第八章 代码生成

《编译原理》

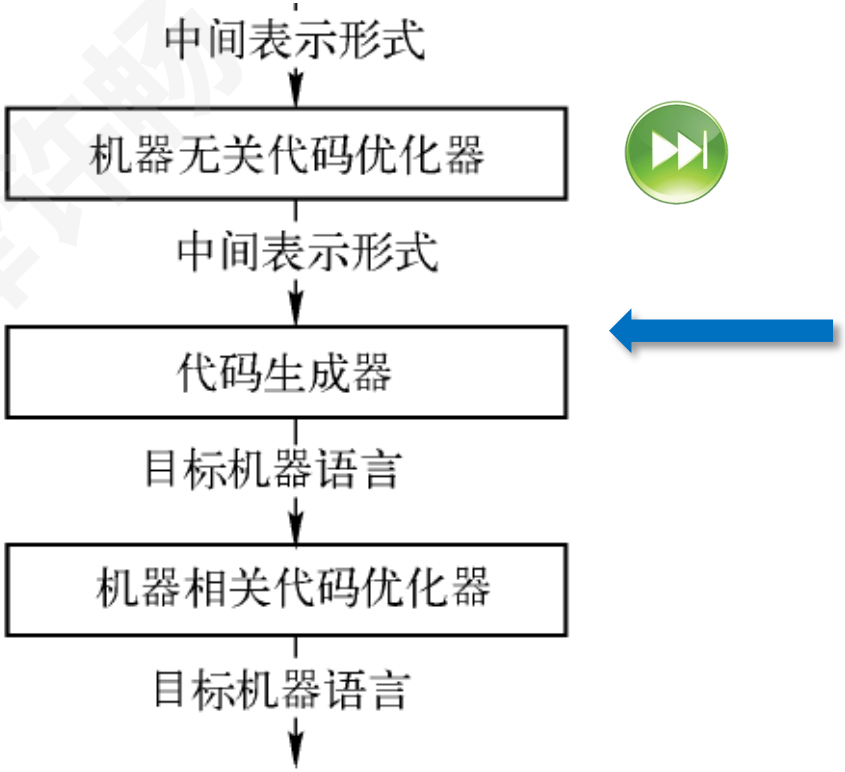
谭添

南京大学计算机系

2022年春季

闯关进度


字符流
词法分析器
符号流
语法分析
语法树
语义分析
语法树
中间代码生成器


中间表示形式
机器无关代码优化器
中间表示形式
代码生成器
目标机器语言
机器相关代码优化器
目标机器语言

主要内容

- 代码生成器的设计
- 目标语言
- 目标代码中的地址
- 基本块和流图
- 基本块优化
- 代码生成器
- 寄存器分配

代码生成器的位置

- 根据中间表示 (IR) 生成代码
- 代码生成器之前可能有一个优化组件
- 代码生成器的三个任务
  - **指令选择**：选择适当的指令实现IR语句
  - **寄存器分配和指派**：把哪个值放在哪个寄存器中
  - 指令排序：按照什么顺序安排指令执行

图 8-1　代码生成器的位置

要解决的问题

- 正确性：正确的机器指令
- 易于实现、测试和维护

- 输入IR的选择
  - 四元式、三元式、字节代码、堆栈机代码、后缀表示、抽象语法树、DAG图、…
- 输出
  - RISC、CISC
  - 可重定向代码、汇编语言

目标机模型

- 使用三地址机器的模型

- 与三地址码的关键区别：寄存器
  - 位于CPU内部，用于存放数据的小型高速存储区域
  - 几乎所有CPU计算都需要寄存器参与 (存放参数/结果)
  - 几乎所有参与计算的程序数据都位于内存中 (使用外存数据也须先读入内存)
  - 机器码需要频繁地在寄存器与内存之间搬运数据

目标机模型

- 指令
  - 加载：LD *dst*, *addr* (把地址*addr*中的内容加载到*dst*所指的寄存器)
  - 保存：ST *x*, *r* (把寄存器*r*中的内容保存到*x*中)
  - 计算：<*OP*> *dst*, src_1, src_2 (把src_1和src_2中的值运算后将结果存放到*dst*中)
  - 无条件跳转：BR *L* (控制流转向标号*L*的指令)
  - 条件跳转：B<*cond*> *r*, *L* (对*r*中的值进行测试，如果为真则转向*L*)

寻址模式

- 变量x：指向分配x的内存位置
- $a(r)$：地址是a的左值加上寄存器r中的值　　$a+r$
- constant(r)：寄存器r中内容加上前面的常数即其地址　　constant + r
- *r：寄存器r的内容所表示的位置上存放的内容位置　　解引用r
- *constant(r)：寄存器r中内容加上常量所代表的位置上的内容所表示的位置　　解引用(constant + r)
- 常量#constant

例子 (1)

- x = y – z
  - LD R1, y // R1 = y
  - LD R2, z // R2 = z
  - SUB R1, R1, R2 // R1 = R1 – R2
  - ST x, R1 // x = R1

例子 (2)

- b = a[i]
  - LD R1, i // R1 = i
  - MUL R1, R1, 8 // R1 = R1 * 8 (8字节长元素)
  - LD R2, a(R1) // R2 = content(a + content(R1))
  - ST b, R2 // b = R2
- a[j] = c
  - LD R1, c // R1 = c
  - LD R2, j // R2 = j
  - MUL R2, R2, 8 // R2 = R2 * 8 (8字节长元素)
  - ST a(R2), R1 // content(a + content(R2)) = R1

例子 (3)

- x = *p
  - LD R1, p // R1 = p
  - LD R2, 0(R1) // R2 = content(0 + content(R1))
  - ST x, R2 // x = R2

- *p = y
  - LD R1, p // R1 = p
  - LD R2, y // R2 = y
  - ST 0(R1), R2 // content(0 + content(R1)) = R2

例子 (4)

- if x < y goto L
  - LD R1, x // R1 = x
  - LD R2, y // R2 = y
  - SUB R1, R1, R2 // R1 = R1 – R2
  - BLTZ R1, M // if R1 < 0 jump to M

程序及指令的代价

- 不同的目的有不同的度量
  - 最短编译时间、运行时间、目标程序大小、能耗
- 不可判定一个目标程序是否最优

- 假设每个指令有固定的代价，设定为1加上运算分量寻址模式的代价
  - LD R0, R1：代价为1
  - LD R0, M：代价是2
  - LD R1, *100(R2)：代价为2

目标代码中的地址

- 如何为过程调用和返回生成代码?
  - 静态分配 (活动记录)
  - 栈式分配 (活动记录)

- 如何将IR中的名字 (过程名或变量名) 转换成为目标代码中的地址?
  - 不同区域中的名字采用不同的寻址方式

活动记录的栈式分配

- 寄存器SP指向栈顶活动记录起始处
- 第一个过程 (main) 初始化栈区
- 过程调用指令序列
  - ADD SP, SP, #*caller*.*recordSize*　　// 增大栈指针
  - ST 0(SP), #*here* + 16　　// 保存返回地址
  - BR *callee*.*codeArea*　　// 转移到被调用者
- 返回指令序列
  - BR *0(SP)　　// 被调用者执行，返回调用者
  - SUB SP, SP, #*caller*.*recordSize*　// 调用者减小栈指针

例子

```
                          // m的代码
100:  LD SP, #600         // 初始化栈
108:  ACTION1             // action1的代码
128:  ADD SP, SP, #msize  // 调用指令序列的开始
136:  ST *SP, #152        // 将返回地址压入栈
144:  BR 300              // 调用q
152:  SUB SP, SP, #msize  // 恢复SP的值
160:  ACTION12
180:  HALT
      ...
                          // p的代码
200:  ACTION3
220:  BR *0(SP)           // 返回
      ...
                          // q的代码
300:  ACTION4             // 包含有跳转到456
320:  ADD SP, SP, #qsize
328:  ST *SP, #344        // 将返回地址压入栈
336:  BR 200              // 调用p
344:  SUB SP, SP, #qsize
352:  ACTION5
```

图 8-6　栈式分配时的目标代码

m调用q，q调用p

- m: main
- q: quicksort
- p: partition

```
372:  ADD SP, SP, #qsize
380:  BR *SP, #396        // 将返回地址压入栈
388:  BR 300              // 调用q
396:  SUB SP, SP, #qsize
404:  ACTION6
424:  ADD SP, SP, #qsize
432:  ST *SP, #440        // 将返回地址压入栈
440:  BR 300              // 调用q
448:  SUB SP, SP, #qsize
456:  BR *0(SP)           // 返回
      ...
600:                      // 栈区的开始处
```

名字的运行时刻地址

- 在三地址语句中使用名字 (实际上是指向符号表条目) 来引用变量

- 语句x = 0
  - 如果x分配在静态区域，且静态区开始位置为*static*
    - static[12] = 0　　LD 112, #0 // *static* = 100
  - 如果x分配在栈区，且相对地址为12，则
    - LD 12(SP), #0

代码生成器

- 根据三地址指令序列生成机器指令
  - 假设每个三地址指令只有一个对应的机器指令
  - 有一组寄存器用于计算基本块内部的值
- 主要的目标是减少加载 (LD) 和保存 (ST) 指令，即最大限度地利用寄存器
- 寄存器的使用方法
  - 执行运算时，运算分量必须放在寄存器中
  - 存放临时变量
  - 存放全局的值
  - 进行运行时刻管理 (比如栈顶指针)

算法的基本思想和数据结构

- 依次考虑各三地址指令，尽可能把值保留在寄存器中，以减少寄存器/内存之间的数据交换
- 为一个三地址指令生成机器指令时
  - 只有当运算分量不在寄存器中时，才从内存载入
  - 尽量保证只有当寄存器中值不被使用 (称之为不活跃) 时，才覆盖掉
- 数据结构 (编译期)
  - **寄存器描述符**：跟踪各个寄存器都存放了哪些变量的当前值

    $R_1 \rightarrow \{x\}, R_2 \rightarrow \{a, b\}$
  - **地址描述符**：各个变量的当前值存放在哪些位置 (包括内存位置和寄存器) 上

    $x \rightarrow \{R_1\}, a \rightarrow \{a, R_2\}$

代码生成算法 (1)

- 重要子函数：*getReg*(*I*)
  - 根据寄存器描述符和地址描述符等数据流信息，为三地址指令*I*选择最佳的寄存器
  - 得到的机器指令的质量依赖于*getReg*函数选取寄存器的算法
- 代码生成算法逐个处理三地址指令

代码生成算法 (2)

- 运算语句：$x = y + z$
  - $getReg(x = y + z)$为x, y, z选择寄存器R_x, R_y, R_z
  - 检查R_y的寄存器描述符，如果y不在R_y中则生成指令
    - LD R_y, y' // y'表示存放y值的当前位置
    - 类似地确定是否生成LD R_z, z'
  - 生成指令ADD R_x, R_y, R_z
- 复制语句：$x = y$
  - $getReg(x = y)$为x和y选择相同的寄存器 (运行后值相同)
  - 如果y不在R_y中，则生成指令LD R_y, y
- 基本块的收尾
  - 如果变量x活跃，且不在内存中，则生成指令ST x, R_x

如果$R_i \rightarrow \{\ldots y \ldots\}$，选择$R_i$作为$R_y$

代码生成算法 (3)

- 代码生成同时更新寄存器和地址描述符

1. 处理指令时生成的LD R, x
   - R的寄存器描述符：只包含x
   - x的地址描述符：R作为新位置加入到x的位置集合中
   - 从任何不同于x的变量的地址描述符中删除R
2. 生成的ST x, R
   - x的地址描述符：包含自己的内存位置 (新增)

代码生成算法 (4)

3. ADD R_x, R_y, R_z
   - R_x的寄存器描述符：只包含x
   - x的地址描述符：只包含R_x (不包含x的内存位置)
   - 从任何不同于x的变量的地址描述符中删除R_x

4. 处理$x = y$时
   - 如果生成LD R_y, y，按照规则1处理
   - 把x加入到R_y的寄存器描述符中 (即R_y同时存放了x和y的当前值)
   - x的地址描述符：只包含R_y (不包含x的内存位置)

例子 (1)

- a、b、c、d在5)后仍活跃
- t、u、v是局部临时变量

1) t = a – b
2) u = a – c
3) v = t + u
4) a = d
5) d = v + u

例子 (2)

寄存器描述符

地址描述符

```
t = a - b
     LD R1, a
     LD R2, b
     SUB R2, R1, R2

u = a - c
     LD R3, c
     SUB R1, R1, R3

v = t + u
     ADD R3, R2, R1
```

R1	R2	R3
a	t	
u	t	c
u	t	v

a	b	c	d	t	u	v
a	b	c	d			
a, R1	b	c	d	R2		
a	b	c, R3	d	R2	R1	
a	b	c	d	R2	R1	R3

例子 (3)

寄存器描述符　　地址描述符

```
a = d
     LD R2, d
```

R1	R2	R3
u	a, d	v

a	b	c	d	t	u	v
R2	b	c	d, R2		R1	R3

```
d = v + u
     ADD R1, R3, R1
```

R1	R2	R3
d	a	v

a	b	c	d	t	u	v
R2	b	c	R1			R3

```
exit
     ST a, R2
     ST d, R1
```

R1	R2	R3
d	a	v

a	b	c	d	t	u	v
a, R2	b	c	d, R1			R3

*getReg*函数 (1)

- 目标：减少LD/ST指令
- 任务：为运算分量和结果分配寄存器

- 为$x = y\ op\ z$的运算分量y和z分配寄存器
  - 如果y已经在某个寄存器中，不需要进行处理，选择这个寄存器作为R_y
  - 如果y不在寄存器中，且有空闲寄存器，选择一个空闲寄存器作为R_y
  - 如果不在寄存器中，且没有空闲寄存器?

*getReg*函数 (2)

- 如果不在寄存器中，且没有空闲寄存器?
  - 需要选择一个非空闲寄存器存放值
- 若选择寄存器R，且已知其寄存器描述符表示某变量v的值在R中，则
  - 如果v的地址描述符表明可在别的地方找到v，DONE
  - v就是x (即结果)，且x不是运算分量z，DONE $x = y\ op\ z$
  - 如果v在此之后不会被使用 (不活跃)，DONE
  - 生成保存指令ST v, R (溢出操作) 并修改v的地址描述符；如果R中存放了多个变量的值，那么需要生成多条ST指令

*getReg*函数 (3)

- 为$x = y\ op\ z$的结果x选择寄存器R_x的方法基本上和上面要把y从内存LD时一样，但是
  - 只存放x值的寄存器总是可接受的
  - 如果y在指令之后不再使用，且R_y仅仅保存了y的值，那么R_y同时也可以作为R_x (对z也一样)

- 处理$x = y$时
  - 先选择R_y
  - 然后让$R_x = R_y$

如何判断一个变量是否活跃?

- 如何判断一个变量的值之后是否还会被使用

  1) t = a – b
  2) u = a – c
  3) **v** = t + **u**
  4) a = d
  5) d = **v** + **u**

- 需要沿着代码执行路径向“前”看

后续使用信息

- 变量值的使用
  - 三地址语句i向变量x赋值，如果另一个语句j的运算分量为x，且从i开始有一条路径到达j，且路径上没有对x赋值，那么j就使用了i处计算得到的x的值
  - 我们说变量x在语句i后的程序点上**活跃**
    - 程序执行完语句i时，x中存放的值将被后面的语句使用
    - **不活跃**是指变量的值不会被使用，而不是变量不会被使用
- 这些信息可以用于代码生成
  - 如果x在i处不活跃，且x占用了一个寄存器，我们可以把这个寄存器用于其它目的

基本块 (Basic block)

- 对每个过程内的指令进行划分
- 必然连续执行的指令划入一组，称为基本块
  - 控制流只能从基本块的第一条指令进入
  - 除基本块的最后一条指令外，控制流不会跳转/停机
- 每个基本块内只有一条执行路径
  - 易于分析

划分基本块的算法

- 输入：三地址指令序列
- 输出：基本块的列表

- 方法 什么会“打断”指令的连续执行？跳转
  - 确定**首指令**leader (基本块的第一个指令)
    - 第一个三地址指令
    - 任意一个 (条件或无条件) 转移指令的目标指令
    - 紧跟在一个 (条件或无条件) 转移指令之后的指令
  - 确定基本块
    - 每个首指令对应于一个基本块：从首指令开始到下一个首指令

基本块划分的例子

- 第一个指令
  - 1
- 跳转指令的目标指令
  - 3, 2, 13
- 跳转指令的下一条指令
  - 10, 12
- 基本块
  - 1-1, 2-2, 3-9, 10-11, 12-12, 13-17

```
 1)  i = 1
 2)  j = 1
 3)  t1 = 10 * i
 4)  t2 = t1 + j
 5)  t3 = 8 * t2
 6)  t4 = t3 - 88
 7)  a[t4] = 0.0
 8)  j = j + 1
 9)  if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

确定基本块中的活跃性、后续使用

- 输入
  - 基本块B，开始时B中的所有非临时变量都是活跃的
- 输出
  - 各个语句i上变量的活跃性、后续使用信息
- 方法
  - 从B的最后一个语句开始反向扫描
  - 对于每个语句i：$x = y + z$
    - 令语句i和x、y、z的当前活跃性信息/使用信息关联
    - 设置x为“不活跃”和“无后续使用”
    - 设置y和z为“活跃”，并指明它们的下一次使用设置为语句i

例子

- i, j, a非临时变量 (出口处活跃)，其余变量不活跃
  - 8) i, j, a活跃，j在8上被使用
  - 7) i, j, a, t4活跃，a和t4被7使用
  - 6) i, j, a, t3活跃，t4不活跃，t3被6使用
  - 5) i, j, a, t2活跃，t4, t3不活跃，t2被5使用
  - 4) i, j, a, t1活跃，t4, t3, t2不活跃，t1和j被4使用
  - 3) i, j, a活跃，t4, t3, t2, t1不活跃，i被3使用

```
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
7)   a[t4] = 0.0
8)   j = j + 1
```

代码生成算法 (2)

1) t = a – b
2) u = a – c
3) v = t + u
4) a = d
5) d = v + u

- 基本块的收尾
  - 如果变量x活跃，且不在内存中，则生成指令ST x, R_x

```
a = d
    LD R2, d

d = v + u
    ADD R1, R3, R1

exit
    ST a, R2
    ST d, R1
```

R1	R2	R3	a	b	c	d	t	u	v
u	a, d	v	R2	b	c	d, R2		R1	R3
d	a	v	R2	b	c	R1			R3
d	a	v	a, R2	b	c	d, R1			R3

寄存器分配和指派

- **寄存器分配**
  - 确定在程序的每个点上，哪个值应该存放在寄存器中
- **寄存器指派**
  - 各个值应该存放在哪个寄存器中

- 简单方法：把特定类型的值分配给特定的寄存器
  - 数组基地址指派给一组寄存器，算术计算分配给一组寄存器，栈顶指针分配一个寄存器，循环，……
  - 缺点：寄存器的使用效率较低

全局寄存器分配

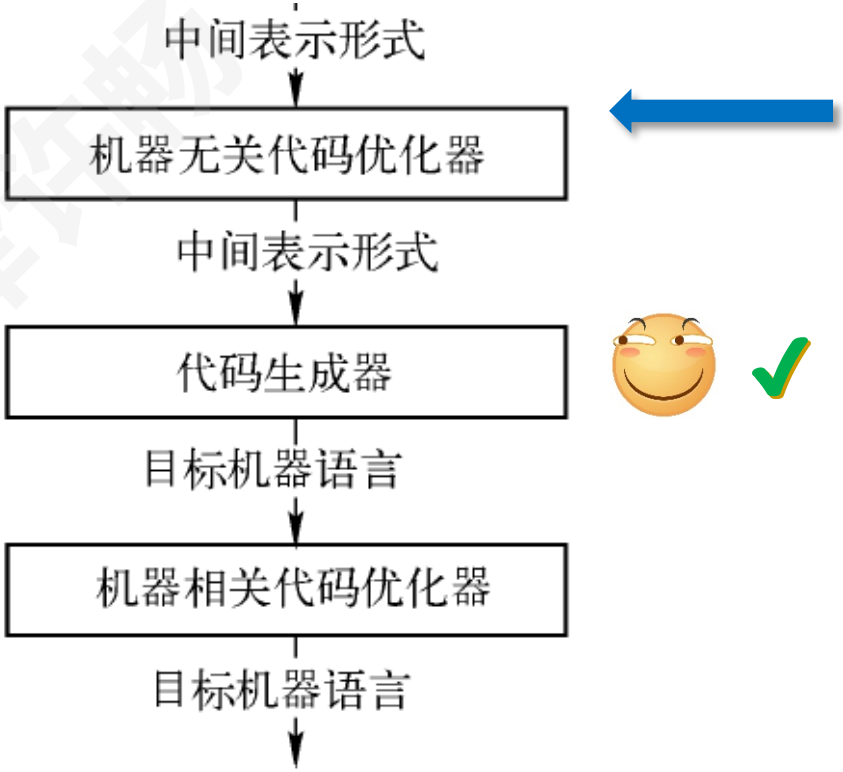
- 在循环中频繁使用的值存放在固定寄存器
  - 分配固定多个寄存器来存放内部循环中最活跃的值

- 可以通过使用计数的方法来估算把一个变量放到寄存器中会带来多大好处，然后根据这个估算来分配寄存器


勝ったッ！
第8章完！

闯关进度

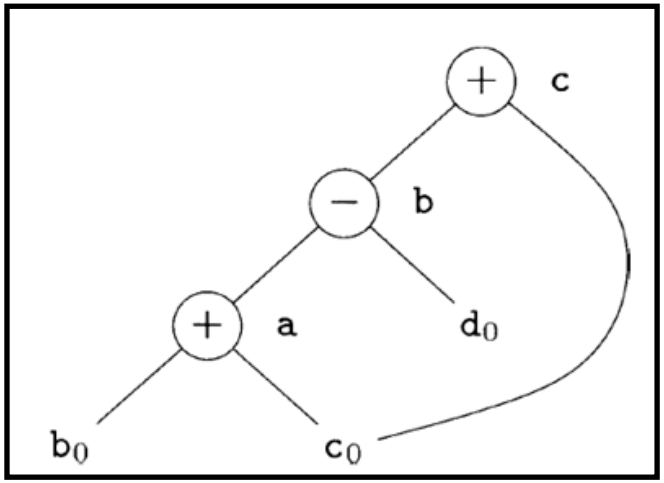

字符流
词法分析器
符号流
语法分析
语法树
语义分析
语法树
中间代码生成器


中间表示形式
机器无关代码优化器
中间表示形式
代码生成器
目标机器语言
机器相关代码优化器
目标机器语言

基本块的优化

- 针对基本块的优化可以有很好的效果 (局部优化)
- 许多局部优化技术需要先将基本块内的指令转化为有向无环图 (Directed Acyclic Graph, DAG)
- DAG可反映变量及其值对其他变量的依赖关系
  - 结点表示变量的值
  - 边表示计算值形成的依赖关系

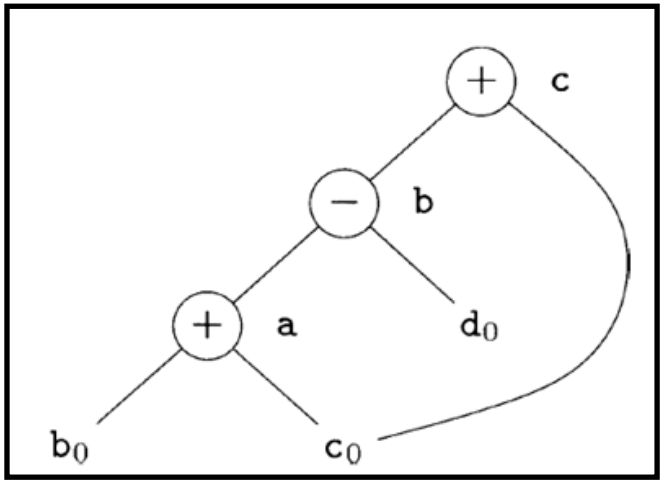
a = b + c
b = a – d
c = b + c



基本块的DAG构造

- 构造方法
  - 每个变量都有一个对应的DAG结点表示其初始值
  - 每个语句s有一个相关的结点N，代表此计算得到的值
    - N的子结点对应于(得到其运算分量当前值的)其它语句
    - N的标号是s中的运算符，同时还有一组变量被关联到N，表示s是最新对这些变量进行定值的语句

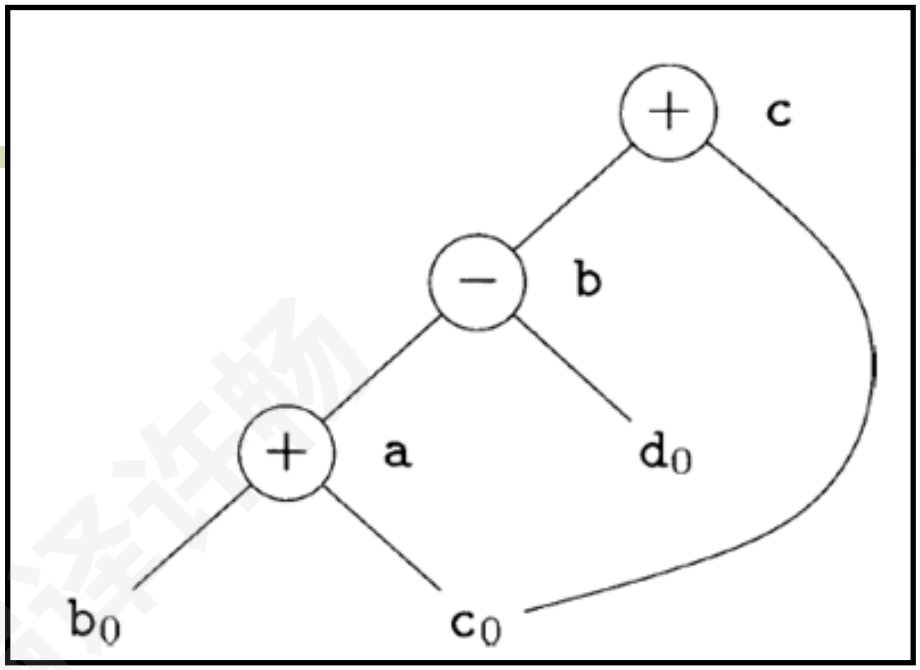
$$a = b + c$$
$$b = a - d$$
$$c = b + c$$



DAG图的构造

- 为基本块中出现的每个变量建立结点 (表示初始值)，各变量和相应结点关联
- 顺序扫描各三地址指令，进行如下处理
  - 指令$x = y\ op\ z$
    - 为该指令建立结点N，标号为op，令x和N关联
    - N的子结点为y、z当前关联的结点
  - 指令$x = y$
    - 假设y关联到N，那么x现在也关联到N
- 扫描结束后，对所有在出口处活跃的变量x，将x所关联的结点设置为**输出结点**

例子



- 指令序列
  - a = b + c
  - b = a – d
  - c = b + c
- 过程
  - 结点b_0、c_0和d_0对应于b、c和d的初始值
  - a = b + c：构造第一个加法结点，a与之关联
  - b = a – d：构造减法结点，b与之关联
  - c = b + c：构造第二个加法结点，c与之关联(注意第一个子结点对应于减法结点)

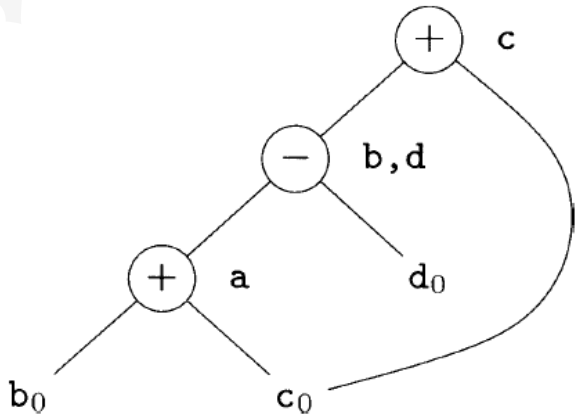
DAG的作用

- DAG图描述了基本块运行时各变量的值 (和初始值) 之间的关系

- 以DAG为基础，对代码进行转换
  - 寻找局部公共子表达式
  - 消除死代码
  - 代数恒等式的使用
  - 数组引用的表示
  - 指针赋值和过程调用

局部公共子表达式

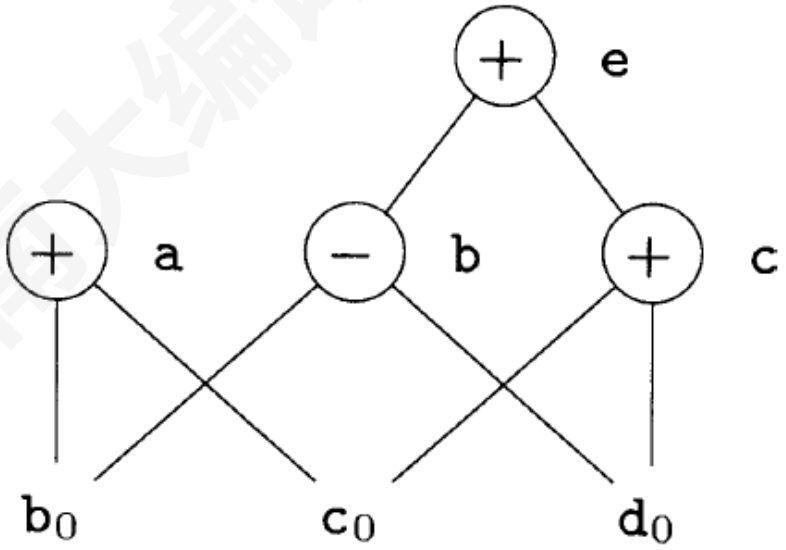
- 局部公共子表达式的发现
  - 建立某个结点M之前，检查是否存在一个结点N，它和M具有相同的运算符和子结点(顺序也相同)
  - 如果存在，则不需要生成新的结点，用N代表M
- 例如
  - a = b + c
  - b = a – d
  - c = b + c
  - d = a – d



- 注意：两个b + c不是公共子表达式(但a – d是)

消除死代码

- 在DAG图上消除没有附加活跃变量的根结点，即**消除死代码**
- 如果图中c、e不是活跃变量(但a、b是)，则可以删除标号为e、c的结点



基于代数恒等式的优化

- 消除计算步骤
  - $x + 0 = 0 + x = x$　　$x - 0 = x$
  - $x * 1 = 1 * x = x$　　$x / 1 = x$
- 降低计算强度
  - $x^2 = x * x$　　$2 * x = x + x$　　$x / 2 = x * 0.5$
- 常量折叠
  - 2 * 3.14可以用6.28替换

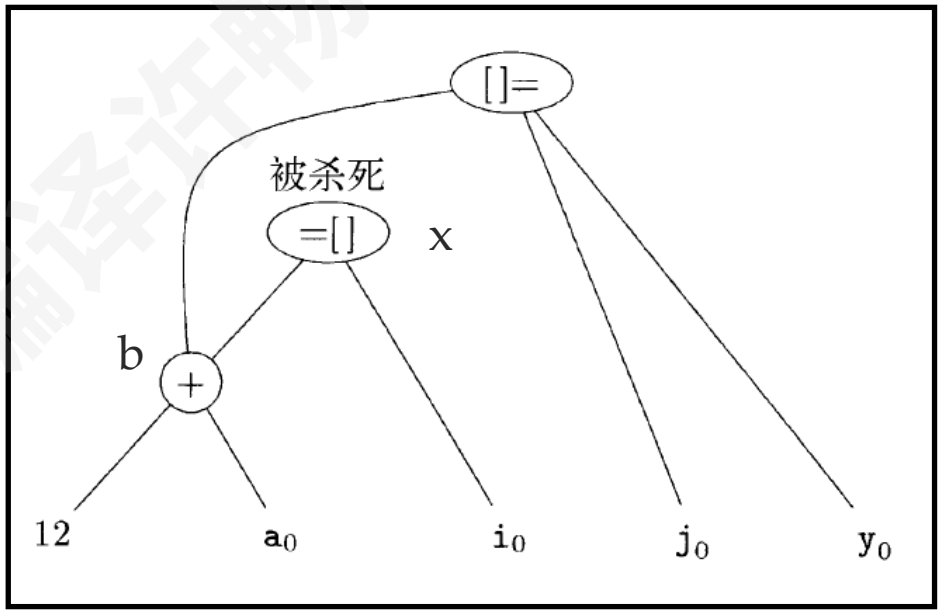
- 实现这些优化，只需在DAG图上寻找特定的模式

数组引用

- a[j]可能改变a[i]的值 (别名)，因此不能像普通运算符一样构造结点
  - x = a[i]　　a[j] = y　　z = a[i]
- **从数组取值**的运算x = a[i]对应于=[]的结点
  - 这个结点的左右子节点是数组初始值a_0和下标i
  - 变量x是这个结点的标号之一
- **对数组赋值**的运算a[j] = y对应于[]=的结点
  - 这个结点的三个子节点分别表示a_0、j和y
  - 杀死所有依赖于a_0的变量

数组引用DAG的例子

- 设a是数组，b是指针
  - b = 12 + a
  - x = b[i]
  - b[j] = y
- 一个结点被杀死，意味着它不能被复用
  - 考虑再有指令m = b[i]



指针赋值/过程调用

- 通过指针进行取值/赋值：x = *p、*q = y
  - x使用了任意变量，因此无法消除死代码
  - *q = y对任意变量赋值，因此杀死了全部其他结点
- 可通过 (全局/局部) 指针分析部分地解决这个问题

- 过程调用也类似，必须安全地假设它
  - 使用了可访问范围内的所有变量
  - 修改了可访问范围内的所有变量

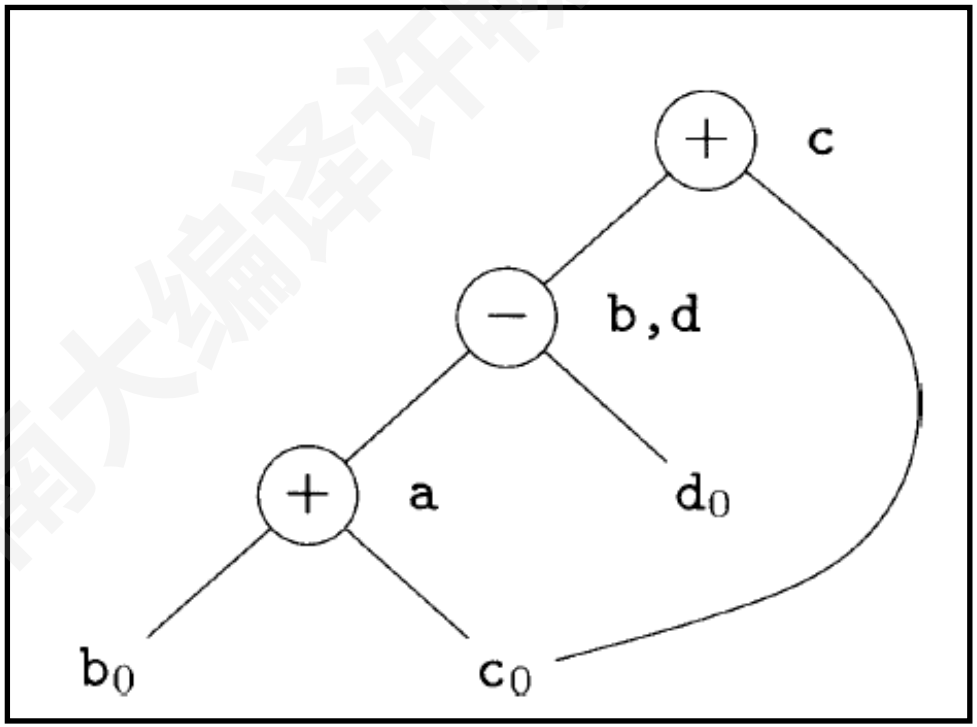
从DAG到基本块

- **重构**的方法
  - 每个结点构造一个三地址语句，计算对应的值
  - 结果应该尽量赋给一个活跃的变量
  - 如果结点有多个关联的变量，则需要用复制语句进行赋值

重组基本块的例子

- 根据DAG构造时结点产生的顺序
  - a = b + c
  - b = a – d
  - d = b
  - c = b + c

a = b + c
b = a – d
c = b + c
d = a – d



重组的规则

- 注意求值顺序
  - 指令顺序必须遵守DAG中结点的顺序
  - 对数组赋值 (write) 要跟在原来之前的赋值/求值之后
  - 对数组求值 (read) 要跟在原来之前的赋值指令之后
  - 对变量的使用必须跟在所有原来在它之前的过程调用和指针间接赋值之后
  - 任何过程调用或指针间接赋值必须跟在原来在它之前的变量求值之后
- 即保证
  - 如果两个指令之间相互影响，它们的顺序就不该改变

窥孔优化

- 使用一个滑动窗口 (**窥孔**) 来检查目标指令，在窥孔内实现优化
  - 冗余指令消除
  - 控制流优化
  - 代数化简/强度消减
  - 机器特有指令的使用

滑动窗口 (窥孔) 并无准确定义，可理解为只需关注**少量**相关指令即可完成的优化

冗余指令消除

- 多余的LD/ST指令
  - LD R_0, a
  - ST a, R_0　　// 非跳转目标；可删除

- 级联跳转代码
  - if debug == 1 goto L1; goto L2; L1: …; L2: …;
    - => if debug != 1 goto L2; L1: …; L2: …;
  - 如果已知debug一定是0，那么替换成为goto L2

控制流优化

- goto L1; … …; L1: goto L2
  - => goto L2; … …; L1: goto L2

- if a < b goto L1; … … ; L1: goto L2
  - => if a < b goto L2; … … ; L1: goto L2

代数化简/强度消减和机器特有指令

- 应用代数恒等式进行优化
  - 消除$x = x + 0, x = x * 1, \ldots$
  - 用$x * x$替换x^2
- 使用机器特有指令
  - INC, DEC, …

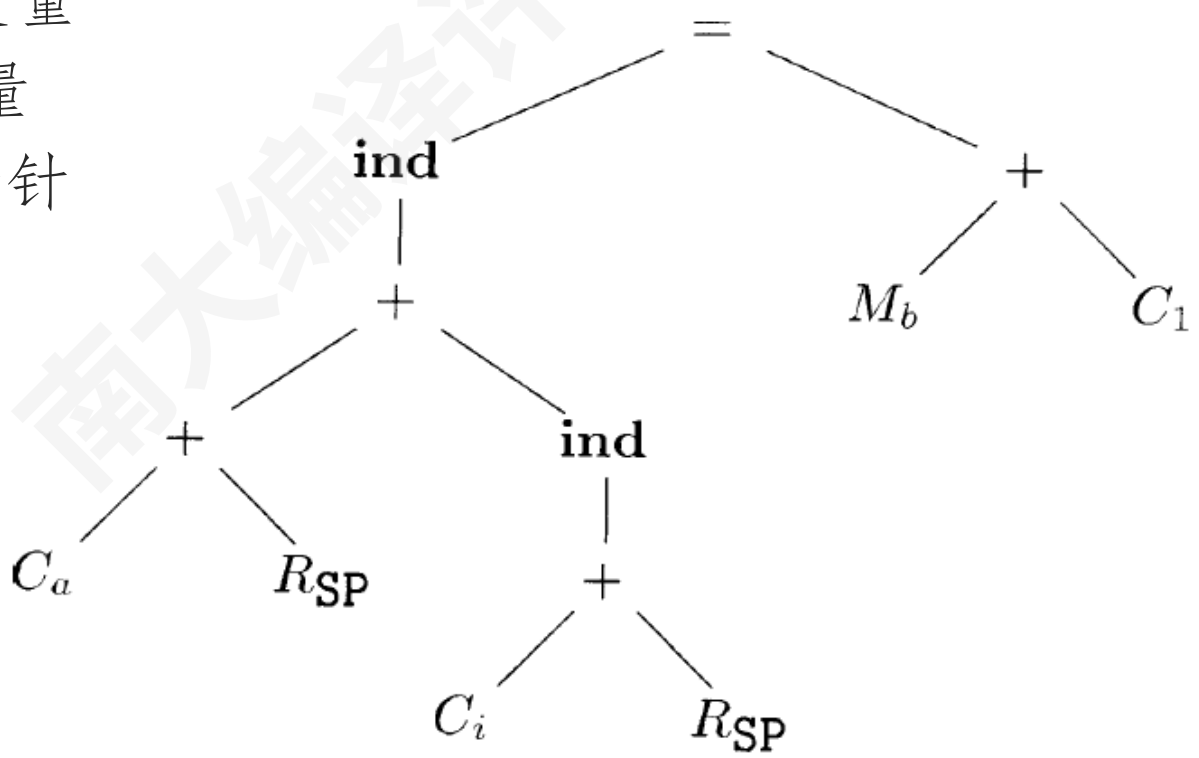
树重写实现指令选择

- 在某些机器上，同一个三地址指令可以使用多种机器指令实现，有时多个三地址指令可以使用一个机器指令实现

- **指令选择**
  - 为实现中间表示形式中出现的运算符选择适当的机器指令
  - 用树来表示中间代码，按照特定的规则不断覆盖这棵树并生成机器指令

例子

- a[i] = b + 1
  - ind：把参数作为内存地址
  - a, i：局部变量
  - b：全局变量
  - SP：栈顶指针



目标指令选择

- 通过应用一个树重写规则序列来生成
- 重写规则形式

$$replacement \leftarrow template\ \{action\}$$

其中，*replacement*（　替换结点）是一个结点，*template*（模板）是一棵树，*action*（动作）是一个像语法制导翻译方案中那样的代码片断。

- 一组树重写规则被称为一个**树翻译方案**
- 树重写规则示例

一些重写规则 (1)

1)	$R_i \leftarrow C_a$	{ LD Ri, #a }
2)	$R_i \leftarrow M_x$	{ LD Ri, x }
3)	$M \leftarrow =(M_x, R_i)$	{ ST x, Ri }
4)	$M \leftarrow =(\text{ind}(R_i), R_j)$	{ ST *Ri, Rj }
5)	$R_i \leftarrow \text{ind}(+(C_a, R_j))$	{ LD Ri, a(Rj) }

一些目标机器指令的树重写规则

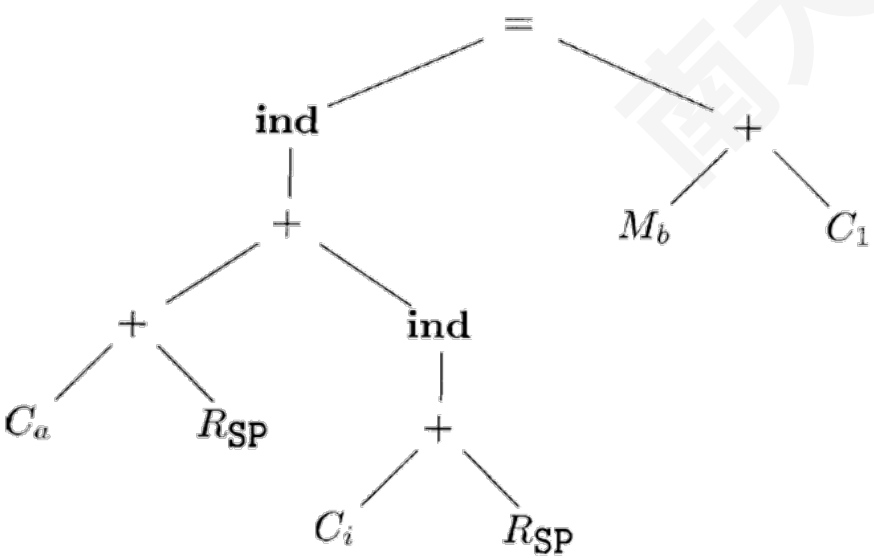
一些重写规则 (2)

6)	$R_i \leftarrow$ + (R_i, ind(+(C_a, R_j)))	{ ADD R*i*, R*i*, *a*(R*j*) }
7)	$R_i \leftarrow$ + (R_i, R_j)	{ ADD R*i*, R*i*, R*j* }
8)	$R_i \leftarrow$ + (R_i, C_1)	{ INC R*i* }

一些目标机器指令的树重写规则

覆盖重写过程

- 规则1)：{ LD R_0, #a }
- 规则7)：{ ADD R_0, R_0, SP }
- 规则6)：{ ADD R_0, R_0, i(SP) }
- 规则2)：{ LD R_1, b }
- 规则8)：{ INC R_1 }
- 规则4)：{ ST *R_0, R_1 }



1)	$R_i \leftarrow C_a$	{ LD Ri, #a }
2)	$R_i \leftarrow M_x$	{ LD Ri, x }
3)	$M \leftarrow$ = (M_x, R_i)	{ ST x, Ri }
4)	$M \leftarrow$ = (ind(R_i), R_j)	{ ST *Ri, Rj }
5)	$R_i \leftarrow$ ind(+(C_a, R_j))	{ LD Ri, a(Rj) }
6)	$R_i \leftarrow$ +(R_i, ind(+(C_a, R_j)))	{ ADD Ri, Ri, a(Rj) }
7)	$R_i \leftarrow$ +(R_i, R_j)	{ ADD Ri, Ri, Rj }
8)	$R_i \leftarrow$ +(R_i, C_1)	{ INC Ri }

树翻译方案的工作模式

- 给定一颗输入树，树重写规则中的模板被用来匹配输入树的子树
- 如果找到一个匹配的模板，那么输入树中匹配的子树将被替换为相应规则中的替换结点，并执行相应的动作，这可能是生成相应的机器指令序列
- 不断匹配，直到这颗树被规约成单个结点，或找不到匹配的模板为止
- 在此过程中生成的机器指令代码序列就是树翻译方案作用于给定输入树而得到的输出

树翻译方案生成目标指令示例

- 如何完成树匹配?
  - 把树重写规则替换成相应的上下文无关文法的产生式
  - 产生式的右部是其指令模板的前缀表示
- 如果在某个时刻有多个模板可以匹配
  - 匹配到大树优先

1)	R_i	$\rightarrow$ $\mathbf{c}_a$	{ LD Ri, #a }
2)	R_i	$\rightarrow$ M_x	{ LD Ri, x }
3)	M	$\rightarrow$ $= M_x\ R_i$	{ ST x, Ri }
4)	M	$\rightarrow$ $=$ **ind** $R_i\ R_j$	{ ST *Ri, Rj }
5)	R_i	$\rightarrow$ **ind** $+$ $\mathbf{c}_a\ R_j$	{ LD Ri, a(Rj) }
6)	R_i	$\rightarrow$ $+\ R_i$ **ind** $+$ $\mathbf{c}_a\ R_j$	{ ADD Ri, Ri, a(Rj) }
7)	R_i	$\rightarrow$ $+\ R_i\ R_j$	{ ADD Ri, Ri, Rj }
8)	R_i	$\rightarrow$ $+\ R_i\ \mathbf{c}_1$	{ INC Ri }
9)	R	$\rightarrow$ **sp**	
10)	M	$\rightarrow$ **m**	

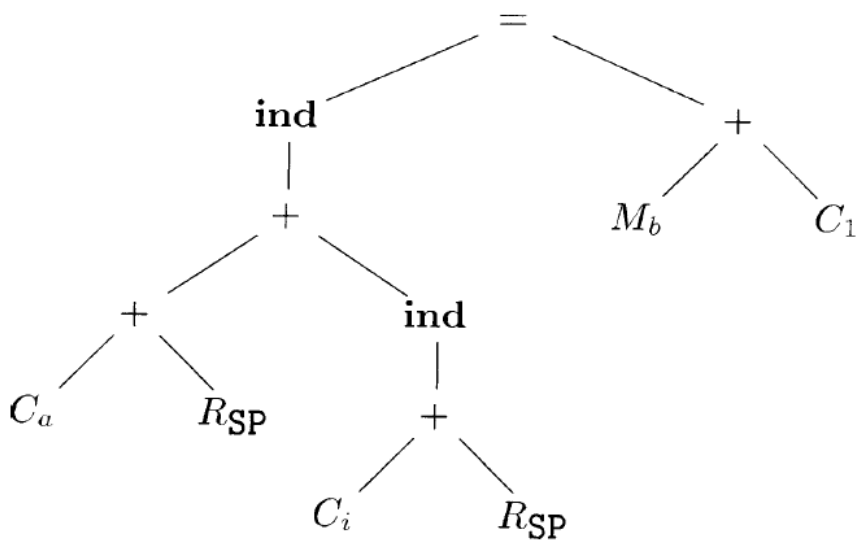


图 8-21　由图 8-20 构造得到的语法制导翻译方案