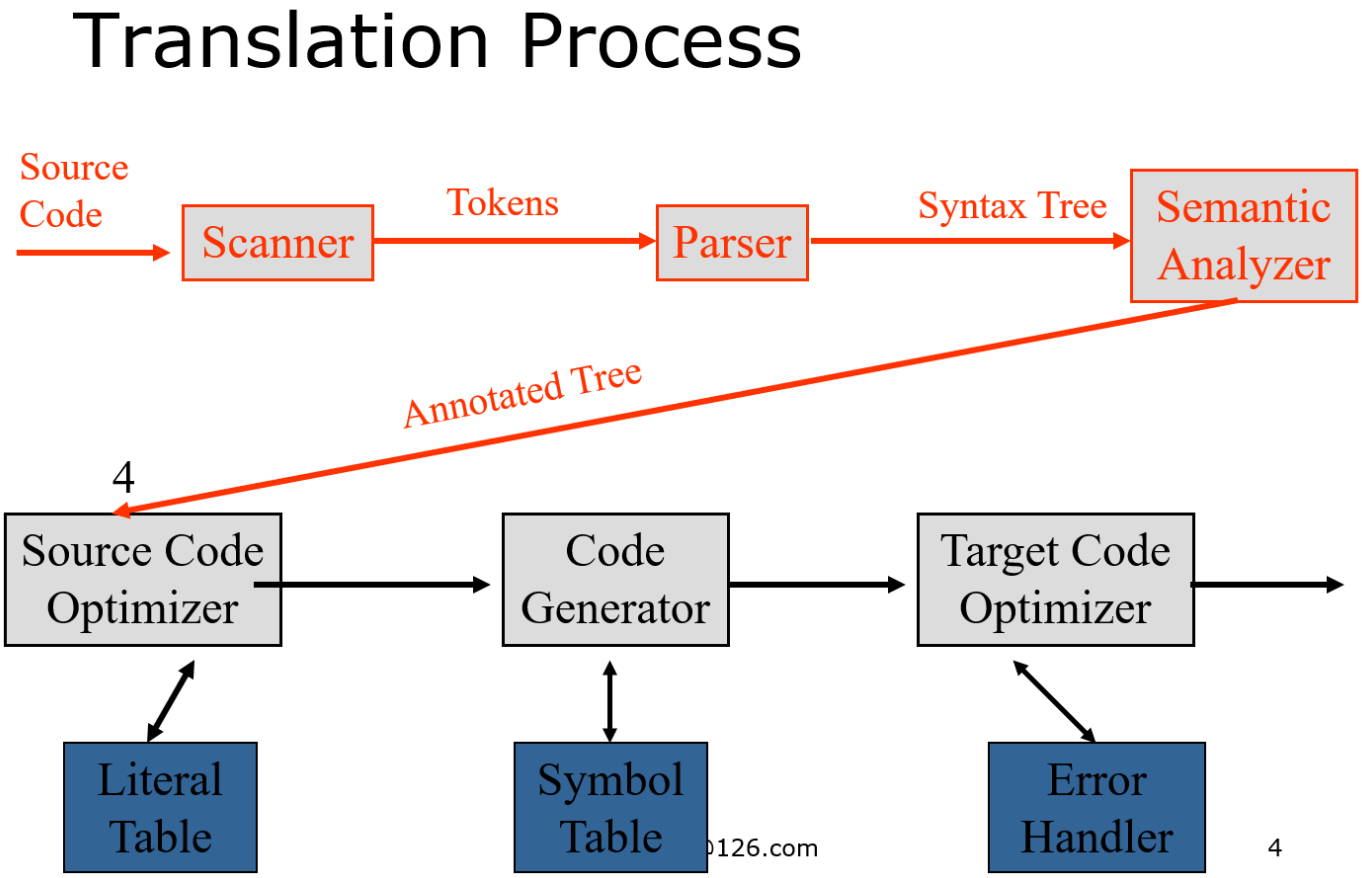
第六章 语义分析



属性和属性文法

属性(attribute)是编程语言结构的任意特性。属性在其包含的信息和复杂性等方面变化很大，特别是当它们能确定时翻译/执行过程的时间。属性的典型例子有：

- 变量的数据类型
- 表达式的值
- 存储器中变量的位置
- 程序的目标代码

- 数的有效位数

属性的计算及将计算值与正在讨论的语言结构联系的过程称作属性的联编(binding)。联编属性发生时编译/执行过程的时间称作联编时间 (binding time)

在执行之前联编的属性称作静态的(static)，而只在执行期间联编的属性是动态的(dynamic)

注意：编译器绝不会产生动态属性，因为先编译再执行，编译时还未执行，因此不可能产生动态的属性。

属性文法

在语法制导语义(syntax-directed semantics)中，属性直接与语言的文法符号相联系 (终结符号或非终结符号) 。如果X是一个文法符号，a是X的一个属性，那么我们把与X关联的a的值记作X.a。

> 若有一个属性的集合$a_1, \ldots, a_k$，语法制导语义的原理应用于每个文法规则$X_0 \rightarrow X_1 X_2 \ldots Xn$ (这里X_0是一个非终结符号，其他的X_i都是任意符号)，每个文法符号X_i的属性$X_i.a_j$的值与规则中其他符号的属性值有关。如果同一个符号X_i在文法规则中出现不止一次，那么每次必须用合适的下标与在其他地方出现的符号区分开来。每个关系用属性等式 (attribute equation)或语义规则(semantics rule) [e] 表示，形式如下
>
> $$X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, X_1.a_1, \ldots, X_1.a_k, \ldots, X_n.a_1, \ldots, X_n.a_k)$$
>
> 这里的f_{ij}是一个数学函数。属性$a_1, \ldots, a_k$的属性文法(attribute grammar)是对语言的所有文法规则的所有这类等式的集合。

eg:

> **例6.1** 考虑下面简单的无符号数文法：
>
> $$number \rightarrow number\ digit \mid digit$$
> $$digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$
>
> 一个数最重要的属性是它的值，我们将其命名为 *val*。每个数字都有一个值，可以用它表示的实际数直接计算。因此，例如，文法规则 $digit \rightarrow 0$ 表明在这个情况下 *digit* 的值为0。这可以用属性等式$digit.val = 0$ 表示，我们将这个等式和规则$digit \rightarrow 0$ 联系在一起。此外，每个数都有一个基于它所包含的数字的值。如果一个数使用了下面的规则推导
>
> $$number \rightarrow digit$$
>
> 那么这个数就只包含了一个数字，其值就是这个数字的值。用属性等式表示为
>
> $$number.val = digit.val$$
>
> 如果一个数包含的数字多于1个，可以使用下列文法规则推导
>
> $$number \rightarrow number\ digit$$
>
> 我们必须表示出这个文法规则左边符号的值和右边符号的值之间的关系。请读者注意，在这个文法规则中对*number* 的两次出现必须进行区分，因为右边的*number* 和左边的*number*的值不相同。我们使用下标进行区分，将这个文法写成如下形式：
>
> $$number_1 \rightarrow number_2\ digit$$

eg: 考点

例6.2 考虑下列简单的整数算术表达式文法：

$$exp \rightarrow exp + term \mid exp - term \mid term$$
$$term \rightarrow term * factor \mid factor$$
$$factor \rightarrow (exp) \mid \textbf{number}$$

val的属性等式:注意，左右两侧如果有相同的部分（比如第一行左右两侧都有exp），要通过添加下标加以区分

文法规则	语义规则
$exp_1 \rightarrow exp_2 + term$	$exp_1.val = exp_2.val + term.val$
$exp_1 \rightarrow exp_2 - term$	$exp_1.val = exp_2.val - term.val$
$exp \rightarrow term$	$exp.val = term.val$
$term_1 \rightarrow term_2 * factor$	$term_1.val = term_2.val * factor.val$
$term \rightarrow factor$	$term.val = factor.val$
$factor \rightarrow (exp)$	$factor.val = exp.val$
$factor \rightarrow \textbf{number}$	$factor.val = \textbf{number}.val$

eg:

例6.3 考虑下列类似C语法中变量声明的简单文法：

$$decl \rightarrow type\ var\text{-}list$$
$$type \rightarrow \textbf{int} \mid \textbf{float}$$
$$var\text{-}list \rightarrow \textbf{id}, var\text{-}list \mid \textbf{id}$$

我们要为在声明中标识符给出的变量定义一个数据类型属性，并写出一个等式来表示数据类型属性是如何与声明的类型相关的。通过构造 *dtype*属性的一个属性文法可以实现这一点 (使用名字*dtype*和非终结符*type*的属性进行区分)。*dtype*的属性文法在表6-3中给出。在图中关于属性等式我们做了以下标记。

首先，从{*integer* , *real*}集合中得出*dtype*的值，相应的记号为**int**和**float**。非终结符*type*有一个它表示的记号给定的 *dtype*。通过*decl*文法规则的等式，这个 *dtype*对应于全体*var-list*的*dtype*。通过*var-list*的等式，表中的每个 **id**都有相同的*dtype*。注意，没有等式包含非终结符*decl* 的*dtype*。实际上*decl*并不需要*dtype*，一个属性的值没有必要为所有的文法符号指定。

文法规则	语义规则
$decl \rightarrow type\ var\text{-}list$	$var\text{-}list.dtype = type.dtype$
$type \rightarrow \textbf{int}$	$type.dtype = integer$
$type \rightarrow \textbf{float}$	$type.dtype = real$
$var\text{-}list_1 \rightarrow \textbf{id}, var\text{-}list_2$	$\textbf{id}.dtype = var\text{-}list_1.dtype$
	$var\text{-}list_2.dtype = var\text{-}list_1.dtype$
$var\text{-}list \rightarrow \textbf{id}$	$\textbf{id}.dtype = var\text{-}list.dtype$

相关图和赋值顺序

给定一个属性文法，每个文法规则选择有一个相关依赖图 (associated dependency graph)。文法规则中的每个符号在这个图中都有用每个属性 $X_i.a_j$ 标记的节点，对每个属性等式

$$X_i.a_j = f_{ij}(\ldots, X_m.a_k, \ldots)$$

相关于文法规则从在右边的每个节点 $X_m.a_k$ 到节点 $X_i.a_j$ 有一条边(表示 $X_i.a_j$ 对 $X_m.a_k$ 的依赖)。依据上下文无关文法，在语言产生时给定一个合法的字符串，这个字符串的依赖图 (dependency graph)就是字符串语法树中选择表示每个(非叶子)节点文法规则依赖图的联合。

在绘制每个文法规则或字符串的依赖图时，与每个符号 X 相关的节点均画在一组中，这样依赖就可以看作是语法树的构造。

eg：见6.1ppt

顺序约束:

给定要转换的一个特定的记号字符串，字符串语法树的相关图根据计算字符串属性的算法给出了一系列顺序约束。实际上，任一个算法在试图计算任何后继节点的属性之前，必须计算相关图中每个节点的属性。遵循这个限制的相关图遍历顺序称作拓扑排序 (topological sort)，而且众所周知,存在拓扑排序的充分必要条件是相关图必须是非循环 (acyclic)的。这样的图形称作确定非循环图 (directed acyclic graphs,DAGs)。

图的根节点没有前驱节点，这些节点的值不依赖于任何其他属性。

合成属性和继承属性

（客观题常考）

定义 一个属性是合成的(synthesized)，如果在语法树中它所有的相关都从子节点指向父节点。等价地，一个属性 a 是合成的，如果给定一个文法规则 $A \rightarrow X_1 X_2 \ldots X_n$，左边仅有一个 a 的相关属性等式有以下形式

$$A.a = f(X_1.a_1, \ldots, X_1.a_k, \ldots, X_n.a_1, \ldots, X_n.a_k)$$

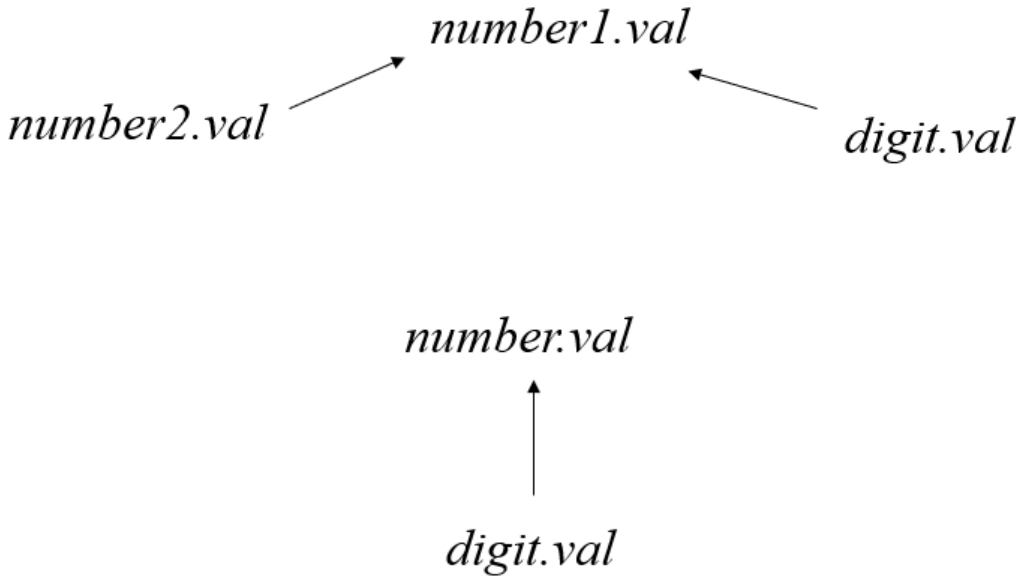
一个属性文法中所有的属性都是合成的，就称作S属性文法(S-attributed grammar)。

eg：给定由分析程序构造的分析树或语法树，S属性文法的属性值可以通过对树进行简单的自底向上或后序遍历来计算。

```
procedure PostEval (T: treenode);
for each child C of T do
    PostEval(C);
compute all synthesized attributes of T
```

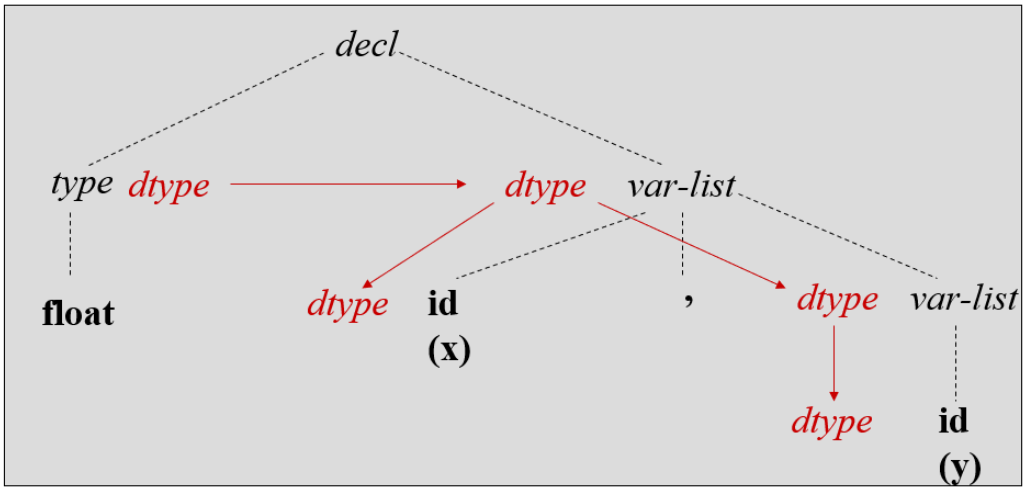
eg：合成属性所有的相关都从子节点指向父结点

Example

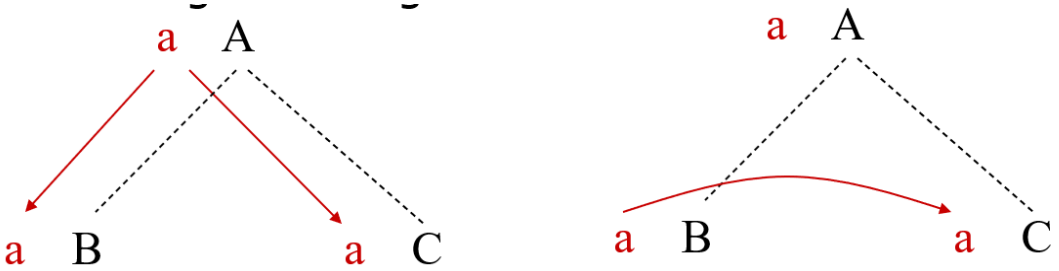


继承属性：一个属性如果不是合成的，就称作继承属性

eg:



无论分析树中，从祖先到子孙的继承属性还是从同属的继承属性都有依赖



继承属性的计算可以通过对分析树或语法树的前序遍历或前序/中序遍历的组合来进行

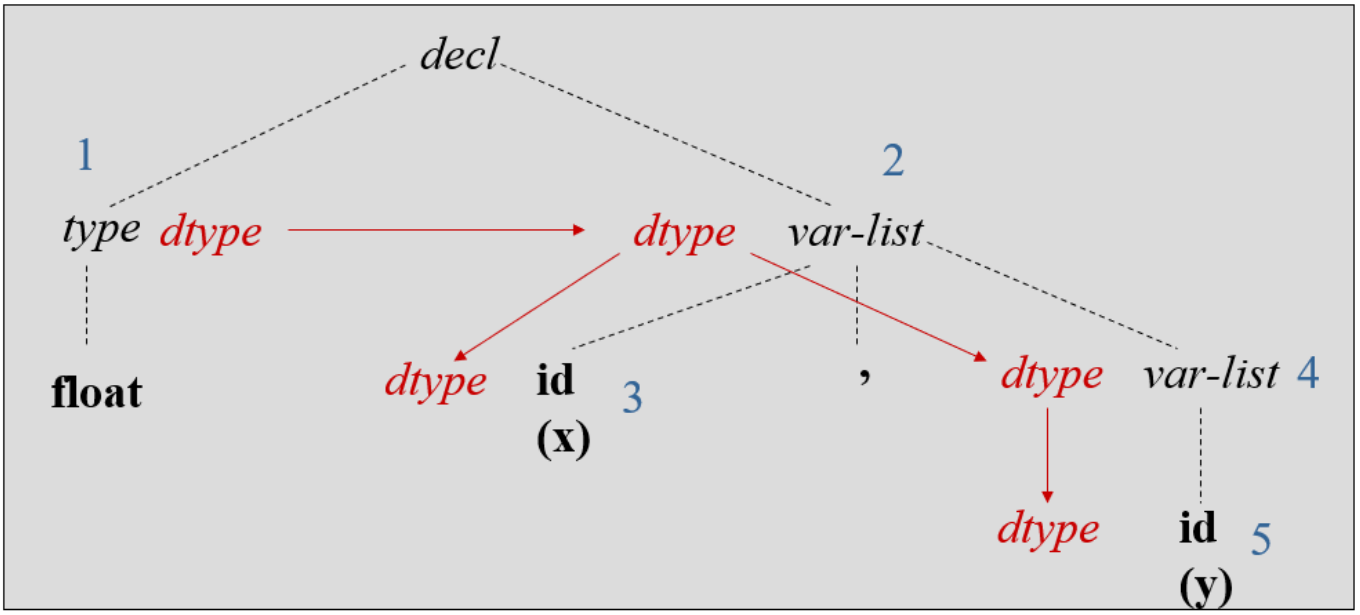
eg:

```
procedure PreEval (T: treenode);
for each child C of T do
    compute all inherited attributes of C
    PreEval(C);
```

eg:

Example

decl → *type var-list*
type → **int** | **float**
var-list → **id** , *var-list* | **id**



语法中属性计算的相关性

属性严重依赖于文法结构。可能会有这样的情况，不改变语言合法的字符串而修改文法会使属性的计算更简单或更复杂。

定理：给定一个属性文法，通过适当地修改文法，而无需改变文法的语言，所有的继承属性可以改变成合成属性

eg:

例6.18　考虑前一个例子中简单声明的文法：

$$decl \rightarrow type\ var\text{-}list$$
$$type \rightarrow \texttt{int} \mid \texttt{float}$$
$$var\text{-}list \rightarrow \texttt{id,}\ var\text{-}list \mid \texttt{id}$$

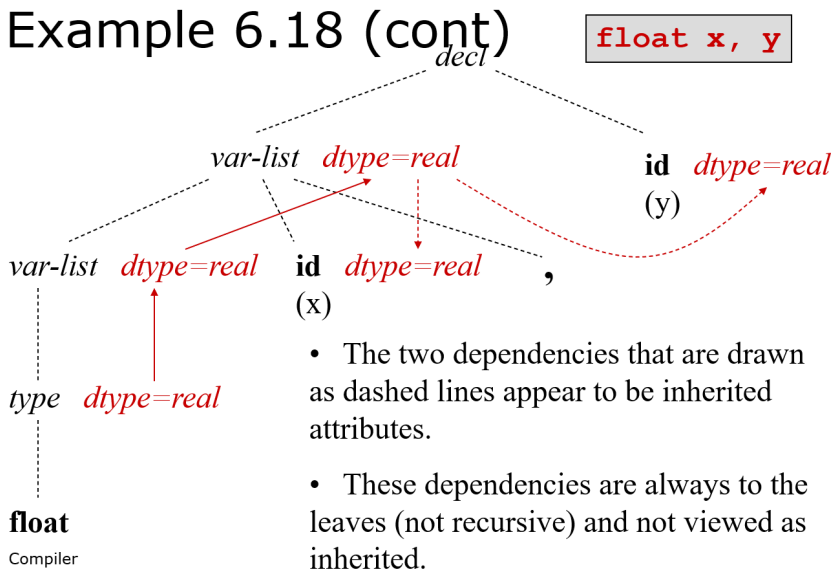
表6-3的属性文法的*dtype*属性是继承属性。然而，如果重写这个文法如下：

$$decl \rightarrow var\text{-}list\ \texttt{id}$$
$$var\text{-}list \rightarrow var\text{-}list\ \texttt{id,} \mid type$$
$$type \rightarrow \texttt{int} \mid \texttt{float}$$

那么产生了相同的字符串，但根据下面的属性文法，属性*dtype*现在变成了合成属性：

文法规则	语义规则
$decl \rightarrow var\text{-}list\ \texttt{id}$	$\texttt{id}.dtype = var\text{-}list.dtype$
$var\text{-}list_1 \rightarrow var\text{-}list_2\ \texttt{id}\ ,$	$\texttt{id}.dtype = var\text{-}list_2.dtype$ $var\text{-}list_1.dtype = var\text{-}list_2.dtype$
$var\text{-}list \rightarrow type$	$var\text{-}list.dtype = type.dtype$
$type \rightarrow \texttt{int}$	$type.dtype = integer$
$type \rightarrow \texttt{float}$	$type.dtype = real$

两个继承的值用虚线标出



符号表

【编译原理—符号表和类型表】 https://www.bilibili.com/video/BV1Xt421b7bV/?share_source=copy_web&vd_source=d80f309439bdd49df0b0808ccb80749a

笔记：http://t.csdnimg.cn/SGAt9

符号表的组织与操作

符号表的作用

- 登记各类名字的信息
- 编译各阶段都需要使用符号表
  - 一致性检查和作用域分析
  - 辅助代码生成

符号表的组织

- 符号表的每一项(入口)包含两大栏
  - 名字栏，也称主栏，关键字栏
  - 信息栏，记录相应的不同属性，分为若干子栏
- 按名字的不同种属建立多张符号表，如常数表、 变量名表、过程名表、…

符号表的组织

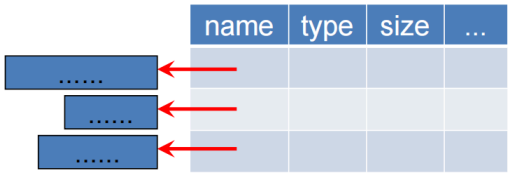
- 符号表的每一项(入口)包含两大栏
- 按名字的不同种属建立多张符号表，如常数表、 变量名表、过程名表、…

▶ 对符号表的操作
- ▶ 填入名称
- ▶ 查找名字
- ▶ 访问信息
- ▶ 填写修改信息
- ▶ 删除

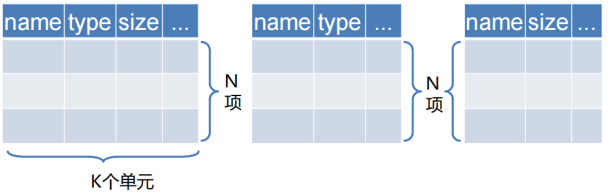
	NAME	INFORMATION
(1)	index	整型，变量
(2)	score	实型，变量
(3)	p	整型，形式参数

▶ 对符号表进行操作的时机
- ▶ 定义性出现
  - ▶ int index
- ▶ 使用性出现
  - ▶ if index < 100 …

▶ 栏目的长度
- ▶ 安排各项各栏的存储单元为固定长度
- ▶ 用间接方式安排各栏存储单元



▶ 符号表的存放
- ▶ 把每一项置于连续K存储单元中，构成一张K*N的表(N为符号表的项数)
- ▶ 把整个符号表分成M个子表，如T_1、T_2、…、T_m，每个子表含N项

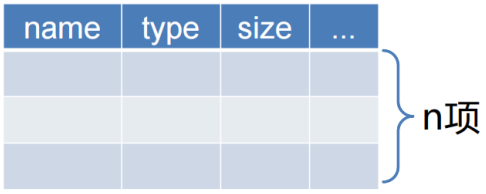


整理和查找

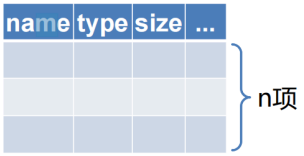
- 线性查找

- 二分查找

▶ 按关键字出现的顺序填写各项
  ▶ 结构简单，节省空间，填表快，
  ▶ 查找慢，时间复杂度O(n)
  ▶ 改进：自适应线性表

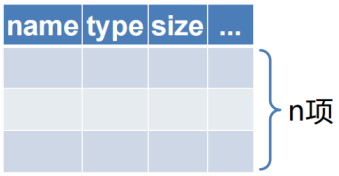


▶ 表格中的项按名字的“大小”顺序整理排列
▶ 填表慢，查找快
▶ 时间复杂度:O(Log_2n)
▶ 改进：组织成二叉树



- 杂凑查找(HASH技术):

▶ 杂凑函数H(SYM)：0~n-1
  ▶ n：符号表的项数
▶ 填表快，查找快
▶ 要求
  ▶ 计算简单高效
  ▶ 函数值分布均匀



符号表的内容

- 符号表的信息栏中登记了每个名字的有关性质
- 类型：整、实或布尔等
- 种属：简单变量、数组、过程、函数等
- 大小：长度，即所需的存储单元字数
- 相对数：指分配给该名字的存储单元的相对地址

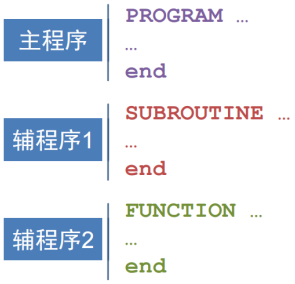
利用符号表分析名字的作用域

名字的作用范围

- 在许多程序语言中，允许同一个标识符在不同过程中代表不同的名字
- 名字都有一个确定的作用范围
- 作用域
  - 一个名字能被使用的区域范围称作这个名字的作用域
- 两种程序体结构
  - 并列结构，如FORTRAN
  - 嵌套结构，如PASCAL，PL

FORTRAN：

- 一个程序由一个主程序段和若干辅程序段组成
- 辅程序段可以是子程序、函数段或数据块
- 每个程序段由一系列的说明语句和执行语句组成，各段可以独立编译
- 模块结构，没有嵌套和递归
- 各程序段中的名字相互独立，同一个标识符在不同的程序段中代表不同的名字
- 把局部名和全局名分别存在不同的地方



PASCAL/PL

- PASCAL/PL程序本身可以看成是一个操作系统调用的过程，过程可以嵌套和递归
- 一个PASCAL/PL过程

```
过程头;
    说明段（由一系列的说明语句组成）;
begin
    执行体（由一系列的执行语句组成）;
end
```