中间代码表示

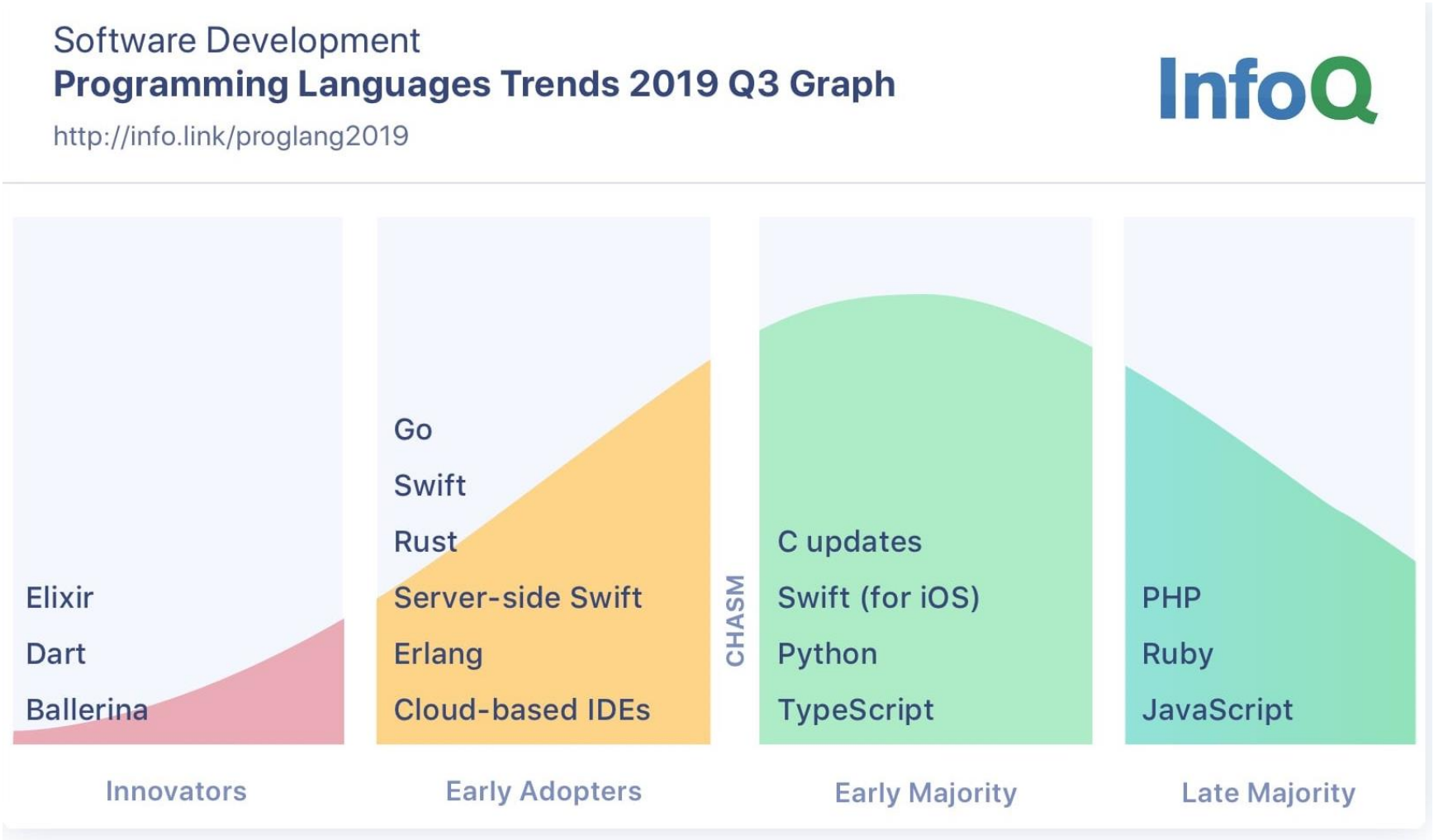
李 诚

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

2023年09月27日

为什么需要中间代码表示?

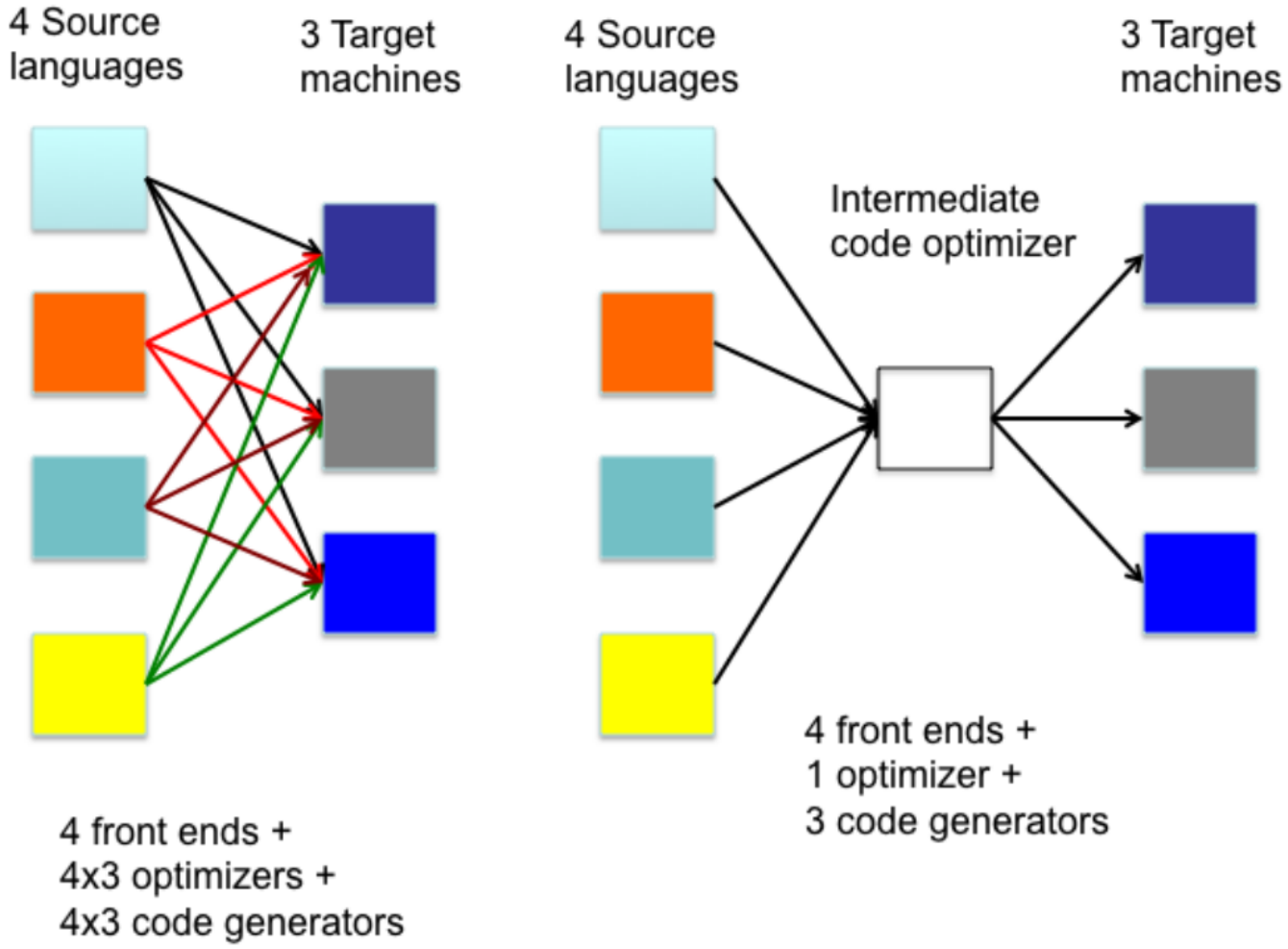


为什么需要中间代码表示?

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	MDMX, MIPS-3D	1981
VLIW	Elbrus	64	Just-in-time dynamic translation: x87, IA-32, MMX, SSE, SSE2, x86-64, SSE3, AVX	2014

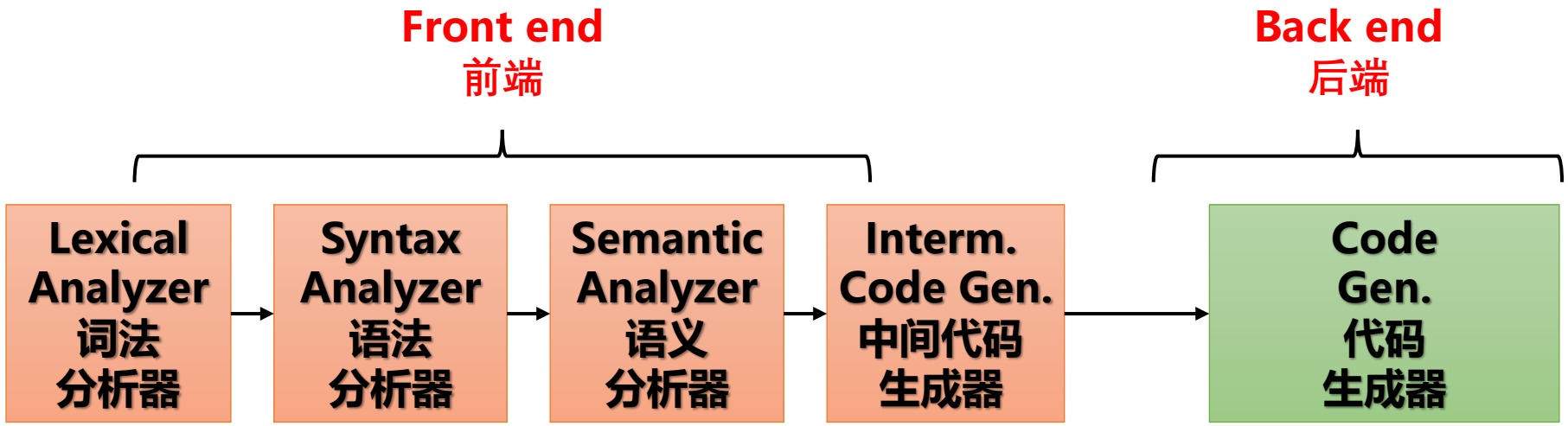
指令集体系结构(ISA)的发展

为什么需要中间代码表示?



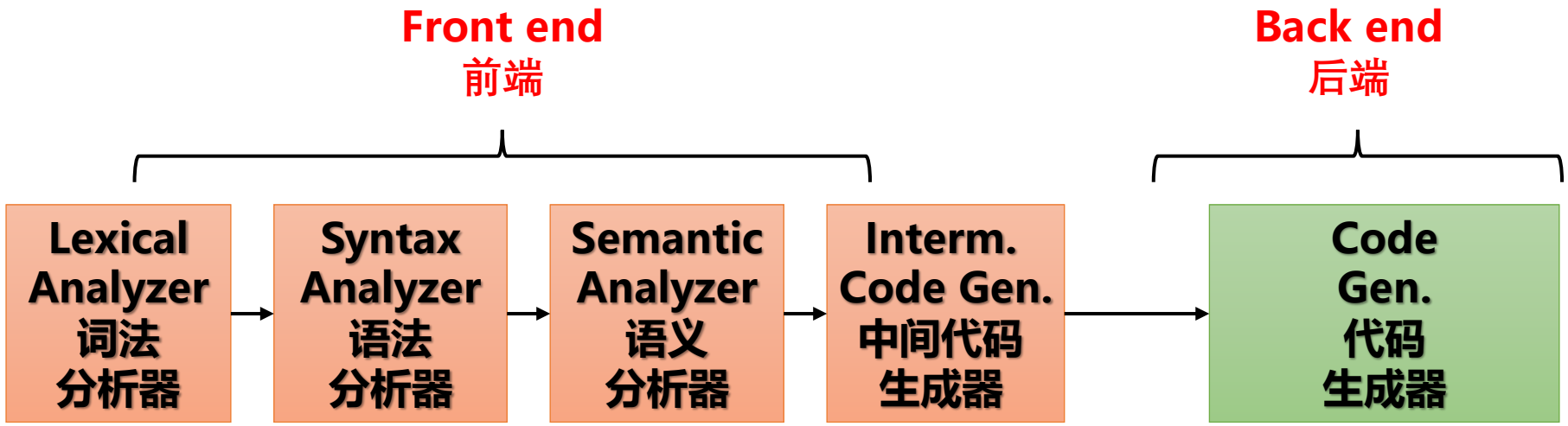
实践过程中，推陈出新的语言、不断涌现的指令集、开发成本之间的权衡

为什么需要中间代码表示



- **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

为什么需要中间代码表示



- **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**

中间表示有哪些类型?

- **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

后缀表示

uop是一元运算符

$$E \rightarrow E\ opE \mid uopE \mid (E) \mid \mathrm{id} \mid \mathrm{num}$$

表达式E	后缀式E'
id	id
num	num
$E_1\ op\ E_2$	$E_1'\ E_2'\ op$
$uopE$	$E'uop$
(E)	E'

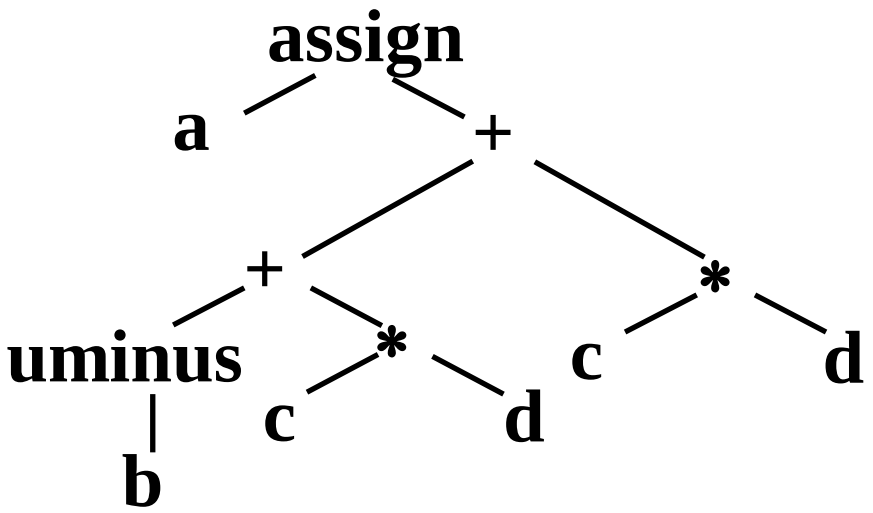
后缀表示

- 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	

图形表示

- 语法树是一种图形化的中间表示

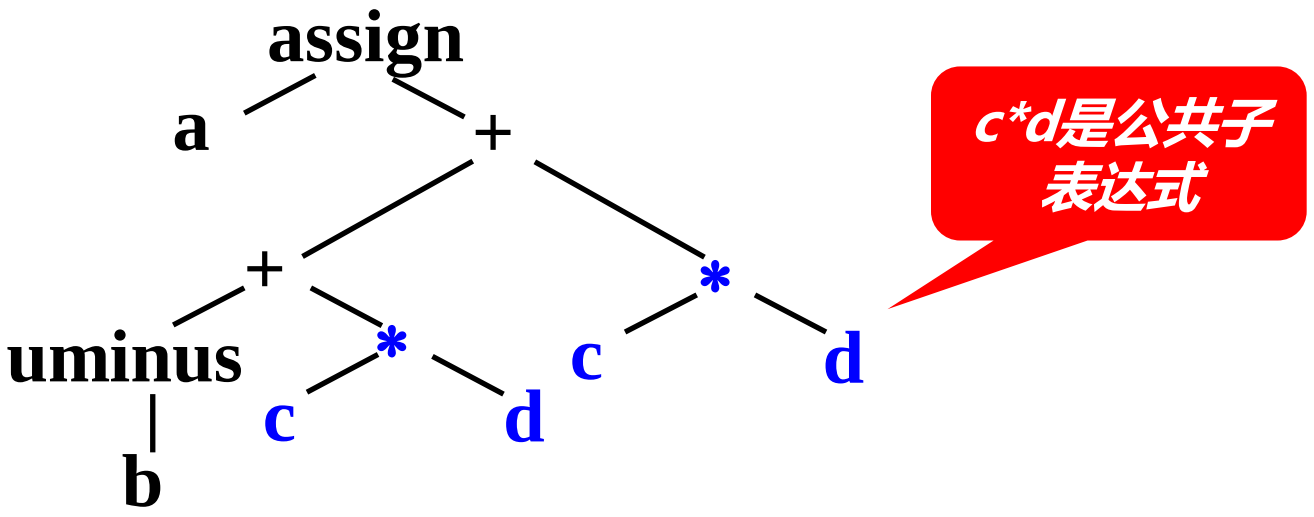


(a) 语法树

a = (–b + c*d) + c*d的图形表示

图形表示

- 语法树是一种图形化的中间表示

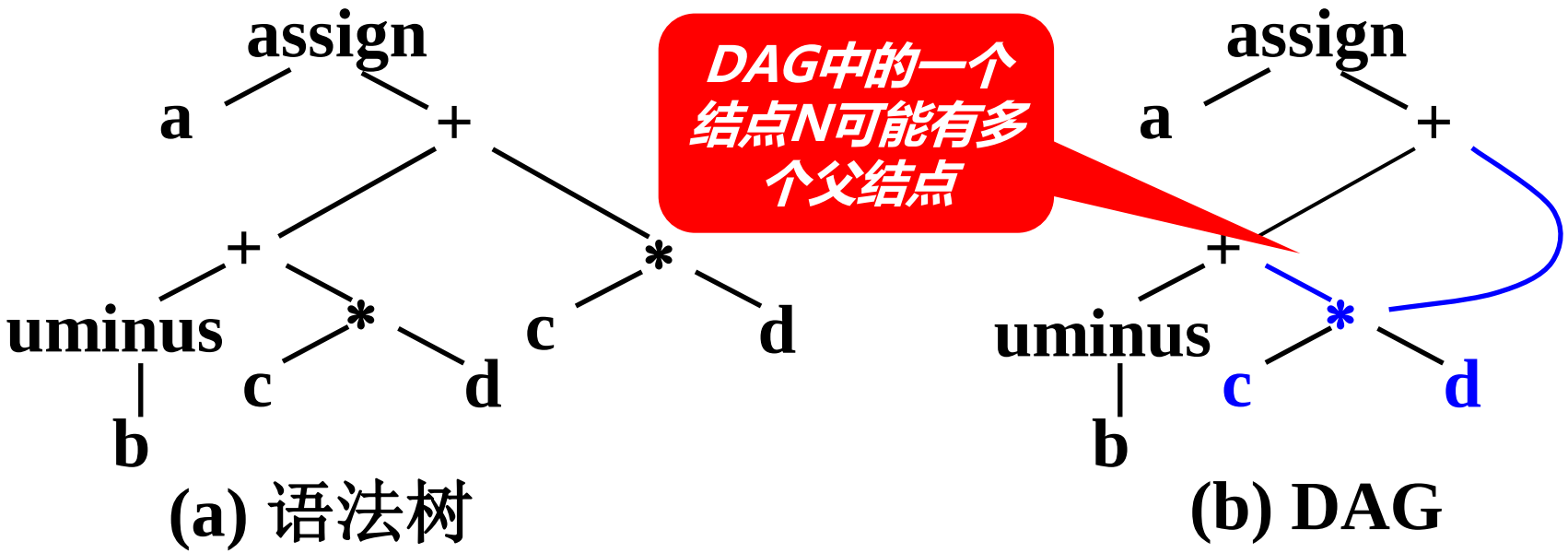


(a) 语法树

a = (–b + c*d) + c*d的图形表示

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　　(b) DAG

a = (–b + c*d) + c*d的图形表示

三地址代码

- **三地址代码 (Three-Address Code，TAC)**

一般形式：$x = y\ op\ z$

- 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

- 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

$t_1 = -b$

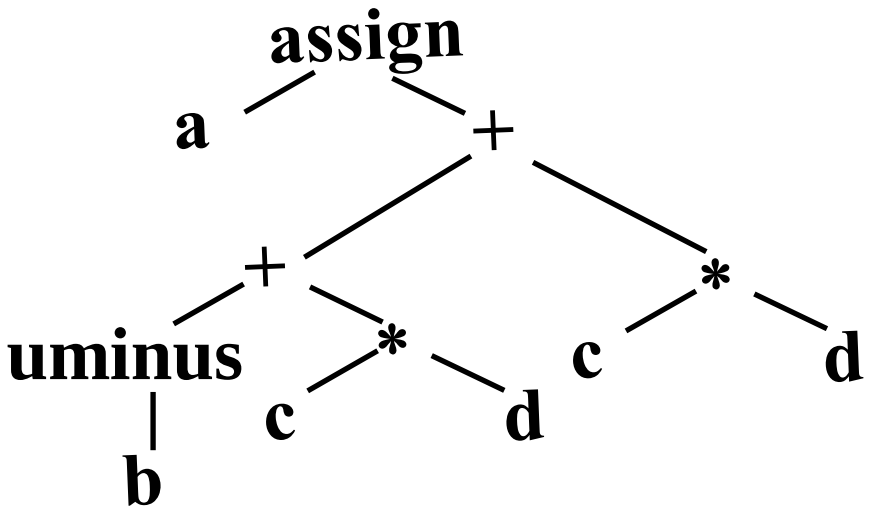
$t_2 = c * d$

$t_3 = t_1 + t_2$

$t_4 = c * d$

$t_5 = t_3 + t_4$

$a = t_5$



三地址代码

- 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

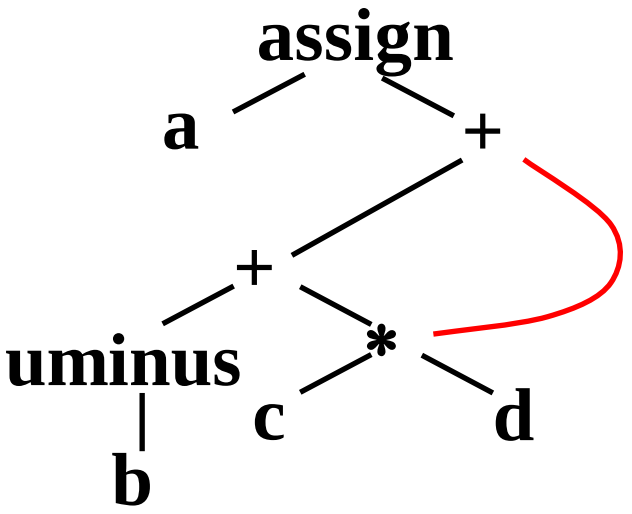
$t_1 = -b$

$t_2 = c * d$

$t_3 = t_1 + t_2$

$t_4 = t_3 + t_2$

$a = t_4$



三地址代码

- **常用的三地址语句**
  - 运算/赋值语句　　***x = y op z*，　*x = op y*，　*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**

三地址代码

- **常用的三地址语句**
  - 过程调用
    - **param** x_1 //设置参数
    - param x_2
    - …
    - param x_n
    - **call p , *n*** //调用子过程p，n为参数个数
  - 过程返回 **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***

三地址代码翻译：举例

- **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:  t1 = i + 1
    i  = t1
    t2 = i * 8
    t3 = a[ t2 ]
    if t3 < v goto L
```

符号标号

```
100:  t1 = i + 1
101:  i  = t1
102:  t2 = i * 8
103:  t3 = a[ t2 ]
104:  if t3 < v goto 100
```

位置标号

静态单赋值形式

- 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	静态单赋值形式
$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$

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

静态单赋值形式

- 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) x1 = –1; else x2 = 1;
x3 = φ(x1, x2);        //由flag的值决定用x1还是x2
y = x3 * a;
```

举例

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 >= 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 >= 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
```

基本块(Basic block)

- 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）

基本块划分算法

- **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 >= 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_1
(1) i := m –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 >= j goto (23)

B_5
(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_6
(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 (Flow graph)

- **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 (Flow graph)

- **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

B_1
(1) i := m –1
(2) j := n
(3) t1 := 4 * n
(4) v := a[t1]

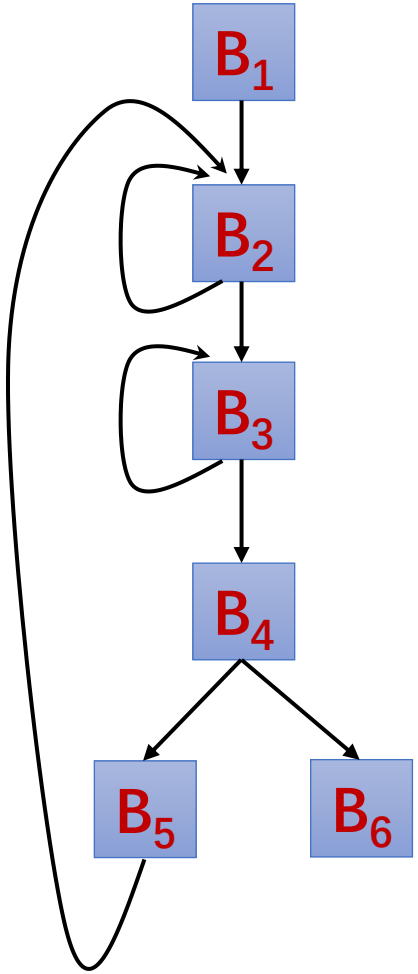
B_2
(5) i := i + 1
(6) t2 := 4 * i
(7) t3 := a[t2]
(8) if t3 < v goto (5)

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 >= j goto (23)

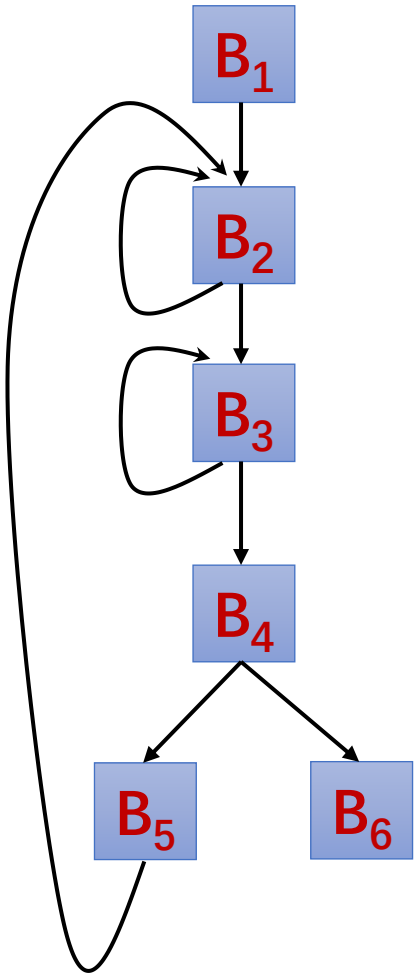
B_5
(14) t6 := 4 * i
(15) x := a[t6]
(16) t7 := 4 * i
(17) t8 := 4 * j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4 * j
(21) a[t10] := x
(22) goto (5)

B_6
(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



循环

- **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
  - $\{B_2, B_3, B_4, B_5\}$



一起努力
打造国产基础软硬件体系！

李 诚

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

2023年09月27日