第五章 语法制导的翻译 语义分析

《编译原理》

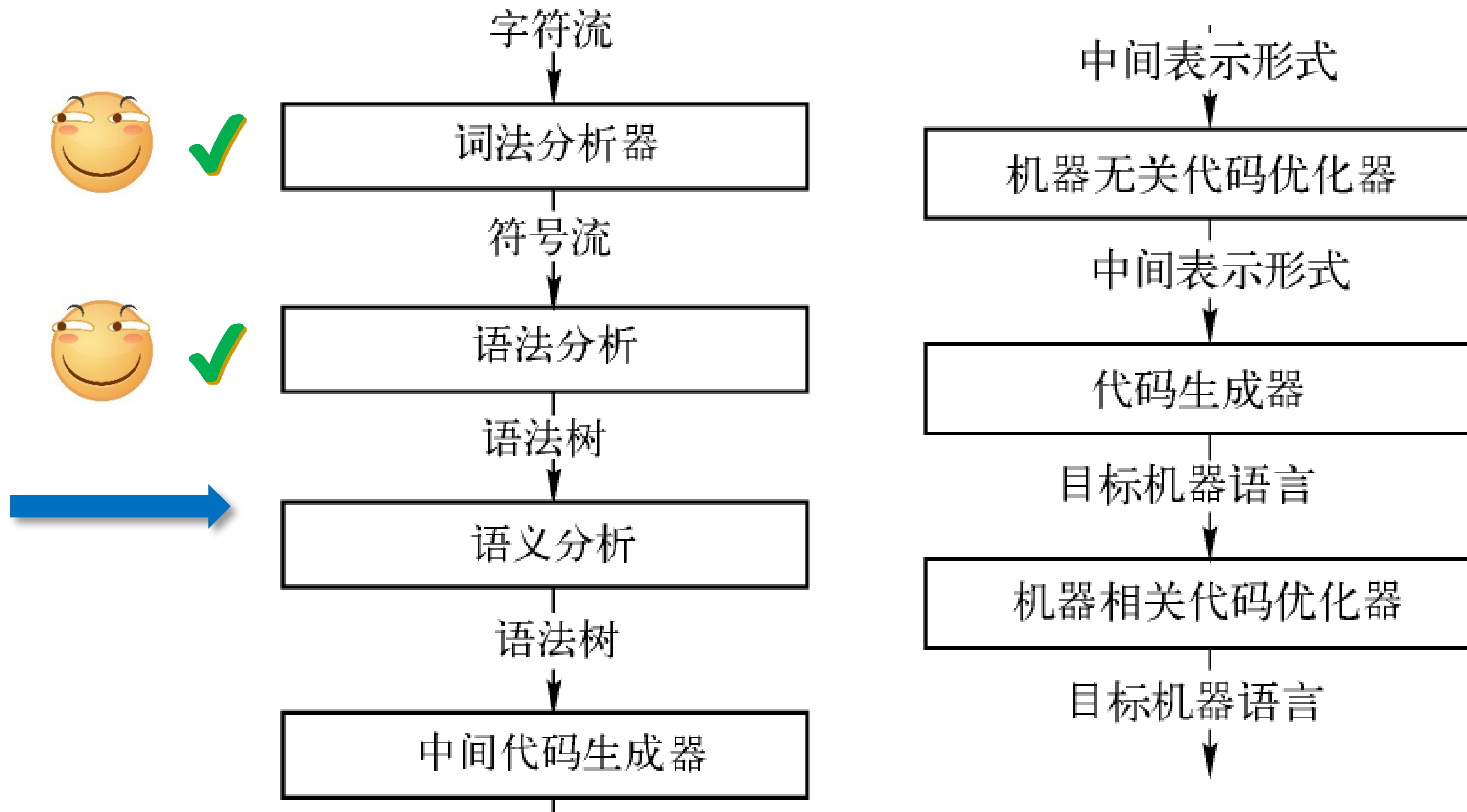
谭添

南京大学计算机系

2026年春季

版权所有 南京大学计算机科学与技术系 许畅 2026春季版（有修改）

闯关进度



目前得到的信息

- 输入程序通过了词法分析
  - 输入程序符合词法规则
  - 过滤了注释、空白字符等无用信息
  - 得到合法的符号流（合法的变量名、关键字…)
- 输入程序通过了语法分析
  - 输入程序符合语法规则
  - 得到合法的语法分析树

是否已经得到了合法的程序?

一个例子 (1)

```
int foo(int x) {
    int z;
    z = x + y;
    char s[] = “foo”;
    while (z > 0) {
        int n = x * s;
        --z;
    }
    return n;
}
```

一个例子 (2)

```
int foo(int x) {
    int z;
    z = x + y;
    char s[] = “foo”;
    while (z > 0) {
        int n = x * s;
        --z;
    }
    return n;
}
```

y未声明

整型与字符数组相乘，违反类型规则

在n的作用域之外无法访问n

- 符合词法、语法规则
- 违反语义规则（与具体语言设计相关）
  - 类型规则
    - 数值类型之间才能相乘
    - ……
  - 变量使用规则
    - 变量要先声明再使用
    - 变量要在作用域内使用
    - ……
  - ……

语义分析 (1)

- 检查程序是否符合语言的语义规则
  - 类型规则
  - 变量使用规则
  - 函数调用 (foo(…)) 规则
  - 属性访问 (x.f) 规则
  - ……

为什么语法分析技术不能检查语义？

- 语法分析基于CFG（Context-Free Grammar）上下文无关文法
- 语义分析往往与语法元素的上下文密切相关

```
int foo(int x) {
    int z;
    z = x + y;
    char s[] = “foo”;
    while (z > 0) {
        int n = x * s;
        --z;
    }
    return n;
}
```

完全相同的语句在不同上下文中可以具有不同语义

语义分析 (2)

- 检查程序是否符合语言的语义规则
  - 类型规则
  - 变量使用规则
  - 函数调用 (foo(…)) 规则
  - 属性访问 (x.f) 规则
  - ……

- 对于程序中的构造（函数定义、语句、表达式等）
  - 根据其对应的CFG文法结构赋予其语义
  - 检查语义是否符合语言的规则

文法与语义

- 使用上下文无关文法引导语义的分析
  - CFG的非终结符号代表了语言的某个构造 $E \rightarrow E + T$
  - 程序设计语言的构造由更小的构造组合而成
  - 一个构造的语义可以由小构造的含义综合而来
    - 比如：表达式$x + y$的类型由x、y的类型和运算符+决定
  - 也可以从附近的构造继承而来
    - 比如：声明int x中x的类型由它左边的类型表达式决定

语法制导的定义 (SDD)

- **Syntax-Directed Definition** (SDD) 是上下文无关文法和属性/规则的结合
  - 属性和文法符号相关联，按照需要来确定各个文法符号需要哪些属性
  - 规则和产生式相关联

SDD三要素：文法、属性、规则

属性可视为语义的具象化表示：类型、值、名字、地址……

- 对于文法符号X和属性*a*，我们用X.*a*表示分析树中某个标号为X的结点的值
  - 一个分析树结点和它的分支对应一个产生式规则，而对应的语义规则确定了这些结点上属性的取值和计算

基于SDD，我们可以在一个程序的分析树上进行语义信息的计算

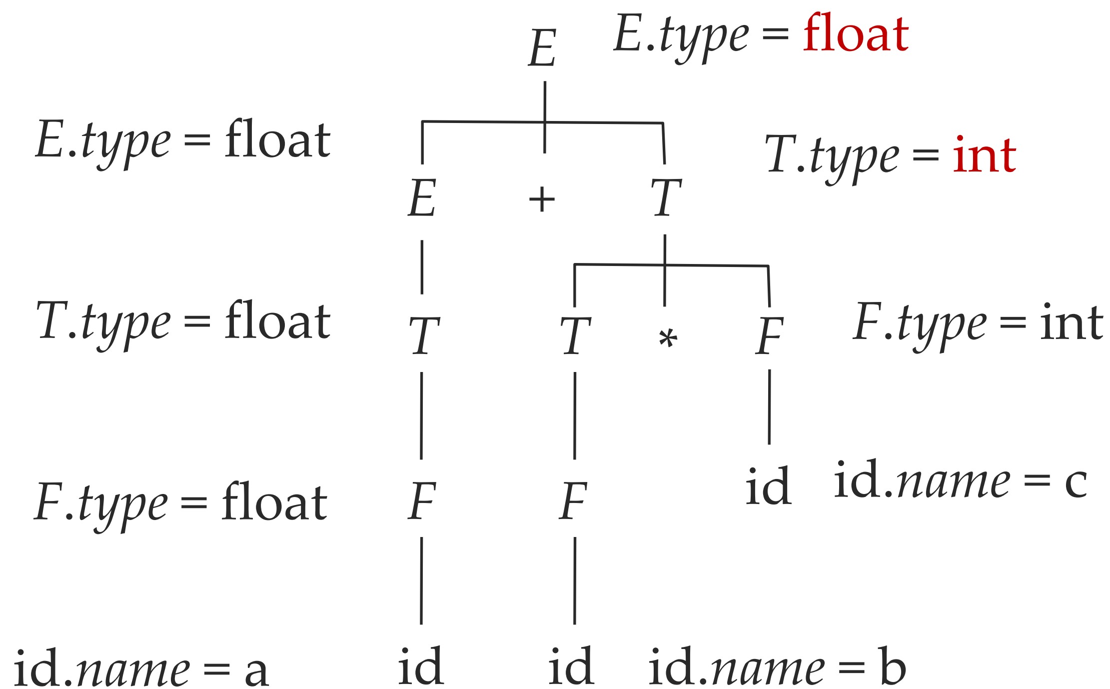
一个SDD的例子 (1)

- 假设需要知道一个表达式的类型
  - 我们需要属性*type*
- 产生式规则： $E \rightarrow E_1 + T$
  - 假设只有int/float类型
  - $E.type$ = if ($E_1.type$ == $T.type$) $T.type$ else float;
- 产生式规则： $E \rightarrow T$ $T \rightarrow F$
  - $E.type = T.type$; $T.type = F.type$;
- 产生式规则： $F \rightarrow$ id
  - $F.type$ = lookupIDTable(id.*name*)->*type*;

SDD三要素：文法、属性、规则

一个SDD的例子 (2)

- 输入a + b * c的语法分析树以及属性值



(1) 假设a, b, c是已声明的变量，a的类型为float，b和c的类型为int

(2) 中间未标明的T和F的$type$是int

- 产生式规则：$E \rightarrow E_1 + T$
  - $E.type$ = if ($E_1.type$ == $T.type$) $T.type$ else float;
- 产生式规则：$F \rightarrow$ id
  - $F.type$ = lookupIDTable(id.$name$)->$type$;
- 产生式规则：$E \rightarrow T$
  - $E.type = T.type$;
- 产生式规则：$T \rightarrow F$
  - $T.type = F.type$;

综合属性和继承属性

- **综合属性 (Synthesized Attribute)** 自底向上
  - 结点N的属性值由N的产生式所关联的语义规则来定义
  - 通过N的子结点或N本身的属性值来定义
- **继承属性 (Inherited Attribute)** 自顶向下
  - 结点N的属性值由N的父结点所关联的语义规则来定义
  - 依赖于N的父结点、N本身和N的兄弟结点上的属性值
- 几条约束
  - 不允许N的继承属性通过N的子结点上的属性来定义，但允许N的综合属性依赖于N本身的继承属性
  - 终结符号有综合属性 (来自词法分析)，但无继承属性

方便属性的定义

又一个SDD的例子

- 计算表达式行L的值 (属性val)
- 计算L的val值需要E的val值，E的val值又依赖于E_1和T的val值…
- 终结符号digit有综合属性lexval

产生式	语义规则
1) $L \to E$ **n**	$L.val = E.val$
2) $E \to E_1 + T$	$E.val = E_1.val + T.val$
3) $E \to T$	$E.val = T.val$
4) $T \to T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \to F$	$T.val = F.val$
6) $F \to (E)$	$F.val = E.val$
7) $F \to$ **digit**	$F.val =$ **digit**.lexval

S属性的SDD

- 只包含综合(Synthesized)属性的SDD称为**S属性**的SDD
  - 每个语义规则都根据产生式体中的属性值来计算头部非终结符号的属性值
- S属性的SDD可以和LR语法分析器一起实现
  - 栈中的状态/文法符号可以附加相应的属性值
  - 归约时，按照语义规则计算归约得到的符号的属性值
- 语义规则一般不应该有复杂的副作用
  - 要求副作用不影响其它属性的求值
  - 没有副作用的SDD称为**属性文法** (Attribute Grammar)

方便语义信息的计算，每次可专注于当前的产生式

语法分析树上的SDD求值 (1)

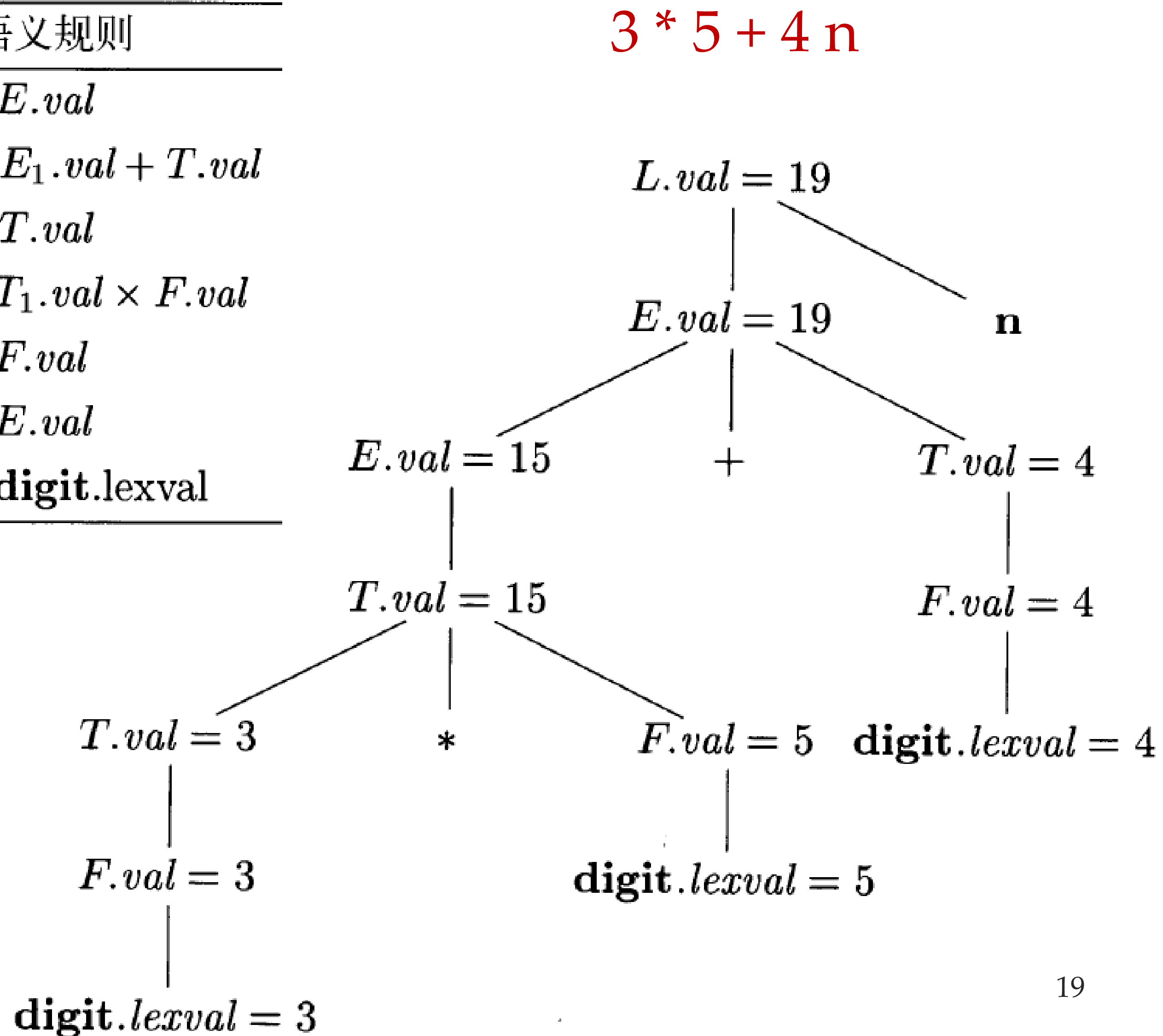
- 在分析树上求值有助于属性计算的可视化，便于理解
- **注释（Annotated）语法分析树**
  - 包含了各个结点的各属性值的语法分析树
- 步骤
  - 对于任意的输入串，首先构造出相应的分析树
  - 给各个结点 (根据其文法符号) 加上相应的属性
  - 按照语义规则计算这些属性的值

语法分析树上的SDD求值 (2)

- 按照分析树中的分支对应的文法产生式，应用相应的语义规则计算属性值
- 计算顺序
  - 如果某个结点N的属性a为$f(N_1.b_1, N_2.b_2, \ldots, N_k.b_k)$，那么我们需要先算出$N_1.b_1, N_2.b_2, \ldots, N_k.b_k$的值
- 如果可以给各个属性值排出计算顺序，那么这个注释分析树就可以计算得到
  - S属性的SDD一定可以按照自底向上的方式求值
- 下面的SDD不能计算
  - $A \rightarrow B$　　$A.s = B.i$　　$B.i = A.s + 1$

注释分析树的例子

	产生式	语义规则
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval



SDD的求值顺序

- 在对SDD的求值过程中
  - 如果结点N的属性a依赖于结点M_1的属性a_1，M_2的属性a_2，…那么我们必须先计算出M_i的属性a_i，才能计算N的属性a

$$N.a = \ldots M_1.a_1 \ldots M_2.a_2 \ldots$$

- 使用**依赖图**来表示计算顺序
  - 这些值的计算顺序形成一个偏序关系，如果依赖图中出现了环，表示属性值无法计算

依赖图

- 描述了某棵特定的分析树上各个属性之间的信息流 (计算顺序)
  - 从实例a_1到实例a_2的有向边表示计算a_2时需要a_1的值
- 对于分析树结点N，与N关联的每个属性a都对应依赖图的一个结点$N.a$

产生式	语义规则
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

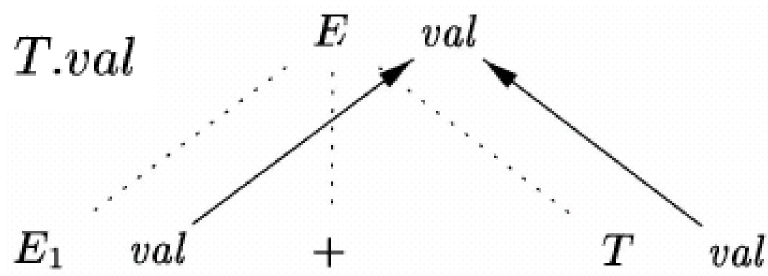


图 5-6 $E.val$ 由 $E_1.val$ 和 $T.val$ 综合得到

属性值的计算顺序

- 各个属性值需要按照依赖图的拓扑顺序计算
  - 如果依赖图中存在环，则属性计算无法进行
- 给定一个SDD，很难判定是否存在一棵分析树，其对应的依赖图包含环

- 但是特定类型的SDD一定不包含环，且有固定的计算顺序
  - 如：S属性的SDD，L属性的SDD（后面讲解）

Synthesized属性的SDD

- 每个属性都是综合属性，都是根据子构造的属性计算出父构造的属性
- 在依赖图中，总是通过子结点的属性值来计算父结点的属性值，可以与自底向上或自顶向下的语法分析过程一起计算
  - 自底向上
    - 在构造分析树结点的同时计算相关的属性 (此时其子结点的属性必然已经计算完毕)
  - 自顶向下
    - 在递归子程序法中，在过程$A()$的最后计算A的属性 (此时A调用的其他过程 (对应于其子结构) 已经调用完毕)

在分析树上计算SDD

- 按照后序遍历的顺序计算属性值即可

```
postorder(N)
{
    for (从左边开始，对N的每个子结点C)
        postorder(C);
    // 递归调用返回时，各子结点的属性已计算完毕
    对N的各个属性求值; // N.a = … C_i.a_i …
}
```

- 在LR分析过程中，实际上不需要构造分析树结点

回顾

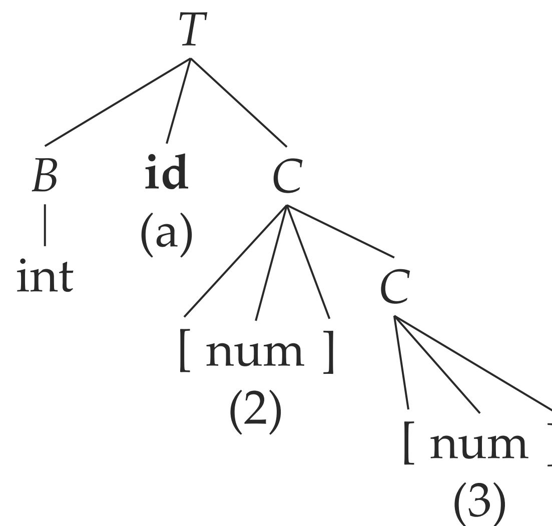
- 语义分析
  - 计算程序构造的语义信息
  - 检查程序中的语义错误
- 语法制导定义SDD (Syntax-Directed Definition)
  - 赋予语法树结点属性以表示语义信息
  - 通过产生式规则计算属性 (语义信息)
- 属性与计算
  - 综合属性与继承属性
  - 计算方式
    - 在语法分析过程中进行计算
    - 遍历语法树进行计算

S属性SDD的局限

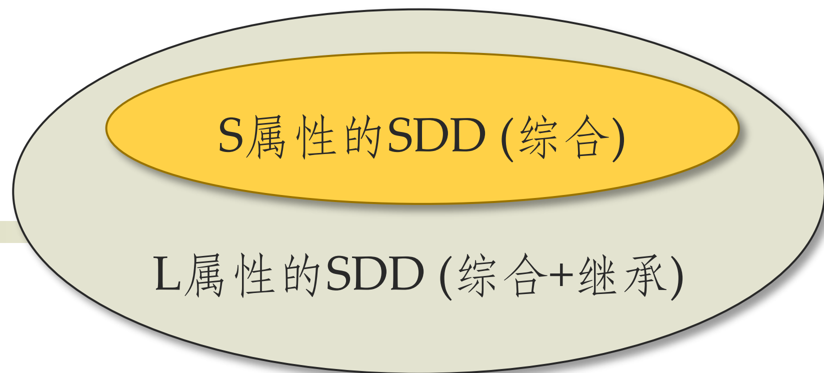
- S属性的SDD中每个属性都是综合属性
  - 便于计算

- 无法计算同时需要继承和综合属性的语义信息
  - $T \rightarrow B$ **id** C
  - $B \rightarrow$ int | float
  - $C \rightarrow$ [num] C | ε

  - 对于int a[2][3]，a的类型是?



L属性的SDD



- 每个属性
  - 是综合属性，或
  - 是继承属性，且$A \rightarrow X_1X_2\ldots X_n$中计算$X_i.a$的规则只用
    - A的继承属性，或
    - X_i左边的文法符号X_j的继承属性或综合属性 (j<i)，或
    - X_i自身的继承或综合属性 (这些属性间的依赖关系不形成环)
- 特点
  - 依赖图中的边
    - 综合属性从下到上
    - 继承属性从上到下，或从左到右
  - 计算一个属性值时，它所依赖的属性值都已计算完毕

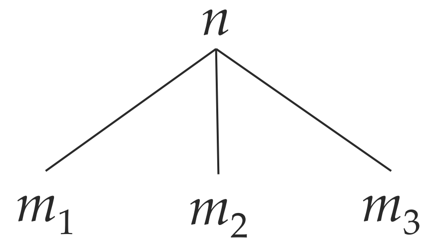
L属性SDD和自顶向下语法分析 (1)

- 在递归子程序法中实现L属性
  - 对于每个非终结符号A，调用其对应过程前计算继承属性，从过程返回前计算综合属性
- 在处理规则$A \rightarrow X_1X_2 \ldots X_n$时
  - 在调用$X_i()$之前计算X_i的继承属性
  - 在该产生式对应代码的最后计算A的综合属性
  - 如果所有文法符号的属性计算按上面的方式进行，计算顺序必然和依赖关系一致

L属性SDD和自顶向下语法分析 (2)

- L属性SDD其属性总可以按如下方式计算

```
L_dfvisit(n)
{
    for m = 从左到右n的每个子节点 do
    {
        计算m的继承属性;
        L_dfvisit(m);
    }
    计算n的综合属性;
}
```



L属性SDD的反例和例子

- 非L属性的例子
  - $A \rightarrow BC$　　$A.s = B.b;$　　$B.i = f(C.c, A.s)$
- L属性的例子
  - 基本类型和数组类型的L属性定义

类型

- 什么是类型?
  - 不同语言有不同定义
- 非严谨的概括：类型≈数据+操作
  - 数据：一个类型包含的所有取值集合
  - 操作：一个类型支持的所有操作集合
- 为什么需要类型？类型有什么用?
- 类型检查
  - 计算各表达式的类型
  - 检查各个运算是否是相关类型所支持的合法操作

类型系统

- 类型可视为一种属性，为程序中的元素（变量、语句、表达式...）赋予类型的一套规则称为**类型系统**
- 类型系统规定了每个类型的元素能支持的操作，违反规定的操作称为**类型错误**（Type error）
- 检查程序中是否有违反类型错误的过程称为**类型检查**（Type checking）

类型系统的作用

- 编译器视角
  - 类型可以决定数据的大小（size）
  - 指导编译器生成操作数据的相应指令
  - 指导编译器进行正确的内存布局
- 程序员视角
  - 通常类型错误意味着程序逻辑错误
    - 整型与字符串相加、为接收整型的函数传入字符串参数……
  - 帮助找出程序中的bug

类型系统：静态vs动态

- 静态类型：在编译期进行类型检查
- 动态类型：在运行时进行类型检查
- 并非绝对
  - 静中有动：反射（Reflection）
  - 动中有静：类型注解，Gradual Typing

类型系统：强类型vs弱类型

- 并无精确定义（不同人有不同理解），可理解为类型检查的严格程度
  - 强类型的类型规则更为严格
  - 一些在强类型的类型系统中被视为类型错误的行为可能在一个弱类型的系统中是被允许的
- 通常强类型的语言倾向于在编译器进行类型检查
  - 强类型 != 静态类型

类型系统：强类型vs弱类型

- 并无精确定义（不同人有不同理解），可理解为类型检查的严格程度
  - 强类型的类型规则更为严格
  - 一些在强类型的类型系统中被视为类型错误的行为可能在一个弱类型的系统中是被允许的
- 通常强类型的语言倾向于在编译器进行类型检查
  - 强类型 != 静态类型

	强类型	弱类型
静态类型		
动态类型		

类型系统：强类型vs弱类型

- 并无精确定义（不同人有不同理解），可理解为类型检查的严格程度
  - 强类型的类型规则更为严格
  - 一些在强类型的类型系统中被视为类型错误的行为可能在一个弱类型的系统中是被允许的
- 通常强类型的语言倾向于在编译器进行类型检查
  - 强类型 != 静态类型

	强类型	弱类型
静态类型	Java, Haskell	C/C++
动态类型	Ruby, Python	JavaScript, VB

基本类型的变量声明

产生式	语义规则
$D \rightarrow T\ L$	$L.inh = T.type$
$T \rightarrow$ **int**	$T.type = \text{int}$
$T \rightarrow$ **float**	$T.type = \text{float}$
$L \rightarrow L_1$, **id**	$L_1.inh = L.inh$ **id**$.type = L.inh$
$L \rightarrow$ **id**	**id**$.type = L.inh$

- int x
- float a, b, c, d

语句文法

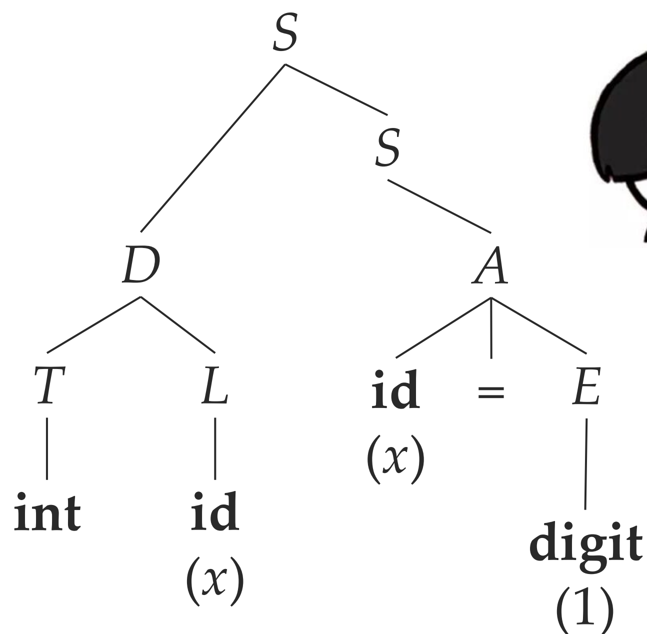
- 语句
  - $S \rightarrow D\ S \mid A\ S \mid \varepsilon$
  - $A \rightarrow \textbf{id} = E$

- 变量声明
  - $D \rightarrow T\ L$
  - $T \rightarrow \textbf{int} \mid \textbf{float}$
  - $L \rightarrow L, \textbf{id} \mid \textbf{id}$

语句文法

- 语句
  - $S \rightarrow D\ S \mid A\ S \mid \varepsilon$
  - $A \rightarrow$ **id** = E

- 变量声明
  - $D \rightarrow T\ L$
  - $T \rightarrow$ **int** | **float**
  - $L \rightarrow L$, **id** | **id**



具有受控副作用的语义规则

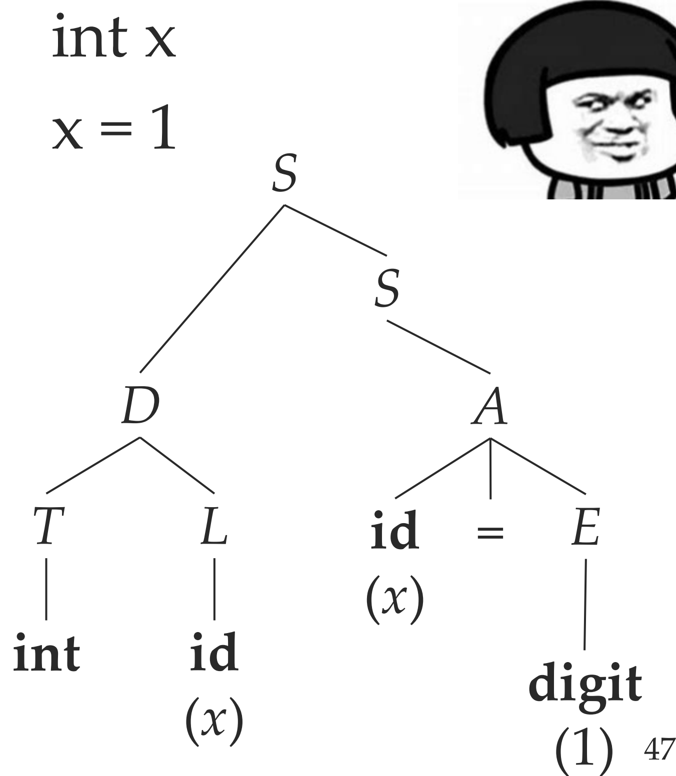
- 属性文法没有副作用，但增加了描述的复杂度
  - 比如语法分析时，如果没有副作用，标识符表就必须作为属性传递
  - 可以把标识符表作为全局变量，然后通过函数来添加新的标识符

- **受控的副作用**
  - 不会对属性求值产生约束，即可以按照任何拓扑顺序求值，不会影响最终结果
  - 或者对求值过程添加简单的约束

受控副作用的例子

- 变量声明SDD中的副作用
  - *addType*将标识符的类型信息加入标识符表中
  - 只要标识符不被重复声明，其类型信息总是正确的

产生式	语义规则
1) $D \rightarrow T\ L$	$L.inh = T.type$
2) $T \rightarrow \textbf{int}$	$T.type = \text{integer}$
3) $T \rightarrow \textbf{float}$	$T.type = \text{float}$
4) $L \rightarrow L_1\ ,\ \textbf{id}$	$L_1.inh = L.inh$ $addType(\textbf{id}.entry, L.inh)$
5) $L \rightarrow \textbf{id}$	$addType(\textbf{id}.entry, L.inh)$



产生式规则： $A \rightarrow \textbf{id} = E$

- **id**.*type* = lookupIDTable(**id**.*entry*)->*type*;

类型结构

- 简化的类型表达式的语法
  - $T \rightarrow B\ C$
  - $B \rightarrow$ int | float
  - $C \rightarrow$ [num] C | ε
- 生成类型表达式的SDD

产生式	语义规则
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow \textbf{int}$	$B.t = integer$
$B \rightarrow \textbf{float}$	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

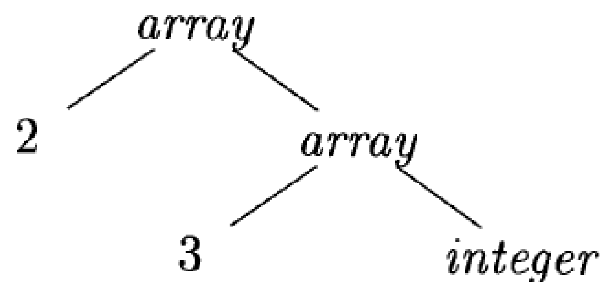
类型的含义

- 类型包括两个部分：$T \rightarrow B\ C$
  - 基本类型 B
  - 分量 C
- 分量形如[2][3]
  - 表示2×3的二维数组
  - 如：int [2][3]

- 数组构造算符array(n, t) [复合类型]
  - array(2, array(3, int))表示抽象的2×3的二维数组

类型表达式的生成过程

- 输入：int [2][3]



$T.t = array(2, array(3, integer))$

$B.t = integer$

$C.b = integer$
$C.t = array(2, array(3, integer))$

int

[2]

$C.b = integer$
$C.t = array(3, integer)$

[3]

$C.b = integer$
$C.t = integer$

ϵ

产生式	语义规则
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow [\ \mathbf{num}\ ]\ C_1$	$C.t = array(\mathbf{num}.val,\ C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

回顾

- L属性的SDD
  - 比S属性能力更强的SDD
    - 允许综合属性
    - 允许受限制的继承属性
  - 属性的计算方法
    - 自顶向下分析法（边建树边计算）
    - 遍历法（建完树再计算）
- 类型检查
  - 通过带副作用的SDD分析变量类型
  - 通过L属性的SDD分析数组类型

SDD (语法制导定义)的应用

- 类型分析只是SDD的一个应用
- SDD的其它编译相关应用
  - 抽象语法树构造
  - 代码翻译

版权所有 南京大学计算机科学与技术系 许畅 2026春季版（有修改）

语法分析树 (Parse Tree)

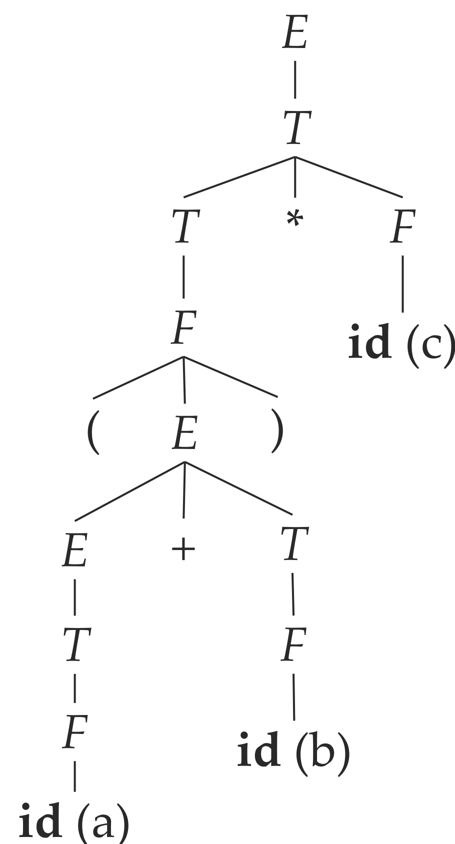
- **具体语法树 (Concrete Syntax Tree)**
  - 保留所有词法元素
  - 所有词法元素都有对应节点
  - 完整地还原非终结符到串的推导过程
  - 严格符合源语言的上下文无关文法

语法分析树 (Parse Tree)

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \text{id}$

- **具体语法树 (Concrete Syntax Tree)**
  - 保留所有词法元素
    - 包含对分析无用的噪音
  - 所有词法元素都有对应节点
    - 节点数量大
  - 完整地还原非终结符到串的推导过程
    - 包含所有中间推导过程，引入大量中间节点
  - 严格符合源语言的上下文无关文法
    - 可根据文法自动生成

$(a + b) * c$

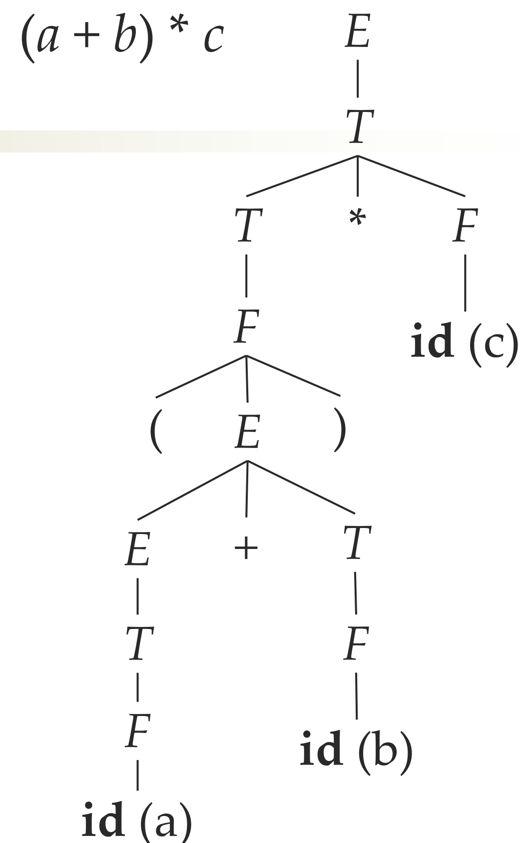


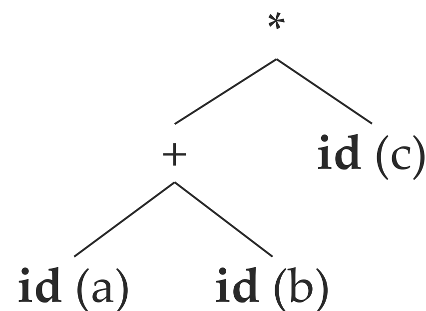
实际当中很少直接在CST上进行分析/处理

语法树 (Syntax Tree)

- **具体语法树 (Concrete Syntax Tree)**
  - 保留所有词法元素
  - 所有词法元素都有对应节点
  - 完整地还原非终结符到串的推导过程
  - 严格符合源语言的上下文无关文法
- **抽象语法树 (Abstract Syntax Tree)**
  - 只保留必要的词法元素
  - 某些词法元素信息存入父节点的属性
  - 移除没有实质信息的中间推导过程
  - 不符合源语言的上下文无关文法

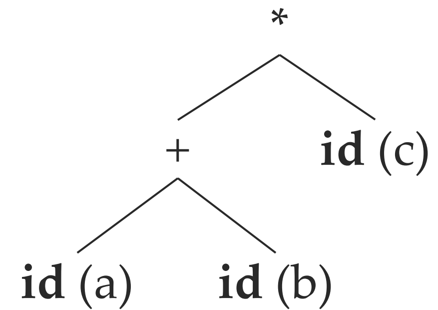
AST的优势
- 更少的结点数量与种类
- 更易于分析和处理
- 独立于具体文法





构造抽象语法树的SDD

- **抽象语法树**
  - 每个结点代表一个语法结构，对应于运算符
  - 结点的每个子结点代表其子结构，对应于运算分量
  - 表示这些子结构按照特定的方式组成了较大的结构
  - 可以忽略掉一些标点符号等非本质的东西
- 抽象语法树的表示方法
  - 每个结点用一个对象表示
  - 对象有多个域
    - 叶子结点中只存放词法值
    - 内部结点中存放了op值和参数(通常指向其子结点)



构造简单表达式的抽象语法树的SDD

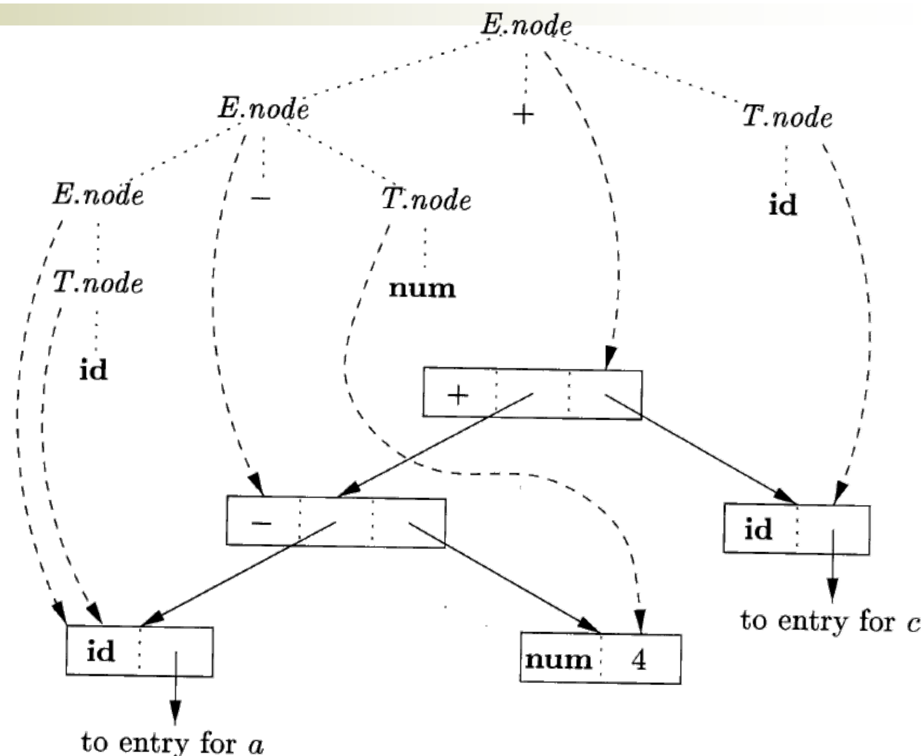
- 属性*E.node*指向*E*对应的抽象语法树的根结点

产生式	语义规则
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) $T \to \mathbf{num}$	$T.node = \mathbf{new}\ Leaf(\mathbf{num}, \mathbf{num}.val)$

- **抽象语法树 (Abstract Syntax Tree)**
  A. 只保留必要的词法元素
  B. 某些词法元素信息存入父节点的属性
  C. 移除没有实质信息的中间推导过程

表达式抽象语法树的构造过程

- 输入
  - a – 4 + c



- 步骤

$p_1 = \text{new Leaf(id, entry_a)};$

$p_2 = \text{new Leaf(num, 4)};$

$p_3 = \text{new Node('–', } p_1, p_2);$

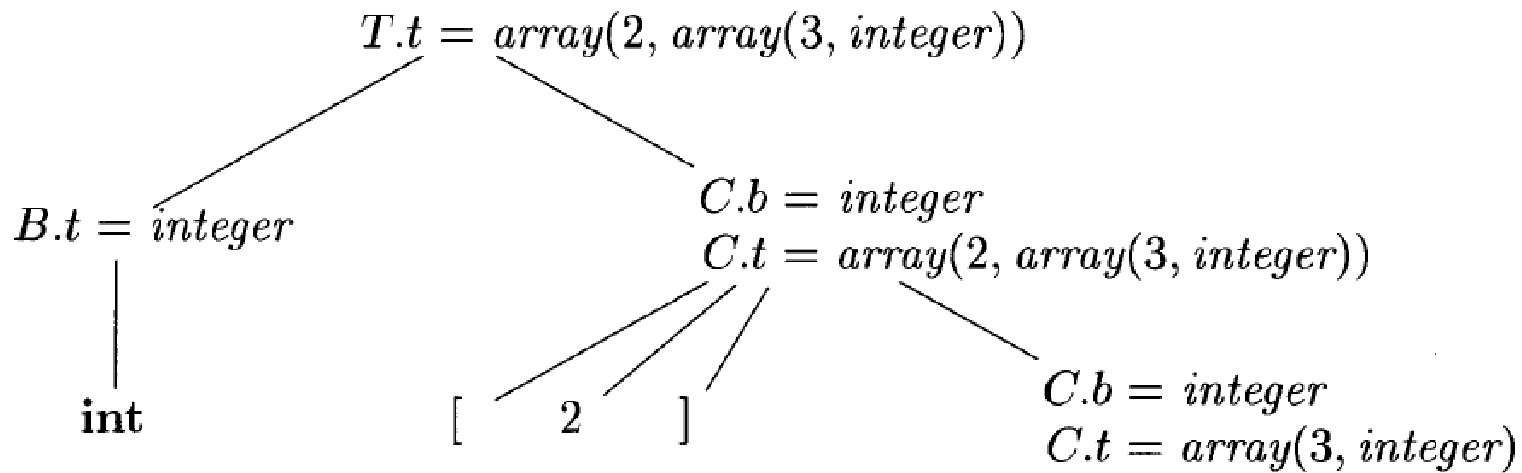
$p_4 = \text{new Leaf(id, entry_c)};$

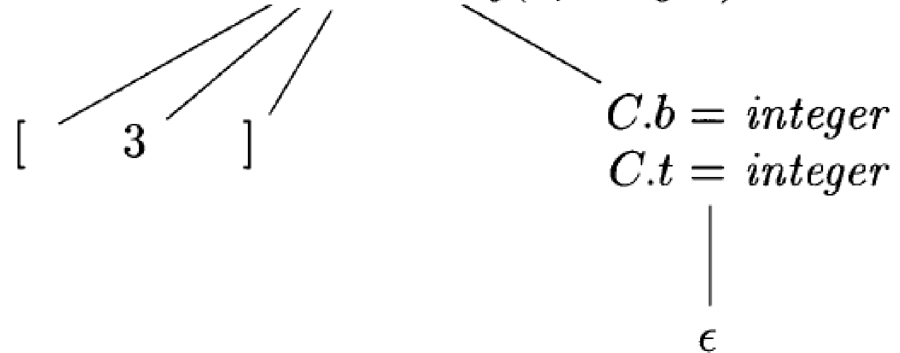
$p_5 = \text{new Node('+', } p_3, p_4);$

产生式	语义规则
1) $E \to E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2) $E \to E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3) $E \to T$	$E.node = T.node$
4) $T \to (\ E\ )$	$T.node = E.node$
5) $T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6) $T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

语义规则的执行时机

- 输入： int [2][3]





产生式	语义规则
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow \textbf{int}$	$B.t = integer$
$B \rightarrow \textbf{float}$	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

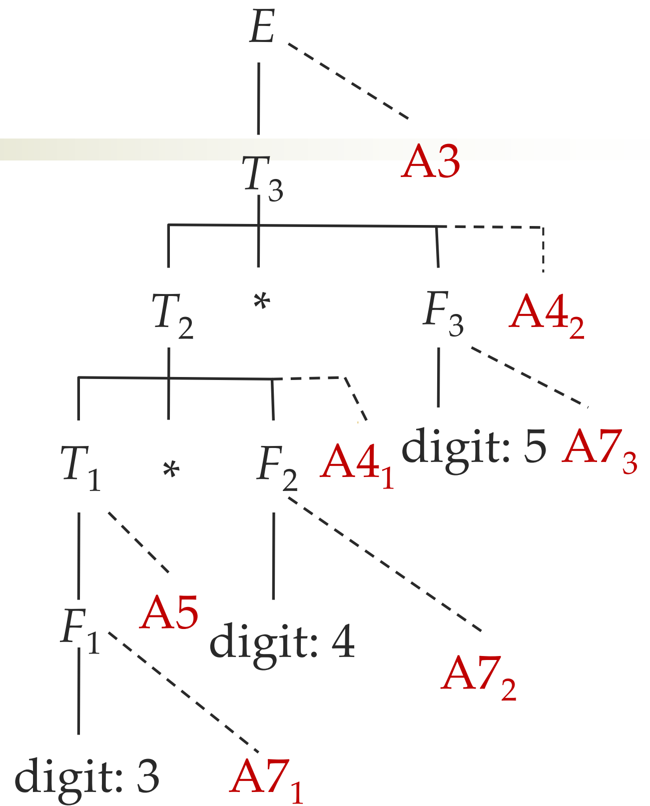
语法制导的翻译方案

- **语法制导的翻译方案** (SDT) 是在产生式体中嵌入语义动作 (程序片断) 的上下文无关文法
- SDT的基本实现方法
  - 建立语法分析树 (CST)
  - 将语义动作看作是虚拟结点 (绑定语义动作)
  - 深度优先、从左到右地遍历分析树，在访问虚拟结点时执行相应的语义动作
- 用SDT实现两类重要的SDD
  - 基本文法是LR的，SDD是S属性的
  - 基本文法是LL的，SDD是L属性的

例子

- 语句3 * 4 * 5的分析树
- 动作执行顺序
  - $A7_1$, A5, $A7_2$, $A4_1$, $A7_3$, $A4_2$, A3
  - 动作的不同实例所访问的属性值属于不同的结点

L	$\rightarrow$	E **n**	{ print($E.val$); }	A1
E	$\rightarrow$	$E_1 + T$	{ $E.val = E_1.val + T.val$; }	A2
E	$\rightarrow$	T	{ $E.val = T.val$; }	A3
T	$\rightarrow$	$T_1 * F$	{ $T.val = T_1.val \times F.val$; }	A4
T	$\rightarrow$	F	{ $T.val = F.val$; }	A5
F	$\rightarrow$	(E)	{ $F.val = E.val$; }	A6
F	$\rightarrow$	**digit**	{ $F.val =$ **digit**.$lexval$; }	A7



可在语法分析过程中实现的SDT

- 实现SDT时，实际上并不会真的构造语法分析树，而是在分析过程中执行语义动作
- 即使基础文法可以应用某种语法分析技术，仍可能因为动作的缘故导致此技术不可应用

- 判断是否可在分析过程中实现
  - 将每个语义动作替换为一个独有的非终结符号M_i，其产生式为$M_i \rightarrow \varepsilon$
  - 如果新的文法可以由某种方法进行分析，那么这个SDT就可以在这个分析过程中实现

SDT可否用特定分析技术实现的例子

- 新文法
  - $L \rightarrow E \text{ n } M_1$　　　　$M_1 \rightarrow \varepsilon$
  - $E \rightarrow E + T M_2$　　　　$M_2 \rightarrow \varepsilon$
  - $E \rightarrow T M_3$　　　　$M_3 \rightarrow \varepsilon$
  - ……

L	$\rightarrow$	E **n**	$\{ \text{print}(E.val); \}$ A1
E	$\rightarrow$	$E_1 + T$	$\{ E.val = E_1.val + T.val; \}$ A2
E	$\rightarrow$	T	$\{ E.val = T.val; \}$ A3
T	$\rightarrow$	$T_1 * F$	$\{ T.val = T_1.val \times F.val; \}$ A4
T	$\rightarrow$	F	$\{ T.val = F.val; \}$ A5
F	$\rightarrow$	(E)	$\{ F.val = E.val; \}$ A6
F	$\rightarrow$	**digit**	$\{ F.val = \textbf{digit}.lexval; \}$ A7

后缀翻译方案

- 文法可以自底向上分析 (即LR的) 且其SDD是S属性的，必然可以构造出后缀SDT
- **后缀SDT**：所有动作都在产生式最右端的SDT

- 构造方法
  - 将每个语义规则看作是一个赋值语义动作
  - 将所有的语义动作放在规则的最右端

后缀翻译方案的例子

- 实现桌上计算器的后缀SDT

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	$\rightarrow$	$T_1 * F$	$\{ T.val = T_1.val \times F.val; \}$
T	$\rightarrow$	F	$\{ T.val = F.val; \}$
F	$\rightarrow$	(E)	$\{ F.val = E.val; \}$
F	$\rightarrow$	**digit**	$\{ F.val =$ **digit**$.lexval; \}$

- 注意动作中对属性值的引用
  - 允许语句引用全局变量、局部变量和文法符号的属性
  - 文法符号的属性只能被赋值一次

后缀SDT的语法分析栈实现

- 可以在LR语法分析的过程中实现
  - 归约时执行相应的语义动作
  - 定义用于记录各文法符号的属性的union结构
  - 栈中的每个文法符号(或状态)都附带一个这样的union类型的值
  - 在按照产生式$A \rightarrow XYZ$归约时，Z的属性可以在栈顶找到，Y的属性可以在下一个位置找到，X的属性可以在再下一个位置找到

	X	Y	Z	状态 / 文法符号
	$X.x$	$Y.y$	$Z.z$	综合属性
			↑ 栈顶	

分析栈实现的例子

- 假设语法分析栈存放在一个被称为*stack*的记录数组中，下标*top*指向栈顶
  - *stack*[*top*]指向这个栈的栈顶
  - *stack*[*top* – 1]指向栈顶下一个位置
- 如果不同的文法符号有不同的属性集合，我们可以使用union来保存这些属性值
  - 归约时能够知道栈顶向下的各个符号分别是什么，因此我们也能够确定各个union中存放了什么值

后缀SDT的栈实现

产生式	语义动作
$L \to E\ \mathbf{n}$	{ print($stack[top-1].val$); $top = top - 1$; }
$E \to E_1 + T$	{ $stack[top-2].val = stack[top-2].val + stack[top].val$; $top = top - 2$; }
$E \to T$	
$T \to T_1 * F$	{ $stack[top-2].val = stack[top-2].val \times stack[top].val$; $top = top - 2$; }
$T \to F$	
$F \to (\ E\ )$	{ $stack[top-2].val = stack[top-1].val$; $top = top - 2$; }
$F \to \mathbf{digit}$	

注意：$stack[top-i]$和文法符号的对应

产生式内部带有语义动作的SDT

- 动作左边的所有符号(以及动作)处理完成后，就立刻执行这个动作：$B \rightarrow X \{a\} Y$
  - 自底向上分析时，在X出现在栈顶时执行动作a
  - 自顶向下分析时，在试图展开Y或者在输入中检测到Y的时刻执行a

- 对一般的SDT，都可以先建立分析树(语义动作作为虚拟结点)，然后进行前序遍历并执行动作
- 不是所有的SDT都可以在分析过程中实现
  - 后缀SDT以及L属性对应的SDT可以在分析时完成

L属性的SDT

- 除了通用的SDT实现技术，若基础文法是LL的，则可以将L属性SDD转换成一个SDT，该SDT可以在自顶向下的分析过程中实现
- 从L属性的SDD到SDT的转换
  - 将每个语义规则看作是一个赋值语义动作
  - 将赋值语义动作放到相应产生式$A \rightarrow X_1 X_2 \dots X_n$的适当位置
    - 计算X_i继承属性的动作插入到产生式体中X_i的左边
    - 计算产生式头A综合属性的动作在产生式的最右边

while语句的SDD和SDT

- 产生式$S \rightarrow$ while (C) S_1
  - 为while语句生成中间代码
  - 主要说明语句控制流中的标号生成

- while语句的含义
  - 首先对C求值，若为真，则控制转向S_1的开始处
  - 若为假，则转向while语句的后续语句开始处
  - S_1结束时，要能够跳转到while语句的代码开始处

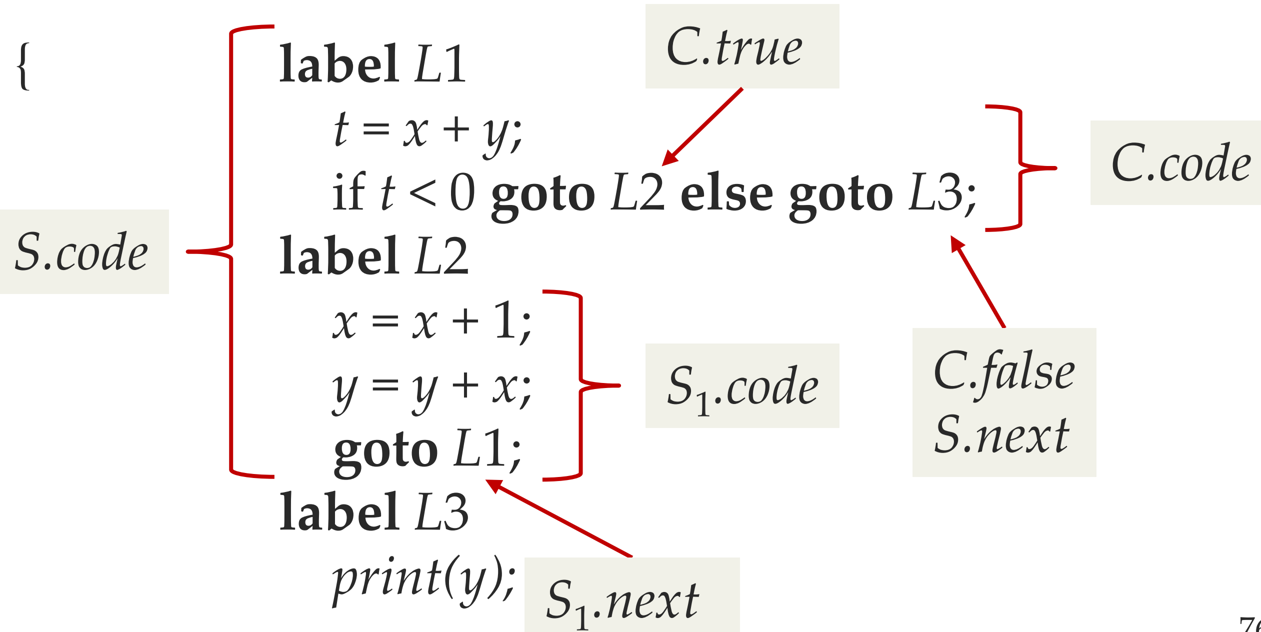
while语句的L属性SDD

$S \rightarrow \textbf{while}\ (\ C\ )\ S_1 \quad L1 = new();$
$L2 = new();$
$S_1.next = L1;$
$C.false = S.next;$
$C.true = L2;$
$S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code$

继承属性

- *next*：语句结束后应该跳转到的标号
- *true, false*：C为真/假时应该跳转到的标号

综合属性code表示代码

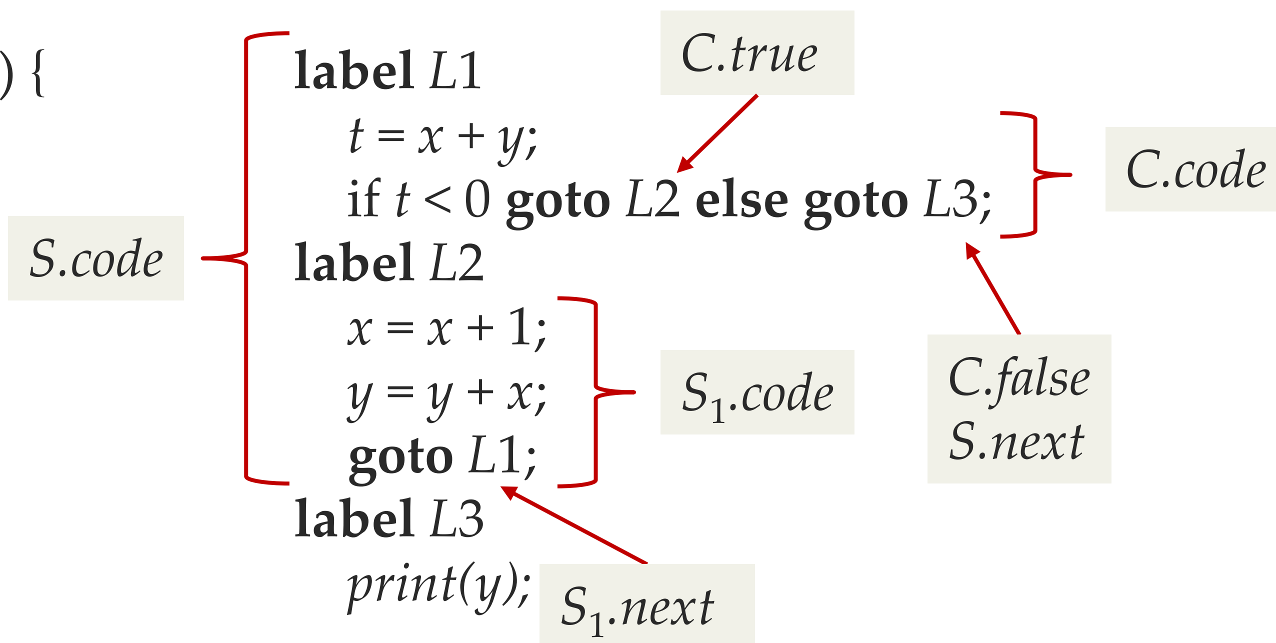


转换为SDT

- 语义动作
  - (a) $L1 = \text{new}(\ )$和$L2 = \text{new}(\ )$：生成新标号
  - (b) $C.\mathit{false} = S.\mathit{next}$; $C.\mathit{true} = L2$：计算C的继承属性
  - (c) $S_1.\mathit{next} = L1$：计算S_1的继承属性
  - (d) $S.\mathit{code} = \ldots$：计算S的综合属型
- 根据放置语义动作的规则得到如下SDT
  - (b)在C之前，(c)在S_1之前，(d)在最右端
  - (a)可以放在最前面

$$
\begin{array}{lll}
S \rightarrow \textbf{while}\ (& \{\ L1 = new();\ L2 = new();\ C.false = S.next;\ C.true = L2;\ \} \\
\quad C\ ) & \{\ S_1.next = L1;\ \} \\
\quad S_1 & \{\ S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code;\ \}
\end{array}
$$

while语句的SDT



$$
\begin{array}{lll}
S \rightarrow \textbf{while} \; (& \{ \; L1 = new(); \; L2 = new(); \; C.false = S.next; \; C.true = L2; \; \} \\
\quad C \;) & \{ \; S_1.next = L1; \; \} \\
\quad S_1 & \{ \; S.code = \textbf{label} \; \| \; L1 \; \| \; C.code \; \| \; \textbf{label} \; \| \; L2 \; \| \; S_1.code; \; \}
\end{array}
$$

L属性SDD的实现

- 使用递归下降的语法分析器
  - 每个非终结符号对应一个函数 $S.next \rightarrow S(next)$
  - 函数的参数接受继承属性，返回值包含了综合属性
- 在函数体中
  - 首先选择适当的产生式
  - 使用局部变量来保存属性
  - 对于产生式体中的终结符号，读入符号并获取其(经词法分析得到的)综合属性
  - 对于非终结符号，使用适当的方式调用相应函数，并记录返回值

避免副作用

递归下降实现L属性SDD的例子

```
string S(label next) {
    string Scode, Ccode; /* 存放代码片段的局部变量 */
    label L1, L2; /* 局部标号*/
    if ( 当前输入 == 词法单元while ) {
        读取输入；
        检查 '(' 是下一个输入符号，并读取输入；
        L1 = new();   C.false   C.true
        L2 = new();
        Ccode = C(next, L2);
        检查 ')' 是下一个输入符号，并读取输入；
        Scode = S(L1);  ←  S.next
        return("label" || L1 || Ccode || "label" || L2 || Scode);
    }
    else /* 其他语句类型 */
}
```

string C(label *false*, label *true*) {…

$S \rightarrow$ **while** ($\{ L1 = new(); L2 = new(); C.false = S.next; C.true = L2; \}$

C) $\{ S_1.next = L1; \}$

S_1 $\{ S.code =$ **label** $\| L1 \| C.code \|$ **label** $\| L2 \| S_1.code; \}$

边扫描边生成属性 (1)

- 当属性值的体积很大，对其进行运算会效率很低
  - code可能是一个上百K的串
  - 许多代码片段被反复复制，很低效
- 可逐步生成属性的各个部分，并增量式地添加到最终的属性值中
- 三个条件
  - 存在一个主属性，且其为综合属性
  - 在产生式中，主属性是通过产生式体中各非终结符号的主属性连接而得到，同时还会连接一些其它元素
  - 各个非终结符号的主属性的连接顺序与它们在产生式体中的顺序相同

边扫描边生成属性 (2)

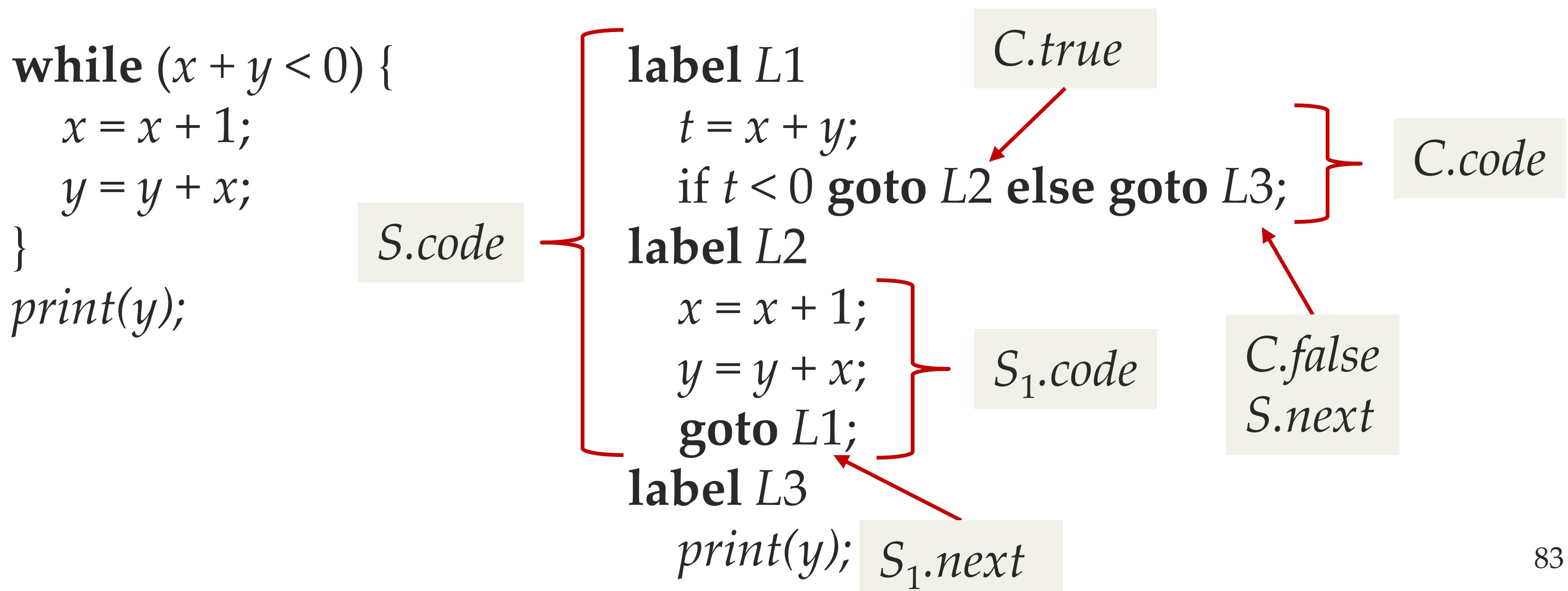
- 基本思想
  - 在适当的时候“发出”元素，保证其能被适当地拼接

emit

- 举例说明
  - 假设扫描一个非终结符号对应的语法结构，调用其相应的函数，并生成主属性
    - $S \rightarrow$ while (C) S_1
    - { $S.code$ = label || $L1$ || $C.code$ || label || $L2$ || $S_1.code$ }
  - 如果各函数都把其主属性 (*code*) 打印出来，则对while语句，只需先打印label $L1$，再调用C (打印C的代码)，再打印label $L2$，再调用S (打印S_1的代码)
  - 需要在适当的时候打印label $L1$和label $L2$

边扫描边生成属性 (3)

- $S \rightarrow$ while (C) S_1
- { $S.code$ = label || $L1$ || $C.code$ || label || $L2$ || $S_1.code$ }
  - 如果各函数都把其主属性 (*code*) 打印出来，则对while语句，只需先打印label $L1$，再调用C (打印C的代码)，再打印label $L2$，再调用S (打印S_1的代码)
  - 需要在适当的时候打印label $L1$和label $L2$



边扫描边生成属性的例子 (1)

- 新的SDT

  $S \rightarrow$ while ({ $L1$ = new(); $L2$ = new(); $C.false = S.next$;
  $C.true = L2$; $print$("label", $L1$); }

  C) { $S_1.next = L1$; $print$("label", $L2$); }

  S_1

- 前提是所有非终结符号的SDT规则都这么做

对照原SDT

$$
\begin{array}{lll}
S \rightarrow \textbf{while}\ (& \{ L1 = new(); L2 = new(); C.false = S.next; C.true = L2; \} \\
\quad C\ ) & \{ S_1.next = L1; \} \\
\quad S_1 & \{ S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code; \}
\end{array}
$$

边扫描边生成属性的例子 (2)

原来的

```
void S(label next) {
    label L1, L2; /* 局部标号 */
    if ( 当前输入 == 词法单元 while ) {
        读取输入 ;
        检查 '(' 是下一个输入符号 , 并读取输入 ;
        L1 = new();
        L2 = new();
        print("label", L1);
        C(next, L2);
        检查 ')' 是下一个输入符号 , 并读取输入 ;
        print("label", L2);
        S(L1);
    }
    else /* 其他语句类型 */
}
```

```
string S(label next) {
    string Scode, Ccode; /* 存放代码片段的局部变量 */
    label L1, L2; /* 局部标号*/
    if ( 当前输入 == 词法单元while ) {
        读取输入 ;
        检查 '(' 是下一个输入符号 , 并读取输入 ;
        L1 = new();
        L2 = new();
        Ccode = C(next, L2);
        检查 ')' 是下一个输入符号 , 并读取输入 ;
        Scode = S(L1);
        return("label" || L1 || Ccode || "label" || L2 || Scode);
    }
    else /* 其他语句类型 */
}
```

边扫描边生成

总结

- 语义分析
  - 检查程序是否符合语言的语义规则
  - 类型规则、变量使用规则、函数调用规则……
- 语法制导定义 (Syntax-Directed Definition)
  - 文法、属性 (继承/综合)、产生式规则
  - 属性值表示语义信息、产生式规则计算语义信息
- SDD的其它应用
  - 抽象语法树 (AST)、代码翻译
- 语法制导翻译 (Syntax-Directed Translation)
  - 明确指定SDD语义动作的执行时机
  - SDT的实现 (遍历语法树/与语法分析同步进行)