Mem2reg

支配树

支配（dominate）：从入口出发，凡是到达节点 Y 的所有路径都必须经过节点 X，就说 X 支配 Y。入口块支配全世界，每个节点也支配自己。

• 严格支配（strict dominate）：X 支配 Y，且 X ≠ Y。

直接支配（idom）：在所有“严格支配 Y”的节点里，离 Y 最近的那个，就是 Y 的 idom。支配树的父子关系就是“父 = idom(子)”，整棵树以入口块为根；

半支配：算法里用的中间量。直观描述：semiDom(v) 是从入口到 v 的 DFS 序里，编号最小的那个祖先 u，满足“在 DFS 树上看某个节点 v，找一个祖先 u（DFS 编号比 v 小），要求存在一条从 u 到 v 的路径，且这条路径上所有“中间节点”的 DFS 编号都大于 u。满足条件的那些 u 里，编号最小的就是 semiDom(v)。”

理解：

1. 首先从开始 然后按照dfs遍历树 得到顺序 root是1 然后依次往下标 并且记住dfs遍历顺序，为了知道谁是谁祖宗
2. 然后就可以求了 记住 x是y的半支配意思就是说：首先x一定得是y的祖宗 并且 x到y之间存在dfs路径使得 所有途经节点dfs编号都>x 且 x一定小于y 且 x是满足到y之间存在dfs路径使得 所有途经节点dfs编号都>x里的可能的众多节点里面 编号最小的;

支配边界：
找节点 X 的支配边界：在原 CFG 中，看所有“至少有一个前驱在 X 支配的子树里，至少有一个前驱不在”的合流节点 Y。这样的 Y 就在 DF(X)。
◦ 直观：从 X 往下的控制范围里分了叉（多个后继或子树内有分叉），这些分叉的路径在某个合流点 Y 再次汇合，而 X 并不支配 Y——那个汇合点就是 X 的支配边界。
◦ 常见情形：一个 if 分叉的两个分支再汇合的块，就是分叉点的 DF 里的元素；循环头也常出现在前驱的 DF 里（例如来自循环体和外侧的前驱）。

理解：
就是首先从cfg看 支配边界（x）=y y一定比x更下面 而且什么时候成立呢 就是当 y 想到entry的话 可以走 x 也可以走其他路 就认为 x的支配边界里面有y

例子：

cfg:
A(entry)
|\
| \
B C
 \ /
  D
  |
  E
  |
  F
  |
  D (回边 F→D)

|
G(exit)

边：A→B, A→C, B→D, C→D, D→E, E→F, F→D（回边）, D→G。

1. DFS 顺序（从 A，先左后右）
   A=1, B=2, D=3, E=4, F=5, C=6, G=7。DFS 树父子：A→B→D→E→F，A→C；G 是 D 的树后继。
2. 支配集合迭代求解
   • 初始化：dom(A)={A}；其余全体节点。
   • 迭代收敛后：
   ◦ dom(A) = {A}
   ◦ dom(B) = {A,B}
   ◦ dom(C) = {A,C}
   ◦ dom(D) = {A,D} （B、C 交汇处只有 A 保留）
   ◦ dom(E) = {A,D,E}
   ◦ dom(F) = {A,D,E,F}
   ◦ dom(G) = {A,D,G}
3. 严格支配 / 直接支配（idom）
   • 严格支配：去掉自身即可，例如 D 严格支配 G。
   • idom（最近的严格支配者）：
   ◦ idom(B)=A
   ◦ idom(C)=A
   ◦ idom(D)=A
   ◦ idom(E)=D
   ◦ idom(F)=E
   ◦ idom(G)=D
   支配树就是：A 为根，孩子 B,C,D；D 的孩子 E,G；E 的孩子 F。
4. 半支配（semi-dom，用 DFS 序最小祖先定义）
   对每个 v，找 DFS 序中编号最小的祖先 u，存在一条 u→…→v 的路径，且路径中间点的 DFS 编号都大于 u。
   • semiDom(B)=A（路径 A→B，中间无点）
   • semiDom(C)=A（路径 A→C）
   • semiDom(D)=A（路径 A→B→D，中间 B 编号 2 > 1）
   • semiDom(E)=A（路径 A→B→D→E，中间 2,3 > 1）
   • semiDom(F)=A（路径 A→B→D→E→F，中间 2,3,4 > 1）
   • semiDom(G)=A（路径 A→B→D→G，中间 2,3 > 1）
   入口 A 的 semiDom(A)=A。
5. 支配边界 DF（定义：DF(X) = { Y | X 支配 Y 的某个前驱，且 X 不支配 Y }）
   先列前驱：
   • pred(B)={A}, pred(C)={A}, pred(D)={B,C,F}, pred(E)={D}, pred(F)={E}, pred(G)={D}
   逐点计算：
   • A 支配所有节点，所以找不到“不被 A 支配”的 Y → DF(A)=∅
   • B：B 支配前驱 B，自身不支配 D，且 D 的前驱含 B → D ∈ DF(B)
   • C：同理 D ∈ DF(C)
   • D：D 支配前驱 D/E/F，但它也支配 E、F、G、本身 D，所以没有 Y 违反第二条件 → DF(D)=∅
   • E：E 支配前驱 E；查看各 Y：
   ◦ 对 D：pred(D) 有 F，E 支配 F（F 的 dom 含 E），但 E 不支配 D → D ∈ DF(E)
   ◦ 其它无
   → DF(E)={D}
   • F：F 只支配自己；pred(D) 有 F，F 不支配 D → D ∈ DF(F)

• G：无后续，DF(G)=∅
结果汇总
• dom 集合：上面第 2 步
• idom：B/A, C/A, D/A, E/D, F/E, G/D
• semi-dom：B,C,D,E,F,G 均为 A（入口），A 自身为 A
• DF：DF(B)={D}, DF(C)={D}, DF(E)={D}, DF(F)={D}, 其余为空

代码相关

只是说自己理解的：

核心思想就是把内存里的值写到寄存器里，实现加速
然后至于SSA，意思就是比如 func1(int &x){int a; x++;} func2(int &y){int a;y++;} 我们就对func1里的a标记为a1 func2里的a标记为a2 那对于局部变量比较好理解 因为后面不会再出现了 但如果是这样 比如 int main{int a=0; func1(a); func2(a); cout<<\a; } 那我们想想 假设最开始声明的是a0 func1里的是a1 func2的是a2 外面cout的a 理论上是a3 那a3的值应该是a1+1还是a2+1呢 a2的值是不是又应该继承a1呢对吧? 这个时候就是通过`phi`来决定

`phi`可以在控制流汇合处决定"从哪个前驱来的值"。上方例子具体就是说：
a3 必须由汇合块的 φ 决定：
◦ 如果 func1 返回后 a 成了 a1，func2 返回后成了 a2，回到 main 时在汇合块放 a3 = φ([来自 func1 路径: a1], [来自 func2 路径: a2])。哪个路径跑完回到这里，就选哪路的版本。
◦ 更常见的 if-else：if 分支里 x1=1，else 里 x2=2，汇合后 x3 = φ([x1, from if], [x2, from else])。

然后有关如何把内存里的值真正的写到寄存器里，这个就是重点了：

我们首先要知道，我们实际操作的是operand，也就是操作数，然后操作数基本可以分为三种，主要是寄存器操作数（就是某个寄存器） 以及 立即数/常量操作数（即某个具体的int/bool等类型的数值），还有一种少见的就是 指针的值（一般就是alloca来产生的地址）。 然后我们不难想像， reg里可能会存 int bool or float等类型，那我怎么知道某个reg存了哪个类型的值呢?

这个就是靠 ir_operand.h 里的 ME :: Operand 里，这里面记录了 Type ，通过读取这个属性 我就可以知道寄存器里存的东西的类型。

然后知道了类型之后，我们就可以开始考虑如何将内存的值写到寄存器里了。

其实一共mem2reg的全流程可以理解为10个步骤：

1. 最先是对程序构建cfg，cfg就是控制流图，其的基本组织结构就是基本块，也就是输出的.ll文件里的`block0 block1`这些。然后各个基本块之间可以通过`br/ret`等语句进行切换和跳转。 实际的构建就是通过先生成一个Analysis的实例AM,这个可以在analysis_manager.cpp内看见，然后再调用get函数，指定生成目标就是CFG，然后就得到了一个CFG(一个基本的CFG里会有id2block，就是block id 到 block指针的映射，还会有G也就是邻接表，存储CFG结构 invG是反向的邻接表（从后往前） G_id（只存id 省的再从指针->block id） 然后还有 inv G_id(反向))
CFG的基本构建思路就是：遍历所有的基本块，然后再读取每个块末尾的所有跳转指令，eg.无条件跳转br、条件跳转 br i1 cond、返回ret，然后根据这些跳转构建cfg的边（br一条，br i1 cond两条，ret一条），然后再保存block id到块指针的映射到id2block&边到G和invG；

---

2. 然后是第一次DCE:DCE就是清理终结指令（也就是跳转指令）后面的不可达的指令。
   那什么是不可达指令呢？其实很简单，如果一条指令前面有一个终结指令（跳转指令），那么执行到前面的终结指令的时候就已经跳转or返回了，肯定执行不到下面的指令，所以就是不可达。
   那为什么这么做呢？因为后续 mem2reg、插 φ的时候都假设基本块里的控制流是“以终结指令收尾、之后没有指令”。若块内还跟着不可达的指令，可能干扰 use/def 统计、支配分析和后续删除。
   函数执行链条？从runOnFunction函数第二步，调用上方的cleanTrailingUnreachable函数。这个函数的基本思想就是对每个基本块，从头扫描到遇到第一条“终结指令”。标记 □seenTerm=true 后，后面所有指令统统删掉。
   那如何判断是不是终结指令呢？就通过isTerminator（）函数，这个函数的返回值在 ir_instruction.h 内已经指明了，比如ret语句的isTerminator就会返回true。

---

3. 收集可以提升的变量：
   什么叫做可以提升的变量?
   这里的变量只包含标量alloca 也就是给单个标量变量分配的栈槽，类型就是类似i32/f32/i1之类的
   然后这里不对数组进行提升 类似int x; float x这样的简单语句就属于标量；

alloca到底是啥?
就是一个指令，作用是在函数栈帧上“划出一块槽位”用来存放局部变量/临时变量。函数一开始先在栈上挖个坑（槽位），返回一个指向这块内存的指针，后面的 store 往里写，load 从里读。

什么是“可以提升”?
alloca后得到的槽，如果地址不逃逸（不作为参数传给call语句&不参与算数orGEP放到别的内存&不能放进phi语句or ret语句返回）且只在这个函数内的load/store中使用，这样的就是可以改写成寄存器的SSA的，不再往内存走。

调用链条?
第三步->removeDeadDefs（）函数，该函数用一个while true循环删除（因为删了a可能导致b也没用，需要被删掉），然后在循环内删除死代码。

什么是死代码?
一条死代码会同时满足三件事：

1. 确实定义了某寄存器（是个 def）；

- 什么叫定义了一个寄存器?
- 如果一条指令产出了一个寄存器值（算术/比较/load/GEP/类型转换/call 返回/phi/alloca），我们就认为这个指令定义了一个寄存器，且其定义的寄存器的号码可以通过函数instructionDefines 得到；

2. 该寄存器的 use 计数为 0（没人用它的结果）；

- use 谁来计算?
- 依然在while内计算：在while循环体刚开始就是计算use:
  把“某个寄存器被当作输入(其实就等价于被读了,也就是load一次)用到几次”数出来，方便后面 DCE 判断“这个定义有没有用”。
  • useCnt：键是寄存器号，值是被用到的次数。
  • addUse(op)：给一个操作数打个“使用+1”记号。只统计寄存器类型，不是寄存器（比如立即数、null）就直接返回；指针为空也返回。
  • 双重循环遍历函数里的每个基本块、每条指令。
  • 对每种 opcode，挑出“这个指令读取了哪些寄存器”，逐个 addUse：
  ◦ LOAD：读取指针操作数（base 指针）。
  ◦ STORE：读取指针和要写入的值。
  ◦ 算术/比较：读取左右两个源操作数。

◦条件跳转：读取条件寄存器。
◦CALL：读取所有实参。
◦RET：读取返回值寄存器（如果有）。
◦GEP：读取基指针和所有索引。
◦转换指令（SITOFP/FPTOSI/ZEXT）：读取源。
◦PHI：读取所有 incoming 值。
• 默认分支：不认识或无读操作的 opcode 不计。
最终得到的 useCnt 映射告诉我们：每个寄存器作为“被使用方”出现了多少次

3. 指令无副作用（hasSideEffect 为假，而我们自己实现的时候，保守的认为 store/br/ret/call等语句都认为有副作用）。

- 什么叫sideeffect?
- 自己定义的 认为：store/br/ret/call等语句都有副作用，不进行删除;

简要总结?
这个函数只删无副作用且没人用的寄存器定义

什么叫寄存器定义?
包含：
算术类：ADD/SUB/MUL/DIV/MOD/SHL/ASHR/LSHR/BITAND/BITXOR/FADD/FSUB/FMUL/FDIV（写结果寄存器）。
• 比较类：ICMP/FCMP。
• 内存读：LOAD（写结果寄存器，值来自内存）。
• 地址计算：GETELEMENTPTR（写结果寄存器，指针计算结果）。
• 类型转换：SITOFP/FPTOSI/ZEXT（写目标寄存器）。
• 函数调用：CALL 若有返回值寄存器。
• ϕ 节点：PHI（写它的结果寄存器）。
• 分配：ALLOCA（写出分配得到的指针寄存器）。

• 外层 while(true) 循环：因为删掉一批指令后，可能让别的定义变成“没人用了”，所以要反复。
• 每轮里做两步：

1. 统计使用次数：useCnt 记录“寄存器号 → 被当作输入读过几次”。遍历全函数，每条指令把读到的寄存器都 addUse（指针、算术左右值、条件寄存器、call 实参、ret 结果、GEP 基址和索引、类型转换源、phi incoming 等）。
2. 删除无用定义：再遍历全函数，对每条指令检查：
   ■ 它是否定义了一个寄存器（instructionDefines）。
   ■ 这个寄存器在 useCnt 中是否为 0（没人读它）。
   ■ 这条指令是否无副作用（hasSideEffect 为 false）。
   三个条件同时满足就删掉这条指令。
   • removed 表示这一轮有没有删东西；若删了，继续下一轮重新统计 use；若一轮都删不掉，removed 仍为 false，跳出循环，函数结束（至少删过一轮）。

---

4. 收集可以提升的变量：

可以分为三步：

1. 先把所有“有机会被提升”的标量栈变量（标量栈就是通过alloca来产生的，所以找alloca）记下来
   • 循环全函数的基本块和指令，找 alloca。
   • 只有 dims.empty(),也就是维度为0的标量 alloca 才考虑（数组直接跳过）。
   • 用 ptr2var 记住以下映射：这个 alloca 产生的指针操作数（alloca->res）对应哪个寄存器号。这样后面看到某个指针就能反查它属于哪个变量。

具体来讲：eg.
Block0:
  %reg_3 = alloca i32　　; alloca 的结果操作数就是 %reg_3，本质是一块栈槽的指针
  store i32 0, ptr %reg_3　; 往这块槽里写 0

构建 ptr2var 时会记一条映射：ptr2var[%reg_3] = 3。
• “键”是 %reg_3 这个操作数（即 alloca 返回的指针值）。
• “值”是它的寄存器号 3。

2. 收集并筛选可提升变量 vars = collectPromotableVars(function, ptr2var)
   • 里面会对每个候选 alloca 做更细的检查：
   ◦ 记录数据类型、将默认值设置为0 或 0.0，这个是自己实现的，像是cpp的实现，防止 还没有store(赋值)就load（读值）的特殊情况可能出错 。
   ◦ 遍历所有指令，看看这个 alloca 是否只被合法的 load/store 使用；若指针被拿去做 GEP/当作实参/算术等“地址逃逸”，就标记 promotable=false。
   ◦ 收集所有对该 alloca 的 store 所在的基本块，放到 defBlocks，后面插 φ 要用。
   • 结果是一个映射：变量寄存器号 → VarInfo（包含 alloca 指针、类型、是否可提升、定义块集合、默认值）。
3. 处理“从未 store 过就被 load”的情况 markUninitializedUses(function, ptr2var, vars)
   • 再扫一遍指令，若发现某个候选变量从没被 store，却有 load，则把它标记为不可提升（这个是我们自己选择的，主要是为了避免把“读未初始化”的行为悄悄换成 SSA 寄存器读，导致结果不确定甚至错优化）。
   • 做完这步，如果 vars 里没有任何可提升的（全被筛掉），函数直接 return，后续 φ/重命名就不用跑了。

• 有了 defBlocks 后，要靠支配树和支配前沿决定 φ 的最小插入位置，并在重命名时 DFS 支配树维护“当前版本栈”。没有支配信息就没法放 φ、也没法正确重命名。

---

5. 获取支配信息（Dom Tree）

Analysis::AM.getAnalysis::DomInfo(function) 会构建/取出该函数的支配树和支配前沿数据。具体get可以在dominfo.cpp内查看 就是先拿到 CFG，new 一个 DomInfo，调用 build(*cfg) 填好支配树/前沿，然后放进缓存并返回

• 支配前沿用来决定哪些基本块要插入 φ；支配树则用来在重命名阶段做 DFS，把每条路径的“当前版本”按支配关系向下传递。

---

6. 插入 φ

6.1 先声明一个blockPhi，其是个索引表，记录“某块里针对某个变量的 φ 指令指针”，方便后续重命名阶段给这些 φ 补 incoming 值。
简单说：它是“块 × 变量 → φ 指针”的查找表，先存放占位 φ，再在重命名阶段用来精准填充每条边的输入。

6.2 跳转到insertPhiNodes函数内（）：
起点：把这个变量所有出现过 store 的基本块（defBlocks）塞进队列work。
• 循环：每次从work取出一个块 x，看它的支配前沿集合 frontier[x]，里面是“x 支配但不立即支配”的后继交汇点。
• 对于前沿里的每个块 y(y属于frontier[x])：
◦ 如果之前没在 y 给这个变量放过 φ（hasPhi判重），现在放一个：

1. 申请新寄存器，创建 φ（类型同变量）。

2. 把 φ 插在块 y 的指令列表最前面。
3. 记到 blockPhi[y][vreg]，后面重命名时用来填 incoming。
   ◦ 额外一步：如果 y 本身不是定义块（即原本没写这个变量），把 y 再入队，让它的支配前沿也被继续探索。这样能覆盖“φ 生成新的定义，再可能在更远的交汇点需要新的 φ”的情形.

循环结束后，对于这个变量，所有需要合流的基本块都已经插好 φ，并分配了结果寄存器。

6.3 例子：
想象一个最常见的菱形 CFG 例子，变量 a 在两条分支都被写过，需要在合流点选对的值：

```
entry:
  store 1 -> a
  br %cond, then, else

then:              else:
  store 2 -> a       store 3 -> a
  br join            br join

join:
  ... use a ...
```

• 先把所有写过 a 的块入队：defBlocks = {entry, then, else}。
• 取队首 entry，查支配前沿 frontier[entry] = {join}：意思是“entry 支配 join，但不是 join 的唯一前驱，说明这里有路径合流”。join 还没放过 φ，于是插一个 phi(a) 在 join 块头。
◦ hasPhi 记录：join 已经有了针对 a 的 φ。
◦ join 不是定义块，把 join 入队，继续传播。
• 取队首 then，它的支配前沿通常也是 {join}，但 join 已经有 φ 了，跳过。
• 取队首 else，同理跳过。
• 取队首 join（之前入队的），看它的支配前沿。如果还有更远的合流点，也会继续插；没有就结束。
这样 join 预先放好了 φ，后续重命名时会给这个 φ 补三路 incoming：
• 来自 entry 直连的路径（如果有），
• 来自 then 的值 2，
• 来自 else 的值 3。

φ 指令的“reg”只有一个结果寄存器（SSA 定义点），但它内部可以挂多条 incoming：每条是「来自哪条前驱边的值」。在例子里：
• 插入时只分配了一个新寄存器给 phi(a)，这是 φ 的结果寄存器。
• 后面重命名阶段，沿每条前驱边调用 phi->addIncoming(value, label)，把三条输入依次塞进这个 φ：
◦ 来自 entry 的（如果直连）；
◦ 来自 then 的值 2；
◦ 来自 else 的值 3。
所以“只插了一个 reg”指 φ 的定义寄存器只有一个；“三路 incoming”是同一个 φ 里存了三组 (值, 前驱) 对。

---

7. 重命名为 SSA

核心思路：沿支配树 DFS，每个变量用一个“值栈”追踪当前可见的版本。

理解：可提升变量的“最新值”始终放在栈顶，load 只是“取最新值”，store 只是“把新值放到栈顶”。既然我们自己在栈里维护了它们，原来的 load/store 就成了多余的内存指令，可以删掉。

具体规则（在重命名 DFS 过程中）：
• 进入一个块：先把这个块头上预插好的 φ（如果有）当作“新定义”压到该变量的栈顶。
• 遍历指令：
◦ 遇到 load ptr，且 ptr 属于可提升变量：

- 怎么判断?
- 先用 ptr 去 ptr2var 查出对应的变量编号（alloca 的寄存器号），再查 vars[varReg].promotable 看看是不是可以提升
  ■ 若可以提升，就再看这个变量的栈顶。如果栈非空，用栈顶的寄存器替换掉所有对这个 load 结果的使用；如果栈空，用默认 0/0.0。
  ■ 然后删掉这条 load，因为“取值”动作由栈直接提供了，内存不需要再读。

◦ 遇到 store val -> ptr，且 ptr 属于可提升变量：
■ 把 val 压到该变量的栈顶，表示“最新版本”。
■ 删掉这条 store，因为值已经在栈里维护，内存不需要写。
◦ 其他指令照旧。

• for循环遍历地去处理后继块（在当前基本块的 CFG 出边指向的块，就是控制流“下一步可能去的块”）的 φ incoming：
◦ 对每个后继块，如果后继针对某变量有 φ，占位在 blockPhi[succ][vreg]，就取当前栈顶（若空用默认值）作为“从当前块流向后继的值”，调用 phi->addIncoming(val, 当前块label)。

• 递归支配树孩子；支配树描述“谁支配谁”，domTree[blockId] 列出当前块在支配树上的子节点（被当前块直接支配的块）。DFS 这些孩子，表示沿着支配关系向下遍历整张函数图，确保每条路径都按顺序做同样的栈维护和替换。
• 返回时弹栈：压栈的目的是在“当前块及其支配的子树”这段路径上，让 load/φ incoming 能拿到正确的“最新值”。等这条路径处理完（返回父节点），弹栈只是恢复现场，让兄弟分支看到的仍是进入当前块前的版本，避免影响兄弟分支的处理。

为什么能删 load/store?
• 对“单个标量变量”的所有读写，我们用栈严密地维护了“当前值”版本；load 只是读，store 只是写，内存不再需要承担“存当前值”的责任，所以这些指令可安全删除。
• φ 的 incoming 就是把“沿着哪条前驱边来的当前值”接进 φ，保证合流处能选对版本。

---

8. 删除提升后的alloca:

删已提升的 alloca（入口块扫描）：在入口块(入口块就是函数的起点基本块：程序一进这个函数，第一条指令所在的块（通常是 Block0) 遍历指令，找到 alloca 且对应变量 vars[alloc->res->getRegNum()].promotable 为 true，就删掉这条指令。只删入口块是因为栈槽分配按约定都在入口；非入口若有 alloca 也不会被提升。

---

9. 第二次DCE:

再跑一次 DCE：重命名后 load/store/alloca 被删掉，原先为它们服务的中间算术、phi 可能变成“定义了寄存器但没人用、且无副作用”的死代码。例如初始化用的 add 0,0、未被引用的 phi 等。第二次调用 removeDeadDefs 把这些新死掉的 def/phi 清理掉

---

10. 失效分析缓存

我刚改动过这个函数的 IR，之前缓存的分析结果都不可信了”，所以调用 Analysis::AM.invalidate(function); 来把与该函数关联的分析（CFG、DomInfo 等）标记为失效

助教可能问到的实现的一点小问题（Little Bug）

为什么有的 alloca 不能删：

◦ 只要指针"逃逸"到你看不清的地方（传给 call、作为 GEP/base 继续做地址算术、参与条件判断、存进别的内存、返回出去），外部或别名可能读写这块栈内存。你如果把它提升成寄存器，就丢失了真实的内存读写，程序可能错。
◦ 所以我们采取的策略是：只有所有 use 都是本地的 load/store 且不逃逸，才能放心删除 alloca，并用寄存器和 ϕ 代替。逃逸时保守保留，保证语义不变。

没删alloca的场景：

指针逃逸/数组：

实际.ll:

```
Block0:
    %reg_59 = alloca [4 x [2 x i32]]
    %reg_44 = alloca [4 x [2 x i32]]
    %reg_27 = alloca [4 x [2 x i32]]
    %reg_10 = alloca [4 x [2 x i32]]
    %reg_1 = alloca [4 x [2 x i32]]
    %reg_2 = getelementptr [4 x [2 x i32]], ptr %reg_1, i32 0, i32 0, i32 0
    store i32 0, ptr %reg_2
    %reg_3 = getelementptr [4 x [2 x i32]], ptr %reg_1, i32 0, i32 0, i32 1
    store i32 0, ptr %reg_3
```

解释：
有 5 个数组 alloca，后续 大量 getelementptr 基于这些指针计算子元素地址，再配合 store/load 初始化和访问。alloca指针通过 GEP 派生出新指针流入后续指令，属于地址逃逸/聚合访问场景。mem2reg 不做 SROA，无法安全用寄存器替代整块数组，因此保留这些 alloca 是必要的。
◦ 佐证：看 getelementptr … ptr %reg_1 … 就能证明 %reg_1 作为基址被进一步派生使用，不能简单删掉。

未初始化读例子：

实际：

```
%reg_61 = alloca [7 x [1 x [5 x i32]]]
%reg_55 = alloca i32
%reg_30 = alloca [2 x [8 x i32]]
%reg_16 = alloca i32
%reg_4 = alloca i32
%reg_1 = alloca i32
%reg_2 = add i32 1, 0
%reg_3 = load i32, ptr %reg_1
store i32 %reg_2, ptr %reg_1
```

解释：
有多个标量 alloca。对 %reg_1 先 load 后才 store 初值 1（同理 %reg_4），这是"首 store 之前先读"模式。如果 mem2reg 直接提升成寄存器，会变成读未定义 SSA 值。当前策略是允许提升但给默认值 0；若改为严格，可选择检测到这种形态就不提升或报错。

◦ 佐证：%reg_3 = load i32, ptr %reg_1 在任意 store 前执行，说明存在潜在未初始化读。

副作用判定例子

```
call void @putint(i32 %reg_9)
%reg_10 = load i32, ptr %reg_4
call void @putint(i32 %reg_10)
%reg_11 = load i32, ptr %reg_1
call void @putint(i32 %reg_11)
%reg_12 = add i32 10, 0
call void @putch(i32 %reg_12)
br label %Block1
```

解释：
位置：多条 call void @putint/putch；这些是有外部可见副作用的指令。DCE 时不会删除这些调用，即便返回值未使用。
◦ 同时，在同文件 的 store 指令也被视为有副作用（写内存），不会被 DCE 当作死代码删除。
◦ 佐证：任何 STORE、CALL、BR、RET 都在 hasSideEffect 列表里；所以即便寄存器定义无人用，也不会删掉这些操作，保证语义不变

未初始化读默认补 0 的原因：

◦ IR 里可能出现“从未 store 就 load”的情况。直接提升成寄存器会变成“读未定义 SSA 值”，结果不确定。
◦ 给它补一个确定的默认常量（整型 0 / 浮点 0.0），让 SSA 有定义点，后续优化和执行都稳定。如果要严谨，可以选择：检测到这种情况就不提升，或直接报错。

ϕ 插入依赖 defBlocks/dom frontier 的原因：

◦ ϕ 要放在“不同定义汇合”的地方，最小化位置就是支配边界（dom frontier）。用 defBlocks 记录所有 store 所在块，再沿支配边界迭代放 ϕ，确保需要的汇合处都有 ϕ，又不会满世界乱插，保持数目最少且覆盖正确。

副作用只按 Operator 枚举的原因：

◦ DCE/替换时必须保留有副作用的指令（写内存、改控制流、外部可见）。已知的副作用操作都在 Operator 枚举里（STORE/BR/RET/CALL 等），只对“确认无副作用”的算术/phi/load 做删除或替换。这样简单且安全。若以后加新指令，要记得更新这个枚举，否则可能误删或漏删。

你项目里优化顺序是什么（答辩先说这段最稳）

- 触发条件：`optimizeLevel > 0` 才跑优化（也就是你看到的 `-O1`）。流水线在 main.cpp。
- 优化顺序：`mem2reg -> SCCP -> CSE`
- Pass 顺序（非常关键）：
    1. UnifyReturn：先把函数里多个 `return` 归一成“一个出口块”，让 CFG 更规整（方便做支配/DF 这类分析）。
    2. mem2reg：把局部变量从“栈槽（alloca/load/store）”提升到 SSA（phi + 寄存器值流）。
    3. SCCP：在 SSA 上做“常量传播 + 条件剪枝 + 删不可达块”。
    4. CSE：块内把重复的纯计算提取复用（你的实现是 block-local）。

- 为什么这个顺序合理（大白话）：
  - mem2reg 先把值流变干净（SSA），SCCP 才能“顺着 def-use 推常量”推得动；
  - SCCP 删掉恒假分支和不可达块后，CSE 在更短的块里更容易命中；
  - UnifyReturn 在最前，减少 CFG 的“多出口边界情况”。
  - mem2reg/complex_phi：在 test_output/complex_phi.ll#L25-L147 可见循环入口用多路 `phi` 同时合并计数器和累加值，且全函数没有 `alloca`。若未插 `phi`，同一变量不同路径的值无法用寄存器表示，只能退回 `load` / `store`，但现在全部寄存器化，说明复杂支配关系下的 `phi` 插入正确。

SCCP（大白话与证据）

- 为什么这样做：提前“算死”能确定的东西，减少计算；把永远走不到的块删掉，让后续优化和生成代码更干净。

结合你自己的 SCCP 实现：老师问“你怎么做的”你可以按这 3 句回答

- 你用的是经典 3 值格（UNDEF/CONST/OVERDEF）：定义在 middleend/pass/sccp.cpp。
  - 大白话：
    - UNDEF：现在还不知道值是什么
    - CONST：已经确定就是某个常量
    - OVERDEF：不确定/可能是很多值（比如来自输入、来自内存 load、来自调用返回等）
- 你怎么把多条路径的信息合起来：用 `join`（格的“合并”）在 middleend/pass/sccp.cpp。
  - 大白话：两条路都是同一个常量 → 还是那个常量；只要冲突/类型不一致 → 变 OVERDEF。
- 你不仅推常量，还做“条件可达性”：
  - 你用 `executable[blockId]` 标记基本块是否可达，初始只把入口块置为 true（见 middleend/pass/sccp.cpp）。
  - 遇到 `br i1 cond`：如果 `cond` 是常量，就只把真边或假边标为可达；否则两边都可达（见 middleend/pass/sccp.cpp）。
  - `phi` 只会合并“可达前驱”的 incoming（见 middleend/pass/sccp.cpp），这是 SCCP 里“Conditional”的灵魂。

功能样例（Basic）：把“能算死的 if”直接变成直达输出

- 源码：见 testcase/functional/Basic/21_short_circuit3.sy
- O0（未优化）：你能在 IR 里看到一大串算术把条件算出来，再 `br i1` 分支，最后在真分支里 `putch(65/66/67...)`
  - 证据片段：见 test_output/21_short_circuit3-O0.ll
  - 典型结构：`icmp ...` → `br i1 %cond, label %T, label %F` → `T` 里 `call @putch`。
- O1（优化后）：这些 if 直接被“折叠”为顺序代码（该输出就输出，不该输出的块直接没了）
  - 证据片段：见 test_output/21_short_circuit3-O1.ll
  - 你可以指给老师看：这里已经直接 `call void @putch(i32 65)`，而不是先算条件再跳。
- 术语小释义：
  - “剪枝”= 把恒假分支删掉或不再传播；“不可达块”= 从入口永远到不了的基本块。

1. 变量被直接替换成常量；2) 条件恒真/恒假后，对应分支块会消失或被绕过；3) IR 里只剩必要的算术链。

结合你自己的 CSE 实现：一句话定位“你做的是哪种 CSE”

- 你的 `CSEPass` 是“块内 CSE”：只在同一个基本块里做复用。
  - 证据：你在 middleend/pass/cse.cpp 对每个 block 建 `expr2reg`，不会跨块携带。
- 你用的表达式 key：`opcode + (type/cond) + operand1 + operand2`，并且对可交换运算做了排序归一。
  - 证据：`isCommutative` 在 middleend/pass/cse.cpp，`buildKey` 在 middleend/pass/cse.cpp。
  - 大白话：`a+b` 和 `b+a` 算同一个 key，所以能命中。
- 你遇到“可能改变值流的指令”会清空表（保守不跨越副作用）：
  - `STORE/CALL/BR/RET` 会触发 `expr2reg.clear()`，见 middleend/pass/cse.cpp。
  - 大白话：只要中间出现了“可能把内存/控制流搞复杂”的东西，就别复用之前的表达式，避免错优化。
- 你的结果如何体现：
  - 源码（testcase/optimize/sccp/sccp2.sy#L1-L34）有多组 if，若没优化，会看到所有条件与 g/3 的除法保留在 IR。
  - 优化后在 test_output/sccp2.ll#L20-L60 只剩常量链：`phi` 直接取常量 9961，再连续 `add` 3253、9760、14、334，最后输出；无条件分支、无除法，证明常量传播+死块删除生效。
  - sccp1 虽长，但执行正确且初始化/边界值直接以 SSA 常量流转，说明常量折叠未破坏语义。

标量 CSE（大白话与证据）

- 它是什么：Common Subexpression Elimination，公共子表达式消除。大白话：同样的纯计算只算一次，后面都用它的结果。
- 为什么这样做：省指令、少重复算；也便于后续寄存器分配、调度。
- 术语小释义：
  - “公共子表达式”= 代码里值完全相同的纯计算（无副作用），重复出现即可提取。
- 优化完成的直观特征：
  1. 重复计算被提取为单一 SSA 值；2) 派生值（如 x+y）只算一次、多处复用；3) 大表达式不再被反复展开。
- 你的结果如何体现：
  - 源码（testcase/optimize/scalar_cse/cse1.sy#L1-L123）在循环里几十次重复 `(i1+...+i15)`；若无 CSE，IR 会每次重建 15 项加法树。
  - 优化后在 test_output/cse1.ll#L31-L140 先形成一次 `%reg_69`（15 参数之和），后续所有累加都复用 `%reg_69`，未再展开 15 项，这是直接证据。
  - `main` 中派生值 `x+y`、`x+z`、`y+z`、`2*y`、`z/3`、`z*2` 各自只算一次并被传参复用（test_output/cse1.ll#L153-L167）。未优化时这些会反复计算。

答辩速讲要点（按优化分类）

- mem2reg：

1. 目标：消除局部 `alloca`，用 SSA+`phi` 维持一致性；减少内存流量。
2. 机制：建支配树/支配前沿，插 `phi`，再重写 `load/store` 为直接 SSA def-use。
3. 证据：noloadstore/samebb/complex_phi 全函数无 `alloca`，分支汇合处有 `phi`（见上方行号链接）。未优化应当能看到 `alloca`+`store`+`load` 的内存往返。

- SCCP：
  1. 目标：在 SSA 上推常量+剪枝，删掉永不可达的块。
  2. 机制：工作队列驱动，格值（常量/Top/Bottom）传播，遇到恒真恒假条件直接折叠。
  3. 证据：sccp2 产物只剩常量链、无分支（test_output/sccp2.ll#L20-L60），对照源码 if 嵌套（testcase/optimize/sccp/sccp2.sy#L1-L34）。
- CSE：
  1. 目标：把重复的纯计算提出来共用，省掉重复指令。
  2. 机制：基于表达式 key (opcode+操作数) 查表/哈希，命中即复用已有 SSA 名；需保证无副作用、操作数等价。
  3. 证据：cse1 的巨型和式被凝结为一次 `%reg_69` 后反复复用（test_output/cse1.ll#L31-L140），`main` 的派生值也各算一次（test_output/cse1.ll#L153-L167）。

如果被问“没优化时会怎样”可以这样回答

- mem2reg：IR 会看到每个局部的 `alloca`，每次用前 `load`、写后 `store`，分支归并不会有 `phi`。
- SCCP：所有 if 条件和分支都保留，g/3 的除法会一直存在；不可达块也还在，代码更长。
- CSE：大和式与派生值会在每次出现时重算一次，IR 中会出现大量重复的同形加法链或乘法链。

结论

- mem2reg：消除局部栈槽、插入 `phi`；证据见 noloadstore/samebb/complex_phi 的“无 allocas + 正确 phi”。
- SCCP：常量折叠+死块删除；证据见 sccp2 常量链 IR（分支全消）。
- CSE：公共子表达式提取复用；证据见 cse1 中 `%reg_69` 与单次派生值计算。

答辩问题：

怎么调用的优化?

main.cpp 有一个If语句判断输入：终端输入优化等级只要>0 就启动优化

优化顺序？为什么？好处?

自己做的选择了 SCCP CSE
一个很自然的顺序：mem2reg -> SCCP -> CSE
好处：
前提：这些优化都是编译器对**中间表示（IR）**（所有优化都在IR层做）的优化。

1. mem2reg（Memory to Register Promotion，内存转寄存器提升）

- 核心术语解释：
  - 栈内存：程序运行时，局部变量/临时变量默认存在栈内存中，CPU访问栈内存-->速度慢；
  - 寄存器：CPU内置的高速存储单元，访问速度比内存快；
  - 提升（Promotion）：把内存中的变量“迁移”到寄存器，并消除不必要的内存读写操作,提升速度。
- 大白话做啥：
  编译器扫描IR，找出满足两个条件的临时变量/局部变量：① 只在一个**基本块**（解释：一段没有分支、能从头执行到尾的代码段）里被赋值一次；② 赋值后只读取、不修改。
  把这些变量从栈内存中移除，直接用寄存器存储和操作它们——比如原本要先把值写到内存的tmp变量，再读内存的tmp计算，优化后直接把值放寄存器，跳过“写内存、读内存”这两步慢操作。
  核心目标：减少慢的内存访问，用快的寄存器替代。

2. SCCP（Sparse Conditional Constant Propagation，稀疏条件常量传播）

- 核心术语解释：
  - 常量传播：如果一个变量的值能100%确定是固定常量（比如a=5且后续无修改），就把所有用到a的地方直接替换成5，无需访问变量；
  - 条件：考虑分支、循环等条件语句的影响（比如if(0)里的代码永远执行不到，里面的变量赋值可忽略）；
  - 稀疏：只针对“能确定是常量”的变量做传播，不做无依据的猜测；
  - 死代码消除（DCE）：SCCP的附带效果，删掉“永远不会执行的代码”。
- 大白话做啥：
  编译器顺着程序所有可能的执行路径遍历IR，做两件事：
  ① 找出“不管走哪条路径，值都固定不变”的变量，把这些变量的所有引用处直接替换成常量；
  ② 找出“永远不会被执行的代码（死代码）”（比如if(1>2)的分支、赋值后永远用不到的变量赋值），直接删掉。
  比如变量a被赋值为10且后续无修改，所有用a的地方都换成10；if(0)里的代码直接删，不用执行。
  核心目标：用常量替换变量减少计算/访问，删除无用代码减少执行量。

3. CSE（Common Subexpression Elimination，公共子表达式消除）

- 核心术语解释：
  - 公共子表达式：程序不同位置出现的“完全相同的表达式”，且参与计算的变量值在两次计算之间没有被修改（比如x=a+b、y=a+b，a和b在两次计算间没改，a+b就是公共子表达式）；
  - 消除：只计算一次该表达式，把结果存到寄存器，后续复用结果，不用重复计算。
- 大白话做啥：
  编译器扫描IR，找出满足条件的重复表达式，只执行一次计算并缓存结果，后续再用到时直接用缓存值。
  比如先算(a+b)*3，又算(a+b)/2，只要a、b在两次计算间没被修改，就先算一次a+b存寄存器，再用这个结果乘3、除以2，避免重复算a+b。
  核心目标：减少重复计算，节省CPU算力。

二、为什么顺序是 mem2reg → SCCP → CSE？（核心：前一步为后一步扫清障碍，避免无用功/误判）

1. 先做mem2reg：给后续优化打基础，避免误判

SCCP和CSE都依赖“变量的值是否稳定、可预测”：

- 内存中的变量存在**别名**（解释：多个指针/引用指向同一块内存，编译器没法确定某段代码是否修改了这个内存值），导致编译器不敢确定变量值是否不变；
- 把变量挪到寄存器后，寄存器无别名（一个寄存器只对应一个变量），编译器能明确知道变量的赋值/修改情况——SCCP能精准判断哪些变量是常量，CSE能精准判断哪些是公共子表达式。

2. 再做SCCP：精简代码，避免CSE做无用功

- 如果先做CSE再做SCCP，CSE会给“后续会被删掉的死代码”做表达式消除，纯浪费编译器算力；
- SCCP替换常量后，原本的变量表达式会变成常量表达式（比如a+b变成10+20），此时CSE再优化，能直接针对“简化后的常量表达式”做消除，而不是优化“未简化的变量表达式”（比如先把a+b换成30，就不用再复用a+b，直接用30即可）。

3. 最后做CSE：精准消除，不遗漏/不误判

经过mem2reg（变量在寄存器，值透明）和SCCP（常量替换、死代码删除）后，剩下的IR是“干净且值可预测的”：

- CSE能精准找到所有真正的公共子表达式，不会漏判（比如寄存器变量无修改，能确定表达式值不变）；
- 不会误判（比如不会把“看似相同但内存值可能变的表达式”当成公共的）；
- 还能覆盖SCCP替换常量后产生的新公共子表达式（比如a+b变成30+b，两次30+b可复用）。

三、这个顺序的核心好处

1. 优化效果最大化：每一步都为下一步消除障碍，最终减少的内存访问、计算量，比乱序优化多得多；
2. 避免无用功：先删死代码、替换常量，CSE只处理“有价值的表达式”，节省编译器优化时间；
3. 减少优化错误：寄存器无别名，避免SCCP误判常量、CSE误判公共子表达式。