简单编译器

为了将程序的语法树转化为汇编代码主要需要经过这样一些环节。以处理 while+db 语言为例：

- 拆分复合表达式

  由于汇编程序不能处理复合表达式，因此需要将复合表达式拆分成为一系列单步计算。在拆分过程中，需要引入额外的辅助变量。

- 生成基本块

  汇编程序的结构不是树形结构，是一个汇编指令的列表，其中包含少量的跳转语句。因此需要将 AST 中的树形结构转化为这样以列表为主的结构。后面我们会详细介绍什么是基本块，什么是控制流图。

- 分配寄存器

  汇编程序中没有变量，只有寄存器和内存。因此需要用寄存器或内存来存储程序变量（含先前阶段生成的辅助变量）。出于运行效率的考虑，我们应当尽量用寄存器来分配。

- 生成汇编代码

  最后生成汇编代码是还可能补充实现一些细节，例如基本块的排布等等。

1 拆分复合表达式

- 关键点 1：每个赋值语句至多进行一步计算
- 关键点 2：转化后的程序仅在必要处保留树结构

  换言之：顺序执行的语句可以用链表存储。之所以在语法分析时使用树结构而此处不再使用树结构主要有两点原因：1. 没有必要，后续生成基本块时，每个基本块内的程序语句都是线性排列的；2. 线性表的操作更方便编程。

- 关键点 3：要妥善处理 while 循环条件的计算语句

  在 AST 中，while 循环条件是一个表达式。但是，经过拆分后，这个表达式的计算将会变为一段程序语句以及一个表达式。例如：

  拆分前

  ```
  x + y + z < 10
  ```

  拆分后

  ```
  #0 = x + y;
  #1 = #0 + z
  ```

  ```
  #1 < 10
  ```

请注意：这对于 if 语句而言不是一个问题

If 语句的处理方法

拆分前

```
if (old_condition)
then { ... }
else { ... }
```

拆分后

```
...
(some computation)
...
if (new_condition)
then { ... }
else { ... }
```

While 语句的处理方法

拆分前

```
while (old_condition) do {
  ...
  (old loop body)
  ...
}
```

拆分后

```
LABEL_1:
  ...
  (some computation)
  ...
  if (! new_condition) then jmp LABEL_2
  ...
  (new loop body)
  ...
  jmp LABEL1
LABEL_2:
  ...
```

- 关键点 4：要妥善处理短路求值

  由于短路求值的存在，and 与 or 必须转化为 if 语句。例如：

  拆分前

```
if (p && * p != 0)
then { ... }
else { ... }
```

  拆分后

```
if (p)
then { #1 = p }
else { #2 = * p;
       #1 = (#2 != p) };
if (#1)
then { ... }
else { ... }
```

2 生成基本块

3 寄存器分配

3.1 活性分析

- 计算方法：

$$in(u) = (out(u) \backslash def(u)) \cup use(u)$$

$$out(u) = \bigcup_{u \to v} in(v)$$

- 实现方法：稠密时使用 bitvector，稀疏时使用链表
- 控制流图的流向与其反方向是不对称的。例如，在下面程序中，`x` 有一个 def 两个 use，`x` 从第一个 def 到最后一个 use 为止始终是 live 的：

```
x = y;
z = x + 1;
u = x * x - 1
```

相反在下面程序中，`x` 有两个 def 一个 use，`x` 从第一个 def 到第二个 def 之间并不 live：

```
x = y;
x = z - 1;
u = x * x - 1
```

- 最终计算得到的 in 与 out 集合是上方等式对应的 Kleen 不动点。
- 实际计算 in 与 out 集合时可以使用迭代更新的算法。不难发现，假设要求的是函数 F 的 Kleen 不动点，并且 n 次迭代后的结果是 X 满足

$$\bot \leqslant X \leqslant F^{(n)}(\bot)$$

那么，

$$\text{lub}(\bot, F(\bot), F^{(2)}(\bot)\dots) \leqslant \text{lub}(X, F(X), F^{(2)}(X)\dots) \leqslant \text{lub}(F^{(n)}(\bot), F^{(n+1)}(\bot), F^{(n+2)}(\bot)\dots)$$

因此 $\text{lub}(X, F(X), F^{(2)}(X)\dots)$ 就是 F 的 Kleen 不动点。

- 生成 interference graph：如果存在一个 u 使得 $x, y \in in(u)$，那么就在 x 与 y 之间连一条边，表示它们不能分配同一个寄存器。

3.2 简单寄存器分配算法

本课程只考虑最简单的寄存器分配，即每个变量对应一个寄存器，多个变量可能对应相同的寄存器。较为前沿高效的寄存器分配算法，有时会对同一寄存器的不同使用位置分配不同的寄存器。

- 假设共有 K 个寄存器；
- 步骤一，Simplify：在 interference graph 中删除度数 $K-1$ 的节点，删除的节点用栈记录；
- 步骤二，Spill：如果无法 Simplify，就任意删除一个节点，再回到前面步骤一；
- 步骤三，Select：按照删除节点的倒序为所有变量分配寄存器；

  注：Simplify 删除的节点能够保证被分配到与 interference graph 上相邻节点不冲突的寄存器，Spill 变量可能可以分配到寄存器（假 Spill），也可能无法分配到寄存器（真 Spill），此时应当存储在内存中；
- 步骤四，Start over：如果有至少一个节点无法分配到寄存器，则改写相关代码并重做 liveness 分析与上述所有步骤。
- 如果 3 号变量无法分配寄存器，并且存储在 `%rbp - 16` 地址，那么：

  ```
  #2 = #3 + 1
  ```

  ```
  #3 = * #4
  ```

  分别会被改写为：

  ```
  #3 = * (%rbp - 16)
  #2 = #3 + 1
  ```

  ```
  #3 = * #4
  * (%rbp - 16) = #3
  ```

  这样，两次使用 `#3` 变量之间的代码 `#3` 都不再是 live 的了。
- 如果所有变量都分配到了寄存器，那么寄存器分配过程结束；如果有至少一个节点无法分配到寄存器，那么在执行万上述程序改写后，变量的 live 区域会大大缩小，此时应当重新进行 liveness 分析，并回到步骤一，从头开始重新分配寄存器，这个过程称为 Start over。往往经过至多 2-3 次 Start over 之后，寄存器分配算法就能顺利结束。

3.3 寄存器分配算法的优化

- 改进思路：如果有 move 指令（形如 `x = y` 的指令）且两个变量的 live 区域不重叠，那么就可以合并这两个变量，并删除 move 指令，这个过程成为合并（Coalesce）；
- 由于合并操作可能导致 interference graph 变稠密，从而导致原本可以不需要 Spill 的情形变得需要 Spill，得不偿失，因此一般只采用保守合并策略:；

  保守合并策略

  - 如果 `x` 与 `y` 两个变量的所有邻居中度数 $\geqslant K$ 的数量 $\leqslant K-1$，那么可以合并 `x` 与 `y` ；

  - 如果 `x` 的每个邻居要么也是 `y` 的邻居，要么度数 $\leqslant K-1$，那么可以合并 `x` 与 `y`；

- 步骤一，Simplify：只删除 move 无关的度数 $\leqslant K-1$ 的节点；
- 步骤二，Coalesce，进行保守合并；
- 步骤三，Freeze：如果没有可行的 Simplify 和 Coalesce 操作可以继续进行，那么就选择一条 move 指令，放弃对他进行合并，之后回到步骤一与步骤二；
- 上述三类操作都无法进行时则采用 Spill，最后 Select 与 Start Over 的流程与前述简化版本的算法一致。