机器无关代码优化
Part3：数据流与可用表达式分析

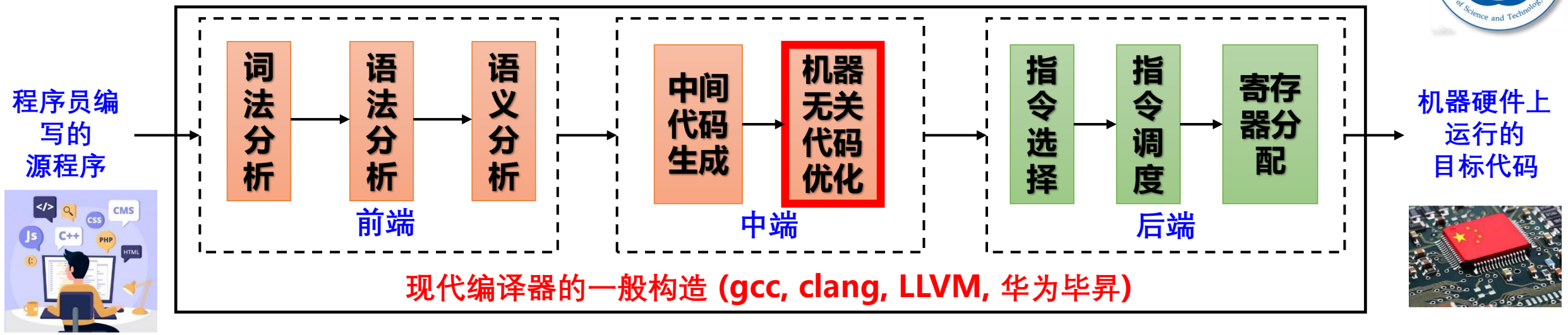
徐 伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年4月24日

本节提纲



- 可用表达式的定义和简单计算
- 可用表达式分析概述及算法介绍
- 可用表达式分析示例

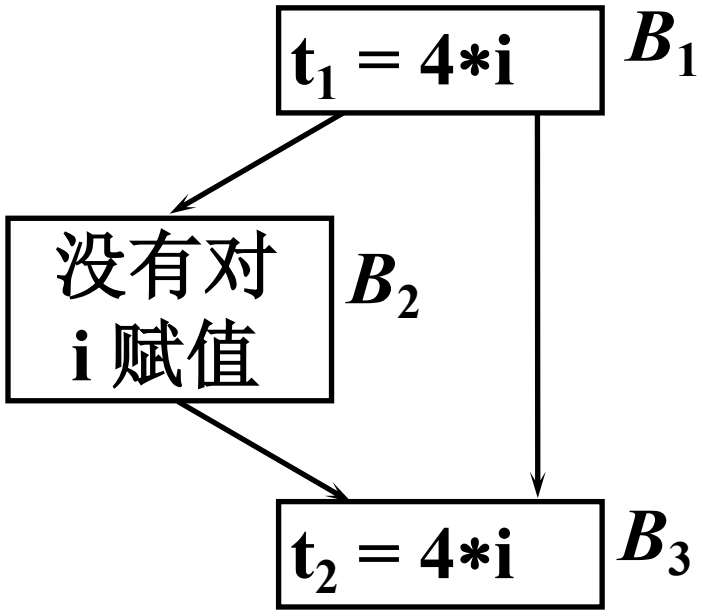
可用表达式

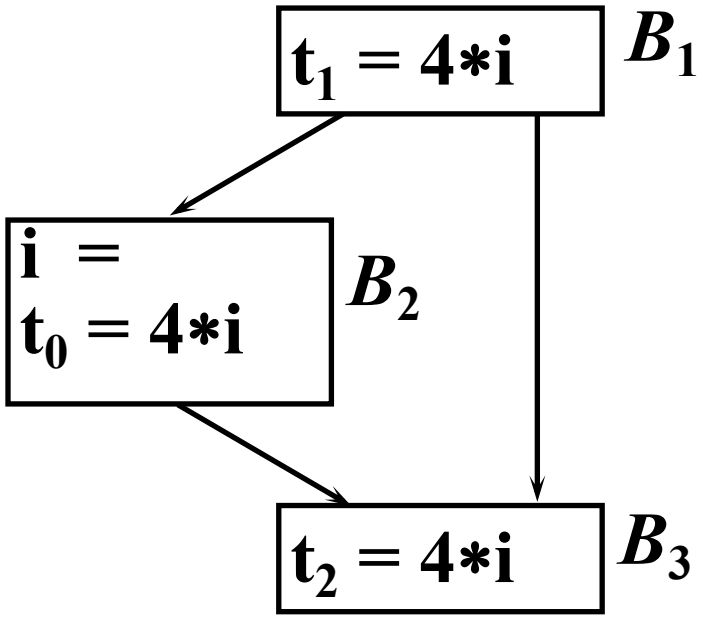
$x = y + z$	$x = y + z$	$x = y + z$
.	.	.
.	$y = \ldots$	$z = \ldots$
.	.	.
p	*p*	*p*
y + z 在*p*点	y + z 在*p*点	y + z 在*p*点
可用	不可用	不可用

可用表达式的应用

消除全局公共子表达式

- 例：下面两种情况下， 4*i在B_3的入口都可用





可用表达式

- **基本块生成的表达式：**

  基本块中语句d：x = y + z的前、后点分别为点p与点q。设在点p处可用表达式集合为S（基本块入口点处S为空集），那么经过语句d之后，在点q处可用表达式集合如下构成：

  (1) S = S ∪ { y+z }

  (2) S = S – { S 中所有涉及变量x的表达式 }

  注意，步骤(1)和(2)不可颠倒

可用表达式

- **基本块生成的表达式：**

  基本块中语句d：x = y + z的前、后点分别为点p与点q。设在点p处可用表达式集合为S（基本块入口点处S为空集），那么经过语句d之后，在点q处可用表达式集合如下构成：

  (1) S = S ∪ { y+z }

  (2) S = S – { S 中所有涉及变量x的表达式 }

  注意，步骤(1)和(2)不可颠倒，x可能就是y或z。

  如此处理完基本块中所有语句后，可以得到基本块生成的可用表达式集合S；

可用表达式

- **基本块生成的表达式：**

  基本块中语句d：x = y + z的前、后点分别为点p与点q。设在点p处可用表达式集合为S（基本块入口点处S为空集），那么经过语句d之后，在点q处可用表达式集合如下构成：

  (1) S = S ∪ { y+z }

  (2) S = S – { S 中所有涉及变量x的表达式 }

  注意，步骤(1)和(2)不可颠倒，x可能就是y或z。

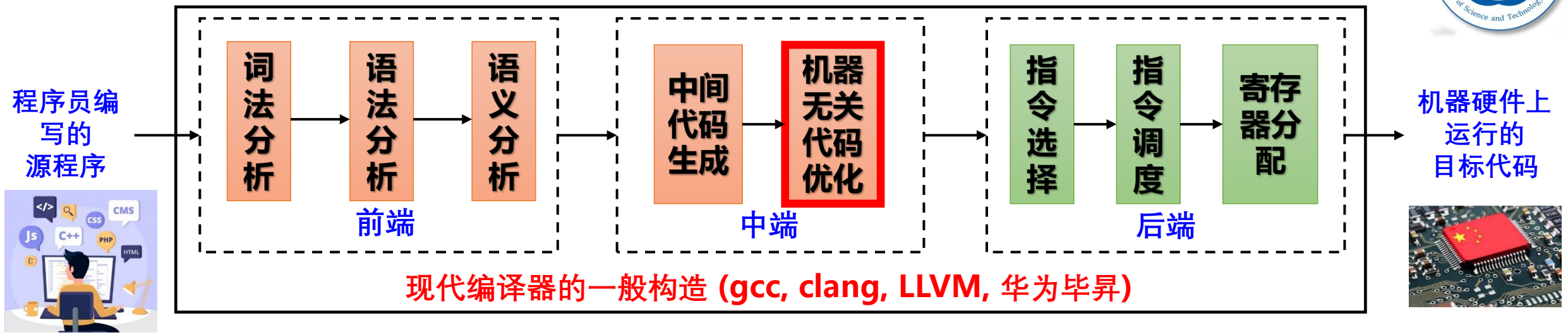
  如此处理完基本块中所有语句后，可以得到基本块生成的可用表达式集合S；

- **基本块杀死的表达式：所有其他类似y+z的表达式，基本块中对y或z定值，但基本块没有生成y+z。**

示例：基本块生成的表达式

语句	可用表达式
	Ø
a = b + c	{ b + c }
b = a – d	{ a – d } // b+c被杀死
c = b + c	{ a – d } // b+c被杀死
d = a – d	Ø // a – d 被杀死

本节提纲



- 可用表达式的定义和简单计算
- **可用表达式分析概述及算法介绍**
- **可用表达式分析示例**

可用表达式分析

- **定义**
  - 若到点p的每条执行路径都计算x op y，并且计算后没有对x或y赋值，那么称x op y在点p可用
  - e_gen_B：块B产生的可用表达式集合
  - e_kill_B：块B注销的可用表达式集合
  - IN [B]：块B入口的可用表达式集合
  - OUT [B]：块B出口的可用表达式集合

可用表达式分析

- **数据流等式**
  - **OUT** $[B] = e\_gen_B \cup (\text{IN } [B] - e\_kill_B)$
  - **IN** $[B] = \cap_{P是B的前驱}$ **OUT** $[P]$
  - OUT **[ENTRY]** $= \varnothing$
    - 在ENTRY的出口处没有可用表达式
- **同先前的主要区别**
  - 使用∩而不是∪作为这里数据流等式的汇合算符
  - 只有当一个表达式在B的所有前驱的结尾处都可用，那么它才会在B的开头可用
  - 求最大解而不是最小解

可用表达式数据流分析

- 迭代算法：

*U*是全体表达式集合

(1) OUT[ENTRY] = ø

(2) for(除ENTRY之外的每个基本块B) OUT[B]= *U*

(3) while(某个OUT值发生变化) {

(4) for(除ENTRY之外的每个基本块B){

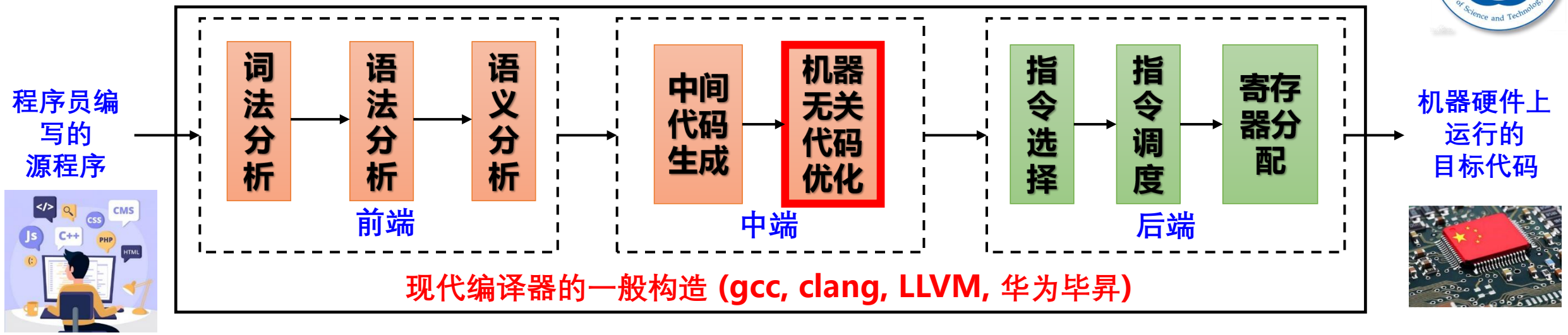
(5) IN[B] = ∩ P是B的前驱基本块(OUT[P])

(6) OUT[B] = $e\_gen_B \cup (IN[B] - e\_kill_B)$

} // end-of-for

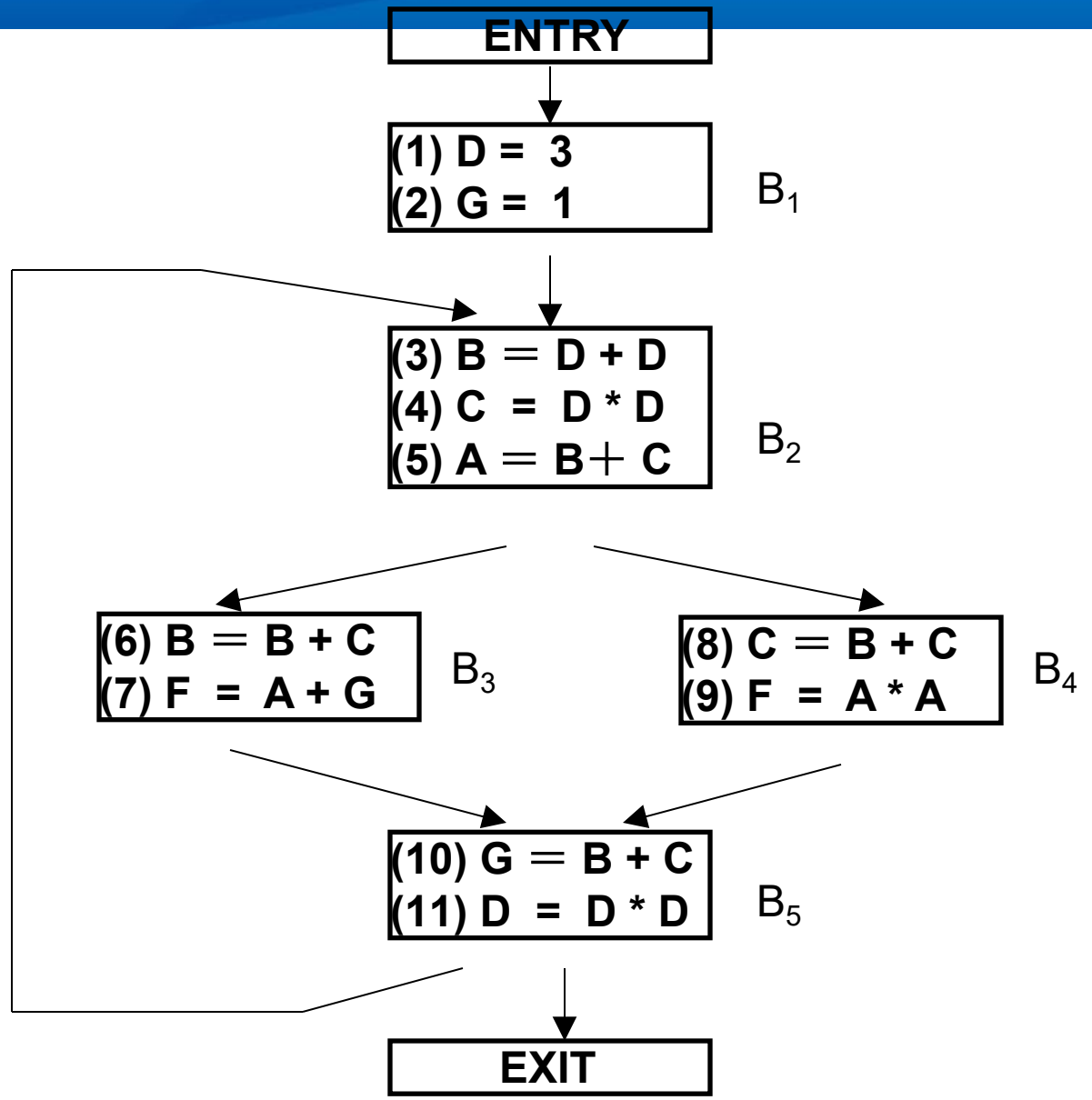
} // end-of-while

本节提纲



- 可用表达式的定义和简单计算
- 可用表达式分析概述及算法介绍
- **可用表达式分析示例**

示例：可用表达式分析


ENTRY
(1) D = 3
(2) G = 1
B1
(3) B = D + D
(4) C = D * D
(5) A = B+ C
B2
(6) B = B + C
(7) F = A + G
B3
(8) C = B + C
(9) F = A * A
B4
(10) G = B + C
(11) D = D * D
B5
EXIT

示例：可用表达式分析

基本块	前驱	后继
ENTRY	—	B_1
B_1	ENRTY	B_2
B_2	B_1 B_5	B_3 B_4
B_3	B_2	B_5
B_4	B_2	B_5
B_5	B_3 B_4	B_2 EXIT
EXIT	B_5	—

示例：可用表达式分析

基本块	e_gen	e_kill
ENTRY	Ø	Ø
B_1	{3, 1}	{ D+D, D*D, A+G }
B_2	{ D+D, D*D, B+C }	{ A*A, A+G }
B_3	{ A+G }	{ B+C }
B_4	{ A * A }	{ B+C }
B_5	{ B+C }	{ A+G, D*D, D+D }
EXIT	Ø	Ø
全部表达式*U*={ 3, 1, D+D, D*D, B+C, A+G, A*A }		

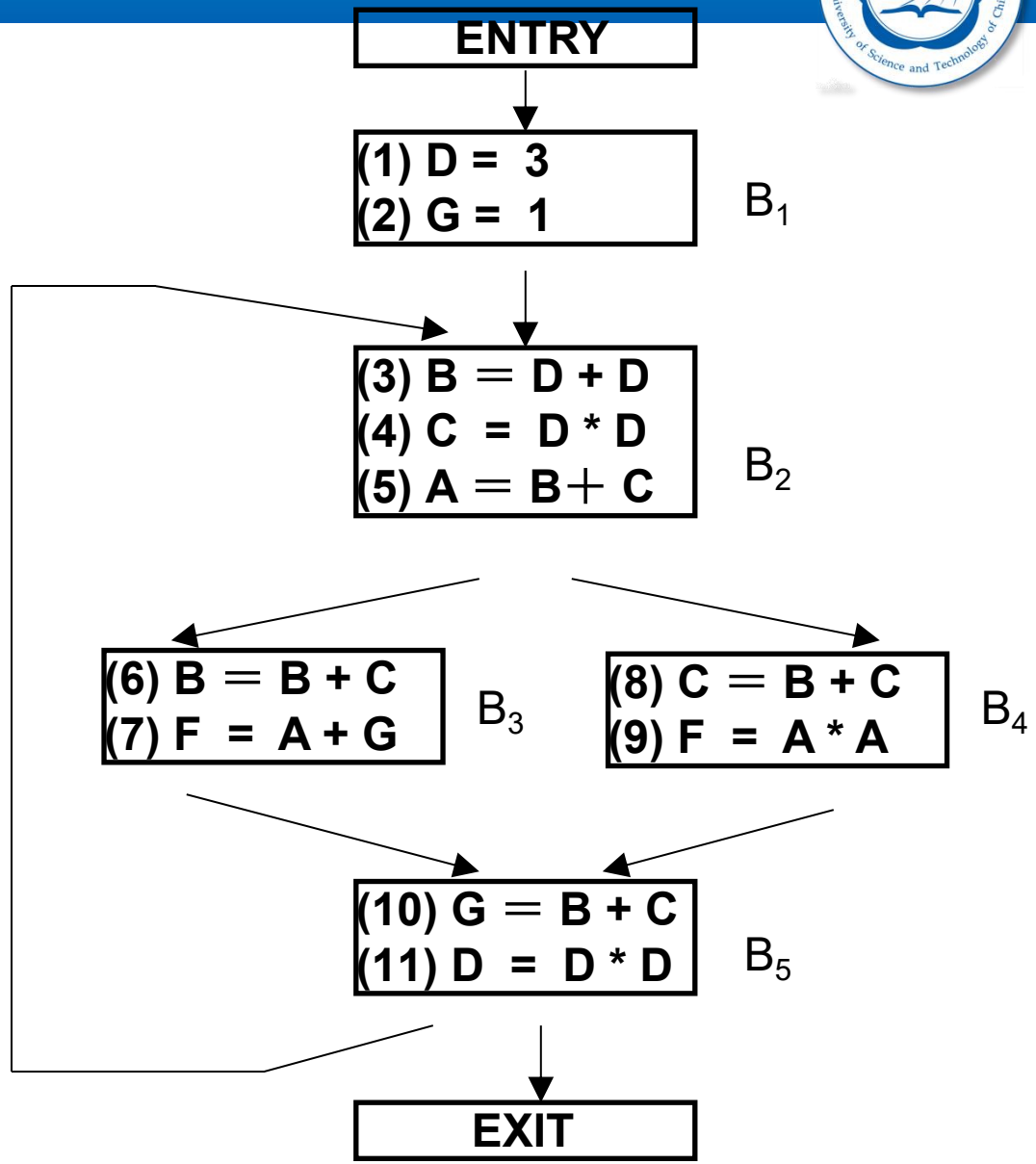
示例：可用表达式分析

基本块		e_kill
ENTRY		Ø
B_1		{ D+D, D*D, A+G }
B_2		{ A*A, A+G }
B_3	{ A+G }	{ B+C }
B_4	{ A * A }	{ B+C }
B_5	{ B+C }	{ A+G, D*D, D+D }
EXIT	Ø	Ø
全部表达式U={ 3, 1, D+D, D*D, B+C, A+G, A*A }		

- B2块的e_kill集合不包含B+C，因为虽然B和C的赋值改变了B+C的值，但是最后一个语句再次计算了B+C，这样B+C又成为可用表达式。生命力顽强，没有被kill掉。
- 从另一个视角来看，即便是e_kill中包含了B+C，OUT集合计算的时候也会被e_gen中的B+C覆盖掉。

示例：可用表达式分析

- 可用表达式的迭代计算
  - 深度优先序，即 B1 -> B2 -> B3 -> B4 -> B5 -> EXIT
  - 边界值：OUT[ENTRY] = ∅ ；
  - 初始化：for all NON-ENTRY B: OUT[B] = U ；



示例：可用表达式分析

- 第一次迭代：（all NON-ENTRY B）

(1) IN[B1] = OUT[ENTRY] = ∅ ； // B1 前驱仅为ENTRY

OUT[B1] = e_gen[B1] ∪ (IN[B1] – e_kill [B1])

= e_gen[B1] = { 3, 1 } //变化

(2) IN[B2] = OUT[B1] ∩OUT[B5]

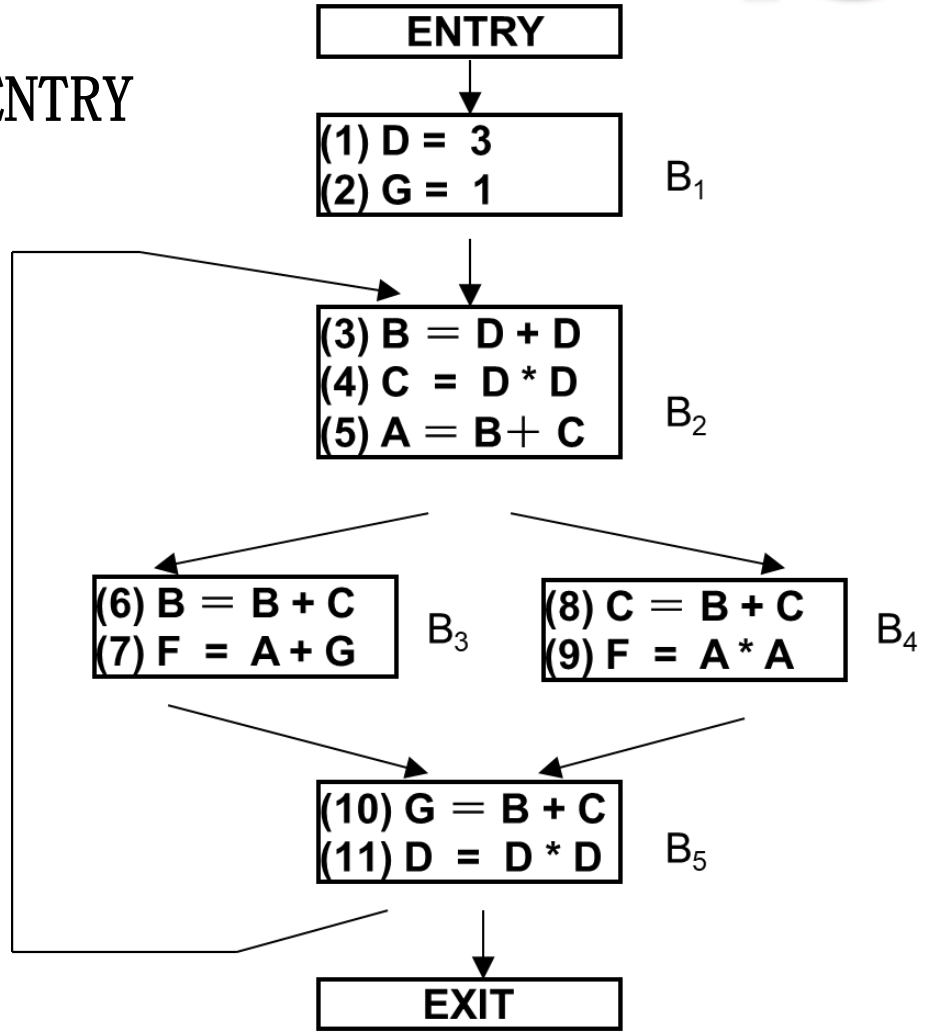
= { 3, 1 } ∩ U = { 3, 1 }

OUT[B2] = e_gen[B2] ∪ (IN[B2] – e_kill [B2])

= { D+D, D*D, B+C } ∪

({ 3, 1 } – {A*A, A+G })

= {3, 1, D+D, D*D, B+C } //变化

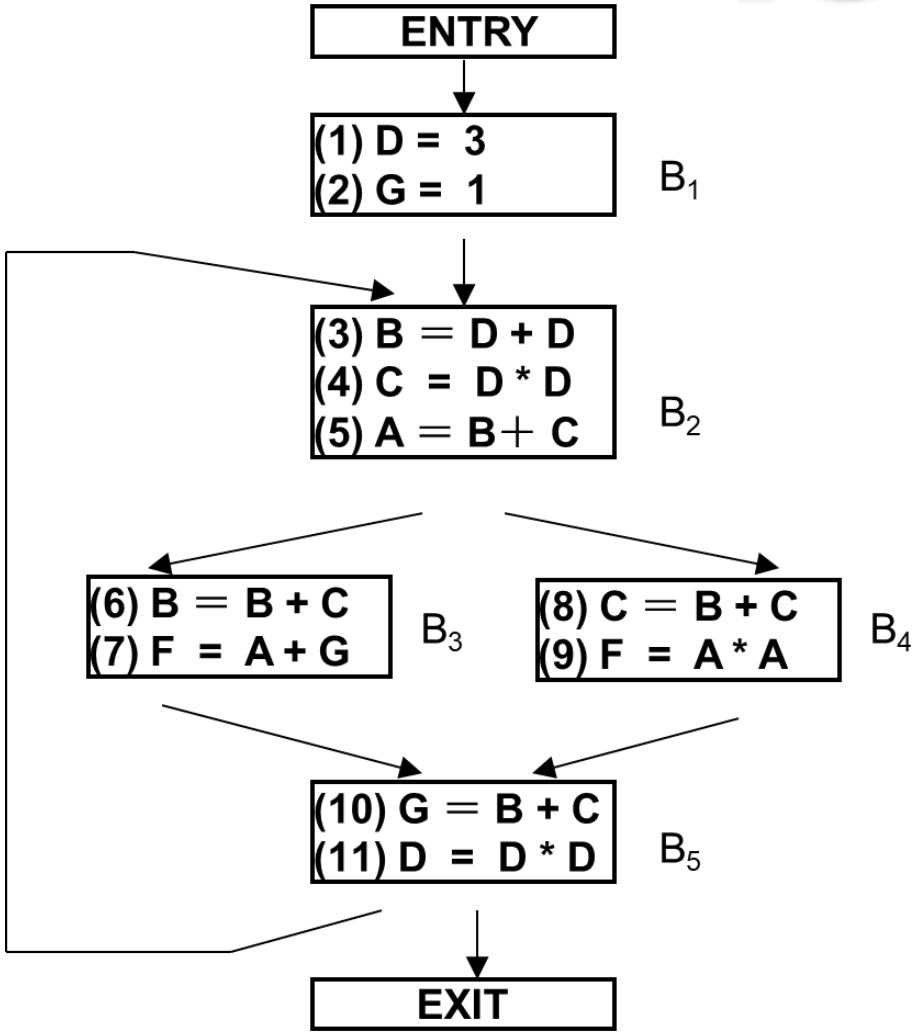


示例：可用表达式分析

- 第一次迭代：（all NON-ENTRY B）

(3) IN[B3] = OUT[B2]
= {3, 1, D+D, D*D, B+C }
OUT[B3] = e_gen[B3] ∪ (IN[B3] – e_kill[B3])
= { A+G } ∪ ({ 3, 1, D+D, D*D, B+C }
– { B+C })
= { 3, 1, D+D, D*D, A+G } //变化

(4) IN[B4] = OUT[B2]
= {3, 1, D+D, D*D, B+C }
OUT[B4] = e_gen[B4] ∪ (IN[B4] – e_kill[B4])
= { A * A } ∪({ 3, 1, D+D, D*D, B+C }
– { B+C })
= { 3, 1, D+D, D*D, A * A } //变化



示例：可用表达式分析

- 第一次迭代：（all NON-ENTRY B）

(5) IN[B5] = OUT[B3] ∩ OUT[B4]

= { 3, 1, D+D, D*D, A+G } ∩

{ 3, 1, D+D, D*D, A * A }

= { 3, 1, D+D, D*D }

OUT[B5] = e_gen[B5] ∪ (IN[B5] – e_kill[B5])

= {B+C} ∪ ({3,1,D+D, D*D} –

{A+G, D*D, D+D})

= { 3, 1, B+C } //变化

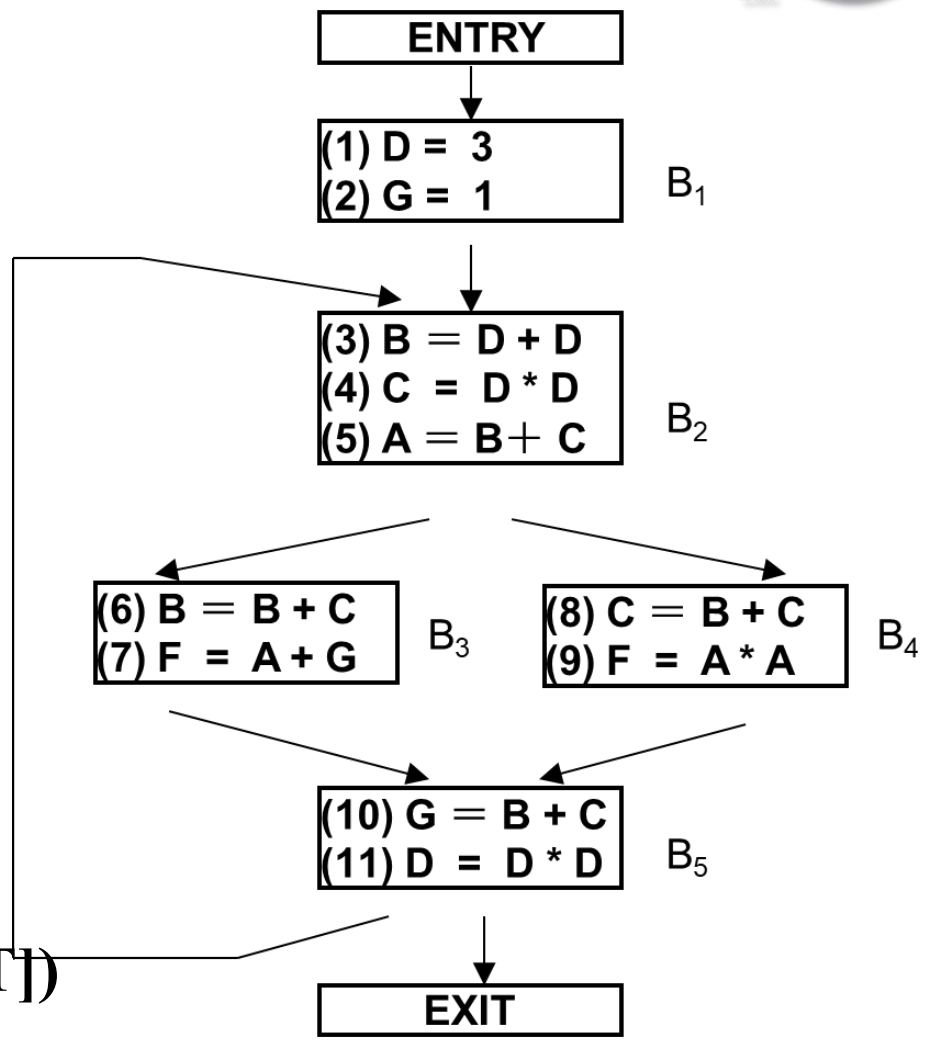
(6) IN[EXIT] = OUT[B5] = { 3, 1, B+C }

OUT[EXIT] = e_gen[EXIT] ∪

(IN[EXIT] –e_kill [EXIT])

= ⌀ ∪ ({ 3, 1, B+C } – ⌀)

= { 3, 1, B+C } //变化



示例：可用表达式分析

- 第二次迭代：（all NON-ENTRY B）

(1) IN[B1] = OUT[ENTRY] = ∅ ;

OUT[B1] = e_gen[B1] ∪ (IN[B1] – e_kill[B1])

= e_gen[B1] = { 3, 1 } // 不变

(2) IN[B2] = OUT[B1] ∩ OUT[B5]

= { 3, 1 } ∩ { 3, 1, B+C } = { 3, 1 } // 不变

OUT[B2] = e_gen[B2] ∪ (IN[B2] – e_kill[B2])

= { D+D, D*D, B+C } ∪ ({ 3, 1 } – {A*A, A+G })

= {3, 1, D+D, D*D, B+C } // 不变

示例：可用表达式分析

- 第二次迭代：（all NON-ENTRY B）

(3) IN[B3] = OUT[B2]
= {3, 1, D+D, D*D, B+C } //不变
OUT[B3] = e_gen[B3] ∪ (IN[B3] – e_kill[B3])
= { A+G } ∪ ({ 3, 1, D+D, D*D, B+C } – { B+C })
= { 3, 1, D+D, D*D, A+G } //不变

(4) IN[B4] = OUT[B2]
= {3, 1, D+D, D*D, B+C } //不变
OUT[B4] = e_gen[B4] ∪ (IN[B4] – e_kill[B4])
= { A * A } ∪({ 3, 1, D+D, D*D, B+C } – { B+C })
= { 3, 1, D+D, D*D, A * A } //不变

示例：可用表达式分析

- 第二次迭代：（all NON-ENTRY B）

(5) IN[B5] = OUT[B3] ∩ OUT[B4]

= { 3, 1, D+D, D*D, A+G } ∩{ 3, 1, D+D, D*D, A * A }

= { 3, 1, D+D, D*D } //不变

OUT[B5] = e_gen[B5] ∪ (IN[B5] – e_kill[B5])

= {B+C} ∪ ({3,1,D+D, D*D} – {A+G, D*D, D+D})

= { 3, 1, B+C } //不变

(6) IN[EXIT] = OUT[B5] = { 3, 1, B+C } //不变

OUT[EXIT] = e_gen[EXIT] ∪ (IN[EXIT] –e_kill[EXIT])

= ∅ ∪ ({ 3, 1, B+C } – ∅)

= { 3, 1, B+C } //不变

一起努力
打造国产基础软硬件体系！

徐 伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年4月24日