第九章 语法制导的语义分析

2024

什么是语义?

- 语言是表达与交流的手段
  - 表达问题如何求解
  - 与机器交流
- 语义?
  - 语义性质
  - 合法性

- 程序的合法性与许多语义性质有关，如变量在声明的作用域内使用是合法的，表达式的类型是由子表达式类型以及操作来决定的等。
- 类型规则、类型检查，如函数原型、数组原型等文法不容易限定的情况（太繁）

语义分析的任务

```
1.  class Myclass implements MyInterface {
2.    string myInteger;
3.    void doSomething()
4.    {
5.      int[] x = myInteger * y;
6.    }
7.    void doSomething ()
8.    {
9.    }
10.   int fibonacci (int n)
11.   {
12.     return doSomething () + fibonacci(n-1);
13.   }
14. }
```

语义错误：

第1行MyInterface没有说明；

第4行myInteger是字符串不能做乘法；变量y没有说明

第6行 重定义了该函数，不允许；

第9行doSomething()是void类型不能做加法；

最后没有定义主函数。

语义分析任务不可替代

- 语义分析不仅仅是找语义错误。直觉地，语义分析就是验证程序没有违背语义性质，若通过了这些验证就说明程序有良定义的语义。语义性质广泛存在于程序语言的规格说明中,不同于词法、语法规则都有形式定义,对于语义规则在更多情况下采用自然语言描述。
- 虽然语义形式化方法如指称语义、公理语义、操作语义等可用于描述程序设计语言的语义，但是难以满足实际语言的要求，因而需要进行具体的设计以形成语义规则。
- 程序经过了语义分析,就终于可以说它是合法的了。但要注意程序的合法性与正确性不完全是同一个概念。

程序的合法性与正确性

```
int main() {
  string x;
  if (false) { x = 137; }
}
```

赋值语句中x与整数137类型不一致，所以该程序段的语义不合法. 当然该语句是死代码，所以这段程序仍然是类型安全的。

```
int Fibonacci(int n) {
  if (n <= 1) return 0;
  return Fibonacci(n – 1) + F
       ibonacci(n – 2);
}
int main() {
  Print(Fibonacci(40));
}
```

其中return语句中的0会导致程序结果在n=1时错误，所以它是不正确的程序. 但是该程序的语义是正确的。

语义分析的细化任务

- 总而言之，语义分析是验证程序的合法性，并不能解决程序正确性方面的问题。
- 但是，发现语义错误或验证了程序合法性还不是语义分析的全部任务，还有更为常见的任务是收集程序的有用信息供后续编译阶段使用，比如表达式的类型转换、变量的类型、名字的作用域等。
- 另外，对语义分析而言，还有一个重要目标是生成中间语言代码，就是就将合法的高级语言句子转换为中间表示。这些中间表示采取的形式有抽象语法树、三地址代码等，本质上跟语法树有相关性。

本章内容

- 认识语义分析任务
- 符号表
- 属性文法
- SLR(1)制导的属性求值框架
- 对程序声明的语义分析
  - 简单变量声明
  - 数组声明
  - 函数声明

9.2 符号表

- 编译过程中编译程序需要汇集和反复查证出现在源程序中各个名字的属性和特征等有关信息，这些信息通常记录在符号表中。
- 每个符号表都由表头和0到多个登记项组成，登记项包含两个部分：一部分是名（标识符）；另一部分是名相关语义信息。
- 一般地，与名相关的语义信息指名的类型以及类型特有信息，类型不同，名的类型特有信息也会多少不一。
- 登记在符号表中的信息为各个编译阶段所使用。
  - 在语义分析阶段用于合法性检查和产生中间代码（检查一个名字的使用和原先的声明是否一致）。
  - 在目标语言代码生成阶段，当对符号名进行地址分配时，符号表是地址分配的依据。
- 为符号表提供相应操作以管理表自身及访问表内容。

名字的类型和类型特有信息

- 登记在符号表中的名字都有类型。
- 类型特有信息也在符号表中。
- 符号表提供用来管理这些信息的操作函数：
  - 添加登记项bind()
  - 查找登记项及各域lookup()
  - 访问登记项各域lookup()
- 此外，符号表除了名字登记项以外还有自身的表头信息，也有相应的表头信息管理操作，如：
  - tab->width访问符号表的表头中的width域。
- 常常以函数（过程）为单位组织符号表，包括为函数创建符号表。因此称这种符号表为过程符号表。
  - newtab()创建一个符号表并初始化。

符号表设计

- 符号表可以是个结构，由表头部分和登记项组成。
  - outer　外层符号表指针
  - width　过程的存储区大小
  - argc　过程参数个数
  - arglist　过程参数表
  - rtype　返回结果类型
  - code　过程代码
  - *entry　登记项

```
struct fst {
    struct fst *outer;
    ...//<header>;
    struct tentry *entry;
};
typedef symboleTable *fst;
```

- 符号表的表头信息的访问方式取决于如何指定符号表：
  - 如果在tab中那么指针形式如tab->width，tab->outer等
  - 如果是在symtab栈顶，那么使用update[]宏进行访问，具体在后面介绍。

符号表的登记项

- 过程符号表的登记项可以有0到多个，每个登记项都有两个固定的域：
  - name域，该过程声明的名字，如变量名、函数名；
  - type域，该名字的类型，如INT，ARRAY，FUNC等。
- 函数bind(x,type)用于给符号表添加一个登记项，它的名字是x，类型是type，同时这个登记项的类型特有域同时被包含在该登记项里并初始化为UNBOUND。
  - 比如，x为简单变量，那么它的登记项还包含offset域，表示x的存储单元相对地址（偏移量）。
- lookup(x,offset:,offs)将x的offset置为offs的值；
- lookup(x,offset:)返回x的offset域的值，若无值则返回UNBOUND。

符号表的数组登记项

- 数组的登记项的一些域：
  - etype元素类型
  - base存储区首地址（相对地址）
  - dims维数
  - dim[0]维长
  - ...
  - dim[dims-1]维长
- bind(a, ARRAY);
- lookup(a, dims:);
- lookup(a,dim: 0,10);
- 当翻译数组声明语句的时候创建该登记项。
- 当翻译到下标变量引用时会用到该登记项。

符号表的函数登记项

- 一些语言允许函数声明嵌套，因而内层函数会作为外层函数的一个函数登记项而出现在外层函数的符号表中。
- 如果局部函数不被允许，则所有函数都登记在同一个符号表里。
- 函数登记项的类型特有域：
  - mytab 指向这个函数自己的符号表（函数都有符号表）
  - offset为双字，当函数作为实参的时候有用，ep和ip
- 如下操作是在符号表中添加函数登记项，并记录层次关系：
  - bind(fn,FUNC);
    lookup(fn,mytab:,newtab());
- 取出函数登记项中的符号表指针：
  - tab1=lookup(fn,mytab:);

符号表的实现

- 运用数据结构课程所学知识，符号表能被优化实现，但已超出本课程范围。
- 在翻译的任何步骤，都有一个当前符号表，约定全局symtab栈顶是当前符号表；
- 可以看到，所有的bind()和lookup()操作都是针对当前符号表来工作的。
- 如果是针对指定符号表tab，tab可以是全局或局部变量，则使用bind1()和lookup1()，它们的第一个参数为符号表，其余参数对应bind()和lookup()。

- 关于符号表还可以有进一步的讨论，在用到时将予以介绍。

9.3 属性文法

- 9.3.1 属性文法概述
- 9.3.2 属性文法定义
- 9.3.3 属性求值

9.3.1 属性文法

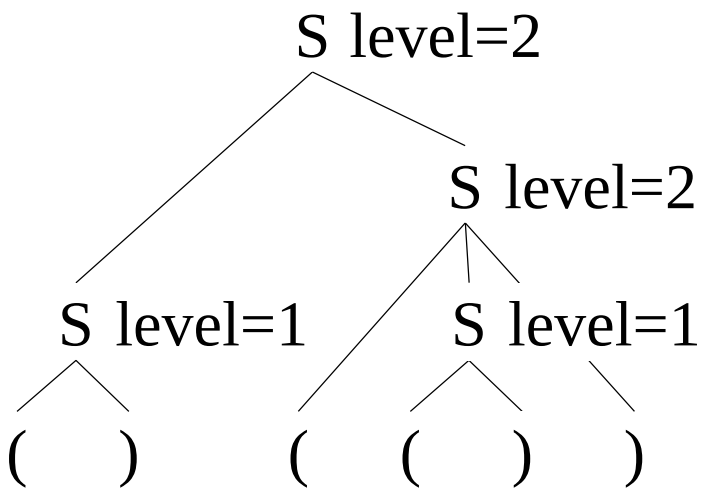
- 属性文法att(G)是对CFG G的扩展，扩展了属性和属性方程，其中属性方程就是语义规则的表现形式。
- att(G)为G的每一条产生式规则都配备0到多个属性方程，属性方程是属性求值规则的代数表达。
- 每个属性名与产生式中的一个文法符号关联，作为承载与该文法符号有关信息的容器。
- 属性方程描述了这些属性的求值规则，常常用作语义规则。这些语义规则可以推广为包含一些函数和过程来描述常规意义下的语义动作，如生成变量、标号、访问符号表、输出代码等等。

关于属性

- 属性名：任意符号名，如助记符。
- 属性：产生式中每个元素与给定属性名关联就构成一个属性类；句型中每个元素与给定属性名关联起来就构成一个属性实例。
- 属性值：属性实例的任意取值。

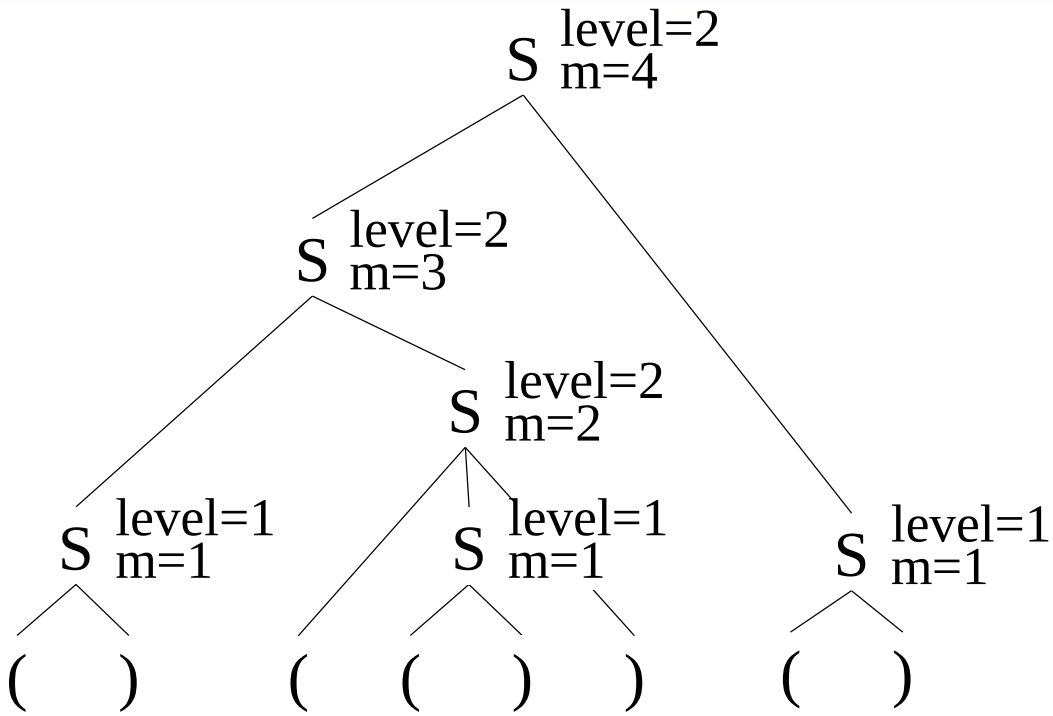
- $S \rightarrow SS$ { S[0].level = max(S[1].level, S[2].level) }
- $S \rightarrow (S)$ { S[0].level = S[1].level + 1 }
- $S \rightarrow ()$ { S.level = 1 }
- 属性名level；属性类S[i].level，i=1,2,3下标指示变元的出现
- ()(())⇨S(())⇨ S(S)⇨SS ⇨S
- 属性及其值：
- $()(())⇨S_{level=1}(())⇨ S_{level=1}(S_{level=1})⇨S_{level=1}S_{level=2} ⇨S_{level=2}$

属性的语法树表示



- $S \rightarrow SS$ { S[0].level = max(S[1].level, S[2].level) }
- $S \rightarrow (S)$ { S[0].level = S[1].level + 1 }
- $S \rightarrow ()$ { S.level = 1 }
- $()(())\mapsto S_{level=1}(())\mapsto S_{level=1}(S_{level=1})\mapsto S_{level=1}S_{level=2}\mapsto S_{level=2}$

属性的语法树表示



- $S \rightarrow SS$ { S[0].level = max(S[1].level, S[2].level) }
- $S \rightarrow (S)$ { S[0].level = S[1].level + 1 }
- $S \rightarrow ()$ { S.level = 1 }
- $()(())\mapsto S_{level=1}(())\mapsto S_{level=1}(S_{level=1})\mapsto S_{level=1}S_{level=2}\mapsto S_{level=2}$

关于属性方程

- 属性文法中，每个产生式可以有0到多个属性方程，用来决定如何对属性（实例）求值，或者说如何计算属性的值。
- 对于产生式A→XYZ可以有属性方程形如，
  - A.a=f(X.b, Y.c, Z.d)
  - Y.a=g(A.b, X.a, Y.b)
  - Y.a=h(A.a, X.b, Z.c)
  - …
- 其中a,b,c,d代表属性名，可以相同也可以不同。
- f()，g()，h()任意函数。
- S → SS　　{ S[0].level = max(S[1].level, S[2].level) }
- S → (S)　　{ S[0].level = S[1].level + 1 }
- S → ()　　{ S.level = 1 }

例：简单台式计数器

- S→En　　{print(E.val)}
- E→E+T　　{E[0].val=E[1].val+T.val}
- E→T　　{E.val=T.val}
- T→T*F　　{T[0].val=T[1].val*F.val}
- T→F　　{T.val=F.val}
- F→(E)　　{F.val=E.val}
- F→i　　{F.val=getv(i)}

属性方程的更一般形式

- $p : X \rightarrow X_1 \ldots X_n$
  - p为产生式编号，这是为了方便引用任意产生式及元素。
  - 左部变元$p[0]=X$;
  - 候选式的符号$p[i]=X_i$, $i=1,\ldots,n$

- 属性方程
  - $p[i].a_j = f_{ij}(p[0].a_1,\ldots,p[0].a_k,\ldots, p[n].a_1,\ldots, p[n].a_k)$

- 假定产生式p中所有属性类为$C_1,\ldots,C_m$, $m>0$，那么属性方程可表示为$f(C_1,\ldots,C_m)=0$，$g(C_1,\ldots,C_m)=0$，…

- 属性方程反映的是属性之间的依赖关系。

综合属性与继承属性

- 综合属性：
- $p[0].a_j = f(p[i].a_k, i>=0)$

- 继承属性：
- $p[m].a_j = f(p[i].a_k, i<=m), m>0$

- 终结符只有综合属性。

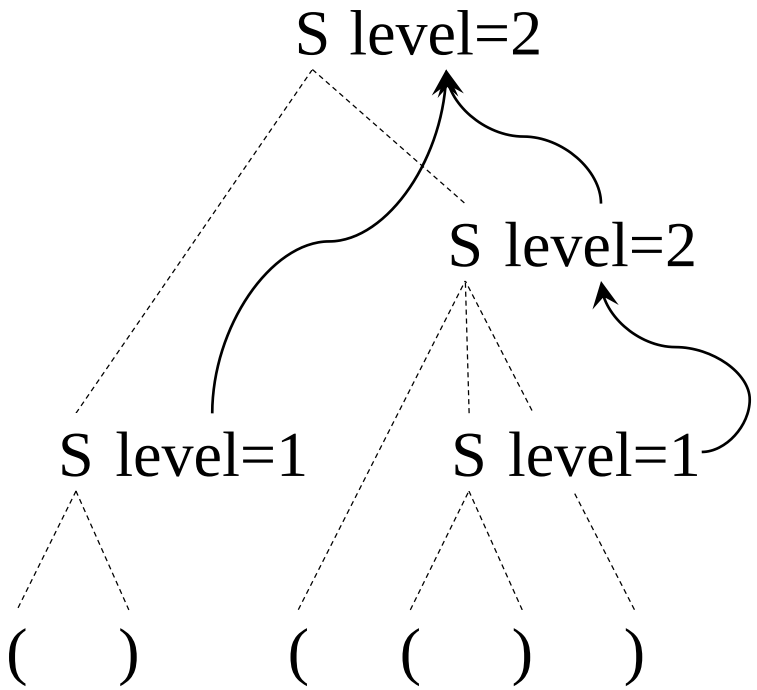
两种属性在语法树上的体现

- 属性方程反映了属性计算中的依赖关系

- 结点的综合属性的值只由孩子的一些属性来决定。
- 即，父结点的综合属性依赖于孩子结点的属性。
- 比如level属性、val属性。

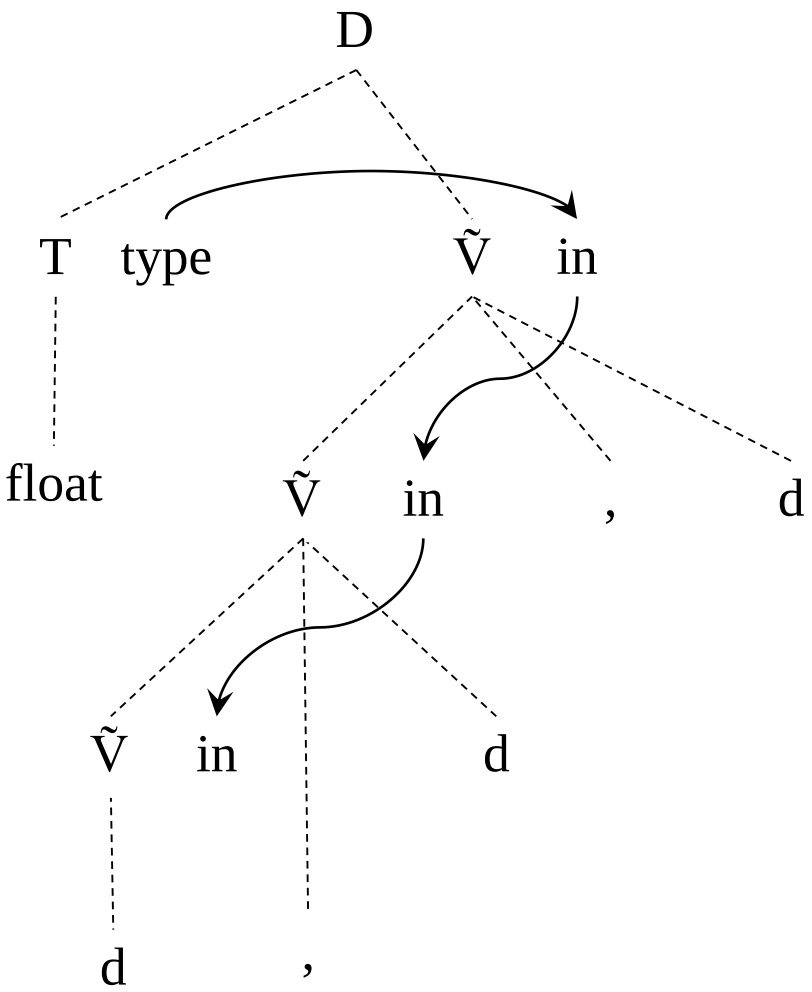
- 结点的继承属性的值由父结点属性或兄长结点属性来决定
- 或者说，结点的继承属性依赖于父结点或兄长结点的属性。
- 这种依赖是值依赖，即先算哪个后算哪个。

综合属性的依赖图



例：继承属性与依赖图

- D→TṼ
- T→int | float
- Ṽ→Ṽ,d | d



属性文法设计

- 设计属性
- 设计语义方程
- 依据属性方程计算属性的值
- 希望在语法分析过程中的归约时刻计算属性值

设计属性

- 属性是与语法单位相关的特性。
  - 语法分析识别出组成程序的语法单位以及它们之间存在的一些关系，相应地可发现有用的语义特性以及这些特性之间存在的计算关系。
- 语义特性的例子：
  - 变量的类型；
  - 表达式的值；
  - 变量在存储器中的位置；
  - 过程的代码；
  - 数中有效位数；
  - ...

一种解释

- An attribute grammar is a context-free grammar that has been extended to provide context-sensitive information by appending attributes to some of its non-terminals.
- Each distinct symbol in the grammar has associated with it a finite, possibly empty, set of attributes.
  - Each attribute has a domain of possible values.
  - An attribute may be assigned values from its domain during parsing.
  - Attributes can be evaluated in assignments or conditions.

属性的特点

- 属性所包含的信息、复杂程度、能够确定其值的时间区间等都有较大的范围。

- 编译之前确定
  - 数中有效位数; ...
- 编译过程中确定
  - ...
- 执行时确定
  - 表达式的值;
  - 动态分配的数据结构的地址; ...

属性求值

- 计算语法单位的属性的值称为属性求值（绑定），属性求值发生的时刻有编译时和运行时，
  - 静态：编译时（执行之前）
  - 动态：执行时
- 这些属性的求值时刻？
  - 变量的类型；(C/Pascal/Lisp Type Checker)
  - 表达式的值；(Code/Constant Folding)
  - 变量在存储器中的位置；(Static/Dynamic Allocation)
  - 过程的目标代码；(Static)
  - 数中有意义数字的个数。(Runtime Env.)
- 语义分析可抽象为编译时的属性求值

设计属性文法。

- 为了完成语义分析，需要做如下工作：
  - 设置一些属性，所包含信息复杂多样且多为非直接的；
  - 确定语法单位间的属性计算关系；
  - 常常伴随有副作用的行为。

- 用属性文法形式化表达出来
  - 按照定义写成属性文法。

属性求值方法

- 属性求值
  - 计算串w的语法树中每个内结点X的所有属性的值；
  - X与子结点对应于一个产生式规则，运用该规则配套的语义方程计算属性X.a的值，其中X.a应出现在方程中。

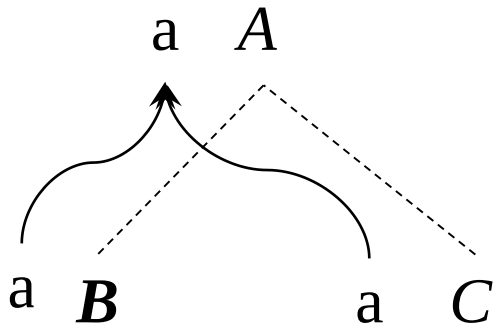
- 属性求值方法
  - 依赖图拓扑排序得到计算次序
  - 遍历语法树的求值方法
  - LL制导的求值方法
  - LR制导的求值方法
  - 一遍与多遍的求值方法

依赖图

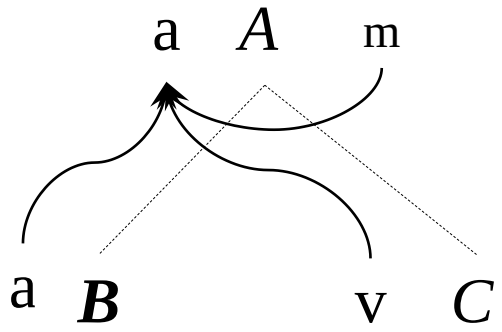
- 语法树中各属性间可能存在依赖，表现为有向无环图；
  - 若有依赖环存在表明属性文法有问题，应另行设计。
- 语法树叶子结点的综合属性值借助于词法分析（若有）。
- 语法树根结点的继承属性应该是已知的（若有）
- 依赖图拓扑排序结果为求值次序，需要遍历语法树完成各个属性计算。

综合属性的计算示意

- 语法分析树中属性X.a的值是由X的子结点属性值和X自身属性值决定。



(a) 综合属性A.a



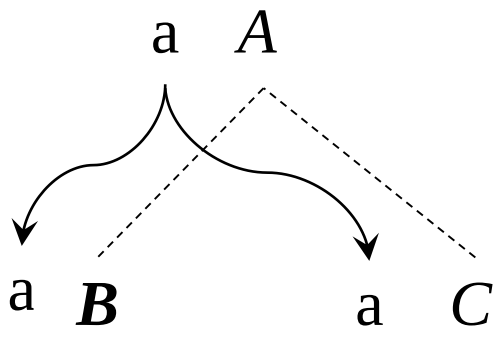
(b) 综合属性A.a

虚线：语法树

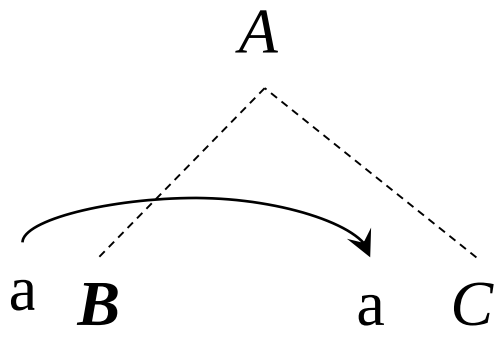
实线：属性依赖关系

继承属性的计算示意

- 不是综合属性的属性是继承属性（注意与前边不同）
  - 语法树中，一个结点的继承属性由此结点的父结点和或兄弟结点的某些属性确定。



(a) 继承自父



(b) 继承自兄

例：继承属性与依赖图

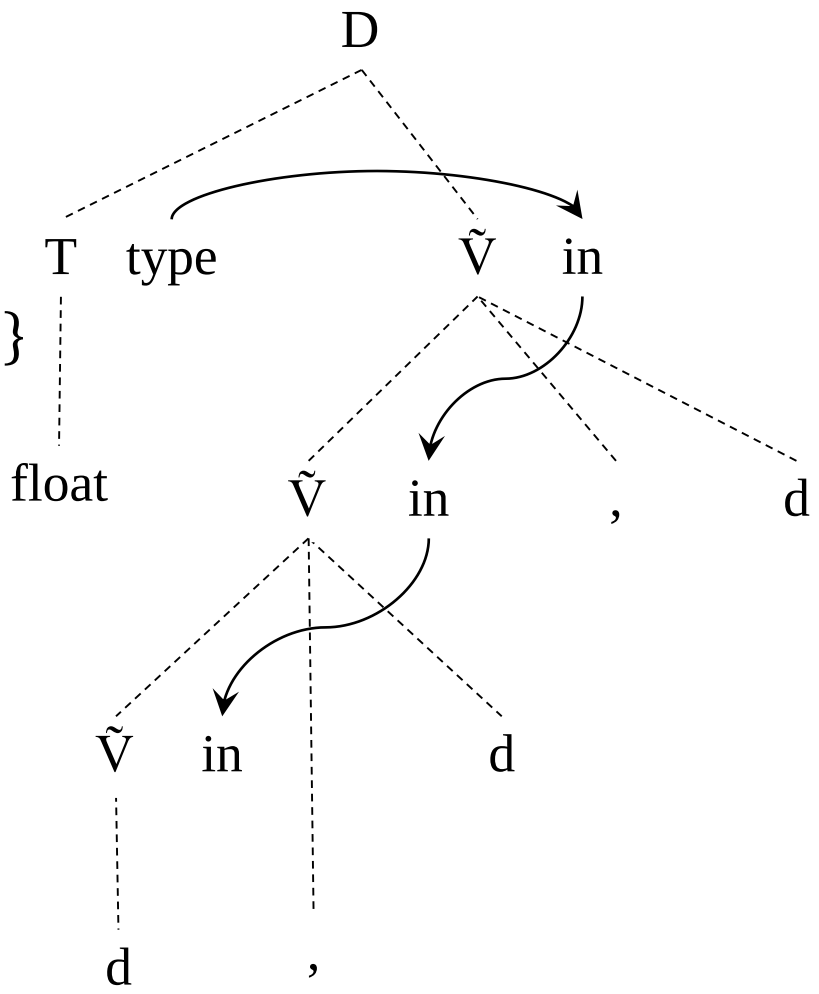
D→T$\tilde{V}$ {$\tilde{V}$.in=T.type}

T→int {T.type=i}

T→float {T.type=f}

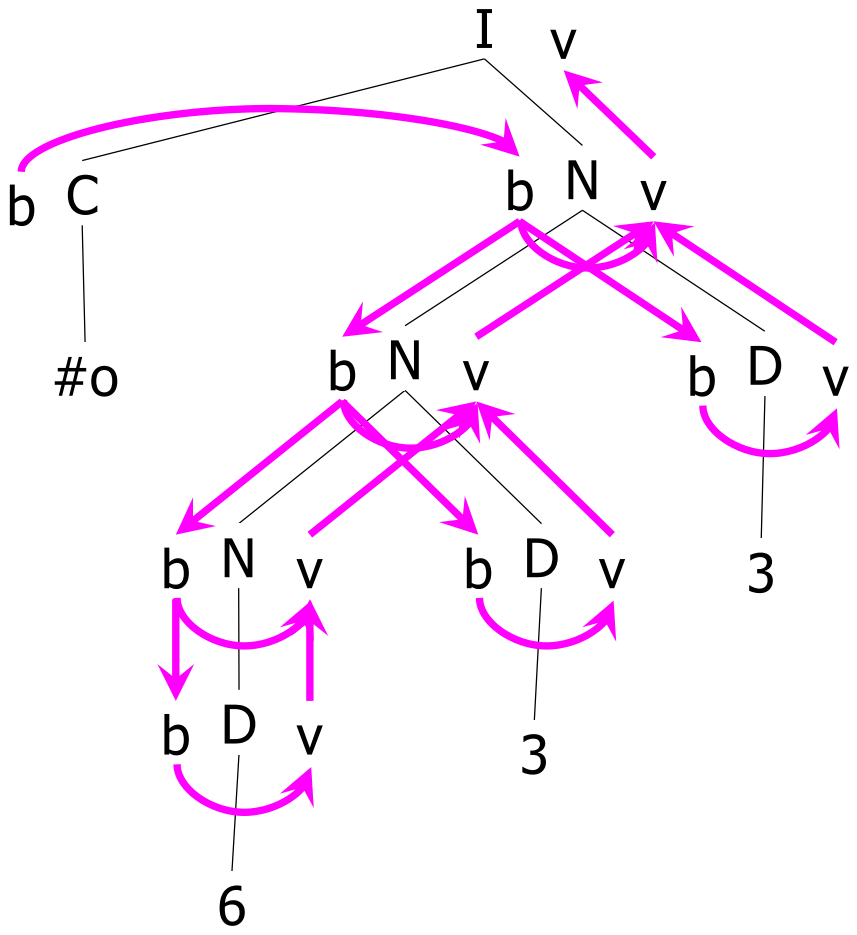
$\tilde{V}$→$\tilde{V}$,d {$\tilde{V}$[1].in=$\tilde{V}$[0].in;
bind(getn(d),$\tilde{V}$[0].in)}

$\tilde{V}$→d {bind(getn(d), $\tilde{V}$.in)}



- 先根遍历语法树。

例：一遍不能完成的依赖图



文法：
I->CN
N->ND|D
D->0|...|9
C->#o|#d

进制数#o633语法树及依赖图

在语法树上计算属性值

- 后根遍历语法树计算综合属性；
- 先根遍历语法树计算继承属性；
- 两类属性有依赖的依赖图可能需要先根-后根多变遍历语法树才能得到全部属性值。

- 面对程序设计语言寻求更简单的属性求值方法。
- S-属性文法
  - 完全是综合属性的文法
- L-属性文法

S属性文法 包含于 L属性文法

  - 如果一个属性文法中的任意一个继承属性X.a都不会依赖于X的右兄弟的属性，则称其为L-属性文法。
  - 换句话说，继承属性只依赖于父、兄。

9.4 语法制导的属性求值

- 将语法树中文法符号翻译为中间代码。
- 得到树根的代码表明完成了语义分析任务。

- 这个翻译过程如果是伴随着语法分析同步进行，就被称为语法制导（引导、指引）的翻译。
- 这个翻译框架的基本任务其实就是属性求值，产生代码为其副作用。
- 相应地属性求值被规范为：
  - S-属性文法的LR制导的属性求值；
  - L-属性文法的LL制导的属性求值。

语法制导的基本思想

S'→S　　{S'.l=S.l; S'.m=S.m}

S→ SS　　{S[0].l=maS(S[1].l,S[2].l);　S[0].m=S[1].m+S[2].m}

S→(S)　　{S[0].l=S[1].l+1;　S[0].m=S[1].m+1;}

S→()　　{S.l=S.m=1}

符号栈	l栈	m栈	剩余串	动作
#	#	#	()(())#	移进
#(	#-	#-	)(())#	移进
#()	#--	#--	(())#	r4; S.l=1; S.m=1
#S	#1	#1	(())#	移进
#S(	#1-	#1-	())#	移进
#S((	#1--	#1--	))#	移进
#S(()	#1---	#1---	)#	r4; S.l=1; S.m=1
#S(S	#1-1	#1-1	)#	移进
#S(S)	#1-1-	#1-1-	#	r3; S[0].l=S[1].l+1; S[0].m=S[1].m+1
#SS	#12	#12	#	r2; S[0].l=max(S[1].l,S[2].l); S[0].m=S[1].m+S[2].m
#S	#2	#3	#	acc

S-属性文法的计算

- 上例中使用了带有综合属性域的分析栈。
- 设当前栈顶指针为sp，设属性文法中产生式A→XYZ配备的语义方程是A.a=f(X.a, Y.a, Z.a)，那么，在归约到XYZ时，栈上要弹出XYZ，每弹出一个符号的同时，将属性a栈也弹出，就依次得到了X.a，Y.a和Z.a，进而计算出A.a，并随A压栈的同时A.a也压到a栈上。
- 如果栈上弹出一个终结符，它的属性也可以从分析栈获得如果需要的话。如，

E → i　　{E.val=i.val}

分析栈	val属性栈	剩余串	动作
#		5#	移进
#5	5	#	归约
#E	5	#	

- E → i　　{E.val=getv(i)}

如果是这种属性文法则直接求（示意中的蓝5改为-）

S-属性文法的计算

- 推广到一般情形，每个属性名都有一个栈，这样LR分析栈就由一个状态栈、一个符号栈、多个属性栈联合联动而成。
- 对于CFG G=(𝕍, T, $\mathcal{P}$, S)的 (p=(A,γ))∈$\mathcal{P}$可配备属性方程

  p[0].a=f

  其中a∈$\mathcal{A}$, 函数f: 2^({γ_i|1≤i≤|γ|}×$\mathcal{A}$)→$\mathcal{V}$,

  $\mathcal{A}$为属性名集合，$\mathcal{V}$为属性值集合。
- 所有给p配备的属性方程的集合组成了关于p的语义行为（动作），这样带有语义行为的文法就是S-属性文法。
- S-属性文法的$\mathcal{A}$都是综合属性，$\mathcal{A}$的每个元素都有一个栈，叫做属性栈。
- 当用(p=(A,γ))∈$\mathcal{P}$归约时，名为a的属性栈，栈顶|γ|个符号中有属性γ_i.a, 1≤i≤|γ|, 的值或未定义，可用于计算A的属性值。注意γ_i.a, 1≤i≤|γ|, 一般写成p[i].a, 1≤i≤|γ|。

S-属性计算简单例子

S→X　　{S.l=X.l; S.c=X.c}

X→ XX　　{X[0].l=max(X[1].l,X[2].l);　X[0].c=X[1].c+X[2].c}

X→(X)　　{X[0].l=X[1].l+1;　　X[0].c=X[1].c+1;}

X→()　　{X.l=X.c=1}

有用的例子将在后边陆续给出。

例：level和count属性计算

符号栈	level栈	m栈	w#	动作
#	#	#	()(())#	移进
#(	#-	#-	)(())#	移进
#()	#--	#--	(())#	r4
#X	#1	#1	(())#	shift
#X(	#1-	#1-	())#	shift
#X((	#1--	#1--	))#	shift
#X(()	#1---	#1---	)#	r4
#X(X	#1-1	#1-1	)#	shift
#X(X)	#1-1-	#1-1-	#	r3
#XX	#12	#12	#	r2
#X	#2	#3	#	r1
#S	#2	#3	#	acc

- D→T$\tilde{V}$　　{$\tilde{V}$.in=T.type}
- T→int　　{T.type=i}
- T→float　　{T.type=f}
- $\tilde{V}$→$\tilde{V}$,d　　{$\tilde{V}$[1].in=$\tilde{V}$[0].in；bind(getn(d),$\tilde{V}$[0].in)}
- $\tilde{V}$→d　　{bind(getn(d),$\tilde{V}$.in)}

S-属性文法计算面临的问题

D→TṼ {Ṽ.in=T.type}

T→int {T.type=i}

T→float {T.type=f}

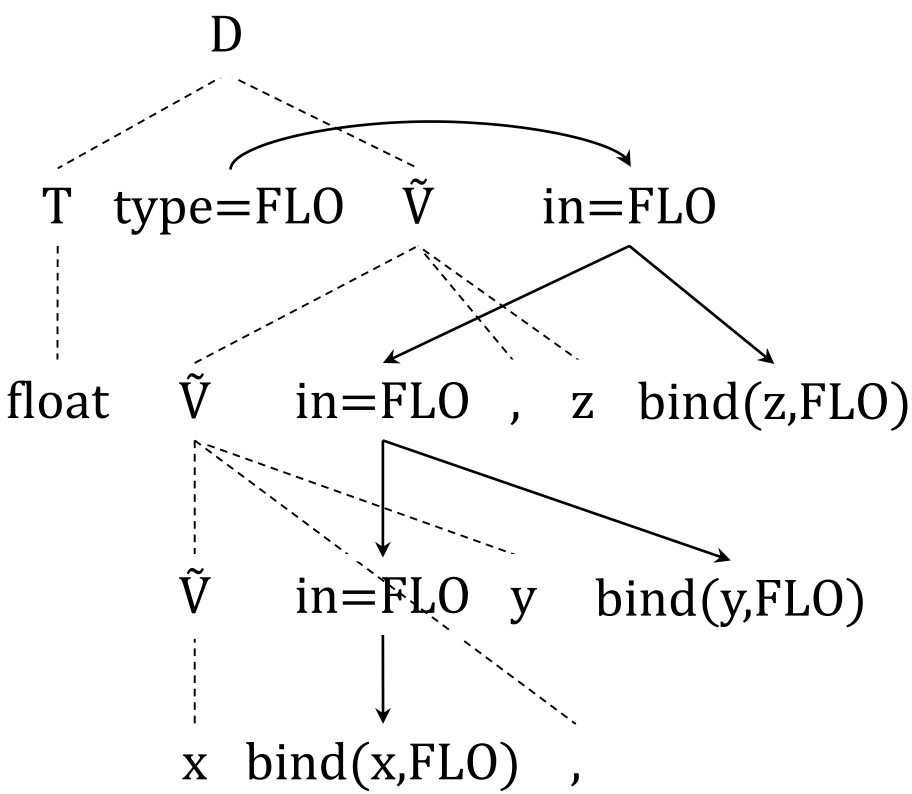
Ṽ→Ṽ,d {

Ṽ[1].in=Ṽ[0].in;

bind(getn(d),Ṽ[0].in)}

Ṽ→d {

bind(getn(d), Ṽ.in)}



问题：对继承属性不能正确处理

in属性栈不能正确工作

符号栈	in栈	type栈	剩余串	动作
#	#	#	float x,y,z#	s
#f	#-	#-	float x,y,z#	r:T→float
#T	#-	#f	x,y,z#	s
#Td	#--	#f-	,y,z#	r:Ṽ→d; in?
#TṼ	#-f	#f-	,y,z#	s
#TṼ,	#-f-	#f--	y,z#	s
#TṼ,d	#-f--#f--		,z#	r:Ṽ→Ṽ,d; in?
#TṼ	#-f	#f-	,z#	s
#TṼ,	#-f-	#f--	z#	s
#TṼ,d	#-f--#f---		#	r:Ṽ→Ṽ,d; in?
#TṼ	#-f	#f-	#	r:D→TṼ; in?
#D	#f	#f	#	acc

继承属性改为综合属性的情景

- 老板给员工发工资，把员工叫到办公室，老板取出一叠钞票给员工，员工领到了工资，也就就完成了工资发放。
- 老板给员工发工资，要了员工的账号，把工资打到账户里，就完成了工资发放。

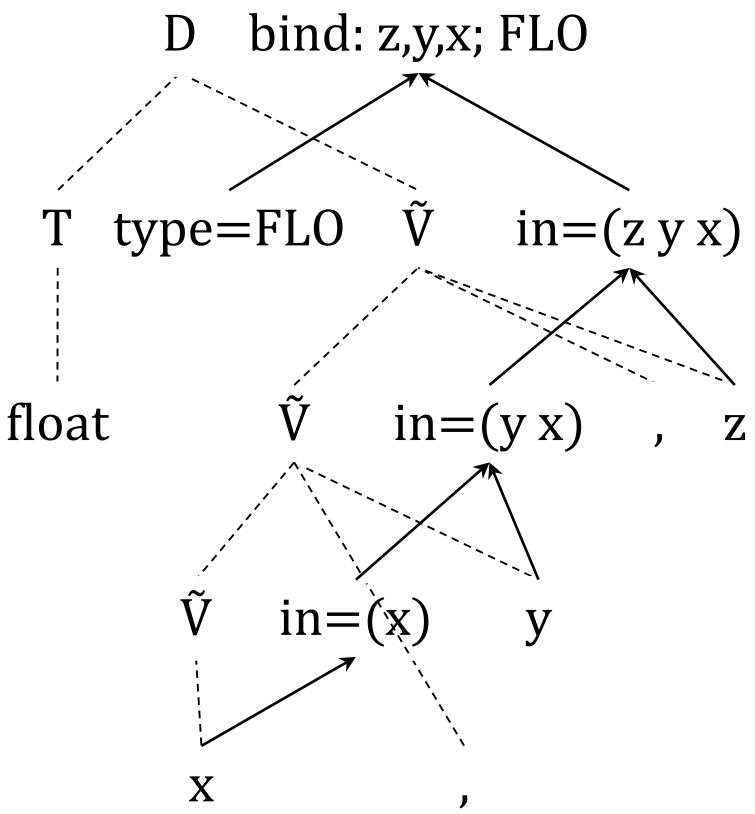
现在的in为综合属性名

D→TṼ {
  for(e∈Ṽ.in)
  bind(e,T.type)}

T→float {
  T.type=FLO}

Ṽ→Ṽ,d {
  Ṽ[0].in=
  cons(getn(d), Ṽ[1].in)}

Ṽ→d {
  Ṽ.in=list(getn(d))}



float x, y, z

将继承属性转变为综合属性

1:D→TṼ {for(e∈Ṽ.in)bind(e,T.type)}
2:T→float {T.type=FLO}
3:Ṽ→Ṽ,d
{v=getn(d);Ṽ[0].in=cons(v, Ṽ[1].in)}
4:Ṽ→d {v=getn(d);Ṽ.in=list(v)}

符号栈	in栈	type栈	剩余串	语法及语义动作
#	#	#	float x,y,z#	shift
#f	#-	#-	x,y,z#	r2;
#T	#-	#F	x,y,z#	shift
#Tx	#-x	#F-	,y,z#	r4; Ṽ.in=(x)♠; ~~d.in=x~~
#TṼ	#-♠	#F-	,y,z#	shift
#TṼ,	#-♠,	#F--	y,z#	shift
#TṼ,d	#-♠,y	#F--	,z#	r3; Ṽ[0].in=(y x)♣; ~~d.in=y~~
#TṼ	#-♣	#F-	,z#	shift
#TṼ,	#-♣-	#F-	z#	shift
#TṼ,d	#-♣-z	#F--	#	r3; Ṽ[0].in=(z y x)◇; ~~d.in=z~~
#TṼ	#-◇	#F-	#	r1; bind:x,FLO; y,FLO; z,FLO;
#D	#-	#F	#	acc

几个函数说明

- set(x,y)参数求值，然后后者赋给前者。
- getn(d)返回标识符记号中的名字。
- newvar()创建并返回一个变量名。
- list(x)参数求值，然后返回以该值为元素的表。
- cons(x,list)参数求值，然后list中新增第一个元素x后被返回。
- for(e∈list){...}按照从左到右的次序逐个元素执行后面的复合语句。

使用符号表

- bind(name, type)在符号表中登记一项，若已存在则报错。
- lookup(name)查找name返回该登记项或UNBOUND

补充：来自Lisp的表概念

- 对象只有两种：atom和cons
- 点对(a . b)是一个cons，其中a和b为原子或cons
- nil表示空表，也是个原子
- (a . nil)=(a)
- (a . (b . (c . nil)))=(a b c)
- (a . (b . (c . d)))=(a b c . d)
- 定义表处理函数用于设计属性文法：
  - cons(e, l)返回(e' . l')其中e'为e的值，l'为l的值
    list(e)返回(e')
    endcons(l, e)返回表l与表list(e)连接在一起
  - car(c)返回a，如果c的值为(a . b)
  - cdr(c)返回b，如果c的值为(a . b)
  - nth(n,l)，返回l的第n个元素，表元素从第0个算起。

9.5 程序声明部分的语义分析

- 程序声明或声明语句的作用：声明名字的类型。
- 名字类型反映了名字的语义信息，这些信息应用于执行语句的中间代码生成。
- 包括：9.5.1 简单变量声明；9.5.2 数组声明；9.5.3 函数声明

9.5.1简单变量声明语句

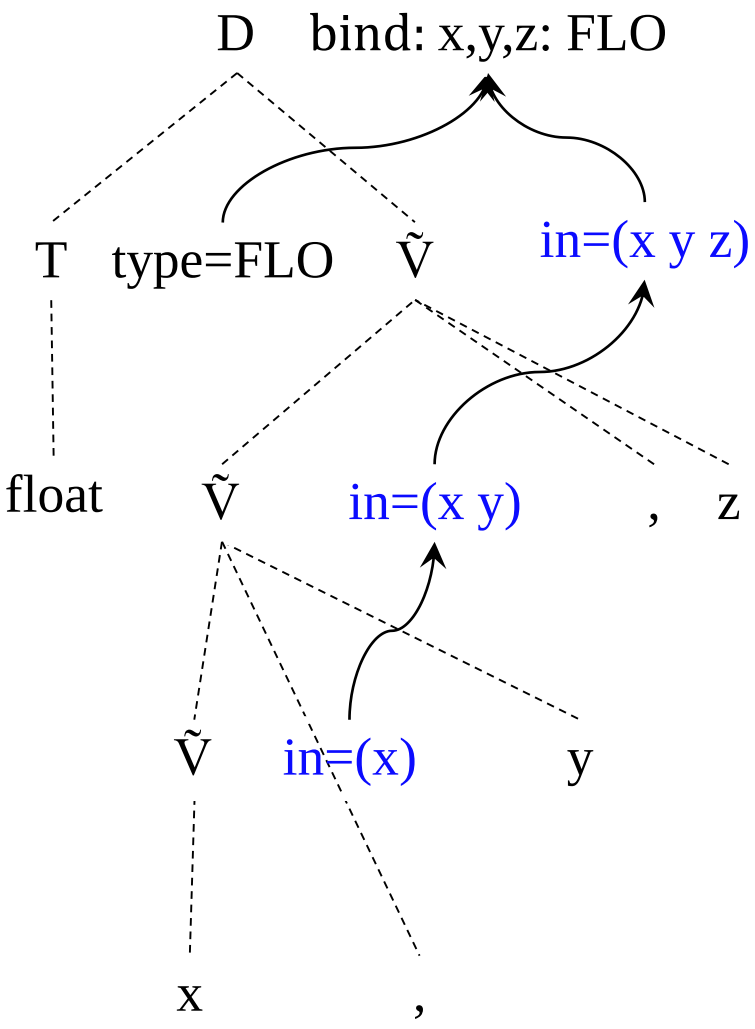
D→T$\tilde{V}$ {
for(e∈$\tilde{V}$.in)bind(e, T.type);
D.place=$\tilde{V}$.in}

T→int {T.type=INT}

T→float {T.type=FLO}

$\tilde{V}$→$\tilde{V}$,d {v=getn(d);
$\tilde{V}$[0].in=endcons($\tilde{V}$[1].in), v)}

$\tilde{V}$→d {v=getn(d);
$\tilde{V}$.in=list(v))}



float x, y, z

9.5.2 数组声明语句

- 例：int a[10,5,20]
- 数组名a，元素类型INT，三维，维长依次为10，5，20
- 一般地，数组有类型（数组类型），其元素也有类型（数组元素类型）
  - 任意维数组的类型都用ARRAY表示
  - 元素类型都是基本类型
  - 一般地写成array(Ǐ,T)，其中Ǐ是各维维长，T是元素类型。
- 数组元素存储区域。数组各元素按照预定次序（如行主序）排列，排列方式决定如何访问数组元素（偏移量）。
- 数组每一维都有上界和下界，之差加1为维长。下界默认为0。那么数组的尺寸就是整数列表Ǐ的各元素的乘积再乘上元素类型尺寸。
- 编译时从数组声明语句获取的有用信息都保存在符号表中。

数组声明与下标变量的语义

- C语言数组声明　　float a[SIZE] ; int i,j;
- 数组元素引用　　a[i+1]=a[j*2]+3;

- PASCAL数组声明　var a: array[0..10] of int;　　i,j:int;
- 数组元素引用　　a[i]:=a[j]+1;

- FORTRAN数组声明　　real x(-13,13)
- 　　integer i,j
- 数组元素引用　　x[i]=x[j]+1

- 数组元素的存放区域如何安排？
- 如何确定数组元素的偏移量？

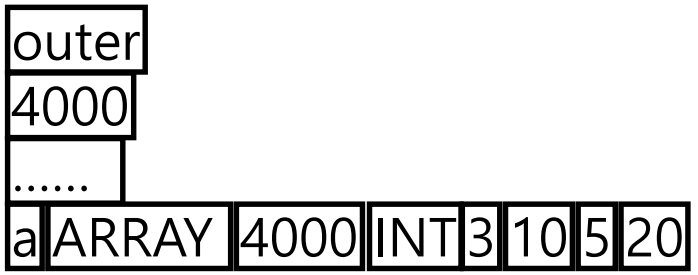
数组声明语句

- D → T d[Ǐ]

- int a[10,5,20]
- D.place是表(a)，表元素是声明语句声明的数组名。
- Ǐ.val=(10 5 20)，其中表元素为整数。
- T.type是类型INT。类型大小按4字节计。

数组声明举例

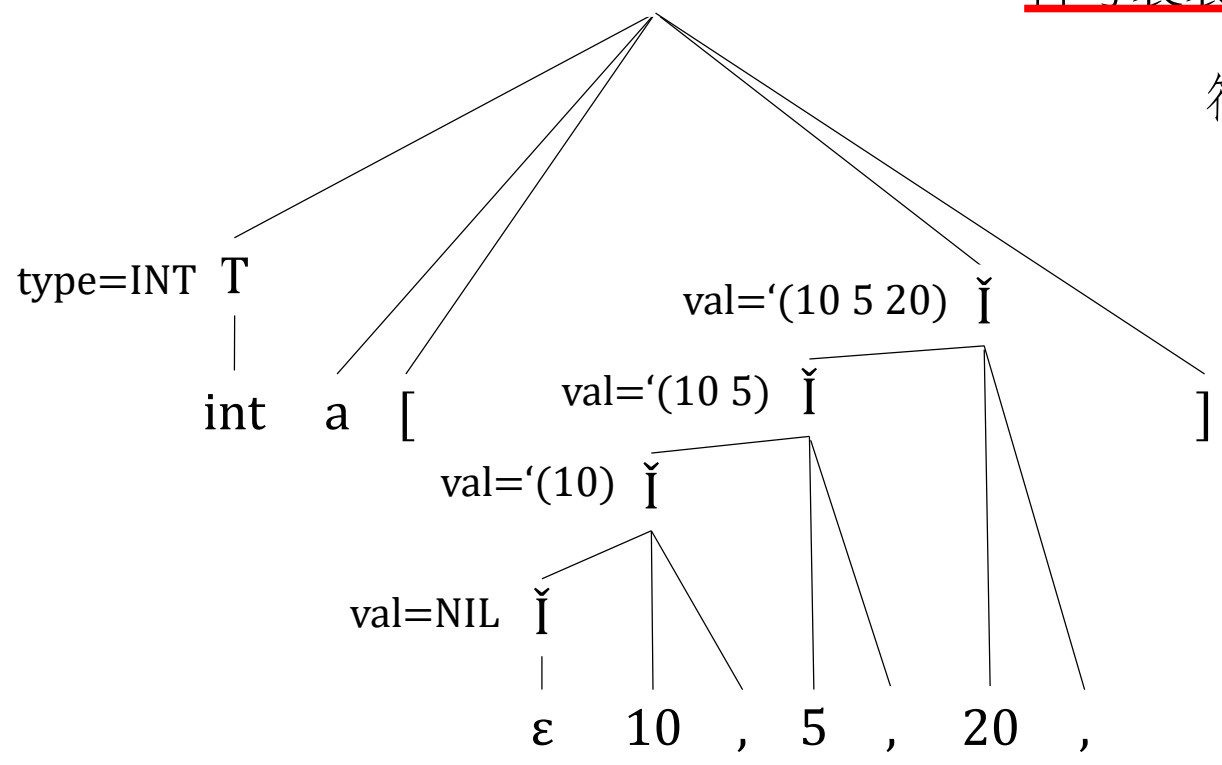
- int a[10,5,20]
- 数组登记项内容为，
  - name:a;
  - type:ARRAY;
  - base:有; // tab->width
  - etype:INT;
  - dims:3;
  - dim[0]:10; dim_length计算：(10-1)-(0)+1
  - dim[1]:5;
  - dim[2]:20
- tab->width增加4000



outer	外层符号表指针
width	过程的存储区大小
argc	过程参数个数
arglist	过程参数表
rtype	返回结果类型
code	过程代码
*entry	登记项

继承属性实现为综合属性

int a[10, 5, 20,]



size=4000

符号表表头width+=4000

符号表添加登记项

name:'a

type:ARRAY

base: width

etype:INT

dims:3

dim[0]:10

dim[1]:5

dim[2]:20

数组声明语句翻译

- D → T d[Ǐ] {
  x=getn(d);
  bind(x,ARRAY);
  k=length(Ǐ.val);
  lookup(x,dims:,k);
  size=1;
  for(i=0;i<k;i++){
    c=nth(i, Ǐ.val);
    lookup(x, dim:i, c);
    size*=c}
  size*=sizeof(T.type);
  lookup(x, etype:, T.type);
  update[width, ?size, add];
  lookup(x, base:, lookup(width:));
  D.place=list(x)}

- int a[10,5,20]
  x='a
  登记'a及各域到tab中
  dims=3
  维长记入该登记项
  为了计算'a尺寸
  循环每一维
  第i个维长
  记入当前登记项中
  计算'a尺寸
  考虑到元素类型尺寸
  元素类型记入登记项
  空间首址记入登记项
  更新符号表的offset

符号表数组登记项访问示例

- bind(name, type)如果已经存在则报错，否则登记一项，具体长度根据type可决定。
- 数组类型是ARRAY
- lookup(name)在不在，在了返回name，不在返回UNBOUND
- lookup(name, type:)返回类型或UNBOUND表示name不存在。
- lookup(name, type:, INT)不能设置类型，建立时就已知。
- lookup(a,etype:,INT)设置数组元素类型。
- lookup(a,dims:,n)设置数组维数。
- lookup(a,dim[i]:,k_i), i=0,...,n-1，设置数组每一维维长。或写为lookup(a,dim: i, k_i)。
- lookup(a,base:,addr)设置数组存储区首地址。

9.5.3 函数声明

- n-参数函数的类型表达式为$D_1 \times D_2 \times \ldots D_n \rightarrow R$，其中$D_k$为函数第k个参数的类型，R为函数返回结果类型。函数类型表达式提供的语义信息有：形式参数个数，形式参数表，每个参数的类型及其类型特有域的值，返回结果类型。
- 函数参数的类型可以是任意类型，参数可以多个。
  - 数组作为参数，可声明为数组或数组原型；
  - 函数作为参数，声明为函数签名。
- D → T d (Ă) {Ď Š} 其中Ď中可以声明函数（类似PASCAL）或者不允许再声明函数（类似C语言）。
- 函数声明的有用信息都在语义分析时保存到符号表中。
- Ă中只能是函数签名、数组原型、简单变量名、标号名。

形参表

- Ă → ε {
  Ă.place=NIL; push(symtab, newtab());}

- A → T d　　{
  x=getn(d);
  bind(x, T.type);
  update[width, ?sizeof(T.type), add];
  lookup(x, offset:, lookup(width:));
  update[arglist, ?x, endcons];
  update[argc, 1, add];
  A.place=list(x);}

- Ă → Ă A ; {
  Ă[0].place=append(A.place, Ă[1].place);}

变量声明语句翻译

- D→T d {
  x=getn(d);
  bind(x, T.type);
  update[width, ?sizeof(T.type), add];
  lookup(x, offset:, lookup(width:));
  D.place=list(x);}

声明列表

- Ď → ε {Ď.place=();}
- Ď → ĎD ; {Ď[0].place=append(Ď[1].place, D.place);}
- D→T d (Ă) {Ď Š} {tab=pop(symtab); //栈顶是d<x>的
  tab->outer=top(symtab); //栈顶为x的外层即D的
  x=getn(d);
  bind(x, FUNC);
  lookup(x, mytab:, tab);
  update[width, ?sizeof(FUNC), add];
  lookup(x, offset:, lookup(width:));
  D.place=list(x);
  tab->code=Š.code;} //Š.code在第十章生成

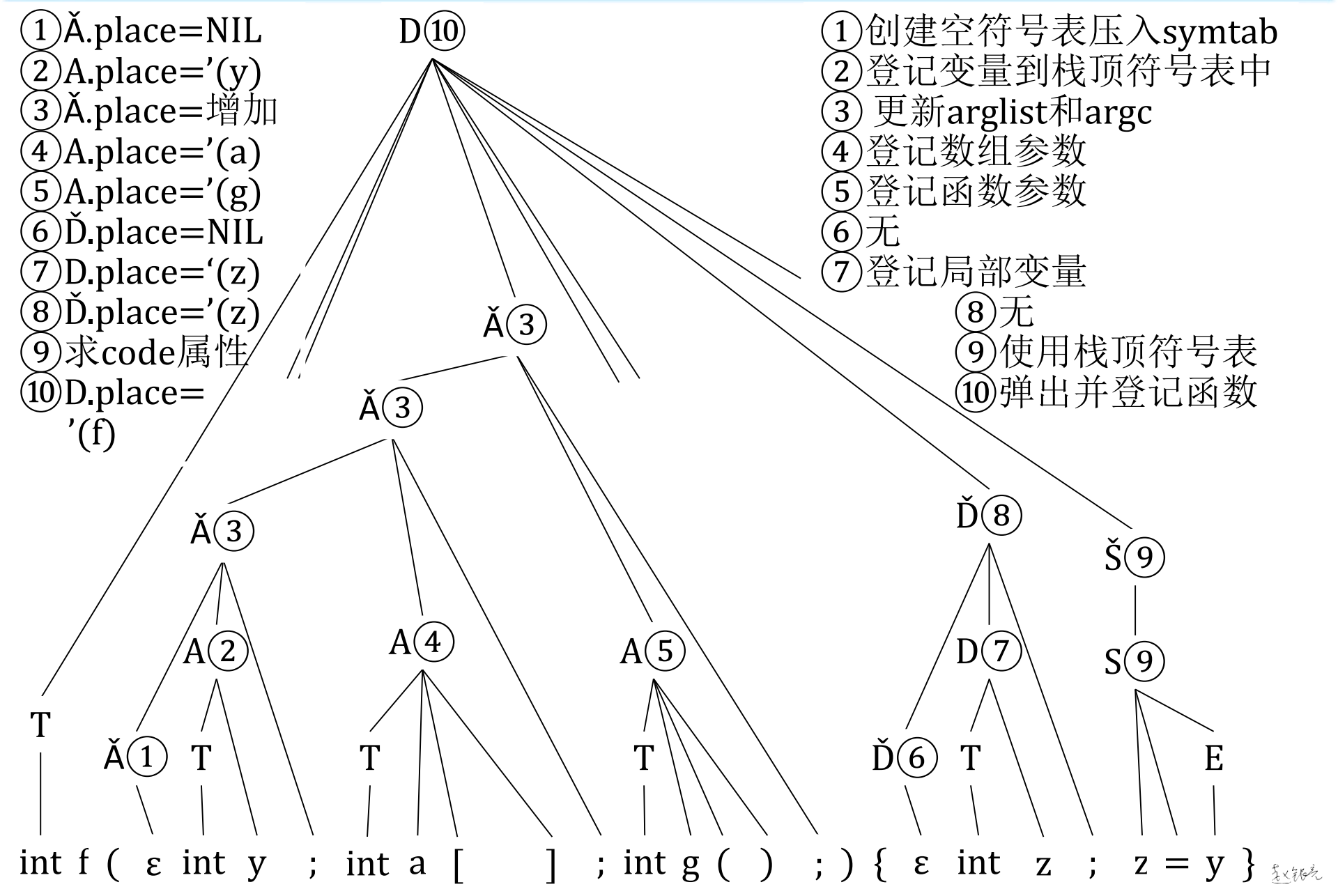
数组形参

- A →T d[]　　{
  x=getn(d); bind(x, ARRPTT);
  update[width, sizeof(ARRPTT), add];　　//=4
  lookup(x, base:, lookup(width:));
  lookup(x, etype:, T.type);
  update[arglist, ?x, endcons];
  update[argc, 1, add];
  A.place=list(x);}

函数形参

- A →T d()　　{
  x=getn(d); bind(x, FUNPTT);
  update[width, sizeof(FUNPTT), add];　　//ep,ip; =8
  lookup(x, offset:, lookup(width:));
  lookup(x, rtype:, T.type);
  update[arglist, ?x, endcons];
  update[argc, 1, add];
  A.place=list(x);}

翻译语句的时候当前符号表是栈顶符号表

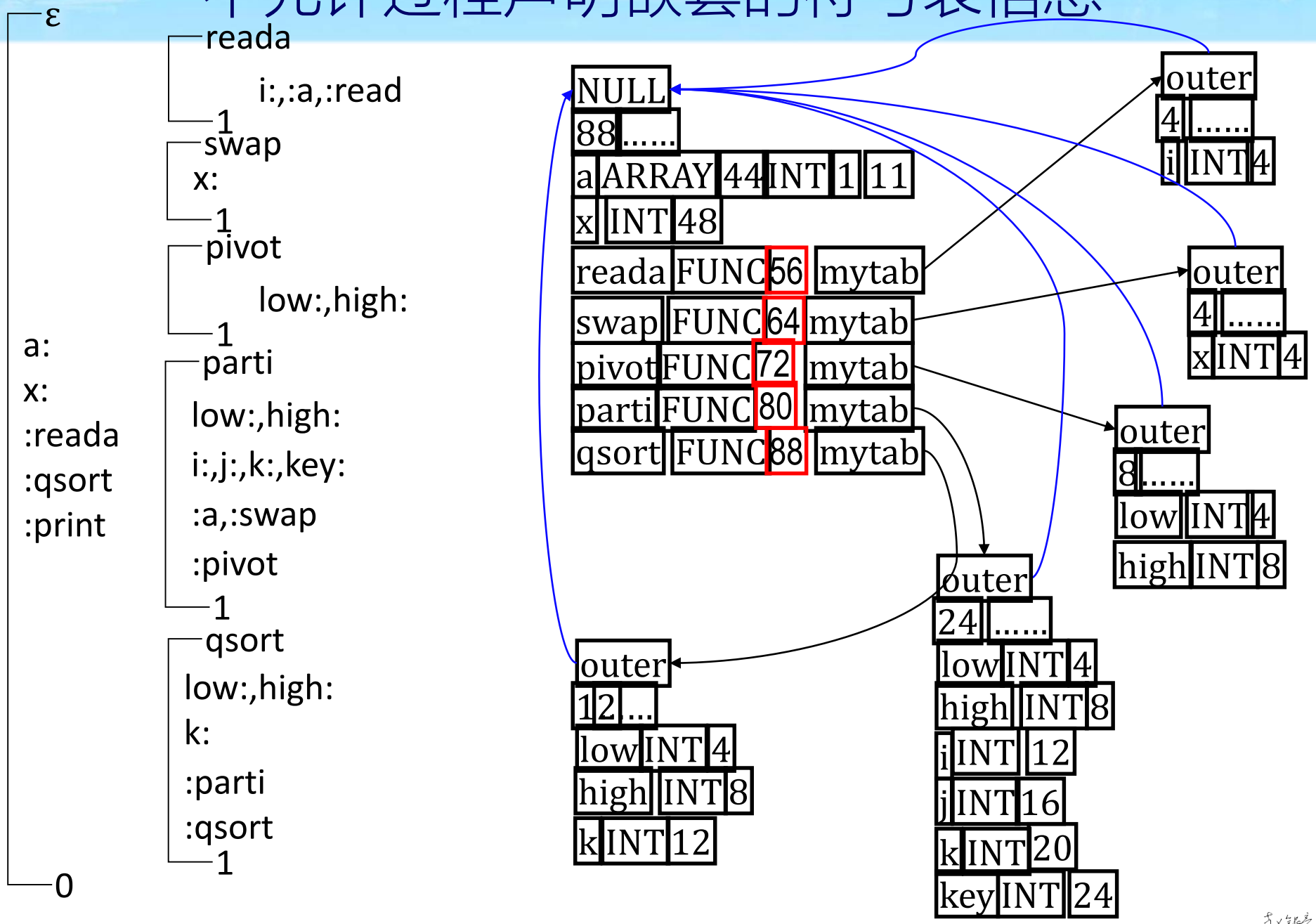


不允许过程嵌套定义

```
int a[11];
int x;
void reada(){
  int i;
  i=0;
  while(i<11){
    read(a[i]);
    i++}
}//readArray
void swap(int i,j){
    int x
    x=a[i];
    a[i]=a[j];
    a[j]=x
}//exchange
int pivot(int low,high){
   return low
  }
int parti(int low, int high){
  int i,j,k,key;
  k=pivot(low,high);
  if (k!=high)swap(k,high);
  key=a[high];
  i = low - 1;
  for (j=low;j<=high-1;j++){
    if (a[j] < key) {
      i++;
      swap(i,j)  }
  }
  swap(i+1, high);
  return i+1
} //partition
```

```
void qsort(int low, high)
{
  int k;
  if (low < high)
  {
     int k = parti(low, high);
     qsort(low, k - 1);
     qsort(k + 1, high);
   }
}//end of qsort
reada();
qsort(0,10);
print().
```

不允许过程声明嵌套的符号表信息



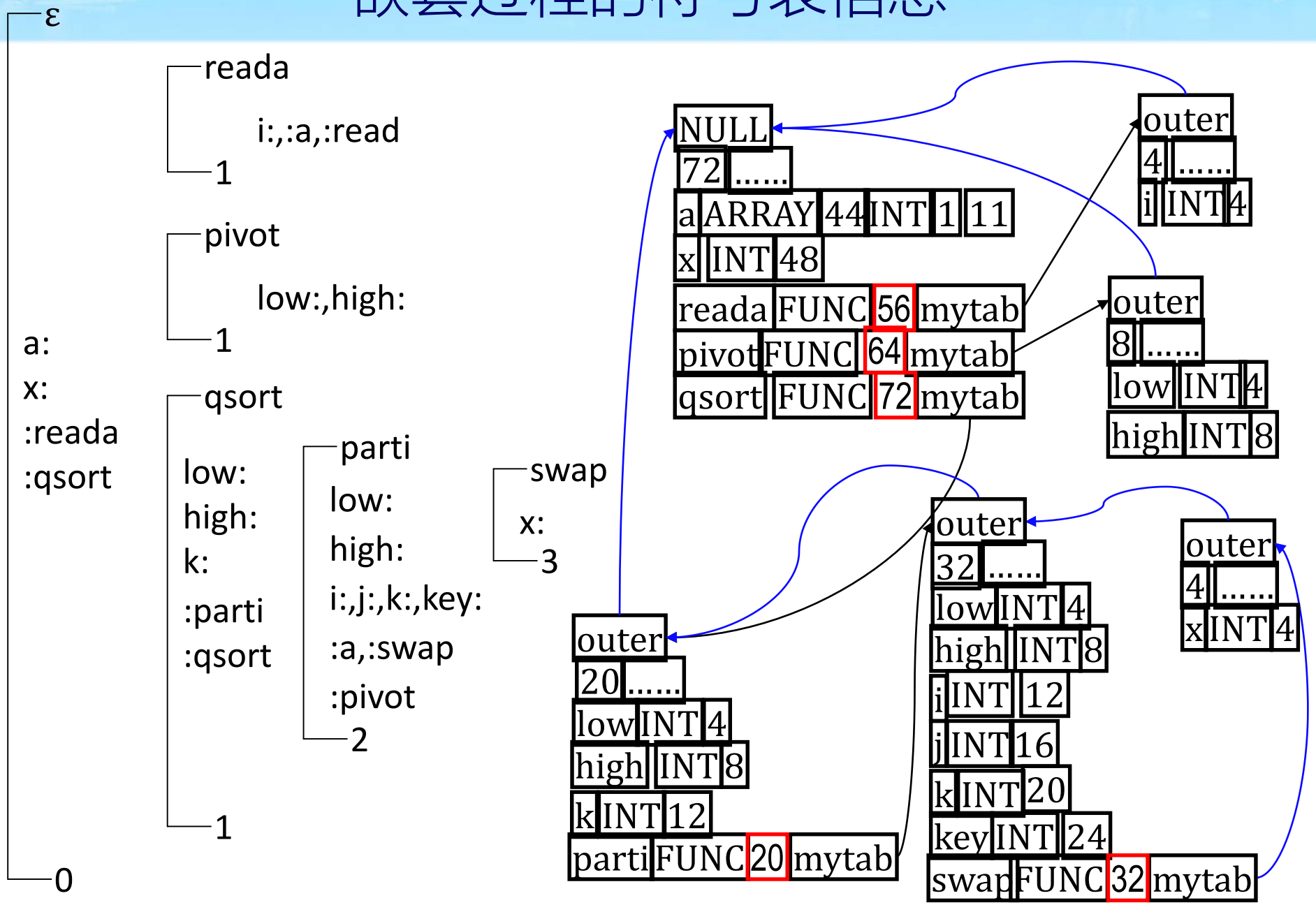
允许过程嵌套声明

```
int a[11];
int x;
void reada(){
 int i;
 i=0;
 while(i<11){
  read(a[i]);
  i++}
}
int pivot(int low,high){
 return low
}
void qsort(int low,
      int high){
 int k;
 int parti(int low,high){
```

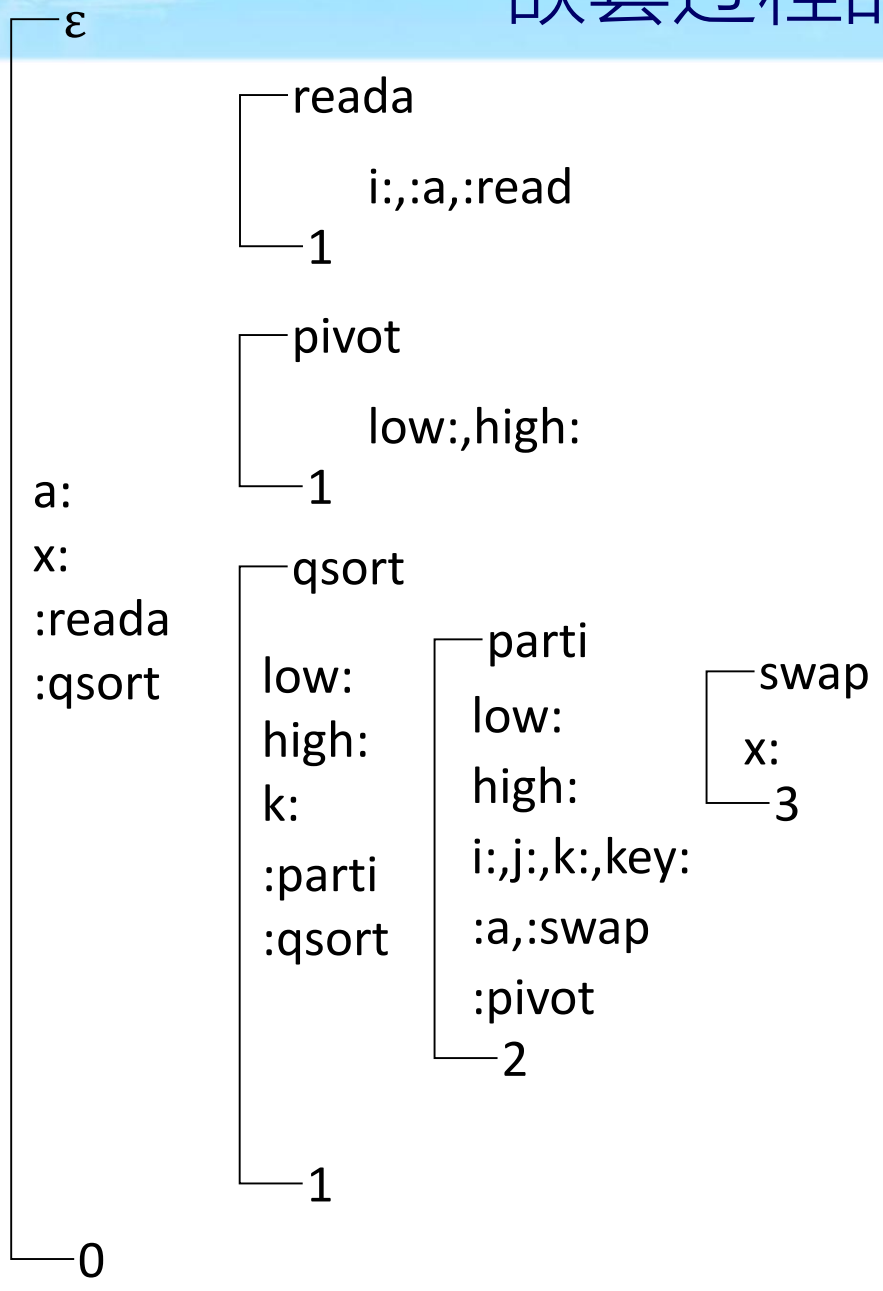
```
void swap(int i,j){
 int x;
 x=a[i];
 a[i]=a[j];
 a[j]=x}//end of swap
int i,j,k,key;
k=pivot(low,high);
if (k!=high)swap(k,high);
key=a[high];
i = low - 1;
for (j=low;j<=high-1;j++){
   if (a[j] < key) {
      i++;
      swap(i,j)  }
}
swap(i+1, high);
```

```
  return i+1
 } //end of parti
 if (low < high) {
   k=parti(low, high);
   qsort(low, k - 1);
   qsort(k + 1, high);
 }
return 1
} //end of qsort
reada();
qsort(0,10);
print a.
```

嵌套过程的符号表信息


ε
reada
i:,:a,:read
1
pivot
low:,high:
1
a:
x:
:reada
:qsort
qsort
low:
high:
k:
:parti
:qsort
parti
low:
high:
i:,j:,k:,key:
:a,:swap
:pivot
2
swap
x:
3
1
0
NULL
72 .....
a ARRAY 44 INT 1 11
x INT 48
reada FUNC 56 mytab
pivot FUNC 64 mytab
qsort FUNC 72 mytab
outer
4 .....
i INT 4
outer
8 .....
low INT 4
high INT 8
outer
20 .....
low INT 4
high INT 8
k INT 12
parti FUNC 20 mytab
outer
32 .....
low INT 4
high INT 8
i INT 12
j INT 16
k INT 20
key INT 24
swap FUNC 32 mytab
outer
4 .....
x INT 4

嵌套过程的符号表信息


ε
reada
i:,:a,:read
1
pivot
low:,high:
1
a:
x:
:reada
:qsort
qsort
low:
high:
k:
:parti
:qsort
parti
low:
high:
i:,j:,k:,key:
:a,:swap
:pivot
2
swap
x:
3
1
0

```
int a[100];
int x;
void reada(){ // 读入a各元素
    int i;
    …
}
int pivot(int low;int high;){ //选取主元返回之
    …
}
void qsort(int low; int high;){ //在a的指定范围内快排
    int k;
    int parti(int low;int high;){//在指定范围内划分并返回边界
        void swap(int i; int j;){ // 交换a[i]和a[j]
            int x;
            …
        }
        int i,j,k,key;
        …
    }
    …
}
…/对输入快排并输出结果
```

```
int a[100];
int x;
void reada(){                          // 读入a各元素
    int i;
    …
}
void qsort(int low, int high){         // 在a的指定范围内快排
    int k;
    int parti(int low, int high){      // 划分并返回边界
        void swap(int i,j){            // 交换a[i]和a[j]
            int x;
            …
        }
        int i,j,k,key;
        …
    }
    …
}
…                                      /对输入快排并输出结果
```

Level 1

Level 2

Level 3

快排程序的符号表

```
int a[100];
int x;
void reada(){ // 读入a各元素
   int i;
   …
}
int pivot(int low;int high;){ //选取主元返回之
   …
}
void qsort(int low; int high;){ //在a的指定范围内快排
   int k;
   int parti(int low;int high;){//在指定范围内划分并返回边界
      void swap(int i; int j;){ // 交换a[i]和a[j]
         int x;
         …
      }
      int i,j,k,key;
      …
   }
   …
}
…//对输入快排并输出结果
```

@table: (
outer: NULL
width: 428 argc: 0 arglist: NIL rtype: VOID level: 0 code: [...]
entry: (name: a type: ARRAY base: 400 dims: 1 dim[0]: 100 etype: INT)
entry: (name: x type: INT offset: 404)
entry: (name: reada type: FUNC offset: 412 mytab: reada@table)
entry: (name: pivot type: FUNC offset: 420 mytab: pivot@table)
entry: (name: qsort type: FUNC offset: 428 mytab: qsort@table))

全局@table，已经没有人比你更靠外了

全局运行域的level默认是0

100个int => 400

400+4(int x)

函数与其他之间的entry都是8

函数与其他之间的entry都是8

函数与其他之间的entry都是8

reada@table: (
outer: @table
width: 4 argc: 0 arglist: NIL rtype: VOID level: 1 code: [...]
entry: (name: i type: INT offset: 4))

它的外层显然是全局@table

第一层定义的函数（看右上角）

Entry表是每个形参/局部变量都要写的函数的@table下，不同entry之间的间隔按照类型来

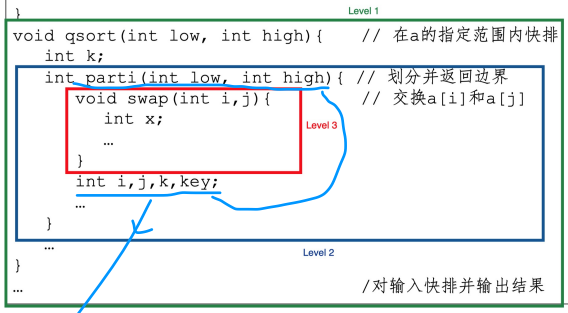
pivot@table: (
outer: @table
width: 8 argc: 2 arglist: (low high) rtype: INT level: 1 code: [...]
entry: (name: low type: INT offset: 4)
entry: (name: high type: INT offset: 8))

第一层定义的函数（看右上角）

Int low => 4

Int high => 4 + 4

快排程序的符号表（续）



qsort@table: (
outer: @table
width: 20 argc: 2 arglist: (low high) rtype: VOID level: 1 code: [...]
entry: (name: low type: INT offset: 4) int low
entry: (name: high type: INT offset: 8) 4 + 4(int high)
entry: (name: k type: INT offset: 12) 8 + 4(int k) 12 + 8（函数与其他的entry之间间隔都是8）
entry: (name: parti type: FUNC offset: 20 mytab: parti@table))

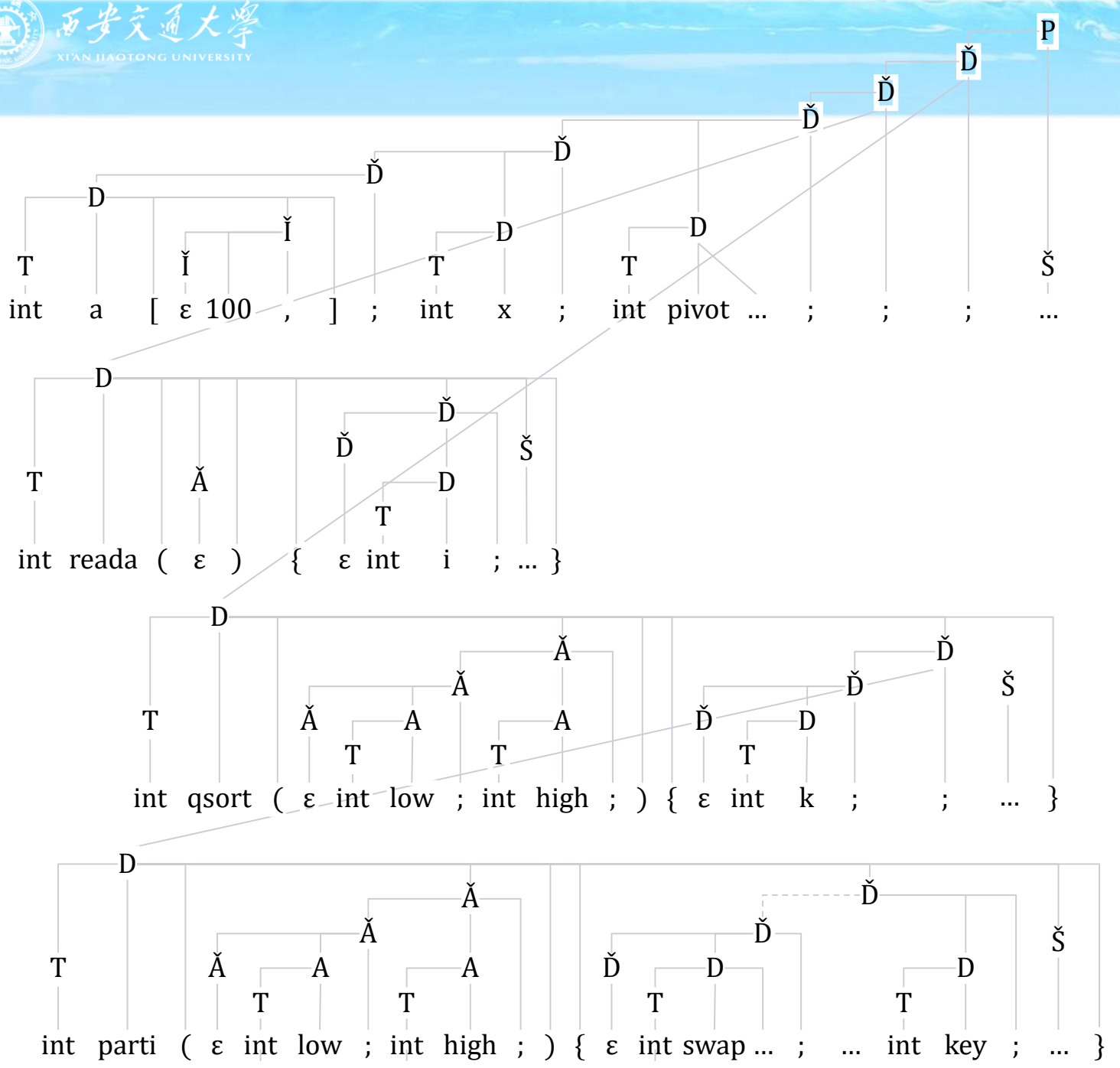
parti@table: (
outer: qsort@table
width: 32 argc: 2 arglist: (low high) rtype: INT level: 2 code [...]
entry: (name: low type: INT offset: 4)
entry: (name: high type: INT offset: 8)
entry: (name: i type: INT offset: 12)
entry: (name: j type: INT offset: 16)
entry: (name: k type: INT offset: 20)
entry: (name: key type: INT offset: 24) 24 + 8（函数与其他的entry之间间隔都是8）
entry: (name: swap type: FUNC offset: 32 mytab: parti@table))

swap@table: (
outer: parti@table
width: 4 argc: 2 arglist: (i j) rtype: VOID level: 3 code: [...]
entry: (name: x type: INT offset: 4))

为什么low和high在parti中还要再写一遍entry，而I和j在swap中不需要再写一遍?

Low and high 在 parti的外层（即qsort）不是被定义的，是形参，因此要写

I and j 在swap的外层（即parti）已被定义，不是形参，是局部变量，因此不用写


P
Ď
Ď
Ď
Ď
Ď
Ď
D
Ǐ
Ǐ
D
D
T
T
T
T
Š
int a [ε 100 ,] ; int x ; int pivot ... ; ; ; ...
D
Ď
Ď
Š
Ă
D
T
T
int reada (ε) { ε int i ; ... }
D
Ă
Ď
Ă
Ď
Š
Ă
A
A
Ď
D
T
T
T
int qsort (ε int low ; int high ;) { ε int k ; ; ... }
D
Ă
Ď
Ă
Ď
Š
Ă
A
A
Ď
D
D
T
T
T
T
int parti (ε int low ; int high ;) { ε int swap ... ; ... int key ; ... }

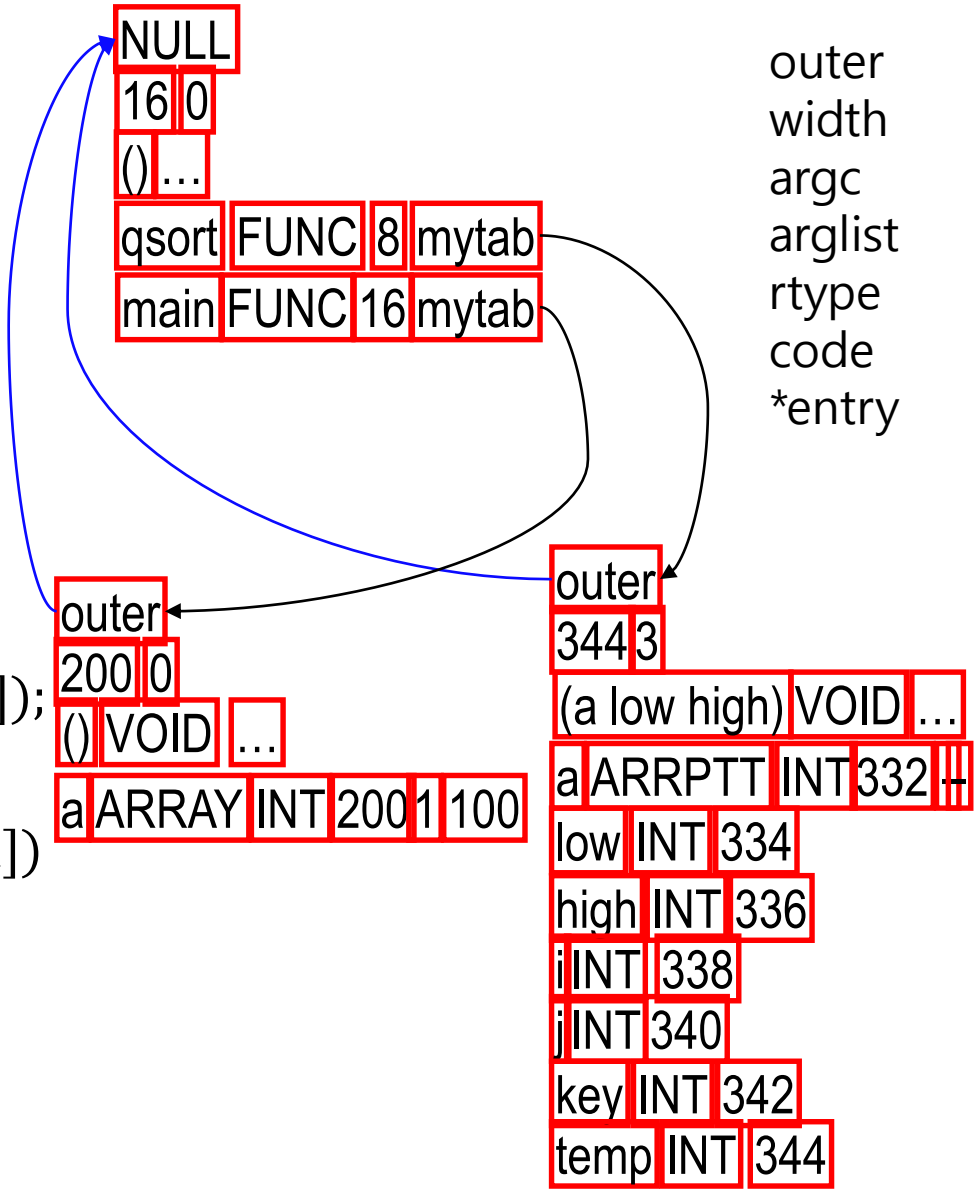
无嵌套声明的函数翻译举例

```
void qsort(int a[166], int low, int high){
int i,j,key,temp;
if (high <= low) return;
i = low;
j = high;
key = a[low];
while (true)   {      //从左向右找比key大的值
        while (a[i] <= key)      {
                i++;
                if (i == high) break;      } //从右向左找比key小的值
        while (a[j] >= key)      {
                 j--;
                if (j == low){          break;        }      }
        if (i >= j) break;      /*交换i,j对应的值*/
        temp = a[i];      a[i] = a[j];      a[j] = temp;   }
```

无嵌套声明的函数翻译举例

```
        a[low] = a[j];
        a[j] = key;
        qsort(a, low, j - 1);
        qsort(a, j + 1, high);
}

void main(){
        int a[100];
        for(i=0;i<100;i++)read(a[i]);
        qsort(a, 0, 99);
        for(i=0;i<100;i++)print(a[i])
}
main()
```



补充SLR(1)附加冲突消解

- 如果有效项目集中同时包含{E→d(Ě)•, S→d(Ě)•}且FOLLOW(E)∩FOLLOW(S)≠φ，那么，
  - 若lookup(getn(d), rtype:)==VOID则归约为S；
  - 否则，归约为E。

- 为此增加一个返回值类型VOID。

常用属性类

- D.place是一个表，元素是D声明的名字。
- D.tab是一个不完全符号表，有width和D声明的名字登记项
- Ď.place是一个表，表元素同D.place，顺序与语句列表一致。
- Ď.tab是将列表元素的符号表合并成为的一个符号表。
- E.code用于存放由表达式翻译产生的中间代码。
- E.place中存放一个变量，这个变量存放代码的执行结果。
- Ě.code是一个表，表元素同E.code，顺序与列表一致。
- Ě.place是一个表，表元素同E.place，顺序与列表一致。
- Ǐ.val是一个表，表元素为整数，顺序与列表一致。
- S.code语句S的代码。
- Š.code是一个表，表元素同S.code，顺序与语句列表一致。
- B.tc是个表，表元素为标号，表示B.code执行结果为真时后继
- B.fc是个表，表元素为标号，表示B.code执行结果为假时后继
- B.code用于存放由布尔表达式翻译产生的中间代码。

常用功能函数

- newtab()
  - 创建并返回一个符号表tab，其中，tab->width=0; tab->outer=NULL; tab->entry=NULL; tab->arglist=NIL; tab->argc=0; tab->offset=0
- bind1(tab, x, type)
  - if (lookup1(tab,x,type:)!=UNBOUND) error(); tab->entry=newentry(x,type,tab->entry)

采用全局名的翻译样例

```
int z;
int a[10,20];
int bar(int x){
        return x++};
float foo(int x, int b[], int boo()){
        if (x >0) z=sqrt(b[0,0])
        else return boo(z)};
print foo(0, a[],bar())
```

```
@table:
outer:NULL;
width:820;
argc:0;
arglist:NIL;
rtype:INT;
code:@code;
entry:(name:z; type:INT; offset:4);
entry:(name:a; type:ARRAY; base:804; etype:INT;
        dims:2; dim[0]:10; dim[1]:20);
entry:(name:bar; type:FUNC; offset:812;
        mytab:bar@table);
entry:(name:foo; type:FUNC; offset:820;
        mytab:foo@table)
```

800

800+4

任何@table之下：
函数(FUNC)的entry之间默认是8
（函数与函数之间
函数与变量之间
函数与数组之间）

数组如果没有给定参数，即数组指针，entry之间默认是4

```
foo@table:
outer:@table; width:16; argc:3;
arglist:(x b boo);
rtype:FLO; code:foo@code;
entry:(name:x; type:INT; offset:4);
entry:(name:b; type:ARRPTT;
        offset:8; etype:INT);
entry:(name:boo; type:FUNPTT;
        offset:16; rtype:INT)
```

三个参数

参数列表

“对外”

返回值类型是float

0 + 4(int)

4 + 4(数组指针是4)

8 + 8(函数指针是8)

函数内的@table

Entry之间的间隔使用类型计算

```
bar@table:
outer:@table; width:4; argc:1; arglist:(x);
rtype:INT; code:bar@code;
entry(name:x; type:INT; offset:4)
```

因为entry遍历完以后的累积offset是4，即width

“对外”

“对内，描述参数”

类型检查

- 检查类型声明是否满足类型的语义性质。
- 设计时允许文法修剪与附加语义约束取得折中，
  - D→Td(Ď){ĎŠ}修剪为D→Td(Ă){ĎŠ}；或者给D→Td(Ď){ĎŠ}附加语义约束，将Ď[1]约束为只能是Ă情况；
  - D→Td[Ě]修剪为D→Td[Ĭ]；或者给D→Td[Ě]附加语义约束，将Ě约束为只能是整数列表否则报错；
  - 各种边界及类型范围的约束等等。
- 总之在简化文法与简化语义性质之间平衡。
- 当然有些语义性质是无法通过修剪文法来平衡掉的，因为这些语义性质是固有的。
- 此外还有类型推理问题如类型转换等（略）

小结

- 符号表表示及相关操作
- 属性文法概念及设计
  - 属性名、属性、属性值
  - 综合属性、继承属性、S属性文法、L属性文法
- 基于属性栈的SLR(1)制导语义分析框架
  - 继承属性转换为综合属性
  - 扩展SLR(1)分析框架计算属性值
  - 语法树上计算属性值、带注释语法树
- 声明的语义分析
  - 简单变量声明
  - 数组声明
  - 函数声明、函数符号表、函数声明的嵌套与并列
- 层次符号表的属性表表示（注意属性表与属性文法无关）

本章作业

- 作业：编译p164，1、9（要求产生三地址代码）
- 补充见下页

补充

- **习题9.1**利用本章为主文法配套的属性文法，翻译下列声明为符号表表示（提示：共4个符号表），假定对于Š的代码生成都没有产生临时变量，Š的代码用省略号表示。假定FLO类型宽度为8。然后写出h()函数声明语句的带注释语法树或带注释的规范归约，提示：画出语法树并将树中每个变元的属性逐一列出，考虑到树上标注不方便，可对相同变元的多次出现用上标区分如图8-1，然后在树外一一列出即可。

```
int x; int a[5];
float b[3, 6];
int g(int r(); int y; float b[];){
        int c[10];
        Š};
int h(int f(); int y;){
        int g(int c[];){Š};
        Š};
Š
```