第 5 章 中间代码生成及优化

5.1 中间代码生成简介

在语法分析和语义检查阶段，始终在与语句、表达式和外部声明这 3 个概念打交道。通过声明最终建立了相应的类型结构，并在符号表中保存了相关标识符的类型信息。到了中间代码生成阶段，就不再需要处理外部声明了，只需要为语句和表达式生成相应的中间代码即可。文件 ucl\tranexpr.c 用于为表达式生成中间代码，文件名 tranexpr 是 translate expression 的缩写，而 ucl\transtmt.c 则用于为语句生成中间代码。下面先结合图 5.1 所示的例子来讨论。图 5.1 第 1～9 行的递归函数用于计算 n 阶乘，第 12～15 行的 while 循环用于打印出 1！～10！的值，第 19～45 行是 UCC 编译器所生成的中间代码的字符串形式，UCC 编译器内部会用三地址码的结构来表示中间代码。“龙书”把语法树和三地址码都视为中间代码，本书在讨论 UCC 编译器时，如果不作特别声明，中间代码指的是三地址码。三地址码包含两个源操作数、一个目的操作数及一个运算符。例如对于 t1:a＋b 而言，加号＋是运算符，a 和 b 是两个源操作数，而 t1 是目的操作数(即用于存放运算结果)，这三个操作数对应 3 个地址，所以称这样的中间代码为三地址码。

```
1   // hello.c                              16      return 0;
2   #include<stdio.h>                      17  }
3   int f(int n){                          18  // hello.uil
4     if(n<1){                             19  function f
5         return 1;                        20      if (n >=1) goto BB3;
6     }else{                               21  BB1:
7         return n * f(n-1);               22      return 1;
8     }                                    23      goto BB4;
9   }                                      24  //
10  int main(int argc,char * argv[]){      25  BB3:
11     int i=1;                            26      t0 : n+-1;
12     while(i<=10){                       27      t1 : f(t0);
13       printf("f(%d)=%d\n",i,f(i));      28      t2 : n * t1;
14       i++;                              29      return t2;
15     }                                    30  BB4:
```

图 5.1 基本块

```
31     ret                                    39     printf(t0, i, t1);
32  ////////////////////////                  40     ++i;
33  function main                             41  BB7:
34     i=1;                                   42     if (i<=10) goto BB6;
35     goto BB7;                              43  BB8:
36  BB6:                                      44     return 0;
37     t0 :&str0;                             45     ret
38     t1 : f(i);
```

图 5.1　（续）

图 5.1 第 20 行的中间代码表示有条件的跳转，第 23 行的 goto 表示无条件的跳转，在汇编语言中，都有与之对应的汇编指令。低级语言中的有条件或无条件的跳转会引起控制流的转移，由此可以实现高级语言中的分支语句 if 和循环语句 while 等控制结构。当然，函数返回也是一种控制流的变化，在 UCC 编译器生成的中间代码中，第 22 行的 return 1 只是把返回值设为 1，真正的返回动作由第 31 行的 ret 指令来完成。这样处理的好处在于，即便在 C 语言中出现了多条 return 语句（如第 5 行和第 7 行的 return 语句），但在中间代码层次，只在第 22 行设置返回值为 1，第 29 行设置返回值为 t2，真正的返回动作只需要在同一个地方处理（即第 31 行的 ret 指令），这也意味着从函数的入口处开始执行，离开函数时也只有一个出口。执行函数调用时，要依次把实参和返回地址入栈，之后无条件跳转到函数起始地址，因此函数调用也可看成是一种无条件跳转。生成中间代码后，还需要进行代码优化，此时需要考虑控制流的变化。我们期望把相邻的若干条中间代码当作一个整体来处理，例如第 26～29 行的这 4 条中间代码，控制流只能从这个整体的第一条指令进入（此处即第 26 行对应的中间代码），离开时从最后一条指令离开（此处即第 29 行的中间代码），称这样的“整体”为一个基本块（basic block），第 25 行的 BB3 表示第 3 个基本块。由此，整个 C 程序就可由若干个基本块构成。

对图 5.1 第 19～45 行的中间代码而言，从静态来看，这些基本块是从上至下依次排列的，用一个链表就可以存放这些基本块。但从条件跳转和无条件跳转的动态语义来看，若把第 41 行的基本块 BB7 看成一个结点，则执行完第 42 行的条件跳转语句后，接下来可能执行的是第 44 行的中间代码，也可能是第 37 行的中间代码。这就相当于存在一条由第 41 行的基本块 BB7 指向第 43 行的基本块 BB8 的有向边，同时从 BB7 到 BB6 也存在一条有向边。由此，基本块成了点，而控制流的转移就成了有向边，可把整个程序的动态执行的路线看成数据结构中的“图”，这个由基本块构成的图被称为控制流图（Control Flow Graph），简写为 CFG。后续的分析和优化都是基于控制流图这样的数据结构进行的。需要说明的是，UCC 编译器的中间代码优化工作只在基本块中进行，并没有进行函数间（过程间）的代码优化，因此 UCC 编译器在划分基本块时，并没有把函数调用当作基本块的最后一条指令，例如在图 5.1 第 36 行的基本块 BB6 中，第 39 行调用的函数 printf()并不是基本块的最后一条指令。在中间代码优化阶段，ucl\simp.c 中的 PeepHole()函数会对 CALL 指令进行优化处理，PeepHole 译为窥孔，其含义是通过一个小孔或小窗口

来看世界，一次只看到世界的一小部分(局部)。在 UCC 编译器中，这个孔的大小一般就限定在一个基本块内。例如，对于以下两条中间代码来说：

```
t1=f();
num=t1;
```

由于 UCC 编译器没有把函数调用 f()置于基本块的最末尾，因此这两条中间代码就可以处于同一个基本块中，在 PeepHole()函数对其所处基本块进行窥孔优化时，就会发现临时变量 t1 是多余的，这两条中间代码可优化为如下所示的中间代码：

```
num=f();
```

为了方便进行这样的优化，UCC 编译器在划分基本块时，并没有把函数调用当作控制流的转移。但我们知道，在真正执行机器指令时，函数调用确实会导致控制流的转移，UCC 编译器在后续阶段产生 x86 汇编指令时，还会在 ucl\x86.c 的 EmitBlock()函数中对 CALL 指令进行特殊判断，从而保存函数调用前用到的一些寄存器。

简单来说，为了存放如图 5.1 第 19～45 行的中间代码，需要由若干个基本块构成的链表结构；为了描述控制流在基本块间的动态转移，需要控制流图的数据结构。图 5.2 第 2～8 行的结构体 irinst 用于描述一条三地址码中间代码，第 7 行的 opds[3]用于存放 3 个操作数，每个操作数由一个符号对象 struct symbol 来表示，在第 2.5 节时简介过符号的相关概念。第 6 行的 opcode 用于存放运算符。由于一个基本块可以包含若干条中间代码，可用第 3 行的 prev 和第 4 行的 next 来构成双向链表结构，第 5 行的 ty 用于记录运算结果的类型信息。

```
1  // Intermediate Representation
2  typedef struct irinst{
3    struct irinst *prev;
4    struct irinst *next;
5    Type ty;
6    int opcode;
7    Symbol opds[3];
8  } *IRInst;
9  // control flow graph edge
10 typedef struct cfgedge{
11    BBlock bb;
12    struct cfgedge *next;
13 } *CFGEdge;
14 // Basic Block
15 struct bblock{
16    struct bblock *prev;
17    struct bblock *next;
```

图 5.2　与中间代码有关的数据结构

```
18    Symbol sym;
19    // successors
20    CFGEdge succs;
21    // predecessors
22    CFGEdge preds;
23    struct irinst insth;
24    // number of instructions
25    int ninst;
26    // number of successors
27    int nsucc;
28    // number of predecessors
29    int npred;
30    // number of references
31    int ref;
32 };
33 static void AddPredecessor(BBlock bb, BBlock p){
34    CFGEdge e;
35    ALLOC(e);
36    e->bb=p;
37    e->next=bb->preds;
38    bb->preds=e;
39    bb->npred++;
40 }
41 static void AddSuccessor(BBlock bb, BBlock s){
42    CFGEdge e;
43    ALLOC(e);
44    e->bb=s;
45    e->next=bb->succs;
46    bb->succs=e;
47    bb->nsucc++;
48 }
49 void DrawCFGEdge(BBlock head, BBlock tail){
50    AddSuccessor(head, tail);
51    AddPredecessor(tail, head);
52 }
53
54 void AppendInst(IRInst inst){
55    assert(CurrentBB !=NULL);
56    CurrentBB->insth.prev->next=inst;
57    inst->prev=CurrentBB->insth.prev;
58    inst->next=&CurrentBB->insth;
```

图 5.2 （续）

```
59      CurrentBB->insth.prev=inst;
60      CurrentBB->ninst++;
61  }
```

图 5.2 （续）

图 5.2 第 15～32 行的结构体用于刻画一个基本块，第 16 行的 prev 和第 17 行的 next 用于构造由若干个基本块形成的双向链表。双向链表描述了如图 5.4 所示的基本块的静态结构；而在动态结构控制流图中，一个结点可以有多个前驱，也可以有多个后继，第 20 行的 succs 用于记录当前基本块的所有后继结点，而第 22 行的 preds 用于记录其所有的前驱结点。第 25 行的 ninst 用于记录当前基本块中有多少条中间代码，第 27 行的 nsucc 用于记录后继结点的个数，而 29 行的 npred 则用于记录前驱结点的个数。第 18 行的 sym 用于存放基本块的名称，例如 BB3 等。第 23 行的 insth 对应一条占位用的中间代码，仅仅用于充当双向链表的头结点，并不对应任何实际的代码。

由于一个基本块 Bx 可以有多个前驱{B1，B2，B3，…，Bn}，每个前驱到基本块 Bx 都存在一条有向边；同理，该基本块 Bx 也可以有多个后继，从 Bx 到各个后继结点也存在相应的有向边。图 5.2 第 10～13 行的结构体 struct cfgedge 用于描述一条有向边，第 11 行的 bb 域用于存放基本块的前驱（或者后继）；第 12 行的 next 域用于构成若干个前驱（或者后继）形成的单向链表。第 49 行的函数 DrawCFGEdge(head，tail)用于构造一条从基本块 head 指向基本块 tail 的有向边，这意味着 tail 是 head 的后继结点，通过第 50 行的函数 AddSuccessor()把 tail 加入到基本块 head 的 succs 域所指向的后继链表中；同时，head 是 tail 结点的前驱，UCC 把 head 加入到基本块 tail 的 preds 域所指向的前驱链表中，这个工作由第 51 行的函数 AddPredecessor()来实现。

图 5.2 第 54 行的函数 AppendInst()用于往当前基本块中添加一条中间代码，第 55～59 行的 4 条语句用于实现双向链表的插入操作。

接下来，初步了解一下中间代码生成的总体执行过程，如图 5.3 所示，第 55～64 行的 Translate()函数实现了由抽象语法树到三地址码的翻译，第 57 行的 while 循环依次对当前 C 文件中的各个函数进行翻译，实际的工作由第 60 行的 TranslateFunction()函数来完成。图 5.3 第 38～54 行给出了函数 TranslateFunction()的主要代码，按照前面对 return 语句的处理，不论函数内部的控制流有多复杂，整个函数定义都只有一个入口和一个出口。第 43 行和第 44 行分别调用 CreateBBlock()来创建这两个基本块，第 20～26 行给出了 CreateBBlock()函数的代码，第 23 行用于设置头结点为空指令 NOP（No OPeration 的缩写，即“无任何实际运算”）。第 46 行调用 TranslateStatement()函数实现了对函数体的翻译，函数体实际上是一个复合语句。第 1～16 行列出了一个函数指针表，第 2～15 行的各函数完成了各语句的翻译工作，第 17～19 行的 TranslateStatement()函数只是查询这个函数指针表而已。在后续章节将对第 2～15 行中的各个函数进行分析。第 48 行的 Optimize()函数用于对生成的中间代码进行优化，第 50 行的 while 循环用于给各个基本块命名，形如 BB1 和 BB2 等。图 5.3 第 38～54 行就是中间代码生成及优化的主要流程。

```
1  static void (*StmtTrans[])(AstStatement)={
2    TranslateExpressionStatement,
3    TranslateLabelStatement,
4    TranslateCaseStatement,
5    TranslateDefaultStatement,
6    TranslateIfStatement,
7    TranslateSwitchStatement,
8    TranslateWhileStatement,
9    TranslateDoStatement,
10   TranslateForStatement,
11   TranslateGotoStatement,
12   TranslateBreakStatement,
13   TranslateContinueStatement,
14   TranslateReturnStatement,
15   TranslateCompoundStatement
16 };
17 static void TranslateStatement(AstStatement stmt){
18   (*StmtTrans[stmt->kind -NK_ExpressionStatement])(stmt);
19 }
20 BBlock CreateBBlock(void){
21    BBlock bb;
22    CALLOC(bb);
23    bb->insth.opcode=NOP;
24    bb->insth.prev=bb->insth.next=&bb->insth;
25    return bb;
26 }
27 void StartBBlock(BBlock bb){
28   IRInst lasti;
29   CurrentBB->next=bb;
30   bb->prev=CurrentBB;
31   lasti=CurrentBB->insth.prev;
32   if (lasti->opcode !=JMP
33        && lasti->opcode !=IJMP){
34        DrawCFGEdge(CurrentBB, bb);
35   }
36   CurrentBB=bb;
37 }
38 static void TranslateFunction(AstFunction func){
39   BBlock bb;
40   FSYM=func->fsym;
```

图 5.3 Translate()函数

```
41    …
42    TempNum=0;
43    FSYM->entryBB=CreateBBlock();
44    FSYM->exitBB=CreateBBlock();
45    CurrentBB=FSYM->entryBB;
46    TranslateStatement(func->stmt);
47    StartBBlock(FSYM->exitBB);
48    Optimize(FSYM);
49    bb=FSYM->entryBB;
50    while (bb !=NULL){
51        bb->sym=CreateLabel();
52        bb=bb->next;
53    }
54  }
55  void Translate(AstTranslationUnit transUnit){
56    AstNode p=transUnit->extDecls;
57    while (p){
58        if (p->kind==NK_Function
59                && ((AstFunction)p)->stmt){
60            TranslateFunction((AstFunction)p);
61        }
62        p=p->next;
63    }
64    …
65  }
```

图 5.3 （续）

需要注意的是，图 5.3 第 43 行调用的 CreateBBlock()函数返回后，新创建的基本块对象并未加入到双向链表中。只有调用了第 27 行的 StartBBlock(bb)函数后，才会把函数参数 bb 所指向的基本块对象加入到双向链表中，第 29、30 行的代码完成了这个插入操作。图 5.3 第 32 行的 if 语句用于检查当前基本块的最后一条指令是否会使控制流转移到参数 bb 所指向的基本块，如果可能出现这样的转移，就会在第 34 行调用 DrawCFGEdge()函数，构造一条由当前基本块指向参数 bb 基本块的有向边，之后在第 36 行使参数 bb 成为新的当前基本块。基本块末尾的无条件跳转指令 JMP 会导致控制流转移到不相邻的基本块上，例如图 5.1 基本块 BB1 第 23 行的 goto BB4 指令就会跳往与 BB1 不相邻的基本块 BB4。对于这种情况，在生成无条件跳转指令时，UCC 会进行有向边的构造，相应的操作可参见 ucl\gen.c 中的函数 GenerateJump()。类似地，如果基本块的最末尾一条指令是如下所示的间接跳转，也不调用图 5.3 第 34 行的 DrawCFGEdge()函数，而是由 ucl\gen.c 中的函数 GenerateIndirectJump()根据实际的跳转目标来构造有向边。

```
//根据 t0 的值来确定跳转的目标
goto (BB1,BB2,BB3,)[t0];
```

这两个函数并不复杂，这里从略。在进行 switch 语句的翻译时，会用到间接跳转。

而如果当前基本块的最末一条指令是有条件跳转指令，则控制流还是可能流向相邻的下一个基本块的，此时需要调用图 5.3 第 34 行 DrawCFGEdge()函数来构造有向边，例如图 5.1 的第 42 行。当然，如果最末一条指令不是跳转指令，那么控制流一定是流向相邻的下一个基本块，此时也需要调用图 5.3 第 34 行的 DrawCFGEdge()函数，例如图 5.1 的第 40 行。对于图 5.1 中的 main()函数而言，图 5.4 给出了其基本块的静态结构和动态结构，静态结构采用的是双向链表，而动态结构采用的是控制流图。

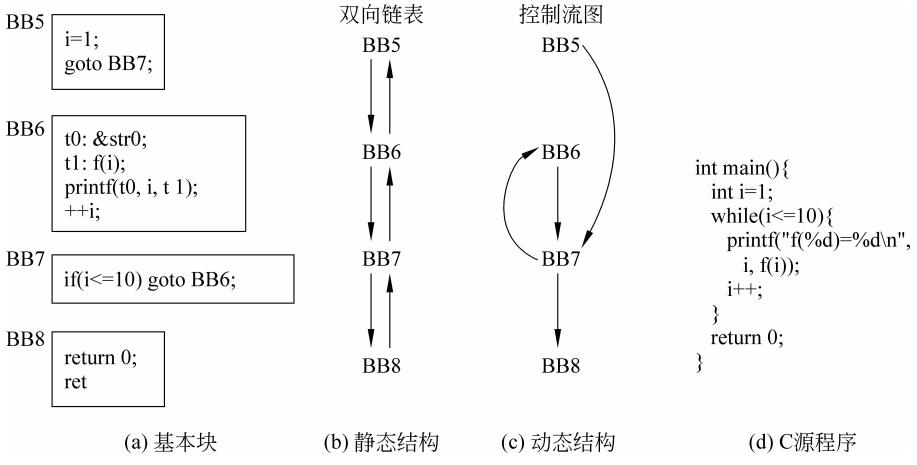

图 5.4　基本块的静态结构和动态结构

5.2　表达式的翻译

5.2.1　布尔表达式的翻译

本节仍然按照语法分析和语义检查时的思路，先讨论表达式的翻译，再处理语句。表达式从概念上来说可分为算术表达式和布尔表达式，在一些编程语言(例如 Java)中对这两者是有严格区分的，算术表达式的结果是整数或浮点数等，而布尔表达式的结果是逻辑上的真或假。布尔是英国数学家，由于布尔较早进行了关于“与或非”逻辑运算的研究，为了纪念这位先驱，在 Java 中引入了关键字 boolean，而在 C++ 中引入了关键字 bool 来表达逻辑上的真或假。C 语言中并没有专门的布尔类型，而是把非 0 的值当作真，把 0 当作假。

```
int a, b;
if(a+b){              // Java 中非法，a+b 为算术值，非布尔值；C 语言中合法
}
if(a==b){             //a==b 为布尔值，在 C 语言或 Java 中都合法
}
```

为了讨论的方便，不妨认为C语言存在布尔表达式，例如a&&b、a‖b或者a>b等，也存在算术表达式，例如a+b或者a&b，但是不存在C程序员可见的布尔类型关键字bool或者boolean。对于算术表达式而言，C编译器需要对整个表达式进行求值，例如，要计算a和b相加的值或者a和b按位与的值，并把运算结果保存到一个临时变量中，这个临时变量的值就代表了整个表达式的值。而对于布尔表达式来说，并不会对整个表达式进行求值，而是生成相应的跳转指令。对于a&&b而言，当a为假时，不论b为何值，a&&b的值都为假，此时我们并不需要对b进行求值。若把a&&b换成f()&&g()，则当f()的返回值为假时，不需要对函数g()进行求值，此时函数g()根本就没有被调用，这就是C语言中的短路运算。

先举个简单的例子来说明这些概念，如图5.5所示，第1～21行给出了一个简单的C程序，第22～60行是与之对应的中间代码。对于第4～8行的C代码来说，与其对应的中间代码在第24～31行，需要先对第4行的算术表达式a+b进行求值，第25行的中间代码t0:a+b表示把a+b求值后的结果存于临时变量t0中，对表达式进行求值的工作由ucl\tranexpr.c中的TranslateExpression()函数来完成。但对于第9行的布尔表达式a&&b来说，只是生成第33行和第35行的条件跳转指令，并没有对整个布尔表达式a&&b进行求值，当a为假时，控制流直接由第33行转移到基本块BB6处，即不再需要判断b的值。产生这些跳转指令的工作由ucl\tranexpr.c中的TranslateBranch()函数来实现。在x86汇编语言中，跳转指令对应的助记符为Jump，缩写为J；但在Arm汇编中，跳转指令对应的是Branch，缩写为B，单词Branch是分支的意思（也有些人写为分枝），这意味着，如果是有条件跳转，控制流就会“开叉”，相当于一根树枝分成若干个细枝了，例如图5.5第35行的有条件跳转，控制流可能流向第37行，也可能流向第40行。

```
1   // hello.c                                  16      }else{
2   int a , b, c;                               17          c=6;
3   int main(int argc,char * argv[]){           18      }
4     if(a+b){                                  19      c=a||b;
5         c=1;                                  20      return 0;
6     }else{                                    21  }
7         c=2;                                  22  // hello.uil
8     }                                         23  function main
9     if(a && b){                               24  //  if(a+b)
10        c=3;                                  25      t0 : a+b;
11    }else{                                    26      if (! t0) goto BB2;
12        c=4;                                  27  BB1:
13    }                                         28      c=1;
14    if(a & b){                                29      goto BB3;
15        c=5;                                  30  BB2:
```

图5.5　布尔表达式和算术表达式

```
31      c=2;                                46      goto BB10;
32  BB3:  // if(a && b)                     47  BB9:
33      if (! a) goto BB6;                  48      c=6;
34  BB4:                                    49  BB10:  // c=a||b
35      if (! b) goto BB6;                  50      if (a) goto BB13;
36  BB5:                                    51  BB11:
37      c=3;                                52      if (b) goto BB13;
38      goto BB7;                           53  BB12:
39  BB6:                                    54      c=0;
40      c=4;                                55      goto BB14;
41  BB7:  // if(a & b)                      56  BB13:
42      t1 : a & b;                         57      c=1;
43      if (! t1) goto BB9;                 58  BB14:
44  BB8:                                    59      return 0;
45      c=5;                                60      ret
```

图 5.5　（续）

对于图 5.5 第 14 行的算术表达式 a&b 来说，需要先对整个表达式 a&b 进行求值，再根据其结果进行跳转，如第 42、43 行所示。比较特殊的是第 19 行的布尔表达式 a‖b，按之前的介绍，翻译布尔表达式时，只是产生一些跳转指令来改变控制流，如第 50 行和第 52 行所示。但按照第 19 行赋值语句的语义，需要在 a‖b 为真时给变量 c 赋值 1；在 a‖b 为假时给变量赋值 0。因此，C 编译器需要产生第 54～57 行的代码。这些工作由 ucl\tranexpr.c 中的函数 TranslateBranchExpression()来完成。

简而言之，如果需要显式地求出表达式的值，就调用函数 TranslateExpression()；如果只是需要生成一些跳转指令来改变当前的控制流，则使用 TranslateBranch()函数。如果在调用 TranslateExpression()函数来对表达式求值时遇到的表达式是形如图 5.5 第 19 行的赋值语句中的布尔表达式 a‖b，此时就再调用 TranslateBranchExpression()函数来处理，进而产生如图 5.5 第 50～57 行的代码。

趁热打铁，现在就来看一下函数 TranslateBranchExpression()的代码，如图 5.6 第 1～18 行所示。第 4 行通过调用 CreateTemp()函数创建了一个临时变量，不妨设这个临时变量为 t。需要根据布尔表达式的控制流来执行 t=0 或者 t=1 的代码，第 11 行调用 GenerateMove()函数产生 t=0 的代码，第 15 行则用于产生 t=1。当表达式 expr 为假时，在执行完 t=0 后，还要通过第 12 行的 GenerateJump()函数产生一条无条件跳转指令，跳过 t=1 这个指令。中间代码 t=0 是基本块 falseBB 的第一条指令，而 t=1 为基本块 trueBB 的第一条指令。在调用第 8 行的 TranslateBranch()函数来产生跳转指令时，需要把 trueBB 和 falseBB 作为跳转目标传递给 TranslateBranch()函数，这样才能生成形如图 5.5 第 50～52 行的跳转指令。