信息与软件工程学院

编译技术

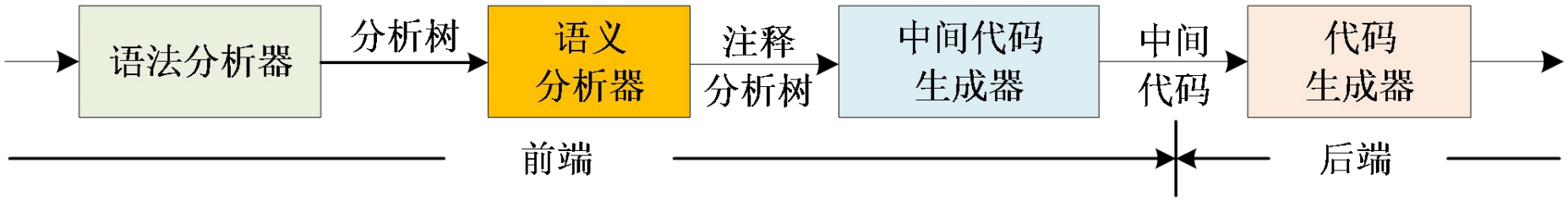
主讲教师：陈安龙

第4章 语法制导翻译(语义分析)

- 编译器结构回顾
- 语义分析的任务
- 语法制导翻译（语义分析方法）
- 语法制导定义
- 语法制导翻译方案

1. 编译器结构回顾

- 为了提高编译程序的可移植性，一般将编译程序划分为前端和后端。
- 前端通常包括词法分析、语法分析、语义分析、中间代码生成、与机器无关的中间代码优化等，它们的实现一般不依赖于具体的目标机器。
- 后端通常包括与机器有关的代码优化、目标代码的生成、相关的错误处理以及符号表的访问等。



语义分析的任务

① 语义检查： 语义分析阶段负责对源程序进行语义检查，确保程序中的语义符合语言规范和语义约束。

② 类型检查： 语义分析阶段进行类型检查，验证表达式、操作符和函数调用等的类型是否匹配，以确保程序在运行时不会出现类型错误。例如，a+b,要求a和b都是算术型（整型或实型），当a和b不是同一种类型时，需进行类型转换。

③ 作用域分析： 语义分析阶段进行作用域分析，确定变量、函数等标识符的作用域和可见性范围。

④ 中间代码生成： 语义分析阶段生成中间表示形式，将源程序转换为更容易进行优化和目标代码生成的形式。

语义分析采用语法制导翻译来实现

什么是语法制导翻译(Syntax-Directed Translation)?

① 是一种在语法分析过程中根据语法规则和附加的语义规则对源代码进行翻译的方法。它使用上下文无关文法（CFG）来引导对语言的翻译，是一种面向文法的翻译技术。

② 通过在文法规则上附加语义规则来描述和计算语义信息。这些语义规则描述了如何根据语法结构计算语义信息，可以包括计算表达式的值、生成中间代码、填充符号表等。

③ 在语法制导翻译中，每个文法符号都可以关联一个或多个属性，属性可以是终结符的值、非终结符的值、中间结果等。语义规则则描述了如何根据文法符号的属性计算其他属性的值。属性的计算可以通过语法分析树的遍历来实现。

语法制导翻译的应用

在编译器的实现中，特别是在语义分析和中间代码生成阶段。通过语法制导翻译，编译器可以根据语法规则和语义规则来进行语义分析、类型检查、符号解析等操作，并生成相应的中间代码。

语法制导翻译过程

① 语法分析：对输入的源代码进行语法分析，构建语法分析树或语法分析表。

② 属性计算：在语法分析的过程中，根据语法规则和附加的语义规则，计算每个文法符号的属性值。这可以通过自底向上或自顶向下的遍历来实现。

③ 语义动作执行：在属性计算的过程中，执行相应的语义动作。这些动作可以包括生成中间代码、填充符号表、计算表达式的值等。

④ 中间代码生成：根据语义动作的执行结果，生成目标代码或中间代码。

语法制导翻译的类型

① 语法制导定义(Syntax-Directed Definitions，SDD)：语法制导定义是一种扩展了上下文无关文法（CFG）的定义，它将每个产生式和一组语义规则相关联，用于计算产生式中每个文法符号的属性值。语法制导定义描述了如何根据语法结构计算语义信息，并指导编译器的翻译过程。

② 语法制导翻译方案(Syntax-Directed Translation Scheme，SDT)：语法制导翻译方案是对语法制导定义的具体实现方式，它详细规定了如何根据语法制导定义来计算各个属性的值，并完成从源代码到目标代码的转换。语法制导翻译方案包括属性计算和翻译动作的执行，可以用于生成中间代码、填充符号表、进行错误检测等操作。

语法制导定义(SDD)

- 在语法制导定义中，每个文法符号（终结符和非终结符）都关联着一组属性，这些属性可以是任何类型的值，如字符串、数值、类型或表引用等，用于捕捉文法结构的语义信息。
- 每个产生式都配备了一组语义规则，用于计算这些属性的值。这些规则定义了如何根据产生式右部的文法符号的属性值来计算左部文法符号的属性值或产生式右部某个文法符号的继承属性值。

属性文法（Attribute Grammar）

当给描述文法的产生式的每个文法符号都关联有一个或多个属性，这些属性可以用于传递和计算语法结构中的语义信息。这时带有属性的文法产生式就称为属性文法。

属性文法的示例

简单的算术表达式文法，包含加法和乘法运算；

E→E+T

E→T

T→T * F

T→F

F→(E)

F→id

文法语义是计算表达式的值；给每个文法符号附上属性val

E.val：表示表达式的值；

T.val：表示项的值；

F.val：表示因素的值；

id.val：表示标识符的值。

属性文法产生式如下，如果有相同终结符，用下标区分：

$E \rightarrow E_1+T$ {E.val=E_1.val+T.val}

E→T {E.val=T.val}

$T \rightarrow T_1 * F$ {T.val=T_1.val* F.val}

T→F {T.val=F.val}

F→(E) {F.val=E.val}

F→id {F.val=id.val}

属性文法的属性分类：

- 综合属性
- 继承属性

综合属性(Synthesized Attribute)

- 在编译技术中，综合属性的值是通过从子节点的属性值计算得出的;通常用于自下而上地传递信息；在语法树中，节点的综合属性由其子节点或自身的某些属性值确定；综合属性的计算是从叶子节点开始，根据语义规则计算综合属性的值，并将其传递给父节点。通过逐步计算综合属性，最终可以得到根节点的综合属性值。
- 在语法制导翻译中，每个产生式都可以附加一个或多个语义规则，用于计算产生式左侧文法符号的属性值。这些语义规则可以根据子节点的属性值计算出父节点的综合属性值。
- 仅仅使用综合属性的属性文法称S-属性文法。

综合属性的计算过程示例

简单的算术表达式文法，包含加法和乘法运算，有综合属性 val 用于计算表达式的值；假设有表达式2*(3+4)，按照以下步骤计算val 的值：

产生式示例：

$E \rightarrow E_1+T$ {E.val=E_1.val+T.val}

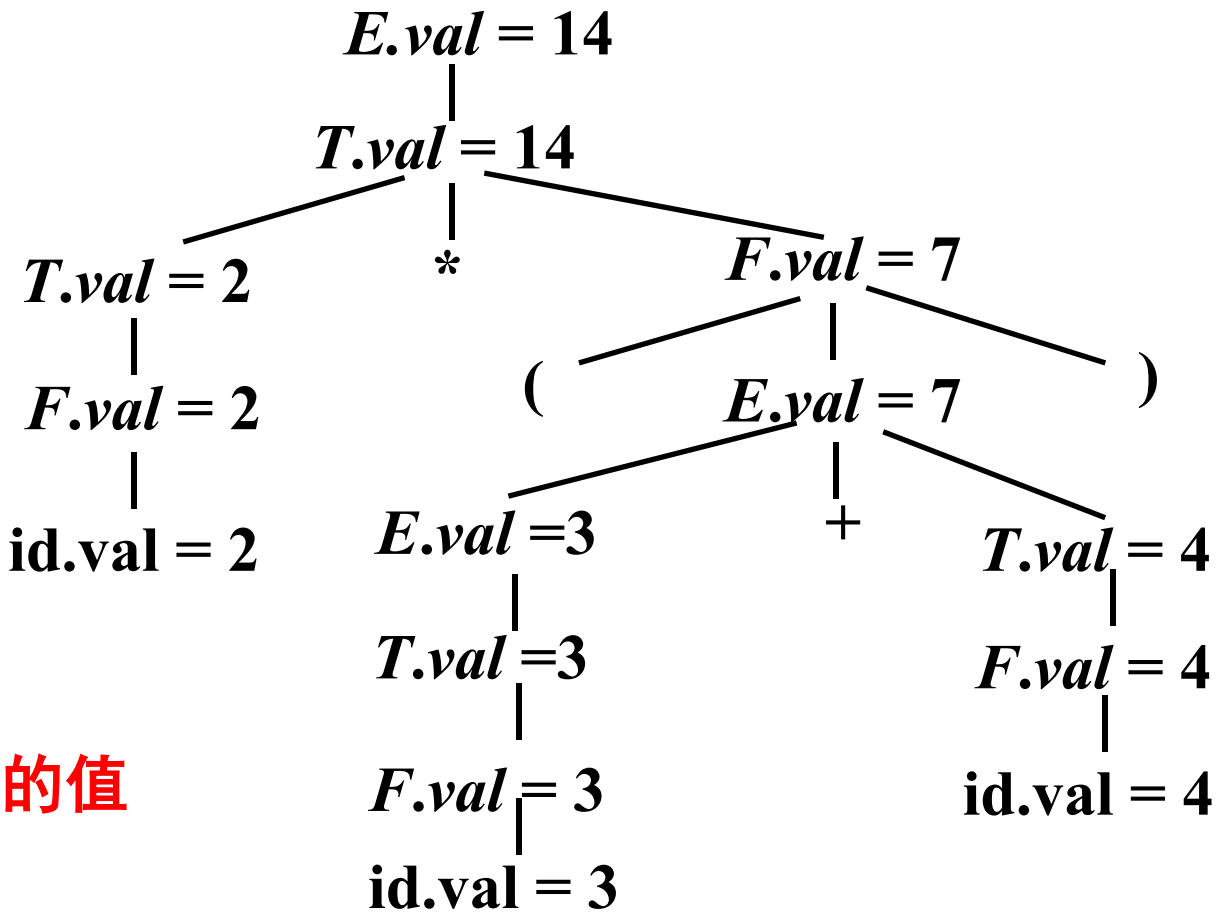
$E \rightarrow T$ {E.val=T.val}

$T \rightarrow T_1*\ F$ {T.val=T_1.val* F.val}

$T \rightarrow F$ {T.val=F.val}

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

$F \rightarrow id$ {F.val=id.val}



类似于按照二叉树的后序遍历计算val的值

继承属性(Inherited Attribute)

- 在编译技术中，继承属性的值是从父节点传递给子节点的。继承属性通常用于自顶向下地传递信息。在语法树中，一个节点的继承属性由其父节点或自身的某些属性值确定。
- 在语法制导翻译中，每个产生式都可以附加一个或多个语义规则，用于计算产生式右侧文法符号的属性值。这些语义规则可以根据父节点的属性值计算出子节点的继承属性值。
- 继承属性的计算通常是自顶向下的过程。从根节点开始，根据语义规则计算继承属性的值，并将其传递给子节点。通过逐步计算继承属性，最终可以得到叶子节点的继承属性值。

继承属性的计算示例---类型检查

假设下面文法是确定标识符的数据类型的属性文法，inh为继承属性，type为综合属性。

D→TL　　{ *L.inh = T.type* }

T→int　　{*T. type = integer* }

T→float　　{*T. type = float* }

L→L_1, id {L_1.*inh = L.inh*; *addType*(id.*entry*, *L.inh*) }

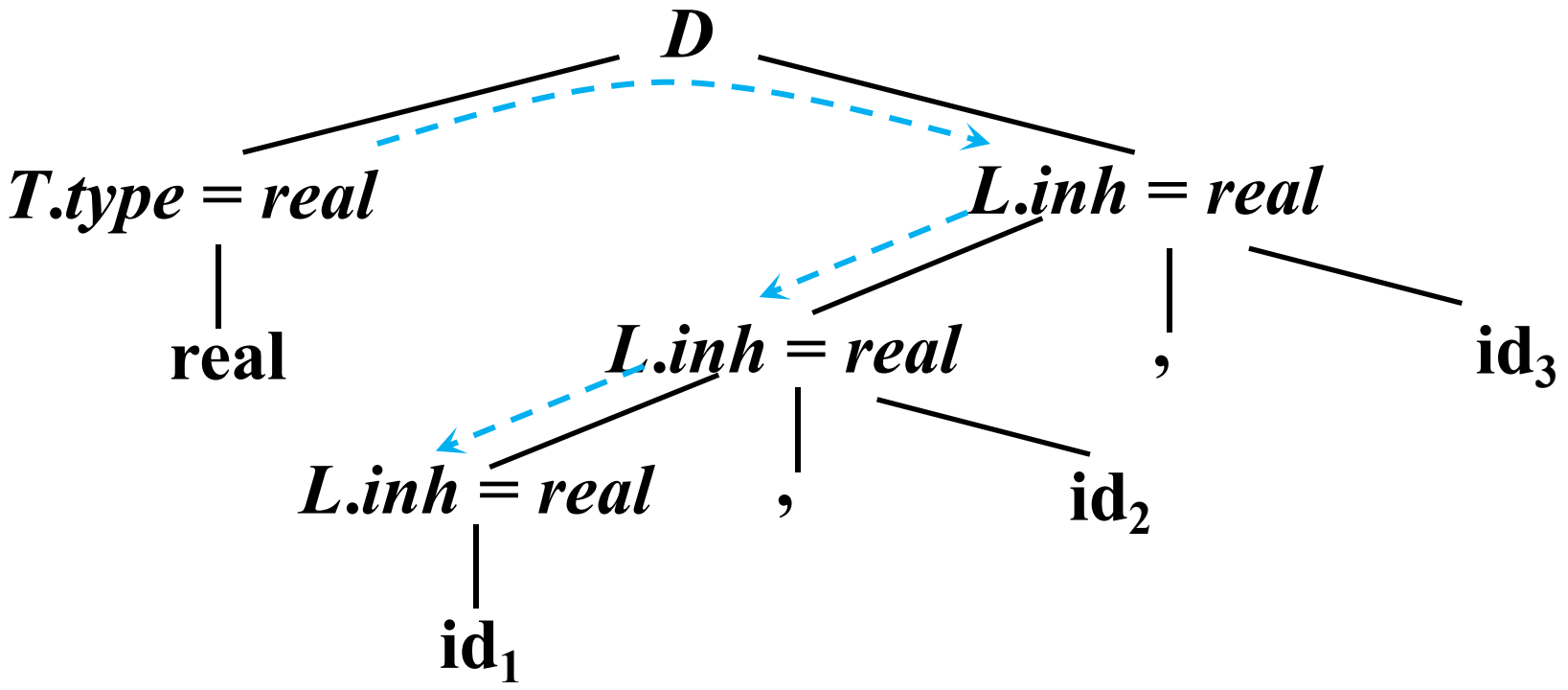
L→ id　　{*addType*(id.*entry*, *L.inh*)}

过程addtype是把每个标志符的类型信息登录在符号表中相关项中。

继承属性的计算示例

由父节点、兄弟结点的属性来定义的属性。

例：real id_1, id_2, id_3的注释分析树



S属性和L属性

- S属性：仅涉及综合属性的语法制导定义
- L属性：一个语法制导定义是L属性定义，如果

  与每个产生式$A \rightarrow X_1X_2 \ldots X_n$相应的每条语义规则计算的属性都是A的综合属性，或是X_j（$1 \leq j \leq n$）的继承属性，而该继承属性仅依赖于：

  ① A的继承属性；

  ② 产生式中X_j左边的符号X_1、X_2、…、X_{j-1}的综合属性或继承属性；
- 每一个S属性定义都是L属性定义

L属性的语法制导定义示例

D→TL　{ *L.inh = T.type* }

T→int　{*T. type = integer* }

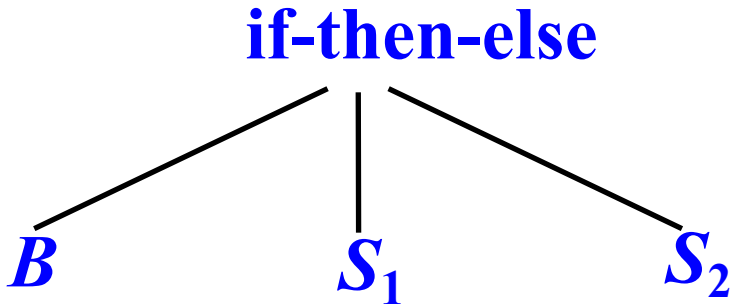
T→float　{*T. type = float* }

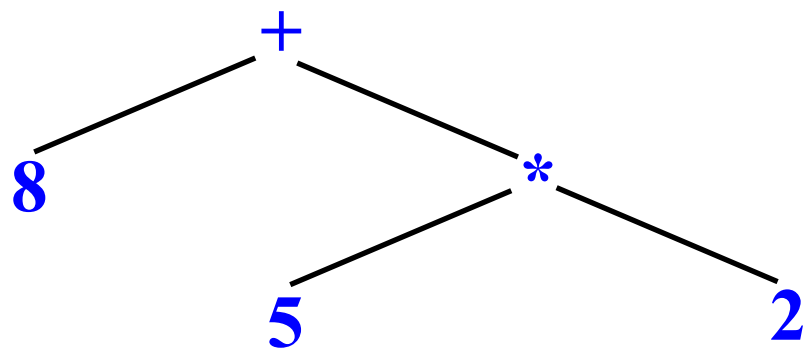
L→L_1, id　{L_1.*inh = L.inh*; *addType*(id.*entry*, *L.inh*) }

L→ id　{*addType*(id.*entry*, *L.inh*)}

语法树（抽象语法树）

表示程序层次结构的树，它把分析树中对语义无关紧要的成分去掉，是分析树的抽象形式。 产生式s→if B then S1 else S2的语法树





在语法树中，运算符号和关键字都不在叶结点，而是在内部结点中出现。

建立表达式的语法树的构造函数

① *node* (*op*, *left*, *right*)：运算符结点；

② *leaf*(*id*, *entry*)：标识符叶子结点；

③ *leaf*(*num*, *val*)：数叶子结点。

语法制导翻译创建语法树的示例

$E \rightarrow E_1 + T$ { E.node := new Node('+', E_1.node, T.node) }

$E \rightarrow E_1$-F { E.node := new Node('-', E_1.node, T.node) }

E→T { E.node := T.node }

T→(E) { T.node := E.node }

T→id { T.node := new Leaf (id, id.entry) } ------id为变量终结符

T→num { T.node := new Leaf (num, num.val) } ------num为常量终结符

a-4+c的抽象语法树构造

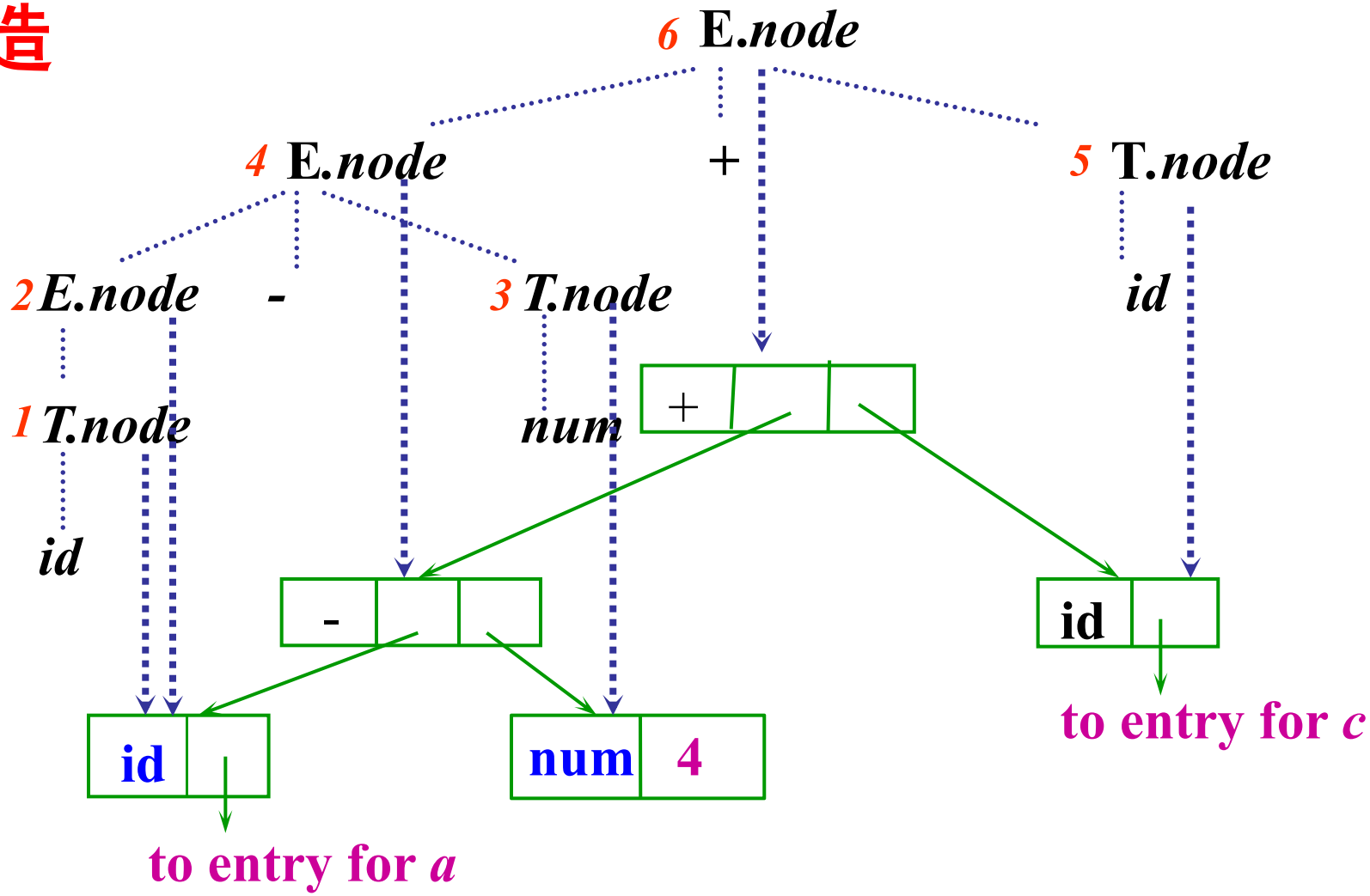
(1) p_1 := new *Leaf*(id, *entrya*);

(2) p_2 := new *Leaf*(num, 4);

(3) p_3 := new *Node*('-', p_1, p_2);

(4) p_4 := new *Leaf*(id, *entryc*);

(5) p_5 := *new* *Node*('+', p_3,p_4);



虚线部分是分析树，一般不用实际创建；实线是抽象语法树，红色的数字表示分析过程的顺序.

语法制导的翻译方案 (SDT)

- 语法制导翻译方案是对语法制导定义的具体实现方式，它详细规定了如何根据语法制导定义来计算各个属性的值，并完成从源代码到目标代码的转换。语法制导翻译方案包括属性计算和翻译动作的执行，可以用于生成中间代码、填充符号表、进行错误检测等操作。
  ① 把SDD的语义规则改写为计算属性值的程序片段用花括号｛ ｝括起来，插入到产生式右部的任何合适的位置上。
  ② 这是一种语法分析和语义动作交错的表示法，它表达在按深度优先遍历分析树的过程中何时执行语义动作
- 原来的不含语义动作的文法称作是基础文法。

翻译方案的设计

1. 只需要综合属性的情况：为每一个语义规则建立一个包含赋值的动作，并把这个动作放在相应的产生式右边的末尾。

- 例如：计算表达式值的左递归文法的语法制导翻译方案，只有综合属性val

$E \rightarrow E_1+T$	{ E.val = E_1.val + T.val }
$E \rightarrow E_1-T$	{ E.val = E_1.val - T.val }
$E \rightarrow T$	{ E.val = T.val }
$T \rightarrow (E)$	{ T.val = E.val }
$T \rightarrow num$	{ T.val = num.val }

翻译方案的设计(续)

2. 既有综合属性又有继承属性：

① 产生式右边的符号的继承属性必须在这个符号以前的动作中计算出来。

② 一个动作不能引用这个动作右边符号的综合属性。

③ 产生式左边非终结符号的综合属性只有在它所引用的所有属性都计算出来以后才能计算。计算这种属性的动作通常可放在产生式右端的末尾。

翻译方案的设计(续)

既有综合属性又有继承属性的文法方案示例

下列左边是变量声明及变量类型计算的文法，文法符号的属性为：

T.type：表示类型，为综合属性；*L.inh* ：为继承性属性；

id.*entry*： 指向符号表的指针；*addType*(id.*entry*, *L.inh*)：在符号表给id的添加类型。

D→T L	D→T { *L.inh* = *T.type* } L	规则①
T→int	T→int {*T. type* = *integer* }	规则②
T→float	T→float {*T. type* = *float* }	规则②
L→ L_1, id	L→ {L_1.*inh* = *L.inh*} L_1, id { *addType*(id.*entry*, *L.inh*) }	规则①③
L→ id	*L*→ id {*addType*(id.*entry*, *L.inh*)}	规则③

左递归文法的翻译方案

- 左递归翻译方案

  $A \rightarrow A_1 Y \quad \{ A.a = h(A_1.a, Y.y) \}$

  $A \rightarrow X \quad \{ A.a = f(X.x) \}$

  假设每个文法符号有一个综合属性，用相应的小写字母表示，h 和 f 是任意函数

- 消除左递归之后，文法转换成

  $A \rightarrow XR$

  $R \rightarrow YR \mid \varepsilon$

- 消除左递归的翻译方案

  $A \rightarrow X \quad \{ R.i = f(X.x) \}$

  $R \quad \{ A.a = R.s \}$

  $R \rightarrow Y \quad \{ R_1.i = h(R.i, Y.y) \}$

  $R_1 \quad \{ R.s = R_1.s \}$

  $R \rightarrow \varepsilon \quad \{ R.s = R.i \}$

  经过转换的翻译方案中

  使用了R的继承属性 i 和综合属性 s

左递归文法翻译方案示例

例: 把带左递归的文法的翻译方案转换成不带左递归的文法的翻译方案。

带左递归的文法的翻译方案

$E \rightarrow E_1+T$ { E.val = E_1.val + T.val }

$E \rightarrow E_1-T$ { E.val = E_1.val - T.val }

$E \rightarrow T$ { E.val = T.val }

$T \rightarrow (E)$ { T.val = E.val }

$T \rightarrow id$ { T.val = id.val }

val为综合属性，

E.val： 表示表达式的值；

T.val： 表示项的值；

id.val： 表示标识符的值

不带有左递归文法的翻译方案

$E \rightarrow T$ { R.i = T.val }

R { E.val = R.s }

$R \rightarrow +T$ { R_1.i = R.i +T.val }

R_1 { R.s = R_1.s }

$R \rightarrow -T$ { R_1.i = R.i - T.val }

R_1 { R.s = R_1.s }

$R \rightarrow \varepsilon$ { R.s = R.i }

$T \rightarrow (E)$ { T.val = E.val }

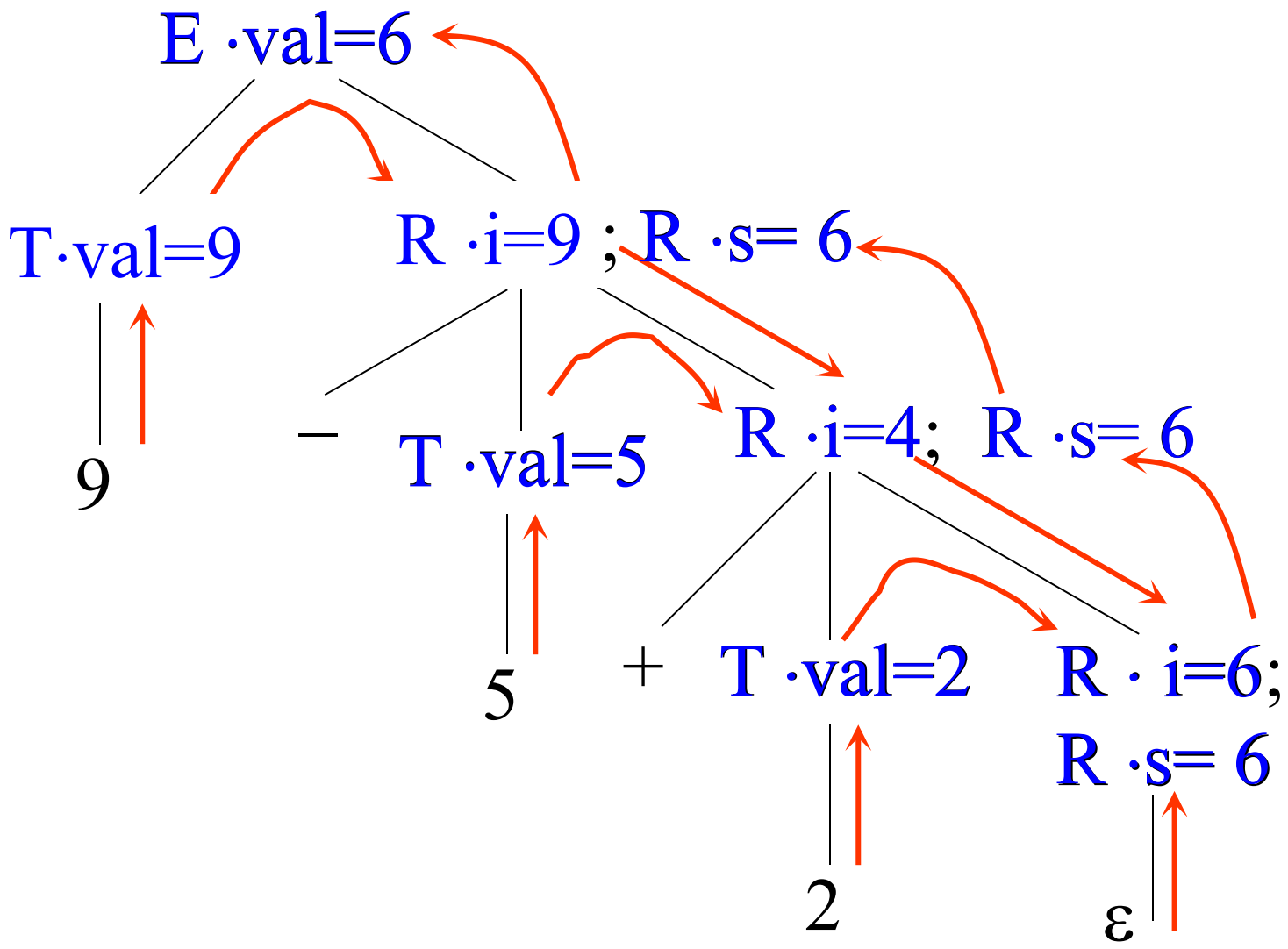
$T \rightarrow id$ { T.val = id.val }

R.i为继承属性

左递归转化非左递归后的文法翻译方案示例

表达式9-5+2的计算

E→T { R.i = T.val }
R { E.val = R.s }
R→ +T { R_1.i = R.i +T.val }
R_1 { R.s = R_1.s }
R→ -T { R_1.i = R.i - T.val }
R_1 { R.s = R_1.s }
R→ε { R.s = R.i }
T→ (E) { T.val = E.val }
T→id { T.val = id.val }



语法制导的预测翻译程序

算法：构造语法制导的预测翻译程序

输入：基础文法适合于预测分析的语法制导翻译方案

输出：语法制导翻译程序

方法：（修改预测分析程序的构造技术）

(1) 为每个非终结符号A建立一个函数(可以是递归函数)

① A的每一个继承属性对应函数的一个形参

② A的综合属性作为函数的返回值

③ A产生式中的每个文法符号的每个属性都对应一个局部变量

(2) A的函数的代码由多个分支组成

(3) 与每个产生式相关的程序代码

- 按照从左到右的顺序考虑产生式右部的记号、非终结符号和语义动作
- 对带有综合属性x的记号X
  - 把属性x的值保存于为X.x声明的变量中
  - 产生一个匹配记号X的调用
  - 推进扫描指针
- 对非终结符号B
  - 产生一个函数调用语句c=B(b_1,b_2,…,b_k)
  - b_i(i=1,2,…,k)是对应于B的继承属性的变量
  - c是对应于B的综合属性的变量
- 对每一个语义动作
  - 把动作代码复制到分析程序中
  - 用代表属性的变量代替翻译方案中引用的属性

例：为简单表达式求值的翻译方案构造翻译程序

E→T { R.i = T.val }

R { E.val = R.s }

R→ +T { R_1.i = R.i +T.val }

R_1 { R.s = R_1.s }

R→ -T { R_1.i = R.i - T.val }

R_1 { R.s = R_1.s }

R→ε { R.s = R.i }

T→ (E) { T.val = E.val }

T→id { T.val = id.val }

- 为每个非终结符号构造一个函数

  int E(void)

  int T(void)

  int R(int in)

  int lookup(id) //在符号表中查找id的值

语法制导的翻译方案实现

E→ T {R.i=T.val}

R {E.val=R.s}

```
int E(void)  {
    int eval, tval, ri, rs;
    tval=T();
    ri=tval;
    rs=R(ri);
    eval=rs;
    return eval;
}
```

语法制导的翻译方案实现

R→ +T { R_1.i = R.i +T.val }

R_1 { R.s = R_1.s }

R→ -T { R_1.i = R.i - T.val }

R_1 { R.s = R_1.s }

R→ε { R.s = R.i }

```
int R(int in)  {
  int tval, i1, s1, s;
  if (lookahead== '+') {
     match( '+' );
     tval=T();  i1=in+tval;  s1=R(i1);
     s=s1;
  } elseif (lookahead== '-') {
     match( '-' );
     tval=T();  i1=in-tval;  s1=R(i1);
     s=s1;
  } ;
  else  s=in; // 产生式R→ε
  return s
}
```

语法制导的翻译方案实现

T→ (E)　{ T.val = E.val }

T→id　　{ T.val = id.val }

```
int T( )  {
   int tval, idval;
   if (lookahead== '(') {
       match( '+' );
       tval=E();
      } else {
      idval=lookup(id) ;
      tval=idval;
   } ;
   return tval ;
}
```

本章作业

习题 1: 给定文法：

E —> E +T

E —>T

T —>T * F

T —> F

F —> integer

设计一个语法制导翻译方案，计算表达式的值。

习题2: 给定文法：

stmt —> id = E

E —> id

E —> int

设计一个语法制导翻译方案，检查赋值语句中的类型匹配。