

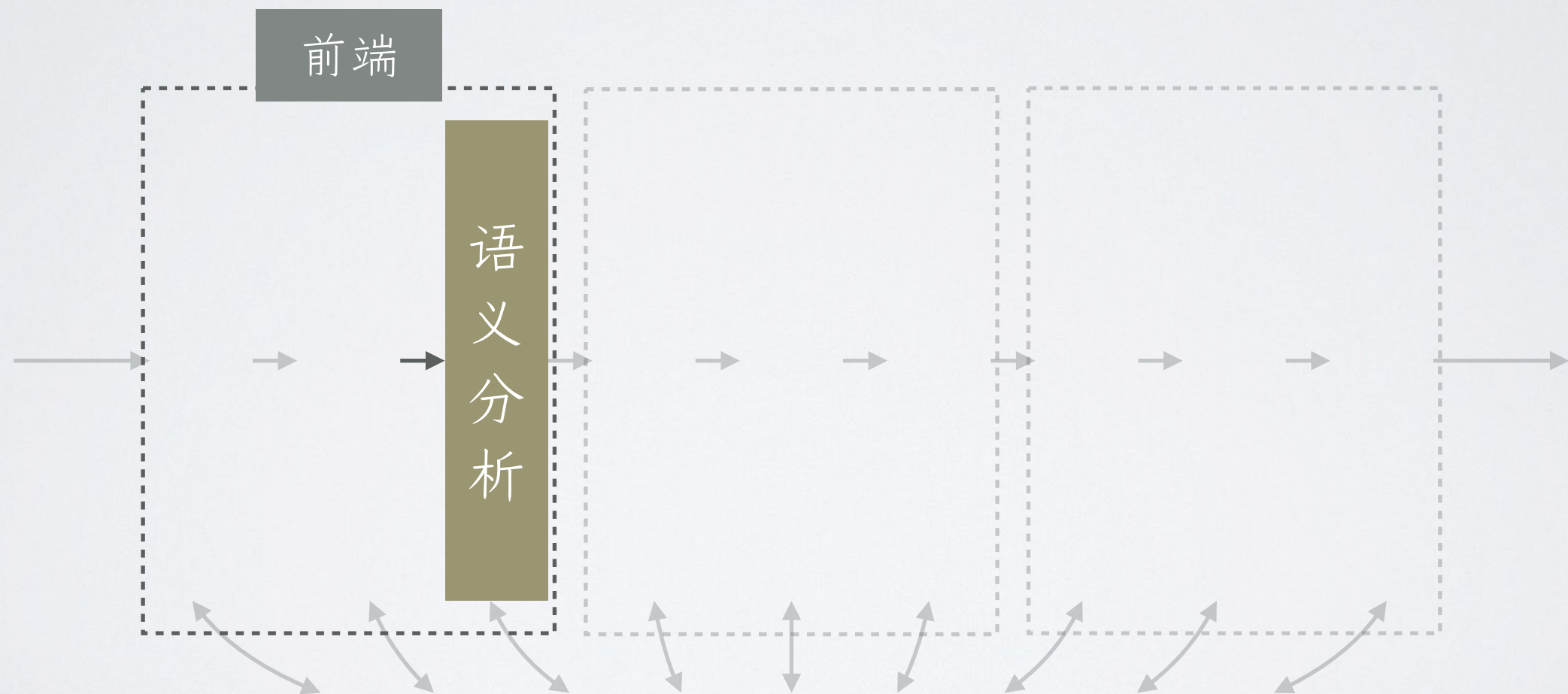


第十讲 语义分析（进阶版）

Advanced Topics in Semantic Analysis

回顾：语义分析的作用

- 基于语法分析树，提取程序的核心语义
- 语义分析在不同场景下可以指不同的工作，但通常包括**翻译到中间表示、类型检查、解释执行**等

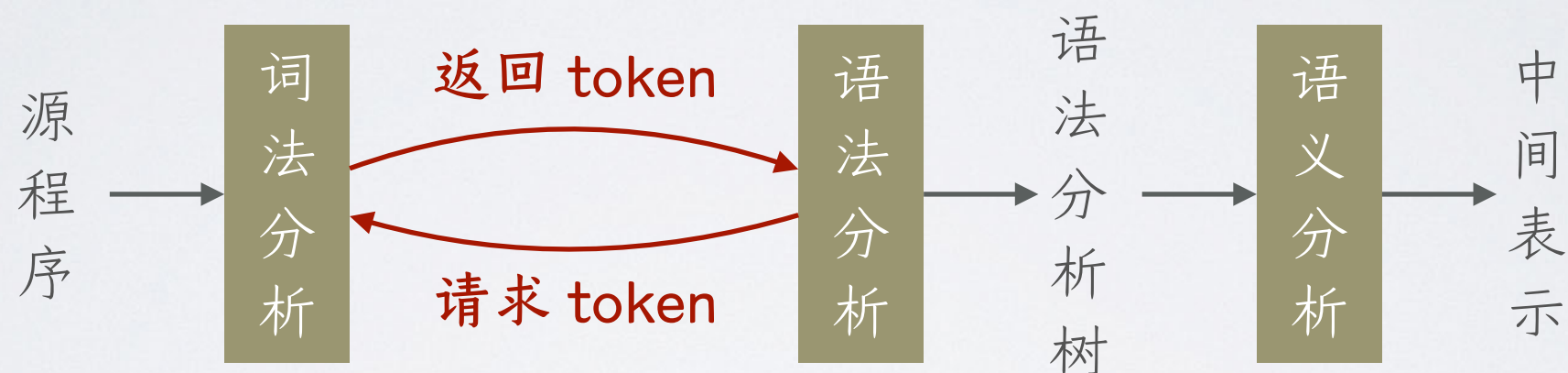


回顾：语义分析的工作和实现

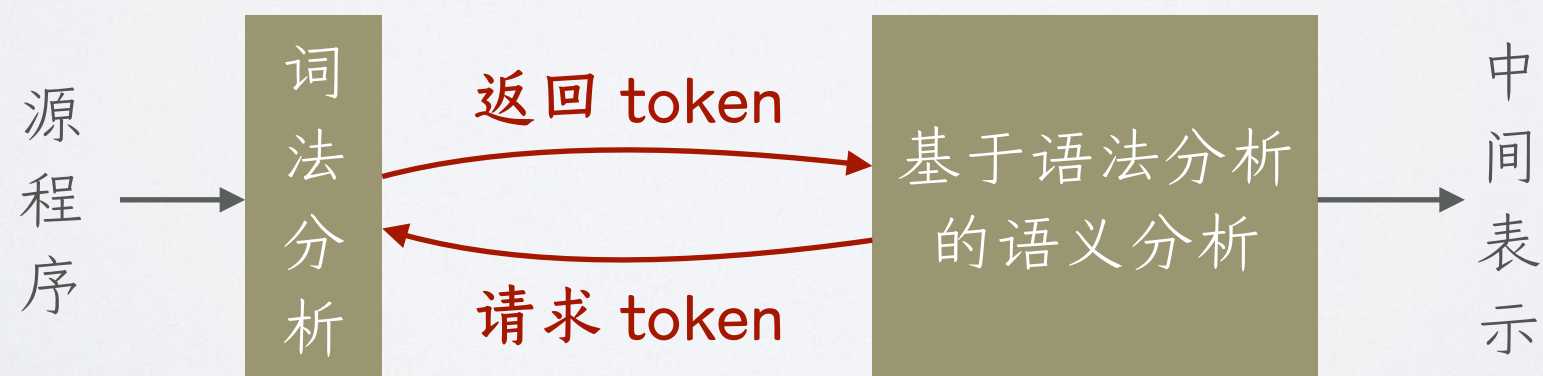
◎ 与优化器中的分析不同，语义分析主要关注**翻译到中间表示**

❖ 也被称为**语义推敲 (semantic elaboration)**

◎ 可以独立于语法分析进行实现



◎ 也可以与语法分析同步进行



回顾：自底向上的语法分析

● 移进-归约分析

- ❖ 用一个分析栈记录移进的终结符号和完成归约的非终结符号
- ❖ 根据分析栈顶的模式和向前看符号确定进行移进还是归约

```

top = 0;
token = next_token();
while (true) {
    if (top == 0) {
        switch (token) {
            case LSQUARE:
                top++; stack[top] = LSQUARE;
                token = next_token(); break;
            case RSQUARE:
                return false;
            case EOF:
                top++; stack[top] = S; break;
        }
    } else if (top == 1 && stack[top] == S) {
        switch (token) {
            case LSQUARE: return false;
            case RSQUARE: return false;
            case EOF: return true;
        }
    } else if .....
}

```

$S' \rightarrow S \text{ EOF}$
 [1] $S \rightarrow \epsilon$
 [2] $S \rightarrow [S] S$

分析栈 \ 向前看	[	]	EOF
ϵ	移进	错误	归约 [1]
S	错误	错误	接受
$\beta[$	移进	归约 [1]	错误
$\beta[S$	错误	移进	错误
$\beta[S]$	移进	归约 [1]	归约 [1]
$\beta[S]S$	错误	归约 [2]	归约 [2]

主要内容

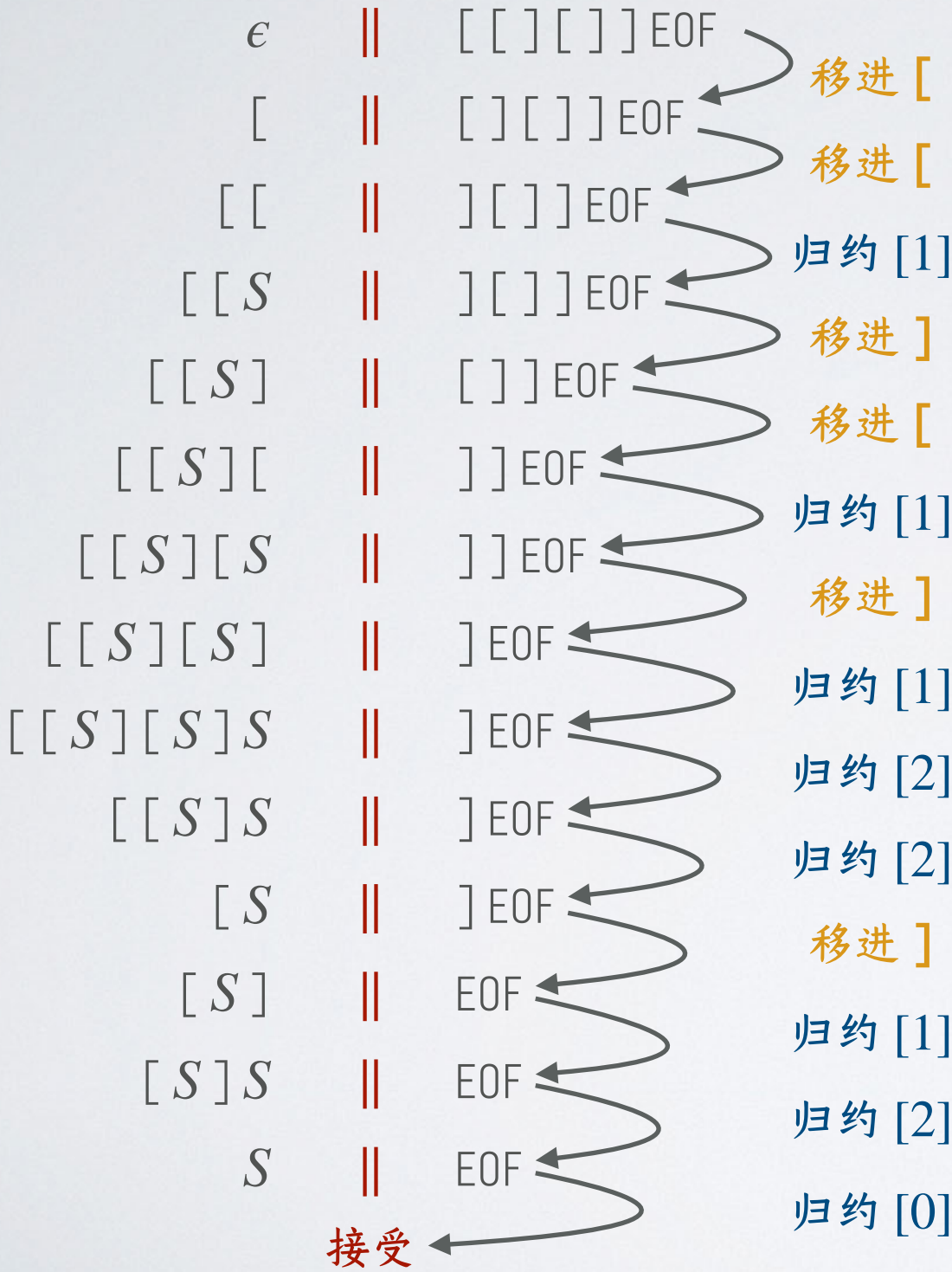
- ◎ 自底向上的语义分析
- ◎ 语义分析的自动生成
- ◎ 类型系统(基础版)

主要内容

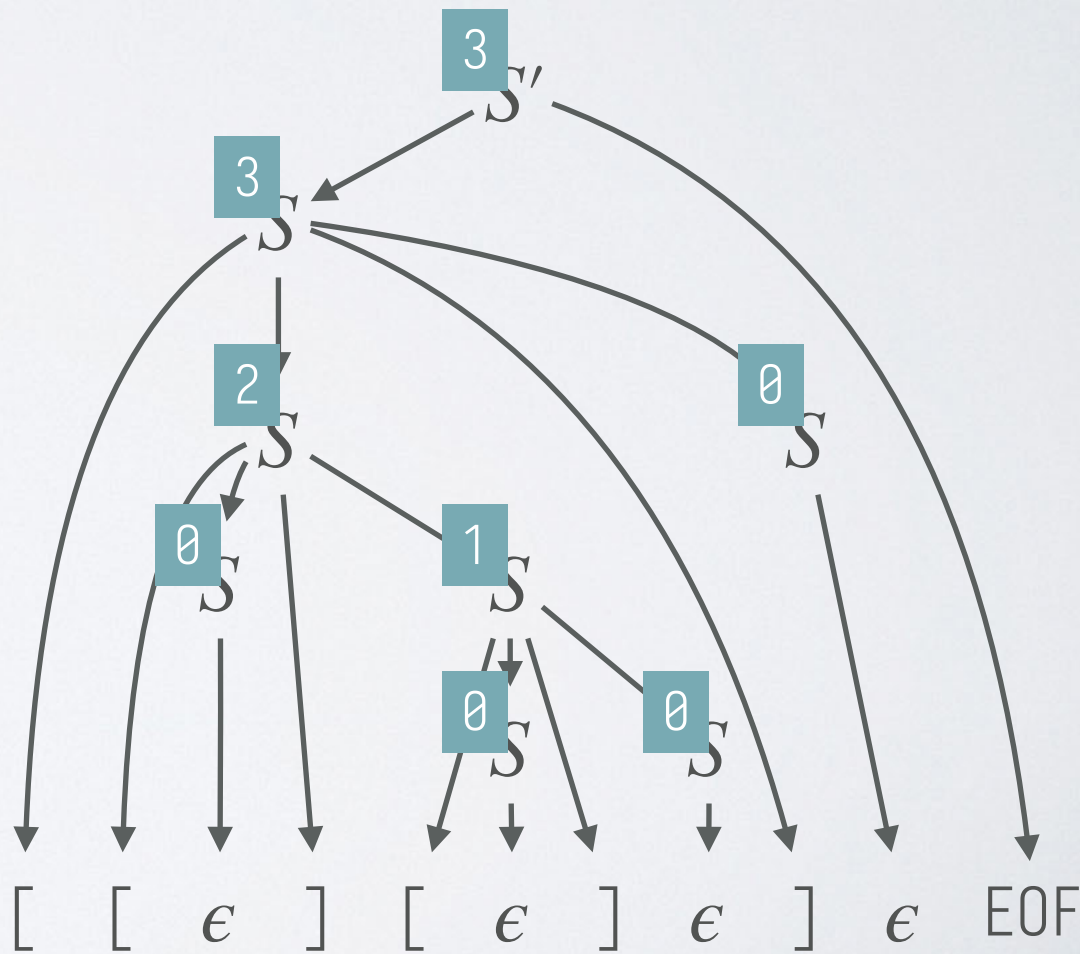
- ◎ 自底向上的语义分析
- ◎ 语义分析的自动生成
- ◎ 类型系统(基础版)

自底向上的语义分析

综合属性 $S'.cnt$ 和 $S.cnt$ 记录匹配的括号对数



产生规则	属性计算规则
[0] $S' \rightarrow S EOF$	$S'.cnt = S.cnt$
[1] $S \rightarrow \epsilon$	$S.cnt = 0$
[2] $S \rightarrow [S_1] S_2$	$S.cnt = 1 + S_1.cnt + S_2.cnt$



S 属性的文法的移进-归约实现

- ◎ 回顾: S 属性的文法中只有综合属性
 - ❖ 信息的流向自底向上, 与语法分析的方向一致
- ◎ S 属性的文法的移进-归约分析
 - ❖ 在分析栈中记录每个非终结符号的综合属性
 - ❖ 在进行归约时, 计算被归约符号的综合属性

```
.....  
} else if (根据分析栈顶和向前看要按照  $A \rightarrow B_1 B_2 \dots B_k$  进行归约) {  
    按照栈顶  $k$  个符号及其记录的综合属性计算  $A$  的综合属性;  
    top = top - k + 1; stack[top] = A;  
    在 stack[top] 的位置上记录之前算出的综合属性;  
} else .....
```

S 属性的文法的移进-归约实现

```

top = 0;
token = next_token();
while (true) {
    if (top == 0) {
        switch (token) {
            case LSQUARE:
                top++; stack[top] = LSQUARE;
                token = next_token(); break;
            case RSQUARE:
                return false;
            case EOF:
                top++; stack[top] = S;
                stack[top].cnt = 0; break;
        }
    } else if (top == 1 && stack[top] == S) {
        switch (token) {
            case LSQUARE: return false;
            case RSQUARE: return false;
            case EOF: return true;
        }
    } else if .....
}

```

$S' \rightarrow S \text{ EOF}$
 [1] $S \rightarrow \epsilon$
 [2] $S \rightarrow [S]S$

产生规则	属性计算规则
[0] $S' \rightarrow S \text{ EOF}$	$S'.cnt = S.cnt$
[1] $S \rightarrow \epsilon$	$S.cnt = 0$
[2] $S \rightarrow [S_1]S_2$	$S.cnt = 1 + S_1.cnt + S_2.cnt$

分析栈 \ 向前看	[	]	EOF
ϵ	移进	错误	归约 [1]
S	错误	错误	接受
$\beta[$	移进	归约 [1]	错误
$\beta[S$	错误	移进	错误
$\beta[S]$	移进	归约 [1]	归约 [1]
$\beta[S]S$	错误	归约 [2]	归约 [2]

S 属性的文法的移进-归约实现

```

.....
} else if (top >= 1 && stack[top] == LSQUARE) {
    switch (token) {
        case LSQUARE:
            top++; stack[top] = LSQUARE;
            token = next_token(); break;
        case RSQUARE:
            top++; stack[top] = S;
            stack[top].cnt = 0; break;
        case EOF:
            return false;
    }
} else if (top >= 2 && stack[top] == S &&
           stack[top - 1] == LSQUARE) {
    switch (token) {
        case LSQUARE: return false;
        case RSQUARE:
            top++; stack[top] = RSQUARE;
            token = next_token(); break;
        case EOF: return false;
    }
} else if .....

```

$S' \rightarrow S \text{ EOF}$
 [1] $S \rightarrow \epsilon$
 [2] $S \rightarrow [S]S$

产生规则	属性计算规则
[0] $S' \rightarrow S \text{ EOF}$	$S'.cnt = S.cnt$
[1] $S \rightarrow \epsilon$	$S.cnt = 0$
[2] $S \rightarrow [S_1]S_2$	$S.cnt = 1 + S_1.cnt + S_2.cnt$

分析栈 \ 向前看	[	]	EOF
ϵ	移进	错误	归约 [1]
S	错误	错误	接受
$\beta[$	移进	归约 [1]	错误
$\beta[S$	错误	移进	错误
$\beta[S]$	移进	归约 [1]	归约 [1]
$\beta[S]S$	错误	归约 [2]	归约 [2]

S 属性的文法的移进-归约实现

$$S' \rightarrow S \text{ EOF}$$

[1] $S \rightarrow \epsilon$

[2] $S \rightarrow [S]S$

```

.....
} else if (top >= 3 &&
           stack[top] == RSQUARE &&
           stack[top - 1] == S &&
           stack[top - 2] == LSQUARE) {
    switch (token) {
    case LSQUARE:
        top++; stack[top] = LSQUARE;
        token = next_token(); break;
    case RSQUARE:
        top++; stack[top] = S;
        stack[top].cnt = 0; break;
    case EOF:
        top++; stack[top] = S;
        stack[top].cnt = 0; break;
    }
} else if .....

```

产生规则	属性计算规则
[0] $S' \rightarrow S \text{ EOF}$	$S'.cnt = S.cnt$
[1] $S \rightarrow \epsilon$	$S.cnt = 0$
[2] $S \rightarrow [S_1]S_2$	$S.cnt = 1 + S_1.cnt + S_2.cnt$

分析栈 \ 向前看	[	]	EOF
ϵ	移进	错误	归约 [1]
S	错误	错误	接受
$\beta[$	移进	归约 [1]	错误
$\beta[S$	错误	移进	错误
$\beta[S]$	移进	归约 [1]	归约 [1]
$\beta[S]S$	错误	归约 [2]	归约 [2]

S 属性的文法的移进-归约实现

```

.....
} else if (top >= 4 && stack[top] == S &&
           stack[top - 1] == RSQUARE &&
           stack[top - 2] == S &&
           stack[top - 3] == LSQUARE) {
    switch (token) {
        case LSQUARE:
            return false;
        case RSQUARE:
            tmp = 1 + stack[top - 2].cnt +
                  stack[top].cnt;
            top = top - 3; stack[top] = S;
            stack[top].cnt = tmp; break;
        case EOF:
            tmp = 1 + stack[top - 2].cnt +
                  stack[top].cnt;
            top = top - 3; stack[top] = S;
            stack[top].cnt = tmp; break;
    }
} else return false;
}

```

$S' \rightarrow S \text{ EOF}$
 [1] $S \rightarrow \epsilon$
 [2] $S \rightarrow [S]S$

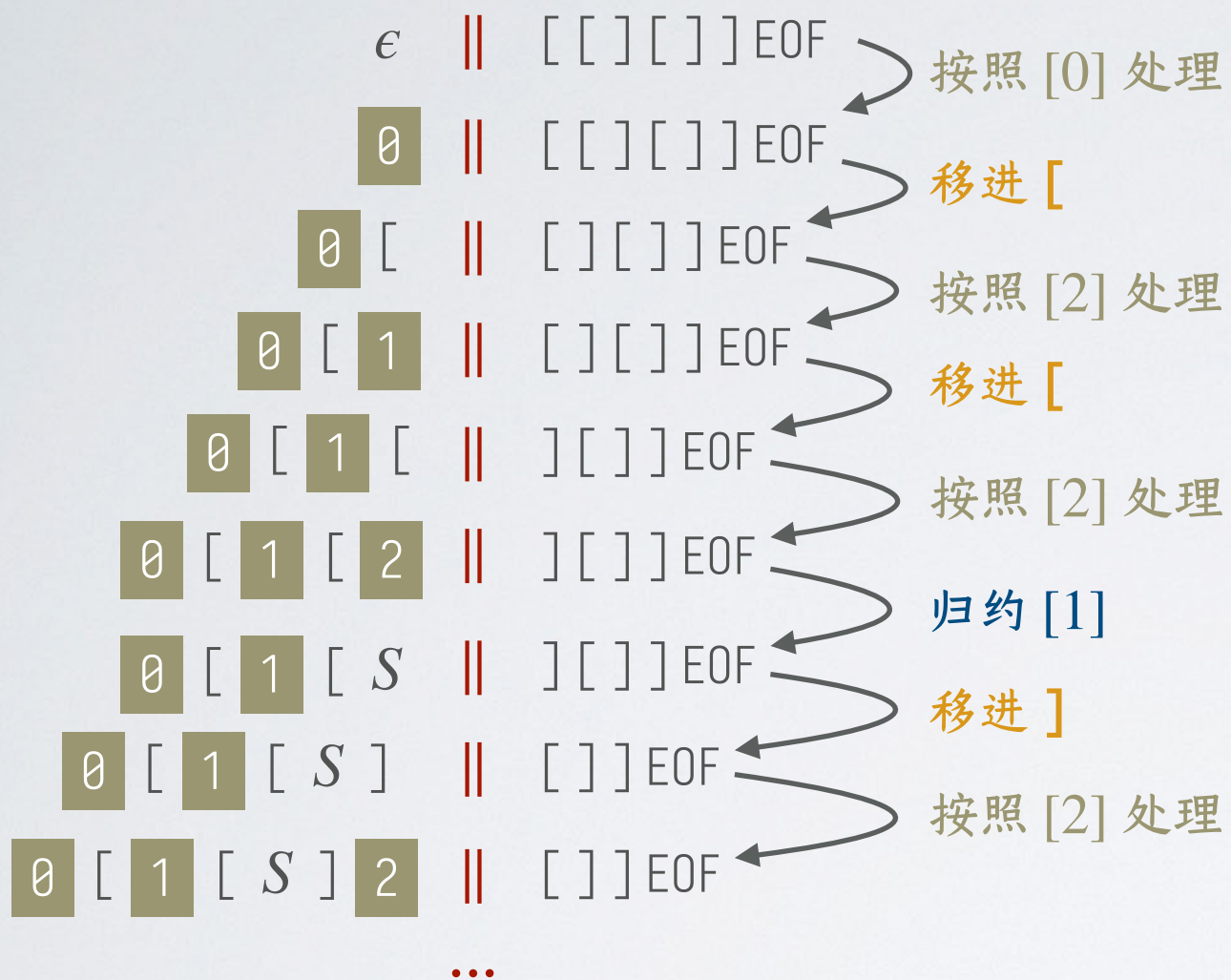
产生规则	属性计算规则
[0] $S' \rightarrow S \text{ EOF}$	$S'.cnt = S.cnt$
[1] $S \rightarrow \epsilon$	$S.cnt = 0$
[2] $S \rightarrow [S_1]S_2$	$S.cnt = 1 + S_1.cnt + S_2.cnt$

分析栈 \ 向前看	[	]	EOF
ϵ	移进	错误	归约 [1]
S	错误	错误	接受
$\beta[$	移进	归约 [1]	错误
$\beta[S$	错误	移进	错误
$\beta[S]$	移进	归约 [1]	归约 [1]
$\beta[S]S$	错误	归约 [2]	归约 [2]

L 属性的文法的移进-归约实现

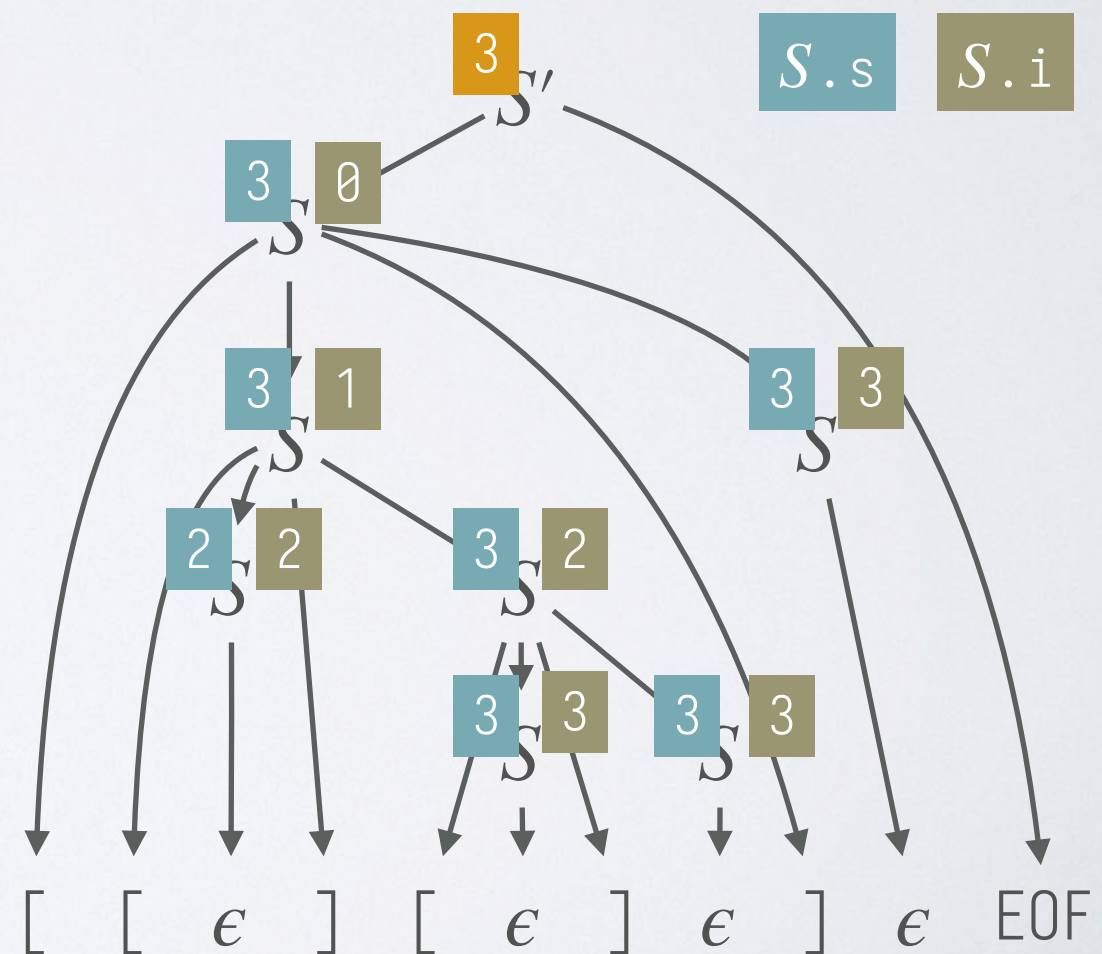
- ◎ 回顾: L 属性的文法中既有综合属性也有继承属性
 - ❖ 每个继承属性可以依赖左侧结点的属性和 parent 结点的继承属性
 - ❖ 问题: 继承属性的信息流向可以自顶向下, 与语法分析的方向不一致
- ◎ 如果文法本身是 LL 的呢?
 - ❖ 从左往右处理 token 时, 根据向前看符号就能确定展开规则
 - ❖ 自底向上分析在移进符号时, 有足够信息获得这个展开规则
- ◎ 以 LL 文法为基础的 L 属性的文法可以通过移进-归约实现
 - ❖ 合理安排分析栈, 需要继承属性时可以在分析栈的特定位置找到

自底向上的带继承属性的语义分析



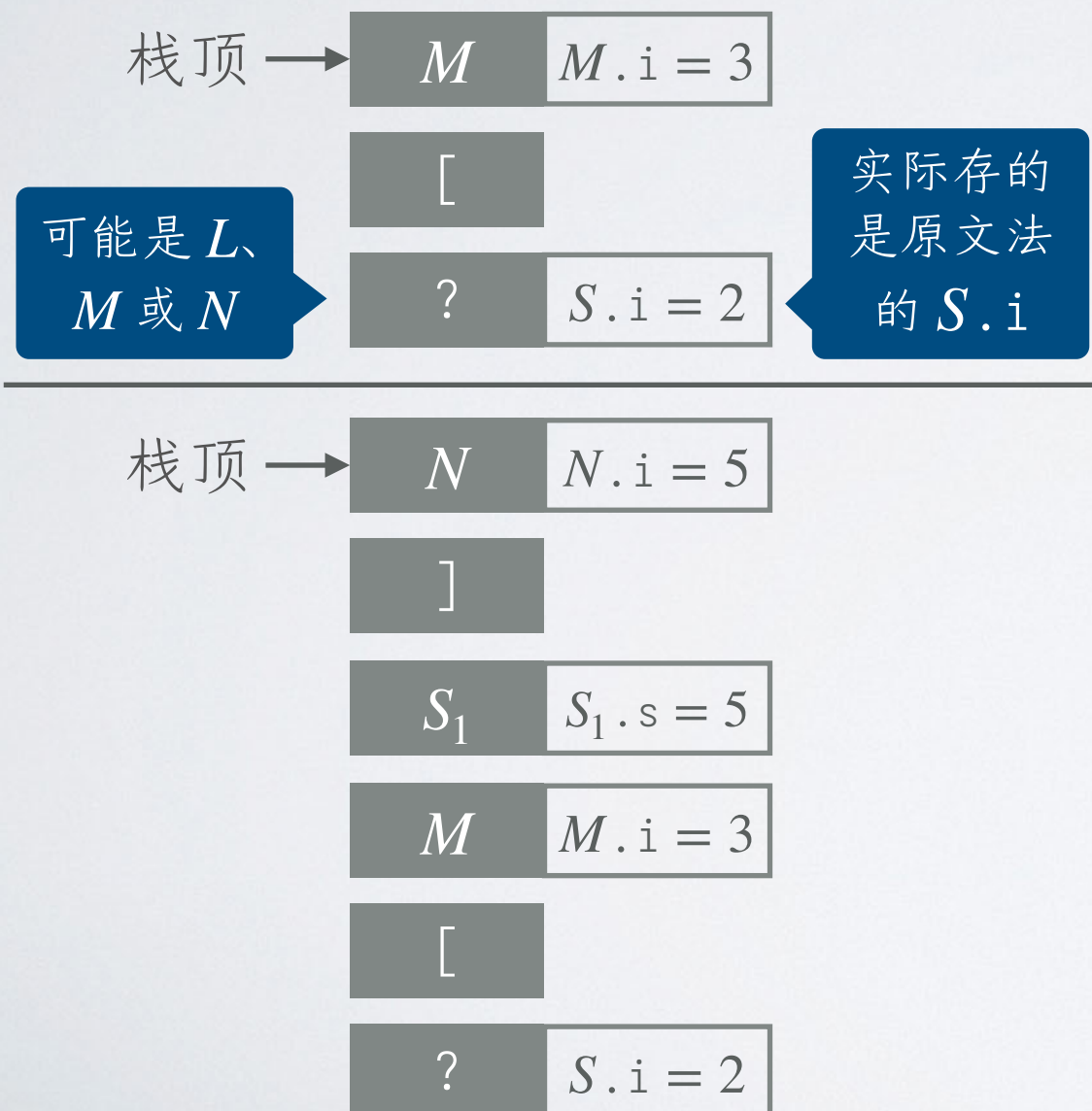
即将展开非终结符号 S 时, 在分析栈中(将来进行归约的位置的下方)存储 S 的继承属性

产生规则	属性计算规则
[0] $S' \rightarrow S \text{ EOF}$	$S.i = 0$ $S'.s = S.s$
[1] $S \rightarrow \epsilon$	$S.s = S.i$
[2] $S \rightarrow [S_1]S_2$	$S_1.i = S.i + 1$ $S_2.i = S_1.s$ $S.s = S_2.s$



L 属性的文法的移进-归约实现

- 安排分析栈使得在**归约位置减 1 处**总能找到需要的继承属性
- 技巧: 引入**标记**非终结符号(如下面的 L 、 M 、 N)



产生规则	属性计算规则
$S' \rightarrow L S \text{ EOF}$	$S'.s = S.s$
$S \rightarrow \epsilon$	$S.s = \text{栈中归约位置减 1 处}.i$
$S \rightarrow [M S_1] N S_2$	$S.s = S_2.s$
$L \rightarrow \epsilon$	$L.i = 0$
$M \rightarrow \epsilon$	$M.i = \text{栈中归约位置减 2 处}.i + 1$
$N \rightarrow \epsilon$	$N.i = \text{栈中归约位置减 2 处}.s$

L 属性的文法的移进-归约实现

- 在 LL 文法中插入这些标记非终结符号 (如下面的 L 、 M 、 N) 后, 得到的文法是一个 LR 文法, 可以用移进-归约实现

```
top++; stack[top] = S;  
stack[top].s = stack[top - 1].i;
```

```
top++; stack[top] = M;  
stack[top].i = stack[top - 2].i + 1;
```

```
top++; stack[top] = N;  
stack[top].i = stack[top - 2].s;
```

产生规则	属性计算规则
$S' \rightarrow L S \text{ EOF}$	$S'.s = S.s$
$S \rightarrow \epsilon$	$S.s = \text{栈中归约位置减 1 处}.i$
$S \rightarrow [M S_1] N S_2$	$S.s = S_2.s$
$L \rightarrow \epsilon$	$L.i = 0$
$M \rightarrow \epsilon$	$M.i = \text{栈中归约位置减 2 处}.i + 1$
$N \rightarrow \epsilon$	$N.i = \text{栈中归约位置减 2 处}.s$

- 注意: 严格来说这种属性计算规则并不规范, 但可以纳入后面会讲的语法制导的翻译方案中

主要内容

- ◎ 自底向上的语义分析
- ◎ **语义分析的自动生成**
- ◎ 类型系统(基础版)

从规约到实现的转换

◎ 语义分析的规约：属性文法

- ❖ 也称为语法制导定义 (Syntax-Directed Definition, SDD)

◎ 语义分析的实现：

- ❖ 独立于语法分析实现：在语法分析树上进行遍历
- ❖ 基于语法分析进行实现：S 属性的文法、L 属性的文法

◎ 语义分析的自动生成：

- ❖ 自动化 S 或 L 属性的文法到实现的转换
- ❖ 基于表驱动的语法分析进行实现
- ❖ **问题：**属性文法中给出的属性计算规则不一定与实际计算顺序一致

语法制导的翻译方案

- ◎ Syntax-Directed Translation, SDT
- ◎ 与属性文法相比, SDT 把属性计算规则改写为计算属性值的**程序片段**, 并用花括号 { } 括起来, 插入到产生规则**右侧**的任何合适的位置上
- ◎ 本质是一种对**语法分析**和**语义动作**交错的表示方法
 - ❖ 理解为深度优先遍历语法分析树时, 何时执行语义动作
- ◎ 与属性文法相比, SDT 明确了属性值的计算顺序

SDT 示例：四则运算表达式

- 后缀翻译方案：每个语义动作都在产生规则的最后
- 对于 S 属性的文法，SDT 看起来跟属性文法差不多

产生规则	语义动作
$Expr \rightarrow Expr_1 + Term$	{ $Expr.val = Expr_1.val + Term.val$; }
$Expr \rightarrow Expr_1 - Term$	{ $Expr.val = Expr_1.val - Term.val$; }
$Expr \rightarrow Term$	{ $Expr.val = Term.val$; }
$Term \rightarrow Term_1 * Factor$	{ $Term.val = Term_1.val * Factor.val$; }
$Term \rightarrow Term_1 / Factor$	{ $Term.val = Term_1.val / Factor.val$; }
$Term \rightarrow Factor$	{ $Term.val = Factor.val$; }
$Factor \rightarrow (Expr)$	{ $Factor.val = Expr.val$; }
$Factor \rightarrow INT$	{ $Factor.val = INT.intval$; }

SDT 示例：四则运算表达式

- 语义动作可以出现在产生规则中间, 比如 $B \rightarrow X \{ a \} Y$
- 对于 L 属性的文法, SDT 在中间插入的动作可以操作继承属性

产生规则	语义动作
$Expr \rightarrow Term$ $Expr'$	$\{ Expr'.inh = Term.val; \}$ $\{ Expr.val = Expr'.syn; \}$
$Expr' \rightarrow + Term$ $Expr'_1$	$\{ Expr'_1.inh = Expr'.inh + Term.val; \}$ $\{ Expr'.syn = Expr'_1.syn; \}$
$Expr' \rightarrow \epsilon$	$\{ Expr'.syn = Expr'.inh; \}$
$Term \rightarrow Factor$ $Term'$	$\{ Term'.inh = Factor.val; \}$ $\{ Term.val = Term'.syn; \}$
$Term' \rightarrow * Factor$ $Term'_1$	$\{ Term'_1.inh = Term'.inh * Factor.val; \}$ $\{ Term'.syn = Term'_1.syn; \}$
$Term' \rightarrow \epsilon$	$\{ Term'.syn = Term'.inh; \}$
$Factor \rightarrow (Expr)$	$\{ Factor.val = Expr.val; \}$
$Factor \rightarrow INT$	$\{ Factor.val = INT.intval; \}$

也可以写在一行上:

$E \rightarrow T \{ E'.inh = T.val; \} E' \{ E.val = E'.syn; \}$

回顾: 按照深度优先遍历语法
分析树的顺序插入语义动作

从结点 E 遍历其孩子结点, 先访问结点 T , 结束后用综合属性 $T.val$ 设置下一个孩子结点的继承属性 $E'.inh$, 然后访问结点 E' , 结束后把综合属性 $E'.syn$ 作为结点 E 的综合属性返回

SDT 示例：四则运算表达式

- ◎ 语义动作也可以不计算属性，而是直接产生**副作用**
- ◎ 下面的 SDT 把中缀表达式转换成等价的前缀表达式输出
 - ❖ 比如输入 $9+5*2$, 输出 $+ 9 * 5 2$

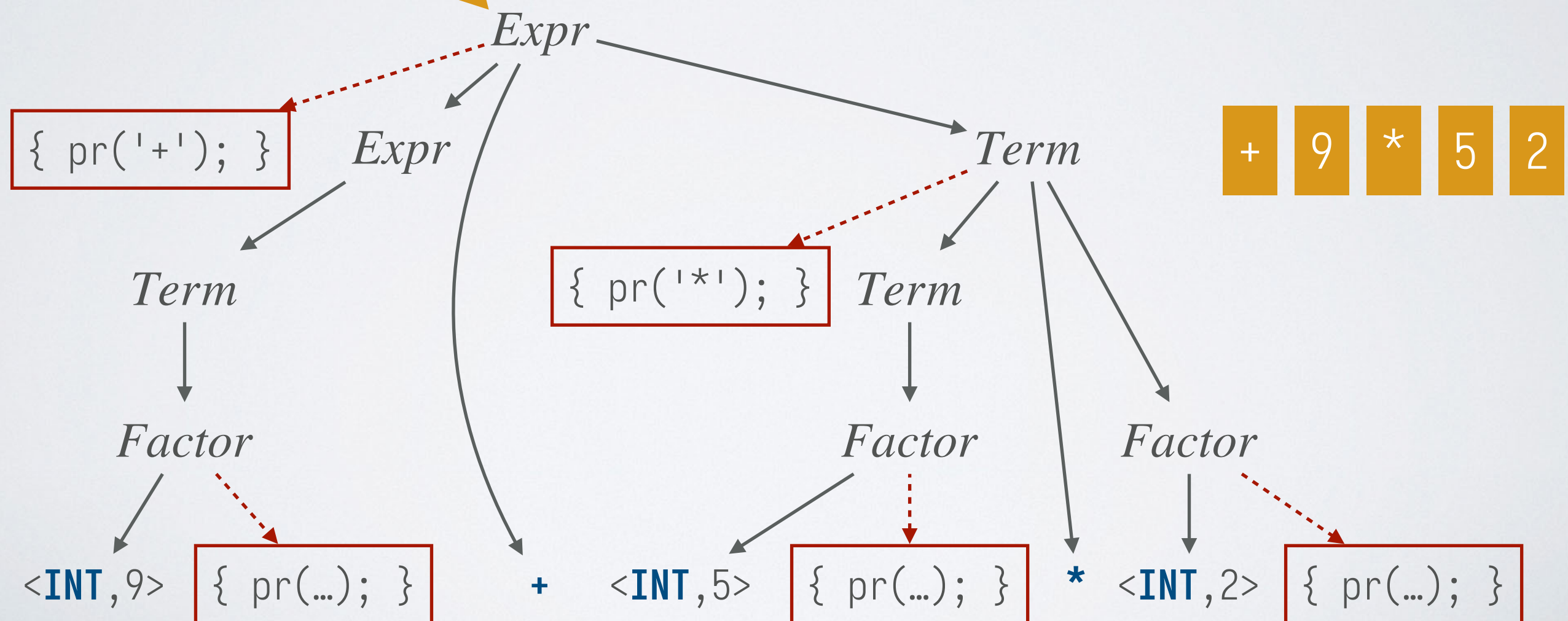
产生规则	语义动作
$Expr \rightarrow$ $Expr_1 + Term$	{ print('+'); }
$Expr \rightarrow Term$	
$Term \rightarrow$ $Term_1 * Factor$	{ print('*'); }
$Term \rightarrow Factor$	
$Factor \rightarrow (Expr)$	
$Factor \rightarrow INT$	{ print(INT.lexeme); }

也可以写在一行上：
 $Expr \rightarrow \{ \text{print('+'); } \} Expr_1 + Term$

嵌入语义动作的语法分析树

◎ SDT 的通用实现方法:

- ❖ 忽略语义动作, 建立语法分析树
- ❖ 对每个内部结点 N , 把它对应产生规则中的动作作为 N 的孩子结点加入
- ❖ 对这棵分析树进行前序遍历, 在访问到动作结点时则立即执行该动作



基于语法分析实现 SDT

◎ 回顾:

❖ LL 文法可以通过递归下降法实现

❖ 支持 S 或 L 属性的文法

❖ LR 文法可以通过移进-归约法实现

❖ 支持 S 属性的文法

❖ 支持以 LL 文法为基础的 L 属性的文法

◎ 回顾: LL 文法和 LR 文法都可以用表驱动的语法分析实现

◎ 基于语法分析实现 SDT:

❖ 把 S 或 L 属性的文法改写为带语义动作的 SDT

❖ 把 SDT 中的语义动作嵌入表驱动的语法分析中

回顾：表驱动的 LL(1) 分析

S' || $[[[]]] EOF$
 $EOF S$ || $[[[]]] EOF$
 $EOF S]S[$ || $[[[]]] EOF$
 $EOF S]S$ || $[[[]]] EOF$
 $EOF S]S]S[$ || $[[[]]] EOF$
 $EOF S]S]S$ || $] [[]] EOF$
 $EOF S]S]$ || $] [[]] EOF$
 $EOF S]S$ || $] [[]] EOF$
 $EOF S]S]S[$ || $] [[]] EOF$
 $EOF S]S]S$ || $] [[]] EOF$
 $EOF S]S]$ || $] [[]] EOF$
 $EOF S]S$ || $] [[]] EOF$
 $EOF S]$ || $] [[]] EOF$
 $EOF S$ || $] [[]] EOF$
 EOF || $] [[]] EOF$

$[0] \quad S' \rightarrow S EOF$
 $[1] \quad S \rightarrow \epsilon$
 $[2] \quad S \rightarrow [S] S$

使用一个栈来记录待匹配的
终结符号和非终结符号

分析栈 \ 向前看	[	]	EOF
$\beta S'$	展开 [0]	错误	展开 [0]
βS	展开 [2]	展开 [1]	展开 [1]
$\beta [$	匹配	错误	错误
$\beta]$	错误	匹配	错误
EOF	错误	错误	接受
ϵ	错误	错误	错误

回顾：表驱动的 LL(1) 分析

```

token = next_token();
top = 1; stack[top] = S';
while (true) {
    if (top == 1 &&
        stack[top] == EOF &&
        token == EOF) {
        return true;
    } else if (top > 0 && stack[top] == token) {
        top--; token = next_token();
    } else if (top > 0 &&
        table[stack[top]][token] ==
        A → B1 B2 ... Bk) {
        top--;
        for (int i = k; i >= 1; i--) {
            if (Bi != ε) {
                top++; stack[top] = Bi;
            }
        }
    } else {
        return false;
    }
}

```

[0]	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow \epsilon$
[2]	$S \rightarrow [S] S$

使用一个栈来记录待匹配的
终结符号和非终结符号

分析栈 \ 向前看	[	]	EOF
$\beta S'$	展开 [0]	错误	展开 [0]
βS	展开 [2]	展开 [1]	展开 [1]
$\beta [$	匹配	错误	错误
$\beta]$	错误	匹配	错误
EOF	错误	错误	接受
ϵ	错误	错误	错误

以 LL 文法为基础的表驱动语义分析

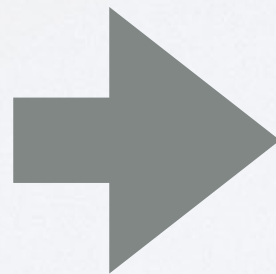
- ◎ 对于规则 $B \rightarrow X \{ a \} Y$, 在自顶向下的语法分析的过程中, 执行动作 a 的时机为:
 - ❖ 如果 Y 是非终结符号, 则在**展开 Y 之前**执行
 - ❖ 如果 Y 是终结符号, 则在**匹配 Y 之前**执行
- ◎ **如何在表驱动的语法分析中支持语义动作?**
- ◎ 需要解决两方面的问题:
 - ❖ 语义动作可能操作尚未分析的符号的继承属性
 - ❖ 展开规则时, 需要把语义动作存放在分析栈上

以 LL 文法为基础的表驱动语义分析

◎ 对于 S 或 L 属性的 LL 文法, 转换为 SDT 时:

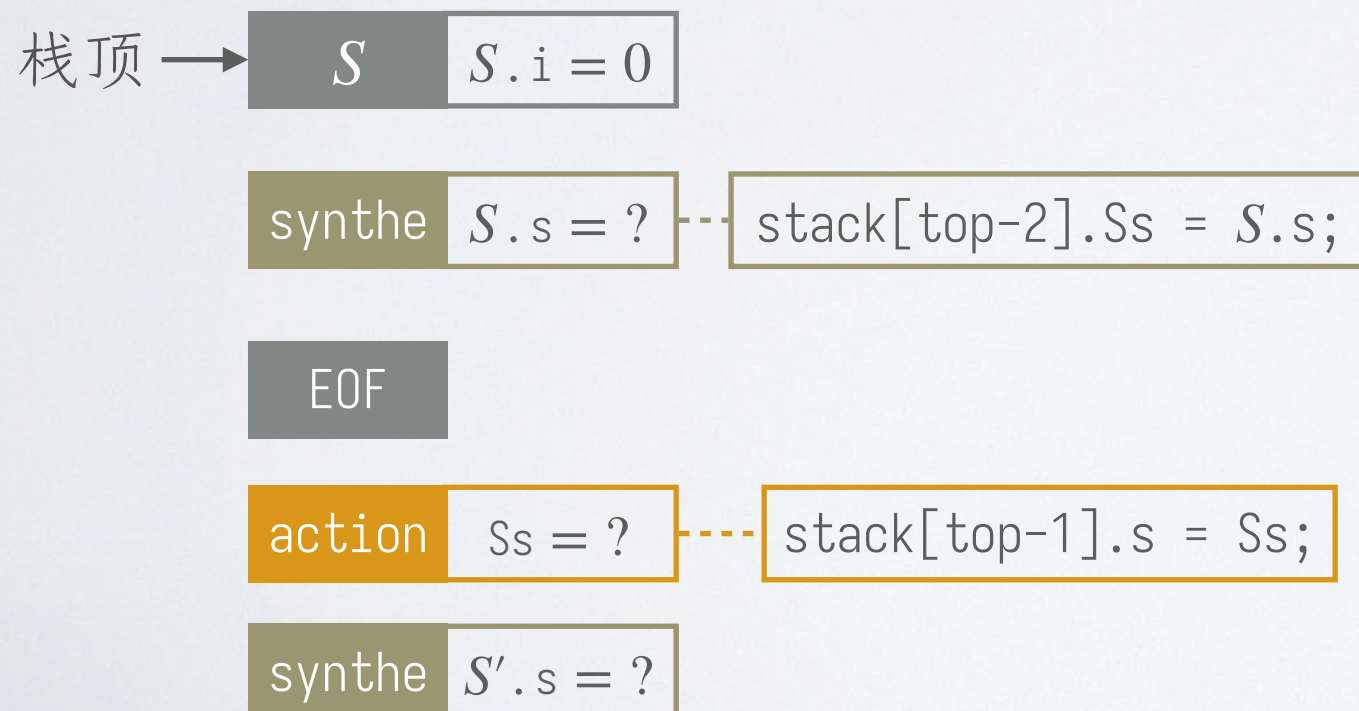
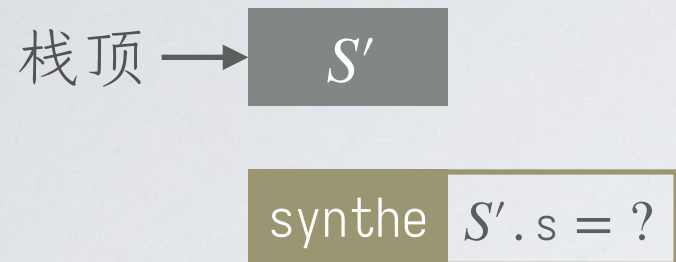
- ❖ 计算某个非终结符号 A 的继承属性的动作插入到产生规则中**紧靠在 A 的本次出现之前的位置**
- ❖ 计算产生规则左侧的符号的综合属性的动作插入到**该产生规则的最右侧**

产生规则	属性计算规则
[0] $S' \rightarrow S \text{ EOF}$	$S.i = 0$ $S'.s = S.s$
[1] $S \rightarrow \epsilon$	$S.s = S.i$
[2] $S \rightarrow [S_1]S_2$	$S_1.i = S.i + 1$ $S_2.i = S_1.s$ $S.s = S_2.s$



产生规则	语义动作
[0] $S' \rightarrow$ $S \text{ EOF}$	$\{ S.i = 0; \}$ $\{ S'.s = S.s; \}$
[1] $S \rightarrow \epsilon$	$\{ S.s = S.i; \}$
[2] $S \rightarrow [$ $S_1]$ S_2	$\{ S_1.i = S.i + 1; \}$ $\{ S_2.i = S_1.s; \}$ $\{ S.s = S_2.s; \}$

以 LL 文法为基础的表驱动语义分析



语法分析栈中需要保存综合记录

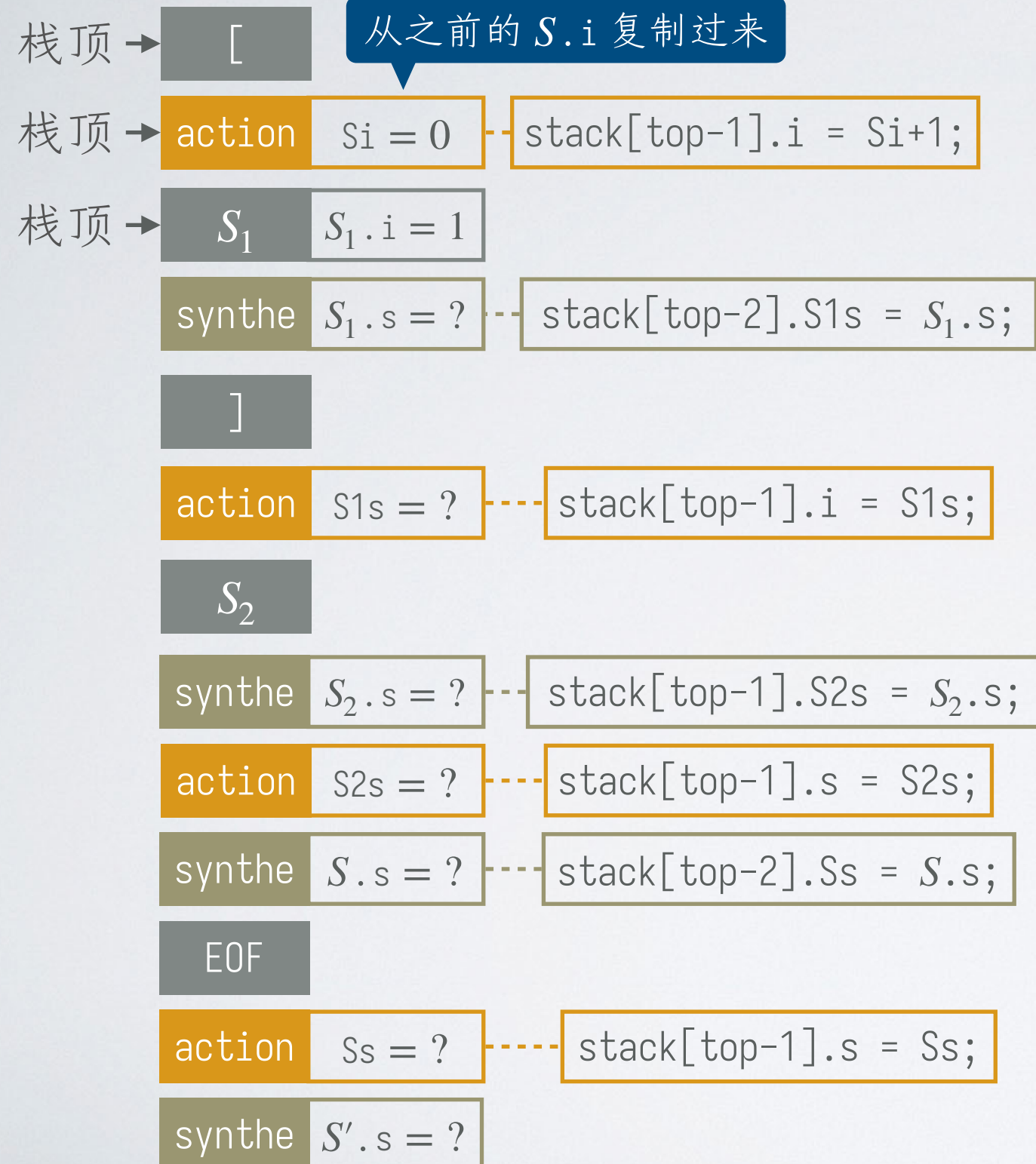
- ❖ 保存非终结符号的综合属性值
- ❖ 它总是紧靠在对应的非终结符号的下方

语法分析栈中需要保存动作记录

- ❖ 保存该动作需要的其它地方计算的属性值
- ❖ 保存该动作的代码, 可以实现为函数指针
- ❖ 栈中符号相对位置是可以静态确定的

产生规则	语义动作
[0] $S' \rightarrow$ $S \text{ EOF}$	$\{ S.i = 0; \}$ $\{ S'.s = S.s; \}$
[1] $S \rightarrow \epsilon$	$\{ S.s = S.i; \}$
[2] $S \rightarrow [$ $S_1]$ S_2	$\{ S_1.i = S.i + 1; \}$ $\{ S_2.i = S_1.s; \}$ $\{ S.s = S_2.s; \}$

以 LL 文法为基础的表驱动语义分析



语法分析栈中需要保存综合记录

- ❖ 保存非终结符号的综合属性值
- ❖ 它总是紧靠在对应的非终结符号的下方

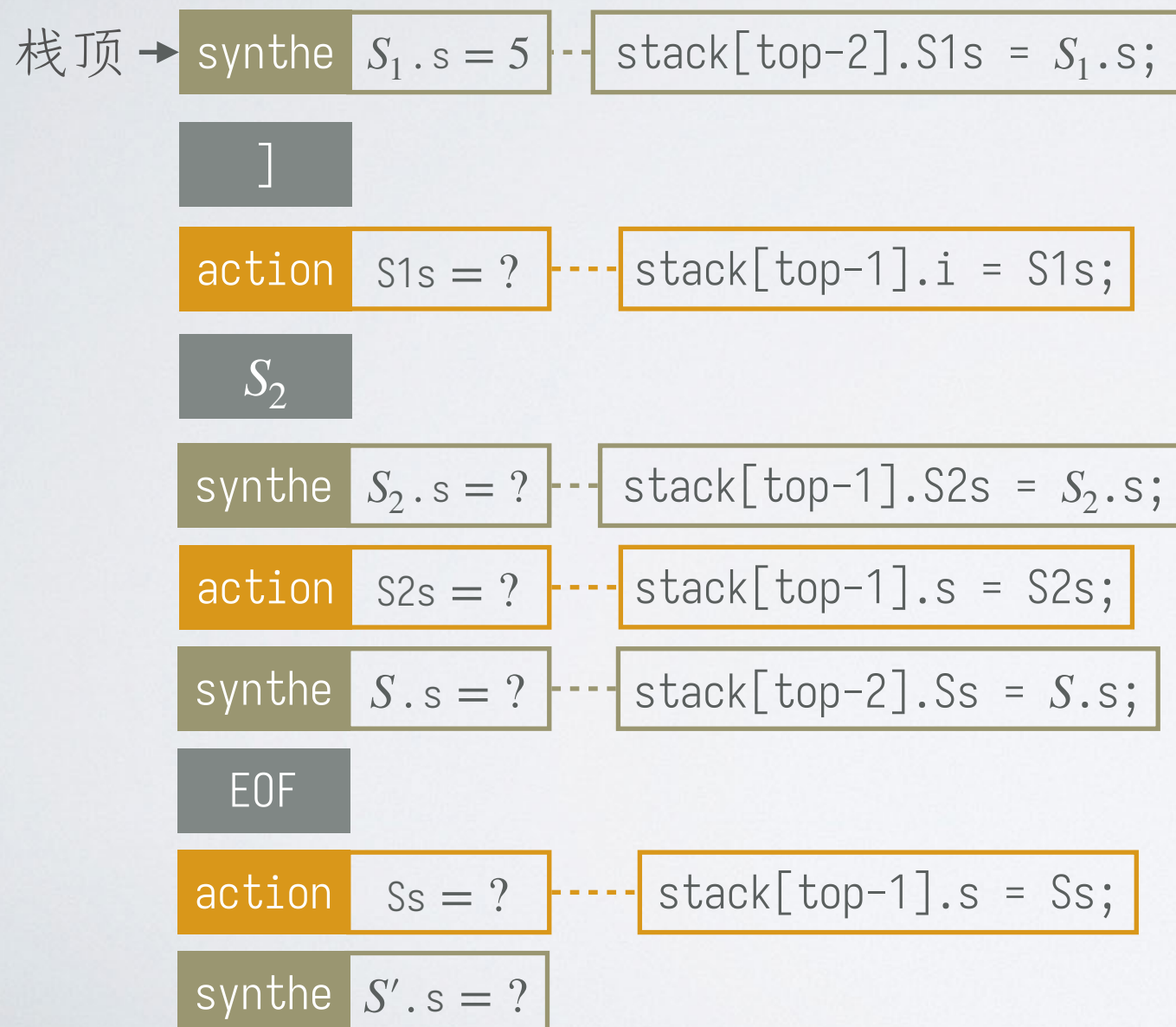
语法分析栈中需要保存动作记录

- ❖ 保存该动作需要的其它地方计算的属性值
- ❖ 保存该动作的代码, 可以实现为函数指针
- ❖ 栈中符号相对位置是可以静态确定的

产生规则	语义动作
[0] $S' \rightarrow$	{ $S.i = 0;$ }
$S EOF$	{ $S'.s = S.s;$ }
[1] $S \rightarrow \epsilon$	{ $S.s = S.i;$ }
[2] $S \rightarrow [$	{ $S_1.i = S.i + 1;$ }
$S_1]$	{ $S_2.i = S_1.s;$ }
S_2	{ $S.s = S_2.s;$ }

以 LL 文法为基础的表驱动语义分析

- 语法分析栈中需要保存**综合记录**
 - 保存非终结符号的综合属性值
 - 它总是紧靠在对应的非终结符号的下方



- 语法分析栈中需要保存**动作记录**
 - 保存该动作需要的其它地方计算的属性值
 - 保存该动作的代码, 可以实现为函数指针
 - 栈中符号相对位置是可以静态确定的

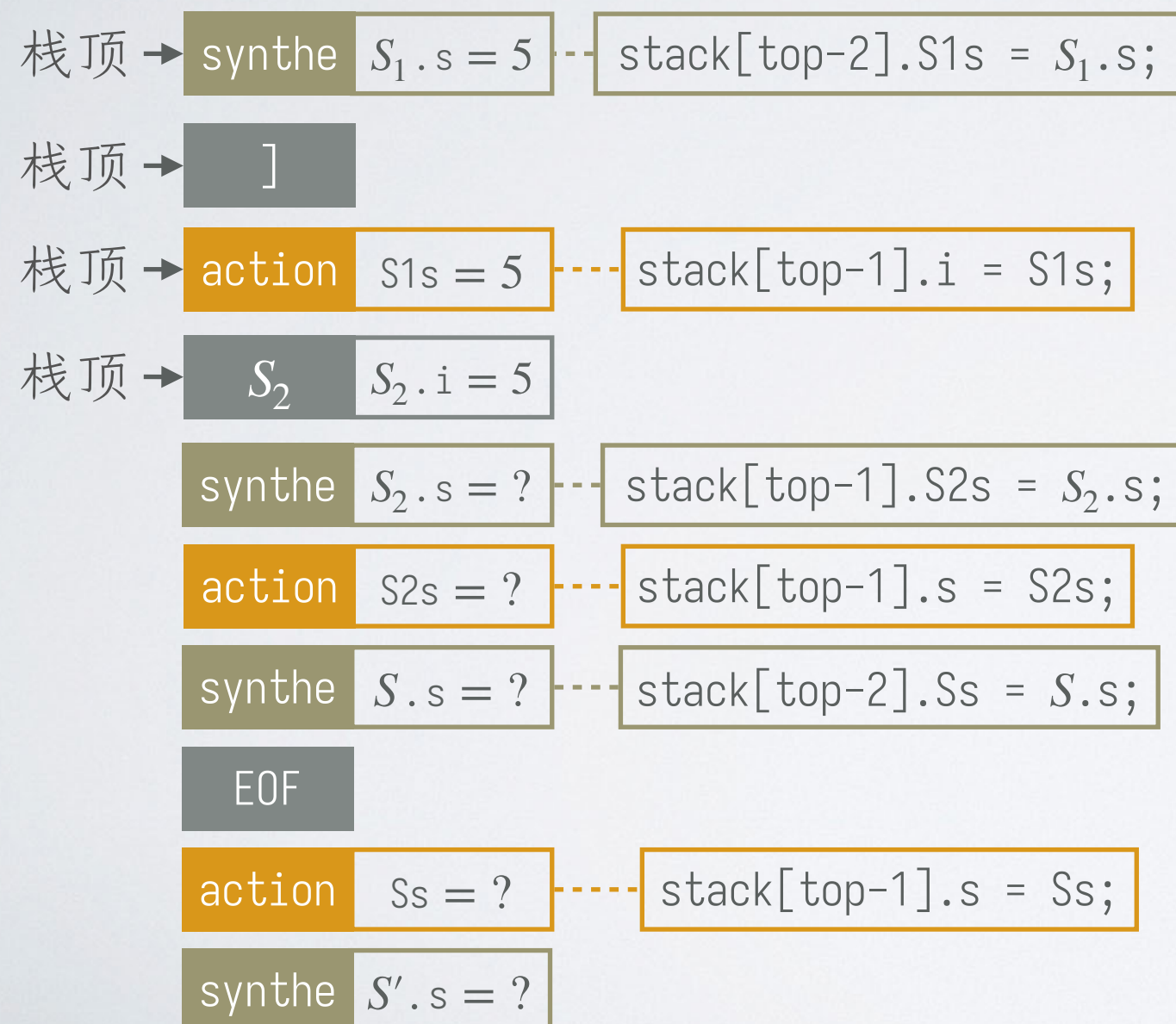
产生规则	语义动作
[0] $S' \rightarrow$	$\{ S.i = 0; \}$
$S EOF$	$\{ S'.s = S.s; \}$
[1] $S \rightarrow \epsilon$	$\{ S.s = S.i; \}$
[2] $S \rightarrow [$	$\{ S_1.i = S.i + 1; \}$
$S_1]$	$\{ S_2.i = S_1.s; \}$
S_2	$\{ S.s = S_2.s; \}$

以 LL 文法为基础的表驱动语义分析

- 语法分析栈中需要保存**综合记录**
 - 保存非终结符号的综合属性值
 - 它总是紧靠在对应的非终结符号的下方

- 语法分析栈中需要保存**动作记录**

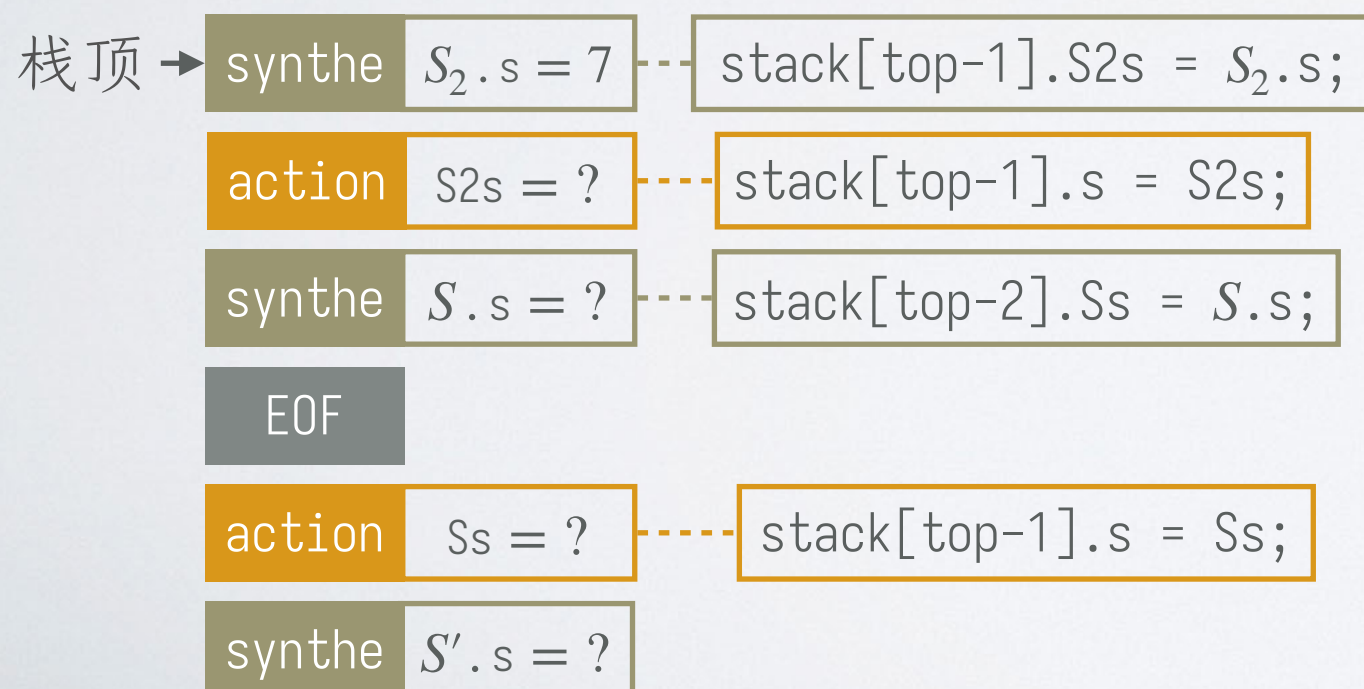
- 保存该动作需要的其它地方计算的属性值
- 保存该动作的代码, 可以实现为函数指针
- 栈中符号相对位置是可以静态确定的



产生规则	语义动作
[0] $S' \rightarrow$	{ $S.i = 0;$ }
$S EOF$	{ $S'.s = S.s;$ }
[1] $S \rightarrow \epsilon$	{ $S.s = S.i;$ }
[2] $S \rightarrow [$	{ $S_1.i = S.i + 1;$ }
$S_1]$	{ $S_2.i = S_1.s;$ }
S_2	{ $S.s = S_2.s;$ }

以 LL 文法为基础的表驱动语义分析

- 语法分析栈中需要保存**综合记录**
 - 保存非终结符号的综合属性值
 - 它总是紧靠在对应的非终结符号的下方
- 语法分析栈中需要保存**动作记录**
 - 保存该动作需要的其它地方计算的属性值
 - 保存该动作的代码, 可以实现为函数指针
 - 栈中符号相对位置是可以静态确定的



产生规则	语义动作
[0] $S' \rightarrow$	{ $S.i = 0$; }
$S EOF$	{ $S'.s = S.s$; }
[1] $S \rightarrow \epsilon$	{ $S.s = S.i$; }
[2] $S \rightarrow [$	{ $S_1.i = S.i + 1$; }
$S_1]$	{ $S_2.i = S_1.s$; }
S_2	{ $S.s = S_2.s$; }

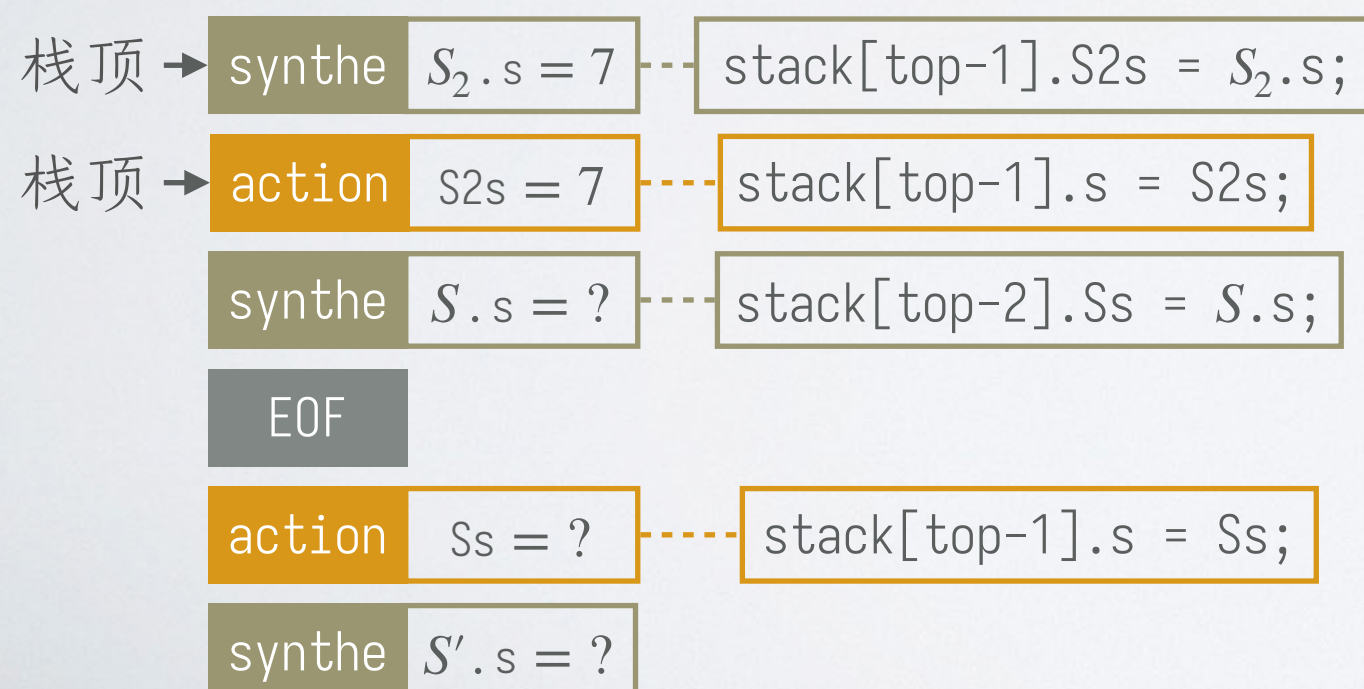
以 LL 文法为基础的表驱动语义分析

● 语法分析栈中需要保存综合记录

- ❖ 保存非终结符号的综合属性值
- ❖ 它总是紧靠在对应的非终结符号的下方

● 语法分析栈中需要保存动作记录

- ❖ 保存该动作需要的其它地方计算的属性值
- ❖ 保存该动作的代码, 可以实现为函数指针
- ❖ 栈中符号相对位置是可以静态确定的



产生规则	语义动作
[0] $S' \rightarrow$ $S EOF$	$\{ S.i = 0; \}$ $\{ S'.s = S.s; \}$
[1] $S \rightarrow \epsilon$	$\{ S.s = S.i; \}$
[2] $S \rightarrow [$ $S_1]$ S_2	$\{ S_1.i = S.i + 1; \}$ $\{ S_2.i = S_1.s; \}$ $\{ S.s = S_2.s; \}$

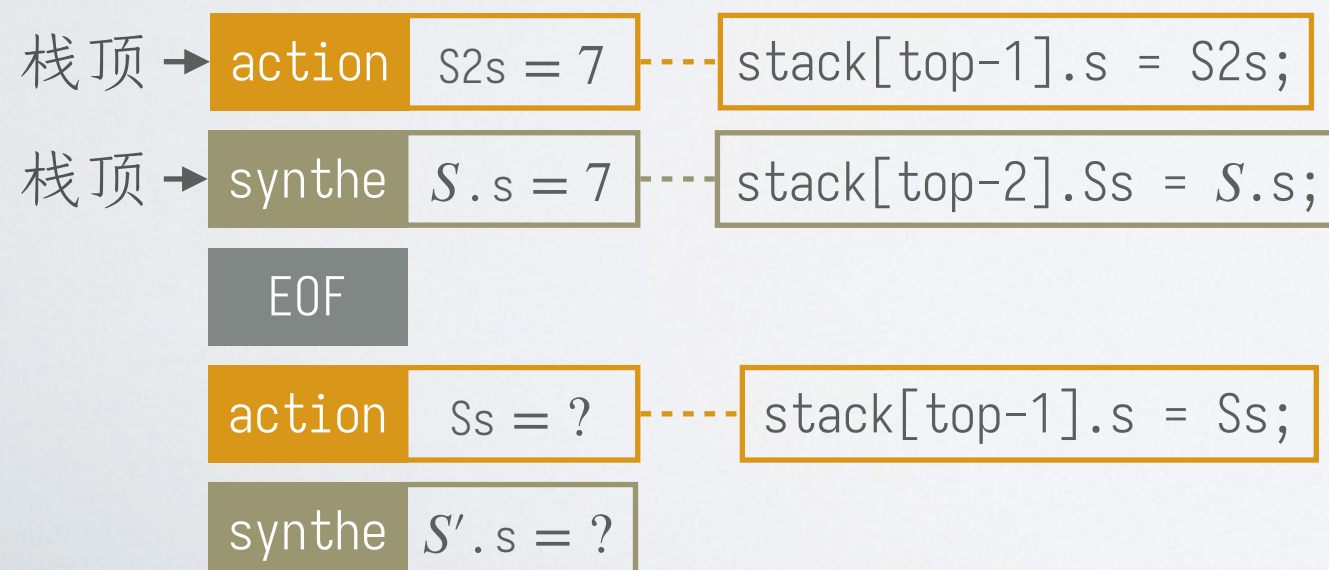
以 LL 文法为基础的表驱动语义分析

语法分析栈中需要保存综合记录

- ❖ 保存非终结符号的综合属性值
- ❖ 它总是紧靠在对应的非终结符号的下方

语法分析栈中需要保存动作记录

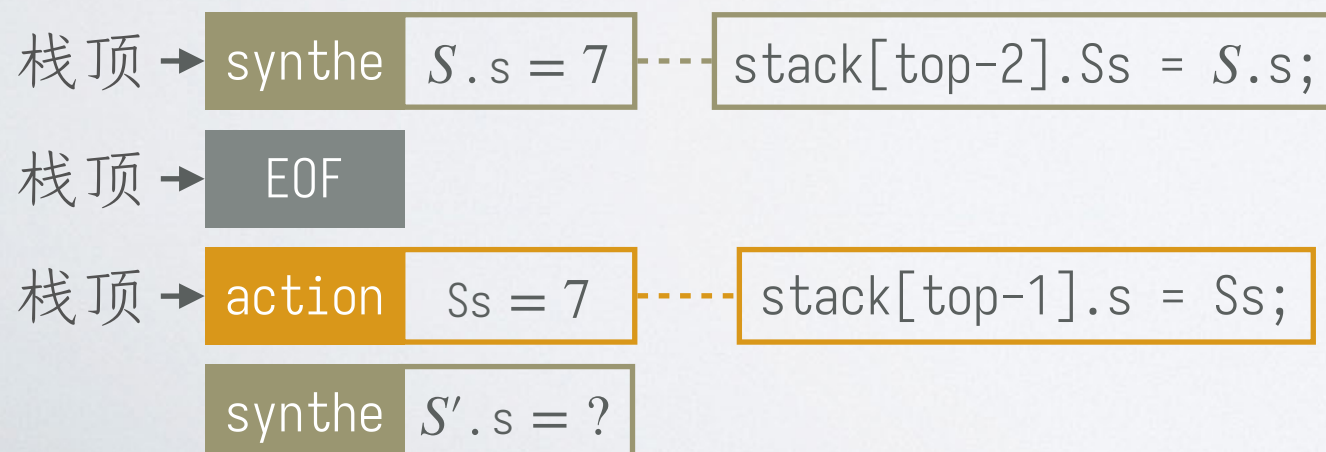
- ❖ 保存该动作需要的其它地方计算的属性值
- ❖ 保存该动作的代码, 可以实现为函数指针
- ❖ 栈中符号相对位置是可以静态确定的



产生规则	语义动作
[0] $S' \rightarrow$ $S \text{ EOF}$	$\{ S.i = 0; \}$ $\{ S'.s = S.s; \}$
[1] $S \rightarrow \epsilon$	$\{ S.s = S.i; \}$
[2] $S \rightarrow [$ $S_1]$ S_2	$\{ S_1.i = S.i + 1; \}$ $\{ S_2.i = S_1.s; \}$ $\{ S.s = S_2.s; \}$

以 LL 文法为基础的表驱动语义分析

- 语法分析栈中需要保存**综合记录**
 - 保存非终结符号的综合属性值
 - 它总是紧靠在对应的非终结符号的下方
- 语法分析栈中需要保存**动作记录**
 - 保存该动作需要的其它地方计算的属性值
 - 保存该动作的代码, 可以实现为函数指针
 - 栈中符号相对位置是可以静态确定的

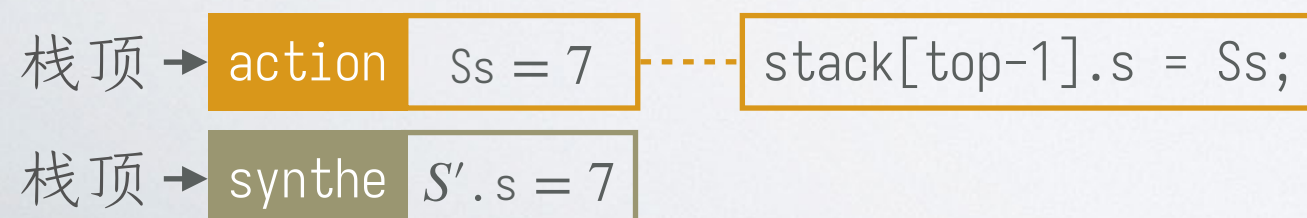


产生规则	语义动作
[0] $S' \rightarrow$ $S \text{ EOF}$	$\{ S.i = 0; \}$ $\{ S'.s = S.s; \}$
[1] $S \rightarrow \epsilon$	$\{ S.s = S.i; \}$
[2] $S \rightarrow [$ $S_1]$ S_2	$\{ S_1.i = S.i + 1; \}$ $\{ S_2.i = S_1.s; \}$ $\{ S.s = S_2.s; \}$

以 LL 文法为基础的表驱动语义分析

- 语法分析栈中需要保存综合记录
 - 保存非终结符号的综合属性值
 - 它总是紧靠在对应的非终结符号的下方
- 语法分析栈中需要保存动作记录
 - 保存该动作需要的其它地方计算的属性值
 - 保存该动作的代码, 可以实现为函数指针
 - 栈中符号相对位置是可以静态确定的

产生规则	语义动作
[0] $S' \rightarrow S \text{ EOF}$	$\{ S.i = 0; \}$ $\{ S'.s = S.s; \}$
[1] $S \rightarrow \epsilon$	$\{ S.s = S.i; \}$
[2] $S \rightarrow [S_1] S_2$	$\{ S_1.i = S.i + 1; \}$ $\{ S_2.i = S_1.s; \}$ $\{ S.s = S_2.s; \}$



LL(1) 文法的语义分析驱动程序

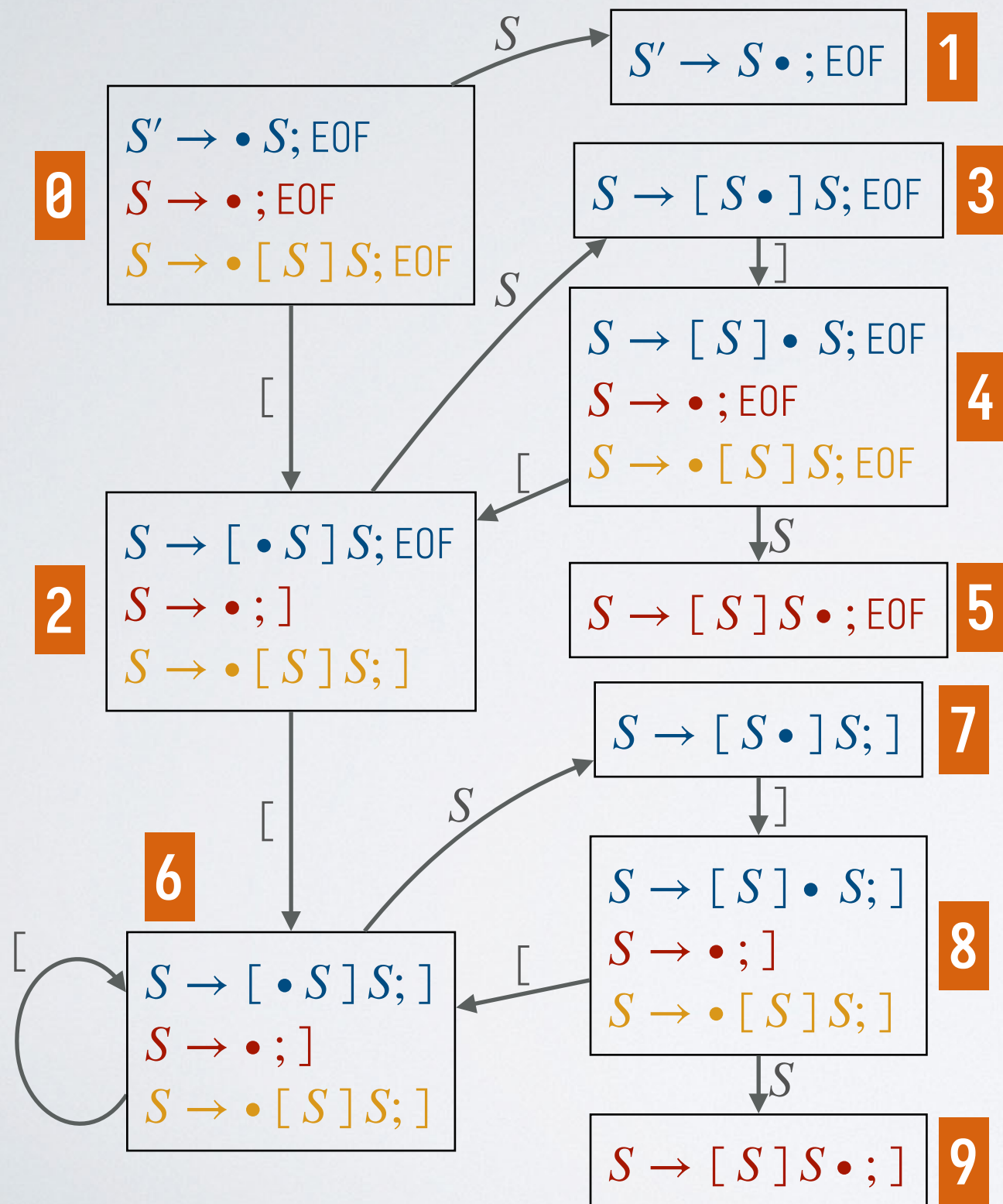
```
token = next_token();
top = 1; stack[top] = S';
while (true) {
    if (top > 0 && stack[top] 是动作/综合记录) {
        stack[top].action_func();
        top--;
    } else if (top > 0 && stack[top] 是非终结符号 && table[stack[top]][token] ==  $A \rightarrow B_1 B_2 \dots B_k$ ) {
        top--;
        for (int i = k; i >= 1; i--) {
            if ( $B_i \neq \epsilon$ ) {
                if ( $B_i$  是非终结符号) {
                    top++; stack[top] =  $B_i$  的综合记录;
                    top++; stack[top] =  $B_i$ ; }
                else if ( $B_i$  是语义动作  $act$ ) { top++; stack[top] =  $act$  的动作记录; }
                else { top++; stack[top] =  $B_i$ ; }
            }
        }
    } else {
        // ...
    }
}
```

记录中保存的动作代码(函数指针)可以操作全局变量 stack 和 top

存储 B_i 的综合属性, 如果后面的语义动作用到这些属性, 在对应的动作代码中复制这些属性到相应的位置

存储 act 会用到的属性, 并在动作代码中使用这些属性完成动作的计算, 把结果复制到相应的位置

回顾：表驱动的 LR(1) 分析



$S' \rightarrow S \text{ EOF}$

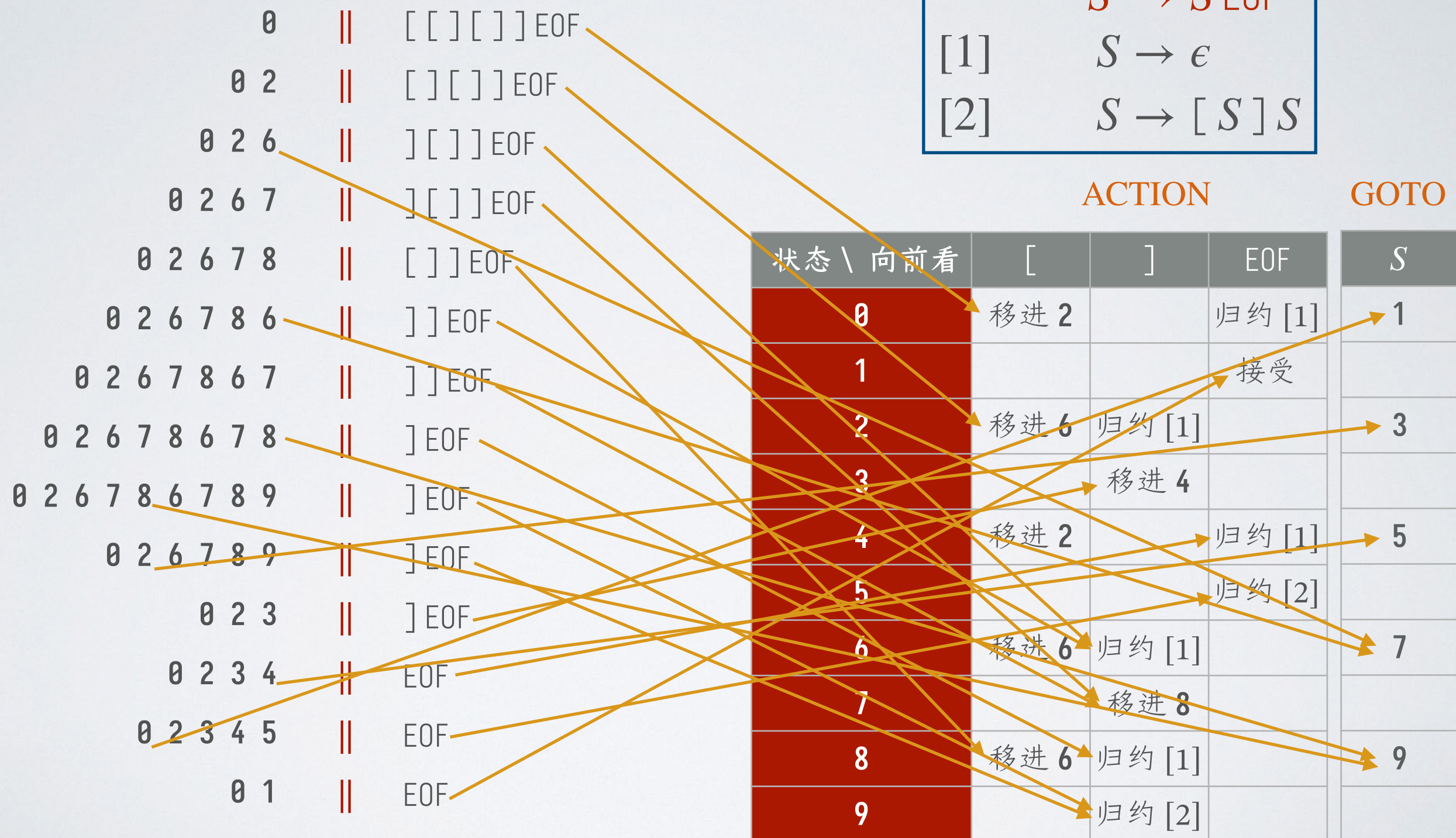
[1] $S \rightarrow \epsilon$

[2] $S \rightarrow [S] S$

状态 \ 向前看	ACTION			GOTO
	[	]	EOF	
0	移进 2		归约 [1]	1
1			接受	
2	移进 6	归约 [1]		3
3		移进 4		
4	移进 2		归约 [1]	5
5			归约 [2]	
6	移进 6	归约 [1]		7
7		移进 8		
8	移进 6	归约 [1]		9
9		归约 [2]		

回顾：表驱动的 LR(1) 分析

$S' \rightarrow S \text{ EOF}$
 $[1] \quad S \rightarrow \epsilon$
 $[2] \quad S \rightarrow [S]S$



回顾：表驱动的 LR(1) 分析

```

token = next_token();
top = 1; stack[top] = I0;
while (true) {
    state = stack[top];
    if (ACTION[state][token] == “归约 A → α”) {
        top = top - |α|;
        state = stack[top];
        top++; stack[top] = GOTO[state][A];
    } else if (ACTION[state][token] == “移进 Ij”) {
        top++; stack[top] = Ij;
        token = next_token();
    } else if (ACTION[state][token] == “接受”) {
        return true;
    } else {
        return false;
    }
}

```

$S' \rightarrow S \text{ EOF}$
 [1] $S \rightarrow \epsilon$
 [2] $S \rightarrow [S] S$

ACTION				GOTO
状态 \ 向前看	[	]	EOF	S
0	移进 2		归约 [1]	1
1			接受	
2	移进 6	归约 [1]		3
3		移进 4		
4	移进 2		归约 [1]	5
5			归约 [2]	
6	移进 6	归约 [1]		7
7		移进 8		
8	移进 6	归约 [1]		9
9		归约 [2]		

以 LR 文法为基础的表驱动语义分析

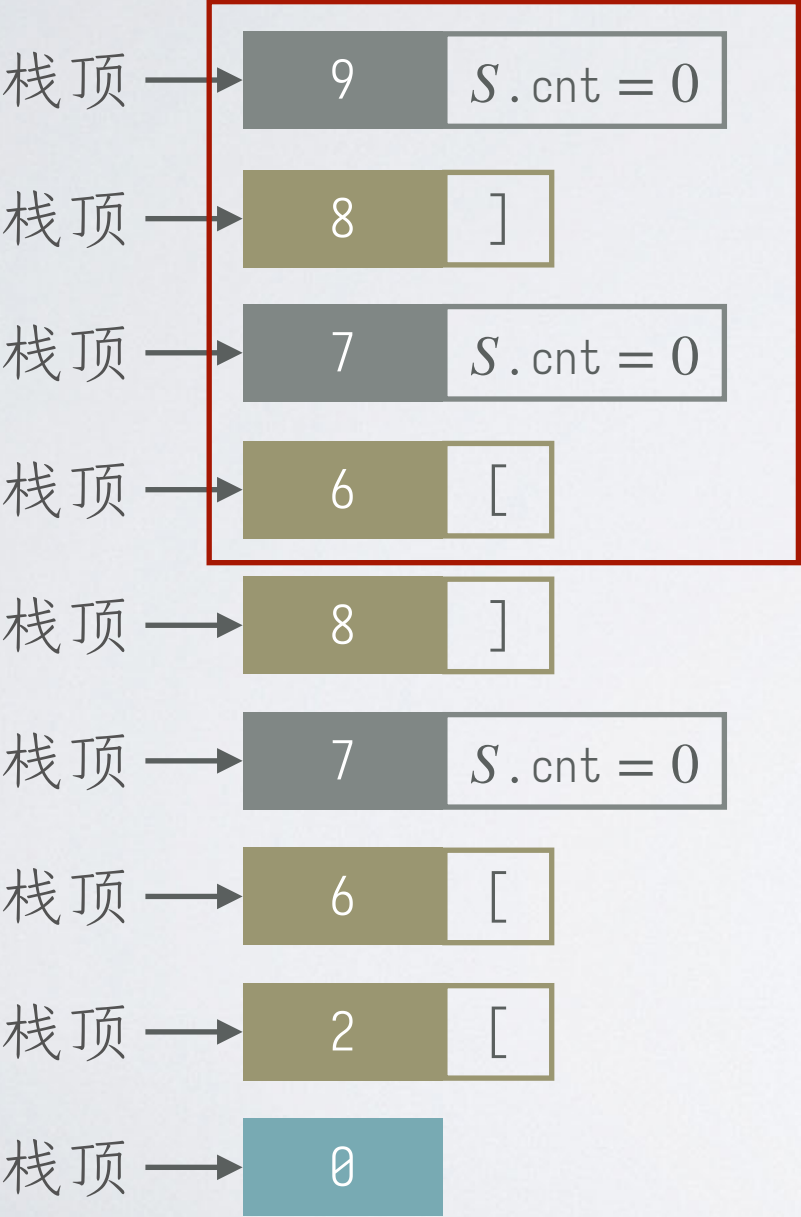
- ◎ 对于规则 $B \rightarrow X \{ a \} Y$, 在自底向上的语法分析的过程中, 执行动作 a 的时机为:
 - ❖ 当 X 的此处出现位于分析栈的栈顶时, 立即执行动作 a
- ◎ 如何知道是在进行 $B \rightarrow XY$ 的推导?
- ◎ 一般而言, 在进行归约时, 才能确定使用的推导规则
- ◎ **解决方案:** 要求 SDT 的语义动作都出现在产生规则的最右侧
 - ❖ 语义动作只会在归约的时候执行
 - ❖ 适用于 S 属性的文法
- ◎ 在分析栈中除了记录自动机状态, 还需记录各符号的属性值

以 LR 文法为基础的表驱动语义分析

[[[]]] EOF



	$S' \rightarrow S \text{ EOF}$	{ $S'.\text{cnt} = S.\text{cnt};$ }
[1]	$S \rightarrow \epsilon$	{ $S.\text{cnt} = 0;$ }
[2]	$S \rightarrow [S_1] S_2$	{ $S.\text{cnt} = 1 + S_1.\text{cnt} + S_2.\text{cnt};$ }

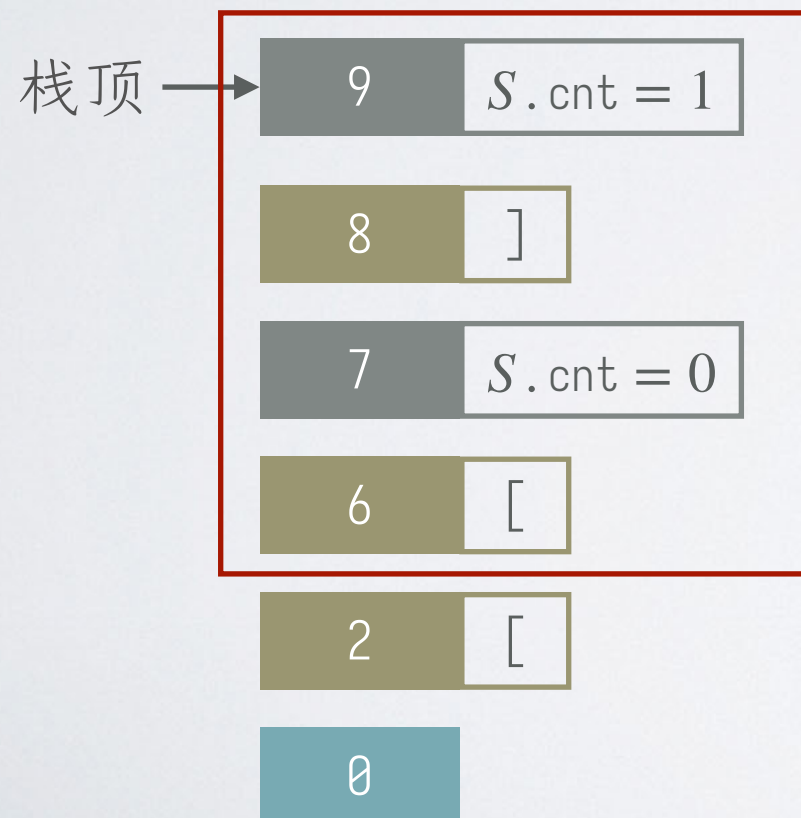


状态 \ 向前看	ACTION			GOTO
	[	]	EOF	
0	移进 2		归约 [1]	1
1			接受	
2	移进 6	归约 [1]		3
3		移进 4		
4	移进 2		归约 [1]	5
5			归约 [2]	
6	移进 6	归约 [1]		7
7		移进 8		
8	移进 6	归约 [1]		9
9		归约 [2]		

以 LR 文法为基础的表驱动语义分析

[[] []] EOF
↑

- [1] $S' \rightarrow S \text{ EOF}$ { $S'.\text{cnt} = S.\text{cnt};$ }
- [2] $S \rightarrow \epsilon$ { $S.\text{cnt} = 0;$ }
- [2] $S \rightarrow [S_1] S_2$ { $S.\text{cnt} = 1 + S_1.\text{cnt} + S_2.\text{cnt};$ }

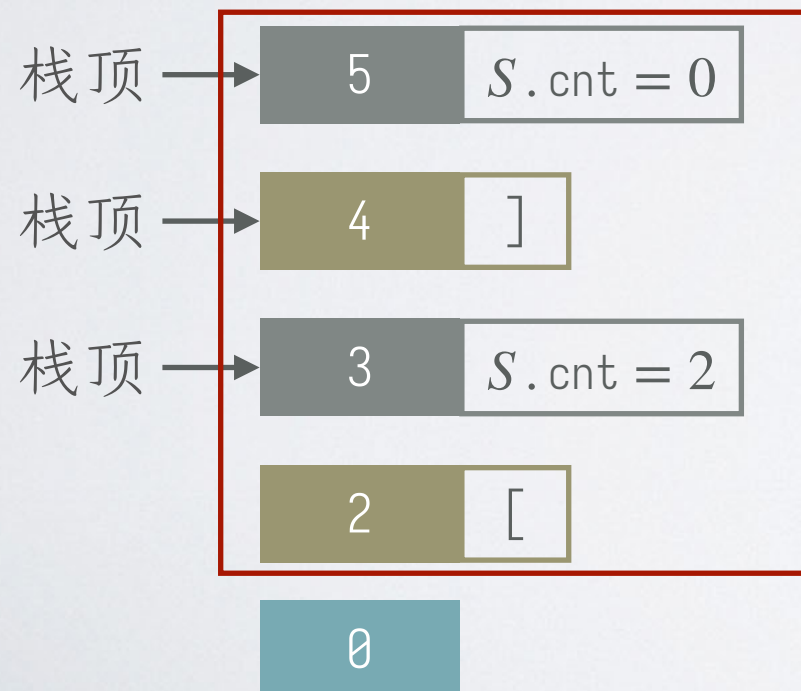


状态 \ 向前看	ACTION			GOTO
	[	]	EOF	S
0	移进 2		归约 [1]	1
1			接受	
2	移进 6	归约 [1]		3
3		移进 4		
4	移进 2		归约 [1]	5
5			归约 [2]	
6	移进 6	归约 [1]		7
7		移进 8		
8	移进 6	归约 [1]		9
9		归约 [2]		

以 LR 文法为基础的表驱动语义分析

[[[]]] EOF
↑

- [1] $S' \rightarrow S \text{ EOF}$ { $S'.\text{cnt} = S.\text{cnt};$ }
- [2] $S \rightarrow \epsilon$ { $S.\text{cnt} = 0;$ }
- [2] $S \rightarrow [S_1] S_2$ { $S.\text{cnt} = 1 + S_1.\text{cnt} + S_2.\text{cnt};$ }



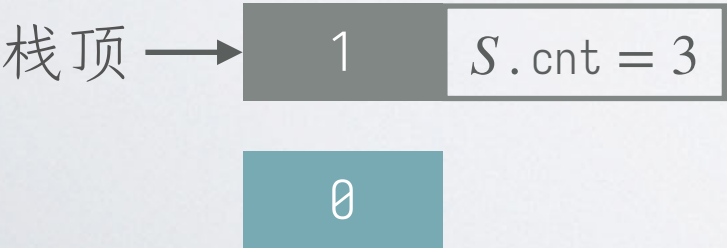
状态 \ 向前看	ACTION			GOTO
	[	]	EOF	S
0	移进 2		归约 [1]	1
1			接受	
2	移进 6	归约 [1]		3
3		移进 4		
4	移进 2		归约 [1]	5
5			归约 [2]	
6	移进 6	归约 [1]		7
7		移进 8		
8	移进 6	归约 [1]		9
9		归约 [2]		

以 LR 文法为基础的表驱动语义分析

[[[]]] EOF
 ↑

	$S' \rightarrow S \text{ EOF}$	{ $S'.\text{cnt} = S.\text{cnt};$ }
[1]	$S \rightarrow \epsilon$	{ $S.\text{cnt} = 0;$ }
[2]	$S \rightarrow [S_1] S_2$	{ $S.\text{cnt} = 1 + S_1.\text{cnt} + S_2.\text{cnt};$ }

状态 \ 向前看	ACTION			GOTO
	[	]	EOF	S
0	移进 2		归约 [1]	1
1			接受	
2	移进 6	归约 [1]		3
3		移进 4		
4	移进 2		归约 [1]	5
5			归约 [2]	
6	移进 6	归约 [1]		7
7		移进 8		
8	移进 6	归约 [1]		9
9		归约 [2]		



以 LR 文法为基础的表驱动语义分析

[[[]]] EOF
↑

- | | | |
|-----|--------------------------------|---|
| | $S' \rightarrow S \text{ EOF}$ | $\{ S'.\text{cnt} = S.\text{cnt}; \}$ |
| [1] | $S \rightarrow \epsilon$ | $\{ S.\text{cnt} = 0; \}$ |
| [2] | $S \rightarrow [S_1] S_2$ | $\{ S.\text{cnt} = 1 + S_1.\text{cnt} + S_2.\text{cnt}; \}$ |



状态 \ 向前看	ACTION			GOTO
	[	]	EOF	S
0	移进 2		归约 [1]	1
1			接受	
2	移进 6	归约 [1]		3
3		移进 4		
4	移进 2		归约 [1]	5
5			归约 [2]	
6	移进 6	归约 [1]		7
7		移进 8		
8	移进 6	归约 [1]		9
9		归约 [2]		

LR(1) 文法的语义分析驱动程序

```
token = next_token(); 自动机状态
top = 1; stack[top].state =  $I_0$ ;
while (true) {
    state = stack[top].state;
    if (ACTION[state][token] == “归约  $A \rightarrow \alpha \{ act \}$ ”) {
        根据 stack[top -  $|\alpha|$  + 1] 到 stack[top] 的属性完成 act 动作;
        top = top -  $|\alpha|$ ;
        state = stack[top].state;
        top++; stack[top].state = GOTO[state][A];
        stack[top].symbol = A; 根据前面的计算结果保存 A 的属性值;
    } else if (ACTION[state][token] == “移进  $I_j$ ”) {
        top++; stack[top].state =  $I_j$ ;
        stack[top].symbol = token;
        token = next_token();
    } else if (ACTION[state][token] == “接受”) {
        根据 stack[top] 的属性计算  $S'$  的属性值并存放于 stack[top];
        return true;
    } else return false;
}
```

显式操作分析栈的 SDT

- 基于 LR 文法的语义分析所使用的 SDT 可以显式操作分析栈

产生规则	语义动作
[0] $S' \rightarrow S \text{ EOF}$	{ $S'.\text{cnt} = S.\text{cnt};$ }
[1] $S \rightarrow \epsilon$	{ $S.\text{cnt} = 0;$ }
[2] $S \rightarrow [S_1] S_2$	{ $S.\text{cnt} = 1 + S_1.\text{cnt} + S_2.\text{cnt};$ }



产生规则	语义动作
[0] $S' \rightarrow S \text{ EOF}$	{ $\text{stack}[\text{top}-1].\text{cnt} = \text{stack}[\text{top}-1].\text{cnt};$ $\text{top} = \text{top}-1;$ }
[1] $S \rightarrow \epsilon$	{ $\text{stack}[\text{top}+1].\text{cnt} = 0;$ $\text{top} = \text{top}+1;$ }
[2] $S \rightarrow [S_1] S_2$	{ $\text{stack}[\text{top}-3].\text{cnt} = 1 + \text{stack}[\text{top}-2].\text{cnt} + \text{stack}[\text{top}].\text{cnt};$ $\text{top} = \text{top}-3;$ }

此处假设 EOF 也像其它 token 一样移进了分析栈

回顾：L 属性的文法的移进-归约实现

- 在 LL 文法中插入标记非终结符号 (如下面的 L 、 M 、 N) 后, 得到的文法是一个 LR 文法, 可以用移进-归约实现语义分析

```
top++; stack[top] = S;
stack[top].s = stack[top - 1].i;
```

```
top++; stack[top] = M;
stack[top].i = stack[top - 2].i + 1;
```

```
top++; stack[top] = N;
stack[top].i = stack[top - 2].s;
```

产生规则	属性计算规则
$S' \rightarrow L S \text{ EOF}$	$S'.s = S.s$
$S \rightarrow \epsilon$	$S.s = \text{栈中归约位置减 1 处}.i$
$S \rightarrow [M S_1] N S_2$	$S.s = S_2.s$
$L \rightarrow \epsilon$	$L.i = 0$
$M \rightarrow \epsilon$	$M.i = \text{栈中归约位置减 2 处}.i + 1$
$N \rightarrow \epsilon$	$N.i = \text{栈中归约位置减 2 处}.s$

- 可以通过显式操作分析栈的 SDT 来进行实现

L 属性的文法的移进-归约实现

文法本身是 LL 的

产生规则	属性计算规则
[0] $S' \rightarrow S \text{ EOF}$	$S.i = 0$ $S'.s = S.s$
[1] $S \rightarrow \epsilon$	$S.s = S.i$
[2] $S \rightarrow [S_1]S_2$	$S_1.i = S.i + 1$ $S_2.i = S_1.s$ $S.s = S_2.s$



产生规则	语义动作
$S' \rightarrow L S \text{ EOF}$	{ stack[top-2].s = stack[top-1].s; top = top-2; }
$S \rightarrow \epsilon$	{ stack[top+1].s = stack[top].i; top = top+1; }
$S \rightarrow [M S_1] N S_2$	{ stack[top-5].s = stack[top].s; top = top-5; }
$L \rightarrow \epsilon$	{ stack[top+1].i = 0; top = top+1; }
$M \rightarrow \epsilon$	{ stack[top+1].i = stack[top-1].i + 1; top = top+1; }
$N \rightarrow \epsilon$	{ stack[top+1].i = stack[top-1].s; top = top+1; }

主要内容

- ◎ 自底向上的语义分析
- ◎ 语义分析的自动生成
- ◎ 类型系统(基础版)

类型化的语言

- ◎ **类型化的语言**：变量都被给定类型的语言
 - ❖ 表达式、语句等语法构造的类型都是静态确定的
 - ❖ 例如，类型 `bool` 的变量 `x` 和 `y` 在程序运行时的值只能是布尔值
 - ❖ 从而表达式 `x && y` 总是良定义的
- ◎ **非类型化的语言**：不限制变量值范围的语言
 - ❖ 一个运算可以作用到任意的运算对象，其结果可能是一个有意义的值、一个错误、一个异常或一个语言未明确定义的结果
 - ❖ 例如：Python、JavaScript 等语言
- ◎ **Gradual Typing**：混合静态类型和动态类型
 - ❖ 平衡语言的**安全性**和**灵活性**

类型表达式

- ◎ 类型自身也有结构
- ◎ **类型表达式 (type expression)** 用来表示程序中变量、常量、表达式、语句等语言成分的类型
 - ❖ 基本类型: `bool`、`char`、`int`、`float`、`void` 等
 - ❖ 面向对象语言中的类名
 - ❖ 数组类型: *array* 作用于一个数字和一个类型表达式
 - ❖ 记录类型: *record* 作用于字段名和相应的类型
 - ❖ 函数类型: 从类型 s 到类型 t 的函数表示为 $s \rightarrow t$
 - ❖ 笛卡尔积: $s \times t$ 可以用来表示类型的元组 (例如函数参数)
 - ❖ 取值为类型表达式的变量

类型表达式的例子

- ◎ C 语言中的结构体:

```
struct {  
    int no;  
    char name[20];  
}
```

- ◎ 它对应的类型表达式为:

- ❖ *record({no : int, name : array(20,char)})*
- ❖ 或者(如果不考虑字段名)
- ❖ *int × array(20,char)*

类型等价

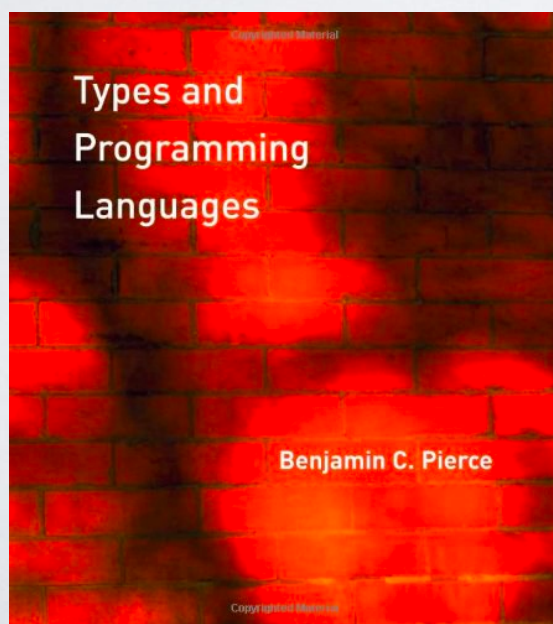
- ◎ **类型等价 (type equivalence)**: 两个类型的值的集合相等并且作用在其上的运算集合相等
 - ❖ 类型等价具有对称性
- ◎ 类型等价主要可以分为两种:
 - ❖ **按名字等价**: 两个类型名字相同, 或者被定义成等价的两个名字
 - ❖ 比如: C++ 和 Java 中的类定义
 - ❖ **按结构等价**: 两个类型的结构完全相同, 但是名字不一定相同 (或者不一定有名字)
 - ❖ 比如: Standard ML 中的记录类型
$$\{x : int, y : int\} \equiv \{y : int, x : int\}$$
 - ❖ 按名字等价一定是按结构等价的

类型检查

◎ 类型系统(type system):

- ❖ 给程序的每一个组成部分赋予一个类型表达式
- ❖ 通过一组逻辑规则来表示这些类型表达式必须满足的条件
- ❖ **健全性(soundness)**: 保证某类类型错误不会在运行时发生

◎ 「类型系统是指一种根据程序所计算的值进行分类从而证明某些(不良)程序行为不会发生的轻量级形式化方法」



Benjamin C. Pierce, **Types and Programming Languages**

类型检查规则分类

● 类型综合(type synthesis)

- ❖ 根据子表达式的类型构造出表达式的类型

如果 f 的类型为 $s \rightarrow t$ 且 x 的类型为 s

那么 表达式 $f(x)$ 的类型为 t

● 类型推导(type inference)

- ❖ 根据语言结构的使用方式来确定该结构的类型

如果 $f(x)$ 是一个表达式

那么 对于某些类型 α 和 β , f 的类型是 $\alpha \rightarrow \beta$ 且 x 的类型为 α

- ❖ α 和 β 可以是类型表达式中的类型变量

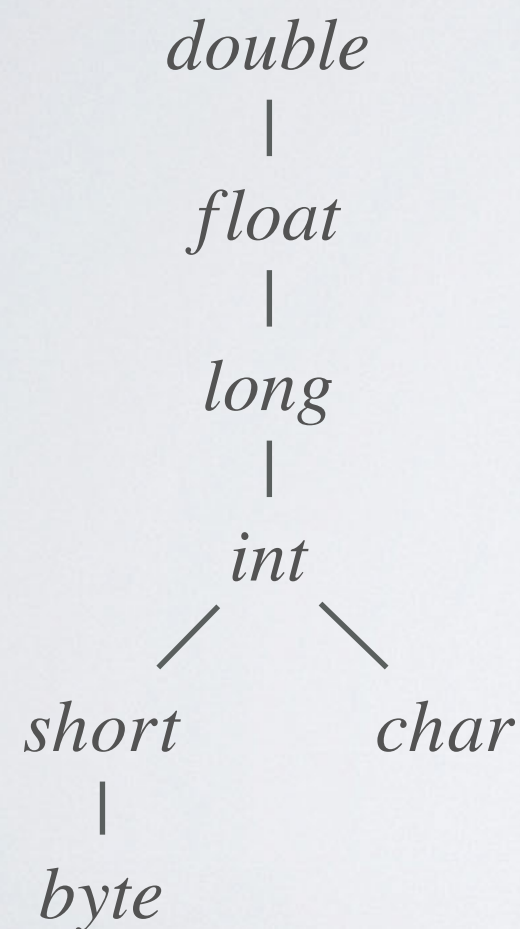
● Bidirectional Typing: 结合类型综合和推导

- ❖ 有确定的类型信息时, 通过类型综合传播信息
- ❖ 出现未知(或未写明的)类型时, 进行类型推导

类型的拓宽和窄化

◎ Java 的类型转换规则:

- ❖ **拓宽(widening)**: 保持转换前的原有数值范围信息(但可能**丢失精度**)
- ❖ **窄化(narrowing)**: 转换可能丢失数值范围信息

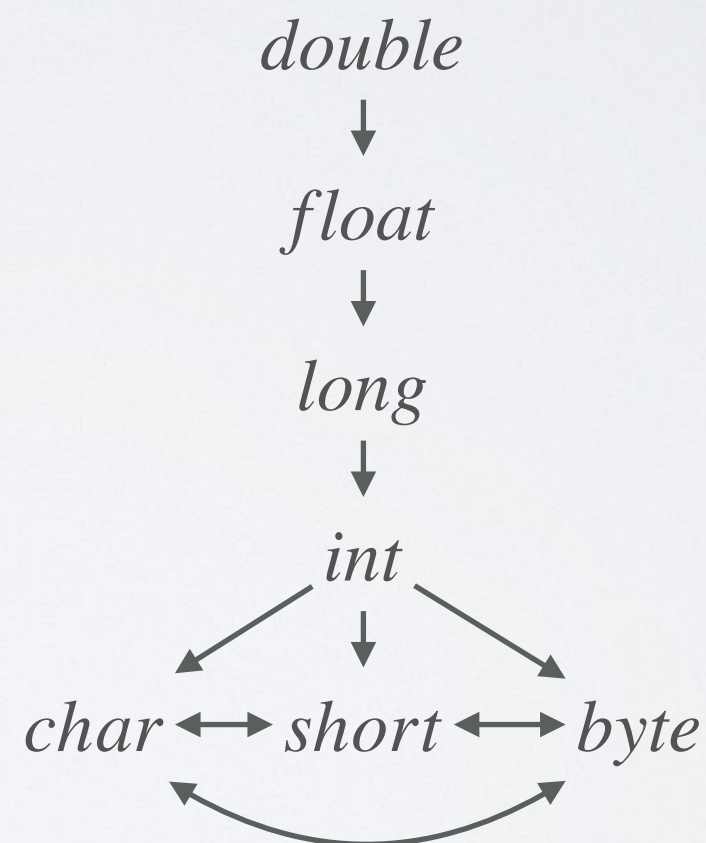


拓宽

```

byte b = 4;
short s = b;
int i = s;
long l = i;
float f = l;
double d = f;
  
```

隐式转换



窄化

```

double d = 4;
float f = (float)d;
long l = (long)f;
int i = (int)l;
short s = (short)i;
byte b = (byte)s;
  
```

显式转换

函数和运算符的重载

- 根据符号所在的上下文不同, 被**重载 (overloaded)** 的符号可能有不同的含义
- 针对重载函数的类型综合规则:
 - 如果 f 可能的类型为 $s_i \rightarrow t_i$ ($1 \leq i \leq n$), 其中 $s_i \neq s_j$ ($i \neq j$)
 - 且 x 的类型为 s_k ($1 \leq k \leq n$)
 - 那么 表达式 $f(x)$ 的类型为 t_k

类型推导

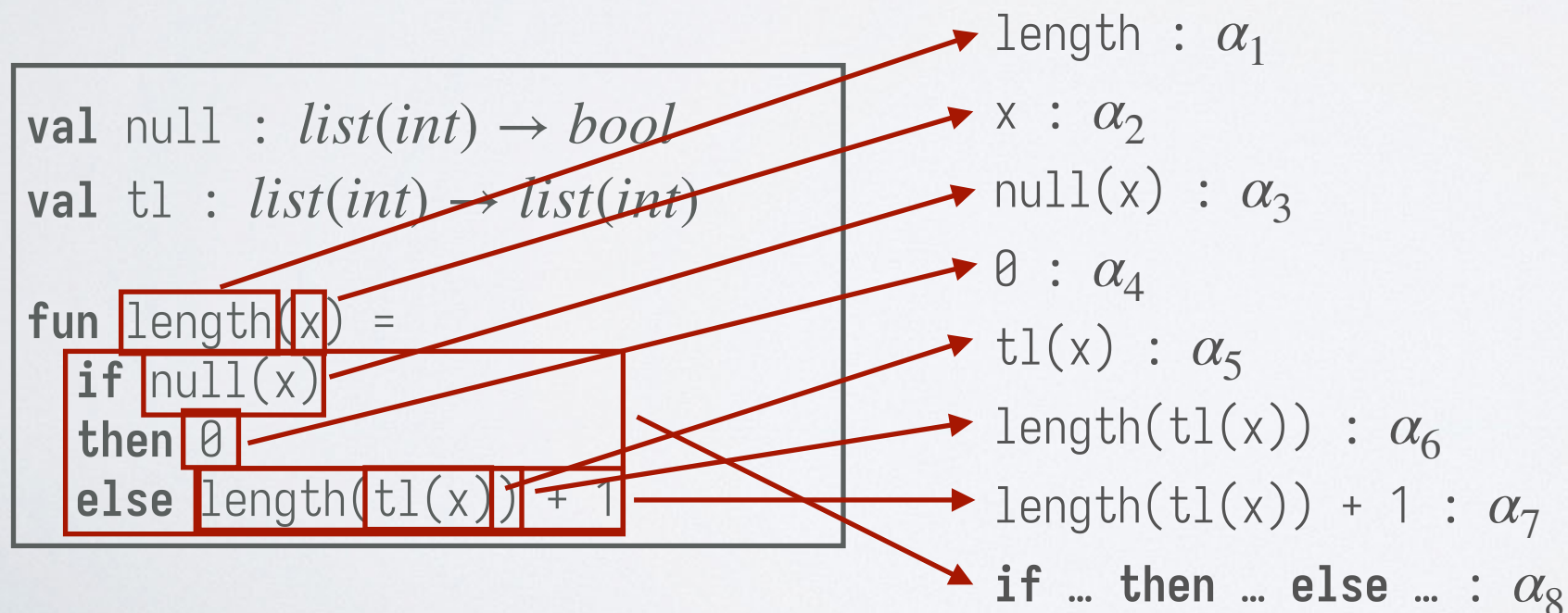
◎ 问题：类型化的语言可能要求程序员 **写很多的类型标注**

❖ 比如 C++ 和 Java 等语言

◎ **类型推导**可以一定程度上降低程序员的负担

❖ 比如 C++ 中的 auto, Rust、Haskell 等语言中不同程度的自动推导

◎ 类型推导的一般方法：**解方程**



$$\alpha_1 = list(int) \rightarrow int$$

↑ 解

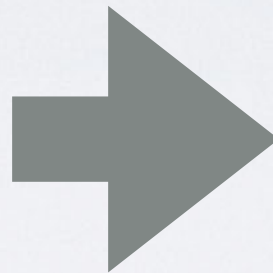
$$\begin{aligned}
 \alpha_2 \rightarrow \alpha_3 &= list(int) \rightarrow bool \\
 \alpha_4 &= int \\
 \alpha_2 \rightarrow \alpha_5 &= list(int) \rightarrow list(int) \\
 \alpha_5 \rightarrow \alpha_6 &= \alpha_1 \\
 \alpha_6 &= int \\
 \alpha_7 &= int \\
 \alpha_4 &= \alpha_7 \\
 \alpha_8 &= \alpha_4 \\
 \alpha_1 &= \alpha_2 \rightarrow \alpha_8
 \end{aligned}$$

参数多态

- 参数多态 (parametric polymorphism): 函数可以在不同类型 (通过类型变量来刻画) 的参数上运行

```
val null : list(int) → bool
val tl : list(int) → list(int)

fun length(x) =
  if null(x)
  then 0
  else length(tl(x)) + 1
```



```
val null :  $\forall \beta. list(\beta) \rightarrow bool$ 
val tl :  $\forall \beta. list(\beta) \rightarrow list(\beta)$ 

fun length(x) =
  if null(x)
  then 0
  else length(tl(x)) + 1
```

- 在类型推导中, 多态函数每次应用可能作用于不同的类型

length : α_1

x : α_2

null(x) : α_3

0 : α_4

tl(x) : α_5

length(tl(x)) : α_6

length(tl(x)) + 1 : α_7

if ... then ... else ... : α_8

方程

$\alpha_2 \rightarrow \alpha_3 = list(\beta_1) \rightarrow bool$

$\alpha_2 \rightarrow \alpha_5 = list(\beta_2) \rightarrow list(int)$

解

$list(\beta_1) = \alpha_2$

$list(\beta_2) = \alpha_2$

$\alpha_1 = \alpha_2 \rightarrow int$

泛化

length : $\forall \beta_1. list(\beta_1) \rightarrow int$

历史：从 B 语言到 C 语言

```
printn(n, b) {  
    extrn putchar;  
    auto a;  
  
    a = n / b;  
    if (a)  
        printn(a, b);  
    putchar(n % b + '0');  
}
```

B 语言

- 1969 年
- 贝尔实验室
- 为系统和编译器开发设计
- **无类型**
- **性能差**

历史：从 B 语言到 C 语言

```
printn(n, b) {  
    extrn putchar;  
    auto a;  
  
    a = n / b;  
    if (a)  
        printn(a, b);  
    putchar(n % b + '0');  
}
```

B 语言

函数参数是**无类型**的
函数也没有返回类型

历史：从 B 语言到 C 语言

```
printn(n, b) {  
    extrn putchar;  
    auto a;  
  
    a = n / b;  
    if (a)  
        printn(a, b);  
    putchar(n % b + '0');  
}
```

B 语言

auto 关键字仅表示为
变量 a 预留存储空间

B 语言中只有一种类型：
所有数据都是字(word)

历史：从 B 语言到 C 语言

```
printn(n, b) {  
    extrn putchar;  
    auto a;  
  
    a = n / b;  
    if (a)  
        printn(a, b);  
    putchar(n % b + '0');  
}
```

B 语言

调用的外部函数**没有函数签名**

编译时**无法检查**函数调用
是否传入了正确的参数

运行时检查降低了程序性能

历史：从 B 语言到 C 语言

```
printn(n, b) {  
    extrn putchar;  
    auto a;  
  
    a = n / b;  
    if (a)  
        printn(a, b);  
    putchar(n % b + '0');  
}
```

B 语言

```
extern int putchar(int);  
void printn(int n, int b) {  
    int a;  
  
    a = n / b;  
    if (a)  
        printn(a, b);  
    putchar(n % b + '0');  
}
```

C 语言

历史：从 B 语言到 C 语言

```
extern int putchar(int);  
void printn(int n, int b) {  
    int a;  
  
    a = n / b;  
    if (a)  
        printn(a, b);  
    putchar(n % b + '0');  
}
```

C 语言

- ◎ 1972 年
- ◎ 贝尔实验室
- ◎ 通用编程语言，常被用来开发操作系统、设备驱动、协议栈等
- ◎ 静态类型
- ◎ 性能优越

类型系统的优点

● 检查错误

- ❖ 函数调用不符合签名, 对浮点数进行按位与操作, ...

● 提供文档

- ❖ 函数的类型标注提供了基本的函数文档

● 编译优化

- ❖ 去除不必要的运行时检查, 优化数据结构的内存布局, ...

● 语言安全

- ❖ Rust 提供的内存安全, TypeScript 提供的针对 JavaScript 的安全, ...

● 抽象机制

- ❖ 带类型的泛型编程, 面向对象编程, 函数式编程, ...

类型系统与语言设计

语言设计应当同类型系统设计并行进行且互相协同

- 不含类型系统的语言在设计时容易引入**难以进行类型检查**的编程特性
- 带类型语言的**语法可能更复杂**，需要在设计中考虑类型标注的清晰性