# 第 5 章 三地址码的生成

从本章开始进入编译器的中段。在语义分析之后，抽象语法树已经包含源代码的所有信息，接下来要把它转化成三地址码的双链表。三地址码是类似汇编的代码，比抽象语法树更接近机器指令。生成三地址码的算法也是为每类节点定义一个回调函数，然后在遍历语法树时调用它并把生成的三地址码挂载在一个双链表上。


## 5.1 回填技术

通常三地址码的生成只需遍历抽象语法树，但在为 break、continue、goto、return 等生成跳转时则要使用回填技术。分析当前语句时还没分析到跳转的目标位置，只能先添加一条三地址码等获得目标位置之后再回写跳转地址叫作回填(Refill)。

### 5.1.1 回填的数据结构

(1) break 要跳到当前循环结束后的第 1 行代码，因为在生成它的三地址码时还没分析完整个循环，并不知道循环结束后的代码在哪里，所以只能回填。

(2) continue 要跳回去检测循环条件，但 do-while 的循环条件在末尾，for 在下次检测之前要更新变量，这两种语句的跳转目标也只能回填。

(3) goto 可能跳到当前函数内的任意一个标签(Label)，它的目标位置更难确定，只能回填。

(4) 标签不用回填，但要记录下来给 goto 回填。

(5) return 表示退出当前函数，它的跳转位置是函数的末尾，只能在分析完函数的代码之后回填。

综上所述，回填的数据结构需要包含 5 种情况，代码如下：

```
//第 5 章/scf_operator_handler_3ac.c
//节选自 SCF 编译器
#include "scf_ast.h"
#include "scf_operator_handler.h"
#include "scf_3ac.h"
```

```
typedef struct { //记录分支跳转的结构体,动态数组里是需要回填的三地址码的指针
    scf_vector_t * _breaks;
    scf_vector_t * _continues;
    scf_vector_t * _gotos;
    scf_vector_t * _labels; //标签
    scf_vector_t * _ends;    //跳转到函数尾的回填,例如 return
} scf_branch_ops_t;
```

### 5.1.2 三地址码的数据结构

三地址码的数据结构,代码如下：

```
//第 5 章/scf_3ac.c
//节选自 SCF 编译器
#include "scf_node.h"
#include "scf_dag.h"
#include "scf_graph.h"
#include "scf_basic_block.h"

typedef struct scf_3ac_operator_s  scf_3ac_operator_t;
typedef struct scf_3ac_operand_s   scf_3ac_operand_t;

struct scf_3ac_operator_s {                    //三地址码的操作码
    int                 type;                  //类型
    const char *        name;                  //名字
};
struct scf_3ac_operand_s {                     //三地址码的操作数
    scf_node_t *        node;                  //对应的抽象语法树节点
    scf_dag_node_t *    dag_node;              //对应的有向无环图节点
    scf_3ac_code_t *    code;                  //对应的三地址码,仅用于跳转指令
    scf_basic_block_t * bb;                    //对应的基本块,仅用于跳转指令
    void *              rabi;
};
struct scf_3ac_code_s {
    scf_list_t          list;                  //用于双链表的元素
    scf_3ac_operator_t * op;                   //三地址码的操作码

    scf_vector_t *      dsts;                  //目的操作数的动态数组
    scf_vector_t *      srcs;                  //源操作数的动态数组

    scf_label_t *       label;                 //标签,仅用于 goto 指令
    scf_3ac_code_t *    origin;

    scf_basic_block_t * basic_block;           //所属的基本块
    uint32_t            basic_block_start:1;   //是否是基本块的开头
    uint32_t            jmp_dst_flag    :1;    //是否是跳转的目标
```

```
    scf_vector_t *      active_vars;          //活跃变量的动态数组
    scf_vector_t *      dn_status_initeds;    //变量的初始化状态
    scf_vector_t *      instructions;         //机器指令的动态数组
    int                 inst_bytes;           //机器指令的字节数
    int                 bb_offset;            //机器指令在基本块内的偏移量
    scf_graph_t *       rcg;                  //变量冲突图
};
```

三地址码的定义包含整个编译器中后段的所有需求，在回填技术里用到的是它的操作码和目的操作数。

(1) op 字段是操作码，只有操作码是跳转指令时才需要回填，其他指令不需要。

(2) dsts 字段是目的操作数的动态数组，跳转指令的目的操作数只有一个，目的操作数的 code 字段即为跳转的目标位置，它是另一条三地址码。

### 5.1.3　回填的步骤

(1) continue 语句在生成完当前循环的所有三地址码之后回填为真正的跳转地址，因为 continue 会跳转到循环的条件或更新表达式，其目标位置可以确定。

(2) break 语句即使在生成完当前循环的所有三地址码之后也无法确定真正的跳转地址，只能确定为循环末尾的下一条三地址码，所以先回填为循环末尾，等处理完整个函数之后再下移一条。

(3) 当发现 goto 语句时查找它的标签位置，当发现标签时回填它的 goto 语句，两者必然一先一后。

(4) 因为标签是下一条三地址码的位置，但刚发现它时只能确定上一条的位置，所以 goto 的回填地址也是目标位置的上一条。

(5) break、goto、return 的回填位置都要在处理完整个函数之后下移一条。

**注意**：return 是跳转到函数末尾而不是直接返回主调函数，因为被调函数退出之前要清理栈。

## 5.2　if-else 的三地址码

if-else 是最常用的控制语句，它在抽象语法树上有 2～3 个子节点，其中条件表达式和 if 主体顺序块的节点是必需的，else 节点可能不存在。

**1. if 的三地址码**

if 的三地址码是在条件表达式之后添加条件跳转(Jump Code with Condition，JCC)。因为条件表达式的结果在运行时才能确定且每次可能不一样，所以这条跳转是不能省略的，除非条件表达式为常量。

(1) 当条件表达式成立时该跳转不会被触发，代码顺势执行到 if 的主体顺序块。

（2）当条件表达式不成立时若 else 节点存在，则跳转到 else，若不存在，则跳转到整个 if 语句块的下一条三地址码。

（3）若 else 存在，则 if 主体顺序块之后要添加一条绝对跳转（Jump Absolutely，汇编码 JMP）以跳过随后的 else 部分，到达整个 if 语句块之后的下一条三地址码。

**注意**：因为这时整个 if 语句块之后的三地址码还没生成，所以只能回填，类似 break 语句。

**2. 条件表达式的三地址码**

条件表达式可以简单到一个常量，也可以复杂到由多个比较和逻辑运算符组成。

（1）如果含有双目逻辑运算符，则要处理与运算（&&）或运算（||）的短路，此时需要添加条件跳转。

（2）如果只含比较运算符，则跳转条件与比较运算符相反，即 if（a==b）的跳转为 JNZ，因为比较结果为 False 时才需要跳转，而比较结果为 True 时不需要跳转。

（3）逻辑非运算（!）和纯表达式都等价于比较目标是否为 NULL，只是二者的跳转条件相反，例如 if（!p）的跳转为 JNZ 而 if(p)的跳转为 JZ。

**注意**：JNZ 是非零跳转，JZ 是零跳转，比较在汇编指令中是先做减法，然后看结果，即查看是大于 0、等于 0，还是小于 0。

**3. else 的三地址码**

else 不需要生成跳转指令，只需把它的内容依次转化成三地址码。嵌套的 else if 在抽象语法树上属于 else 的子节点，它是新的 if 语句且会触发对生成函数的递归调用，如图 5-1 所示。

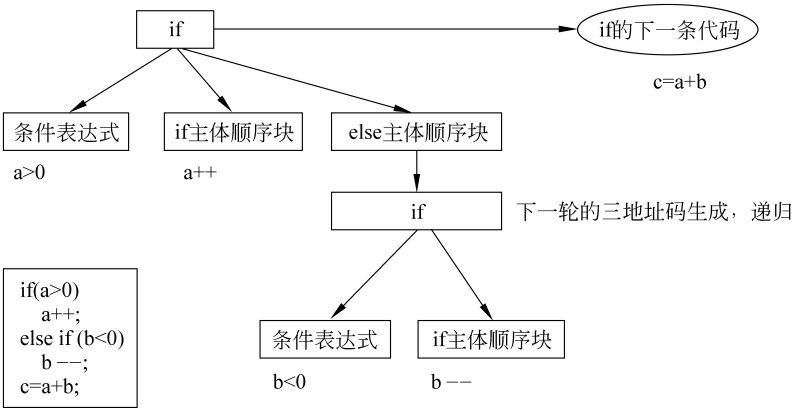



图 5-1　嵌套的 else if 和它的抽象语法树

图 5-1 的抽象语法树转化成的三地址码序列，代码如下：

```
//第 5 章/if_else_3ac.c
CMP a, 0     //第 1 个 if 的条件表达式
JLE _else    //若不成立，则跳转到 else
```

```
INC a       //第 1 个 if 的主体顺序块
JMP _next   //跳转到整个 if 语句块的下一行

_else:      //第 1 个 if 的 else 分支
CMP b, 0    //第 2 个 if 的条件表达式
JGE _next   //若不成立,则跳转到 else 的下一行,实际为_next
DEC b       //第 2 个 if 的主体顺序块,其 else 分支不存在

_next:      //整个 if 语句块的下一行
ADD c; a, b
```

可以看出三地址码与汇编的不同在于它不需要确定每个变量占用的寄存器。

**4. 代码实现**

在 SCF 编译器中 if-else 的三地址码由_scf_op_if()函数生成,代码如下:

```
//第 5 章/scf_operator_handler_3ac.c
#include "scf_ast.h"
#include "scf_operator_handler.h"
#include "scf_3ac.h"
  int _scf_op_if(scf_ast_t * ast,scf_node_t** nodes,int nb_nodes, void * data){
scf_handler_data_t * d =data;                    //三地址码生成的上下文
scf_3ac_operand_t * dst;
scf_list_t * l;
int i;
    if (2 !=nb_nodes && 3 !=nb_nodes)            //子节点必须为 2 个或 3 个
        return -1;
    scf_expr_t * e      =nodes[0];               //0 号子节点为条件表达式
    scf_node_t * parent =e->parent;
    int jmp_op =_scf_op_cond(ast, e, d);         //生成条件表达式的三地址码
    if (jmp_op<0)
        return -1;

    //添加到 else 分支的跳转语句
    scf_3ac_code_t * jmp_else  =scf_branch_ops_code(jmp_op, NULL, NULL);
    scf_3ac_code_t * jmp_endif =NULL;
    scf_list_add_tail(d->_3ac_list_head, &jmp_else->list);

    for (i =1; i<nb_nodes; i++) {
        scf_node_t * node =nodes[i];
        if (_scf_op_node(ast, node, d)<0)
            return -1;
        if (1 ==i) {                             //1 号子节点为 if 主体部分
            if (3 ==nb_nodes) {                  //若存在 else 分支,则跳过
                jmp_endif =scf_branch_ops_code(SCF_OP_GOTO, NULL, NULL);
                scf_list_add_tail(d->_3ac_list_head, &jmp_endif->list);
```

```
            }
            l   =scf_list_tail(d->_3ac_list_head);
            dst =jmp_else->dsts->data[0];       //设置到 else 的跳转目标
            dst->code =scf_list_data(l, scf_3ac_code_t, list);
        }
    }
    int ret =scf_vector_add(d->branch_ops->_breaks, jmp_else);//添加到回填数组
    if (ret<0)
        return ret;

    if (jmp_endif) { //设置跳过 else 分支的目标位置并添加到回填数组
        l   =scf_list_tail(d->_3ac_list_head);
        dst =jmp_endif->dsts->data[0];
        dst->code =scf_list_data(l, scf_3ac_code_t, list);
        ret =scf_vector_add(d->branch_ops->_breaks, jmp_endif);
        if (ret<0)
            return ret;
    }
    return 0;
}
```

在上述代码中，_scf_op_cond()函数为条件表达式生成三地址码，其返回值是条件不成立时的跳转类型，在这里用于到 else 分支的跳转。当 else 分支存在时 if 分支末尾要添加绝对跳转，跳转类型为 SCF_OP_GOTO。在语义完全相同时三地址码的操作码都使用了与抽象语法树节点相同的类型，高级语言中的 GOTO 与绝对跳转 JMP 语义相同。

## 5.3 循环的入口和出口

源代码中除了 if-else 之外最多的是 for 循环。循环是程序中最耗时间的代码，是运行速度的瓶颈所在，循环的优化是生成高效机器码的关键。

**1. 结构化循环**

循环可分为结构化循环和非结构化循环，其中入口和出口都唯一的循环是结构化循环，而不唯一的是非结构化循环。while、do-while、for 都是结构化循环，goto 形成的循环可能是结构化的，也可能不是。

依照《编译原理》，如果把 while、do-while、for 都变成 if{do{}while}的形式，则 if 与 do-while 之间的两个位置就分别是循环的入口和出口，代码如下：

```
//第 5 章/if_do_while.c
if (cond0) {
//循环的入口
    do {
        //循环体
```

```
    } while (cond1);   //cond0 和 cond1 都为条件表达式
//循环的出口
}
```

其中 while 和 for 转换之后的两个条件表达式与原来的相同，而 do-while 转换之后只有 cond1 与原来的相同，cond0 恒为 True。

**2. while 循环的结构化**

while 循环在抽象语法树上有 1～2 个子节点，其中条件表达式是必需的，主体顺序块当循环体为空时可省略。生成 while 循环的三地址码并转化成 if＋do-while 结构的步骤如下：

(1) 记录三地址码的双链表尾部，因为新生成的三地址码要挂载在它之后，所以它是 while 循环的前一条代码(Start Prev)，它之后就是条件表达式的第 1 条代码。

(2) 遍历条件表达式，把生成的三地址码依次挂载在双链表上，三地址码的生成顺序即为表达式的计算顺序。

(3) 获取条件表达式的根运算符，即它在抽象语法树上的根节点，像 if 语句一样确定比较和跳转条件，并添加条件跳转的三地址码(JCC End)。

**注意**：因为当条件表达式不成立时 JCC End 会跳到循环结束(End)之后的下一条代码，当条件表达式成立时进入循环，所以 JCC End 之后即为循环入口。

(4) 记录 JCC End 的位置，它和 Start Prev 之间的即为条件表达式的三地址码(不包含它俩)。

(5) 若循环体不为空，则遍历 while 的主体顺序块，并把生成的三地址码挂载在双链表上。

(6) 把条件表达式的三地址码序列复制到双链表的尾部(Cond Prev)，这就是转换成的 do-while 的循环条件。

**注意**：Cond Prev 既是循环体的结束，也是 do-while 条件表达式之前的第 1 条三地址码。

(7) 添加一条到循环体首部的条件跳转(JCC Loop)即可构成循环，它的跳转条件与 JCC End 相反。

while 循环的三地址码结构如图 5-2 所示。

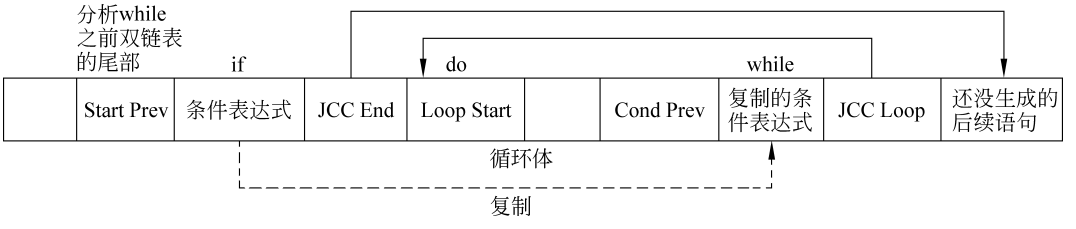



图 5-2　while 循环的三地址码结构

如果循环中包含 continue 语句，则需要跳转到 Cond Prev 的下一条三地址码检查 do-while 的表达式，而不是回到开头去检查 if 的表达式。如果循环中包含 break 语句，则需

要跳转到循环结束后的下一条三地址码，因为暂时还没生成，只能记录为循环的最后一条代码，以后再下移一条。循环中的 goto 语句只跟标签的出现时机有关，与循环无关。return 语句则只能记录下来，留待函数结束时回填。

### 3. do-while 循环的结构化

只需删除图 5-2 中的 if 条件表达式和 JCC End，因为 do-while 要先运行循环体再判断条件表达式，相当于开头的 if 条件肯定为真。

### 4. for 循环的结构化

1）for 循环的抽象语法树

for 循环在抽象语法树上有 4 个子节点，编号从 0～3 依次为初始化表达式组、条件表达式、更新表达式组、循环体，与它们在源代码中的出现顺序一致。for 循环的 4 个子节点可能为空，但即使为空也得用 NULL 作为占位符以保证其他节点的编号正确。若条件表达式为 NULL，则条件恒为 True，若其他子节点为 NULL，则忽略。

2）初始化表达式组

初始化表达式组可能只包含 1 个表达式，也可能包含多个。它只运行一次，并不是真正的循环部分。在生成三地址码时要把它放在循环之前，然后由其他部分组成循环结构。

3）continue 语句

for 循环里的 continue 要跳转到更新表达式，而不是像 while 循环一样跳转到条件表达式。在生成循环体的三地址码之后要记录更新表达式的起始位置，它同时是 continue 语句的跳转目标的前一个位置（Continue Prev），它之后是真正的跳转目标。

4）for 循环的三地址码

（1）先生成初始化表达式组的三地址码并记录双链表的末尾，这是循环开始前的上一条代码（Start Prev）。

（2）生成条件表达式的三地址码，然后添加一条跳转指令（JCC End），Start Prev 和 JCC End 之间的是循环条件。

（3）生成循环体的三地址码并记录双链表的末尾，这是更新表达式的上一条代码，也是 continue 的跳转目标（Continue Prev）。

（4）生成更新表达式的三地址码并记录双链表的末尾，这是条件表达式要复制的起始位置。

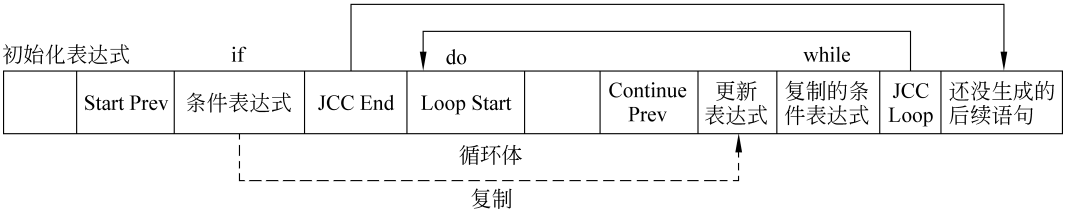
（5）将条件表达式复制到双链表的末尾，然后添加一条跳转指令（JCC Loop），其目标位置指向循环体的开头以构成循环。

for 循环的三地址码结构如图 5-3 所示。

### 5. 代码实现

for 循环的三地址码生成由_scf_op_for()函数处理，代码如下：

```
//第 5 章/scf_operator_handler_3ac.c
#include "scf_ast.h"
#include "scf_operator_handler.h"
```




注释如下。
Start Prev:循环开始前的上一个位置；JCC End：到循环之后的跳转；Loop Start:循环的开头；Continue Prev:continue语句跳转位置的前一个；JCC Loop:到循环开头的跳转

图 5-3　for 循环的三地址码结构

```
#include "scf_3ac.h"
  int _scf_op_for(scf_ast_t * ast,scf_node_t** nodes,int nb_nodes,void * data){
scf_handler_data_t * d =data;                    //三地址码生成的上下文
scf_3ac_operand_t * dst;
scf_3ac_code_t * jmp_end =NULL;                  //到循环结束后的跳转
scf_list_t * l;
int i;
    assert(4 ==nb_nodes);                        //子节点必须为 4 个
    if (nodes[0]) {                              //0 号子节点为初始化表达式,在循环外
        if (_scf_op_node(ast, nodes[0], d)<0)
            return -1;
    }
    scf_list_t * start_prev =scf_list_tail(d->_3ac_list_head); //循环开始位置

    if (nodes[1]) {                              //1 号子节点为条件表达式
        assert(SCF_OP_EXPR ==nodes[1]->type);
        int jmp_op =_scf_op_cond(ast, nodes[1], d);
        if (jmp_op<0)
            return -1;
        jmp_end =scf_branch_ops_code(jmp_op, NULL, NULL); //当条件不成立时的跳转
        scf_list_add_tail(d->_3ac_list_head, &jmp_end->list);
    }

    scf_branch_ops_t * local_branch_ops =scf_branch_ops_alloc();
    scf_branch_ops_t * up_branch_ops =d->branch_ops; //记录上层的回填数组
    d->branch_ops =local_branch_ops;             //更新为 for 循环的回填数组
    if (nodes[3]) {                              //3 号节点为循环主体
        if (_scf_op_node(ast, nodes[3], d)<0)
            return -1;
    }

     //记录 continue 语句的跳转位置
    scf_list_t * continue_prev =scf_list_tail(d->_3ac_list_head);
```

```
    if (nodes[2]) {                                    //2 号节点为更新表达式
        if (_scf_op_node(ast, nodes[2], d)<0)
            return -1;
    }

    if (_scf_op_end_loop(start_prev, continue_prev, jmp_end,
                         up_branch_ops, d)<0)          //回填并结束循环
        return -1;

    d->branch_ops    =up_branch_ops;                   //恢复上层的回填数组
    scf_branch_ops_free(local_branch_ops);
    local_branch_ops =NULL;
    return 0;
}
```

因为 for 循环的主体也是一个作用域，所以它也是一个对上层作用域有屏蔽作用的回填区域。在生成循环体的三地址码之前要更新回填数组 d->branch_ops，在生成结束后再更新回来。因为 for 循环在抽象语法树上的子节点顺序与源代码一致，但三地址码的生成顺序要与实际运行顺序一致，所以 2 号子节点和 3 号子节点的处理顺序是相反的。

**6. 循环的入口和出口**

（1）当循环被转化成 if+do-while 的结构之后，在循环内不含 goto 语句的情况下循环的入口和出口都是唯一的。

（2）如果把变量的加载提前到循环的入口并把保存推迟到循环的出口，则可在循环运行期间将变量保持在寄存器内，这样可以降低内存的读写次数。

学过汇编的人都知道寄存器的读写速度比内存更快而循环又是最消耗时间的地方，降低循环的内存读写次数可以提高运行效率。若循环的入口和出口不唯一，则会给循环优化带来复杂性。在生成三地址码时让循环变得结构化就为下一步的循环优化打下了基础。

## 5.4 指针与数组的赋值

指针和数组都是对变量的间接使用，这在带来灵活性的同时也导致了它们在做源操作数和目的操作数时的不同，带来了更多的风险。

**1. 指针的赋值**

当星号（*）是目的操作数时修改的是指针指向的变量，当它是源操作数时将指向的变量值读取到一个临时变量中，当不含星号时则是对指针的普通赋值，如图 5-4 所示。

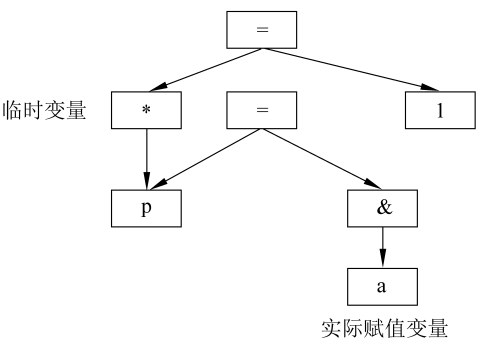



图 5-4 指针的赋值解引用

（1）p=&a 是对指针的普通赋值，它让 p