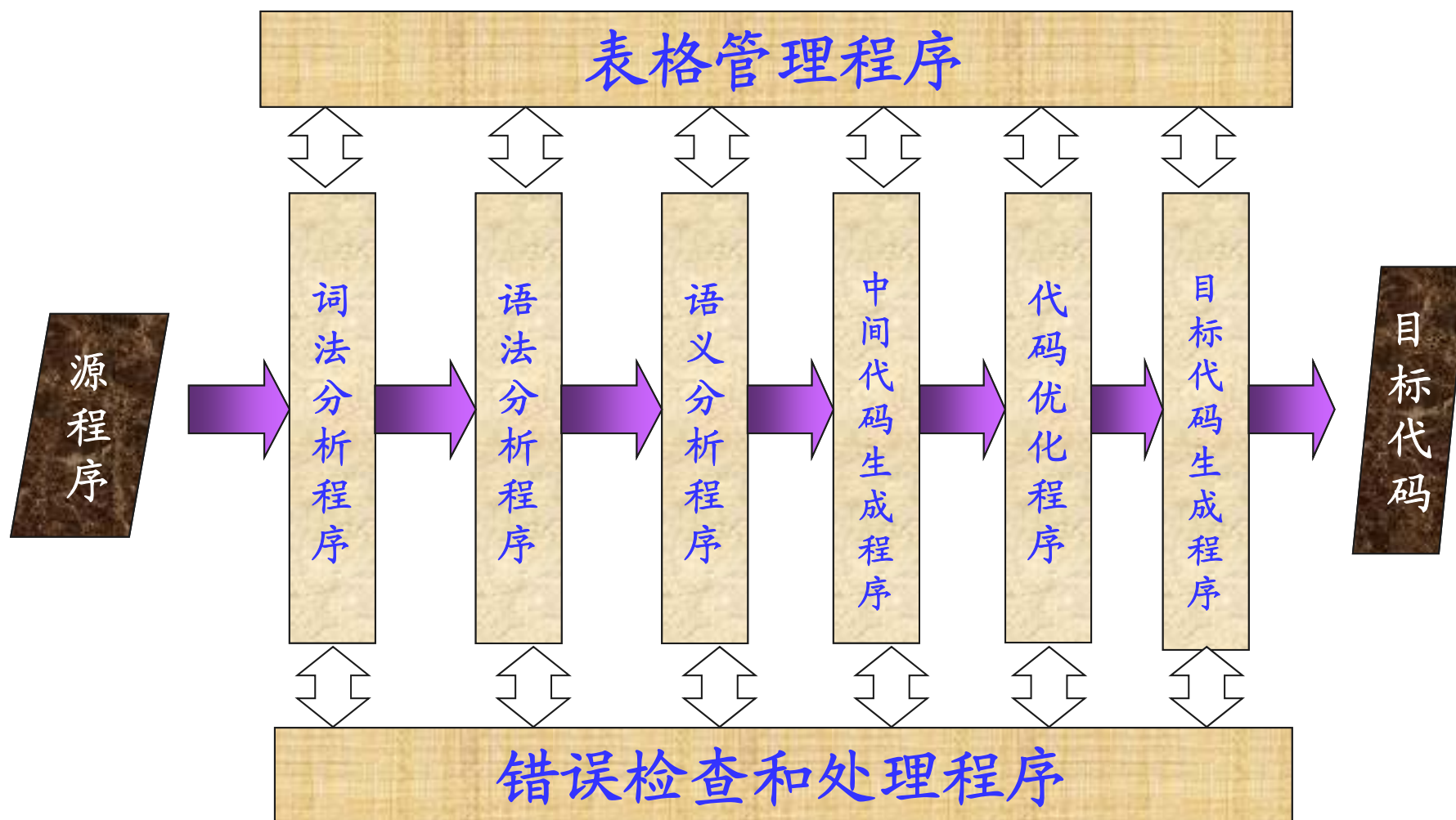




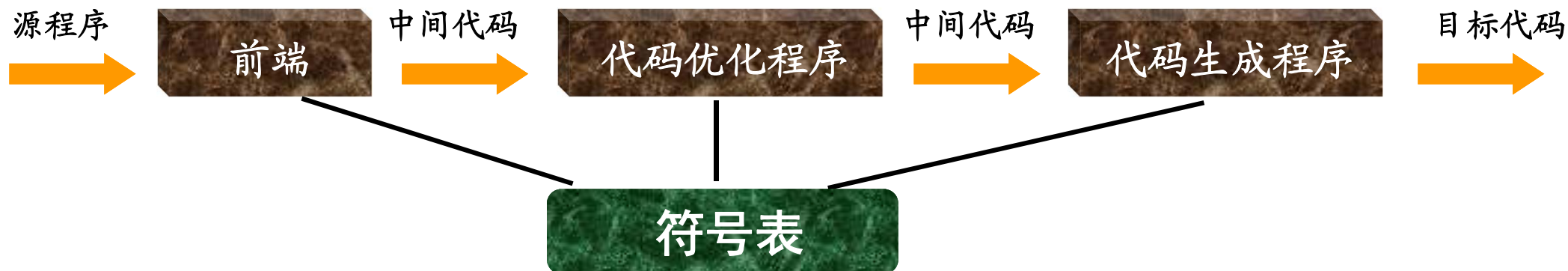
编译原理

武汉大学计算机学院
编译原理课程组

编译程序的结构



第10章 符号表

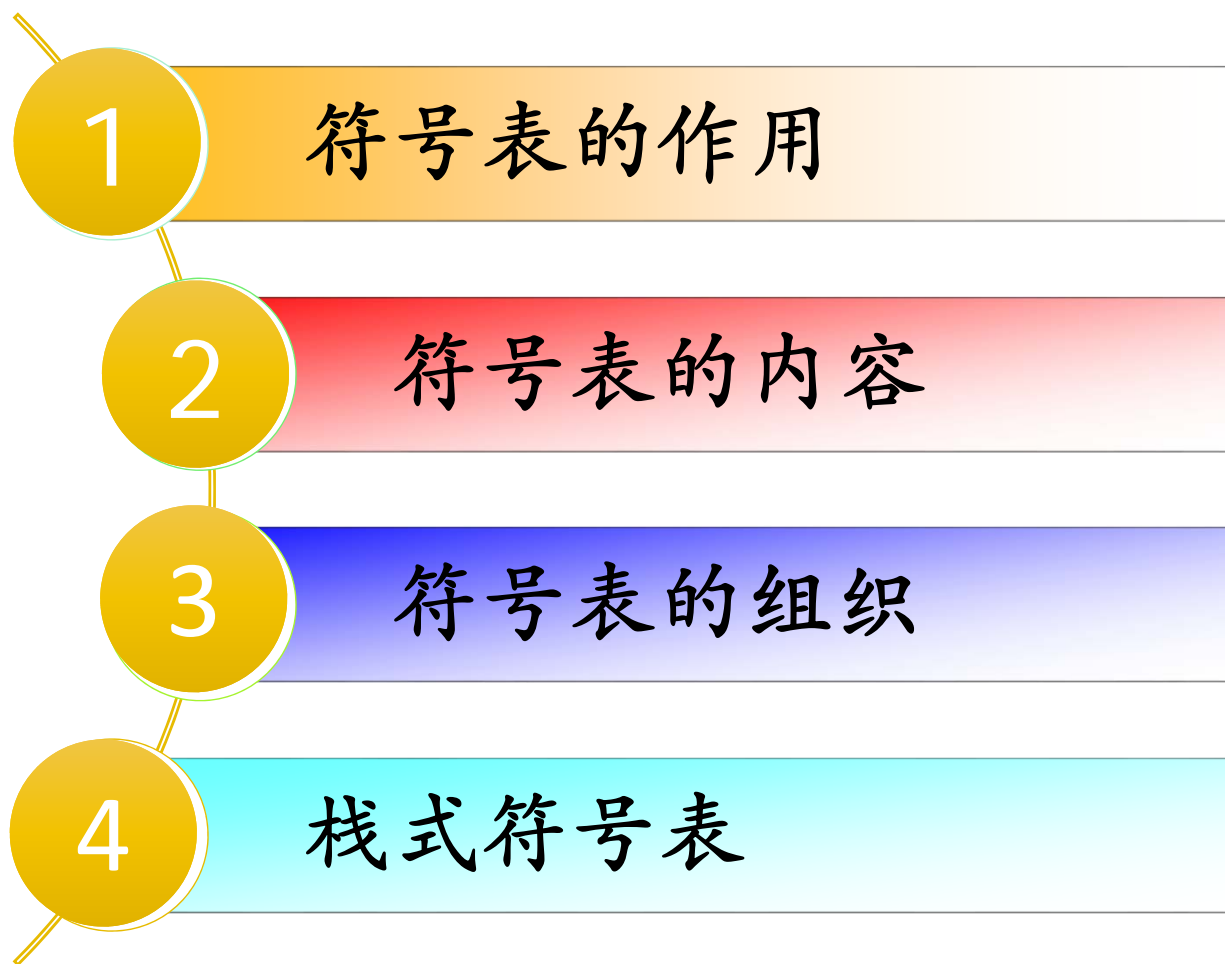


登记源程序中出现的每个名字以及名字的各种属性。

有些名字的属性需要在各个阶段才能填入。

NAME	TYPE	CAT	...	VAL	ADDR
position	...				
initial	...				
rate	...				

第10章 符号表





10.1 符号表的作用

◆ 符号表的作用

符号表在整个编译期间的作用主要有两条：一是辅助语义的（即上下文有关的）正确性检查；二是辅助代码生成。

◆ 符号表的生存期

符号表的建立可以开始于词法分析阶段，也可以放到语法语义阶段，但符号表的使用，有时会延续到目标代码的运行阶段（如运行时刻为了诊断的需要，数组下标地址计算的需要等）。



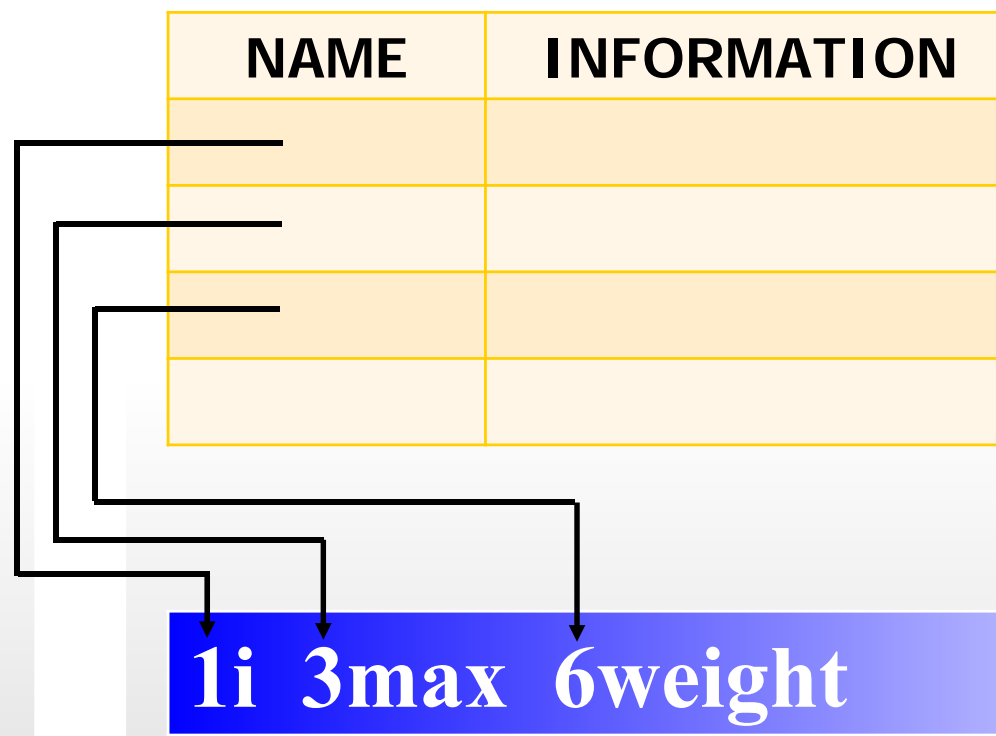
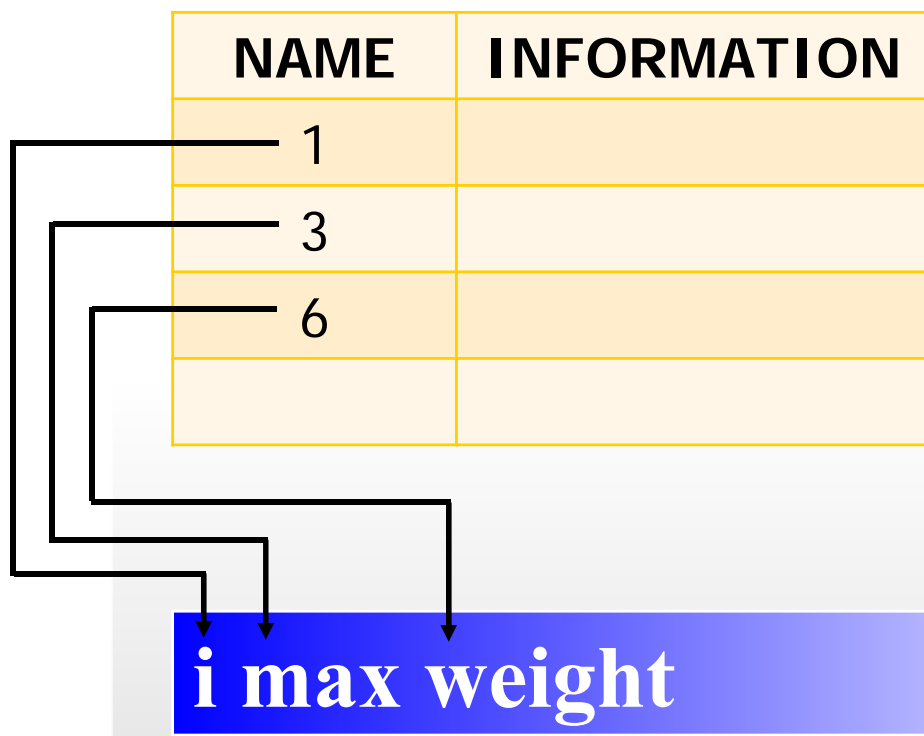
10.2 符号表的内容

- ◆ 标识符的~~名字~~ —— ~~主目~~
- ◆ 与标识符~~有关的信息~~ —— ~~值~~

不同的标识符在符号表中具有不同的信息。

- 1) 数组——信息向量表(内情向量表)
- 2) 记录——域紧接着相应的记录变量名字相继存放
- 3) 过程或函数——参数的个数、类型、次序、是否允许递归等

符号表的结构





10.2 符号表的内容

① 类型信息

包括**种属**(常量、变量、数组、标号、函数或过程等)与**类型**(整型、实型、字符型、布尔型等)。

② 地址码

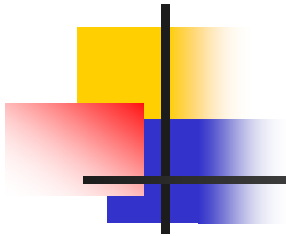
i. 简单变量或常量—— 一般是在数据区中的绝对或相对地址

ii. 数组—— 在数据区中的首地址

iii. 过程或函数—— 过程或函数的分程序入口地址

③ 层次信息—— 标识符所属分程序(过程)的静态层次

④ 行号信息—— 标识符在源程序中的行号, 包括说明行与引用行。



符号表举例

程序说明部分:

```
CONST A=35, B=49;  
VAR C, D, E;  
PROCEDURE P;  
VAR G ; ...
```

名字	种属	层次/值	地址	存储空间
NAME: A	KIND: CONSTANT	VAL: 35		
NAME: B	KIND: CONSTANT	VAL: 49		
NAME: C	KIND: VARIABLE	LEVEL: LEV	ADR: DX	
NAME: D	KIND: VARIABLE	LEVEL: LEV	ADR: DX+1	
NAME: E	KIND: VARIABLE	LEVEL: LEV	ADR: DX+2	
NAME: P	KIND: PROCEDUR	LEVEL: LEV	ADR:	SIZE: 4
NAME: G	KIND: VARIABLE	LEVEL: LEV+1	ADR: DX	
.....	



10.3 符号表的结构与组织

1. 符号表的结构

线性符号表、树结构、散列表或桶等。

NAME	INFORMATION
sum
x	
y	
temp	
z	
a	
b	
.....	



10.3 符号表的结构与组织

1. 符号表的结构

线性符号表、树结构、散列表或桶等。

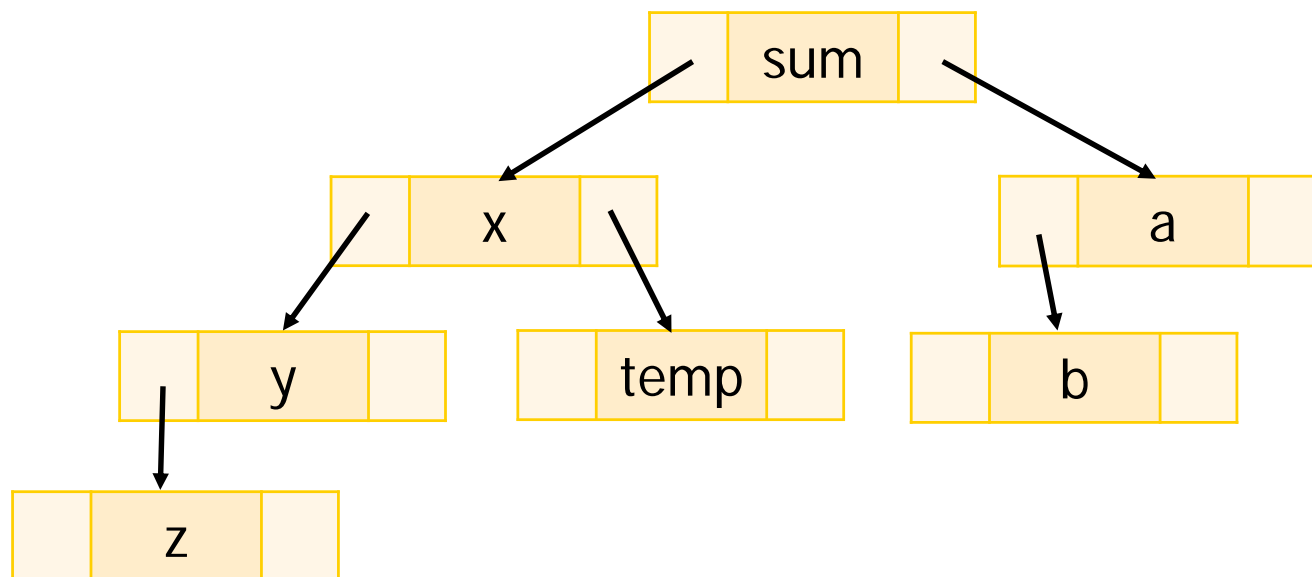
NAME	INFORMATION
a
b	
sum	
temp	
x	
y	
z	
.....	

10.3 符号表的结构与组织

1. 符号表的结构

线性符号表、**树结构**、散列表或桶等。

NAME
a
b
sum
temp
x
y
z
.....





10.3 符号表的结构与组织

1. 符号表的结构

线性符号表、树结构、散列表或桶等。

x	→	hash(x)
---	---	---------

地址冲突



10.3 符号表的结构与组织

2. 符号表的组织

符号表的条目一般由两部分组成，即名字栏与信息栏。

通常，条目用连续的存储字构成的记录来实现。为保持符号表条目记录的统一，往往把与名字相关联的构造类型等信息保存在符号表的某处(如数组的信息向量表等)，而把相应的指针或序号放在记录内。

不同种属的符号表组织

NAME	INFORMATION		
	CAT	ADD
Name1			
Name2	A (数组)	→
Name3			

数组的信息向量表

l_1	u_1	d_1
l_2	u_2	d_2
...
l_n	u_n	d_n
n	Conspart	
type	Baseloc	

不同种属使用不同符号表（信息栏长度统一，处理方便）

常数表、变量名表、过程名表、标号表等。



10.3 符号表的结构与组织

3. 符号表的操作

- (1) 判定一给定的名字是否在表中;
- (2) 在表中填入一个新名字;
- (3) 访问与给定名字相关的信息;
- (4) 为给定的名字填入或更新其某些信息;
- (5) 从表中删除一个或一组无用的项.

大量的查找、填表与删除等操作。



10.3 符号表的结构与组织

4. 符号表的构造与查找

大量的查找、填表与删除等操作。

对于符号表组织、构造和管理方法的好坏会直接影响编译系统的运行效率。

- ◆ 线性符号表 → 线性查找：填表快、查表慢
- ◆ 有序符号表 → 折半查找（二叉树查找）：查表快、填表慢
- ◆ 散列符号表 → 散列查找：查、填快；存储空间要求高
- ◆ 栈式符号表：嵌套结构的程序设计语言



10.4 栈式符号表——PASCAL符号表的设计

1. 语言的特点

在很多程序设计语言中,对名字的作用域有相应的规定,即同一名字的标识符,在不同的作用域里标识了不同的对象,且占用了不同的存储空间.因此,在组织符号表时,应能反映各个标识符的作用域.

PASCAL按照最近嵌套作用域原则,一个名字的作用域是那个包含了这个名字的说明的最小过程或函数。

PASCAL的过程是嵌套的,内层可引用外层过程中说明的名字。



10.4 栈式符号表——PASCAL符号表的设计

2. 符号表的设计

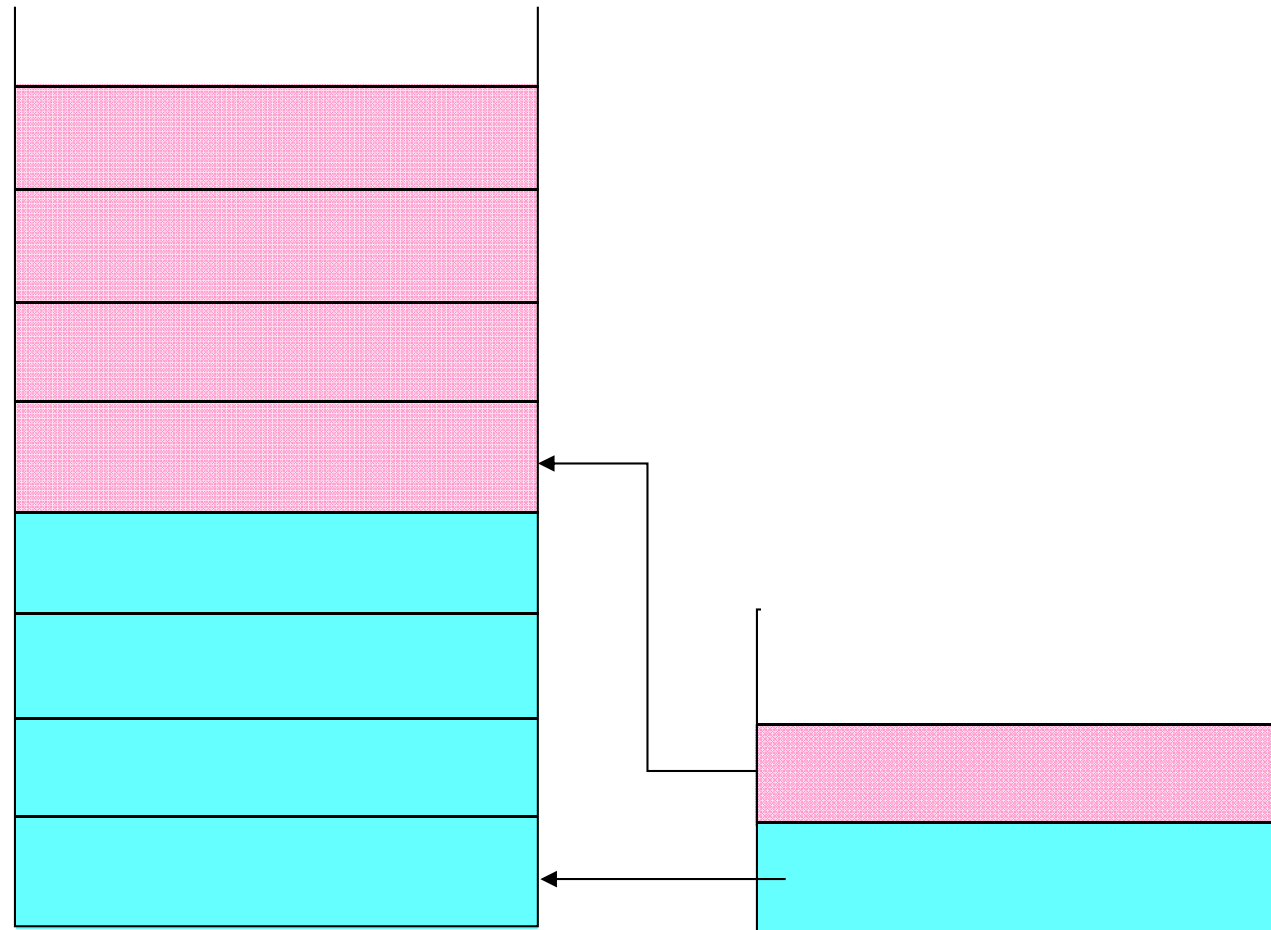
符号表设计为栈符号表，新的名字总是从栈顶填入。每当进入一层过程时，为该过程建立一张子符号表，在退出此过程时，则删除相应的子符号表，使现行符号表与进入此过程前的内容保持一致。

由于过程是嵌套的，所以，子符号表也是嵌套的，并按序生成。这样，不同层次的同名标识符不会导致混乱。

查找操作从符号表的栈顶往底部查(保证先查最近出现的名字)。

10.4 栈式符号表——PASCAL符号表的设计

3. 举例



栈符号表

DISPLAY

```
PROGRAM main;  a=10;  b,c: integer; d,e: real;
```

0

```
  PROCEDURE P(x: real);  f: real;
```

1

```
    PROCEDURE q(y: real);  g=5;  n: boolean;
```

2 BEGIN

```
        ... IF e<0 THEN p(f);  ...
```

```
      END
```

```
    BEGIN
```

```
        ... q(e);  ...
```

```
    END;
```

```
  PROCEDURE t;  j: real;
```

1

```
    BEGIN
```

```
        ...p(e);  ...
```

```
    END;
```

```
  BEGIN
```

```
    ... WHILE c>0 Do t;
```

```
    p(d);  ...
```

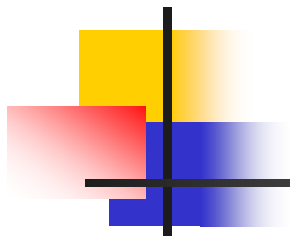
```
  END, {main}
```



对比 & 注意

第 9 章 栈式 存储分配

第 10 章 栈式 符号表



下节内容

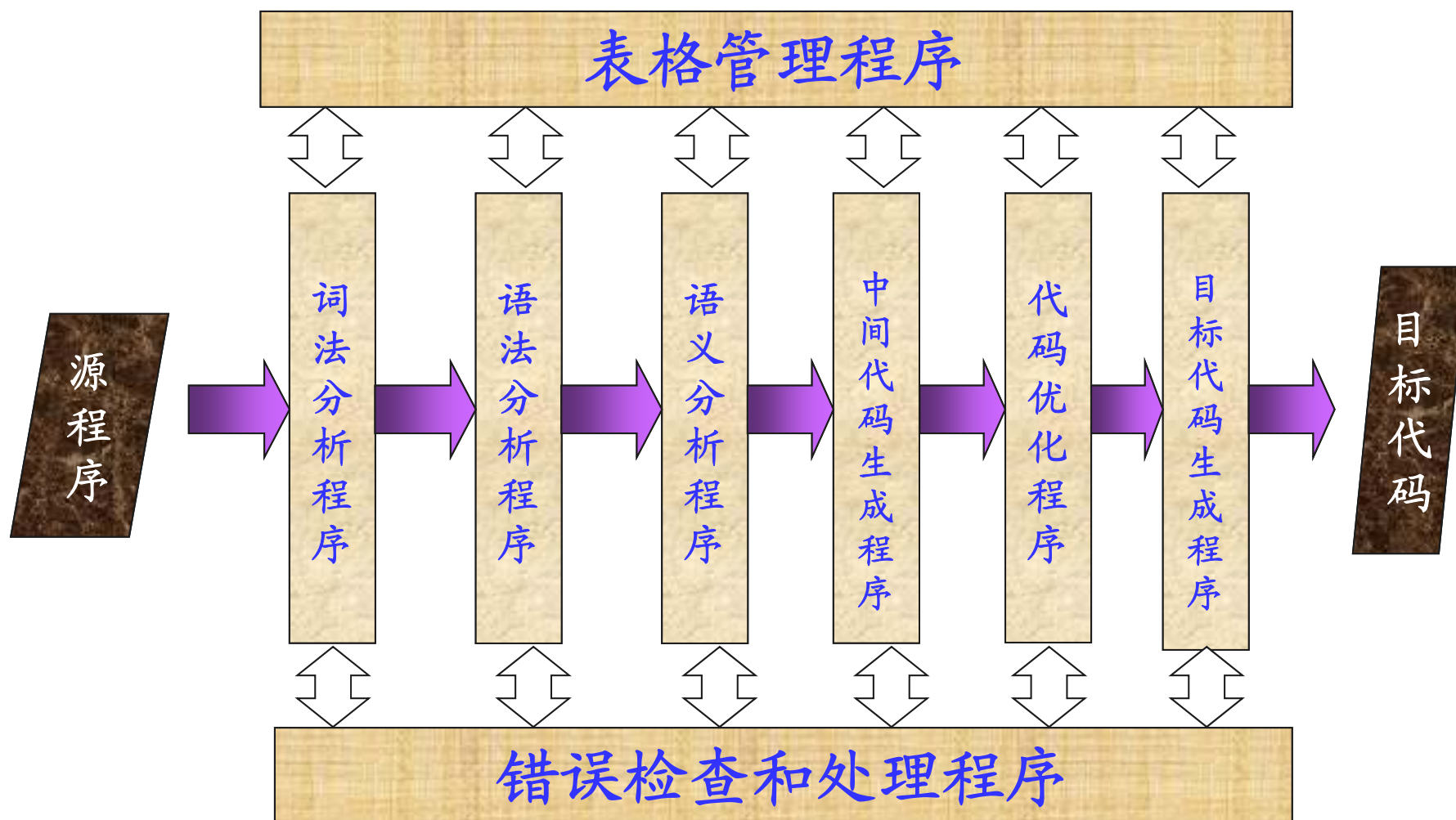
代码优化



编译原理

武汉大学计算机学院
编译原理课程组

编译程序的结构



编译过程——代码优化

(1) (inttoreal, 60 - t1)

(2) (* , id3 t1 t2)

(3) (+ , id2 t2 t3)

(4) (:= , t3 - id1)

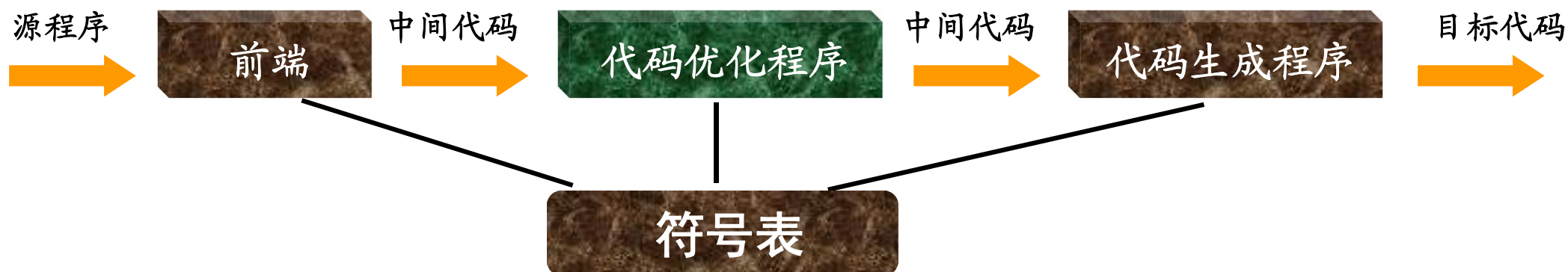
id1 := id2 + id3 * 60

Intermediate code optimizer

(1) (* , id3 60.0 t1)

(2) (+ , id2 t1 id1)

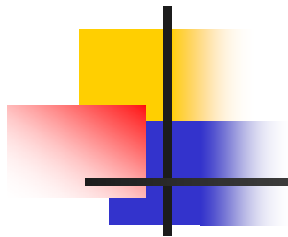
第11章 代码优化



- ◆ 中间代码的优化
- ◆ 目标代码的优化

第11章 代码优化





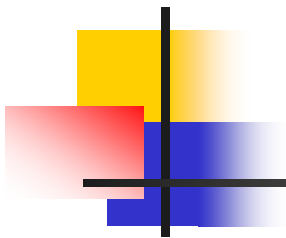
11.1 概述

1. 目的

提高目标程序的效率

2. 改进程序效率的途径

- ◆ 源程序级等价变换
- ◆ 利用程序库
- ◆ 编译时刻的优化



11.1 概述

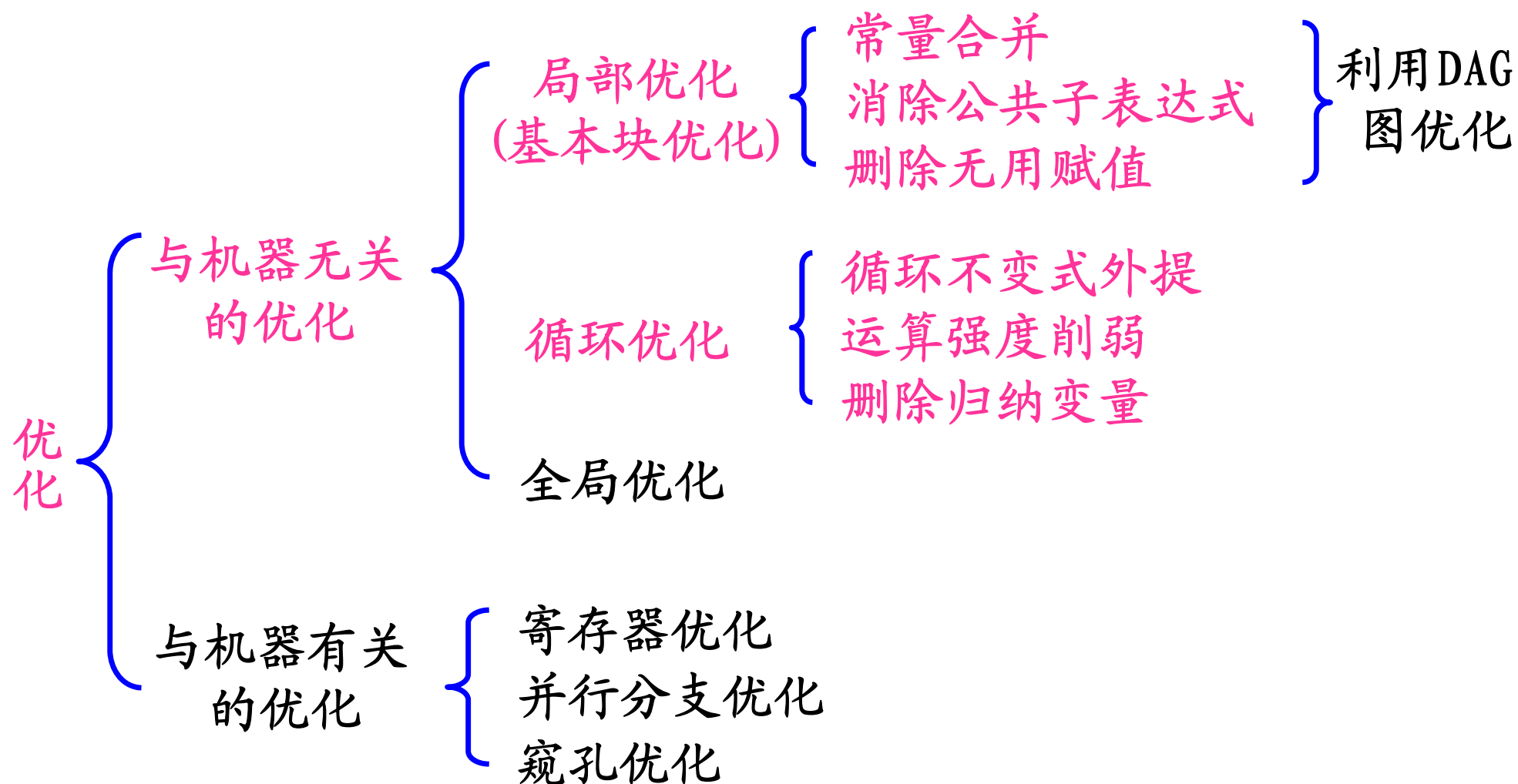
3. 编译时刻的优化

在中间代码上进行，仅在目标程序上进行窥孔优化。

注意事项：

- (1) 等价原则——不应改变程序的功能。
- (2) 有效原则——优化后的目标代码效率确实提高。
- (3) 合算原则——以较低的代价取得较好的优化效果。

11.2 优化的种类





11.3 基本块优化

1. 基本块

所谓**基本块**，是指程序中**一顺序执行的语句序列**，其中只有一个入口和一个出口，入口就是第一个语句，出口就是最后一个语句。对于一个基本块来说，执行时，**只能**从其入口进入，从其出口退出。



11.3 基本块优化

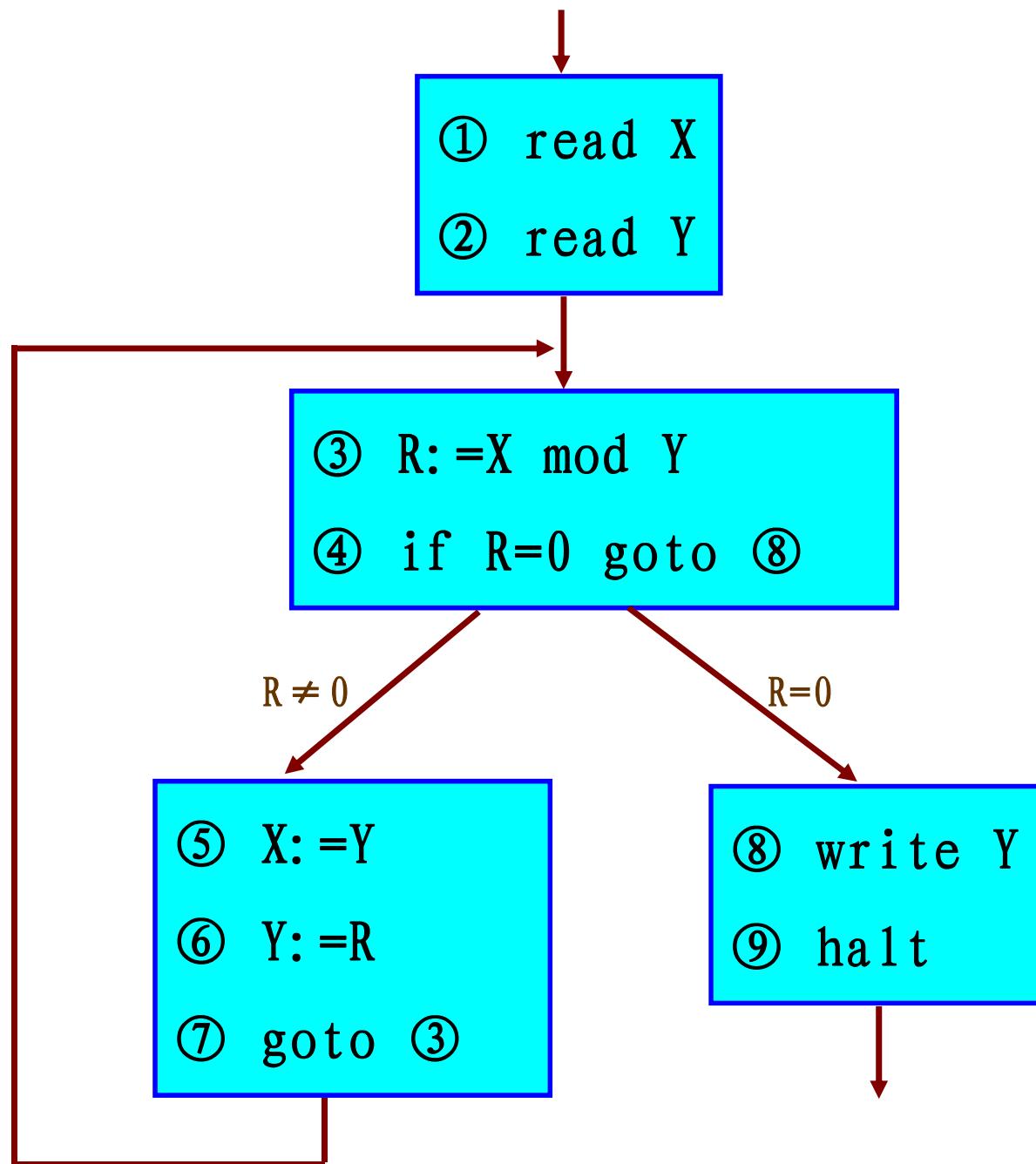
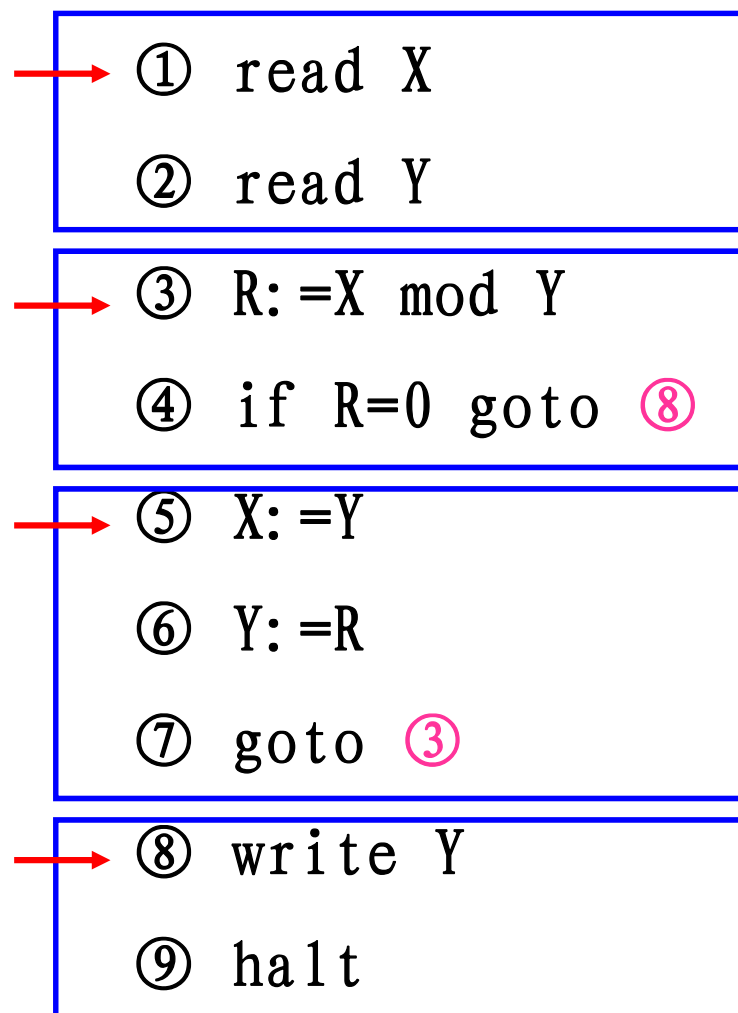
2. 划分基本块的算法

①求入口语句，它们是：

- i. 程序的第一个语句；或者
- ii. 能由条件转移语句或无条件转移语句转移到的语句；或者
- iii. 紧跟在条件转移语句后面的语句。

②对以上求出的每一入口语句，构造其所属的基本块。它是由该入口语句到另一入口语句(不包括该入口语句)，或到一转移语句(包括该转移语句)，或到一停语句(包括该停语句)之间的语句序列组成。

③凡未被纳入某一基本块的语句，都是程序中控制流程无法到达的语句，从而也是不会被执行到的语句，将其删除。

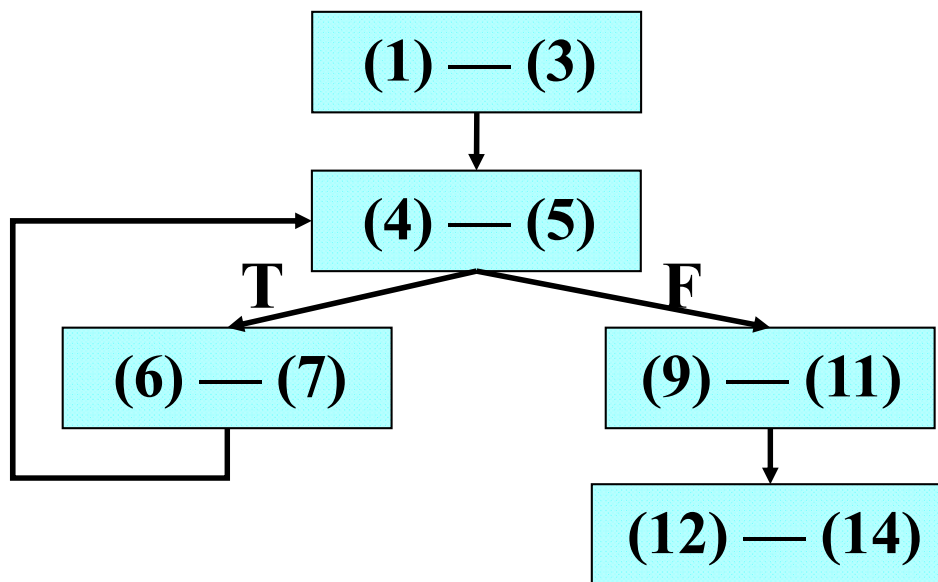


程序的流图

11.3 基本块优化

冗余语句

凡未被纳入某一基本块的语句，都是程序中控制流程无法到达的语句，从而也是不会被执行到的语句，将其删除。



→	(1)	Block		
	(2)	:=	100	k
	(3)	+	i	j
				T1
→	(4)	>	k	T1
				T2
	(5)	jumpf	(9)	T2
→	(6)	−	k	1
				k
	(7)	jump		(4)
	(8)	jump		(12)
→	(9)	*	i	2
				T3
	(10)	*	j	2
				T4
	(11)	−	T3	T4
				k
→	(12)	:=	0	j
				i
	(13)	:=	0	
	(14)	Blockend		



11.3 基本块优化

3. 基本块内可进行的优化

删除公共子表达式 (Common Sub-expression Elimination)

删除无用代码 (Dead Code Elimination)

复写传播 (Copy Propagation)

合并已知常量 (Constant folding)

.....



11.3 基本块优化

3. 基本块内可进行的优化

$T_6 := 4*i$

$x := a[T_6]$

$T_7 := 4*i$

$T_8 := 4*j$

$T_9 := a[T_8]$

$a[T_7] := T_9$

$T_{10} := 4*j$

$a[T_{10}] := x$

goto L

$T_6 := 4*i$

$x := a[T_6]$

$T_7 := T_6$

$T_8 := 4*j$

$T_9 := a[T_8]$

$a[T_7] := T_9$

$T_{10} := T_8$

$a[T_{10}] := x$

goto L

删除公共子表达式

$T_6 := 4*i$

$x := a[T_6]$

$T_7 := T_6$

$T_8 := 4*j$

$T_9 := a[T_8]$

$a[T_6] := T_9$

$T_{10} := T_8$

$a[T_8] := x$

goto L

复写传播

$T_6 := 4*i$

$x := a[T_6]$

$T_7 := T_6$

$T_8 := 4*j$

$T_9 := a[T_8]$

$a[T_6] := T_9$

$T_{10} := T_8$

$a[T_8] := x$

goto L

删除无用代码

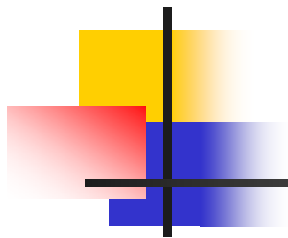


11.2 基本块优化

4. 基本块的DAG (Directed Acyclic Graph) 表示

基本块的DAG图的特征:

- i. 叶结点用标识符(变量名)或常数作为其唯一的标记, 当叶结点是标识符时, 代表名字的初值;
- ii. 内部结点用运算符标记, 它表示计算的值;
- iii. 各结点可能附加有一个或若干个标识符, 附加于同一个结点上的若干个标识符有相同的值。



11.3 基本块优化

5. 利用DAG图进行基本块的优化

①从基本块构造DAG

②从DAG重写基本块

11.3 基本块优化

5. 利用DAG图进行基本块的优化

$t1 := 3 * A$

$t2 := 2 * C$

$t3 := t1 + t2$

$t4 := t3 + 5$

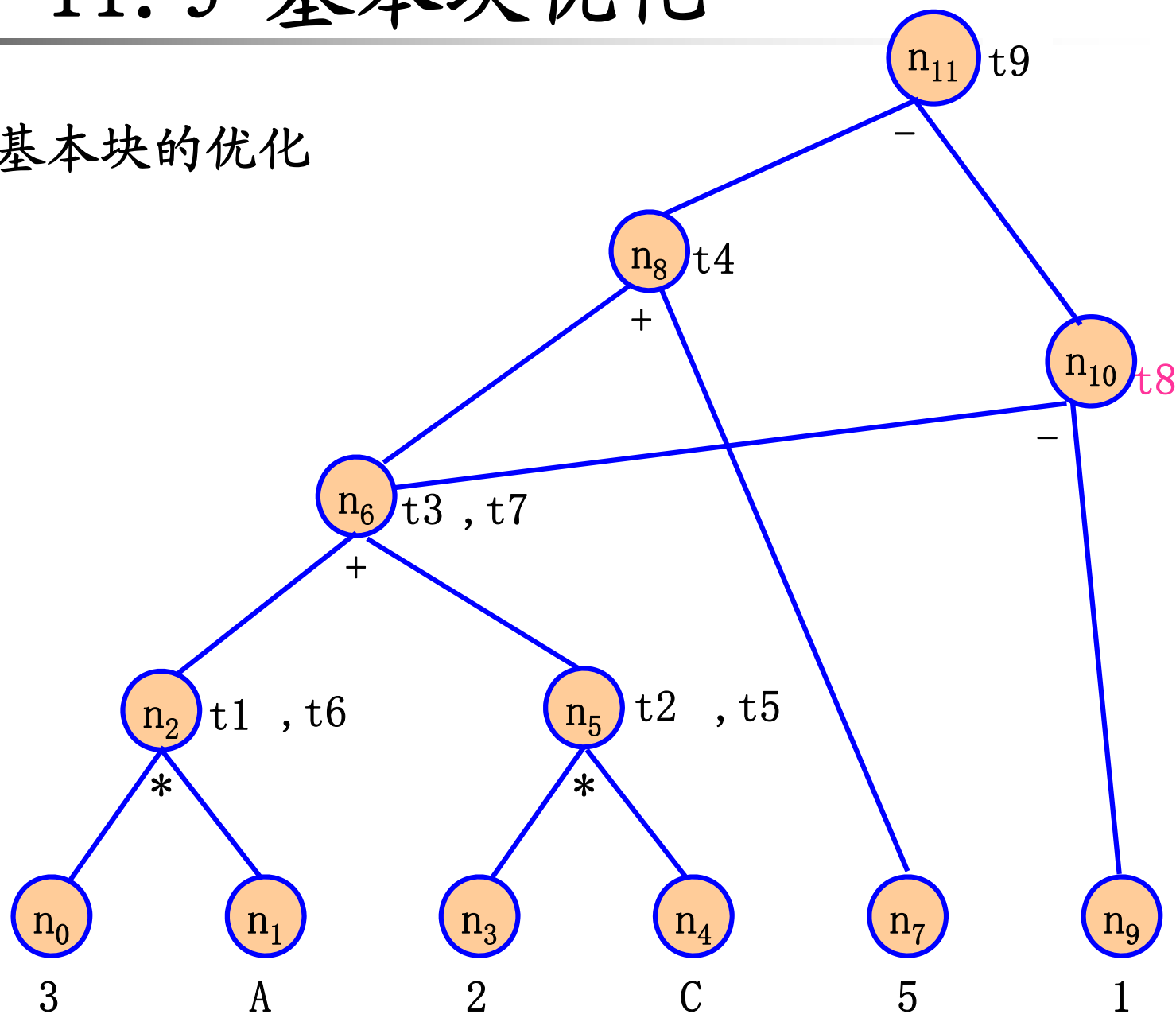
$t5 := 2 * C$

$t6 := 3 * A$

$t7 := t6 + t5$

$t8 := t7 - 1$

$t9 := t4 - t8$





11.4 循环优化

1. 循环优化的种类

◆ 循环不变表达式外提

若循环的某一运算所涉及的运算对象是该循环的不变量，则将此运算外提至本循环外。循环中外提的代码统统外提到前置结点中。

◆ 运算强度削弱

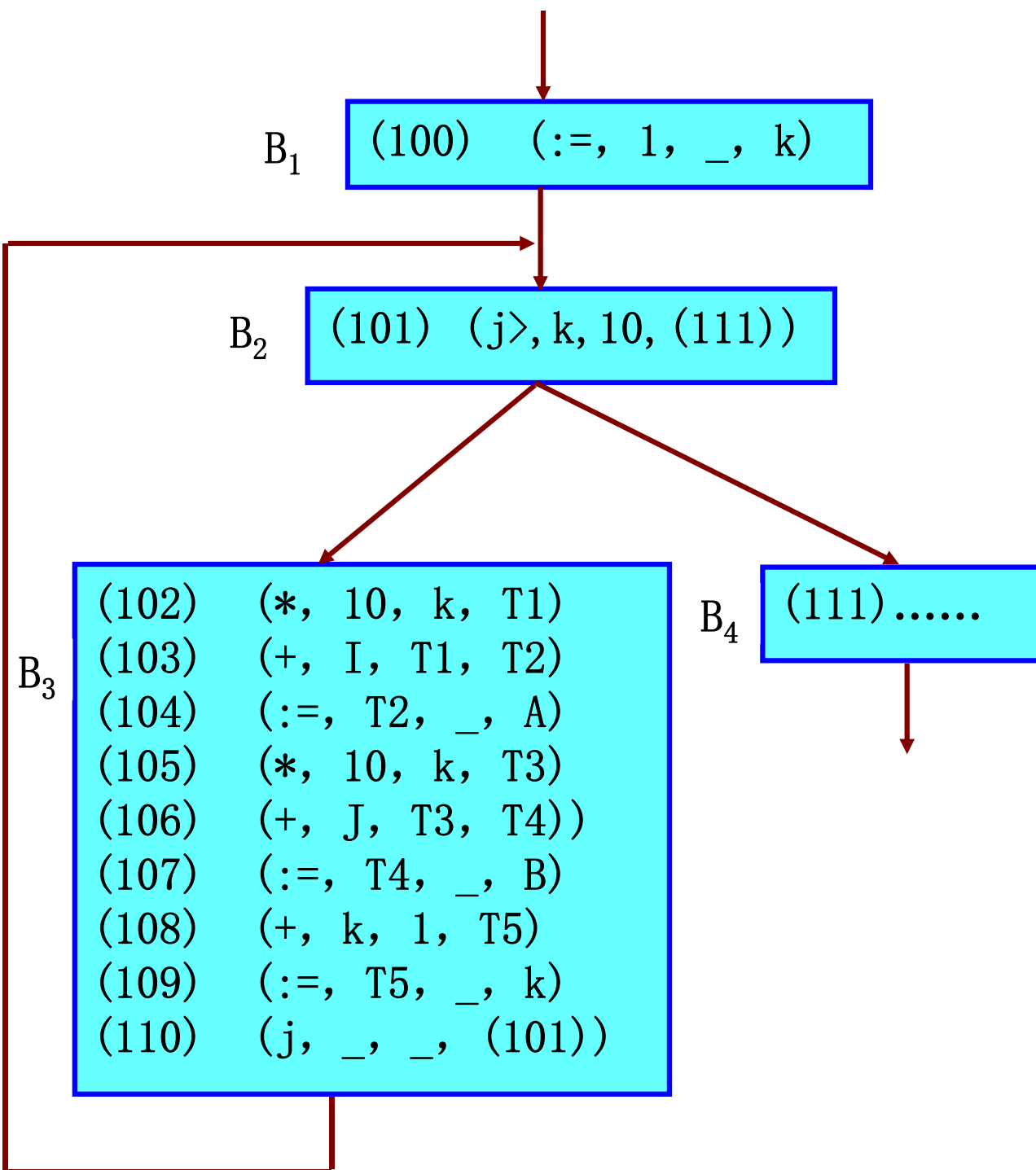
将循环中执行时间较长的运算替换为执行时间较短的运算。

◆ 删除归纳变量

如果循环中对变量 I 只有惟一的形如 $I := I \pm C$ 的赋值，且 C 为循环的**区域常量**，则称 I 为循环中的**基本归纳变量**。基本归纳变量除用于自身的递归定值外，往往只在循环中用来计算其它归纳变量以及用来控制循环的进行。

(100) (: =, 1, -, k)
(101) (j>, k, 10, (111))
(102) (*, 10, k, T1)
(103) (+, I, T1, T2)
(104) (: =, T2, -, A)
(105) (*, 10, k, T3)
(106) (+, J, T3, T4))
(107) (: =, T4, -, B)
(108) (+, k, 1, T5)
(109) (: =, T5, -, k)
(110) (j, -, -, (101))
(111)

划分基本块



B₁

(100) (:=, 1, _, k)

B₂

(101) (j>, k, 10, (111))

B₃

(102) (*, 10, k, T1)
(103) (+, I, T1, T2)
(104) (:=, T2, _, A)
(105) (*, 10, k, T3)
(106) (+, J, T3, T4))
(107) (:=, T4, _, B)
(108) (+, k, 1, T5)
(109) (:=, T5, _, k)
(110) (j, _, _, (101))

B₄

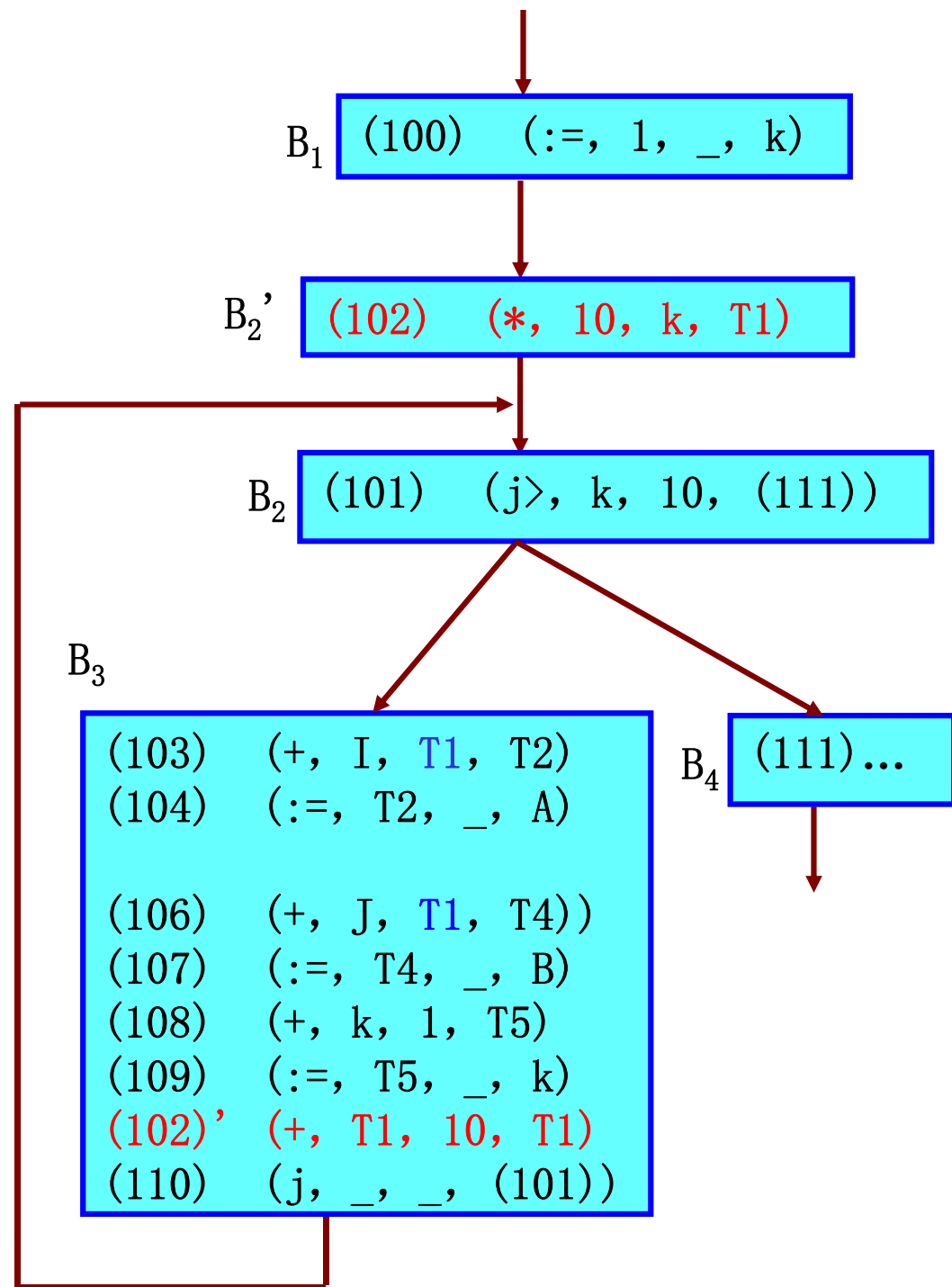
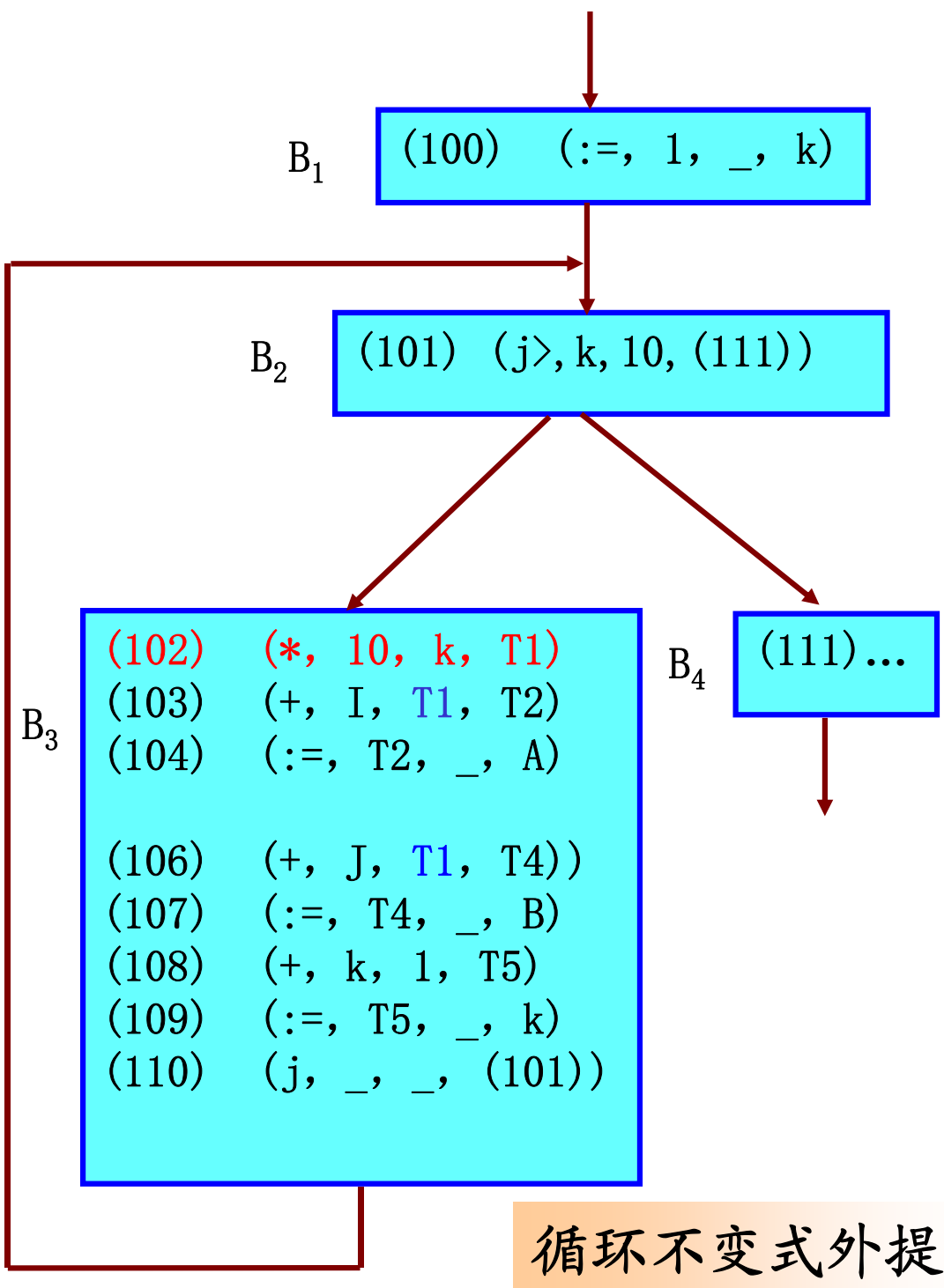
(111)

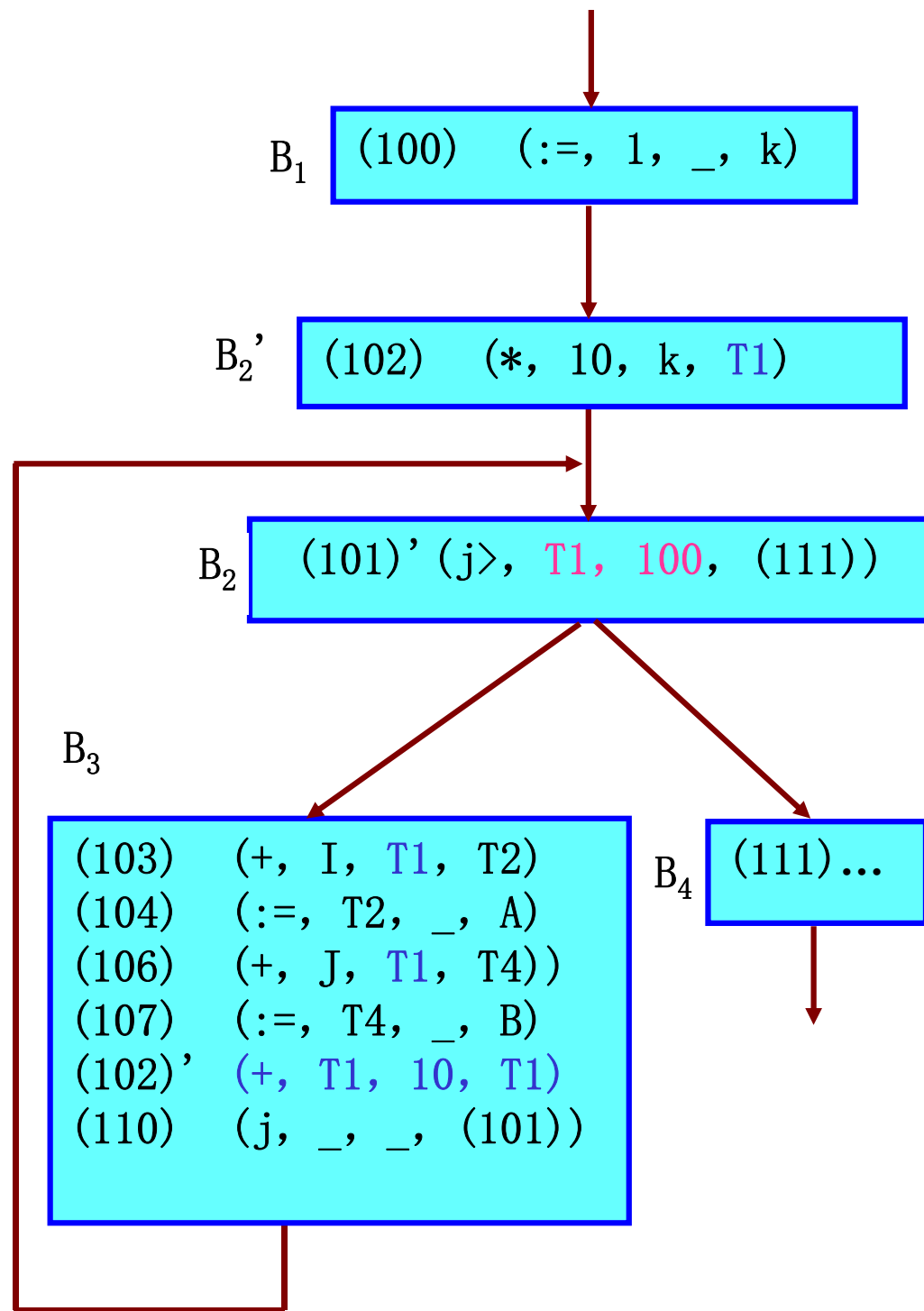
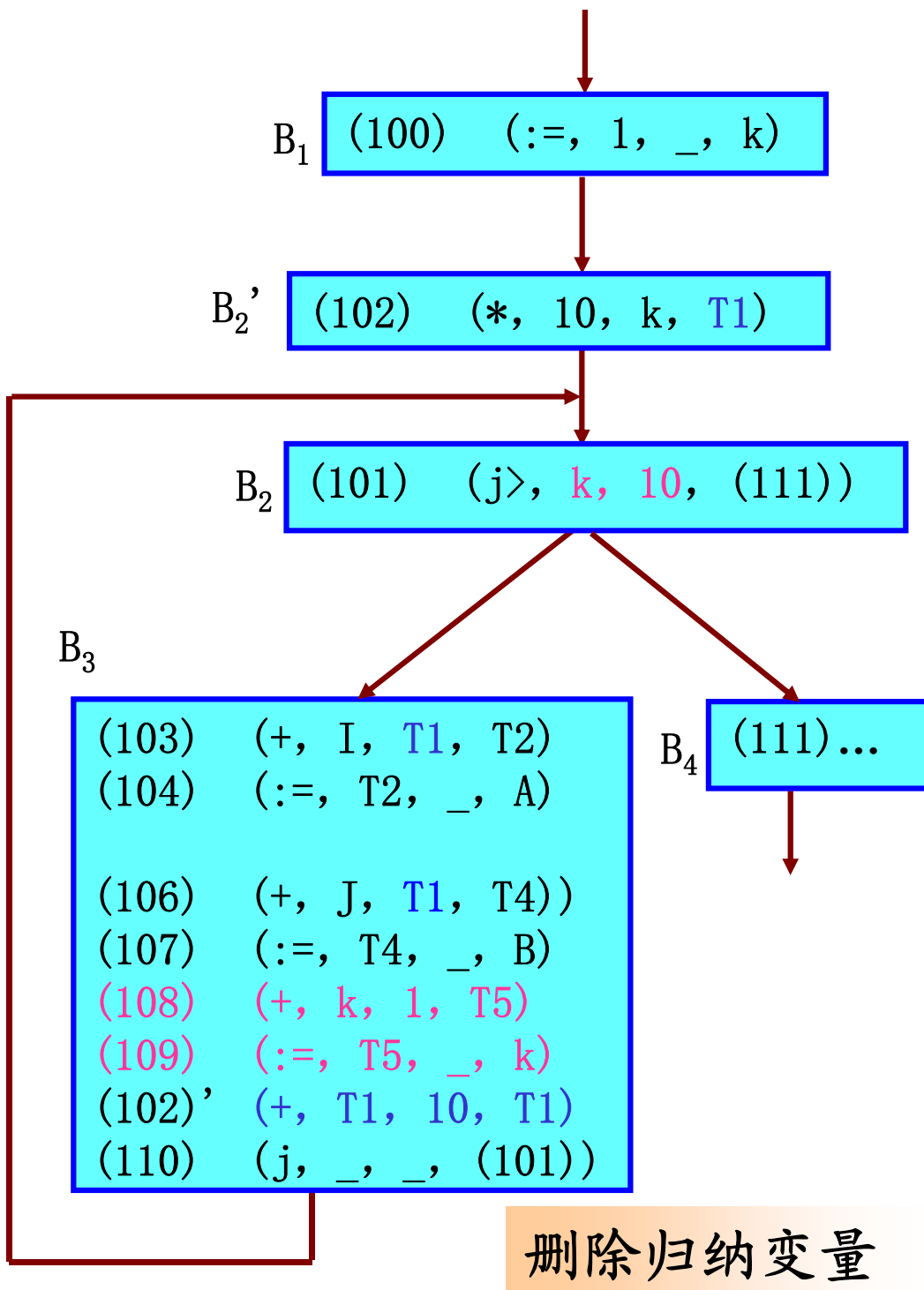
B₃

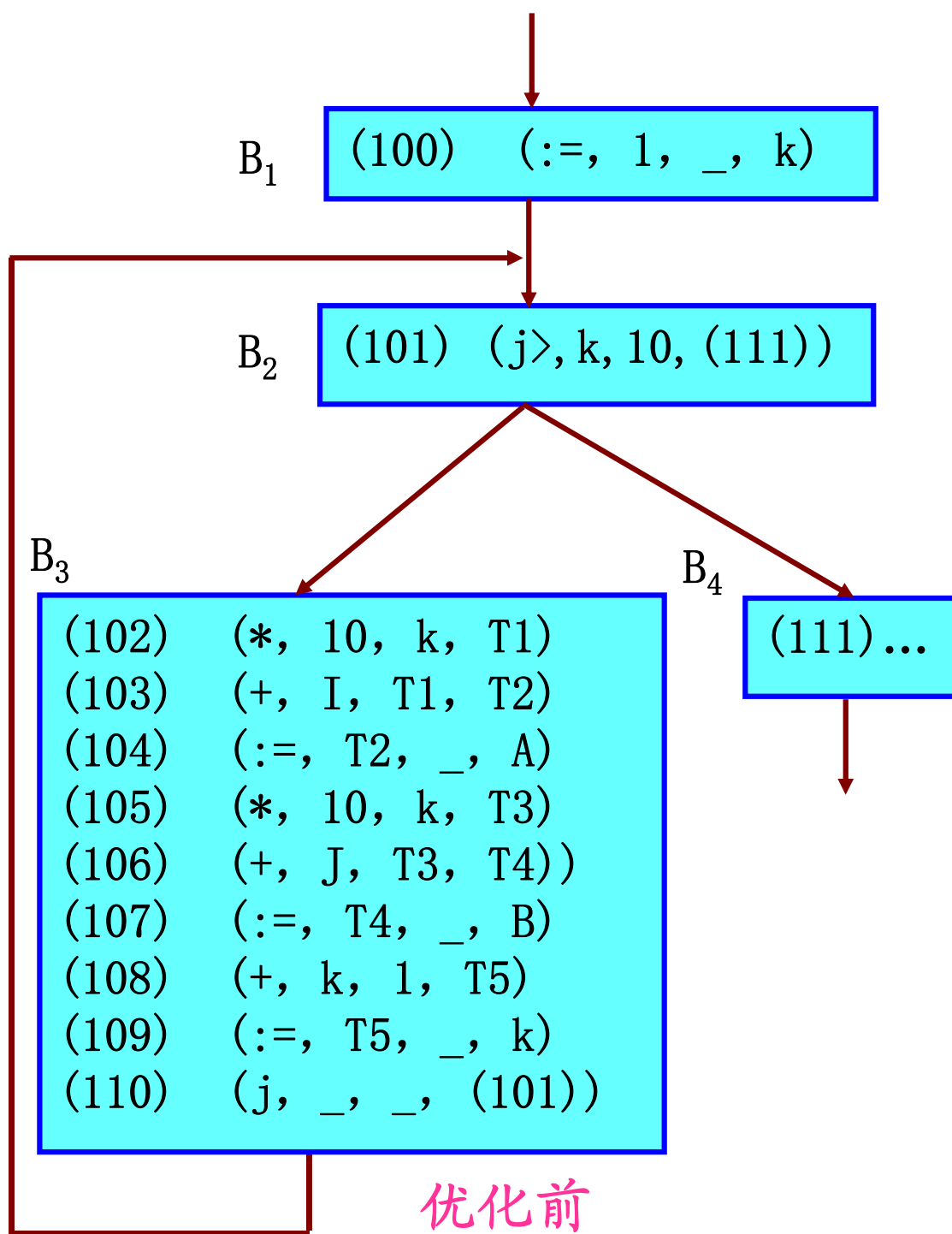
(102) (*, 10, k, T1)
(103) (+, I, T1, T2)
(104) (:=, T2, _, A)

(106) (+, J, T1, T4))
(107) (:=, T4, _, B)
(108) (+, k, 1, T5)
(109) (:=, T5, _, k)
(110) (j, _, _, (101))

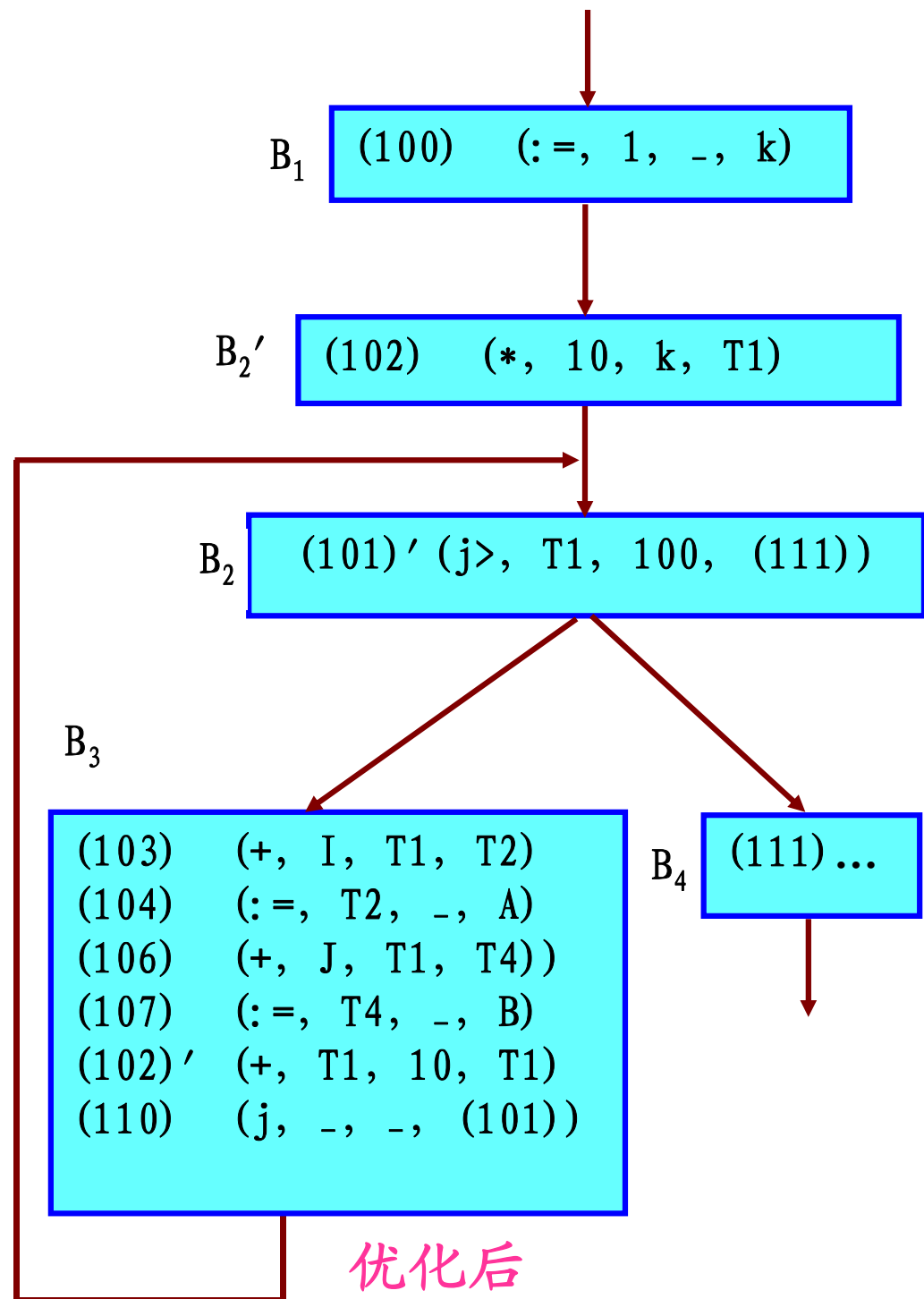
基本块优化







优化前



优化后

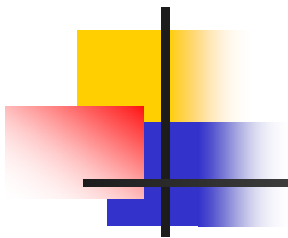
课后作业



- 试把下面的程序段划分为基本块并作出其程序流图。

```
A:=0;  
i:=1;  
h1: B:= j+1;  
      C:= B+i;  
      A:=C+A  
      if i>=100 then goto h2  
      i:=i+1;  
      goto h1;  
h2: write A;
```

- 对上题进行尽可能多的优化。



下节内容

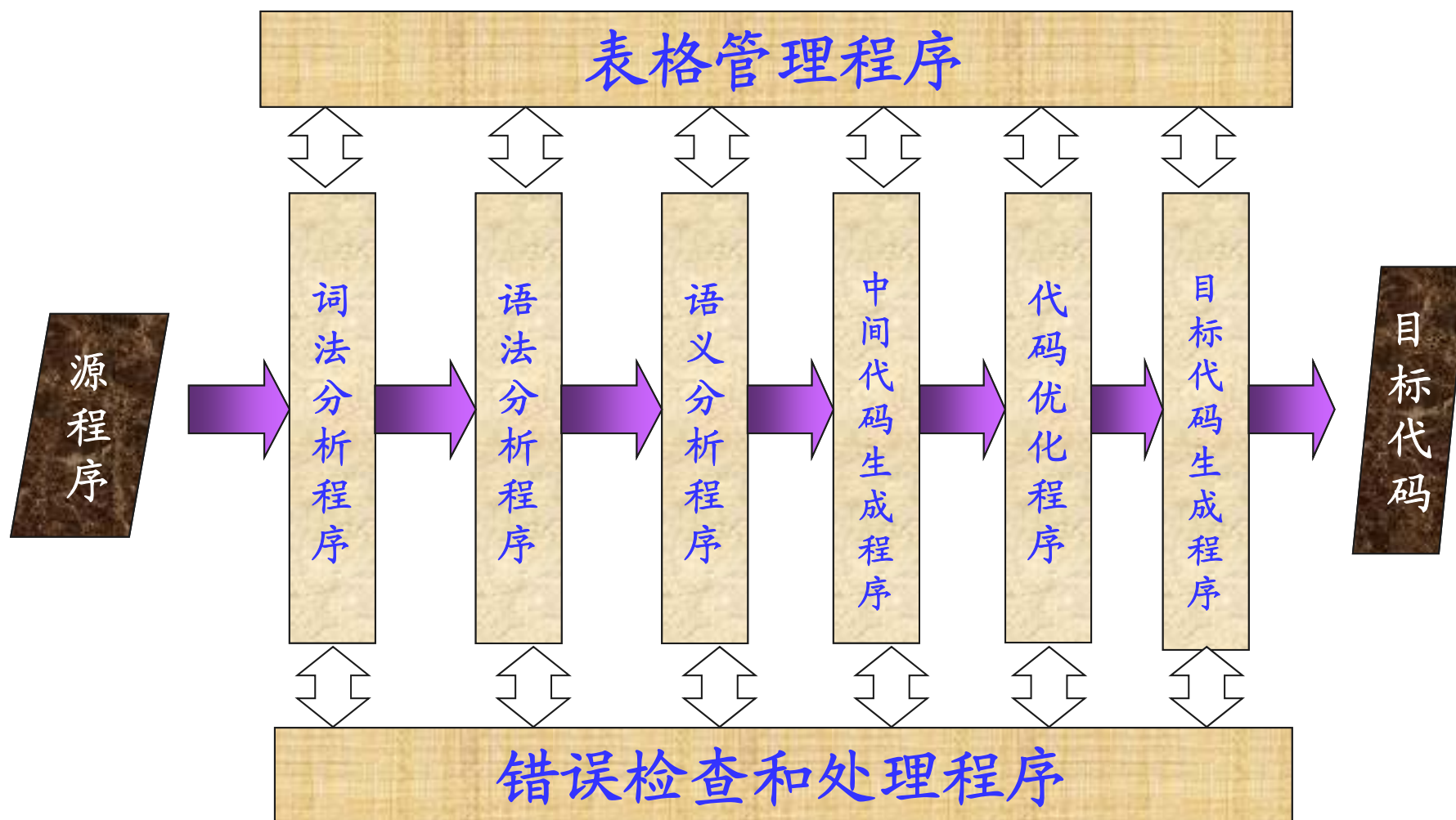
代码生成



编译原理

武汉大学计算机学院
编译原理课程组

编译程序的结构



编译过程——代码生成

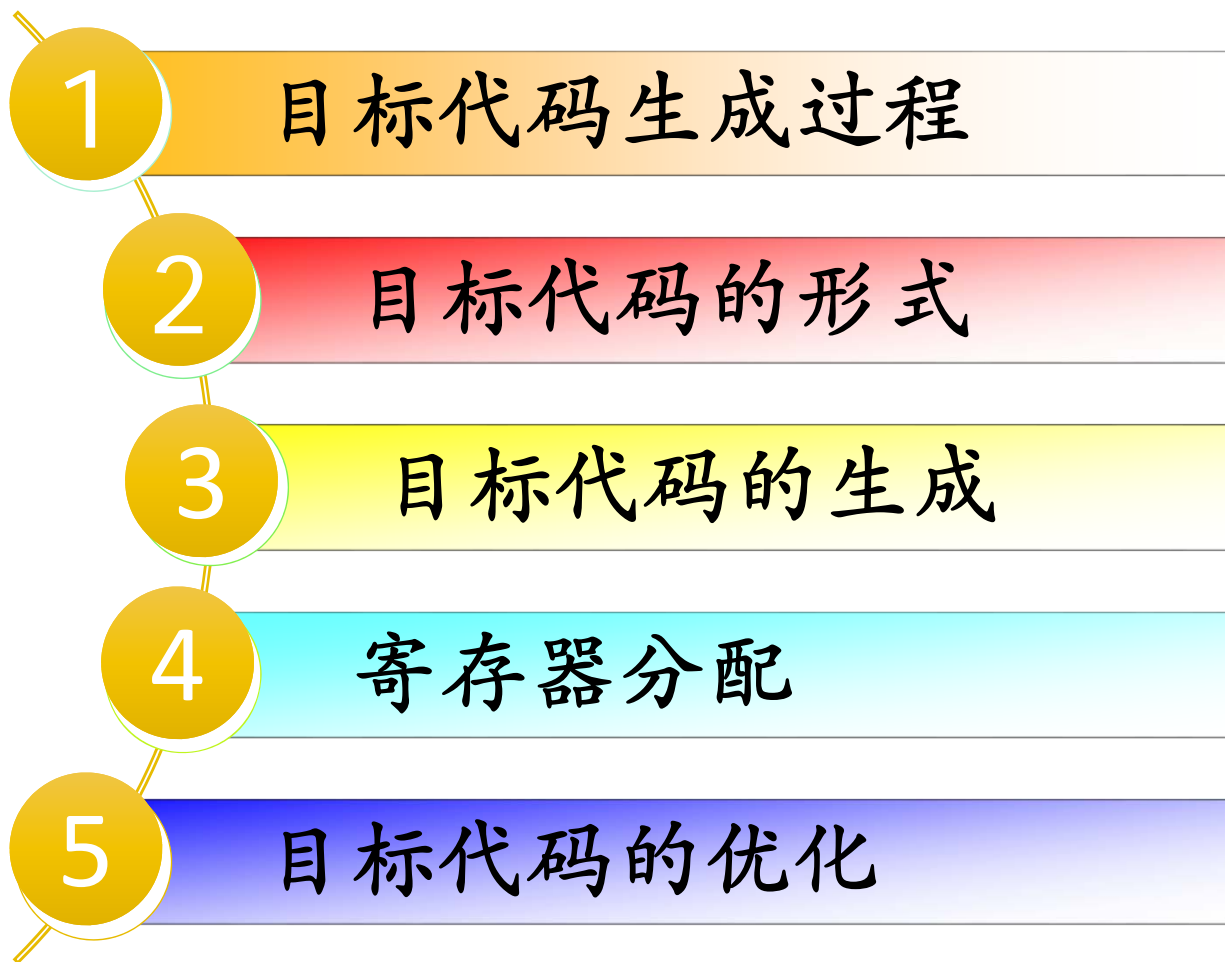
(1) (* , id3 60.0 t1)
(2) (+ , id2 t1 id1)

Target code generator

$id1 := id2 + id3 * 60$

```
movf    id3,  R2
mulf    #60.0, R2
movf    id2,  R1
addf    R2,   R1
movf    R1,   id1
```

第12章 代码生成



12.1 目标代码生成过程



- 着重考虑的两个问题：
 - 使生成的目标代码较短
 - 利用计算机的寄存器，减少目标代码访问存储单元的次数（目标代码的执行速度）



12.2 目标代码的形式

三种形式:

- (1) 可立即执行的机器语言代码
- (2) 待装配的机器语言模块
- (3) 汇编语言形式的代码



12.2 目标代码的形式

(1) 可立即执行的机器语言代码

- 可立即执行
- 不可重定位，灵活性较差
- 不能独立编译各程序块，整个源程序要一起编译



12.2 目标代码的形式

(2) 待装配的机器语言模块

- 执行前要进行代码定位
- 浮动的机器语言代码，比较灵活
- 可分别编译，可从目标模块中调用先前已编译好的其它程序模块



12.2 目标代码的形式

(3) 汇编语言形式的代码

- 汇编后执行
- 比前两种方式更灵活
- 可产生符号指令和利用宏机制来帮助生成代码



12.3 目标代码的生成

假想的计算机模型：中间代码形式 → 汇编语言代码

- ◆ 从四元式生成代码
- ◆ 从三元式生成代码
- ◆ 从树型表示生成代码
- ◆ 从逆波兰表示生成代码



从四元式生成代码

■ 假想计算机中的符号指令

指令		含义
LD	m	m单元内容送入累加器
ST	m	累加器内容存入m单元
ADD/MULT	m	m单元内容加（乘）到累加器中
SUB/DIV	m	累加器内容减去（除以）m单元内容
ABS		对累加器内容取绝对值
CHS		改变累加器内容的正负号

从四元式生成代码

■ 代码生成程序

- 四元式序列排列顺序与运算顺序相同
- 运行时所有运算都要在累加器中进行，生成代码前要知道累加器内容

过程/变量		含义
GEN	(w, x)	生成指令 “w x”
INACC	(A, B)	生成运算指令之前，将（第）一运算对象存入累加器
ACC		指明运行时刻累加器的状态（累加器中存放的变量名或临时变量）

```
Procedure INACC(A, B);  
  string A, B;  
  begin string T;  
    if ACC=" then  
      begin  
        GEN('LD', A);  
        ACC:=A;  
        return  
      end;  
    if ACC=B then  
      begin  
        T:=A; A:=B; B:=T  
      end;  
    else if ACC≠A then  
      begin  
        GEN('ST', ACC);  
        GEN('LD', A);  
        ACC:=A  
      end;  
  end
```

从四元式生成代码

- 加法（乘法）的代码生成程序
 - 运算量可交换的

```
INACC(quad(i).OPER1, quad(i).OPER2);
```

```
GEN('ADD', quad(i).OPER2);
```

```
ACC:= quad(i).RESULT;
```

```
INACC(quad(i).OPER1, quad(i).OPER2);
```

```
GEN('MULT', quad(i).OPER2);
```

```
ACC:= quad(i).RESULT;
```

```
Procedure INACC(A, B);  
string A, B;  
begin string T;  
  if ACC=' ' then  
    begin  
      GEN('LD', A);  
      ACC:=A;  
      return  
    end;  
  if ACC=B then  
    begin  
      T:=A; A:=B; B:=T  
    end;  
  else if ACC≠A then  
    begin  
      GEN('ST', ACC);  
      GEN('LD', A);  
      ACC:=A  
    end;  
end
```

从四元式生成代码

- 减法（除法）的代码生成程序
 - 运算量不可交换的

```
INACC(quad(i).OPER1, “”);
```

```
GEN(‘SUB’, quad(i).OPER2);
```

```
ACC:= quad(i).RESULT;
```

```
INACC(quad(i).OPER1, “”);
```

```
GEN(‘DIV’, quad(i).OPER2);
```

```
ACC:= quad(i).RESULT;
```

```
Procedure INACC(A, B);  
string A, B;  
begin string T;  
  if ACC=‘’ then  
    begin  
      GEN(‘LD’, A);  
      ACC:=A;  
      return  
    end;  
  if ACC=B then  
    begin  
      T:=A; A:=B; B:=T  
    end;  
  else if ACC≠A then  
    begin  
      GEN(‘ST’, ACC);  
      GEN(‘LD’, A);  
      ACC:=A  
    end;  
end
```

从四元式生成代码

- 单目减的代码生成程序
 - 一个运算量的

```
INACC(quad(i).OPER1, “”);
```

```
GEN(‘CHS’, “”);
```

```
ACC:= quad(i).RESULT;
```

```
Procedure INACC(A, B);  
string A, B;  
begin string T;  
  if ACC=‘’ then  
    begin  
      GEN(‘LD’, A);  
      ACC:=A;  
      return  
    end;  
  if ACC=B then  
    begin  
      T:=A; A:=B; B:=T  
    end;  
  else if ACC≠A then  
    begin  
      GEN(‘ST’, ACC);  
      GEN(‘LD’, A);  
      ACC:=A  
    end;  
end
```



从四元式生成代码

■ 举例: $A \times ((A \times B + C) - (C \times D))$

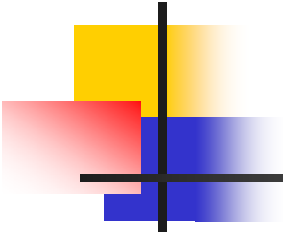
四元式	代码	ACC内容
* A B T1	LD A	
	MULT B	T1
+ T1 C T2	ADD C	T2
* C D T3	ST T2	
	LD C	
	MULT D	T3
- T2 T3 T4	ST T3	
	LD T2	
	SUB T3	T4
* A T4 T5	MULT A	T5



12.4 寄存器分配

■ 寄存器分配原则

- 计算表达式时，使所需的寄存器个数最少（存取指令条数）
 - 需要使用变量 v 值，考虑寄存器的分配
- 基本块代码生成需遵循的寄存器分配一般原则
 - 尽量使用寄存器保存变量的值或计算结果
 - 在基本块的出口，将变量的值存放到内存中
 - 尽早释放基本块内不再被引用的变量所占用寄存器



12.4 寄存器分配

- 考虑循环，按照什么标准分配寄存器呢？——访问主存单元的次数
 - 指令的**执行代价** = 指令访问主存单元次数 + 1

操作码	操作数1	操作数2		执行代价
OP	Rj	Ri		1
OP	Rj	M		2
OP	Rj	*Ri		2
OP	Rj	*M		3

- 分配中尽量把变量值保存在寄存器中，减少对内存的访问



12.5 目标代码的优化

窥孔优化（ peephole optimization ）

◆ 删除冗余存取

(1) **ST R0, A**

(2) **LD R0, A**



12.5 目标代码的优化

窥孔优化（ peephole optimization ）

◆ 控制流优化

```
goto L1  
.....  
L1: goto L2
```



12.5 目标代码的优化

窥孔优化（ peephole optimization ）

◆ 删除无用操作

```
ADD R, #0
```

```
.....
```

```
MUL R, #1
```

小 结

