



中国科学技术大学  
University of Science and Technology of China



# 《编译原理与技术》

## 中间代码

计算机科学与技术学院

李 诚

2021-10-11



# 为什么需要中间代码表示?

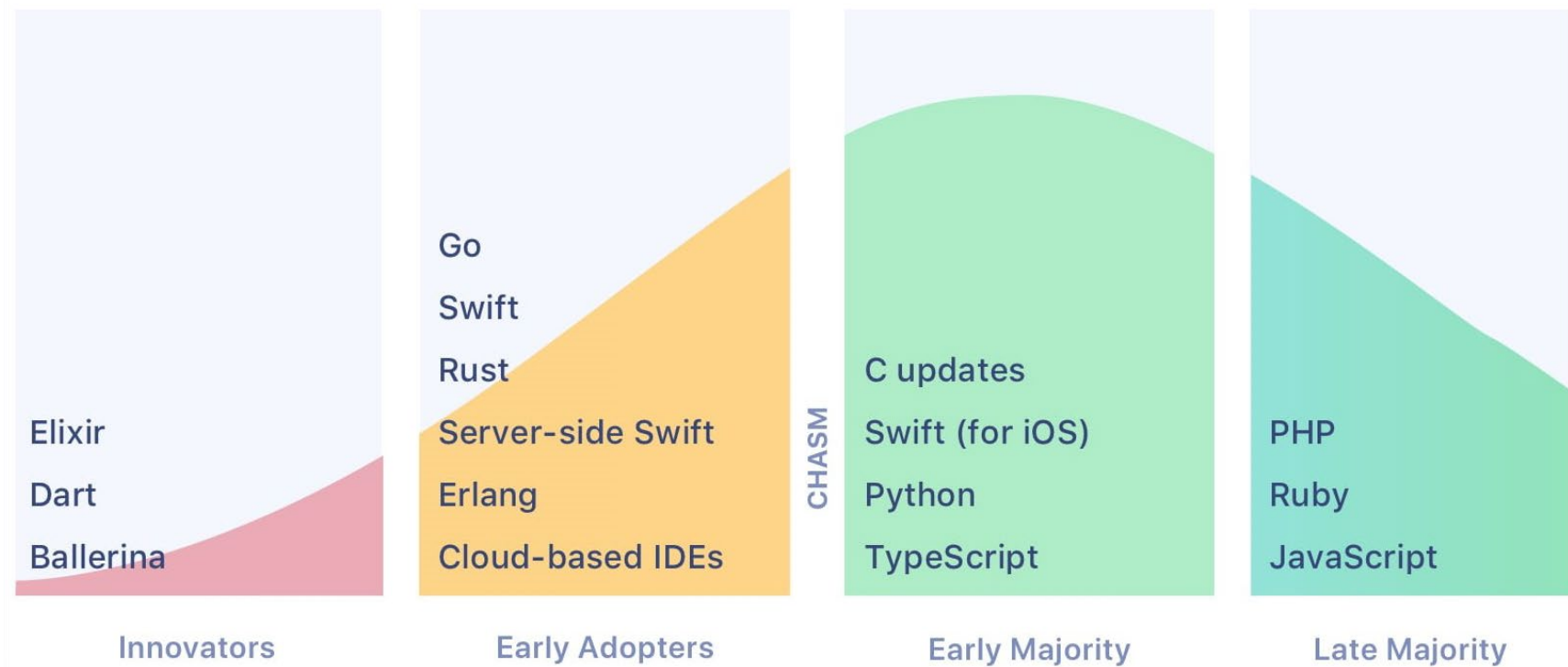


中国科学技术大学  
University of Science and Technology of China

## Software Development Programming Languages Trends 2019 Q3 Graph

<http://info.link/proglang2019>

InfoQ





# 为什么需要中间代码表示?



中国科学技术大学  
University of Science and Technology of China

分类	名称	版本	扩展	初始年份
CISC	x86	16, 32, 64 (16→32→64)	x87, IA-32, MMX, 3DNow!, SSE, SSE2, PAE, x86-64, SSE3, SSSE3, SSE4, BMI, AVX, AES, FMA, XOP, F16C	1978
RISC	MIPS	32	<a href="#">MDMX</a> , <a href="#">MIPS-3D</a>	1981
VLIW	Elbrus	64	Just-in-time dynamic translation: x87, IA-32, MMX, SSE, SSE2, x86-64, SSE3, AVX	2014

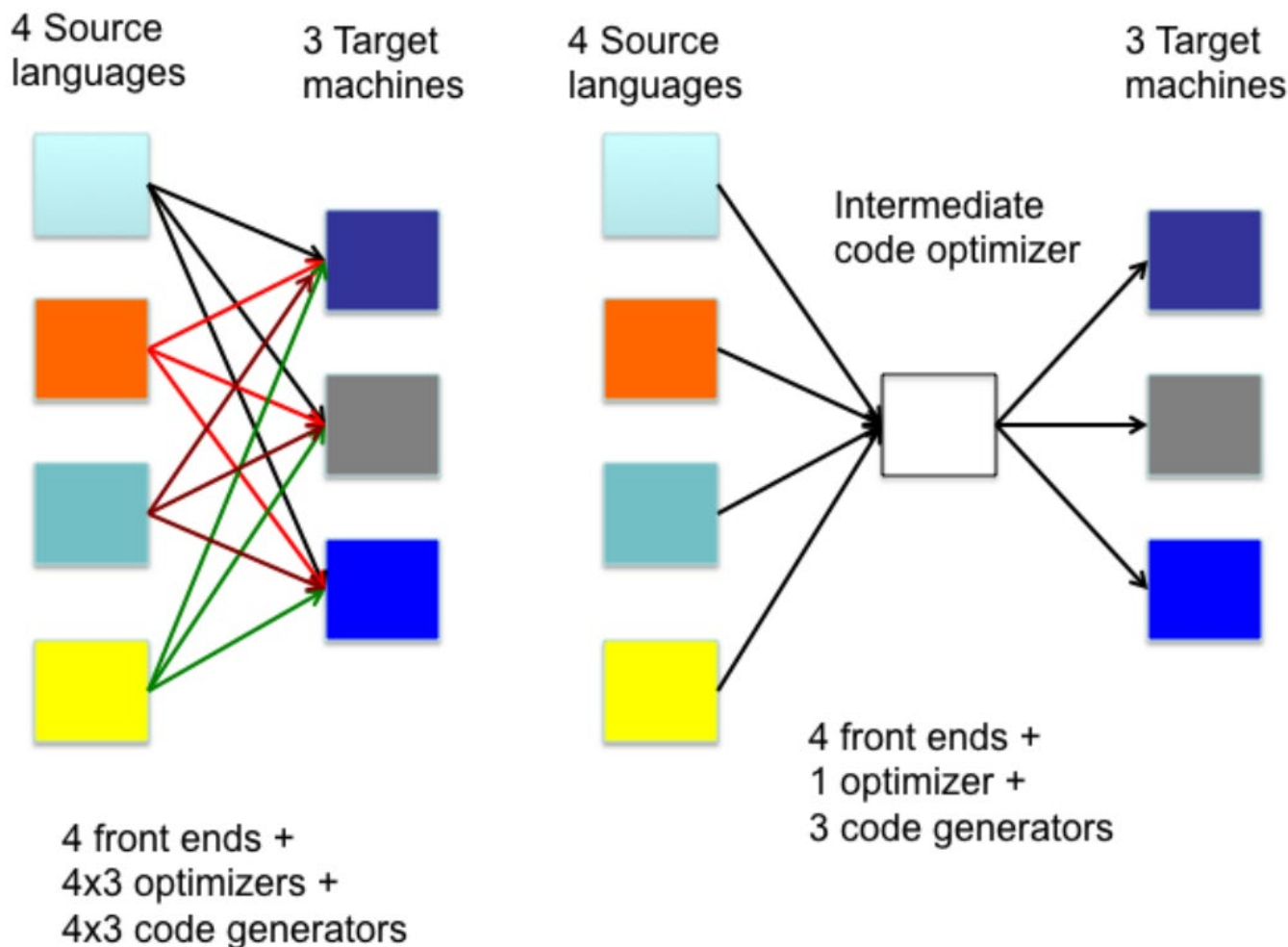
指令集体系结构(ISA)的发展



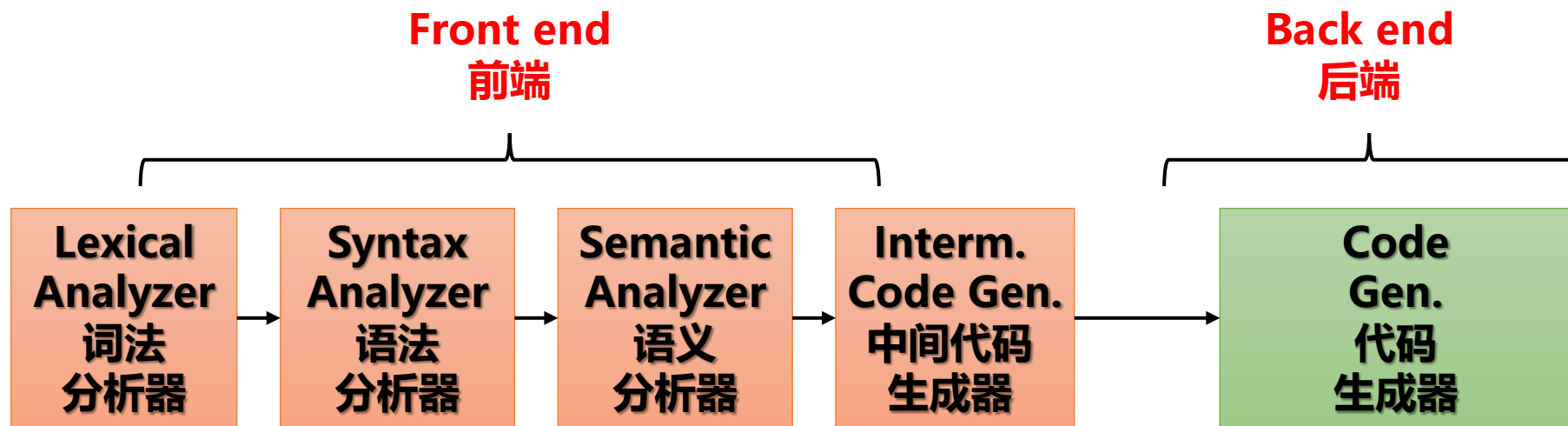
# 为什么需要中间代码表示？



中国科学技术大学  
University of Science and Technology of China

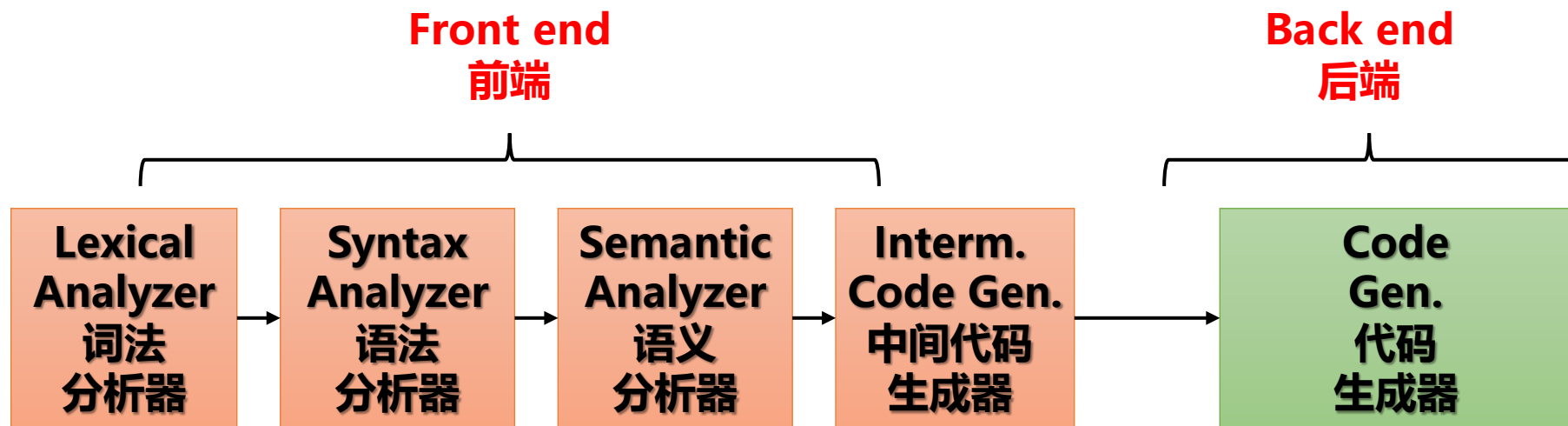


**实践过程中，推陈出新的语言、不断涌现的指令集、开发成本之间的权衡**



## □前端与后端分离

- ❖ 不同的源语言、不同的机器可以得到不同的编译优化组合
- ❖ 前端只关注和分析与源语言相关的细节，与目标机器无关



## □前端与后端分离

❖ 为新机器构建编译器，只需要设计从中间代码到新的目标机器代码的编译器 (前端独立)

## □中间代码优化与源语言和目标机器均无关



□简而言之，编译器任何完整的中间输出都是中间代码表示形式

□常见类型有：

❖ 后缀表示

❖ 语法树或DAG图

❖ 三地址码(TAC)

❖ 静态单赋值形式(SSA)

重点关注  
LLVM IR是TAC类型



*uop*是一元运算符

$$E \rightarrow E \text{ op } E \mid uop E \mid (E) \mid \text{id} \mid \text{num}$$

表达式  $E$

$\text{id}$

$\text{num}$

$E_1 \text{ op } E_2$

$uop E$

$(E)$

后缀式  $E'$

$\text{id}$

$\text{num}$

$E'_1 E'_2 \text{ op}$

$E' uop$

$E'$





## □ 后缀表示不需要括号

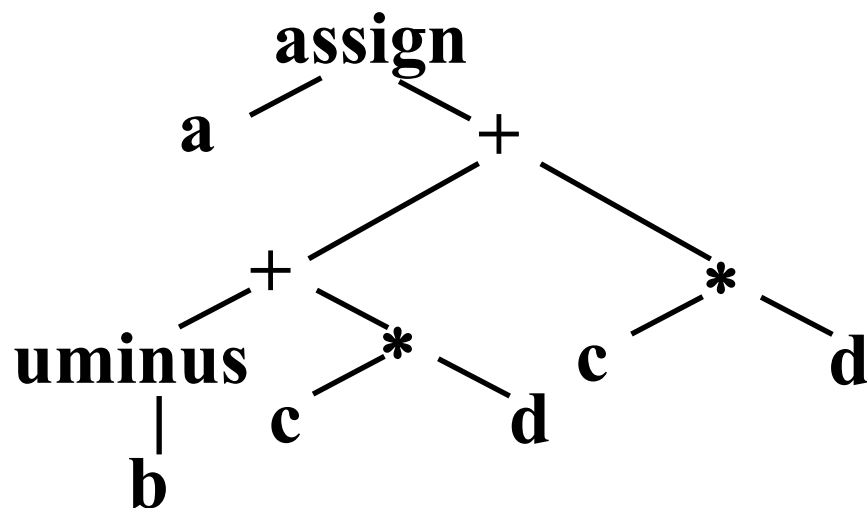
❖  $(8 - 5) + 2$  的后缀表示是  $8\ 5\ -2\ +$

## □ 后缀表示的最大优点是便于计算机处理表达式

计算栈	输入串
	$8\ 5\ -2\ +$
8	$5\ -2\ +$
8 5	$-2\ +$
3	$2\ +$
3 2	$+$
5	



## □语法树是一种图形化的中间表示

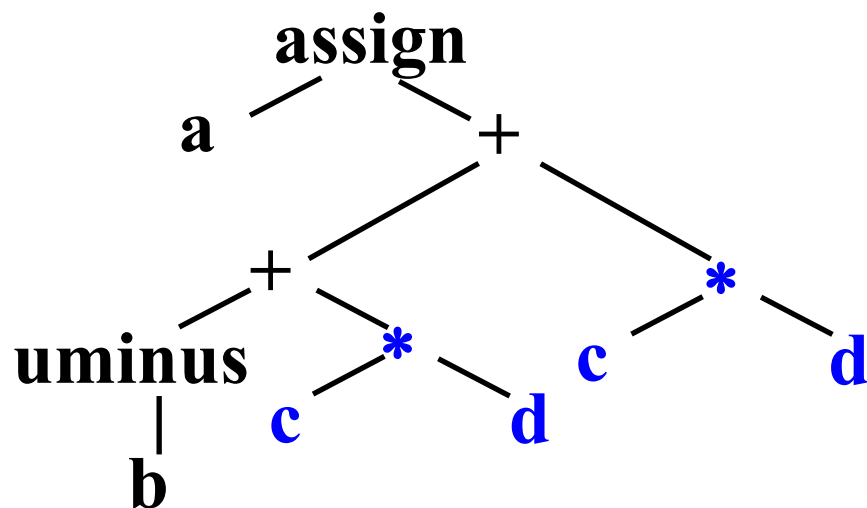


(a) 语法树

$a = (-b + c*d) + c*d$ 的图形表示



## □语法树是一种图形化的中间表示



(a) 语法树

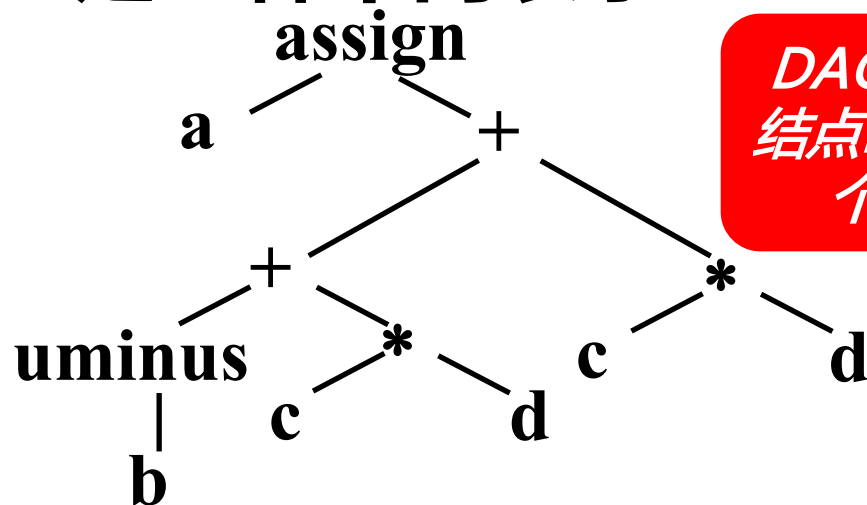
$c*d$ 是公共子  
表达式

$a = (-b + c*d) + c*d$ 的图形表示



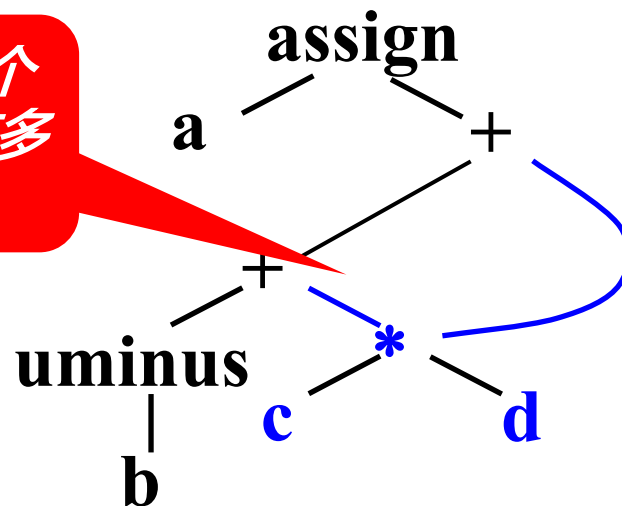
□语法树是一种图形化的中间表示

□有向无环图(Directed Acyclic Graph, DAG)也是一种中间表示



(a) 语法树

DAG中的一个  
结点N可能有多  
个父结点



(b) DAG

$a = (-b + c*d) + c*d$ 的图形表示



## □ 三地址代码 (Three-Address Code, TAC)

一般形式:  $x = y \text{ op } z$

- 最多一个算符
- 最多三个计算分量
- 每一个分量代表一个地址, 因此三地址

## □ 例 表达式 $x + y * z$ 翻译成的三地址语句序列

$$t_1 = y * z$$

$$t_2 = x + t_1$$



## □三地址代码是语法树或DAG的一种线性表示

❖例  $a = (-b + c * d) + c * d$

语法树的代码

$$t_1 = -b$$

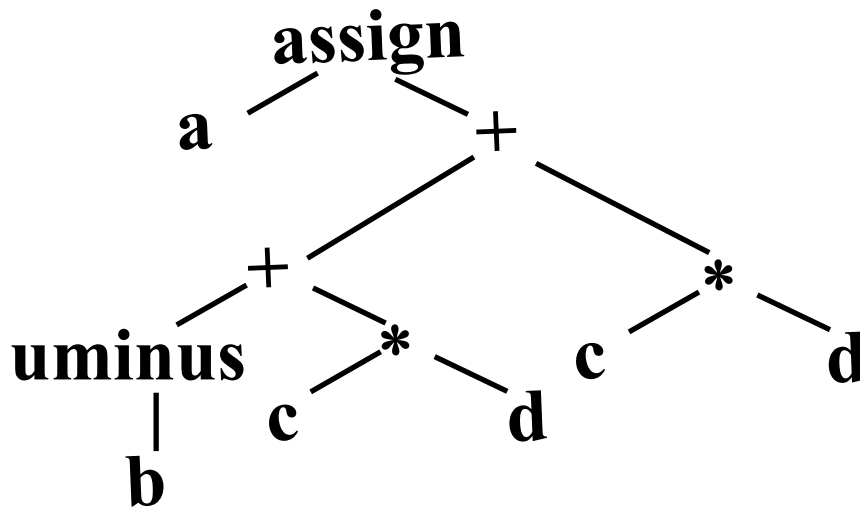
$$t_2 = c * d$$

$$t_3 = t_1 + t_2$$

$$t_4 = c * d$$

$$t_5 = t_3 + t_4$$

$$a = t_5$$





## □三地址代码是语法树或DAG的一种线性表示

❖例  $a = (-b + c * d) + c * d$

语法树的代码

$$t_1 = -b$$

$$t_2 = c * d$$

$$t_3 = t_1 + t_2$$

$$t_4 = c * d$$

$$t_5 = t_3 + t_4$$

$$a = t_5$$

DAG的代码

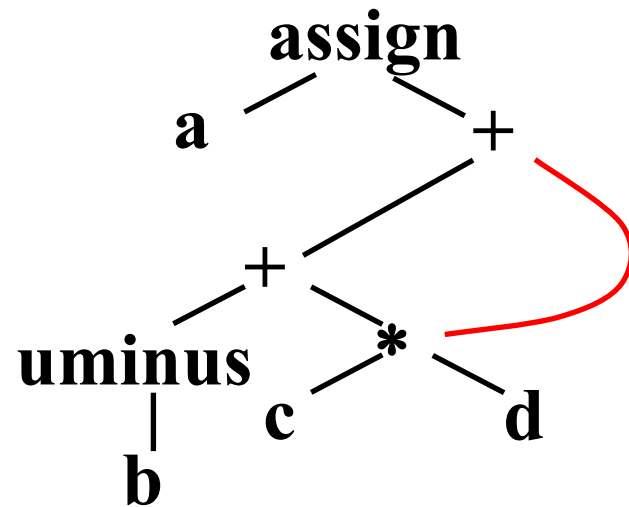
$$t_1 = -b$$

$$t_2 = c * d$$

$$t_3 = t_1 + t_2$$

$$t_4 = t_3 + t_2$$

$$a = t_4$$





## □常用的三地址语句

- ❖ 运算/赋值语句  $x = y \text{ op } z, \quad x = \text{op } y, \quad x = y$
- ❖ 无条件转移 `goto L`
- ❖ 条件转移1 `if x goto L, if False x goto L`
- ❖ 条件转移2 `if x relop y goto L`





## □常用的三地址语句

### ❖过程调用

- $\text{param } x_1$  //设置参数
- $\text{param } x_2$
- ...
- $\text{param } x_n$
- $\text{call } p, n$  //调用子过程 $p$ ,  $n$ 为参数个数

### ❖过程返回

$\text{return } y$

### ❖索引赋值

$x = y[i]$  和  $x[i] = y$

- 注意:  $i$ 表示距离 $y$ 处 $i$ 个内存单元

### ❖地址和指针赋值 $x = \&y$ , $x = *y$ 和 $*x = y$



□考虑语句，令数组a的每个元素占8存储单元

❖do  $i = i + 1$ ; while ( $a[i] < v$ );

```
L:   $t_1 = i + 1$   
     $i = t_1$   
     $t_2 = i * 8$   
     $t_3 = a[t_2]$   
    if  $t_3 < v$  goto L
```

符号标号

```
100:  $t_1 = i + 1$   
101:  $i = t_1$   
102:  $t_2 = i * 8$   
103:  $t_3 = a[t_2]$   
104: if  $t_3 < v$  goto 100
```

位置标号



□三地址代码只说明了指令的组成部分，我们还需关注其在编译器中的具体数据结构实现

□常见的实现方式有三种：

❖四元式： (op, arg1, arg2, result)

❖三元式： (op, arg1, arg2)

❖间接三元式： (三元式的指针表)



## □四元式(Quadruple)

❖包括4个字段:

$op \quad arg_1 \quad arg_2 \quad result$

❖ $op$ : 运算符的内部编码

❖ $arg_1 \quad arg_2 \quad result$ 是地址

❖例如:  $x = y + z$  的四元式  $+ \quad y \quad z \quad x$



## □四元式(Quadruple)

❖例： $a = b * -c + b * -c$

$t_1 = \text{minus } c$

$t_2 = b * t_1$

$t_3 = \text{minus } c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

	<i>op</i>	<i>arg</i> <sub>1</sub>	<i>arg</i> <sub>2</sub>	<i>result</i>
0	minus	c		$t_1$
1	*	b	$t_1$	$t_2$
2	minus	c		$t_3$
3	*	b	$t_3$	$t_4$
4	+	$t_2$	$t_4$	$t_5$
5	=	$t_5$		a
	...			

**缺点：临时变量太多，增加时间和空间成本**



## □三元式(Triple)

❖格式:  $op \ arg_1 \ arg_2$

❖将存储结果的临时变量隐藏起来, 用指令所代表的的位位置来表示其运算结果

❖ $x = y \ op \ z$  将被拆分为(?是编号)

➤  $? \ op \ y \ z$

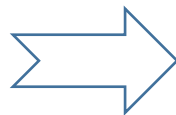
➤  $x = (?)$



## □三元式(Triple)

❖例： $a = b * -c + b * -c$

	<i>op</i>	<i>arg</i> <sub>1</sub>	<i>arg</i> <sub>2</sub>	<i>result</i>
0	minus	c		t <sub>1</sub>
1	*	b	t <sub>1</sub>	t <sub>2</sub>
2	minus	c		t <sub>3</sub>
3	*	b	t <sub>3</sub>	t <sub>4</sub>
4	+	t <sub>2</sub>	t <sub>4</sub>	t <sub>5</sub>
5	=	t <sub>5</sub>		a
...				



	<i>op</i>	<i>arg</i> <sub>1</sub>	<i>arg</i> <sub>2</sub>
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
...			

**缺点：隐式的临时变量，代码位置调整会造成引用该位置的代码也要修改。**



## □ 间接三元式 (Indirect triple)

❖ 例:  $a = b * -c + b * -c$

*instruction*

35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

指令序列可以任意调整顺序

	<i>op</i>	<i>arg<sub>1</sub></i>	<i>arg<sub>2</sub></i>
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

**优势：比四元式空间开销小，比三元式更灵活**





# 三地址代码的实现方式总结



中国科学技术大学  
University of Science and Technology of China

四元式	按编号次序计算	计算结果存于 <b>result</b>	方便移动，计算 次序容易调整	大量引入临时变量
三元式	按编号次序计算	由编号代表	不方便移动	在代码生成时进行临时变量的分配
间接三元式	按指令列表次序计算	由编号代表	方便移动，计算 次序容易调整	在代码生成时进行临时变量的分配



□ 一种便于某些代码优化的中间表示

□ 和三地址代码的主要区别

❖ 所有赋值指令都是对不同名字的变量的赋值

三地址代码

$$p = a + b$$

$$q = p - c$$

$$p = q * d$$

$$p = e - p$$

$$q = p + q$$

静态单赋值形式

$$p_1 = a + b$$

$$q_1 = p_1 - c$$

$$p_2 = q_1 * d$$

$$p_3 = e - p_2$$

$$q_2 = p_3 + q_1$$

SSA由Barry K. Rosen、Mark N. Wegman和  
F. Kenneth Zadeck于1988年提出



□ 一种便于某些代码优化的中间表示

□ 和三地址代码的主要区别

❖ 所有赋值指令都是对不同名字的变量的赋值

❖ 同一个变量在不同控制流路径上都被定值

```
if (flag) x = -1; else x = 1;
```

```
y = x * a;
```

改成

```
if (flag)  $x_1 = -1$ ; else  $x_2 = 1$ ;
```

```
 $x_3 = \phi(x_1, x_2);$  // 由flag的值决定用 $x_1$ 还是 $x_2$ 
```

```
y =  $x_3$  * a;
```

## 快速排序程序片段如下

$i = m - 1; j = n; v = a[n];$

**while (1) {**

**do  $i = i + 1$ ; while( $a[i] < v$ );**

**do  $j = j - 1$ ; while ( $a[j] > v$ );**

**if ( $i \geq j$ ) break;**

**$x = a[i]; a[i] = a[j]; a[j] = x;$**

**}**

**$x = a[i]; a[i] = a[n]; a[n] = x;$**

(1)  $i := m - 1$

(2)  $j := n$

(3)  $t1 := 4 * n$

(4)  $v := a[t1]$

(5)  $i := i + 1$

(6)  $t2 := 4 * i$

(7)  $t3 := a[t2]$

(8) if  $t3 < v$  goto (5)

(9)  $j := j - 1$

(10)  $t4 := 4 * j$

(11)  $t5 := a[t4]$

(12) if  $t5 > v$  goto (9)

(13) if  $i \geq j$  goto (23)

(14)  $t6 := 4 * i$

(15)  $x := a[t6]$

(16)  $t7 := 4 * i$

(17)  $t8 := 4 * j$

(18)  $t9 := a[t8]$

(19)  $a[t7] := t9$

(20)  $t10 := 4 * j$

(21)  $a[t10] := x$

(22) goto (5)

(23)  $t11 := 4 * i$

(24)  $x := a[t11]$

(25)  $t12 := 4 * i$

(26)  $t13 := 4 * n$

(27)  $t14 := a[t13]$

(28)  $a[t12] := t14$

(29)  $t15 := 4 * n$

(30)  $a[t15] := x$



**□连续的三地址指令序列，控制流从它的开始进入，并从它的末尾离开，中间没有停止或分支的可能性（末尾除外）**



□输入：三地址指令序列

□输出：基本块列表

□算法：

❖首先确定基本块的第一个指令，即**首指令(leader)**

➤指令序列的**第一条三地址指令**是一个首指令

➤任意转移指令的**目标指令**是一个首指令

➤紧跟一个**转移指令的指令**是一个首指令

❖然后，每个首指令对应的基本块包括了从它自己开始，直到**下一个首指令(不含)或指令序列结尾**之间的所有指令



# 举例



```
(1) i := m - 1
(2) j := n
(3) t1 := 4 * n
(4) v := a[t1]
(5) i := i + 1
(6) t2 := 4 * i
(7) t3 := a[t2]
(8) if t3 < v goto (5)
(9) j := j - 1
(10) t4 := 4 * j
(11) t5 := a[t4]
(12) if t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 := 4 * i
(15) x := a[t6]
```

```
(16) t7 := 4 * i
(17) t8 := 4 * j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4 * j
(21) a[t10] := x
(22) goto (5)
(23) t11 := 4 * i
(24) x := a[t11]
(25) t12 := 4 * i
(26) t13 := 4 * n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4 * n
(30) a[t15] := x
```



# 举例——首指令



(1)  $i := m - 1$

(2)  $j := n$

(3)  $t1 := 4 * n$

(4)  $v := a[t1]$

(5)  $i := i + 1$

(6)  $t2 := 4 * i$

(7)  $t3 := a[t2]$

(8) if  $t3 < v$  goto (5)

(9)  $j := j - 1$

(10)  $t4 := 4 * j$

(11)  $t5 := a[t4]$

(12) if  $t5 > v$  goto (9)

(13) if  $i \geq j$  goto (23)

(14)  $t6 := 4 * i$

(15)  $x := a[t6]$

(16)  $t7 := 4 * i$

(17)  $t8 := 4 * j$

(18)  $t9 := a[t8]$

(19)  $a[t7] := t9$

(20)  $t10 := 4 * j$

(21)  $a[t10] := x$

(22) goto (5)

(23)  $t11 := 4 * i$

(24)  $x := a[t11]$

(25)  $t12 := 4 * i$

(26)  $t13 := 4 * n$

(27)  $t14 := a[t13]$

(28)  $a[t12] := t14$

(29)  $t15 := 4 * n$

(30)  $a[t15] := x$





# 举例——基本块



**B<sub>1</sub>**

(1)  $i := m - 1$   
(2)  $j := n$   
(3)  $t1 := 4 * n$   
(4)  $v := a[t1]$

---

**B<sub>2</sub>**

(5)  $i := i + 1$   
(6)  $t2 := 4 * i$   
(7)  $t3 := a[t2]$   
(8) if  $t3 < v$  goto (5)

---

**B<sub>3</sub>**

(9)  $j := j - 1$   
(10)  $t4 := 4 * j$   
(11)  $t5 := a[t4]$   
(12) if  $t5 > v$  goto (9)

---

**B<sub>4</sub>**

(13) if  $i \geq j$  goto (23)  
(14)  $t6 := 4 * i$   
(15)  $x := a[t6]$

(16)  $t7 := 4 * i$   
(17)  $t8 := 4 * j$   
(18)  $t9 := a[t8]$   
(19)  $a[t7] := t9$   
(20)  $t10 := 4 * j$   
(21)  $a[t10] := x$   
(22) goto (5)

---

**B<sub>5</sub>**

(23)  $t11 := 4 * i$   
(24)  $x := a[t11]$   
(25)  $t12 := 4 * i$   
(26)  $t13 := 4 * n$   
(27)  $t14 := a[t13]$   
(28)  $a[t12] := t14$   
(29)  $t15 := 4 * n$   
(30)  $a[t15] := x$



- 流图的结点是一些基本块
- 从基本块 $B$ 到基本块 $C$ 之间有一条边，当且仅当 $C$ 的第一个指令可能紧跟在 $B$ 的最后一条指令之后执行
  - ❖  $B$ 是 $C$ 的前驱 (predecessor)
  - ❖  $C$ 是 $B$ 的后继 (successor)



- 流图的结点是一些基本块
- 从基本块 $B$ 到基本块 $C$ 之间有一条边，当且仅当 $C$ 的第一个指令可能紧跟在 $B$ 的最后一条指令之后执行，**判定方法如下**：
  - ❖ 有一个从 $B$ 的结尾跳转到 $C$ 的开头的跳转指令
  - ❖ 参考原来三地址指令序列中的顺序， $C$ 紧跟在 $B$ 之后，且 $B$ 的结尾没有无条件跳转指令

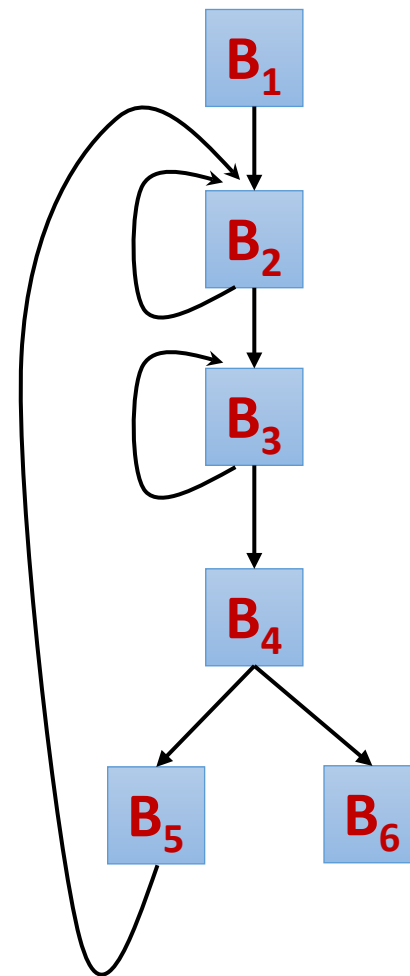


# 举例——流图



(1)  $i := m - 1$   
 $B_1$  (2)  $j := n$   
(3)  $t1 := 4 * n$   
(4)  $v := a[t1]$   
-----  
 $B_2$  (5)  $i := i + 1$   
(6)  $t2 := 4 * i$   
(7)  $t3 := a[t2]$   
(8) if  $t3 < v$  goto (5)  
-----  
 $B_3$  (9)  $j := j - 1$   
(10)  $t4 := 4 * j$   
(11)  $t5 := a[t4]$   
(12) if  $t5 > v$  goto (9)  
-----  
 $B_4$  (13) if  $i \geq j$  goto (23)  
(14)  $t6 := 4 * i$   
(15)  $x := a[t6]$

(16)  $t7 := 4 * i$   
(17)  $t8 := 4 * j$   
(18)  $t9 := a[t8]$   
(19)  $a[t7] := t9$   
 $B_5$  (20)  $t10 := 4 * j$   
(21)  $a[t10] := x$   
(22) goto (5)  
-----  
(23)  $t11 := 4 * i$   
(24)  $x := a[t11]$   
(25)  $t12 := 4 * i$   
 $B_6$  (26)  $t13 := 4 * n$   
(27)  $t14 := a[t13]$   
(28)  $a[t12] := t14$   
(29)  $t15 := 4 * n$   
(30)  $a[t15] := x$



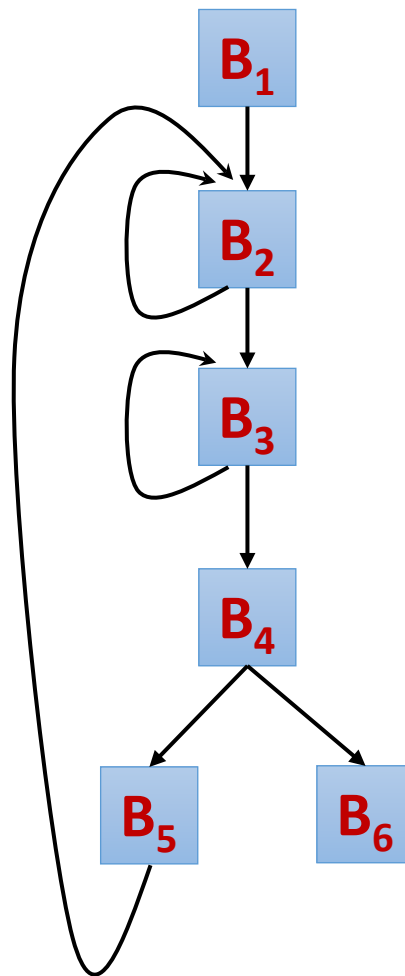
□ 流图中的一个结点集合L是一个循环，如果它满足：

- ❖ 该集合有唯一的入口结点
- ❖ 任意结点都有一个到达入口结点的非空路径，且该路径全部在L中

□ 不包含其他循环的循环叫做内循环

□ 右图中的循环

- ❖  $B_2$  自身
- ❖  $B_3$  自身
- ❖  $\{B_2, B_3, B_4, B_5\}$





中国科学技术大学  
University of Science and Technology of China



# 《编译原理与技术》

## 中间代码

**Done!**