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

  看似前面三个环节已经完成了全部的从 AST 到汇编代码转化的工作，那么为什么还需要这个额外的生成汇编代码环节呢？因为有些先前步骤中缺乏准确的用于生成汇编代码的信息。例如，表达式拆分时还没有进行寄存器分配，在内存中的数据和寄存器中的数据需要用不同的汇编指令进行操作，因此不能在表达式拆分时就完成所有指令生成的工作。同样，在生成基本块时，基本块之间的跳转指令究竟应当如何实现，也可以等到当前这个代码生成环节再实现。

在上面的四阶段划分中，我们暂时忽略的编译器中的一个非常重要的组成部分：编译优化。编译优化的代码可能占现代编译器代码的百分之 80 以上。而由于编译优化的代码非常复杂，人们往往希望能够各种编译器复用这些编译优化的代码。换言之：无论是编译 C 程序，还是 C++。还是 ML 语言，还是 rust 语言等等，无论要编译生成的是 x86 的汇编代码，还是 ARM 的汇编代码，还是 PowerPC 的汇编代码，我们都希望能够尽可能的使用同一套编译优化代码。例如下面就是一个可行的编译流程，我们将着重讲解编译优化之外的各个环节。

抽象语法树
→ 拆分复合表达式 →
→ 生成基本块 →
中间表达（intermediate representation, IR）
→ 编译优化（主要）→
中间表达（intermediate representation, IR）
→ 分配寄存器 →
→ 生成汇编代码（含少量优化）→
汇编代码

1 拆分复合表达式

- 在普通编译器中，在此环节之前还需要做一些其他工作，例如，类型分析，将源代码运算法转化为硬件运算符（例如除法有无符号、指针加法减法等），给变量编号等等。在我们的 while 语言中，情况比较简单，只需要对变量编号，我们在这个拆分步骤中一并完成。例子：

  拆分前

  ```
  x = y * y + z * z
  ```

  拆分后

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

  拆分的同时编号

  ```
  #3 = #1 * #1;
  #4 = #2 * #2;
  #0 = #3 + #4
  ```

- 关键点 1：每个赋值语句至多进行一步计算

  这里的一步计算可能是：运算后存储到变量；从变量存储到内存；从内存读出到变量。

  ```
  enum L1_ConstOrVar {
    L1_T_CONST = 0,
    L1_T_VAR
  };

  struct L1_const_or_var {
    enum L1_ConstOrVar t;
    union {
      struct {unsigned int value; } CONST;
      struct {unsigned int name; } VAR;
    } d;
  };
  ```

  ```
  enum L1_ExprType {
    L1_T_CONST_OR_VAR = 0,
    L1_T_BINOP,
    L1_T_UNOP,
    L1_T_DEREF,
    L1_T_MALLOC,
    L1_T_RI,
    L1_T_RC
  };
  ```

```
struct L1_expr {
  enum L1_ExprType t;
  union {
    struct {struct L1_const_or_var cv; } CONST_OR_VAR;
    struct {enum BinOpType op;
            struct L1_const_or_var cv1;
            struct L1_const_or_var cv2; } BINOP;
    struct {enum UnOpType op;
            struct L1_const_or_var cv; } UNOP;
    struct {struct L1_const_or_var cv; } DEREF;
    struct {struct L1_const_or_var cv; } MALLOC;
    struct {void * none; } RI;
    struct {void * none; } RC;
  } d;
};
```

```
struct L1_const_or_var * L1_TConst(unsigned int value);
struct L1_const_or_var * L1_TVar(unsigned int name);
struct L1_expr * L1_TConstOrVar(struct L1_const_or_var * cv);
struct L1_expr * L1_TBinop(enum BinOpType op,
                           struct L1_const_or_var * cv1,
                           struct L1_const_or_var * cv2);
struct L1_expr * L1_TUnop(enum UnOpType op,
                          struct L1_const_or_var * cv);
struct L1_expr * L1_TDeref(struct L1_const_or_var * cv);
struct L1_expr * L1_TMalloc(struct L1_const_or_var * cv);
struct L1_expr * L1_TReadInt();
struct L1_expr * L1_TReadChar();
```

```
enum L1_CmdType {
  L1_T_ASGN_VAR = 0,
  L1_T_ASGN_MEM,
  L1_T_IF,
  L1_T_WHILE,
  L1_T_WI,
  L1_T_WC
};

struct L1_cmd {
  enum L1_CmdType t;
  union {
    struct {unsigned int name;
            struct L1_expr right; } ASGN_VAR;
    struct {unsigned int name;                          最小的计算单元
            struct L1_const_or_var cv; } ASGN_MEM;
    ...
  } d;
};
```

- 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。

```
struct L1_cmd_list {
  struct L1_cmd * head;
  struct L1_cmd_list * tail;
};

struct L1_cmd_listbox {
  struct L1_cmd_list * head;
  struct L1_cmd_list * * tail;
};
```

二级指针，写入新的cmd要写到哪里去。

此处`tail`为二阶指针，是为了方便的链表尾部插入指令。

```
struct L1_cmd_listbox * L1_CmdListBox_Empty() {
  struct L1_cmd_listbox * res = new_L1_cmd_listbox_ptr();
  res -> head = NULL;
  res -> tail = & (res -> head);
  return res;
}

void L1_CmdListBox_Add1
       (struct L1_cmd_listbox * cs, struct L1_cmd * c)
{
  struct L1_cmd_list * cl = new_L1_cmd_list_ptr();
  * (cs -> tail) = cl;
  cl -> head = c;
  cl -> tail = NULL;
  cs -> tail = & (cl -> tail);
}
```

```
struct L1_cmd {
  enum L1_CmdType t;
  union {
    ...
    struct {struct L1_expr cond;
            struct L1_cmd_listbox * left;
            struct L1_cmd_listbox * right; } IF;
    ...
  } d;
};
```

- 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:
  ...
```

C 代码实现

```
struct L1_cmd {
  enum L1_CmdType t;
  union {
    ...
    struct {struct L1_cmd_listbox * pre;
            struct L1_expr cond;
            struct L1_cmd_listbox * body; } WHILE;
    struct {struct L1_const_or_var cv; } WI;
    struct {struct L1_const_or_var cv; } WC;
  } d;
};
```

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

- 整体框架：对语法树递归处理

  表达式的拆解

所有新的辅助计算都加到box里去

```
struct L1_expr * TAC_gen_expr
                   (struct expr * e,
                    struct L1_cmd_listbox * cs) {
  switch (e -> t) {
  case T_CONST:
    return L1_TConstOrVar(L1_TConst(e -> d.CONST.value));
  case
    ...
  }
}
```

程序语句的拆解：C 函数定义方案一

```
struct L1_cmd_listbox * TAC_gen_cmd(struct cmd * c) {
  ...
}
```

程序语句的拆解：C 函数定义方案二

```
void TAC_gen_cmd(struct cmd * c,
                 struct L1_cmd_listbox * cs) {
  ...
}
```

2 生成基本块（basic block）

什么是基本块呢？每个基本块应当包含一列赋值语句和一个跳转指令。这个跳转指令，可以是有条件跳转，也可以是无条件跳转。

```
struct L1_basic_block_end {
  enum L1_BasicBlockType t;
  struct L1_expr cond;
  unsigned int dst1;
  unsigned int dst2;
};

struct L1_basic_block {
  struct L1_cmd_list * head;
  struct L1_basic_block_end end_info;
};

void L1_set_BBEnd
       (struct L1_basic_block * bb,
        struct L1_basic_block_end * end_info);
```

大家可能会有疑问：

- 有些基本块之后其实不需要跳转语句，从汇编程序的角度看，程序会直接进入下一个基本块；
- 汇编代码中没有双向跳转语句，只有 je 和 jne 这样的条件跳转语句。即指令中说明一个分支的跳转地址，而另一个分支就默认为紧接在跳转指令之后汇编程序。

我们选择在当前阶段统一生成无条件跳转指令或者双分支条件跳转指令，而具体汇编代码的选择（例如是否可以省略无条件跳转指令，是否需要用条件跳转指令和无条件跳转指令的组合来实现双分支跳转等等）则在之后汇编代码生成时再处理。

- 整体框架：遍历程序语句链表，遇到分支或循环语句时创建新基本块

  L1.h

```
void basic_block_gen(struct L1_cmd_list * p,
                     struct L1_basic_block_end * end_info,
                     struct L1_basic_block * bb) {
  struct L1_cmd_list * * ptr_p = & (bb -> head);
  * ptr_p = p;
  while (p) {
    switch (p -> head -> t) {
    case L1_T_IF:
      ...
      basic_block_gen(p -> tail, end_info, ...);
      * ptr_p = NULL;
      return;
    case L1_T_WHILE:
      ...
    default:
      ptr_p = & (p -> tail);
      p = * ptr_p;
  } }
  L1_set_BBEnd(bb, end_info);
}
```

- 处理 if 语句时，如果
  - 当前 BB 是 BB_now，预定的跳转信息是 end_info，
  - if 语句的后续语句不为空

  那么
  - 创建 BB_then 处理 if-then 分支，

  - 创建 BB_else 处理 if-else 分支，
  - 创建 BB_next 处理 if 语句后续的语句，
  - 结束 BB_now 的构造，令其条件跳转到 BB_then 与 BB_else，

  并且

  - 递归调用 basic_block_gen 处理 if-then 分支,预定跳转信息为无条件跳转到 BB_next，
  - 递归调用 basic_block_gen 处理 if-else 分支,预定跳转信息为无条件跳转到 BB_next，
  - 递归调用 basic_block_gen 处理 if 语句后续的语句，预定跳转信息为 end_info。

- 处理 if 语句时，如果
  - if 语句无后续程序，预定进行条件跳转，

  那么
  - 应当创建不包含任何指令的 BB_next，同上进行

- 处理 if 语句时，如果
  - 当前 BB 是 BB_now，
  - if 语句无后续程序，预定无条件跳转到 BB0，

  那么
  - 创建 BB_then 处理 if-then 分支，
  - 创建 BB_else 处理 if-else 分支，
  - 结束 BB_now 的构造，令其条件跳转到 BB_then 与 BB_else，

  并且
  - 递归调用 basic_block_gen 处理 if-then 分支，预定跳转信息为无条件跳转到 BB0，
  - 递归调用 basic_block_gen 处理 if-else 分支，预定跳转信息为无条件跳转到 BB0，

- 处理 while 语句时，如果
  - 当前 BB 是 BB_now，
  - while 语句有后续程序，预定跳转信息为 end_info，

  那么
  - 创建 BB_body 处理 while 语句的循环体，
  - 创建 BB_next 处理 while 语句后续的语句，
  - 如果当前 BB_now 为空，那么令 BB_pre = BB_now 处理 while 语句中计算循环条件的语句，
  - 如果当前 BB_now 不为空，那么创建 BB_pre 处理 while 语句中计算循环条件的语句，结束 BB_now 的构造，令其无条件跳转到 BB_pre，

  并且
  - 递归调用 basic_block_gen 处理 while 语句的循环体，预定跳转信息为无条件跳转到 BB_pre，

  - 递归调用 basic_block_gen 处理 while 语句中计算循环条件的语句，预定跳转信息为条件跳转到 BB_body 与 BB_next，
  - 递归调用 basic_block_gen 处理 while 语句的后续语句，预定跳转信息为 end_info。

- 处理 while 语句时，如果
  - 当前 BB 是 BB_now，
  - while 语句无后续程序，预定跳转信息为条件跳转，

  那么
  - 应当创建不包含任何指令的 BB_next，同上进行

- 处理 while 语句时，如果
  - 当前 BB 是 BB_now，
  - while 语句无后续程序，预定跳转信息为无条件跳转到 BB0，

  那么
  - 创建 BB_body 处理 while 语句的循环体，
  - 如果当前 BB_now 为空，那么令 BB_pre = BB_now 处理 while 语句中计算循环条件的语句，
  - 如果当前 BB_now 不为空，那么创建 BB_pre 处理 while 语句中计算循环条件的语句，结束 BB_now 的构造，令其无条件跳转到 BB_pre，

  并且
  - 递归调用 basic_block_gen 处理 while 语句的循环体，预定跳转信息为无条件跳转到 BB_pre，
  - 递归调用 basic_block_gen 处理 while 语句中计算循环条件的语句，预定跳转信息为条件跳转到 BB_body 与 BB0，

3 寄存器分配

3.1 Liveness 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 bitvectore，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 Bourbaki-Witt 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 Bourbaki-Witt 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 Bourbaki-Witt 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；

  这个变量就放弃了，实在找不到寄存器给你了，准备放到内存里
- 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之前的值不用管

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

从这里先加载出来，加减常数再取值是可以一起操作的。

这样的话 #3的liveness范围会更小

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

rbp一般不算做use

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 改进的寄存器分配算法

尽量分配同一个寄存器，这样赋值操作就不用做了

- 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）；

不能以导致寄存器分配失败和真spill为代价。
Move指令的代价小于一次spill的代价。

- 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

  度数大的不多

  保证coalesce的新节点最后一定会被simply

  - 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`；

simplify到不能simplify之后再走coalesce

和move有关就意味着可能做coalesce更好

不保证合并一定被simply，但合并之后不比合并以前更糟糕（spill 出 更多的变量）

- 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；

coalesce可以全部coalesce完，也可以先simplify，因为simplify不会导致不能合并

- 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；

使得有更多的simply可以做

- 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。