

9.3 对图9.32的流图，计算：

- (a) 为到达-定值分析，计算每个块的gen、kill、IN和OUT集合。
- (b) 为可用表达式分析，计算每个块的e_gen、e_kill、IN和OUT集合。
- (c) 为活跃变量分析，计算每个块的def、use、IN和OUT集合。

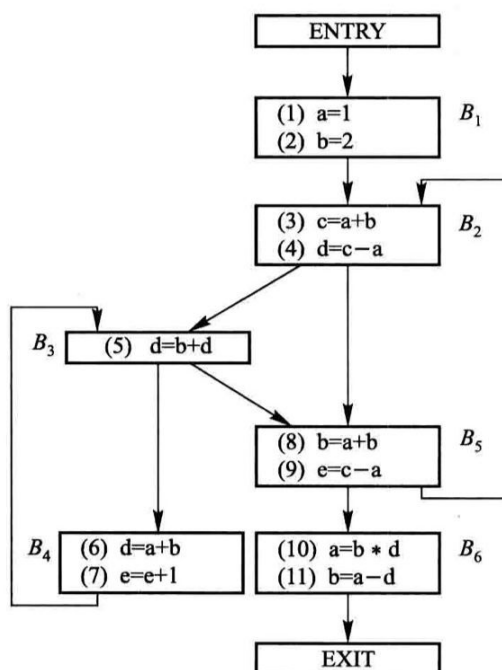


图 9.32 一个流图

(a) 到达-定值分析

	gen	kill
B1	(1), (2)	(8), (10), (11)
B2	(3), (4)	(5), (6)
B3	(5)	(4), (6)
B4	(6), (7)	(4), (5), (9)
B5	(8), (9)	(2), (7), (11)
B6	(10), (11)	(1), (2), (8)

	IN	OUT	IN	OUT
B1	0000 0000 000	1100 0000 000	0000 0000 000	1100 0000 000
B2	1100 0000 000	1111 0000 000	1111 1001 100	1111 0001 100
B3	1111 0000 000	1110 1000 000	1111 0111 100	1110 1011 100
B4	1110 1000 000	1110 0110 000	1110 1011 100	1110 0111 000
B5	1111 1000 000	1011 1001 100	1111 1011 100	1011 1001 100
B6	1011 1001 100	0011 1000 111	1011 1001 100	0011 1000 111

第三次迭代后无变化，算法停止

(b) 可用表达式分析

	e_gen	e_kill
B1	{1, 2}	{a+b, c-a, b+d, b*d, a-d}
B2	{a+b, c-a}	{b+d, b*d, a-d}
B3	{}	{b+d, b*d, a-d}
B4	{a+b}	{b+d, b*d, a-d, e+1}
B5	{c-a}	{a+b, b+d, b*d, e+1}
B6	{a-d}	{1, 2, a+b, c-a, b+d, b*d}

	OUT	IN	OUT
B1	U	{}	{1, 2}
B2	U	{1, 2}	{1, 2, a+b, c-a}
B3	U	{1, 2, a+b, c-a}	{1, 2, a+b, c-a}
B4	U	{1, 2, a+b, c-a}	{1, 2, a+b, c-a}
B5	U	{1, 2, a+b, c-a}	{1, 2, c-a}
B6	U	{1, 2, c-a}	{1, 2, a-d}

第二次迭代后无变化，算法停止

(C) 活跃变量分析

	use	def
B6	{}	{a , b}
B5	{a , b}	{c , d}
B4	{b , d}	{}
B3	{a , b , e}	{d}
B2	{a , b , c}	{e}
B1	{b , d}	{a}

	IN	OUT	IN	OUT
B6	{}	{b , d}	{}	{b , d}
B5	{b , d}	{a , b , c , d}	{a , b , d , e}	{a , b , c , d}
B4	{}	{a , b , e}	{a , b , c , d , e}	{a , b , c , e}
B3	{a , b , c , d , e}	{a , b , c , d , e}	{a , b , c , d , e}	{a , b , c , d , e}
B2	{a , b , c , d , e}	{a , b , e}	{a , b , c , d , e}	{a , b , e}
B1	{a , b , e}	{e}	{a , b , e}	{e}

第三次迭代后无变化，算法停止

9.31 下面的C程序分别经非优化编译和2级以上（含2级，如命令行选项 `-O2`）的优化编译后，生成的两个目标程序运行时的表现不同（例如，编译器是GCC: (GNU) 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)）。请回答它们运行时的表现有何不同，并说明原因。

```
int f(int g( )) {
    return g(g);
}
main() {
    f(f);
}
```

我的编译器版本：GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0

通过非优化编译得到以下汇编代码（部分主要代码）：

```
f:
.LFB0:
    .cfi_startproc
    endbr64
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq     %rsp, %rbp
    .cfi_def_cfa_register 6
    subq     $16, %rsp
    movq     %rdi, -8(%rbp)
```

```

    movq    -8(%rbp), %rax
    movq    -8(%rbp), %rdx
    movq    %rax, %rdi
    movl    $0, %eax
    call    *%rdx
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size    f, .-f
    .globl   main
    .type    main, @function
main:
.LFB1:
    .cfi_startproc
    endbr64
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq     %rsp, %rbp
    .cfi_def_cfa_register 6
    leaq     f(%rip), %rdi
    call     f
    movl     $0, %eax
    popq     %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc

```

通过优化编译 `gcc -S tmp.c -O2 tmp.s` 得到以下汇编代码（部分主要代码）：

```

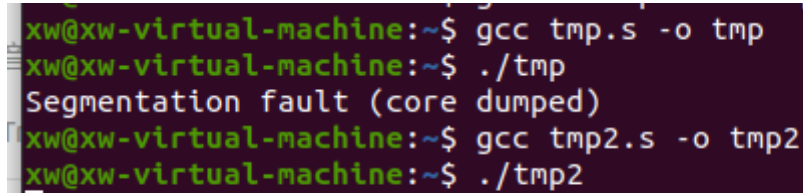
    .file    "tmp.c"
    .text
    .p2align 4
    .globl   f
    .type    f, @function
f:
.LFB23:
    .cfi_startproc
    endbr64
    xorl     %eax, %eax
    jmp     *%rdi
    .cfi_endproc
.LFE23:
    .size    f, .-f
    .section .text.startup,"ax",@progbits
    .p2align 4
    .globl   main
    .type    main, @function
main:
.LFB24:
    .cfi_startproc
    endbr64
    subq     $8, %rsp
    .cfi_def_cfa_offset 16
    leaq     f(%rip), %rdi

```

```
xorl    %eax, %eax
call    f
xorl    %eax, %eax
addq    $8, %rsp
.cfi_def_cfa_offset 8
ret
.cfi_endproc
```

可以看出，编译优化了后，在 `f` 函数中没有为形参 `g` 开辟内存，而是通过寄存器进行相应操作，且优化后用 `jmp` 处理函数尾递归，避免了函数调用的代价。

通过 `gcc tmp.s -o tmp` 和 `./tmp` 命令后，未优化的由于函数不同递归调用导致栈溢出，出现 `segmentation fault`，而做了优化的由于用 `jmp` 处理函数尾递归，避免了栈溢出，就一直没有显示出结果。



```
xw@xw-virtual-machine:~$ gcc tmp.s -o tmp
xw@xw-virtual-machine:~$ ./tmp
Segmentation fault (core dumped)
xw@xw-virtual-machine:~$ gcc tmp2.s -o tmp2
xw@xw-virtual-machine:~$ ./tmp2
```