



中山大學
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

DCS²⁹⁰

Compilation Principle 编译原理

第六章 中间代码生成 (1)

郑馥丹

zhengfd5@mail.sysu.edu.cn

CONTENTS

目录

01

中间代码概述
Introduction

02

类型和声明
Types and
Declarations

03

表达式和语句
Assignment and
Expressions

04

类型检查
Type
Checking

05

布尔表达式
Boolean
Expressions

06

回填技术
Backpatching

1. 中间代码生成[Intermediate Code Generation]

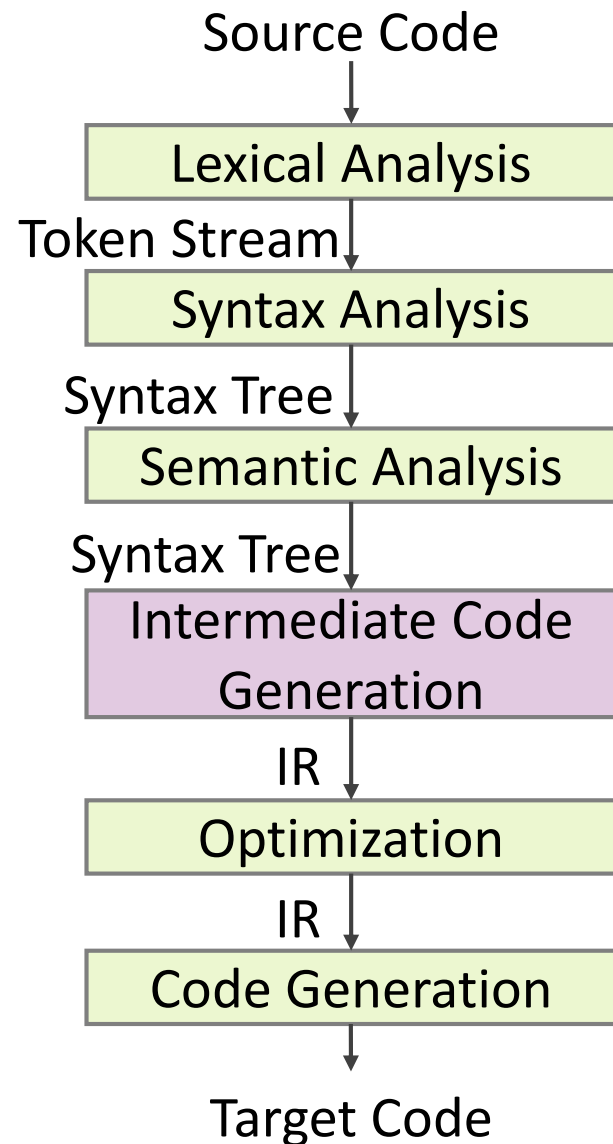
01 中间代码概述

——从这里开始真正做翻译工作

- 初步翻译，生成**等价**于源程序的中间表示（IR）
 - 输入：语法树，输出：IR
 - 建立源和目标语言的桥梁，易于翻译过程的实现，利于实现某些优化算法
 - IR形式：通常为**三地址码（TAC）**

```
void main()  
{  
    int arr[10], i, x = 1;  
    for (i = 0; i < 10; i++)  
        arr[i] = x * 5;  
}
```

```
i := 0  
loop:  
    t1 := x * 5  
    t2 := &arr  
    t3 := sizeof(int)  
    t4 := t3 * i  
    t5 := t2 + t4  
    *t5 := t1  
    i := i + 1  
    if i < 10 goto loop
```



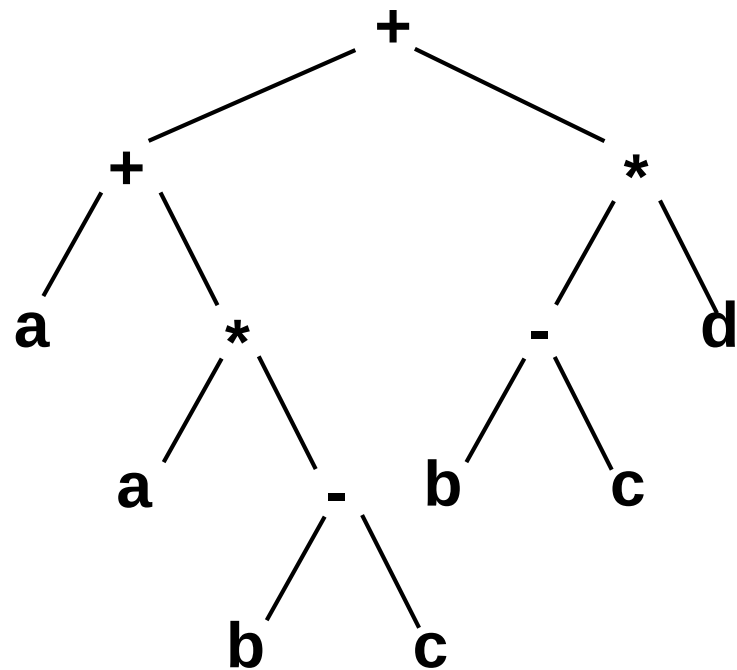
2. 中间表示[Intermediate Representation, IR]

- 高级中间表示
 - AST(Abstract Syntax Tree, 抽象语法树)和DAG(Directed Acyclic Code, 有向无环图)
 - 适用于静态类型检查等任务
- 低级中间表示
 - **3-地址码(Three-Address Code, TAC)**: $x = y \text{ op } z$
 - 适用于依赖于机器的任务, 如寄存器分配和指令选择。
- IR的选择/设计都是针对具体应用的
 - LLVM IR: 通用
 - TensorFlow XLA IR: 专门针对机器学习计算图优化
 - 常用C语言 (AT&T贝尔实验室高级C++)

2. 中间表示[Intermediate Representat

- AST和DAG

– 例：算术表达式 $a+a*(b-c)+(b-c)*d$

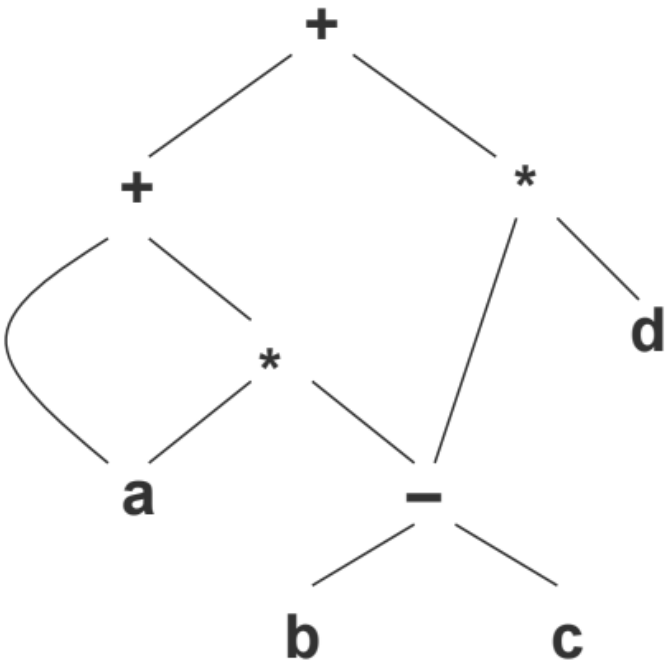


AST

- (1) `p1 = new Leaf(id, entry-a)`
- (2) `p2 = new Leaf(id, entry-a)`
- (3) `p3 = new Leaf(id, entry-b)`
- (4) `p4 = new Leaf(id, entry-c)`
- (5) `p5 = new Node('-', p3 , p4 )`
- (6) `p6 = new Node('*', p2 , p5 )`
- (7) `p7 = new Node('+', p1 , p6 )`
- (8) `p8 = new Leaf(id, entry-b)`
- (9) `p9 = new Leaf(id, entry-c)`
- (10) `p10 = new Node('-', p8 , p9)`
- (11) `p11 = new Leaf(id, entry-d)`
- (12) `p12 = new Node('*', p10, p11)`
- (13) `p13 = new Node('+', p7 , p12)`

No.	Productions	Semantic Rules
1	$E \rightarrow E_1 + T$	$E.\text{node} = \text{new Node}('+', E_1.\text{node}, T.\text{node})$
2	$E \rightarrow E_1 - T$	$E.\text{node} = \text{new Node}('-', E_1.\text{node}, T.\text{node})$
3	$E \rightarrow T$	$E.\text{node} = T.\text{node}$
4	$T \rightarrow (E)$	$T.\text{node} = E.\text{node}$
5	$T \rightarrow \text{id}$	$T.\text{node} = \text{new Leaf}(\text{id}, \text{id}.\text{entry})$
6	$T \rightarrow \text{num}$	$T.\text{node} = \text{new Leaf}(\text{num}, \text{num}.\text{val})$

SDD

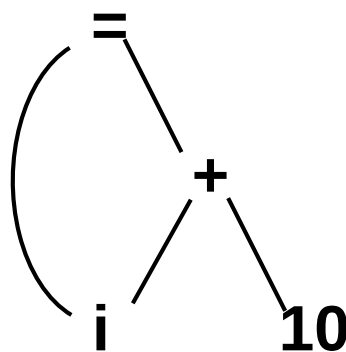


DAG

2. 中间表示[Intermediate Representation, IR]

• 构造DAG的值编码方法

- 语法树或DAG中的结点存放在一个记录数组中
- 数组的每一行表示一个记录，即一个结点
- 每个记录中，都有结点编号
- 叶子结点：一个附加字段，存放标识符的词法值lexval
- 内部结点：两个附加字段，分别指明其左右结点
- 例： $i=i+10$

**DAG**

1	id	i	
2	num	10	
3	+	1	2
4	=	1	3
5	...	...	...

值编码

随堂练习 (1)

- 为下列表达式构造DAG，并指出其值编码，假定+是左结合的。

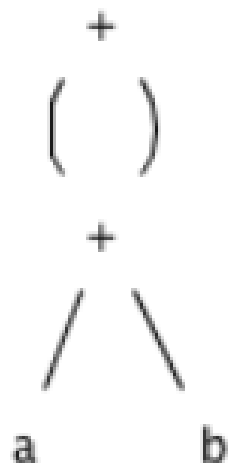
(1) $a+b+(a+b)$

(2) $a+b+a+b$

(3) $a+a+(a+a+a+(a+a+a+a))$

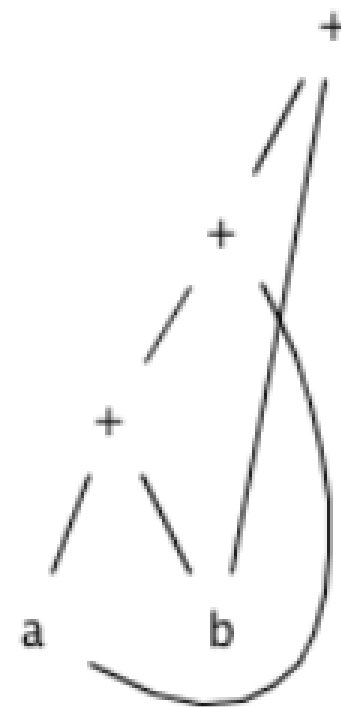
• 参考答案

(1) $a+b+(a+b)$



1	id	a	
2	id	b	
3	+	1	2
4	+	3	3

(2) $a+b+a+b$



1	id	a	
2	id	b	
3	+	1	2
4	+	3	1
5	+	4	2

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, **TAC**]

- 三地址码中，**一条指令右侧最多有一个运算符**，即，不允许出现组合的算术表达式

- 像 $x + y * z$ ，需翻译成如下三地址指令序列：

- $t1 = y * z$

- $t2 = x + t1$

- 例：算术表达式 $a + a * (b - c) + (b - c) * d$

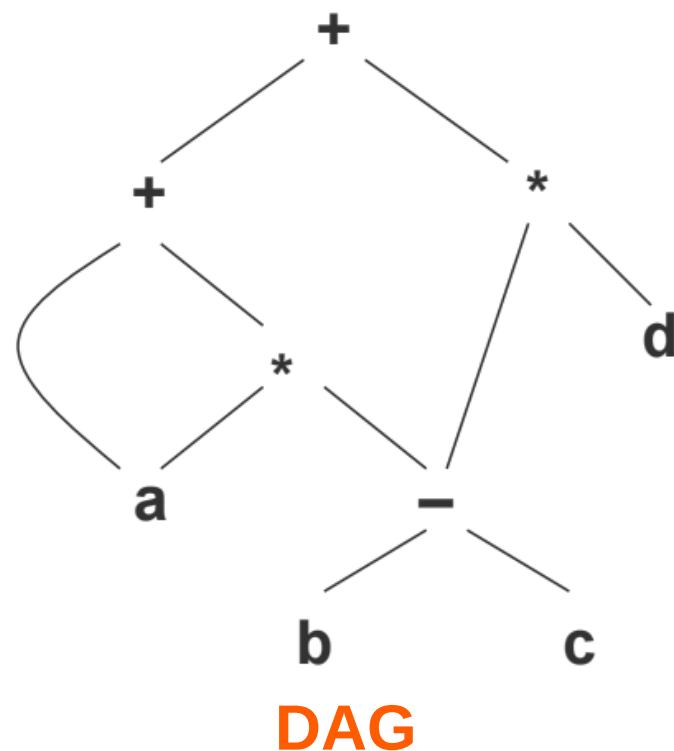
$t1 = b - c$

$t2 = a * t1$

$t3 = a + t2$

$t4 = t1 * d$

$t5 = t3 + t4$



2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]
 - 两个基本概念：地址和指令
 - 地址：
 - ✓ 名字：源程序的名字作为三地址码中的地址：源程序名字被替换为指向符号表条目的指针，关于该名字的所有信息均存放在该条目中
 - ✓ 常量：需考虑表达式中的类型转换问题
 - ✓ 编译器生成的临时变量

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]

- 指令

- ✓ $x = y \text{ op } z$ // 双目运算或逻辑运算, x 、 y 、 z 为地址

- ✓ $x = \text{op } y$ // 单目运算: 单目减(取负)、逻辑非、转换运算(整数转成浮点数等)

- ✓ $x = y$ // 赋值运算

- ✓ **goto** L // 无条件跳转

- ✓ **if** x **goto** L // 有条件跳转

- ✓ **ifFalse** x **goto** L // 有条件跳转

- ✓ **if** $x \text{ op } y$ **goto** L // 关系运算跳转

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]

- 指令

- ✓ **param** x_1 // 参数传递

- ✓ **param** x_2

- ✓ ...

- ✓ **param** x_n

- ✓ **call** p, n // 过程调用

- ✓ $y = \text{call } p, n$ // 函数调用

- ✓ **return** y // 返回值

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]

- 指令

- ✓ $x = y[i]$ // 带下标的赋值指令, **注意i代表内存单元位置, 而非数组位置**

- ✓ $x[i] = y$ // 同上

- ✓ $x = \&y$ // 取y变量的地址赋值给x

- ✓ $x = *y$ // 取y指针中的内容赋值给x

- ✓ $*x = y$ // 取y的值赋值给x指针所指空间

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]

- 例：源码 **do** $i=i+1$;
 while($a[i]>v$);

带符号标号的三地址码：

```
L:  $t_1 = i + 1$   
    $i = t_1$   
    $t_2 = i * 8$  //假设数组中每个元素  
               //占8个存储单元  
    $t_3 = a[t_2]$   
   if  $t_3 < v$  goto L
```

带位置号的三地址码：

```
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
```

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]
 - ① 三元式[triple]
 - ② 间接三元式[indirect triple]
 - ③ 四元式[quadruple]

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]

- ① 三元式[triple]

- ✓ 三个字段: **op, arg1, arg2**

- ✓ 用运算 $x \text{ op } y$ 的位置来表示其结果, 而不是用一个显式的临时名字

- ✓ 带括号的数字表示指向相应三元式结构的指针

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]

- ① 三元式[triple]

✓ 例：源码 $a = b * -c + b * -c$

三地址码：

三元式：

$t1 = \text{minus } c$

$t2 = b * t1$

$t3 = \text{minus } c$

$t4 = b * t3$

$t5 = t2 + t4$

$a = t5$

	op	arg ₁	arg ₂
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
...	...		

对运算结果的引用是通过位置完成的，因此如果改变一条指令的位置，则引用该指令结果的所有指令都要相应修改！

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]

- ② 间接三元式[indirect triple]

- ✓ 同样三个字段： **op, arg1, arg2**

- ✓ 包含 **一个指向三元式的指针的列表**，而不是列出三元式序列本身，从而化解了前述三元式由于指令改变所引起的问题

2. 中间表示[Intermediate Representation, IR]

• 三地址码[Three-Address Code, TAC]

② 间接三元式[indirect triple]

✓ 例：源码 $a = b * -c + b * -c$

三地址码：

$t1 = \text{minus } c$

$t2 = b * t1$

$t3 = \text{minus } c$

$t4 = b * t3$

$t5 = t2 + t4$

$a = t5$

间接三元式：

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

	op	arg ₁	arg ₂
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
...	...		

优化编译器可以通过对指令列表的重新排序来移动指令位置，而不影响三元式本身。

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]

- ③ 四元式[quadruple]

- ✓ 四个字段: **op, arg1, arg2, result**

- ✓ 一些特例:

- 形如 $x = \text{minus } y$ 的单目运算符指令和赋值指令 $x = y$, 不使用arg2
 - 像param这样的运算既不使用arg2, 也不使用result
 - 条件或非条件转移指令将目标标号放入result字段

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]

- ③ 四元式[quadruple]

✓ 例：源码 $a = b * -c + b * -c$

三地址码：

$t_1 = \text{minus } c$

$t_2 = b * t_1$

$t_3 = \text{minus } c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

四元式：

	op	arg ₁	arg ₂	result
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
...	...			

相对于三元式的优势：当移动一个计算临时变量t的指令时，那些使用t的指令不需要做任何改变

随堂练习 (2)

- 对于以下的表达式，分别给出三元式和四元式序列

(1) $a = b[i] + c[j]$

(2) $a[i] = b * c - b * d$

(3) $x = f(y + 1) + 2$

(4) $x = *p + \&y$

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

• 参考答案

(1) $a = b[i] + c[j]$

#	OP	ARG1	ARG2
1	=[]	b	i
2	=[]	c	j
3	+	(1)	(2)
4	=	a	(3)

三元式

#	OP	ARG1	ARG2	RESULT
1	=[]	b	i	t1
2	=[]	c	j	t2
3	+	t1	t2	t3
4	=	t3	—	a

四元式

随堂练习 (2)

• 参考答案

(2) $a[i] = b * c - b * d$

#	OP	ARG1	ARG2
1	*	b	c
2	*	b	d
3	-	(1)	(2)
4	[]=	a	i
5	=	(4)	(3)

三元式

#	OP	ARG1	ARG2	RESULT
1	*	b	c	t1
2	*	b	d	t2
3	-	t1	t2	t3
4	[]=	a	i	t4
5	=	t3	—	*t4

四元式

随堂练习 (2)

• 参考答案

(3) $x = f(y + 1) + 2$

#	OP	ARG1	ARG2
1	+	y	1
2	param	(1)	—
3	call	f	1
4	+	(3)	2
5	=	x	(4)

三元式

#	OP	ARG1	ARG2	RESULT
1	+	y	1	t1
2	param	t1	—	—
3	call	f	1	t2
4	+	t2	2	t3
5	=	t3	—	x

四元式

随堂练习 (2)

• 参考答案

(4) $x = *p + \&y$

#	OP	ARG1	ARG2
1	*	p	—
2	&	y	—
3	+	(1)	(2)
4	=	x	(3)

三元式

#	OP	ARG1	ARG2	RESULT
1	*	p	—	t1
2	&	y	—	t2
3	+	t1	t2	t3
4	=	t3	—	x

四元式

随堂练习 (2)

• 参考答案

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

#	OP	ARG1	ARG2
1	+	a	b
2	@	—	(1)
3	+	c	d
4	*	(2)	(3)
5	+	a	b
6	+	(5)	c
7	-	(4)	(6)

三元式

#	OP	ARG1	ARG2	RESULT
1	+	a	b	t1
2	@	t1	—	t2
3	+	c	d	t3
4	*	t2	t3	t4
5	+	a	b	t5
6	+	t5	c	t6
7	-	t4	t6	t7

四元式