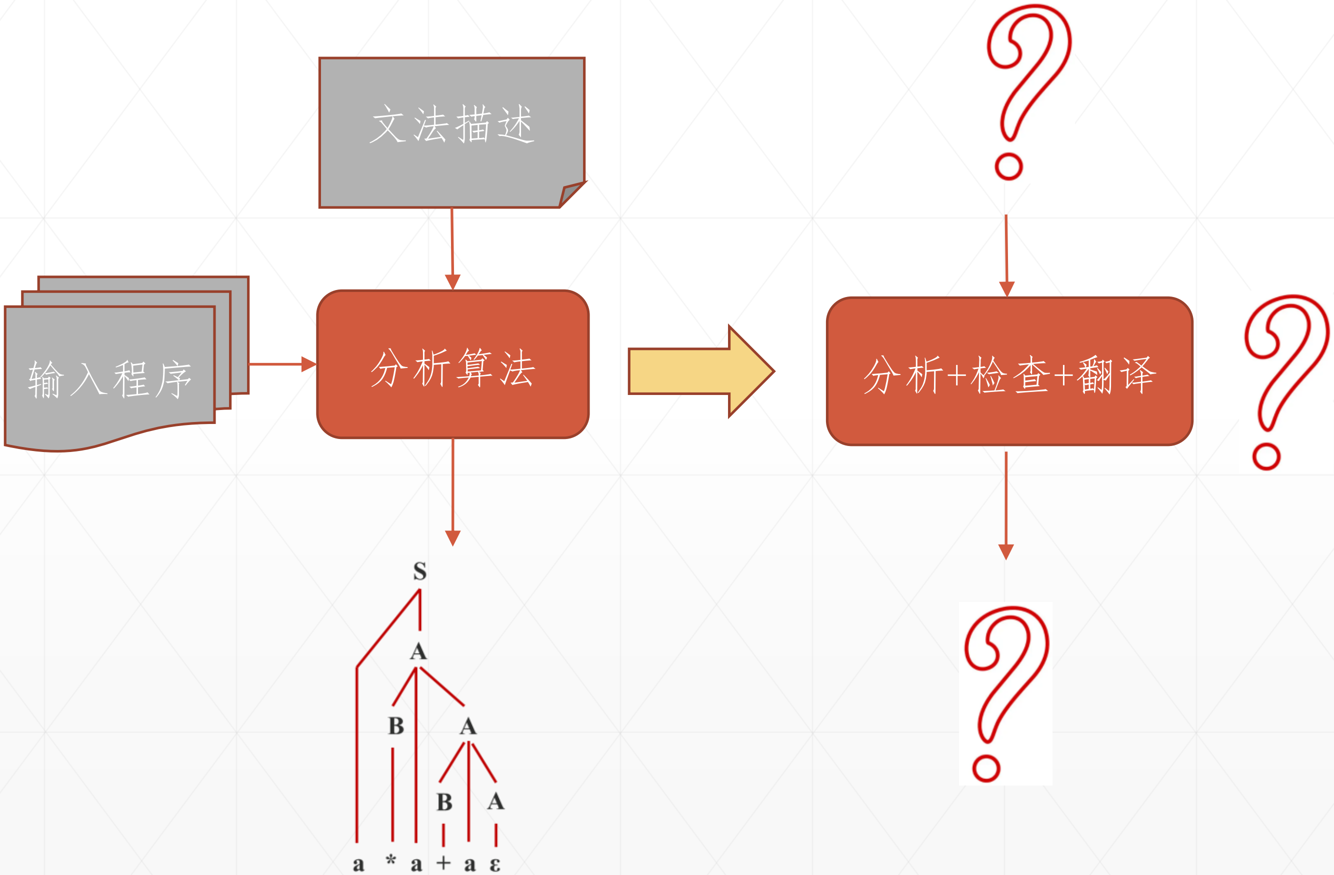
编译原理与设计

北京理工大学 计算机学院

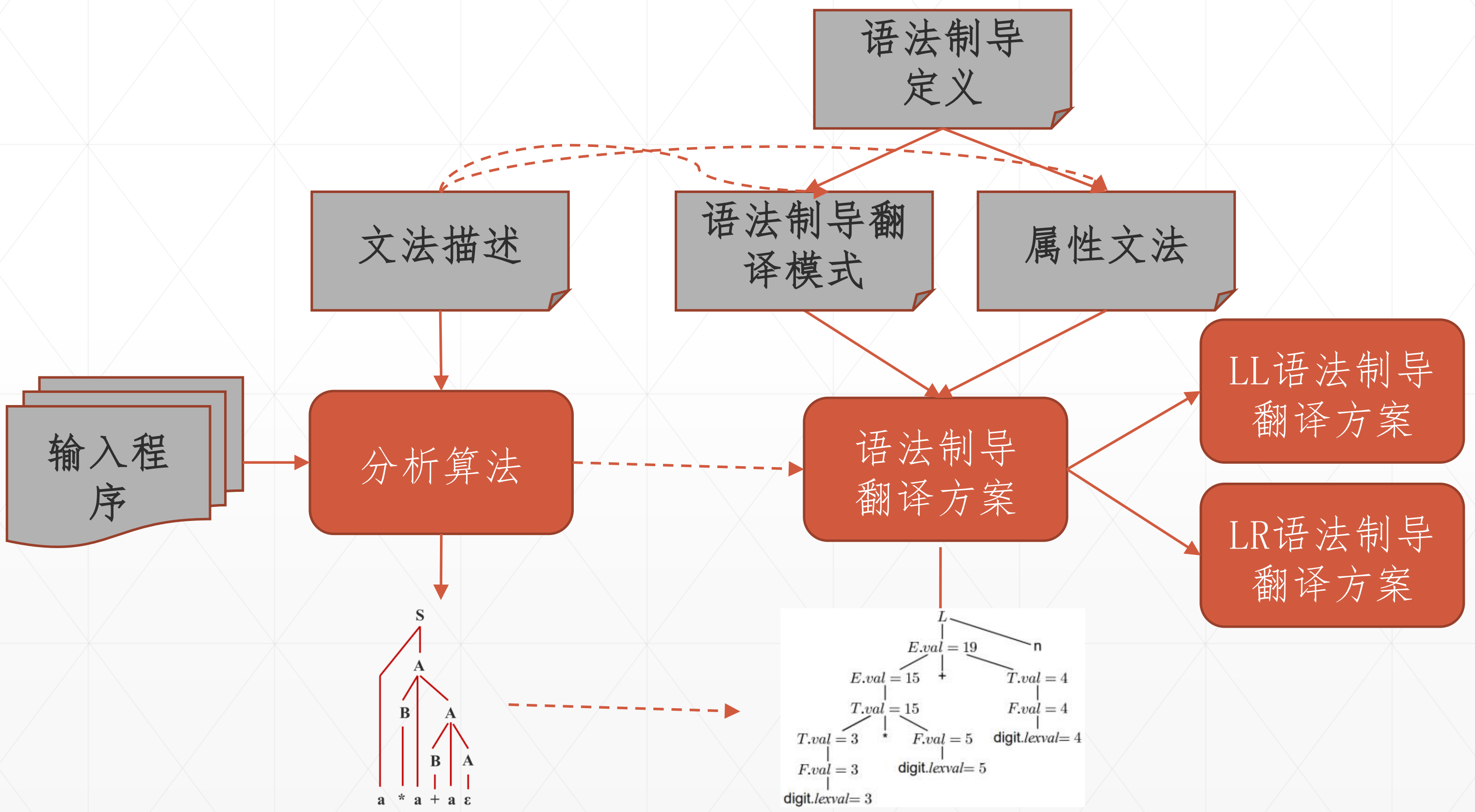
语法制导翻译

- 相关背景



语法制导翻译

- 在语法分析的同时完成代码翻译



分析树　　Decorated AST

语法制导翻译

- **基本思想**：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涉及到的两个概念
  - 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：

$A \to \alpha$，β

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

$\boxed{A \to \alpha}$

源文法

$\boxed{A \to \beta}$

翻译文法

语法制导翻译模型

- 例：简单表达式中缀到后缀翻译的SDTS。

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

***R*中规则为：**

$E \rightarrow E + T$,	$E\ T\ +$
$E \rightarrow E - T$,	$E\ T\ -$
$E \rightarrow -T$,	$T\ @$
$E \rightarrow T$,	T
$T \rightarrow (E)$,	E
$T \rightarrow i$,	i

翻译对偶

SDTS=(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($\alpha A\beta$, $\alpha' A\beta'$)$\Rightarrow$ ($\alpha\delta\beta$, $\alpha'\gamma\beta'$)

翻译对偶

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

$-(i+i)-i$是该文法的句子，其对应的翻译为：

$$(E, E) \Rightarrow (E-T, ET-) \Rightarrow (-T-T, T@T-) \Rightarrow (-(E)-T, E@T-)$$

$$\Rightarrow (-(E+T)-T, ET+@T-) \Rightarrow (-(T+T)-T, TT+@T-)$$

$$\Rightarrow (-(i+T)-T, iT+@T-) \Rightarrow (-(i+i)-T, ii+@T-)$$

$$\Rightarrow (-(i+i)-i, \boxed{ii+@i-})$$

- 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：

$E \to E+T$,　　$ET+$

$E \to E-T$,　　$ET-$

$E \to -T$,　　$T@$

$E \to T$,　　T

$T \to (E)$,　　E

$T \to i$,　　i

语法制导翻译

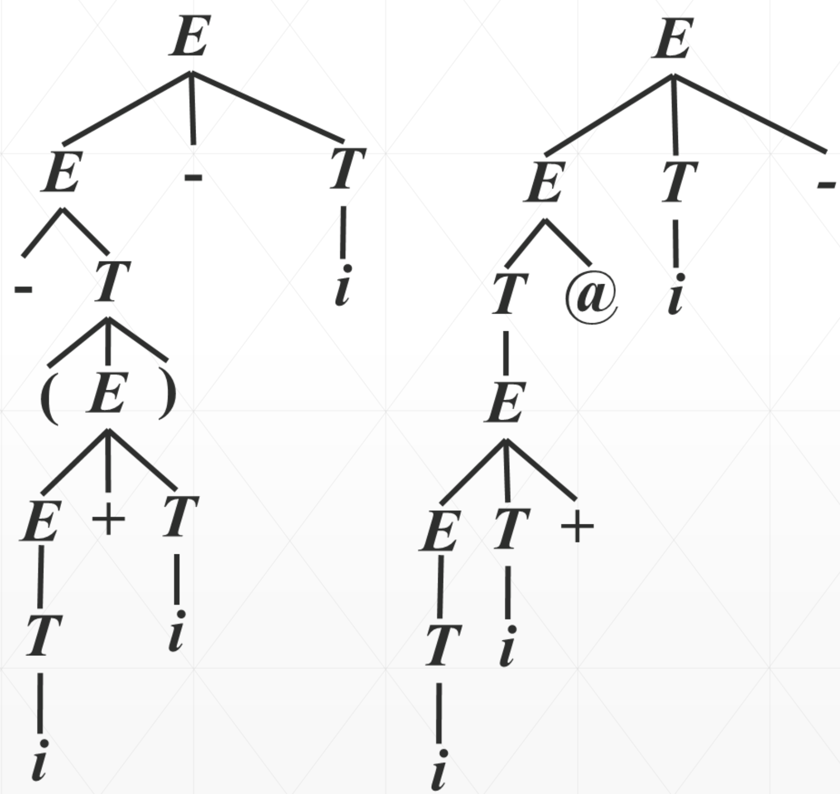
- 语法制导的翻译过程也可以用分析树表示

- 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：

$E \rightarrow E + T$,	$E\ T +$
$E \rightarrow E - T$,	$E\ T -$
$E \rightarrow -T$,	T @
$E \rightarrow T$,	T
$T \rightarrow (E)$,	E
$T \rightarrow i$,	i



语法制导翻译

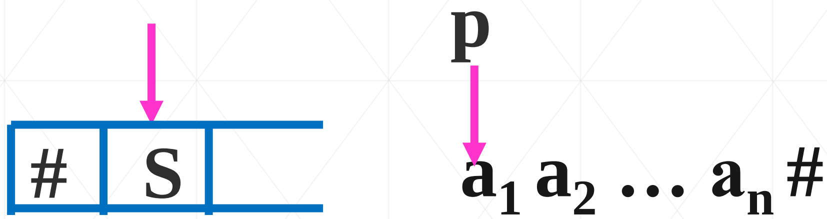
- 自上而下翻译：简单SDTS

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



(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<u>终结符号</u>，则对于：

① X＝a_i＝“#”，处理成功，停止处理过程；

② X＝a_i≠“#”，则将X从分析栈顶退掉，p指向下一个输入字符；

③ X≠a_i，表示不匹配的出错情况。

(3)若X是文法的<u>输出符号</u>，则从栈中弹出并输出X。

语法制导翻译

- 带翻译处理的LL(1)分析

(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

$$X \rightarrow \alpha_0 A_1 \alpha_1 A_2 \ldots A_k \alpha_k,\ \beta_0 A_1 \beta_1 A_2 \ldots A_k \beta_k$$

(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。

语法制导翻译

- 带翻译处理的LL(1)分析

例：简单的SDTS的R规则为

$S \to (S)S$， $xSySz$

$S \to \varepsilon$， w

源文法的LL(1)分析表

	(	)	#
S	$S \to (S)S$	$S \to \varepsilon$	$S \to \varepsilon$

步骤	分析栈	待匹配串	分析动作
0	#S	() ()#	
1	#zS)yS(x	() ()#	输出x
2	#zS)yS(	() ()#	P++
3	#zS)yS	)()#	
4	#zS)yw	)()#	
5	#zS)y	)()#	输出y
6	#zS)	)()#	P++
7	#zS	()#	
8	#zzS)yS(x	()#	输出x
9	#zzS)yS(	()#	P++
10	#zzS)yS	)#	
11	#zzS)yw	)#	
12	#zzS)y	)#	
13	#zzS)	)#	P++
14	#zzS	#	
15	#zzw	#	

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

翻译结果为：

xwyxwywzz

语法制导翻译

- 自下而上翻译器：简单后缀SDTS

1. 简单**SDTS;**

2. 每一条规则都有如下形式

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

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。

语法制导翻译

- 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

$X \to \alpha_1 A_1 \alpha_2 A_2 \ldots A_k \alpha_k, A_1 A_2 \ldots A_k \beta$，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。

语法制导翻译

- 例：简单表达式的中缀到后缀翻译的SDTS

简化为二义性文法

①$E \to E+E$，$EE+$

②$E \to E-E$，$EE-$

③$E \to -E$，$E@$

④$E \to (E)$，E

⑤$E \to i$，i

状态	ACTION							GOTO
	+	-(双目)	-	(	)	i	#	E
0			S2	S3		S4		1
1	S5	S6					acc	
2			S2	S3		S4		7
3			S2	S3		S4		8
4	r5	r5	r5	r5	r5	r5	r5	
5			S2	S3		S4		9
6			S2	S3		S4		10
7	r3	r3	r3	r3	r3	r3	r3	
8	S5	S6			S11			
9	r1	r1	r1	r1	r1	r1	r1	
10	r2	r2	r2	r2	r2	r2	r2	
11	r4	r4	r4	r4	r4	r4	r4	

语法制导翻译

句子-(*i*+*i*)-*i* 的翻译过程

步	栈		余留串	分析动作
	符号栈	状态栈		
0	#	0	-(*i*+*i*)-*i* #	S2
1	#-	02	(*i*+*i*)-*i*#	S3
2	#-(	023	*i*+*i*)-*i*#	S4
3	#-(*i*	0234	+*i*)-*i*#	r5,输出*i*
4	#-(*E*	0238	+*i*)-*i*#	S5
5	#-(*E*+	02385	*i*)-*i*#	S4
6	#-(*E*+*i*	023854	)-*i* #	r5,输出*i*
7	#-(*E*+*E*	023859	)-*i* #	r1,输出+
8	#-(*E*	0238	)-*i*#	S11
9	#-(*E*)	023811	-*i*#	r4
10	#-*E*	027	-*i*#	r3,输出@
11	#*E*	01	-*i*#	S6
12	#*E*-	016	*i*#	S4
13	#*E*-*i*	0164	#	r5,输出*i*
14	#*E*-*E*	01610	#	r2,输出-
15	#*E*	01	#	acc

状态	ACTION							GOTO
	+	-(双目)	-	(	)	i	#	E
0			S2	S3		S4		1
1	S5	S6					acc	
2			S2	S3		S4		7
3			S2	S3		S4		8
4	r5	r5	r5	r5	r5	r5	r5	
5			S2	S3		S4		9
6			S2	S3		S4		10
7	r3	r3	r3	r3	r3	r3	r3	
8	S5	S6			S11			
9	r1	r1	r1	r1	r1	r1	r1	
10	r2	r2	r2	r2	r2	r2	r2	
11	r4	r4	r4	r4	r4	r4	r4	

①*E*→*E*+*E*，*EE*+　②*E*→*E*-*E*，*EE*-　③*E*→-*E*，*E*@

④*E*→(*E*)，*E*　⑤*E*→*i*，*i*

语法制导翻译

- 简单SDTS修改为简单后缀SDTS

条件语句文法：

S→if E then S else S

对应的简单SDTS为：

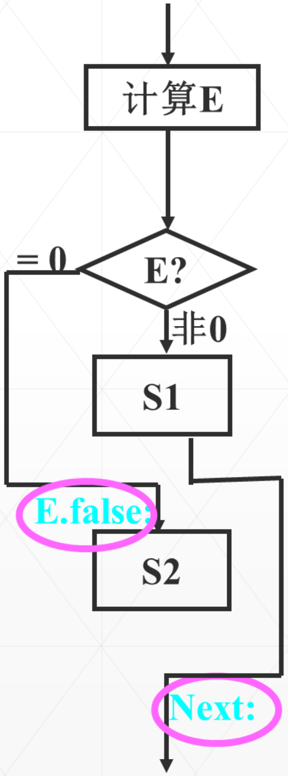
S→if E then S else S ， <E>若E的值为假跳到标号E.false <S1>跳转到标号Next E.false：<S2> Next:

修改为对应的简单后缀SDTS：

S→T else S，TS Next:

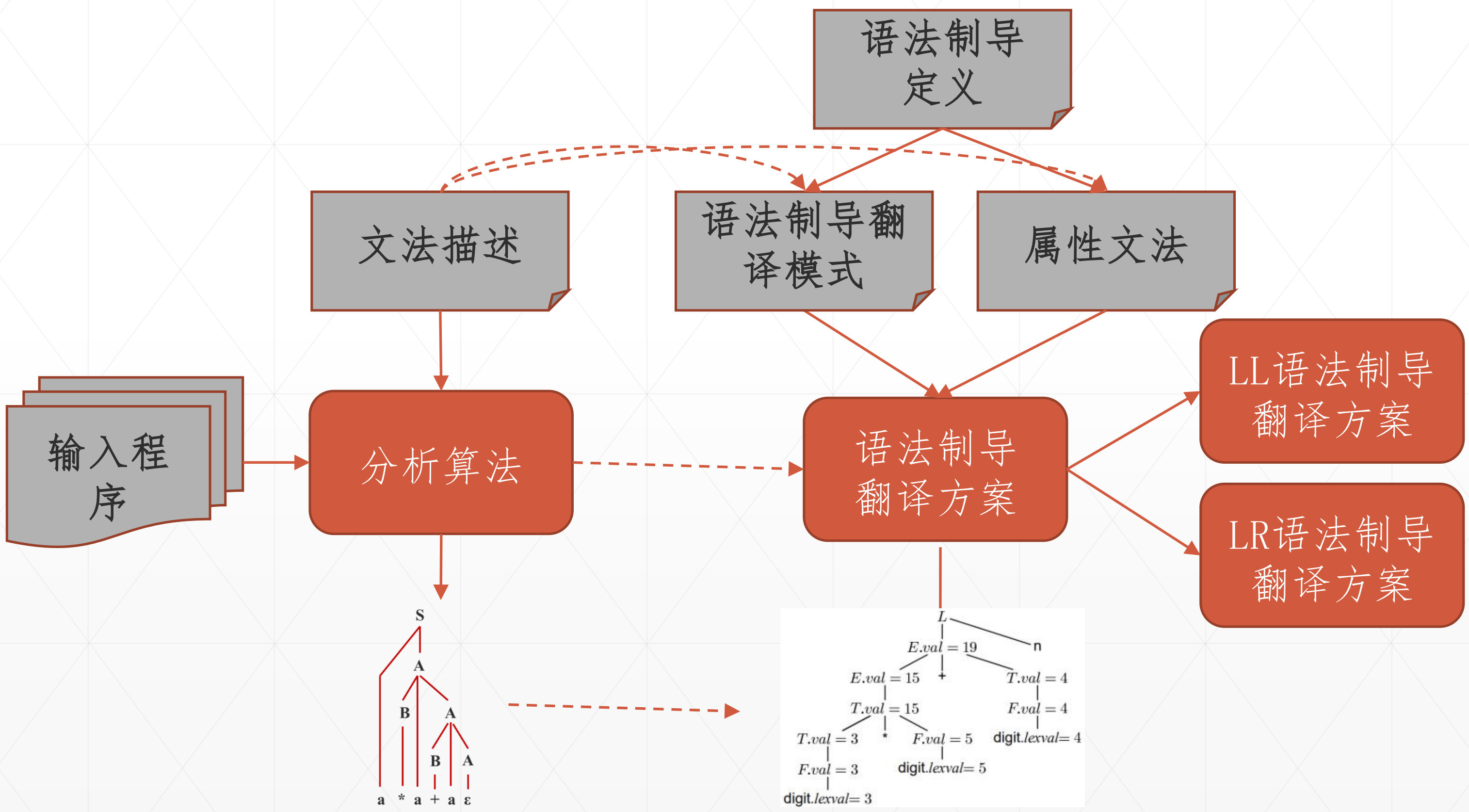
T→I then S，IS跳转到标号Next E.false：

I→if E，E 若E的值为假跳到标号E.false



语法制导翻译

- 在语法分析的同时完成代码翻译



分析树　　　　Decorated AST

属性文法与语法制导定义

- 高层次的语义描述，隐藏了实现细节
- 借助属性文法进行语法制导定义表述

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

SDD

$E \rightarrow E + T$	$E.code = E.code \| \| T.code \| \| '+'$
$E \rightarrow E - T$	$E.code = E.code \| \| T.code \| \| '-'$
$E \rightarrow T$	$E.code = T.code$
$T \rightarrow 0$	$T.code = '0'$
$T \rightarrow 1$	$T.code = '1'$
$\cdots$	
$T \rightarrow 9$	$T.code = '9'$

中缀到后缀表达式翻译

属性文法与语法制导翻译

- **属性文法**
  - **综合属性/Synthesized Attributes.**
  - **继承属性/Inherited Attributes.**

PRODUCTION	SEMANTIC RULE
$L \to E\mathrm{n}$	$print(E.val)$
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to \mathrm{digit}$	$F.val := \mathrm{digit}.lexval$

PRODUCTION	SEMANTIC RULE
$D \to TL$	$L.in := T.type$
$T \to \mathrm{int}$	$T.type := integer$
$T \to \mathrm{real}$	$T.type := real$
$L \to L_1, \mathrm{id}$	$L_1.in := L.in$; $addtype(\mathrm{id}.entry, L.in)$
$L \to \mathrm{id}$	$addtype(\mathrm{id}.entry, L.in)$

属性文法与语法制导翻译

- **无限制属性文法**：可以使用任何综合属性和继承属性
  - 建立属性计算的依赖图
  - 根据属性依赖图计算属性
- S-**属性文法**. 仅使用综合属性
  - 自下而上的属性翻译文法
  - 可以通过后续遍历语法分析树计算完所有属性
- L-**属性文法**：使用综合属性+继承属性（左侧兄弟节点任何属性或者左端非终结符继承属性）
  - 自上而下的属性翻译文法
  - 可通过深度优先遍历语法分析树计算完所有属性

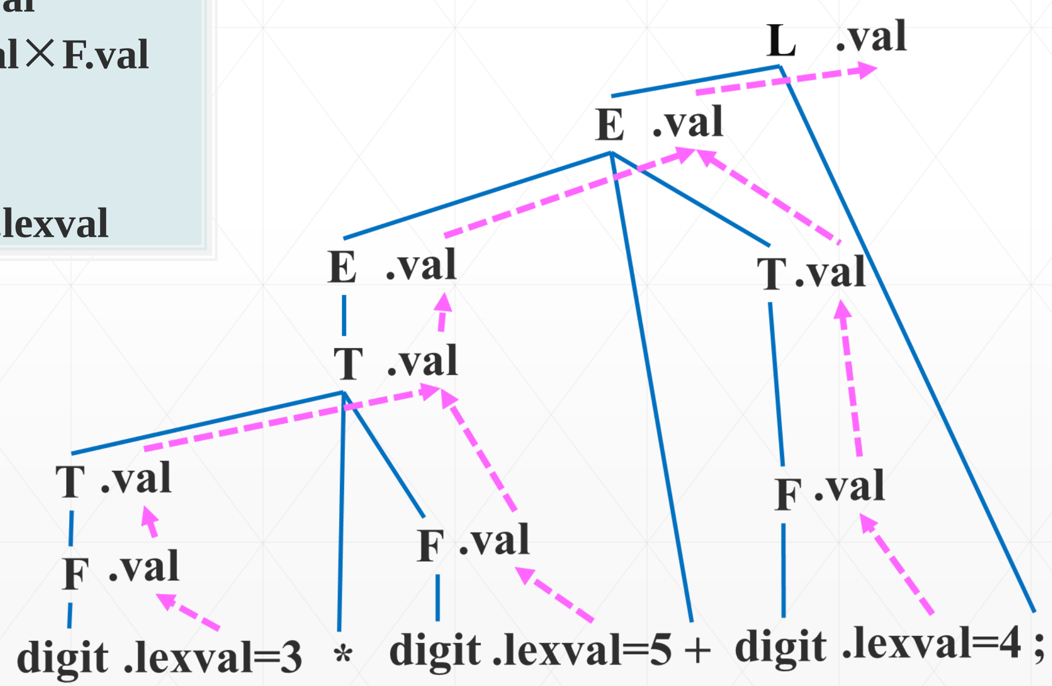
S-属性文法/L-属性文法适合在语法分析过程中处理语义的两类属性文法

属性文法与语法制导翻译

▪ S-属性文法

$L \rightarrow E$	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

例如， 3 * 5 + 4 ;



属性文法与语法制导翻译

- Evaluation of S-Attributed Definitions

  - The parser keeps the values of the synthesized attributes in its stack.
  - Whenever a reduction $A \to \alpha$ is made, the attribute for A is computed from the attributes of α which appear on the stack.
  - Thus, a translator for an S-Attributed Definition can be implemented by extending the stack of an LR-Parser.

- Synthesized attributes are computed just before each reduction:
  - Before the reduction $A \to XYZ$ is made, the attribute for A is computed: $A.a := f(val[top], val[top-1], val[top-2])$.

state	*val*
Z	$Z.x$
Y	$Y.x$
X	$X.x$
$\ldots$	$\ldots$

属性文法与语法制导翻译

- Evaluation of S-Attributed Definitions

- **Example.** Consider the S-attributed definitions for the arithmetic expressions. To evaluate attributes the parser executes the following code

PRODUCTION	CODE
$L \to E\mathsf{n}$	$print(val[top-1])$
$E \to E_1 + T$	$val[ntop] := val[top] + val[top-2]$
$E \to T$	
$T \to T_1 * F$	$val[ntop] := val[top] * val[top-2]$
$T \to F$	
$F \to (E)$	$val[ntop] := val[top-1]$
$F \to \mathsf{digit}$	

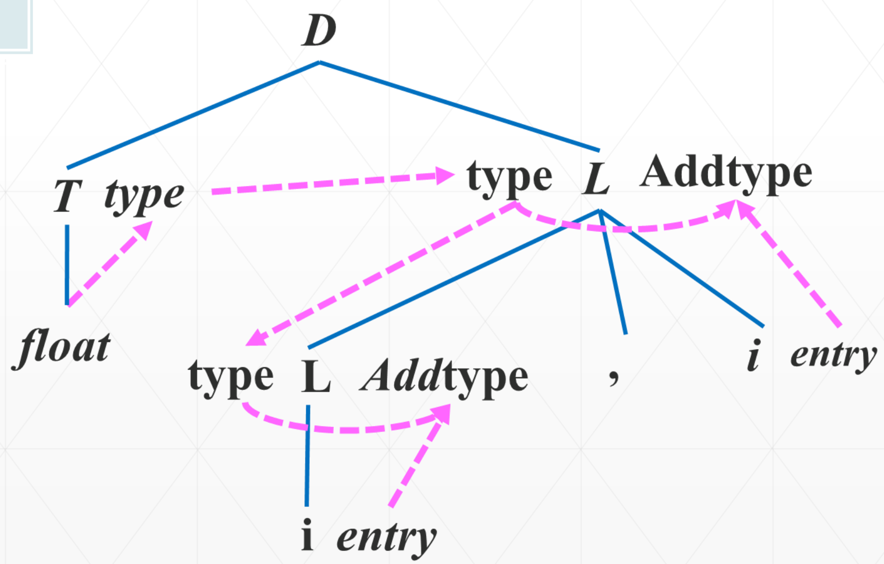
- The variable *ntop* is set to the *new top of the stack*. After a reduction is done *top* is set to *ntop*.
  - When a reduction $A \to \alpha$ is done with $\mid \alpha \mid = r$, then $ntop = top - r + 1$.

属性文法与语法制导翻译

- L-属性文法

D→TL	*L.type=T.type*
T→float	*T.type=float*
L→L,i	L_1.type =L.type
	Addtype (*i.entry*,*L.type*)
L→i	*Addtype*(*i.entry*, L.type)

例如，float i_1， i_2



属性文法与语法制导翻译

- Evaluation of L-Attributed Definitions

- **L-Attributed Definitions** contain both synthesized and inherited attributes but do not need to build a dependency graph to evaluate them.
- **Definition.** A syntax directed definition is *L-Attributed* if each *inherited attribute* of X_j in a production $A \to X_1 \ldots X_j \ldots X_n$, depends only on:
  1. The attributes of the symbols to the **left** (this is what L in *L-Attributed* stands for) of X_j, i.e., $X_1 X_2 \ldots X_{j-1}$, and
  2. The inherited attributes of A.
- **Note.** An S-Attributed definition is also L-Attributed since the restrictions only apply to inherited attributes.

属性文法与语法制导翻译

- Evaluation of L-Attributed Definitions

  - L-Attributed Definitions are a class of syntax directed definitions whose attributes can always be evaluated by single traversal of the parse-tree.
  - The following procedure evaluate L-Attributed Definitions by mixing PostOrder (synthesized) and PreOrder (inherited) traversal.

  Algorithm L-Eval(n: Node). *Input:* Parse-Tree node from an L-Attribute Definition. *Output:* Attribute evaluation.

  Begin
      For each child m of n, from left-to-right Do Begin;
          evaluate inherited attributes of m;
          L-Eval(m)
      End;
      evaluate synthesized attributes of n
  End.

属性文法与语法制导翻译

- **无限制属性文法**
  - **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。

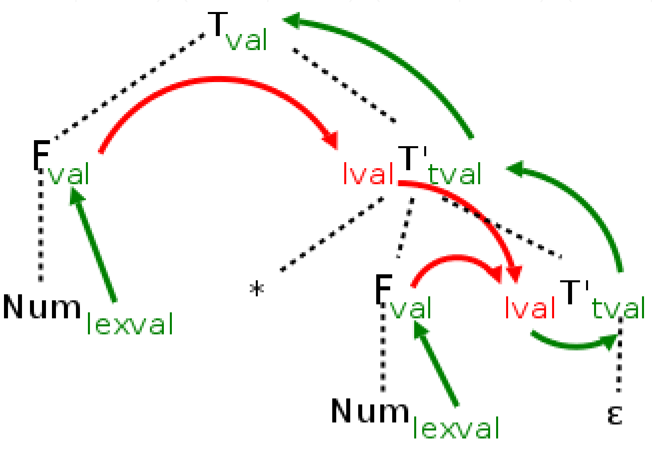
实际应用中，结合语法分析时语法分析树的构造顺序，定义出相对应的属性文法，使属性计算顺序与分析树的展开顺序一致。

属性文法与语法制导翻译

Dependency Graph

- Attribute at the head of an arrow depends on the on the one at the tail
- Must evaluate the head attribute **after** evaluating the tail attribute

Production	Semantic Rules	Type
T → F T'	T'.lval = F.val	Inherited
	T.val = T'.tval	Synthesized
T' → * F T_1'	T'_1.lval = T'.lval * F.val	Inherited
	T'.tval = T'_1.tval	Synthesized
T' → ε	T'.tval = T'.lval	Synthesized
F → num	F.val = num.lexval	Synthesized



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

- 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
- 具体处理：
  - 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。

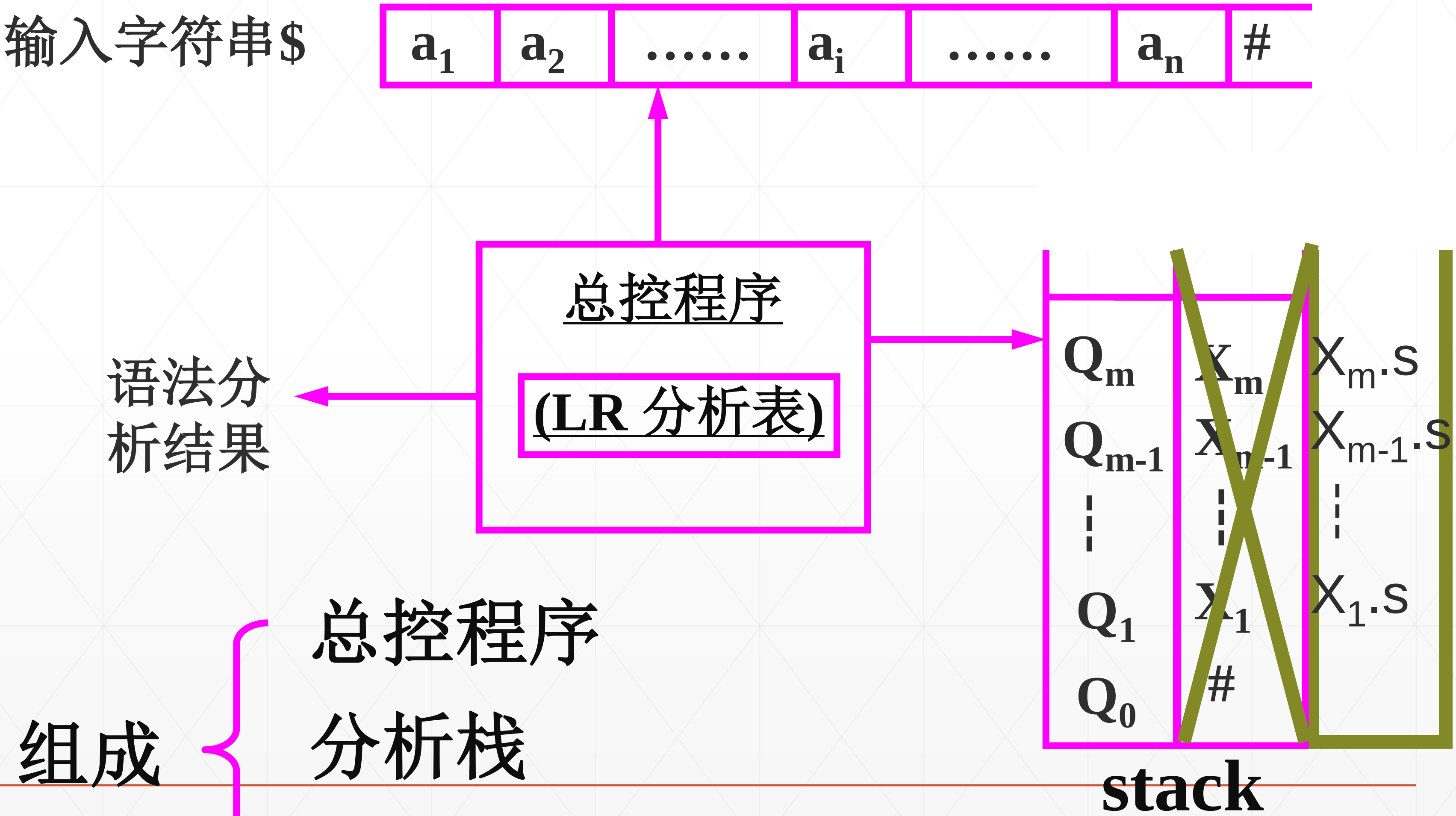
语法制导的翻译方案

- 简单运算的台式计算器的SDT

$$
\begin{aligned}
&L \to E \; \{L.\text{val} = E.\text{val}\} \\
&E \to E + T \; \{E.\text{val} = E_1.\text{val} + T.\text{val}\} \\
&E \to T \;\; \{E.\text{val} = T.\text{val}\} \\
&T \to T * F \;\; \{T.\text{val} = T_1.\text{val} \times F.\text{val}\} \\
&T \to F \;\; \{T.\text{val} = F.\text{val}\} \\
&F \to (E) \;\; \{F.\text{val} = E.\text{val}\} \\
&F \to \text{digit} \;\; \{F.\text{val} = \text{digit}.\text{lexval}\}
\end{aligned}
$$

语法制导的翻译方案

- LR分析器逻辑结构



组成
- 总控程序
- 分析栈
- LR分析表

语法制导的翻译方案

简单运算的台式计算器的LR分析的SDT

$L \rightarrow E$ {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}

$F \rightarrow \text{digit}$

语法制导的翻译方案

简单运算的台式计算器的LL分析的SDT

$L \to E \ \{L.\text{val} = E.\text{val}\}$
$E \to E + T \ \{E.\text{val} = E_1.\text{val} + T.\text{val}\}$
$E \to T \ \{E.\text{val} = T.\text{val}\}$
$T \to T * F \ \{T.\text{val} = T_1.\text{val} \times F.\text{val}\}$
$T \to F \ \{T.\text{val} = F.\text{val}\}$
$F \to (E) \ \{F.\text{val} = E.\text{val}\}$
$F \to \text{digit} \ \{F.\text{val} = \text{digit}.\text{lexval}\}$

R有两个属性值:
继承：前面的结果
综合：最终的结果

$E \to TR$
$R \to +TR|\varepsilon$
$T \to FM$
$M \to *FM|\varepsilon$
$F \to (E)$
$F \to \text{digit}$

只考虑加法的情况

$E \to T\{\text{R.i=T.val}\}$
$\quad R\{\text{E.val=R.s}\}$
$R \to +T\{\text{R}_1\text{.i=R.i+T.val}\}$
$\quad R\{\text{R.s=R}_1\text{.s}\}$
$R \to \varepsilon\{\text{R.s=R.i}\}$

语法制导的翻译方案

简单运算的台式计算器的LL分析的SDT

$E \to TE'$
$E' \to +TE' | \varepsilon$
$T \to FT'$
$T' \to *FT' | \varepsilon$
$F \to (E)$
$F \to \text{digit}$

$E \to T\{E'.i=T.val\}$
$\quad R\{E.val=E'.s\}$
$E' \to +T\{E'_1.i=E'.i+T.val\}$
$\quad R\{E'.s=E'_1.s\}$
$E' \to \varepsilon\ \{E'.s=E'.i\}$
$T \to F\{T'.i=F.val\}$
$\quad T'\{T.val=T'.s\}$
$T' \to *F\{T'_1.i=T'.i \times F.val\}$
$\quad T'\{T'.s=T'_1.s\}$
$T' \to \varepsilon\{T'.s=T'.i\}$
$F \to (E)\ \{F.val=E.val\}$
$F \to \text{digit}\{F.val=num.val\}$

*R*和*M*有两个属性值：

继承：前面的结果

综合：最终的结果

语法制导的翻译方案

- 简单运算的台式计算器的LL分析的SDT

$E \to T$ {E'.i=T.val}

　　E' {E.val=E'.s}

$E' \to +T$ {E'_1.i=E'.i+T.val}

　　E_1' {E'.s=E'_1.s}

$E' \to \varepsilon$ {E'.s=E'.i}

$T \to F$ {T'.i=F.val}

　　T' {T.val=T'.s}

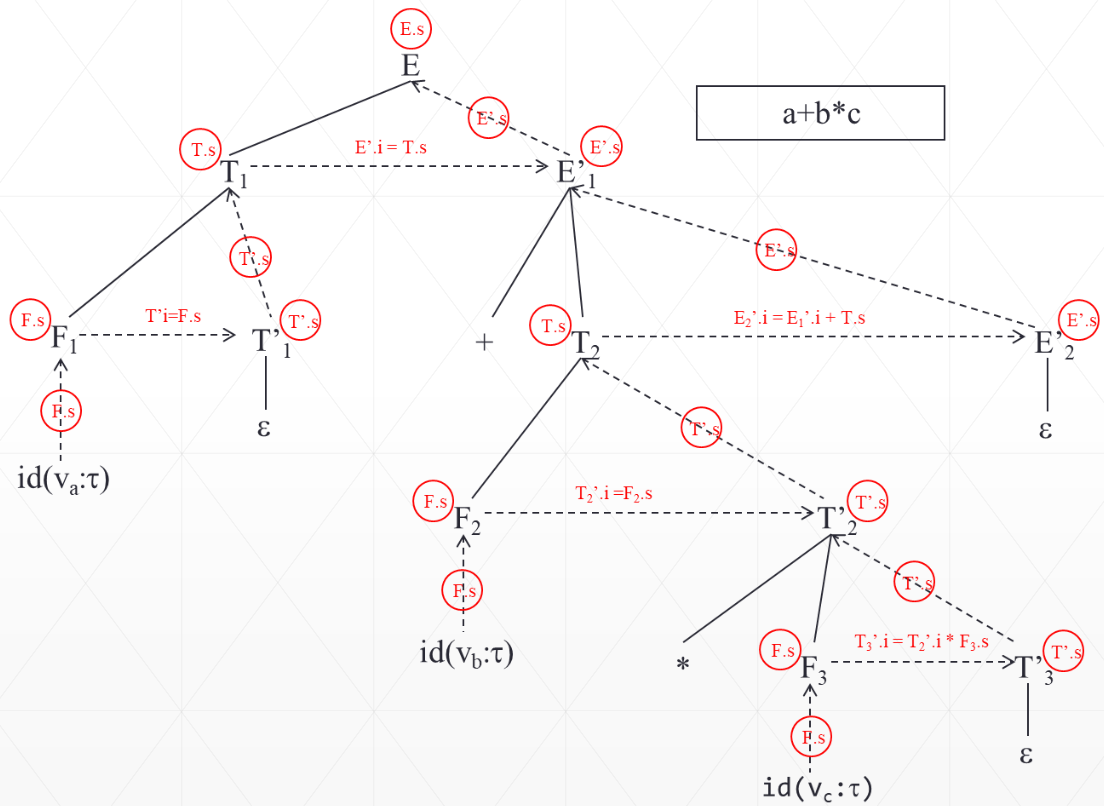
$T' \to *F$ {T'_1.i=T'.i×F.val}

　　T' {T'.s=T'_1.s}

$T' \to \varepsilon$ {T'.s=T'.i}

$F \to (E)$ {F.val=E.val}

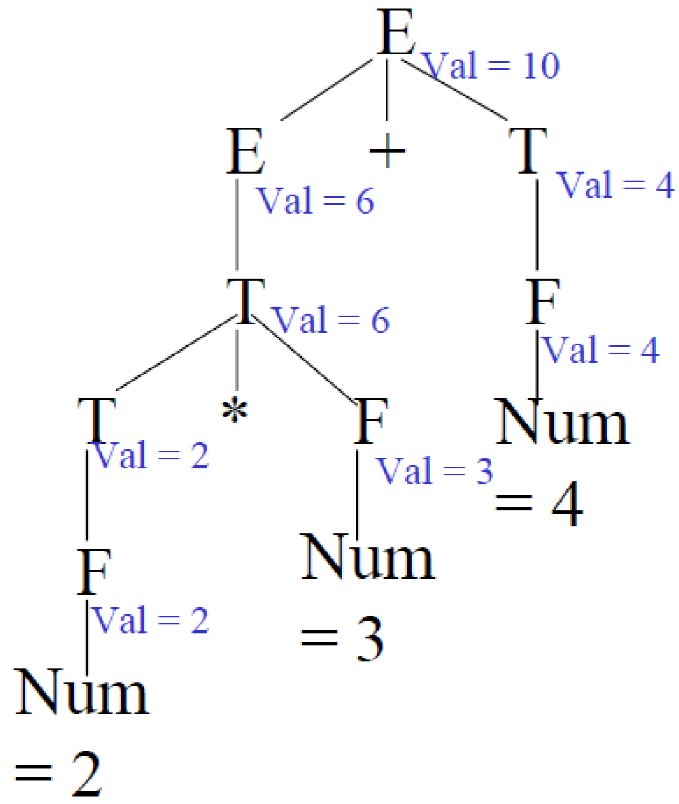
$F \to$ digit {F.val=num.val}



YACC与语法制导翻译

Production	Semantic Actions
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → num	F.val = value(num)
F → (E)	F.val = E.val

Input: 2 * 3 + 4



YACC与语法制导翻译

```
%token NUMBER CR
%%
lines     :  lines line
          |  line
          ;
line      :   expr   CR              {printf("Value = %d",$1); }
          ;
expr      :   expr '+' term          { $$ = $1 + $3; }
          |   term                   { $$ = $1;  /* default - can omit */}
          ;
term      :    term '*' factor       { $$ = $1 * $3; }
          |    factor
          ;
factor    :    '(' expr ')'          { $$ = $2; }
          |    NUMBER
          ;
%%
```

YACC与语法制导翻译

```
%token NUMBER CR
%%
lines     :  lines line
          |  line
          ;

line      :   expr   CR             {System.out.println($1.ival); }
          ;

expr      :   expr '+' term         {$$ = new ParserVal($1.ival + $3.ival); }
          |   term
          ;

term      :    term '*' factor       {$$ = new ParserVal($1.ival * $3.ival);
          |    factor
          ;
factor    :    '(' expr ')'         {$$ = new ParserVal($2.ival); }
          |    NUMBER
          ;
%%
```

YACC与语法制导翻译

- Embedding actions in productions not always guaranteed to work. However, productions can always be rewritten to change embedded actions into end actions

```
A   :  B  {action1}  C  {action2} D {action3};
```

```
A          : new_B  new_C  D  {action3};
new_b    :  B {action1};
new_C    : C  {action 2}  ;
```

AST 构建与语法制导翻译

- AST = Abstract Syntax Tree

```
Parse Tree
==========

        exp
         |
       term
        /|\
   term * factor
     |     /|\
     |    / | \
factor ( exp )
   |      /|\
   3   exp + term
        |     |
      term  factor
        |     |
     factor   2
        |
        4
```

```
Abstract Syntax Tree
====================

          *
         / \
        3   +
           / \
          4   2
```

AST 构建与语法制导翻译

- AST = Abstract Syntax Tree

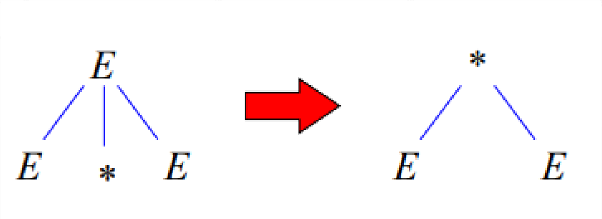
```
Input
=====

{
   x = 0;
   while (x<10) {
      x = x+1;
   }
   y = x*2;
}
```

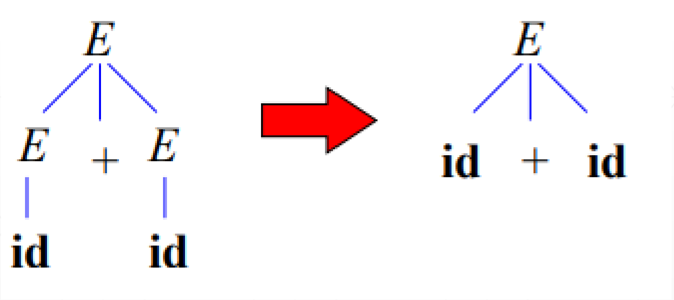
```
                Parse Tree
                ==========

        ____ methodBody __________
       /    /              \      \
      {  declList         stmtList }
            |             /        \
         epsilon    stmtList        stmt___
                   /        \      /  | \  \
            stmtList      stmt    ID  = exp ;
           /        |        \   (y) / |  \
    stmtList      stmt       ...   exp * term
       |         / | | \            |      |
    epsilon     ID = exp ;        term   factor
               (x)    |             |      |
                  INTLITERAL      factor  INT
                     (0)            |     (2)
                                    ID
                                   (x)
```

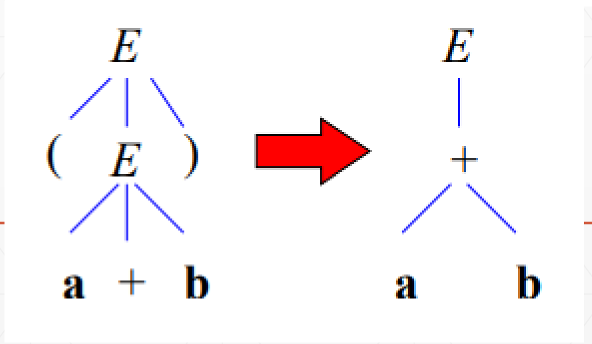
```
            AST
            ===

            methodBody
           /          \
   declList          stmtList
                    /    |    \
             assign   while   assign
             /    \    ...    /     \
           ID     INT        ID      *
          (x)     (0)       (y)     / \
                                   ID  INT
                                  (x)  (2)
```

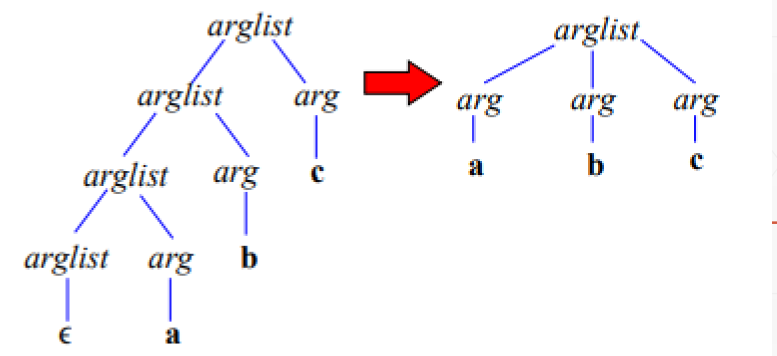
AST 构建与语法制导翻译

- CST(Concrete Syntax Tree)转为 AST(Abstract Syntax Tree)
  - Operators are promoted from leaves to internal nodes
  - Chains of single productions are collapsed
  - Syntactic details like parentheses, semi-colons, and commas are omitted
  - Subtree lists are flattened









AST 构建与语法制导翻译

- LL分析中使用语义动作构建 AST
  - **Example:**

$$S \rightarrow ES'$$
$$S' \rightarrow \varepsilon \mid + S$$
$$E \rightarrow num \mid (S)$$

```
void parse_S() {
  switch (token) {
    case num: case '(':
      parse_E();
      parse_S'();
      return;
    default:
      throw new ParseError();
  }
}
```

⇒

```
Expr parse_S() {
  switch (token) {
    case num: case '(':
      Expr left = parse_E();
      Expr right = parse_S'();
      if (right == null) return left;
      else return new Add(left, right);
    default: throw new ParseError();
  }
}
```

AST 构建与语法制导翻译

- LR分析

- **LR parsing**
  - Need to add code for explicit AST construction

- **AST construction mechanism for LR Parsing**
  - With each symbol X on stack, also store AST sub-tree for X on stack
  - When parser performs reduce operation for $A \to \beta$, create AST subtree for A from AST fragments on stack for β, pop $|\beta|$ subtrees from stack, push subtree for β.

AST 构建与语法制导翻译

- LR parsing

$$\boxed{E \rightarrow \text{num} \mid (E) \mid E+E \mid E*E}$$

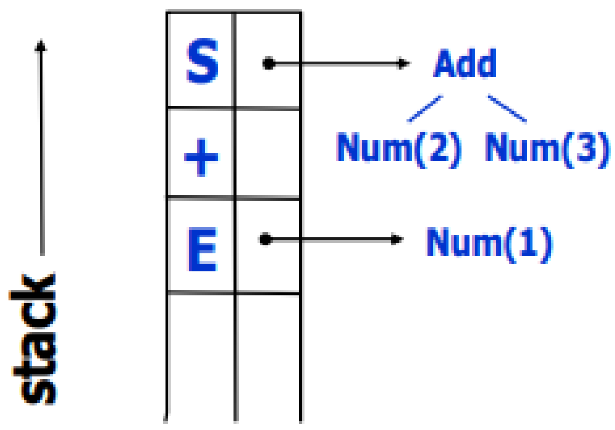
Parser stack stores value of each symbol

		(1+2)*3	
(1		+2)*3	
(E	Num(1)	+2)*3	RESULT=new Num(1)
(E+2		)*3	
(E+E	Num(2)	)*3	RESULT=new Num(2)
(E	Add(,)	)*3	RESULT=new Add(e1,e2)
(E)		*3	
E		*3	RESULT=e

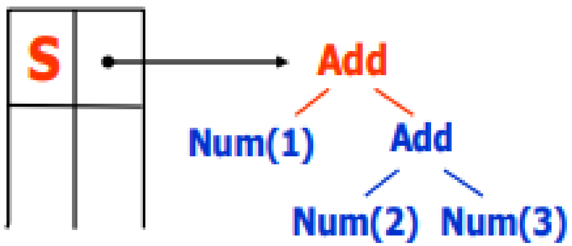
AST 构建与语法制导翻译

- **Example**

$$S \to E+S \mid S$$
$$E \to num \mid (S)$$



Before reduction
$S \to E+S$

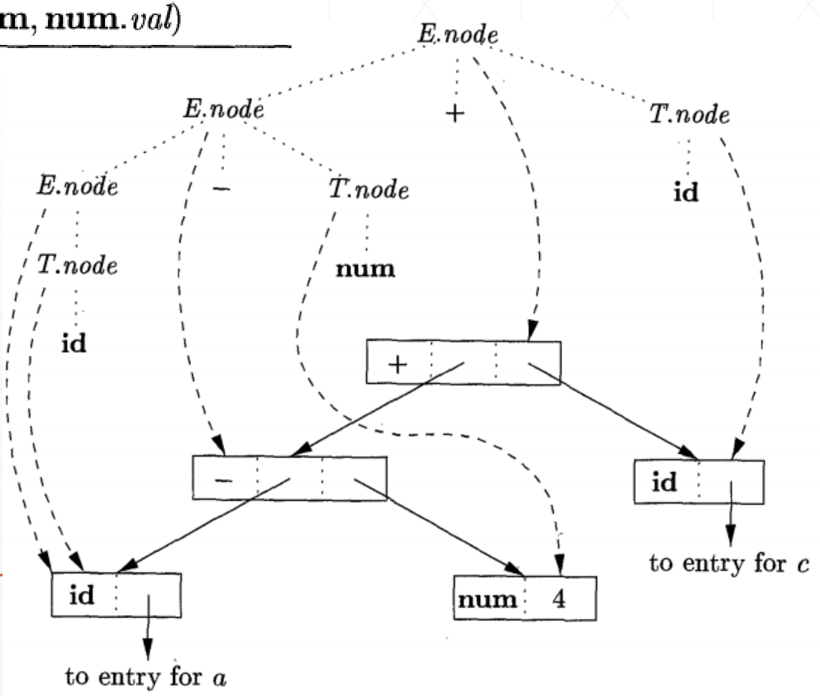


After reduction
$S \to E+S$

AST 构建与语法制导翻译

- Build the AST for Synthesized Attributes

	Production	Semantic Rules
1)	$E \to E_1 + T$	$E.node = \mathbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \mathbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (\ E\ )$	$T.node = E.node$
5)	$T \to \mathbf{id}$	$T.node = \mathbf{new}\ Leaf(\mathbf{id}, \mathbf{id}.entry)$
6)	$T \to \mathbf{num}$	$T.node = \mathbf{new}\ Leaf(\mathbf{num}, \mathbf{num}.val)$



AST 构建与语法制导翻译

- Build the AST for Synthesized Attributes

	Production	Semantic Rules
1)	$E \to T\ E'$	$E.node = E'.syn$ $E'.inh = T.node$
2)	$E' \to +\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3)	$E' \to -\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4)	$E' \to \epsilon$	$E'.syn = E'.inh$
5)	$T \to (\ E\ )$	$T.node = E.node$
6)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
7)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$