习题解答HW4+6

金泽文

Fall 2023

作业

- HW4:
  - 讲义——布尔表达式的翻译
- HW6:
  - 7.2 c、7.5、7.12、8.1 e、8.2 e、8.6

HW4题目

- **考虑布尔表达式**

a>b or true and not c < f

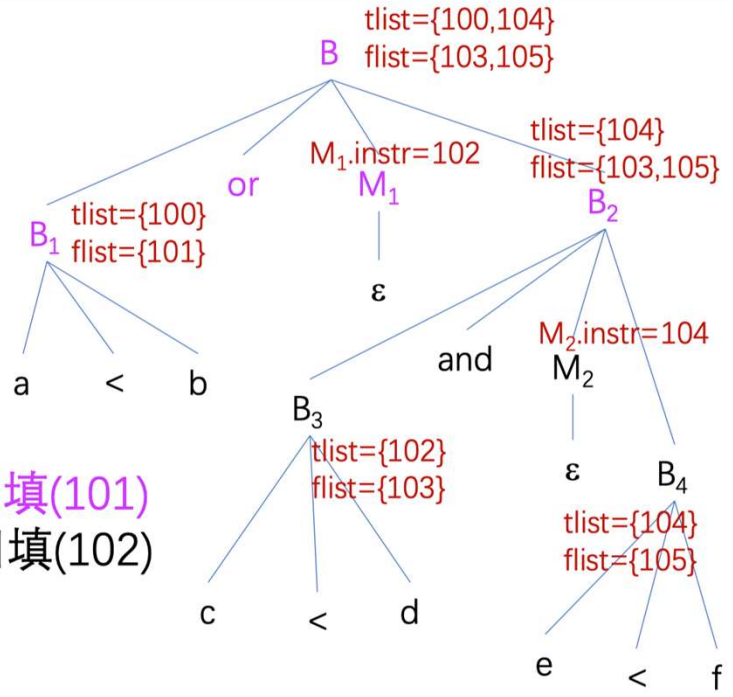
- 参考“中间代码生成——part3”ppt中布尔表达式短路计算、标号回填等翻译技术，生成对应的三地址代码。
- 假设nextinstr = 200
- 除了三地址代码外，画出LR分析方法对应的注释分析树（如slide 35），标注出属性和属性值。
- 结合LR分析方法指出回填的具体细节
  - 在使用哪一个产生式归约时候进行的回填
  - 用哪一个标号，回填了哪一个不完整的goto指令

讲义PPT里的第35页

布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

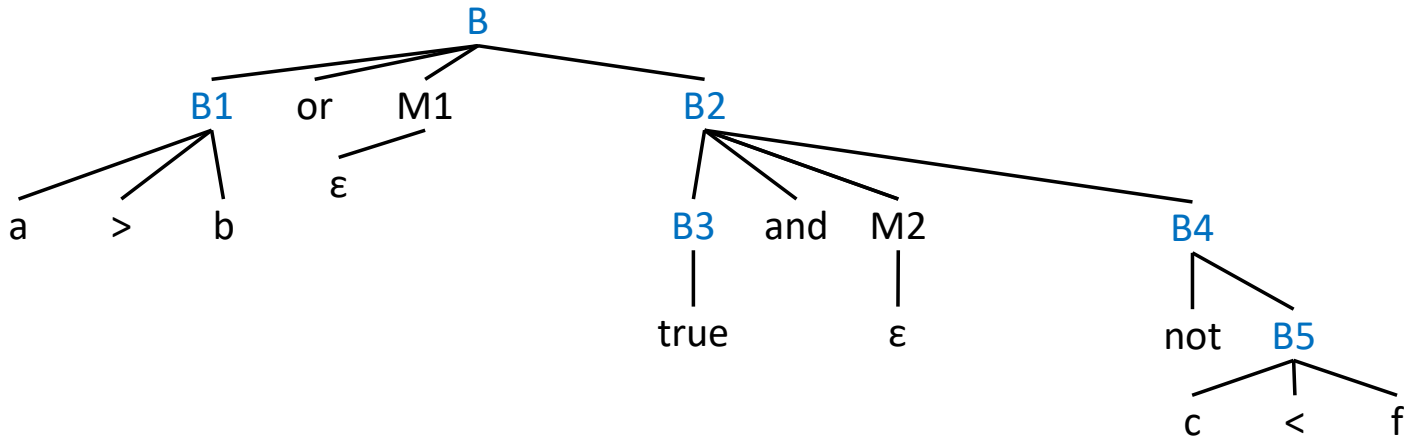


(100) if a<b goto -
(101) goto 102 //用102回填(101)
(102) if c<d goto 104//用104回填(102)
(103) goto -
(104) if e<f goto -
(105) goto -

其他部分的回填要依赖与其他语句的翻译

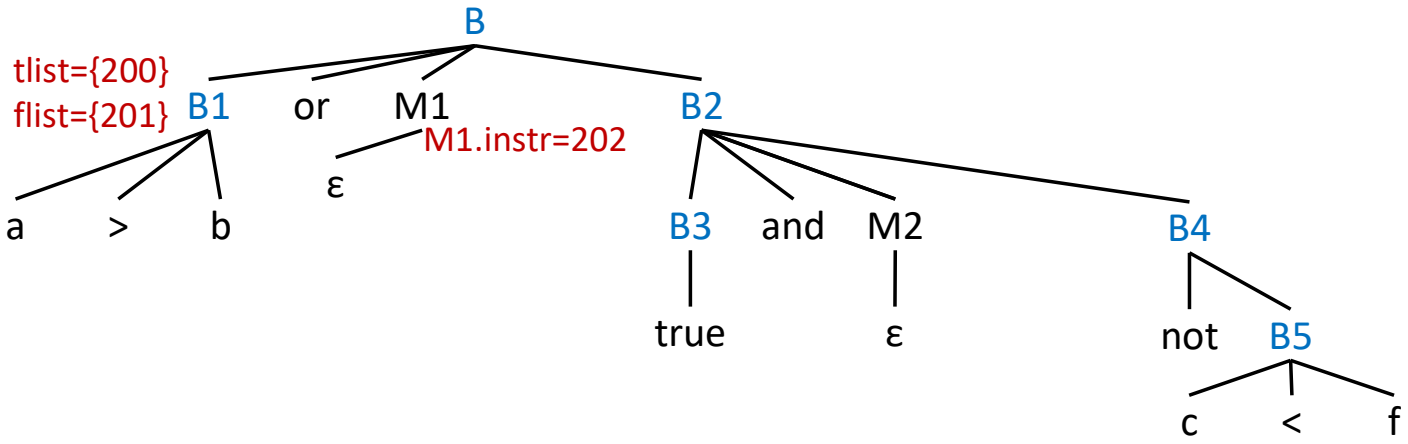
HW4题目

a>b or true and not c < f



HW4题目

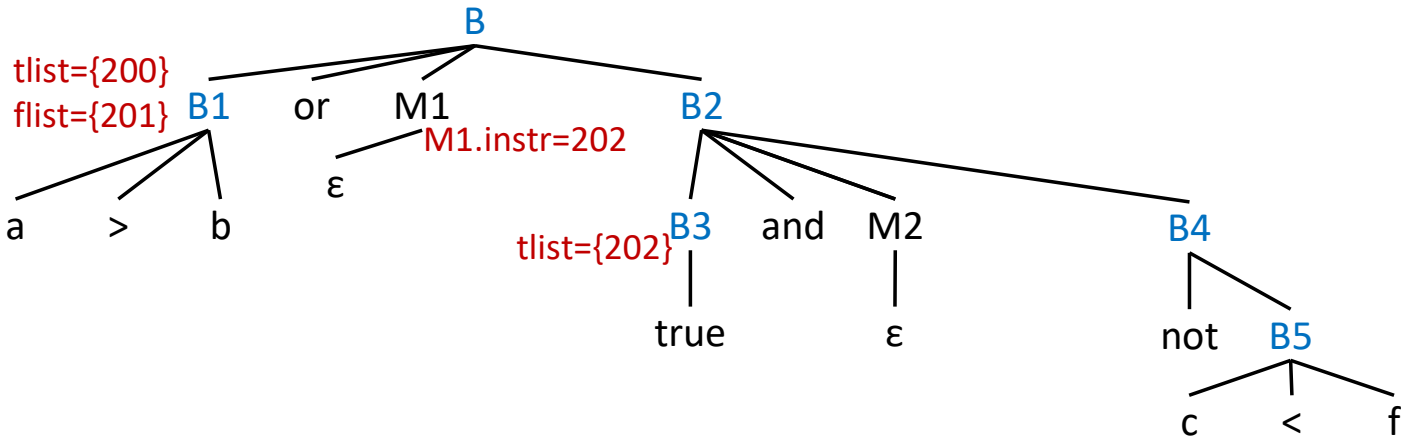
a>b or true and not c < f



(200) if a > b goto -
(201) goto -

HW4题目

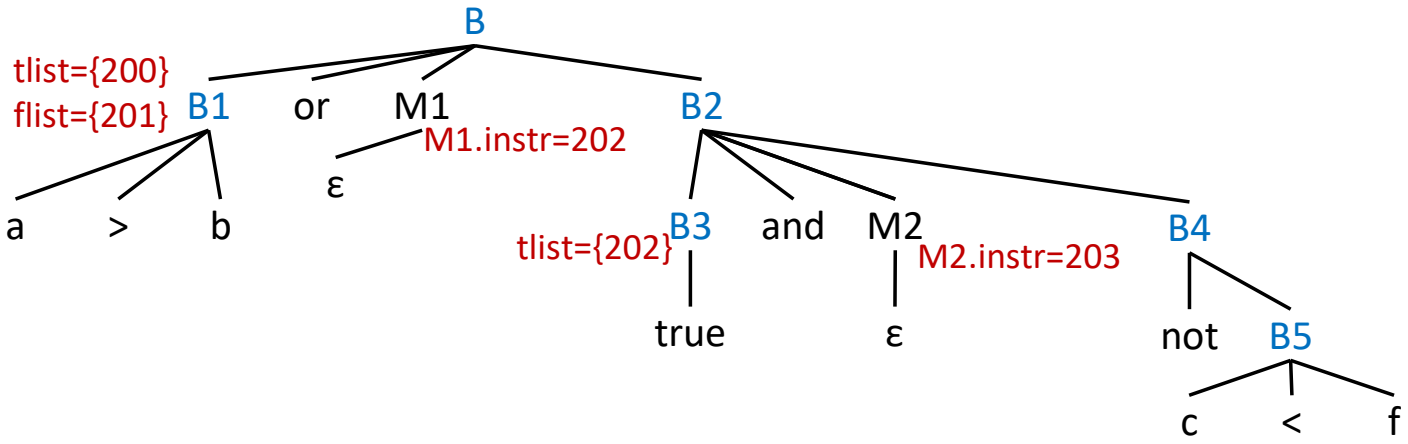
a>b or true and not c < f



(200) if a > b goto -
(201) goto -
(202) goto -

HW4题目

a>b or true and not c < f

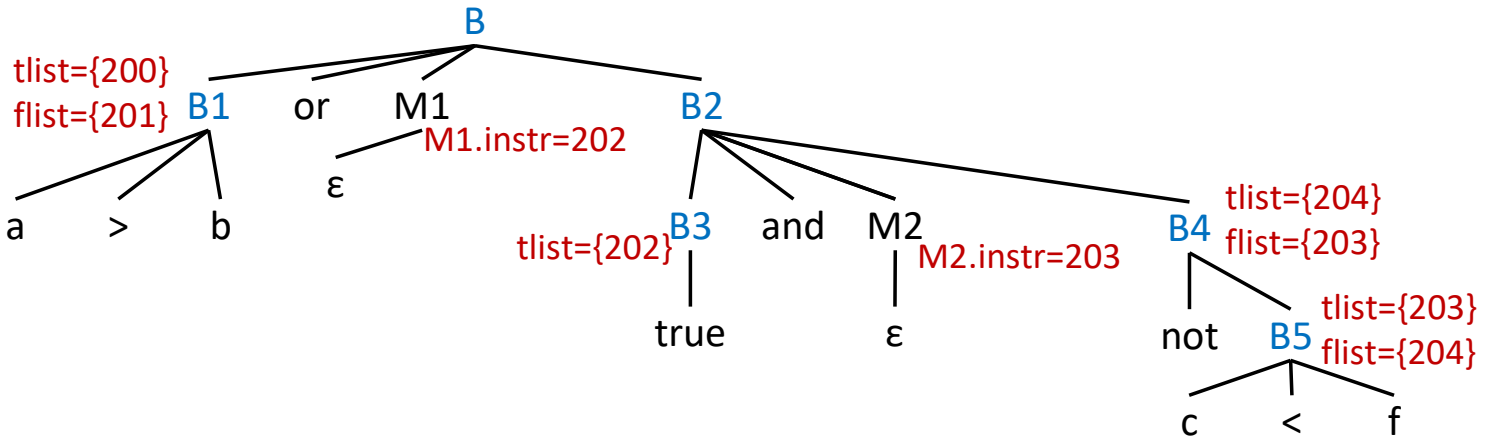


(200) if a > b goto -
(201) goto -
(202) goto -

HW4题目

USTC, CHINA ADSLAB

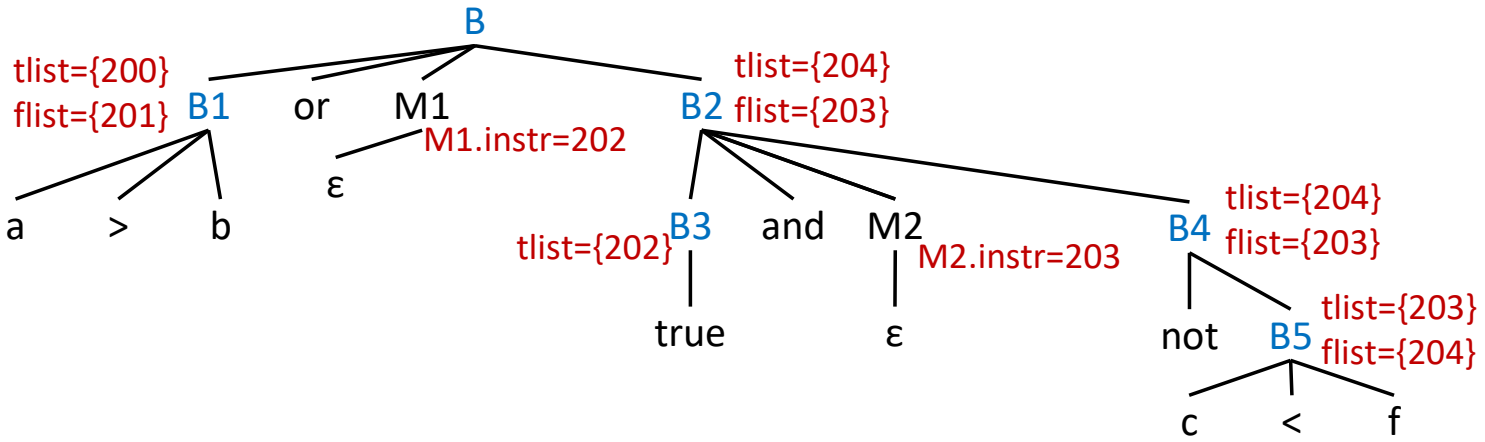
a>b or true and not c < f



(200) if a > b goto -
(201) goto -
(202) goto -
(203) if c < f goto -
(204) goto -

HW4题目

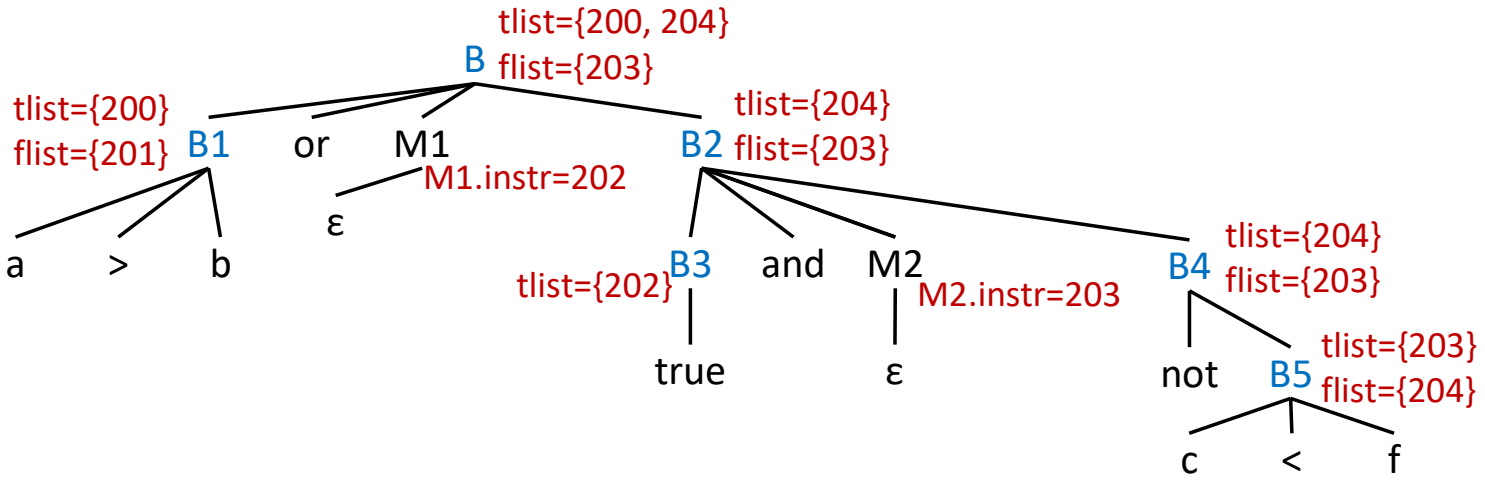
a>b or true and not c < f



(200) if a > b goto -
(201) goto -
(202) goto **203 // B→B1 and M B2**时回填，用**M2.instr = 203**回填
(203) if c < f goto -
(204) goto -

HW4题目

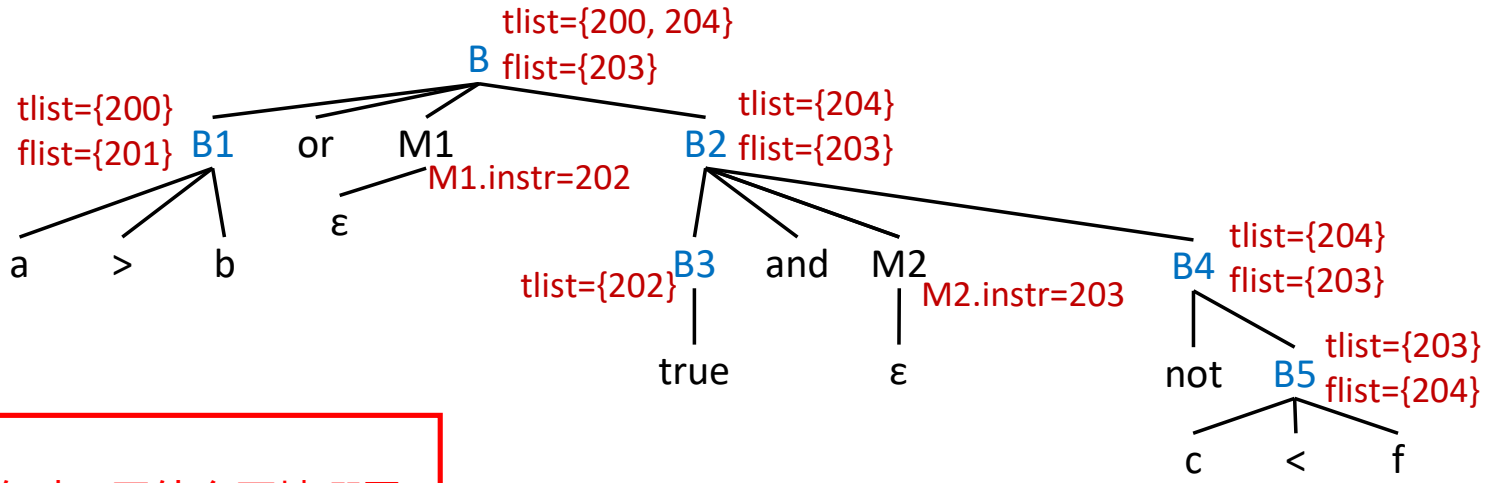
a>b or true and not c < f



(200) if a > b goto -
(201) goto **202 // B→B1 or M B2** 时回填，用**M1.instr = 202**回填
(202) goto 203 // B→B1 and M B2时回填，用M2.instr = 203回填
(203) if c < f goto -
(204) goto -

HW4题目

a>b or true and not c < f



注：
需要标记遇到什么规约时，用什么回填哪里

(200) if a > b goto -
(201) goto 202 // B→B1 or M B2 时回填，用M1.instr = 202回填
(202) goto 203 // B→B1 and M B2时回填，用M2.instr = 203回填
(203) if c < f goto -
(204) goto -

其他部分的回填依赖于其他语句的翻译

7.2c

题目：
把 C 程序的可执行语句翻译成三地址代码

```
main() {
    int i;
    int a[10];
    while (i <= 10)
        a[i] = 0;
}
```

假定 int 的大小为 4 字节：

```
L1: if i <= 10 goto L2
    goto L3
L2: t1 = i * 4 # 计算数组元素地址
    a[t1] = 0
    goto L1
L3: ...
```

合理即可
注：
需要计算数组元素地址

7.5

题目：
修改图 7.5 中计算声明名字的类型和相对地址的翻译方案，offset 不是全局变量，而是文法符号的继承属性。

P →	{offset = 0;}
D;S	
D → D;D	
D → **id**:T	{enter(**id**.lexeme, T.type, offset); offset = offset + T.width}
T → **integer**	{T.type = integer; T.width = 4;}
T → **real**	{T.type = real; T.width = 8;}
T → **Array**[**num**] **of** T_1	{T.type = array(**num**.val, T_1.type); T.width = **num**.val * T_1.width}
T → ↑T_1	{T.type = pointer(T_1.type); T.width = 4;}

7.5

题目：
修改图 7.5 中计算声明名字的类型和相对地址的翻译方案，offset 不是全局变量，而是文法符号的继承属性。

修改后的翻译方案如下：

P →	{D.offset = 0;}
D;S	
D →	{D_1.offset = D.offset;}
D_1;	{D_2.offset = D_1.offset + D_1.width;}
D_2	
D → **id**:T	{enter(**id**.lexeme, T.type, D.offset); D.width = T.width}
T → **integer**	{T.type = integer; T.width = 4;}
T → **real**	{T.type = real; T.width = 8;}
T → **Array**[**num**] **of** T_1	{T.type = array(**num**.val, T_1.type); T.width = **num**.val * T_1.width}
T → ↑T_1	{T.type = pointer(T_1.type); T.width = 4;}

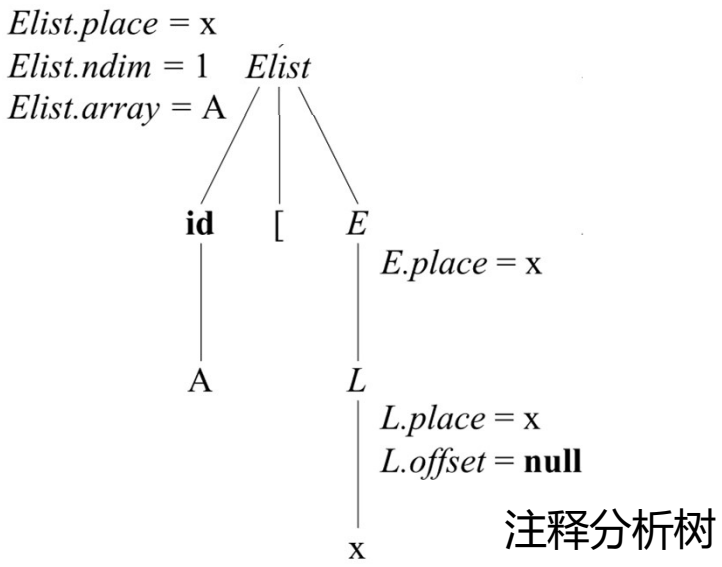
这里也可以为 D 添加一个综合属性 end，两个操作修改为：
{D_2.offset = D_1.end}
{…; D.end = D.offset + T.width}

7.12

题目：
用 7.3 节的翻译方案，把赋值语句 A[x, y] := z 翻译成三地址代码（其中 A 是 10 * 5 的数组）

7.12

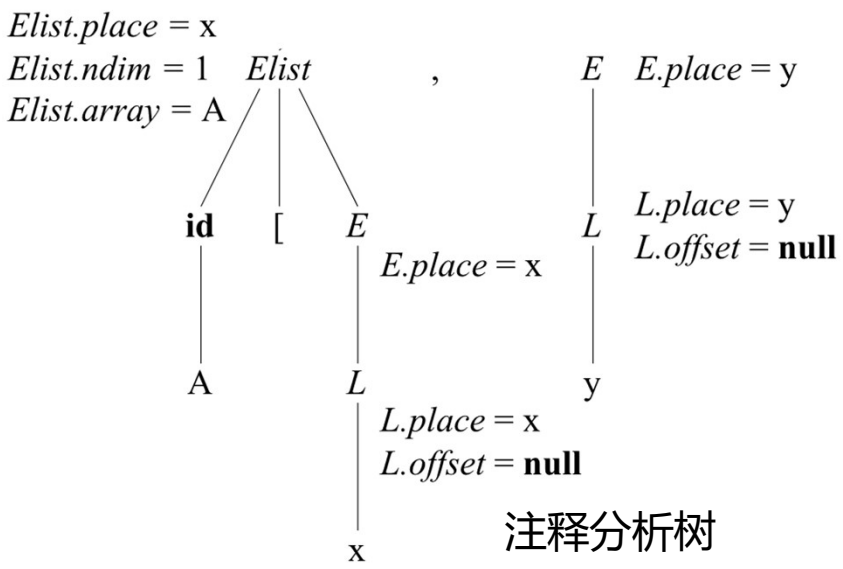
题目：
用 7.3 节的翻译方案，把赋值语句 A[x, y] := z 翻译成三地址代码（其中 A 是 10 * 5 的数组）



注释分析树

7.12

题目:
用 7.3 节的翻译方案，把赋值语句 A[x, y] := z 翻译成三地址代码（其中 A 是 10 * 5 的数组)



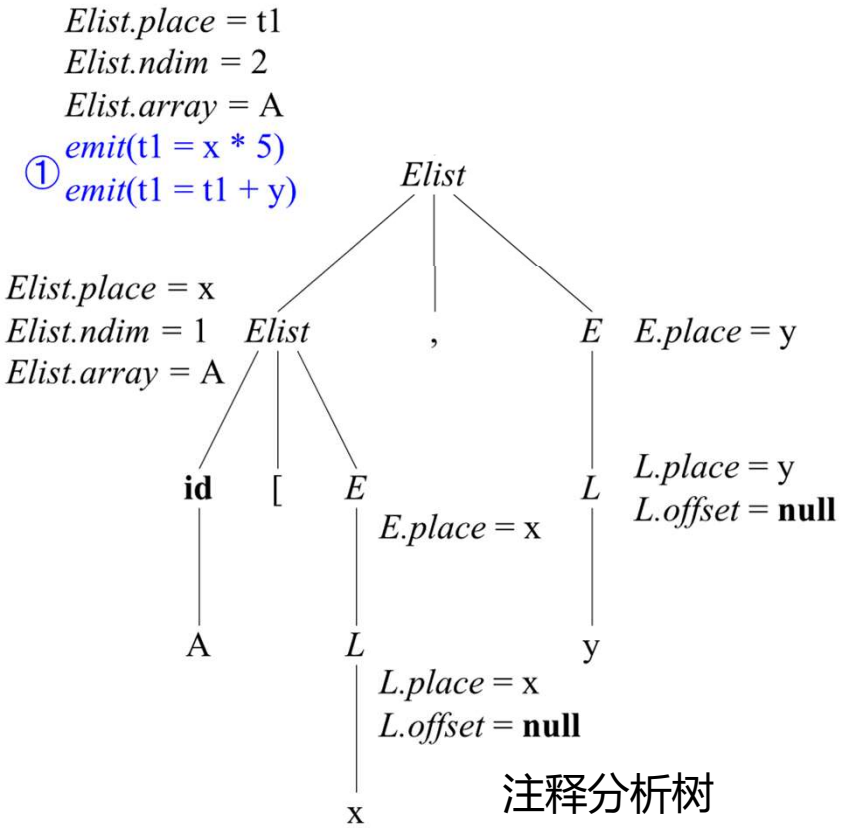
注释分析树

7.12

题目：
用 7.3 节的翻译方案，把赋值语句 A[x, y] := z 翻译成三地址代码（其中 A 是 10 * 5 的数组）

输出的三地址代码：

1: t1 = x * 5
2: t1 = t1 + y



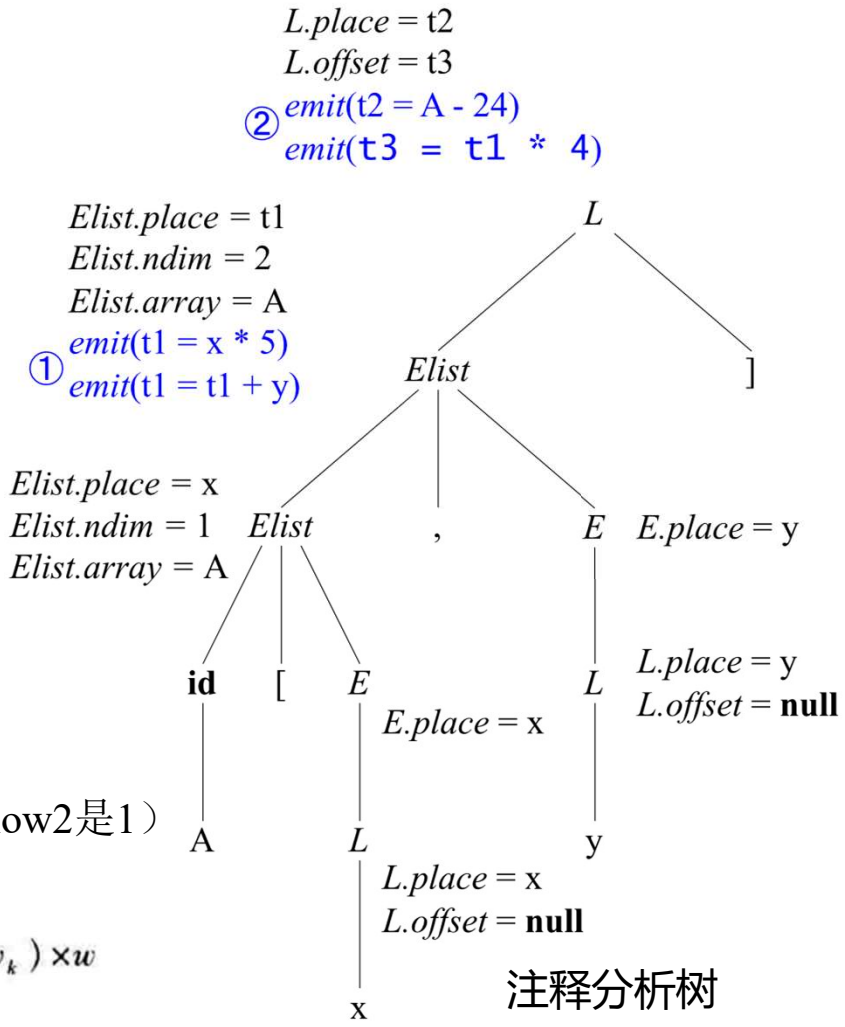
注释分析树

7.12

题目：
用 7.3 节的翻译方案，把赋值语句 A[x, y] := z 翻译成三地址代码（其中 A 是 10 * 5 的数组）

输出的三地址代码：

1: t1 = x * 5
2: t1 = t1 + y
3: t2 = A - 24
4: t3 = t1 * 4



注释分析树

24=((1*5)+1)*4（参考书上式7.4，注意low1和low2是1）

$$A[i_1, i_2, \cdots, i_k]$$

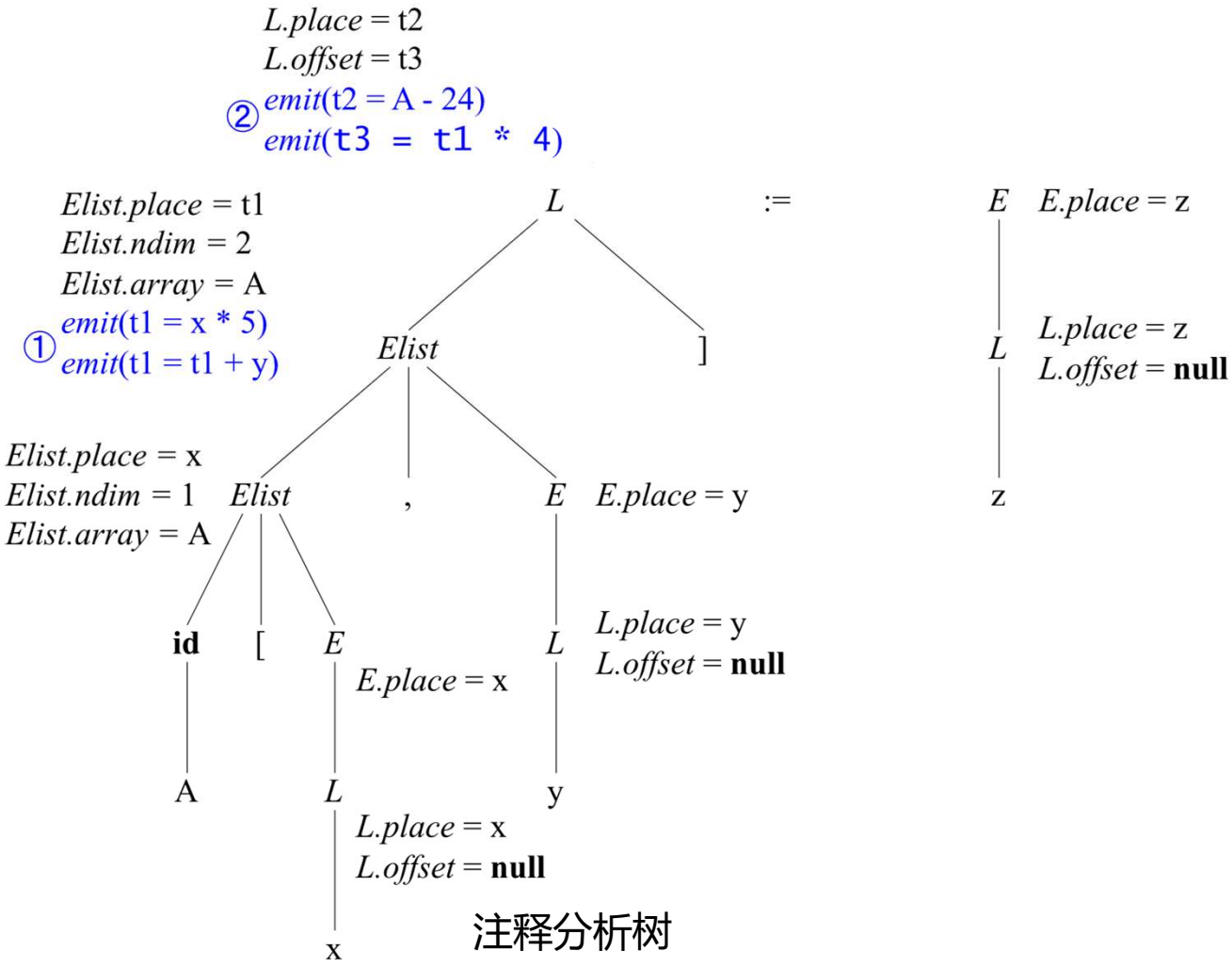
$$base-((\cdots((low_1 \times n_2 + low_2) \times n_3 + low_3)\cdots) \times n_k + low_k) \times w$$

7.12

题目：
用 7.3 节的翻译方案，把赋值语句 A[x, y] := z 翻译成三地址代码（其中 A 是 10 * 5 的数组）

输出的三地址代码：

1: t1 = x * 5
2: t1 = t1 + y
3: t2 = A - 24
4: t3 = t1 * 4



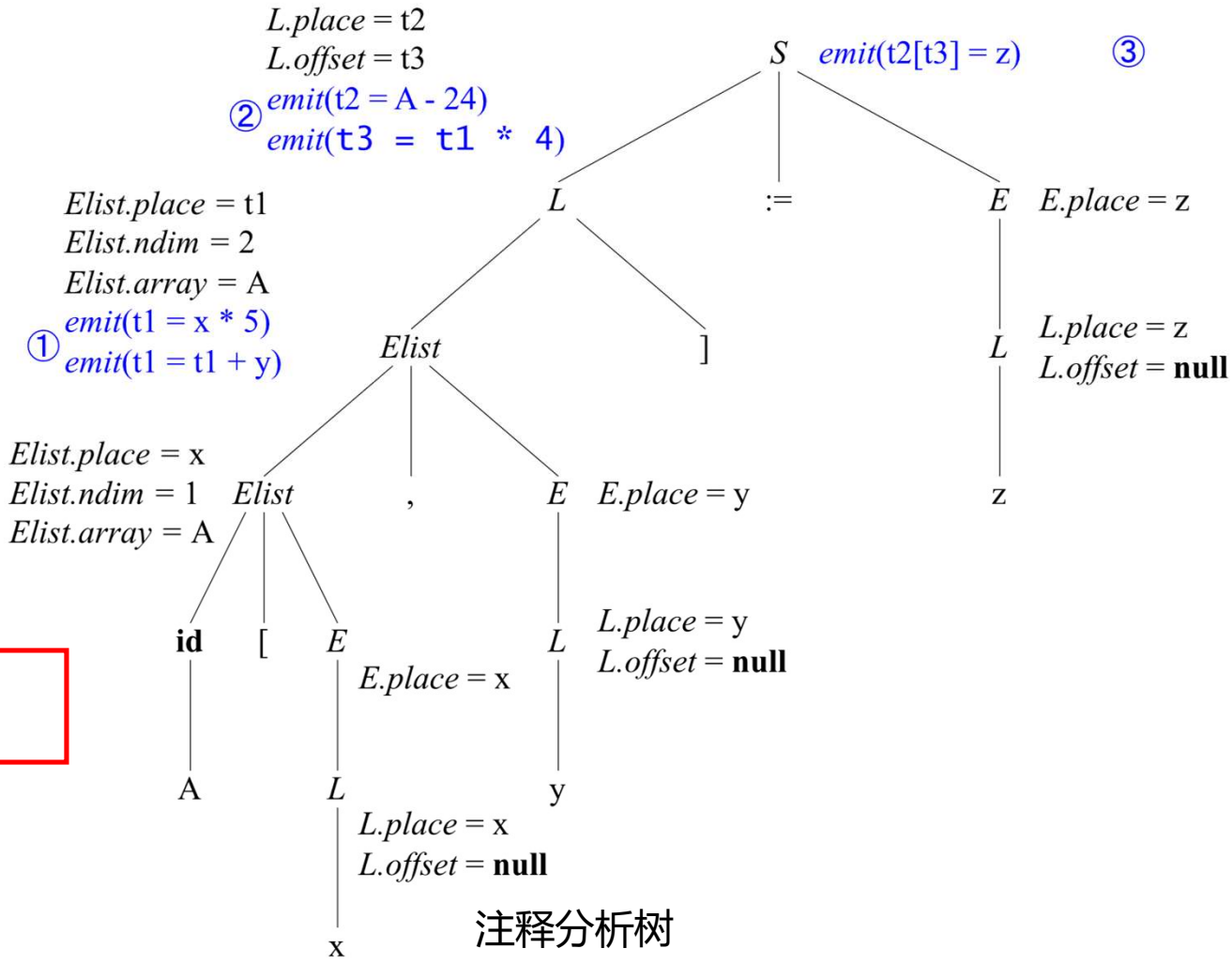
7.12

题目：
用 7.3 节的翻译方案，把赋值语句 A[x, y] := z 翻译成三地址代码（其中 A 是 10 * 5 的数组）

输出的三地址代码：

1: t1 = x * 5
2: t1 = t1 + y
3: t2 = A - 24
4: t3 = t1 * 4
5: t2[t3] = z

注：
解答里不要求画出注释分析树，但最好有



注释分析树

8.1e

题目：
为下列 C 语句产生 8.2 节目标机器的代码，假定所有的变量都是静态的，并假定有三个寄存器可用于保存计算结果。

x = a / (b + c) – d * (e + f)

目标机器代码如下：

```
MOV b, R0  # R0 = b
ADD c, R0  # R0 = b + c
MOV a, R1  # R1 = a
DIV R0, R1 # R1 = R1 / R0 = a / (b + c)
MOV e, R0  # R0 = e
ADD f, R0  # R0 = e + f
MUL d, R0  # R0 = d * (e + f)
SUB R0, R1 # R1 = R1 – R0 = a / (b + c) – d * (e + f) MOV R1, x  # x = R1
```

8.2e

题目：
为下列 C 语句产生 8.2 节目标机器的代码，假定所有的变量都是自动变量（分配在栈上），并假定有三个寄存器可用于保存计算结果。

x = a / (b + c) – d * (e + f)

直接使用 Sx(Rs) 表示 x
目标机器代码如下：

```
MOV Sb(Rs), R0  # R0 = b
ADD Sc(Rs), R0  # R0 = b + c
MOV Sa(Rs), R1  # R1 = a
DIV R0, R1      # R1 = R1 / R0 = a / (b + c)
MOV Se(Rs), R0  # R0 = e
ADD Sf(Rs), R0  # R0 = e + f
MUL Sd(Rs), R0  # R0 = d * (e + f)
SUB R0, R1      # R1 = R1 – R0 = a / (b + c) – d * (e + f)
MOV R1, Sx(Rs)  # x = R1
```

8.6

题目：

一个 C 语言程序如下：

```
main() {
    long i;
    i = 0;
    printf("%ld\n", (++i)+(++i)+(++i));
}
```

该程序在 x86/Linux 系统上，编译后运行的结果是 7（编译器版本是 GCC: (GNU) egcs-2.91.6619990314/Linux(egcs-1.1.2 release)），而在早先的 SPARC/SunOS 系统上编译后运行的结果是 6。试分析运行结果不同的原因。

这个结果似乎和系统无关，更像是编译器相关（比如 GCC 输出是 7，Clang 输出是6）

因为这是 Undefined Behavior，不同的编译器会对这条语句作出不同的解释：

- 输出 7 是因为编译器将 i 作为操作数（先将 i 自增了两次，再算 i + i，然后再 ++i，再作加法，结果为 2 + 2 + 3）。
- 输出 6 是因为编译器将每次自增的中间结果作为加法的操作数（1 + 2 + 3）

注：
答到点上即可，即说明两种情况下对不同的数据进行加法