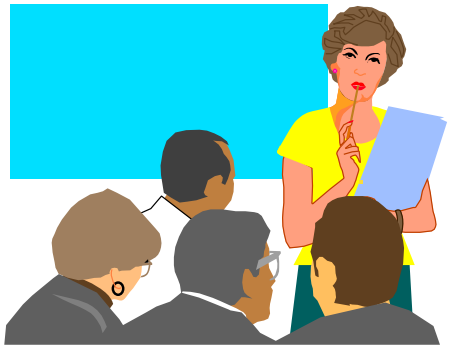
编译技术



胡春明
hucm@buaa.edu.cn

2019.9-2019.12

编译过程

编译过程是指将高级语言程序翻译为等价的目标程序的过程。

习惯上是将编译过程划分为5个基本阶段：

第十四章 代码优化

- 概述
- 优化的基本方法和例子
- 基本块和流图
- 基本块内的优化
- 全局优化

代码优化的基本任务

概述

代码优化（code optimization)

指编译程序为了生成高质量的目标程序而做的各种加工和处理。

目的：提高目标代码运行效率
- 时间效率（减少运行时间）
- 空间效率（减少内存容量）
- 能耗使用？（如在手机上）

原则：进行优化必须严格遵循“不能改变原有程序语义”原则。

为什么要进行代码优化?

翻译引入结构性的冗余

在从高级特性向低级特性翻译时，会引入一些冗余动作。这些动作，可以在优化中被合并、共享或删除

论程序员的个人修养

程序员很难照顾到各种细节，书写的程序可能存在冗余、低效的因素。

为什么要进行代码优化?

用开发时的开销代替运行时开销

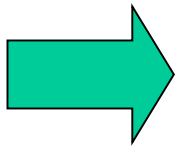
- 大型计算程序运行时间长（数十分钟，甚至小时、天级），为优化即使付出些代价是值得的
- 简单小程序（占机器内存，运行速度均可接受），或在程序的**调试阶段**，优化不那么必要

循环：程序中的“8-2原则”

- 循环往往占用大量计算时间
- 为减少循环执行时间所进行的优化对减少整个程序运行时间有很大的意义

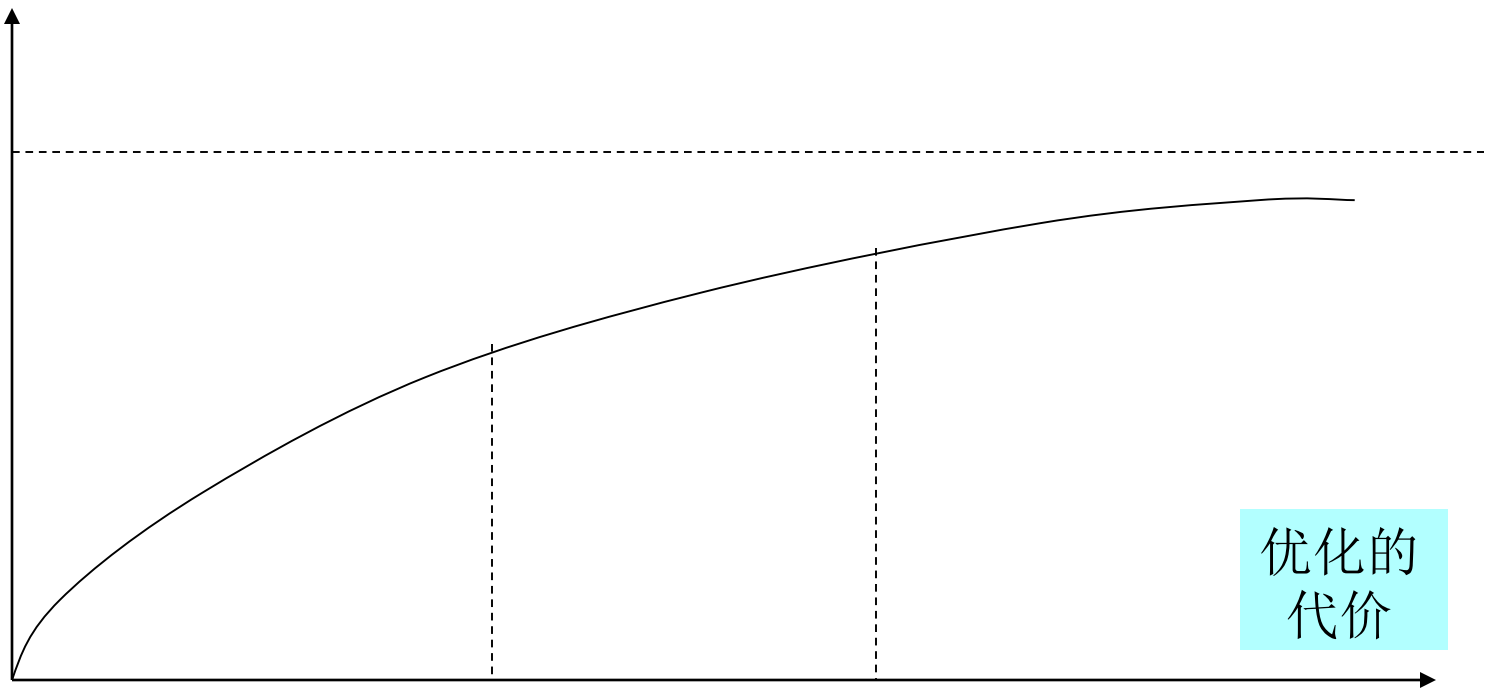
下面的优化合理吗?

```
int foo(int a)
{
   return a+5050;
}
```



```
int foo(int a)
{
   int count = 0 ;
   for(int i=1; i<=100; i++){
      count += i ;
   }
   return a + count;
}
```

优化所花费的代价和优化产生的效果



只要做些简单的处理，便能得到明显的优化效果

若要进一步提高优化效果，就要逐步付出更大的代价。

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

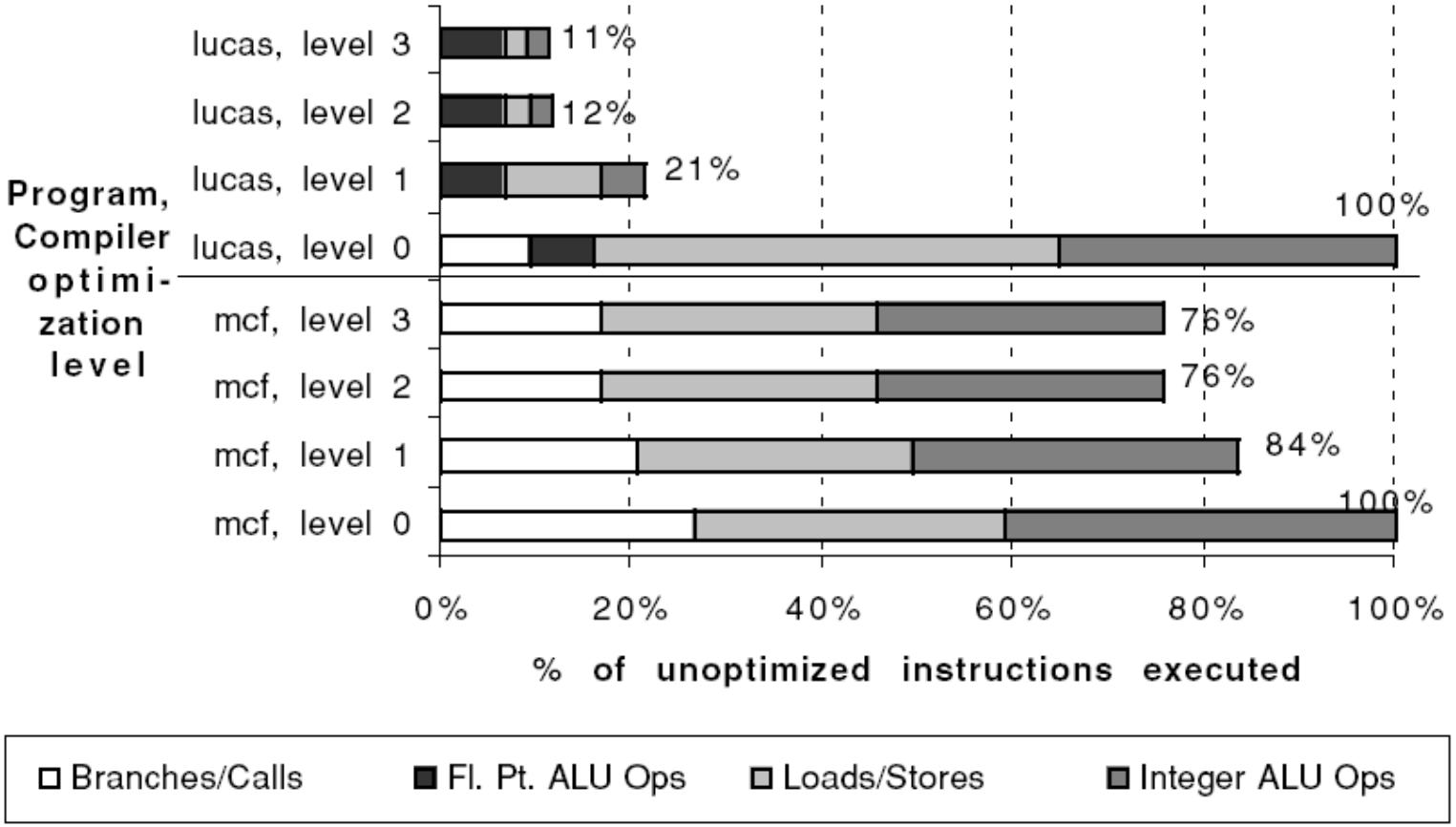
Source: Stanford CS143 (2012)

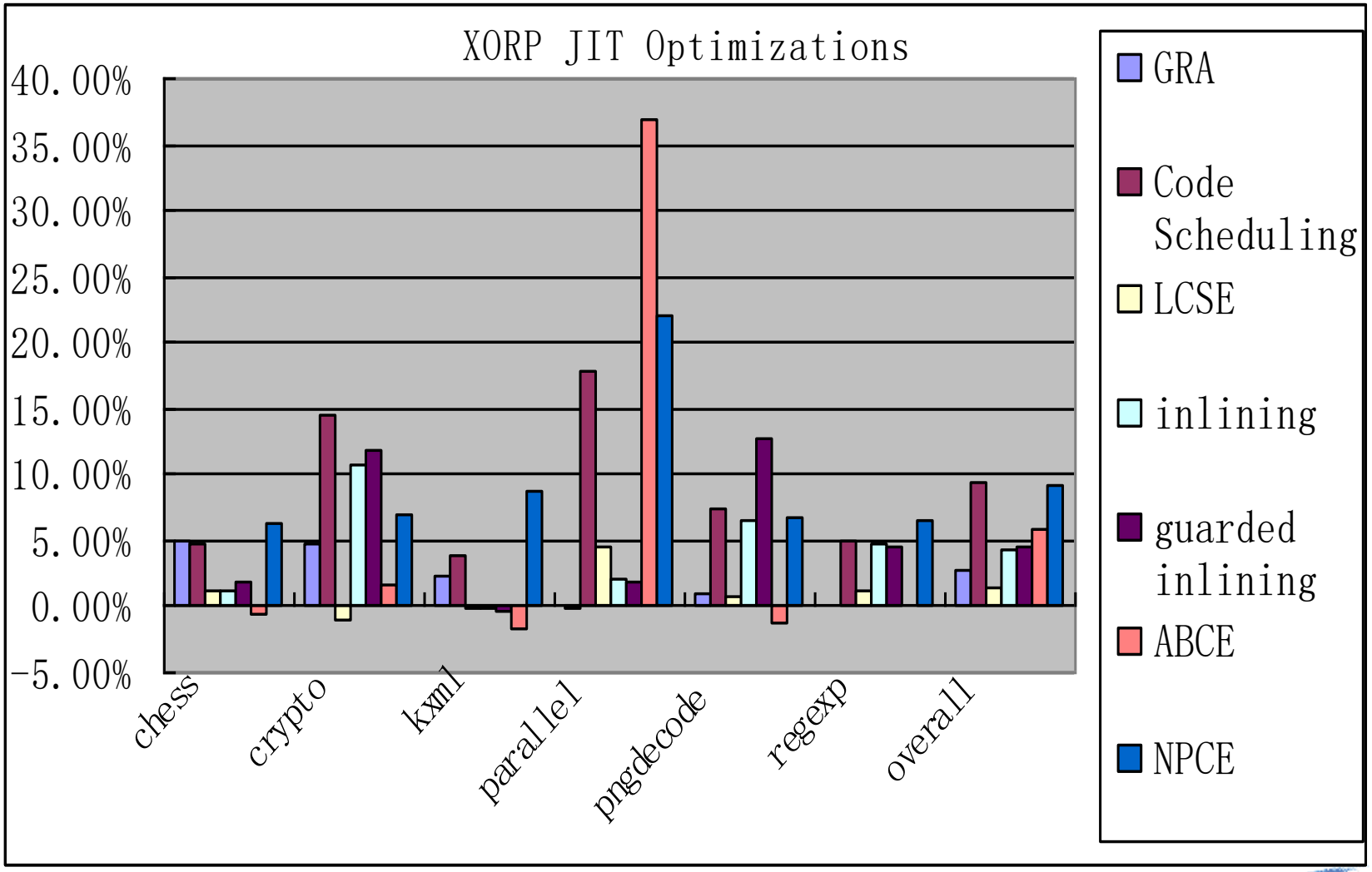
```
while (x < y + z) {
    x = x - y;
}
```

```
_L0:
    _t0 = y + z;
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

Source: Stanford CS143 (2012)

针对 SPEC2000中 *lucas* 和 *mcf* 施加不同级别的编译优化后的运行结果（编译器Alpha Compiler）





全局寄存器分配

指令调度

局部子表达式删除

代码内联

有保护的代码内联

消除数组边界检查

消除空指针检查

优化方法的分类1：

- **与机器无关的优化技术：即与目标机无关的优化，通常是在中间代码上进行的优化。**
  - **如：**数据流分析，常量传播，公共子表达式删除，死代码删除，循环交换，代码内联等等

- **与机器相关的优化技术：充分利用系统资源，（指令系统，寄存器资源）。**
  - 面向超标量超流水线架构、VLIW或者EPIC架构的指令调度方法；面向SMP架构的同步负载优化方法；面向SIMD、MIMD或者SPMD架构的数据级并行优化方法等
  - **特点：仅在特定体系结构下有效**

优化方法的分类2：

- **局部优化技术**
  - 指在基本块内进行的优化
  - 例如，局部公共子表达式删除
- **全局优化技术**
  - 函数/过程内进行的优化
  - 跨越基本块
  - 例如，全局数据流分析
- **跨函数优化技术**
  - 整个程序
  - 例如，跨函数别名分析，逃逸分析 等

基本块、流图

```
void foo(int* a, int* b)
{
  int prod = 0 ;
  int i ;

  for(i = 1;i<=20;i++){
   prod=prod+a[4*i]*b[4*i];
  }
  …
}
```

```
(1)   prod := 0
(2)   i := 1
(3)   i <= 20?
(4)   ifz goto (15)
(5)   t1 := 4 * i
(6)   t2 := a [ t1 ]
(7)   t3 := 4 * i
(8)   t4 := b [ t3 ]
(9)   t5 := t2 * t4
(10)  t6 := prod + t5
(11)  prod := t6
(12)  t7 := i + 1
(13)  i := t7
(14)  goto (3)
(15)  …
```

基本块定义

基本块中的代码是**连续的**语句序列

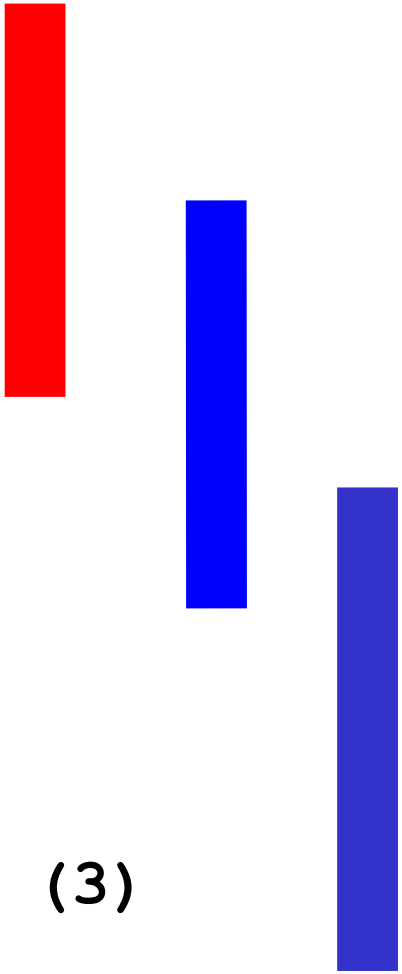
程序的执行（控制流）只能从基本块的第一条语句进入

程序的执行只能从基本块的最后一条语句离开

```
(1)   prod := 0
(2)   i := 1
(3)   i <= 20?
(4)   ifz goto (15)
(5)   t1 := 4 * i
(6)   t2 := a [ t1 ]
(7)   t3 := 4 * i
(8)   t4 := b [ t3 ]
(9)   t5 := t2 * t4
(10)  t6 := prod + t5
(11)  prod := t6
(12)  t7 := i + 1
(13)  i := t7
(14)  goto (3)
(15)  …
```

基本块的例子：划分基本块

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if i <= 20 goto (3)
(13)  …
```



下列语句序列，哪些属于同一个基本块，哪些不属于？

（1）~（6）

（3）~（8）

（7）~（13）

算法14.1　划分基本块

- **输入**：四元式序列
- **输出**：基本块列表，每个四元式仅出现在一个基本块中
- **方法**：
  - 1、**首先确定入口语句（每个基本块的第一条语句）的集合**
    - 规则1：整个语句序列的第一条语句属于入口语句
    - 规则2：任何能由条件/无条件跳转语句转移到的第一条语句属于入口语句
    - 规则3：紧跟在跳转语句之后的第一条语句属于入口语句
  - 2、**每个入口语句直到下一个入口语句，或者程序结束，它们之间的所有语句都属于同一个基本块**

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if i <= 20 goto (3)
(13)  …
```

- 1、首先确定入口语句（每个基本块的第一条语句）的集合
  - 1.1 整个语句序列的第一条语句属于入口语句

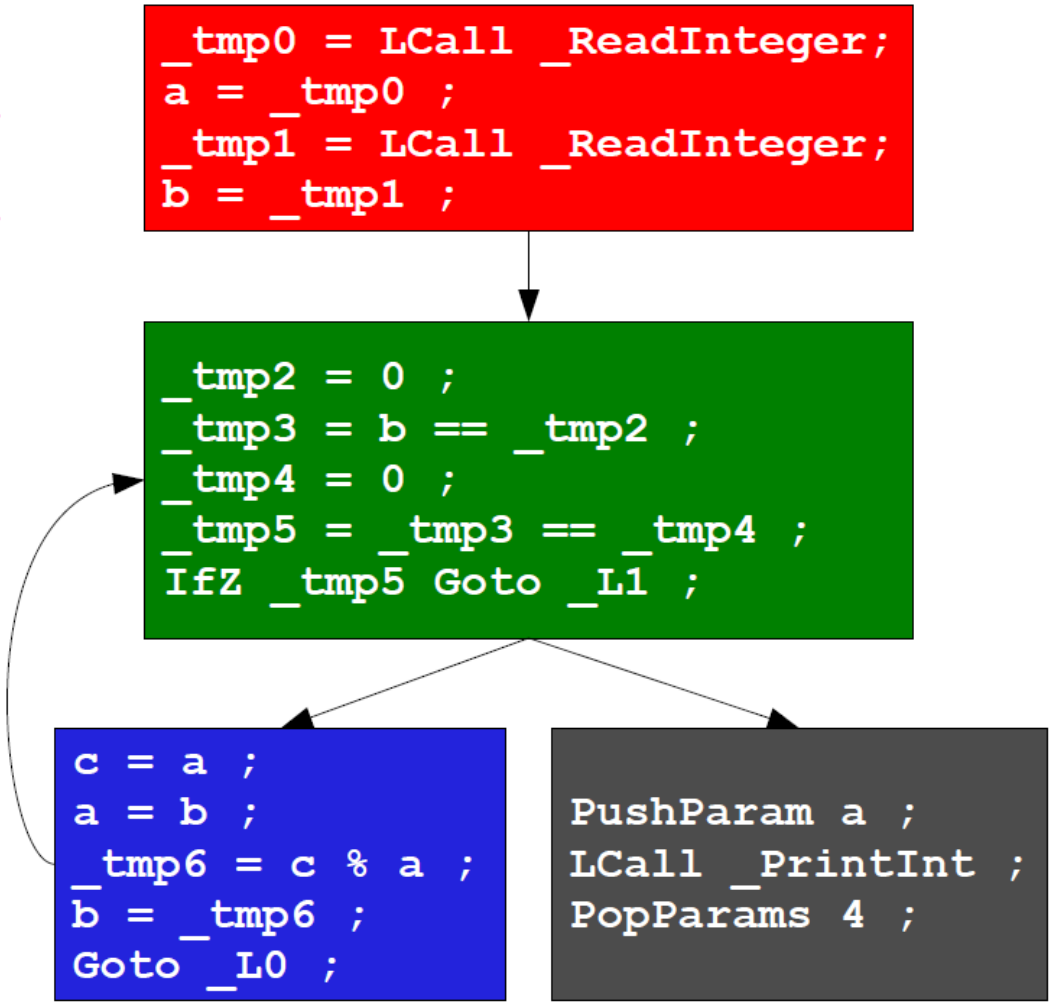
    （1）
  - 1.2 任何能由条件/无条件跳转语句转移到的第一条语句属于入口语句

    （3）
  - 1.3 紧跟在跳转语句之后的第一条语句属于入口语句

    （13）
- 2、每个入口语句直到下一个入口语句，或者程序结束，之间的所有语句都属于同一个基本块
- 基本块：
  - （1）~（2）
  - （3）~（12）
  - （13）~…

```
main:
    BeginFunc 40;
    _tmp0 = LCall _ReadInteger;
    a = _tmp0;
    _tmp1 = LCall _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```

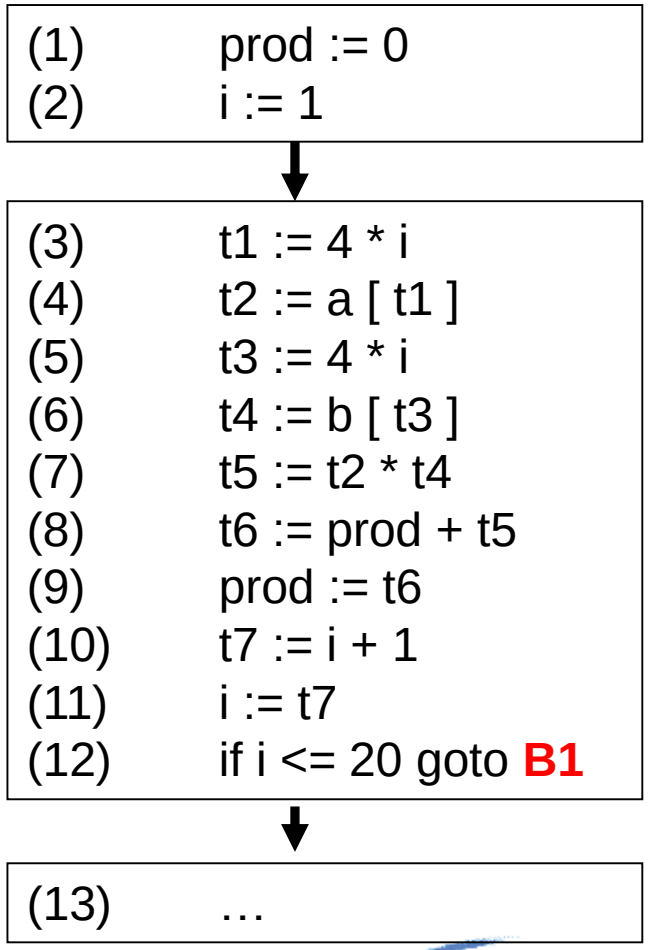
```
main:
    BeginFunc 40;
    _tmp0 = LCall _ReadInteger;
    a = _tmp0;
    _tmp1 = LCall _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```



Source: Stanford CS143 (2012)

基本块中的代码是连续的语句序列
程序的执行（控制流）只能从基本块的第一条语句进入
程序的执行只能从基本块的最后一条语句离开

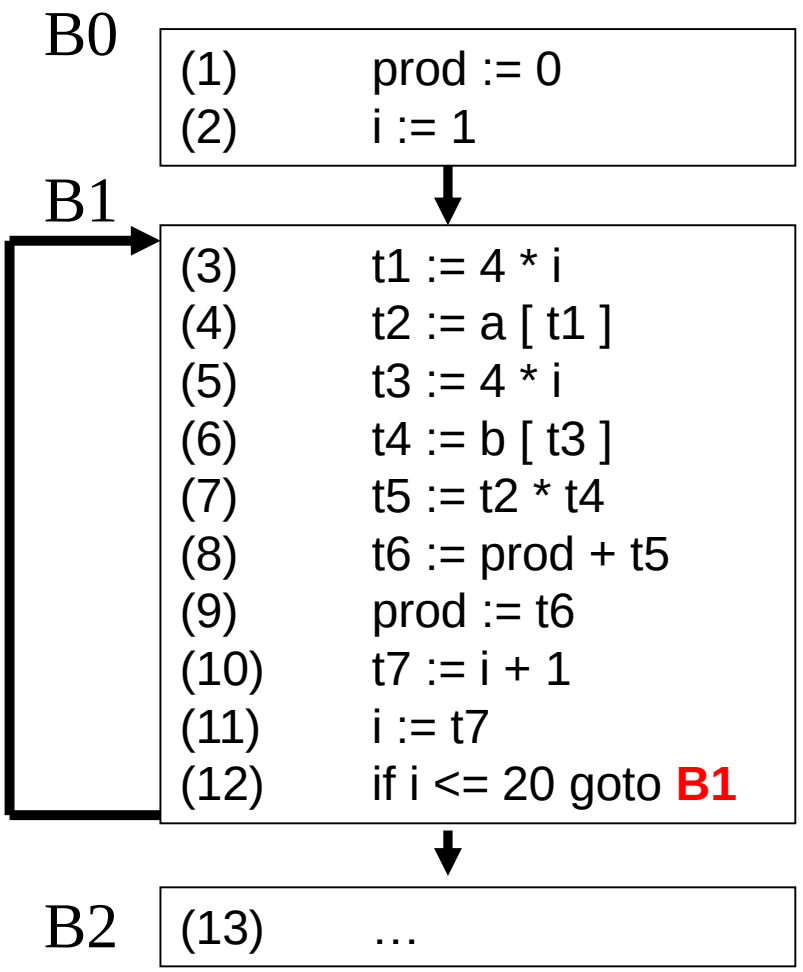
```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if i <= 20 goto (3)
(13)  …
```



流图

- 流图是一种有向图
- 流图的节点是基本块
- 如果在某个执行序列中，B2的执行紧跟在B1之后，则从B1到B2有一条有向边
- 我们称B1为B2的*前驱*，B2为B1的*后继*
  - 从B1的最后一条语句有条件或者无条件转移到B2的第一条语句；或者
  - 按照程序的执行次序，B2紧跟在B1之后，并且B1没有无条件转移到其他基本块

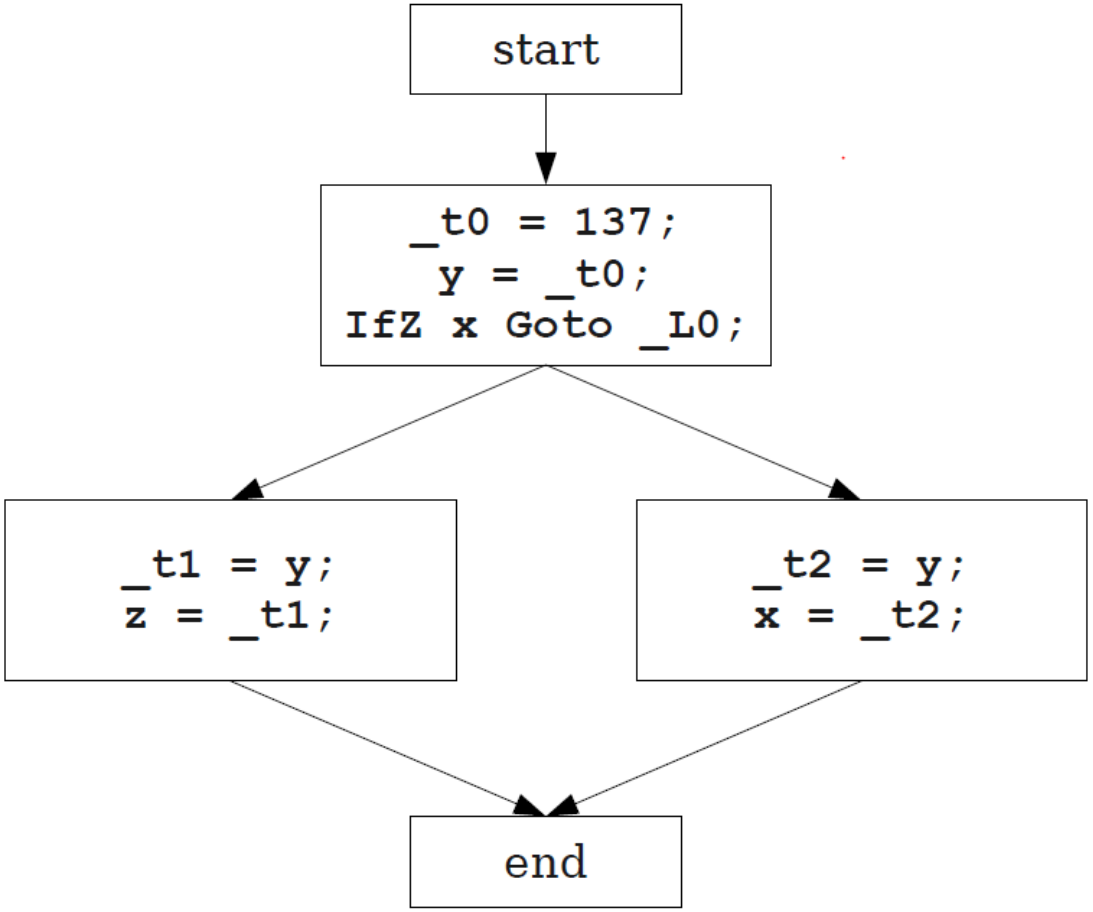
```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12) if i <= 20 goto (3)
(13)  …
```



- 编译器按照“程序—流图----基本块—中间代码”，选择合理的数据结构组织和管理中间代码。

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



局部优化（基本块内的优化）

基本块内优化

（1）利用代数性质（代数变换）

- 编译时完成常量表达式的计算，整数类型与实型的转换。

例： a := 5+6+x → a := 11+x （常数合并）

PI=3.141592

TO-R=PI/180.0 → TO-R = 0.0174644

又如： 设x为实型，x := 3+1 可变换成x := 4.0

- 下标变量引用时，其地址计算的一部分工作可在编译时预先做好（运行时只需计算“可变部分”即可）。

- **运算强度削弱：** 用一种需要较少执行时间的运算代替另一种运算，以减少运行时的运算强度时、空开销）

如 `x**2` → `x*x`

`3*x` → `x+x+x`

`8*x` , `4*x` 等换成左移运算

`x/2` , `x/16` 等换成右移运算

`x:=x+1` 变为 `INC x` 指令

`x/5` → `x*0.2` 等

利用机器硬件所提供的一些功能，如左移，右移操作，利用它们做乘法或除法，具有更高的代码效率。

(2) 常数合并和传播

如 x:=y 这样的赋值语句称为复写语句。由于 x 和 y 值相同，所以当满足一定条件时，在该赋值语句下面出现的 x 可用 y 来代替。

例如：

```
x:=y ;          x:=y ;
u:=2*x;    →    u:=2*y ;
v:=x+1;         v:=y+1  ;
```

这就是所谓的**复写传播 (copy propagation)**

若以后的语句中不再用到x时，则上面的x:=y可删去。

若上例中不是x := y而是x := 3。则复写传播变成了常量传播，即

```
x := y;
u := 2*x;
v := x+1;
```

```
x := 3;
u := 2*x;
v := x+1;
```

u := 6;　　v := 4;

又如　　t_1 := y/z;　　x := t_1;

若这里t_1为暂时（中间）变量，以后不再使用，则可变换为

x := y/z;

此外常量传播，引起常量计算，如：

pi = 3.14159　　r = pi/180.0

此时：pi = 3.14159　　r = 0.0174644　　（常量计算）

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```


Source: Stanford CS143 (2012)

(3) 删除冗余代码

冗余代码就是毫无实际意义的代码，又称死代码(dead code)或无用代码(useless code)。

例如： x := x + 0 ; x := x * 1 ; 等

又例： FLAG := TRUE

IF FLAG THEN…

…

ELSE…

} FLAG永真

另外在程序中为了调试常有如下：

if debug then … 的语句。

但当debug为false时，then后面的语句便永远不会执行，这就是可删去的冗余代码。

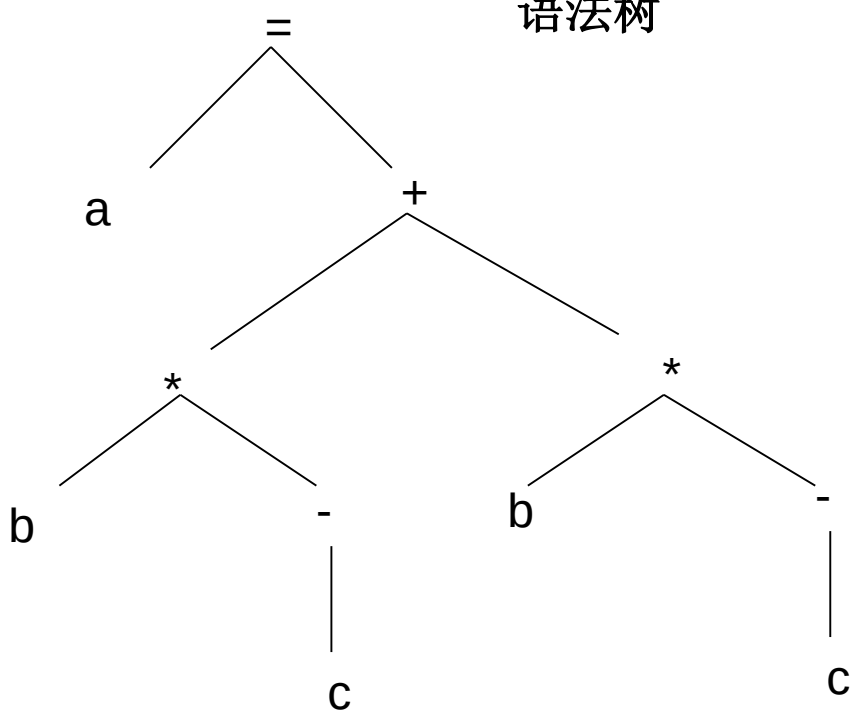
(可用条件编译 #if DEBUG 编写程序，而源代码中还应留着)

基本块内优化: 消除公共子表达式

- 赋值语句: a=b*(-c) + b*(-c)

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```

语法树



```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

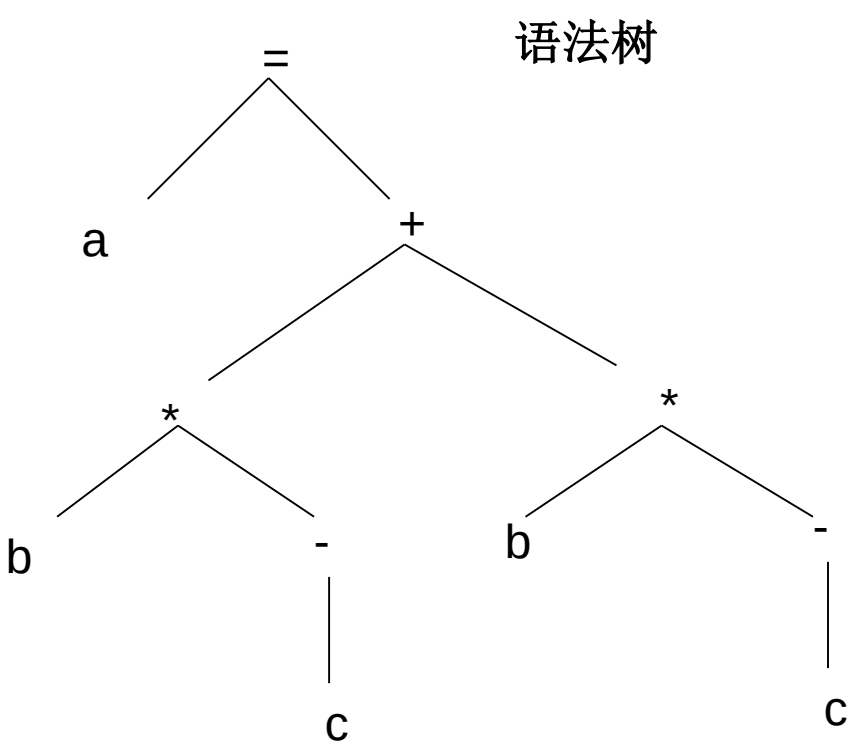
```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

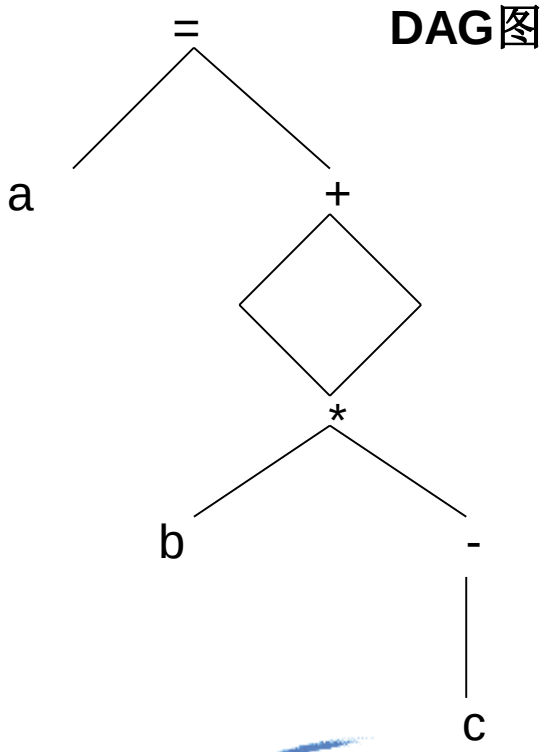
基本块内优化: 消除公共子表达式

- DAG图:
  - Directed Acyclic Graph 有向无环图
  - 用来表示基本块内各中间代码之间的关系
- 可通过DAG图消除公共子表达式

基本块的DAG图表示

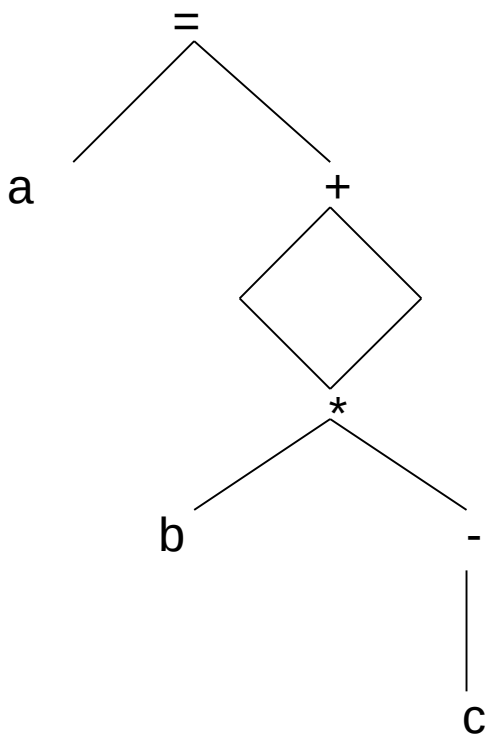
- 赋值语句： a=b*(-c) + b*(-c)





基本块DAG图的定义

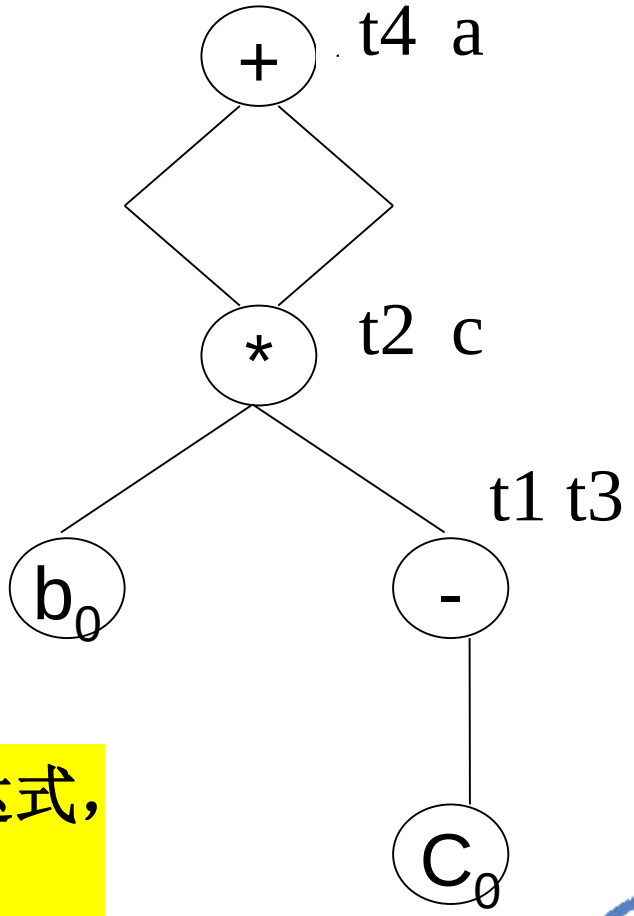
DAG图



- 图的叶节点由变量名或常量所标记。对于那些在基本块内先引用再赋值的变量，可以采用变量名加下标0的方式命名其初值。
- 图的中间节点由中间代码的操作符所标记，代表着基本块中一条或多条中间代码。
- 基本块中变量的最终计算结果，都对应着图中的一个节点；具有初值的变量，其初值和最终值可以分别对应不同的节点。

DAG表示

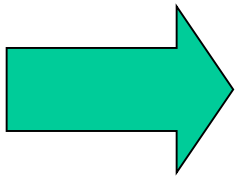
```
(1)  t1 = - c
(2)  t2 = b * t1
(3)  t3 = - c
(4)  c = b * t3
(5)  t4 = t2 + c
(6)  a = t4
```



通过DAG图可消除公共子表达式，得到更简洁的优化代码

消除局部公共子表达式

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```

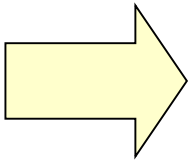


```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t2
   (t4)
a := t5
```

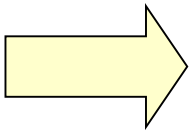
c := c + 1 ?

消除局部公共子表达式

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```



DAG图



```
t1 := - c
t2 := b * t1
t5 := t2 + t2
a := t5
```

需要两个算法：
1、DAG图的生成算法
2、从DAG图导出代码的算法

构建DAG图的算法-消除公共子表达式

- **输入**：基本块内的中间代码序列
- **输出**：完成局部公共子表达式删除后的DAG图
- **方法：**

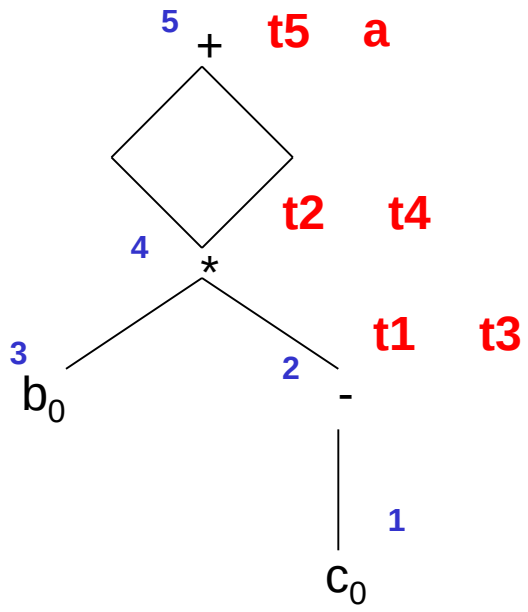
1. 首先建立节点表，该表记录了变量名和常量值，以及它们当前所对应的DAG图中节点的序号。该表初始状态为空。
2. 从第一条中间代码开始，按照以下规则建立DAG图。
3. 对于形如z = x op y的中间代码，其中z为记录计算结果的变量名，x为左操作数，y为右操作数，op为操作符：首先在节点表中寻找x，如果找到，记录下x当前所对应的节点号i；如果未找到，在DAG图中新建一个叶节点，假设其节点号仍为i，标记为x（如x为变量名，该标记更改为x_0）；在节点表中增加新的一项(x, i)，表明二者之间的对应关系。右操作数y与x同理，假设其对应节点号为j。

构建DAG图的算法-消除公共子表达式

4. 在DAG图中寻找中间节点，其标记为op，且其左操作数节点号为i，右操作数节点号为j。如果找到，记录下其节点号k；如果未找到，在DAG图中新建一个中间节点，假设其节点号仍为k，并将节点i和j分别与k相连，作为其左子节点和右子节点；
5. 在节点表中寻找z，如果找到，将z所对应的节点号更改为k；如果未找到，在节点表中新建一项(z, k)，表明二者之间的对应关系。
6. 对输入的中间代码序列依次重复上述步骤3～5。

建立DAG图，例1 a=b*(-c) + b*(-c)

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```

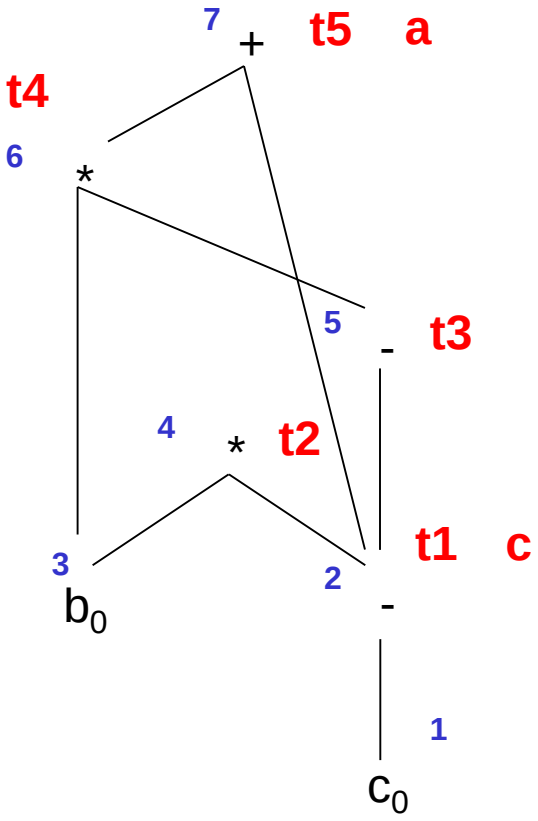


node(x)

c	1
t1	2
b	3
t2	4
t3	2
t4	4
t5	5
a	5

建立DAG图，例2

```
t1 := - c
t2 := b * t1
c := t1
t3 := - c
t4 := b * t3
t5 := c + t4
a := t5
```



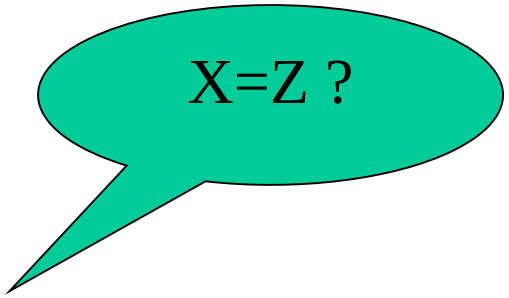
node(x)

c	1
t1	2
b	3
t2	4
t3	5
t4	6
t5	7
a	7

数组、指针及函数调用的DAG图

- 当中间代码序列中出现了数组成员、指针或函数调用时，算法11.2需要作出一定的调整，否则将得出不正确的优化结果 。

例：
x = a[i]
a[j] = y
z = a[i]



不一定。
如果 **j=i**

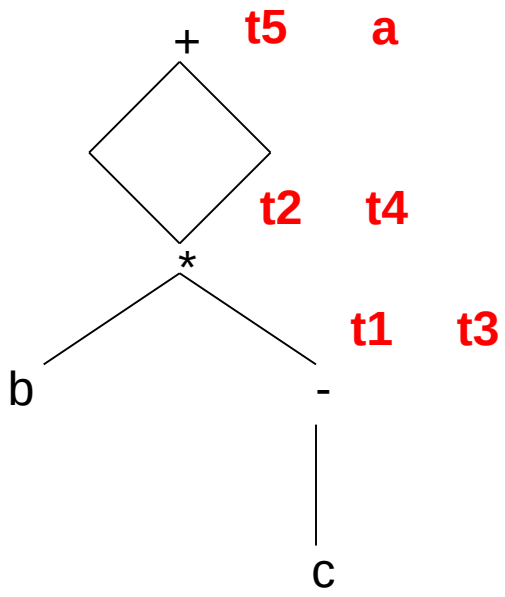
将数组变量a作为一个单独的变量进行考虑，将形如x = a[i]的中间代码都表示为x = a [] i，其中[]为数组取值操作符；形如a[j] = y的中间代码都表示为a = j “[]=” y，其中“[]=”为数组成员赋值操作符。

指针：保守处理

```
x = *p
*q = y
z = *p
```

- 函数调用
  - 在缺乏跨函数数据流分析的支持下，需要保守地假设函数调用改变了所有它可能改变的数据，

从DAG图重新导出中间代码

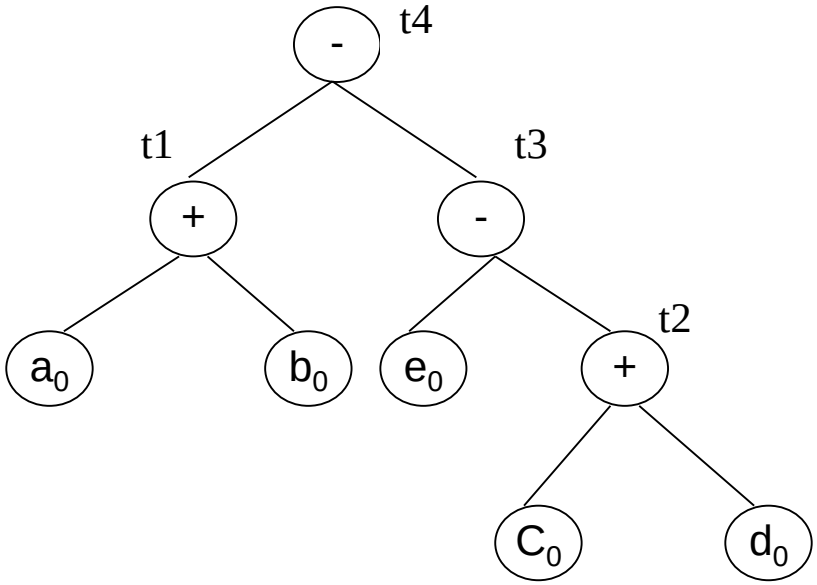



```
t1 := -c

t2 := b * t1

a := t2 + t2
```

从DAG图重新导出中间代码




```
(1)  t1 = a + b
     t2 = c + d
     t3 = e - t2
     t4 = t1 - t3
```

```
(2)  t2 = c + d
     t3 = e - t2
     t1 = a + b
     t4 = t1 - t3
```

```
(1) t1 = a + b
    t2 = c + d
    t3 = e - t2
    t4 = t1 - t3
```

假设：

- 局部变量a, b, c, d, e均不占用寄存器
- 仅有两个寄存器eax, edx 可供使用

```
mov eax, a     ; t1 = a + b
add eax, b
mov edx, c     ; t2 = c + d
add edx, d
mov [ESP+08H], eax  ; t3 = e - t2
mov eax, e
sub eax, edx
mov [ESP+0CH], edx; t4 = t1 - t3
mov edx, [ESP+08H]
sub edx, eax
```

- ***[ESP+08H],*** **[ESP+0CH] 均为临时变量在运行栈上的临时保存单元地址**

(1)
```
t1 = a + b
t2 = c + d
t3 = e - t2
t4 = t1 - t3
```

```
mov eax, a   ; t1 = a + b
add eax, b
mov edx, c   ; t2 = c + d
add edx, d
mov [ESP+08H], eax
                ; t3=e-t2
mov eax, e
sub eax, edx
mov [ESP+0CH], edx
                ; t4=t1-t3
mov edx, [ESP+08H]
sub edx, eax
```

(2)
```
t2 = c + d
t3 = e - t2
t1 = a + b
t4 = t1 - t3
```

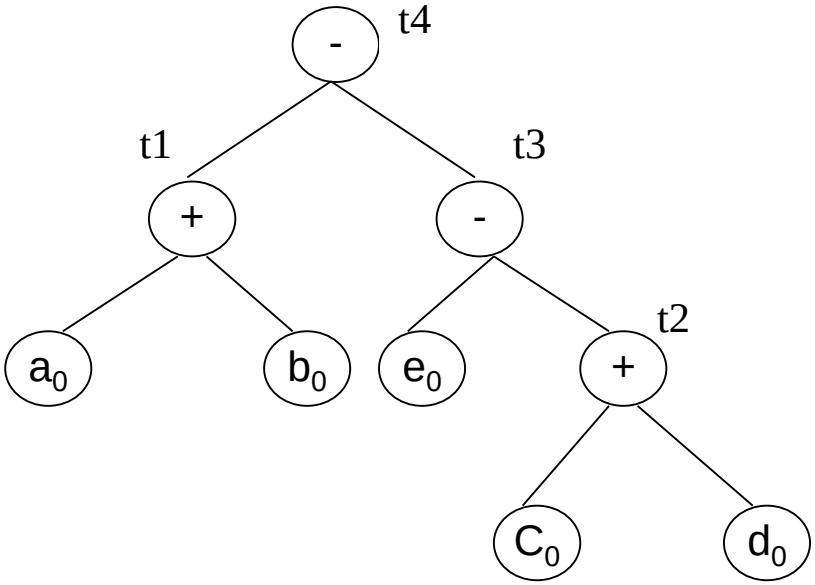
```
mov eax, c   ; t2 = c + d
add eax, d
mov edx, e   ; t3 = e - t2
sub edx, eax
mov [ESP+0CH], eax
                ; t1 = a + b
mov eax, a
add eax, b
mov [ESP+08H], eax
                ; t4= t1 - t3
sub eax, edx
```

从DAG导出中间代码的启发式算法

- **输入：** DAG图
- **输出：** 中间代码序列
- **方法：**

1. 初始化一个放置DAG图中间结点的队列。
2. 如果DAG图中还有中间节点未进入队列，则执行步骤3，否则执行步骤5
3. 选取一个尚未进入队列，但其**所有父节点均已进入队列**的中间节点n，将其加入队列；或选取**没有父节点**的中间节点，将其加入队列
4. 如果n的**最左子节点**符合步骤3的条件，将其加入队列；并沿着当前节点的最左边，**循环访问其最左子节点**，最左子节点的最左子节点等，将符合步骤3条件的中间节点依次加入队列；如果出现不符合步骤3条件的最左子节点，执行步骤2
5. 将中间节点队列**逆序输出**，便得到中间节点的计算顺序，将其整理成中间代码序列

算法14.3 从DAG导出中间代码的启发式算法



1、初始化一个放置DAG图中间节点的队列

2、如果DAG图中还有中间节点未进入队列，则执行步骤3，否则执行步骤5。

3、选取一个尚未进入队列，但其**所有父节点均已进入队列**的中间节点n，将其加入队列；或选取**没有父节点的中间节点**，将其加入队列

4、如果n的最左子节点符合步骤3的条件，将其加入队列；并沿着当前节点的最左边，循环访问其**最左子节点**，最左子节点的最左子节点等，将符合步骤3条件的中间节点依次加入队列；如果出现不符合步骤3条件的最左子节点，执行步骤2。

5、将中间节点队列逆序输出，便得到中间节点的计算顺序，将其整理成中间代码序列

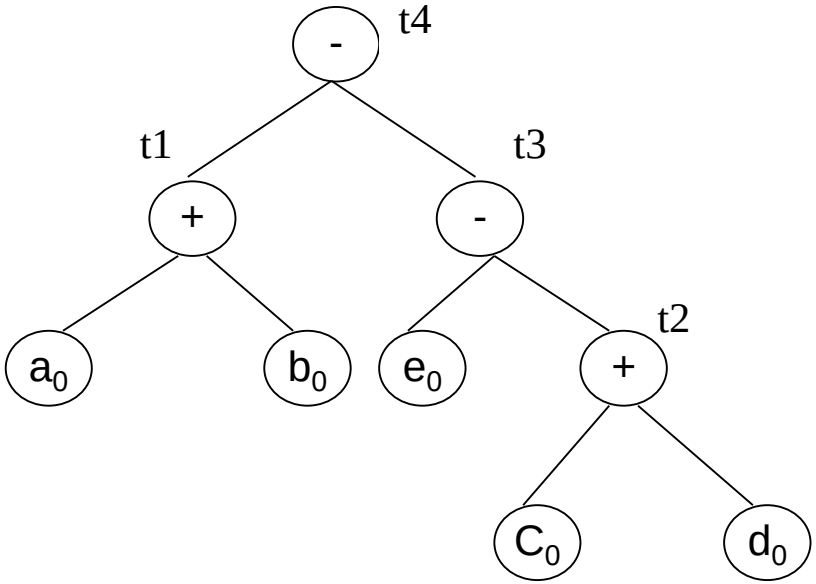
中间节点队列：

t4	t1	t3	t2

中间节点队列：

t4	t1	t3	t2

5、将中间节点队列逆序输出，便得到中间节点的计算顺序，将其整理成中间代码序列




```
t2 = c + d
t3 = e - t2
t1 = a + b
t4 = t1 - t3
```

窥孔优化

- 窥孔优化关注在目标指令的一个较短的序列上，通常称其为“窥孔”
- 通过删除其中的冗余代码，或者用更高效简洁的新代码来替代其中的部[illegible]码，达到提升目标代码质量的目的

但是窥孔优化并不局限在同一个基本块中。以下方法，如利用代数性质等都可以用。

```
mov EAX, [ESP+0[illegible]
mov [ESP+08H], EAX
```

```
B2: ...
```

作业： 14章 1-3题

谢谢！