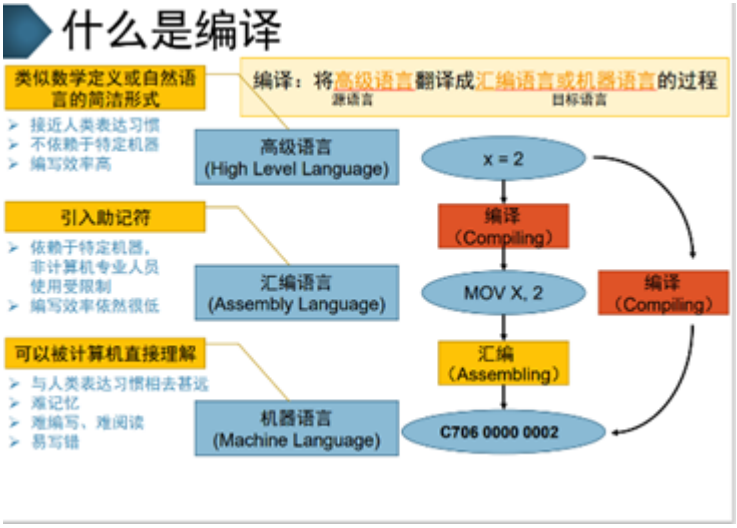
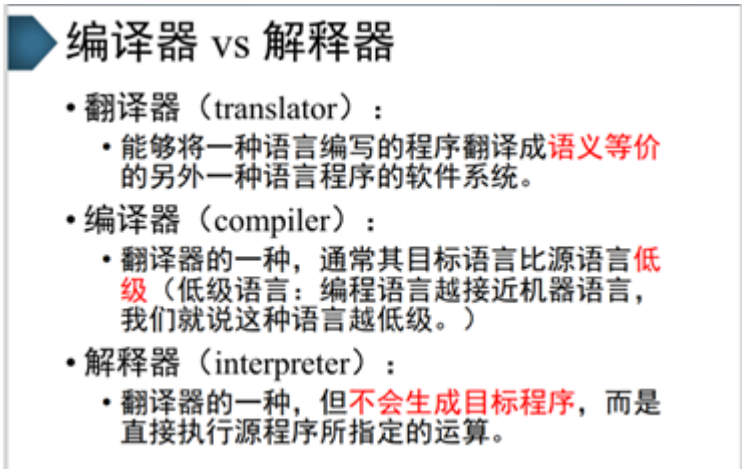
第一章



翻译器：将一种语言保语义变换到另一种语言

解释器：直接执行源程序所指定的运算。每次执行到源程序的某个语句，都要对它进行依次词法分析、语法分析和语义分析，再执行指定运算。所以解释执行的效率比编译器生成的机器代码的执行效率低



交叉编译：在当前编译平台下，编译出来的程序能运行在体系结构不同的另一种目标平台上，但是编译平台本身却不能运行该程序

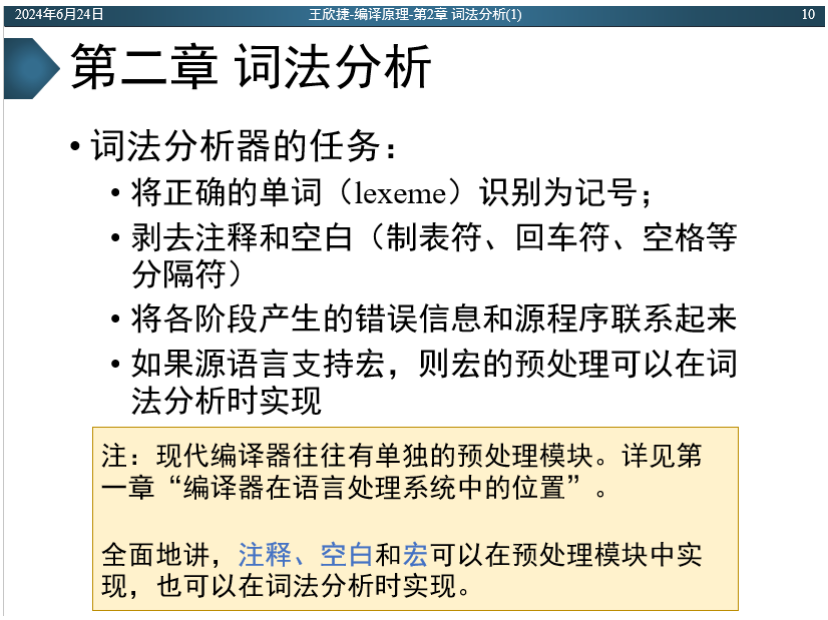
第二章

概念：

- 词法记号（**token**）：<记号名，属性值>
- 字母表：符号的有限集合
- 串：字母表符号的有穷序列
- 语言：字母表上的一个串集
- 句子或字：属于该语言的串

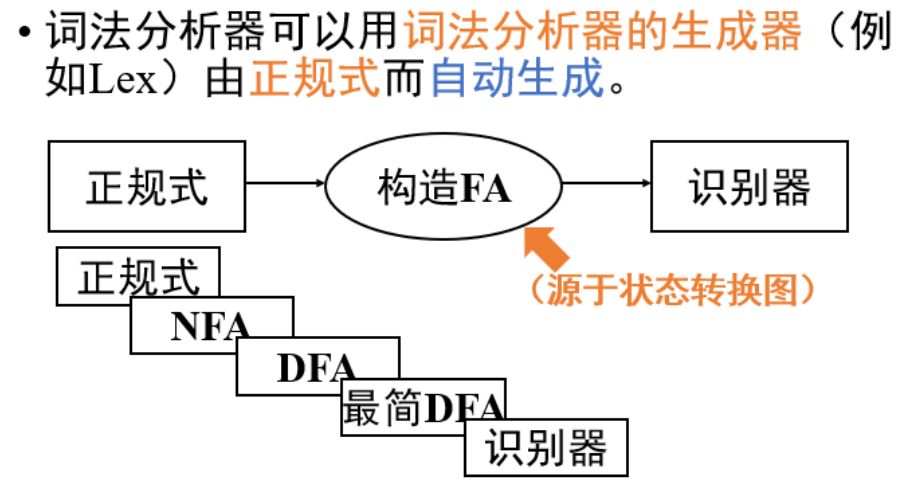
词法分析的错误恢复

- 紧急方式恢复：删掉输入指针当前指向的若干个字符，直到词法分析器能发现一个正确的记号为止
- 错误修补尝试：看剩余输入的前缀能否通过删除、插入、替换或交换这四个变换方式变成一个合法的词法单元。



正规式表示的语言称为正规语言或正规集

运算优先级：闭包>连接>选择



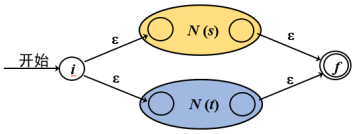
“ 不确定”的含义是：存在这样的状态，对于某个输入符号，它存在不止一种转换

NFA	DFA
• 有限的状态集合S • 输入符号集合Σ（输入符号字母表，决不包含ε） • 转换函数$move: S \times (\Sigma \cup \{\varepsilon\}) \to P(S)$（$S$的幂集） ∀s∈S,a∈Σ∪{ε},$move(s,a)$表示从状态s出发，沿着标记为a的边所能到达的状态集合 • 状态s_0是唯一开始状态 • $F \subseteq S$是接受状态（终态）集合	• 有限的状态集合S • 输入符号集合Σ（输入符号字母表，决不包含ε） • 转换函数$move: S \times \Sigma \to S$。∀s∈S,a∈Σ,$move$(s,a)表示从状态s出发，沿着标记为a的边所能到达的状态 • 状态s_0是唯一开始状态 • $F \subseteq S$是接受状态（终态）集合

正规式→NFA（Thompson算法）

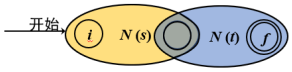
记住三种运算对应的画法，记得空转换，开始和结束。

• 构造识别主算符为“或”的正规式的NFA



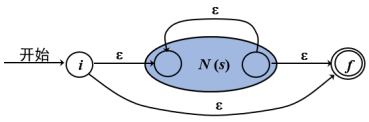
识别正规式$s \mid t$的NFA

• 构造识别主算符为连接的正规式的NFA



识别正规式st的NFA

• 构造识别主算符为闭包的正规式的NFA



识别正规式s^*的NFA

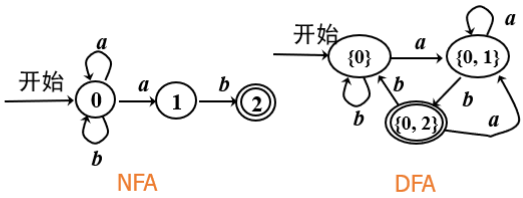
NFA→DFA（子集构造法）

求解的时候画表比较方便，一定要看清楚状态，别漏掉，求闭包时可以做好标记

构造得到的DFA不一定最简

- 从NFA转换到DFA：子集构造法(Subset Construction)
- 思路：
  - DFA的一个状态是NFA的一个状态集合
  - 读了输入$a_1\ a_2\ \dots\ a_n$后，
    NFA能到达的所有状态：$s_1, s_2, \dots, s_k$，则DFA到达状态$\{s_1, s_2, \dots, s_k\}$
  - 例1：

状态	输入符号	
	a	b
0	{0, 1}	{0}
1	∅	{2}
2	∅	∅



2是接受状态

首先初始状态为s_0，找到其通过空转换能到达的所有状态的集合A，即**A=ε-closure(s_0)**

再看A中所有的状态输入非空字符后(比如a)能到达的所有状态move(A,a)，再求得下一个状态为**B=ε-closure(move(A,a))**

DFA→最简DFA

如果转换函数并非全函数，可以引入死状态（“dead” state）变换成全函数。这个在上一步画表时就可引入，当结果为空时，设为死状态

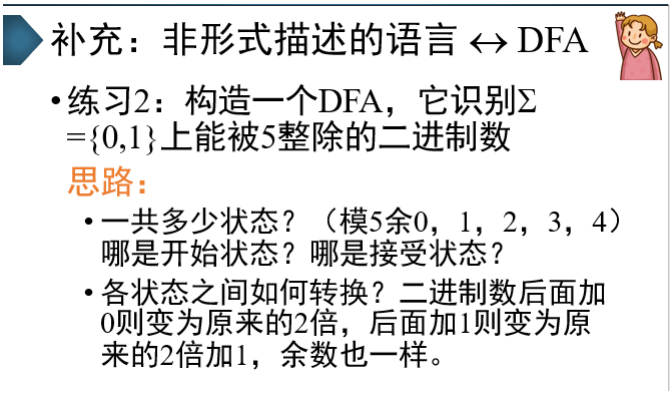
可区别状态：两个状态输入同一个串后，一个停在可接受状态，另一个停在不可接受状态

化简方法：

- 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，若一个子集里有可区别状态，则继续划分子集。直到没有任何一个子集可划分为止

手工构造DFA

重点是理解其中的状态变化



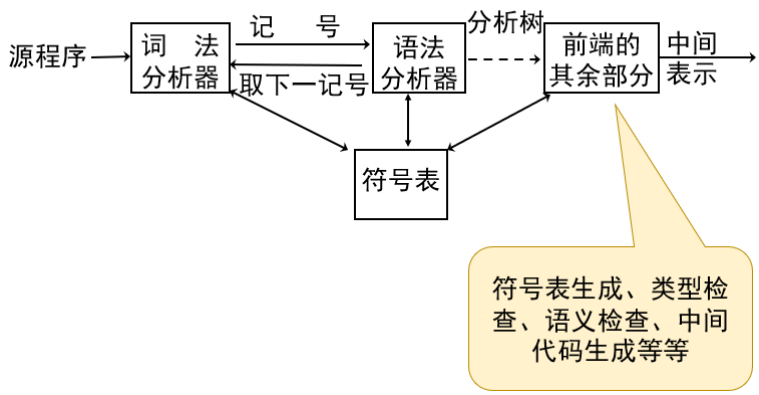
DFA→语言识别器

补充：从DFA构造语言识别器

- *move*(*s*,*c*)：返回状态*s*在面临输入*c*时转换到的状态
- *nextChar*()：返回输入串*x*中的下一个字符

```
s=s0;
c=nextChar();
while(c!=eof) {
   if (move(s,c)未定义)
      return "no";
   else s=move(s,c);
  c=nextChar();
}
if (s∈F)
  return "yes";
else return "no";
```

第三章 语法分析



上下文无关文法（CFG）

Chomsky的文法分类： **0-3**型文法（短语文法0 ⊃上下文有关文法1 ⊃上下文无关文法2 ⊃正规文法3）

- 四种文法之间的关系：逐级限制
  - **0**型文法：α中至少包含**1**个非终结符
  - **1**型文法（*CSG*）：$|\alpha| \leq |\beta|$
  - **2**型文法（*CFG*）：$\alpha \in V_N$
  - **3**型文法（*RG*）：$A \rightarrow wB$或$A \rightarrow w$($A \rightarrow Bw$或$A \rightarrow w$)
- 逐级包含

第三章 语法分析

- 上下文无关文法G是四元组（V_T, V_N, S, P）
  - V_T：　终结符集合（非空，有限）
  - V_N：　非终结符集合（非空，有限）
    $V_T \cap V_N = \varnothing$
  - S：　开始符号
  - P：　产生式集合，产生式形式 ：$A \to \alpha$
    其中$A \in V_N$, $\alpha \in (V_T \cup V_N)^*$,
    S至少要出现在某产生式左部。
- 经过一些约定，上下文无关文法可仅用产生式集合表示。
- 一个经典的上下文无关文法的例子：算术表达式
  - $E \to E+E \mid E*E \mid (E) \mid \mathrm{id}$

1.推导

- $E \to E+E \mid E*E \mid (E) \mid -E \mid \mathrm{id}$
- 注意→和⇒的区别， A→ α是文法的产生式，
  E⇒E+E读作“E推导出E+E”

对任何串，有$\alpha \Rightarrow^* \alpha$;
如果$\alpha \Rightarrow^* \beta, \beta \Rightarrow \gamma$, 那么$\alpha \Rightarrow^* \gamma$

- 一些概念和记号：
  - $A \to \gamma$, 则$\alpha A \beta \Rightarrow \alpha\gamma\beta$
  - $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$：　α_1 推导出 α_n
  - ⇒：一步推导
  - ⇒*：零步或多步推导
  - $\Rightarrow^+$ ：一步或多步推导

- 由文法G产生的全部句子组成的集合称为G产生的语言，记做L(G)。
- 由上下文无关文法产生的语言叫做上下文无关语言（context-free language）。
- 如果S⇒a， a可能含有非终结符，则a叫做该语言的句型（sentential form）
- 句子（sentence）是只含终结符的句型。
- 如果两个文法产生同样的语言，则称这两个文法等价

- 例3.3： $E \to E+E \mid E*E \mid (E) \mid -E \mid \mathrm{id}$
- 最左推导（leftmost derivation）

$$E \Rightarrow_{lm} -E \Rightarrow_{lm} -(E) \Rightarrow_{lm} -(E+E)$$
$$\Rightarrow_{lm} -(\mathrm{id}+E) \Rightarrow_{lm} -(\mathrm{id}+\mathrm{id})$$

- 最右推导（rightmost derivation）

$$E \Rightarrow_{rm} -E \Rightarrow_{rm} -(E) \Rightarrow_{rm} -(E+E)$$
$$\Rightarrow_{rm} -(E+\mathrm{id}) \Rightarrow_{rm} -(\mathrm{id}+\mathrm{id})$$

- 最右推导又叫规范推导（canonical derivation）

2.分析树（推导的图形表示）

又叫语法分析树

二义文法：其某个句子有两棵分析树

二义语言：当产生一个语言的所有文法都是二义时，这个语言才称为二义的

构造非二义文法：用优先级和结合性，优先级低的产生优先级高的

3.消除二义性

思路：限制每一步推导都只有唯一的选择

- 解决优先级：引入新的非终结符，增加一个子结构并提高一级优先级，越靠近开始符号的文法符号优先级越低
- 解决结合性：对于递归产生式**E→E+a**，E出现在[+]左边，则产生式具有左结合性

4.消除左递归

直接左递归改写规则：

A → Aa | b

- A → bA'
- A' → aA' | ε

非直接左递归：

先代换，使之变为直接左递归，这里需要注意代换的顺序，要看每一个式子是否需要代换

5.提左因子

$$A \to \alpha\beta_1 \mid \alpha\beta_2$$

- 提左因子

$$A \to \alpha A'$$

$$A' \to \beta_1 \mid \beta_2$$

自上而下分析

LL(1) 属于预测分析法，不需要回溯，是一种确定的自顶向下分析方法

- 先定义两个和文法有关的函数
  - **FIRST** $(\alpha)=\{a \mid \alpha \Rightarrow^* a\ldots, a \in V_T\}$
    特别的，$\alpha \Rightarrow^* \varepsilon$时，规定$\varepsilon \in$ **FIRST** (α)
  - **FOLLOW** $(A) = \{a \mid S\$ \Rightarrow^* \ldots Aa\ldots, a \in V_T\}$
    特别的，如果A是某个句型的最右符号，那么\$属于**FOLLOW**(A)

FIRST

从后往前看比较好分析，一定要注意ε

- 计算**FIRST** (X)：重复下列算法直到无新的终结符或ε可以加入**FIRST** (X)
  - 若X是终结符，则**FIRST**$(X)=\{X\}$
  - 若X是非终结符，且$X \to Y_1 Y_2 \ldots Y_k \in P$，则对某个$i$，若$\varepsilon \in$**FIRST**$(Y_1)\ldots$**FIRST**$(Y_{i-1})$(即$Y_1 \ldots Y_{i-1} \Rightarrow^* \varepsilon$)，而$a \in$**FIRST**$(Y_i)$，则$a \in$**FIRST**$(X)$。若$\varepsilon \in$**FIRST**$(Y_1)\ldots$**FIRST**$(Y_k)$，则$\varepsilon \in$**FIRST**$(X)$
  - 若$X \to \varepsilon \in P$，则$\varepsilon \in$**FIRST**(X)

FOLLOW

这里不包含ε，需要循环多次计算直至没有变化

看产生式右边

- 计算**FOLLOW** (A)：A是非终结符。
  重复下列算法直到无新的符号可以加入**FOLLOW** (A)。
  - 如果A是开始符号，则\$$\in$**FOLLOW**（$A$），\$是输入右端的结束标记
  - 若有产生式$B \to \alpha A \beta$，则**FIRST**(β)中除ε以外的一切符号都属于**FOLLOW**(A)
  - 若有产生式$B \to \alpha A$，或产生式$B \to \alpha A \beta$而$\varepsilon \in$**FIRST**(β)，则**FOLLOW**$(B) \subseteq$**FOLLOW**(A)，即**FOLLOW**(B)中的一切符号都要放入**FOLLOW**(A)中

判断是否为LL(1)文法：

- 方法一：两个选择的FIRST集交集为空，若其中包含空，则其FOLLOW集与之交集也为空

- **LL(1)文法：**
  任何两个产生式$A \to \alpha \mid \beta$都满足下列条件：
  - $\mathbf{FIRST}(\alpha) \cap \mathbf{FIRST}(\beta) = \varnothing$
  - 若$\beta \Rightarrow^* \varepsilon$，那么$\mathbf{FIRST}(\alpha) \cap \mathbf{FOLLOW}(A) = \varnothing$

- 例如，对于下面文法，面临串$a\ldots$时不知用什么规则，所以它不是LL(1)文法

  $S \to A\ B$
  $A \to a\ b \mid \varepsilon$　　$a \in \mathbf{FIRST}(ab) \cap \mathbf{FOLLOW}(A)$
  $B \to a\ C$
  $C \to \ldots$

- 方法二：预测分析表中没有多重定义的条目

递归下降预测分析

先判断当前指向的符号是否在产生式右边式子的FIRST集合中，若在，则进行后面的匹配；如果有ε，则考虑FOLLOW集合

```
void S(){
  if (lookahead == '(' ){
    match( '(' ); L(); match( ')' ); }
  else  if  (lookahead == 'a'){
          match ('a');}
  else error( );
}

void L(){
  if (lookahead == '(' || lookahead == 'a' ){
    S(); L'();}
  else error( );
}
```

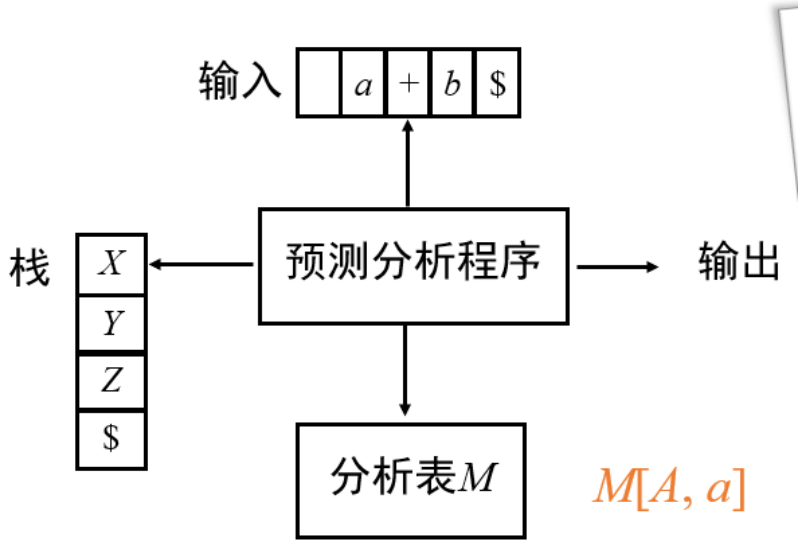
$$S \to (L) \mid a$$
$$L \to S\ L'$$

```
void L'(){
  if (lookahead == ',' ){
    match( ',' ); S(); L'();}
  else if (lookahead != ')')
    error( );
}
```

$$L' \rightarrow , S L' \mid \varepsilon$$

ε产生式的情况：查看输入字符是否属于其FOLLOW集，如果属于，则什么也不做，继续执行递归上一层的代码

非递归的预测分析



栈的初始状态：$S　栈的结束状态：$　产生式倒着放进去

栈顶=输入指针指向的字符时，消去

例：

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

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)		

构造预测分析表

构造方法：

(1)对文法的每个产生式$A\to\alpha$，执行(2)和(3)

(2)对FIRST(α)的每个终结符a，把$A\to\alpha$加入$M[A,a]$

(3)如果ε在FIRST(α)中，对FOLLOW(A)的每个终结符b（包括$），把$A\to\alpha$加入$M[A,b]$

(4)M的其它没有定义的条目都是error

预测分析的错误恢复

- 紧急方式（panic mode）
  - 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（Synchronizing set）为止
  - 该方法的效果依赖于同步记号集合的选择。集合的选择应该使得语法分析器能从实际遇到的错误中快速恢复
    - 例如，可以把FOLLOW(A)中的所有终结符放入非终结符A的同步记号集合

- 同步记号集合的选择
  - 1、把FOLLOW(A)的所有终结符放入非终结符A的同步记号集合
  - 2、把高层结构的开始符号加到低层结构的同步记号集合中
  - 3、把FIRST(A)的终结符加入A的同步记号集
  - 4、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

又叫移进-归约分析

最右推导的逆过程是最左归约，最左规约是规范归约

进行归约，找到句柄，句柄是每个句型里与前面相比发生变化的部分

例 3.18

$$E \to E+E \mid E*E \mid (E) \mid \mathrm{id}$$

$$\begin{aligned} E &\Rightarrow_{rm} E*E \\ &\Rightarrow_{rm} E*E+E \\ &\Rightarrow_{rm} E*E+\mathrm{id}_3 \\ &\Rightarrow_{rm} E*\mathrm{id}_2+\mathrm{id}_3 \\ &\Rightarrow_{rm} \mathrm{id}_1*\mathrm{id}_2+\mathrm{id}_3 \end{aligned}$$

其中id_1是$id_1 * id_2 + id_3$的句柄

用栈实现移进-归约分析

根据归约过程可以得到分析过程：

这里需要把非终结符移入栈中，然后将栈顶的句型归约

3.4.4 移

假定已经知道了归约过程（即推导的逆过程）为：

$E \Rightarrow_{rm} E*E \Rightarrow_{rm} E*E+E \Rightarrow_{rm} E*E+\mathrm{id} \Rightarrow_{rm} E*\mathrm{id}+\mathrm{id} \Rightarrow_{rm} \mathrm{id}*\mathrm{id}+\mathrm{id}$

- 例3.19：$E \to E+E \mid E*E \mid (E) \mid \mathrm{id}$

栈	输　入	动　作
$	$\mathrm{id}_1 * \mathrm{id}_2 + \mathrm{id}_3$$	移进
$$\mathrm{id}_1$	$* \mathrm{id}_2 + \mathrm{id}_3$$	按$E \to \mathrm{id}$归约
$$E$	$* \mathrm{id}_2 + \mathrm{id}_3$$	移进
$$E*$	$\mathrm{id}_2 + \mathrm{id}_3$$	移进
$$E*\mathrm{id}_2$	$+ \mathrm{id}_3$$	按$E \to \mathrm{id}$归约
$$E*E$	$+ \mathrm{id}_3$$	移进
$$E*E+$	$\mathrm{id}_3$$	移进
$$E*E+\mathrm{id}_3$	$	按$E \to \mathrm{id}$归约
$$E*E+E$	$	按$E \to E+E$归约
$$E*E$	$	按$E \to E*E$归约
$$E$	$	接受

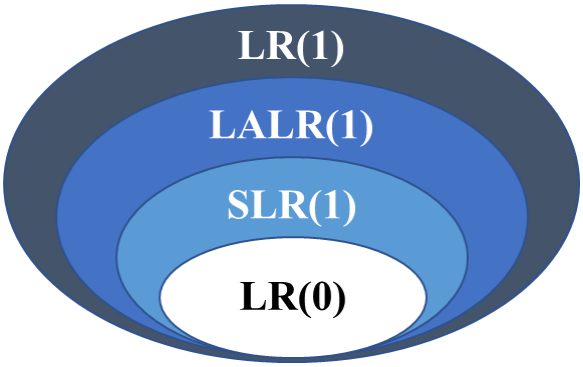
这个过程中可能会出现：

- 移进-归约冲突：下一个输入符号是移进还是归约？
- 归约-归约冲突：有多个产生式，归约成哪一个？

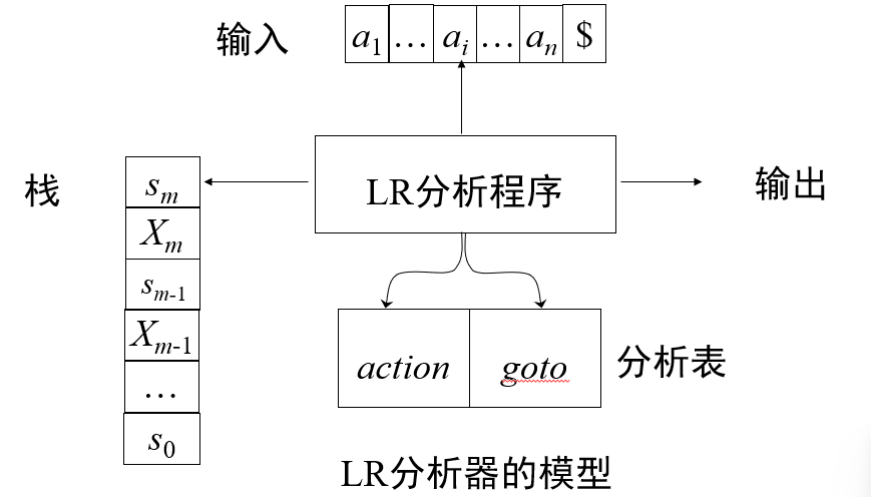
出现冲突的就不是LR文法

LR分析

L：从左到右扫描；R：构造最右推导的逆；k决定分析动作时向前搜索的符号个数，省略表示k=1。程序设计语言的文法一般属于LR(1)类



LR分析技术的包含关系



LR分析器的模型

初始：

栈	输入
$	w$

分析成功：

栈	输入
$S	$

LR分析算法：

看栈顶的数字和指针指向的输入字符在分析表中对应的动作

- 若是移进，则将字符移入栈中，并将对应的数字压栈
- 若是归约，将需要归约的符号弹出栈，对应终结符入栈，并根据转移表获取数字并压栈

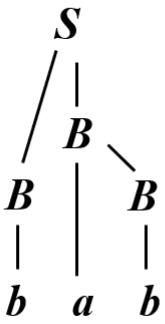
2024年6月26日 编译原理-王欣捷-第三章 语法分析(5)

3.5.1 LR分析算法

- ①$S\rightarrow BB$ ②$B\rightarrow aB$ ③$B\rightarrow b$
- 输入：$b\ a\ b$

状态	ACTION			GOTO	
	a	b	$\$$	S	B
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

例：



栈	输　入	动　作
0	$b\ a\ b\ \$$	移进
0 b 4	$a\ b\ \$$	按$B\rightarrow b$归约
0 B 2	$a\ b\ \$$	移进
0 B 2 a 3	$b\ \$$	移进
0 B 2 a 3 b 4	$\$$	按$B\rightarrow b$归约
0 B 2 a 3 B 6	$\$$	按$B\rightarrow aB$归约
0 B 2 B 5	$\$$	按$S\rightarrow BB$归约
0 S 1	$\$$	接受

算法：

```
令指针a指向w$的第一个符号；
while(1) { /* 永远重复*/
    令s是栈顶的状态；
    if ( ACTION [s，a] = 移进t ) {
        将a和t依次压入栈中；
        令指针a指向下一个输入符号；
    } else if (ACTION [s，a] =归约A → β ) {
        从栈中弹出 2*|β| 个符号；
        将A和GOTO [t，A]压入栈中；
        输出产生式A → β ；
    } else if (ACTION [s，a] =接受) break；  /* 语法分析完成*/
    else调用错误恢复例程；
}
```

活前缀：右句型的前缀，该前缀不超过最右句柄的右端

LR分析法栈中的文法符号总是形成一个活前缀

比较：

	LR(1)方 法	**LL(1)方 法**
建立分析树的方式	自 下 而 上	自 上 而 下
归约还是推导	规 范 归 约 （最左归约）	最 左 推 导
决定使用产生式的时机	看见产生式整个右部推出的东西后才算是看准了用哪个产生式进行归约	看见产生式右部推出的第一个终结符后便确定用哪个产生式进行推导
对文法的显式限制	对文法没有限制	无左递归、无公共左因子
分析表比较	状态×文法符号 分析表大	非终结符×终结符 分析表小
分析栈比较	状态栈，通常状态比文法符号包含更多信息	文法符号栈
确定句柄	根据栈顶状态和下一个符号便可以确定句柄和归约所用产生式	无句柄概念

	LR(1)方 法	LL(1)方 法
语法错误	决不会将出错点后的符号移入分析栈	和LR一样，决不会读过出错点而不报错

构造LR分析表

一些概念：

- LR(0)项目：在右部的某个地方加点的产生式，加点的目的是用来表示分析过程中的状态
- A→ε 对应的LR(0)项目为 A→ ·
- 拓广文法：若开始符号为S，则拓广文法为加入文法符号 S' 及产生式 S'→S 的文法，用这条产生式归约就表明分析成功
- closure函数：对核心项目求闭包得到非核心项目
- goto函数：
- 搜索符：若有项目[A→ $\alpha \cdot \beta$, a] ，其中a就是搜索符。它是子串$\alpha\beta$所在的右句型中直接跟在β后面的终结符，它决定了何石匠$\alpha\beta$归约为A，它通常是FOLLOW(A)的真子集。
- 这个搜索符其实计算的就是整条产生式后面的符号
- 同心的LR(1)项目集：忽略搜索符后它们是相同的集合

SLR分析表

步骤：

- 构造识别活前缀的DFA
  - 拓广文法，从0开始为产生式编号
  - 构造LR(0)项目集规范族：用closure函数确定一个项目，用goto函数找到下一个项目

例如项目集I_0：

$$
\begin{array}{ll}
E' \to \cdot E & \text{（核心项目）} \\
\hline
E \to \cdot E + T & \text{（非核心项目，通} \\
E \to \cdot T & \text{过对核心项目求闭} \\
T \to \cdot T * F & \text{包而获得} \\
T \to \cdot F & \\
F \to \cdot (E) & \\
F \to \cdot \mathbf{id} &
\end{array}
$$

- 用该DFA构造分析表
  - 项目集中的项目分四类：
    - 接受项目[S'→S·]：action[i, $] = acc
    - 移进项目[A→α·aβ]：action[i,a]=sj
    - 归约项目[A→α·]：action[i,b]=rj，**b∈FOLLOW(A)**
    - 待归约项目[A→α·Bβ]：goto[i, B] = j

若SLR分析表中出现移进-归约冲突或归约-归约冲突，则该文法不是SLR文法

- SLR文法描述能力有限的原因：
  - 归约时缺乏对上下文的考虑：即归约时只考虑了下一个输入符号是否属于与归约项目$[A\rightarrow\alpha\cdot]$相关联的$FOLLOW(A)$集合，而没考虑串$\alpha$所在的右句型的上下文（即在该右句型中$\alpha$能否被归约为$A$）。
  - $b\in FOLLOW(A)$只是归约α的一个必要条件，而非充分条件
- 解决方法：规范LR分析（LR(1)分析）

LR(1)分析表

步骤：在SLR的基础上，改为构造LR(1)项目集规范族，带上搜索符，归约时根据搜索符进行归约

计算**搜索符**步骤：

- 初始状态：[S'→·S, $]，即直接加上$符号
- goto(I, X)中状态X的搜索符
  - 核心项目：与I中原项目搜索符相同
  - 非核心项目：若非核心项目[B→·γ,b]的核心项目为[A→$\alpha\cdot B\beta$, a]，则b=FIRST(βa)
  - 例：

$$
\begin{aligned}
&I_0:\\
&S'\rightarrow\cdot S,\ \$\\
&S\rightarrow\cdot BB,\ \$\\
&B\rightarrow\cdot bB,\ b/a\\
&B\rightarrow\cdot a,\ b/a
\end{aligned}
$$

LALR分析表

步骤：在LR(1)的基础上，**合并同心项目集**，得到LALR(1)项目集族

可能会增加**归约-归约冲突**

二义文法的应用

二义文法绝不是LR文法

可以利用优先级和结合性解决冲突

二义文法：$E \to E+E \mid E*E \mid (E) \mid \mathbf{id}$
规定： *优先级高于+，两者都是左结合
$\mathbf{FOLLOW}(E) = \{+, *,), \$\}$

例：

LR(0)项目集I_7

$E \to E+E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

$\mathbf{id}+\mathbf{id}\quad +\mathbf{id}$
$\mathbf{id}+\mathbf{id}\quad *\mathbf{id}$

面临+，归约　面临*，移进

- 若归约项的优先级比较高，则先归约
- 若将移进的优先级比较高，则先移进
- 若优先级一样，则看结合性，若左结合就先归约

- **LR**分析器在什么情况下发现错误？
  - 访问动作表时若遇到出错条目
  - 访问转移表时它决不会遇到出错条目
  - 决不会把不正确的后继移进栈
  - 规范的**LR**分析器甚至在报告错误之前决不做任何无效归约

3.6.3 LR分析的错误恢复

- 两种错误恢复策略：
  - 紧急方式错误恢复：抛弃若干个输入符号，直到合法为止
  - 短语级错误恢复：针对分析表中每个空白条目确定不同的最合适的恢复方法

用Yacc处理二义文法

方法：在程序中的声明部分，为二义文法规定优先级和结合性

例：

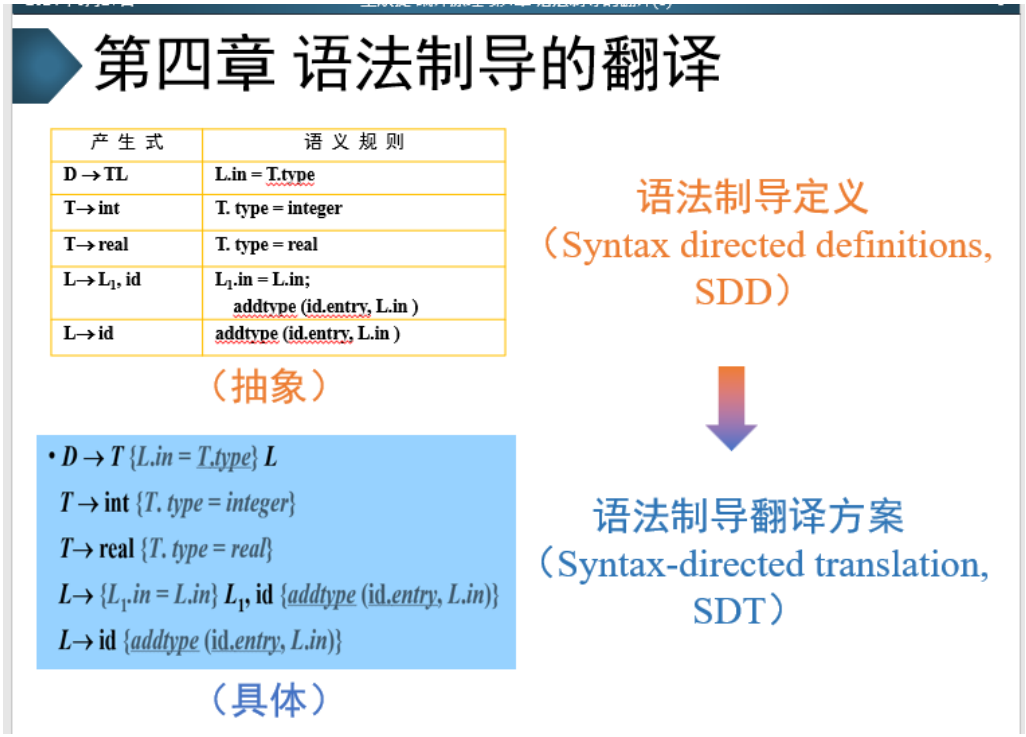
```
%left '+' '-'
%left '*' '/'
```

这里规定+-的优先级低于*/，且都是左结合的

第四章 语法制导的翻译

- 重点：构造语法制导定义和翻译方案
  - 语法制导定义、综合属性、继承属性、翻译方案的概念，能区分综合属性和继承属性
  - S属性定义的概念
  - 能看懂构造抽象语法树的语法制导定义
  - 知道S属性定义如何自下而上计算
  - 会构造语法制导定义和注释分析树
    - 作业1、作业2
  - 会写翻译方案
    - 作业3

重点是下面这两个内容，要会自己写语法制导定义和翻译方案：



一些概念：

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

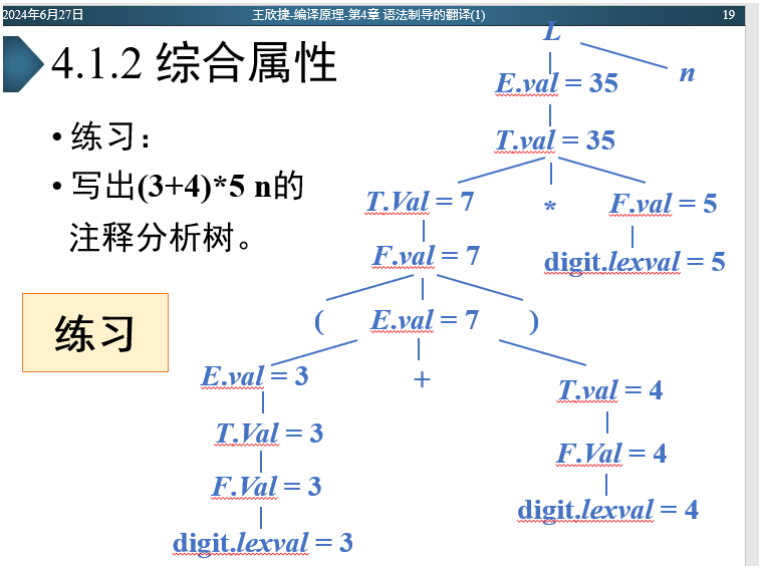
翻译方案：语义动作放到{}内，插入到产生式中，综合属性写后面，继承属性写对应符号的前面

综合属性：通过分析树中它的子结点的属性值来计算

继承属性：由结点的兄弟结点、父结点和自己的属性值来计算

注释分析树：结点的属性值都标注出来的分析树

例：

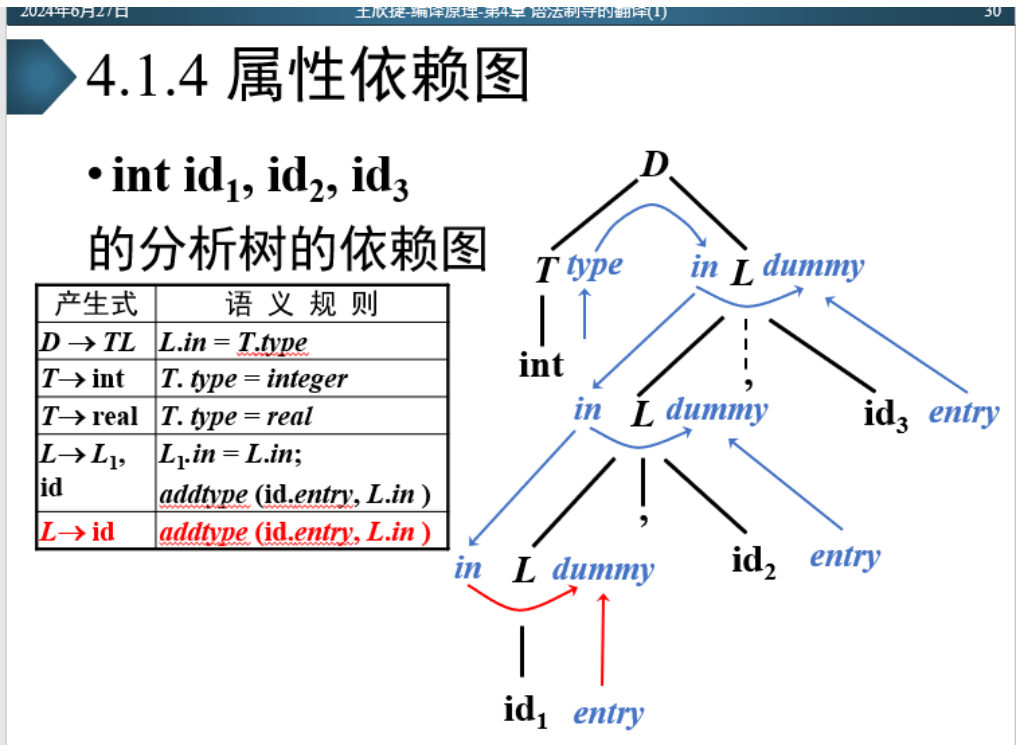


语法制导的定义：

- **S属性的定义**（重点掌握这个）：仅仅使用**综合属性**的语法制导定义
  - 分析树各结点综合属性的计算可以自下而上地完成
- L属性的定义：要么是综合属性，要么是依赖左边兄弟节点或父结点的继承属性

• *L*属性定义：如果每个产生式$A \rightarrow X_1\ X_2\ \ldots\ X_n$ 的每条语义规则计算的属性是*A*的综合属性；或者是X_j 的继承属性，$1 \le j \le n$，但它仅依赖：
  • 该产生式中X_j左边符号$X_1, X_2, \ldots, X_{j-1}$的属性；
  • *A*的继承属性
• *S*属性定义属于*L*属性定义

属性依赖图



在通过拓扑排序依次计算属性值

重点例题：

构造抽象语法树的语法制导定义

使用指针

声明叶子节点：mkleaf(num, val) 和 mkleaf(id, entry)

声明非叶子节点：mknode(op, left, right)

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

第五章 类型检查

- 动态的：
  - 执行错误：程序运行时出现的错误（包括会被捕获和不会被捕获的错误）
  - 良行为的程序：运行时不会引起不会被捕获的错误
  - 安全语言：任何合法程序都是良行为的
- 静态的：
  - 类型化语言：
    - 语言的规范为所有变量的每种运算都定义了各种运算对象和运算结果所允许的类型
    - 分为显示类型化和隐式类型化：可以没有类型声明
    - 汇编语言为无类型语言
  - 类型系统：由一组定型规则构成，目的是用静态检查的方式来保证合法程序运行时的良行为
  - 类型检查：根据定型规则来确定各语法的构造类型
  - 良类型的程序（合法程序）：能通过类型检查的程序
- 类型可靠的语言：良类型程序都是良行为的
- 禁止错误：
  - 所有不会被捕获错误集合 + 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
- 类型检查器：完成类型检查的算法

类型表达式

- 基本类型
  - integer
  - real
  - char
  - boolean
  - void
  - type_error
- 构造类型
  - array(num, T)

    例：array(3, int)
  - pointer(T)

    例：pointer(int)
  - 笛卡尔积构造符：×
  - 函数构造符：→

    例：若有一个函数 bool foo(int a, float b)，其类型表达式为：int×float → bool
  - 记录构造符：$record(N_1 : T_1, N_2 : T_2, \ldots, N_n : T_n)$，这里面N是name，T是类型

断言

分类：

- 环境断言

  Γ |– ◊　　该断言表示**Γ**是良形**(*well formed*)**的环境(i.e., it has been properly constructed)
  - 后面将用推理规则来定义环境的语法（而不是用文法）
- 语法断言

  Γ |– nat　　在环境**Γ**下，**nat**是类型表达式
  - 后面将用推理规则来定义类型表达式的语法（而不是用文法）
- 定型断言（最重要）

  Γ |– *M* : *T*　　在环境**Γ**下，***M***具有类型***T*** **(*M*的自由变量都出现在Γ中)**

  例：**∅ |– true : boolean**　　**{*x* : nat} |– *x*+1 : nat**
  - 后面将用推理规则来确定语法结构实例的类型

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

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

  - 横线上方是前提(premise)
  - 横线下方是结论(conclusion)
  - 前提为零的规则叫做公理

看懂这些规则：

（规则名）（注释）　推理规则　（注释）

- 环境规则

  (Env ∅)
  $$\frac{}{\varnothing \vdash \Diamond}$$

  - 空环境是良形的，它是一个公理。

- 语法规则

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

  - 任何良形环境Γ下，**boolean**是一个布尔表达式

- 定型规则

  (Exp +)
  $$\frac{\Gamma \vdash M : \mathbf{int}, \Gamma \vdash N : \mathbf{int}}{\Gamma \vdash M + N : \mathbf{int}}$$

  - 推理规则的结论是定型断言，则称之为定型规则

类型检查器的例子

先设计类型系统：

- 用环境规则在环境中增加一个变量到类型的映射

环境规则

(Env ∅)
$$\frac{}{\varnothing \vdash \Diamond}$$

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

其中$\mathrm{id} : T$是该简单语言的一个声明语句

遇到一个声明语句，则向静态定型环境中增加一个变量到类型的映射

这里先在前提上说明了符号表里面有T这个类型

- 用语法规则声明类型表达式

语法规则

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

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

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

语法断言中用到了类型表达式

语法规则

(Type Ref) $(T \neq void)$ $\dfrac{\Gamma \vdash T}{\Gamma \vdash pointer(T)}$

(Type Array) $(T \neq void)$ $\dfrac{\Gamma \vdash T,\ \Gamma \vdash N : integer}{\Gamma \vdash array\ (N,\ T)}$ $(N>0)$

(Type Function) $(T_1,\ T_2 \neq void)$ $\dfrac{\Gamma \vdash T_1,\ \Gamma \vdash T_2}{\Gamma \vdash T_1 \rightarrow T_2}$

语法断言中用到了类型表达式

- 用定型规则确定表达式的类型

定型规则——表达式

(Exp Truth) $\dfrac{\Gamma \vdash \Diamond}{\Gamma \vdash \mathbf{truth} : boolean}$

(Exp Num) $\dfrac{\Gamma \vdash \Diamond}{\Gamma \vdash \mathbf{num} : integer}$

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

- **Exp Id表示，如果id出现在环境Γ中，则id的类型就是它在Γ中的类型。**

定型规则——表达式

(Exp Mod) $\dfrac{\Gamma \vdash E_1 : integer,\ \Gamma \vdash E_2 : integer}{\Gamma \vdash E_1 \ \mathbf{mod}\ E_2 : integer}$

(Exp Index) $\dfrac{\Gamma \vdash E_1 : array(N, T), \Gamma \vdash E_2 : integer}{\Gamma \vdash E_1[E_2] : T}$ $(0 \leq E_2.\mathrm{val} < N)$

(Exp Deref) $\dfrac{\Gamma \vdash E : pointer(T)}{\Gamma \vdash E\uparrow : T}$

(Exp FunCall) $\dfrac{\Gamma \vdash E_1 : T_1 \rightarrow T_2,\quad \Gamma \vdash E_2 : T_1}{\Gamma \vdash E_1\ (E_2) : T_2}$

（Exp Index）规则表示数组中一个元素的类型

- 用定型规则确定语句

定型规则——语句

(State Assign) $(T=boolean$ or $interger)$

$\dfrac{\Gamma \vdash \mathbf{id} : T,\ \Gamma \vdash E : T}{\Gamma \vdash \mathbf{id} := E : void}$

(State If) $\dfrac{\Gamma \vdash E : boolean,\ \Gamma \vdash S : void}{\Gamma \vdash \mathbf{if}\ E\ \mathbf{then}\ S : void}$

(State While) $\dfrac{\Gamma \vdash E : boolean,\ \Gamma \vdash S : void}{\Gamma \vdash \mathbf{while}\ E\ \mathbf{do}\ S : void}$

(State Seq) $\dfrac{\Gamma \vdash S_1 : void,\ \Gamma \vdash S_2 : void}{\Gamma \vdash S_1; S_2 : void}$

简单起见，没有设计整个程序段的定型规则

进行类型检查

根据上面的规则设计语义动作

$$E \rightarrow E_1\,[E_2\,]\ \{\text{if}\ (\,E_2.type == integer\ \&\&\ E_1.type == array(s,t))\ \ E.type = t;\ \text{else}\ E.type = type\_error;\ \}$$

$$E \rightarrow E_1\uparrow\quad \{\text{if}\ (\,E_1.type == pointer(t)\,)\ \ E.type = t;\ \text{else}\ E.type = type\_error;\ \}$$

$$E \rightarrow E_1\,(E_2\,)\ \{\text{if}\ (\,E_2.type == s\ \&\&\ E_1.type == s \rightarrow t\,)\ \ E.type = t;\ \text{else}\ E.type = type\_error;\ \}$$

$$S \rightarrow \text{id} := E\quad \{\text{if (id}.type == E.type\ \&\&\ E.type \in \{boolean, integer\})\ S.type = void;\ \text{else}\ S.type = type\_error;\ \}$$

$$S \rightarrow \text{if}\ E\ \text{then}\ S_1\ \{\text{if}\ (E.type == boolean)\ S.type = S_1.type;\ \text{else}\ S.type = type\_error;\ \}$$

对应定型规则：

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

类型表达式等价

结构等价:

- 把所有的类型名字用它们定义的类型表达式代换后，两个类型表达式完全相同
- 可以用算法递归得检查，不能检查环

名字等价

- 两个类型表达式不做名字代换就结构等价

第六章 运行时存储空间的组织和管理

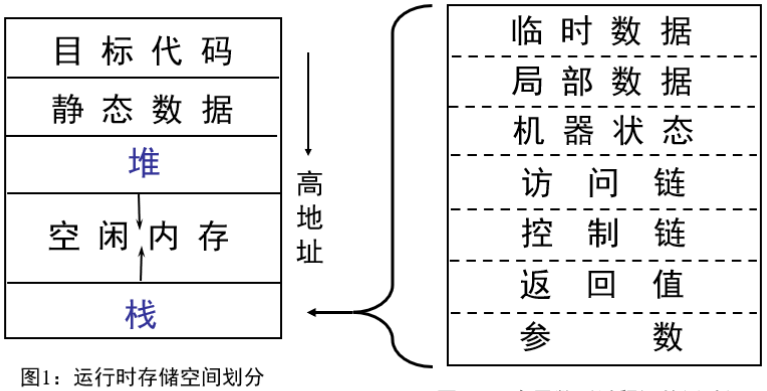


图1：运行时存储空间划分
C编译器就是图上这种方式，
但有的编译器会把堆和栈位置互换

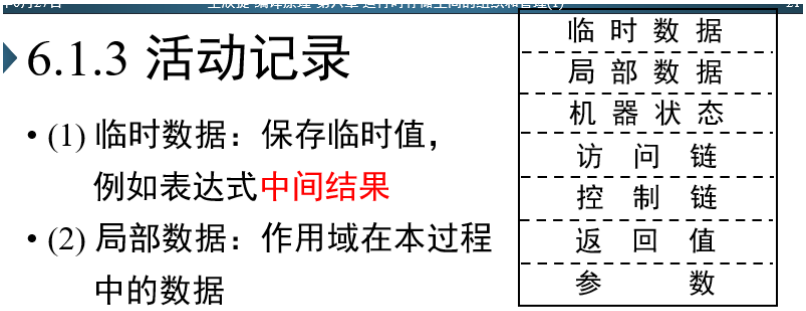
图2：一个函数（过程）的活动记录
仅表示一般组织形式，
不同的编译器该形式可能不同

一些概念：

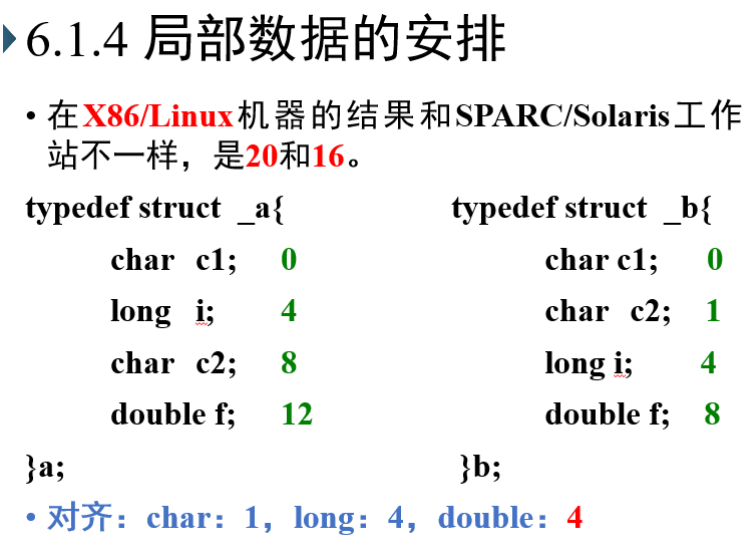
- 过程：是一个声明，将过程名和过程体联系起来
- 作用域：声明起作用的程序部分
- 绑定：将名字映射到存储单元

活动记录

布局：



局部数据的布局：

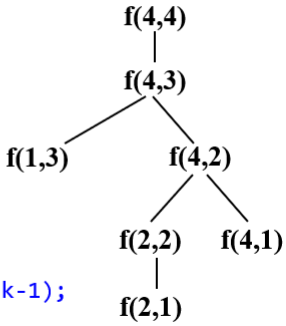


这里要求了不同类型在内存中的对齐，比如double的地址必须是4的倍数

活动树

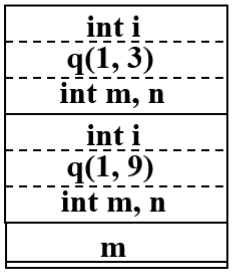
- 练习：画出下列C程序的活动树。
- 初值n=4，k=4

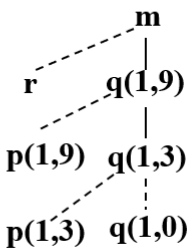
```
int f(int n, int k) {
   if (n==1 || k==1)
     return 1;
   else if (n<k)
     return f(n,n);
   else if (n==k)
     return f(n,n-1)+1;
   else
     return f(n-k,k)+f(n,k-1);
}
```



运行栈

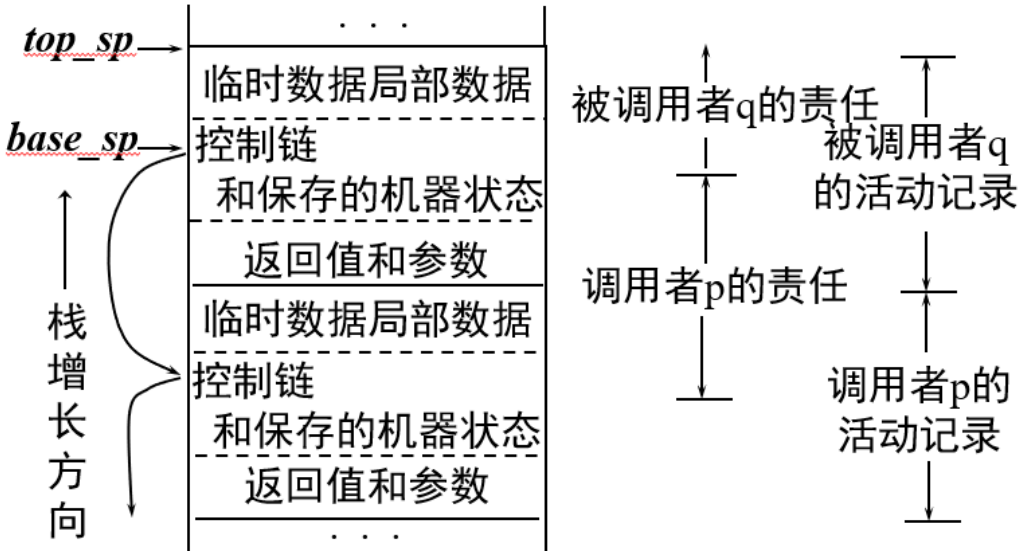
- 运行栈：把控制栈中的信息拓广到包括过程活动的活动记录





控制栈：m, q(1,9), q(1,3)

- 调用者p和被调用者q之间的任务划分（无访问链）



调用序列

- 1、过程p调用过程q的调用序列
  - p计算实参，依次放入栈顶，并在栈顶留出放返回值的空间。*top_sp*的值在此过程中减小
  - p把返回地址压入q的活动记录，控制转到q。
  - q将要保存的寄存器（*base_sp*除外）的值和其他机器状态信息压入栈，然后把*base_sp*的当前值（p的*base_sp*）压入栈，并把当前*top_sp*的值作为自己*base_sp*的值。
  - q根据局部数据域和临时数据域的大小减小*top_sp*的值，初始化它的局部数据，并开始执行过程体

返回序列

- 过程p调用过程q的返回序列
  - q把返回值置入活动记录中存放返回值的地方
  - q对应调用序列步骤（4），增加*top_sp*的值
  - q恢复寄存器（包括*base_sp*）和机器状态，返回p
  - p对应调用序列步骤（1）增加*top_sp*的值，并取出返回值

参数传递

- 值调用：实参的右值传给被调用过程
- 引用调用：实参的左值传给被调用过程
- 换名调用：用实参表达式对形参进行正文替换

printf 函数

- 下面程序中printf只有一个参数，但却会输出3个整数，说明一下为什么会这样。

  main(){ printf(“%d, %d, %d\n”); }
- 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。
- 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。因此printf的实现首先取到了第一个参数——格式控制字符串，然后分析它的格式要求，发现有三个参数，于是去栈上相对地址处取第2、3、4个参数（因为%d是整型，所以取到的是整数），而不管调用者是否提供了这些参数。所以最终输出了三个整数，这三个整数的值是不确定的。

第七章 中间代码生成

中间语言

后缀表示

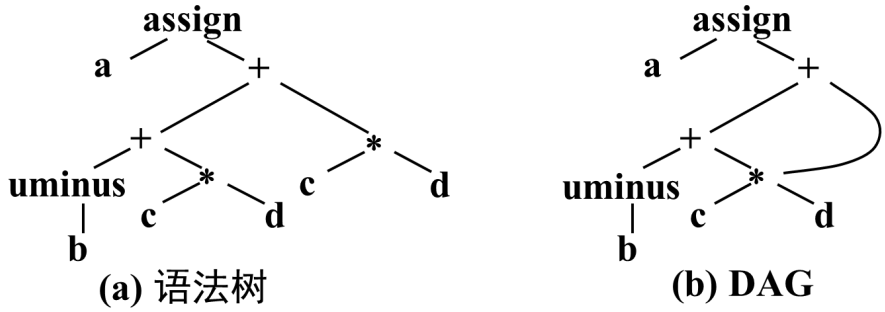
后缀表示不需要括号

$(8-4)+2$ 的后缀表示是8 4 - 2 +

图形表示

用抽象语法树和有向无环图表示

例：



(a) 语法树　　(b) DAG

a = (–b + c*d) + c*d的图形表示

其后缀表示为：a b uminus c d * + c d * + assign

三地址码（重要）

语法树的代码	dag的代码
$t_1 = -b$	$t_1 = -b$
$t_2 = c * d$	$t_2 = c * d$
$t_3 = t_1 + t_2$	$t_3 = t_1 + t_2$
$t_4 = c * d$	$t_4 = t_3 + t_2$
$t_5 = t_3 + t_4$	$a = t_4$
$a = t_5$	

静态单赋值

- 源程序：
  if (flag) x = -1; else x = 1;
  y = x * a;

- SSA表示：
  if (flag) $x_1 = -1$; else $x_2 = 1$;
  $x_3 = \Phi(x_1, x_2)$;
  y = x_3 * a;

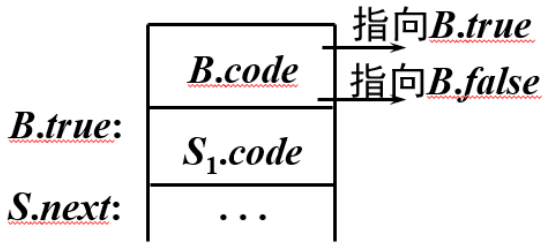
若控制流通过条件为真的部分，则Φ为x_1，若通过条件为假的部分，则Φ为x_2

语句翻译成三地址码

$S \rightarrow$ if B then S_1

语义规则：

- *B.true* = *newLabel*();
  B.false = *S.next*;
  S_1.*next* = *S.next*;
  S.code = *B.code* || *gen*(*B.true*, ':') || S_1.*code*

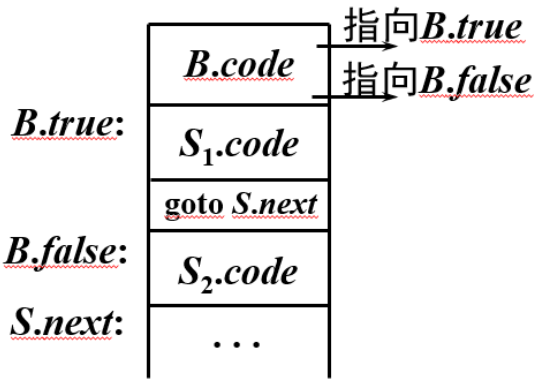


(a) if-then

$S \rightarrow$ if B then S_1 else S_2

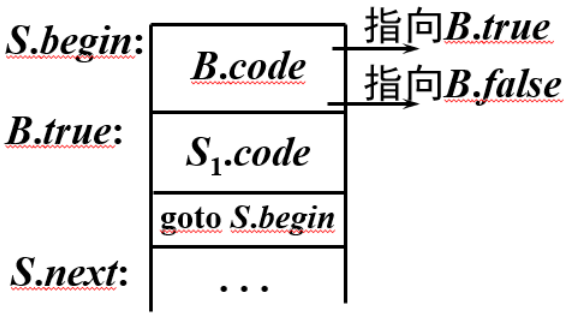
语义规则：

- *B.true* = *newLabel*();
  B.false = *newLabel*();
  S_1.*next* = *S.next*;
  S_2.*next* = *S.next*;
  S.code = *B.code* || *gen*(*B.true*, ':') || S_1.*code* ||
  gen('goto', *S.next*) || *gen*(*B.false*, ':')
  ||S_2.*code*



(b) if-then-else

$S \rightarrow$ while B do S_1



(c) while-do

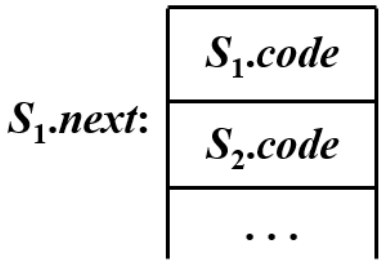
语义规则：

- $S.begin = newLabel()$;
  $B.true = newLabel()$;
  $B.false = S.next$;
  $S_1.next = S.begin$;
  $S.code = gen(S.begin, ':') \| B.code \|$
  $gen(B.true, ':') \| S_1.code$
  $\| gen('goto', S.begin)$

$S \rightarrow S_1; S_2$

语义规则：

- $S_1.next = newLabel()$; $S_2.next = S.next$;
  $S.code = S_1.code \| gen(S_1.next, ':') \| S_2.code$



(d) $S_1; S_2$

code为综合属性，next，begin，true，false为继承属性

$B \rightarrow B_1$ or B_2

语义规则：

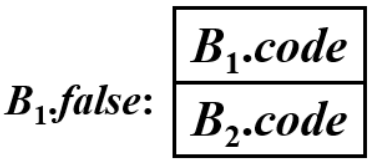
$B_1.true = B.true$;

$B_1.false = newLabel()$;

$B_2.true = B.true$;

$B_2.false = B.false$;

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



$B \rightarrow B_1$ and B_2

语义规则：

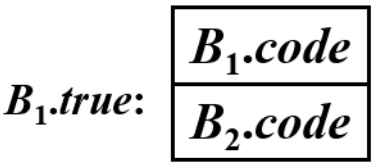
$B_1.true = newLabel()$;

$B_1.false = B.false$;

$B_2.true = B.true$;

$B_2.false = B.false$;

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



$B \rightarrow$ not B_1

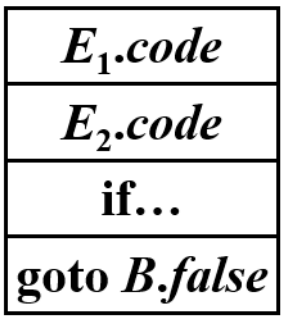
语义规则：

$B_1.true = B.false$;

$B_1.false = B.true$;

$B.code = B_1.code$

- $B \rightarrow E_1 \text{ relop } E_2$



语义规则：

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

$B \rightarrow \text{true}$

语义规则：

$B.code = gen(\text{'goto'}, B.true)$

- $B \rightarrow \text{false}$

语义规则：

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

练习

- [ ] 理解、写正规式
- [ ] 构造DFA（两种方法）
- [ ] 最左推导和最右推导
- [ ] 会画分析树
- [ ] 判断二义文法，构造等价的二义文法和非二义文法，消除二义性（改写文法或在程序中规定优先级）
- [ ] 提左因子，消除左递归
- [ ] 求FIRST和FOLLOW，据此判断是否为LL(1)文法
- [ ] 构造预测分析表，用它分析，多重定义的修改，错误恢复
- [ ] 句柄，根据归约写移进-归约分析器的步骤（非常容易写错）
- [ ] LR分析技术的包含关系
- [ ] 四种文法的包含关系（乔普斯基分类）

- [ ] 进行LR分析
- [ ] 构造LR分析表（SLR，LR，LALR）
- [ ] 计算搜索符，一定记得考虑所有可以展开这个非终结符的产生式的搜索符
- [ ] 使用优先级和结合性解决冲突
- [ ] 画注释分析树
- [ ] 判断是综合属性和继承属性，判断是不是L属性定义或S属性定义
- [ ] 会写S属性的SDD，写L属性的SDT
- [ ] 区别类型检查的定义
- [ ] 会写类型表达式
- [ ] 画活动树，控制栈和活动栈
- [ ] 调用序列和返回序列
- [ ] 中间语言（后缀，图形，三地址码）

优先级：

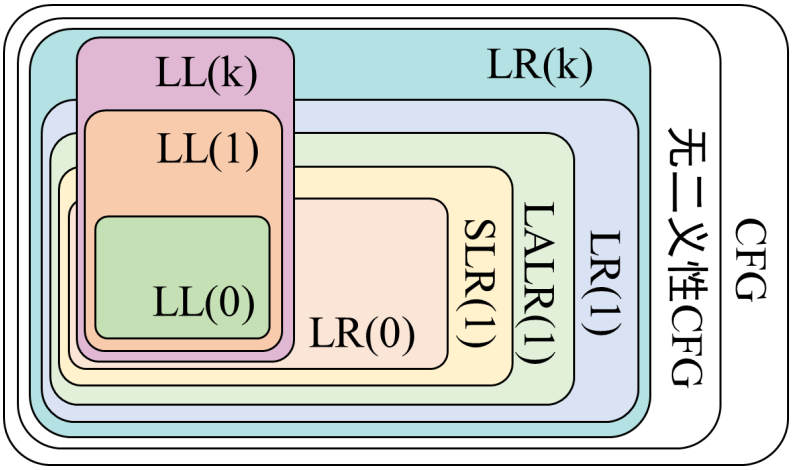
闭包>连接>选择

not > and > or

文法分类：

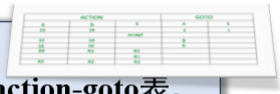
- 四种文法之间的关系：逐级限制
  - 0型文法：α中至少包含1个非终结符
  - 1型文法（CSG）：$|\alpha| \leq |\beta|$
  - 2型文法（CFG）：$\alpha \in V_N$
  - 3型文法（RG）：$A \to wB$或$A \to w$($A \to Bw$或$A \to w$)
- 逐级包含

包含关系：



• **SLR分析表构造过程：**

- 1. 构造识别活前缀的**DFA**
  - 1.1 构造拓广文法，为产生式编号
  - 1.2 构造**LR(0)**项目集规范族（**closure, goto** 函数）
  - 1.3 画出**DFA**，所有状态都为接收状态



- 2. 构造**SLR**分析表
  - 2.1 根据项目集族和**DFA**填**action-goto**表，其中在项目集内遇到归约项目时，按**FOLLOW**集合填归约

- 若有一个类型 int a[10][30], a是数组首元素的地址，指向a[0]，&a指向整个数组的地址

代码表达式	输出结果	类型表达式
a	4223040	***pointer*(*array*(0, *integer*))**
a+1	4223160	***pointer*(*array*(0, *integer*)) × *integer* →*integer***
&a[0]+1	4223160	***pointer*(*array*(0, *integer*)) × *integer* →*integer***
&a+1	4224240	***pointer*(*array*(0, *array*(0, *integer*))) × *integer* →*integer***
&a[0][0]+1	4223044	***pointer(integer)* × *integer* →*integer***