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

  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;
            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;
};
```

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;
       #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

```
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 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 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，

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

  因此 $\mathrm{lub}(X, F(X), F^{(2)}(X)\dots)$ 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。