语义分析与中间代码生成

语义分析：概览

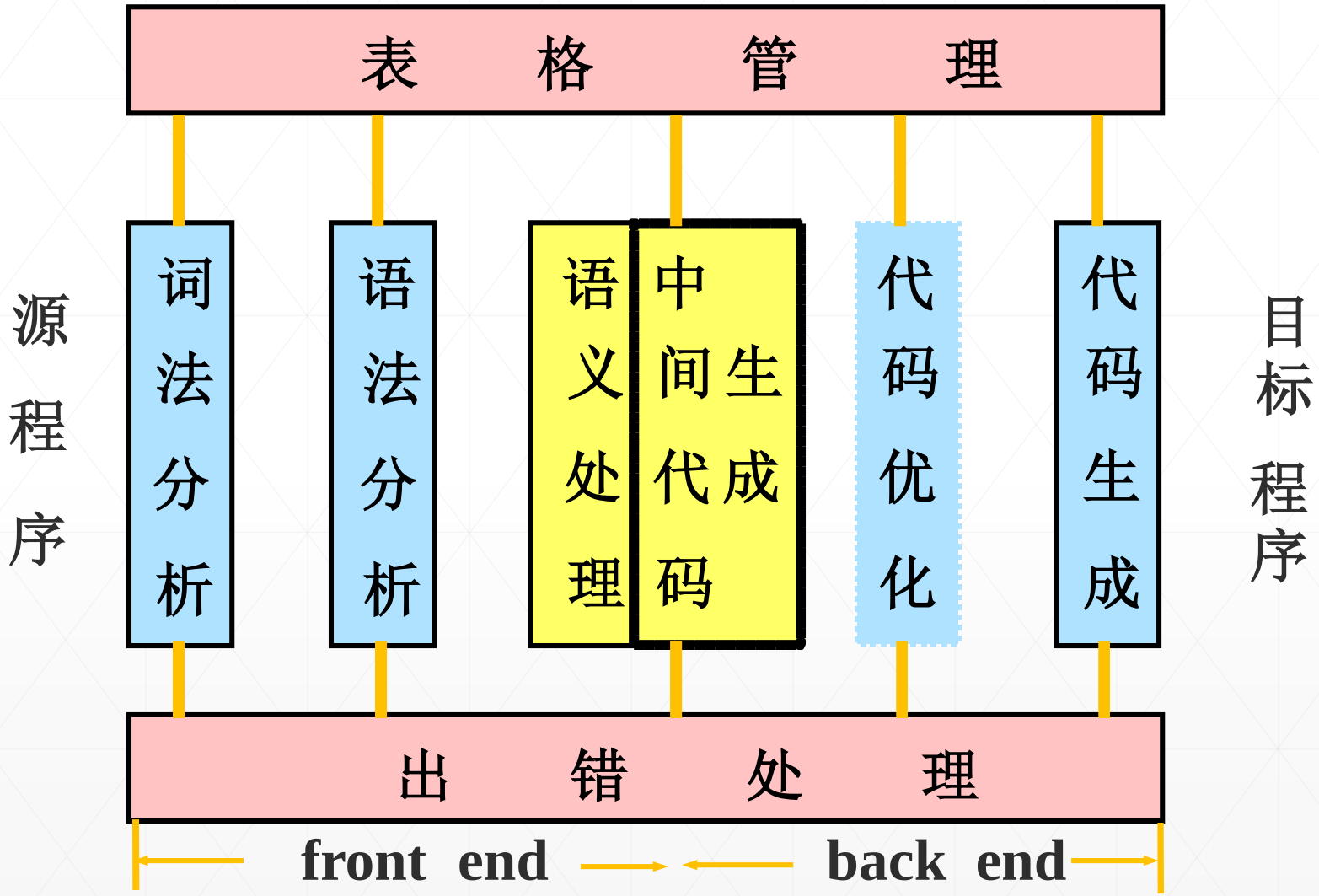
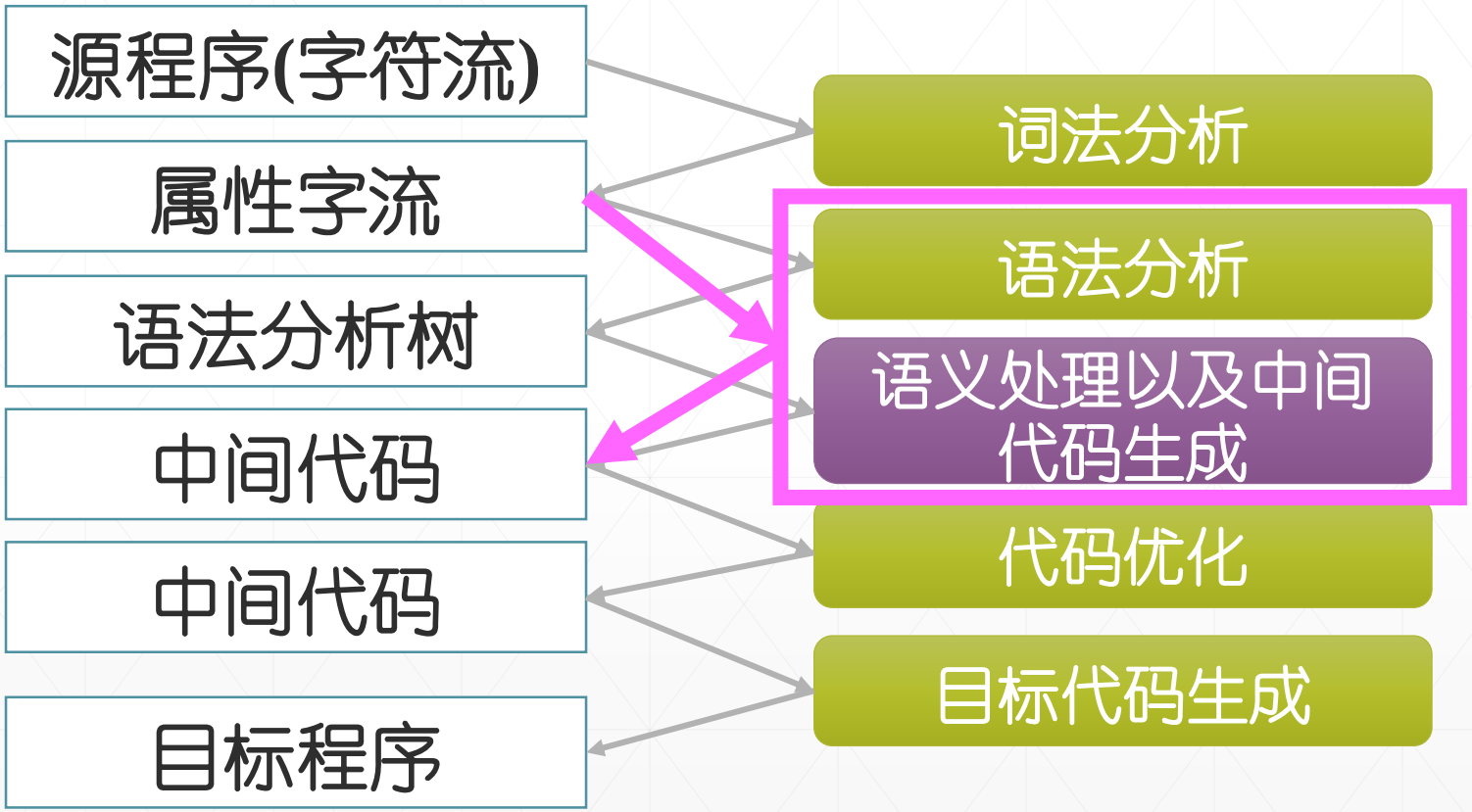
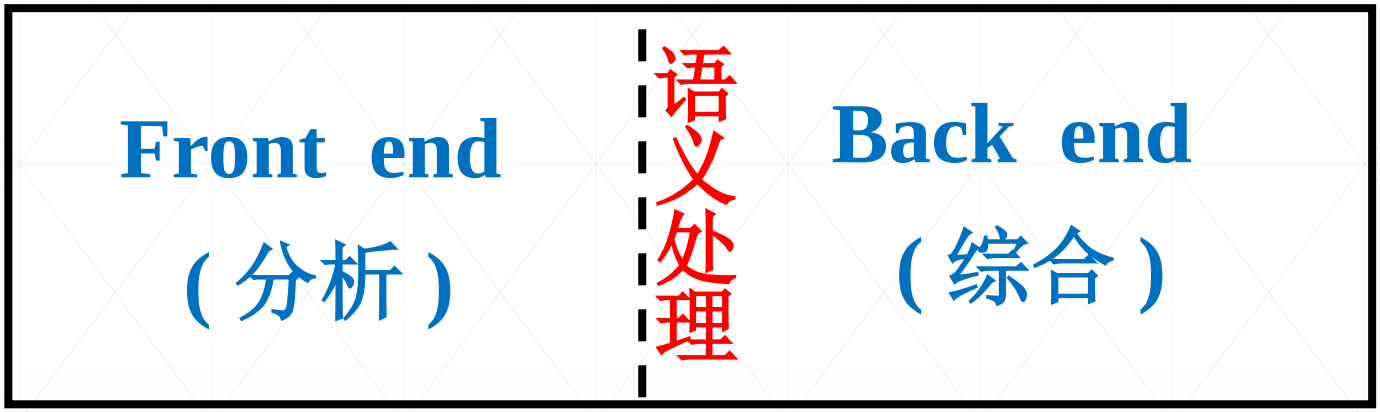


表 格 管 理
源程序
词法分析
语法分析
语义处理
中间代码生成
代码优化
代码生成
目标程序
出 错 处 理
front end
back end

- 基本功能


源程序(字符流)
属性字流
语法分析树
中间代码
中间代码
目标程序
词法分析
语法分析
语义处理以及中间代码生成
代码优化
目标代码生成

语义分析的地位



编译程序最实质性的工作；

第一次对源程序的语义作出解释，引起源程序质的变化。

语义分析的任务

按照语法分析器识别的语法范畴，

输入

依据其语义规则

依据

进行语义检查和处理，

输出

产生相应的中间代码或目标代码。

语法的局限(上下文无关文法)，没有在语法中定义的结构通过语义处理描述

语法正确，但是不正确的程序

- 重复定义的标识符
- 函数参数不匹配
- 类型不兼容的访问
- Break语句的位置
- 未声明的标识符
- Goto的目标不存在
- …

```
foo(int a, char * s){...}

int bar() {
  int f[3];
  int i, j, k;
  char q, *p;
  float k;
  foo(f[6], 10, j);
  break;
  i->val = 42;
  j = m + k;
  printf("%s,%s.\n",p,q);
  goto label42;
}
```

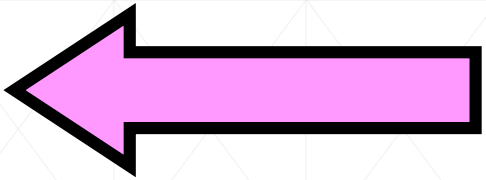
中间代码

介于源语言和目标代码之间的一种代码。

引入中间代码的目的

1. 方便生成目标代码；
2. 便于优化；
3. 便于移植。

第 6 章　语义分析与中间代码生成



6.1　语法制导翻译

6.2　符号表

6.3　类型检查

6.4　中间语言

6.5　中间代码生成

6.1 语法制导翻译

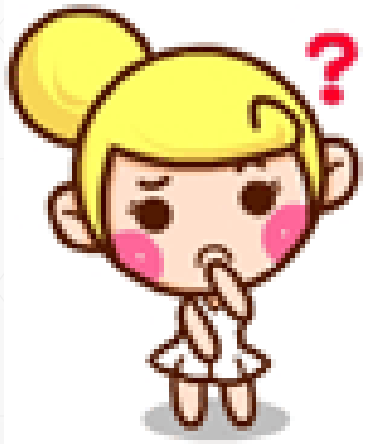
6.1.1 语法制导翻译基本原理

6.1.2 自顶向下翻译器

6.1.3 自底向上翻译器

6.1.4 语法制导定义SDD—属性文法

6.1.5 语法制导翻译方案SDT



语法制导翻译思想

为文法的每一个产生式添加一个成分(语义动作或翻译子程序)，

在执行语法分析的同时调用它。

查填表、修改值、打印信息、输出中间语言…

语法制导翻译做语义处理设计的两个概念

语法制导翻译模型

翻译对偶

语法制导翻译模型(Syntax_Directed Translation Schemata，SDTS)

五元组T=(V_T，V_N，Δ，R，S)

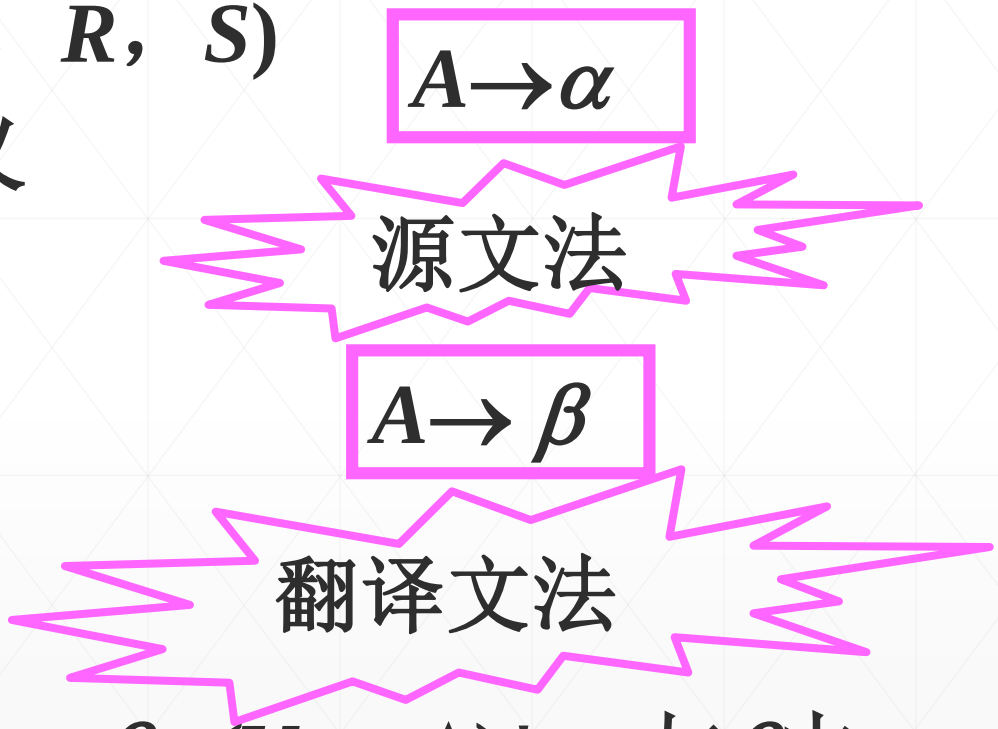
V_T，V_N，S：文法定义

Δ：输出字符表

R：规则的集合

规则形式：

$A\rightarrow\alpha$，β



其中:$A\rightarrow\alpha$为文法规则，$\beta\in(V_N\cup\Delta)^*$，$\alpha$与$\beta$中的非终结符一一对应。

例：简单表达式的中缀到后缀翻译的SDTS。

SDTS=({*i*,+,-,(,)}，{*E*,*T*}，{+,-, @,*i*}，*R*，*E*)

*R*中规则为：

E→*E*+*T*，*ET*+

E→*E*-*T*，*ET*-

E→-*T*，*T*@

E→*T*，*T*

T→(*E*)，*E*

T→*i*，*i*

翻译对偶

T=(V_T，V_N，Δ，R，S)的一个翻译对偶

1)(S,S)是一个翻译对偶，两个S是相关的(S是SDTS的开始符号)。

2) ($\alpha A\beta$, $\alpha' A\beta'$)是一个翻译对偶，其中的两个A相关；若$A\rightarrow\delta$，γ是R中的一条规则，那么($\alpha\delta\beta$, $\alpha'\gamma\beta'$)也是一个翻译对偶，另外δ和γ的非终结符号之间的相关性也必须带进这种翻译对偶。

可表示为($\alpha A\beta$, $\alpha' A\beta'$)$\Rightarrow$ ($\alpha\delta\beta$, $\alpha'\gamma\beta'$)

一个SDTS=(V_T，V_N，Δ，R，S)所定义的翻译是特殊翻译对偶的集合

$\{(\alpha, \beta)|(S,S)\overset{*}{\Rightarrow}(\alpha, \beta), \quad \alpha \in V_T^*, \beta \in \Delta^*,\}$

例：简单表达式的中缀到后缀翻译的SDTS。

-(*i*+*i*)-*i*是该文法的句子，其对应的翻译为：

*R*中规则为：

	(*E*,*E*) ⇒(*E*-*T*,*ET*-)⇒(-*T*-*T*,*T*@*T*-)
E→*E*+*T*，*ET*+	⇒(-(*E*)-*T*,*E*@*T*-)⇒(-(*E*+*T*)-*T*,*ET*+@*T*-)
E→*E*-*T*，*ET*-	⇒(-(*T*+*T*)-*T*,*TT*+@*T*-)
E→-*T*，*T*@	⇒(-(*i*+*T*)-*T*,*iT*+@*T*-)
E→*T*，*T*	⇒(-(*i*+*i*)-*T*,*ii*+@*T*-)
T→(E)，*E*	⇒(-(*i*+*i*)-*i*, *ii*+@*i*-)
T→*i*，*i*	

语法制导的翻译过程也可以用分析树表示。

R中规则为：

$E \to E+T$，$ET+$

$E \to E-T$，$ET-$

$E \to -T$，$T@$

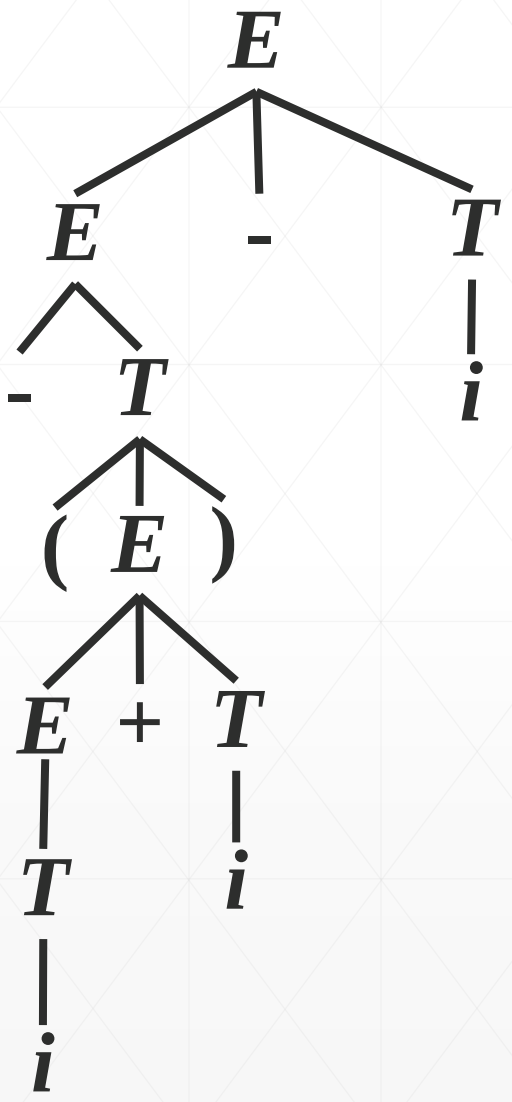
$E \to T$，T

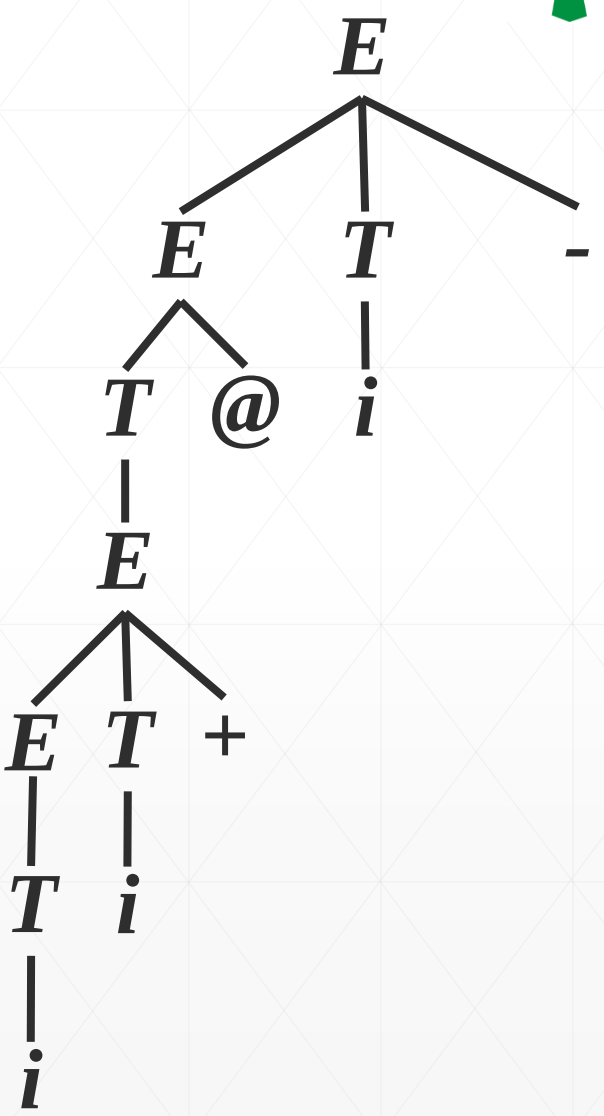
$T \to (E)$，E

$T \to i$，i

句子：

$-(i+i)-i$





6.1 语法制导翻译

6.1.1 语法制导翻译基本原理

6.1.2 自上而下翻译器

6.1.3 自下而上翻译器

6.1.4 语法制导定义SDD—属性文法

6.1.5 语法制导翻译方案SDT

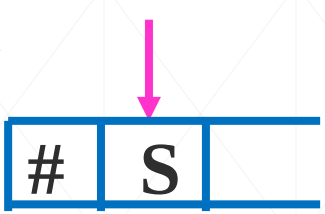
简单SDTS：

*R*中的每一条规则$A \to \alpha$，β

β中的非终结符号出现次序与α中的非终结符号出现次序相同。

源文法是LL(1)文法，若构造出关于该文法的一简单SDTS，则可以在自上而下的LL(1)的语法分析中加入翻译动作实现翻译，简称自上而下的翻译。

带翻译处理的LL(1)分析



p

$a_1\ a_2 \ldots a_n\ \#$

(1) 初始化工作：

为描述方便，设栈顶符号为X，p指向的符号为a_i，

(2)若X是文法的终结符号，则

对于：

① $X=a_i=$“#”，处理成功，停止处理过程；

② $X=a_i\neq$“#”，则将X从分析栈顶退掉，p指向下一个输入字符；

③ $X\neq a_i$，表示不匹配的出错情况。

(3)若X是文法的输出符号，则

从栈中弹出并输出X。

(4) 若$X \in V_N$，则查分析表中的项$M(X,a_i)$:

①若$M(X,a_i)$中为一个产生式规则，设其对应的SDTS规则为$X \to \alpha_0 A_1 \alpha_1 A_2 \ldots A_k \alpha_k, \beta_0 A_1 \beta_1 A_2 \ldots A_k \beta_k$

(A_i是非终结符号，α_i是终结符号串，β_i是输出符号串)

则将X从栈中弹出,并将串$\beta_0 \alpha_0 A_1 \beta_1 \alpha_1 A_2 \ldots A_k \beta_k \alpha_k$ 按倒序推进栈。

② 若$M(X,a_i)$中为空白，表示出错，可调用语法出错处理子程序。

例：简单的SDTS的*R*规则为

$S\rightarrow(S)S$，$xSySz$

$S\rightarrow\varepsilon$，w

源文法的LL(1)分析表

	(	)	#
S	$S\rightarrow(S)S$	$S\rightarrow\varepsilon$	$S\rightarrow\varepsilon$

步骤	分析栈	待匹配串	分析动作
0	#S	()()#	$S\rightarrow(S)S$
1	#z*S*)y*S*(x	()()#	输出*x*
2	#z*S*)y*S*(	()()#	*P*++
3	#z*S*)y*S*	)()#	$S\rightarrow\varepsilon$
4	#z*S*)y*w*	)()#	输出*w*
5	#z*S*)y	)()#	输出*y*
6	#z*S*)	)()#	*P*++
7	#z*S*	()#	$S\rightarrow(S)S$
8	#z*z**S*)y*S*(x	()#	输出*x*
9	#z*z**S*)y*S*(	()#	*P*++
10	#z*z**S*)y*S*	)#	$S\rightarrow\varepsilon$
11	#z*z**S*)y*w*	)#	输出*w*
12	#z*z**S*)y	)#	输出*y*
13	#z*z**S*)	)#	*P*++
14	#z*z**S*	#	$S\rightarrow\varepsilon$
15	#z*z*w	#	输出*w*

步骤	分析栈	待匹配串	分析动作
16	#z*z*	#	输出*z*
17	#z	#	输出*z*
18	#	#	成功结束

句子()()的翻译过程

翻译结果为：

xwyxwywzz

6.1 语法制导翻译

6.1.1 语法制导翻译基本原理

6.1.2 自上而下翻译器

6.1.3 自下而上翻译器

6.1.4 语法制导定义SDD—属性文法

6.1.5 语法制导翻译方案SDT

简单后缀SDTS

1.简单SDTS；

2.每一条规则都有如下形式

$$A \rightarrow \alpha_1 B_1 \alpha_2 B_2 \ldots B_k \alpha_k, B_1 B_2 \ldots B_k \beta$$

即除了最右边的β输出符号串，输出符号不能出现在翻译成分中。

源文法是LR文法，若构造出关于该文法的一简单后缀SDTS，则可以在自下而上的LR语法分析中加入翻译动作实现翻译，简称自下而上的翻译。

带翻译处理的LR分析

① 初始化:将开始状态Q_0及”#”压入分析栈

②据当前分析栈栈顶Q_m，当前输入符号a_i查action表：

i)若action(Q_m , a_i)＝S_{Qj}，完成移进动作；

ii)若action (Q_m , a_i)＝r_j，对应的SDTS规则为

$X\rightarrow\alpha_1A_1\alpha_2A_2\ldots A_k\alpha_k, A_1A_2\ldots A_k\beta$，完成归约动作，并输出$\beta$。

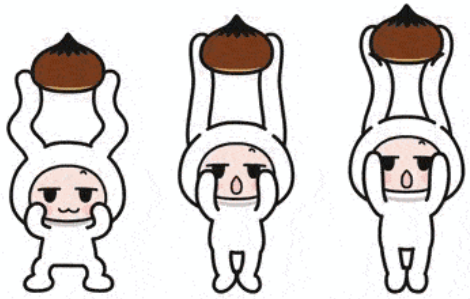
iii)若action(Q_m , a_i)＝acc，分析成功；

iv)若action(Q_m , a_i)＝error，出错处理。

③转②。

简单SDTS修改为简单后缀SDTS

条件语句的处理



条件语句文法：

S→if E then S else S

条件语句文法修改为：

S→T else S

T→I then S

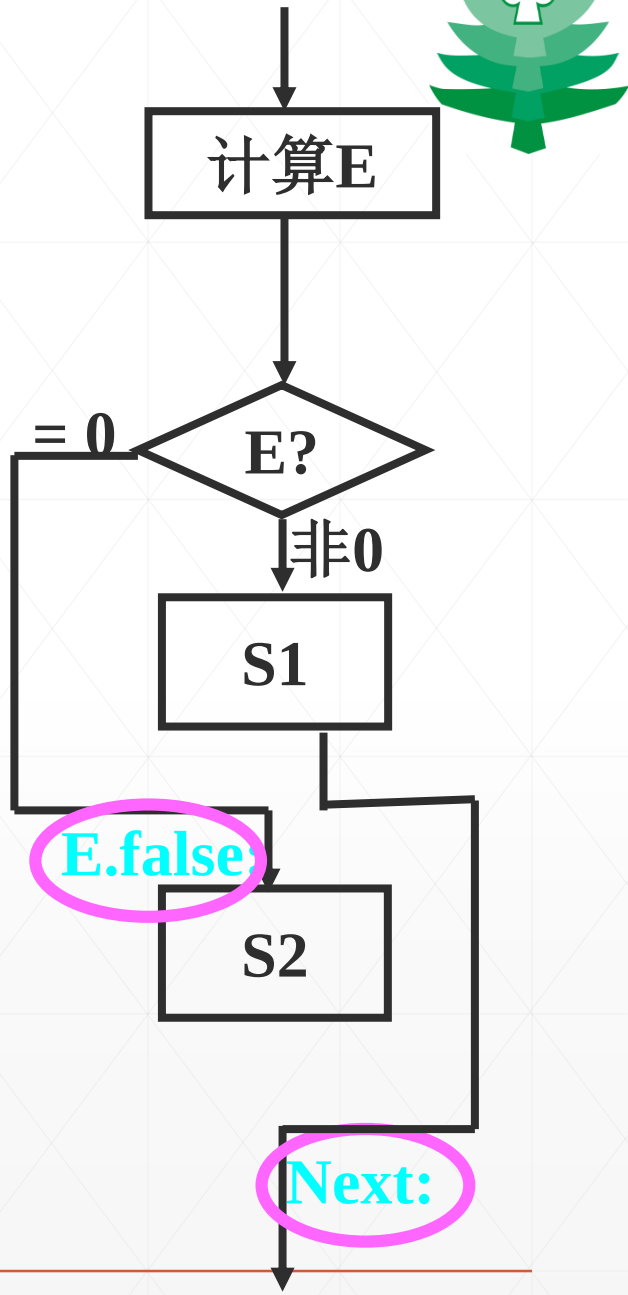
I→if E

对应的简单后缀SDTS为：

S→T else S，TS;标号Next:

T→I then S，IS; J标号Next；标号E.false:

I→if E，E;JF标号E.false;



简单SDTS修改为简单后缀SDTS

条件语句的处理

条件语句文法：

S→if E then S else S

条件语句文法修改为：

S→if E T S L S

T→then

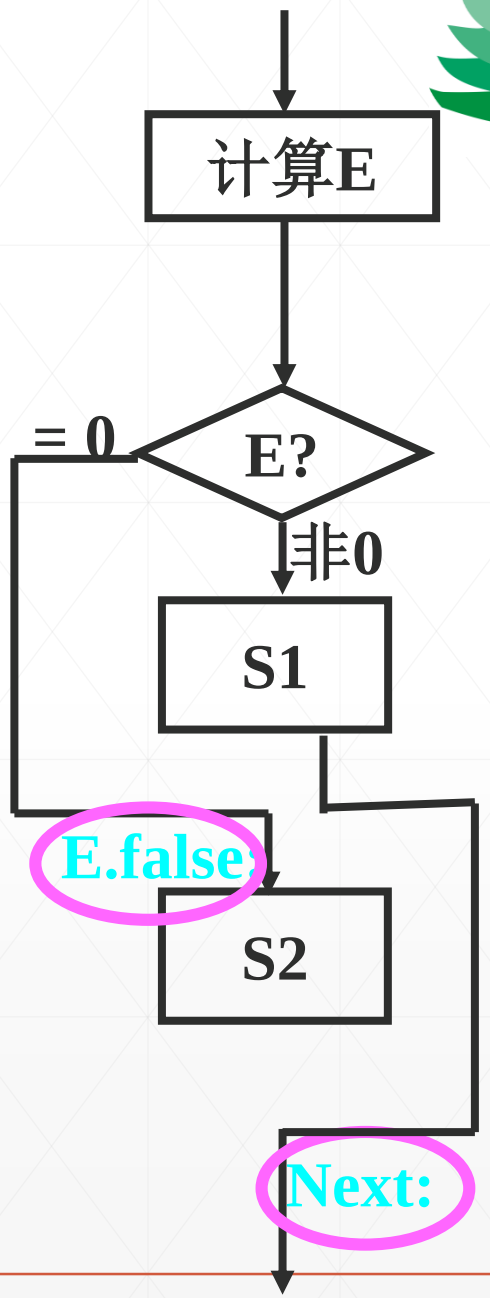
L→else

对应的简单后缀SDTS为：

S→if E T S L S， E T S L S;标号Next:

T→then， JF标号E.false;

L→else， J标号Next；标号E.false:



简单SDTS修改为简单后缀SDTS

条件语句的处理

条件语句文法：

S→if E then S else S

条件语句文法修改为：

S→if E then T S else L S

T→ε

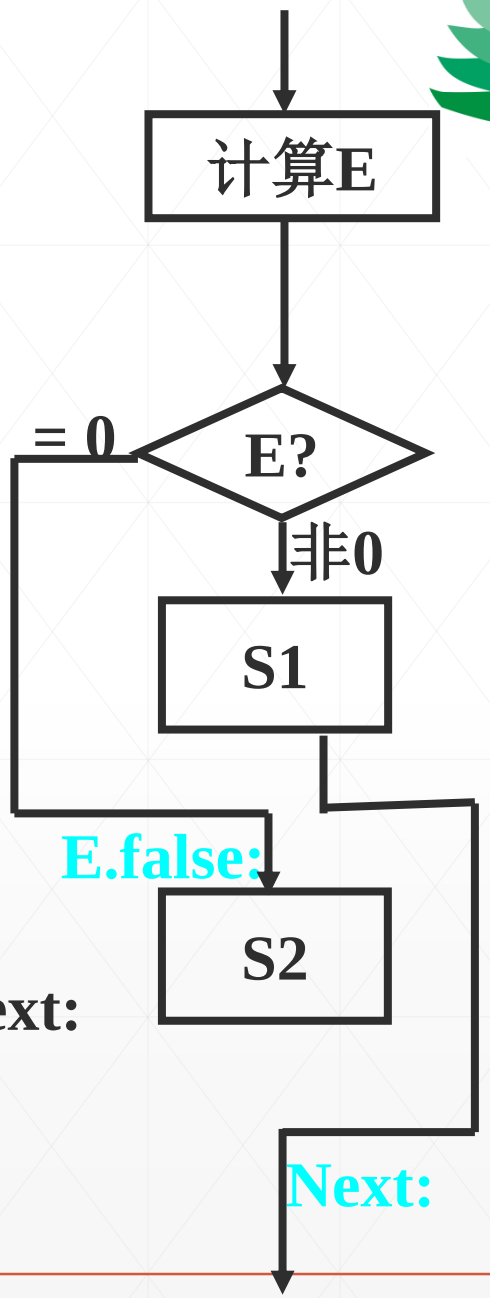
L→ε

对应的简单后缀SDTS为：

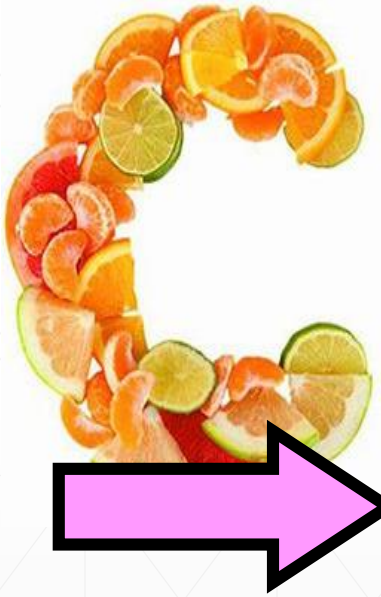
S→if E then T S else L S， E T S L S;标号Next:

T→ε，JF标号E.false;

L→ε，J标号Next；标号E.false:



6.1 语法制导翻译



6.1.1 语法制导翻译基本原理

6.1.2 自上而下翻译器

6.1.3 自下而上翻译器

6.1.4 语法制导定义SDD—属性文法

6.1.5 语法制导翻译方案SDT

语义用文法结构定义，而结构表示为文法符号，把语法制导模型中的输出符号集合用文法符号的属性表示

文法符号的属性

- 文法符号对应一“属性”集；

表示为：

$N.t$ （N是文法符号，t是N的属性）

属性可以为：数、符号串、代码、类型、存储空间…

- 属性的值由产生式的语义规则来定义；
- 属性同变量一样，可以进行计算和传递；
- 属性计算的过程既是语义处理的过程。

- 属性翻译文法(语法制导定义Syntax-Directed Definition,SDD)

- 形式定义 $A=(G,V,F)$

其中：

G： 二型文法；

V： 属性的有穷集；

F：用属性描述的与产生式相关的语义规则

简单运算的台式计算器的属性文法

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

终结符号的属性

实际应用中：存在非属性描述的语义规则(动作)看做虚属性

非终结符号的两类属性

- **综合属性(Synthesized Attributes,*s*属性)**

产生式左部符号的某些属性由候选式中符号的属性和(或)自己的其他属性定义；

- **继承属性(Inherited Attributes,*i*属性)**

候选式中符号的某些属性由产生式左侧非终结符号的属性和(或)候选式中其他符号的某些属性定义。

终结符号只有综合属性，由词法分析提供

简单运算的台式计算器的属性文法

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

综合属性

简单类型说明语句的属性文法

产生式	语义规则
$D \rightarrow T\ L$	$L.type = T.type$
$T \rightarrow int$	$T.type = int$
$T \rightarrow float$	$T.type = float$
$L \rightarrow L, i$	$L_1.type = L.type$
$L \rightarrow i$	$Addtype(i.entry, L.type)$

继承属性

虚属性

依赖图

语法分析树中结点属性计算的依赖关系图

在语法分析树的基础上构建

增加属性结点：

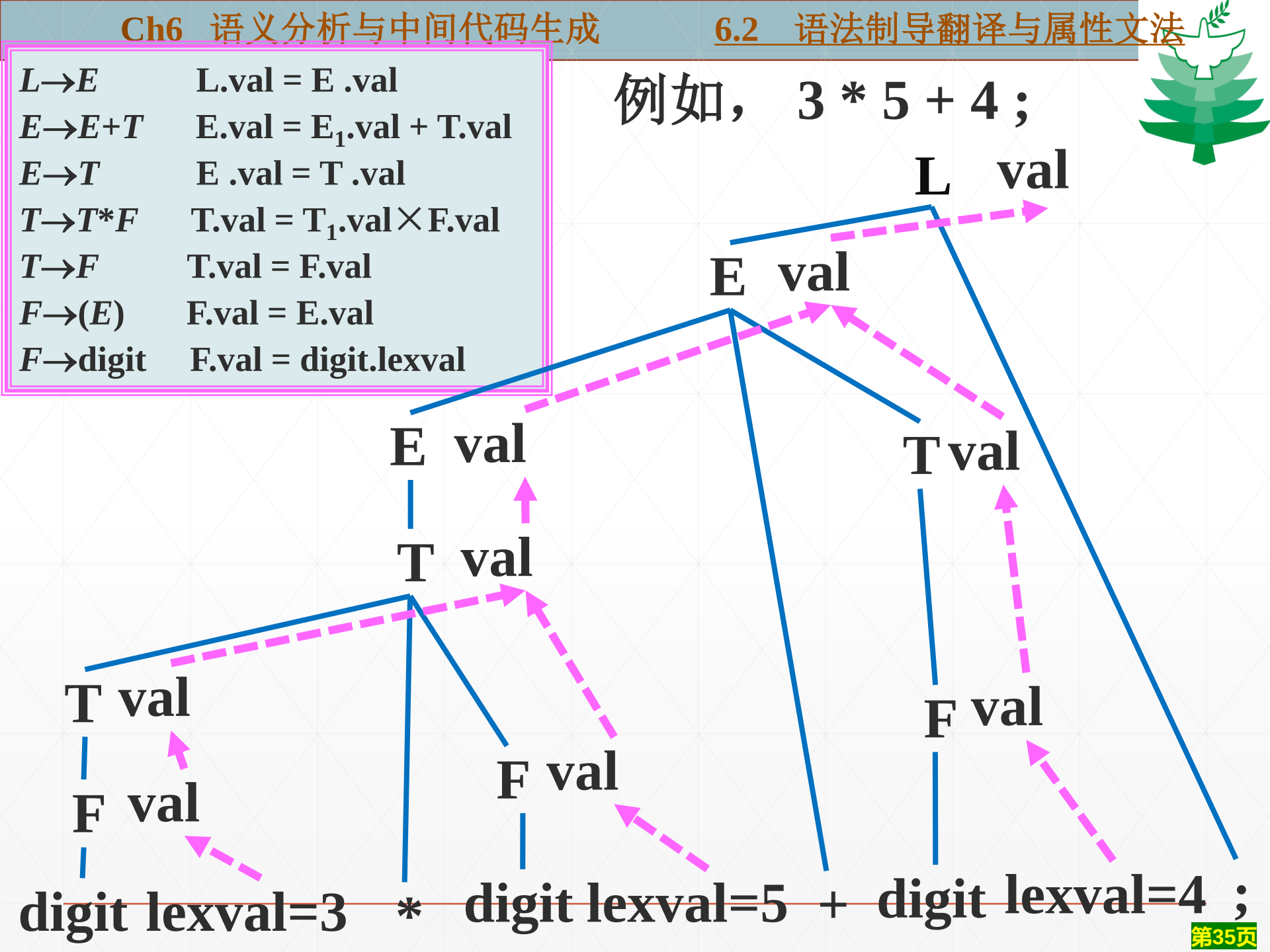
继承属性标在符号的左边，综合属性标在符号的右边

增加属性节点之间的边：

若属性b的计算依赖于属性c，则从结点c到结点b画一条有向边。

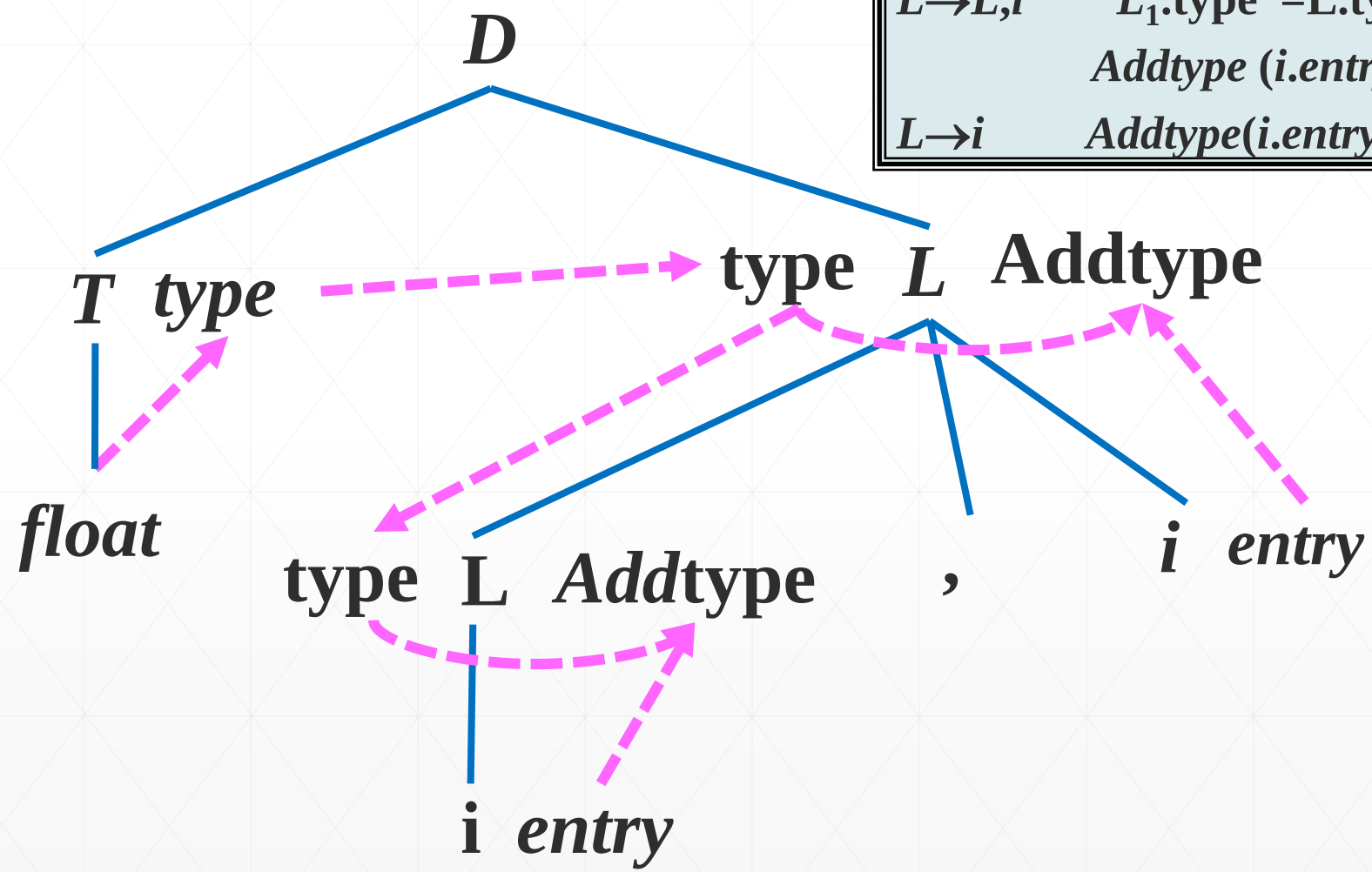
$L \rightarrow E$	L.val = E .val
$E \rightarrow E+T$	E.val = E_1.val + T.val
$E \rightarrow T$	E .val = T .val
$T \rightarrow T*F$	T.val = T_1.val × F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$digit	F.val = digit.lexval

例如， 3 * 5 + 4 ;



例如，float i_1，i_2

$D \rightarrow TL$	$L.type = T.type$
$T \rightarrow float$	$T.type = float$
$L \rightarrow L,i$	$L_1.type = L.type$
	$Addtype(i.entry, L.type)$
$L \rightarrow i$	$Addtype(i.entry, L.type)$



6.1 语法制导翻译

6.1.1 语法制导翻译基本原理

6.1.2 自上而下翻译器

6.1.3 自下而上翻译器

6.1.4 语法制导定义SDD—属性文法

6.1.5 语法制导翻译方案SDT

属性计算(加工)的过程即是语义处理的过程

依赖图刻画了属性计算时的一些顺序要求，

属性结点M到结点N有一条边，那么计算结点N对应的属性时，必须计算出M结点对应的属性。

根据依赖图，求出属性结点的一个拓扑排序，此排序就是属性结点的一个计算顺序。

可能无拓扑排序

实际应用中，结合语法分析时语法分析树的构造顺序，定义出相对应的属性文法，使属性计算顺序与分析树的展开顺序一致。

语法制导的翻译方案(Syntax-directed translation scheme,SDT)语法制导翻译模式

语法制导定义的补充.

给出SDD中的属性计算的可行实现方案。

根据两种语法分析方法，我们主要介绍两种相对应的属性文法

S-属性文法

L-属性文法

适合在语法分析过程中处理语义的两类属性文法

- S-属性文法 自下而上的属性翻译文法

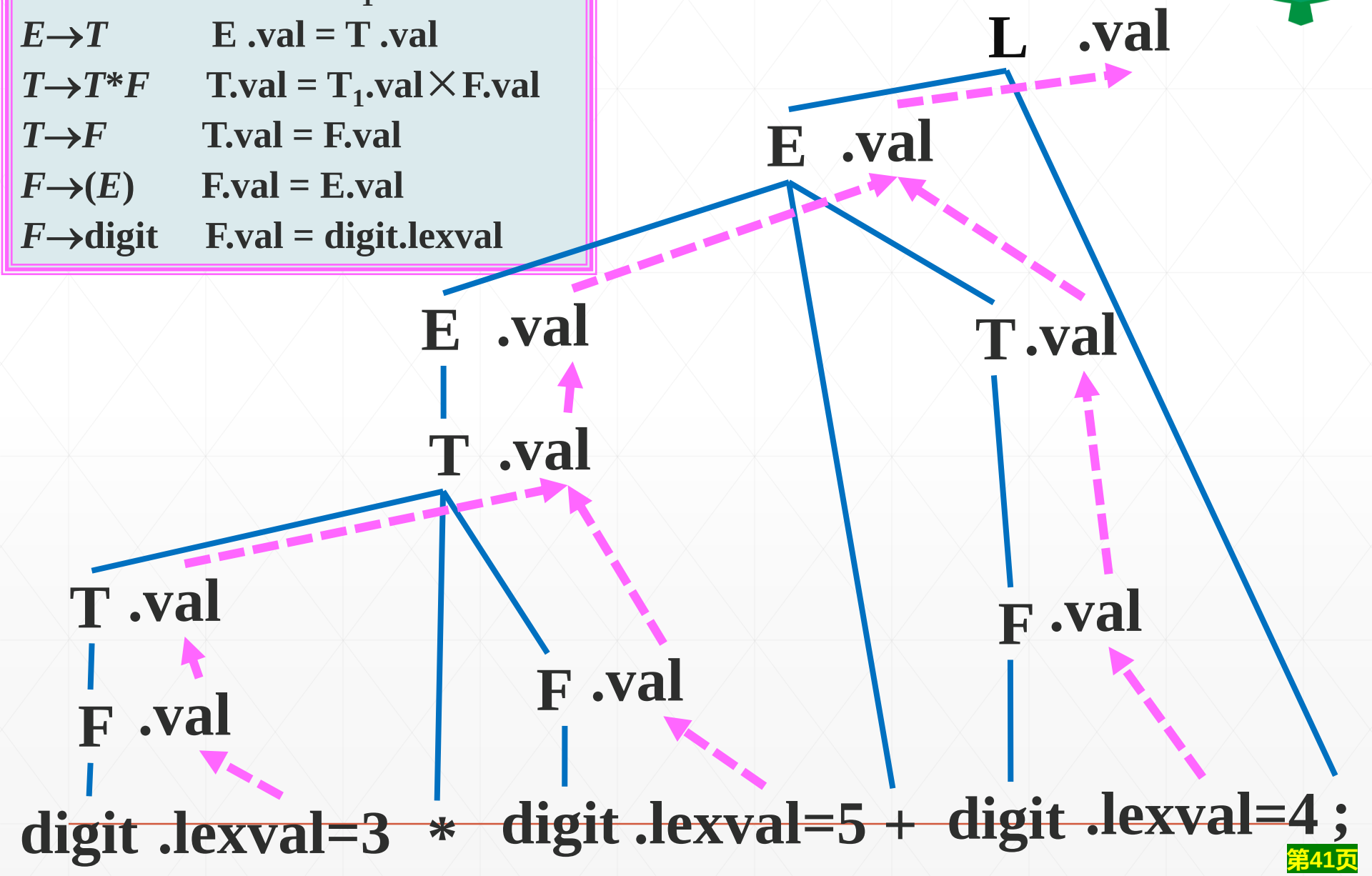
1.所有非终结符号只有综合属性

可以通过后续遍历语法分析树计算完所有属性

最简单的S属性描述
可能不严格
但实用

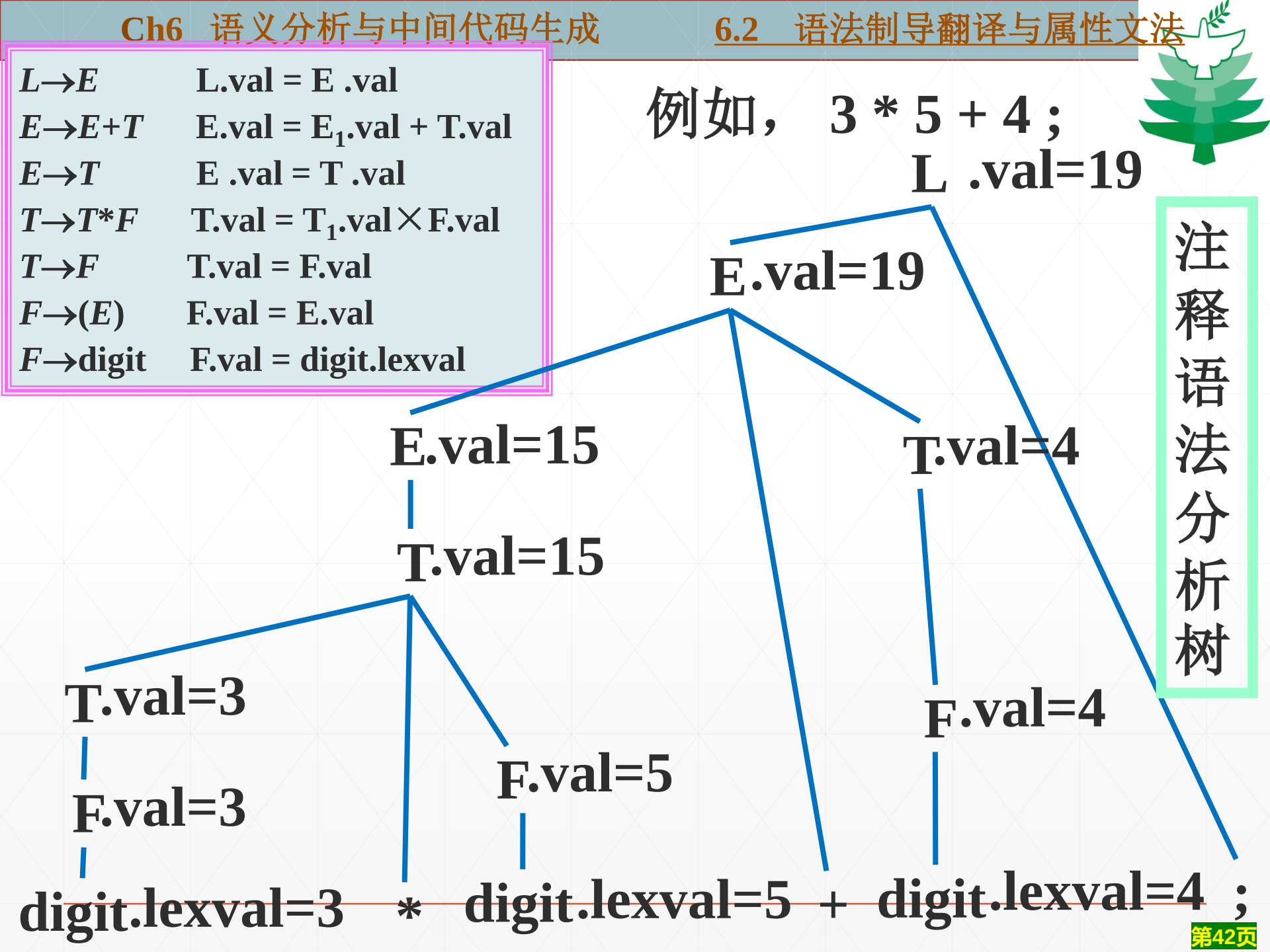
$L \rightarrow E$	L.val = E .val
$E \rightarrow E+T$	E.val = E_1.val + T.val
$E \rightarrow T$	E .val = T .val
$T \rightarrow T*F$	T.val = T_1.val × F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$digit	F.val = digit.lexval

例如， 3 * 5 + 4 ;



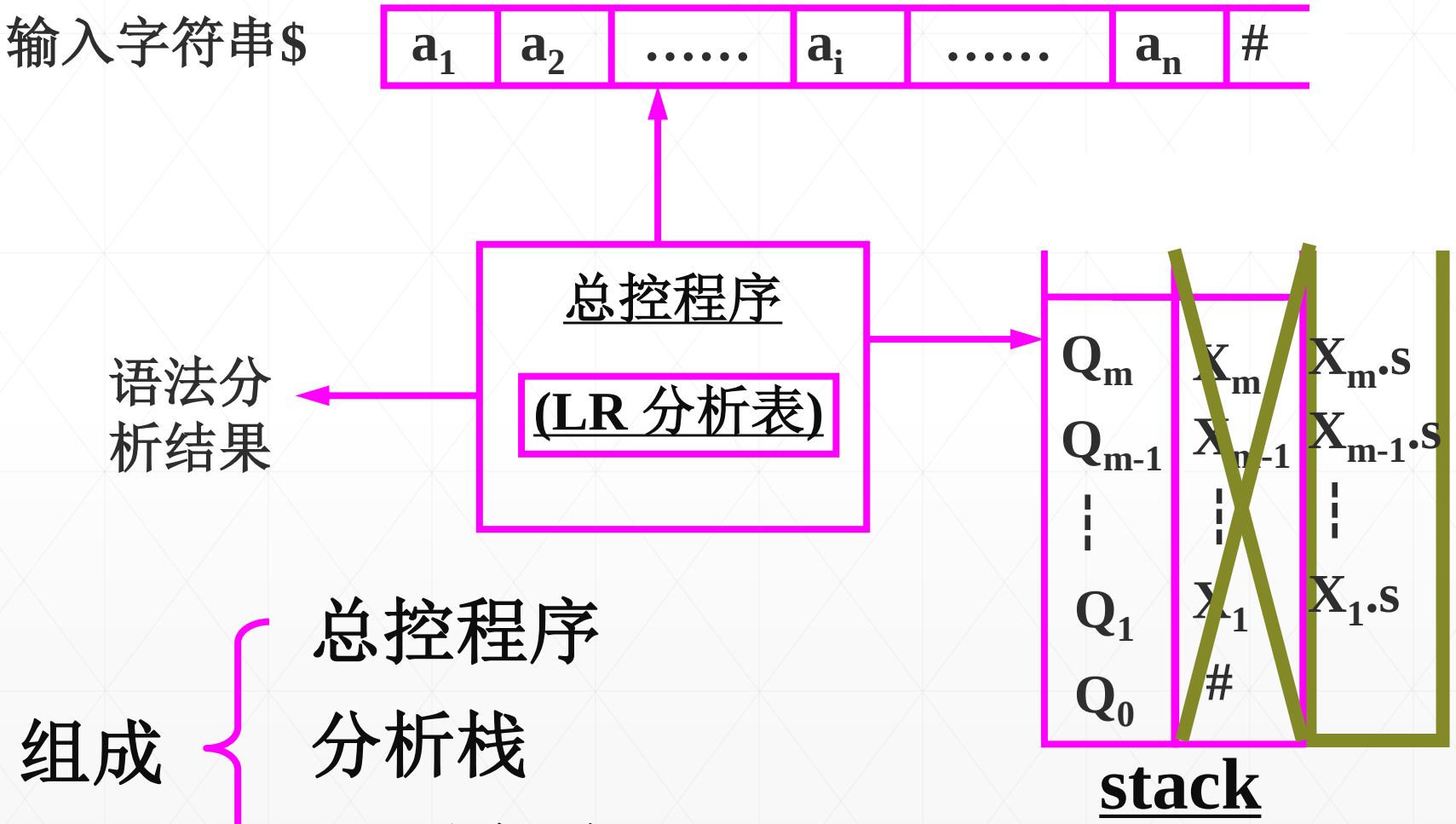
$L \rightarrow E$	L.val = E .val
$E \rightarrow E+T$	E.val = E_1.val + T.val
$E \rightarrow T$	E .val = T .val
$T \rightarrow T*F$	T.val = T_1.val × F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$digit	F.val = digit.lexval

例如， 3 * 5 + 4 ;



注释语法分析树

LR分析器逻辑结构


输入字符串$
a1 a2 …… ai …… an #
总控程序
(LR 分析表)
语法分析结果
Qm Qm-1 Q1 Q0
Xm Xm-1 X1 #
Xm.s Xm-1.s X1.s
stack
组成
总控程序
分析栈
LR分析表

简单运算的台式计算器的LR分析的SDT

产生式	语义动作
$L \rightarrow E$	{输出stack[top]}
$E \rightarrow E + T$	{stack[top-2]=stack[top-2]+stack[top];top=top-2}
$E \rightarrow T$	
$T \rightarrow T * F$	{stack[top-2]=stack[top-2]+stack[top];top=top-2}
$T \rightarrow F$	
$F \rightarrow (E)$	{stack[top-2]=stack[top-1];top=top-2}
$F \rightarrow$ digit	

适合在语法分析过程中处理语义的两类属性文法

- **L-属性文法**

自上而下的属性翻译文法

1.综合属性

2.继承属性

要求：

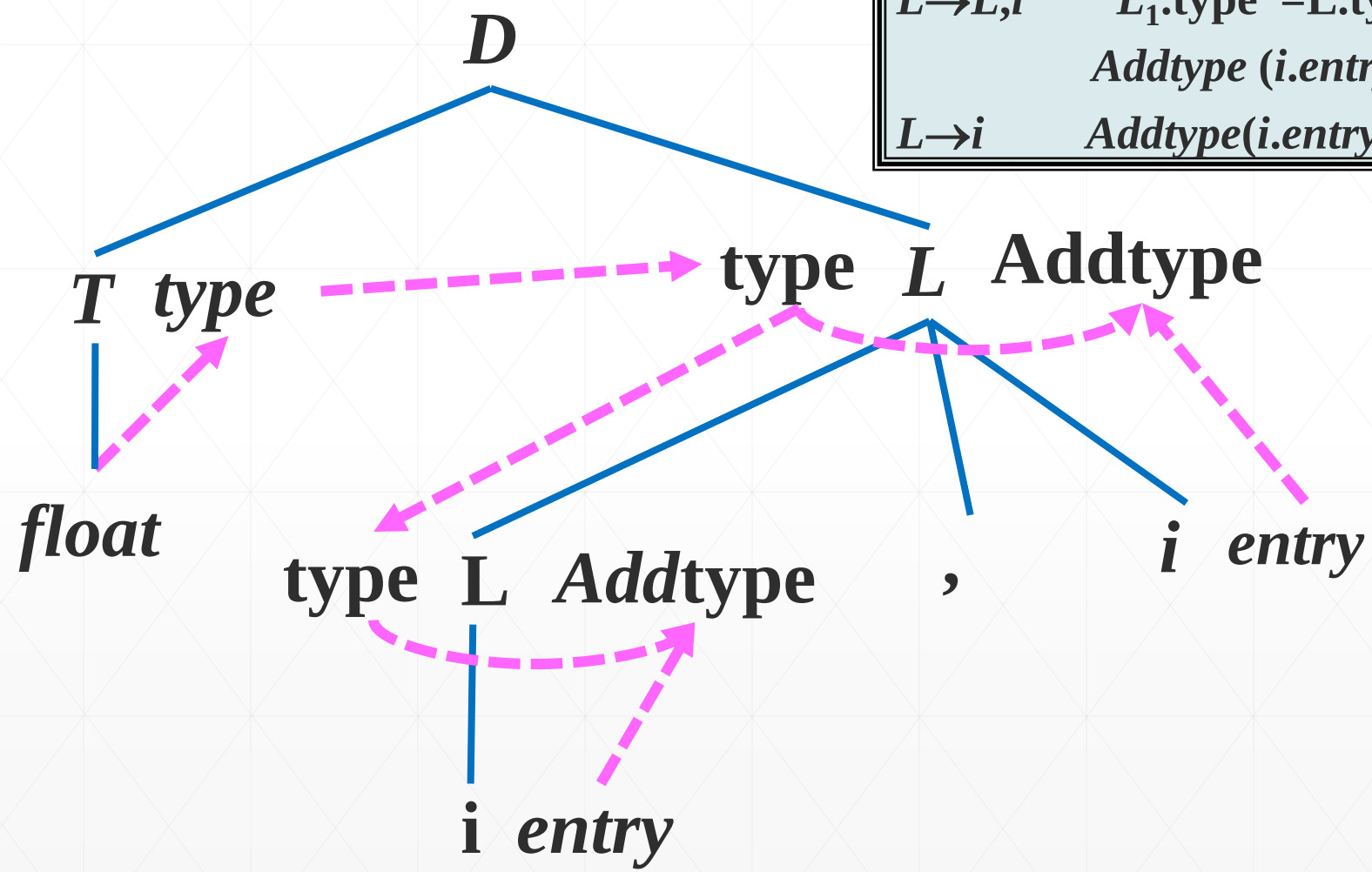
或者是左部非终结符号的继承属性

或者是位于其左侧兄弟的任何属性

可通过深度优先遍历语法分析树计算完所有属性

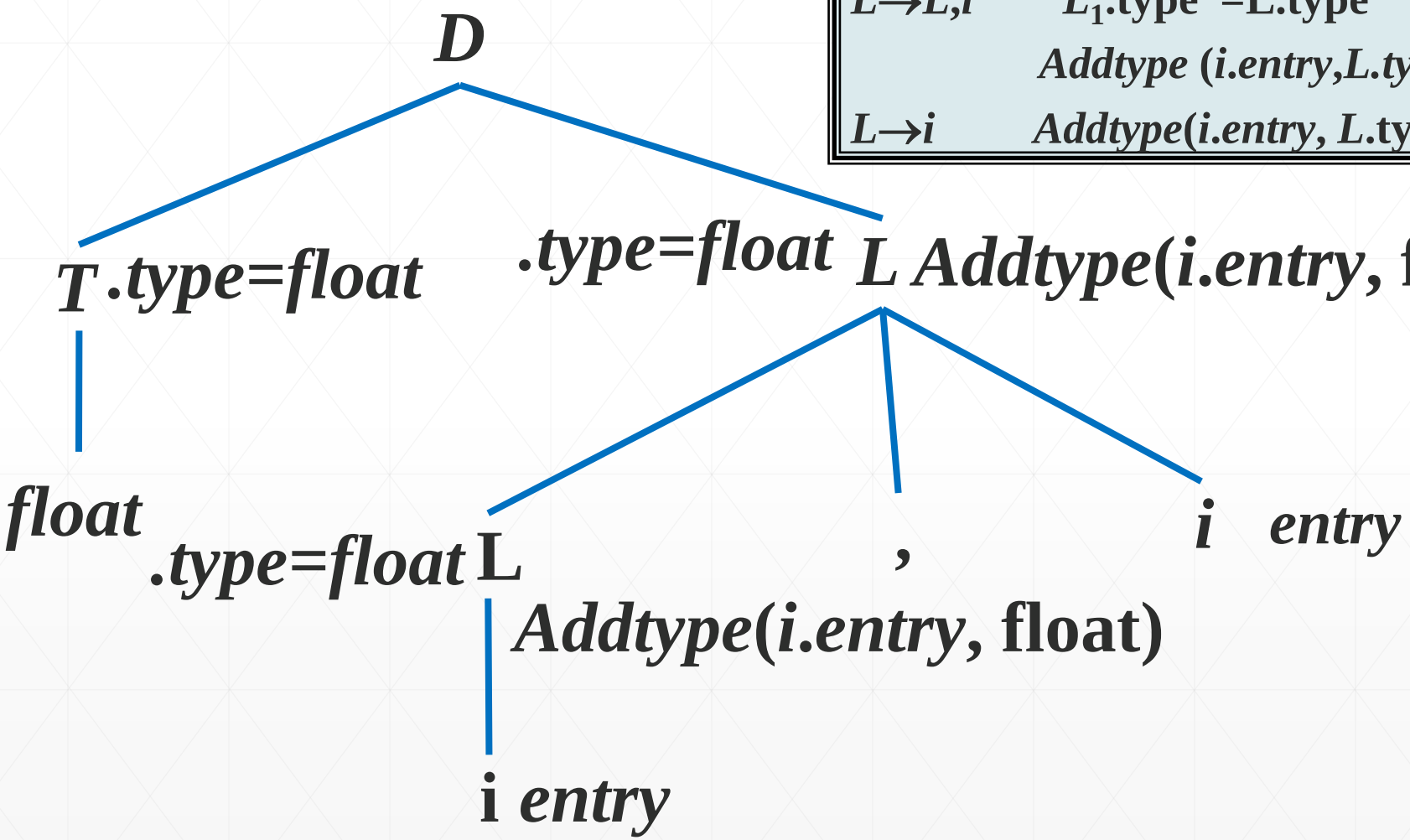
例如，float i_1，i_2

$D \rightarrow TL$	$L.type = T.type$
$T \rightarrow float$	$T.type = float$
$L \rightarrow L,i$	$L_1.type = L.type$
	$Addtype(i.entry, L.type)$
$L \rightarrow i$	$Addtype(i.entry, L.type)$



例如，float i_1，i_2

$D \rightarrow TL$	$L.type = T.type$
$T \rightarrow float$	$T.type = float$
$L \rightarrow L,i$	$L_1.type = L.type$
	$Addtype(i.entry, L.type)$
$L \rightarrow i$	$Addtype(i.entry, L.type)$



注释语法分析树

类型说明的SDT

产生式	语义动作
D→*T*	{*L*.type= *T*.type}
L	
T→*float*	*T.type=float*
T→*int*	*T.type=int*
L→	L_1.type=*L*.type
L,*i*	*Addtype* (*i.entry*,*L.type*)
L→*i*	*Addtype*(*i.entry*, *L*.type)

把属性计算(语义动作)用“{}”嵌入在产生式中，嵌入的位置表示他相对应的“计算时间”,综合属性在符号后，继承属性在符号前。

左递归文法修改后的语义处理

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

*R*有两个属性值：
继承：前面的结果
综合：最终的结果

$E \to TR$
$R \to +TR \mid \varepsilon$
$T \to FM$
$M \to *FM \mid \varepsilon$
$F \to (E)$
$F \to digit$

只考虑加法的情况

产生式	语义动作
$E \to T$	{R.i=T.val}
R	{E.val=R.s}
$R \to +T$	{R_1.i=R.i+T.val}
R	{R.s=R_1.s}
$R \to \varepsilon$	{R.s=R.i}

左递归文法修改后的语义处理

$E \rightarrow TR$

$R \rightarrow +TR|\varepsilon$

$T \rightarrow FM$

$M \rightarrow *FM|\varepsilon$

$F \rightarrow (E)$

$F \rightarrow$ digit

产生式	语义动作
$E \rightarrow T$	{R.i=T.val}
R	{E.val=R.s}
$R \rightarrow +T$	{R_1.i=R.i+T.val}
R	{R.s=R1.s}
$R \rightarrow \varepsilon$	{R.s=R.i}
$T \rightarrow F$	{M.i=F.val}
M	{T.val=M.s}
$M \rightarrow *F$	{M_1.i=M.i×F.val}
M	{M.s=M1.s}
$M \rightarrow \varepsilon$	{M.s=M.i}
$F \rightarrow (E)$	{F.val=E.val}
$F \rightarrow$ digit	{F.val=num.val}

*R*和*M*有两个属性值：

继承：前面的结果

综合：最终的结果