Lab 0 实验报告

PB20000296 郑滕飞

Docker 镜像最终效果：

```
PS D:\> docker start -ai compiler-labs
root@2684c043e391:~# cd /labs
root@2684c043e391:/labs# clang -S -emit-llvm fibonacci.c
root@2684c043e391:/labs# lli fibonacci.ll; echo $?
55
root@2684c043e391:/labs#
```

对 Docker 理解：

虚拟化（英语：Virtualization）是一种资源管理技术，是将计算机的各种实体资源，如服务器、网络、内存及存储等，予以抽象、转换后呈现出来，打破实体结构间的不可切割的障碍，使用户可以比原本的组态更好的方式来应用这些资源。这些资源的新虚拟部份是不受现有资源的架设方式，地域或物理组态所限制。一般所指的虚拟化资源包括计算能力和资料存储。

Docker 对进程进行封装隔离，属于操作系统层面的虚拟化技术。由于隔离的进程独立于宿主和其它的隔离的进程，因此也称其为容器。比起直接使用虚拟机进行硬件层面的虚拟化，Docker 的启动速度更快、占用体积小。由于确保了执行环境的一致，应用的迁移、维护、拓展也更加简单。

(参考 https://blog.csdn.net/qq_34936541/article/details/104890251)

个人总结：可以当作一个更小单位，拥有部分组件的轻量化的虚拟机，虽然运行时环境并不能提供内核[如上学期操作系统涉及内核模块的实验就无法用 Docker 解决]，但这也带来了更加高效、快速的体验。

总结：

在 Windows 配到一半，想了想感觉是不是去 Linux 虚拟机搞更合适，然后整了两个小时也没研究出来 docker.service 启动失败：Unit not found 到底怎么解决，但解决不了这个也搞不定 docker daemon 的启动，遂放弃，乖乖用 Windows 了。

Lab 1 实验报告

PB20000296 郑滕飞

词法分析：

本部分要求对.y 文件与.l 文件进行适当填空以完成词法分析器。

1、tokens 与 union

根据之后代码的要求，每一个 token 的类型都需要是语法树结点类型，而通过.h 文件即可看出它对应的类型(最开始写的时候漏掉了 struct 导致报错，后来才意识到规则)：

```
%union {
    struct syntax_tree_node *node;
}
```

而根据基础知识里的要求即可写出所有对应的 tokens：

```
%token <node> ERROR
%token <node> ELSE IF INT RETURN VOID WHILE FLOAT ADD MINUS TIMES DIV LESS LEQ GREATER GEQ EQ
NEQ ASSIGN SEMI COMMA LSMALL RSMALL LMID RMID LBIG RBIG ID INTEGER FLOATPOINT
```

值得注意的是，对注释与空白只需要忽略即可，因此也不需要作为 tokens 返回。

2、正则表达式

通过查阅文件 flex regular expression 知具体符号可通过转义符表达，连续的字符串则通过双引号。大部分的正则表达式要求都很简单，直接仿照已经给出的 ADD 处理即可：

```
"while" {pos_start = pos_end; pos_end += 5; pass_node(yytext); return WHILE;}
"float" {pos_start = pos_end; pos_end += 5; pass_node(yytext); return FLOAT;}
\+ {pos_start = pos_end; pos_end++; pass_node(yytext); return ADD;}
\- {pos_start = pos_end; pos_end++; pass_node(yytext); return MINUS;}
\* {pos_start = pos_end; pos_end++; pass_node(yytext); return TIMES;}
\/ {pos_start = pos_end; pos_end++; pass_node(yytext); return DIV;}
\< {pos_start = pos_end; pos_end++; pass_node(yytext); return LESS;}
"<=" {pos_start = pos_end; pos_end += 2; pass_node(yytext); return LEQ;}
\> {pos_start = pos_end; pos_end++; pass_node(yytext); return GREATER;}
">=" {pos_start = pos_end; pos_end += 2; pass_node(yytext); return GEQ;}
```

对于空白的忽略，由于需要维护行数与列数，将\n、\t 与其他空白分别处理：

```
[\r ] {pos_start = pos_end; pos_end++;}
\t {pos_start = pos_end; pos_end += 4;}
\n {pos_end = 0; lines++;}
```

对 ID、INTEGER 与 FLOATPOINT，也直接根据基础知识文档书写即可：

```
[a-zA-Z]+ {pos_start = pos_end; pos_end += strlen(yytext); pass_node(yytext); return ID;}
[0-9]+ {pos_start = pos_end; pos_end += strlen(yytext); pass_node(yytext); return INTEGER;}
([0-9]+\.)|([0-9]*\.[0-9]+) {pos_start = pos_end; pos_end += strlen(yytext); pass_node(yytext);
return FLOATPOINT;}
```

由于 flex 自动匹配最长的特性，如此书写即可保证正则表达式被正确识别。

比起以上这些，匹配注释就难得多了。我最初写成了”/*”[0x00-0xFF]*”*/”，但简单测试即发现，由于匹配最长的特性，当出现两个注释时，这种写法会导致第一个注释的开头和第二个注释的结尾匹配，从而忽略了中间的内容。

因此，不仅需要保证注释开头结尾匹配，由于注释不能嵌套，还需要中间不能出现连续的/*或*/。此外，当识别完注释之后，需要通过对里面的\n 进行计数来确定 lines 增加的次数，由此，最后的结果是这样的：

```
"/*"(\**|\/*)([^\*\/]+(\**|\/*))*"*/" {int i = 0; while(yytext[i]) {if (yytext[i++]
{pos_end = 0; lines++;} else pos_end++;}}
```

在注释的开始结束之间，任意连续个/或*中间必须有其他字符，即可保证两者不会连续出现。

3、结果展示

```
root@0eac26b38678:/labs/2022fall-compiler_cminus# ./build/lexer ./tests/parser/normal/local-decl.cminus
Token         Text      Line    Column (Start,End)
261            int      1       (0,3)
285           main      1       (4,8)
279              (      1       (8,9)
263           void      1       (9,13)
280              )      1       (13,14)
283              {      1       (15,16)
261            int      2       (4,7)
285              i      2       (8,9)
277              ;      2       (9,10)
265          float      2       (11,16)
285              j      2       (17,18)
277              ;      2       (18,19)
263           void      3       (4,8)
285              v      3       (9,10)
277              ;      3       (10,11)
262         return      4       (4,10)
286              0      4       (11,12)
277              ;      4       (12,13)
284              }      5       (0,1)
```

上图为对 normal/local-decl.cminus 进行词法分析的结果，可以验证维护正确(由于将\t 视为了四个字符，首行的缩进相当于四个空格的效果)。

语法分析：

本部分要求对.y 文件进行适当填空以完成词法分析器。

1、实现过程

大部分的实现过程直接参考给出的例子即可：

```
return-stmt : RETURN SEMI {$$ = node("return-stmt", 2, $1, $2);}
| RETURN expression SEMI {$$ = node("return-stmt", 3, $1, $2, $3);}
```

根据 node 函数，只需要标出结点的名字与孩子个数，并进行列举即可。

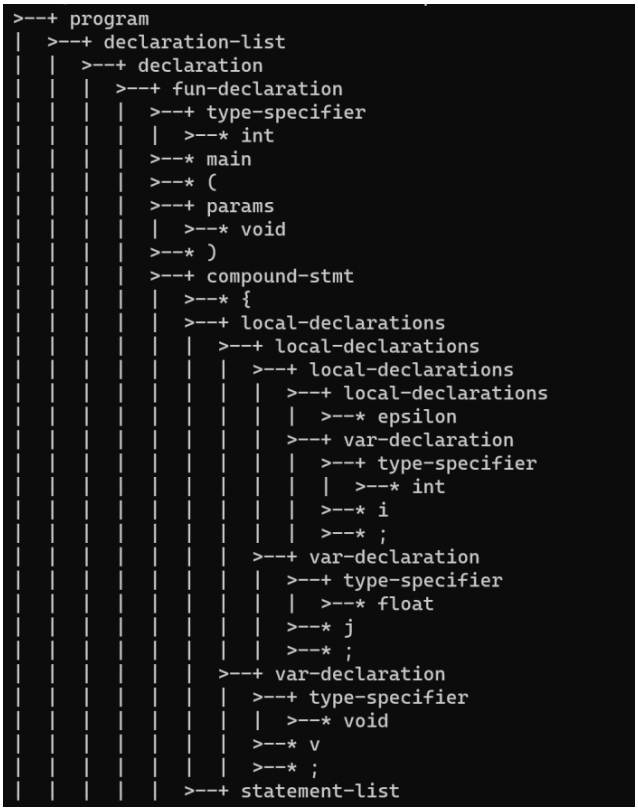
唯一需要研究的是对 empty 的处理，根据基础知识文档，empty 直接在|后为空即可，而根据 node 中的提示：

```
if (children_num == 0) {
    child = new_syntax_tree_node("epsilon");
    syntax_tree_add_child(p, child);
```

为空时设置 children_num 为 0 即可，从而类似可知：

```
args : arg-list {$$ = node("args", 1, $1);}
| {$$ = node("args", 0);}
```

2、结果展示



上图为对 normal/local-decl.cminus 进行词法分析的结果，由于换行区别，直接进行.sh 脚本分析会提示存在不同，不过对比 std 可发现实际上是相同的，因此可知正确。

思考题：

1.

由于 Bison 是将 LALR 文法转换成可编译的 C 代码，而 LALR 文法是可以处理左递归的。

2.

维护了一棵树，树的结点类型是.y 中定义的 union，叶子结点为 token 中的，而其他为 type 中的，在归约过程中自然得到了每个结点的值。

3.

浮点型除以 0 会成为无穷大，整型则会报错。

4.

将 `case '/': $$ = $1 / $3;`这句中，添加检测`$3` 是否为 0 的 `if`，若为 0 则将结点值设为特殊值。每层都检测特殊值，若孩子为特殊值则将自己也设为特殊值，根节点若检测到特殊值则报发生了除以 0 的错误(或在检测到除以 0 时直接退出报错)。

总结：

全场最难任务：识别注释的正则表达式，从研究规则开始弄了一个多小时。

语法分析纯纯体力活，强烈谴责(不是)助教在写 `readme` 的时候没把那 31 条规则弄成方便直接复制的形式-__-

不过 `type` 可以起成自己定的缩写，还是比较友好的。

Lab 2 实验报告

PB20000296 郑滕飞

ll 文件：

本部分要求根据 c 文件写出对应的 ll 文件。

通过学习 gcd 的例子生成的 ll 文件，可以找到大致思路：

1、assign

这个文件主要涉及数组的位置和存取，建立数组时需要分配空间(关于分配空间后%1 的类型见后方的问题回答)：

```
%1 = alloca [10 x i32]
```

并在对应位置存/取，如对 a[0]操作，需要先用%2 找到 a[0]的地址，再存入数据：

```
%2 = getelementptr inbounds [10 x i32], [10 x i32]* %1, i64 0, i64 0
store i32 10, i32* %2
```

运算指令则直接仿照 gcd 中的 ll 进行计算即可，如：

```
%4 = mul nsw i32 %3, 2
```

2、fun

函数的定义部分与 main 相同：

```
define dso_local i32 @callee(i32 %0) #0 {...}
```

而调用亦仿照 gcd 进行即可：

```
%1 = call i32 @callee(i32 110)
ret i32 %1
```

3、if

这里由于涉及浮点数操作，先利用 clang 生成 ll 文件进行对照。浮点数的存储如下：

```
%1 = alloca float
store float 0x40163851E0000000, float* %1
```

(奇怪的事：store float 5.5 ...是可以的，5.55 就不行了)

而涉及的比较操作为：

```
%3 = fcmp ogt float %2, 1.0
```

if 产生了两个分支，由于此处没有 else，第一个分支结束要无条件跳转第二个分支，即：

```
br i1 %3, label %4, label %6
4:
    ret i32 233
    br label %6
6:
    ret i32 0
```

4、while

while 事实上会涉及四个分支，while 前、比较、执行、while 后。

while 前无条件跳转到比较分支，比较成功则执行，否则跳到 while 后。每次执行结束，无条件跳转回比较。由此，代码如下：

```
...
br label %3

    3:
        %4 = load i32, i32* %2
        %5 = icmp slt i32 %4, 10
        br i1 %5, label %6, label %10

    6:
        %7 = add nsw i32 %4, 1
        %8 = load i32, i32* %1
        %9 = add nsw i32 %7, %8
        store i32 %7, i32* %2
        store i32 %9, i32* %1
        br label %3

10:
    %11 = load i32, i32* %1
    ret i32 %11
```

cpp 文件：

本部分要求根据 c 文件写出生成对应 ll 的 cpp 文件。

仍然通过学习 gcd 的例子的 cpp 文件寻找大致思路：

1、assign

数组的构建与存取如下：

```cpp
auto *arrayType = ArrayType::get(Int32Type, 10);
auto aAlloca = builder->create_alloca(arrayType);
auto a0GEP = builder->create_gep(aAlloca, {CONST_INT(0), CONST_INT(0)});
builder->create_store(CONST_INT(10), a0GEP);
auto aLoad = builder->create_load(a0GEP);
```

类似之前写 ll 过程，先生成了数组指针，再生成对应位置的指针，最后存取具体的值。

2、fun

对每个函数，都需要定义对应的基本块：

```cpp
auto calleeFunTy = FunctionType::get(Int32Type, Ints);
auto calleeFun = Function::create(calleeFunTy, "callee", module);
auto bb = BasicBlock::create(module, "entry", calleeFun);
builder->set_insert_point(bb);
```

```
auto mainFun = Function::create(FunctionType::get(Int32Type, {}), "main", module);
bb = BasicBlock::create(module, "entry", mainFun);
builder->set_insert_point(bb);
```

获取参数的过程仿照 gcd 中：

```
std::vector<Value *> args;  // 获取callee函数的形参,通过Function中的iterator
for (auto arg = calleeFun->arg_begin(); arg != calleeFun->arg_end(); arg++) {
    args.push_back(*arg);   // * 号运算符是从迭代器中取出迭代器当前指向的元素
}
builder->create_store(args[0], aAlloca);
```

而调用直接生成即可：

```
auto call = builder->create_call(calleeFun, {CONST_INT(110)});
builder->create_ret(call);
```

3、if

浮点数需要先定义指针类型，即利用 get_float_type。涉及分支时逻辑已在 ll 部分解释过，实现如下：

```
auto fcmp = builder->create_fcmp_gt(aLoad, CONST_FP(1));
auto trueBB = BasicBlock::create(module, "trueBB", mainFun);    // true分支
auto falseBB = BasicBlock::create(module, "falseBB", mainFun);
auto br = builder->create_cond_br(fcmp, trueBB, falseBB);

builder->set_insert_point(trueBB);
builder->create_ret(CONST_INT(233));
builder->create_br(falseBB);

builder->set_insert_point(falseBB);
builder->create_ret(CONST_INT(0));
```

4、while

逻辑同 ll 中所述，分支实现如下：

```
auto judgeBB = BasicBlock::create(module, "judgeBB", mainFun);
auto whileBB = BasicBlock::create(module, "whileBB", mainFun);
auto retBB = BasicBlock::create(module, "retBB", mainFun);
builder->create_br(judgeBB);

builder->set_insert_point(judgeBB);
auto iLoad = builder->create_load(iAlloca);
auto icmp = builder->create_icmp_lt(iLoad, CONST_INT(10));
builder->create_cond_br(icmp, whileBB, retBB);

builder->set_insert_point(whileBB);
iLoad = builder->create_load(iAlloca);
auto aLoad = builder->create_load(aAlloca);
auto iAdd = builder->create_iadd(iLoad, CONST_INT(1));
auto aAdd = builder->create_iadd(aLoad, iAdd);
builder->create_store(iAdd, iAlloca);
builder->create_store(aAdd, aAlloca);
builder->create_br(judgeBB);

builder->set_insert_point(retBB);
aLoad = builder->create_load(aAlloca);
builder->create_ret(aLoad);
```

问题解答：

1、getelementptr

当定义了`%1 = alloca [10 x i32]`后，`%1` 的类型事实上是一个指向 `10` 元数组的指针，即 `int (*p)[10]`。

于是，为找到其中的元素的地址，必须使用

```
%2 = getelementptr [10 x i32], [10 x i32]* %1, i32 0, i32 %0
```

第一个 `0` 代表 p 所指的第一个十元数组，第二个 `0` 代表十元数组中的第一个元素。

而传递形参等情况时，传递的是 `int*`类型的指针，即 `p[0]`，这时只需要找到其中对应的元素即可，于是只需要:

```
%2 = getelementptr i32, i32* %1 i32 %0
```

(也即两种写法针对的指针类型不同)

2、cpp 与 .ll 的对应

assign 中，只有 main 函数的一个基本块，直接对应:

```
define i32 @main() {
label_entry:
  %op0 = alloca [10 x i32]
  %op1 = getelementptr [10 x i32], [10 x i32]* %op0, i32 0, i32 0
  store i32 10, i32* %op1
  %op2 = load i32, i32* %op1
  %op3 = mul i32 2, %op2
  %op4 = getelementptr [10 x i32], [10 x i32]* %op0, i32 0, i32 1
  store i32 %op3, i32* %op4
  %op5 = load i32, i32* %op4
  ret i32 %op5
}
```

fun 中，callee 与 main 各有一个基本块，cpp 中的定义见上文 `2.2`:

```
define i32 @callee(i32 %arg0) {
label_entry:
  %op1 = alloca i32
  store i32 %arg0, i32* %op1
  %op2 = load i32, i32* %op1
  %op3 = mul i32 2, %op2
  ret i32 %op3
}
define i32 @main() {
label_entry:
  %op0 = call i32 @callee(i32 110)
  ret i32 %op0
}
```

if 中，除 entry 外有 true 分支和 false 分支两个基本块，cpp 中的定义见上文 `2.3`:

```
define i32 @main() {
label_entry:
  %op0 = alloca float
  store float 0x40163851e0000000, float* %op0
  %op1 = load float, float* %op0
  %op2 = fcmp ugt float %op1,0x3ff0000000000000
  br i1 %op2, label %label_trueBB, label %label_falseBB
label_trueBB:                                     ; preds = %label_entry
  ret i32 233
  br label %label_falseBB
label_falseBB:                                    ; preds = %label_entry, %label_trueBB
  ret i32 0
}
```

while 中，需要前、判断、循环体、后四个基本块，cpp 中的定义见上文 2.4：

```
define i32 @main() {
label_entry:
  %op0 = alloca i32
  %op1 = alloca i32
  store i32 10, i32* %op0
  store i32 0, i32* %op1
  br label %label_judgeBB
label_judgeBB:                                    ; preds = %label_entry, %label_whileBB
  %op2 = load i32, i32* %op1
  %op3 = icmp slt i32 %op2, 10
  br i1 %op3, label %label_whileBB, label %label_retBB
label_whileBB:                                    ; preds = %label_judgeBB
  %op4 = load i32, i32* %op1
  %op5 = load i32, i32* %op0
  %op6 = add i32 %op4, 1
  %op7 = add i32 %op5, %op6
  store i32 %op6, i32* %op1
  store i32 %op7, i32* %op0
  br label %label_judgeBB
label_retBB:                                      ; preds = %label_judgeBB
  %op8 = load i32, i32* %op0
  ret i32 %op8
}
```

3、Visitor Pattern

语法树

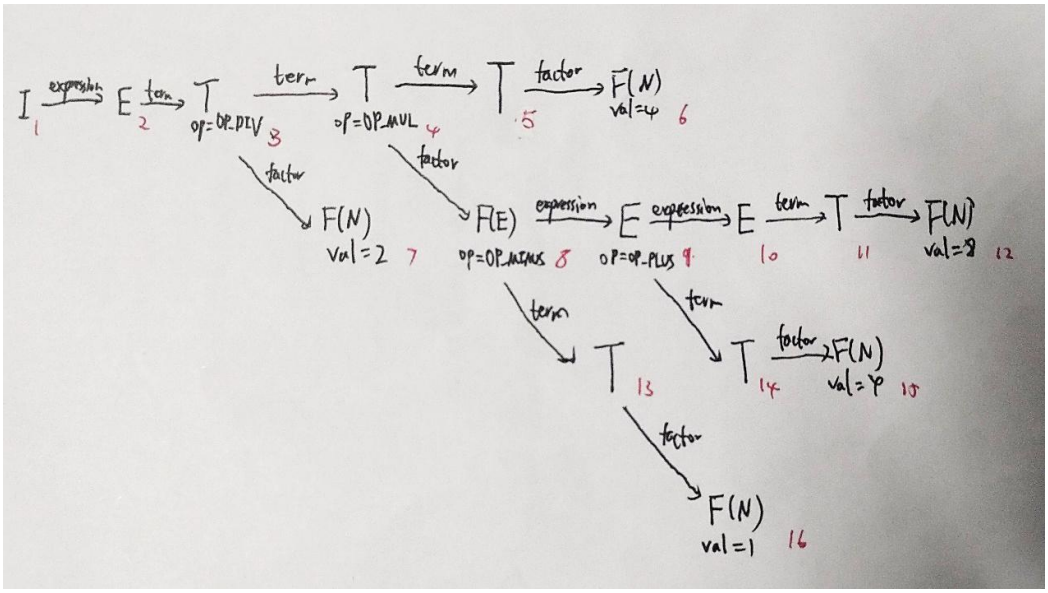
I: CalcASTInput*

E: CalcASTExpression*

T: CalcASTTerm*

F(N): CalcASTNum*强制转换为的 CalcASTFactor*

F(E): CalcASTExpression*强制转换为的 CalcASTFactor*



访问顺序：

1->2->3->4->5->6->8->9->10->11->12->14->15->13->16->7

总结：

代码部分实现并不算复杂，主要就是仿照，大部分的 bug 都是编译报错，能直接通过报错信息解决。

Vistor Pattern 手搓语法树的时候参考了一个资料：

父类子类指针相互转换

转载 matrix2020 于 2021-10-28 09:45:17 发布 1041 收藏 8 版权

分类专栏： C++ 文章标签： c++ mfc cocos2d

C++ 专栏收录该内容 0 订阅 3 篇文章 订阅专栏

父类子类指针相互转换

1.当自己的类指针指向自己类的对象时，无论调用的是 虚函数 还是实函数，其调用的都是自己的；

2.当指向父 类对象 的父类指针被强制转换成子类指针时候，子类指针调用函数时，只有非重写函数是自己的，虚函数是父类的；

3.当指向子类对象的子类指针被强制转换成父类指针的时候，也就是父类指针指向子类对象，此时，父类指针调用的虚函数都是子类的，而非虚函数都是自己的；

当父类子类有同名非虚函数的时候，调用的是**转换后**的指针类型的函数；
当父类子类有同名虚函数的时候，调用的是指针**转换前**指向的对象类型的函数。

代码写得真妙，通过派生直接减了一层代码树（感叹

Lab 3 实验报告

PB20000296 郑滕飞

总结：

这次破例把总结写在最开始，既是解释一下接下来报告的组织，也是表达一下感叹。

[宣布 Lab 2 + Lab 3 是大学到现在见过最好的实验设计]

Lab 2 的三个阶段真的起到了非常不错的引导作用[写代码的时候经常回去看上个实验写的 ll 和 cpp]，包括最后的三个问题，第一个提示了个人认为全场最难处理的地方[也是我最后一个解决的 bug，对应 lv2 的测试中的 func array array]，第二个问题则是提示基本块的对应关系[这里有个正常人几乎考虑不到但是很容易犯的错，对应 lv1 的测试中的 selection 系列]，第三个问题如果认真做的话，事实上是在学习读代码的方式[ast.cpp, builder.cpp, builder.hpp 联动，加上寻找其他文件一些类定义中有效的部分]。

而 Lab 3 的难度合适，并没有过于复杂，也能在一点一点看到正确率提升的过程中感到理解增加。testcase 的针对性很强，也充足，因此定位问题无需无头苍蝇一样寻找。而每个问题的解决，其实都是编译过程考虑到的一个边界情况。

所以，这次报告并不是按照最终结果顺序进行组织，而是基本靠解决 bug 的时间顺序去解释结构[第一次编译过直接提交然后通过测试 15/80 的惨痛画面实在记忆犹新]。

整体结构：

首先，需要理解的是整个框架的结构。

框架已经从.y 文件生成的符合 cminus 文法结构的语法树，并将其转化成了需要的树结构。写 visit 函数，事实上是需要在遍历中进行计算。

于是，通过建立全局变量 Value* result，储存当前 builder 构建命令后返回的 Value。之后的大部分计算都是依托此。

例如，访问 Num 结点时，直接通过整数或浮点计算结果即可：

```
void CminusfBuilder::visit(ASTNum &node) {
    //!TODO: This function is empty now.
    // Add some code here.
    if (node.type == TYPE_INT)
        result = CONST_INT(node.i_val);
    else result = CONST_FP(node.f_val);
}
```

[这里之前犯了个小错，type 是与 TYPE_INT/TYPE_FLOAT 比较的，而 Value 类型的方法.get_type()得到的才是直接与 INT32_T 等比较的]

对变量声明语句，通过 node.num 确定是数还是数组，并直接对应插入声明语句，将声明变量的名称与**地址**放入 scope。

```
if (node.num == nullptr) {
    if (node.type == TYPE_INT) {
        auto val = builder->create_alloca(INT32_T);
        scope.push(node.id, val);
    }
    else {
        auto val = builder->create_alloca(FLOAT_T);
        scope.push(node.id, val);
    }
}
```

```
else {
    if (node.type == TYPE_INT) {
        auto *arrayType = ArrayType::get(INT32_T, node.num->i_val);
        auto val = builder->create_alloca(arrayType);
        scope.push(node.id, val);
    }
    else {
        auto *arrayType = ArrayType::get(FLOAT_T, node.num->i_val);
        auto val = builder->create_alloca(arrayType);
        scope.push(node.id, val);
    }
}
```

在函数调用语句中，需要对参数类型进行处理：

```
if (param->type == TYPE_INT) {
    if (param->isarray) param_types.push_back(INT32PTR_T);
    else param_types.push_back(INT32_T);
}
else {
    if (param->isarray) param_types.push_back(FLOATPTR_T);
    else param_types.push_back(FLOAT_T);
}
```

并且为形参建立对应的变量：

```
auto val = builder->create_alloca(param_types[i]);
builder->create_store(args[i], val);
scope.push(node.params[i]->id, val);
```

[由于并不想重复判断类型操作，并不会对 param 调用 visit，对应的 visit 函数直接为空。此外，这里一开始忘记了 create store 一句，回去看 Lab 2 的 gcd 的对应 cpp 才发现。]

对于 Compound Statement，起初也没有意识到需要加的代码是什么，直到测试中的 scope 错误才终于发现，大括号语句需要有一个自己的作用域，也即：

```
void CminusfBuilder::visit(ASTCompoundStmt &node) {
    //!TODO: This function is not complete.
    // You may need to add some code here
    // to deal with complex statements.

    scope.enter();

    for (auto &decl : node.local_declarations) {
        decl->accept(*this);
    }

    for (auto &stmt : node.statement_list) {
        stmt->accept(*this);
        if (builder->get_insert_block()->get_terminator() != nullptr)
            break;
    }

    scope.exit();
}
```

对于需要新建基本块的语句，也存在一个隐蔽的问题。如果直接固定基本块的命名，会导致基本块命名相同，从而 selection 测试无法通过，因此，需要全局变量管理基本块的序号：

```
int c = 0;
```

在条件语句与循环语句建立基本块时，要利用此计数器保证名字不同，如循环语句：

```
auto judgeBB = BasicBlock::create(module.get(), "judge_"+std::to_string(c), cur_fun);
auto whileBB = BasicBlock::create(module.get(), "while_"+std::to_string(c), cur_fun);
auto nextBB = BasicBlock::create(module.get(), "next_"+std::to_string(c++), cur_fun);
builder->create_br(judgeBB);
```

```
builder->set_insert_point(judgeBB);
node.expression->accept(*this);
auto cmp = result;
if (result->get_type() == INT32_T) {
    cmp = builder->create_icmp_ne(result, CONST_INT(0));
}
else if (result->get_type() == FLOAT_T) {
    cmp = builder->create_fcmp_ne(result, CONST_FP(0.));
}
builder->create_cond_br(cmp, whileBB, nextBB);

builder->set_insert_point(whileBB);
node.statement->accept(*this);
builder->create_br(judgeBB);

builder->set_insert_point(nextBB);
c++;
```

由于计算出来的结果未必是 INT1 类型，可能需要类型转换，条件也是同理：

```
node.expression->accept(*this);
auto cmp = result;
if (result->get_type() == INT32_T) {
    cmp = builder->create_icmp_ne(result, CONST_INT(0));
}
else if (result->get_type() == FLOAT_T) {
    cmp = builder->create_fcmp_ne(result, CONST_FP(0.));
}
auto trueBB = BasicBlock::create(module.get(), "true_"+std::to_string(c), cur_fun);
auto nextBB = BasicBlock::create(module.get(), "next_"+std::to_string(c), cur_fun);
if (node.else_statement != nullptr) {
    auto falseBB = BasicBlock::create(module.get(), "false_"+std::to_string(c++), cur_fun);
    builder->create_cond_br(cmp, trueBB, falseBB);
    builder->set_insert_point(falseBB);
    node.else_statement->accept(*this);
    builder->create_br(nextBB);
}
else {
    c++;
    builder->create_cond_br(cmp, trueBB, nextBB);
}
builder->set_insert_point(trueBB);
node.if_statement->accept(*this);
builder->create_br(nextBB);
builder->set_insert_point(nextBB);
```

对变量与赋值语句，需要注意的是，当访问变量时，需要获取变量的值，而赋值则需要获取地址，此部分具体内容在之后细节处叙述。

对表达式的计算，只要两边一方为浮点数，就需要进行浮点转换，否则进行整数运算。而布尔表达式，通过类型转化与零扩展参与浮点数/整数的运算，此处以 Term 为例：

```
if (node.additive_expression == nullptr) {
    node.term->accept(*this);
    return;
}
node.additive_expression->accept(*this);
auto lval = result;
node.term->accept(*this);
auto rval = result;
int if_float = 0;
if (lval->get_type() == FLOAT_T || rval->get_type() == FLOAT_T) if_float = 1;
if (if_float) {
    if (lval->get_type() != FLOAT_T) lval = builder->create_sitofp(lval, FLOAT_T);
    if (rval->get_type() != FLOAT_T) rval = builder->create_sitofp(rval, FLOAT_T);
    if (node.op == OP_PLUS) result = builder->create_fadd(lval, rval);
    else if (node.op == OP_MINUS) result = builder->create_fsub(lval, rval);
}
else {
    if (lval->get_type() != INT32_T) lval = builder->create_zext(lval, INT32_T);
    if (rval->get_type() != INT32_T) rval = builder->create_zext(rval, INT32_T);
    if (node.op == OP_PLUS) result = builder->create_iadd(lval, rval);
    else if (node.op == OP_MINUS) result = builder->create_isub(lval, rval);
}
```

Simple Expression 与 Addictive Expression 也是类似的，只是具体生成的运算指令有差异而已。

最后，调用语句，需要先计算每个参数再进行调用，具体细节见下一部分。

细节处理：

1、全局变量

上面所述的变量建立只考虑了局部变量，而当 cur_fun 为空时，代表变量的建立不在任何函数中，即为全局变量，需要额外进行建立，这里仿照 gcd builder 的写法：

```
if (cur_fun == nullptr) {
    if (node.num == nullptr) {
        if (node.type == TYPE_INT) {
            auto ini = ConstantZero::get(INT32_T, module.get());
            auto val = GlobalVariable::create(node.id, module.get(), INT32_T, false, ini);
            scope.push(node.id, val);
        }
        else {
            auto ini = ConstantZero::get(FLOAT_T, module.get());
            auto val = GlobalVariable::create(node.id, module.get(), FLOAT_T, false, ini);
            scope.push(node.id, val);
        }
    }
    else {
        if (node.type == TYPE_INT) {
            auto *arrayType = ArrayType::get(INT32_T, node.num->i_val);
            auto ini = ConstantZero::get(INT32_T, module.get());
            auto val = GlobalVariable::create(node.id, module.get(), arrayType, false, ini);
            scope.push(node.id, val);
        }
        else {
            auto *arrayType = ArrayType::get(FLOAT_T, node.num->i_val);
            auto ini = ConstantZero::get(FLOAT_T, module.get());
            auto val = GlobalVariable::create(node.id, module.get(), arrayType, false, ini);
            scope.push(node.id, val);
        }
    }
}
```

2、越界检查

对于数组的下标，需要额外进行检查。也即增加一个 if 语句，当计算出的下标(类型转换为 INT32)小于 0 时，直接调用报错[这里展示了变量中的，赋值语句也是同理]：

```
if (node.expression != nullptr) {
    node.expression->accept(*this);
    auto relpos = result;
    if (result->get_type() == FLOAT_T)
        relpos = builder->create_fptosi(result, INT32_T);
    else if (result->get_type() == INT1_T)
        relpos = builder->create_zext(result, INT32_T);
    auto cmp = builder->create_icmp_lt(relpos, CONST_INT(0));
    auto falseBB = BasicBlock::create(module.get(), "false_"+std::to_string(c), cur_fun);
    auto nextBB = BasicBlock::create(module.get(), "next_"+std::to_string(c++), cur_fun);
    builder->create_cond_br(cmp, falseBB, nextBB);
    builder->set_insert_point(falseBB);
    builder->create_call(scope.find("neg_idx_except"), {});
    builder->create_br(nextBB);
    builder->set_insert_point(nextBB);
```

4、赋值语句

赋值语句中，有一点十分值得注意：其左侧的变量需要寻找地址，因此不能调用 `visit`，而是直接寻找类似下图：

```
node.expression->accept(*this);
auto res = result;
auto pos = scope.find(node.var->id);
if (node.var->expression != nullptr) {
    node.var->expression->accept(*this);
    auto relpos = result;
    if (result->get_type() == FLOAT_T) {
        relpos = builder->create_fptosi(result, INT32_T);
    }
    else if (result->get_type() == INT1_T) {
        relpos = builder->create_zext(result, INT32_T);
    }
```

3、一般类型转换

除了比较时的类型转换，事实上还有很多地方需要进行类型的匹配，例如变量赋值时与函数调用时，都必须保证类型的一致。较为简单的类型转换是 `INT1`、`INT32` 与 `FLOAT` 之间，基本可以通过调用解决。如 `call` 部分的代码：

```
auto pos = scope.find(node.id);
std::vector<Value *> arguments;
for (int i = 0; i < node.args.size(); i++) {
    node.args[i]->accept(*this);
    auto argument = result;
    auto argtype = static_cast<FunctionType *>(pos->get_type())->get_param_type(i);
    if (argtype == INT32_T) {
        if (result->get_type() == INT1_T)
            argument = builder->create_zext(result, INT32_T);
        else if (result->get_type() == FLOAT_T) {
            argument = builder->create_fptosi(result, INT32_T);
        }
    }
    else if (argtype == FLOAT_T) {
        if (result->get_type() != FLOAT_T)
            argument = builder->create_sitofp(result, FLOAT_T);
    }
```

通过每次考虑参数列表中的参数类型来实现类型的转换，之后再将参数传入

```
    arguments.push_back(argument);
}
result = builder->create_call(pos, arguments);
```

4、数组操作

这里大概是全实验最复杂的部分。实际操作中的指针有两种类型，以整型为例，有直接建立数组所出现的 `int (*p)[10]`类型与函数形参中的 `int *p` 类型，而对两者的 `gep` 操作也不同。

更值得注意的是，由于变量建立时的不同，通过名字寻找 `int *p` 类型的变量实际找到的是 `int **p` 类型，因此需要的是先 `load`，再 `gep`。以 `call` 函数调用数组为例，实际的

操作如下：

```
if (argtype == INT32_T) {
    if (result->get_type() == INT1_T)
        argument = builder->create_zext(result, INT32_T);
    else if (result->get_type() == FLOAT_T) {
        argument = builder->create_fptosi(result, INT32_T);
    }
}
else if (argtype == FLOAT_T) {
    if (result->get_type() != FLOAT_T)
        argument = builder->create_sitofp(result, FLOAT_T);
}
else {
    auto ty = result->get_type()->get_pointer_element_type();
    if (ty != INT32PTR_T && ty != FLOATPTR_T)
        argument = builder->create_gep(result, {CONST_INT(0), CONST_INT(0)});
    else argument = builder->create_load(result);
}
arguments.push_back(argument);
```

当参数类型不是数时，代表为数组指针，而这里需要传入的是 `int *`类型。判断类型是 `int (*p)[10]`还是 `int **p` 可以通过指向的内容来判断，若指向的是指针类型，则为后者。

对赋值语句，同理进行判断：

```
node.var->expression->accept(*this);
auto relpos = result;
if (result->get_type() == FLOAT_T) {
    relpos = builder->create_fptosi(result, INT32_T);
}
else if (result->get_type() == INT1_T) {
    relpos = builder->create_zext(result, INT32_T);
}
auto cmp = builder->create_icmp_lt(relpos, CONST_INT(0));
auto falseBB = BasicBlock::create(module.get(), "false_"+std::to_string(c), cur_fun);
auto nextBB = BasicBlock::create(module.get(), "next_"+std::to_string(c++), cur_fun);
builder->create_cond_br(cmp, falseBB, nextBB);
builder->set_insert_point(falseBB);
builder->create_call(scope.find("neg_idx_except"), {});
builder->create_br(nextBB);
builder->set_insert_point(nextBB);
auto ty = pos->get_type()->get_pointer_element_type();
if (ty != INT32PTR_T && ty != FLOATPTR_T) {
    pos = builder->create_gep(pos, {CONST_INT(0), relpos});
}
else {
    pos = builder->create_load(pos);
    pos = builder->create_gep(pos, {relpos});
}
```

通过不同的情况得到真正需要赋值的指针位置。

Lab 4.1 实验报告

PB20000296 郑滕飞

思考题：

1、

支配性：基本块 i 位于起点到 j 的每条路径上，则称 i 支配 j。
严格支配性：i 支配 j 且不是 j，则称 i 严格支配 j。
直接支配性：i 为严格支配 j 的结点中离 j 最近的结点，则称 i 是 j 的直接支配结点。
[算了一下，虽然这个定义写得看起来很不靠谱，不过唯一性是可以证明的；入口没有直接支配结点。]
支配边界：n 支配 m 的某个前驱且 n 不严格支配 m，所有这样的 m 构成 n 的支配边界。

2、

概念：通过控制流从哪条边进入基本程序块确定取值选择的函数。
意义：合并来自不同边的值，保证静态单赋值形式能在汇合处调和为一个名字，从而每个定义创建唯一名字、每个使用处引用一个定义的要求能实现。

3、

1 func label_entry 中的 store：被消除。对局部变量的 store 压栈作为 lval 的定值。
2 func label_entry 中的 load：被消除，对局部变量的 load 用最新定值替换。
3 func label6 中的 store：被消除，同 1。
4 func label7 中的 load：基本块汇合处，最新定值由 phi 替换。
5 main 对 globVar 的 store：不变，全局变量不会被替代。
6 main 对 arr 的 store：不变，数组不会被替代。
7 main 对 b 的 store：被消除，同 1。
8 main 对 b 的 load：被消除，同 2。
9 main 对 globVar 的 load：不变，同 5。

4、

步骤一中在 global_live_var_name 中保存所有局部变量，而在 live_var_2blocks 中保存了它所活跃的基本块，步骤二对每个局部变量的所有活跃基本快执行:

```
for (auto bb_dominance_frontier_bb : dominators_->get_dominance_frontier(bb)) {
    if (bb_has_var_phi.find({bb_dominance_frontier_bb, var}) == bb_has_var_phi.end()) {
        // generate phi for bb_dominance_frontier_bb & add bb_dominance_frontier_bb to work list
        auto phi =
            PhiInst::create_phi(var->get_type()->get_pointer_element_type(), bb_dominance_frontier_bb);
        phi->set_lval(var);
        bb_dominance_frontier_bb->add_instr_begin(phi);
        work_list.push_back(bb_dominance_frontier_bb);
        bb_has_var_phi[{bb_dominance_frontier_bb, var}] = true;
    }
}
```

其中利用 dominators_的成员函数 get_dominance_frontier 获取了支配树中此基本块的支配者，只要其中还没有建立对应的 phi，就需要进行建立。
[此处只是建立，补充完整需要在确定基本块出口处的值之后]

5、

代码中为每个变量维护了一个栈 `var_val_stack`，用于替换 `lval`。

在单个基本块中，如果有对此局部变量的 `phi` 指令，将最新定值确定为 `phi` 指令的结果压栈：

```
if (instr->is_phi()) {
    auto l_val = static_cast<PhiInst *>(instr)->get_lval();
    var_val_stack[l_val].push_back(instr);
}
```

若出现 `store` 指令，将右值作为新定值压栈：

```
if (instr->is_store()) {
    auto l_val = static_cast<StoreInst *>(instr)->get_lval();
    auto r_val = static_cast<StoreInst *>(instr)->get_rval();

    if (!IS_GLOBAL_VARIABLE(l_val) && !IS_GEP_INSTR(l_val)) {
        var_val_stack[l_val].push_back(r_val);
        wait_delete.push_back(instr);
    }
}
```

使用时从栈顶调取：

```
if (instr->is_load()) {
    auto l_val = static_cast<LoadInst *>(instr)->get_lval();

    if (!IS_GLOBAL_VARIABLE(l_val) && !IS_GEP_INSTR(l_val)) {
        if (var_val_stack.find(l_val) != var_val_stack.end()) {
            // 此处指令替换会维护 UD 链与 DU 链
            instr->replace_all_use_with(var_val_stack[l_val].back());
            wait_delete.push_back(instr);
        }
    }
}
```

最后离开基本快时消除基本块中的定值：

```
for (auto &instr1 : bb->get_instructions()) {
    auto instr = &instr1;

    if (instr->is_store()) {
        auto l_val = static_cast<StoreInst *>(instr)->get_lval();
        if (!IS_GLOBAL_VARIABLE(l_val) && !IS_GEP_INSTR(l_val)) {
            var_val_stack[l_val].pop_back();
        }
    } else if (instr->is_phi()) {
        auto l_val = static_cast<PhiInst *>(instr)->get_lval();
        if (var_val_stack.find(l_val) != var_val_stack.end()) {
            var_val_stack[l_val].pop_back();
        }
    }
}
```

Lab 4.2 实验报告

PB20000296 郑滕飞

准备工作：

1、常量折叠扩充

为了方便常量折叠的计算，我扩充了用 op 计算折叠的方式，如：

```
class ConstFolder {
  public:
    ConstFolder(Module *m) : module_(m) {}
    Constant *compute(Instruction *instr, Constant *value1, Constant *value2);
    Constant *compute(Instruction::OpID op, Constant *value1, Constant *value2);
    Constant *compute(CmpInst::CmpOp op, Constant *value1, Constant *value2);
    Constant *compute(FCmpInst::CmpOp op, Constant *value1, Constant *value2);
    Constant *compute(Instruction *instr, Constant *value1);

  private:
    Module *module_;
};
```

实现例如：

```
Constant *ConstFolder::compute(Instruction::OpID op, Constant *value1, Constant *value2) {
    switch (op) {
    case Instruction::add: return ConstantInt::get(get_const_int_value(value1) + get_const_int_value(value2), module_);
    case Instruction::sub: return ConstantInt::get(get_const_int_value(value1) - get_const_int_value(value2), module_);
    case Instruction::mul: return ConstantInt::get(get_const_int_value(value1) * get_const_int_value(value2), module_);
    case Instruction::sdiv: return ConstantInt::get(get_const_int_value(value1) / get_const_int_value(value2), module_);
    case Instruction::fadd: return ConstantFP::get(get_const_fp_value(value1) + get_const_fp_value(value2), module_);
    case Instruction::fsub: return ConstantFP::get(get_const_fp_value(value1) - get_const_fp_value(value2), module_);
    case Instruction::fmul: return ConstantFP::get(get_const_fp_value(value1) * get_const_fp_value(value2), module_);
    case Instruction::fdiv: return ConstantFP::get(get_const_fp_value(value1) / get_const_fp_value(value2), module_);
    }
}
```

2、表达式类型定义

根据 instr 的区分，可以将类型分为如下部分：

```
class Expression {
  public:
    // TODO: you need to extend expression types according to testcases
    enum gvn_expr_t { e_constant, e_bin, e_phi, e_cmp, e_fcmp, e_gep, e_uni, e_call, e_incom};
    Expression(gvn_expr_t t) : expr_type(t) {}
    virtual ~Expression() = default;
    virtual std::string print() = 0;
    gvn_expr_t get_expr_type() const { return expr_type; }

  private:
    gvn_expr_t expr_type;
};
```

依次为：常数类型、二元运算(整数/浮点数的加减乘除)、整数比较、浮点数比较、取指针操作、单元运算(三个类型转换)、纯函数调用，与其他的“不可比较”类型。

这其中，不可比较类型事实上是非纯函数调用、load 指令与 alloca 指令，共同特点是当且仅当同一句才会认为相等。

对于已经实现的 constant、bin 与 phi，我加入了一些取部分的函数，方便之后操作，如：

```
Constant *get_const() const { return c_; }
```

```
Instruction::OpID get_op() const { return op_; }
std::shared_ptr<Expression> get_l() const { return lhs_; }
std::shared_ptr<Expression> get_r() const { return rhs_; }
```

其他则是类似进行处理，例如纯函数调用的 `call` 指令：

```
class CallExpression : public Expression {
  public:
    static std::shared_ptr<CallExpression> create(Function *func,
                                                  std::vector<std::shared_ptr<Expression>> opes) {
        return std::make_shared<CallExpression>(func, opes);
    }
    virtual std::string print() {
        std::string ret = "(purecall " + func_->get_name();
        for (int i = 0; i < opes_.size(); i++)
            ret += " " + opes_[i]->print();
        ret += ")";
        return ret;
    }

    bool equiv(const CallExpression *other) const {
        if (func_ != other->func_)
            return false;
        if (opes_.size() != other->opes_.size())
            return false;
        for (int i = 0; i < opes_.size(); i++)
            if (!(*opes_[i] == *other->opes_[i]))
                return false;
        return true;
    }

    CallExpression(Function *func, std::vector<std::shared_ptr<Expression>> opes)
        : Expression(e_call), func_(func), opes_(opes) {}

  private:
    Function *func_;
    std::vector<std::shared_ptr<Expression>> opes_;
};
```

整体框架：

1、Detect Equivalences

这个函数的整体构造分为四个部分：预处理、计算 `pin`、计算 `pout`、终止检测。预处理部分，需要初始化所有 `pout` 为 `Top`，第一个块的 `pin` 为空，以及加入函数的参数和全局变量，如加入参数部分：

```
for (auto it = func_->arg_begin(); it != func_->arg_end(); it++) {
    Value *arg = *it;
    std::shared_ptr<Expression> exp = InComparible::create(arg);
    auto cc = createCongruenceClass(next_value_number_++);
    cc->leader_ = arg;
    cc->members_ = {arg};
    cc->value_expr_ = exp;
    cc->value_phi_ = nullptr;
    pin_[entry].insert(cc);
}
```

对顶元 `Top` 的设计，我定义为“只有一个 `index` 为 `0` 的等价类的 `partition`”，在 `join`

中对此单独区分。

接着，在 do while 循环内，根据前驱计算 pin：

```
std::list<BasicBlock *> pres = bb.get_pre_basic_blocks();
if (pres.size() == 1) {
    BasicBlock *pre = *pres.begin();
    pin_[&bb] = clone(pout_[pre]);
} else if (pres.size() == 2) {
    BasicBlock *pre1 = *pres.begin();
    BasicBlock *pre2 = *pres.rbegin();
    pin_[&bb] = join(pout_[pre1], pout_[pre2]);
}
partitions p = clone(pin_[&bb]);
```

从 pin 出发，遍历当前基本块语句(返回空的语句无需处理，phi 是在前驱中处理的)与之后的 phi，计算 pout：

```
// iterate through all instructions in the block
for (auto &instr : bb.get_instructions()) {
    if (!instr.is_void() && !instr.is_phi())
        p = transferFunction(&instr, &instr, p, &bb);
}

// and the phi instruction in all the successors
for (auto &next : bb.get_succ_basic_blocks()) {
    pres = next->get_pre_basic_blocks();
    if (pres.size() == 2) {
        int v = 2;
        if (&bb == *pres.begin())
            v = 0;
        for (auto &instr : next->get_instructions()) {
            if (instr.is_phi()) {
                p = transferFunction(&instr, instr.get_operand(v), p, &bb);
            } else
                break;
        }
    }
}
```

最后，检测并复制，得到终止条件：

```
// check changes in pout
if (p != pout_[&bb]) {
    changed = true;
}
pout_[&bb] = std::move(p);
```

只要一轮循环后，没有 partition 发生改变，就终止迭代。

2、join 与 intersect

在计算 pin 的过程中若有两个前驱，需要进行汇合，代码如下：

```
GVN::partitions GVN::join(const partitions &P1, const partitions &P2) {
    // TODO: do intersection pair-wise
    if (P1.empty() || P2.empty())
        return {};
    if ((*P1.begin())->index_ == 0)
        return P2;
    if ((*P2.begin())->index_ == 0)
        return P1;
    partitions P = {};
    for (auto i = P1.begin(); i != P1.end(); i++)
        for (auto j = P2.begin(); j != P2.end(); j++) {
            std::shared_ptr<CongruenceClass> k = intersect(*i, *j);
            if (k != nullptr)
                P.insert(k);
        }
    return P;
}
```

若有空直接返回空，否则进行顶元检测，若均非顶元则类似伪代码进行 `intersect`。取交的具体操作如下：

```
std::shared_ptr<CongruenceClass> GVN::intersect(std::shared_ptr<CongruenceClass> Ci,
                                                std::shared_ptr<CongruenceClass> Cj) {
    // TODO
    std::set<Value *> common_members;
    std::set_intersection(Ci->members_.begin(), Ci->members_.end(),
                          Cj->members_.begin(), Cj->members_.end(),
                          std::inserter(common_members, common_members.begin()));
    if (common_members.empty()) return nullptr;
    if (Ci->value_expr_ == Cj->value_expr_) {
        CongruenceClass k = *Ci;
        k.members_ = common_members;
        k.leader_ = *common_members.begin();
        return std::make_shared<CongruenceClass>(k);
    }
    CongruenceClass k(next_value_number_++);
    k.members_ = common_members;
    k.leader_ = *common_members.begin();
    k.value_expr_ = k.value_phi_ = PhiExpression::create(Ci->value_expr_, Cj->value_expr_);
    return std::make_shared<CongruenceClass>(k);
}
```

由于汇合到新基本块中，不会与旧的共用 `index`，当 ve 相等时可以直接更改 `members`。不等时，则直接创建新类并建立 `phi` 函数。

3、运算符重载

[对于所有==运算符，我都重载了对应的 `x != y` 为`!(x == y)`，避免错误]

等价类的比较事实上只需要比较成员是否对应相等：

```
bool CongruenceClass::operator==(const CongruenceClass &other) const {
    // TODO: which fields need to be compared?
    return members_ == other.members_;
}
```

这是由于当出现循环时，value expression 事实上是 phi 函数无穷递归的结构，每次合并都会更新。现实中，每走一层会计算一层的 expression，通过惰性求值避免了无限继续，但分析时会无限递归，因此只要 members 收敛就认为收敛。

对 partitions 的比较，只需要比较其中等价类：

```
bool operator==(const GVN::partitions &p1, const GVN::partitions &p2) {
    // TODO: how to compare partitions?
    if (p1.size() != p2.size()) return false;
    auto a1 = p1.begin(), a2 = p2.begin();
    while (a1 != p1.end()) {
        if (**a1 != **a2)
            return false;
        a1++;
        a2++;
    }
    return true;
}
```

Tranfer Function:

1、结构

我的 transfer function 整体思路是：

找 x，若找到，删除并对应调整等价类
找 e，若找到，x 加入 e 所在等价类，return
ve = valueexpr(e,pout)
vpf = valuephifunc(ve,bb)
if (vpf != null) ve = vpf;
找 ve，若找到，x 加入 ve 是 value expr 的等价类，return
若 ve 是 const，以 x 建 const_leader 等价类，插入，return
否则，以 x 建 x_leader 等价类，插入，return

这里，所有的表达式最终统一展开为 ve，事实上我的代码里不需要 vpf 字段，只需要通过 ve 是否是 phi 来控制。

这个结构没有特别处理 copy statement，因为大部分情况都会在找 e 的部分解决，除非 e 是常数，这样进等价类不影响结果。

2、getVN 与重载

getVN 我进行了如下重载：

```
std::shared_ptr<Expression> GVN::getVN(const partitions &pout, Value *v) {
    Constant *t = dynamic_cast<Constant *>(v);
    if (t != nullptr)
        return ConstantExpression::create(t);
    for (auto it = pout.begin(); it != pout.end(); it++)
        for (auto m = (*it)->members_.begin(); m != (*it)->members_.end(); m++)
            if (*m == v)
                return (*it)->value_expr_;
    return nullptr;
}
```

这个重载代表对 Value *找对应的 expression，常数或者找到都会返回(同样，对 ve

找 expression 若找到也是直接返回 ve)。

3、具体实现

有了 getVN，就可以实现寻找 x 与寻找 e(这里，若 e 常数未找到，会进行后续代码)：

```
auto a = getVN(pout, static_cast<Value *>(x));
if (a != nullptr) {
    for (auto it = pout.begin(); it != pout.end(); it++)
        if ((*it)->value_expr_ == a) {
            (*it)->members_.erase(static_cast<Value *>(x));
            if ((*it)->members_.empty()) {
                pout.erase(it);
            }
            else if ((*it)->leader_ == x) {
                (*it)->leader_ = *(*it)->members_.begin();
            }
            break;
        }
}

a = getVN(pout, e);
if (a != nullptr) {
    for (auto it = pout.begin(); it != pout.end(); it++)
        if ((*it)->value_expr_ == a) {
            (*it)->members_.insert(x);
            return pout;
        }
}
```

接着按伪代码确定是否要新建等价类：

```
std::shared_ptr<Expression> exp = valueExpr(e, pout);

if (exp == nullptr)
    return pout;
std::shared_ptr<Expression> phi = valuePhiFunc(exp, bb);
if (phi != nullptr) {
    exp = phi;
    if (phi->get_expr_type() != Expression::e_phi)
        phi = nullptr;
}
auto t = getVN(pout, exp);
if (t != nullptr) {
    for (auto it = pout.begin(); it != pout.end(); it++)
        if ((*it)->value_expr_ == t) {
            (*it)->members_.insert(x);
            return pout;
        }
}
auto cc = createCongruenceClass(next_value_number_++);
if (exp->get_expr_type() == Expression::e_constant)
    cc->leader_ = find_const(exp);
else
    cc->leader_ = x;
cc->members_ = {x};
cc->value_expr_ = exp;
if (phi == nullptr)
    cc->value_phi_ = nullptr;
else
    cc->value_phi_ = to_phi(phi);
pout.insert(cc);
return pout;
```

值得一提的是，这里 valuePhiFunc 的返回值未必是 PhiExpression 的智能指针，因为我判断了若左右表达式相等则不建立 phi，所以会有第一段的讨论。

to_phi(x)是宏定义：(std::static_pointer_cast<PhiExpression>(x))。

4、ValueExpr

这个函数用于将指令转化为对应的表达式，仍以 call 举例：

```
if (ins->is_call()) {
    std::vector<Value *> opes = ins->get_operands();
    Function *func = static_cast<Function *>(opes[0]);
    if (!func_info_->is_pure_function(func))
        return InComparible::create(instr);
    int op_num = opes.size() - 1;
    std::vector<std::shared_ptr<Expression>> opes_e(op_num);
    for (int i = 0; i < op_num; i++) {
        opes_e[i] = getVN(p, opes[i + 1]);
        if (opes_e[i] == nullptr)
            return nullptr;
    }
    return CallExpression::create(func, opes_e);
}
```

若是纯函数调用，则建立 Call，这里 getVN 是之前重载的，无论是找到还是常数都得到了对应的表达式。否则，进入不可比较类型，相等当且仅当是同一句。

而对于常数折叠，以 bin 举例，左右都是常数时进行折叠：

```
if (ins->isBinary()) {
    Value *left = ins->get_operand(0);
    Value *right = ins->get_operand(1);
    std::shared_ptr<Expression> l, r;
    bool c = true;
    l = getVN(p, left);
    if (l == nullptr)
        return nullptr;
    c = c and (l->get_expr_type() == Expression::e_constant);
    r = getVN(p, right);
    if (r == nullptr)
        return nullptr;
    c = c and (r->get_expr_type() == Expression::e_constant);
    if (c) {
        Constant *temp = folder_->compute(ins, find_const(l), find_const(r));
        return ConstantExpression::create(temp);
    }
    return BinaryExpression::create(ins->get_instr_type(), l, r);
}
```

其中 to_const(x)宏定义函数：

```
(static_cast<ConstantExpression *>(exp.get())->get_const(x))
```

5、ValuePhiFunc

这个函数总体实现与伪代码相同，实际上只需要考虑类型是 bin 且两个子表达式类型都是 phi 的情况。

函数的过程分为判定、计算左右(折叠)、递归，具体如下：

```
if (ve->get_expr_type() == Expression::e_bin) {
    BinaryExpression *b = static_cast<BinaryExpression *>(ve.get());
    shared_ptr<Expression> l1, l2, r1, r2;
    if (b->get_l()->get_expr_type() == Expression::e_phi) {
        shared_ptr<PhiExpression> t = to_phi(b->get_l());
        l1 = t->get_l();
        r1 = t->get_r();
    } else {
        return nullptr;
    }
    if (b->get_r()->get_expr_type() == Expression::e_phi) {
        shared_ptr<PhiExpression> t = to_phi(b->get_r());
        l2 = t->get_l();
        r2 = t->get_r();
    } else {
        return nullptr;
    }
```

```
shared_ptr<Expression> l, r;
if (l1->get_expr_type() != Expression::e_constant || l2->get_expr_type() != Expression::e_constant) {
    l = BinaryExpression::create(b->get_op(), l1, l2);
} else {
    l = ConstantExpression::create(folder_->compute(b->get_op(), find_const(l1), find_const(l2)));
}
if (r1->get_expr_type() != Expression::e_constant || r2->get_expr_type() != Expression::e_constant) {
    r = BinaryExpression::create(b->get_op(), r1, r2);
} else {
    r = ConstantExpression::create(folder_->compute(b->get_op(), find_const(r1), find_const(r2)));
}
```

```
std::list<BasicBlock *> pres = bb->get_pre_basic_blocks();
shared_ptr<Expression> l0, r0;
if (l->get_expr_type() == Expression::e_constant)
    l0 = l;
else {
    BasicBlock *prel = *pres.begin();
    partitions Pl = pout_[prel];
    l0 = getVN(Pl, l);
    if (l0 == nullptr)
        l0 = valuePhiFunc(l, prel);
}
if (r->get_expr_type() == Expression::e_constant)
    r0 = r;
else {
    BasicBlock *prer = *pres.rbegin();
    partitions Pr = pout_[prer];
    r0 = getVN(Pr, r);
    if (r0 == nullptr)
        r0 = valuePhiFunc(r, prer);
}
if (l0 != nullptr && r0 != nullptr) {
    if (l0 != r0)
        return PhiExpression::create(l0, r0);
    else
        return l0;
}
```

我传入了 bb，用于找到前驱的 pout，其他基本与伪代码相同。

利用完成的 transfer function 与上方框架，就能得到最终的结果。

思考题：

1．请简要分析你的算法复杂度。

对每个 bb 里的每个语句至多进入一次 transfer function（作为 phi 在之前进入，或作为普通语句在遍历时进入），transfer function 中，除了 ValuePhiFunc 之外都是接近常数时间复杂度的(这是由于 ValueExpr 运用了动态规划的思想优化)，而 vpf 的递归层数也至多是不重复的基本块嵌套层数，因此，当基本块数量与每个基本块中的语句数量不太大时，基本是与外循环总次数乘总非空语句数正比。

当数量较大时，每次查找所在等价类与 vpf 的遍历阶数都近似 log，因此假设循环次数 c，语句数 m，bb 数 n，则复杂度可以被控制在 O(cm*log m*log n)。

2、std::shared_ptr 如果存在环形引用，则无法正确释放内存，你的 Expression 类是否存在 circular reference?

不存在，每次嵌套会产生新的值表达式，是严格的树的形式。

3、尽管本次实验已经写了很多代码，但是在算法上和工程上仍然可以对 GVN 进行改进，请简述你的 GVN 实现可以改进的地方。

从算法的角度，value_phi_func_字段并无必要，可以统一成 value_expr_的形式。ValuePhiFunc 还可以进行更多合并，如针对 cmp 与 fcmp 的合并，或者是 a + Phi(b, c) 这类情况的合并处理，统一成展开形式。替换的逻辑中，如果 br 确定为 true 或 false，可以将不可到达的 bb 删除，或是 bb 前驱减少后 phi 变成直接的等价语句，重新进入循环。针对 load 和 store 也可以进行一定的等价类分析(如两个 load 中无 store 则等价)。

从实现的角度，中间的一些搜索过程是重复的，可以考虑分类得更细以降低整体复杂度。

总结：

1.09 14:25 正式开工
1.11 00:39 过 single_bb1
1.11 23:43 几乎完成(39.2/40)
1.12 14:08 最终完成
确实很棒的体验，不过这个体验是以 bug 都 de 出来了为前提的）
总之恭喜编译实验完结!