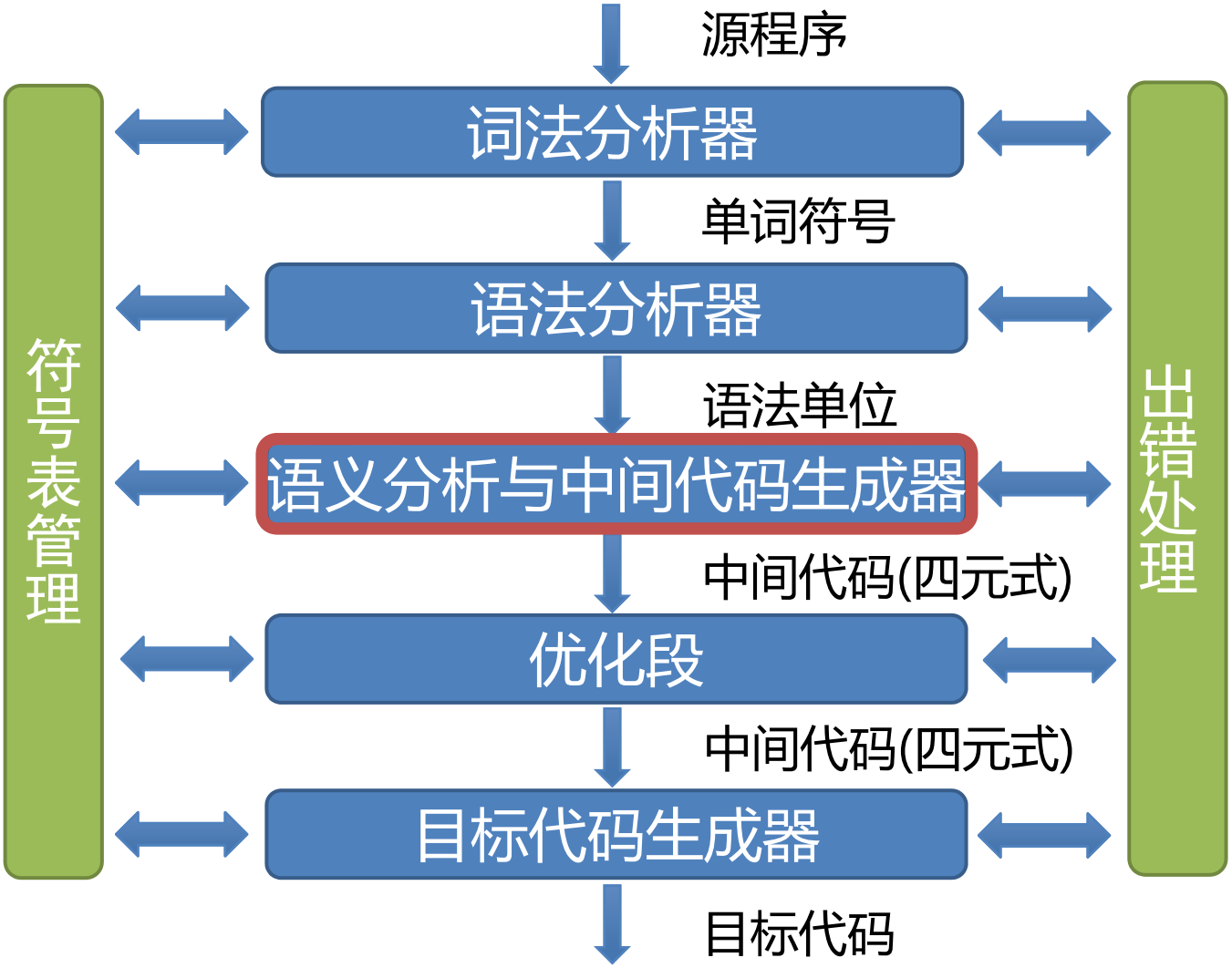
编译原理

属性文法

编译原理

属性文法概念

编译程序总框


源程序
词法分析器
单词符号
语法分析器
语法单位
语义分析与中间代码生成器
中间代码(四元式)
优化段
中间代码(四元式)
目标代码生成器
目标代码
符号表管理
出错处理

属性文法

Donald Ervin Knuth

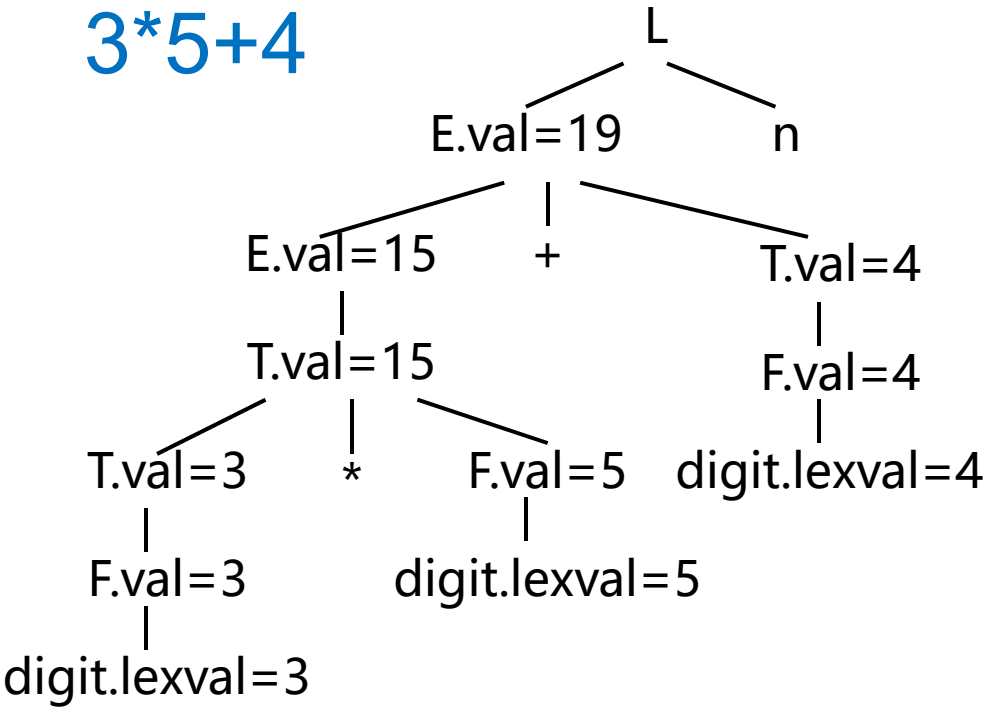
产生式	语 义 规 则
L→En	print(E.val)
$E \rightarrow E_1+T$	E.val := E_1.val+T.val
E→T	E.val :=T.val
$T \rightarrow T_1*F$	T.val :=T_1.val* F.val
T→F	T.val :=F.val
F→ (E)	F.val :=E.val
F→digit	F.val :=digit.lexval

- 属性文法,也称属性翻译文法
- Knuth在1968年提出
- 以上下文无关文法为基础
  - 为每个文法符号（终结符或非终结符）配备若干相关的“值”（称为属性），代表与文法符号相关信息，如类型、值、代码序列、符号表内容等
  - 对于文法的每个产生式都配备了一组属性的语义规则，对属性进行计算和传递

综合属性

- 自下而上传递信息
- 语法规则：根据右部候选式中的符号的属性计算左部被定义符号的综合属性
- 语法树：根据子结点的属性和父结点自身的属性计算父节点的综合属性

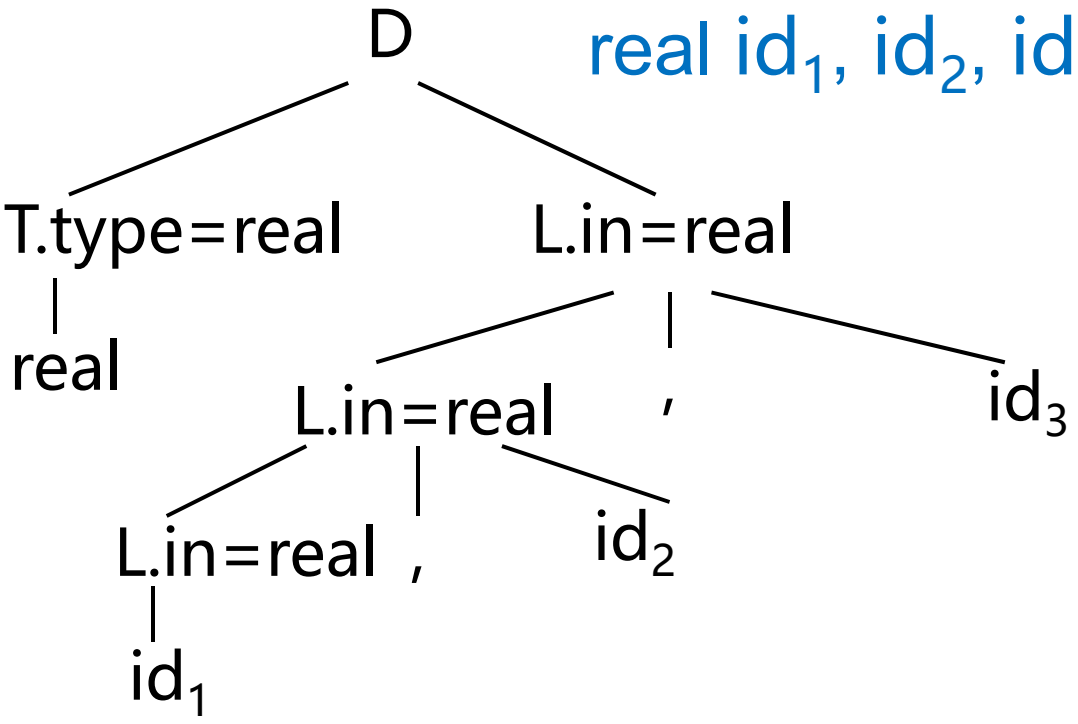
3*5+4



产生式	语 义 规 则
L→En	print(E.val)
$E→E_1+T$	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→ (E)	F.val :=E.val
F→digit	F.val :=digit.lexval

继承属性

- 自上而下传递信息
- 语法规则：根据右部候选式中的符号的属性和左部被定义符号的属性计算右部候选式中的符号的继承属性
- 语法树：根据父结点和兄弟节点的属性计算子结点的继承属性



产生式	语义规则
D→TL	L.in := T.type
T→int	T.type := integer
T→real	T.type := real
$L \rightarrow L_1$, id	L_1.in :=L.in
	addtype(id.entry, L.in)
L→id	addtype(id.entry, L.in)

编译原理

属性依赖

属性依赖

- 对应于每个产生式A→α都有一套与之相关联的语义规则，每条规则的形式为(f是一个函数)：

  $b:=f(c_1,c_2,\ldots,c_k)$

- 属性b依赖于属性$c_1,c_2,\ldots,c_k$
  - b是A的一个综合属性并且$c_1,c_2,\ldots,c_k$是产生式右边文法符号的属性，或者
  - b是产生式右边某个文法符号的一个继承属性并且$c_1,c_2,\ldots,c_k$是A或产生式右边任何文法符号的属性

产生式	语 义 规 则
L→En	print(E.val)
$E→E_1+T$	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→ (E)	F.val :=E.val
F→digit	F.val :=digit.lexval

产 生 式	语 义 规 则
D→TL	L.in := T.type
T→int	T.type := integer
T→real	T.type := real
$L→L_1$, id	L_1.in :=L.in
	addtype(id.entry, L.in)
L→id	addtype(id.entry, L.in)

属性依赖

- 终结符只有综合属性，由词法分析器提供
  - F→digit
  - digit.lexval
- 非终结符既可有综合属性也可有继承属性，文法开始符号的所有继承属性作为属性计算前的初始值

产生式	语 义 规 则
L→En	print(E.val)
$E \rightarrow E_1+T$	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→digit	F.val :=digit.lexval

语义规则

- 对出现在产生式右边的继承属性和出现在产生式左边的综合属性都必须提供一个计算规则。属性计算规则中只能使用相应产生式中的文法符号的属性。
- 出现在产生式左边的继承属性和出现在产生式右边的综合属性不由所给的产生式的属性计算规则进行计算，由其它产生式的属性规则计算或者由属性计算器的参数提供。

产生式	语 义 规 则
L→En	print(E.val)
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→ (E)	F.val :=E.val
F→digit	F.val :=digit.lexval

产 生 式	语 义 规 则
D→TL	L.in := T.type
T→int	T.type := integer
T→real	T.type := real
L→L_1, id	L_1.in :=L.in
	addtype(id.entry, L.in)
L→id	addtype(id.entry, L.in)

语义规则

- 语义规则所描述的工作可以包括属性计算、静态语义检查、符号表操作、代码生成等。

产生式	语 义 规 则
$L \rightarrow En$	print(E.val)
$E \rightarrow E_1+T$	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
$F \rightarrow digit$	F.val :=digit.lexval

产 生 式	语 义 规 则
$D \rightarrow TL$	L.in := T.type
$T \rightarrow int$	T.type := integer
$T \rightarrow real$	T.type := real
$L \rightarrow L_1$, id	L_1.in :=L.in
	addtype(id.entry, L.in)
$L \rightarrow id$	addtype(id.entry, L.in)

测试：语义规则

- 考虑非终结符A，B和C，其中，A有一个继承属性a和一个综合属性b，B有综合属性c，C有继承属性d。产生式A→BC不可能有规则

  A. C.d:=B.c+1

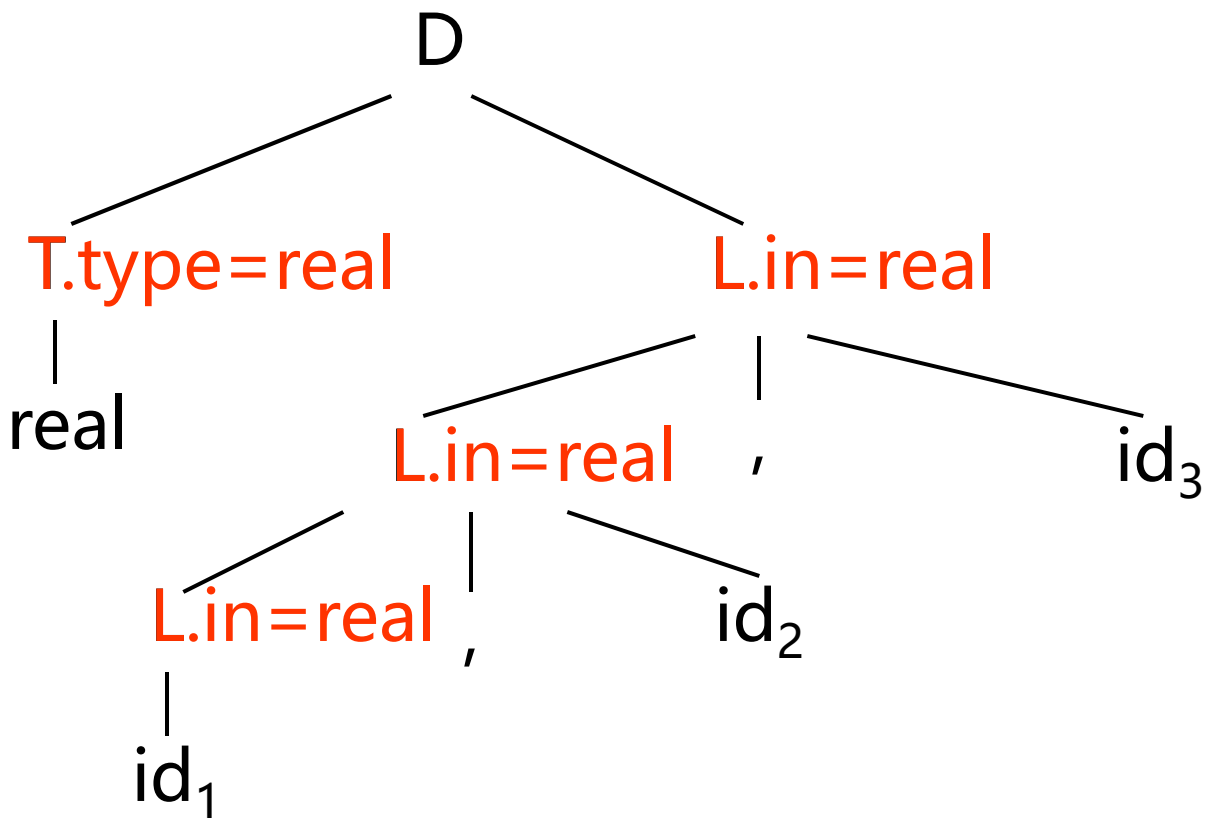
  B. A.b:=B.c+C.d

  C. B.c := A.a

编译原理

带注释的语法树

带注释的语法树



带注释的语法树

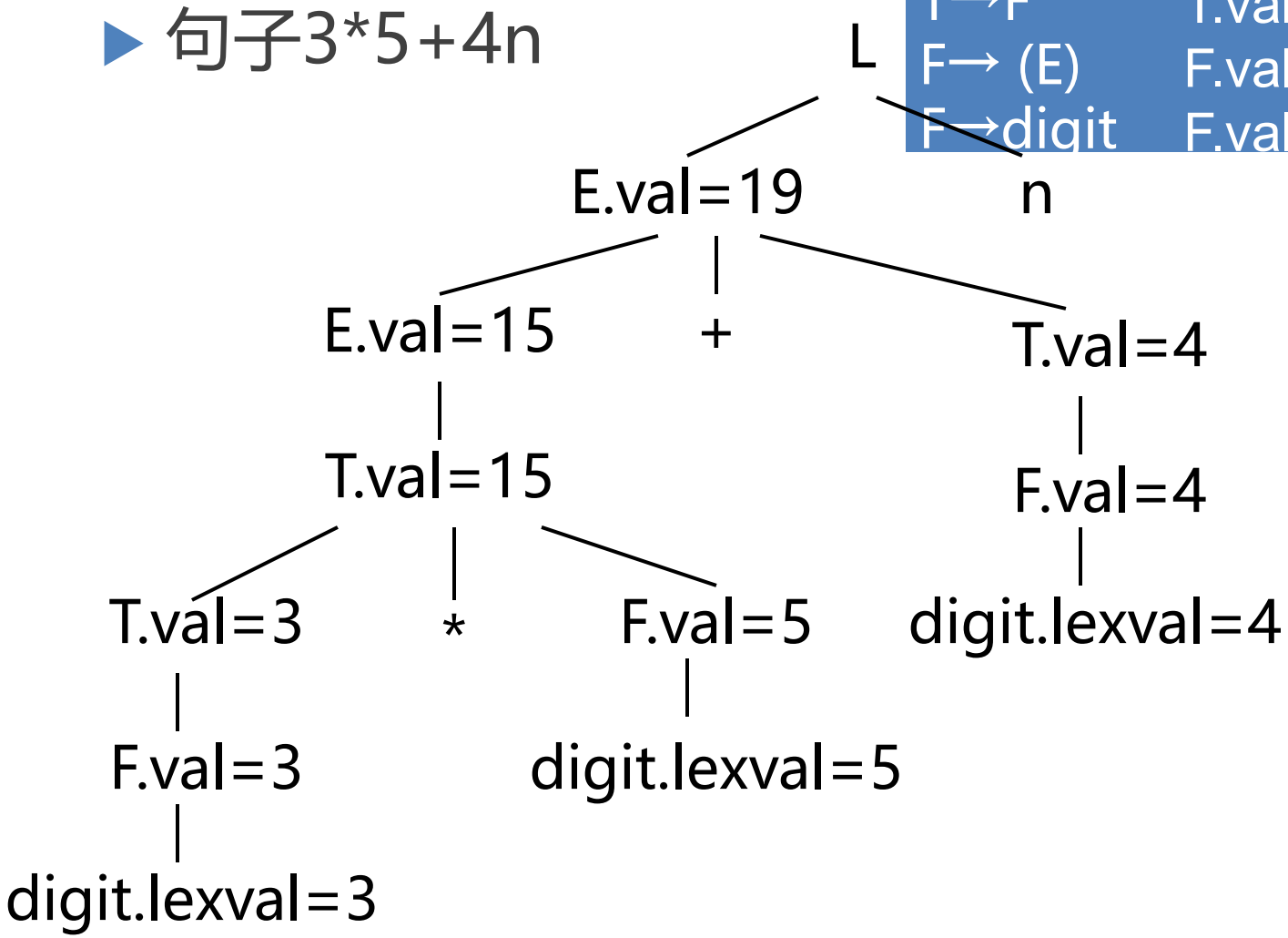
- 在语法树中，一个结点的综合属性的值由其子结点和它本身的属性值确定
- 使用自底向上的方法在每一个结点处使用语义规则计算综合属性的值
- 仅使用综合属性的属性文法称S - 属性文法

产生式	语 义 规 则
L→En	print(E.val)
$E \rightarrow E_1+T$	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→digit	F.val :=digit.lexval

带注释的语法树

产生式	语 义 规 则
$L \rightarrow En$	print(E.val)
$E \rightarrow E_1+T$	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
$F \rightarrow digit$	F.val :=digit.lexval

- 句子3*5+4n



带注释的语法树

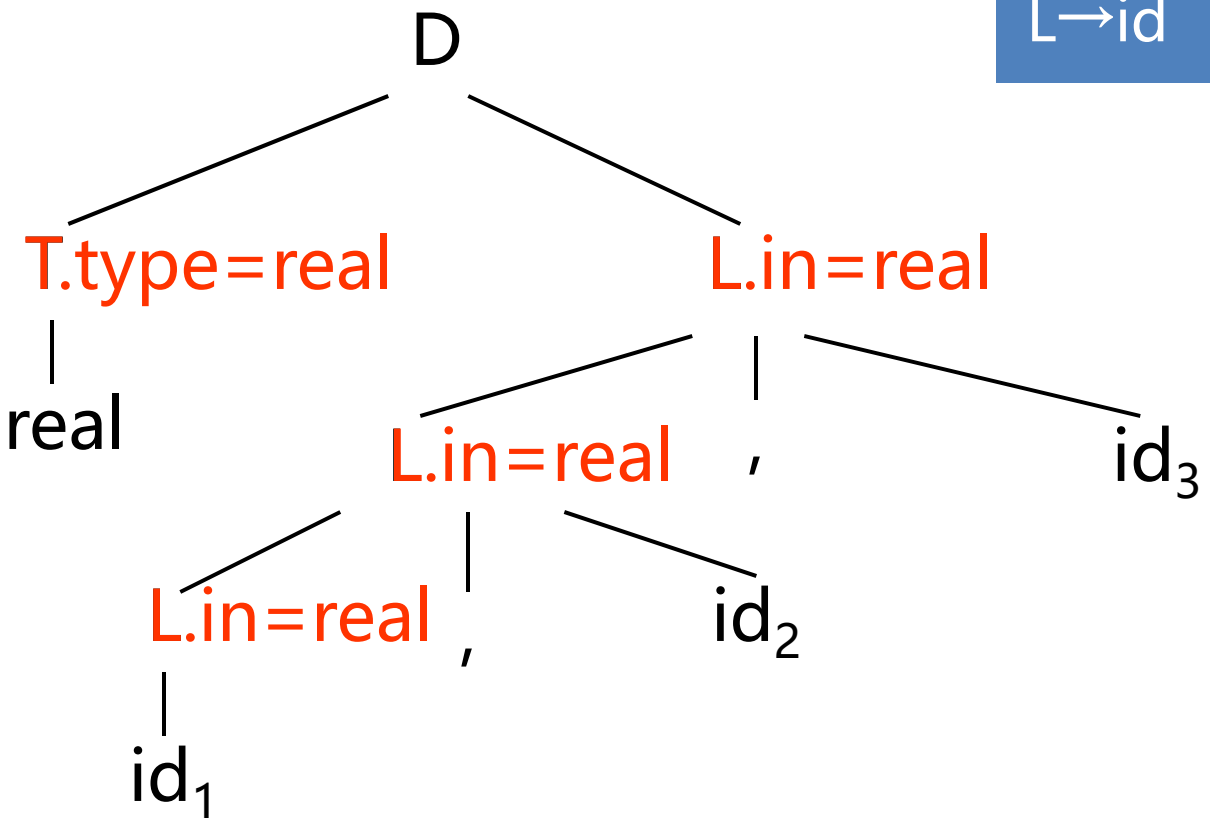
- 在语法树中，一个结点的继承属性由其父结点、其兄弟结点和其本身的某些属性确定
- 用继承属性来表示程序设计语言结构中的上下文依赖关系很方便

产生式	语义规则
D→TL	L.in := T.type
T→int	T.type := integer
T→real	T.type := real
$L \rightarrow L_1$, id	L_1.in :=L.in addtype(id.entry, L.in)
L→id	addtype(id.entry, L.in)

带注释的语法树

- 句子real id_1, id_2, id_3

产生式	语义规则
D→TL	L.in := T.type
T→int	T.type := integer
T→real	T.type := real
$L→L_1$, id	L_1.in :=L.in
	addtype(id.entry, L.in
L→id	addtype(id.entry, L.in)



name	type
...	...
id_3	real
id_2	real
id_1	real

符号表

小结

- 属性文法
- 综合属性
- 继承属性
- 带注释的语法树

编译原理

属性计算

基于属性文法的处理方法

- 语义规则的计算
  - 产生代码
  - 在符号表中存放信息
  - 给出错误信息
  - 执行任何其它动作
- 对输入串的翻译就是根据语义规则进行计算

基于属性文法的处理方法

- 由源程序的语法结构所驱动的处理办法就是语法制导翻译法

  输入串⟶语法树 ⟶ 按照语义规则计算属性

基于属性文法的处理方法

- 依赖图
- 树遍历
- 一遍扫描

编译原理

依赖图

依赖图

- 在一棵语法树中的结点的继承属性和综合属性之间的相互依赖关系可以由依赖图(有向图)来描述。
- 为每一个包含过程调用的语义规则引入一个虚综合属性b，这样把每一个语义规则都写成

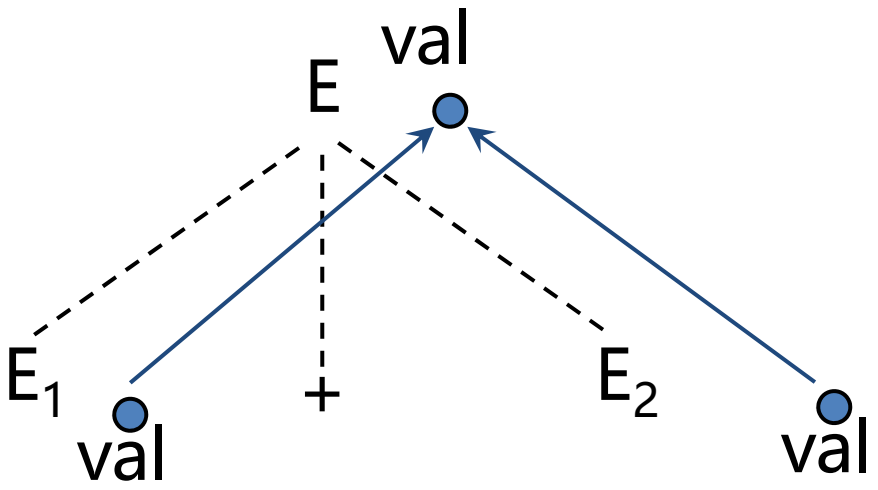
$$b:=f(c_1,c_2,\ldots,c_k)$$

的形式。

依赖图

- 依赖图中为每一个属性设置一个结点，如果属性b依赖于属性c，则从属性c的结点有一条有向边连到属性b的结点。

$E \rightarrow E_1 + E_2$　　$E.val := E_1.val + E_2.val$



依赖图的构建算法

for 语法树中每一结点n do

for 结点n的文法符号的每一个属性a do

为a在依赖图中建立一个结点；

for 语法树中每一个结点n do

for 结点n所用产生式对应的每一个语义规则

b:=f(c_1,c_2,...,c_k) do

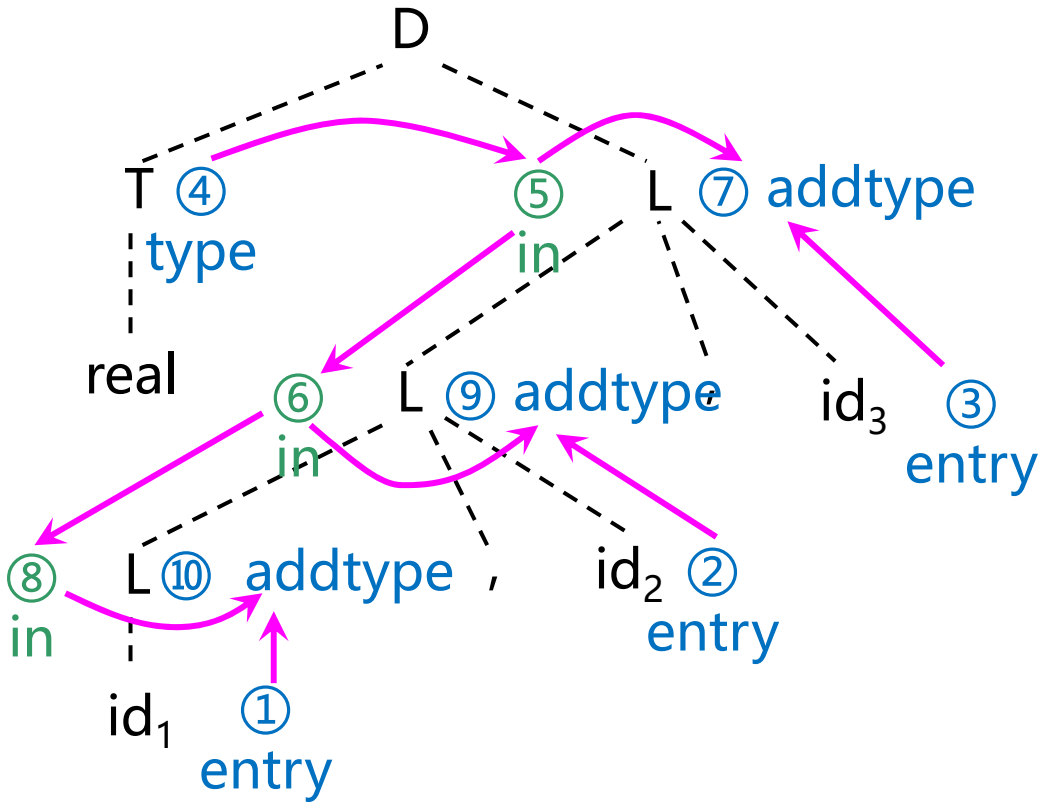
for i:=1 to k do

从c_i结点到b结点构造一条有向边；

依赖图示例

- 句子real id_1，id_2，id_3的依赖图

产生式	语义规则
D→TL	L.in := T.type
T→int	T.type := integer
T→real	T.type := real
$L \rightarrow L_1$, id	L_1.in :=L.in
	addtype(id.entry, L.in)
L→id	addtype(id.entry, L.in)



良定义的属性文法

- 如果一属性文法不存在属性之间的循环依赖关系，则称该文法为良定义的
- 一个依赖图的任何拓扑排序都给出一个语法树中结点的语义规则计算的有效顺序

属性的计算次序

- 基础文法用于建立输入符号串的语法分析树
- 根据语义规则建立依赖图
- 根据依赖图的拓扑排序，得到计算语义规则的顺序

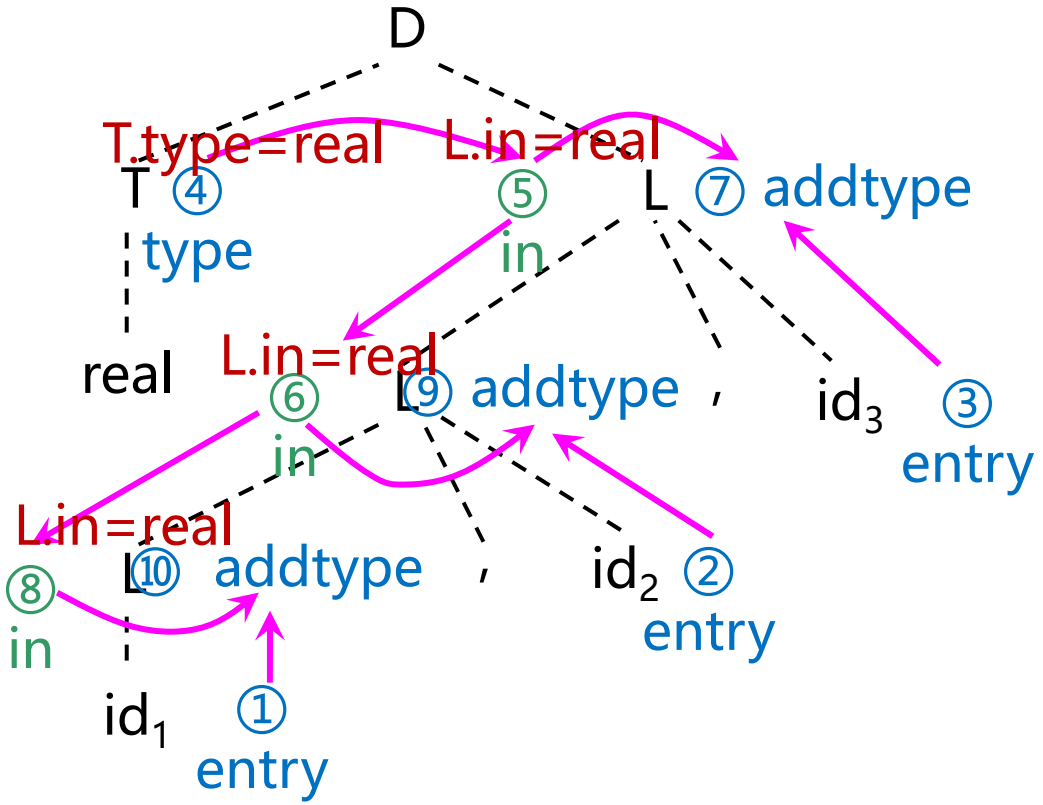
输入串 ⟶ 语法树 ⟶ 依赖图 ⟶ 语义规则计算次序

依赖图示例

- 句子real id_1，id_2，id_3的依赖图

name	type
...	...
id_3	real
id_2	real
id_1	real

产生式	语义规则
D→TL	L.in := T.type
T→int	T.type := integer
T→real	T.type := real
$L→L_1$, id	L_1.in :=L.in
	addtype(id.entry, L.in)
L→id	addtype(id.entry, L.in)



编译原理

树遍历

树遍历的属性计算方法

- 通过树遍历的方法计算属性的值
  - 假设语法树已建立，且树中已带有开始符号的继承属性和终结符的综合属性
  - 以某种次序遍历语法树，直至计算出所有属性
  - 深度优先，从左到右的遍历

输入串 → 语法树 → 遍历语法树
计算属性

树遍历算法

- 计算思维的典型方法--递归
  - 问题的解决又依赖于类似问题的解决，只不过后者的复杂程度或规模较原来的问题更小
  - 一旦将问题的复杂程度和规模化简到足够小时，问题的解法其实非常简单

```
While 还有未被计算的属性 do
   VisitNode(S)   /*S是开始符号*/

procedure VisitNode (N:Node) ;
begin
   if N是一个非终结符 then  /*假设其产生式为N→X1...Xm*/
      for i :=1 to  m  do
         if Xi∈VN then /*即Xi是非终结符*/
         begin
            计算Xi的所有能够计算的继承属性;
            VisitNode (Xi)
         end;
      计算N的所有能够计算的综合属性
end
```

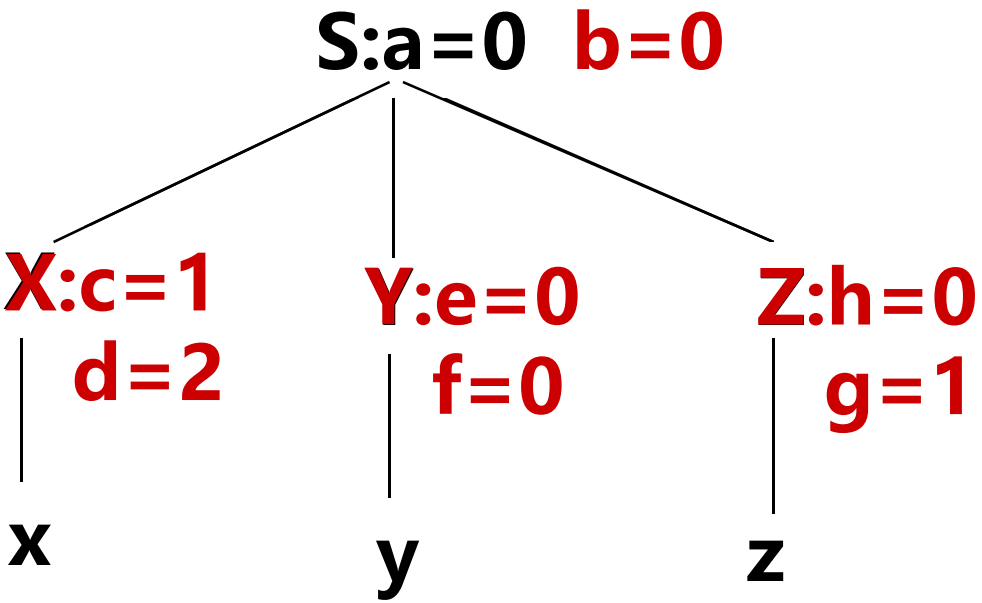
树遍历算法示例

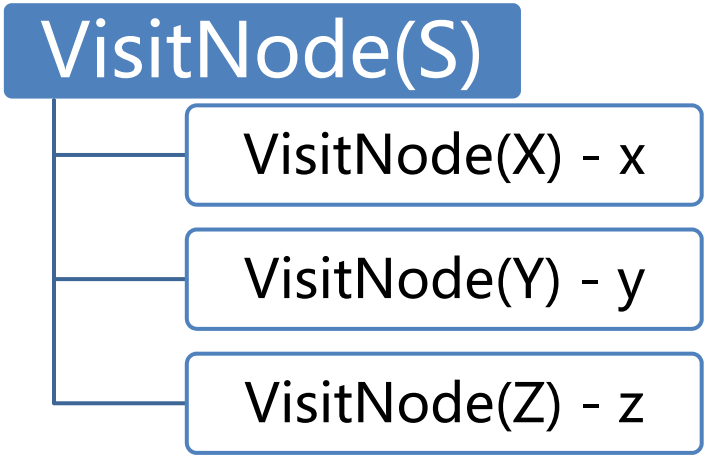
- 考虑属性的文法G(S)，其中
  - S有继承属性a，综合属性b
  - X有继承属性c、综合属性d
  - Y有继承属性e、综合属性f
  - Z有继承属性h、综合属性g

产生式	语义规则
S→XYZ	Z.h := S.a
	X.c := Z.g
	S.b := X.d -2
	Y.e := S.b
X→x	X.d :=2*X.c
Y→y	Y.f := Y.e*3
Z→z	Z.g :=Z.h+1

树遍历算法示例

- 输入串为xyz
- 假设S.a的初始值为0





产生式	语义规则
S→XYZ	Z.h := S.a
	X.c := Z.g
	S.b := X.d -2
	Y.e := S.b
X→x	X.d :=2*X.c
Y→y	Y.f := Y.e*3
Z→z	Z.g :=Z.h+1

编译原理

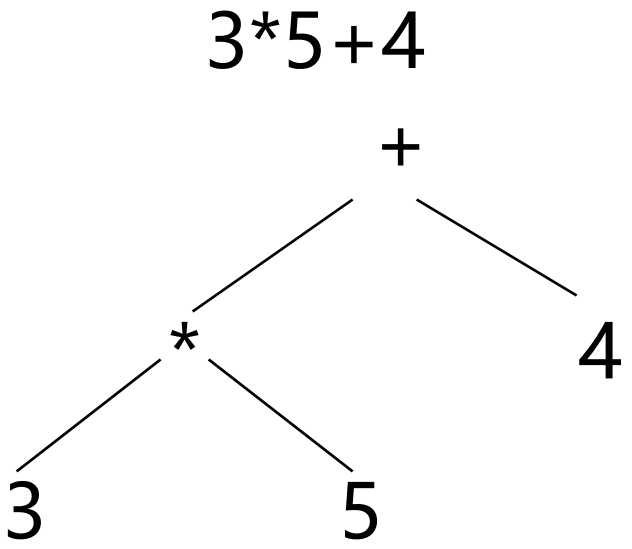
一遍扫描

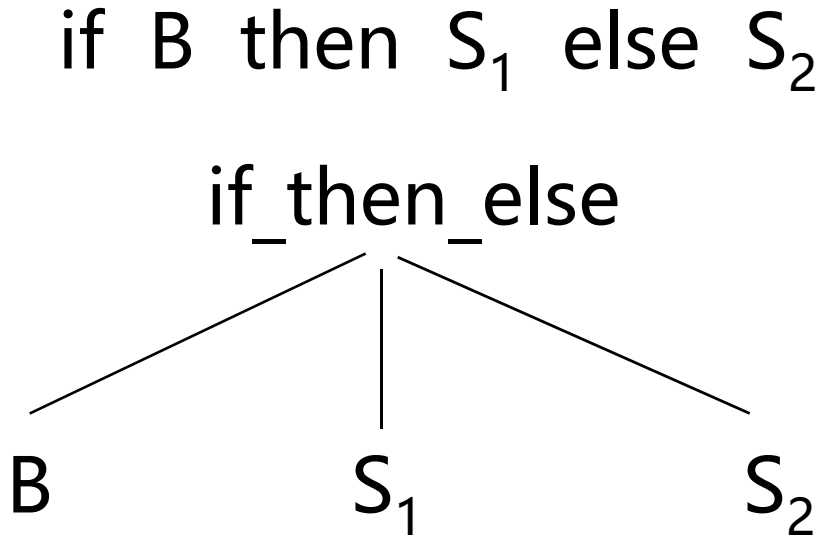
一遍扫描的处理方法

- 在语法分析的同时计算属性值
  - 所采用的语法分析方法
  - 属性的计算次序
- 所谓语法制导翻译法，直观上说就是为文法中每个产生式配上一组语义规则，并且在语法分析的同时执行这些语义规则
- 语义规则被计算的时机
  - 自上而下分析，一个产生式匹配输入串成功时
  - 自下而上分析，一个产生式被用于进行归约时

抽象语法树

- 抽象语法树(Abstract Syntax Tree，AST)，在语法树中去掉那些对翻译不必要的信息，从而获得更有效的源程序中间表示





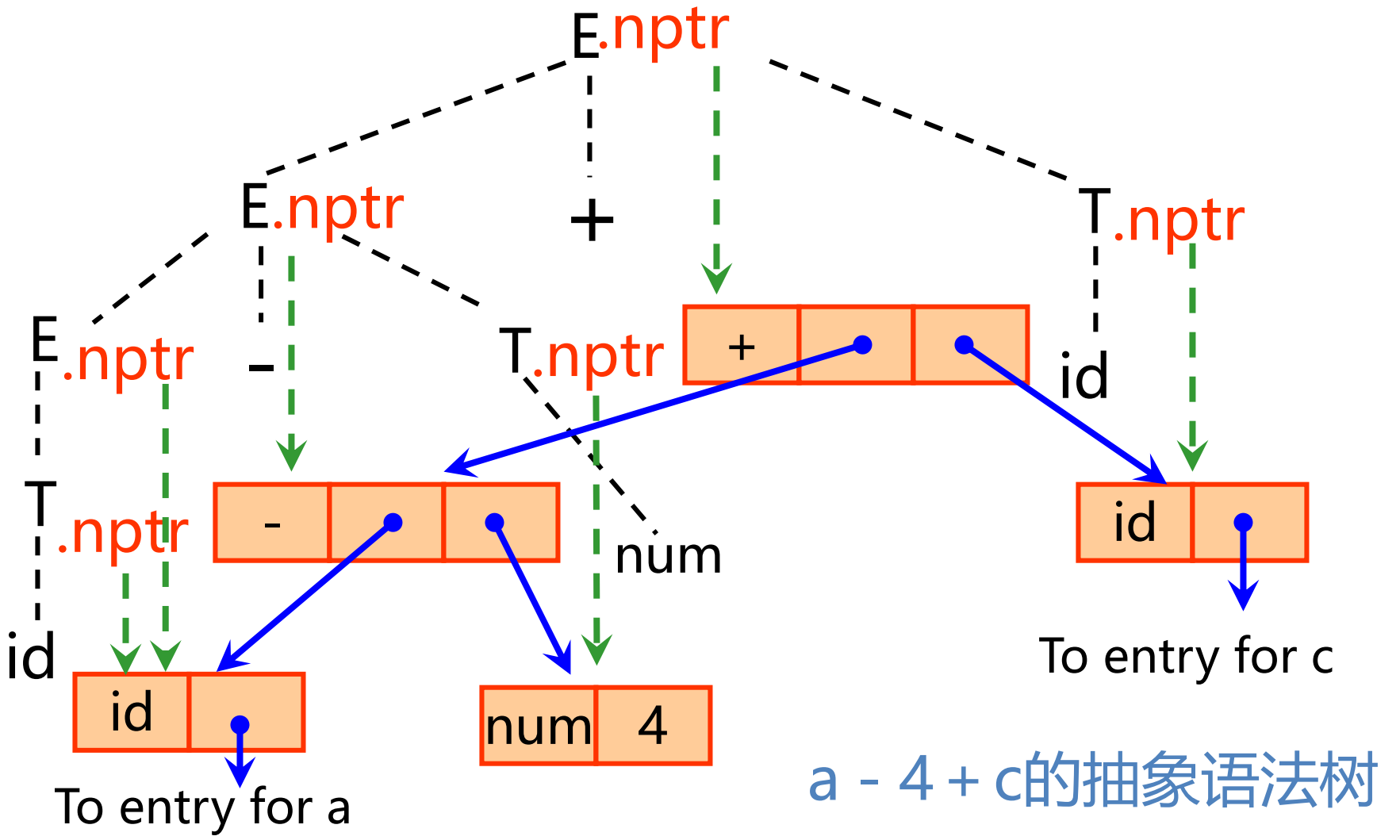
建立表达式的抽象语法树

- mknode(op，left，right) 建立一个运算符号结点，标号是op，两个域left和right分别指向左子树和右子树
- mkleaf(id，entry) 建立一个标识符结点，标号为id，一个域entry指向标识符在符号表中的入口
- mkleaf(num，val) 建立一个数结点，标号为num，一个域val用于存放数的值

建立抽象语法树的语义规则

产 生 式	语 义 规 则
$E \rightarrow E_1+T$	E.nptr := mknode(‘+’, E_1.nptr, T.nptr)
$E \rightarrow E_1-T$	E.nptr := mknode(‘-’, E_1.nptr, T.nptr)
$E \rightarrow T$	E.nptr := T.nptr
$T \rightarrow (E)$	T.nptr := E.nptr
$T \rightarrow id$	T.nptr := mkleaf (id, id.entry)
$T \rightarrow num$	T.nptr := mkleaf (num, num.val)

$E \rightarrow E_1 + T$	E.nptr := mknode('+' , E_1.nptr, T.nptr)
$E \rightarrow E_1 - T$	E.nptr := mknode('-' , E_1.nptr, T.nptr)
$E \rightarrow T$	E.nptr := T.nptr
$T \rightarrow id$	T.nptr := mkleaf (id, id.entry)
$T \rightarrow num$	T.nptr := mkleaf (num, num.val)



a - 4 + c的抽象语法树

小结

- 基于属性文法的处理方法
  - 依赖图
  - 树遍历
  - 一遍扫描