Code Generation

代码生成器的作用是根据中间表示生成机器代码，根据实现的不同在其之前可能会有一个机器无关的优化组件。代码生成器的主要任务有

- 指令选择：选择适当的指令实现 IR 语句
- 寄存器分配和指派：把哪个值放在哪个寄存器
- 指令排序：按照什么顺序执行指令

目标语言

目标语言是一个三地址语言，其指令包括

- 加载指令
- 保存指令
- 运算指令
- 无条件跳转指令
- 条件跳转指令

指令的代价为 1 加上其运算分量寻址的代价，寄存器寻址代价为 0，涉及内存的代价为 1

目标代码中的地址

即如何将 IR 中的名字转换为目标代码中的地址

根据之前的讨论，逻辑地址空间被划分为四个部分，静态的代码区和数据区，动态的堆区和栈区

对于为过程调用和返回生成代码，有两种策略：静态分配和栈式分配

静态分配

静态分配的思想是根据符号表确定每个活动记录的大小和布局，为其静态的分配一个区域

返回地址存放在过程的活动记录中

静态分配不支持递归调用

栈式分配

栈式分配的思想是保存活动记录时使用相对地址。由于运行时才能得知活动记录的位置，故可以将这个位置存放在寄存器中，然后采用相对于开始位置的偏移量来访问

名字的运行时刻地址

IR 中使用名字来引用变量。这种策略使得编译器在移植到使用不同 run-time 组织方式的机器上时不用修改前端

最终名字都将被替换为访问存储位置的代码

基本块和流图

可以用图的形式来表示中间代码

- 将中间代码划分成**基本块**（basic block），基本块是满足以下条件的最大连续三地址指令序列
  - 控制流只能从基本块的第一个指令进入基本块，没有跳转到基本块中间的代码
  - 除了基本块最后一条指令，控制流离开基本块之前不会停机或跳转
- 基本块构成**流图**（flow graph）的节点，流图的边指明了哪些基本块会紧随一个基本块运行

划分基本块

按照以下算法可以确定中间代码的首指令（leader）

- 中间代码的第一条指令是首指令
- 任意转移指令的目标是首指令
- 紧跟在转移指令之后的指令是首指令

则每个首指令对应一个基本块，其包含的指令是从首指令到下一条首指令之前的三地址指令

后续使用信息

后续使用信息可以用于优化寄存器指派，一个变量的值如果在之后都不会被用到，则存放该变量值的寄存器可以分配给其他变量使用

变量的**使用**（use）定义为：假设语句 i 对 x 赋值，且语句 j 的运算分量为 x，如果从 i 开始可以沿一条不对 x 赋值的路径到达 j，则称 j 使用了语句 i 计算的 x 的值，可以说 x 在 i 处**活跃**（live）

可以反向扫描基本块，得到每个变量的使用信息。因为过程调用可能有副作用，为方便起见，假设每个过程调用是一个新基本块的开始

初始假设所有变量都是活跃的，从基本块结束开始反向扫描，对于每条语句 $i: x = y + z$

- 将 x, y, z 的活跃信息和语句 i 关联起来
- 设置 x 为不活跃，无后续使用
- 设置 y, z 为活跃，下次使用位置设为 i

需要注意的是第二步和第三步顺序不能颠倒，因为要正确处理形如 $x = x + 1$ 的语句

流图

流图的顶点是基本块，两个基本块 B, C 之间有边 $\iff$ 基本块 C 的第一条指令有可能在 B 的最后一条指令后执行（包括跳转和程序自然地流动）

在流图中可以添加入口和出口节点，入口节点到第一条指令所在基本块有一条边，而任何可能最后执行的基本块到出口节点有一条边

循环是程序大部分的运行内容，是流图中识别的重点。一个循环 L 是流图中节点的集合，满足

- L 中有一个被称为**循环入口**（loop entry）的节点，其是唯一的前驱可能在 L 之外的节点，即从整个流图的入口到达 L 中任意节点都需要经过循环入口，且其不是整个流图的入口节点本身
- L 中每个节点都有到 L 循环入口的非空路径，且该路径全部在 L 中

基本块的优化

针对基本块局部的优化可以有很好的效果

基本块的 DAG

基本块可以用 DAG 表示（类似表达式的 DAG）

- 每个变量有对应的节点，代表初始值
- 每个语句有对应的节点，代表运算结果，其节点与一组变量关联，表示节点对应的语句是基本块中对这些变量最晚定值的语句
- 有输出节点，节点对应的变量在基本块出口处活跃

根据 DAG 可以知道各个变量最终的值和初始值的关系

基本块的构造可以通过顺序扫描三地址语句实现

- 初始时为基本块中各变量建立节点，将变量和节点关联
- 顺序扫描三地址指令

    - 如果指令为 $x = y \text{ op } z$，则建立标号为 op 的节点，其子节点为 y, z 当前关联的节点，然后令 x 与该节点关联
    - 如果指令为 $x = y$，不建立节点，如果 y 关联到节点 N，则同样令 x 关联 N
- 扫描结束后，对于所有在出口活跃的变量 x，将其关联的节点设为输出节点

基本块优化

根据基本块的 DAG，可以进行更多优化，如

- 消除局部公共子表达式：检查局部公共子表达式的方法是在建立某个节点之前，检查是否存在一个与其有相同运算符和子节点的节点
- 消除死代码：消除没有附加活跃变量的**根节点**
- 对相互独立的语句重新排序，降低一个临时值需要保存的时间
- 使用代数规则重排三地址指令的运算分量顺序：如利用代数恒等式将计算步骤消除或是将计算强度消减（如用 $x \times x$ 代替 x^2）

数组引用

数组的引用不能简单地当成一般运算处理

对一个数组元素取值的运算，如 `x = a[i]` 用一个运算符 `=[]` 表示，左右子节点为数组初值和下标，x 与该节点关联

对数组的赋值，如 `a[j] = y` 用一个运算符 `[]=` 表示，三个子节点分别为数组初值，下标 j 和变量 y，没有变量与这个节点关联，且其创建会“杀死”所有已建立的依赖于数组 a 的节点。

一个被杀死的节点不能再获得标号，不能作为公共子表达式

指针赋值/过程调用

在没有额外信息的情况下，语句 `x = *p` 可能使用了任意变量，故影响了对死代码的消除，而语句 `*q = y` 可以对任意变量赋值，杀死了其他所有的节点

通过全局/局部的指针分析可以解决部分问题，如指令序列 `p = &x; *p = y` 只需要杀死关联了 x 的节点

过程调用是类似的思想，在没有额外信息的情况下，必须安全地假设其使用了所有可访问范围内的变量，同时修改了所有可访问范围内的变量

从 DAG 到基本块

可以从 DAG 重构基本块

- 为每个节点构造一个三地址语句，计算其对应的值

- 将值赋给会在出口处活跃的变量，如果没有额外信息，需要假设所有变量都在出口处活跃
- 如果有多个活跃变量与节点关联，需要添加复制语句

重构基本块时指令的**顺序**需要遵循

- 指令顺序和 DAG 中一致，只有在计算出一个节点的子节点的值后才能计算该节点的值
- 对数组的赋值必须在原基本块中其之前的数组赋值和取值之后
- 对数组的取值必须在原基本块中其之前的数组赋值后，数组赋值之间的求值可以交换顺序
- 变量的使用必须在原基本块中其之前的过程调用和指针赋值之后
- 过程调用和指针赋值必须在原基本块中其之前的变量求值之后，即重构时指令不能越过过程调用和指针赋值

代码生成器

生成代码的主要问题是如何最大限度使用寄存器，减少不必要的 load/store 指令。寄存器的主要使用方法有

- 执行运算时部分或全部的运算分量需要存放在寄存器中
- 存储临时变量，如大表达式中的子表达式的值
- 存储在一个基本块中计算而在另一个基本块中使用的全局值，如循环时的下标
- 进行运行时管理，如存放栈指针的寄存器

寄存器和地址描述符

需要数据结构来描述变量的值的位置和寄存器中存储的值

每个可用的寄存器有一个**寄存器描述符**（register descriptor），跟踪有哪些变量的值存放在寄存器中。由于仅考虑一个基本块内的情况，可以假设开始时所有寄存器描述符为空

每个程序变量有一个**地址描述符**（address descriptor），跟踪在哪些位置上可以找到该变量的值，这些位置可以是寄存器，内存位置或是栈中的位置

代码生成算法

代码生成中最重要的部分就是为三地址指令选择寄存器，引入一个函数 $getReg(I)$ 为三地址代码 I 选择寄存器

对于运算语句，三地址指令形如 `x = y + z` ，生成代码的步骤为

- 使用 $getReg$ 为 x, y, z 选择寄存器，记为 R_x, R_y, R_z

- 根据 R_y 的寄存器描述符，如果 y 的值不在 R_y 中，则生成指令 $\mathrm{LD}\ R_y, y'$ ，其中 y' 是 y 的内存位置之一，可根据其地址描述符得到。对于 z 同理
- 生成指令 $\mathrm{ADD}\ R_x, R_y, R_z$

对于复制语句，三地址指令形如 `x = y` ，则我们总是假设 $getReg$ 为 x 和 y 选择同一个寄存器，若 y 不在 R_y 中，生成指令 $\mathrm{LD}\ R_y, y'$ 。如果 y 已在 R_y 中，则不需要做任何事情，只需修改其寄存器描述符，表示其中也存储了 x 的值

在基本块结束时，如果某个变量在出口处活跃，且根据其地址描述符，其值不在其内存位置，则需要生成语句 $\mathrm{ST}\ x, R$ ，其中 R 是存放 x 值的寄存器

在生成指令时，需要同时修改描述符，其规则如下

- 对于指令 $\mathrm{LD}\ R, x$
  - 修改寄存器 R 的描述符，使其只含 x
  - 修改 x 的地址描述符，加入 R
  - 从 x 以外的地址描述符删去 R
- 对于指令 $\mathrm{ST}\ x, R$ ，修改 x 的地址描述符，使之包含 x 的内存位置
- 对于形如 $\mathrm{ADD}\ R_x, R_y, R_z$ 的语句
  - 修改寄存器 R_x 的描述符，使其只含 x
  - 修改 x 的地址描述符使其只含 R_x
  - 从 x 以外的地址描述符删去 R_x
- 对于复制语句 `x = y` ，如果生成了语句 $\mathrm{LD}\ R_y, y$ ，按照 LD 的规则处理，除此之外
  - 将 x 加入 R_y 的寄存器描述符
  - 修改 x 的地址描述符使其只含 R_y

getReg 函数的设计

考虑为三地址代码 `x = y + z` 选择寄存器，考虑为运算分量 y 选择寄存器（z 的情况类似）

- 如果 y 已经在某个寄存器 R_y 中，则选择这个寄存器，不用生成额外的加载指令
- 如果 y 不在寄存器中，但是存在空闲寄存器，则选择该寄存器，将 y 加载进去
- 如果 y 不在寄存器中，且没有空闲寄存器，则必须选择一个可行的寄存器，设该寄存器为 R ，其中的变量为 v
  - 如果 v 的地址描述符表示 v 还保存在 R 以外的地方，则可以使用 R
  - 如果 v 是 x，且 v 不是该语句的另一个分量，则可以使用 R，因为 v 的值在这之后不会被用到
  - 如果 v 在之后不会被使用，则可以使用 R
  - 如果上述条件都不满足，则需要生成存储语句 $\mathrm{ST}\ v, R$ ，该操作称为溢出（spill），如果 R 中存放了多个变量，则需要为所有变量生成 ST 语句，此

时应当选择溢出代价最小的语句

为运算结果 x 选择寄存器的策略基本相同，不同之处在于

- 只包含 x 的值的寄存器总是可选的，因为 x 的新的值正在计算
- 对于运算分量，如果 y 在之后不再使用，且 R_y 中只有 y 的值，则 R_y 也是可选的

对于复制语句 `x = y`，总是选择 R_y 然后令 $R_x = R_y$

窥孔优化

窥孔优化（peephole optimization）是一种局部改进目标代码的方法，在优化时检查目标代码的一个滑动窗口，然后用更高效的指令序列代替原本的指令

窥孔优化可以做到

- 冗余指令消除

  如果有类似

  ```
  LD R0, a
  ST a, R0
  ```

  的语句，可以删去其中的 ST 指令。为了保证这样的优化是安全的，必须保证两条指令都在一个基本块内
- 不可达代码消除：一个紧跟在无条件跳转后的不带标号的指令可以删除，这样的代码在控制流中是不可达的
- 控制流优化
- 代数化简和强度削减：如利用代数恒等式或是将幂运算转换为乘法运算等
- 使用机器特有指令：如对于加一减一的操作使用特有的 INC, DEC

寄存器分配和指派

寄存器的有效使用对于生成优质的代码是很有帮助的

- 寄存器分配：哪些值应当存放在寄存器
- 寄存器指派：各个值应当存放在哪个寄存器

全局寄存器分配

之前的代码生成策略都是在基本块的运行期间使用寄存器保存值，但是在基本块结尾将所有变量的值写回内存，但是对于在全局频繁使用的变量，可以将寄存器指派给这些变量，并使得这些寄存器在不同的基本块的指派保证一致

可以使用计数的方法估计将一个变量存放到寄存器带来的好处

使用计数

在循环中把一个变量保存在寄存器可以减少加载的次数，如果 x 被分配在寄存器中，每一次对 x 的引用都可以节省一次加载的开销

然而如果 x 在基本块中先被计算再被使用，有很大的可能 x 本来就在寄存器中，故只有 x 在基本块中没有被先行赋值，并且在同一基本块中被使用，才认为这次使用节省一次加载的开销

且如果不用在基本块结尾将 x 保存到内存，可以节省两个单位的开销（一次保存，一次加载），这要求 x 在基本块中被赋值且在出口处活跃

则在循环 L 中把寄存器分配给 x 节省的开销可以表示为

$$\sum_{B\in L} use(x,B)+2\times live(x,B)$$

其中 $use(x,B)$ 是 x 在基本块 B 中定值前被使用的次数，如果 x 在 B 出口处活跃且在 B 中被赋值，则 $live(x,B)$ 为 1，否则为 0

树重写指令选择

在某些指令集上，同一个三地址指令可以使用多种机器指令实现，而多个三地址指令可以用一个机器指令实现，此时需要选择适当的机器指令

可以用树来表示中间代码，按照规则覆盖这颗树生成机器指令