的。**

并不显式的求出kill集合到底包括哪些指令，实际上kill集合中的指令包括且仅包括定义的变量与gen集合中的语句定义的变量相同的语句，所以当迭代过程中需要用到kill集合时，分析gen集合定义了哪些语句，然后删除

第一个阶段：求gen和kill。

算法细节：

- 对于每个基本块，维护一个In集合和Out集合。分析的过程即为In集合流出到Out集合的过程。
- 初始时In集合为所有前驱基本块的Out集合的并集。把这个集合作为一个临时集合进行处理。
- 从前到后遍历一个基本块的各个语句。如果产生了一个定义，则加入到临时集合中，并且删去临时集合中其他所有定义变量与当前语句定义的变量相同的语句。
- 遍历完当前基本块的所有语句后，临时集合赋值给Out集合。
- 按照上面的流程从前到后遍历所有基本块，如果发生了In集合或者Out集合的改变，就继续进行迭代，直到无改变。

第二个阶段：求每条语句的每个变量的定义点。

算法细节：

按照下面的流程遍历每个基本块：

- 维护一个变量定义点集合，这个集合初始是当前基本块的In。根据算法和程序的语法规则，这个集合应当包含了所有当前基本块中先使用后定义的变量的（潜在的）定义点。
- 如果一条语句使用了某个变量，那么查找变量定义点集合中所有关于这个变量的定义点，把它们都作为这条语句的这个变量的潜在的定义点。
- 如果一条语句定义了某个变量，那么把这个定义语句加入到变量定义点集合中，并删除定义相同变量的所有定义。
- 如此进行直到基本块结束。

②根据变量定义点的信息，从前向后进行迭代，判断:**任意一条语句使用的任意一个变量的值是否能在编译时确定。**

算法细节：

①中已经获得了任意一条语句中使用的任意一个变量的值是在哪里定义的这个信息。对所有的变量定义语句进行迭代：

- 如果一条语句是常量定义，那么它的结果是已知的。
- 如果一条语句是从常量数组中的已知位置获取值，那么它的结果是已知的。
- 如果一条语句用到的所有变量都是已知的，那么它的结果是已知的。

  按照上面三条原则进行迭代，如果某次迭代产生了新的已知变量，那么就继续进行迭代，否则迭代停止。

③对于那些能在编译时确定值的变量，统一新增相同值的常量，并且修改为对常量引用。这样，在接下来的活跃变量分析阶段，对于能在编译时确定值的变量的赋值语句会被识别为死代码从而被删除。

活跃变量分析

首先求出每个基本块的def和use。然后利用迭代算法求出每个基本块的in和out集合。这样就得到了活跃变量信息。在活跃变量分析的过程中，默认每条语句是“无意义”的。但是如果这条语句是**①函数调用语句②打印语句③读入语句④访问全局数组的语句**。则认为这条语句是有意义的。然后进行迭代：如果一条语句改变了某个变量的值，而这个变量在这里的值也会被另一条“有意义”的语句使用，则当前语句也被判断为“有意义”的。

活跃变量分析与到达定义分析相结合，实际上进行了**死代码删除**。

算法细节：

对各个基本块从后往前进行遍历。遍历每个基本块时，维护活跃变量集合。

活跃变量集合初始为这个基本块的Out，即所有前驱基本块的In。

- 如果打印了某个变量，则把它加入到活跃变量集合中。
- 如果某条语句会定义某个活跃变量，那么把它从活跃变量结合中删除，同时把这条语句用到的所有变量添加到活跃变量中。

按照上述过程进行迭代，如果某次迭代过程中发现了新的活跃变量，那么就继续进行迭代，直到没有新的活跃变量被发现为止。

图着色全局寄存器分配

利用活跃变量分析的结果，容易获得变量冲突图：如果一个变量定义时，另一个变量是活跃的，那么它们冲突。根据冲突图，利用启发式算法进行全局寄存器分配。

算法细节：

- 首先构造一个单独的类作为冲突图，它负责冲突图的构建和寄存器分配结果的导出。
- 在活跃变量分析的过程中，如果把某个新的活跃变量添加到某个活跃变量集中，那么它与活跃变量集中的所有变量都冲突。这部分可以通过在冲突图类中保存活跃变量集来进行优化。
- In集合中的所有变量互相冲突
- Out集合中的所有变量互相冲突。

如果给一个变量分配了全局寄存器，那么正常情况下无需关注它对应的内存地址里存储的值。然而当发生函数调用时，需要把全局寄存器里的值（如果它是函数的局部变量）保存到内存，当函数调用返回时再从内存中取出。

换页算法局部寄存器分配

局部寄存器分配的关键在于：当局部寄存器不够用时，把哪个寄存器中存储的变量换出寄存器，存到内存里。为了比较简便的求出较优的解，编译进行两次，第一次不进行目标代码生成，只进行模拟编译，获得每个基本块中的变量使用顺序，第二次编译时，可以利用第一次编译的结果，当寄存器换出时，把最久的将来才会用到的变量进行换出。

算法细节：

- 在模拟编译模式下，如果把某个变量的值加载到寄存器中，就对当前基本块做一个记录。
- 在正常编译模式下，如果把某个变量的值加载到寄存器中，就“消费”一条记录。

```
/**
 * 必须把变量弄到可以直接使用的寄存器里，不能是hi 或者 lo
 */
int loadValueToRegister(int valueIndex) {
    if (this->type == "common") {
        nowBlock->useOneVar(valueIndex);
    } else if (this->type == "getVarsBelongsTo") {
        nowBlock->addUseVar(valueIndex, varsBelongsTo[valueIndex]);
        return 1;
    }
```

根据编译器的构造流程，把变量加载到寄存器中意味着要用到这个变量。所以按照上面的办法，能够在编译的任何时刻知道，最久的将来才会被用到的变量是哪个。

事实上，调用loadValueToRegister函数时，如果发现某个变量已经在寄存器中了，那么直接返回那个寄存器的编号，而不会分配新的寄存器，也不会生成lw指令。

维护一个Dirty数组，记录哪些寄存器里的变量与内存中的版本不一致，离开当前基本块时，如果一个寄存器里的变量是"脏"的，那么就把它保存到内存。

除法降级

复现论文**Divison by Invariant Integers using Multiplication**的结果：

Inputs: **sword** d and n, with d constant and $d \neq 0$.
udword m;
int ℓ, sh_{post};
$(m, sh_{\text{post}}, \ell) = \text{CHOOSE\_MULTIPLIER}(|d|, N-1)$;
if $|d| = 1$ **then**
  Issue $q = d$;
else if $|d| = 2^{\ell}$ **then**
  Issue $q = \text{SRA}(n + \text{SRL}(\text{SRA}(n, \ell-1), N-\ell), \ell)$;
else if $m < 2^{N-1}$ **then**
  Issue $q = \text{SRA}(\text{MULSH}(m, n), sh_{\text{post}}) - \text{XSIGN}(n)$;
else
  Issue $q = \text{SRA}(n + \text{MULSH}(m - 2^N, n), sh_{\text{post}}) - \text{XSIGN}(n)$;
  Cmt. Caution — $m - 2^N$ is negative.
end if

if $d < 0$ **then**
  Issue $q = -q$;
end if

```
procedure CHOOSE_MULTIPLIER(uword d, int prec);
Cmt. d – Constant divisor to invert. 1 ≤ d < 2^N.
Cmt. prec – Number of bits of precision needed, 1 ≤ prec ≤ N.
Cmt. Finds m, sh_post, ℓ such that:
Cmt.      2^(ℓ-1) < d ≤ 2^ℓ.
Cmt.      0 ≤ sh_post ≤ ℓ. If sh_post > 0, then N + sh_post ≤ ℓ + prec.
Cmt.      2^(N+sh_post) < m * d ≤ 2^(N+sh_post) * (1 + 2^-prec).
Cmt. Corollary. If d ≤ 2^prec, then m < 2^(N+sh_post) * (1 + 2^(1-ℓ))/d ≤ 2^(N+sh_post-ℓ+1).
Cmt.            Hence m fits in max(prec, N - ℓ) + 1 bits (unsigned).
Cmt.
int ℓ = ⌈log2 d⌉,   sh_post = ℓ;
udword m_low = ⌊2^(N+ℓ)/d⌋,   m_high = ⌊(2^(N+ℓ) + 2^(N+ℓ-prec))/d⌋;
Cmt. To avoid numerator overflow, compute m_low as 2^N + (m_low - 2^N).
Cmt. Likewise for m_high. Compare m' in Figure 4.1.
Invariant. m_low = ⌊2^(N+sh_post)/d⌋ < m_high = ⌊2^(N+sh_post) * (1 + 2^-prec)/d⌋.
while ⌊m_low/2⌋ < ⌊m_high/2⌋ and sh_post > 0 do
    m_low = ⌊m_low/2⌋;   m_high = ⌊m_high/2⌋;   sh_post = sh_post - 1;
end while;                    /* Reduce to lowest terms. */
return (m_high, sh_post, ℓ);    /* Three outputs.          */
end CHOOSE_MULTIPLIER;
```

公共子表达式删除

对于每个基本块，利用DAG图求出其公共子表达式，然后利用启发式算法求出新的中间代码。