



# 中间代码表示

# Intermediate Representation

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10月16日



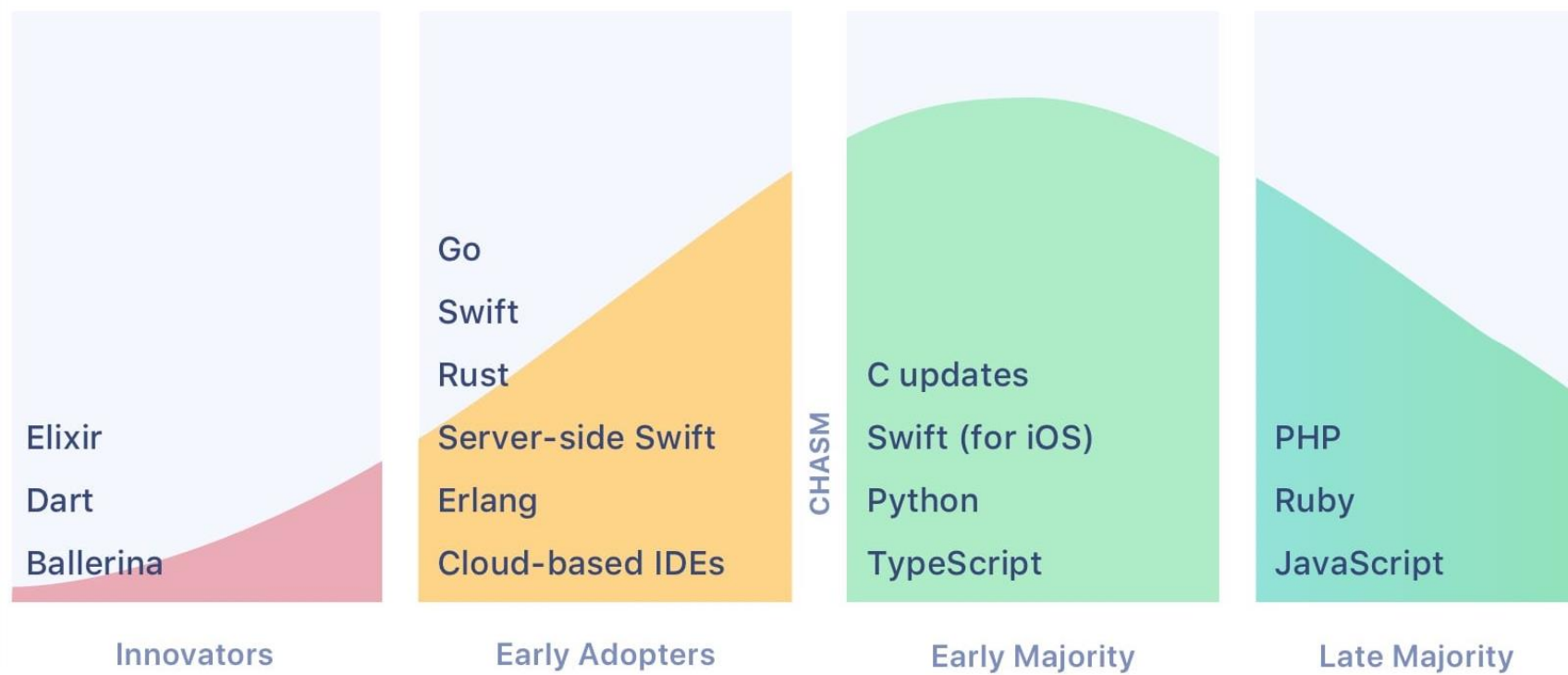
# 为什么需要中间代码表示?



## Software Development Programming Languages Trends 2019 Q3 Graph

<http://info.link/proglang2019>

InfoQ





# 为什么需要中间代码表示?

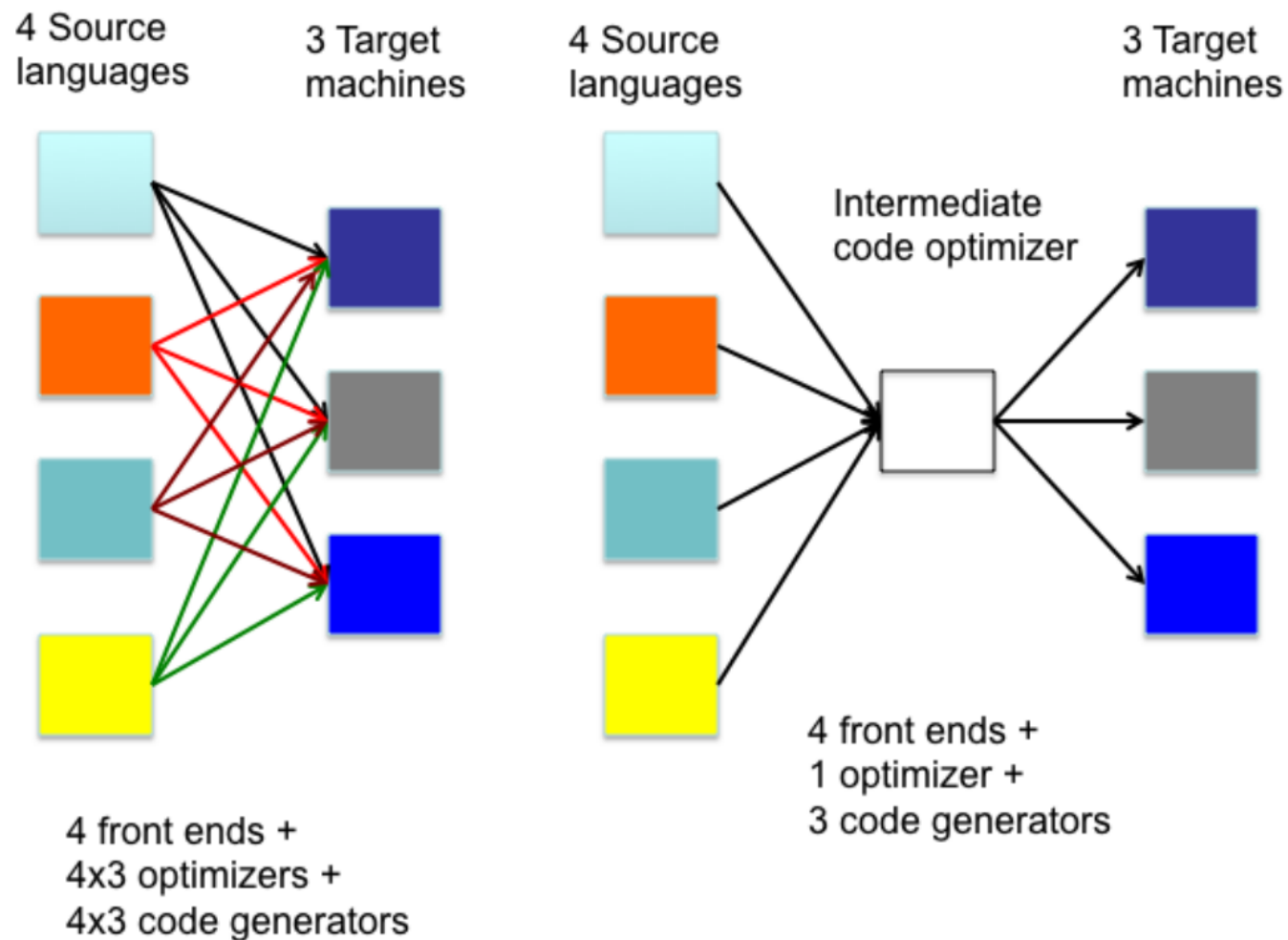


分类	名称	版本	扩展	初始年份
CISC	x86	16, 32, 64 (16→32→64)	x87, IA-32, MMX, 3DNow!, SSE, SSE2, PAE, x86-64, SSE3, SSSE3, SSE4, BMI, AVX, AES, FMA, XOP, F16C	1978
RISC	MIPS	32	<a href="#">MDMX</a> , <a href="#">MIPS-3D</a>	1981
VLIW	Elbrus	64	Just-in-time dynamic translation: x87, IA-32, MMX, SSE, SSE2, x86-64,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

指令集体系结构(ISA)的发展



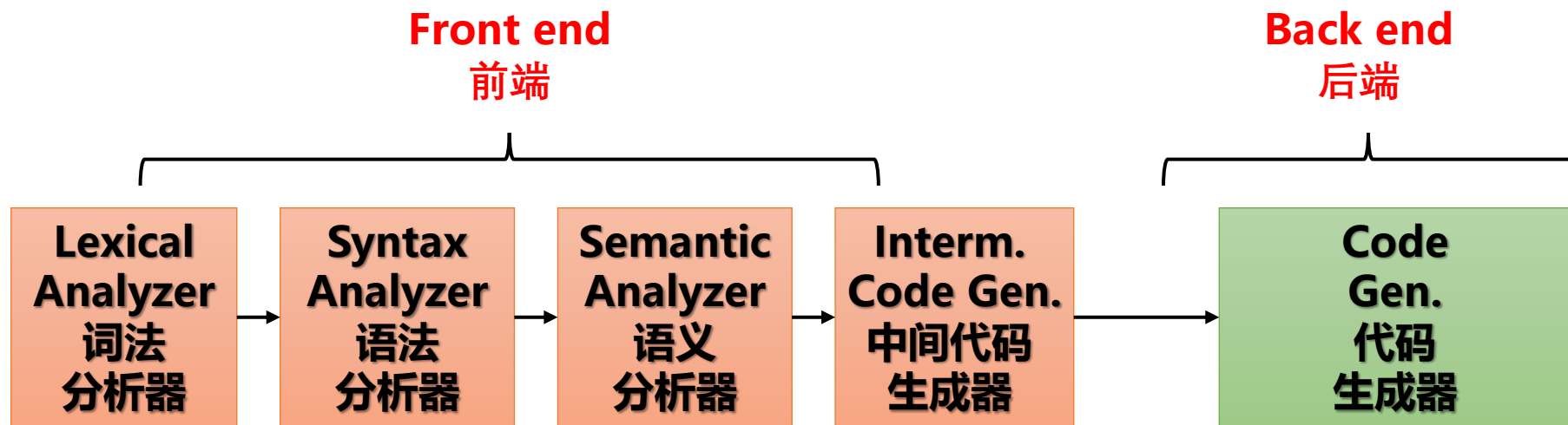
# 为什么需要中间代码表示?



**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

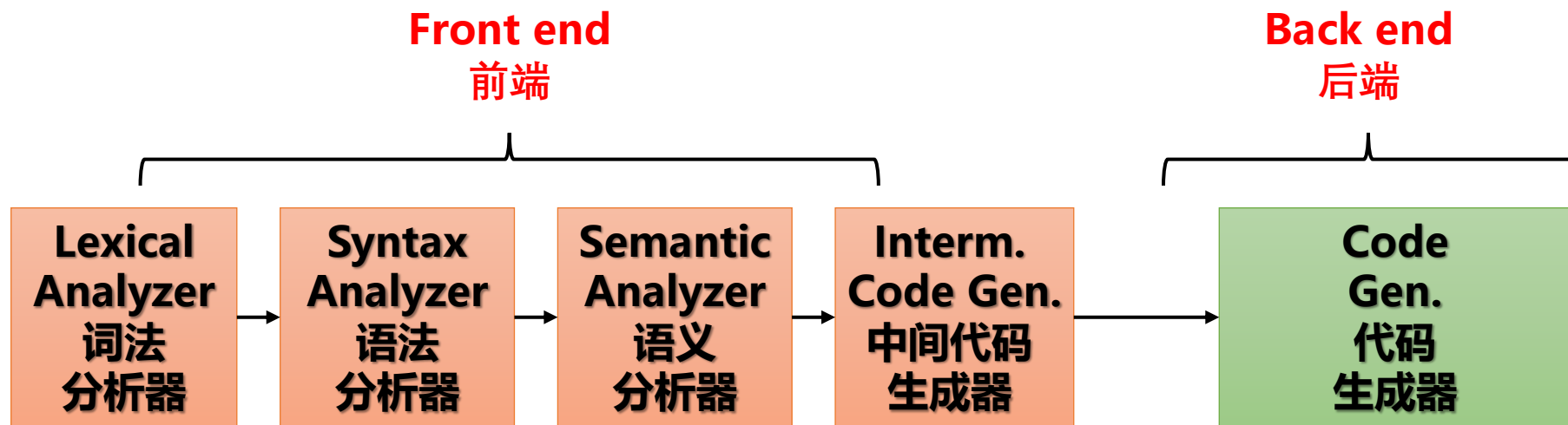


## • 前端与后端分离

- 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)



# 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**



*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  $E$

id

num

$E_1 \text{ op } E_2$

$uop E$

$(E)$

后缀式  $E'$

id

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

8

8 5

3

3 2

5

输入串

8 5 - 2 +

5 - 2 +

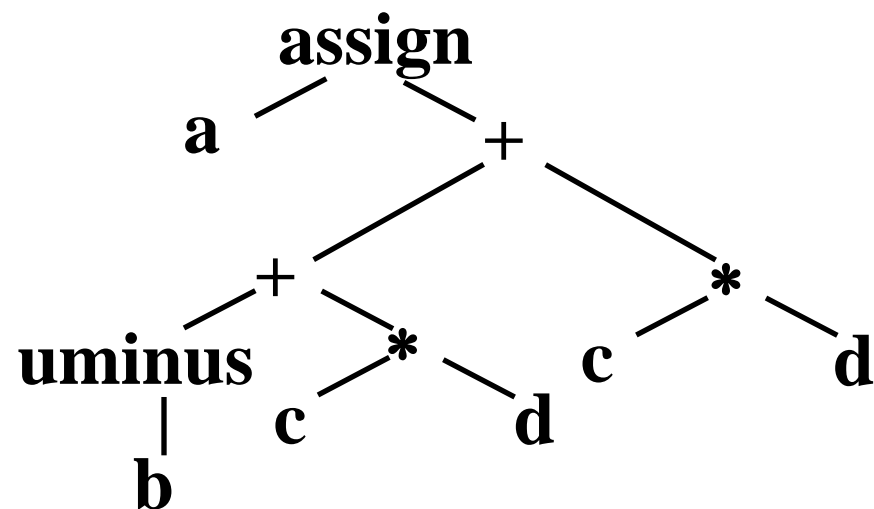
- 2 +

2 +

+



- 语法树是一种图形化的中间表示

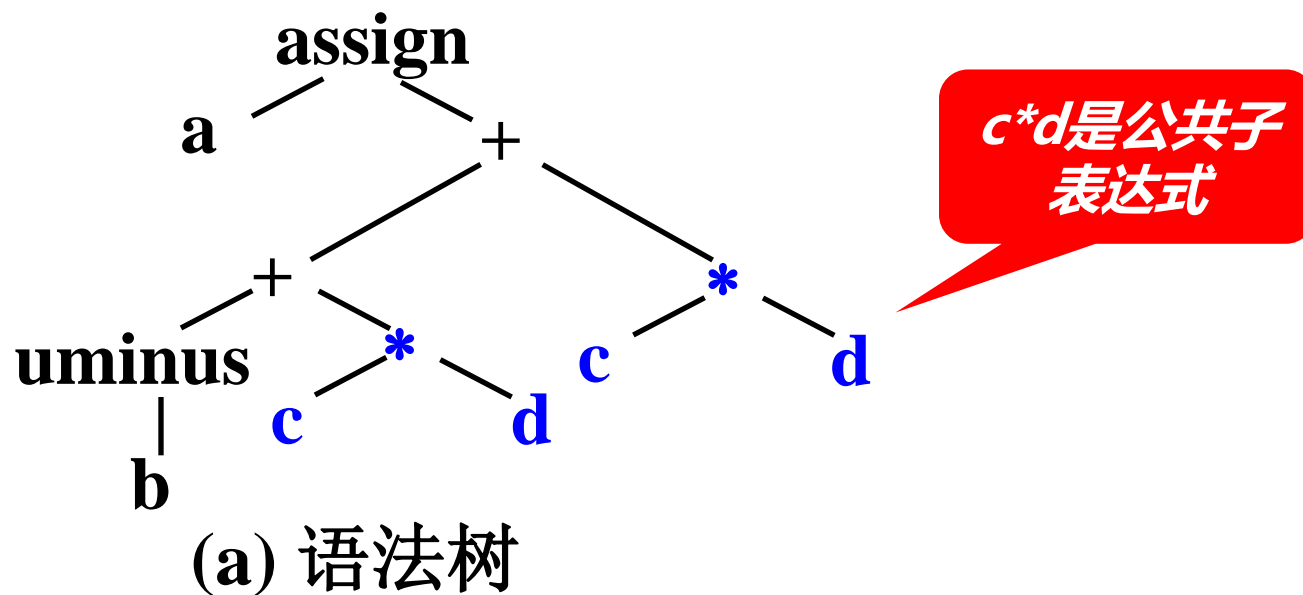


(a) 语法树

$a = (-b + c*d) + c*d$ 的图形表示



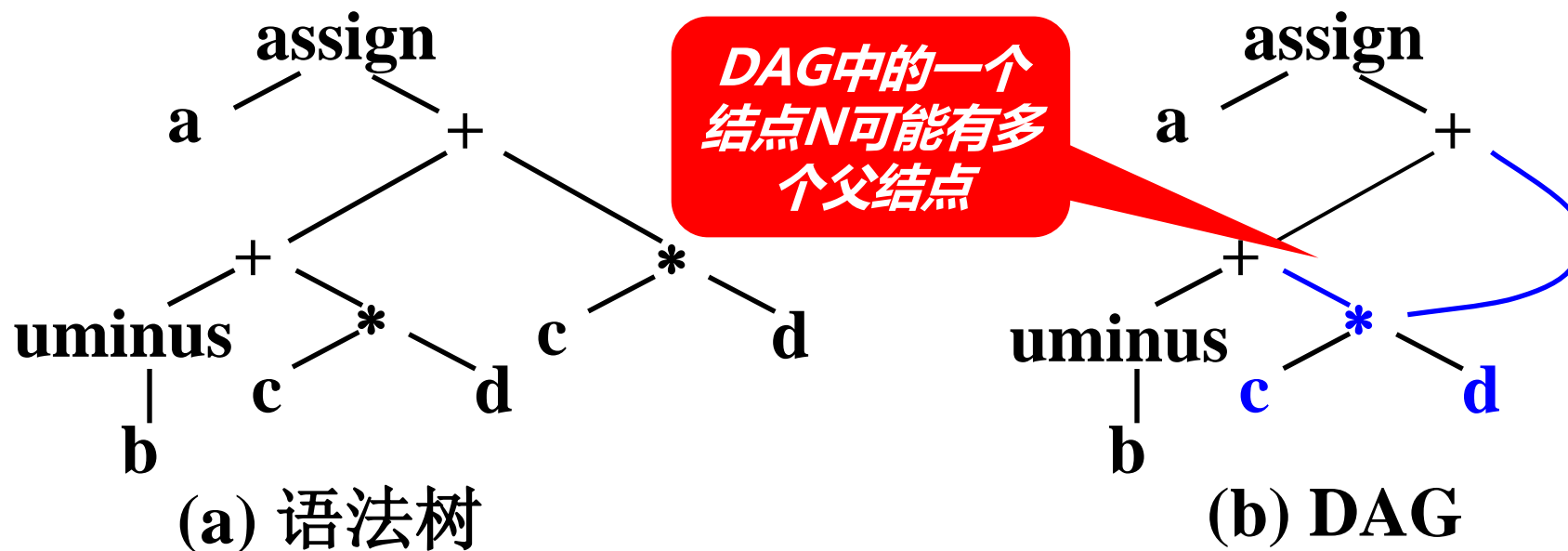
- 语法树是一种图形化的中间表示



$a = (-b + c*d) + c*d$ 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



$a = (-b + c*d) + c*d$ 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

$$t_1 = -b$$

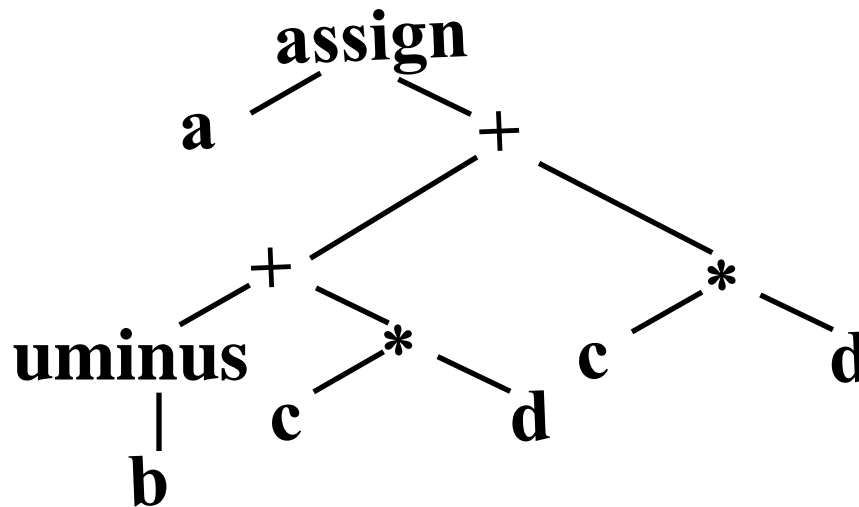
$$t_2 = c * d$$

$$t_3 = t_1 + t_2$$

$$t_4 = c * d$$

$$t_5 = t_3 + t_4$$

$$a = t_5$$





## • 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

DAG的代码

$$t_1 = -b$$

$$t_1 = -b$$

$$t_2 = c * d$$

$$t_2 = c * d$$

$$t_3 = t_1 + t_2$$

$$t_3 = t_1 + t_2$$

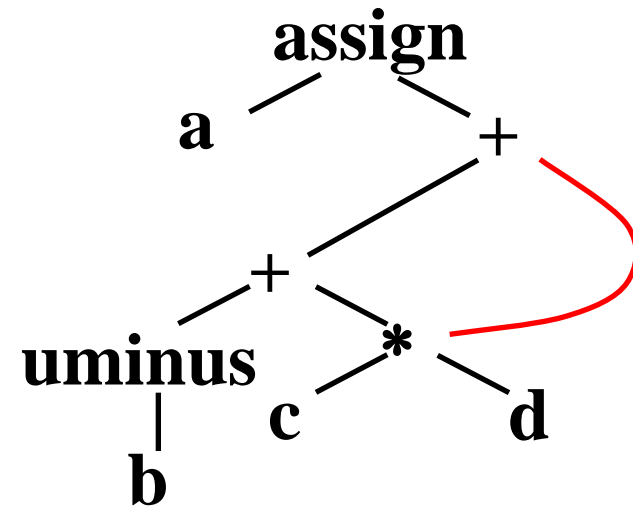
$$t_4 = c * d$$

$$t_4 = t_3 + t_2$$

$$t_5 = t_3 + t_4$$

$$a = t_4$$

$$a = t_5$$





## • 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 L1, if False x goto L2**
- 条件转移2 **if  $x \text{ relop } y$  goto L**





## • 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:   $t_1 = i + 1$   
     $i = t_1$   
     $t_2 = i * 8$   
     $t_3 = a[t_2]$   
    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



- 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赋值

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 $x_2$

$y = x_3 * a$ ;



# 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 ( $i \geq j$ ) 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)  $a[t15] := x$



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

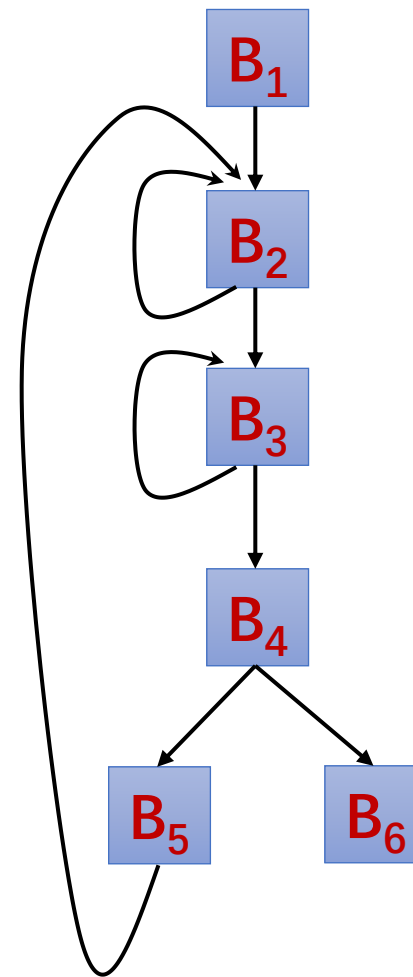
$B_5$

(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)  $a[t15] := x$





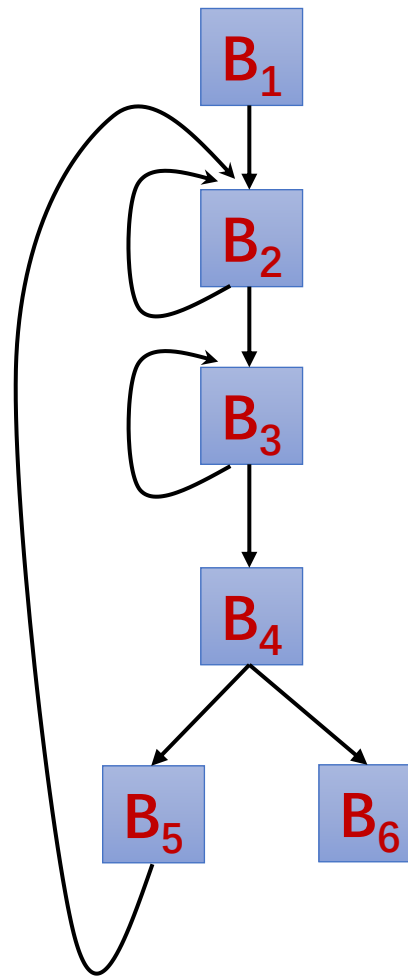
- 流图中的一个结点集合L是一个循环，如果它满足：

- 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

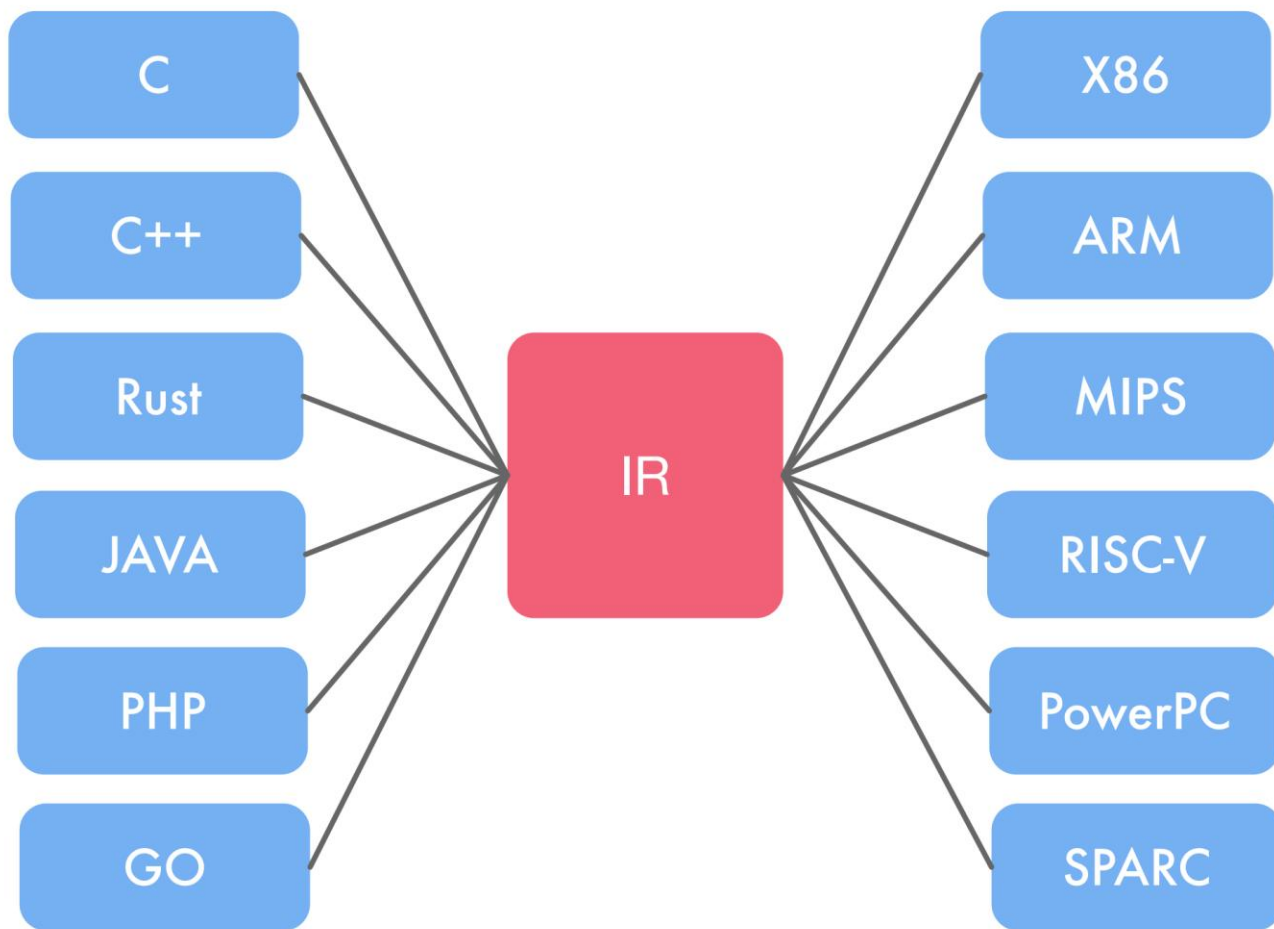
- $B_2$  自身
- $B_3$  自身
- $\{B_2, B_3, B_4, B_5\}$



# LLVM 出现



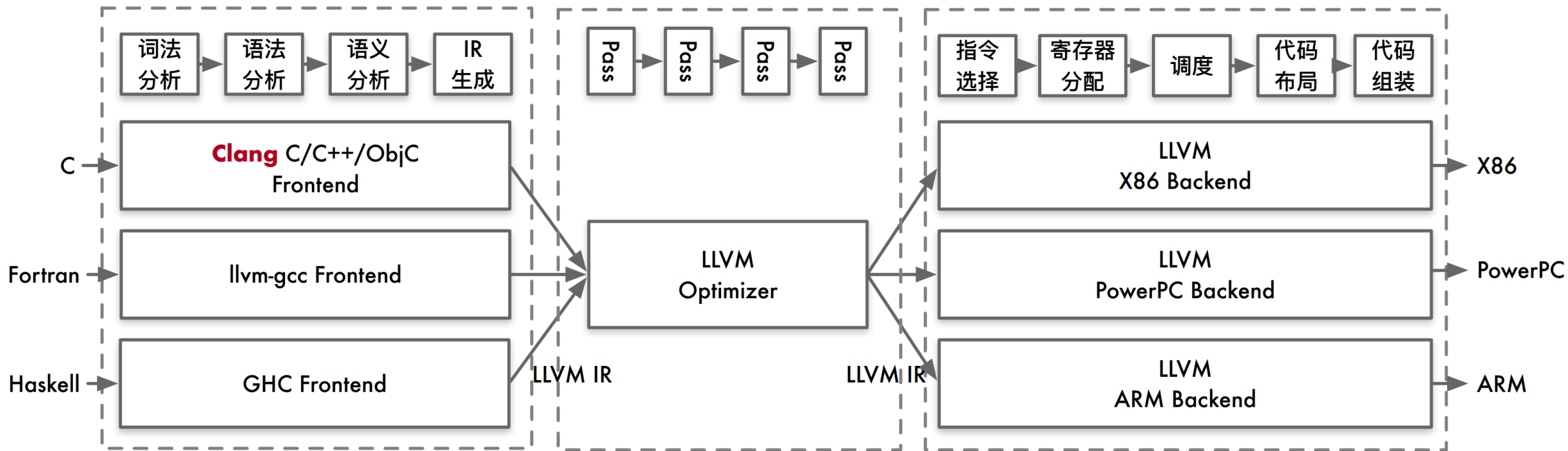
- LLVM定义了LLVM IR，将编译器需要的功能包装成库，解决了编译器代码重用的问题



# LLVM 架构



- LLVM 将很多编译器需要的功能**包装成库**，供其他编译器实现者根据需要**使用或者扩展**



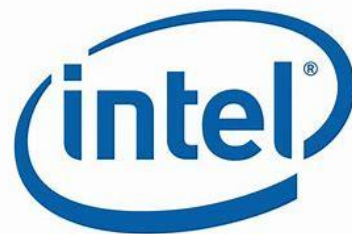


# LLVM 在工业界的广泛使用



- ❑ 工业界使用者（包括 Apple, Huawei, Intel, NVIDIA, Adobe, Sony... **30+** 公司使用 LLVM 开发相应产品）

Company	Description
Apple	iOS, macOS, tvOS and watchOS
Huawei	BiSheng Compiler for Huawei's Kunpeng servers
Intel	OpenCL* compiler, debugger
NVIDIA	OpenCL runtime compiler (Clang + LLVM)



# LLVM 在学术界的广泛使用



- 学术界使用者（包括 **CMU, ETH, UIUC, UCLA, Stanford ...** 在内 **20+** 顶尖大学研究组使用 **LLVM** 完成相关研究项目）

Organization	People	Description
CMU	David Koes	Principled Compilation
Stanford	Dawson Engler's Research Group	KLEE Symbolic Virtual Machine
ETH Zurich	Thomas Lenherr	Language-independent library for alias analysis
UCLA	Jason Cong	xPilot behavioral synthesis system

- 教育界使用者（包括 **CMU, ETH, UIUC, UCLA ...** 在内 **10+** 已公开的顶尖大学编译课程中使用 **LLVM** 工具链设计实验）

## □ LLVM IR 特点

- LLVM IR 采用 RISC 风格的三地址指令方式，形如 **%2 = add i32 %0, %1**
- LLVM IR 是 **SSA** 形式，且具有无限虚拟寄存器的假设
  - 注：SSA形式指的是静态单赋值形式，每个虚拟寄存器变量只会被赋值一次
- LLVM IR 是强类型系统，每个指令及操作数都具有自身的类型

## □ LLVM IR 的目标是成为一种通用 IR（支持包括动态与静态语言），因此 IR 指令种类较为复杂繁多

- 本课程以 cminusf 语言为源语言，从 **LLVM IR** 中裁剪出了适用于教学的精简的 **IR** 子集，并将其命名为 **LightIR**

# LLVM IR 库与 LightIR 库



## ❑ LLVM 提供了辅助生成 IR 的 C++ 库

### ■ LLVM IR 库用于课程实验的问题

- 类继承关系过于复杂，不利于学生理解 IR 抽象
- 存在很多为了编译性能的额外设计，不利于学生理解 IR 抽象

## ❑ 本课程依据 LLVM 的设计，为 LightIR 提供了配套简化的 C++ 库

### ■ LightIR C++ 库与 LLVM IR C++ 库的联系

- 仅保留必要的核心类，简化了核心类的继承关系与成员设计
- 给学生提供与 LLVM 相同的生成 IR 的接口

## ❑ 下面将对 LightIR 的结构层次与具体指令进行详细介绍

## □ 结构层次

- **Module**: 是 Light IR 中最顶层的结构，每个 Cminus-f 程序对应了一个 **Module**，在 **Module** 中存在 **GlobalVariable** 域与 **Function** 域

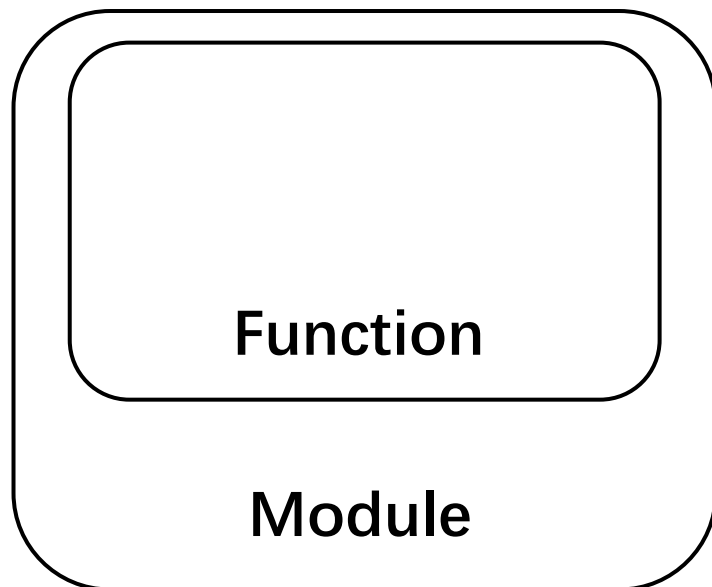
A large, empty rounded rectangle with a black border, representing the structure of a Module.

Module

## □ 结构层次

### ■ Module

- **Function**: 是函数，由头部与函数体组成。函数头部包括返回值类型，函数名，与参数表。函数体由多个**BasicBlock**构成，一个**module**中至少需要包含一个**main**函数

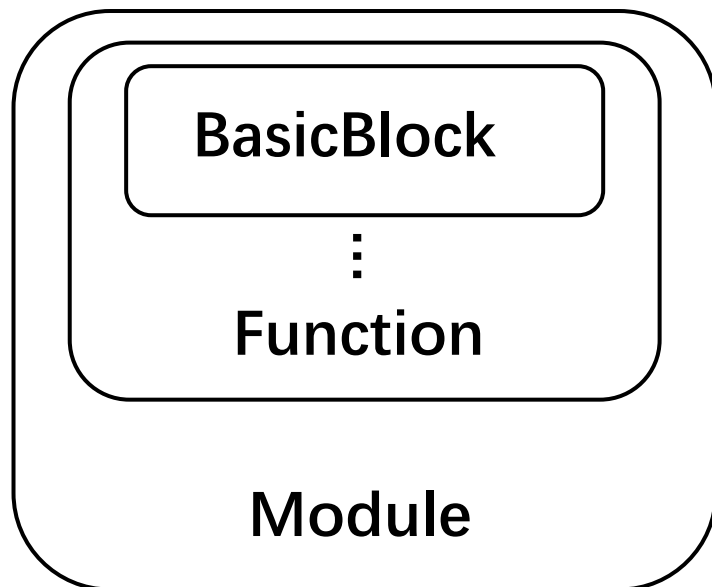


## □ 结构层次

### ■ Module

### ■ Function

- **BasicBlock**: 是指程序顺序执行的语句序列，只有一个入口和一个出口。基本块由若干**Instruction**构成



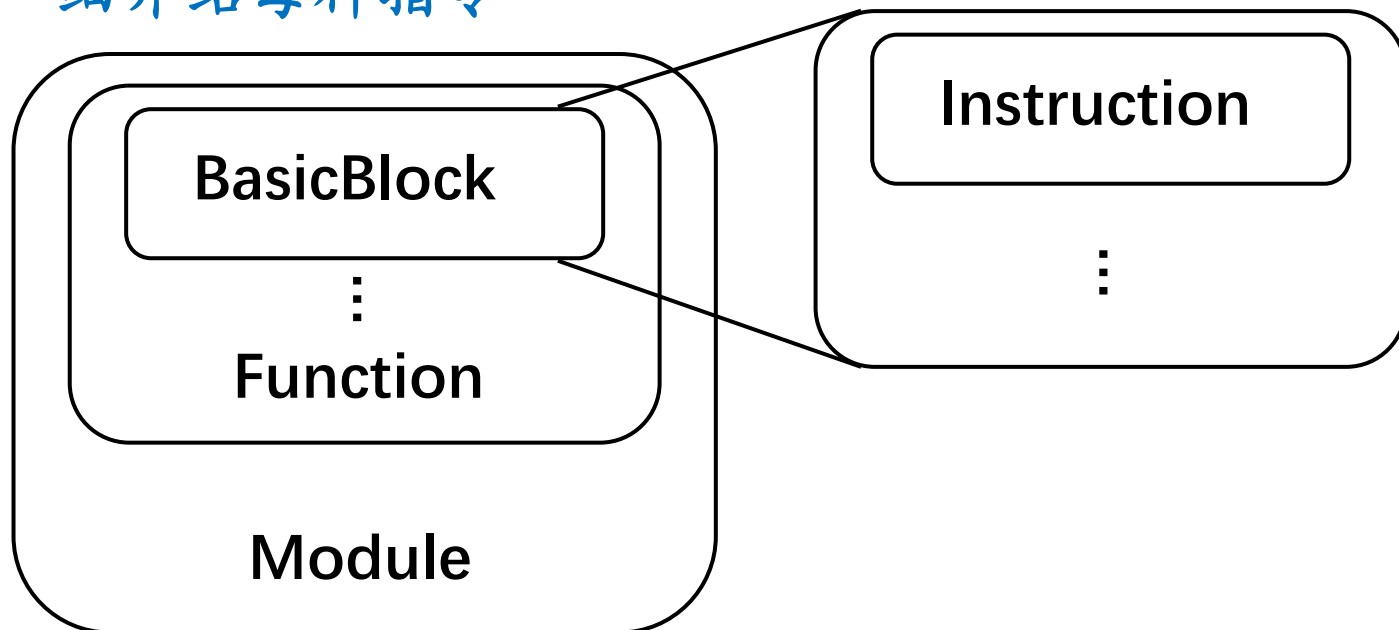
## □ 结构层次

■ Module

■ Function

■ BasicBlock

■ **Instruction**: 包含了Light IR从LLVM IR中裁剪出来的所有指令种类，接下来将详细介绍每种指令





# LightIR 片段



@y = global i32 36

```
define i32 @main() #0 {  
  %1 = alloca i32  
  %2 = alloca i32  
  store i32 0, i32* %1  
  store i32 16, i32* %2  
  %3 = load i32, i32* %2  
  %4 = load i32, i32* @y  
  %5 = icmp sgt i32 %3, %4  
  br i1 %5, label %6, label %10
```

```
6:  
  %7 = load i32, i32* %2  
  %8 = load i32, i32* @y  
  %9 = sub nsw i32 %7, %8  
  store i32 %9, i32* %2  
  br label %10
```

```
10:  
  %11 = load i32, i32* %1  
  ret i32 %11  
}
```

→ **Module**  
; preds = %0

; preds = %6, %0

# LightIR 片段



@y = global i32 36

GlobalVariable

```
define i32 @main() #0 {  
  %1 = alloca i32  
  %2 = alloca i32  
  store i32 0, i32* %1  
  store i32 16, i32* %2  
  %3 = load i32, i32* %2  
  %4 = load i32, i32* @y  
  %5 = icmp sgt i32 %3, %4  
  br i1 %5, label %6, label %10
```

```
6:  
  %7 = load i32, i32* %2  
  %8 = load i32, i32* @y  
  %9 = sub nsw i32 %7, %8  
  store i32 %9, i32* %2  
  br label %10
```

```
10:  
  %11 = load i32, i32* %1  
  ret i32 %11  
}
```

; preds = %0 → Function

; preds = %6, %0

# LightIR 片段



@y = global i32 36

```
define i32 @main() #0 {
```

```
%1 = alloca i32  
%2 = alloca i32  
store i32 0, i32* %1  
store i32 16, i32* %2  
%3 = load i32, i32* %2  
%4 = load i32, i32* @y  
%5 = icmp sgt i32 %3, %4  
br i1 %5, label %6, label %10
```

→ **BasicBlock**

```
6:                                ; preds = %0
```

```
%7 = load i32, i32* %2  
%8 = load i32, i32* @y  
%9 = sub nsw i32 %7, %8  
store i32 %9, i32* %2  
br label %10
```

→ **Instruction**

```
10:                               ; preds = %6, %0
```

```
%11 = load i32, i32* %1  
ret i32 %11
```

```
}
```

## □ 简介

- LightIR 每条指令或操作数都具有自己的类型，其中分为基本类型与组合类型

## □ 基本类型（LightIR类型系统中原生的类型）

- i1: 1位宽的整数类型
- i32: 32位宽的整数类型
- float: 单精度浮点数类型
- label: 标识符类型（基本块的类型）

## □ 组合类型（LightIR类型系统中，通过类型与运算符组合形成的类型）

- 组合类型包括，指针类型，数组类型，函数类型

## □ 指针类型

- 格式: `<type>*`
- 例如: `i32*`, `[10 x i32]*`

## □ 数组类型

- 格式: `[n x <type>]`
- 例如: `[10 x i32]`, `[10 x [10 x i32]]`

## □ 函数类型

- 格式: `<ret-type> @ <func name> (<arg-type>...)`
- 详解: 函数类型格式中有两个组成部分, 返回值类型与参数类型列表

□ LightIR 保留了LLVM IR **SSA 形式、三地址指令、强类型系统的特点**，但指令数量远小于 LLVM，按照种类可分为以下指令：

- 终止指令
- 二元运算指令
- 内存操作指令
- 类型转换指令
- 其他指令...

□ 在介绍具体指令时，遵循以下符号约定

- 以<>符号括起来的代表指令格式中**必选项**，以[ ]括起来的代表指令格式中**可重复项且至少存在一项**。以%标记的代表**IR中的虚拟寄存器**

## □ 概念

- 程序基本块只有一个入口与出口，顺序执行每一条语句，每个程序基本块的最后一条指令称为**终止指令**

## □ ret 指令

- 概念：返回指令，将控制流从函数返回给调用者
- 形式：
  - `ret <type> <value>` 例： `ret i32 %0`
  - `ret void` 例： `ret void`

## □ br 指令

- 概念：跳转指令，使控制流转移到当前功能中的另一个基本块
- 形式：
  - `br i1 <cond>, label <iftrue>, label <iffalse>` 例： `br i1 %cond, label %truebb label %falsebb`
  - `br label <dest>` 例： `br label %bb`

## □概念

- 包括add/sub/mul/sdiv/fadd/fsub/fmul/fdiv八种二元运算操作，返回计算后的结果

## □形式

- `<result> = <binary-op> <type> <op1>, <op2>`
- 注：<binary-op> 为八种二元运算操作之一，其中 add/sub/mul/sdiv 操作数是 i32 类型， fadd/fsub/fmul/fdiv 操作数是float类型

## □例子

- `%2 = sub i32 %1, %0`
- `%2 = fdiv float %1, %0`



## □概念

- **LightIR**是**SSA**格式的，同一个虚拟寄存器只会被赋值一次，对于需要多次赋值的变量，在翻译时在栈上分配相应的空间，并通过内存操作指令对其读写

## □alloca 指令

- **概念**：用于在栈上动态分配内存。该指令会在当前执行函数所在的栈中分配一块内存，内存的大小为**`sizeof(<type>)*NumElements`**，当函数执行结束时，分配的内存会自动释放。
- **形式**：**`<result> = alloca <type>`**
- **例子**：
  - `%ptr = alloca i32`
  - `%ptr = alloca [10 x i32]`

## load 指令

- 概念：从栈上分配的地址空间中取数据到虚拟寄存器中
- 形式：**`<result> = load <type>, <type>* <pointer>`**
- 例子：**`%val = load i32, i32* %ptr`**

## store 指令

- 概念：将虚拟寄存器中数据存到从栈上分配的地址空间中
- 形式：**`store <type> <value>, <type>* <pointer>`**
- 例子：**`store i32 3, i32* %ptr`**

## □ 概念

- 将数据从某个类型转换到另一个类型的指令

## □ zext 指令

- 概念：将操作数从 i1 类型零扩展到 i32 类型
- 形式：<result> = zext i1 <value> to i32
- 例子：%1 = zext i1 %0 to i32

## □ fptosi 指令

- 概念：将操作数从 float 类型转换为 i32 类型
- 形式：<result> = fptosi float <value> to i32
- 例子：%Y = fptosi float 1.0E-247 to i32

## □ sitofp 指令

- 概念：将操作数从 i32 类型转换为 float 类型
- 形式：<result> = sitofp <type> <value> to <type2>
- 例子：%1 = sitofp i32 257 to float

## □ icmp和fcmp指令

■ 形式: **<result> = icmp/fcmp <cond> <type> <op1>, <op2>**

■ 概念: 根据两个整数的比较返回布尔值,

- icmp: <cond> = eq(等于)| ne(不等于)| sgt(大于)| sge(大于等于)| slt(小于)| sle(小于等于)
- fcmp: <cond> = eq(等于)| ne(不等于)| ugt(大于)| uge(大于等于)| ult(小于)| ule(小于等于)

■ 例子:

- %2 = icmp sge i32 %0, %1
- %0 = fcmp ult float 4.0, 5.0

## □ call 指令

■ 概念: **call** 指令是调用指令, 用于使控制流转移到指定的函数

■ 形式: **<result> = call <return ty> <func name>(<function args>)**

■ 例子:

- %2 = call i32 @gcd( i32 %1, i32\* %0)
- call @output( i32 %1)

## ▣ getelementptr 指令

■ 概念：getelementptr指令用于获取数组结构的元素的地址。getelementptr仅执行地址计算，并且不访问内存

■ 形式：<result> = getelementptr <type1>, <type2>\* <ptrval> [, <type> <idx>]

➢ 注：<type1>是初始计算偏移的元素类型（若为数组类型，则在通过偏移值计算索引地址时进行降维，得到新的计算偏移的元素类型），<type2>表示索引开始的指针类型，<ptrval>是计算偏移的起始地址的指针，[]表示可重复参数，里面表示的数组索引的偏移值的类型与偏移值。

### ■ 例子：

➢ %array = [10 x i32]\* @myarray

➢ %ptr = getelementptr [10 x i32], [10 x i32]\* %array, i32 0, i32 1

➢ 这里 %array 是一个指向含有 10 个 i32 类型元素的数组的指针，getelementptr 指令用于获取第二个元素（索引为 1，因为 LLVM 中索引从 0 开始）的地址。



# 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10月16日