Outline

- ✧ 基于语法制导定义的语义处理*
- ✧ 翻译方案*
- ✧ 继承属性的模拟求值**
- ✧ 类型检查
- ✧ 符号表
- ✧ 运行时存储组织*

基于SDD的语义处理

- 基于SDD的语义处理即为语法制导的语义处理（*Syntax-Directed Semantic Process*），也称语法制导的翻译（*Syntax-Directed Translation*）

  处理方法分两类：

  - 树遍历方法

    通过遍历分析树进行属性计算

  - 单遍的方法 （On-the-fly方法 ）

    语法分析遍的同时进行属性计算

✧ 基于树遍历方法的语义处理

– 步骤

- 构造输入串的语法分析树
- 构造依赖图（*Dependency graph*）
- 若该依赖图是无环的，则按照此无环图的一种拓扑排序（*Topological sort*）对分析树进行遍历，就可以计算所有的属性

注：若依赖图含有环，则相应的SDD不可采用这种方法进行语义处理，此类SDD不是良定义的. 所谓良定义的SDD，当且仅当它的规则集合能够为所有分析树中的属性集确定唯一的值集。

基于SDD的语义处理

- ✧ 依赖图是一个有向图，用来描述分析树中的属性与属性之间的相互依赖关系
  - – 构造算法

for 分析树中每一个结点n do

for 结点n对应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 结点n所用产生式中每个形如$f(c_1,c_2,\ldots c_k)$的语义规则 do

为该规则在依赖图中也建立一个结点（称为虚结点）；

for 分析树中每一个结点n do

for 结点n所用产生式对应的每个语义规则 $b:=f(c_1,c_2,\ldots c_k)$ do

（可以只是$f(c_1,c_2,\ldots c_k)$，此时b结点为一个虚结点）

for i :=1 to k do

从c_i结点到b结点构造一条有向边

基于SDD的语义处理

✧ 基于树遍历的处理方法举例

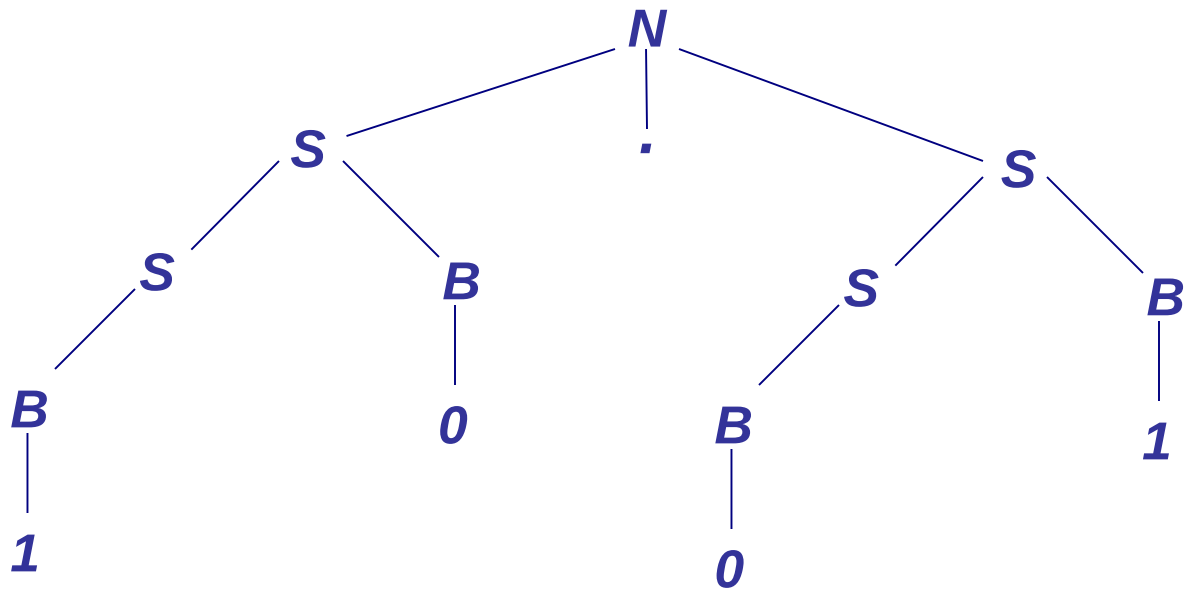
– 设有如下SDD，考虑输入串 10.01 的语义处理过程

产生式	语义规则
$N \rightarrow S_1.S_2$	$\{ N.v := S_1.v+S_2.v;\ S_1.f := 1;\ S_2.f := 2^{-S_2.l} \}$
$S \rightarrow S_1B$	$\{ S_1.f := 2S.f;\ B.f := S.f;\ S.v := S_1.v+B.v;\ S.l := S_1.l+1 \}$
$S \rightarrow B$	$\{ S.l := 1;\ S.v := B.v;\ B.f := S.f \}$
$B \rightarrow 0$	$\{ B.v := 0 \}$
$B \rightarrow 1$	$\{ B.v := B.f \}$

基于SDD的语义处理

✧ 基于树遍历的处理方法举例

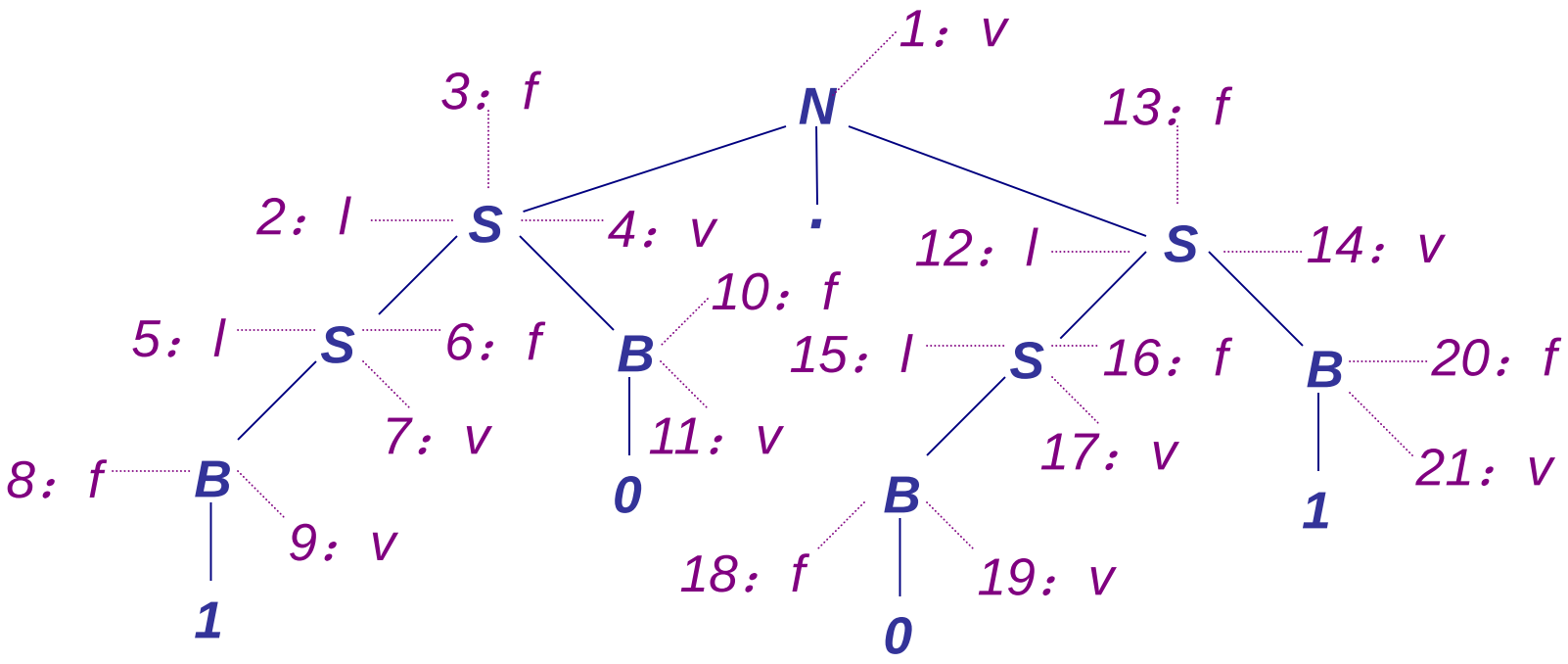
– 步骤一 构造输入串10.01的语法分析树



基于SDD的语义处理

✧ 基于树遍历的处理方法举例

– 步骤二 为分析树中所有结点的每个属性建立一个依赖图中的结点，并给定一个标记序号

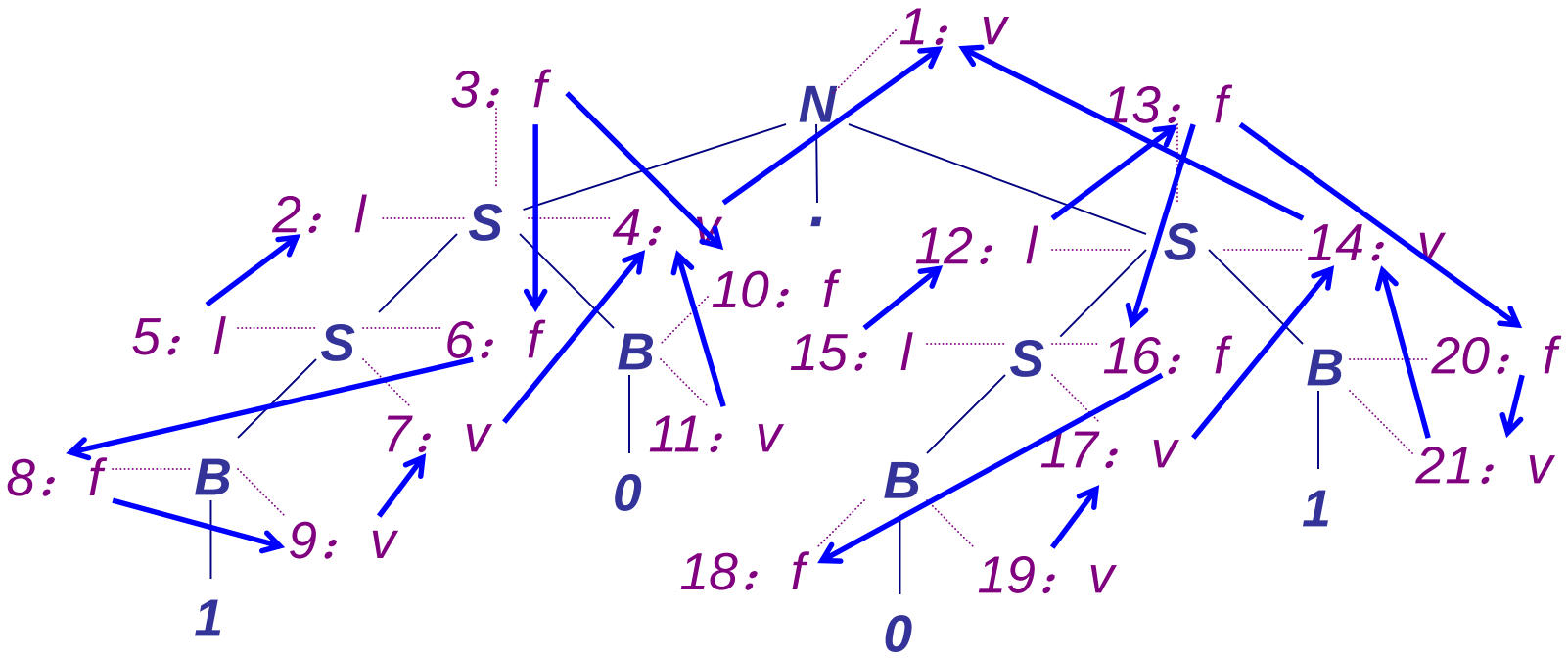


基于SDD的语义处理

✧ 基于树遍历的处理方法举例

– 步骤三 根据语义规则，建立依赖图中的有向边

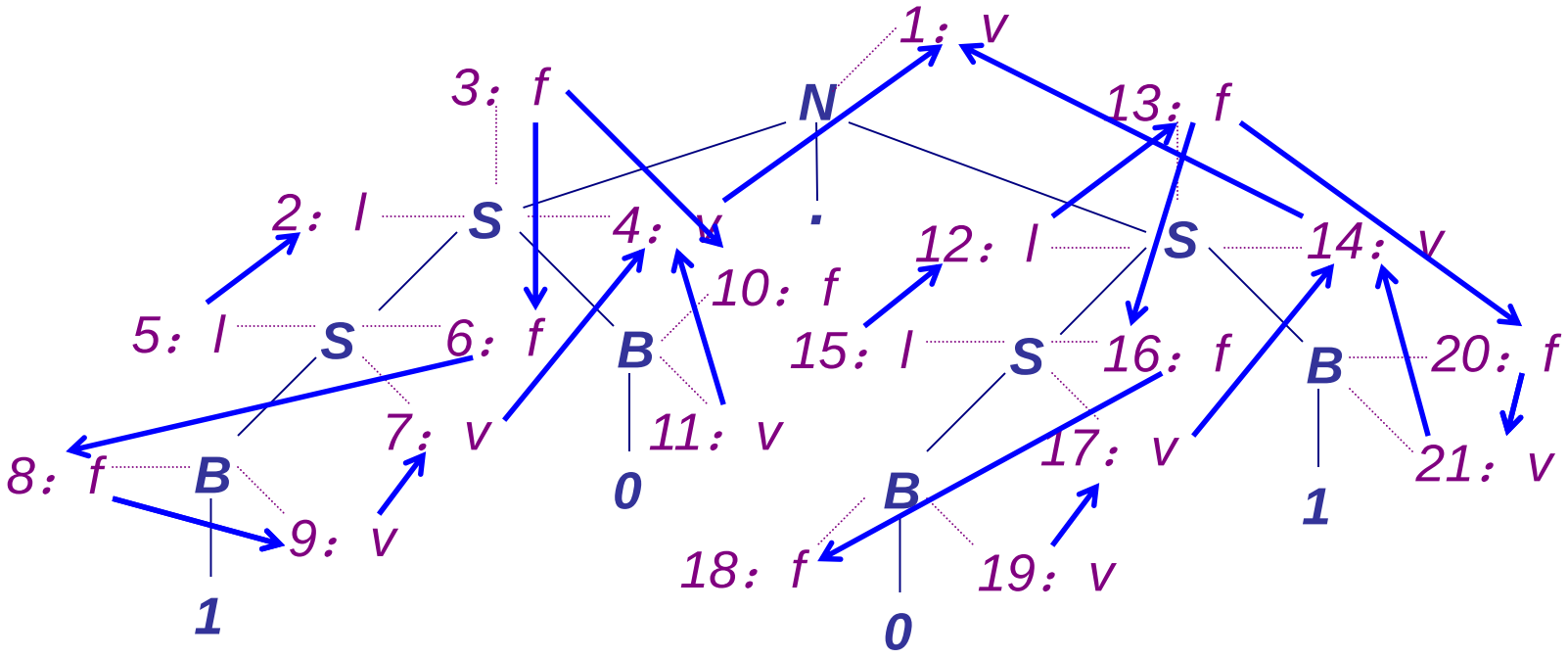
$N \rightarrow B_1 B_2$ { $B_1.f := S.f$; $B_2.f := S.f$ }



基于SDD的语义处理

✧ 基于树遍历的处理方法举例

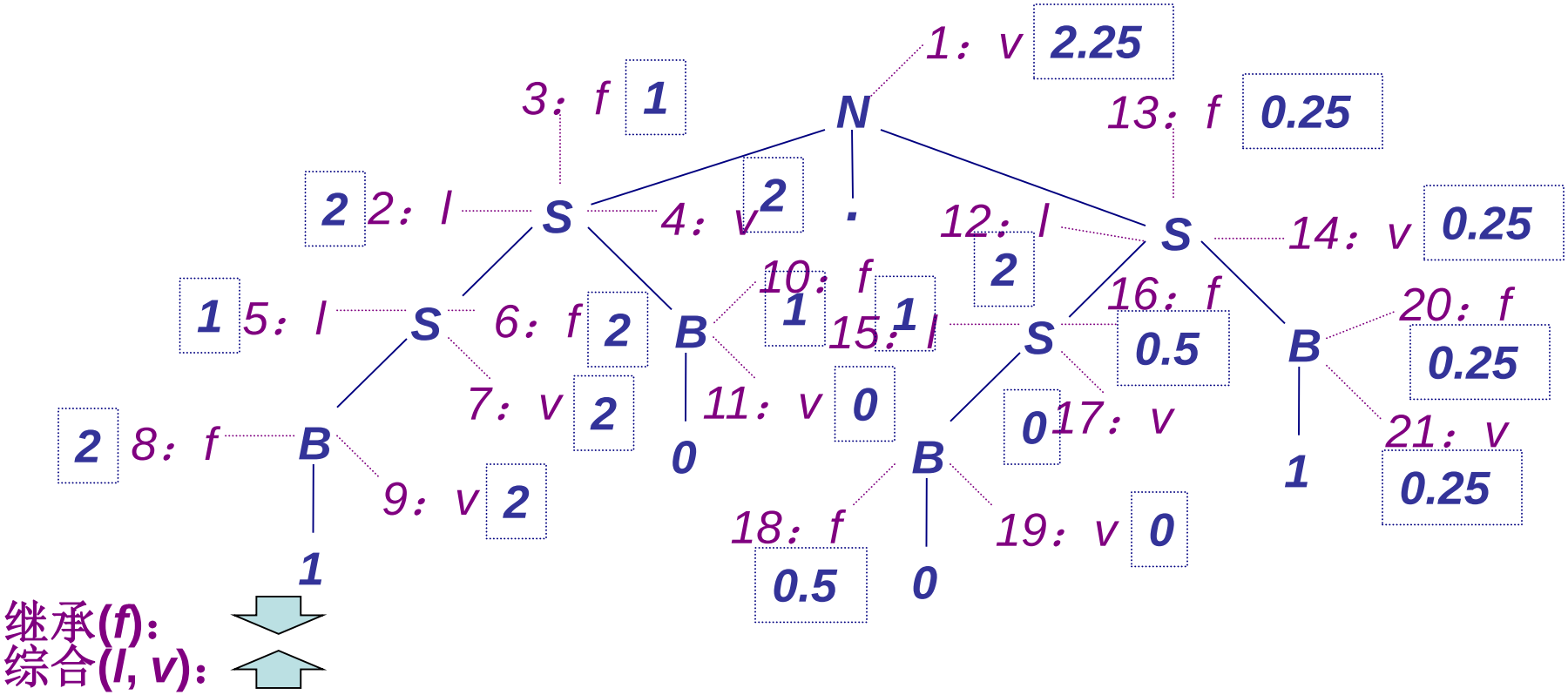
– 步骤四 容易看出，该依赖图是无环的，因此存在拓扑排序. 依任何一个拓扑排序，都能够顺利完成属性值的计算. 如下是一种可能的计算次序：
3,5,2,6,10,8,9,7,11,4,15,12,13,16,20,18,21,19,17,14,1



基于SDD的语义处理

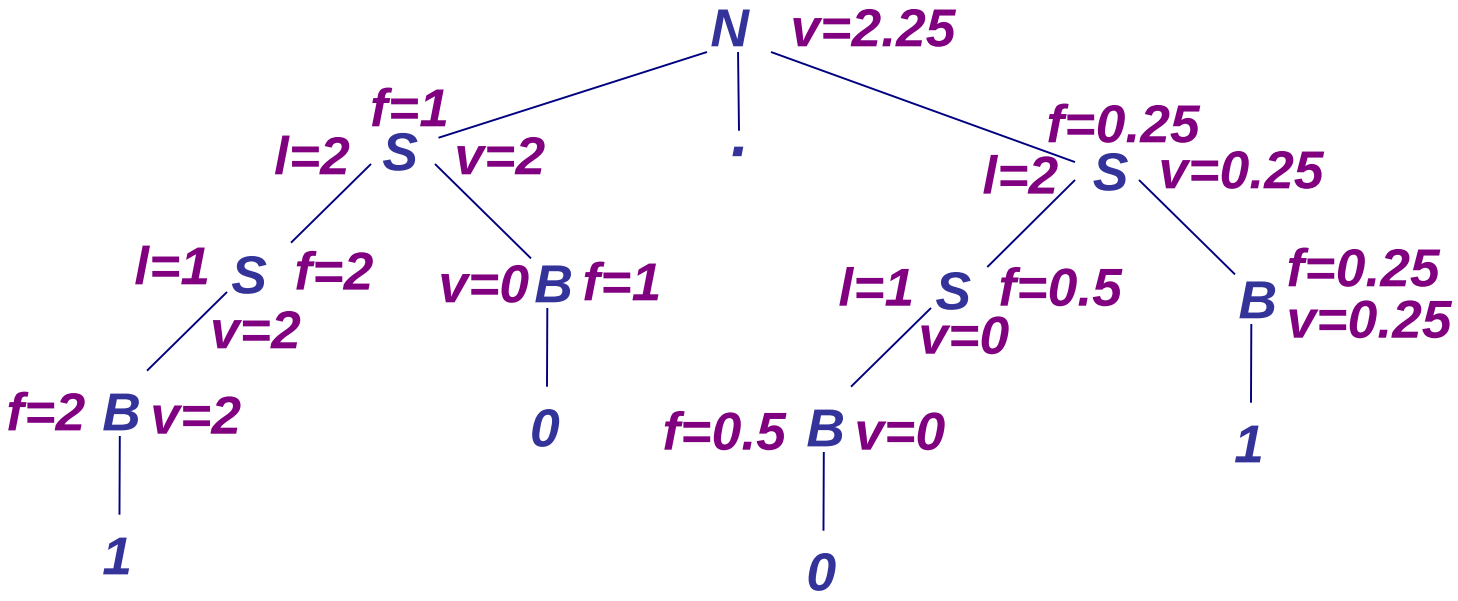
✧ 基于树遍历的处理方法举例

– 步骤五 依计算次序，根据语义规则求出各结点对应的属性值. 对如下结点次序进行计算：
3,5,2,6,10,8,9,7,11,4,15,12,13,16,20,18,21,19,17,14,1



基于SDD的语义处理

- 带标注（annotated）的语法分析树
  - 语法分析树中各结点属性值的计算过程被称为对语法分析树的标注（*annotating*）或修饰（*decorating*），用带标注的语法分析树表示属性值的计算结果，如：



基于SDD的语义处理

- 基于SDD的语义处理即为语法制导的语义处理（*Syntax-Directed Semantic Process*），也称语法制导的翻译（*Syntax-Directed Translation*）

  处理方法分两类：

  - 树遍历方法

    通过遍历分析树进行属性计算

  - 单遍的方法 （On-the-fly方法 ）

    语法分析遍的同时进行属性计算

基于SDD的语义处理

- 单遍的方法（On-the-fly方法）
  - 语法分析遍的同时进行属性计算
    - 自下而上方法
    - 自上而下方法
  - 只适用于特定文法

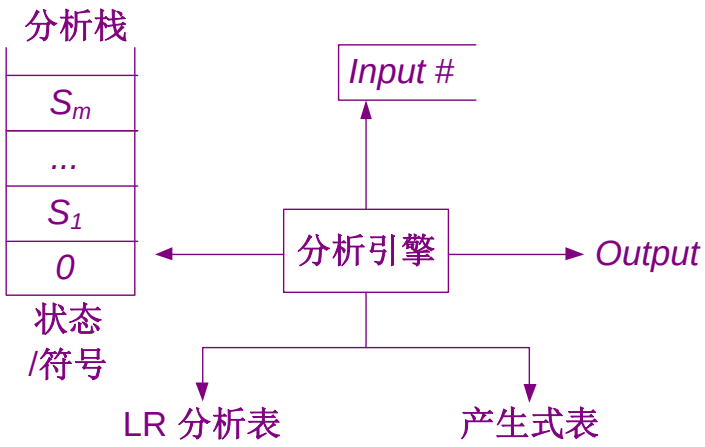
  两类SDD：
    - S-属性SDD
    - L-属性SDD

基于SDD的语义处理

- S-属性的SDD
  - 只包含综合属性
  - S for synthesized
- L-属性的SDD
  - 可以包含综合属性，也可以包含继承属性
  - L for left-to-right
  - 产生式右端某文法符号的继承属性的计算只取决于该符号左边文法符号（包括产生式左边的文法符号）的属性
  - S-属性SDD是L-属性SDD的一个特例

基于SDD的语义处理

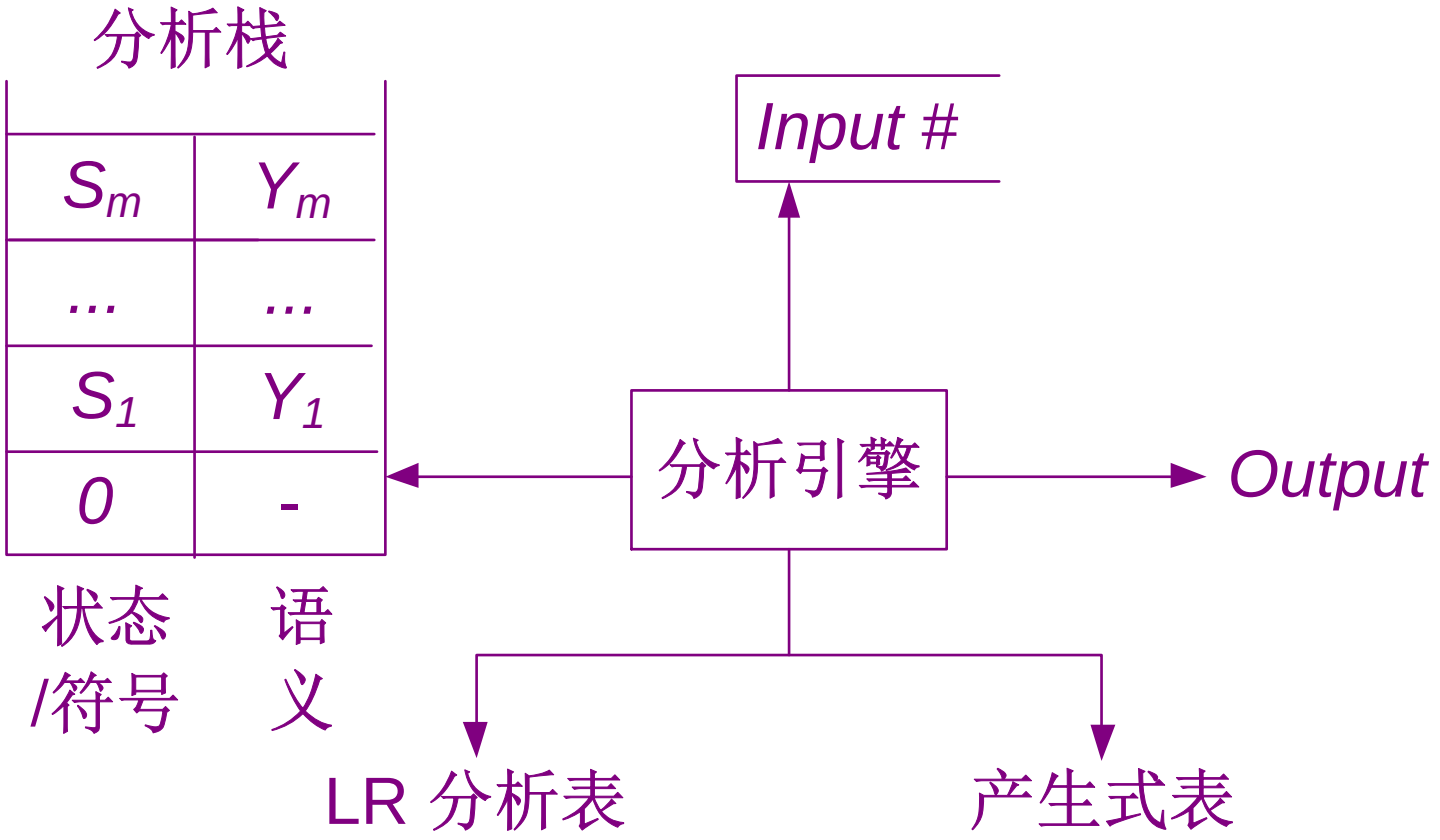
✧ S-属性的SDD的语义处理



- 通常采用自下而上的方式进行
- 若采用LR分析技术，可以
  通过扩充分析栈中的域，
  形成语义栈来存放综合属性的值，
  计算相应产生式
  左部文法符号的综合属性值刚好
  发生在每一步归约之前的时刻

基于SDD的语义处理

- 采用LR分析技术进行S-属性SDD的语义处理
  - 扩充分析栈中的域形成语义栈存放综合属性的值



基于SDD的语义处理

- 采用LR分析技术进行S-属性SDD的语义处理
  - 语义规则中的综合属性可以通过存在于当前语义栈栈顶部分的属性进行计算
  - 例如，假设有相应于产生式 **A→BCD** 的语义规则

    A.a := f（B.b, C.c, D.d）

    在 **BCD** 归约为 **A** 之前，**B.b, C.c,**和 **D.d**分别存放于语义栈的 **top-2**，**top-1** 和 **top** 的相应域中，因此 **A.a** 可以顺利求出

    归约后，**B.b, C.c, D.d**被弹出，而在栈顶 **top** 的位置上存放 **A.a**。

基于SDD的语义处理

✧ 用LR分析技术进行S-属性SDD的语义处理举例

– 通过下列S-属性SDD G'[S]为常量表达式求值

产生式	语义规则
$S \rightarrow E$	{ *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* }
$F \rightarrow d$	{ *F.val* := *d.lexval* }

基于SDD的语义处理

- 文法G'[S] 的LR分析表

（0）$S \to E$ （1）$E \to E+T$ （2）$E \to T$
（3）$T \to T*F$ （4）$T \to F$
（5）$F \to (E)$ （6）$F \to d$

状态	ACTION						GOTO		
	d	*	+	(	)	#	*E*	*T*	*F*
0	s5			s4			1	2	3
1			s6			acc			
2		s7	r2		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8			s6		s11				
9		s7	r1		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

基于SDD的语义处理

✧ LR分析过程伴随常量表达式2 + 3 * 5的求值

（0）$S \to E$ （1）$E \to E+T$ （2）$E \to T$
（3）$T \to T*F$ （4）$T \to F$
（5）$F \to (E)$ （6）$F \to d$

分析栈（状态，语义值）	余留输入串	动作	语义规则
0 –	2 + 3 * 5 #	s5	
0 – 5 2	+ 3 * 5 #	r6	F.val := d.lexval
0 – 3 2	+ 3 * 5 #	r4	T.val := F.val
0 – 2 2	+ 3 * 5 #	r2	E.val := T.val
0 – 1 2	+ 3 * 5 #	s6	
0 – 1 2 6 –	3 * 5 #	s5	
0 – 1 2 6 – 5 3	* 5 #	r6	F.val := d.lexval
0 – 1 2 6 – 3 3	* 5 #	r4	T.val := F.val
0 – 1 2 6 – 9 3	* 5 #	s7	
0 – 1 2 6 – 9 3 7 –	5 #	s5	
0 – 1 2 6 – 9 3 7 – 5 5	#	r6	F.val := d.lexval
0 – 1 2 6 – 9 3 7 – 10 5	#	r3	$T.val := T_1.val \times F.val$
0 – 1 2 6 – 9 15	#	r1	$E.val := E_1.val + T.val$
0 – 1 17	#	acc	print(E.val)

基于SDD的语义处理

✧ L-属性SDD的语义处理

– 采用自上而下的方式可以较方便地进行

– 可以采用下列基于深度优先后序遍历的算法

```
procedure dfvisit(n: node);
  begin
    for n 的每一孩子m, 从左到右 do
        begin
            计算 m 的继承属性值;
            dfvisit(m)
        end;
        计算n的综合属性值
  end
```

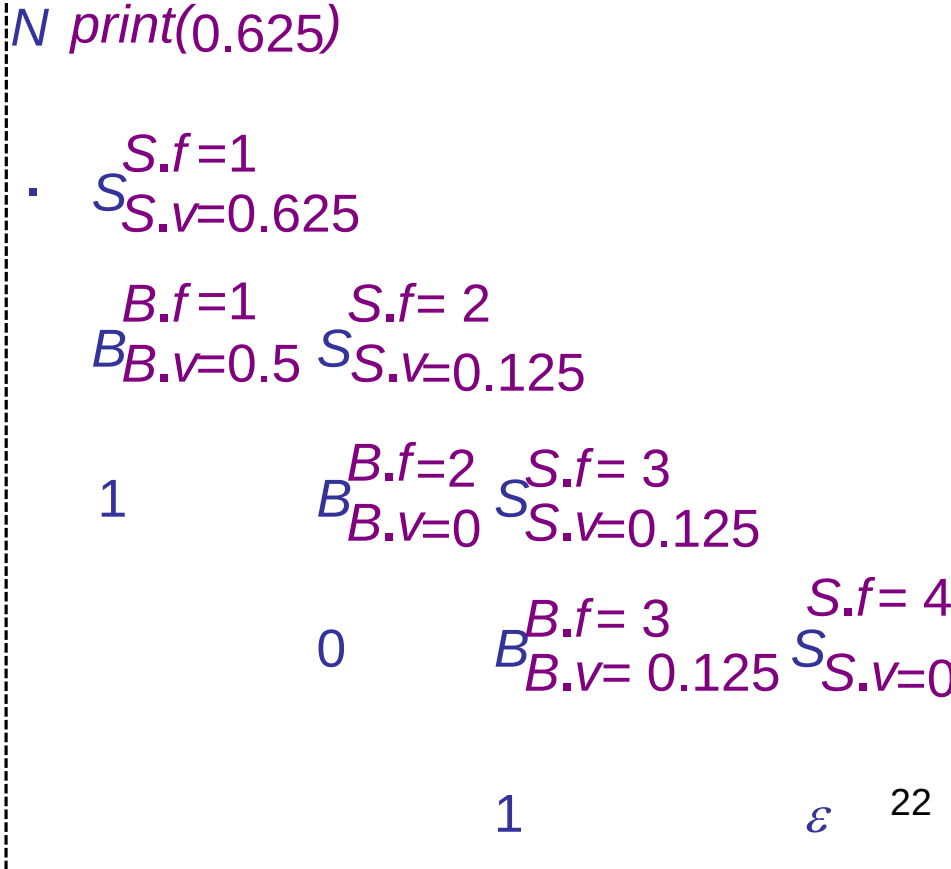
– 该算法与自上而下预测分析过程对应. 因此,基于 LL(1) 文法的 L-属性SDD可以采用这种方法进行语义处理.
（随后将结合翻译方案的进一步讨论分析程序的构造）

基于SDD的语义处理

- 采用基于深度优先后序遍历算法进行 L-属性 SDD 的语义处理举例
  - 考虑对于下列L-属性SDD，输入串为 .101 时的处理过程

产生式	语义规则
$N \to .S$	{ $S.f := 1$; $print(S.v)$ }
$S \to BS_1$	{ $S_1.f := S.f+1$; $B.f := S.f$; $S.v := S_1.v+B.v$ }
$S \to \varepsilon$	{ $S.v := 0$ }
$B \to 0$	{ $B.v := 0$ }
$B \to 1$	{ $B.v := 2^{-B.f}$ }

继承属性：
综合属性：



翻译方案

✧ 语法制导的翻译方案（Syntax-Directed Translation Scheme, SDT）概念

- 在产生式体中嵌入了程序片段的CFG
- 翻译方案 / 翻译模式；语义动作
- 适合语法制导语义处理的另一种描述形式
- 可以体现一种合理调用语义规则的翻译算法
- 形式上类似于SDD，但允许由{ }括起来的语义规则集合出现在产生式右端的任何位置. 优点在于可显式地表达动作和属性计算的次序，而在前述的SDD中不体现这种次序

✧ 受限的翻译方案

– 在设计翻译方案时，必须作某些限制，以确保每个属性值在被访问到的时候已经存在

– 这里仅讨论两类受限的翻译方案

- 受S-属性SDD的启示，对于仅需要综合属性的情形，只要创建一个语义规则集合，放在相应产生式右端的末尾，把属性的计算规则加入其中即可

- 受L-属性定义的启示，对于既包含继承属性又包含综合属性的情形，必须注意：

（1）产生式右端某个符号的继承属性的计算必须位于该符号之前；

（2）每个计算规则不访问位于它右边符号的综合属性；

（3）产生式左部非终结符的综合属性的计算只能在所用到的属性都已计算出来之后进行，通常放在相应产生式右端的末尾

✧ 翻译方案举例

– 定点二进制小数转换为十进制小数

$N \rightarrow .\ \{ S.f := 1 \}\ S\ \{ print(S.v) \}$

$S \rightarrow \{ B.f := S.f \}\ B\ \{ S_1.f := S.f + 1 \}\ S_1\ \{ S.v := S_1.v + B.v \}$

$S \rightarrow \varepsilon\ \{ S.v := 0 \}$

$B \rightarrow 0\ \{ B.v := 0 \}$

$B \rightarrow 1\ \{ B.v := 2^{-B.f} \}$

翻译方案

- 基于翻译方案的语义处理（翻译）
  - 仅考虑单遍的方法
    - 自上而下的语义处理（翻译）

      借助于自上而下的预测分析技术
    - 自下而上的语义处理（翻译）

      借助于自下而上的移进–归约分析技术
  - 仅考虑上述的受限翻译方案，

✧ 基于翻译方案的自上而下语义处理（翻译）

– 对适合于自上而下预测技术的翻译方案，语法制导的语义处理（翻译）程序可以如下构造

- 对每个非终结符 A，构造一个函数，以 A 的每个继承属性为形参，以A的综合属性为返回值（若有多个综合属性，可返回记录类型的值）。如同预测分析程序的构造，该函数代码的流程是根据当前的输入符号来决定调用哪个产生式。
- 与每个产生式相关的代码根据其右端的结构来构造（见下页）

✧ 基于翻译方案的自上而下语义处理（翻译）

– 语法制导的语义处理（翻译）程序的构造中，与每个产生式相关的代码根据产生式右端的终结符，非终结符，和语义规则集（语义动作），依从左到右的次序完成下列工作：

- 对终结符 x，调用匹配终结符（match_token），并继续输入。
- 对非终结符 B，利用相应于 B 的函数调用产生赋值语句 $c:=B(b_1, b_2, \ldots, b_k)$，其中变量 $b_1, b_2, \ldots, b_k$ 对应 B的各继承属性，变量c对应B的综合属性
- 对语义规则集，直接copy其中每一语义规则（动作）来产生代码，只是将对属性的访问替换为对相应变量的访问。

✧ 基于翻译方案的自上而下语义处理举例

– 构造下列翻译方案的自上而下预测分析程序（可以验证其基础文法为 LL（1）文法）

$N \rightarrow .\ \{ S.f := 1 \}\ S\ \{ print(S.v) \}$

$S \rightarrow \{ B.f := S.f \}\ B\ \{ S_1.f := S.f + 1 \}\ S_1\ \{ S.v := S_1.v + B.v \}$

$S \rightarrow \varepsilon\ \{ S.v := 0 \}$

$B \rightarrow 0\ \{ B.v := 0 \}$

$B \rightarrow 1\ \{ B.v := 2^{-B.f} \}$

✧ 基于翻译方案的自上而下语义处理举例

– 根据产生式

$$N \rightarrow . \ \{ S.f := 1 \} \ S \ \{ print(S.v) \}$$

对非终结符 *N*，构造如下函数

```
void ParseN()
{
    MatchToken('.');        //匹配 '.'
    Sf : =1;                //变量 Sf 对应属性S.f
    Sv : = ParseS(Sf);      //变量 Sv 对应属性S.v
    print(Sv);
}
```

✧ 基于翻译方案的自上而下语义处理举例

– 根据产生式

$S \rightarrow \{ B.f := S.f \}\ B\ \{ S_1.f := S.f + 1 \}\ S_1\ \{ S.v := S_1.v + B.v \}$

$S \rightarrow \varepsilon\ \{ S.v := 0 \}$

对非终结符 S，构造如下函数

```
float ParseS( int f )
{
    if (lookahead=='0' or lookahead=='1' )  {
            Bf : = f;   Bv : = ParseB(Bf); S1f := f +1 ;
            S1v := ParseS(S1f); Sv := S1v + Bv;
    }
    else if (lookahead== '#' )  Sv := 0;
    else { printf("syntax error \n"); exit(0); }
    return Sv;
}
```

✧ 基于翻译方案的自上而下语义处理举例

– 根据产生式

$B \rightarrow 0$ { $B.v := 0$ }
$B \rightarrow 1$ { $B.v := 2^{-B.f}$ }

对非终结符 *B*，构造如下函数

```
float ParseB( int f )
{
    if (lookahead=='0')  {MatchToken('0'); Bv : = 0;}
    else if  (lookahead== '1' )  {
            MatchToken('1'); Bv := 2^(-f);
    }
    else { printf("syntax error \n"); exit(0); }
    return Bv;
}
```

✧ 基于翻译方案的自下而上语义处理（翻译）

- 扩展前述的关于S-属性SDD的自下而上处理技术（即在分析栈中增加存放属性值的域）
  - 翻译方案中综合属性的求值采用前述的处理方法
- 对于前述受限的翻译方案，核心问题可转化为L-属性SDD 的自下而上处理，该问题的讨论较复杂，本讲仅涉及如下 3 个方面的简介
  - 翻译方案中去掉嵌在产生式中间的语义规则
  - 分析栈中继承属性的访问
  - 继承属性的模拟求值

✧ 基于翻译方案的自下而上语义处理（翻译）

– 从翻译方案中去掉嵌在产生式中间的语义规则集

- 若语义规则集中未关联任何属性，引入新的非终结符N和产生式$N \rightarrow \varepsilon$，把嵌入在产生式中间的动作用非终结符$N$代替，并把该语义规则集放在产生式后面
- 若语义规则集中有关联的属性，引入新的非终结符N和产生式$N \rightarrow \varepsilon$，以及把该语义规则集放在产生式后面的同时，需要在适当的地方增加复写规则（可参照稍后关于分析栈中继承属性的模拟求值的解决方案）
- 目的：使所有嵌入的语义规则都出现在产生式的末端，以便自下而上处理继承属性

✧ 基于翻译方案的自下而上语义处理（翻译）

– 从翻译方案中去掉嵌在产生式中间的语义规则集举例

$E \rightarrow T\ R$
$R \rightarrow +\ T\ \{ print(\text{‘+’}) \}\ R_1$
$R \rightarrow -\ T\ \{ print(\text{‘–’}) \}\ R_1$
$R \rightarrow \varepsilon$
$T \rightarrow \underline{num}\ \ \{ print(\underline{num}.val) \}$

$E \rightarrow T\ R$
$R \rightarrow +\ T\ M\ R_1$
$R \rightarrow -\ T\ N\ R_1$
$R \rightarrow \varepsilon$
$T \rightarrow \underline{num}\ \ \{ print(\underline{num}.val) \}$
$M \rightarrow \varepsilon\ \{ print(\text{‘+’}) \}$
$N \rightarrow \varepsilon\ \{ print(\text{‘–’}) \}$

✧ 基于翻译方案的自下而上语义处理（翻译）

– 分析栈中继承属性的访问

- 自下而上语义处理程序根据产生式 $A \rightarrow XY$ 的归约过程中，假设X的综合属性 $X.s$ 已经出现在语义栈上.
- 因为在Y 以下子树的任何归约之前，$X.s$的值一直存在，因此它可以被Y 访问.
- 如果用复写规则$Y.i:=X.s$ 来定义 Y 的继承属性$Y.i$，则在需要$Y.i$ 时，可以使用$X.s$

✧ 基于翻译方案的自下而上语义处理（翻译）

– 分析栈中继承属性的访问举例

翻译方案

$D \rightarrow T$ { $L.in := T.type$ } L

$T \rightarrow \underline{int}$ { $T.type := integer$ } | $\underline{real}$ { $T.type := real$ }

$L \rightarrow$ { $L_1.in := L.in$ } L_1 , v { $addtype(v.entry, L.in)$ }

$L \rightarrow v$ { $addtype(v.entry, L.in)$ }

产生式	依产生式归约时语义处理的代码片断
$D \rightarrow T\ L$	
$T \rightarrow \underline{int}$	$val\ [top] := integer$
$T \rightarrow \underline{real}$	$val\ [top] := real$
$L \rightarrow L\ ,\ v$	$addtype(val\ [top]\ ,\ val\ [top-3])$
$L \rightarrow v$	$addtype(val\ [top]\ ,\ val\ [top-1])$

继承属性：

综合属性：

(分析栈*val* 存放文法符号的综合属性，*top*为栈顶指针)

✧ 基于翻译方案的自下而上语义处理（翻译）

– 继承属性的模拟求值

- 从上面的讨论可知，分析栈中继承属性的访问是通过栈中已有文法符号的综合属性值间接进行的，因此设计翻译方案时需要做到的一点就是要保证继承属性总可以通过某个文法符号的综合属性体现出来
- 必要时，通过增加新的文法符号以及相应的复写规则常常可以达到上述目的

✧ 基于翻译方案的自下而上语义处理（翻译）

– 继承属性的模拟求值举例

考虑如下翻译方案：

$$S \rightarrow a\ A\ \{C.i := A.s\}\ C \mid b\ A\ B\ \{C.i := A.s\}\ C$$
$$C \rightarrow c\ \{C.s := g(C.i)\}$$

若直接应用上述复写规则的处理方法，则在使用 $C \rightarrow c$ 进行归约时，C.i 的值或存在于次栈顶（*top-1*），或存在于次次栈顶（*top-2*），不能确定用哪一个.

一种可行的做法是引入新的非终结符 *M*，将以上翻译方案改造为：

$$S \rightarrow a\ A\ \{C.i := A.s\}\ C \mid b\ A\ B\ \{M.i := A.s\}\ M\ \{C.i := M.s\}\ C$$
$$C \rightarrow c\ \{C.s := g(C.i)\}$$
$$M \rightarrow \varepsilon\ \{M.s := M.i\}$$

这样，在使用**C → *c*** 进行归约时，C.i 的值就一定可以通过访问次栈顶（*top-1*）得到

✧ 基于翻译方案的自下而上语义处理（翻译）

– 继承属性的模拟求值举例

考虑如下翻译方案：

$$S \rightarrow a\ A\ \{C.i := f(A.s)\}\ C$$

这里，继承属性 C.i 不是通过复写规则来求值，而是通过普通函数 f(A.s) 调用来计算. 在计算 C.i 时，A.s 在语义栈上，但 f(A.s)并未存在于语义栈.
同样，一种做法是引入新的非终结符*M*，将以上翻译方案改造为：

$$S \rightarrow a\ A\ \{M.i := A.s\}\ M\ \{C.i := M.s\}\ C$$
$$M \rightarrow \varepsilon\ \{M.s := f(M.i)\}$$

这样，就解决了上述问题。

注：从翻译方案中去掉嵌在产生式中间的语义规则集时，若语义规则集中有关联的属性，则可参照此例的解决方案

类型检查

✧ 静态语义分析的主要工作

– 静态语义检查

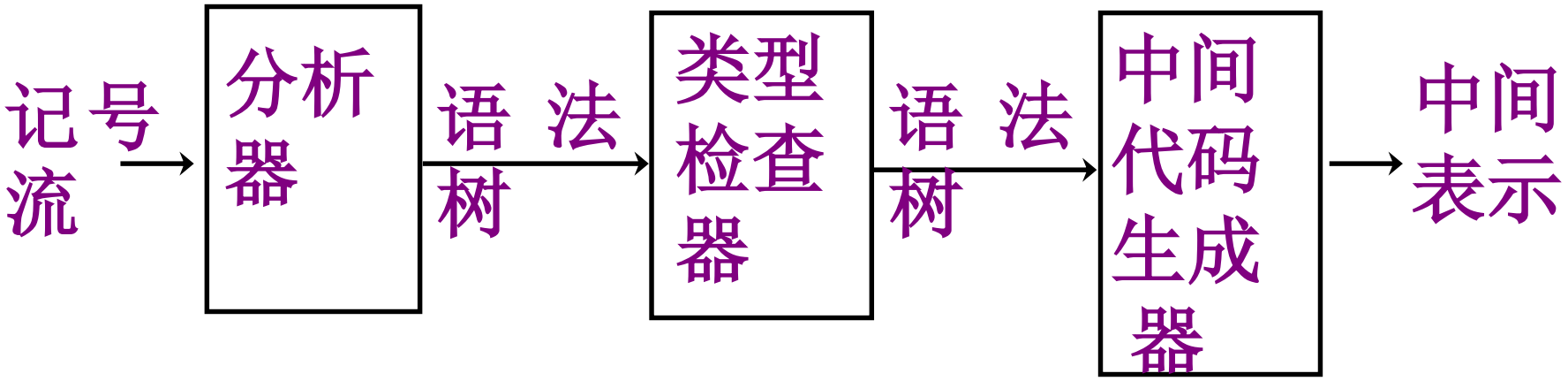
- 类型检查（***type checks***）
  检查每个操作是否遵守语言类型系统的定义
- 名字的作用域（***scope***）分析
  建立名字的定义和使用之间联系
- 控制流检查（***flow-of-control checks***）
  控制流语句必须使控制转移到合法的地方（如 ***break*** 语句必须有合法的语句包围它）
- 唯一性检查（***uniqueness checks***） 很多场合要求对象只能被定义一次（如枚举类型的元素不能重复出现）
- 名字相关检查（***name-related checks***）
  （如，一些名字可能被要求配对出现）
- ……

✧ 语义处理的环境

– 符号表（*symbol tables*）

- 名字信息建立后加入/更改符号表
  名字信息如：类型，偏移地址，占用空间等
- 需要获取名字信息时，查找符号表
- 符号表的组织可以体现名字作用域规则

✧ 类型检查（自学）



Chapter 6
符号表与运行时存储组织

符号表

- 符号表的作用
- 符号表的常见属性
- 关于符号表的操作
- 符号表的实现

符号表

✧ 符号表的作用

- 用来存放有关标识符的属性信息
  - 这些信息会在编译的不同阶段用到
  - 在语义分析中，符号表所登记的内容将用于语义检查和产生中间代码
  - 在目标代码生成阶段，符号表是对符号名进行地址分配的依据
  - 对一个多遍扫描的编译程序，不同遍所用的符号表也会有所不同，因为每遍所关心的信息会有差异
- 用来体现作用域信息

符号表

✧ 符号表的常见属性

- 符号名
- 符号的类型
- 符号的存储类别和存储分配信息
- 符号的作用域/可见性
- 其他属性
  - 数组内情向量
  - 记录结构的成员信息
  - 函数及过程的形参

符号表

✧ 关于符号表的操作

- 创建符号表　在编译开始，或进入一个分程序
- 插入表项　在遇到新的标识符声明时进行
- 查询表项　在引用标识符时进行
- 修改表项　在获得新的语义值信息时进行
- 删除表项　在标识符成为不可见/不再需要它的任何信息时进行
- 释放符号表空间　在编译结束前或退出一个分程序

符号表

✧ 符号表的实现

– 表项属性信息的组成

- 把属性种类完全相同的那些符号组织在一起，构造出多张符号表：常数表、变量名表、过程名表，标号表，等等
- 把程序中的所有符号都组织在一张符号表中，组成一张包括了所有属性的庞大的符号表

✧ 符号表的实现

– 实现符号表的常用数据结构

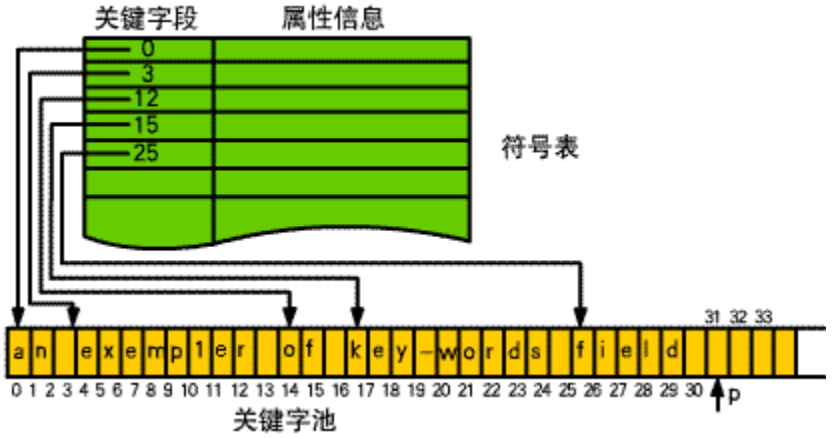
- 一般的线性表
  如：数组、链表，等
- 有序表
  查询较无序表快，如可以采用折半查找
- 二叉搜索树
- Hash表

符号表

✧ 符号表的实现

– 名字域（表项的关键字域）的组织

- 可采用关键字池解决名字串长短不一带来空间浪费的问题



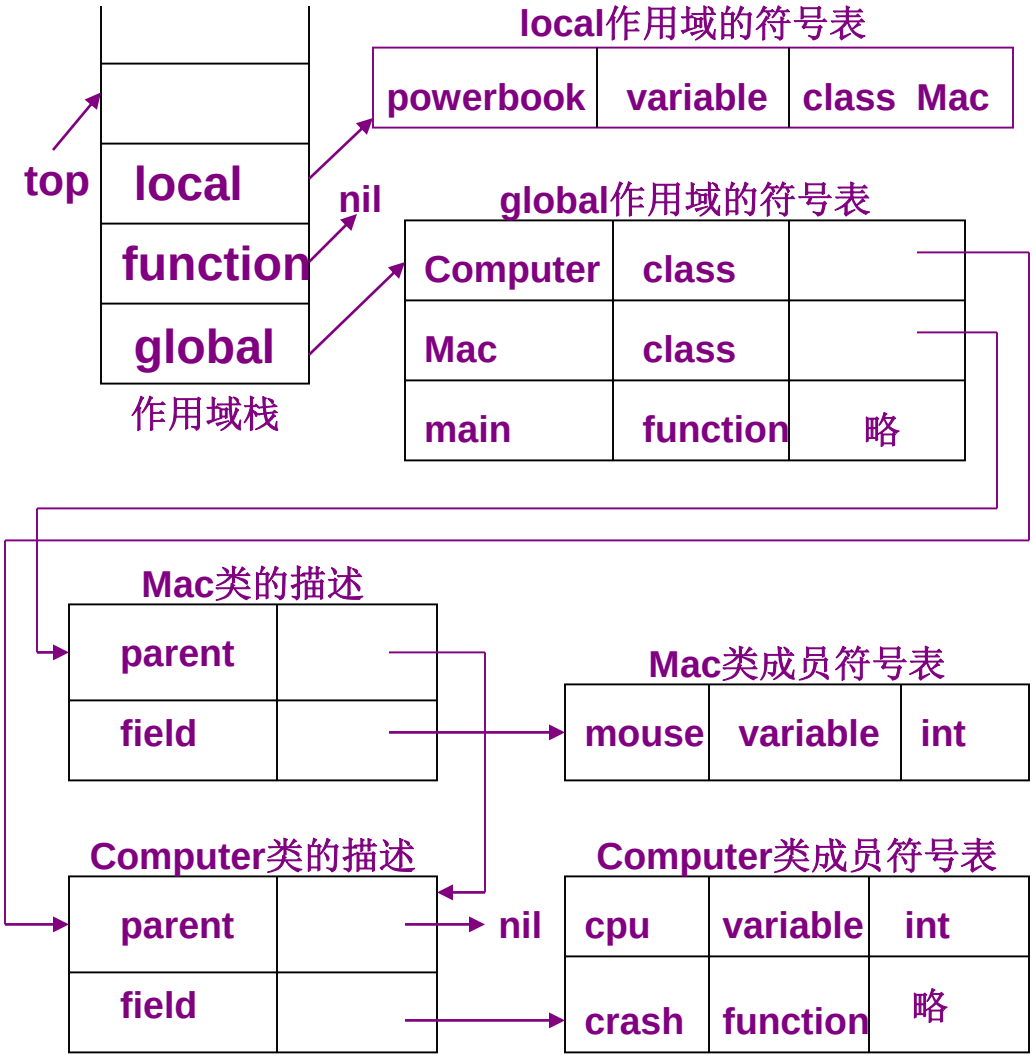
– 其他域的组织

- 解决好不等长属性值问题

一般不把所有属性值都放在符号表项的某个域中，而是另辟空间存放属性值（如数组的内情向量）

符号表

✧ 某编译器的符号表实例（处理到第13行时的情形）




```
(1) class Computer {
(2)         int cpu;
(3)         void Crash(int nTimes) {
(4)                 int i;
(5)                 …
(6)          }
(7) }
(8) class Mac extends Computer {
(9)         int mouse;
(10)        }
(11) void main() {
(12)        class Mac powerbook;
(13)        powerbook.Crash(2);
(14)        …
(15) }
```

运行时存储组织

- ✧ 运行时存储组织的作用与任务
- ✧ 程序在存储器中的布局
- ✧ 存储分配策略
- ✧ 活动记录
- ✧ 垃圾回收（自学）

运行时存储组织

✧ 运行时存储组织的作用与任务

- 代码生成前如何安排目标机资源的使用
- 几个问题
  - 数据表示 如何在目标机中表示每个源语言类型的值
  - 存储分配 如何组织不同作用域变量的存储
  - 表达式计算 如何组织表达式的计算
  - 过程实现 如何以例程实现过程/函数调用，参数传递

运行时存储组织

✧ 数据表示

– 源程序中数据对象在内存或寄存器中的表示形式

- 源程序中数据对象的属性

  名字（*name*），类型（*type*），值（*value*），
  成分（*component*），偏移地址（*offset*），……

- 数据对象在内存或寄存器中的表示形式

  位、字节、字、字节序列、……

- 有些机器要求数据存放时要按某种方式对齐（*align*）

  如：要求所有数据存放的起始地址为能够被4整除

运行时存储组织

✧ 数据表示举例

– 基本类型数据

char 数据 1 *byte* *integer* 数据 4 *bytes*
float 数据 8 *bytes* *boolean* 数据 1 *bit* / 1 *byte*

指针 4 *bytes*

数组 一块连续的存储区（按行/列存放）

结构/记录 所有域（field)存放在一块连续的存储区

对象 实例变量像结构的域一样存放在一块连续的存储区，方法（成员函数）存放在其所属类的代码区

运行时存储组织

✧ 程序在存储器中的布局（*layout*）

– 典型的程序布局

- 代码段

  存放目标代码

- 静态数据段

  静态存放全局数据

- 动态数据段

  运行时动态变化的堆区和栈区

运行时存储组织

✧ 存储分配策略

- 静态分配
  - 在编译期间为数据对象分配存储
- 动态分配
  - 栈式分配

    将数据对象的运行时存储按照栈的方式来管理
  - 堆式分配

    从数据段的堆空间分配和释放数据对象的运行时存储

✧ 静态存储分配

- 在编译期间就可确定数据对象的大小
  - 不能处理递归过程或函数
- 某些语言中所有存储都是静态分配
  - 如汇编语言，FORTRAN语言
- 多数语言只有部分存储进行静态分配
  - 可静态分配的数据对象如大小固定且在程序执行期间可全程访问的全局变量，以及程序中的常量（*literals*，*constants*）
  - 如 C 语言中的 static 和 extern 变量

✧ 栈式存储分配

- 用于有效实现层次嵌套的程序结构
  - 如实现过程/函数，块层次结构
- 可以实现递归过程/函数
  - 比较：静态分配无法实现递归过程/函数
- 运行栈中的数据单元是活动记录（*activation record*）（专门介绍）

运行时存储组织

✧ 堆式存储分配

- 从堆空间为数据对象分配/释放存储
  - 灵活 数据对象的存储分配和释放不限时间和次序
- 显式的分配或释放（*explicit allocation / dealocation*）
  - 程序员负责应用程序的（堆）存储空间管理（借助于）编译器和运行时系统所提供的默认存储管理机制）
- 隐式的分配或释放（*implicit allocation / dealocation*）
  - （堆）存储空间的分配或释放不需要程序员负责，由编译器和运行时系统自动完成

✧ 堆式存储分配

– 堆空间的管理

- 分配算法　面对多个可用的存储块，选择哪一个

  如：最佳适应算法（选择浪费最少的存储块）

  最先适应算法（选择最先找到的足够大的存储块）

  循环最先适应算法（起始点不同的最先适应算法）

- 碎片整理算法　压缩合并小的存储块，使其更可用

（ 部分内容可参考数据结构和操作系统课程）

Conclusions

- 基于SDD的语义处理
  - ✓基于树遍历方法的语义处理
  - ✓单遍的方法
    - ✓自下而上
    - ✓自上而下
- 翻译模式
  - ✓自上而下的翻译
  - ✓自下而上的翻译
- 类型检查
- 符号表
- 运行时存储组织

- § 5.2 Evaluation Orders for SDD's
- § 5.3 Applications of Syntax-Directed Translation
- § 5.4 Syntax-Directed Translation Schemes
- § 5.5 Implementing L-Attributed SDD's
- § 7.1 Storage Organization

课外学习建议

✧ Floyd, R. W. Syntactic Analysis and Operator Precedence. J. ACM. 10:3, 316-333. 1961

✧ Engelfriet, J. Attribute Evaluation Methods. *Methods and Tools for Compiler Construction*. 103-138. 1984

Thank you!