编译原理期末复习-按考点

第一章 引论

翻译器、编译器、解释器

- 翻译器：把一种语言变成另外一种语言（语义等价）
- 编译器：翻译器的一种
- 解释器：不产生目标代码，解释执行源程序（根据程序和输入给出输出）

解释器执行的效率比编译器生成的机器代码的执行效率低（解释器缺少了类似于编译器的优化过程）。

解释器直接执行用编程语言编写的指令。它在执行程序时，会逐条将源代码解释成机器语言并执行，因此运行速度相对较慢。

编译程序的各个阶段

- 词法分析
- 语法分析
- 语义分析
- 中间代码生成（以上是“前端”）
- 独立于机器的代码优化（以下是“后端”）
- 代码生成
- 依赖于机器的代码优化

类型检查通常出现在语义分析中。

编译器在语法分析阶段会将记号流构造成抽象语法树。

编译器会在词法分析阶段去掉源程序的注释。

编译程序各阶段的工作都可能涉及到：错误管理

编译程序的各个阶段都可能会涉及到符号表管理

e.g. 对一段程序进行编译时，将while识别为关键字的编译阶段是词法分析。

语法分析阶段，编译器会检查程序中的单词序列是否遵守特定的语法规则。

语义分析阶段，编译器检查变量和常量是否有正确的类型、函数是否正确地调用。

（往年考题）**编译器各阶段中，词法分析器和语法分析器的典型关系是以语法分析器为主导，词法分析器作为子程序被调用。**

编译器在词法分析阶段可以将来自编译器各个阶段的错误信息和源程序联系起来。

第二章 词法分析

对源程序的翻译

e.g. 如果对下列源程序进行词法分析，请问词法分析器应该返回多少个记号？

```
printf("Total=%d\n", score);
```

答：返回7个记号：`printf`, `(`, `"Total=%d\n"`, `,`, `score`, `)`, `;`.

这里把双引号内的识别为字符串，算一个记号。

正规式

“L^*”：闭包，表示零个或多个

“L^+”：正闭包，表示一个或多个

正规定义

e.g. C语言标识符的正规定义：

letter_ → A | B | ... | Z | a | b | ... | z | _

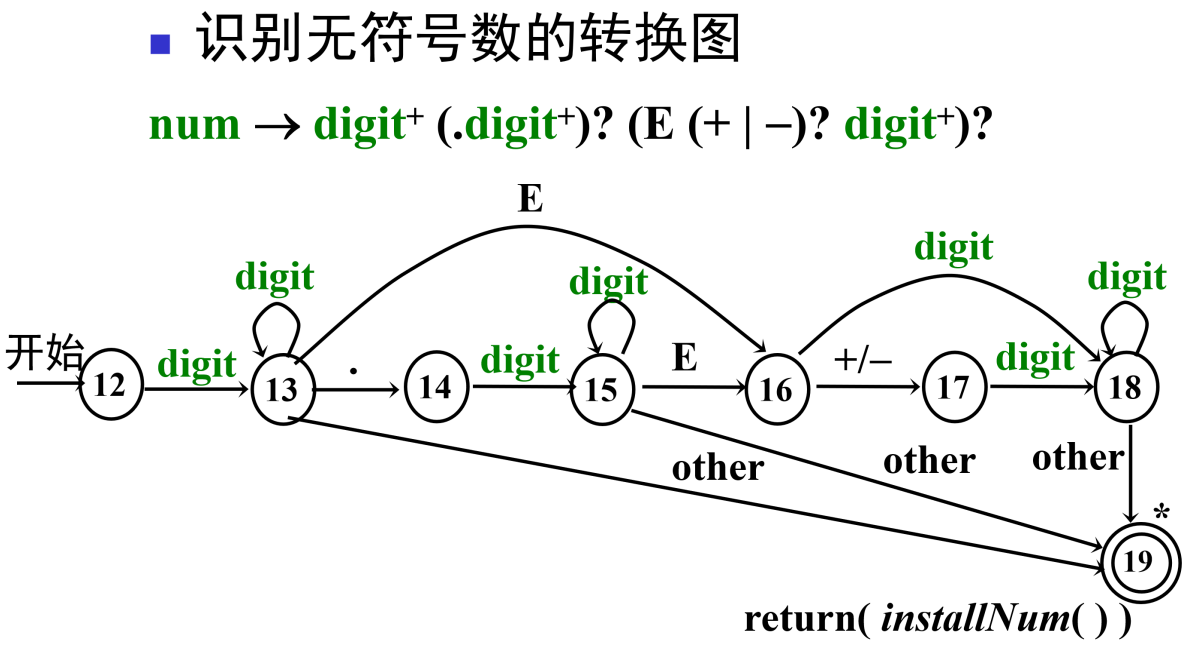
digit → 0 | 1 | ... | 9

id → **letter_** (**letter_** | **digit**)*

一些缩写：

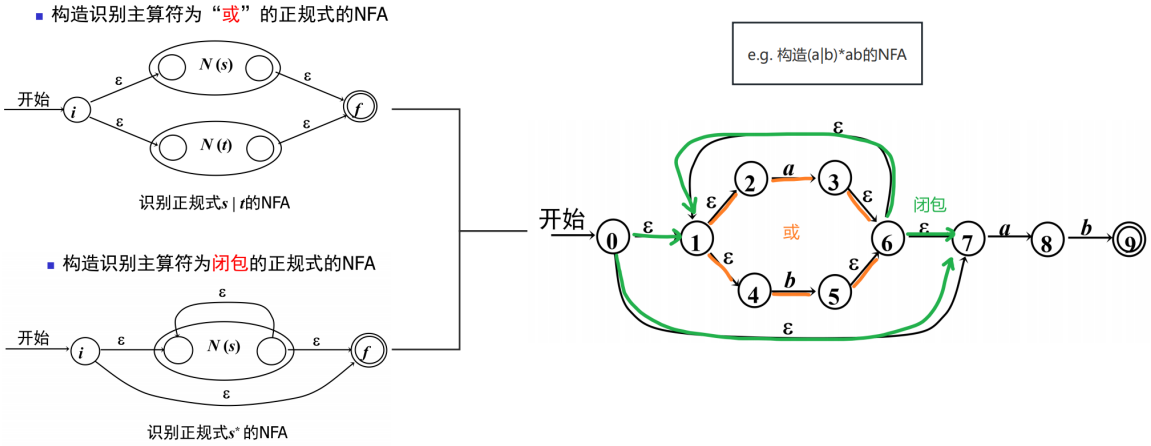
- 一个或多个实例：一元后缀算符“+”。
- **零个或一个**实例：一元后缀算符“？”。
- 字符组：[abc]表示正规式a | b | c。缩写字符组[a-z]表示正规式a | b | ... | z。

状态转换图



开始状态、状态、接受状态（俩圈）、动作。

Thompson算法（正规式→有限自动机）

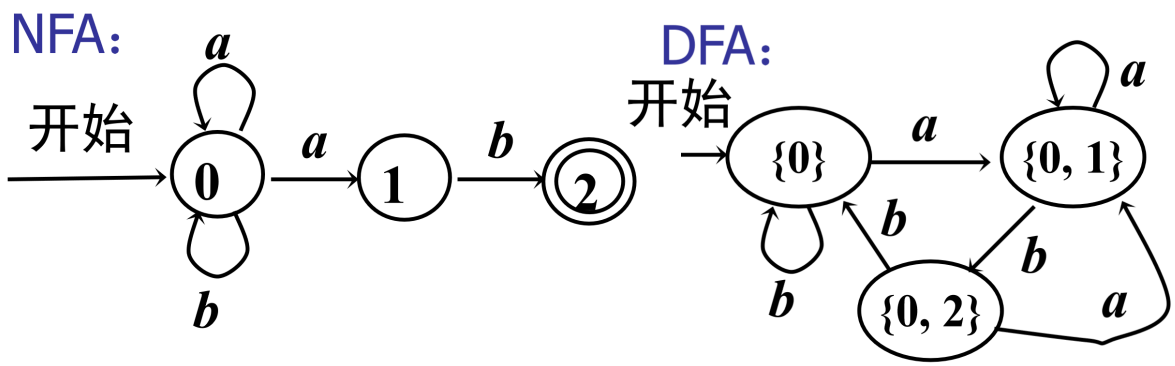


要记住“或”运算和“闭包”运算的构造形式。

NFA&DFA

- NFA：不确定的有限自动机
- DFA：确定的有限自动机
- 理解：NFA有可能会出现一个输入对应一个节点后方的多个箭头，而DFA不会出现这种情况。

子集构造法（NFA→DFA）

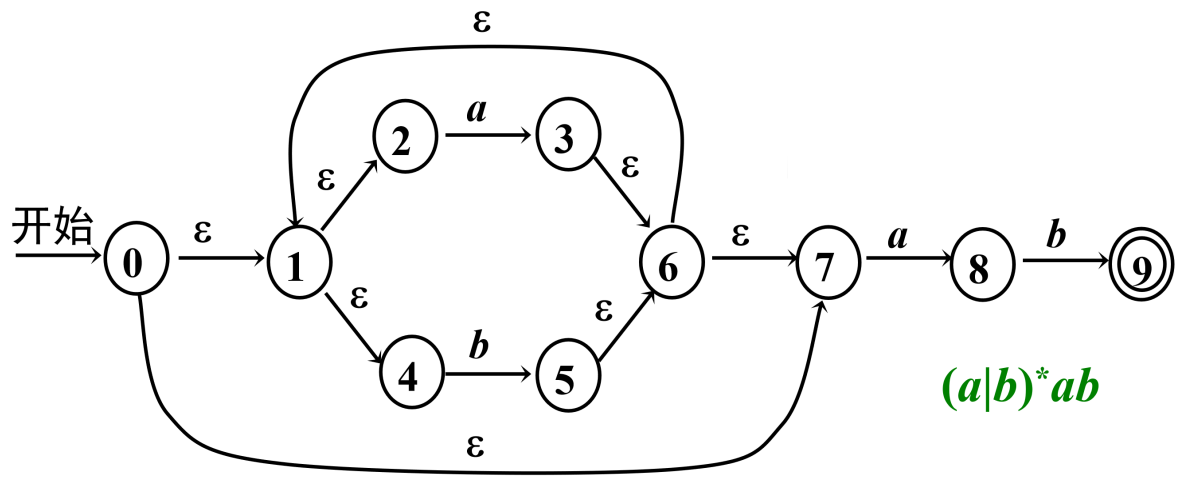


运算	描述
$\varepsilon - closure(s)$	从NFA状态s出发，只用ε转换能达到的NFA状态**集合**； $\varepsilon - closure$**均包含自身**（常用于没有ε转换的题目中）
$\varepsilon - closure(T)$	T是一个集合，包含多个状态（节点）
$move(T, a)$	从状态集合T中的状态（节点）开始，经过a转换能够到达的状态集合
☆$\varepsilon - closure(move(A, a))$	从A状态开始经过**一步a转换**，再可以经过**若干次ε转换**

子集构造法：

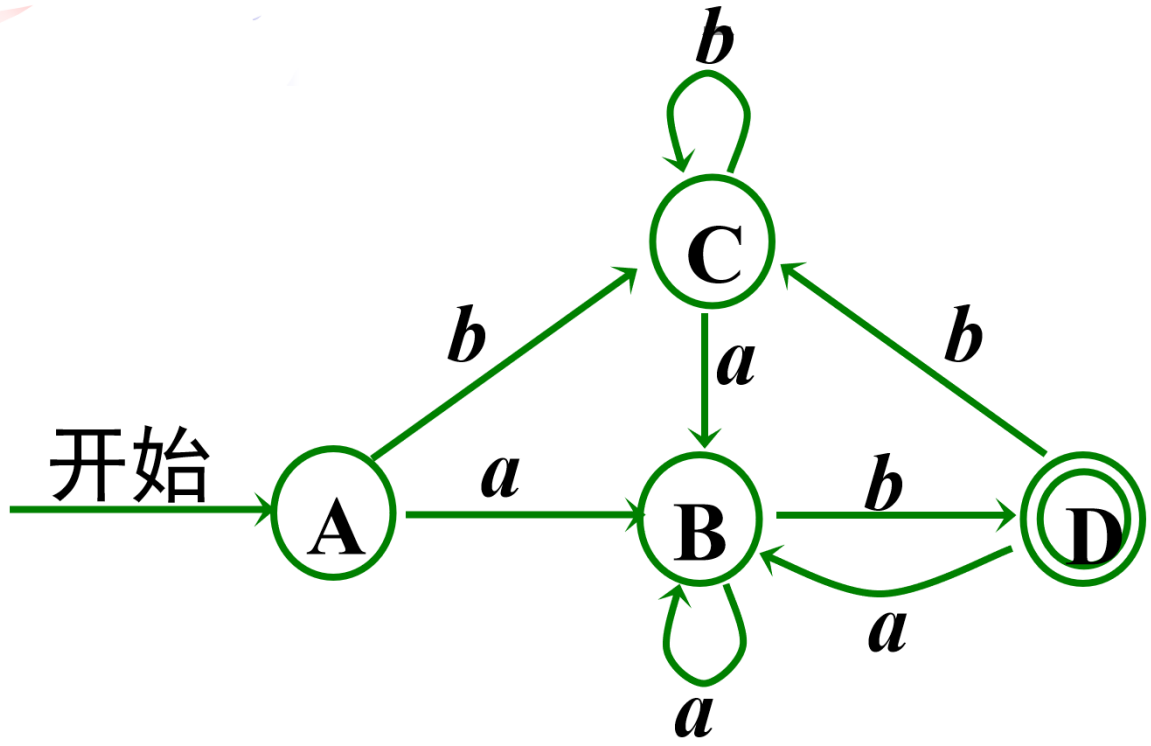
DFA的一个状态是NFA的一个**状态集合。**

算法：

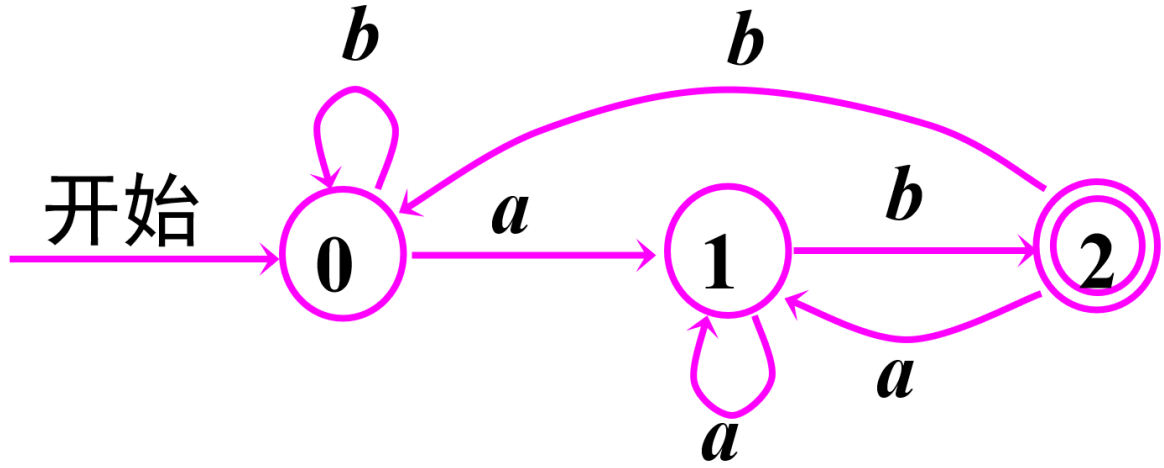


	输入符号	输入符号
	a	b
A	B	C
B	B	D
C	B	C
D	B	C

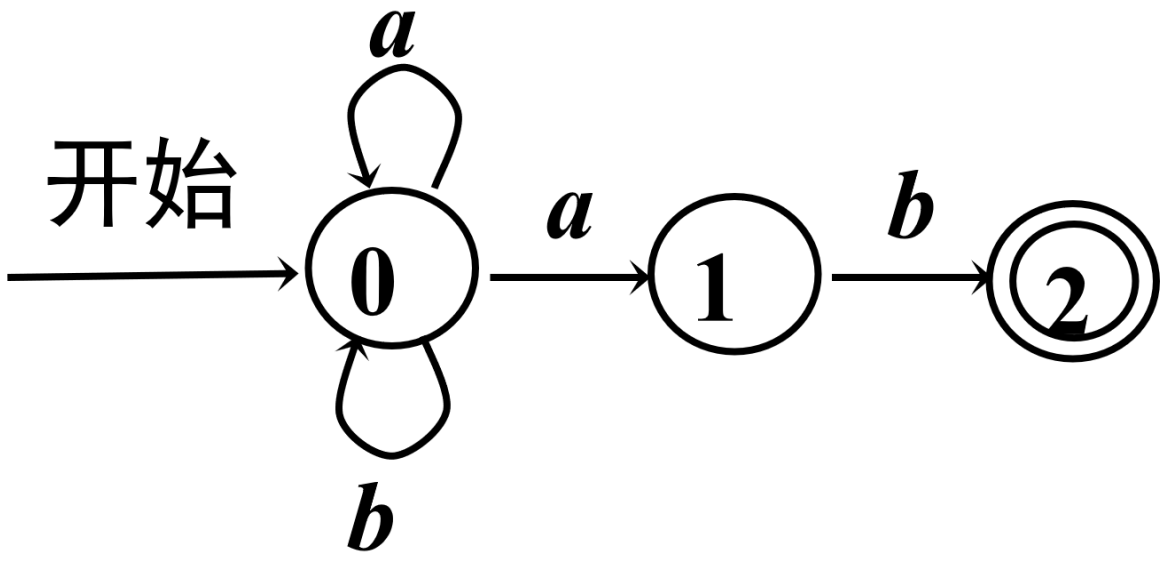
- 初始状态：$A = \varepsilon - closure(s_0) = \{0, 1, 2, 4, 7\}$
- 其他"新"状态：$B = \varepsilon - closure(move(A, a)) = \varepsilon - closure(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$,
  $C = \varepsilon - closure(move(A, b)) = \varepsilon - closure(\{5\}) = \{1, 2, 3, 4, 5, 6, 7\}$,
  $D = \varepsilon - closure(move(B, b)) = \varepsilon - closure(\{5, 9\}) = \{1, 2, 4, 5, 6, 7, 9\}$.
- 其他状态：e.g. $\varepsilon - closure(move(B, a)) = \varepsilon - closure(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$.
- 确定DFA的接受状态：在DFA中，一个状态（或更具体地说，一个状态的子集）被认为是接受状态，如果它包含NFA的一个或多个接受状态。换句话说，如果DFA的某个状态子集**包含NFA的至少一个接受状态**，那么该DFA状态就是接受状态。



子集构造法不一定得到最简单的DFA：



练习：用子集构造法将下面的NFA转换成DFA。



解：

$A = \varepsilon - closure(s_0) = \{0\}$（只包含自身）

$B = \varepsilon - closure(move(A, a)) = \varepsilon - closure(\{0, 1\}) = \{0, 1\}$

$\varepsilon - closure(move(A, b)) = \varepsilon - closure(\{0\}) = \{0\} = A$

$\varepsilon - closure(move(B, a)) = \varepsilon - closure(\{0, 1\}) = \{0, 1\} = B$

$C = \varepsilon - closure(move(B, b)) = \varepsilon - closure(\{0, 2\}) = \{0, 2\}$

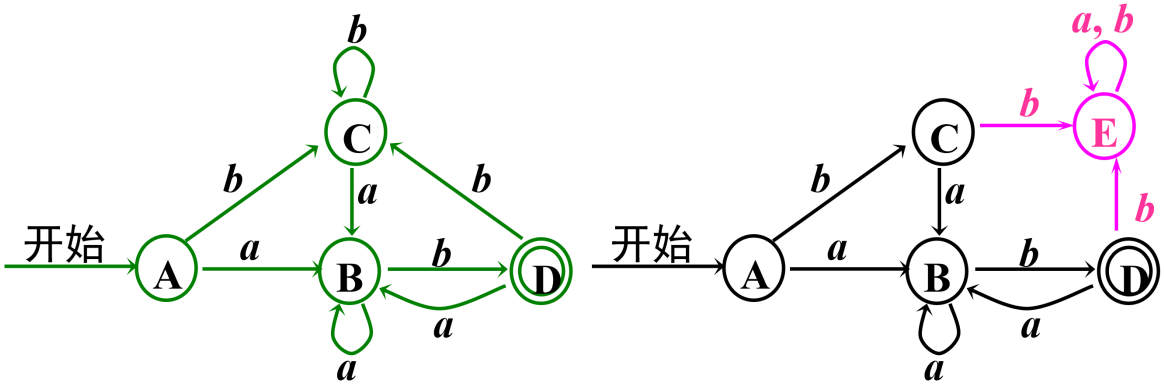
$\varepsilon - closure(move(C, a)) = \varepsilon - closure(\{0,\ 1\}) = \{0,\ 1\} = B$

$\varepsilon - closure(move(C, b)) = \varepsilon - closure(\{0\}) = \{0\} = A$

	输入符号	输入符号
	a	b
A	B	A
B	B	C
C	B	A

DFA的化简

如果转换函数并非全函数，化简前可以引入**死状态**（“dead” state）变换成全函数。

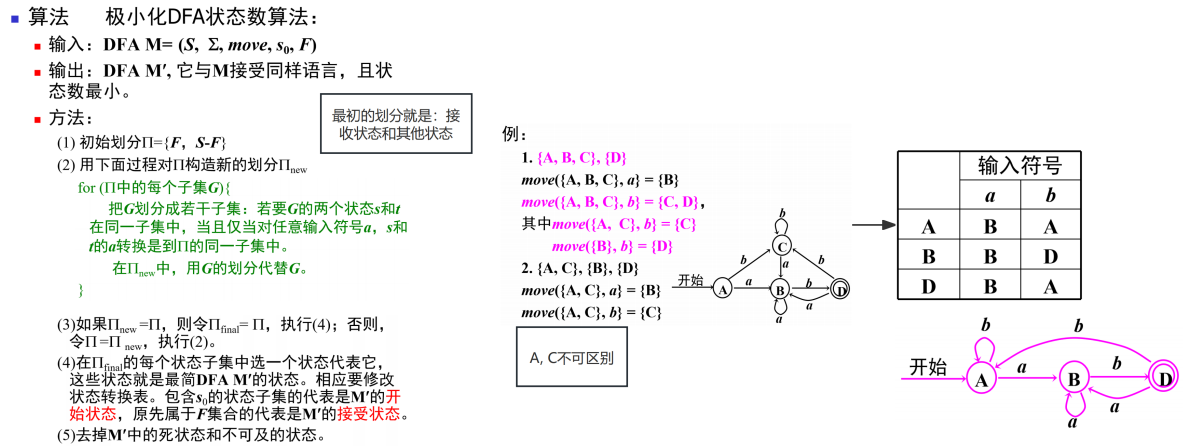


可区别状态&不可区别状态：

- 可区别状态：对该DFA，分别从s和t出发，输入w，一个最终停留在某个**接受状态**，另一个停留在某个**非接受状态。**
- 不可区别状态：找不到任何输入（串）w来区别两个状态。

极小化DFA状态数算法思路：

- 初始，划分成两个子集：接受状态子集和非接受状态子集。
- 检查每个子集，如果还可以划分，则继续划分，直到没有任何一个子集可划分为止。
- **若要G的两个状态s和t在同一子集中，当且仅当对任意输入符号a，s和t的a转换是到同一子集中。**



上述例子中，$move(\{A,B,C\},b)=\{C,D\}$，C和D可区分（一个在接受状态，一个在非接受状态），所以$\{A,B,C\}$可拆。

原本的状态转换表：

	输入符号	输入符号
	a	b
A	B	C
B	B	D
C	B	C
D	B	C

练习：将下面的DFA状态数最小化

	a	b
S0	S5	S2
S1	S6	S2
S2	S0	S4
S3	S3	S5
S4	S6	S2
S5	S3	S0
S6	S3	S1

划分结果：$\{S_0,S_1\},\{S_2\},\{S_3\},\{S_4\},\{S_5,S_6\}.$

第三章 语法分析

上下文无关文法

上下文无关文法G是四元组(V_T, V_N, S, P)，分别是：终结符集合、非终结符集合、开始符号、产生式集合。

例　$G = (\{\text{id}, +, *, -, (,)\}, \{expr, op\}, expr, P)$

其中P由下列产生式组成：

$expr \rightarrow expr\ op\ expr$

$expr \rightarrow (expr)$

$expr \rightarrow - expr$

$expr \rightarrow \text{id}$

$op \rightarrow +$

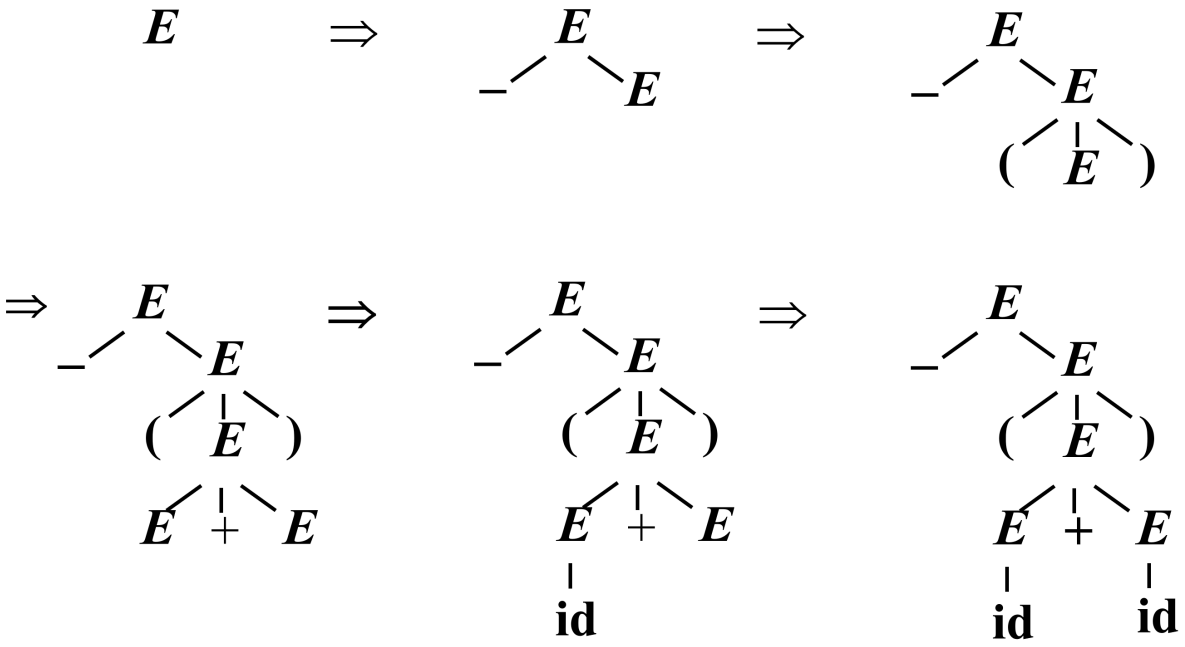
$op \rightarrow *$

为什么是上下文无关？

句型中出现的产生式的左部非终结符可以在任何时候被替换，这种替换不依赖于句型中的其他符号（上下文）

最左推导、最右推导（最右推导又称规范推导）

分析树：（这是$-(id + id)$的推导分析树）



消除二义性

- 有二义性的表达式文法：

  $E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$

- 用一种层次观点看待表达式（考虑优先级和结合性以消除二义性）

  $\underline{\mathbf{id} * \mathbf{id} * (\mathbf{id}+\mathbf{id})} + \underline{\mathbf{id} * \mathbf{id}} + \underline{\mathbf{id}}$

  $\underline{\mathbf{id}} * \underline{\mathbf{id}} * \underline{(\mathbf{id}+\mathbf{id})}$

- 构造非二义的文法

优先级 ↑

$expr \rightarrow expr + term \mid term$

$term \rightarrow term * factor \mid factor$

$factor \rightarrow \mathbf{id} \mid (expr)$

+和*左结合，
*优先级高于+

- 下面的二义文法描述命题演算公式的语法，为它写一个等价的非二义文法

$$S \rightarrow S \text{ and } S \mid S \text{ or } S \mid \text{not } S \mid p \mid q \mid (S)$$

非二义文法的产生式如下：

$$E \rightarrow E \text{ or } T \mid T$$
$$T \rightarrow T \text{ and } F \mid F$$
$$F \rightarrow \text{not } F \mid G$$
$$G \rightarrow (\ E) \mid p \mid q$$

e.g. if-then-else

正确语义：每个else和它左边最接近的还没配对的then相配。

想法：
出现在配对的then和else之间的语句必须是配对的

- 无二义的文法

$stmt \rightarrow matched\_stmt \mid unmatched\_stmt$

$matched\_stmt \rightarrow \text{if } expr \text{ then } matched\_stmt \text{ else } matched\_stmt \mid \text{other}$

$unmatched\_stmt \rightarrow \text{if } expr \text{ then } stmt \mid \text{if } expr \text{ then } matched\_stmt \text{ else } unmatched\_stmt$

消除左递归

左递归：给定文法G，若有$A \to \alpha \to^* A\gamma(\gamma \neq \varepsilon)$，则称非终结符A是左递归的。（→*是"经过任意步推导"的意思）

由形式为$A \to A\alpha$的产生式引起的左递归称为**直接左递归。**

☆【背】$A \to A\alpha_1|A\alpha_2|\beta_1|\beta_2$消除左递归：

$A \to \beta_1 A'|\beta_2 A'$

$A' \to \alpha_1 A'|\alpha_2 A'|\varepsilon$

e.g. 消除下列算术表达文法的左递归：

$E \to E+T|T$

$T \to T*F|F$

$F \to (E)|id$

解：消除E和T的直接左递归：

$E \to TE'$

$E' \to +TE'|\varepsilon$

$T \to FT'$

$T' \to *FT'|\varepsilon$

$F \to (E)|id$

对于非直接左递归的：1.变成直接左递归；2.消除直接左递归

e.g. 文法：

$S \to Aa|b$

$A \to Sd|\varepsilon$

解：1.$S \to Aa|b\ A \to Aad|bd|\varepsilon$

2.$S \to Aa|b\ A \to bdA'|A'\ A' \to adA'|\varepsilon$

无用符号和无用产生式：

- 无用符号：X至少出现在一个句子的推导中，则说X是有用的，否则称X为无用符号。（推导：把产生式看成重写规则，把符号串中的非终结符用其产生式右部的串来代替。）
- 无用产生式：如果一个产生式左部或右部含有无用符号，则此产生式称为无用产生式。

- $R \rightarrow Sa \mid a$
- $Q \rightarrow Sab \mid ab \mid b$
- $S \rightarrow abcS' \mid bcS' \mid cS'$
- $S' \rightarrow abcS' \mid \varepsilon$

- 消除无用符号和产生式：
  - $S \rightarrow abcS' \mid bcS' \mid cS'$
  - $S' \rightarrow abcS' \mid \varepsilon$

提左因子

$A \rightarrow \alpha\beta_1 | \alpha\beta_2$

左因子：α

提左因子：

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 | \beta_2$

形式语言的Chomsky分类

【背】

- 0型文法：短语文法
- 1型文法：上下文有关文法
- 2型文法：上下文无关文法
- 3型文法：正规文法（正规式）

0型文法描述能力最强，3型文法描述能力最弱。

FIRST和FOLLOW集合

FIRST-第一个 FOLLOW-跟着的

本部分举例用的文法如下：

$E \rightarrow TE'$

$E' \to +TE'|\varepsilon$

$T \to FT'$

$T' \to *FT'|\varepsilon$

$F \to (E)|id$

FIRST(X)计算:

- X是终结符→FIRST(X)={X}（终结符：类似于id这样的）
- X→ε是产生式→ε∈FIRST(X)
- $X \to Y_1Y_2 \dots Y_k$是产生式，则对某个i，若$\varepsilon \in FIRST(Y_1) \dots FIRST(Y_{i-1})$，而$a \in FIRST(Y_i)$，则$a \in FIRST(X)$。若$\varepsilon \in FIRST(Y_1) \dots FIRST(Y_k)$，则$\varepsilon \in FIRST(X)$。

对上述第三条规则的理解：前面若干个“空”，就跳过，到没有空的那儿开始看。

求上述例子的FIRST集合：

FIRST(E)=FIRST(TE')=FIRST(T)=FIRST(FT')=FIRST(F)={(, id}.

FIRST(E')=FIRST(+TE')∪FIRST(ε)={+, ε}.

FIRST(T')=FIRST(*FT')∪FIRST(ε)={*, ε}.

FOLLOW(A)计算：

- A是开始符号→$∈FOLLOW(A)
- 若有产生式$B \to \alpha A\beta$，则FIRST(β)中**除ε以外**的一切符号都属于FOLLOW(A).（β的开头都跟在A的后面）
- ✩若有产生式$B \to \alpha A$，或产生式$B \to \alpha A\beta$而$\varepsilon \in FIRST(\beta)$，则$FOLLOW(B) \subseteq FOLLOW(A)$，即FOLLOW(B)中的一切符号都要放入FOLLOW(A)中。（理解：A后面没东西了→B后面的就是A后面的）（✩这里要注意$B \to \alpha A$中B和A的顺序问题，B在左，A在右）

对于上述的第二条和第三条规则，在求解的时候，应该着重看产生式的右部。

FOLLOW集合里不会出现ε。

求上述例子的FOLLOW集合：

E是开始符号→$∈FOLLOW(E)

FOLLOW(E)={$,)}

FOLLOW(E')=FOLLOW(E)={$,)}

FOLLOW(T)=(FIRST(E')-{ε})∪FOLLOW(E')={+, $,)}（因为$\varepsilon \in FIRST(E')$，所以要并上第二部分的“FOLLOW(E')”） **（注意第二条规则里的“除ε以外”）**

FOLLOW(T')=FOLLOW(T)={+, $,)}

FOLLOW(F)=(FIRST(T')-{ε})∪FOLLOW(T)∪FOLLOW(T')={*, +, $,)}

LL(1)文法

含义：从左到右分析，最左推导，每步向前搜索一个输入符号。

定义：

任何两个产生式$A \to \alpha|\beta$都满足下列条件： **(这里的"两个产生式"是指$A \to \alpha$和$A \to \beta$)**

- FIRST(α)∩FIRST(β)=Φ
- 若β经过若干次推导可得ε，那么FIRST(α)∩FOLLOW(A)=Φ

理解：两个产生式的FIRST集合不相交，如果β能推到空，就用FOLLOW(A)代替FIRST(β)。

LL(1)文法的性质：

- 没有公共左因子（显然）
- 不是二义的
- 不含左递归

不符合LL(1)文法定义的文法举例：

$S \to AB$

$A \to ab|\varepsilon$

$B \to aC$

$C \to \dots$

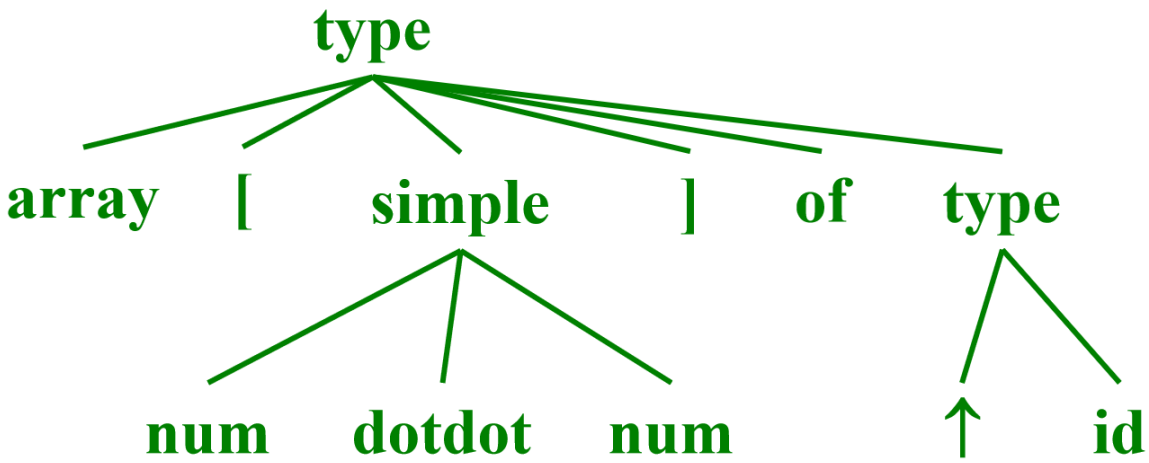
上述文法中，$a \in FIRST(ab) \cap FOLLOW(A).$

递归下降的预测分析

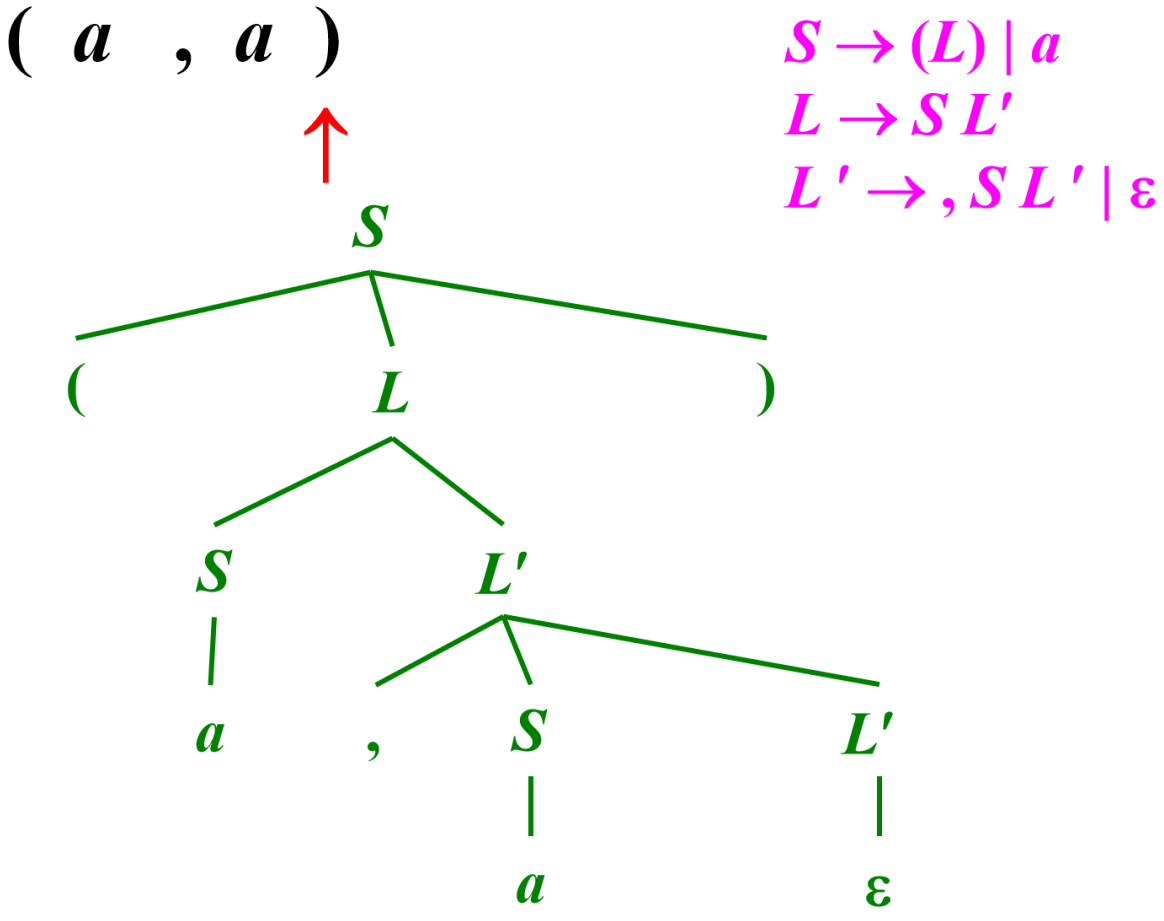
属于**自上而下分析。**

例：Pascal语言的类型子集（该语言是LL(1)的）

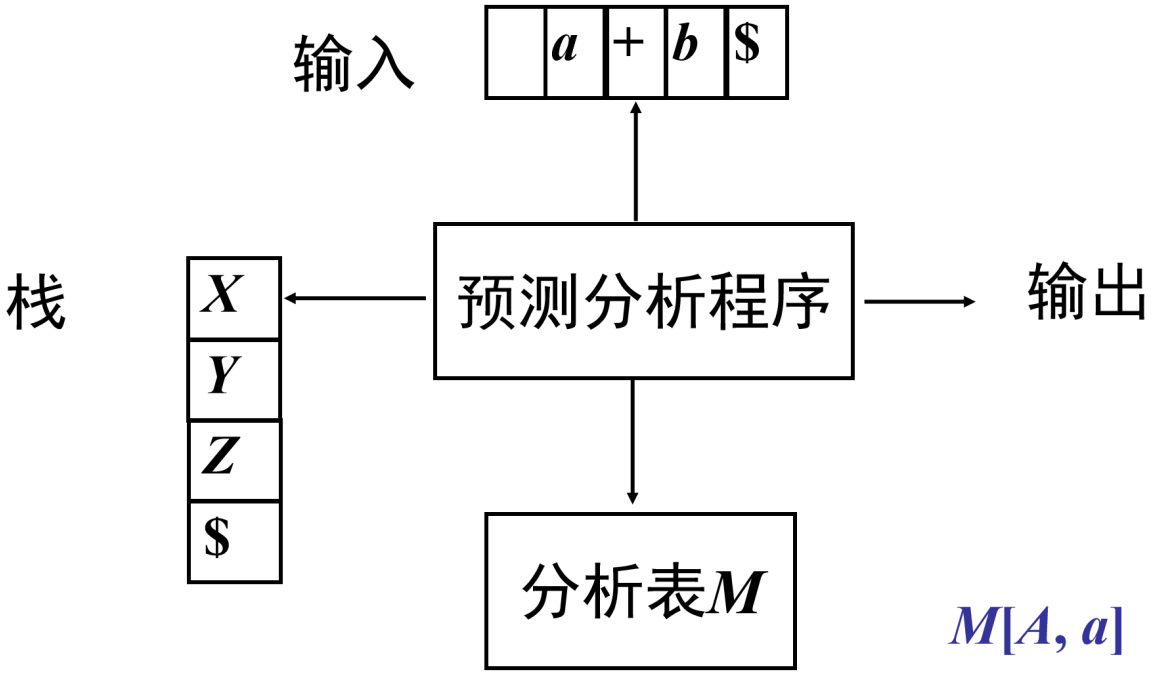
type → ***simple***
| ↑ **id**
| **array** [***simple***] **of** ***type***
simple → **integer**
| **char**
| **num dotdot num**



递归下降预测分析举例：



非递归的预测分析（LL(1)分析）



栈	输入	动作
$...	...$	...

- 栈顶为$且输入只剩$时，分析成功。
- 栈顶与输入的头部相等且不为$时，弹出栈顶，输入指针后移。
- 栈顶是终结符但不是输入头部的那个终结符，则报错。→调用错误恢复例程
- 栈顶是非终结符（一般是那些大写字母）时，访问预测分析表M[X, a]。如果表里有产生式，则用产生式右部代替栈顶（**逆序压栈**）。“动作”栏中打印“输出【产生式】”。如果M[X, a]是空白单元格，则调用错误恢复例程。

给定预测分析表，写接受输入某某某的动作

预 测 分 析 表 M

非终结符	输入符号					
	id	+	*	(	)	$
E	$E\to TE'$			$E\to TE'$		
E'		$E'\to +TE'$			$E'\to\varepsilon$	$E'\to\varepsilon$
T	$T\to FT'$			$T\to FT'$		
T'		$T'\to\varepsilon$	$T'\to *FT'$		$T'\to\varepsilon$	$T'\to\varepsilon$
F	$F\to$id			$F\to(E)$		

接受输入**id*id+id**的动作：

栈	输　入	输　出
$E	id * id + id$	（访问M[E, id]）
$E ′T	id * id + id$	$E \rightarrow TE'$
$E ′T ′F	id * id + id$	$T \rightarrow FT'$
$E ′T ′ id	id * id + id$	$F \rightarrow$ id
$E ′T ′	* id + id$	
$E ′T ′F *	* id + id$	$T' \rightarrow *FT'$
$E ′T ′F	id + id$	
$E ′T ′ id	id + id$	$F \rightarrow$ id

栈	输　入	输　出
$E ′T ′ id	id + id$	$F \rightarrow$ id
$E ′T ′	+ id$	
$E ′	+ id$	$T' \rightarrow \varepsilon$
$E ′T+	+ id$	$E' \rightarrow +TE'$
$E ′T	id$	
$E ′T ′F	id$	$T \rightarrow FT'$
$E ′T ′id	id$	$F \rightarrow$ id
$E ′T ′	$	

栈	输　入	输　出
$E ′T ′	$	
$E ′	$	$T' \rightarrow \varepsilon$
$	$	$E' \rightarrow \varepsilon$

在非递归的预测分析中，已经扫描过的符号加上栈中的文法符号（从栈顶到栈底），构成最左推导的句型。

（句型：一个文法从开始符号能推出的所有串；句子：只含终结符的句型）

构造预测分析表

步骤：对**文法的每个产生式**A→α，执行下述操作：

- 对FIRST(α)中的每个终结符a，把【产生式】A→α加入M[A, a]。 **（注意区分里面的α和a）**
- 如果ε在FIRST(α)中，对FOLLOW(A)的每个终结符b（包括$），把A→α加入M[A, b]

e.g. 对于文法：

$E \to TE'$

$E' \to +TE'|\varepsilon$

$T \to FT'$

$T' \to *FT'|\varepsilon$

$F \to (E)|id$

FIRST(*E*) = FIRST(*T*) = FIRST(*F*) = { (, id }
FIRST(*E* ′) = {+, ε}
FIRST(*T* ′) = {*, ε}
FOLLOW(*E*) = FOLLOW(*E* ′) = {), $}
FOLLOW(*T*) = FOLLOW (*T* ′) = { +,), $}
FOLLOW(*F*) = {+, *,), $}

构造预测分析表：

非终结符	输入符号					
	id	**+**	*****	**(**	**)**	**$**
E	***E*→*TE′***			***E*→*TE′***		
E′		***E′*→+*TE′***			***E′*→ε**	***E′*→ε**
T	***T*→*FT′***			***T*→*FT′***		
T′		***T′*→ε**	***T′*→**FT′***		***T′*→ε**	***T′*→ε**
F	***F*→id**			***F*→(*E*)**		

紧急方式的错误恢复

发现错误时，分析器每次抛弃一个输入记号，直到输入记号属于某个指定的同步记号集合为止。

同步记号选择的一些提示：

- 把FOLLOW(A)的所有终结符放入非终结符A的同步记号集合
- 把高层结构的开始符号加到低层结构的同步记号集合中
- 把FIRST(A)的终结符加入A的同步记号集
- 如果非终结符可以产生空串，若出错时栈顶是这样的非终结符，则可以使用产生空串的产生式

自下而上分析（移进-归约分析）

自下而上分析

编译器常用的移进-归约分析：**LR分析**（L：从左向右扫描；R：构造最右推导的逆）

例 $S \to aABe$

$A \to Abc \mid b$

$B \to d$

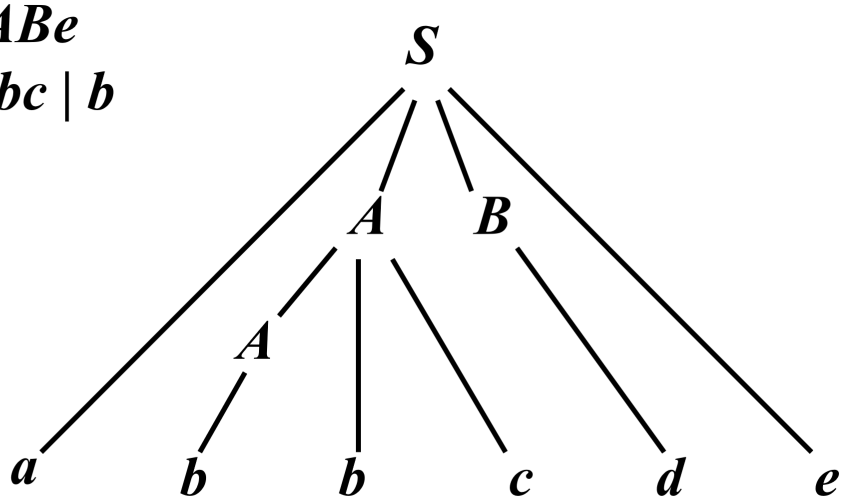
$abbcde$

$aAbcde$

$aAde$

$aABe$

S



$$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$$

注意：推导箭头下面的rm在最右推导时必须要写。

最右（左）推导的逆过程是最左（右）归约；最左归约也称**规范归约**。

句柄：如果一个句型由最右推导得到，则句型的句柄是和某产生式右部匹配的子串。

$S \to aABe$

$A \to Abc \mid b$

$B \to d$

$$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$$

上述例子中，绿色标出的就是“句柄”。

句柄的右边仅含终结符。

移进-归约分析器

分析器基本动作：

- 移进：把下一个输入符号移进栈
- 归约：把栈顶的句柄归约为**非终结符**
- 接受：宣告分析成功
- 报错：发现错误，调用错误恢复例程

用栈实现移进-归约分析：

$E \to E+E|E*E|(E)|id$

栈	输　入	动　作
$	$\mathbf{id_1 * id_2 + id_3\$}$	移进
$ $\mathbf{id_1}$	$\mathbf{* id_2 + id_3\$}$	按$E \to$ id归约
$E	$\mathbf{* id_2 + id_3\$}$	移进
$E*	$\mathbf{id_2 + id_3\$}$	移进
$E*$\mathbf{id_2}$	$\mathbf{+ id_3\$}$	按$E \to$ id归约
$E*E	$\mathbf{+ id_3\$}$	移进
$E*E+	$\mathbf{id_3\$}$	移进
$E*E+$\mathbf{id_3}$	$	按$E \to$ id归约
$E*E+E	$	按$E \to E+E$归约
$E*E	$	按$E \to E*E$归约
$E	$	接受

“归约”只在栈中进行。

移进-归约冲突&归约-归约冲突

移进-归约冲突举例：

stmt → **if** expr **then** stmt | **if** expr **then** stmt **else** stmt | **other**

栈	输入
… **if** expr **then** stmt	**else** … $

无法确定此时应当移进还是归约，故产生移进-归约冲突。（无法确定 **if** expr **then** stmt 是否为句柄）

归约-归约冲突举例：

$$stmt \rightarrow \mathbf{id}\ (parameter\_list) \mid expr = expr$$
$$parameter\_list \rightarrow parameter\_list, parameter \mid parameter$$
$$parameter \rightarrow \mathbf{id}$$
$$expr \rightarrow \mathbf{id}\ (expr\_list) \mid \mathbf{id}$$
$$expr\_list \rightarrow expr\_list, expr \mid expr$$

由$p(i, j)$开始的语句，词法分析后变为记号流**id(id,id)**

栈	输入
… id (id	, id)…

不知道应该归约成过程调用还是数组，所以产生了归约-归约冲突。

☆有关YACC处理移进归约冲突和归约归约冲突：

- YACC在处理语法冲突的时候，默认情况下，对于移进归约冲突，优先于**移进。**
- YACC在处理语法冲突的时候，默认情况下，对于归约归约冲突，优先于**先出现的**产生式。

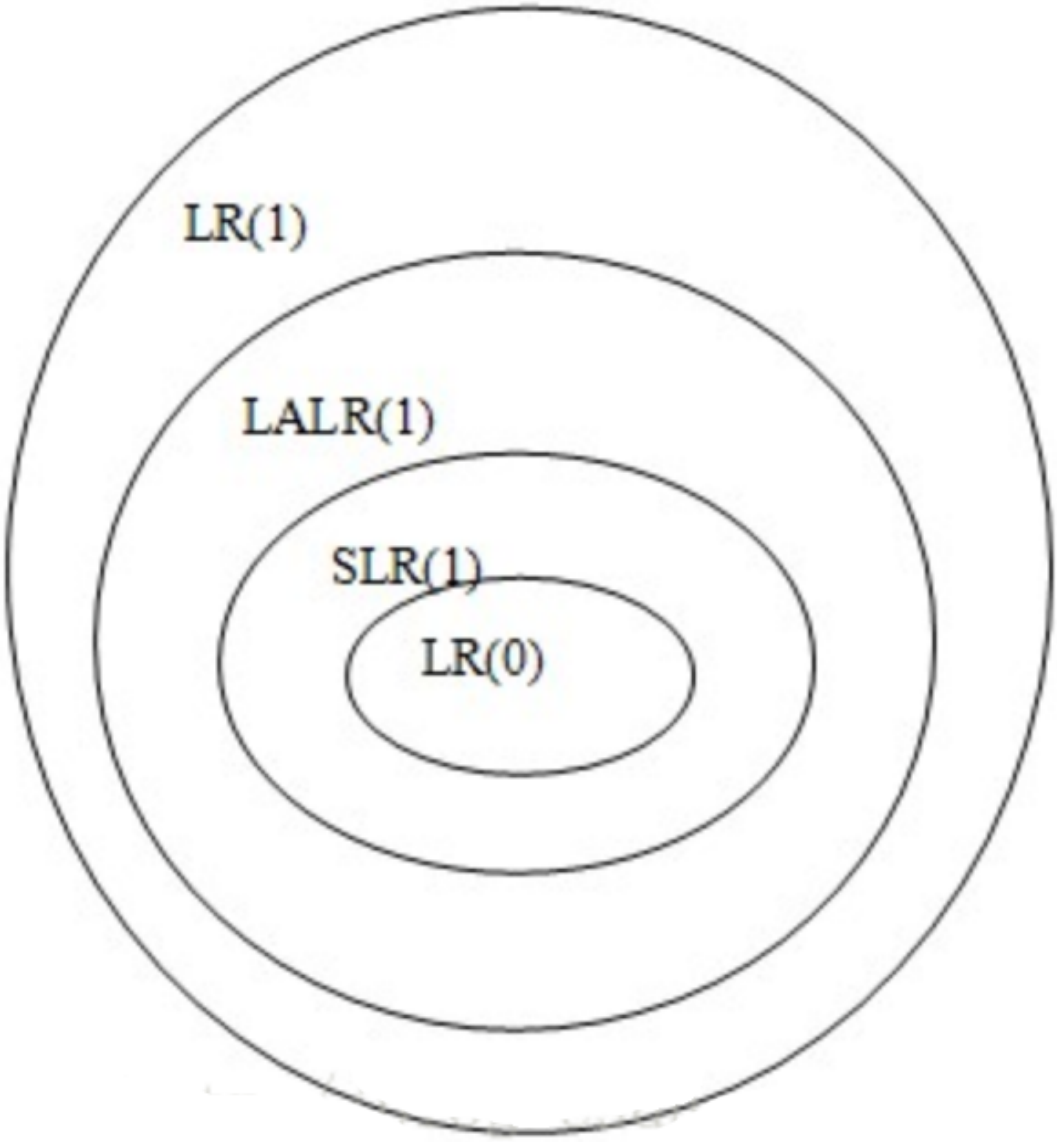
（往年考过）**合并同心项目集**有可能产生新的**归约归约冲突。**

Yacc解决分析动作冲突时，如果定义了终结符的优先级和结合性，则如果产生移进归约冲突时：

- 如果归约的产生式的优先级和移进的终结符的优先级相同，且产生式**左结合**，则进行**归约。**
- Yacc中产生式的优先级与产生式中处在最右端的终结符的优先级相同。
- 如果归约的产生式的优先级**高于**移进的终结符的优先级，则进行归约。

LR分析器

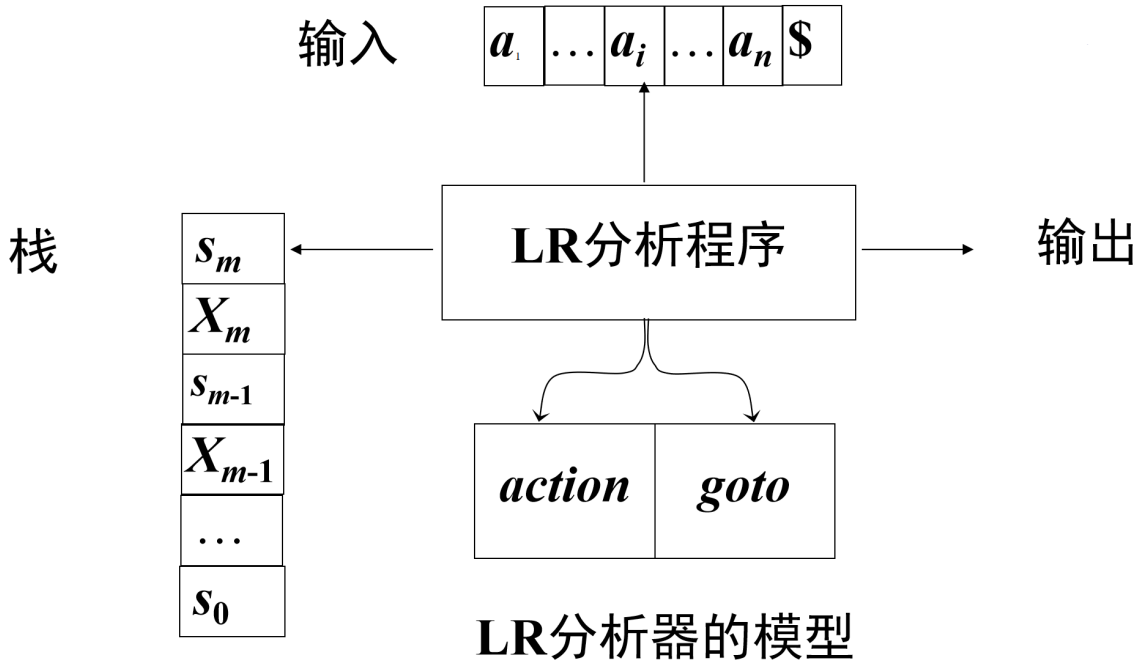
☆包含关系：



- 简单的LR方法（简称SLR）
- 规范的LR方法
- 向前看的LR方法（简称LALR）

所有能够用LL分析法（预测分析法）分析的文法，都能够用LR分析法分析。

它们的差别在于构造action-goto表的方法不同



LR分析器的模型

输入缓冲区；状态栈

LR分析算法：根据当前的输入和当前的栈顶的状态来查action表（移进/归约/出错/接受）：移进-把当前输入放入栈中，再把si中的i（状态）放进去；出错/接受-结束分析或错误处理；归约-按照（ri中的）第i条产生式归约，把产生式右部的所有符号以及它们上面的状态都出栈，把产生式左部的非终结符移到栈里，根据这个非终结符以及它下面的状态查转移表，把状态值压入栈中。

例 (1) $E \to E + T$ | (2) $E \to T$
(3) $T \to T * F$ | (4) $T \to F$
(5) $F \to (E)$ | (6) $F \to \mathrm{id}$

状态	动 作						转 移		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3

状态	动　作						转　移		
	id	+	*	(	)	$	*E*	*T*	*F*
5		*r*6	*r*6		*r*6	*r*6			
6	*s*5			*s*4				9	3
7	*s*5			*s*4					10
8		*s*6			*s*11				
9		*r*1	*s*7		*r*1	*r*1			
10		*r*3	*r*3		*r*3	*r*3			
11		*r*5	*r*5		*r*5	*r*5			

栈	输　入	动　作
0	id * id + id $	移进
0 id 5	* id + id $	按*F* → id归约
0 *F* 3	* id + id $	按*T* → *F*归约
0 *T* 2	* id + id $	移进
0 *T* 2 * 7	id + id $	移进
0 *T* 2 * 7 id 5	+ id $	按*F* → id归约
0 *T* 2 * 7 *F* 10	+ id $	按*T* → *T**F*归约
0 *T* 2	+ id $	按*E* → *T*归约
0 *E* 1	+ id $	移进

栈	输　入	动　作
0 *E* 1	+ id $	移进
0 *E* 1 + 6	id $	移进
0 *E* 1 + 6 id 5	$	按*F* → id归约
0 *E* 1 + 6 *F* 3	$	按*T* → *F*归约
0 *E* 1 + 6 *T* 9	$	按*E* → *E*+*T*归约
0 *E* 1	$	接受

活前缀：右句型的前缀，该前缀不超过最右句柄的右端。

$$A \to \beta$$

$$S \Rightarrow^*_{rm} \gamma A w \Rightarrow_{rm} \gamma \beta w$$

$\gamma\beta$的任何前缀（包括ε和$\gamma\beta$本身）都是一个活前缀

构造SLR分析表☆

构造识别活前缀的DFA

文法的**LR(0)**项目：在右部的某个地方加点的产生式：

$$expr \to expr + \bullet\ term$$

$$term \to \bullet\ term * factor$$

$$term \to term \bullet * factor$$

加点的目的是用来表示分析过程中的状态。

点前面的相当于已经有了的，点后面相当于期望得到的。

核心项目和非核心项目：

- 核心项目：包括初始项目$S' \to S$和所有那些点不在左端的项目。
- 非核心项目：除了$S' \to S$以外，所有点在左端的项目。

闭包函数closure：

如果I是文法G的一个项目集，那么$closure(I)$是用下面两条规则从I构造的项目集：

- 初始时，I的每个项目都加入$closure(I)$。
- 如果$A \to \alpha \cdot B\beta$在$closure(I)$中，且$B \to \gamma$是**产生式**，那么如果项目$B \to \cdot\gamma$还不在$closure(I)$中的话，则把它加入。运用这条规则，直到没有更多的项目可以加入$closure(I)$为止。**【点后面是非终结符的需要进行此操作】**

e.g.【构造SLR分析表】考虑下述文法：

$E \to E + T|T$

$T \to T * F|F$

$F \to (E)|id$

拓广文法：添加$E' \to E$

I_0：在上面的所有产生式箭头右边的头部加点：

$E' \to \cdot E$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

$I_1 = goto(I_0, E), I_2 = goto(I_0, T), I_3 = goto(I_0, F), I_4 = goto(I_0, (), \ldots \ldots$

特别需要注意的是I_4的求法：

先有$F \to (\cdot E)$

☆☆☆【求闭包】点后面是E，要**【根据产生式】**把E产生什么什么的放进来：

$E \to \cdot E + T$

$E \to \cdot T$

点后面是T，要把T产生什么什么的放进来：

$T \to \cdot T * F$

$T \to \cdot F$

点后面是F，要把F产生什么什么的放进来：

$F \to \cdot (E)$

$F \to \cdot id$

所以I_4中有7条产生式。

对于$I_6 = goto(I_1, +)$：（I_1：$E' \to \cdot E, E \to E \cdot + T$）

先有：$E \to E + \cdot T$

因为**产生式**中有T产生什么什么的，所以要补充：

$T \to \cdot T * F$

$T \to \cdot F$

点后面是F，要把F产生什么什么的放进来：

$F \to \cdot (E)$

$F \to \cdot id$

所以I_6有5条产生式。

参考过程：（较为潦草，但条理是清晰的）

$E' \to E$

$E \to E+T$

$E \to T$

$T \to T*F$

$T \to F$

$F \to (E)$

$F \to id$

$I_0: E' \to \cdot E$

$E \to \cdot E+T$

$E \to \cdot T$

$T \to \cdot T*F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

$I_1 = goto(I_0, E)$

$E' \to E \cdot$

$E \to E \cdot +T$

$I_2 = goto(I_0, T)$

$E \to T \cdot$

$T \to T \cdot *F$

$I_3 = goto(I_0, F)$

$T \to F \cdot$

$I_4 = goto(I_0, ()$

$F \to (\cdot E)$

$E \to \cdot E+T$

$E \to \cdot T$

$T \to \cdot T*F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

$I_5 = goto(I_0, id)$

$F \to id \cdot$

$I_6 = goto(I_1, +)$

$E \to E+\cdot T$

$T \to \cdot T*F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

$I_7 = goto(I_2, *)$

$T \to T*\cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

$I_8 = goto(I_4, E)$

$F \to (E \cdot)$

$E \to E \cdot +T$

$I_9 = goto(I_6, T)$

$E \to E+T \cdot$

$T \to T \cdot *F$

$goto(I_6, F)$

$= I_3$

$goto(I_6, ()$

$= I_4$

$goto(I_7, F)$

$T \to T*F \cdot$

$goto(I_8,))$

$F \to (E) \cdot$

$goto(I_8, +)$

$E \to E+\cdot T$

$T \to \cdots$

$T \to \cdots$

$= I_6$

$goto(I_9, *)$

$T \to T*\cdot F$

$I_9 = goto(I_4, T)$

$E \to T \cdot$

$T \to T \cdot *F$

$= I_2$

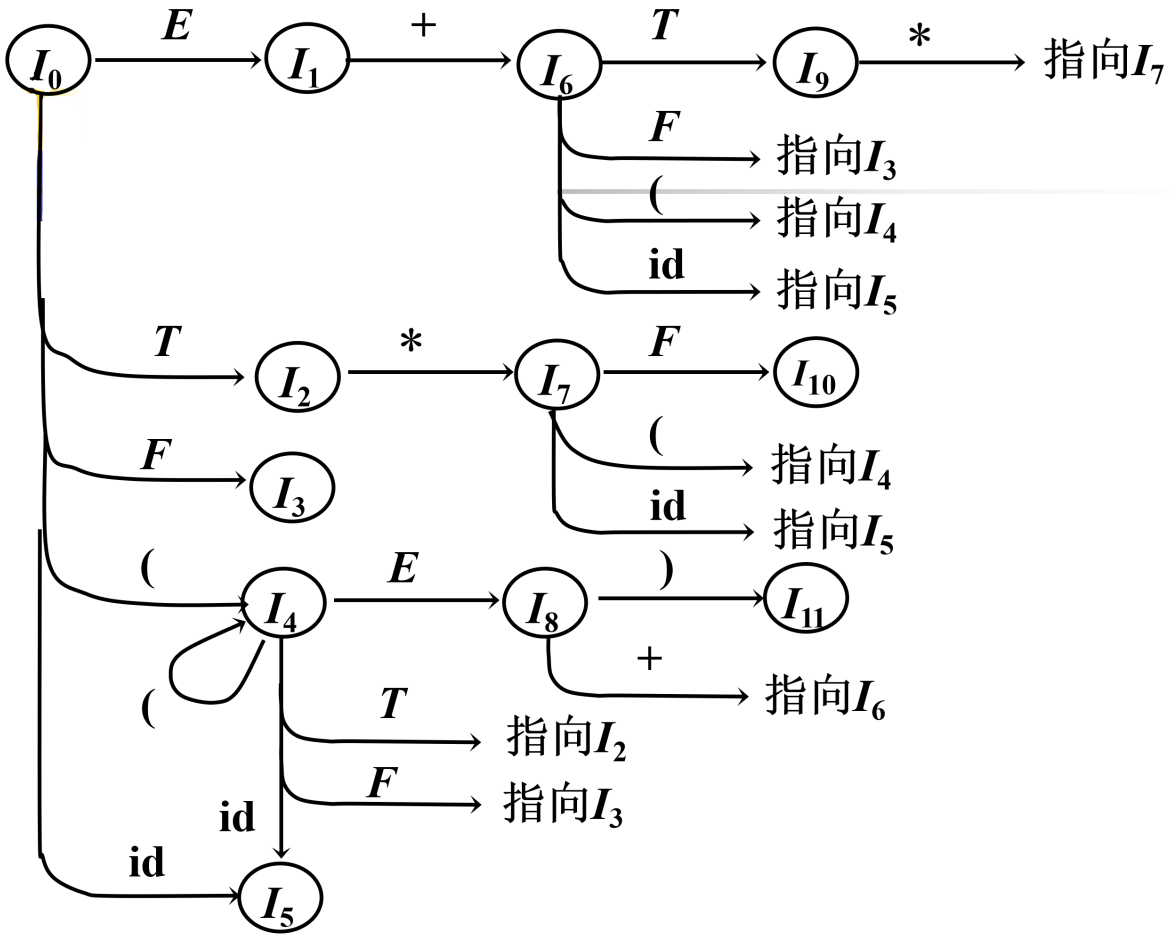
$goto(I_4, F)$

$T \to F \cdot$

$= I_3$

$goto(I_4, ()$

$= I_4$



上图是接受活前缀的**DFA**的转换图

根据LR(0)项目集规范族和识别活前缀的DFA构造action-goto表

项目集规范族：可理解为上面例子里求的那些I_i。

☆☆☆项目集中的项目分四类：（前面的序号是做题时的推荐顺序）

- ①接受项目**（开始符号'→开始符号·）**：$[S' \to S\cdot]\ action[i,\$] = acc$
- ③移进项目**（点后为终结符）**：$[A \to \alpha \cdot a\beta]\ action[i,a] = sj$
- ④归约项目**（点后为空）**：$[A \to \alpha\cdot]\ action[i,b] = rj, b \in FOLLOW(A)$☆ **（这里的"j"表示：按哪条产生式归约）**
- ②待归约项目**（点后为非终结符，或者说goto(I..., E)中的E为非终结符）**：$[A \to \alpha \cdot B\beta]$ $goto[i,B] = j$（仅根据这一条规则写转移表）

构造action-goto表时，要分析每个状态中的每一条产生式。

状态	动作	动作	动作	动作	动作	动作	转移	转移	转移
	id	+	*	(	)	$	E	T	F
0	A		B				C	D	E
1		F				G			
2		H	I		J	K			
3									
...									

状态	动　作						转　移		
	id	+	*	(	)	$	*E*	*T*	*F*
0	*s*5			*s*4			1	2	3
1		*s*6				*acc*			
2		*r*2	*s*7		*r*2	*r*2			
3		*r*4	*r*4		*r*4	*r*4			
4	*s*5			*s*4			8	2	3

状态	动　作						转　移		
	id	+	*	(	)	$	*E*	*T*	*F*
5		*r*6	*r*6		*r*6	*r*6			
6	*s*5			*s*4				9	3
7	*s*5			*s*4					10
8		*s*6			*s*11				
9		*r*1	*s*7		*r*1	*r*1			
10		*r*3	*r*3		*r*3	*r*3			
11		*r*5	*r*5		*r*5	*r*5			

对于I_0：

1. $E' \to \cdot E$
2. $E \to \cdot E + T$
3. $E \to \cdot T$
4. $T \to \cdot T * F$
5. $T \to \cdot F$
6. $F \to \cdot (E)$
7. $F \to \cdot id$

其中的第一、第二条产生式符合“待归约项目”的要求，输入E后转移至状态1（I_1），所以在转移表0-E处写1。

其中的第三、第四条产生式符合“待归约项目”的要求，输入T后转移至状态2（I_2），所以在转移表0-T处写2。

其中的第五条产生式符合“待归约项目”的要求，输入F后转移至状态3（I_3），所以在转移表0-F处写3。

第六条$F \to \cdot (E)$点后是左括号，是终结符，符合移进项目的定义，所以动作表中0-(处写s4。

同理，第七条$F \to \cdot id$点后是id，是终结符，符合移进项目的定义，所以动作表中0-id处写s5。

对于状态1，$E' \to E\cdot$符合接受项目的定义，在1-$处写acc。

I_1中的$E \to E \cdot +T$点后为终结符，符合移进项目的定义，在1-+处写s6。

对于状态2，$E \to T\cdot$符合归约项目（点后为空）的定义，因为$FOLLOW(E) = \{+,), \$\}$，所以2-+, 2-), 2-$三个位置都应该填r2。

（其余填表过程略）

- 使用SLR分析表的LR分析是**SLR分析**，能够构造出无冲突的SLR分析表的文法是**SLR文法。**
- SLR分析表中如果出现动作冲突（移进-归约冲突，归约-归约冲突），则文法就不是SLR的。
- SLR文法都不是二义的，但是它描述能力有限，有些非二义的文法不能用SLR分析。

解决方法：**规范LR分析（LR(1)分析）**

构造规范的LR分析表（LR(1)）

LR(1)项目：重新定义项目，让它带上搜索符：

$[A \to \alpha \cdot \beta, a]$

- 搜索符是在字串$\alpha\beta$所在的右句型中直接跟在β后面的终结符。
- 搜索符在β不为ε的情况下是没什么用处的，但当β为ε时，它决定了何时将$\alpha\beta$归约为A。
- LR(1)中的1实际上指搜索符的长度。

构造规范LR分析表步骤：

- 拓广文法并为产生式编号
- 构造**LR(1)项目**集规范族（顺便计算所有goto函数）
- 构造识别活前缀的DFA
- 根据LR(1)项目集规范族和识别活前缀的DFA构造action-goto表

e.g.

为下面文法构造规范LR分析表

$$S \to BB$$
$$B \to bB \mid a$$

先拓广文法并编号：

1. $S' \to S$
2. $S \to BB$
3. $B \to bB$
4. $B \to a$

构造项目集规范族：

- 初始项目集：$\{[S' \to \cdot S, \$]\}$
- ✩计算$closure(\{[S' \to \cdot S, \$]\})$得到第一个项目集$I_0$。（闭包函数定义第二条：如果$A \to \alpha \cdot B\beta$在$closure(I)$中，且$B \to \gamma$是**产生式**，那么如果项目$B \to \cdot\gamma$还不在$closure(I)$中的话，则把它加入。运用这条规则，直到没有更多的项目可以加入$closure(I)$为止。**【点后面是非终结符的需要进**

行此操作】）

对*closure(I)*的修改：若项目$[A\to\alpha\cdot B\beta, a]$在$I$中，对每一条形式为$B\to\eta$的产生式，令$b\in \text{FIRST}(\beta a)$，把项目$[B\to\cdot\eta, b]$加入闭包。（注意，$b$不可能为ε，因为$a$不为ε）

- 计算$goto(I_0, X)$得到其他项目集（状态）
- 同理，可以用$goto(I_i, X)$计算从任意状态I_i开始，向栈中压入一个文法符号X所得到的状态（项目集），直到没有新的状态（项目集）产生为止。

$S'\to S\cdot, \$$ I_1

$S\to BB\cdot, \$$ I_5

$S'\to\cdot S, \$$ I_0
$S\to\cdot BB, \$$
$B\to\cdot bB, b/a$
$B\to\cdot a, b/a$

$S\to B\cdot B, \$$ I_2
$B\to\cdot bB, \$$
$B\to\cdot a, \$$

$B\to b\cdot B, \$$ I_6
$B\to\cdot bB, \$$
$B\to\cdot a, \$$

$B\to b\cdot B, b/a$ I_3
$B\to\cdot bB, b/a$
$B\to\cdot a, b/a$

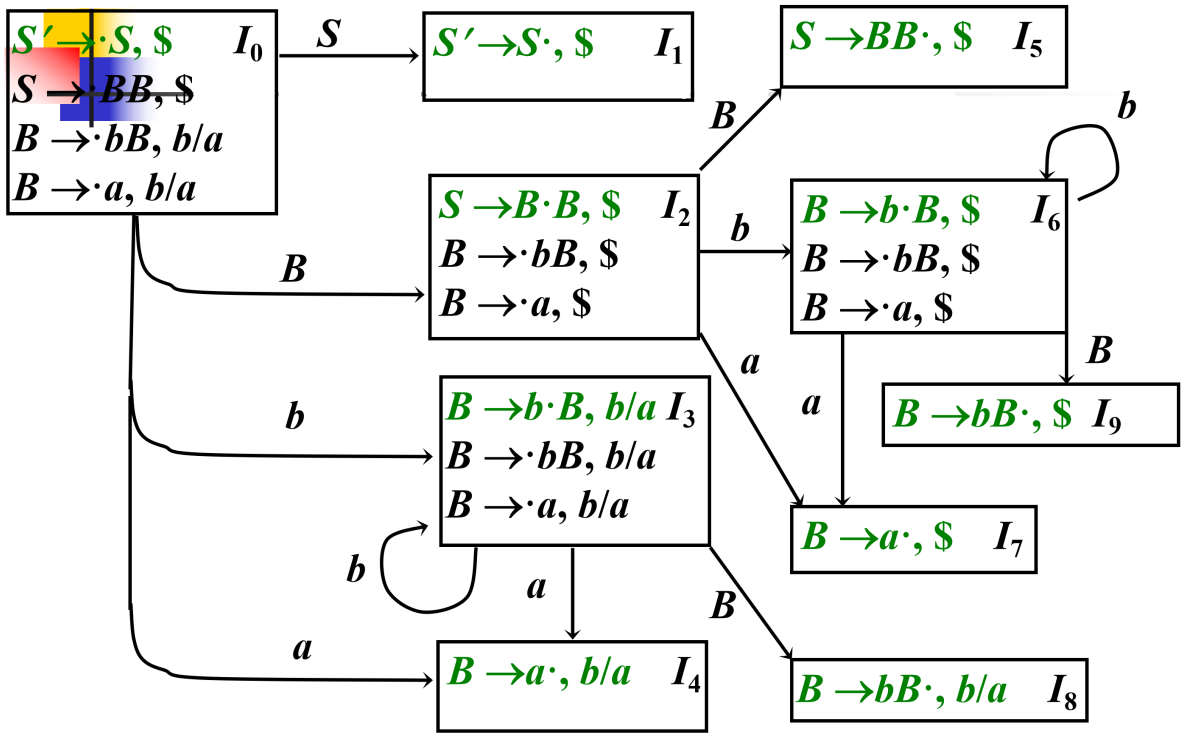
$B\to bB\cdot, \$$ I_9

$B\to a\cdot, \$$ I_7

$B\to a\cdot, b/a$ I_4

$B\to bB\cdot, b/a$ I_8

画出识别活前缀的DFA：



构造action-goto表：

项目集中的项目分四类：

- 接受项目$[S' \to S\cdot, \$]$：$action[i, \$] = acc$
- 移进项目$A \to \alpha \cdot a\beta, b$：$action[i, a] = sj$
- 归约项目$A \to \alpha\cdot, b$：$action[i, b] = rj$
- 待归约项目$A \to \alpha \cdot B\beta$：$goto[i, B] = j$

状态	动　作			转　移	
	a	***b***	**$**	***S***	***B***
0	**s4**	**s3**		**1**	**2**
1			**acc**		
2	**s7**	**s6**			**5**
3	**s4**	**s3**			**8**
4	**r3**	**r3**			

状态	动　作			转　移	
	a	***b***	**$**	***S***	***B***
5			**r1**		
6	**s7**	**s6**			**9**
7			**r3**		
8	**r2**	**r2**			
9			**r2**		

规范LR分析法的问题：状态数庞大，构造难度大。

缓解的办法：LALR分析法

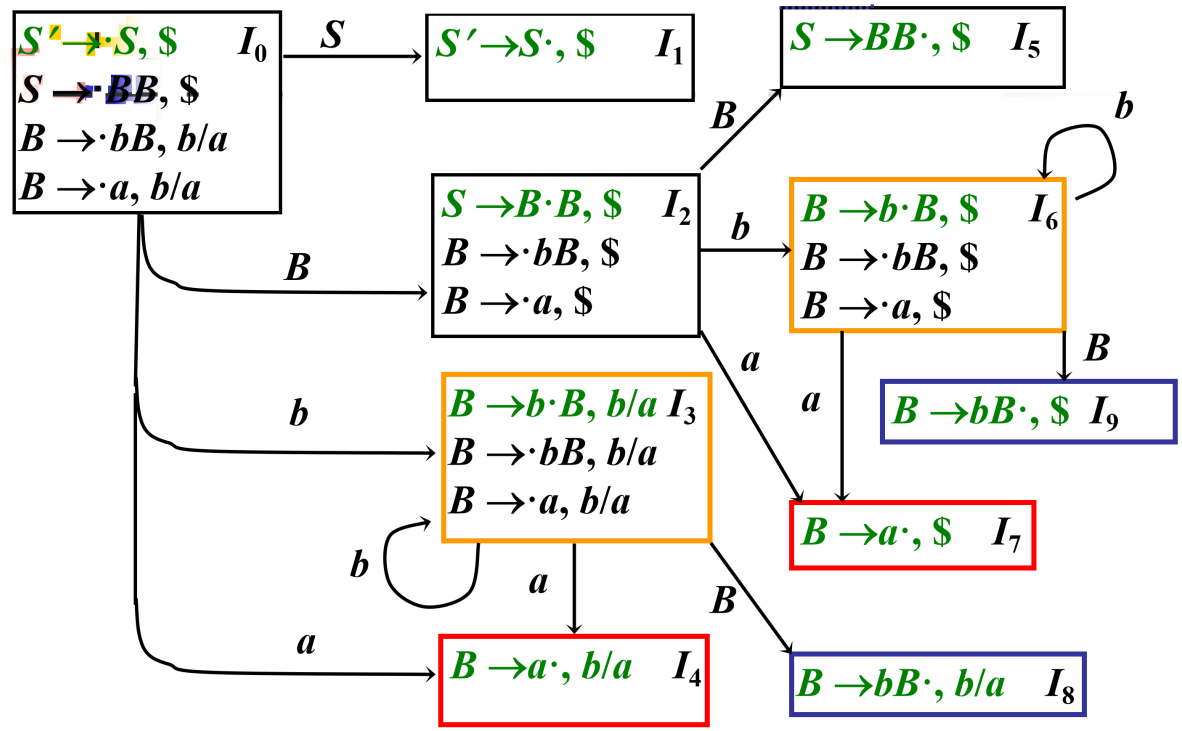
构造LALR分析表

LALR: LookAhead LR

LALR的分析能力介于SLR和规范LR之间。

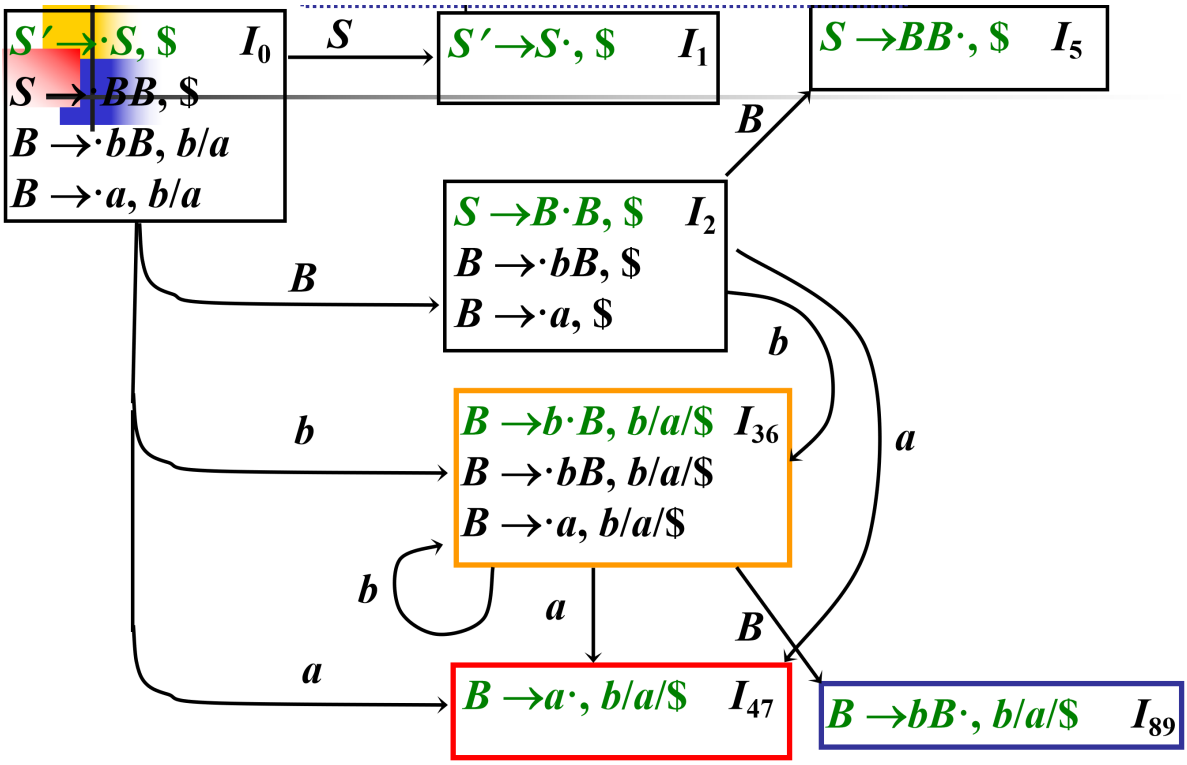
实际的编译器经常使用LALR分析法。

LALR分析就是在LR(1)的基础上合并同心项目集，合并同心项目集后的项目集族叫做LALR(1)项目集族。



其中，I_4与I_7仅搜索符不同，可以将它们视为同心项目集。I_3和I_6、I_8和I_9同理，均为同心项目集。

合并同心项目集后的结果：



合并同心项目集可能会引起冲突，不会引起新的移进归约冲突，但可能产生新的归约归约冲突。

第四章 语法制导的翻译

语法制导定义

一个语法制导定义包括：基础文法（上下文无关文法）、产生式对应的语义规则、文法符号的属性。（不包含属性依赖图）

语法制导定义：上下文无关文法的扩展，每个文法符号多了一组**属性**，每个产生式多了一组语义规则。（语法制导的定义是带属性和语义规则的上下文无关文法；每个文法符号都有一组属性，无论终结符还是非终结符）

在语法制导定义中，每个文法符号有一组属性，每个文法产生式$A \to \alpha$有一组形式为$b = f(c_1, c_2, \ldots, c_k)$的语义规则，其中f是函数，b和$c_1, \ldots, c_k$是该产生式的文法符号的属性：

- **综合属性**：如果b是A（产生式左部）的属性，$c_1, \ldots, c_k$是产生式右部文法符号的属性或A的其他属性，那么b称为A的综合属性。
- **继承属性**：如果b是产生式右部某个文法符号X的属性，$c_1, \ldots, c_k$是A的属性或右部文法符号的属性，那么b称为X的继承属性。

注：综合属性和继承属性没有重合。

综合属性是计算$A \to \alpha$中A的属性；继承属性是计算$A \to \alpha$右部α里的属性。

e.g. 简单计算器的语法制导定义：

产生式	**语义规则**
$L \to E$**n**	$print(E.val)$
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val \times F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to$**digit**	$F.val =$**digit**$.lexval$

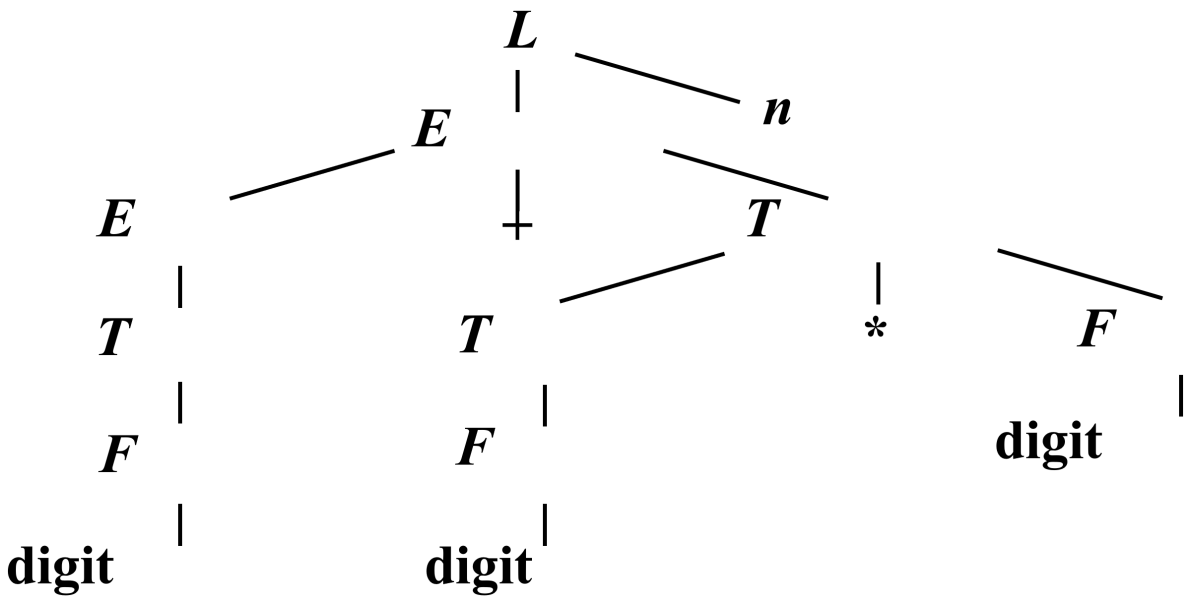
综合属性

如果把上述表格的第一行的语义规则改写为：$L.dummy = print(E.val)$（L的虚拟综合属性dummy），则这里面只有综合属性，符合**S属性定义。**（**S属性定义**：仅仅使用综合属性的语法制导定义称为S属性定义）

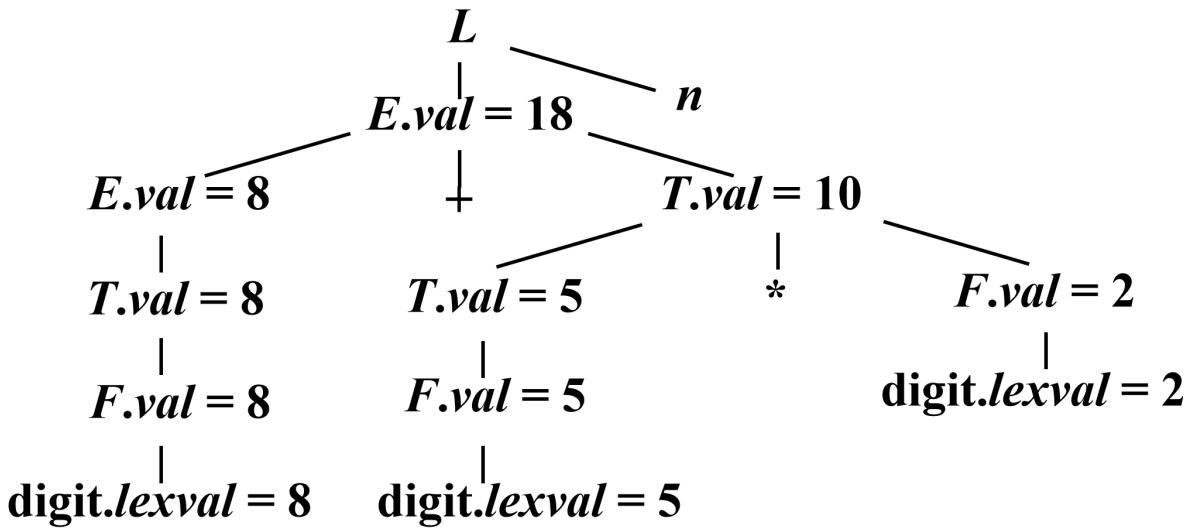
注释分析树：结点的属性值都标注出来的分析树。

- 计算结点属性值的过程叫做**注释（加注）**或**修饰。**

e.g. $8 + 5 * 2n.$



注释后：



分析树各结点属性的计算可以**自下而上**地完成。

继承属性

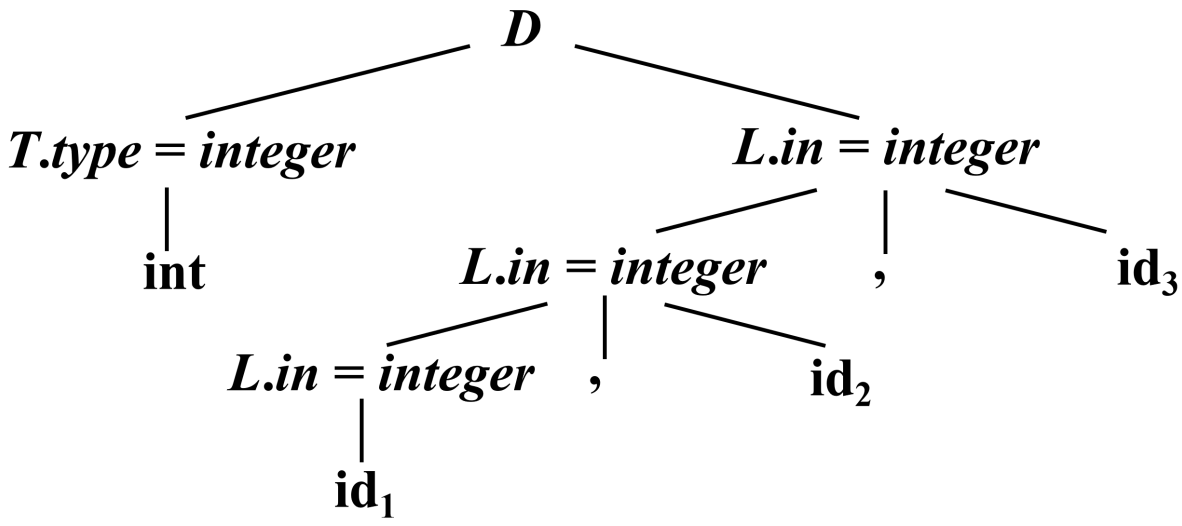
编程语言的一些构造的属性依赖于它们所在的**上下文**，此时使用**继承属性**比较方便。

e.g.

- 有继承属性的语法制导定义：int id, id, id

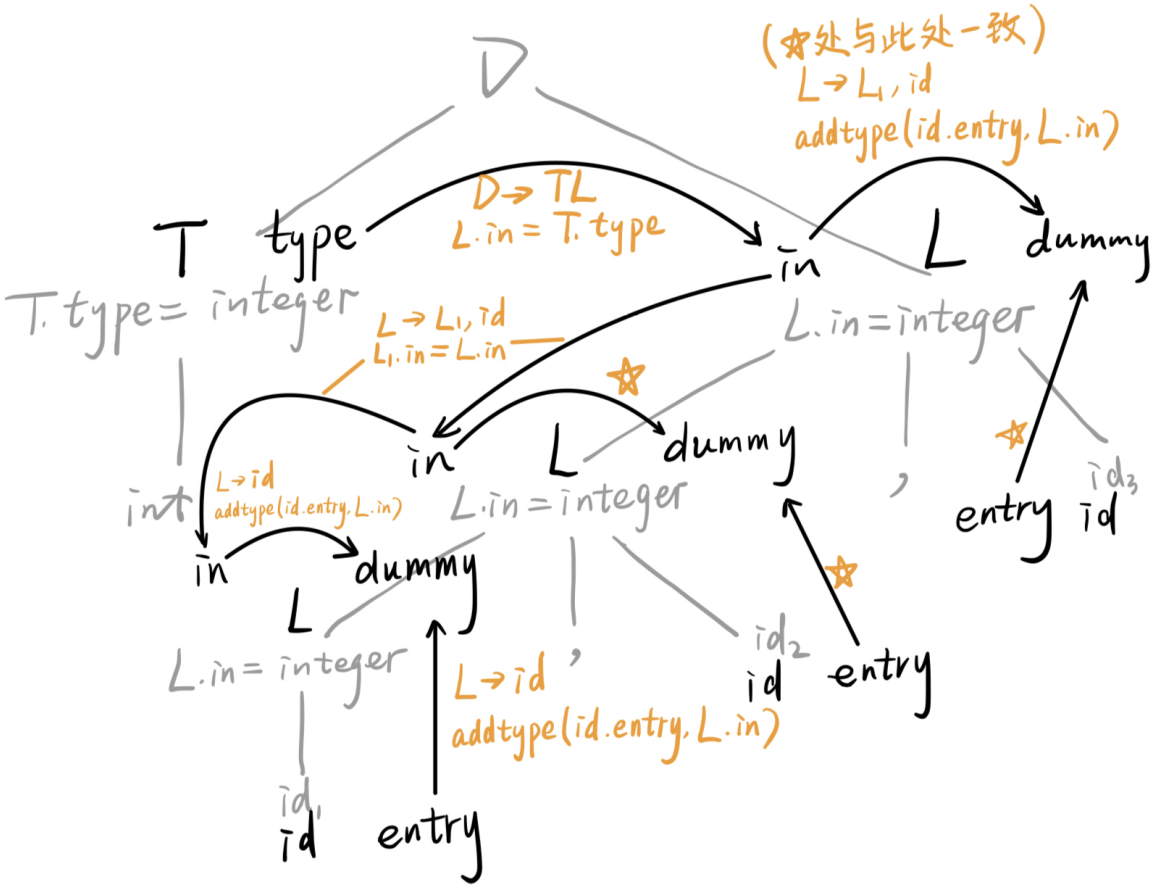
产　生　式	语 义 规 则
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow$ int	$T.type = integer$
$T \rightarrow$ real	$T.type = real$
$L \rightarrow L_1$, id	$L_1.in = L.in$; $addtype$ (id.$entry$, $L.in$)
$L \rightarrow$ id	$addtype$ (id.$entry$, $L.in$)

int id_1, id_2, id_3的注释分析树



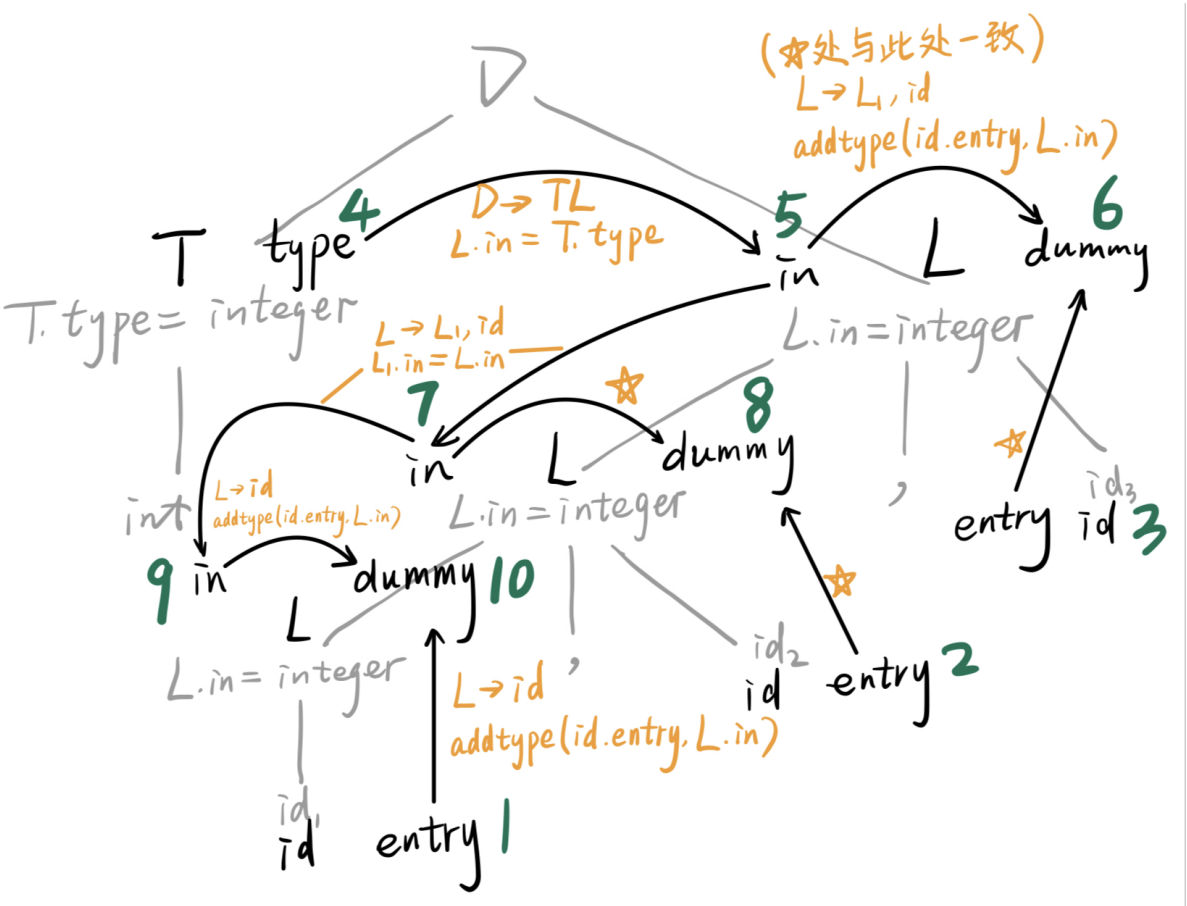
属性依赖图

对上述注释分析树构造属性依赖图：



拓扑排序：有向无环图结点的一种排序，使得边只会从该次序中先出现的结点到后出现的结点。（不能出现逆着箭头走的情况）

下面图中的1-10是一种情况，有的结点的编号可以变顺序。



属性计算次序

如何确定属性计算次序：

- 构造输入的分析树
- 构造属性依赖图
- 对结点进行拓扑排序
- 按拓扑排序的次序计算属性

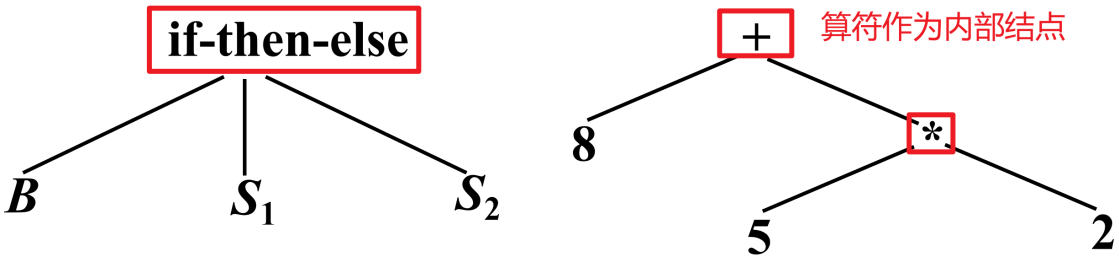
以上是一种语法制导翻译概念上的实现方法；这种方法叫做**分析树方法**。

S属性定义的自下而上计算

抽象语法树是分析树的浓缩表示：算符和关键字是作为内部结点，单非产生式**（右部只有一个非终结符的产生式）**链可能消失。

抽象语法树是一种中间表示，允许把翻译从分析中分离，形成先分析后翻译的方式。

e.g. （红框中均为"算符作为内部结点"）



抽象语法树的数据结构：

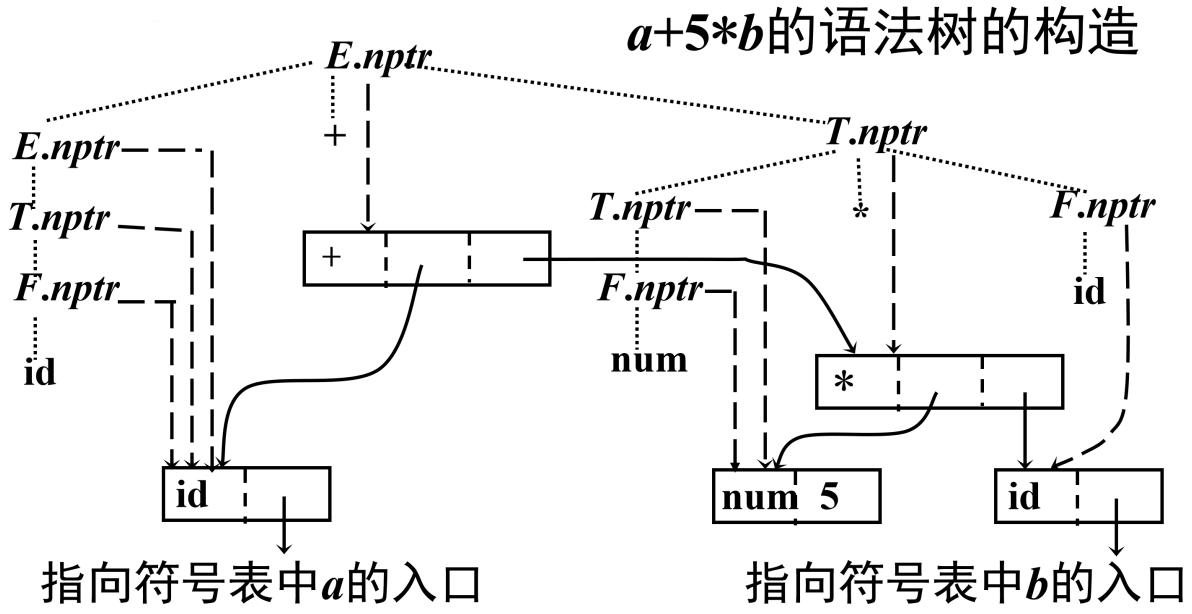
- 算符结点：算符域（结点标记），2个运算对象域（存放指向运算对象的指针）
- 基本运算对象结点：运算对象类别域，运算对象的值

语义规则中的函数：（它们都返回结点指针）

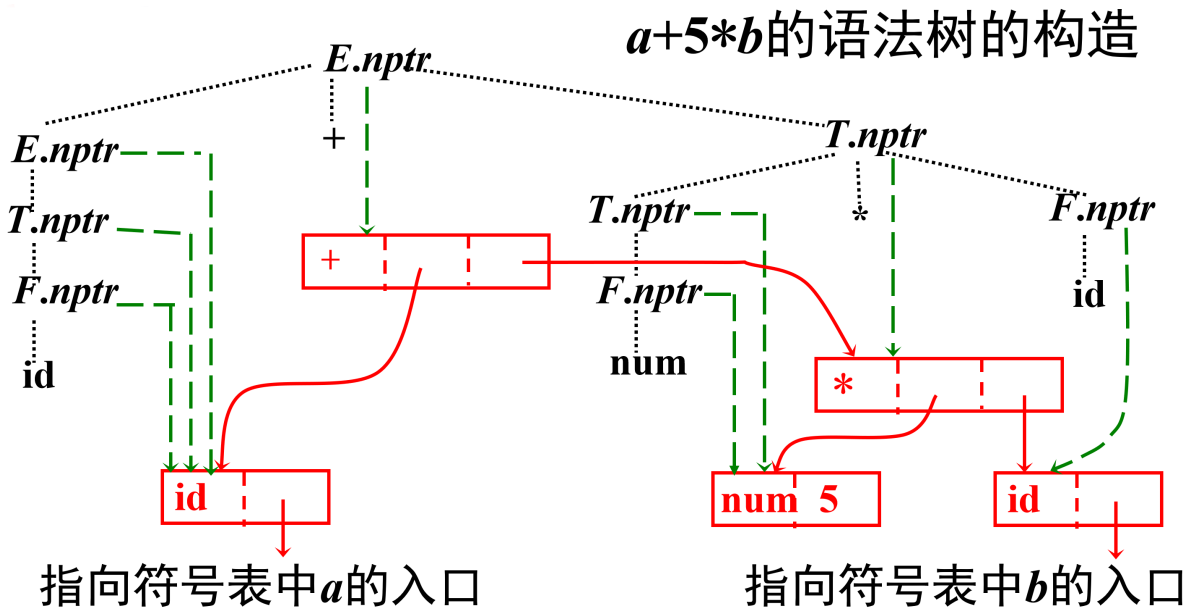
- $mkLeaf(id, entry)$：建立标记为id的标识符结点；entry：符号表中该标识符条目的指针。
- $mkLeaf(num, val)$：建立标记为num的整数结点；结点另有一个域，其值为val，它是该整数的值。
- $mkLeaf(op, left, right)$：算符结点，其余两个指针是该结点的左右子树的指针。

e.g.

产　生　式	语　义　规　则
$E \rightarrow E_1 + T$	$E.nptr = mknode(\text{‘+’}, E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow T_1 * F$	$T.nptr = mknode(\text{‘*’}, T_1.nptr, F.nptr)$
$T \rightarrow F$	$T.nptr = F.nptr$
$F \rightarrow (E)$	$F.nptr = E.nptr$
$F \rightarrow \text{id}$	$F.nptr = mkleaf(\text{id}, \text{id}.entry)$
$F \rightarrow \text{num}$	$F.nptr = mkleaf(\text{num}, \text{num}.val)$

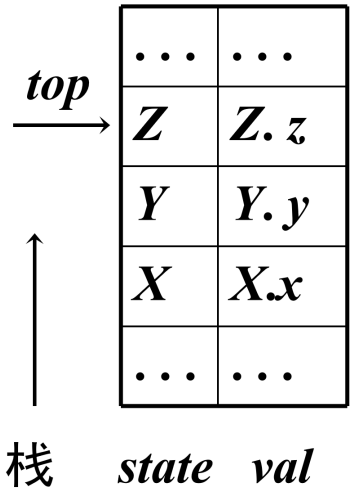


上图中体现出了三种不同的"make leaf"结点。

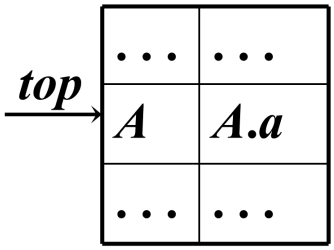


可以通过修改LR分析器的栈使之在分析的同时计算属性。

- 将LR分析器增加一个域来保存综合属性值：



若产生式$A \rightarrow XYZ$的语义规则是
$A.a = f(X.x, Y.y, Z.z)$，
那么归约后：



翻译器面临$8 + 5 * 2n$时的动作：

输入	*state*	*val*	所用产生式
8+5*2n	-	-	
+5*2n	8	8	
+5*2n	*F*	8	$F \rightarrow$ digit
+5*2n	*T*	8	$T \rightarrow F$
+5*2n	*E*	8	$E \rightarrow T$
5*2n	*E* +	8+	
*2n	*E*+5	8+5	
*2n	*E*+*F*	8+5	$F \rightarrow$ digit

输入	*state*	*val*	所用产生式
*2n	*E*+*T*	8+5	$T \rightarrow F$
2n	*E*+*T**	8+5*	
n	*E*+*T**2	8+5*2	
n	*E*+*T**F	8+5*2	$F \rightarrow$ digit
n	*E*+*T*	8+10	$T \rightarrow T * F$
n	*E*	18	$E \rightarrow E + T$
	E n	18-	
	L	18	$L \rightarrow E$ n

L属性定义的自上而下计算

L属性定义

L属性定义：如果每个产生式$A \to X_1 X_2 \dots X_n$的每条语义规则计算的属性要么是A的综合属性；或者计算的是X_j的继承属性（$1 \le j \le n$），它仅依赖：

- 该产生式中X_j左边符号$X_1, X_2, \dots, X_{j-1}$的属性
- A的继承属性

S属性定义属于L属性定义。

翻译方案☆

语法制导翻译方案和语法制导定义的不同之处：**语义动作**放在括号{}里。

e.g. A→α **{action}** β

可以把语义动作想象成一个文法符号，在分析过程中对该符号进行推导或归约的时候，就是该语义动作执行的时候。

A→α **M** β

e.g. 把有加和减的中缀表达式翻译成后缀表达式：如果输入是8 + 5 - 2，则输出是8 5 + 2 -。

翻译方案：

$$E \to T\ R$$
$$R \to \text{addop}\ T\ \{print\ (\text{addop}.lexeme)\}\ R_1 \mid \varepsilon$$
$$T \to \text{num}\ \{print\ (\text{num}.val)\}$$

$$E \Rightarrow \boxed{T}\ R \Rightarrow \boxed{\text{num}\ \{print\ (8)\}}\ R$$
$$\Rightarrow \text{num}\{print\ (8)\}\text{addop}\ T\{print\ (+)\}R$$
$$\Rightarrow \text{num}\{print(8)\}\text{addop num}\{print(5)\}\{print\ (+)\}R$$
$$\dots\ \{print(8)\}\ \dots\{print(5)\}\{print(+)\}\text{addop}\ T\{print(-)\}\ R$$
$$\dots\ \{print(8)\}\ \dots\{print(5)\}\{print(+)\}\ \dots\{print(2)\}\{print(-)\}$$

只有综合属性时，只要将动作放在对应产生式右部末端即可得到翻译方案。

如果有继承属性，则对于L属性定义，总能构造满足下述三条限制的翻译方案：

- 产生式右部符号的继承属性必须在先于这个符号的动作中计算。
- 一个动作不能引用该动作右边符号的综合属性。
- 左部非终结符的综合属性只能在它所引用的所有属性都计算完成后才能计算。计算该属性的动作通常放在产生式右部的末端。

e.g. 数学排版语言EQN

E sub 1 .val $E_1.val$

产 生 式	语 义 规 则
$S \to B$	$B.ps = 10;\quad S.ht = B.ht$
$B \to B_1\ B_2$	$B_1.ps = B.ps;\quad B_2.ps = B.ps;$ $B.ht = max(B_1.ht, B_2.ht)$
$B \to B_1$ sub B_2	$B_1.ps = B.ps;\quad B_2.ps = shrink(B.ps);$ $B.ht = disp\ (B_1.ht, B_2.ht)$
$B \to$ text	$B.ht = \text{text}.h \times B.ps$

- 上述L属性定义定义编排单元大小和高度。
- 继承属性B.ps表示点的大小，它会影响公式高度。
- 综合属性text.h，B.ht和S.ht表示高度。

（粗略理解综合属性&继承属性）表达式左边的对应综合属性，右边的对应继承属性。

$S \to$ $\{B.ps = 10\}$ B.ps是继承属性，在B计算完成前进行

B $\{S.ht = B.ht\}$ S.ht是综合属性，放在后面

$B \to$ $\{B_1.ps = B.ps\}$

B_1 $\{B_2.ps = B.ps\}$

B_2 $\{B.ht = max(B_1.ht, B_2.ht)\}$

$B \to$ $\{B_1.ps = B.ps\}$

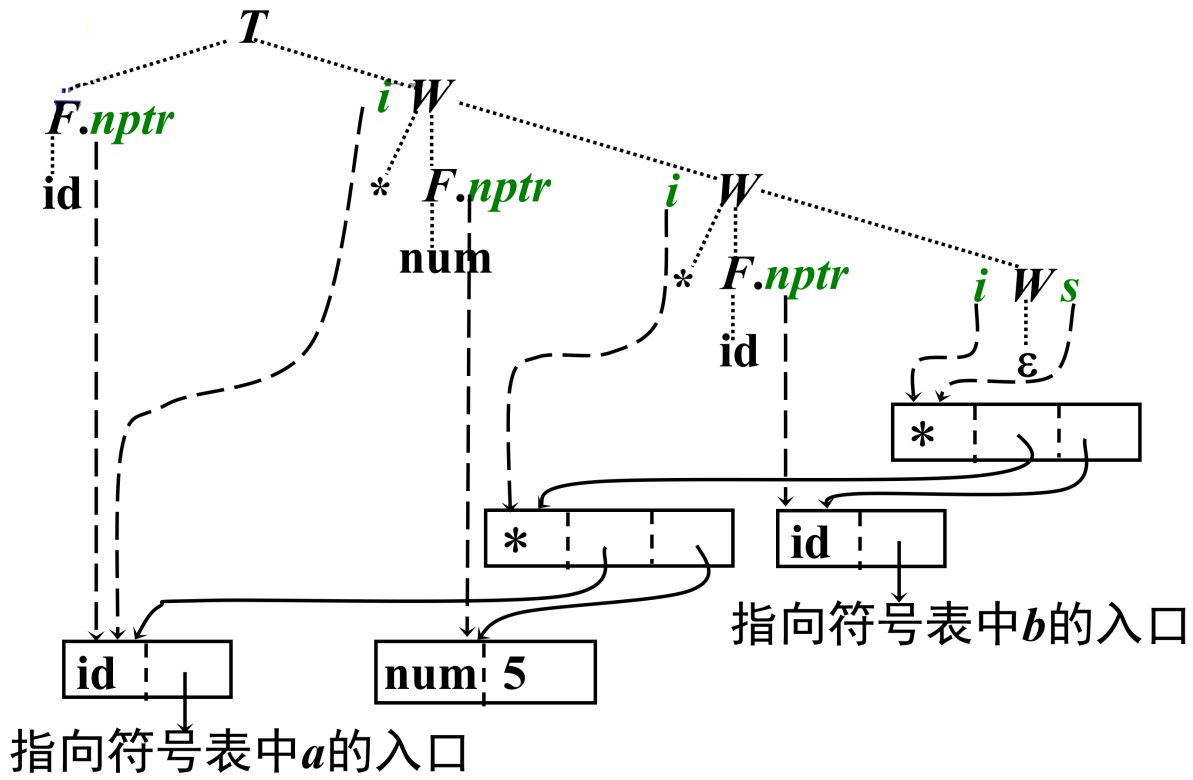
B_1

sub $\{B_2.ps = shrink(B.ps)\}$

B_2 $\{B.ht = disp\ (B_1.ht, B_2.ht)\}$

$B \to$ text $\{B.ht = \text{text}.h \times B.ps\}$

抽象语法树：



第五章 类型检查

执行错误和安全语言

执行错误：程序运行时出现的错误。

程序运行时的执行错误分成两类：

- 会被捕获的错误(trapped error)
- 不会被捕获的错误(untrapped error)

良行为的程序：一个程序的运行不可能引起不会被捕获的错误，则称它是良行为的。

安全语言：所有合法程序都是良行为的语言。

禁止错误：

- 全部不会被捕获的错误
- 一部分会被捕获的错误

为语言设计类型系统的目标是排除禁止错误。

C语言不是类型可靠的语言。（Java语言是类型可靠的）

类型化语言和类型系统

变量的类型：变量在程序执行期间的取值范围。

类型化的语言：若语言的规范为其每种运算都定义了各运算对象和运算结果所允许的类型，则该语言称为**类型化语言**。

- 显式类型化语言：在一种语言中，若函数和变量的类型必须显式声明，则该语言为显式类型化语言。（e.g. Java, C++）
- 隐式类型化语言：类型声明不是必不可少的语言。

在类型化语言中，**类型系统**由一组**定型规则**构成。

类型检查：

- 根据定型规则来确定程序中各语法构造的类型。
- 类型检查的目的是拒绝那些有**类型错误**的程序。

能够通过类型检查的程序称为**良类型的**程序。

如果良类型程序一定是良行为的，则称该语言是**类型可靠的**。类型可靠的语言一定是安全语言。

类型系统主要用来说明程序设计语言的定型规则，它独立于类型检查算法。

e.g. 有关自然数的逻辑系统：

- 自然数表达式：$a+b, 3$
- 合式公式：$a+b=3, (d=3) \wedge (c<10)$
- 推理规则：$\frac{a<b, b<c}{a<c}$

e.g. 类型系统：（可能考小题）

- 类型表达式：int→int
- （在逻辑学中，“⊢”表示推导出、推理出）断言：$\{x:int\} \vdash x+3:int$
- 定型规则：$\frac{\mathrm{T} \vdash M:int, \mathrm{T} \vdash N:int}{\mathrm{T} \vdash M+N:int}$

断言有三种具体形式：

- 环境断言：$\mathrm{T} \vdash \Diamond$
- 语法断言：$\mathrm{T} \vdash nat$
- 定型断言：$\mathrm{T} \vdash M:T$

推理规则：

- 推理规则是在一组已知有效断言的基础上，声称某个断言的有效性。
- 其一般形式：
  - （规则名）（注释） 推理规则 （注释）
- 其中的推理规则习惯表示法

$$\frac{\Gamma_1 \vdash S_1, \ldots, \Gamma_n \vdash S_n}{\Gamma \vdash S}$$

  - 前提、结论
  - 公理

（规则名）（注释）　推理规则　（注释）

- 环境规则
  (Env ∅)

$$\frac{}{\varnothing \vdash \Diamond}$$

- 语法规则
  (Type Bool)

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \textbf{boolean}}$$

- 定型规则
  (Exp +)

$$\frac{\Gamma \vdash M : \textbf{int}, \Gamma \vdash N : \textbf{int}}{\Gamma \vdash M + N : \textbf{int}}$$

  - 推理规则的结论是定型断言，则称之为定型规则

类型检查和类型推断：

- 类型检查：用语法制导的方式，根据上下文有关的定型规则来判定程序构造是否为良类型的过程。
- 类型判断：类型信息不完全的情况下的定型判定问题。

类型系统：

环境规则
(Env ∅)

$$\frac{}{\varnothing \vdash \Diamond}$$

(Decl Var)

$$\frac{\Gamma \vdash T,\ \text{id} \notin dom\,(\Gamma)}{\Gamma, \text{id} : T \vdash \Diamond}$$

语法规则

(Type Bool) $$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \textit{boolean}}$$

(Type Int) $$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \textit{integer}}$$

(Type Void) $$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \textit{void}}$$

*void*用于表示语句类型

下面三条分别是指针、数组和函数调用。（函数调用曾在考试中出现过）

语法规则

(Type Ref) **(*T*≠*void*)** $$\frac{\Gamma \vdash T}{\Gamma \vdash \textit{pointer}(T)}$$

(Type Array) **(*T*≠*void*)** $$\frac{\Gamma \vdash T,\ \Gamma \vdash N : \textit{integer}}{\Gamma \vdash \textit{array}\ (N, T)}\ (N>0)$$

(Type Function) **(T_1, T_2≠*void*)** $$\frac{\Gamma \vdash T_1,\ \Gamma \vdash T_2}{\Gamma \vdash T_1 \rightarrow T_2}$$

定型规则——表达式

(Exp Truth) $$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \textbf{truth} : \textit{boolean}}$$

(Exp Num) $$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \textbf{num} : \textit{integer}}$$

(Exp Id) $$\frac{\Gamma_1, \textbf{id} : T, \Gamma_2 \vdash \Diamond}{\Gamma_1, \textbf{id} : T, \Gamma_2 \vdash \textbf{id} : T}$$

定型规则——表达式

(Exp Mod)

$$\frac{\Gamma \vdash E_1: integer,\ \Gamma \vdash E_2: integer}{\Gamma \vdash E_1 \text{ mod } E_2: integer}$$

(Exp Index)

$$\frac{\Gamma \vdash E_1: array(N, T), \Gamma \vdash E_2: integer \ \ (0 \le E_2.\text{val} < N)}{\Gamma \vdash E_1[E_2] : T}$$

(Exp Deref)

$$\frac{\Gamma \vdash E : pointer(T)}{\Gamma \vdash E\uparrow : T}$$

定型规则——表达式

(Exp FunCall)

$$\frac{\Gamma \vdash E_1: T_1 \to T_2, \qquad \Gamma \vdash E_2: T_1}{\Gamma \vdash E_1\ (E_2) : T_2}$$

定型规则——语句

(State Assign) (*T*=*boolean* or *integer*)

$$\frac{\Gamma \vdash \text{id} : T,\ \Gamma \vdash E : T}{\Gamma \vdash \text{id} := E : void}$$

(State If)

$$\frac{\Gamma \vdash E : boolean,\ \Gamma \vdash S : void}{\Gamma \vdash \text{if } E \text{ then } S : void}$$

(State While)

$$\frac{\Gamma \vdash E : boolean,\ \Gamma \vdash S : void}{\Gamma \vdash \text{while } E \text{ do } S: void}$$

定型规则——语句

(State Seq)

$$\frac{\Gamma \vdash S_1: void,\ \Gamma \vdash S_2: void}{\Gamma \vdash S_1; S_2 : void}$$

类型表达式的等价

结构等价&名字等价

（常考概念辨析、分析大题）

【理解】

- 结构等价：把名字换成具体的类型，如果类型一样，则满足结构等价。
- 名字等价：名字不一样就不符合名字等价。

C语言对除记录（结构体）以外的所有类型采用结构等价，而对记录（结构体）类型采用名字等价，以避免类型表示中的环。（考试考过）

e.g. 分别在名字等价和结构等价下，下面的哪些变量具有相同的类型？

```
type link = ↑cell;
var next : link;
    last : link;
    p : ↑cell;
    q, r : ↑cell;
```

对于next和p：它们的名字不同，不符合名字等价，但由于link=↑cell，所以它们结构等价。

其余略。

e.g. 分别在名字等价和结构等价下，下面的哪些变量具有相同的类型？

```
type link = ↑cell;
     np = ↑cell;
     nqr = ↑cell;
var next : link;
    last : link;
    p : np;
    q : nqr;
    r : nqr;
```

对于p和q：它们的名字不同，不符合名字等价，但由于np=↑cell且nqr=↑cell，所以它们结构等价。

其余略。

记录类型

记录类型从某种意义上来说是它**各域类型的积。**

(Type Record) (l_i是有区别的)

$$\frac{\Gamma \vdash T_1, \ldots, \Gamma \vdash T_n}{\Gamma \vdash record(l_1:T_1, \ldots, l_n:T_n)}$$

(Val Record) (l_i是有区别的)

$$\frac{\Gamma \vdash M_1:T_1, \ldots, \Gamma \vdash M_n: T_n}{\Gamma \vdash \textbf{record}\ (l_1=M_1, \ldots, l_n=M_n) : record\ (l_1:T_1, \ldots, l_n:T_n)}$$

(Val Record Select) ($1 \le j \le n$)

$$\frac{\Gamma \vdash M : record(l_1:T_1, \ldots, l_n:T_n)}{\Gamma \vdash M.l_j : T_j}$$

考察方向：写类型表达式

e.g. C的程序段：

```
typedef struct {
    int address;
    char lexeme[15];
}row;
```

定义类型名row代表类型表达式：

record(address: int, lexeme: array(15, char))

e.g. 假如有下列C的声明：

```
typedef struct {
    int a, b;
} CELL, *PCELL;
CELL foo[100];
PCELL bar(x, y) int x; CELL y; {...}
```

为变量foo和函数bar的类型写出类型表达式。

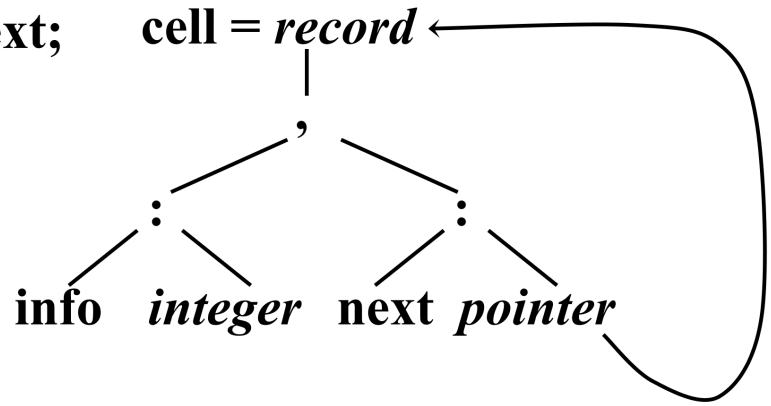
答：

foo: array(100, record(a: integer, b: integer))

bar: integer×record(a: integer, b: integer) → pointer(record(a: integer, b: integer))

类型表示中的环

```
typedef  struct _cell {
    int info;
    struct _cell *next;
} cell;
```



第六章 运行时存储空间的组织和管理

局部存储分配

过程

过程定义是一个声明，它的最简单形式是将一个名字和一个语句联系起来。

- 形式参数（形参）：出现在过程定义中的某些名字
- 实在参数（实参）：出现在过程调用语句中

e.g. （用于理解形参和实参，程序中主函数为max_val()传入的参数a和b是实参，max_val()定义中的参数x和y是形参）

```cpp
#include <iostream>
using namespace std;

int max_val(int x, int y) { // 形参
    if (x > y) return x;
    return y;
}

int main() {
    int a = 3;
    int b = 2;
    int res = max_val(a, b); // 实参
    cout << res << endl;
    return 0;
}
```

名字的作用域和绑定

名字的作用域：一个声明起作用的程序部分称为该声明的作用域。（可参考"程序块"处的例子理解）

从名字到值的两步映射：

- 环境把名字映射到左值（存储空间），而状态把左值映射到右值（实际的值）
- 赋值改变状态，但不改变环境；过程调用改变环境。
- 如果环境将名字x映射到存储单元s，我们就说x被**绑定**（binding）到s

【补充知识】左值和右值：

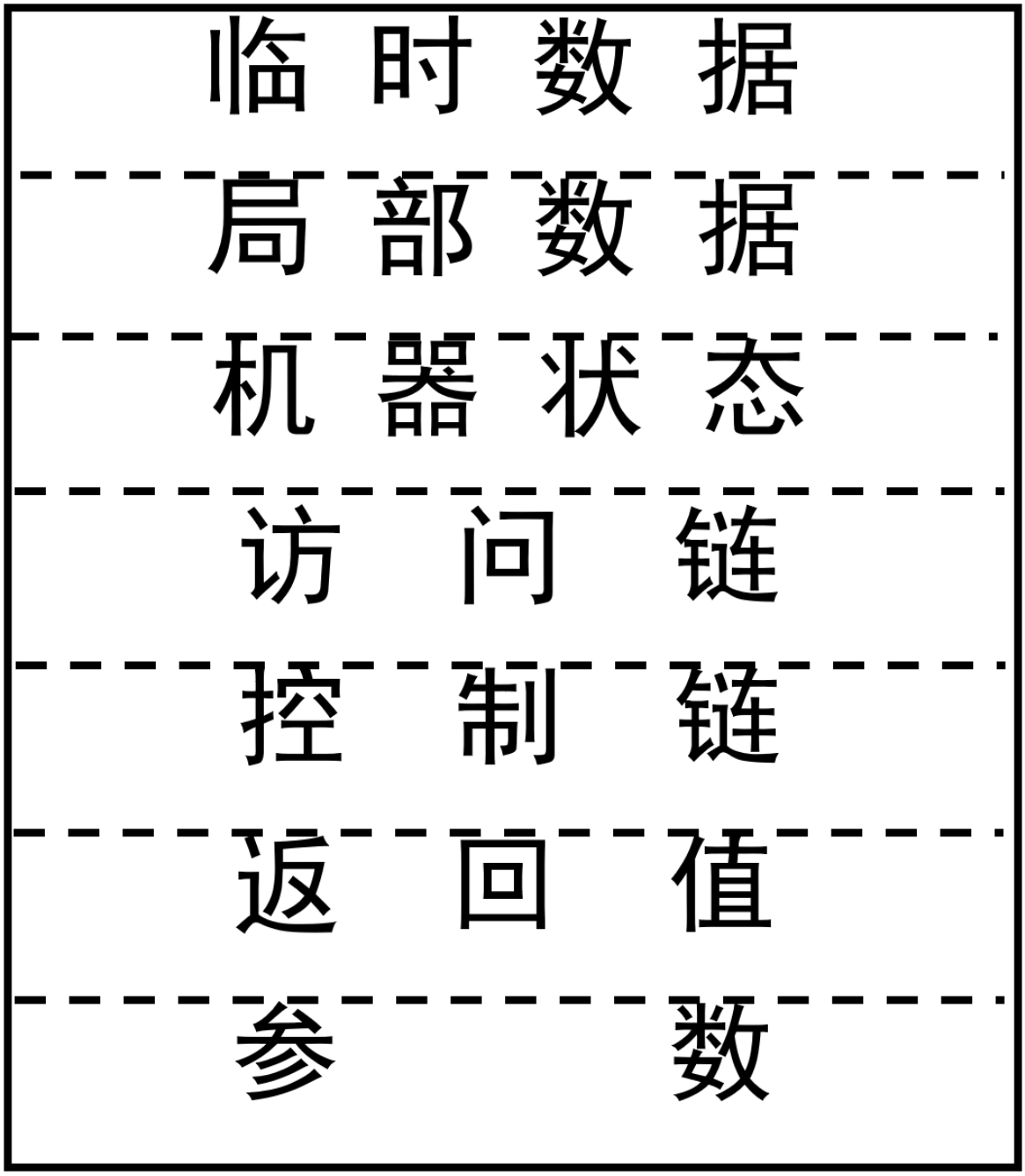
- 左值是可寻址的变量，有持久性。
- 右值一般是不可寻址的常量，或在表达式求值过程中创建的无名临时变量，不具备持久性。

静态和动态概念的对应：

静态概念	动态对应
过程的定义	过程的活动
名字的声明	名字的绑定
声明的作用域	绑定的生存期

活动记录

一般的活动记录布局：



代码、全局变量、常量不存在活动记录中。

局部数据的布局

数据对象的存储安排有一个**对齐**的问题。（因对齐而引起的无用空间称为**衬垫区**）

按照定义的顺序存储。

e.g. 在SPARC/Solaris工作站上：（double占8字节，char占1字节，long占4字节）

```
typedef struct  _a{
        char   c1;     0
        long   i;      4
        char   c2;     8
        double f;      16
}a;
```

size = 24

```
typedef struct  _b{
        char c1;     0
        char   c2;   1
        long i;      4
        double f;    8
}b;
```

size = 16

在X86/Linux机器上：（double类型要求按照4字节对齐，并非8字节对齐）

```
typedef struct  _a{
        char   c1;     0
        long   i;      4
        char   c2;     8
        double f;      12
}a;
```

size = 20

```
typedef struct  _b{
        char c1;     0
        char   c2;   1
        long i;      4
        double f;    8
}b;
```

size = 16

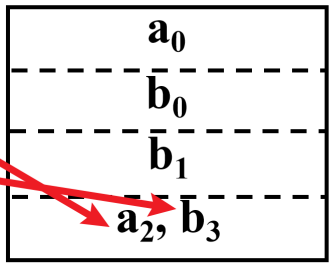
程序块

程序块特点：可以嵌套，不能重叠。

程序块结构的声明作用域由**最接近的嵌套作用域规则**给出。

```
main()
{ /* begin of B0 */
    int a = 0;
    int b = 0;
    { /* begin of B1 */
        int b = 1;
        {/* begin of B2 */
            int a = 2;
        }/* end of B2 */
        {/* begin of B3 */
            int b = 3;
        }/* end of B3 */
    }/* end of B1 */
}/* end of B0 */
```

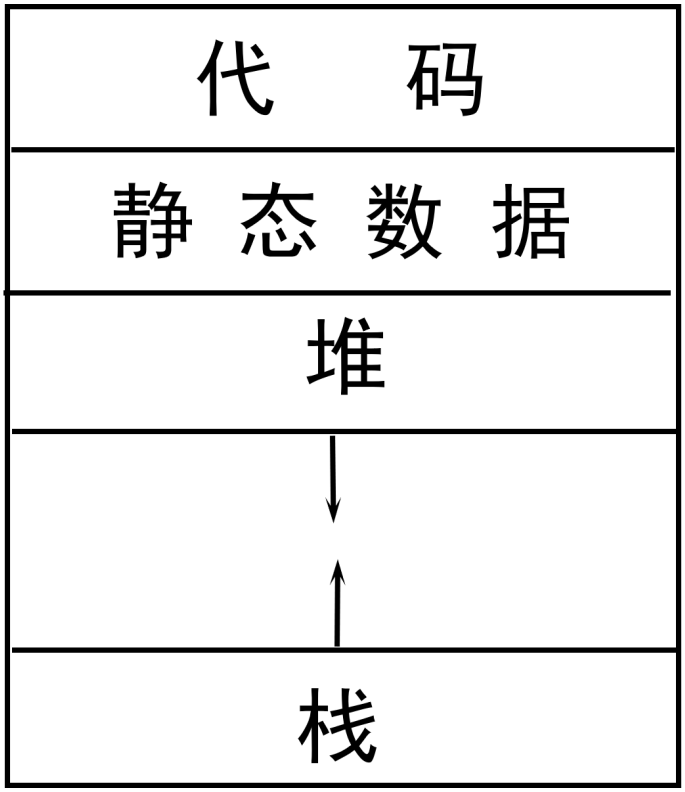
声　明	作 用 域
int a = 0;	$B_0 - B_2$
int b = 0;	$B_0 - B_1$
int b = 1;	$B_1 - B_3$
int a = 2;	B_2
int b = 3;	B_3



这里是减号！！

重叠分配存储单元

全局栈式存储分配



低地址

高地址

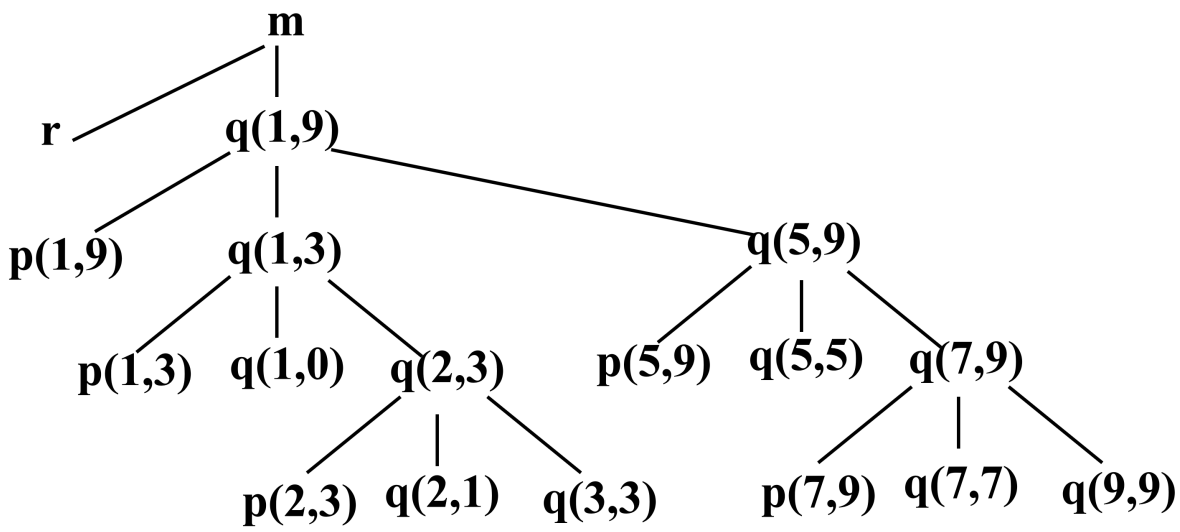
活动树和运行栈

e.g.

```
int a[11];
void readArray() {
```

```
  /* Reads 9 integers into a[1], …, a[9]. */
  int i;
  . . .
}
int partition(int m, int n) {
  // **找到一个数i，让a[i]左边都小于a[i]，右边都大于a[i]。**
  /* Picks a separator value v, and partitions
     a[m..n] so that a[m..p-1] are less than v,
     a[p]=v, and a[p+1..n] are equal to or great
     than v. Returns p. */
}
void quickSort(int m, int n) {
  int i;
  if (n > m) {
    i = partition(m, n);
    quickSort(m, i-1);
    quickSort(i+1,n);
  }
}
main() {
  readarray();
  a[0] = -9999; a[10] = 9999;
  quickSort(1, 9);
}
```

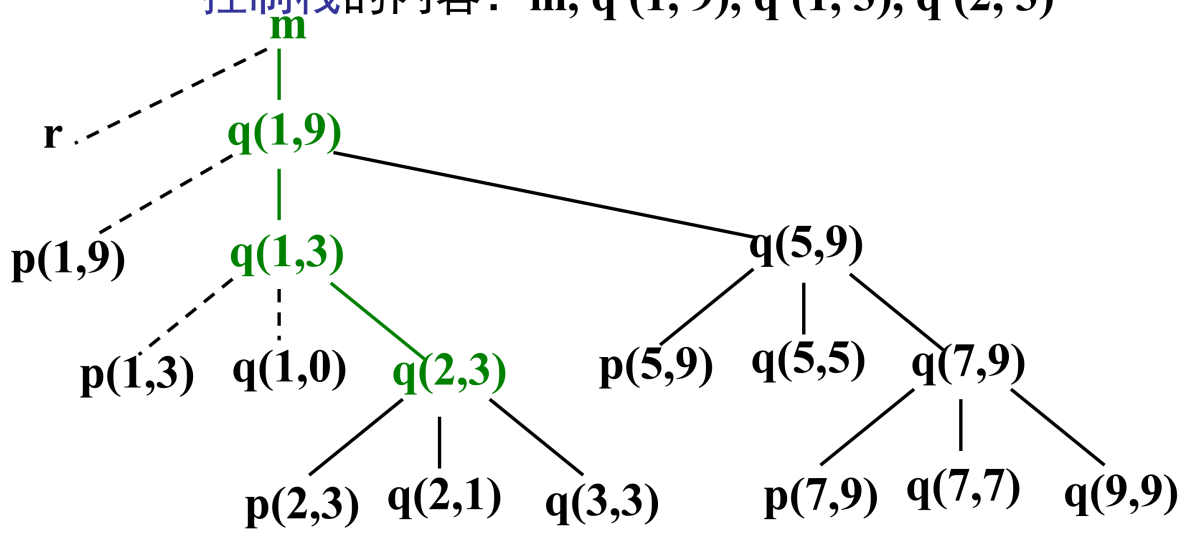
活动树：（程序对应活动树的深度优先遍历-DFS）



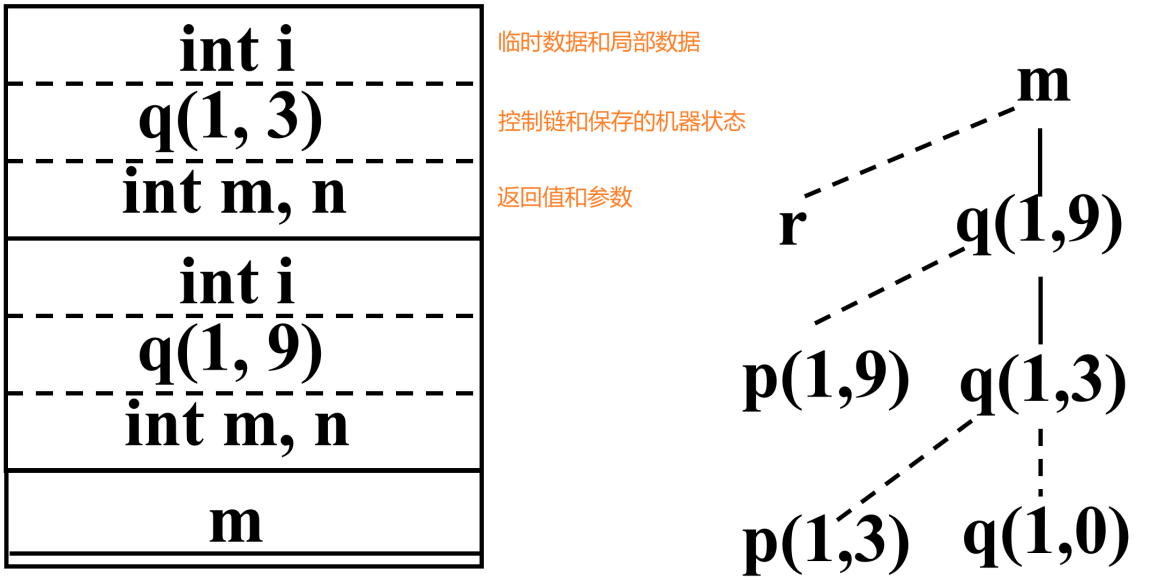
控制栈：当前**活跃着的过程活动**可以保存在一个栈中。

控制栈中当前存在的过程活动都是处在生存期内的活动。

e.g.



运行栈：把控制栈中的信息拓广到包括过程活动的活动记录。（活动记录栈）

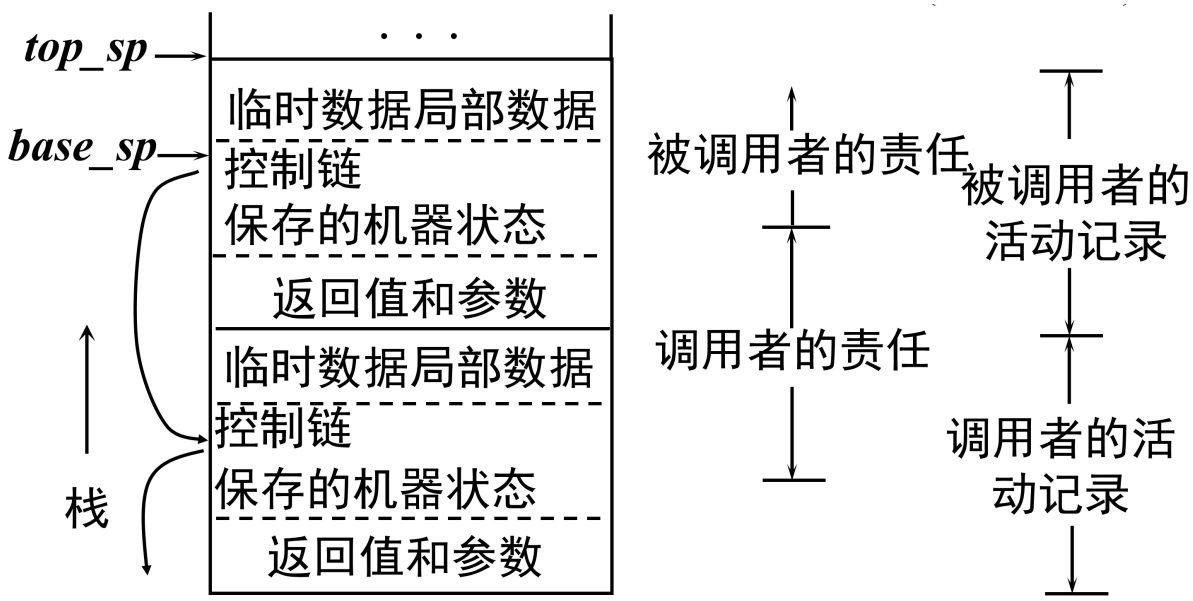


树每向下一层，运行栈就多一层（实线分割为一层）。

调用序列☆

过程调用和**过程返回**：（考过概念辨析）

- 过程调用序列：过程调用时执行的分配活动记录、把信息填入它的域中的代码。
- 过程返回序列：过程返回时执行的恢复机器状态、释放活动记录、使调用过程能够继续执行的代码。
- 调用序列和返回序列常常都分成两部分，**分处于调用过程和被调用过程中。**（也就是说，过程调用序列和过程返回序列**每个都是有两段代码。错误**描述：“过程调用序列通常处于调用过程中，而过程返回序列通常处于被调用过程中”）



栈上可变长度数据

【非重点】

局部数组的大小要等到过程激活时才能确定。

悬空引用

悬空引用：引用某个已经被释放的存储单元

e.g.

```
int * dangle() {
    int j = 20;
    return &j; // j的存储单元在函数结束后被释放了
}

int main() {
    int *q;
    q = dangle();
}
```

非局部名字的访问

- 无过程嵌套的静态作用域（C语言）
- 有过程嵌套的静态作用域（Pascal语言）

C语言属于无过程嵌套的语言，不需要访问链。

参数传递

- 值调用：传右值（对形参的任何操作不会影响调用者的实参的值）
  e.g.

```c
void process(int a, int b) {
    a = a + 2;
    b = b + 3;
}
int main() {
    int x = 5;
    int y = 8;
    process(x, y);
    printf("%d %d\n", x, y);
    /* 此时x和y的打印结果应为5和8，因为C语言采用值调用，
    被调用函数中的操作对此处的结果没有影响。*/
    return 0;
}
```

- 引用调用（地址调用）：传递实参的左值（对形参的任何赋值都会影响调用者的实参）
  在被调用过程的目标代码中，任何对形参的引用都是通过传给该过程的指针来间接引用实参的。
  数组常采用引用调用。
- 换名调用
  e.g.

```
procedure swap(var x, y: integer);
  var temp: integer;
    begin
      temp := x;
      x := y;
      y := temp
    end
```

当调用swap(i, a[i])时，直接“换名”（用文字替代）：

```
temp := i;
i := a[i];
a[i] := temp
```

从这里可以看出换名调用和其他方式的重要区别。第2行引用的a[i]和第3行被赋值的a[i]可能是不同的数据单元，因为i的值在第2行可能被改变了。

习题

1. 假定使用：（a）值调用、（b）引用调用、（c）换名调用，下面的程序打印的结果是什么？

```
program main(input, output);
var a, b: integer;
procedure p(x, y, z: integer);
    begin
        y := y + 1;
        z := z + x;
    end;
begin
    a := 2;
    b := 3;
    p(a + b, a, a);
    print a;
end.
```

答:

值调用时，传入函数p中的为值，因此a原来的值不改变，打印结果为a=2;

引用调用时，传入的是地址，a和b会随函数体内容改变，因此结果为：y=3, z=z+x=5+3=8, a=8；（执行p时对y和z的操作都相当于对a的操作）（**把x: a+b看作一个整体**）

换名调用时，y=a“=”a+1=3, z=z+x=a+a+b=3+3+3=9。

2. C语言函数f的定义如下：

```
int f(int x, int * py, int * * ppz) {
    * * ppz += 1; // 语句1
    * py += 2; // 语句2
    x += 3; // 语句3
    return x + * py + * * ppz;
}
```

变量a是指向b的指针，变量b是指向c的指针，c是整型变量并且当前值是4。那么执行f(c, b, a)的返回值是多少?

答:

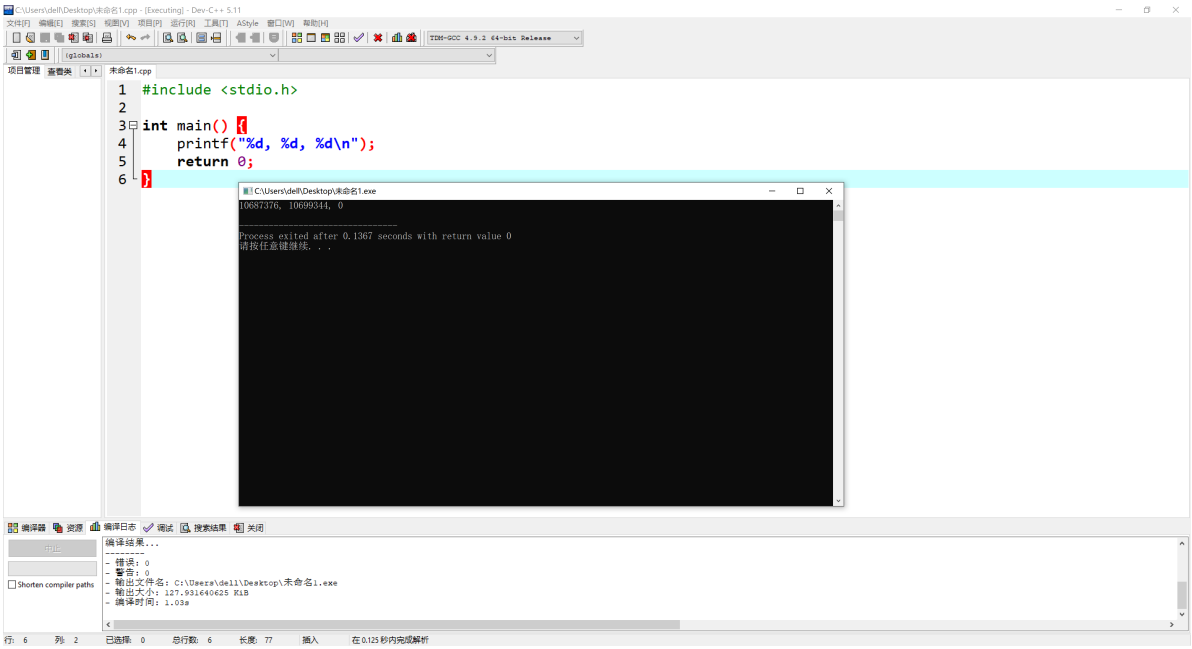
执行语句1后，c=*b=**a=5；执行语句2后，c=*b=**a=7；执行语句3后，x=7，而c=*b=**a=7。最终返回的是7+7+7=21。

3. （✩往年考过）下面的C语言程序中，函数printf的调用仅含格式控制字符串一个参数，该程序在x86/Linux系统上，经某编译器编译后，运行时输出三个整数。试从运行环境和printf的实现来分析，为什么此程序会有三个整数输出。

```
int main() {
    printf("%d, %d, %d\n");
    return 0;
}
```

答:

C语言编译器不做实参、形参个数、类型一致性检查，因此printf函数并不知道究竟调用者提供了多少个参数，它只根据第一个参数格式控制串的要求取参数，而C语言编译器的实现保证了被调用函数能准确取到第一个实参。需要取其他参数的时候编译器会按第一个参数的要求在栈上按顺序取得，而不管该位置是否是真正的参数。



【附往年考试真题】下面这段C语言程序，在gcc下编译运行的结果是什么？

```
#include <stdio.h>
void main() {
    printf("%d,%d,%d,%d\n");
}
```

答案：输出由逗号分隔的4个十进制数和换行。

第七章 中间代码生成

核心考点：给定程序，写出翻译方案（三地址码）。

中间语言

后缀表示

如果E是形式为$E_1 \, op E_2$的表达式，那么E的后缀表示是$E_1' \, E_2' \, op$，其中E_1'和E_2'分别是E_1和E_2的后缀表示。

e.g.

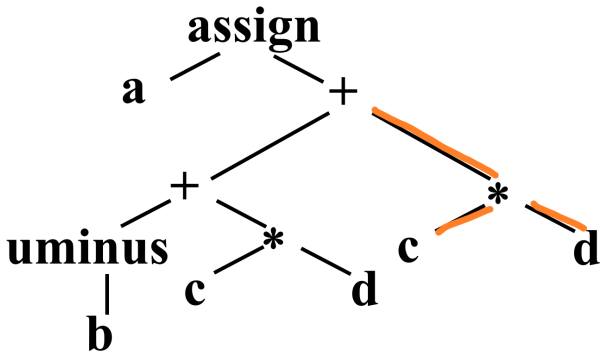
$8-4+2$的后缀表示是$8 \; 4 - 2+$，而$8-(4+2)$的后缀表示是$8 \; 4 \; 2 + -$.

对于单目运算符：$op \; E \rightarrow$后缀表达式$\rightarrow E_{\text{后缀表示}} \; op$.

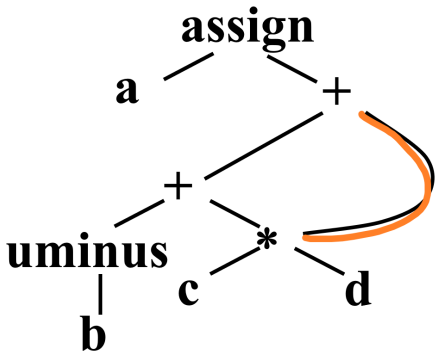
后缀表示不需要括号。后缀表示便于计算机处理。

图形表示（语法树、有向无环图DAG）

e.g. a=(-b+c*d)+c*d的图形表示：



(a) 语法树



(b) DAG

只需后序遍历抽象语法树，即可得到后缀表示。

构造赋值语句语法树的语法制导定义：

产 生 式	语 义 规 则
$S \rightarrow \text{id} = E$	$S.nptr = mknode(\text{'assign'}, mkleaf(\text{id}, \text{id}.entry), E.nptr)$
$E \rightarrow E_1 + E_2$	$E.nptr = mknode(\text{'+'}, E_1.nptr, E_2.nptr)$
$E \rightarrow E_1 * E_2$	$E.nptr = mknode(\text{'*'}, E_1.nptr, E_2.nptr)$
$E \rightarrow -E_1$	$E.nptr = mkunode(\text{'uminus'}, E_1.nptr)$
$E \rightarrow (E_1)$	$E.nptr = E_1.nptr$
$E \rightarrow \text{id}$	$E.nptr = mkleaf(\text{id}, \text{id}.entry)$

第一行构建的是树中的根节点：assign。

uminus: 取相反数。

三地址代码

一般形式：$x = y\ op\ z.$

- x, y, z表示名字、常数或临时变量。
- op表示运算符。

一条三地址码只有一个运算符。

e.g. 表达式$x + y * z$翻译成的三地址语句序列是：

$t_1 = y * z$

$t_2 = x + t_1$

（使用临时变量t_1等保存中间结果）

三地址代码是语法树或dag的一种线性表示

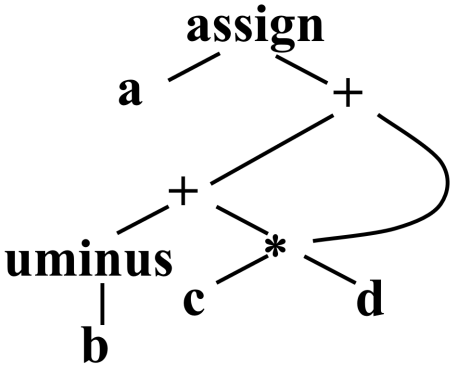
$$a = (-b + c*d) + c*d$$

语法树的代码

$$t_1 = -b$$
$$t_2 = c * d$$
$$t_3 = t_1 + t_2$$
$$t_4 = c * d$$
$$t_5 = t_3 + t_4$$
$$a = t_5$$

dag的代码

$$t_1 = -b$$
$$t_2 = c * d$$
$$t_3 = t_1 + t_2$$
$$t_4 = t_3 + t_2$$
$$a = t_4$$



常用的三地址语句：（要记，考试会考，但卷面上不给）

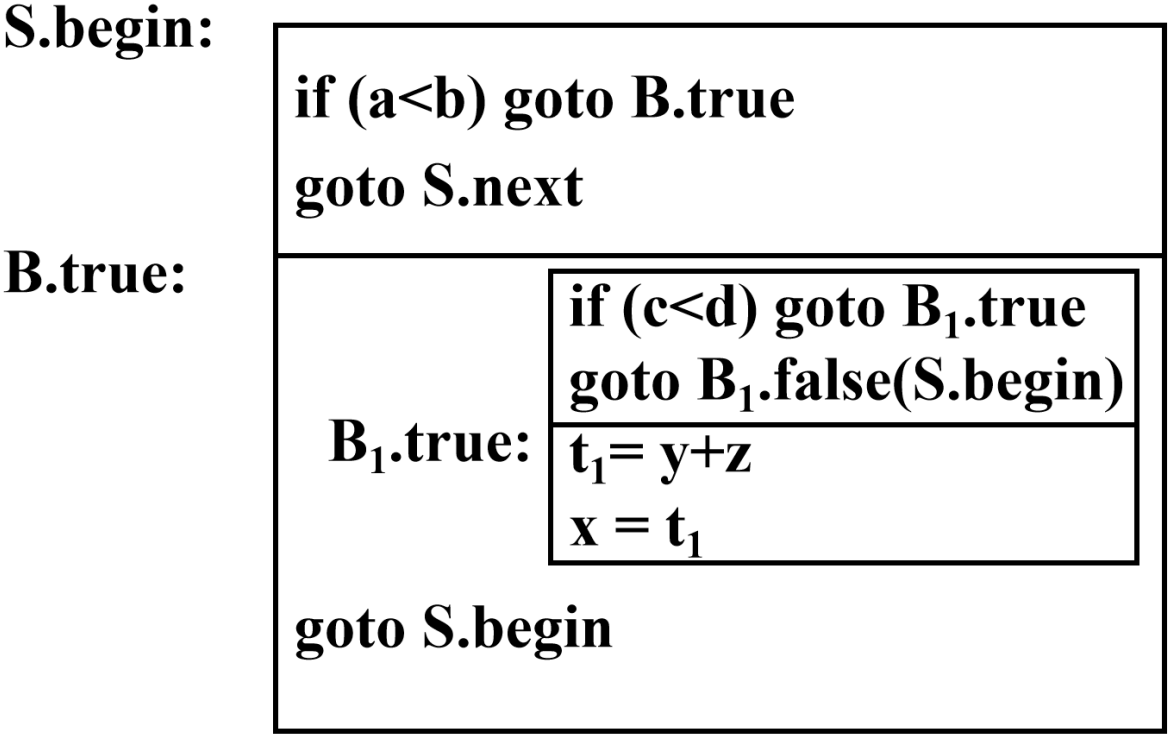
- 赋值语句 $x = y\ op\ z$，　$x = op\ y$，　$x = y$
- 无条件转移 goto L
- 条件转移 if x *relop* y goto L
- 过程调用 **param x** 和 call p , n　n表示实参个数
  指定x为过程调用的参数
- 过程返回 return y
- 索引赋值 $x = y[i]$ 和 $x[i] = y$
- 地址和指针赋值 $x = \&y$，$x = *y$ 和 $*x = y$

三地址指令是中间代码的抽象表示。在编译器中，三地址指令可以用记录实现。

【习题】写出下列程序的三地址代码：

```
while (a < b) {
    if (c < d) x = y + z;
}
```

答：



（上面的分析图考试时可以不画）

```
S.begin:   if (a<b) goto B.true
           goto S.next
B.true:    if (c<d) goto B1.true
           goto S.begin
B1.true:   t1 =y+z
           x = t1
           goto S.begin
```

S.next:

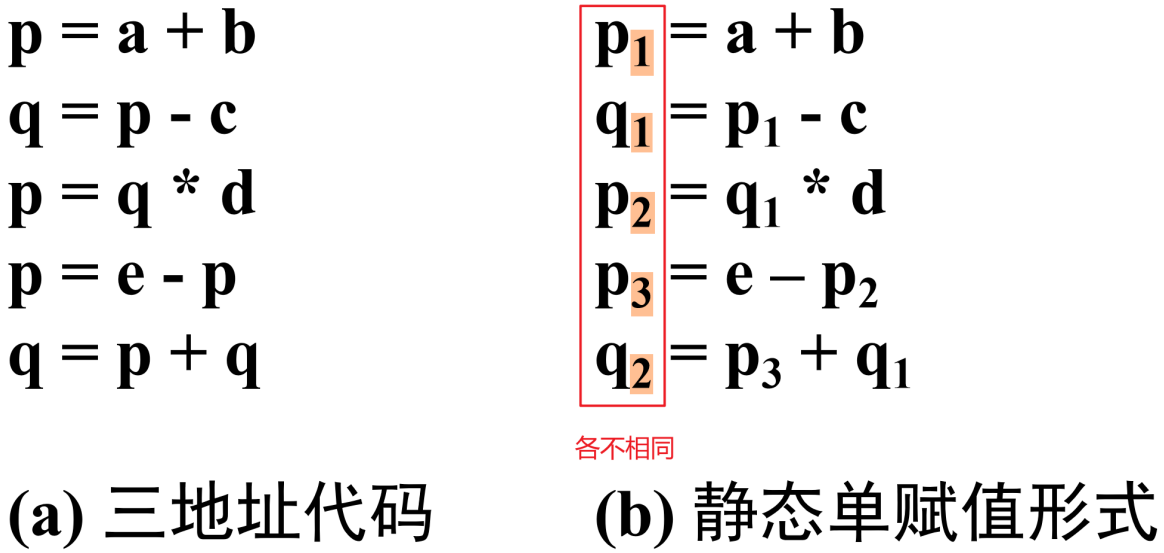
最好将其中的S.begin, B.true, B1.true, S.next替换成L1, L2, L3, Lnext。

静态单赋值形式(SSA)

两个特点：

- SSA中所有复制指令都是对**不同变量**的赋值。
- Φ函数：用于解决分支语句。（对于if语句来说，程序走“then”分支和走“else”分支时，变量名不同→用Φ函数解决）

第一个特点：



第二个特点：

e.g.

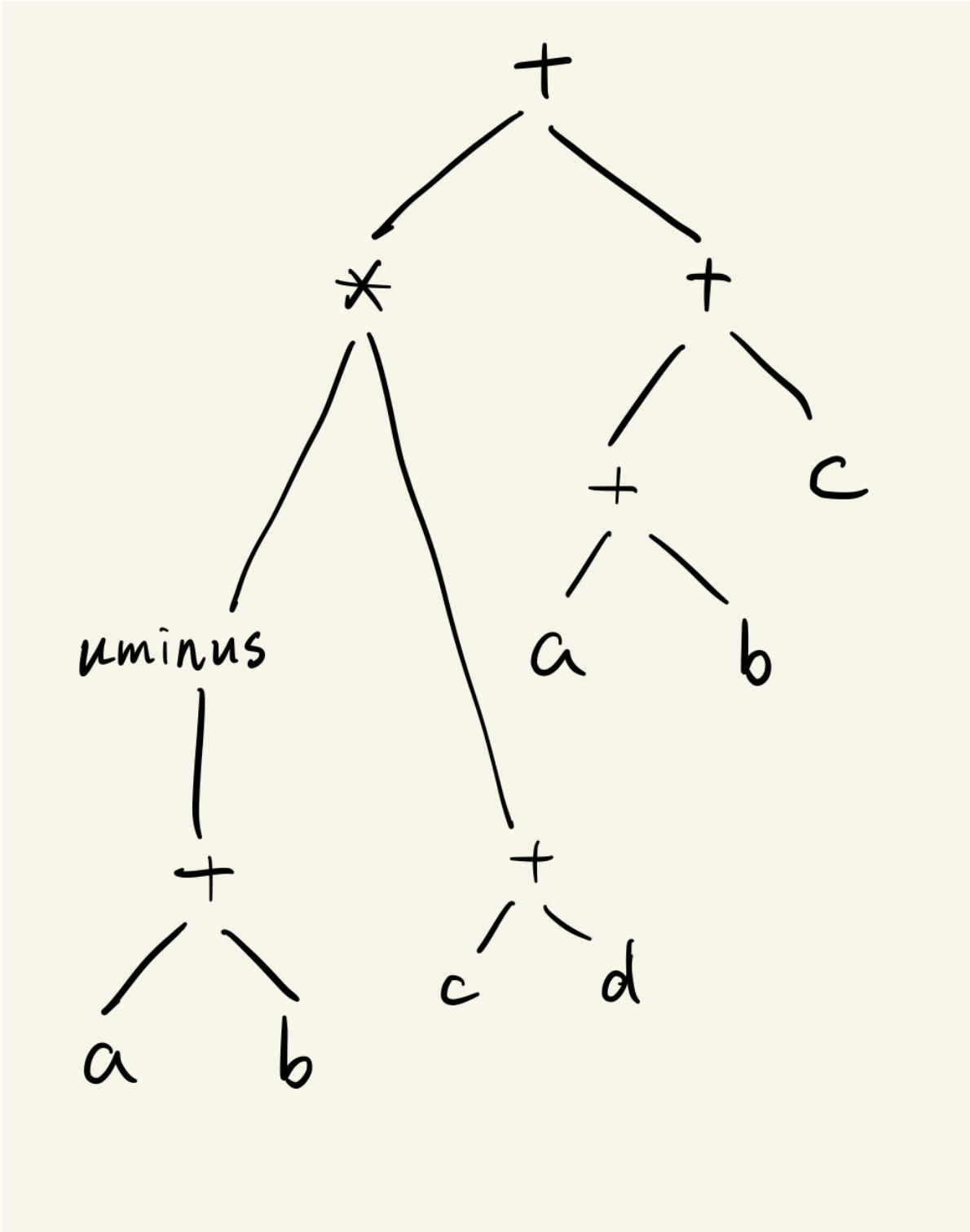
```
if (flag) x1 = -1; else x2 = 1;
x3 = Φ(x1, x2);
```

如果控制流通过条件为真的部分，则$\Phi(x_1,x_2)$的值是x_1，如果控制流通过条件为假的部分，则$\Phi(x_1,x_2)$的值是x_2。

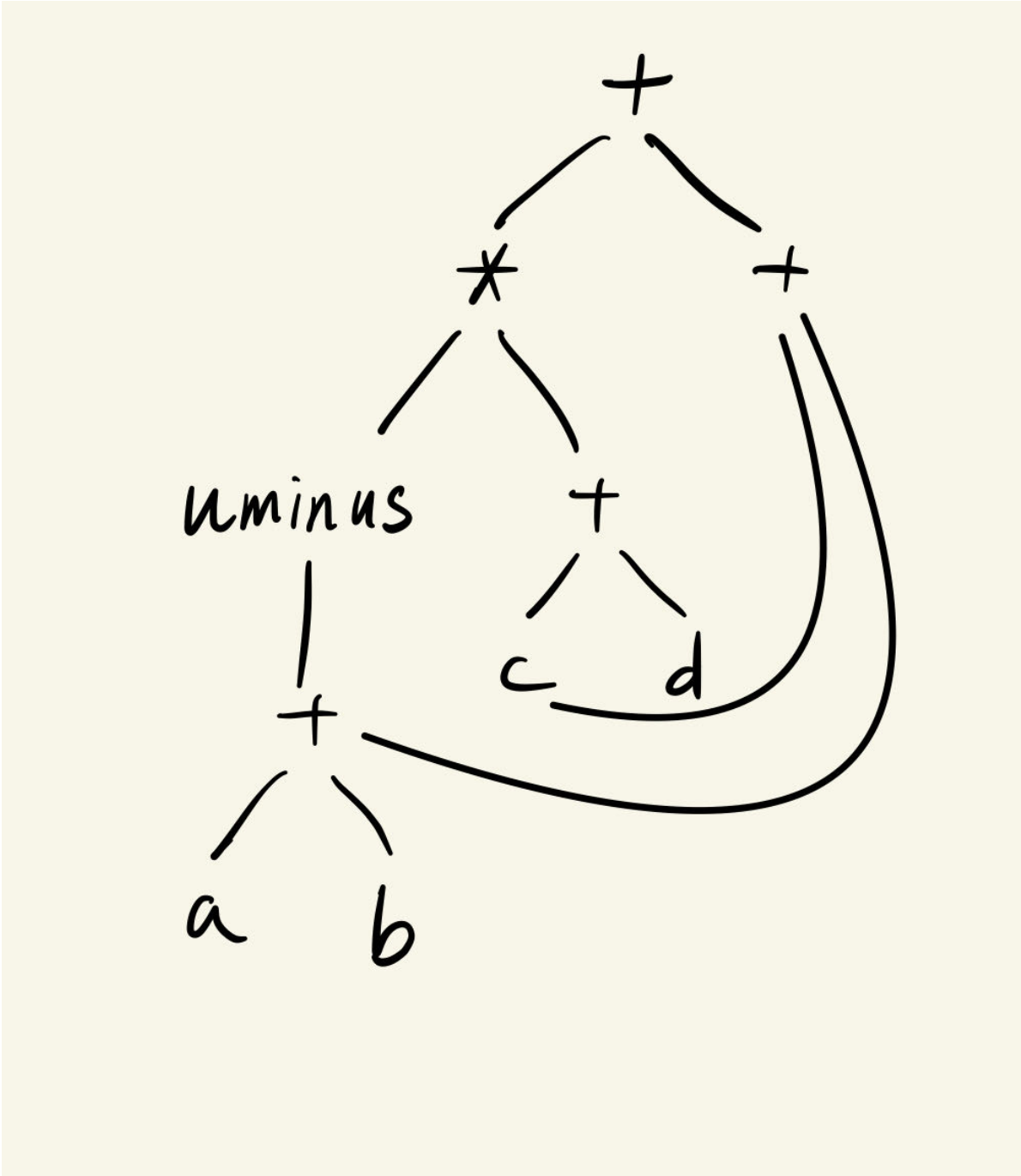
$\Phi(x_1,x_2)$返回它某个变元的值，取决于到达$\Phi(x_1,x_2)$时所通过的控制流路径。

【习题】把算术表达式$-(a+b)*(c+d)+(a+b+c)$翻译成：

- 语法树：



- 有向无环图：



- 后缀表示：a b + uminus c d + * a b + c + +
- 三地址代码：（两种写法，分别根据语法树和有向无环图写）

①

t1=a+b

t2=-t1

t3=c+d

t4=t2*t3

t5=a+b

t6=t5+c

t7=t4+t6

②

t1=a+b

t2=-t1

t3=c+d

t4=t2*t3

t5=t1+c

t6=t4+t5

声明语句

在分析过程或程序块的声明序列时，为局部名字建立符号表条目，并为它分配存储单元。（分配一个“偏移量”）

这样，符号表包含各名字的**类型**和分配给它们的存储单元的**相对地址**等信息。（相对地址是对静态数据区基址的偏移或是对活动记录中某个基址的偏移）

这部分的讨论中忽略数据对象的对齐等问题。

过程中的声明

offset的初值可能不为0。

$P \rightarrow$ $\{offset = 0\}$ offset：偏移（全局变量）

$D\ S$

$D \rightarrow D\ ;\ D$

$D \rightarrow \text{id} : T$ 为名字name建立符号表条目，该名字的类型是type，它在数据区的相对地址是offset。

$\{enter(\text{id}.name, T.type, offset);$
$offset = offset + T.width\}$

$T \rightarrow \text{integer}$ $\{T.type = integer;\ T.width = 4\}$

$T \rightarrow \text{real}$ $\{T.type = real;\ T.width = 8\}$

$T \rightarrow \text{array}[\text{num}]\ \text{of}\ T_1$
$\{T.type = array(\text{num}.val, T_1.type);$
$T.width = \text{num}.val \times T_1.width\}$

四种类型：整型、实型、数组、指针

$T \rightarrow \uparrow T_1$ $\{T.type = pointer(T_1.type);\ T.width = 4\}$

S: 可执行语句

P → D S: 语言的程序→声明+语句

花括号中涉及的对象：

- offset
- X.type
- X.width
- enter(name, type, offset)
- array()
- pointer()

赋值语句

符号表中的名字

介绍中间语言时，为直观，让名字直接出现在三地址码中，实际上应该**把名字理解为它们在符号表中位置的指针。**

编译器在处理名字时，需要在符号表中查找其定义，获得其属性，然后在生成的三地址码中使用它在符号表中位置的指针。

为赋值语句产生三地址代码的翻译方案：

$S \rightarrow \text{id} := E$ {*p = lookup*(id.*name*); if (*p* != *nil*) *emit* (*p*, '=', *E.place*); else *error*; }

$E \rightarrow E_1 + E_2$ {*E.place = newTemp*(); *emit* (*E.place*, '=', E_1*.place*, '+', E_2*.place*) }

$E \rightarrow -E_1$ {*E.place = newTemp*(); *emit* (*E.place*, '=', 'uminus', E_1*.place*); }

$E \rightarrow (E_1)$ {*E.place* = E_1*.place*; }

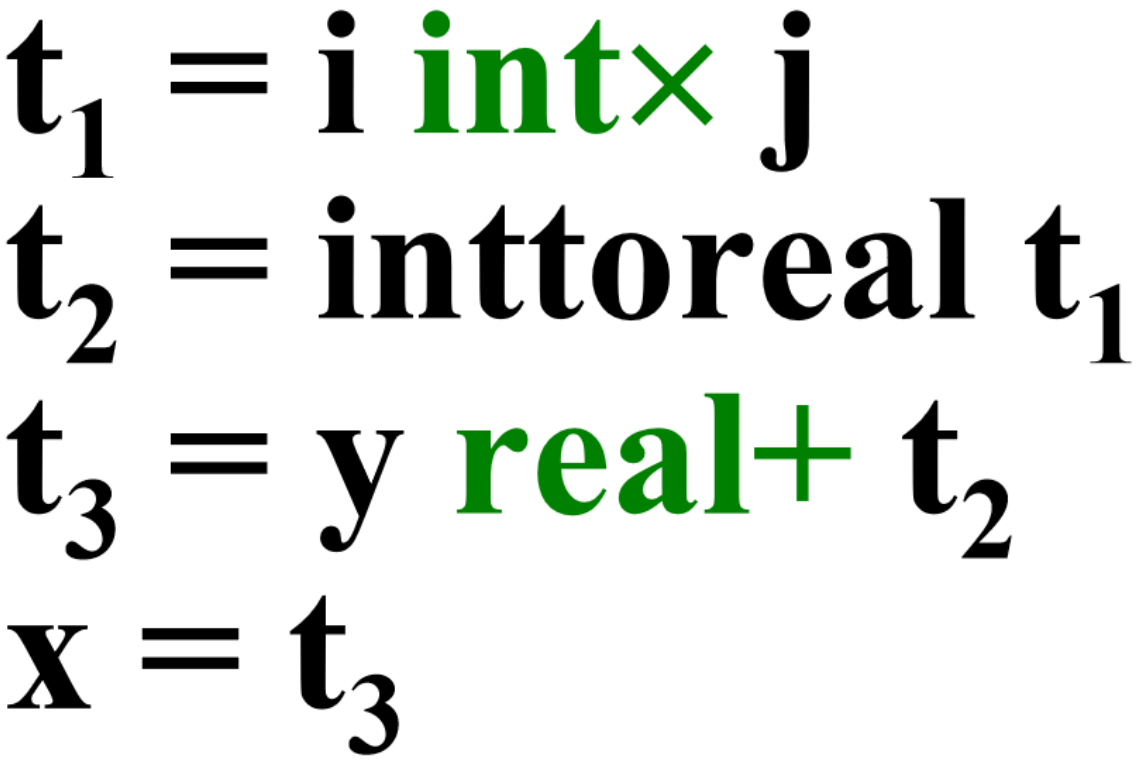
$E \rightarrow \text{id}$ {*p = lookup*(id.*name*); if (*p* != *nil*) *E.place = p*; else *error*; }

类型转换

e.g. 假定x和y的类型是real，i和j的类型是integer，对于输入：

$x = y + i * j$

输出的三地址指令序列是：



产生式$E \rightarrow E_1 + E_2$的完整语义动作：

```
E.place=newTemp();
if(E1.type==integer)&&(E2.type==integer)begin
    emit(E.place,'=',E1.place,'int+',E2.place);
    E.type=integer;
end
else if(E1.type==real)&&(E2.type==real)begin
    emit(E.place,'=',E1.place,'real+',E2.place);
    E.type=real;
end
else if(E1.type==integer)&&(E2.type==real)begin
    u=newTemp();
    emit(u,'=','inttoreal',E1.place);
    emit(E.place,'=',u,'real+',u);
    E.type=real;
end
else if(E1.type==real)&&(E2.type==integer)begin
    u=newTemp();
    emit(u,'=','inttoreal',E2.place);
    emit(E.place,'=',E1.place,'real+',u);
    E.type=real;
end
else
    E.type=type_error
```

对上述代码的解释：就是$E = E_1 + E_2$的右侧两个变量可以是整型或实型，共有四种组合，它们都是整型时或都是实型时可以直接相加，否则需要进行类型转换（int→real）后再相加。其他情况视为error。

布尔表达式和控制流语句

布尔表达式

布尔表达式的文法：

$B \rightarrow B\ or\ B \mid B\ and\ B \mid not\ B \mid (B) \mid E\ relop\ E \mid true \mid false$

优先级：or<and<not；or和and是左结合的，not是右结合的。

实现布尔表达式往往会使用**短路计算。**（短路计算：对于or语句，如果左侧对象为true就直接返回true；对于and语句，如果左侧对象为false就直接返回false）

短路计算有严格的语义规定，把$B_1\ or\ B_2$定义成：

$if\ B_1\ then\ true\ else\ B_2$

把$B_1\ and\ B_2$定义成：

$if\ B_1\ then\ B_2\ else\ false$

控制流语句的翻译

【考大题：if或while或序列语句】

考虑控制流语句由下列文法产生：其中B是布尔表达式

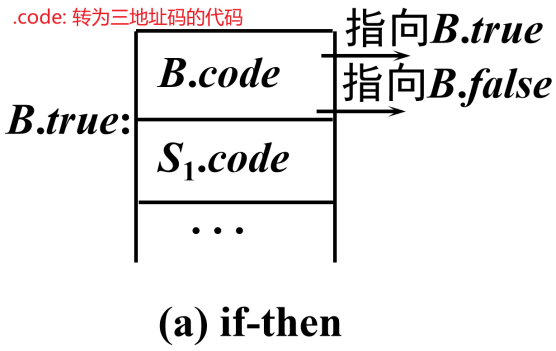
S → if B then S1

| if B then S1 else S2

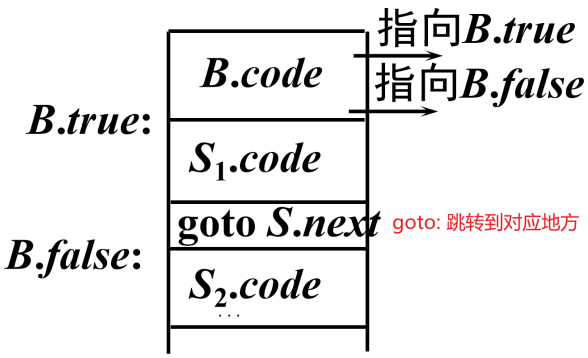
| while B do S1

| S1; S2

.code: 转为三地址码的代码

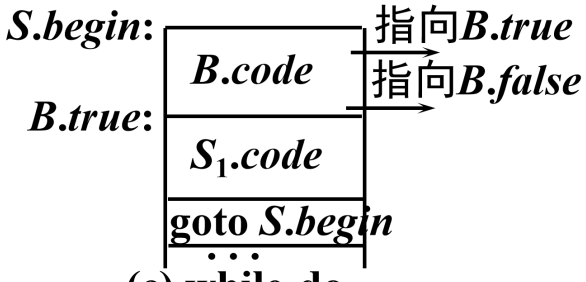


(a) if-then

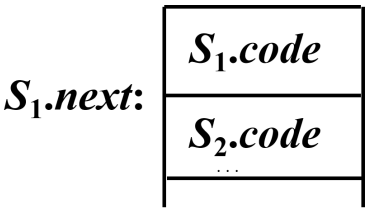


goto: 跳转到对应地方

(b) if-then-else



(c) while-do



(d) S_1; S_2

上图中，B.true, B.false, S.begin, S.next几个属性都表示三地址指令的标号，是继承属性。

B.code和S.code分别表示B和S的三地址指令序列，是综合属性。

定义翻译过程中用到的函数和符号：

- 函数newLabel()返回一个新的标号。

- 函数gen产生一个三地址指令或标号，并把这个三地址指令或标号的串值作为返回值。
- “||”是串连接符号。

表 7.2　控制流语句的语法制导定义　(对应着图 7.12看)

图7.12	产生式	语义规则
(a)	$S \rightarrow$ **if** B **then** S_1	$B.true = newLabel()$ $B.false = S.next$　　$S_1.next = S.next$ $S.code = B.code \parallel gen(B.true, ':') \parallel S_1.code$
(b)	$S \rightarrow$ **if** B **then** S_1 **else** S_2	$B.true = newLabel()$　$B.false = newLabel()$ $S_1.next = S.next$　　$S_2.next = S.next()$ $S.code = B.code \parallel gen(B.true, ':') \parallel S_1.code \parallel$ $gen('goto', S.next) \parallel gen(B.false, ':') \parallel S_2.code$
(c)	$S \rightarrow$ **while** B **do** S_1	$S.begin = newLabel()$　　$B.true = newLabel()$ $B.false = S.next$　　$S_1.next = S.begin()$ → 对于goto的情况，直接让等号右边等于goto的位置 $S.code = gen(S.begin, ':') \parallel B.code \parallel gen(B.true, ':') \parallel$ $S_1.code \parallel gen('goto', S.begin)$
(d)	$S \rightarrow S_1 ; S_2$	$S_1.next = newLabel()$　　$S_2.next = S.next()$ $S.code = S_1.code \parallel gen(S_1.next, ':') \parallel S_2.code$

布尔表达式的控制流翻译

假定B是a<b的形式，那么生成的代码形式为：

```
if a<b goto B.true
goto B.false
```

这里也使用了“短路计算”的思想。

$$B \rightarrow B_1 \text{ or } B_2$$

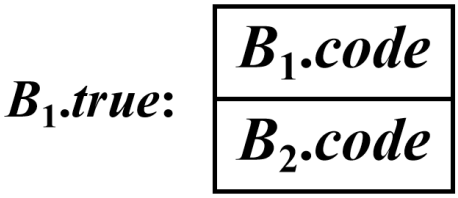
$B_1.false$:

$B_1.code$
$B_2.code$

语义规则：

$$B_1.true = B.true;$$
$$B_1.false = newLabel();$$
$$B_2.true = B.true;$$
$$B_2.false = B.false;$$
$$B.code = B_1.code \parallel gen(B_1.false, ':') \parallel B_2.code$$

$B \rightarrow B_1$ and B_2



语义规则：

$B_1.true = newLabel()$;
$B_1.false = B.false$;
$B_2.true = B.true$;
$B_2.false = B.false$;
$B.code = B_1.code \,||\, gen(B_1.true, ':') \,||\, B_2.code$

$B \rightarrow \text{not } B_1$

语义规则：

$B_1.true = B.false$;
$B_1.false = B.true$;
$B.code = B_1.code$

$$B \rightarrow (B_1)$$

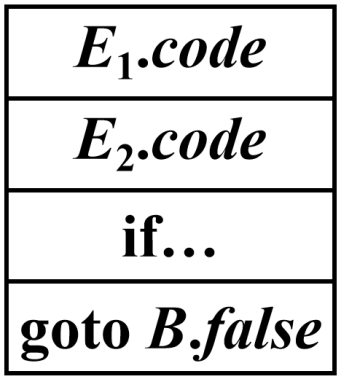
语义规则：

$$B_1.true = B.true;$$

$$B_1.false = B.false;$$

$$B.code = B_1.code$$

$B \rightarrow E_1$ relop E_2



语义规则：

$B.code = E_1.code \,||\, E_2.code \,||$
$gen(\text{‘if’}, E_1.place, \text{relop}.op, E_2.place,$
$\text{‘goto’}, B.true) \,||$
$gen(\text{‘goto’}, B.false)$

$B \rightarrow$ true

语义规则：

$B.code = gen(\text{‘goto’}, B.true)$

$B \rightarrow$ false

语义规则：

$B.code = gen(\text{‘goto’}, B.false)$

e.g. 下面的布尔表达式是控制流中的条件，请按前面的语法制导定义将其翻译成三地址码。

$a < b \ or \ c < d \ and \ e < f$

优先级：先算and，后算or。

<table>
<tr><td colspan="2">if (a<b) goto B1.true(B.true)
goto B1.false</td></tr>
<tr><td>B1.false:</td><td>
<table>
<tr><td>if (c<d) goto B3.true
goto B3.false(B2.false,B.false)</td></tr>
<tr><td>B3.true:</td></tr>
<tr><td>if (e<f) goto B4.true
(B2.true, B.true)
goto B4.false(B2.false,B.false)</td></tr>
</table>
</td></tr>
</table>

答案：

if (*a* < *b*) goto B.true
goto B1.false
B1.false: if (*c* < *d*) goto B3.true
goto B.false
B3.true: if (*e* < *f*) goto B.true
goto B.false

可以将上面的B, B1, B3换成L_i的形式；goto B1.false多余，可以删去。

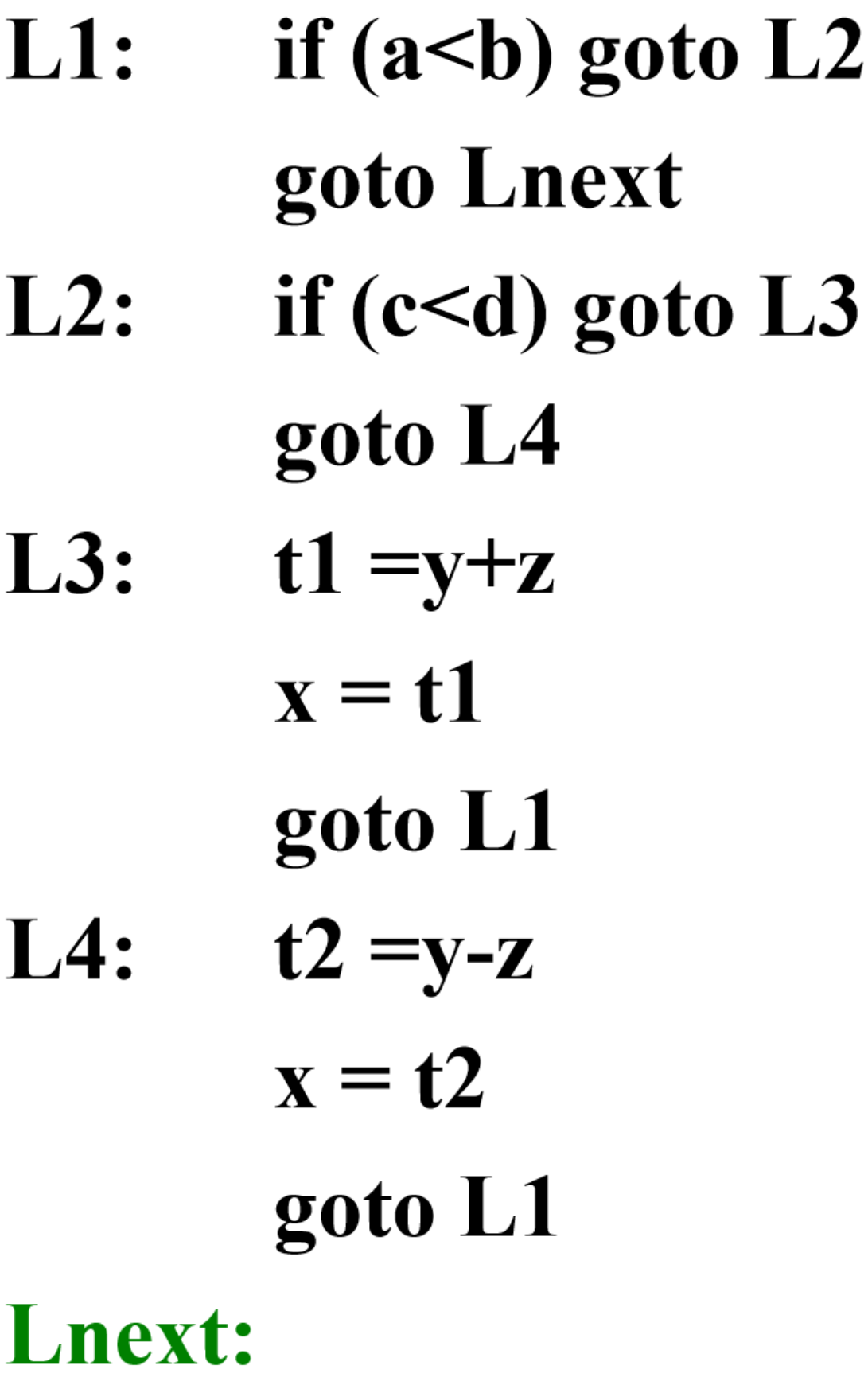
假转方式：a<b翻译成：

$if\ a \leq b\ goto\ B.\ false$

例：为下列语句产生三地址码

```
while (a < b) {
  if (c < d) x = y + z;
  else x = y – z;
}
```

答案：



开关语句的翻译

switch-case（了解，非考察核心）

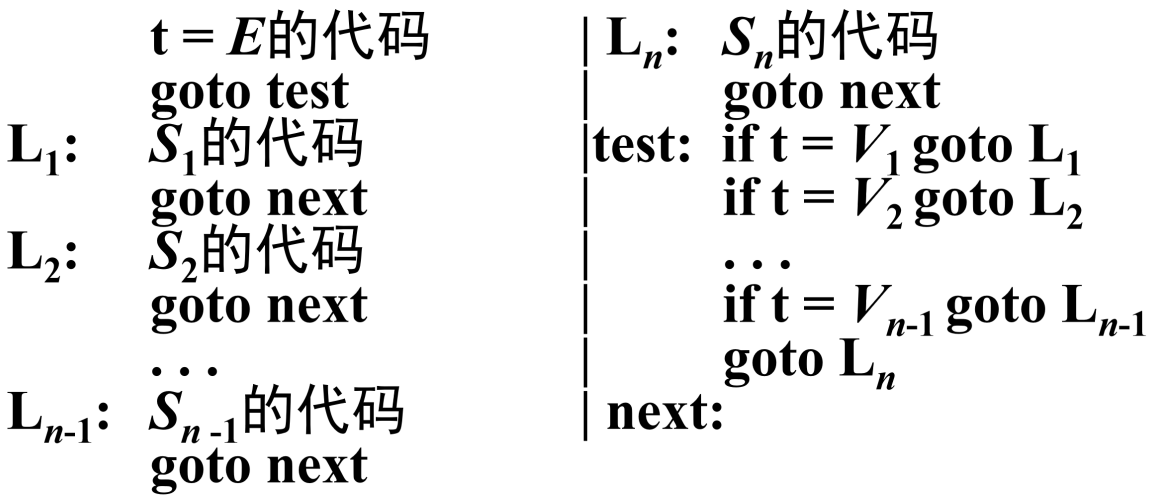
```
switch E
    begin
        case V1: S1
        case V2: S2
        . . .
        case Vn-1: Sn-1
        default: Sn
    end
```

执行流程：计算E，分支测试（常量匹配），执行匹配的分支语句。

分支数较少时这样翻译：（分支数小于10）

```
        t = E的代码
        if t ≠ V1 goto L1
        S1的代码
        goto next
L1:     if t ≠ V2 goto L2
        S2的代码
        goto next
L2:     . . .
        . . .
Ln-2:   if t ≠ Vn-1 goto Ln-1
        Sn-1的代码
        goto next
Ln-1:   Sn的代码
next:
```

分支数较多时，把分支测试的代码集中在test中：



也可以对中间代码增加一种case语句，便于代码生成器对它进行特别处理。

过程调用的翻译

大概率考试中不涉及

日常实验（Lex和Yacc相关）

Lex: 词法分析器驱动程序yylex()

Yacc如何解决二义文法的冲突：

在Yacc中，可以使用%left, %right和%nonassoc指令来声明运算符的优先级和结合性。这些指令可以帮助Yacc生成正确的分析表，并且确保分析表中的操作顺序满足指定的优先级和结合性规则。

y.tab.c包含yyparse()和LALR分析表。

Yacc把错误产生式当作普通产生式处理。

如果不想让Yacc用最右终结符来决定产生式的优先级和结合性，则使用$prec强制指定该产生式和UMINUS的优先级和结合性一致。

【刷题有感】

Lex是一个词法分析器的生成工具。

YACC是一个语法分析器的生成工具。

【日常练习】

Lex根据Lex源程序生成的词法分析器源程序中，必须包含的两部分内容是：

- 根据Lex源程序中正规式构造的DFA状态转换表。
- 词法分析器驱动程序yylex()。

Lex生成的词法分析器使用什么原则匹配被选中的词法单元?

最长匹配原则。如果词法单元可与若干正规式匹配，则优先选择排在前面的正规式。

Lex源程序中有两条正规式：A可以匹配所有标识符（以字母开头的字母数字组合），B可以匹配While关键字。如果想让输入文件中的While被识别为While关键字，则在Lex源程序中：要把B放在A前面。

语法分析器在使用Lex生成的词法分析器时，每次调用yylex()获取一个词法记号，如果词法分析器的输入文件并未扫描到结尾，则关于这个过程正确的说法是：每次调用yylex()后，输入缓冲区不会被重置。每次调用yylex()后，词法分析器当前所属状态不变。