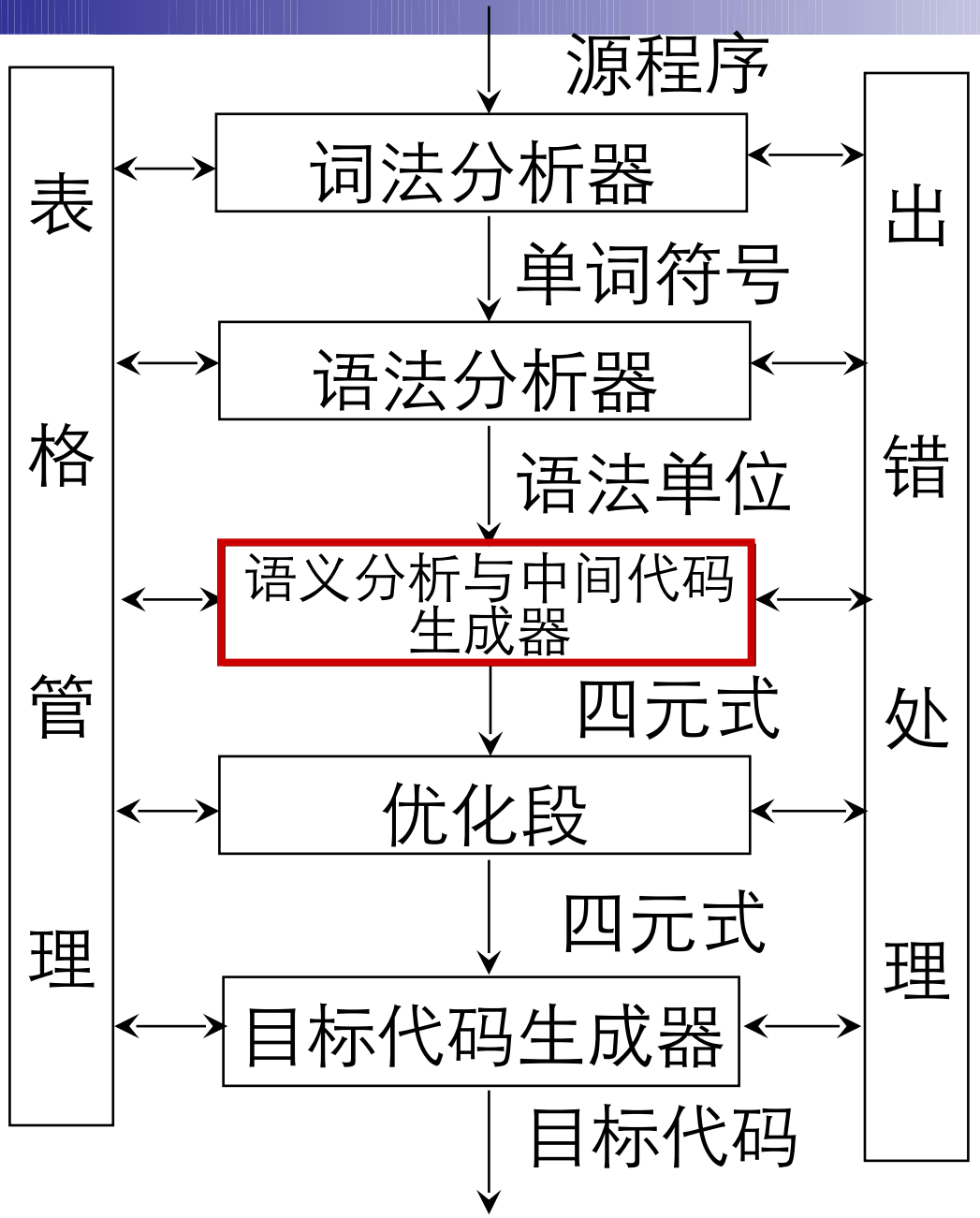
# 编译原理

## 第七章 语义分析和中间代码产生

# 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
四元式
优化段
四元式
目标代码生成器
目标代码
表格管理
出错处理

# 第七章 语义分析和中间代码产生

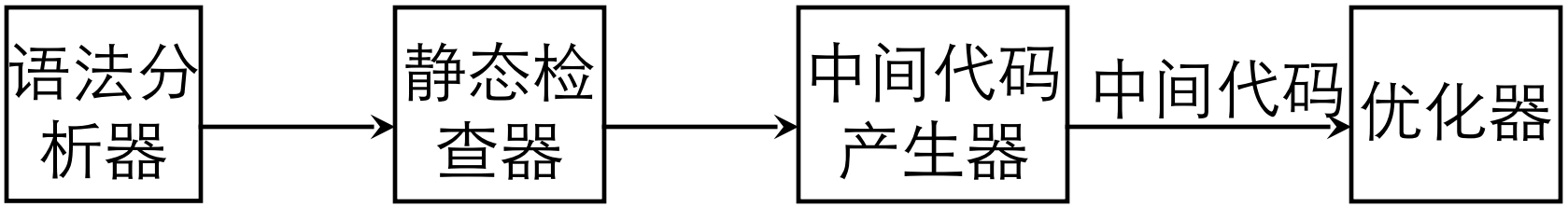
- 中间语言
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译
- 过程调用的处理

# 第七章 语义分析和中间代码产生

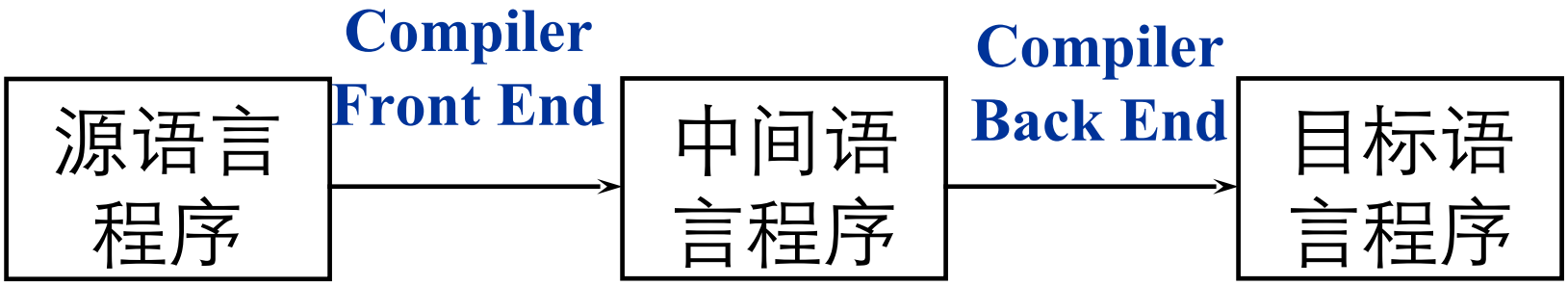
- 中间语言
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译
- 过程调用的处理

# 第七章 语义分析和中间代码产生

- 静态语义检查
  - 类型检查
  - 控制流检查
  - 一致性检查
  - 相关名字检查
  - 名字的作用域分析



- 中间语言
  - 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
  - 便于进行与机器无关的代码优化工作
  - 易于移植
  - 使编译程序的结构在逻辑上更为简单明确



# 7.1 中间语言

- 常用的中间语言
  - 后缀式，逆波兰表示
  - 图表示： DAG 、抽象语法树
  - 三地址代码
    - 三元式
    - 四元式
    - 间接三元式

# 7.1.1 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

- 逆波兰表示法不用括号
  - 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，都能对它进行唯一分解。
- 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语义规则

| 产生式 | 语义规则 |
| --- | --- |
| $E \rightarrow E^{(1)} op\ E^{(2)}$ | $E.code := E^{(1)}.code \mid\mid E^{(2)}.code \mid\mid op$ |
| $E \rightarrow (E^{(1)})$ | $E.code := E^{(1)}.code$ |
| $E \rightarrow id$ | $E.code := id$ |

- 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

| | |
|---|---|
| $E \rightarrow E^{(1)} op\ E^{(2)}$ | E.code:= $E^{(1)}$.code \|\| $E^{(2)}$.code \|\|op |
| $E \rightarrow (E^{(1)})$ | E.code:= $E^{(1)}$.code |
| $E \rightarrow id$ | E.code:=id |

- 数组 POST 存放后缀式： k 为下标，初值为 1
- 上述语义规则可实现为：

| 产生式 | 程序段 |
|---|---|
| $E \rightarrow E^{(1)} op\ E^{(2)}$ | {POST[k]:=op;k:=k+1} |
| $E \rightarrow (E^{(1)})$ | {} |
| $E \rightarrow i$ | {POST[k]:=i;k:=k+1} |

- 例： 输入串 a+b+c 的分析和翻译

POST :

| a 1 | b 2 | + 3 | c 4 | + 5 | … |
|---|---|---|---|---|---|

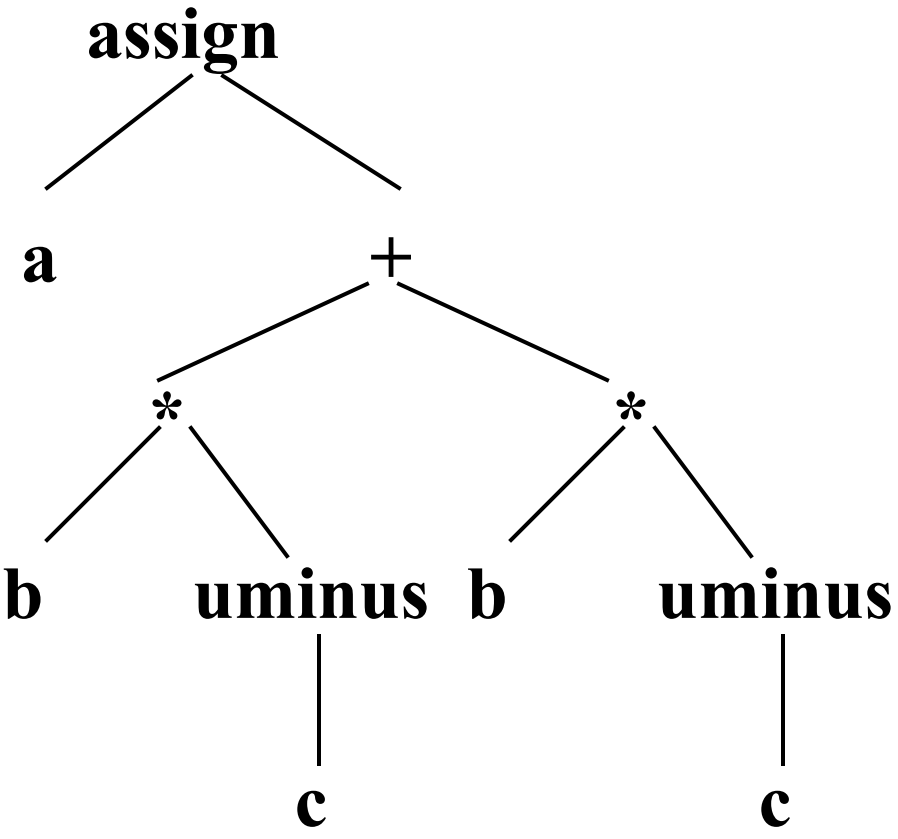
# 7.1.2 图表示法

- 图表示法
  - DAG
  - 抽象语法树

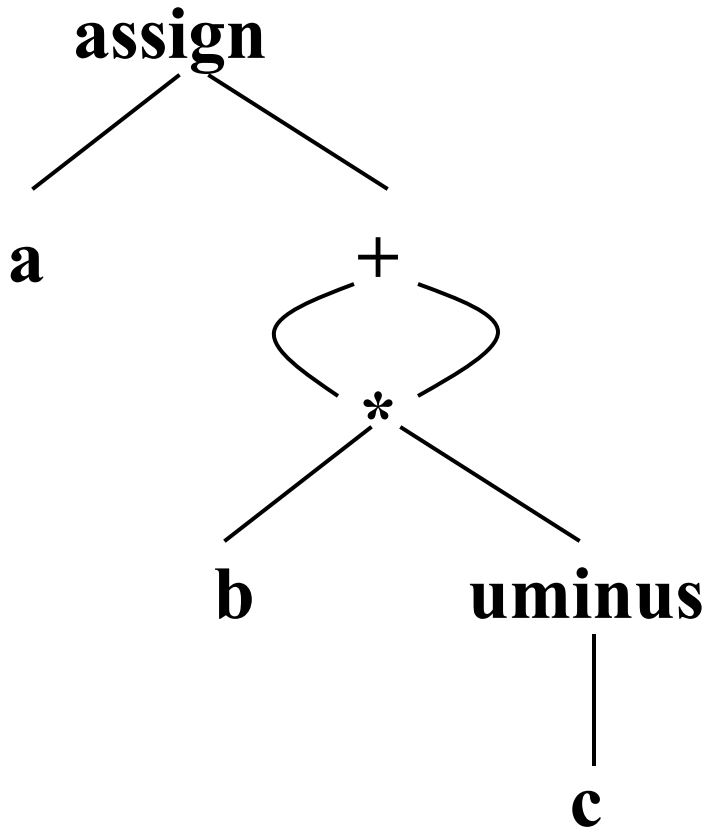
# 无循环有向图 (DAG)

- 无循环有向图 (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

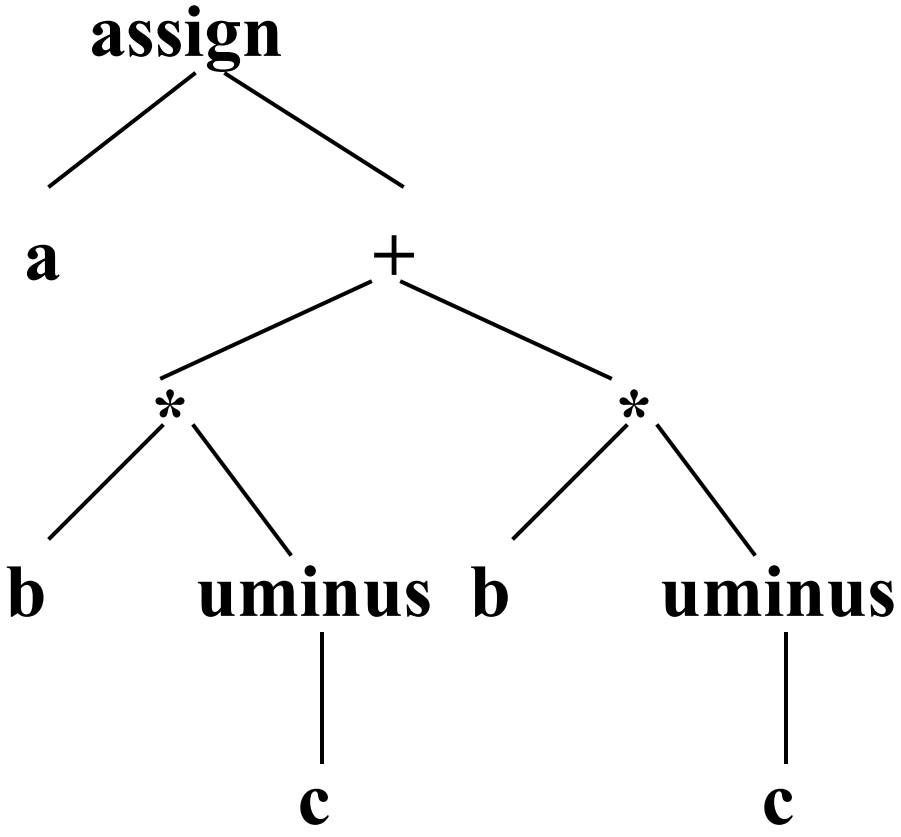
# a:=b*(-c)+b*(-c) 的图表示法




抽象语法树




DAG




抽象语法树

抽象语法树对应的代码：

$T_1$:=-c

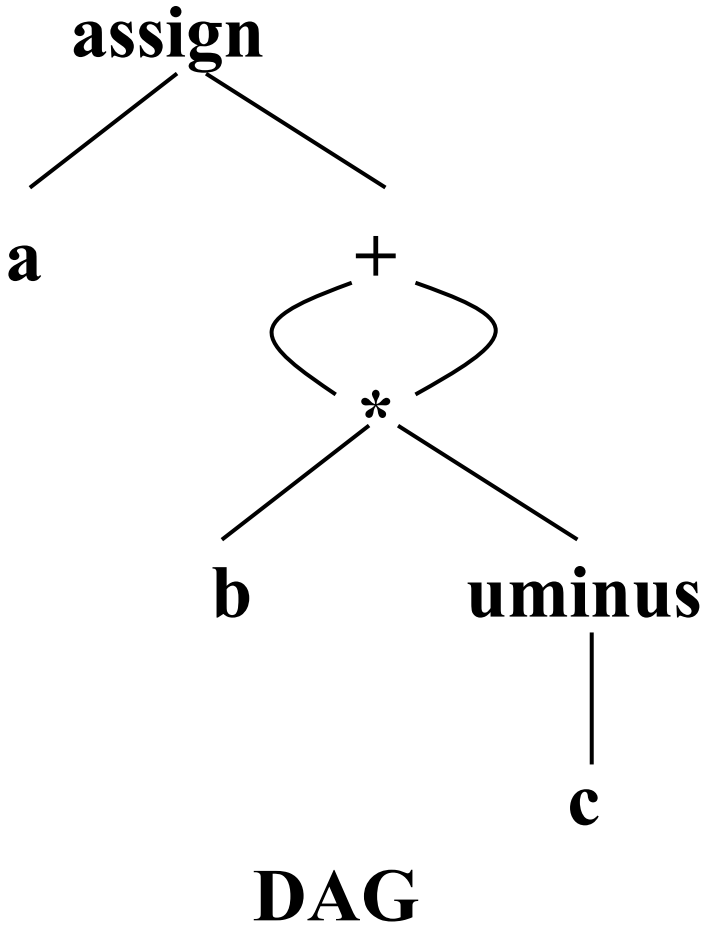
$T_2$:=b*$T_1$

$T_3$:=-c

$T_4$:=b*$T_3$

$T_5$:=$T_2$+$T_4$

a:=$T_5$




DAG

抽象语法树对应的代码：

$T_1$:=-c

$T_2$:=b*$T_1$

$T_3$:=-c

$T_4$:=b*$T_3$

$T_5$:=$T_2$+$T_4$

a:=$T_5$

DAG 对应的代码：

$T_1$:=-c

$T_2$:=b*$T_1$

$T_5$:=$T_2$+$T_2$

a:=$T_5$

# 产生赋值语句抽象语法树的属性文法

| 产 生 式 | 语义规则 |
| --- | --- |
| $S \rightarrow id:=E$ | 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 |
| $E \rightarrow id$ | E.nptr:=mkleaf( id, id.place ) |

# 7.1.3 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 DAG 的一种线性表示

# a:=b*(-c)+b*(-c) 的图表示法

DAG 对应的三地址代码：

$T_1$:=-c

$T_2$:=b*$T_1$

$T_5$:=$T_2$+$T_2$

a:=$T_5$

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:=$T_5$

# 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

a:=b*(-c)+b*(-c)

- 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

| | 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

a:=b*(-c)+b*(-c)

- 三元式
  - 三个域： op 、 arg1 和 arg2
  - 引用临时变量 ( 中间结果 ) ： 通过计算该值的语句的位置

| | op | arg1 | arg2 |
|---|---|---|---|
| (0) | uminus | c | |
| (1) | * | b | (0) |
| (2) | uminus | c | |
| (3) | * | b | (2) |
| (4) | + | (1) | (3) |
| (5) | assign | a | (4) |

# 三地址语句

- x[i]:=y

| | op | arg1 | arg2 |
|---|---|---|---|
| (0) | [ ] = | x | i |
| (1) | assign | (0) | y |

- x:=y[i]

| | op | arg1 | arg2 |
|---|---|---|---|
| (0) | = [ ] | y | i |
| (1) | assign | x | (0) |

# 三地址语句

a:=b*(-c)+b*(-c)

- 三元式
  - 三个域： op 、 arg1 和 arg2
  - 引用临时变量 ( 中间结果 ) ： 通过计算该值的语句的位置

| | op | arg1 | arg2 |
|---|---|---|---|
| (0) | uminus | c | |
| (1) | * | b | (0) |
| (2) | uminus | c | |
| (3) | * | b | (2) |
| (4) | + | (1) | (3) |
| (5) | assign | 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

- 例如，语句 a:=b*(-c)+b*(-c)

的间接三元式表示如下表所示

| 间接代码 | | 三元式表 op | arg1 | arg2 |
|---|---|---|---|---|
| (0) | (0) | uminus | c | |
| (1) | (1) | * | b | (0) |
| (2) | (2) | + | (1) | (1) |
| (3) | (3) | assign | a | (2) |

- 例如，语句

X:=(A+B)*C;
Y:=D↑(A+B)

的间接三元式表示如下表所示

| 间接代码 | 三元式表 | | | |
|---|---|---|---|---|
| | | OP | ARG1 | ARG2 |
| (1) | (1) | + | A | B |
| (2) | (2) | * | (1) | C |
| (3) | (3) | := | X | (2) |
| (1) | (4) | ↑ | D | (1) |
| (4) | (5) | := | Y | (4) |
| (5) | | | | |

# 小结

- 常用的中间语言
  - 后缀式，逆波兰表示
  - 图表示： DAG 、抽象语法树
  - 三地址代码
    - 三元式
    - 四元式
    - 间接三元式

# 作业

- P217-1， 3