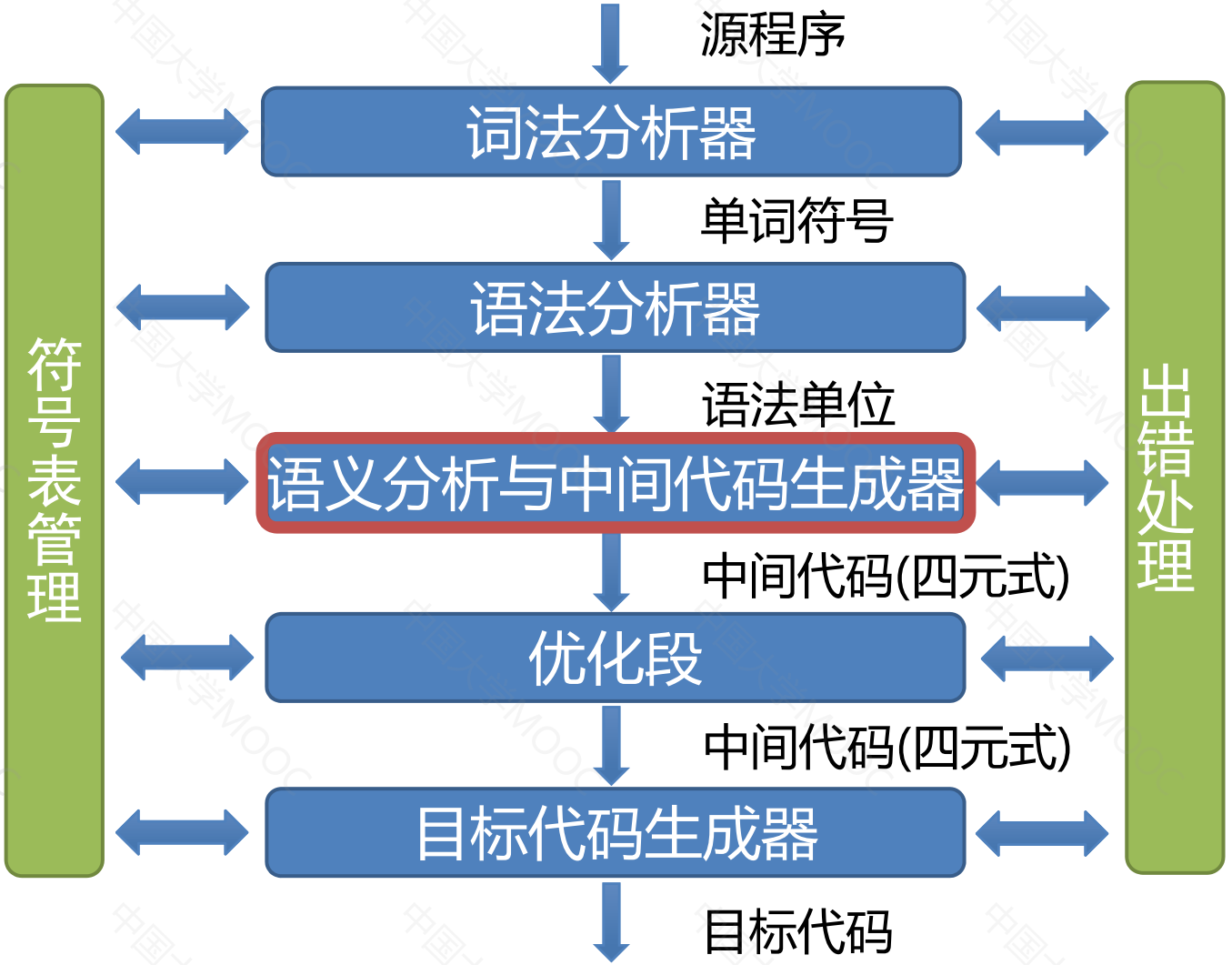
编译原理

中间语言

编译程序总框


源程序
词法分析器
单词符号
语法分析器
语法单位
语义分析与中间代码生成器
中间代码(四元式)
优化段
中间代码(四元式)
目标代码生成器
目标代码
符号表管理
出错处理

编译原理

中间语言的特点和作用

中间语言的特点和作用

- 计算思维
  - 分解
  - 权衡

- 特点
  - 独立于机器
  - 复杂性界于源语言和目标语言之间
- 引入中间语言的优点
  - 使编译程序的结构在逻辑上更为简单明确
  - 便于进行与机器无关的代码优化工作
  - 易于移植

常用的中间语言

- 后缀式，逆波兰表示
- 图表示： 抽象语法树(AST)、有向无环图(DAG)
- 三地址代码
  - 三元式
  - 四元式
  - 间接三元式

编译原理

后缀式

后缀式

- 后缀式表示法：Lukasiewicz发明的一种表示表达式的方法，又称逆波兰表示法。
- 一个表达式E的后缀形式可以如下定义
  - 如果E是一个变量或常量，则E的后缀式是E自身。
  - 如果E是E_1 op E_2形式的表达式，其中op是任何二元操作符，则E的后缀式为E_1' E_2' op，其中E_1'和E_2'分别为E_1和E_2的后缀式。
  - 如果E是(E_1)形式的表达式，则E_1的后缀式就是E的后缀式。

后缀式

- 后缀式表示法不用括号
  - 只要知道每个算符的目数，对于后缀式，不论从哪一端进行扫描，都能对它进行无歧义地分解。
- 后缀式的计算
  - 用一个栈实现
  - 自左至右扫描后缀式，每碰到运算量就把它推进栈。每碰到k目运算符就把它作用于栈顶的k个项，并用运算结果代替这k个项。

将表达式翻译成后缀式的属性文法

产生式	语义规则
$E \rightarrow E_1 op\ E_2$	$E.code := E_1.code \parallel E_2.code \parallel op$
$E \rightarrow (E_1)$	$E.code := E_1.code$
$E \rightarrow id$	$E.code := id$

- E.code表示E后缀形式
- op表示任意二元操作符
- “||”表示后缀形式的连接

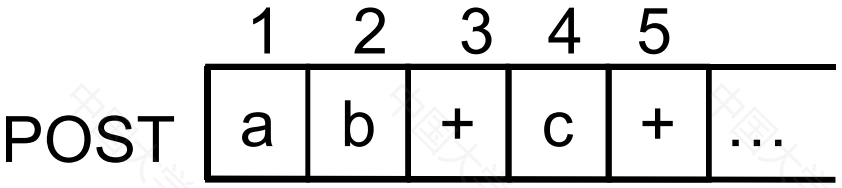
中缀表达式翻译成后缀式的翻译模式

- 数组POST存放后缀式：k为下标，初值为1

产生式	语义规则
$E \rightarrow E_1 op\ E_2$	$E.code := E_1.code \parallel E_2.code \parallel op$
$E \rightarrow (E_1)$	$E.code := E_2.code$
$E \rightarrow id$	$E.code := id$

产生式	程序段
$E \rightarrow E_1 op\ E_2$	{ POST[k]:=op; k:=k+1 }
$E \rightarrow (E_1)$	{ }
$E \rightarrow id$	{ POST[k]:=id; k:=k+1 }

a+b+c的分析和翻译：



编译原理

图表示法

图表示法

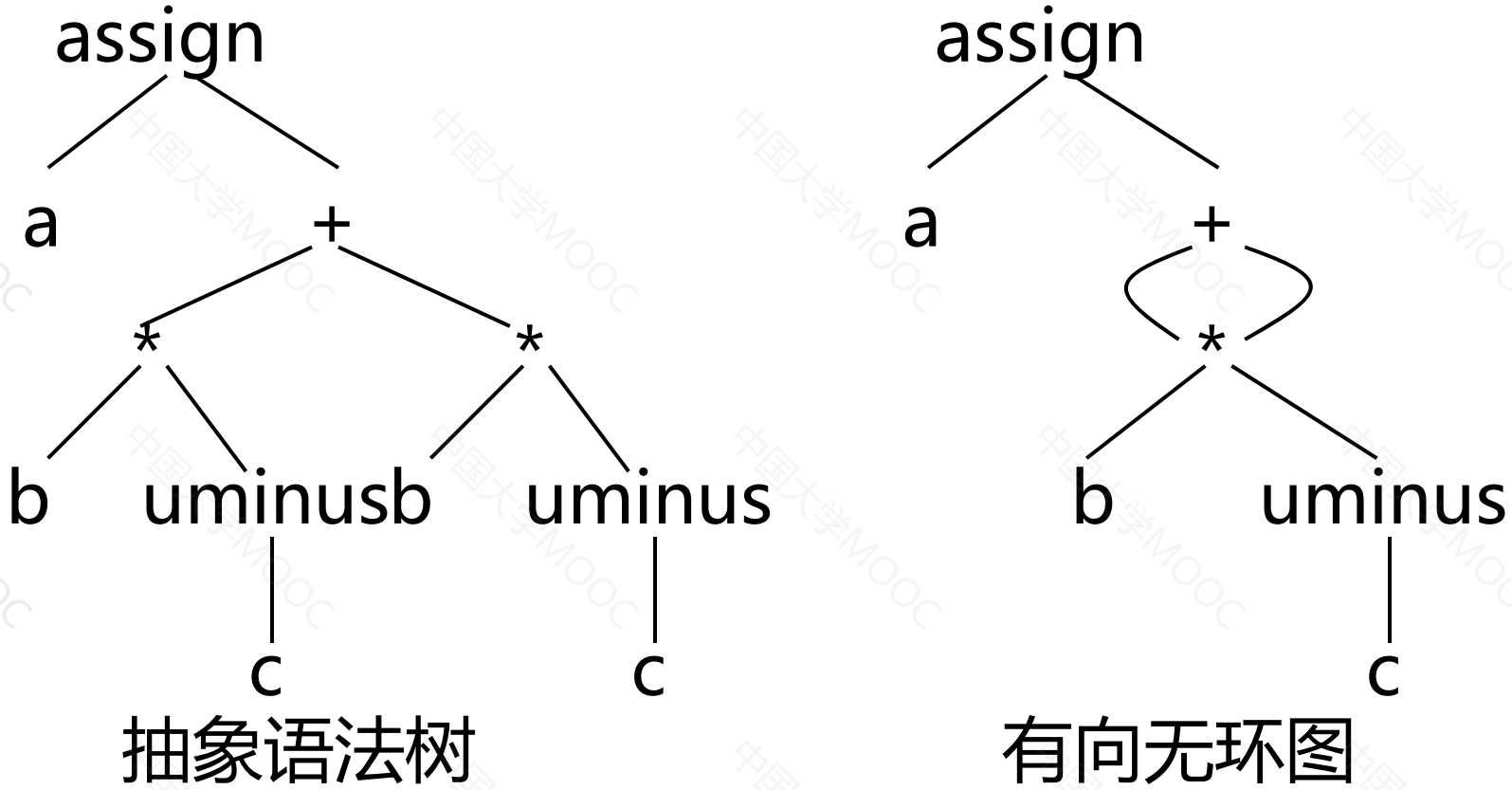
- 抽象语法树(AST)
- 有向无环图(DAG)

有向无环图(DAG)

- 有向无环图(Directed Acyclic Graph，简称DAG)
  - 对表达式中的每个子表达式，DAG中都有一个结点
  - 一个内部结点代表一个操作符，它的孩子代表操作数
  - 在一个DAG中代表公共子表达式的结点具有多个父结点

抽象语法树 vs. 有向无环图

- a:=b*(-c)+b*(-c)的图表示法



赋值语句翻译成抽象语法树的属性文法

产生式	语义规则
S→id:=E	S.nptr:=mknode(‘assign’ , mkleaf(id, id.place), E.nptr)
$E \rightarrow E_1+E_2$	E.nptr:=mknode(‘+’ , E_1.nptr, E_2.nptr)
$E \rightarrow E_1*E_2$	E.nptr:=mknode(‘*’ , E_1.nptr, E_2.nptr)
$E \rightarrow -E_1$	E.nptr:=mknode(‘uminus’ , E_1.nptr)
$E \rightarrow (E_1)$	E.nptr:=E_1.nptr
E→id	E.nptr:=mkleaf(id, id.place)

编译原理

三地址代码

三地址代码

- 三地址代码

  x:=y op z

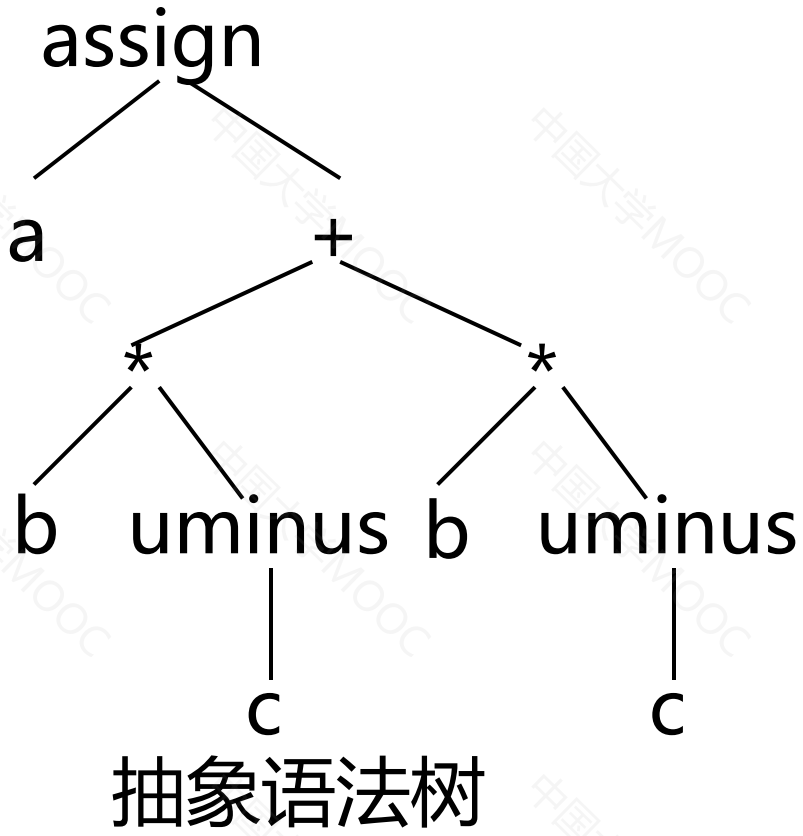
- 三地址代码可以看成是抽象语法树或有向无环图的一种线性表示

抽象语法树 vs. 三地址代码

- a:=b*(-c)+b*(-c)的图表示法

抽象语法树对应的三地址代码：

$T_1 := -c$
$T_2 := b * T_1$
$T_3 := - c$
$T_4 := b * T_3$
$T_5 := T_2 + T_4$
$a := T_5$



抽象语法树

有向无环图 vs. 三地址代码

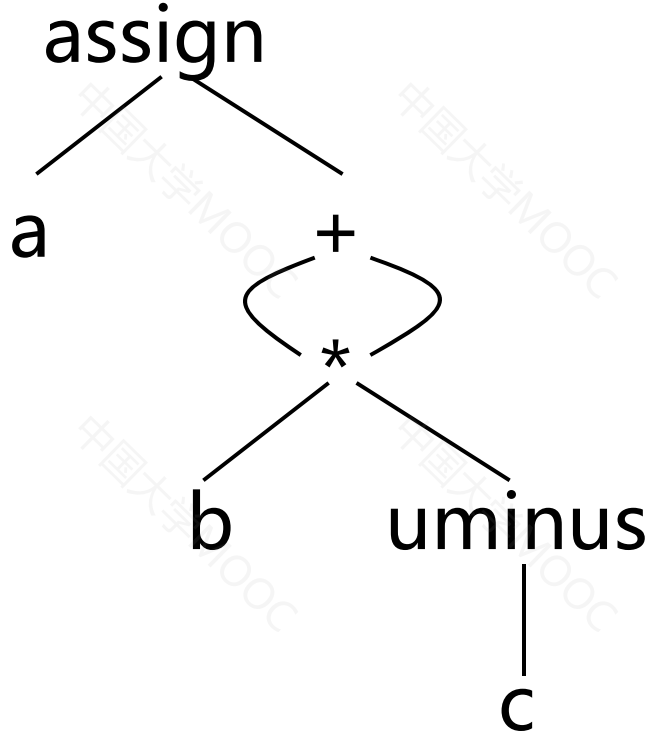
- a:=b*(-c)+b*(-c)的图表示法

抽象语法树对应的三地址代码：

$T_1 := -c$
$T_2 := b * T_1$
$T_3 := - c$
$T_4 := b * T_3$
$T_5 := T_2 + T_4$
$a := T_5$

有向无环图对应的三地址代码：

$T_1 := - c$
$T_2 := b * T_1$
$T_5 := T_2 + T_2$
$a := T_5$



有向无环图

三地址语句的种类

- x:=y op z
- x:=op y
- x:=y
- goto L
- if x relop y goto L或if a goto L
- 传参、转子：param x、call p,n
- 返回语句：return y
- 索引赋值：x:=y[i]、x[i]:=y
- 地址和指针赋值：x:=&y、x:=*y、*x:=y

三地址语句——四元式

- 一个带有四个域的记录结构，这四个域分别称为op, arg1, arg2及result
- a:=b*(-c)+b*(-c)的四元式形式

序号	OP	arg1	arg2	result
(0)	uminus	c	-	T_1
(1)	*	b	T_1	T_2
(2)	uminus	c	-	T_3
(3)	*	b	T_3	T_4
(4)	+	T_2	T_4	T_5
(5)	:=	T_5	-	a

三地址语句——三元式

- 用三个域表示：op、arg1和arg2
- 计算结果引用：引用计算该值的语句的位置
- a:=b*(-c)+b*(-c)的三元式形式

序号	OP	arg1	arg2
(0)	uminus	c	-
(1)	*	b	(0)
(2)	uminus	c	-
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

三地址语句——三元式

- x[i]:=y

序号	OP	arg1	arg2
(0)	[]=	x	i
(1)	:=	(0)	y

- x:=y[i]

序号	OP	arg1	arg2
(0)	=[]	y	i
(1)	:=	x	(0)

三地址语句——三元式

- 用三个域表示：op、arg1和arg2
- 计算结果引用：引用计算该值的语句的位置
- a:=b*(-c)+b*(-c)的三元式形式

序号	OP	arg1	arg2
(0)	uminus	c	-
(1)	*	b	(0)
(2)	uminus	c	-
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

三地址语句——间接三元式

- 三元式表+间接码表
- 间接码表
  - 一张指示器表，按运算的先后次序列出有关三元式在三元式表中的位置
- 优点
  - 方便优化，节省空间

三地址语句——间接三元式

- a:=b*(-c)+b*(-c)的间接三元式形式

间接码表

(0)
(1)
(2)
(3)

序号	OP	arg1	arg2
(0)	uminus	c	-
(1)	*	b	(0)
(2)	+	(1)	(1)
(3)	:=	a	(2)

三地址语句——间接三元式

- 语句

  X:=(A+B)*C;

  Y:=D↑(A+B)

  的间接三元式

间接码表

(0)
(1)
(2)
(0)
(3)
(4)

序号	OP	arg1	arg2
(0)	+	A	B
(1)	*	(0)	C
(2)	:=	X	(1)
(3)	↑	D	(0)
(4)	:=	Y	(3)

小结

- 后缀式，逆波兰表示
- 图表示：抽象语法树(AST)、有向无环图(DAG)
- 三地址代码
  - 三元式
  - 间接三元式
  - 四元式