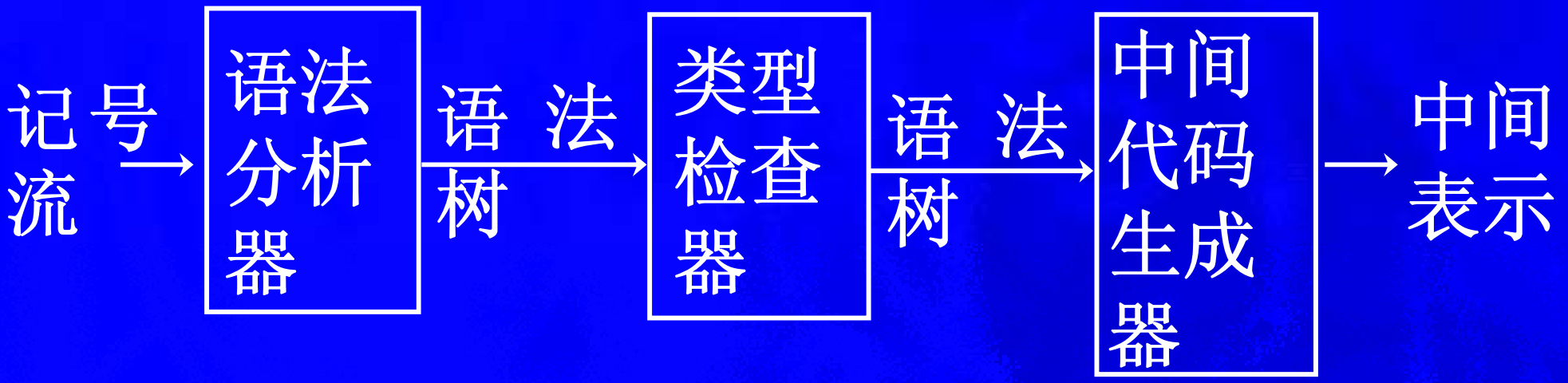
第4章 类 型 检 查



本章内容

- 简要介绍语法制导的翻译
- 介绍静态检查中最典型的部分 —— 类型检查

  忽略其他的静态检查：控制流检查、唯一性检查、关联名字检查等

4.1 语法制导的翻译

4.1.1 翻译方案

- 例 简单台式计算器的语法制导定义

产 生 式	语 义 动 作
$L \rightarrow E$ n	*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*

4.1 语法制导的翻译

- 语法制导的翻译方案的形式
  - 基础文法
  - 每个文法符号有一组属性
  - 每个文法产生式$A \to \alpha$有一组形式为

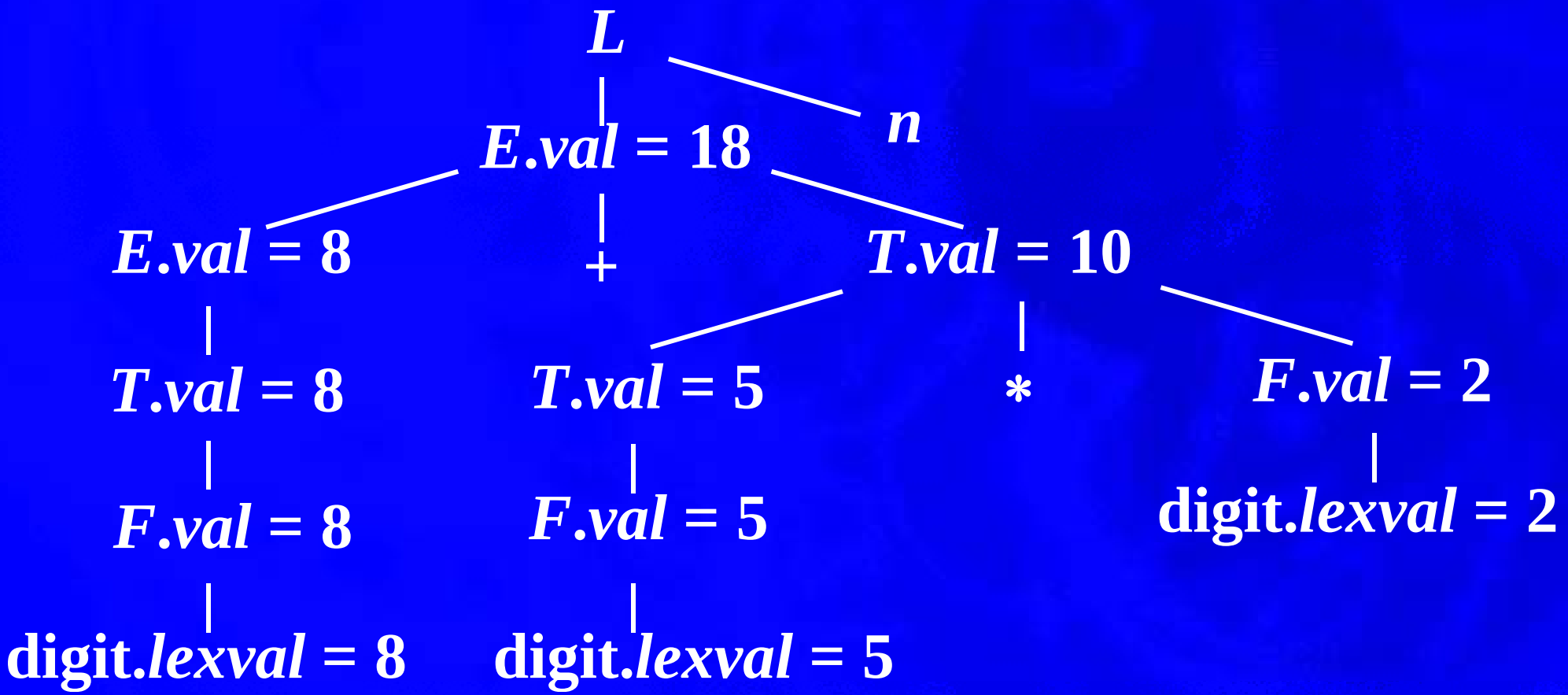
$$b = f(c_1, c_2, \ldots, c_k)$$

的语义动作，其中b 是A 的属性，$c_1, c_2, \ldots, c_k$ 是产生式右部文法符号的属性或 A 的其他属性，f 是函数

  - 属性可以表示任何东西：串、数值、类型、表引用，或其他想表示的东西

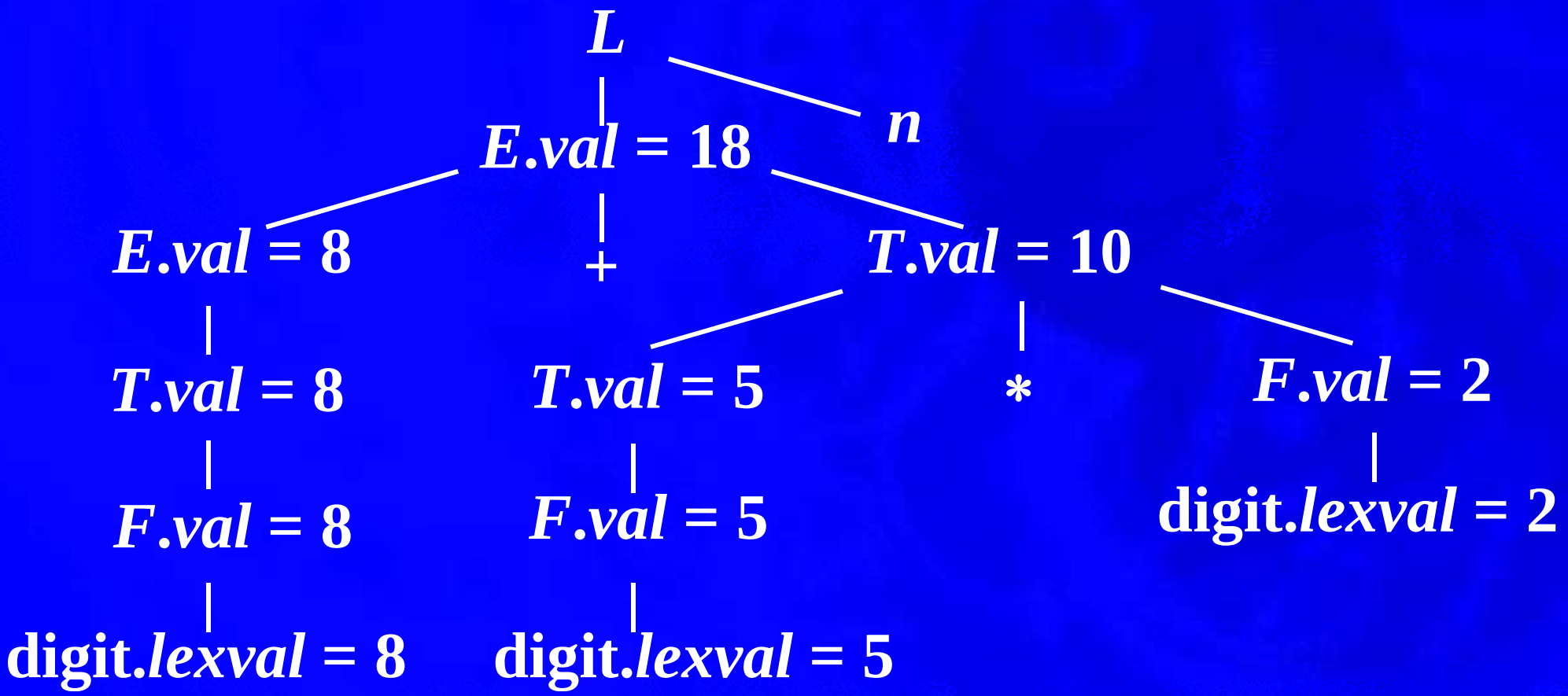
4.1 语法制导的翻译

- 例 8+5*2 n的注释分析树



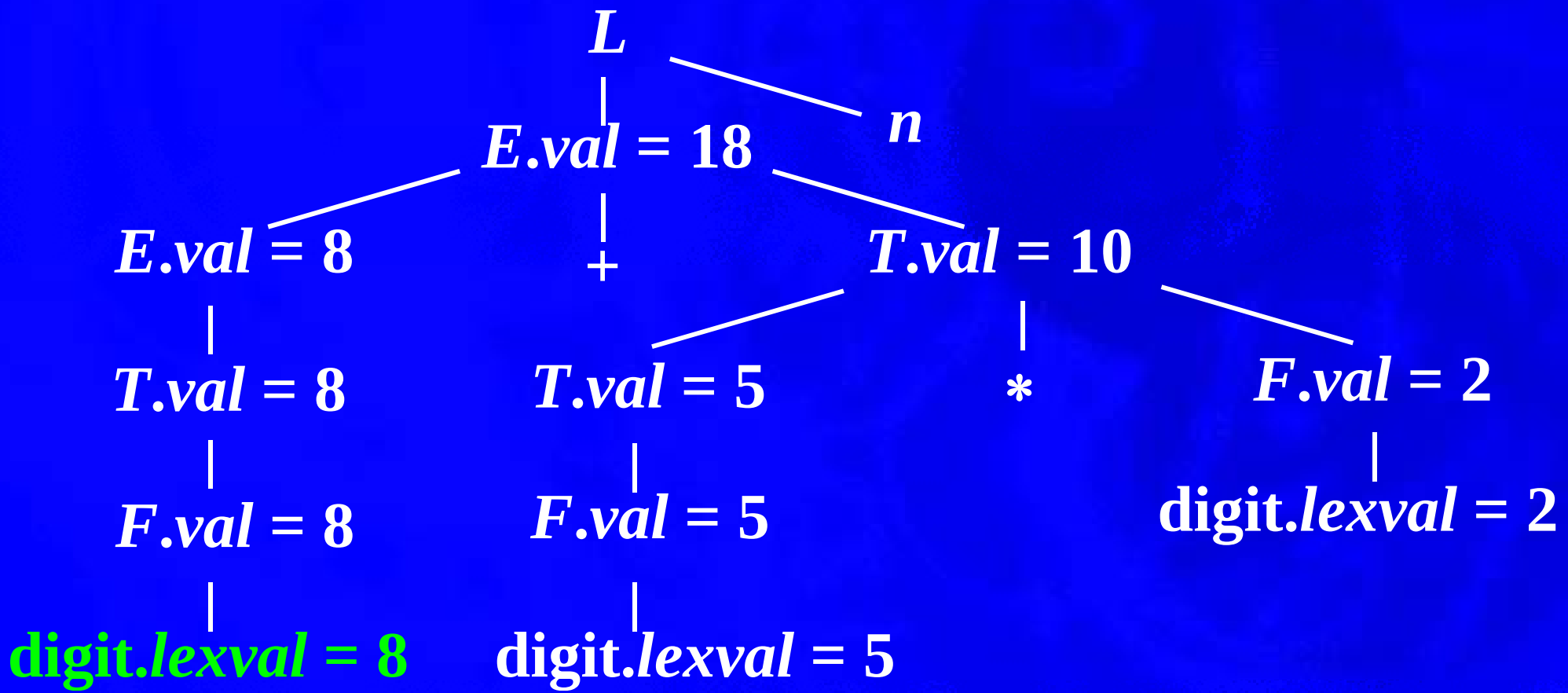
4.1 语法制导的翻译

分析树各结点属性的计算可以自下而上地完成



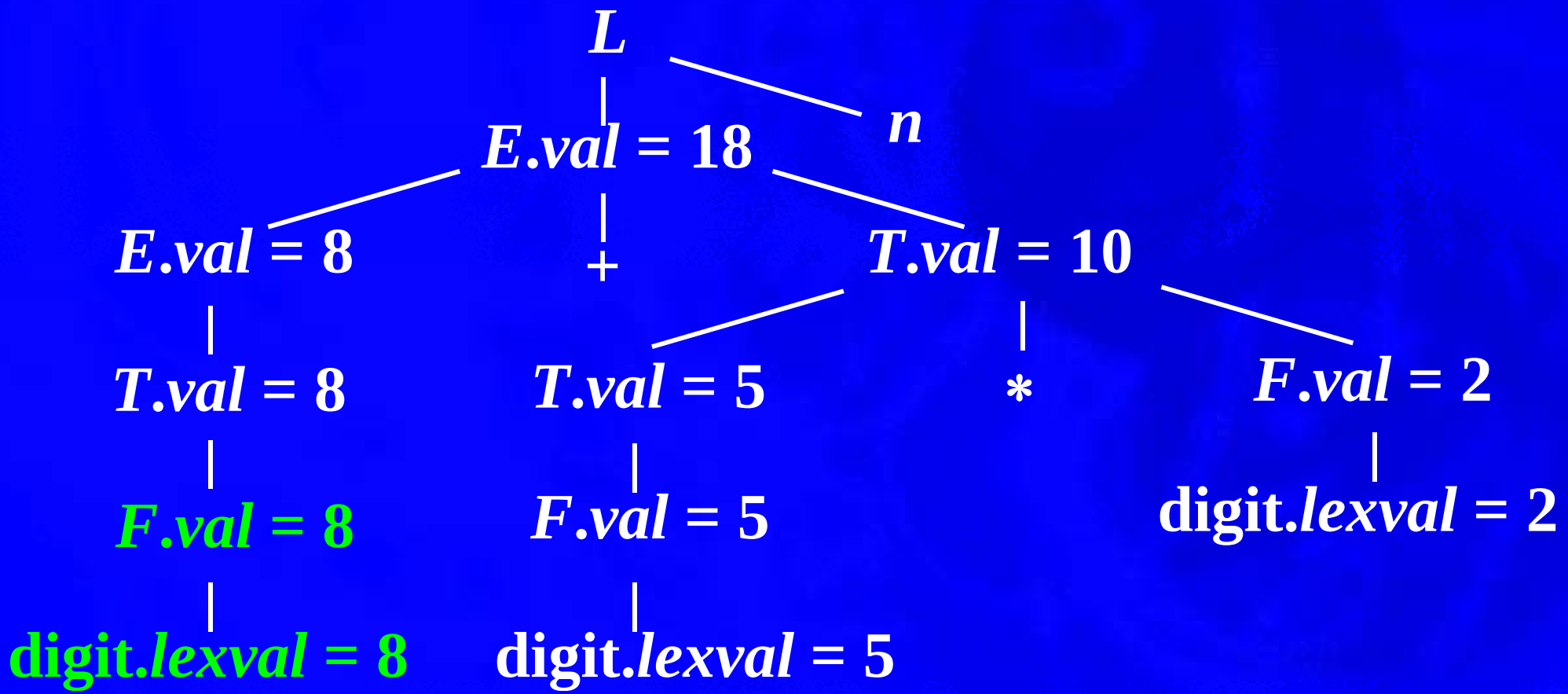
4.1 语法制导的翻译

分析树各结点属性的计算可以自下而上地完成



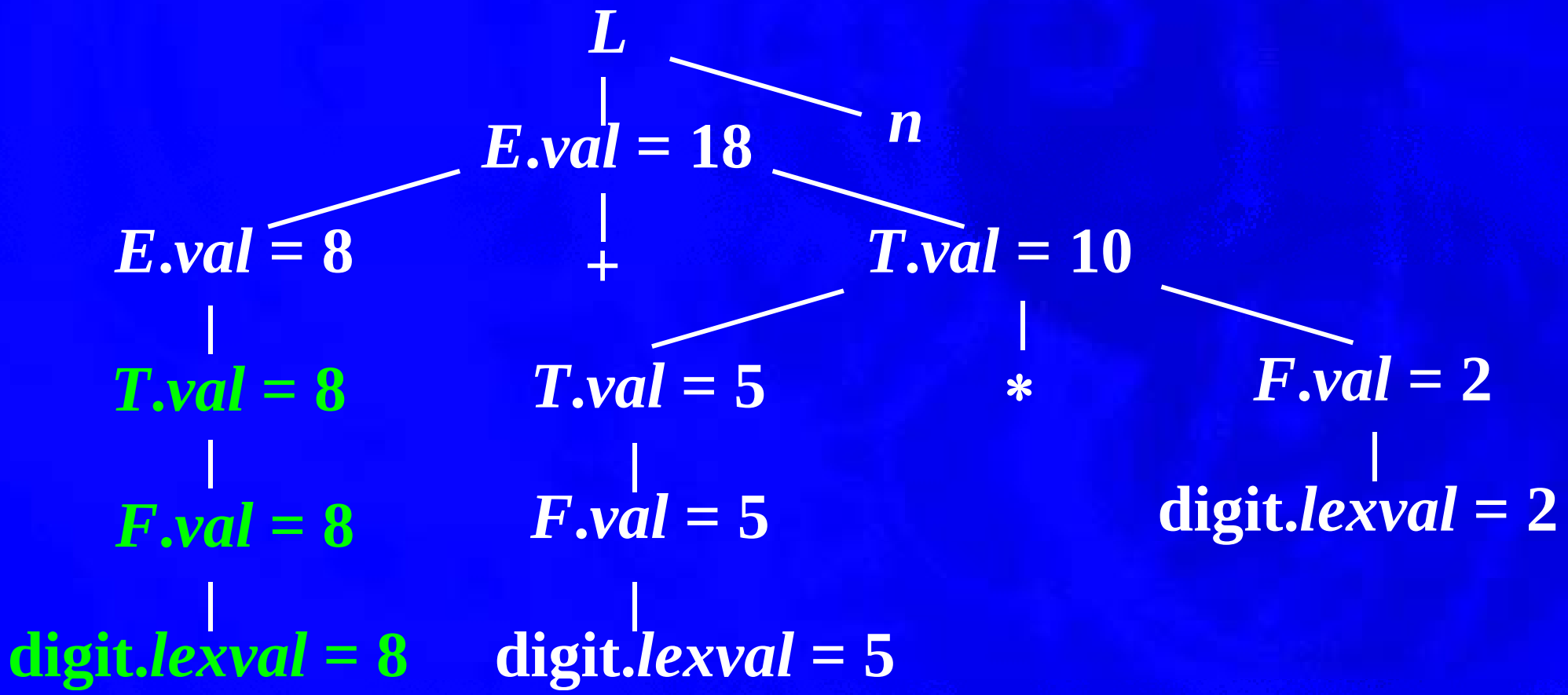
4.1 语法制导的翻译

分析树各结点属性的计算可以自下而上地完成



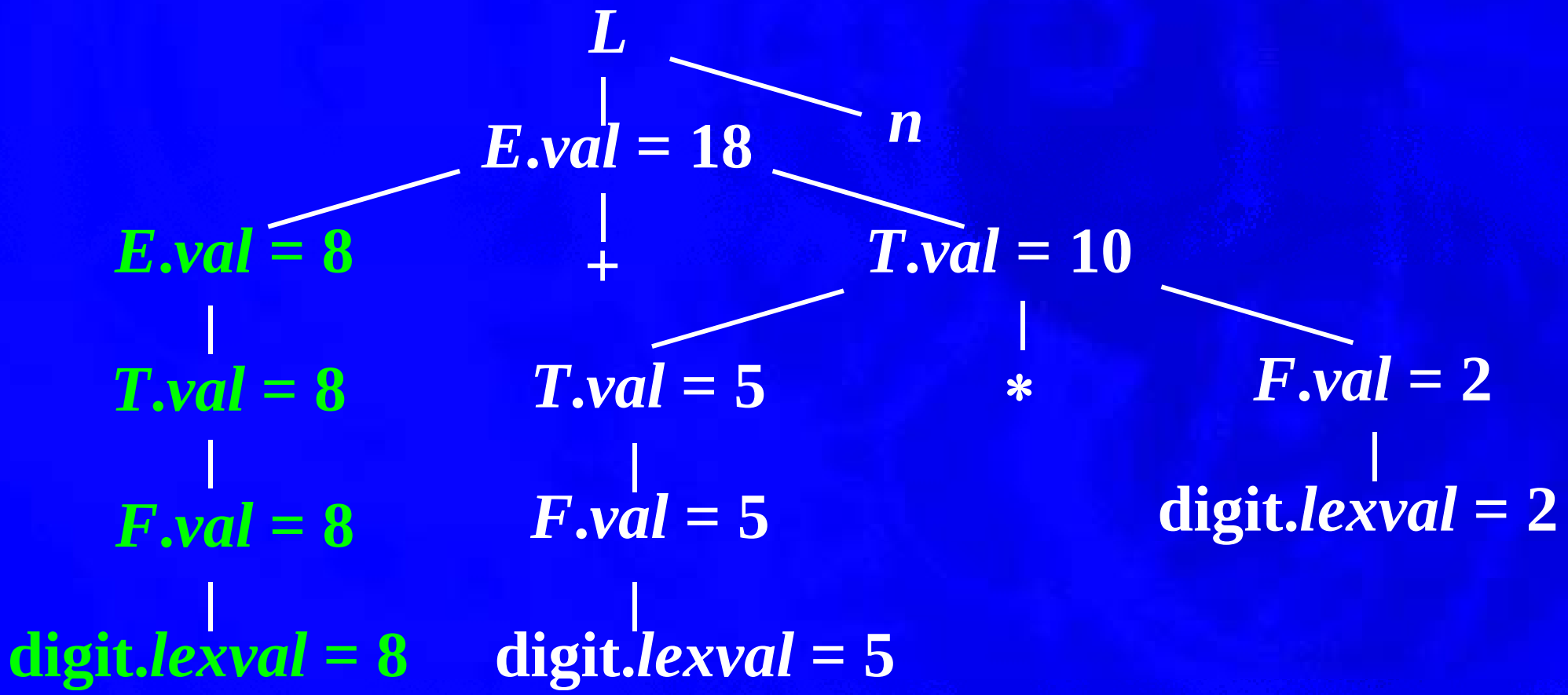
4.1 语法制导的翻译

分析树各结点属性的计算可以自下而上地完成



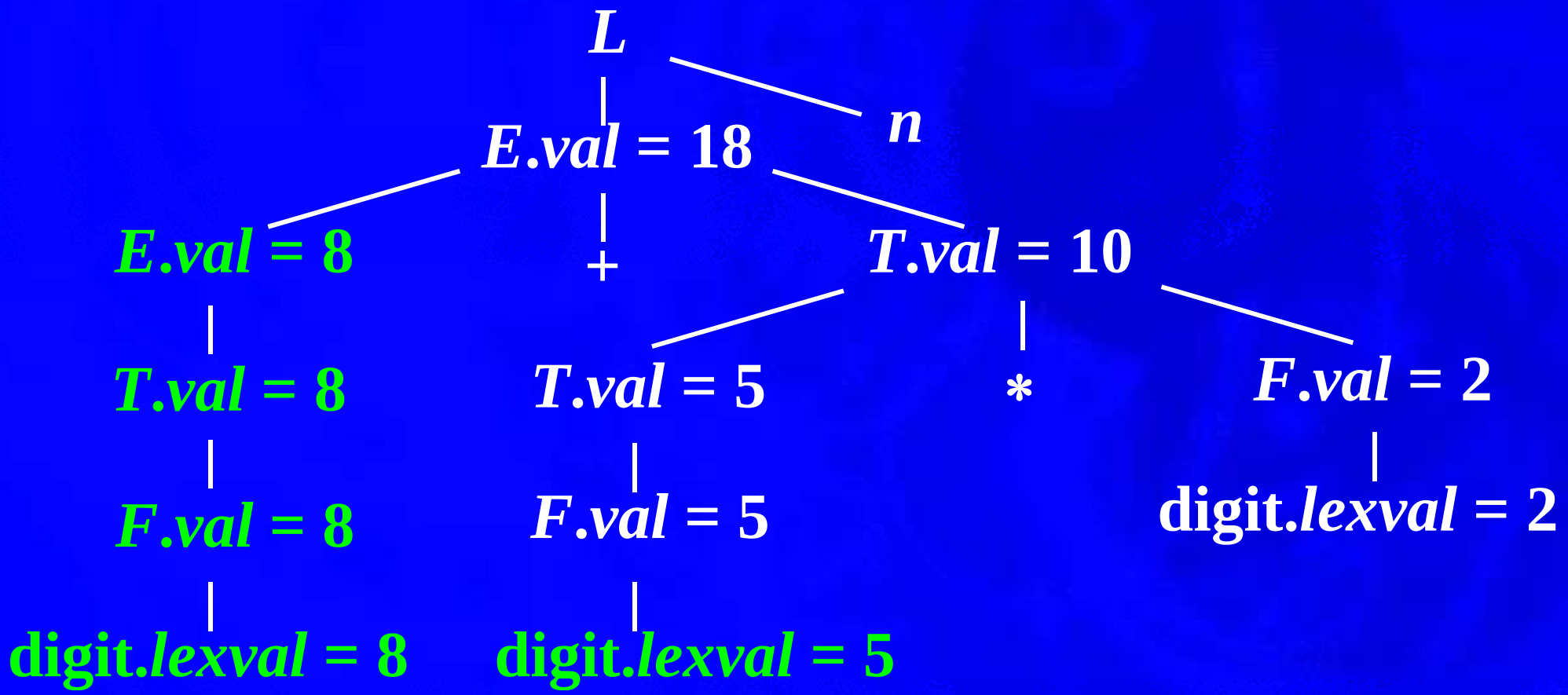
4.1 语法制导的翻译

分析树各结点属性的计算可以自下而上地完成



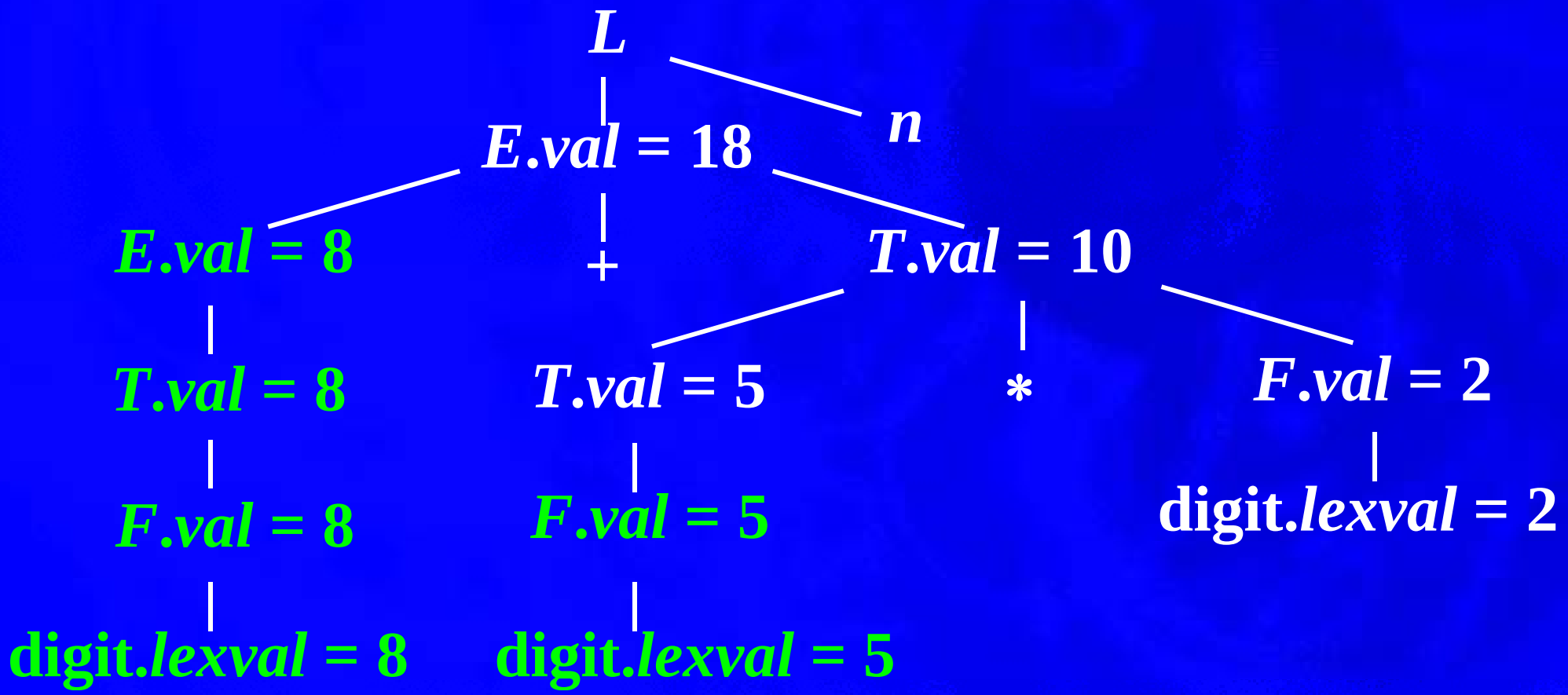
4.1 语法制导的翻译

分析树各结点属性的计算可以自下而上地完成



4.1 语法制导的翻译

分析树各结点属性的计算可以自下而上地完成



4.1 语法制导的翻译

分析树各结点属性的计算可以自下而上地完成

4.1 语法制导的翻译

分析树各结点属性的计算可以自下而上地完成

4.1 语法制导的翻译

分析树各结点属性的计算可以自下而上地完成

4.1 语法制导的翻译

分析树各结点属性的计算可以自下而上地完成

4.1 语法制导的翻译

分析树各结点属性的计算可以自下而上地完成

4.1 语法制导的翻译

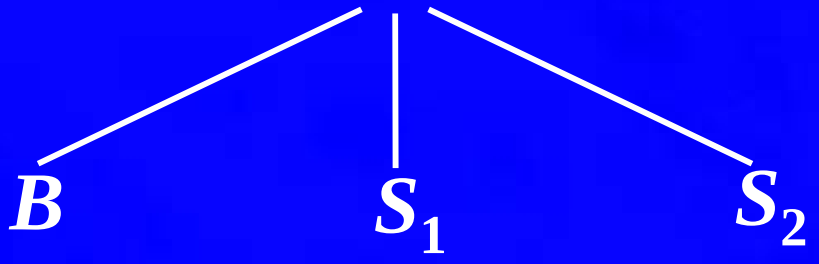
注释分析树:结点的属性值都标注出来的分析树

4.1 语法制导的翻译

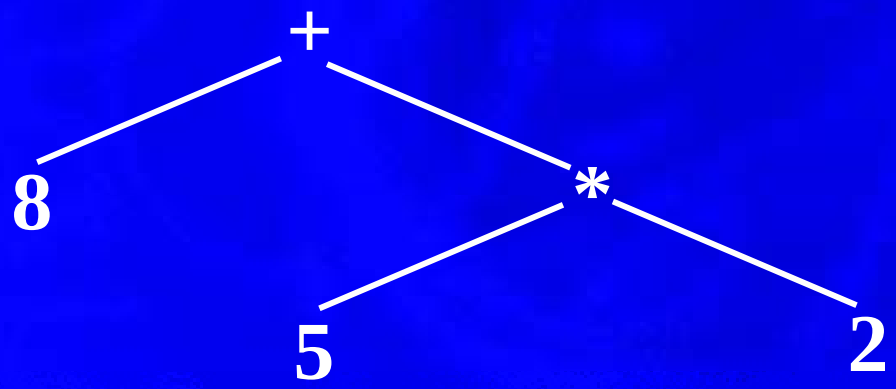
4.1.2 语法树

- 语法树是分析树的浓缩表示：算符和关键字是作为内部结点
- 语法制导翻译可以基于分析树，也可以基于语法树
- 语法树的例子：

if *B* then S_1 else S_2



8 + 5 * 2



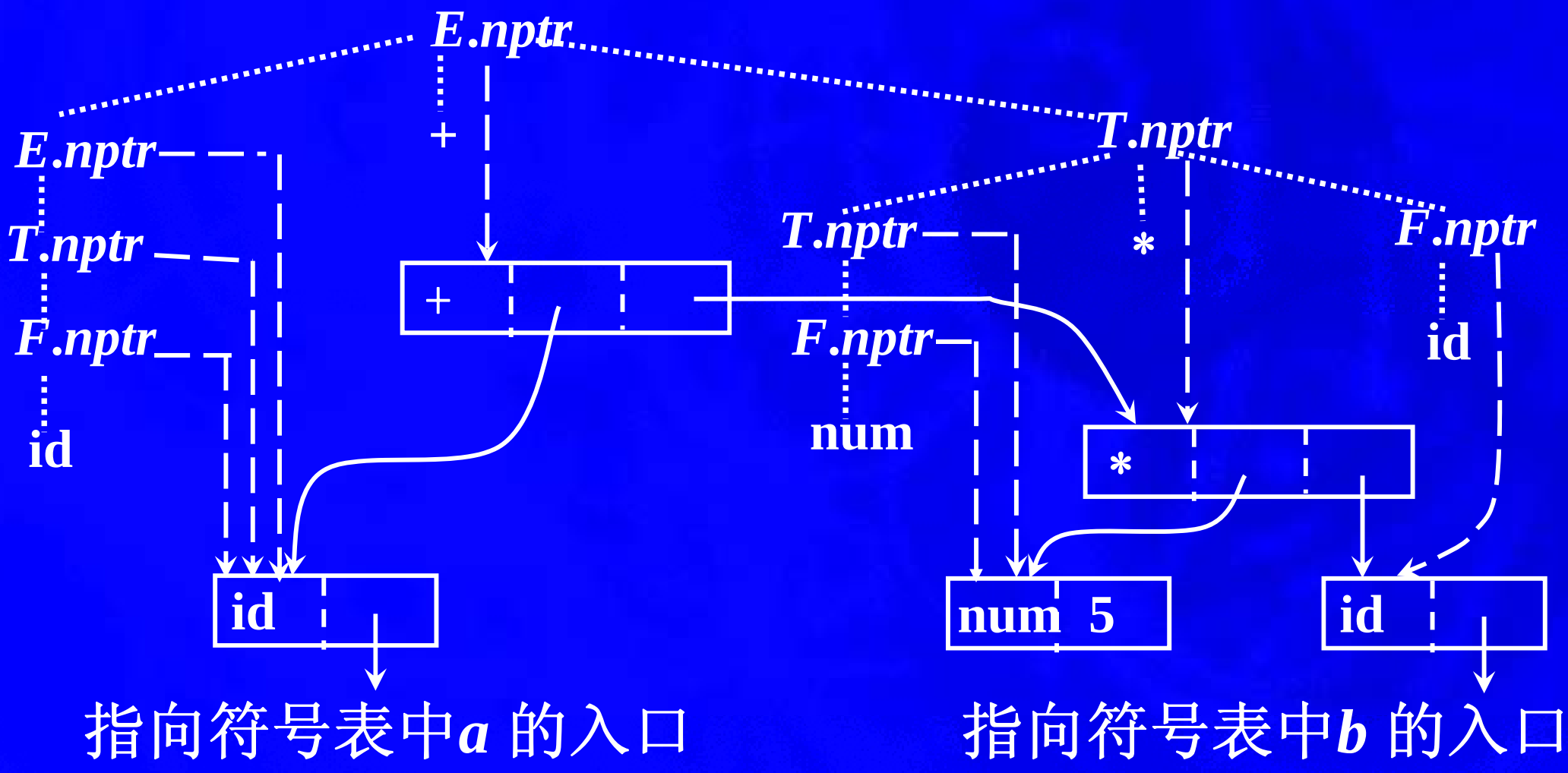
4.1 语法制导的翻译

4.1.3 构造语法树的翻译方案

产 生 式	语 义 动 作
$E \rightarrow E_1 + T$	$E.nptr = mkNode(\text{‘+’}, E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow T_1 * F$	$T.nptr = mkNode(\text{‘*’}, T_1.nptr, F.nptr)$
$T \rightarrow F$	$T.nptr = F.nptr$
$F \rightarrow (E)$	$F.nptr = E.nptr$
$F \rightarrow$ id	$F.nptr = mkLeaf$ (id, id.*entry*)
$F \rightarrow$ num	$F.nptr = mkLeaf$ (num, num.*val*)

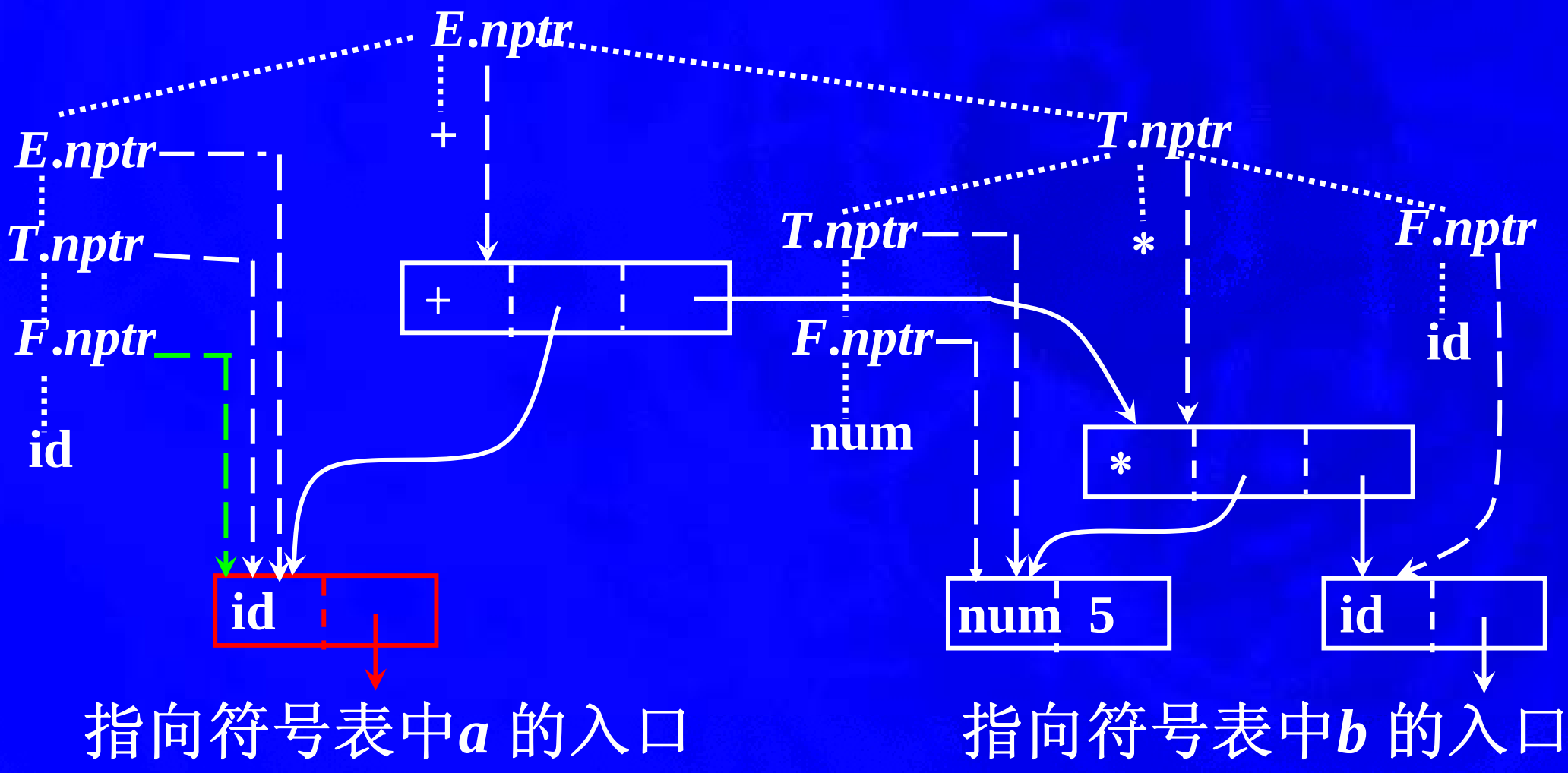
4.1 语法制导的翻译

a+5**b*的语法树的构造



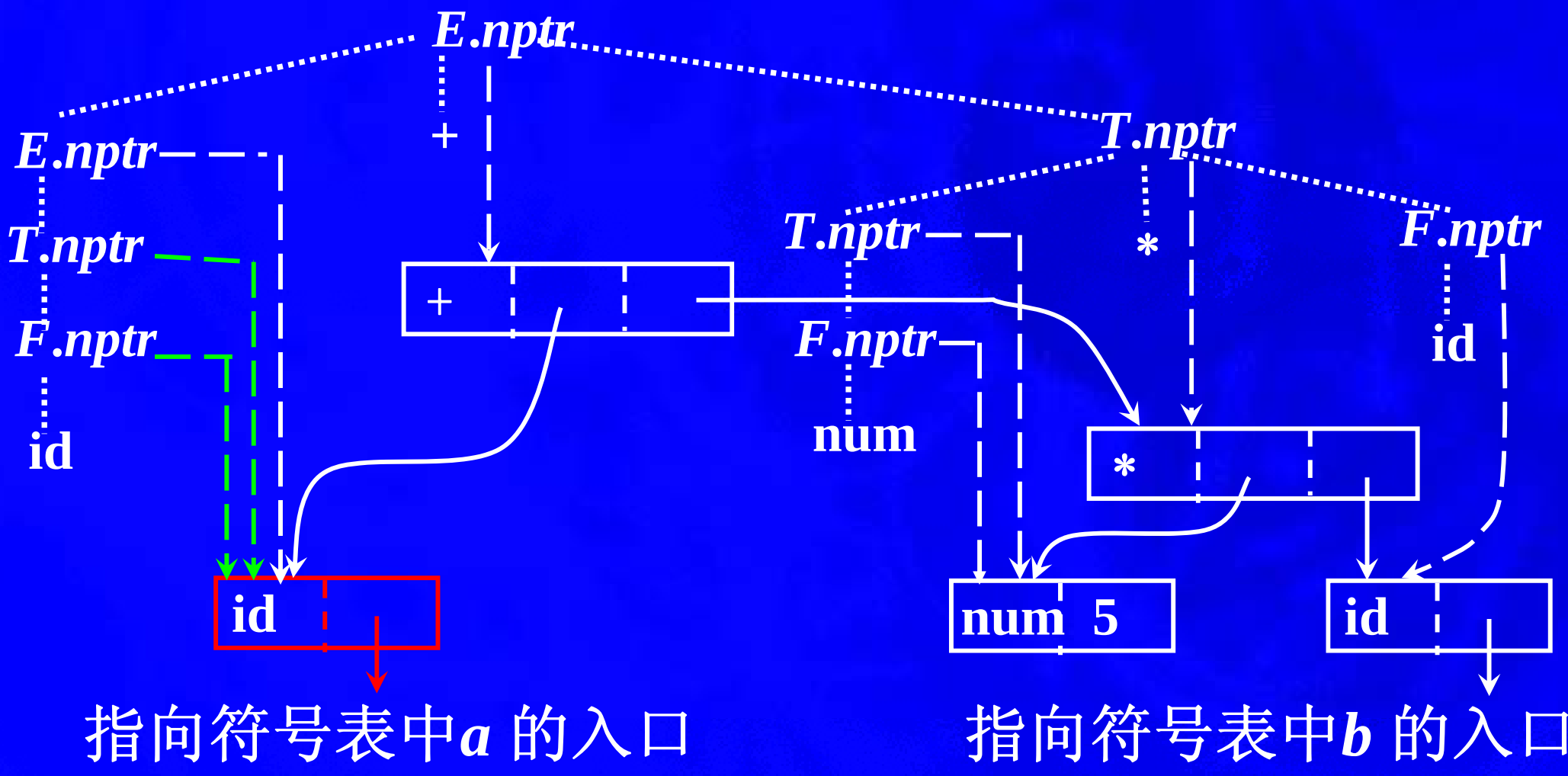
4.1 语法制导的翻译

a+5**b*的语法树的构造



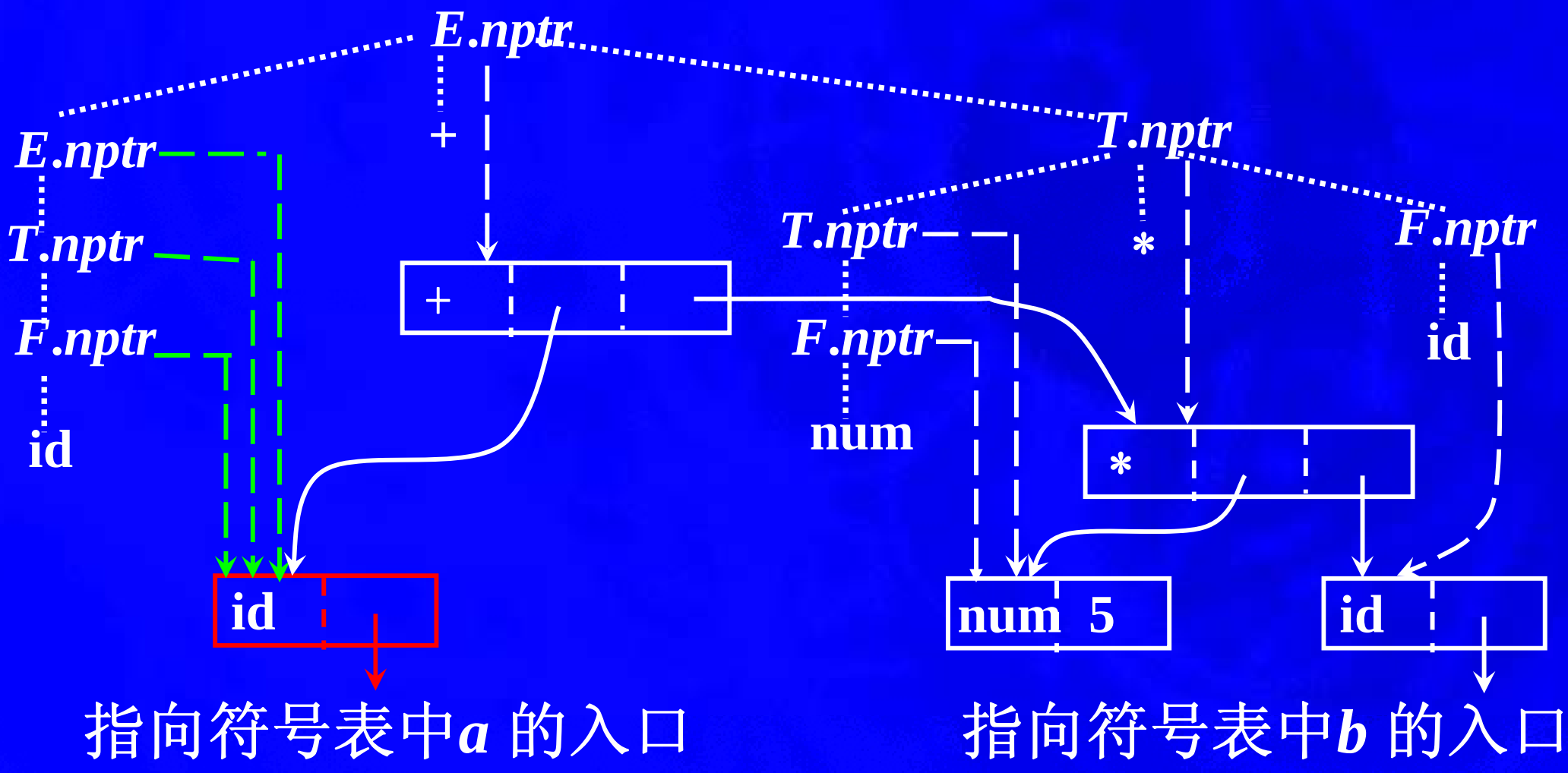
4.1 语法制导的翻译

a+5**b*的语法树的构造



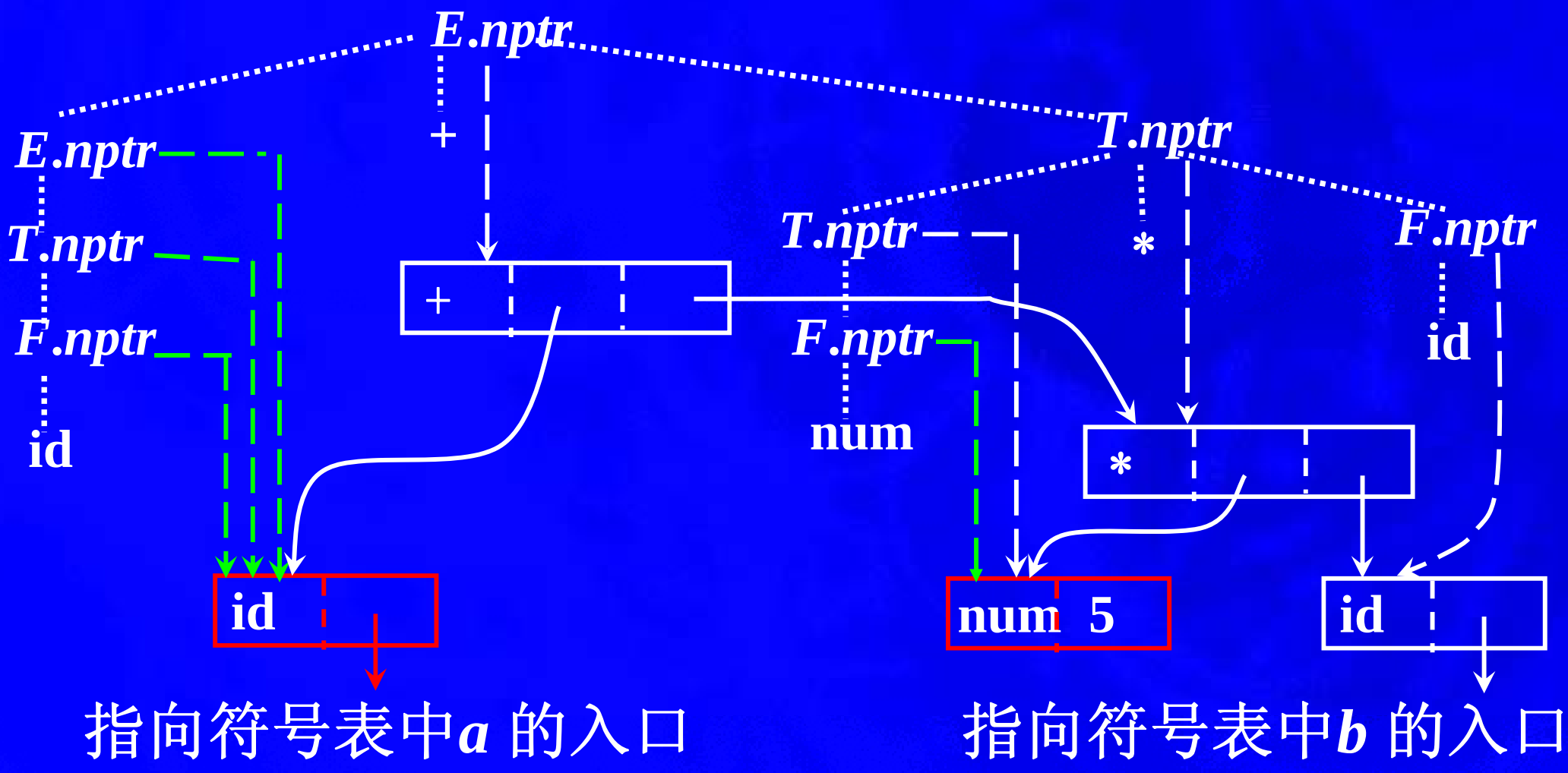
4.1 语法制导的翻译

a+5**b*的语法树的构造



4.1 语法制导的翻译

a+5**b*的语法树的构造

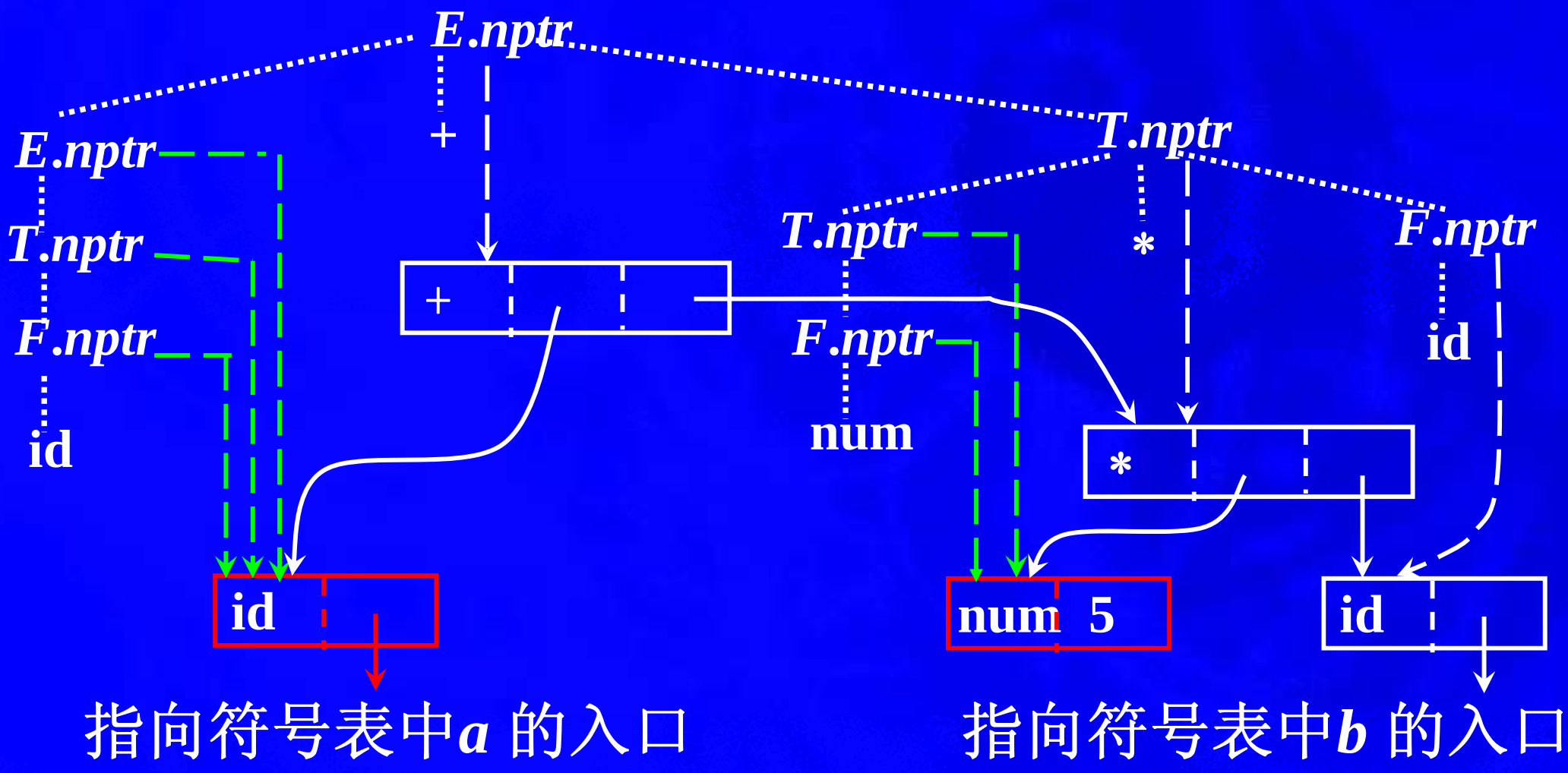


指向符号表中*a* 的入口

指向符号表中*b* 的入口

4.1 语法制导的翻译

a+5**b*的语法树的构造

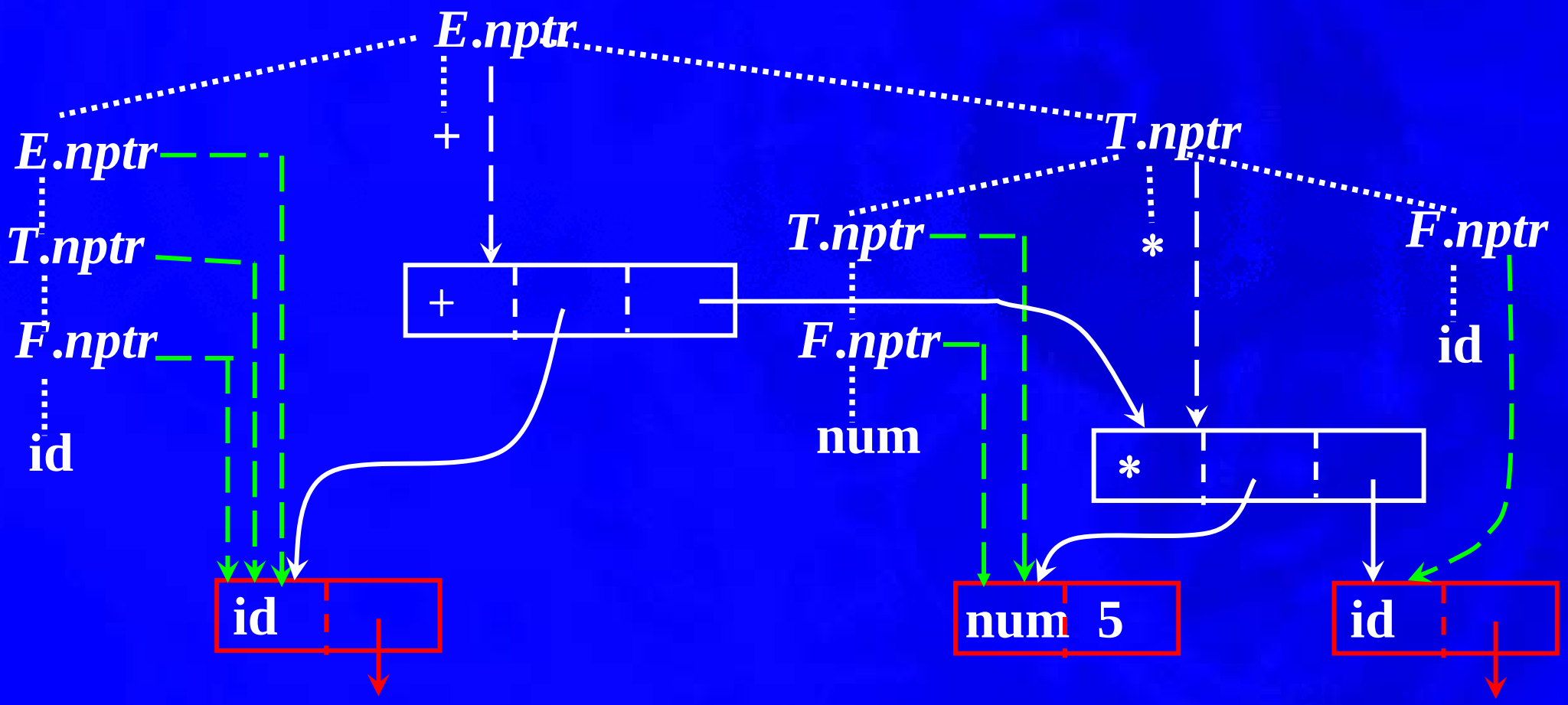


指向符号表中*a* 的入口

指向符号表中*b* 的入口

4.1 语法制导的翻译

a+5**b*的语法树的构造

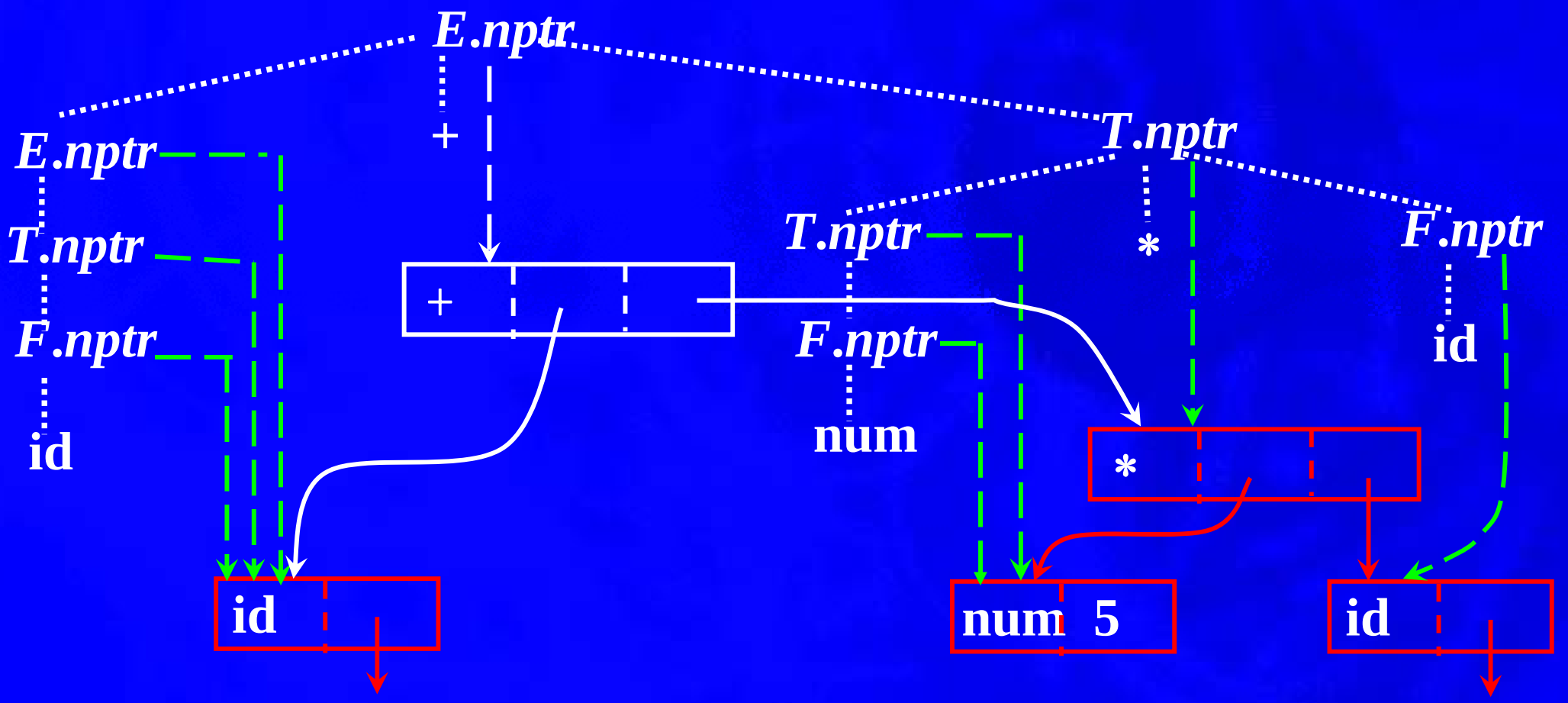


指向符号表中*a* 的入口

指向符号表中*b* 的入口

4.1 语法制导的翻译

a+5**b*的语法树的构造

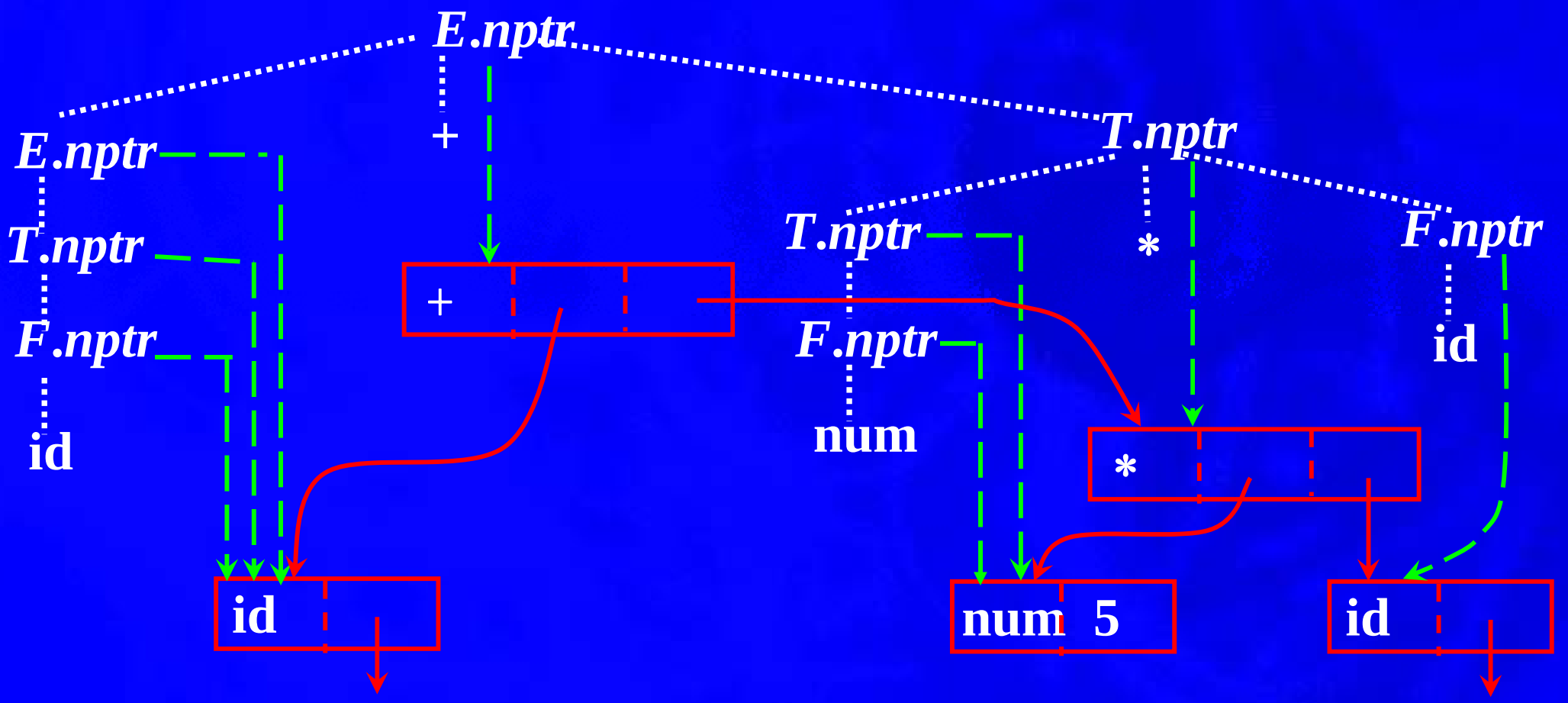


指向符号表中*a* 的入口

指向符号表中*b* 的入口

4.1 语法制导的翻译

a+5**b*的语法树的构造



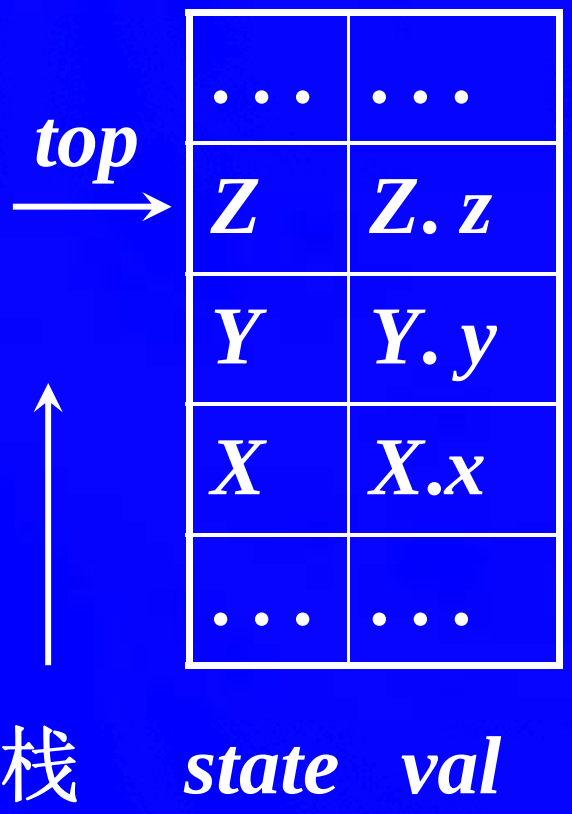
指向符号表中*a* 的入口

指向符号表中*b* 的入口

4.1 语法制导的翻译

4.1.4 翻译方案中属性的自下而上计算

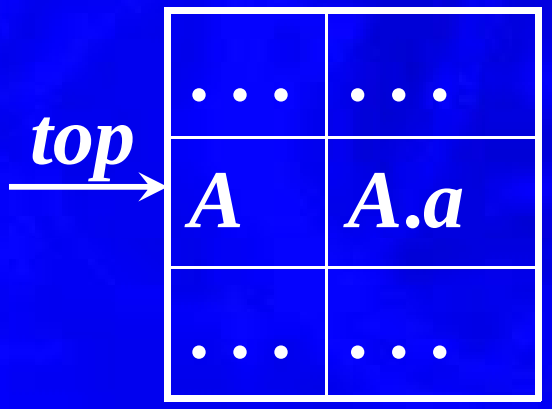
在LR分析器的栈中增加一个域来保存综合属性值



若产生式$A \rightarrow XYZ$的语义动作是

$A.a = f(X.x, Y.y, Z.z)$,

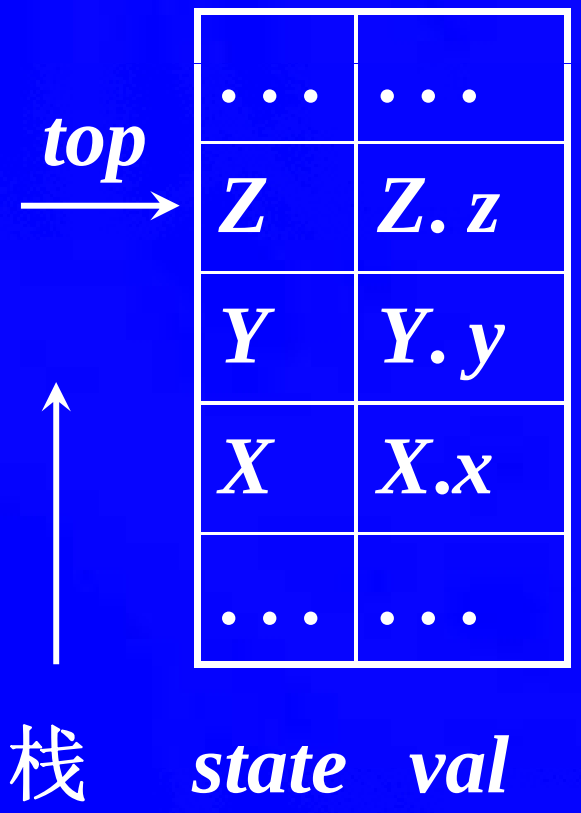
那么归约后：



4.1 语法制导的翻译

台式计算器的语法制导定义改成栈操作代码

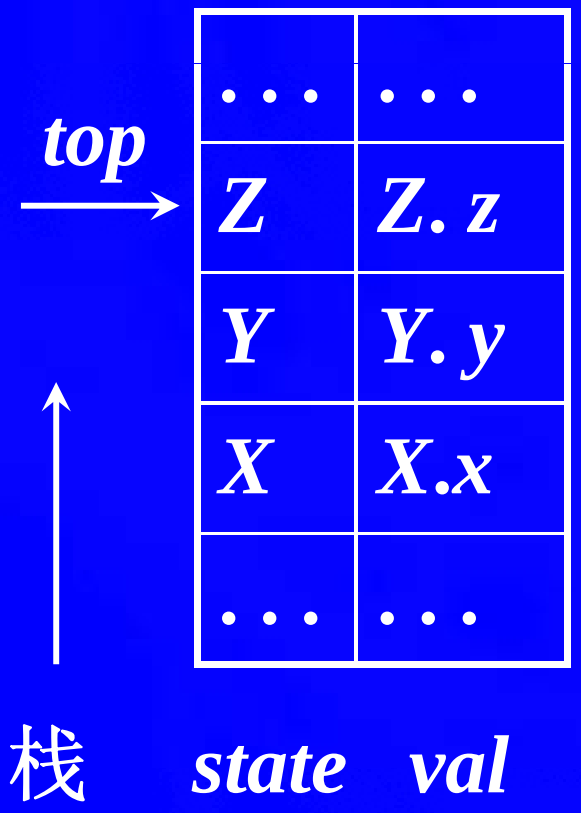
产 生 式	语 义 动 作
$L \rightarrow E$ n	*print* (*E.val*)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	*E.val* = *T.val*
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	*T.val* = *F.val*
$F \rightarrow (E)$	*F.val* = *E.val*
$F \rightarrow$ digit	*F.val* = digit.*lexval*



4.1 语法制导的翻译

台式计算器的语法制导定义改成栈操作代码

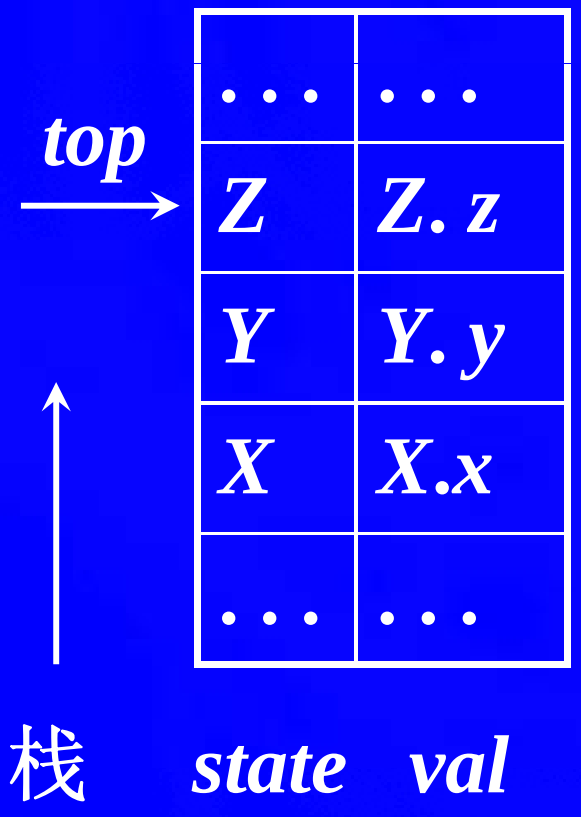
产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*E.val*)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	*E.val* = *T.val*
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	*T.val* = *F.val*
$F \rightarrow (E)$	*F.val* = *E.val*
$F \rightarrow$ digit	*F.val* = digit.*lexval*



4.1 语法制导的翻译

台式计算器的语法制导定义改成栈操作代码

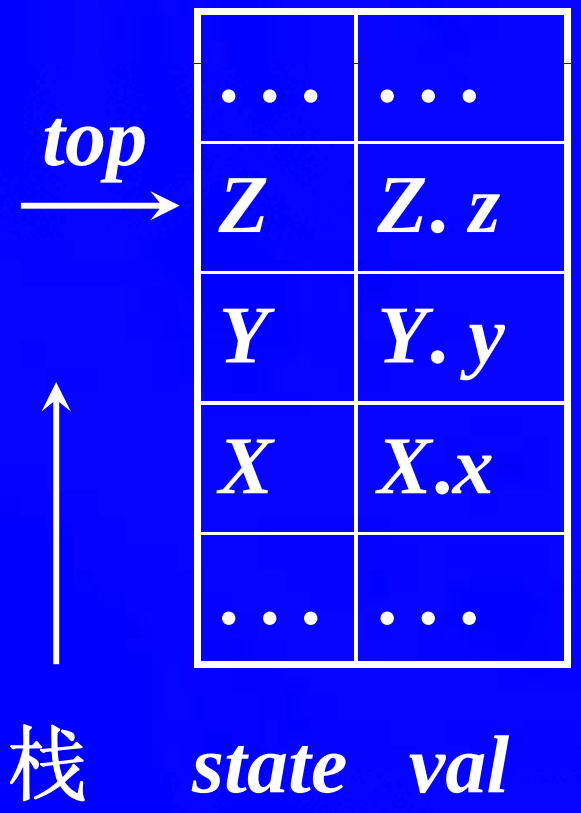
产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val* [*top*–1])
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ digit	*F.val* = digit.*lexval*



4.1 语法制导的翻译

台式计算器的语法制导定义改成栈操作代码

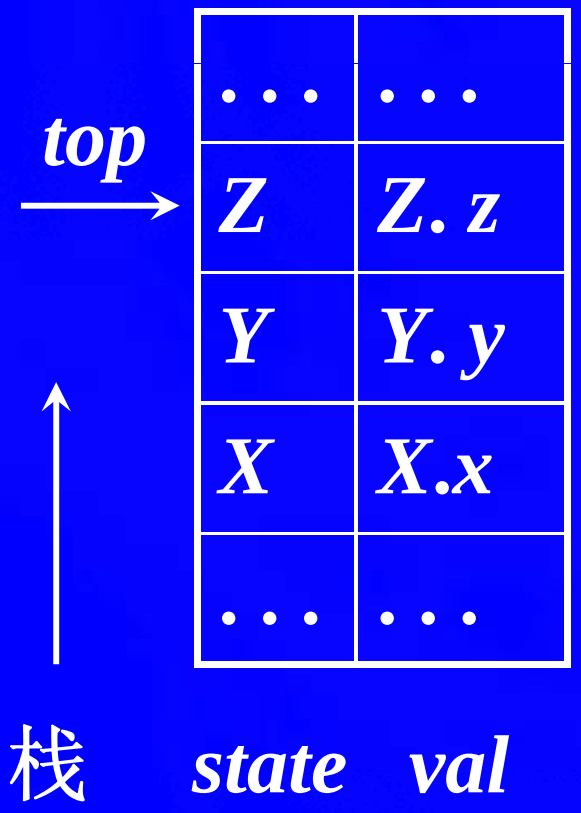
产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val* [*top*–1])
$E \rightarrow E_1 + T$	*val* [*top* –2] = *val* [*top* –2]+*val* [*top*]
$E \rightarrow T$	*E.val* = *T.val*
$T \rightarrow T_1 * F$	*T.val* = T_1.*val* * *F.val*
$T \rightarrow F$	*T.val* = *F.val*
$F \rightarrow (E)$	*F.val* = *E.val*
$F \rightarrow$ digit	*F.val* = digit.*lexval*



4.1 语法制导的翻译

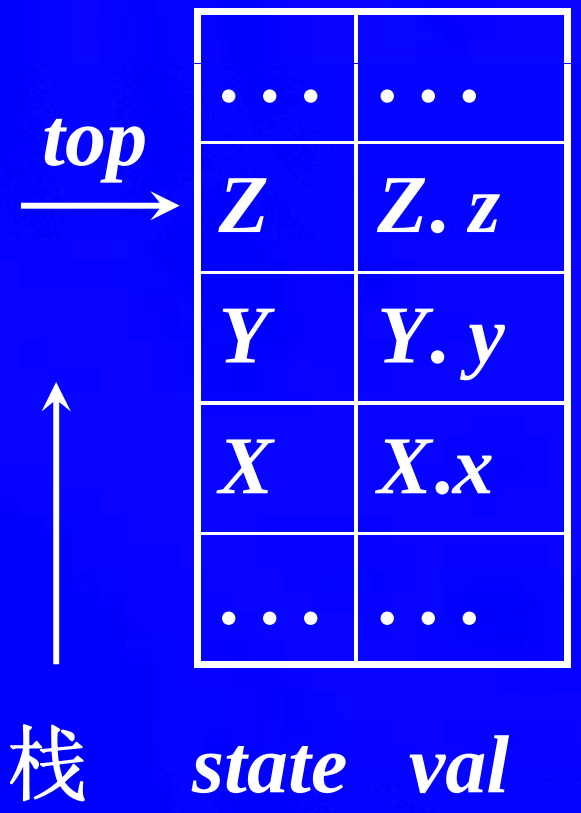
台式计算器的语法制导定义改成栈操作代码

产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val* [*top*–1])
$E \rightarrow E_1 + T$	*val* [*top* –2] = *val* [*top* –2]+*val* [*top*]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ digit	$F.val =$ digit.*lexval*



4.1 语法制导的翻译

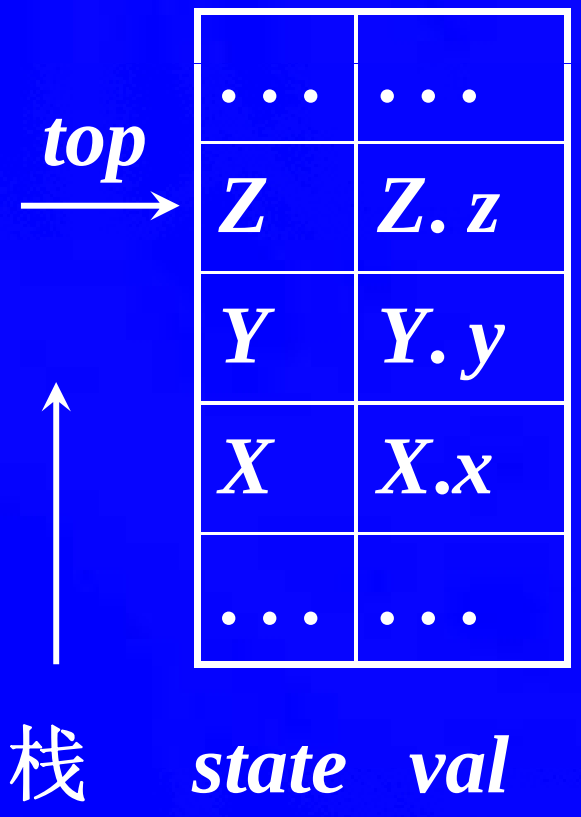
台式计算器的语法制导定义改成栈操作代码



产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val* [*top*–1])
$E \rightarrow E_1 + T$	*val* [*top* –2] = *val* [*top* –2]+*val* [*top*]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val* [*top* –2] = *val* [*top* –2]×*val* [*top*]
$T \rightarrow F$	*T.val* = *F.val*
$F \rightarrow (E)$	*F.val* = *E.val*
$F \rightarrow$ digit	*F.val* = digit.*lexval*

4.1 语法制导的翻译

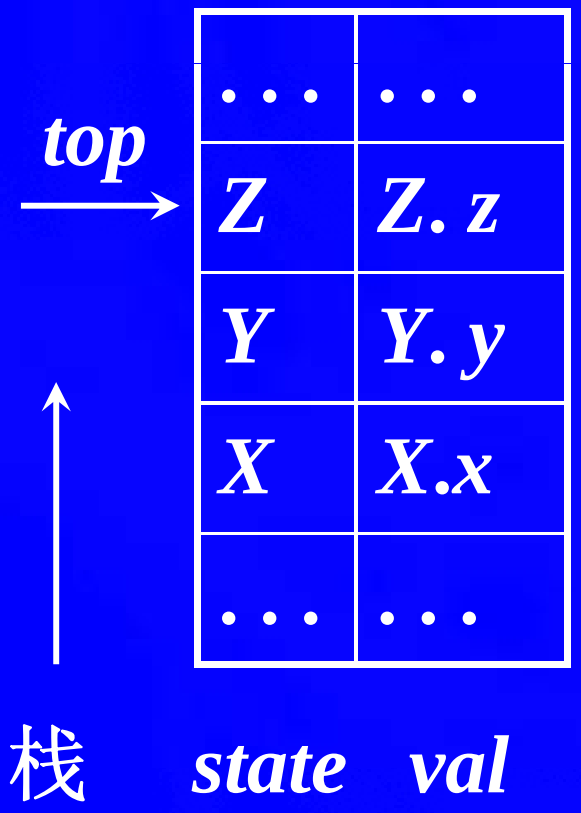
台式计算器的语法制导定义改成栈操作代码



产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val* [*top*–1])
$E \rightarrow E_1 + T$	*val* [*top* –2] = *val* [*top* –2]+*val* [*top*]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val* [*top* –2] = *val* [*top* –2]×*val* [*top*]
$T \rightarrow F$	
$F \rightarrow (E)$	*F.val* = *E.val*
$F \rightarrow$ digit	*F.val* = digit.*lexval*

4.1 语法制导的翻译

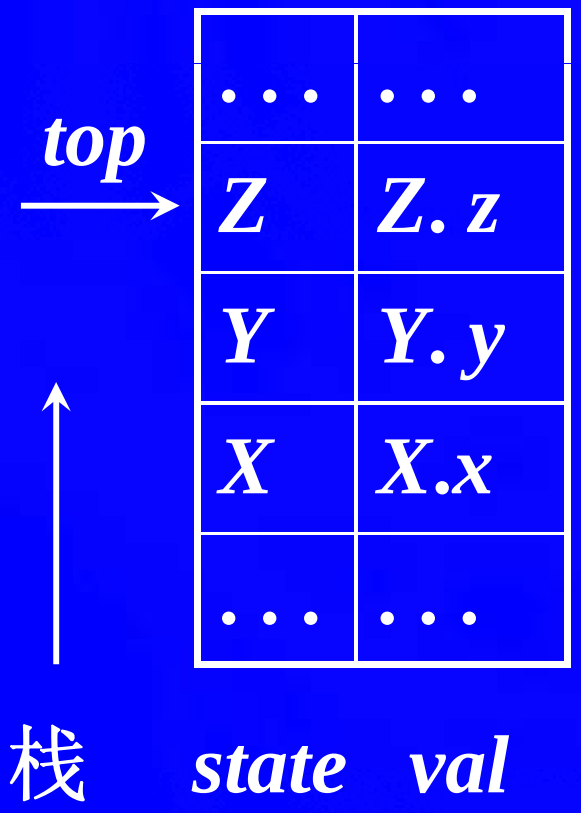
台式计算器的语法制导定义改成栈操作代码



产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val* [*top*–1])
$E \rightarrow E_1 + T$	*val* [*top* –2] = *val* [*top* –2]+*val* [*top*]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val* [*top* –2] = *val* [*top* –2]×*val* [*top*]
$T \rightarrow F$	
$F \rightarrow (E)$	*val* [top –2] = *val* [top –1]
$F \rightarrow$ digit	*F.val* = digit.*lexval*

4.1 语法制导的翻译

台式计算器的语法制导定义改成栈操作代码



产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val* [*top*–1])
$E \rightarrow E_1 + T$	*val* [*top* –2] = *val* [*top* –2]+*val* [*top*]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val* [*top* –2] = *val* [*top* –2]×*val* [*top*]
$T \rightarrow F$	
$F \rightarrow (E)$	*val* [top –2] = *val* [top –1]
$F \rightarrow$ digit	

4.1 语法制导的翻译

4.1.5 设计翻译方案的一些技巧

- 先前设计的语法制导翻译方案的局限
  - 除非是有副作用的语义动作，否则语义动作只能计算非终结符的综合属性
- 综合属性

  非终结符的属性由它本身推出的部分来确定
  - 实际编程语言会出现：

    语言构造的属性并不依赖于本身推出的部分

    需要用继承属性来描述

4.1 语法制导的翻译

- 继承属性

  其值由它的上下文决定

  – 例

  Pascal语言的变量声明，如**m, n : integer**

  $D \rightarrow L : T$

  $T \rightarrow$ **integer | char**

  $L \rightarrow L$, **id | id**

  L需要一个继承属性，其值由它的上下文决定

  – 先前介绍的翻译方案不具备计算继承属性的能力

4.1 语法制导的翻译

- 解决办法

  用综合属性代替继承属性

Pascal的声明，如m, n : integer

$D \rightarrow L : T$

$T \rightarrow$ **integer** | **char**

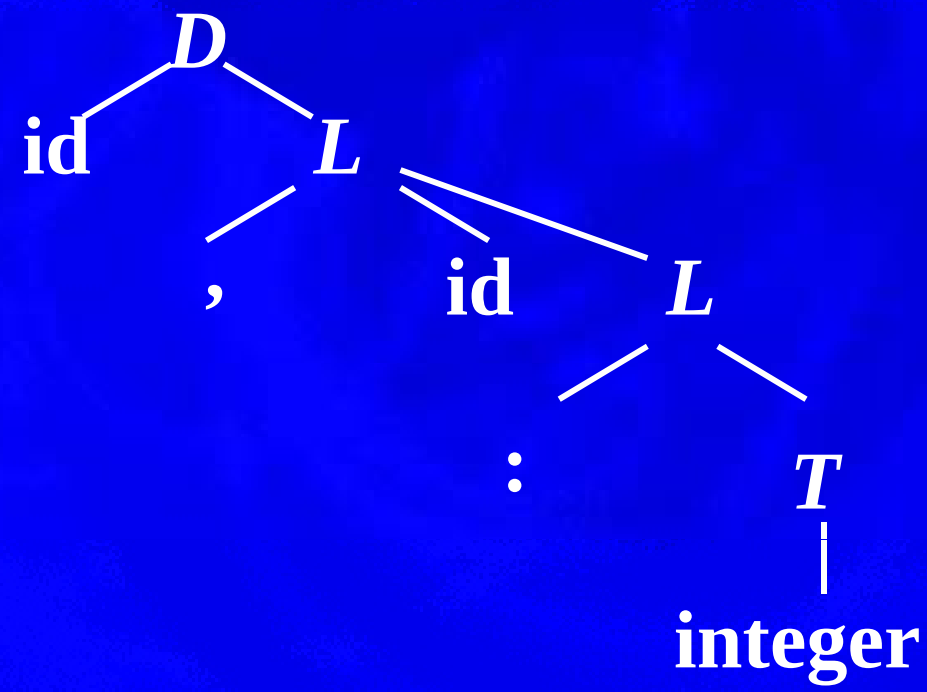
$L \rightarrow L$, **id** | **id**

改成从右向左归约

$D \rightarrow$ **id** L

$L \rightarrow$, **id** L | : T

$T \rightarrow$ **integer** | **char**



4.1 语法制导的翻译

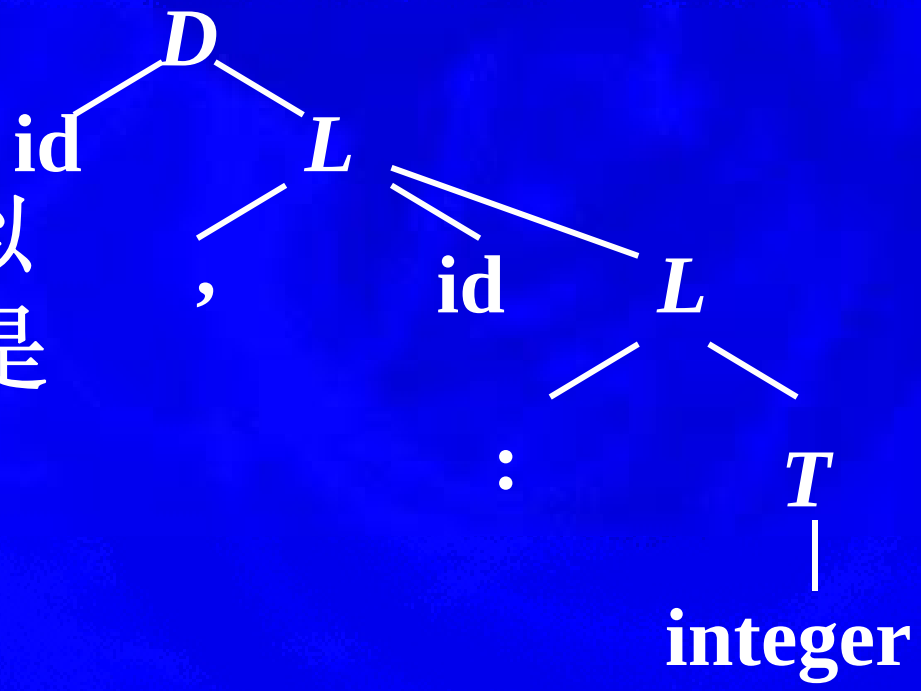
$D \rightarrow$ id L { *addType* (id. *entry*, *L.type*)}

$L \rightarrow$, id L_1 {*L. type* = L_1.*type*;
addType (id. *entry*, L_1.*type*)}

$L \rightarrow$: T {*L. type* = *T. type*}

$T \rightarrow$ integer {*T. type* = integer}

$T \rightarrow$ char {*T. type* = char}



- 设计翻译方案时遇到类似问题时的另一种解决办法是使用全局变量

4.2 类型在编程语言中的作用

4.2.1 执行错误和安全语言

介绍一些和程序运行有联系的概念

4.2 类型在编程语言中的作用

4.2.1 执行错误和安全语言

1、程序运行时的执行错误分成两类

- 会被捕获的错误（***trapped error***）

4.2 类型在编程语言中的作用

4.2.1 执行错误和安全语言

1、程序运行时的执行错误分成两类

- 会被捕获的错误（***trapped error***）
  - 例：非法指令错误、非法内存访问、除数为零

4.2 类型在编程语言中的作用

4.2.1 执行错误和安全语言

1、程序运行时的执行错误分成两类

- 会被捕获的错误（***trapped error***）
  - 例：非法指令错误、非法内存访问、除数为零
  - 引起计算立即停止

4.2 类型在编程语言中的作用

4.2.1 执行错误和安全语言

1、程序运行时的执行错误分成两类

- 会被捕获的错误（***trapped error***）
  - 例：非法指令错误、非法内存访问、除数为零
  - 引起计算立即停止
- 不会被捕获的错误（***untrapped error***）

4.2 类型在编程语言中的作用

4.2.1 执行错误和安全语言

1、程序运行时的执行错误分成两类

- 会被捕获的错误（***trapped error***）
  - 例：非法指令错误、非法内存访问、除数为零
  - 引起计算立即停止
- 不会被捕获的错误（***untrapped error***）
  - 例：下标变量的访问越过了数组的末端

4.2 类型在编程语言中的作用

4.2.1 执行错误和安全语言

1、程序运行时的执行错误分成两类

- 会被捕获的错误（***trapped error***）
  - 例：非法指令错误、非法内存访问、除数为零
  - 引起计算立即停止
- 不会被捕获的错误（***untrapped error***）
  - 例：下标变量的访问越过了数组的末端
  - 例：跳到一个错误的地址，该地址开始的内存正好代表一个指令序列

4.2 类型在编程语言中的作用

4.2.1 执行错误和安全语言

1、程序运行时的执行错误分成两类

- 会被捕获的错误（***trapped error***）
  - 例：非法指令错误、非法内存访问、除数为零
  - 引起计算立即停止
- 不会被捕获的错误（***untrapped error***）
  - 例：下标变量的访问越过了数组的末端
  - 例：跳到一个错误的地址，该地址开始的内存正好代表一个指令序列
  - 错误可能会有一段时间未引起注意

4.2 类型在编程语言中的作用

4.2.1 执行错误和安全语言

2、良行为的程序

- 不同场合对良行为的定义略有区别
- 例如，没有任何不会被捕获的错误的程序

3、安全语言

- 任何合法程序都是良行为的
- 通常是设计一个类型系统，通过静态检查来拒绝不会被捕获的错误
- 但是，设计一个类型系统，它正好只拒绝不会被捕获的错误是非常困难的

4.2 类型在编程语言中的作用

4.2.1 执行错误和安全语言

- 禁止错误（***forbidden error***）
  - 所有不会被捕获的错误 + 部分会被捕获的错误
  - 为语言设计类型系统的目标是在排除禁止错误

良行为程序和安全语言也可以基于禁止错误来定义

4.2 类型在编程语言中的作用

4.2.2 类型化语言和类型系统

4、类型化的语言

- 变量的类型
  - 变量在程序执行期间的取值范围

4.2 类型在编程语言中的作用

4.2.2 类型化语言和类型系统

4、类型化的语言

- 变量的类型
- 类型化的语言
  - 变量都被给定类型的语言
  - 表达式、语句等程序构造的类型都可以静态确定
  - 例如， ***boolean*** 类型的变量 ***x*** 在程序每次运行时的值只能是布尔值，***not*** **(*x*)**总有意义

4.2 类型在编程语言中的作用

4.2.2 类型化语言和类型系统

4、类型化的语言

- 变量的类型
- 类型化的语言
- 未类型化的语言
  - 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加定义的结果
  - 例如：**LISP**语言

4.2 类型在编程语言中的作用

4.2.2 类型化语言和类型系统

4、类型化的语言

- 变量的类型
- 类型化的语言
- 未类型化的语言
- 显式类型化语言
  - 类型是语法的一部分

4.2 类型在编程语言中的作用

4.2.2 类型化语言和类型系统

4、类型化的语言

- 变量的类型
- 类型化的语言
- 未类型化的语言
- 显式类型化语言
- 隐式类型化的语言
  - 不存在隐式类型化的主流语言，但可能存在忽略类型信息的程序片段，例如不需要程序员声明函数的参数类型

4.2 类型在编程语言中的作用

4.2.2 类型化语言和类型系统

5、类型系统

- 语言的组成部分, 由一组定型规则（***typing rule***）构成，这组规则用来给各种程序构造指派类型
- 设计类型系统的根本目的是用静态检查的方式来保证合法程序运行时的良行为
- 类型系统的形式化
  - 类型表达式、定型断言、定型规则
- 类型检查算法
  - 通常是静态地完成类型检查

4.2 类型在编程语言中的作用

4.2.2 类型化语言和类型系统

6、良类型的程序

- 没有类型错误的程序

7、合法程序

- 良类型程序（若语言定义中，除了类型系统外，没有用其他方式表示对程序的约束）

8、类型可靠（*type sound*）的语言

- 所有良类型程序（合法程序）都是良行为的
- 类型可靠的语言一定是安全的语言

4.2 类型在编程语言中的作用

4.2.2 类型化语言和类型系统

语法的和静态的概念

类型化语言

良类型程序

动态的概念

安全语言

良行为的程序

4.2 类型在编程语言中的作用

4.2.2 类型化语言和类型系统

9、类型检查：未类型化的语言

- 可以通过彻底的运行时详细检查来排除所有的禁止错误
  - 如LISP语言

10、类型检查：类型化语言

- 类型检查也可以放在运行时完成，但影响效率
- 一般都是静态检查，类型系统被用来支持静态检查
- 静态检查语言通常也需要一些运行时的检查
  - 数组访问越界检查

4.2 类型在编程语言中的作用

4.2.2 类型化语言和类型系统

实际使用的一些语言并不安全

- 禁止错误集合没有囊括所有不会被捕获的错误

4.2 类型在编程语言中的作用

4.2.2 类型化语言和类型系统

实际使用的一些语言并不安全

- 禁止错误集合没有囊括所有不会被捕获的错误
- 用C语言的共用体（***union***）来举例

```
union U { int u1; int *u2;} u;
int *p;
u.u1 = 10;
p = u.u2;
*p = 0;
```

4.2 类型在编程语言中的作用

4.2.2 类型化语言和类型系统

实际使用的一些语言并不安全

- C语言
  - 还有很多不安全的并且被广泛使用的特征，如：
    指针算术运算、类型强制、参数个数可变
  - 在语言设计的历史上，安全性考虑不足是因为当时强调代码的执行效率
- 在现代语言设计上，安全性的位置越来越重要
  - C 语言的一些问题已经在C++ 语言中得以缓和
  - 更多一些问题在Java 语言中已得到解决

4.2 类型在编程语言中的作用

4.2.3 类型化语言的优点

从工程的观点看，类型化语言有下面一些优点

- 开发的实惠
  - 较早发现错误
  - 类型信息还具有文档作用
- 编译的实惠
  - 程序模块可以相互独立地编译
- 运行的实惠
  - 可得到更有效的空间安排和访问方式

4.3 一个简单类型检查器的规范

4.3.1 一个简单的语言

$P \rightarrow D\ ;\ S$

$D \rightarrow D\ ;\ D \mid \mathbf{id}\ :\ T$

$T \rightarrow \mathbf{boolean} \mid \mathbf{integer} \mid \mathbf{array}\ [\mathbf{num}\ ]\ \mathbf{of}\ T \mid$
$\uparrow T \mid T\ \text{`}\rightarrow\text{'}\ T$

$S \rightarrow \mathbf{id}\ :=\ E \mid \mathbf{if}\ E\ \mathbf{then}\ S \mid \mathbf{while}\ E\ \mathbf{do}\ S \mid S\ ;\ S$

$E \rightarrow \mathbf{truth} \mid \mathbf{num} \mid \mathbf{id} \mid E\ \mathbf{mod}\ E \mid E\ [\ E\ ] \mid$
$E\uparrow \mid E\ (E\ )$

4.3 一个简单类型检查器的规范

- 例

i : integer;

j : integer;

j := i mod 2000

4.3 一个简单类型检查器的规范

4.3.2 类型表达式

- 语法树形式的类型表达式的文法
  - 简单类型
  - 构造类型：类型构造符作用于若干个子类型

type_exp → ***type_error*** | **void** | ***valid_type_exp***

valid_type_exp → **integer** | **boolean** |
***array*(*N*, *valid_type_exp*)** | ***pointer*(*valid_type_exp*)**|
valid_type_exp* ‘→’ *valid_type_exp

***type_error*：** 出现类型错误的语法构造的类型

void： 没有类型错误的语句的类型

*N*是正整数，表示数组的大小

4.3 一个简单类型检查器的规范

4.3.3 类型检查——声明语句

$D \rightarrow D ; D$

$D \rightarrow \mathbf{id} : T$ {*addType* (id.*entry*, *T.type*);}

addType：把类型信息填入符号表

4.3 一个简单类型检查器的规范

4.3.3 类型检查——声明语句

$D \rightarrow D ; D$

$D \rightarrow \text{id} : T$ {*addType* (id.*entry*, *T*.*type*);}

$T \rightarrow \text{boolean}$ {*T*.*type* = boolean;}

$T \rightarrow \text{integer}$ {*T*.*type* = integer;}

$T \rightarrow \uparrow T_1$ {*T*.*type* = *pointer*(T_1.*type*);}

4.3 一个简单类型检查器的规范

4.3.3 类型检查——声明语句

$D \rightarrow D; D$

$D \rightarrow$ id : T {*addType* (id.*entry*, T.*type*);}

$T \rightarrow$ boolean {T.*type* = boolean;}

$T \rightarrow$ integer {T.*type* = integer;}

$T \rightarrow \uparrow T_1$ {T.*type* = *pointer*(T_1.*type*);}

$T \rightarrow$ array [num] of T_1

{T.*type* = *array*(num.*val*, T_1.*type*);}

4.3 一个简单类型检查器的规范

4.3.3 类型检查——声明语句

$D \rightarrow D; D$

$D \rightarrow$ id : T {*addType* (id.*entry*, T.*type*);}

$T \rightarrow$ boolean {T.*type* = boolean;}

$T \rightarrow$ integer {T.*type* = integer;}

$T \rightarrow \uparrow T_1$ {T.*type* = *pointer*(T_1.*type*);}

$T \rightarrow$ array [num] of T_1

{T.*type* = *array*(num.*val*, T_1.*type*);}

$T \rightarrow T_1$ '$\rightarrow$' T_2 {T.*type* = T_1.*type* $\rightarrow$ T_2.*type*; }

4.3 一个简单类型检查器的规范

4.3.3 类型检查——表达式

$E \rightarrow$ **truth** {*E.type* = boolean; }

$E \rightarrow$ **num** {*E.type* = integer;}

$E \rightarrow$ **id** {*E.type* = *lookup*(id.*entry*);}

4.3 一个简单类型检查器的规范

4.3.3 类型检查——表达式

$E \rightarrow$ **truth** {*E.type* = **boolean**; }

$E \rightarrow$ **num** {*E.type* = **integer**;}

$E \rightarrow$ **id** {*E.type* = *lookup*(**id**.*entry*);}

$E \rightarrow E_1$ **mod** E_2

{**if** (*E*$_1$.*type* == **integer** && *E*$_2$. *type* == **integer**)

E.type = **integer**;

else *E.type* = *type_error*; }

4.3 一个简单类型检查器的规范

4.3.3 类型检查——表达式

$E \rightarrow E_1\ [E_2\ ]$ { if (E_2. *type* == integer &&
E_1. *type* == *array*(*s*, *t*))
E.type = *t*;
else *E.type* = *type_error*;}

4.3 一个简单类型检查器的规范

4.3.3 类型检查——表达式

$E \rightarrow E_1\,[E_2\,]$ { if (E_2. *type* == integer &&
E_1. *type* == *array*(*s*, *t*)) *E.type* = *t*;
else *E.type* = *type_error*;}

$E \rightarrow E_1\uparrow$ { if (E_1.*type* == *pointer*(*t*)) *E.type* = *t*;
else *E.type* = *type_error*;}

4.3 一个简单类型检查器的规范

4.3.3 类型检查——表达式

$E \rightarrow E_1\,[E_2\,]$ { if (E_2. *type* == integer &&
E_1. *type* == *array*(*s*, *t*)) *E.type* = *t*;
else *E.type* = *type_error*;}

$E \rightarrow E_1\uparrow$ { if (E_1.*type* == *pointer*(*t*)) *E.type* = *t*;
else *E.type* = *type_error*;}

$E \rightarrow E_1\,(E_2\,)$ { if (E_2.*type* ==*s* && E_1.*type* == $s \rightarrow t$)
E. *type* = *t* ;
else *E*. *type* = *type_error*;}

4.3 一个简单类型检查器的规范

4.3.3 类型检查——语句

$S \rightarrow$ **id** := E { **if** (*id.type* == *E.type* && *E.type* $\in$ {**boolean, integer**}) *S.type* = **void**;

else *S.type* = *type_error*;}

4.3 一个简单类型检查器的规范

4.3.3 类型检查——语句

$S \rightarrow$ **id** := E { **if** (*id.type* == *E.type* && *E.type* $\in$ {**boolean, integer**}) *S.type* = **void**;

else *S.type* = *type_error*;}

$S \rightarrow$ **if** E **then** S_1 { **if** (*E. type* == **boolean**)

S. type = S_1. *type*;

else *S. type* = *type_error*; }

4.3 一个简单类型检查器的规范

4.3.3 类型检查——语句

$S \rightarrow$ **while** E **do** S_1

{ if (*E.type* == boolean) *S.type* = S_1. *type*;

else *S.type* = *type_error*; }

4.3 一个简单类型检查器的规范

4.3.3 类型检查——语句

$S \rightarrow$ **while** E **do** S_1

{ if (*E.type* == boolean) *S.type* = S_1. *type*;

else *S.type* = *type_error*; }

$S \rightarrow S_1; S_2$

{ if (S_1. *type* == void && S_2.*type* == void)

S. *type* = void;

else *S. type* = *type_error*; }

4.3 一个简单类型检查器的规范

4.3.4 类型转换

- 类型转换的根源
  - 数学上可以一起运算的数据，例如整数和实数，因不同的内部表示而分属不同的类型
- 语言设计
  - 决定采用隐式还是显式转换
- 类型系统
  - 指明允许哪些不同类型的运算以及结果类型
- 语言实现
  - 安排插入类型转换操作

4.3 一个简单类型检查器的规范

4.3.4 类型转换

$E \rightarrow E_1$ op E_2

```
{  if  (E1.type == integer && E2.type == integer)
          E.type  = integer;
   else if (E1.type == integer && E2.type == real)
          E.type  = real;
   else if (E1.type == real && E2.type == integer)
          E.type  = real;
   else if (E1.type == real && E2.type == real )
          E.type  = real;
   else E.type  = type_error; }
```

4.4 类型表达式的等价

当允许对类型表达式命名后

- 类型表达式是否相同就有了不同的解释
- 出现了结构等价和名字等价两个不同的概念

```
type link = ↑cell;
var   next  : link;
      last  : link;
      p     : ↑cell;
      q, r  : ↑cell;
```

4.4 类型表达式的等价

4.4.1 类型表达式的结构等价

- 两个类型表达式完全相同（当无类型名时）

```
type link = ↑cell;
var   next  : link;
      last  : link;
      p     : ↑cell;
      q, r  : ↑cell;
```

4.4 类型表达式的等价

4.4.1 类型表达式的结构等价

- 两个类型表达式完全相同（当无类型名时）
  - 类型表达式树一样

```
type link = ↑cell;
var   next  : link;
      last  : link;
      p     : ↑cell;
      q, r  : ↑cell;
```

4.4 类型表达式的等价

4.4.1 类型表达式的结构等价

- 两个类型表达式完全相同（当无类型名时）
  - 类型表达式树一样
  - 相同的类型构造符作用于相同的子表达式

```
type link = ↑cell;
var   next  : link;
      last  : link;
      p     : ↑cell;
      q, r  : ↑cell;
```

4.4 类型表达式的等价

4.4.1 类型表达式的结构等价

- 两个类型表达式完全相同（当无类型名时）
- 有类型名时，用它们所定义的类型表达式代换它们，所得表达式完全相同(类型定义无环时)

```
type link = ↑cell;
var   next  : link;
      last  : link;
      p     : ↑cell;
      q, r  : ↑cell;
```

next, last, p, q和r结构等价

4.4 类型表达式的等价

类型表达式的结构等价测试**sequiv(*s*, *t*)** (无类型名时)

if (*s* 和 *t* 是相同的基本类型)

return true;

else if (*s* == *array*(s_1, s_2) && *t* == *array*(t_1, t_2))

return s_1== t_1 && sequiv(s_2, t_2);

else if (*s* == $s_1 \times s_2$ && *t* == $t_1 \times t_2$)

return sequiv(s_1, t_1) && sequiv(s_2, t_2);

else if (*s* == *pointer* (s_1) && *t* == *pointer*(t_1))

return sequiv(s_1, t_1);

else if (s == $s_1 \rightarrow s_2$ && t == $t_1 \rightarrow t_2$)

return squiv(s_1, t_1) && sequiv(s_2, t_2);

else return false;

4.4 类型表达式的等价

4.4.2 类型表达式的名字等价

- 把每个类型名看成是一个可区别的类型

```
type link = ↑cell;
var   next  : link;
      last  : link;
      p     : ↑cell;
      q, r  : ↑cell;
```

4.4 类型表达式的等价

4.4.2 类型表达式的名字等价

- 把每个类型名看成是一个可区别的类型
- 两个类型表达式名字等价当且仅当这两个类型表达式不做名字代换就结构等价

```
type link = ↑cell;
var   next  : link;
      last  : link;
      p     : ↑cell;
      q, r  : ↑cell;
```

next和last名字等价

p, q和r名字等价

4.4 类型表达式的等价

Pascal语言的许多实现用隐含的类型名和每个声明的标识符联系起来

```
type link = ↑cell;
var  next : link;
     last : link;
     p    : ↑cell;
     q, r : ↑cell;
```

```
type link = ↑cell;
     np = ↑cell;
     nqr = ↑cell;
var next : link;
     last : link;
     p : np;
     q : nqr;
     r : nqr;
```

这时，p与q和r不是名字等价

4.4 类型表达式的等价

注意：

类型名字的引入只是类型表达式的一个语法约定问题，它并不像引入类型构造符或类型变量那样能丰富所能表达的类型

4.4 类型表达式的等价

4.4.3 记录类型

– 记录类型可看成其各个域类型的积类型

– 记录和积之间的主要区别是记录的域被命名

- 例如，C语言的记录类型

```
typedef struct {
        int address;
        char lexeme [15 ];
}row;
```

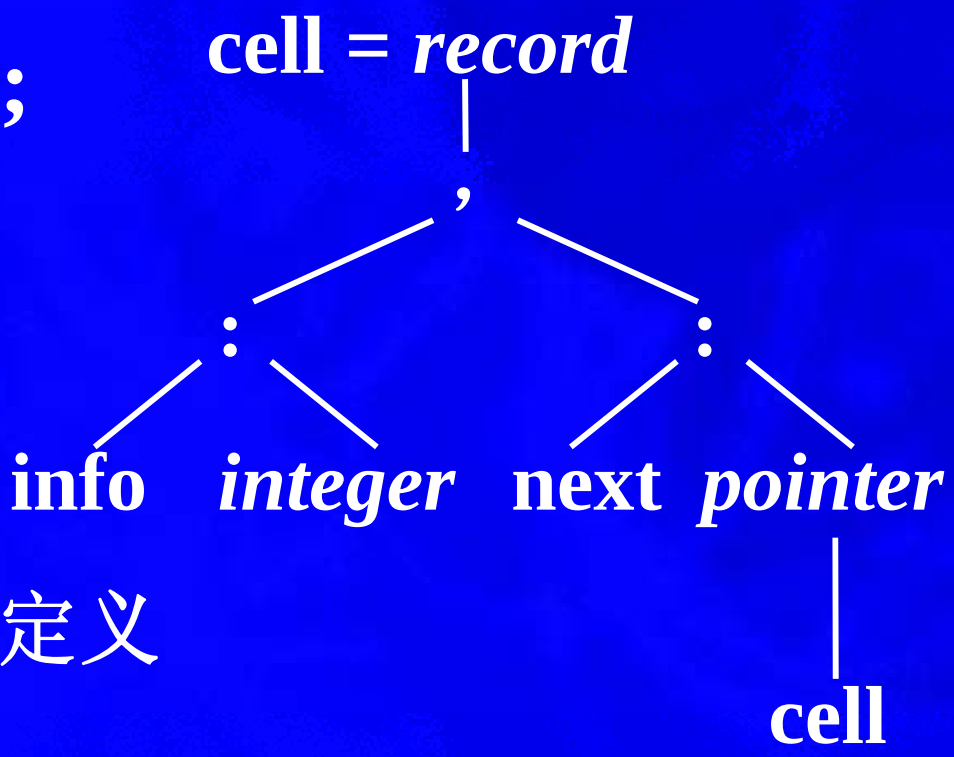
的类型表达式是

record(address : int, lexeme : ***array***(15, char))

4.4 类型表达式的等价

4.4.4 类型表示中的环

```
type link = ↑ cell ;
cell = record
        info : integer ;
        next : link
end;
```

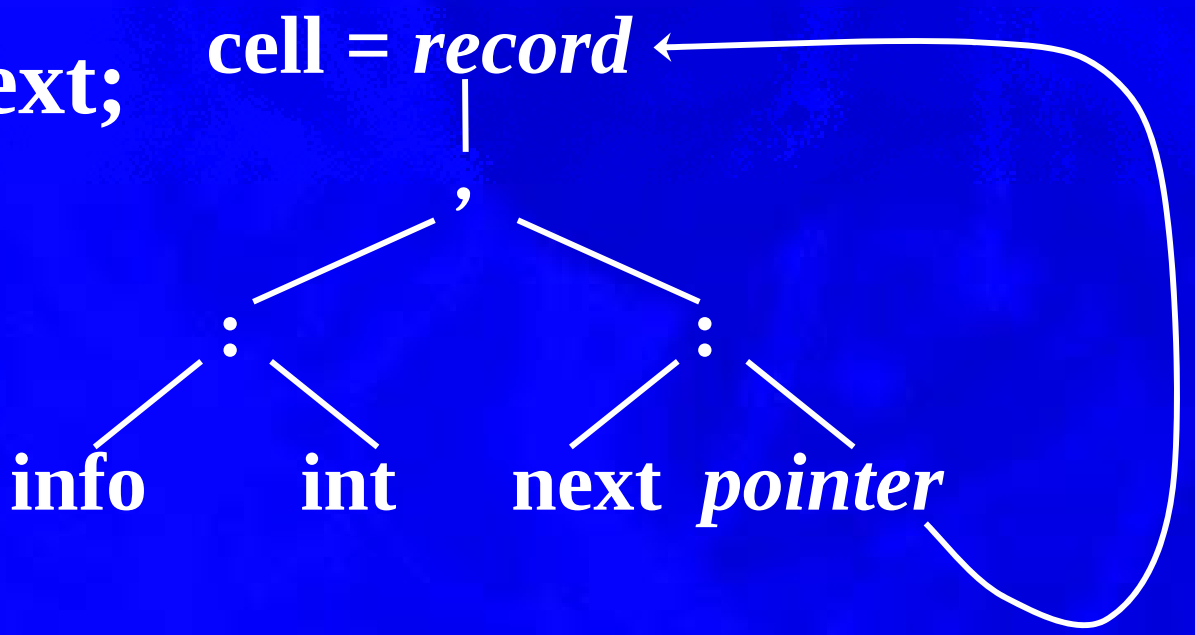


引入环的话，递归定义的类型名可以替换掉

4.4 类型表达式的等价

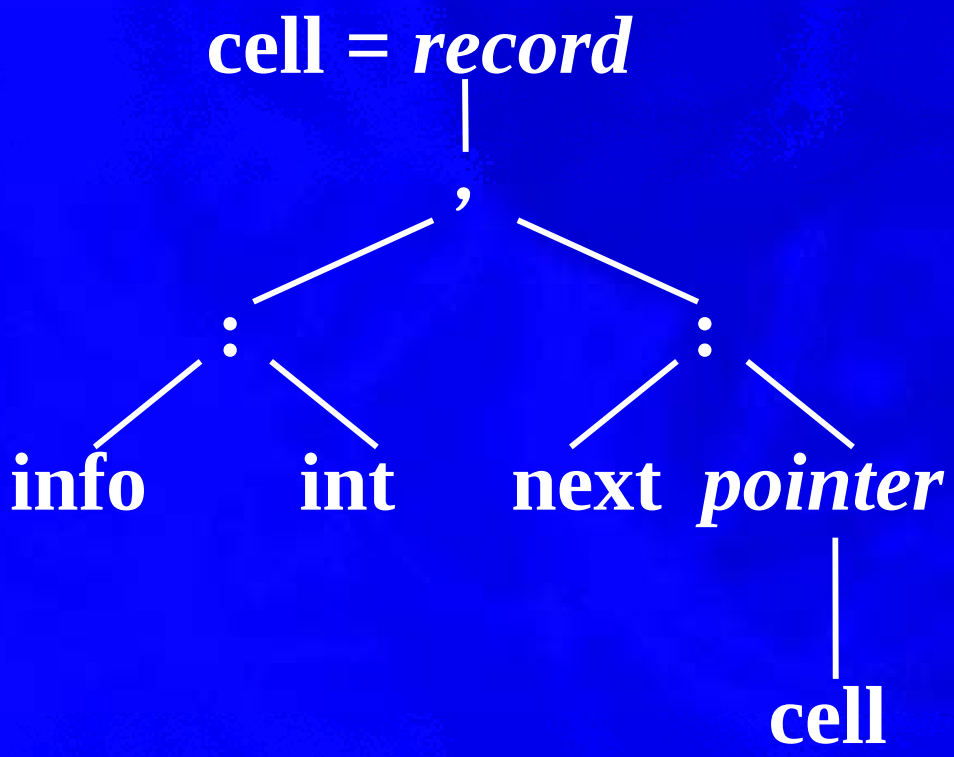
4.4.4 类型表示中的环

```
typedef struct cell {
        int info;
        struct cell *next;
} cell;
```



4.4 类型表达式的等价

C语言对除记录（结构体）以外的所有类型使用结构等价，而对记录类型用的是名字等价，以避免类型图中的环



本 章 要 点

- 设计简单问题的语法制导翻译方案
- 理解语法制导翻译的自下而上计算
- 静态检查所涉及的内容：类型检查、控制流检查、唯一性检查和关联名字检查等
- 类型在编程语言中的作用
- 类型系统和类型化语言
- 设计简单语言类型检查的翻译方案
- 理解类型表达式的结构等价和名字等价

例　题　1

编译时的控制流检查的例子

```
main( )
{
        printf(“\n%ld\n”, gcd(4,12));
        continue;
}
```

编译时的报错如下:

```
continue.c:  In function ‘main’:
continue.c:4: continue statement not within a loop
```

例　题　2

编译时的唯一性检查的例子

```
main( )
{
    int i;
    switch(i){
        case 10: printf(“%d\n”, 10); break;
        case 20: printf(“%d\n”, 20); break;
        case 10: printf(“%d\n”, 10); break;
    }
}
```

编译时的报错如下：

```
switch.c: In function ‘main’:
switch.c:7: duplicate case value
switch.c:5: this is the first entry for that value
```

例　题　3

下面是产生字母表Σ = {0, 1, 2}上数字串的一个文法

$S \rightarrow D\ S\ D \mid 2$

$D \rightarrow 0 \mid 1$

写一个翻译方案，判断它接受的句子是否为回文数

$S' \rightarrow S$	{ print(S.*val*); }
$S \rightarrow D_1\ S_1\ D_2$	{ S.*val* = (D_1.*val* == D_2.*val*) && S_1.*val*; }
$S \rightarrow 2$	{ S.*val* = true; }
$D \rightarrow 0$	{ D.*val* = 0; }
$D \rightarrow 1$	{ D.*val* = 1; }

例　题　4

为下面文法写一个翻译方案，用S的属性*val* 给出下面文法中S产生的二进制数的值。

例如，输入**101.101**时，

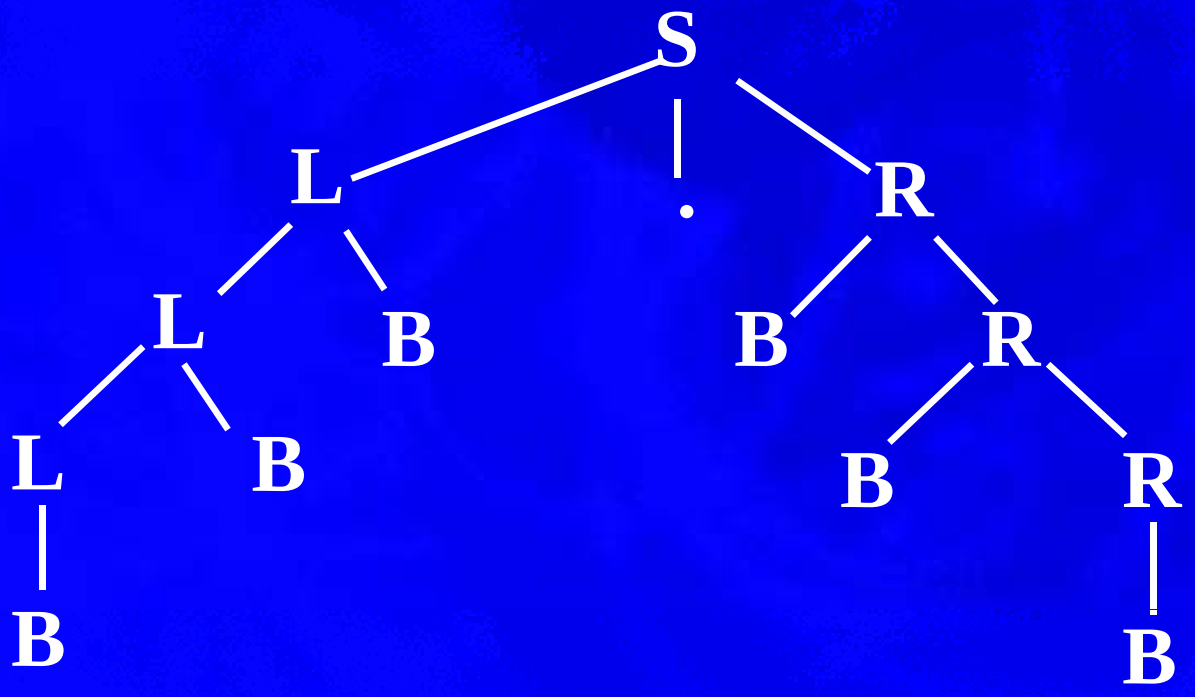
S.*val* = 5.625

S → L . R | L

L → L B | B

R → B R | B

B → 0 | 1



例　题　4

$S \rightarrow L\ .\ R$	{ S. *val* = L. *val* + R. *val*; }
$S \rightarrow L$	{ S. *val* = L. *val*; }
$L \rightarrow L_1\ B$	{ L. *val* = L_1. *val* *2 + B. *val*; }
$L \rightarrow B$	{ L. *val* = B. *val*; }
$R \rightarrow B\ R_1$	{ R. *val* = R_1. *val* / 2 + B. *val* / 2;}
$R \rightarrow B$	{ R. *val* = B. *val* / 2; }
$B \rightarrow 0$	{ B. *val* = 0; }
$B \rightarrow 1$	{ B. *val* = 1; }

例　题　5

C语言

- 称&为地址运算符，&a为变量a的地址
- 数组名代表数组第一个元素的地址

问题：

如果*a*是一个数组名，那么表达式*a*和&*a*的值都是数组*a*第一个元素的地址，它们的使用是否有区别的？

用四个C程序的编译报错或运行结果来提示

例　题　5

```
typedef  int  A[10][20];
A  a;

A  *fun( ) {
   return(a);
}
```

该函数在Linux上用gcc编译，报告的错误如下：

第 5 行： warning: return from incompatible pointer type

例　题　5

```
typedef  int  A[10][20];
A  a;

A  *fun( ) {
   return(&a);
}
```

该函数在Linux上用gcc编译时，没有错误

例　题　5

```
typedef  int  A[10][20];
typedef  int  B[20];
A  a;

B  *fun( ) {
   return(a);
}
```

该函数在Linux上用gcc编译时，没有错误

例　题　5

```
typedef  int  A[10][20];
A  a;

fun( ) { printf("%d,%d,%d\n", a, a+1, &a+1);}

main( ) { fun( ); }
```

该程序的运行结果是：

134518112, 134518192, 134518912

例　题　5

结论

对一个元素为t类型的数组$a[\,i_1\,][\,i_2\,]\ldots[\,i_n\,]$来说,

表达式a的类型是:

pointer(array(0.. i_2 –1, … array(0.. i_n –1, t)…))

表达式&a的类型是:

pointer(array(0.. i_1 –1, … array(0.. i_n –1, t)…))

例　题　6

在x86/Linux机器上，编译器报告最后一行有错误：**incompatible types in return**

```
typedef int A1[10];                | A2 *fun1( )
typedef int A2[10];                | { return(&a); }
A1 a;                              |
typedef struct {int i;}S1;         | S2  fun2( )
typedef struct {int i;}S2;         | { return(s); }
S1 s;
```

在C语言中，数组和结构体都是构造类型，为什么上面第2个函数有类型错误，而第1个函数却没有？

例　题　7

编译器和连接装配器没能发现函数调用错误

```
long gcd (p, q)      /* 这是参数声明的传统形式 */
long p, q;       /*现代形式是long gcd ( long p, long q) */
{
    if (p%q == 0)
        return q;
    else
        return gcd (q, p%q);
}
main( )
{ printf("%ld,%ld\n", gcd(5), gcd(5, 10, 20)); }
```

习　　题

- 第一次 **4.1，4.3，4.5，4.7**
- 第二次 **4.9，4.11，4.14**
- 第三次 **4.15，4.17**