北京郵電大學

第6章　语义分析

LI Wensheng, SCS, BUPT

知识点：符号表

类型体制

各语法成分的类型检查

语义分析

6.1 语义分析概述

6.2 符号表

6.3 类型检查

6.4 一个简单的类型检查程序

小 结

6.1 语义分析概述

- 程序设计语言的结构可由上下文无关文法来描述。语法分析可以检查源程序中是否存在语法错误。
- 没有语法错误的源程序一定正确吗？
- 程序正确与否与结构的上下文有关
  - 变量的作用域问题
  - 同一作用域内同名变量的重复声明问题
  - 表达式、赋值语句中的类型一致性问题等
- 思考：
  - 设计上下文有关文法来描述语言中上下文有关的结构？
  - 理论可行，构造困难，构造分析程序更困难。
- 解决办法：
  - 利用语法制导翻译技术实现语义分析
  - 设计专门的语义动作补充上下文无关文法的分析程序

```
main()
{
    int i, j;
    i=0;  j=1;
    {
        int i, k;
        k=10;
        j=k+j;
    };
    i=j*k;
}
```

6.1.1 语义分析的任务

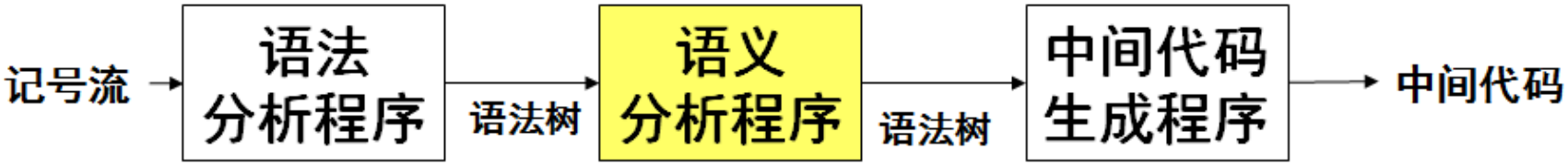
- 语义分析程序通过将变量的定义与变量的引用联系起来，对源程序的含义进行检查。
  检查每一个语法成分是否具有正确的语义。
- 语义分析的任务
  （1）收集并保存上下文有关的信息；
  （2）类型检查。
- 符号表的建立和管理
  - 在分析声明语句时，收集所声明标识符的有关信息，如类型、存储位置、作用域等，并记录在符号表中。
  - 只要编译时控制处于声明该标识符的程序块中，就可以从符号表中查到它的记录。

类型检查

- 动态检查：目标程序运行时进行的检查
- 静态检查：读入源程序但不执行源程序的情况下进行的检查
- 由类型检查程序完成。
  - 检验结构的类型是否与其上下文所期望的一致，检查操作的合法性和数据类型的相容性。如：
    - 内部函数 mod 的运算对象的类型
    - 表达式中各运算对象的类型
    - 用户自定义函数的参数类型、返回值类型
  - 唯一性检查
    - 一个标识符在同一作用域中必须且只能被说明一次
    - CASE语句中用于匹配选择表达式的常量必须各不相同
    - 枚举类型定义中的各元素不允许重复
  - 控制流检查
    - 检查控制语句是否使控制转移到一个合法的位置。

6.1.2 语义分析程序的位置

- 语义分析程序的位置：



  - 以语法树为基础，根据源语言的语义，检查每个语法成分在语义上是否满足上下文对它的要求。
  - 输出的是带有语义信息的语法树。
- 语义分析的结果有助于生成正确的目标代码
  - 重载运算符：一个运算符在不同的上下文中表示不同的运算
  - 类型强制：编译程序把运算对象变换为上下文所期望的类型

6.1.3 错误处理

- 语义相关的错误：
  - 同一作用域内标识符重复声明
  - 标识符未声明
  - 可执行语句中的类型错误
  - 如程序：
- 错误处理：
  - 显示出错信息。报告错误出现的位置和错误性质。
  - 错误恢复。恢复分析器到某同步状态，为了能够对后面的结构继续进行检查。

```
main()
{
    int i, j;
    float x;
    i=0;  j=1;
    x=2;
    {
       int i, k;
       k=10;
    };
    i=j*k;
    j=i+x;
}
```

6.2 符号表

- 符号表在翻译过程中起两方面的重要作用：
  - 检查语义（即上下文有关）的正确性
  - 辅助正确地生成代码
- 通过在符号表中插入和检索标识符的属性来实现
- 符号表是一张动态表
  - 在编译期间符号表的入口不断地增加
  - 在某些情况下又在不断地删除
- 编译程序需要频繁地与符号表进行交互，符号表的效率直接影响编译程序的效率。

符号表

6.2.1 符号表的建立和访问时机

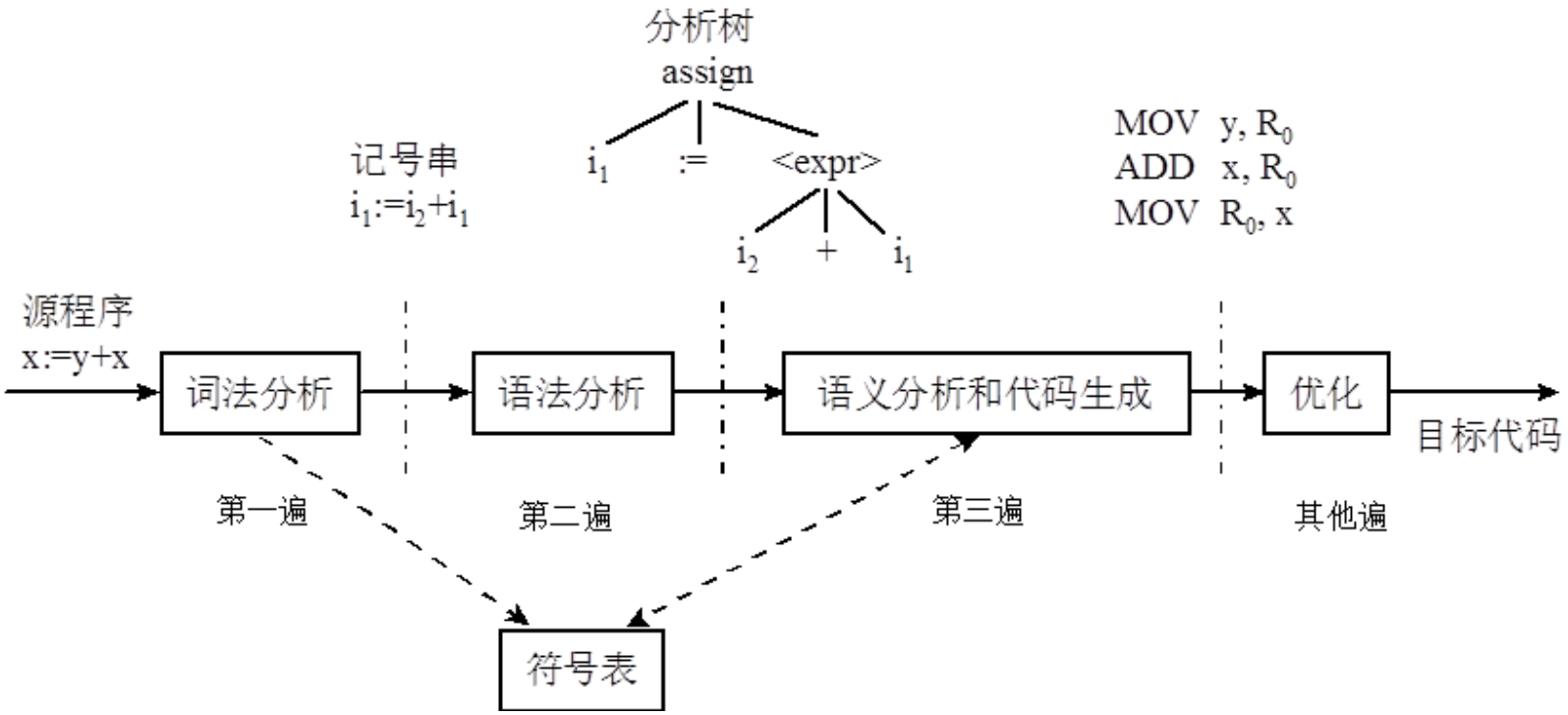
6.2.2 符号表内容

6.2.3 符号表操作

6.2.4 符号表组织

6.2.1 符号表的建立和访问时机

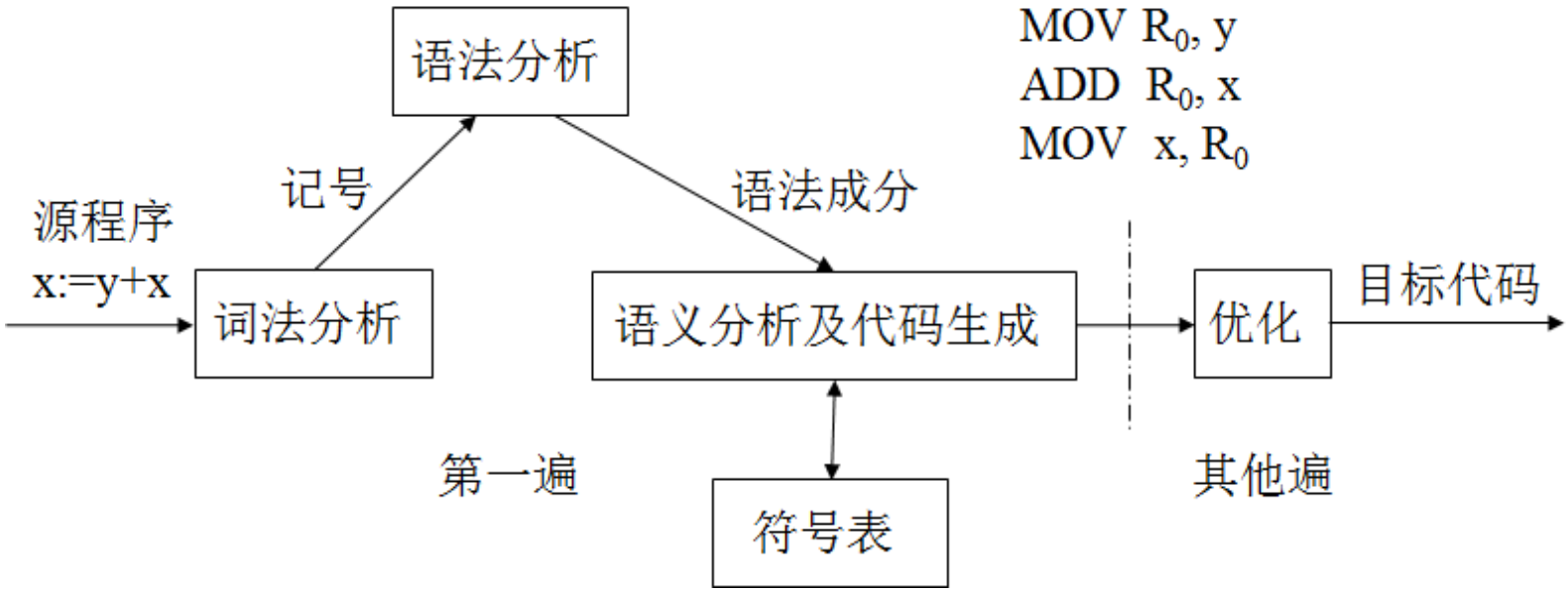
1. 多遍编译程序



- 词法分析阶段建立符号表
- 标识符在符号表中的位置作为记号的属性
- 适用于非块结构语言的编译

符号表的建立和访问时机（续）

2. 合并遍的编译程序



- 语法分析程序是核心模块
- 当声明语句被识别出来时，标识符和它的属性一起写入符号表中。

6.2.2 符号表内容

- 符号表中记录的是和标识符相关的属性
- 出现在符号表中的属性种类，在一定程度上取决于程序设计语言的性质。
- 符号表的典型形式：

序号	名字	类型	存储地址	维数	声明行	引用行	指针
1	counter	2	0	1	2	9, 14, 15	7
2	num_total	1	4	0	3	12, 14	0
3	func_form	3	8	2	4	36, 37, 38	6
4	b_loop	1	48	0	5	10, 11, 13	1
5	able_state	1	52	0	5	11, 23, 25	4
6	mklist	6	56	0	6	17, 21	2
7	flag	1	64	0	7	28, 29	3

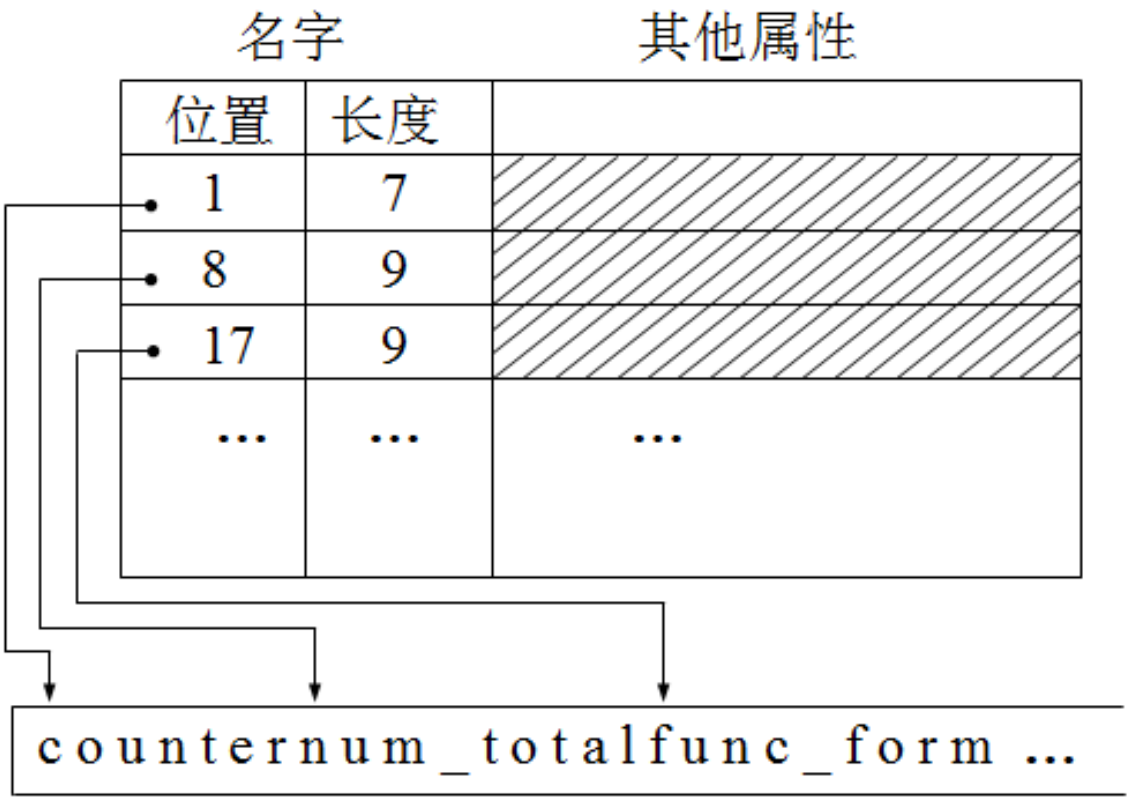
名字

- 编译程序识别一个具体标识符的依据，是符号表必须记录的一个属性。
- 必须常驻内存
- 问题：标识符长度是可变的
- 解决办法：
  - 标识符长度有限制：设置一个长度固定的域，它的长度为该语言允许的标识符最大长度。
  - 标识符长度没有限制：设置一个长度固定的域，域内存放一个串描述符，包含位置指针和长度两个子域，指针域指示该标识符在总的串区内的开始位置，长度域记录该标识符中的字符数。

存取速度较快，
存储空间利用率较低

存取速度较慢，
节省存储空间。

使用串描述符表示变量



类型

- 当所编译的语言有数据类型（隐式或显式的）时，必须把类型属性存放到符号表中。
- 对于无类型的语言，可删除该域。

- 标识符的类型属性用于：
  - 类型检查
  - 生成代码
  - 空间分配

- 类型属性以一种编码形式存放在符号表中。

存储地址

- 记录运行时变量值存放空间的相对位置。
  - 分析声明语句时，将变量的存储地址写入符号表中。
  - 分析对变量的引用语句时，从符号表中取出该地址、并写入相应的目标指令中，生成对该存储地址进行访问的指令。
- 对于静态存储分配的语言（如FORTRAN），目标地址按顺序连续分配，从0开始到m（m是分配给一个程序的数据区的最大值）。
- 对于块结构的语言(如Pascal、C)，通常采用二元地址<blkn, offset>
  - blkn ：块的嵌套深度，用于确定分配给声明变量的块的数据区的基址。
  - offset ：变量的目标地址偏移量，指示该变量的存储单元在数据区中相对于基址的位置。

数组维数/参数个数

- 数组引用时，其维数应当与数组声明时定义的维数一致。
  - 类型检查阶段需要对这种一致性（维数、每维的长度）进行检查
  - 维数用于数组元素地址的计算。
- 过程调用时，实参必须与形参一致。
  - 实参的个数与形参的个数一致
  - 实参的类型与相应形参的类型一致
- 在符号表组织中：
  - 把参数的个数看作它的维数是很方便的，因此，可将这两个属性合并成一个。
  - 这种方法也是协调的，因为对这两种属性所做的类型检查是类似的。

交叉引用表

- 编译程序可以提供的一个十分重要的程序设计辅助工具：交叉引用表
- 编译程序一般设一个选项，用户可以选择是否生成交叉引用表

名字	类型	维数	声明行	引用行
able_state	1	0	5	11, 23, 25
b_loop	1	0	5	10, 11, 13
counter	2	1	2	9, 14, 15
flag	1	0	7	28, 29
func_form	3	2	4	36, 37, 38
mklist	6	0	6	17, 21
num_total	1	0	3	12, 14

链域/指针

- 为了便于产生按字母顺序排列的交叉引用表
- 链域中保存的是符号表中表项的编号，即指针。
- 通过链域，将符号表中所有的表项，按照名字的升序组织成一个链表。
- 如果编译程序不产生交叉引用表，则链域以及语句的行号属性都可以从符号表中删除。

序号	名字	类型	存储地址	维数	声明行	引用行	指针
1	counter	2	0	1	2	9, 14, 15	7
2	num_total	1	4	0	3	12, 14	0
3	func_form	3	8	2	4	36, 37, 38	6
4	b_loop	1	48	0	5	10, 11, 13	1
5	able_state	1	52	0	5	11, 23, 25	4
6	mklist	6	56	0	6	17, 21	2
7	flag	1	64	0	7	28, 29	3

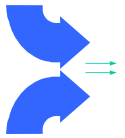
6.2.3 符号表操作

- 最常执行的操作：插入和检索
- 要求标识符显式声明的强类型语言：
  - 编译程序在处理声明语句时调用两种操作
    - 检索：查重、确定新表目的位置
    - 插入：建立新的表目
  - 在程序中引用名字时，调用检索操作
    - 查找信息，进行语义分析、代码生成
    - 可以发现未定义的名字
- 允许标识符隐式声明的语言：
  - 标识符的每次出现都按首次出现处理
  - 检索：
    - 已经声明，进行类型检查。
    - 首次出现，插入操作，从其作用推测出该变量的相关属性。

定位和重定位操作

- 对于块结构的语言，在建立和删除符号表时还要使用两种附加的操作，即定位和重定位。
- 当编译程序识别出块开始时，执行定位操作。
- 当编译程序遇到块结束时，执行重定位操作。
- 定位操作：
  - 建立一个新的子表（包含于符号表中），在该块中声明的所有名字的属性都存放在此子表中。
- 重定位操作：
  - “删除”存放该块中局部名字的子表
  - 这些名字的作用域局部于该块，出了该块后不能再被引用。
- 不同分程序中的相同变量名问题的解决方法：
  - 分程序是按层次嵌套的，如果多层嵌套（如n层），那么这些子符号表也是多层嵌套，并按序生成（即先产生子符号表1，最后产生子符号表n）
  - 在第n层分程序需要查找某一标识符时，先从子符号表n开始查找，如果未查到，再依次查子符号表n-1, n-2, ..., 1，如果在符号表i首先出现该标识符名，这便正是所要查找的结果

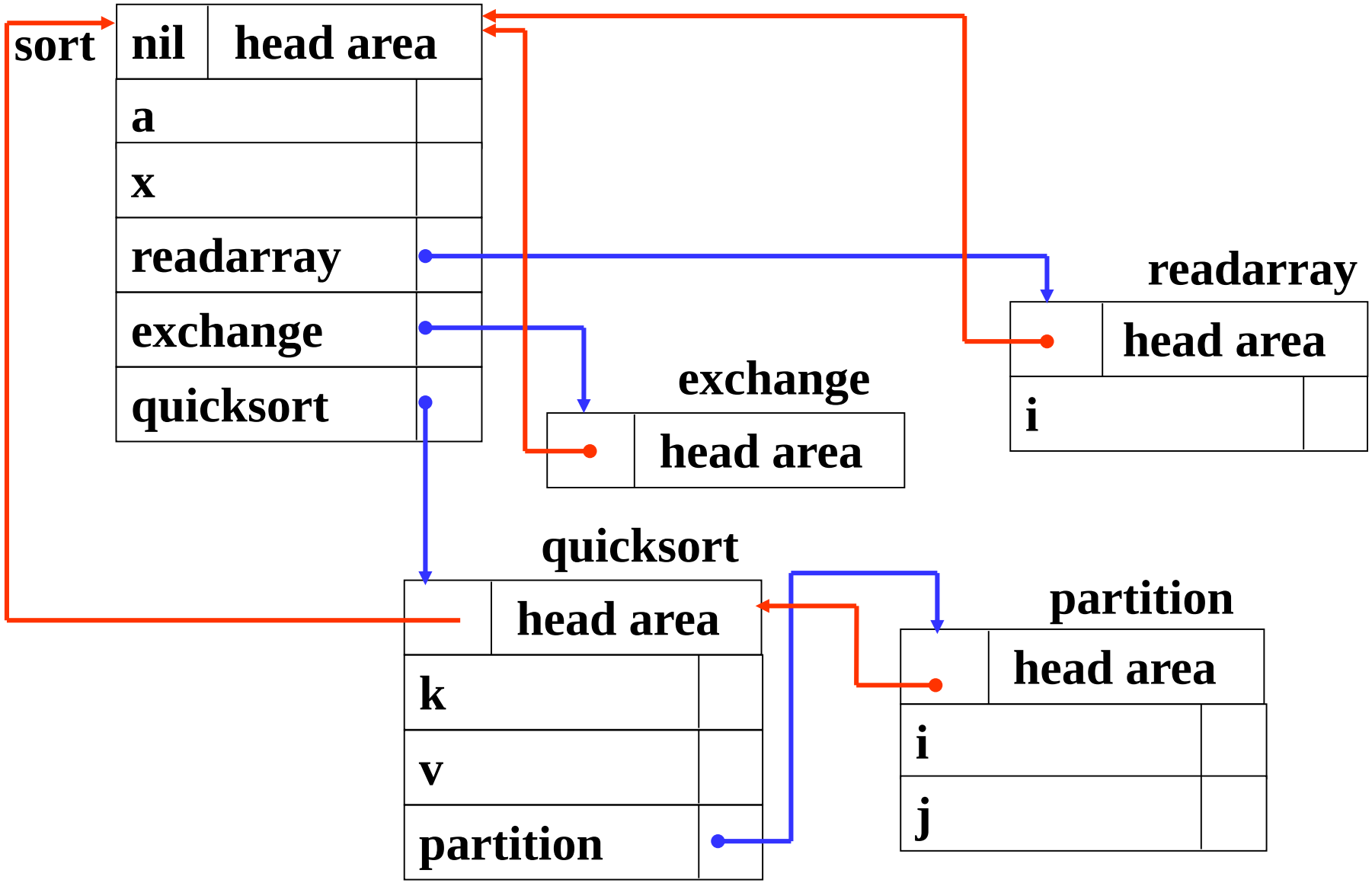
读入数据，并进行排序的PASCAL程序



```
program sort (input,output);
  var a : array[0..10] of integer;
      x : integer;
procedure readarray;
    var i : integer;
    begin
      for i:=1 to 9 do read(a[i])
    end;
procedure exchange (i,j:integer)
    begin
     x:=a[i]; a[i]:=a[j];   a[j]:=x
    end;
```

```
peocedure quicksort (m,n:integer);
  var k,v : integer;
function partition (y,z :integer):integer;
  var i,j : integer;
  begin       …a…;   …v…;
     exchange(i,j);      ……
  end;
  begin   ……
          k=partition(m,n);
          quicksort(m,k-1);
          quicksort(k+1,n);   ……
 end;{quicksort}
begin readarray; quicksort(1,9)
 end. {sort }
```

符号表的逻辑结构


sort
nil
head area
a
x
readarray
exchange
quicksort
readarray
head area
i
exchange
head area
quicksort
head area
k
v
partition
partition
head area
i
j

6.2.4 符号表组织

1. 非块结构语言的符号表组织
2. 块结构语言的符号表组织

1. 非块结构语言的符号表组织

- 非块结构语言：
  - 编写的每一个可独立编译的程序单元是一个不含子模块的单一模块
  - 模块中声明的所有变量属于整个程序
- 符号表组织
  - 无序线性表
    - 属性记录按变量声明/出现的先后顺序填入表中
    - 插入前都要进行检索，若发现同名变量
      - 对显式声明的语言：错误
      - 对隐式声明的语言：引用
    - 适用于程序中出现的变量很少的情况

非块结构语言的符号表组织（续1）

- 有序线性表
  - 按字母顺序对变量名排序的表
  - 线性查找：
    - 遇到第一个比查找变量名值大的表项时，就可以判定该变量名不在表中了。
    - 执行插入操作时，要增加额外的比较和移动操作。
    - 若使用单链结构表的话，可省去表记录的移动，但需要在每个表记录中增加一个链接字段。
  - 折半查找：
    - 首先把变量名与中间项进行比较，结果或是找到该变量名，或是指出下一次要在哪半张表中进行。
    - 重复此过程，直到找到该变量名或确定该变量名不在表中为止。

非块结构语言的符号表组织（续2）

- 散列/哈希表
  - 查找时间与表中记录数无关的一种符号表组织方式

  - 名字空间（即标识符空间）K：
    - 是允许在程序中出现的标识符的集合。
    - 由于在编译程序的具体实现中必须限定标识符的最大长度，故名字空间K总是有限的。
  - 地址空间（也称表空间）A：
    - 是散列表中存储单元的集合{1，2，…，m }。

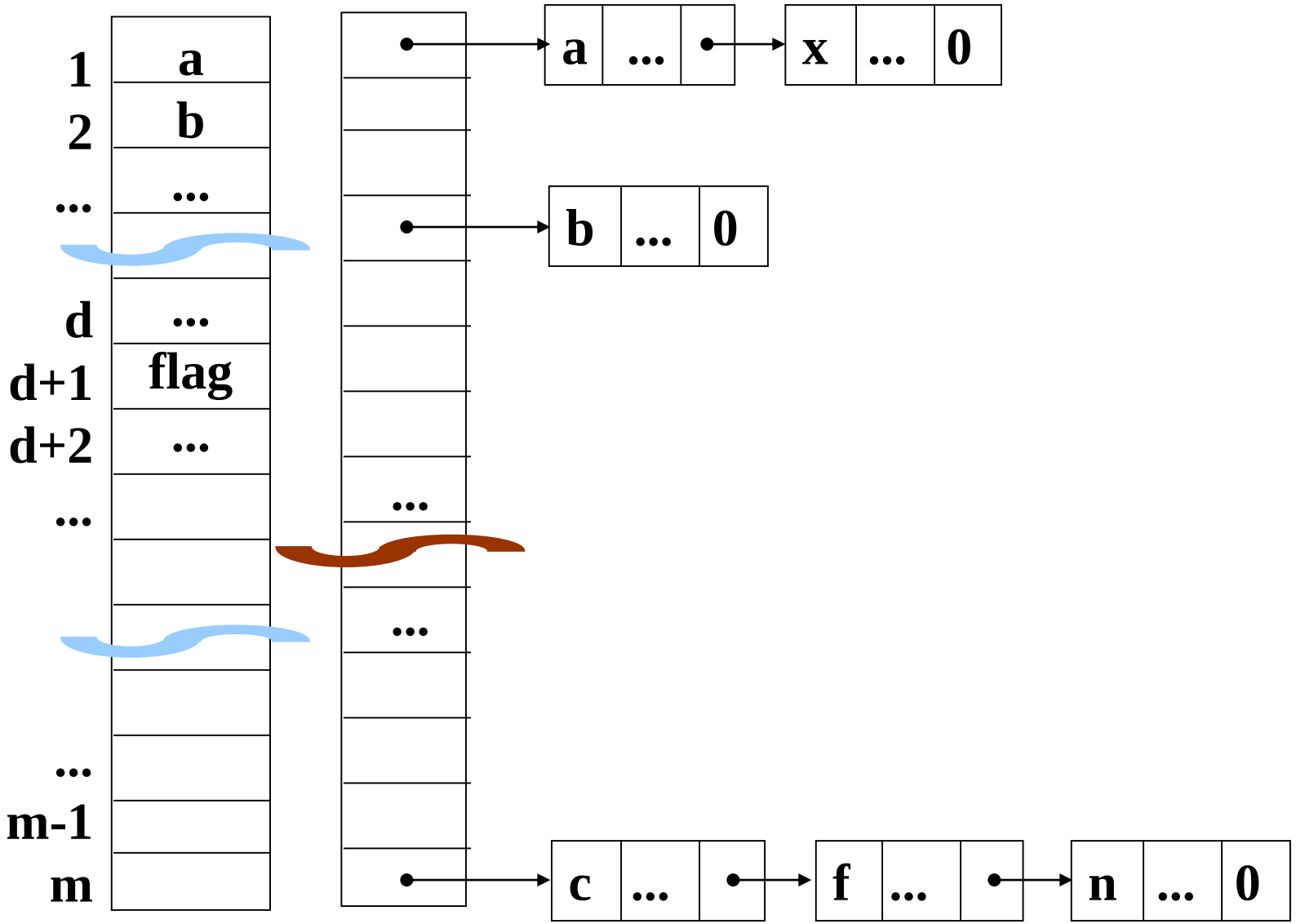
非块结构语言的符号表组织（续3）

- 散列/哈希函数H
  - **除法**：最常用的函数，H(x)=(x mod m)+1，通常m为一个大素数，可使标识符尽可能均匀地分散在表中。
  - **平方取中法**：先求出标识符的平方值，然后按需要取平方值的中间几位作为散列地址。
    因为平方值中间的几位与标识符中每一符号都相关，故不同标识符会以较高的概率产生不同的散列地址。
  - **折叠法**：将标识符按所需地址长度分割成位数相同的几段，最后一段的位数可以不同，然后取这几段的叠加和（忽略进位）作为散列地址。
  - **长度相关法**：标识符的长度和标识符的某个部分一起用来直接产生一个散列地址，或更普遍的方法是产生一个有用的中间字，然后再用除法产生一个最终的散列地址。

解决冲突的方法

- 冲突：变量名被映射到一个存储单元d中，而这个单元已被占用
- 开放地址法
  - 按照顺序d, d+1, …, m, 1, 2, …, d-1进行扫描，直到找到一个空闲的存储单元为止，或者在扫描完m个单元之后搜索停止。
  - 在查找一个记录时，按同样的顺序扫描，或找到要找的记录、或找到一个空闲单元（从未使用过）为止。
- 分离链表法
  - 将发生冲突的记录链到一个专门的溢出区，该溢出区与主区相分离。
  - 为每一组冲突的记录设置一个链表，主区和溢出区的每一个记录都必须有一个链接字段。
  - 为节省存储空间，建立一个中间表（散列表），所有记录都存入溢出区，而主区（散列表）只有链域。

开放地址、分离链表示意图



2. 块结构语言的符号表组织

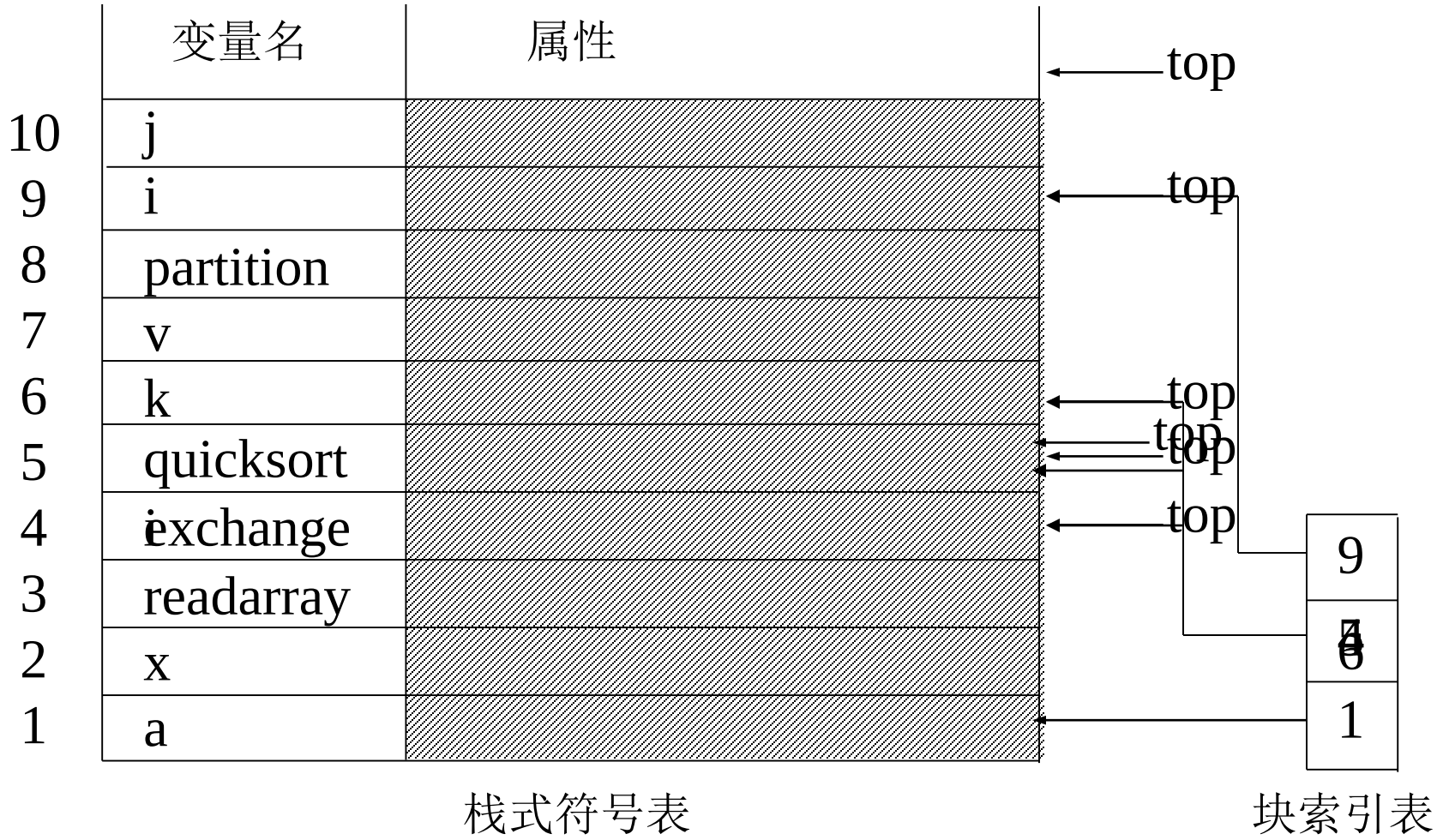
- 块结构语言：
  - 模块中可嵌套子块
  - 每个块中均可以定义局部变量
- 每个程序块有一个子表，保存该块中声明的名字及其属性。
- 符号表组织
  - 栈式符号表
  - 栈式散列符号表

读入数据，并进行排序的PASCAL程序

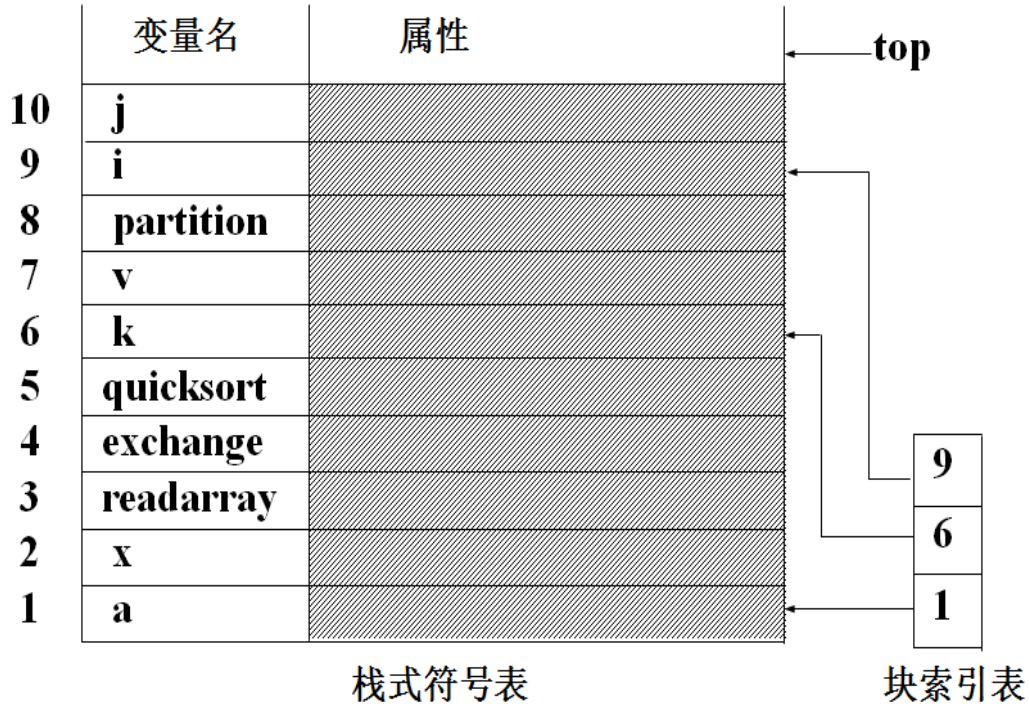
```
program sort (input,output);
  var a : array[0..10] of integer;
      x : integer;
  procedure readarray;
    var i : integer;
    begin
      for i:=1 to 9 do read(a[i])
    end;
procedure exchange (i,j:integer)
    begin
     x:=a[i]; a[i]:=a[j];   a[j]:=x
    end;
```

```
peocedure quicksort (m,n:integer);
  var k,v : integer;
  function partition (y,z :integer):integer;
    var i,j : integer;
    begin
        …a…;    …v…;
        exchange(i,j);        ……
    end;
  begin   ……
          k=partition(m,n);
          quicksort(m,k-1);
          quicksort(k+1,n);   ……
 end;{quicksort}
begin
  readarray; quicksort(1,9)
end. {sort }
```

栈式符号表


变量名
属性
top
10
j
9
i
top
8
partition
7
v
6
k
top
5
quicksort
top
top
4
exchange
top
3
readarray
2
x
1
a
9
1
栈式符号表
块索引表

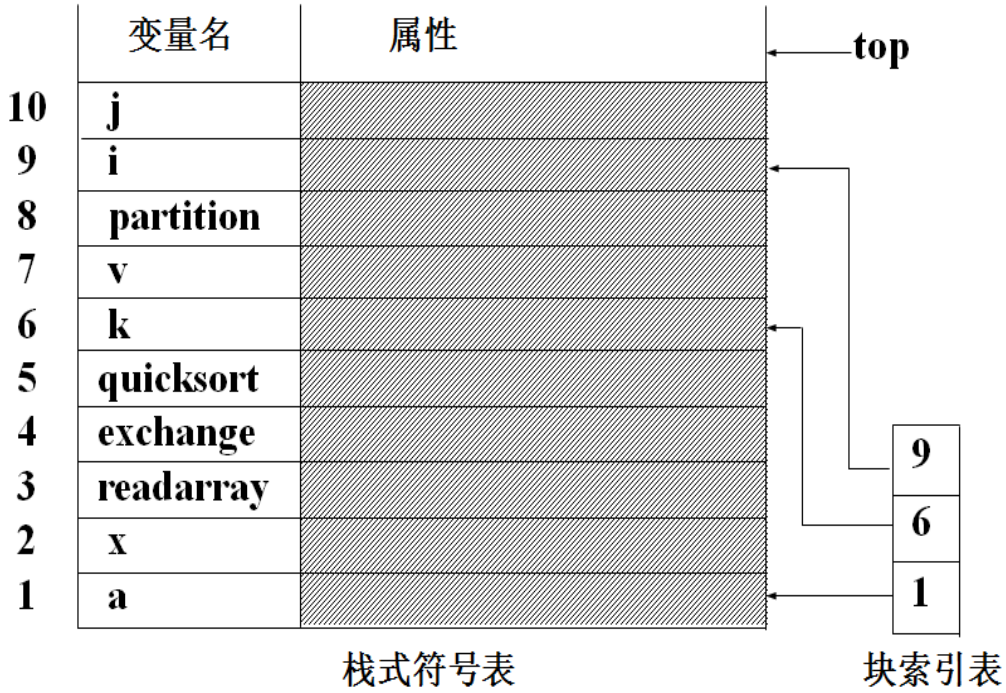
栈式符号表操作



- 插入
  - 检查子表中是否有重名变量
    - 无，新记录压入栈顶
    - 有，报告错误
- 检索
  - 从栈顶到栈底线性检索
    - 在当前子表中找到，局部变量
    - 在其他子表中找到，非局部名字
  - 实现了最近嵌套作用域原则
    - 第一次检索到的名字就是按照最近嵌套作用域原则要求的名字的申明

栈式符号表操作（续）

- 定位
  - 将栈顶指针top的值压入块索引表顶端。
  - 块索引表的元素是指针，指向相应块的子表中第一个记录在栈中的位置。



- 重定位
  - 用块索引表顶端元素的值恢复栈顶指针top，完成重定位操作。
  - 清除刚刚被编译完的块在栈式符号表中的所有记录。
  - top指示符号表栈顶第一个空闲的记录存储单元。

栈式散列符号表

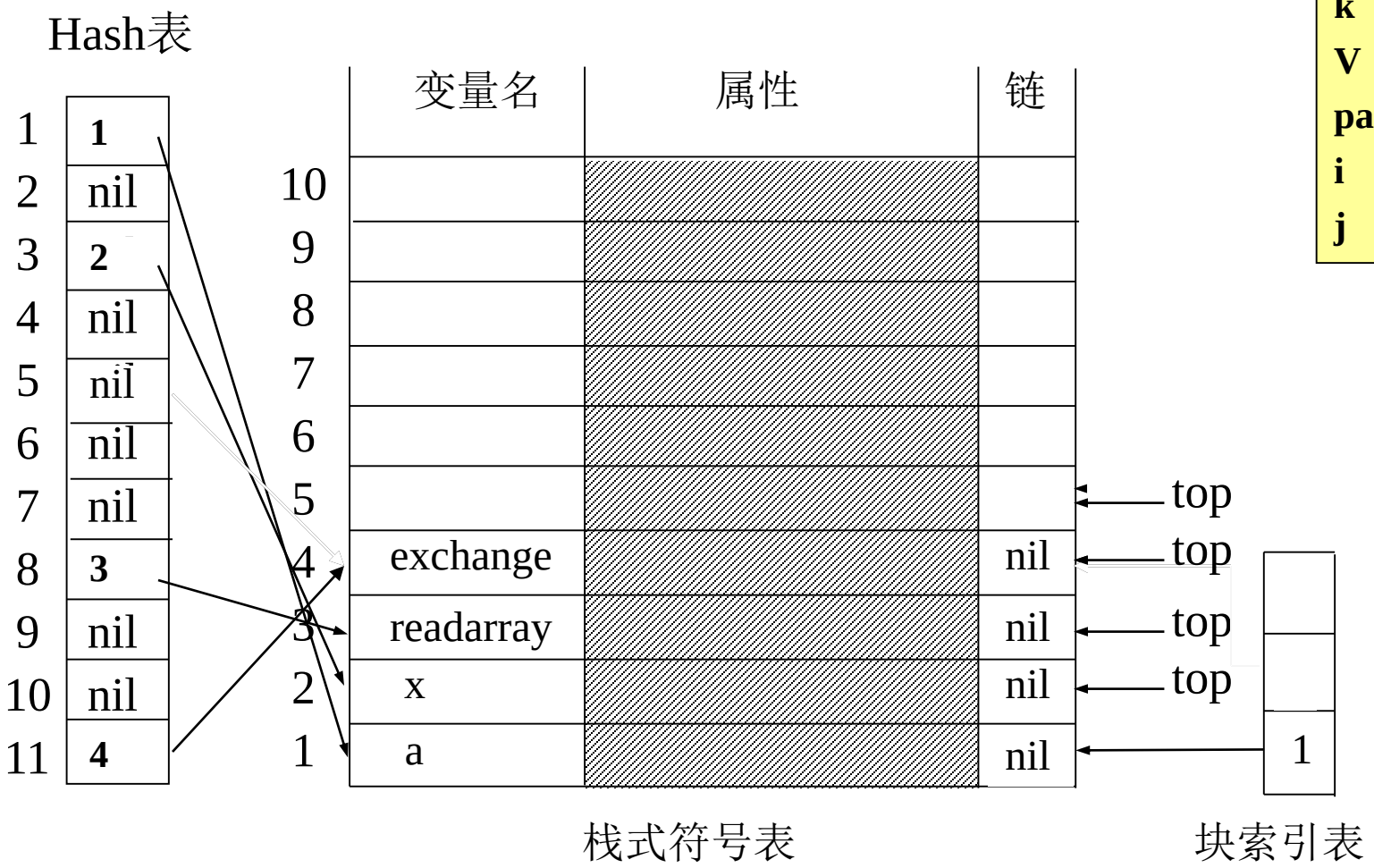
- 假设散列表的大小为11，散列函数执行如下变换：

名字	映射到地址
a、quicksort	1
x、v、j	3
partition	4
i	5
k、readarray	8
exchange	11

栈式散列符号表示意图

a, x, readarray, exchange, quicksort, k, v, partition, i, j

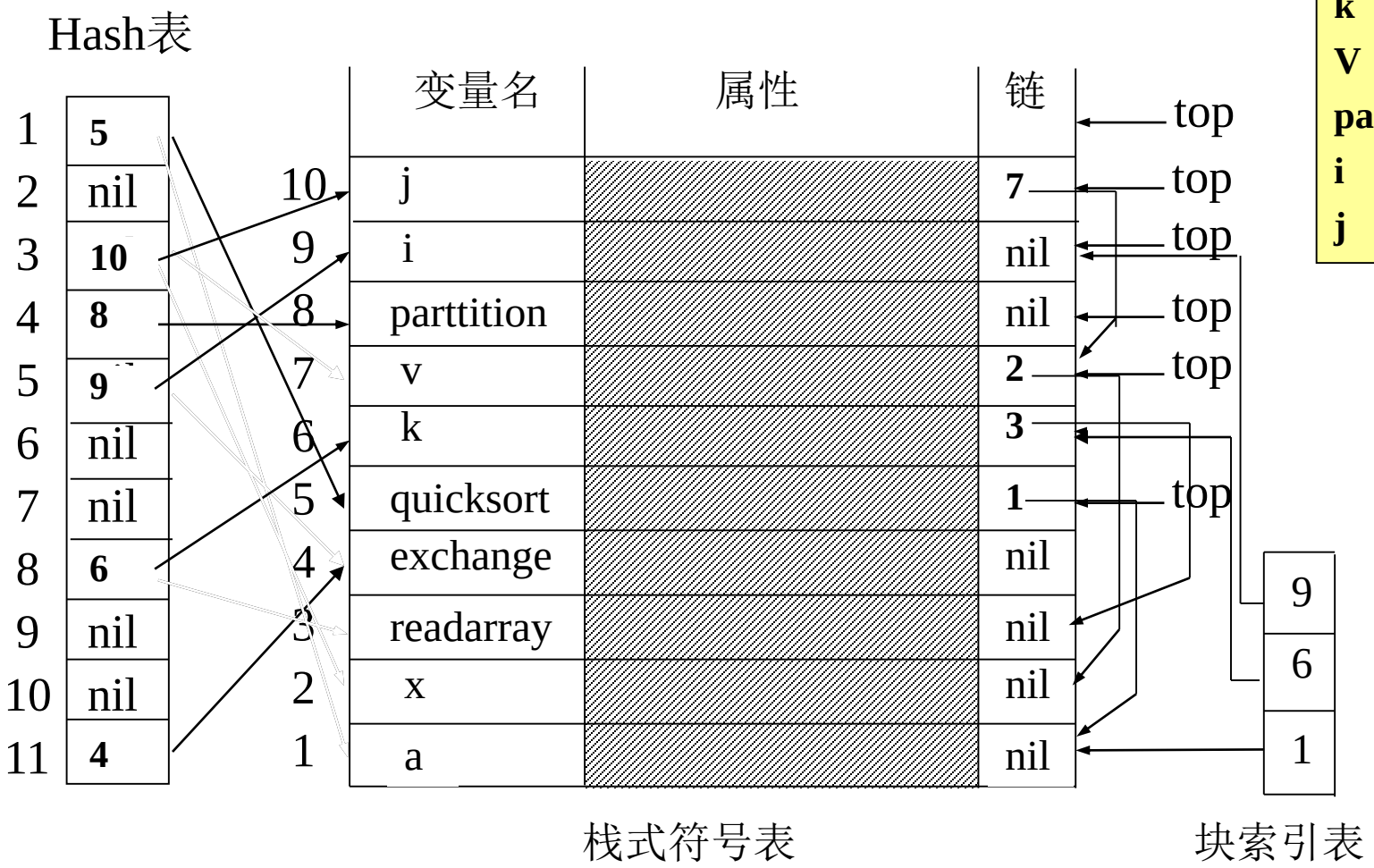
a	1
x	3
readarray	8
exchange	11
quicksort	1
k	8
V	3
partition	4
i	5
j	3



栈式散列符号表示意图

a, x, readarray, exchange, quicksort, k, v, partition, i, j

a	1
x	3
readarray	8
exchange	11
quicksort	1
k	8
V	3
partition	4
i	5
j	3



栈式散列符号表操作

- 插入
  - 散列函数将标识符映射到散列表单元
  - 是否存在冲突？该表单元是否为空？
    - 无冲突：
      - 将栈指针top的值记入该散列表单元
      - 将新记录压入栈顶
    - 有冲突
      - 检查冲突链中是否有同名标识符的重复定义
        - » 没有：将新记录插入冲突链的链头
        - » 有：检查同名标识符是否属于当前子表

          同名标识符在栈中的位置 >= 块索引表顶端元素的值？

          >=：在当前子表中，报告错误

          <：不在当前子表中，将新记录插入冲突链的链头

栈式散列符号表操作（续1）

- 检索
  - 散列函数将标识符名字映射到散列表单元
  - 该散列表单元是否为空？
    - 空：名字未定义，报告错误
    - 不空：沿冲突链检索
      - 未找到：名字未定义，报告错误
      - 找到：名字在栈中的位置>=块索引表顶端元素的值

        >=：局部名字

        <：非局部名字

栈式散列符号表操作（续2）

- 定位
  - 识别出一个新块的开始时，执行定位操作。
  - 将栈顶指针top的值压入块索引表的顶端。
  - 标识新块的符号子表的开始位置
- 重定位
  - 分析到一个块结束时，执行重定位操作。
  - 将该块的有关记录从符号表中“逻辑”/“物理”删除。
    - 用栈顶指针top确定要删除的栈单元
    - 依次取出栈单元中的名字
      - 通过散列函数将该名字映射到散列表单元
      - 从链中把链头记录删除
      - 重复，直到新链头在栈中的位置<块索引表顶端单元的值
    - 用块索引表顶端单元的值设置栈顶指针top。

6.3 类型检查

- 不同的观点
  - 强调最大程度的限制，要求执行严格的类型检查。
    - 如Ada语言要求显式声明类型，其编译程序严格执行类型检查，Ada被称为“强类型语言”。
  - 强调数据类型应用的灵活性，建议采用隐式类型，翻译时无需进行类型检查。
    - 如Scheme语言是隐式类型语言，它的编译程序不进行类型检查。
    - Scheme中的每一个数据值都有一个类型，在程序运行期间，系统将对每一个值的类型进行扩展检查。

类型检查（续）

- 强类型语言的编译程序不可能产生包含导致类型错误的不安全程序
  - 类型规则的严格性，确保了大多数不安全的程序在编译阶段被检出
  - 在编译阶段没有被检出的不安全程序将在数据被损害之前给出一个执行错误
  - 有些安全的程序也可能被检出类型错误
  - 给程序员带来了额外的负担
- Pascal语言不如Ada要求严格，但通常也被认为是强类型的语言。
- C语言有较多漏洞，有时被称为是弱类型语言。
- C++企图去除C的一些最严重的类型漏洞，但由于兼容性原因仍然不是完全强类型的语言。
- 没有类型系统的语言通常被称为无类型语言或动态类型语言
  - 像Lisp、Scheme，以及大多数脚本语言等。
  - 意味着所有安全检查都是在程序执行期间进行的。

类型检查（续1）

- 静态类型检查：由编译程序完成的检查
- 动态类型检查：目标程序运行时完成的检查
  - 如果目标代码把每个对象的类型和该对象的值一起保存，那么任何检查都可以动态完成。
- 一个健全的类型体制不需要动态检查类型错误。
- 如果一种语言的编译程序能够保证它所接受的程序不会有运行时的类型错误，则称这种语言是强类型语言。
- 有些检查只能动态完成，如：

```
char table[10];
int i;
…
table[i]:=9;
…
```

显式类型和静态类型检查

- 显式类型
  - 可提高程序的可读性，有助于理解程序中每一个数据结构的作用，及其所允许的操作。
  - 可以用类型信息去除运算符的重载。
- 静态类型信息
  - 编译程序更有效地进行存储分配、产生高效的机器代码。
  - 可提高编译效率。
  - 使用静态接口类型，通过证明接口一致性和正确性，可以提高大型程序的开发效率。
- 显式类型和静态类型检查相结合
  - 可尽早检出标准程序错误、减少可能出现的执行错误。
  - 编译时发现不正确的程序设计。
- 大多数现代程序设计语言都使用显式类型，由编译程序进行静态类型检查。

6.3.1 类型表达式

- 设计类型检查程序时要考虑的因素：
  - 语法结构：由上下文无关文法描述
  - 数据类型：数据集合以及其上的操作集合
  - 类型体制：把类型指派给语法结构的规则
- Pascal、C语言报告中有关于类型的描述：
  - 如果算术运算符加、减和乘的两个运算对象都是整型，那么结果是整型。
  - 一元运算符&的结果是指向运算对象所代表的实体的指针，如果运算对象的类型是‘…’，结果类型就是指向‘…’的指针。
- 暗示的概念：
  - 每一个表达式有一个类型
  - 类型有结构

数据类型

- 使用类型声明的语言都有一套类型声明规则
  - 基本类型：对程序员来说没有内部结构的类型，如int,char
  - 类型构造器：由基本类型构造复杂类型的方法，构造的类型称为用户定义类型，如数组、结构（记录）等
- 例：数组的类型构造器

  Pascal语言的定义：a:array[1..10]of integer

  C语言的定义：int a[10]
- 由类型构造器创建的类型不能自动获得名字，需要由类型声明（定义）创建，名字不仅是对新的数据类型的命名，对于类型检查、递归类型的构造都具有重要作用

  Pascal语言的类型声明:

```
type Array_integer_ten=array[1..10]of integer;
a: Array_integer_ten;
```

  C语言的类型声明:

```
typedef int Array_integer_ten[10];
Array_integer_ten a;
```

类型表达式的递归定义

- 类型表达式：
  - 或者由一个基本类型组成，或者由类型构造器施于其他类型表达式组成
  - 基本类型和类型构造器都取决于具体语言
- 本章使用的类型表达式定义（定义6.1）：

1. 基本类型是类型表达式：boolean、char、integer和 real

   专用基本类型：

   type_error错误类型：标志类型检查期间的错误

   void回避类型：检查语句、语句序列等没有数据类型的语言结构
2. 类型表达式可以命名，类型名是类型表达式。
3. 类型构造器作用于类型表达式的结果仍是类型表达式

   1）数组：如果T是类型表达式，那么**array(I, T)**是一个类型表达式，表示一个数组类型， T为元素类型， I为下标集合，I通常是一个整数域。

   Pascal的例子：var a:array[1..10] of integer;

   与a数组相关的类型表达式为：array(1..10,integer)

   C语言的例子： int a[10];

   与a数组相关的类型表达式为：array(0..9,integer)

类型表达式的递归定义（续1）

2）笛卡儿乘积：如果T_1和T_2是类型表达式，那么它们的笛卡儿乘积$T_1 \times T_2$也是类型表达式，假定×是左结合的。

引入笛卡尔积主要是为了保证定义的完整性，它可以用于描述类型的列表或元组（例如，用于表示函数参数和结构的分量）

3）记录：记录类型是它的各域类型的笛卡儿乘积，把类型构造器 record 作用于由域名和与之相关的类型表达式组成的二元组，就形成记录的类型表达式

如Pascal语言中的申明：

```
type  row=record
            addr:integer;
            name:array[1..10] of char
          end;
var table:array[1..10] of row;
```

类型名row代表类型表达式：

record((addr×integer)×(name×array(1..10,char)))

变量table的类型表达式为：array(1..10,row)

类型表达式的递归定义（续2）

功能相同的申明在C语言中的表示：

```
typedef  struct{
              int addr;
              char name[10] ;
 }row;
row table[10] ;
```

类型名row代表类型表达式：

record((addr×integer)×(name×array(0..9,char)))

变量table的类型表达式为：array(0..9,row)

4) 指针：如果T是类型表达式，那么**pointer(T)**是表示“指向T类型对象的指针”的类型表达式。

Pascal的例子：var p:↑row;

与p相关的类型表达式为：pointer(row)

C的例子：int * p;

p的类型表达式为：pointer(integer)

类型表达式的递归定义（续3）

5） 函数：从定义域类型D到值域类型R的映射

函数类型由类型表达式 D→R 表示。

Pascal的内部函数mod的类型表达式：int×int→int

这里，“×”的优先级高于“→”，“→”是右结合的

用户定义的Pascal函数：function fun(a,b:char):↑integer;

函数fun的类型表达式：char×char→pointer(integer)

用户定义的C语言函数：int square(int x){return x*x}

函数square的类型表达式：integer→integer

函数g：

参数是把整数映射成整数的函数

返回结果是和参数类型相同的另一函数

g的类型表达式为：

(integer→integer)→(integer→integer)

4. 类型表达式可以包含变量（称为类型变量），变量的值是类型表达式。

6.3.2 类型等价

- 关键：

  精确地定义什么情况下两个类型表达式等价

- 问题：
  - 类型表达式可以命名，且这个名字可用于随后的类型表达式中
  - 名字代表它自己？代表另一个类型表达式？
  - 新的名字是一个类型表达式，这个表达式与名字所代表的类型表达式是否总是等价的？

1. 结构等价

- 如果类型表达式仅由类型构造器作用于基本类型组成，则两个类型表达式等价的自然概念是结构等价
- 结构等价
  - 两个类型表达式：

    要么是同样的基本类型

    要么是同样的构造器作用于结构等价的类型表达式。
- 两个类型表达式结构等价当且仅当它们完全相同

例如：

integer 仅等价于 integer

pointer(integer) 仅等价于 pointer(integer)

例：考虑如下Pascal声明

A: record i: integer f: real end; √	B: record i: integer f: real end; √	C: record f: real i: integer end;	D: record x: real y: integer end;

考虑如下C语言声明：

struct recA { int i; char c; } a; √	typedef struct { int i; char c; } recB; recB b; √	struct { int i; char c; } c; √

算法6.1 测试两个类型表达式结构等价的算法

输入：两个类型表达式s和t
输出：如果s和t结构等价，则返回1（true），否则返回0（false）
方法：

```
boolean  seqtest(texpr s, texpr t)
{
    if  (s和t是同样的基本类型) return 1;
    else  if (s==array(s1, s2))&&(t==array(t1, t2))
            return seqtest (s1, t1) && seqtest (s2, t2);
    else  if (s==s1×s2)&&(t==t1×t2)
            return seqtest (s1, t1) && seqtest (s2, t2);
    else  if (s==pointer(s1))&&(t==pointer(t1))
            return seqtest (s1, t1);
    else  if (s==s1→s2)&&(t==t1→t2)
            return seqtest (s1, t1) && seqtest (s2, t2);
    else  return 0;
}.
```

实际应用中结构等价概念的修改

- 如：

  int a[10];

  int b[20];

- a的类型表达式：array(0..9, integer)
  b的类型表达式：array(0..19, integer)
- a和b不等价。
- 当数组作为参数传递时，数组的界不作为类型的一部分
- 算法调整

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

  return seqtest(s_2,t_2);

类型表达式的内部表示

- 为提高测试效率，可以对类型表达式进行编码。
- 做法：
  - 对基本类型规定确定位数、确定位置的二进制编码；
  - 对类型构造器规定确定位数的二进制编码。
- 将类型表达式表示为一个二进制序列。
- 结构等价测试，比较二进制序列：
  - 不同，不等价；
  - 相同，用算法6.1作进一步测试。

编码方式实例

- D. M. Ritchie设计的C语言编译程序中采用，也应用于Johnson[1979]描述的C语言编译程序中。
- 基本类型采用4位二进制编码，类型构造器采用2位编码。

基本类型	编码
boolean	0000
char	0001
integer	0010
real	0011

类型构造器	编码
pointer	01
array	10
freturns	11

- 示例：

类型表达式	编码
integer	0000000010
pointer(integer)	0000010010
array(pointer(integer))	0010010010
freturns(array(pointer(integer)))	1110010010

2. 名字等价

- 多数语言(如Pascal、C等)允许用户定义类型名
- C类型定义和变量声明（6.1）

```
typedef struct {
        int age;
        char name[20];
} recA;
typedef recA *recP;
recP a;
recP b;
recA *c, *d;
recA *e;
```

变量	类型表达式
a	recP
b	recP
c	pointer(recA)
d	pointer(recA)
e	pointer(recA)

C语言报告中没有定义“类型等价”这个术语

- 问题：a、b、c、d、e 是否具有相同的类型?
- 关键：类型表达式 recP 和 pointer(recA) 是否等价？
- 回答：依赖于具体的实现系统

名字等价（续2）

- 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，当且仅当它们名字完全相同。
- 所有的名字被替换后，两个类型表达式成为结构等价的类型表达式，那么这两个表达式结构等价。
- 名字等价的两个类型表达式也是结构等价的
- 变量a、b、c、d、e的类型表达式：

变量	类型表达式
a	recP
b	recP
c	pointer(recA)
d	pointer(recA)
e	pointer(recA)

结构等价

a、b：名字等价

c、d、e：名字等价

名字不等价

类型等价实例：C语言

- 使用介于名字等价和结构等价之间的一种类型等价形式
  - 对于struct和union采用名字等价，其他采用结构等价。
- 有如下C语言声明：

```
struct {
    short j;
    char c;
} x, y;
struct {
    short j;
    char c;
}b;
```

x、y 名字等价
x、y 与 b 名字不等价

Pascal语言采用名字等价的规则，
几乎所有类型构造器（如记录、数组、指针等）的每次应用，都将产生一个新的内部名字，建立一个新的名字不等价类型。

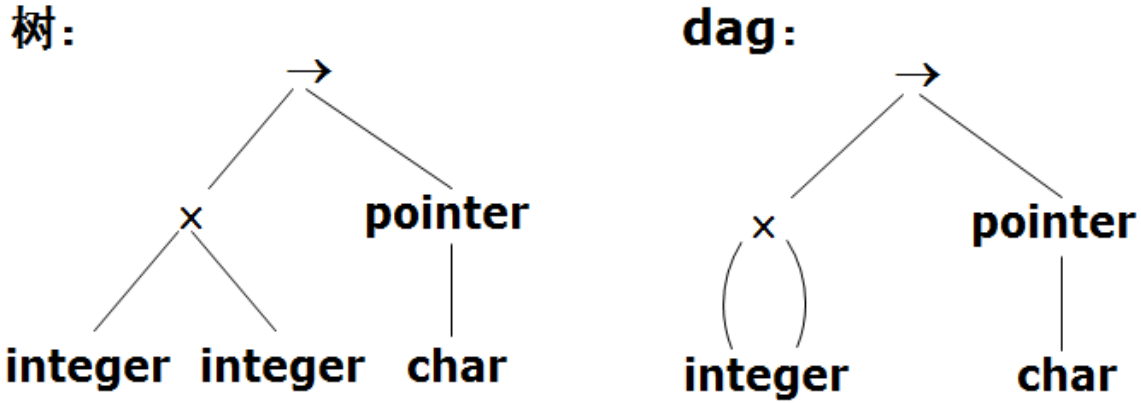
与声明6.1等效的声明6.2：

```
typedef struct {
    int age;
    char name[20];
} recA;
typedef recA  *recP;
typedef recA  *recD;
typedef recA  *recE;
recP a;
recP b;
recD c, d;
recE e;
```

- 名字等价
  - a 和 b 具有等价的类型
  - c 和 d 具有等价的类型
  - a、c 和 e 具有不同的类型
- 结构等价和名字等价这两个概念，可以用来揭示各种语言中通过说明使类型和标识符相关的规则
- C语言采用结构等价，Pascal采用名字等价

类型表达式的图形表示：类型图

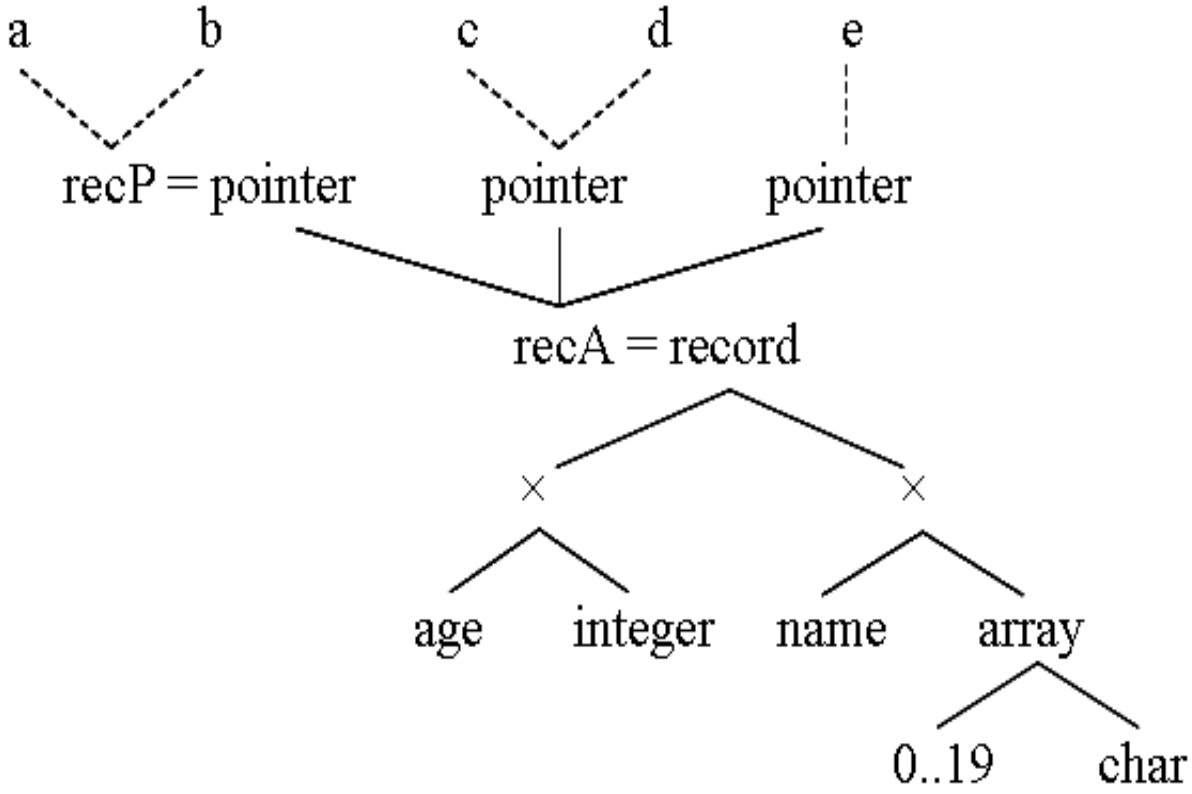
- 利用语法制导翻译技术为类型表达式构造树或dag
  - 内部结点：类型构造器
  - 叶结点：基本类型、类型名、或类型变量
- 构造方法：
  - 每当出现一个类型构造器或基本类型，就建立一个新的结点；
  - 每当出现一个新的类型名时，就建立一个叶结点；
  - 跟踪该名字所代表的类型表达式。
- 如：integer×integer →pointer(char)



类型表达式的图形表示：类型图（续）

- 在类型图中，如果两个类型表达式用相同的结点表示，则它们是名字等价的。

```
typedef struct {
    int age;
    char name[20];
} recA;
typedef recA  *recP;
typedef recA  *recD;
typedef recA  *recE;
recP a;
recP b;
recD c, d;
recE e;
```

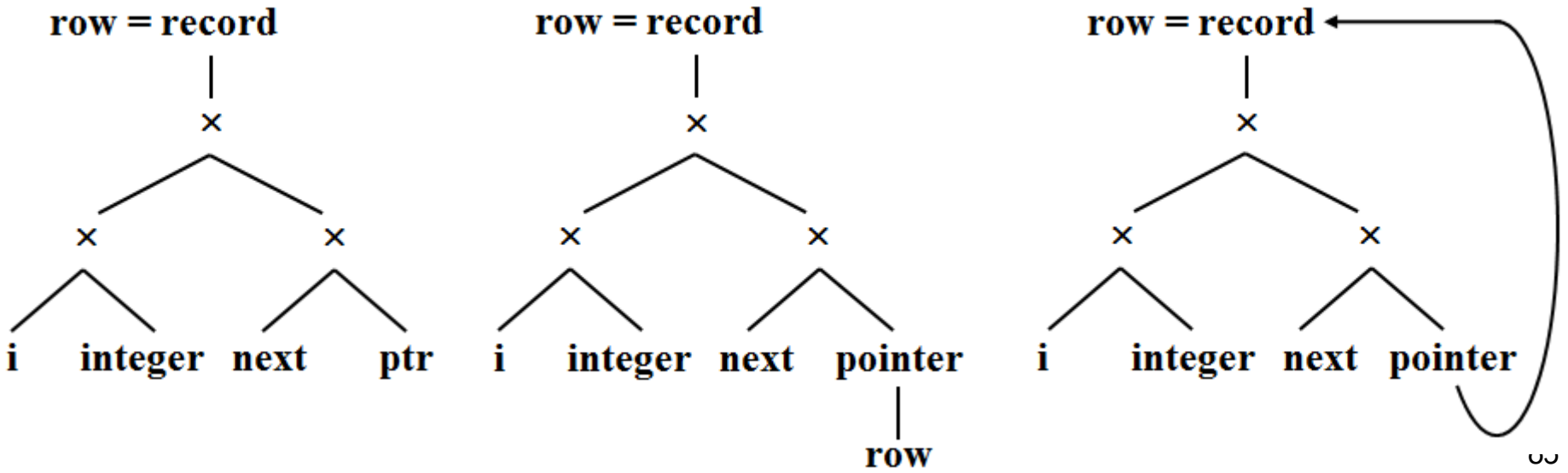


类型表示中的环

- 常用的动态数据结构，如链表、树等，递归定义。
- 结点用记录表示，记录中含有指向其他结点的指针，如：

```
type  ptr=↑row;
        row=record
                i: integer;
                next: ptr
              end;
```

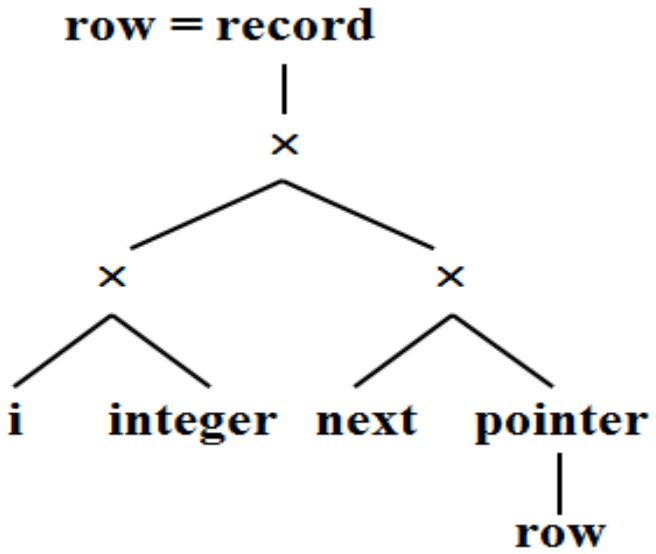
- row的类型表达式对应的类型图：



C语言实例

- 对除strucr和union以外的其他类型使用结构等价，避免在类型图中出现环。

```
struct row{
    int i;
    struct row * ptr;
    };
```



- 要求类型名在使用之前定义，但允许指针指向尚未定义的结构类型。
- 所有潜在的环均是由指向结构的指针引起的。

C语言实例（续）

- 结构名是其类型的一部分，在测试结构等价时，当遇到类型构造器 struct 时，测试停止。
  - 结果被比较的类型或者由于它们有同样的命名结构类型而等价，或者不等价。
  - 这样就避免了类型等价性验证出现无限循环。
  - 如：

```
struct  cell_1 {
      int  info;
      struct  cell_1 * next;
   };
```

```
struct  cell_2 {
      int  info;
      struct  cell_2 * next;
   };
```

- cell_1 和 cell_2 结构等价
- 如果开始时就假定cell_1和cell_2是结构等价的，那么采用无环表示，就可很容易地得出cell_1和cell_2结构等价的结论。

6.4 一个简单的类型检查程序

- 类型体制：把类型表达式指派到程序各组成部分的一组规则。
- 类型体制由类型检查程序实现。
- 同一语言的不同编译程序实现的类型体制可能不同，如：
  - Pascal语言规定：数组的类型由数组的下标集合和数组元素的类型共同决定。
    - 在数组作为参数传递时，实参必须和形参具有完全相同的类型，即具有同样的下标集合和数组元素类型。
  - 实际上，许多Pascal语言的编译程序允许数组的下标集合没有指明，即只检查元素的类型。
    - 实现的类型体制不同于Pascal语言定义的体制。
- UNIX系统提供的静态代码分析程序lint是一个比大多数C语言编译程序更加严密的编译工具。
  - 可以对源程序进行更加广泛的错误分析，可以检查出一般的C语言编译程序查不出来的一些类型错误。
  - 因为lint所实现的类型体制更详细。

6.4.1 语言说明

- 假设语言文法如下：

$P \rightarrow D;S$

$D \rightarrow D;D \mid \text{id}:T \mid \text{proc id}(A);\ D;\ S \mid \text{func id}(A):\ T;\ D;\ S$

$T \rightarrow \text{char} \mid \text{integer} \mid \text{real} \mid \text{boolean}$
$\quad \mid \text{array [num] of } T \mid \uparrow T \mid \text{record } D \text{ end}$

$A \rightarrow \varepsilon \mid paramlist$

$paramlist \rightarrow \text{id}:T \mid paramlist, \text{id}:T$

$S \rightarrow \text{id}:=E \mid \text{if } E \text{ then } S \mid \text{while } E \text{ do } S \mid \text{id}(rparam) \mid S;S$

$E \rightarrow \text{literal} \mid \text{num} \mid \text{num.num} \mid \text{id}$
$\quad \mid E + E \mid E * E \mid -E \mid (E)$
$\quad \mid \text{id}<\text{id} \mid E \text{ and } E \mid E \text{ mod } E \mid \text{id}[E] \mid E\uparrow \mid \text{id}(rparam)$

$rparam \rightarrow \varepsilon \mid rplist$

$rplist \rightarrow rplist, E \mid E$

```
i: integer;
k: integer;
i:=7;
k:=k mod i
```

6.4.1 语言说明（续）

- 语言中的类型：
  - 基本类型：char、integer、real和boolean
    type_error和void
  - 构造类型：数组、指针和记录
- 说明：
  - 所有名字必须先声明后引用
  - 变量声明：每个声明语句声明一个名字
  - 过程和函数声明：过程和函数声明允许嵌套
  - 数组下标从1开始
    类型array[256] of char的类型表达式是：
    array(1..256,char)
  - 前缀算符↑用于建立指针类型
    ↑integer的类型表达式是pointer(integer)

6.4.2 符号表的建立

- 处理声明语句时，编译程序的任务：
  - 分离出每一个被声明的实体；
  - 尽可能多地将该实体的信息填入符号表，名字、类型、存储地址等。
- 声明语句的形式
  - 类型关键字的位置
    - 在标识符表的前面，如 int a,b;
    - 在标识符表的后面，如 a,b: integer;
  - 标识符表
    - 单一实体，如 Ada
    - 多个同类型的实体，如 Pascal、C、Java
    - 不同种类的实体，如 FORTRAN

1. 过程中声明语句的处理

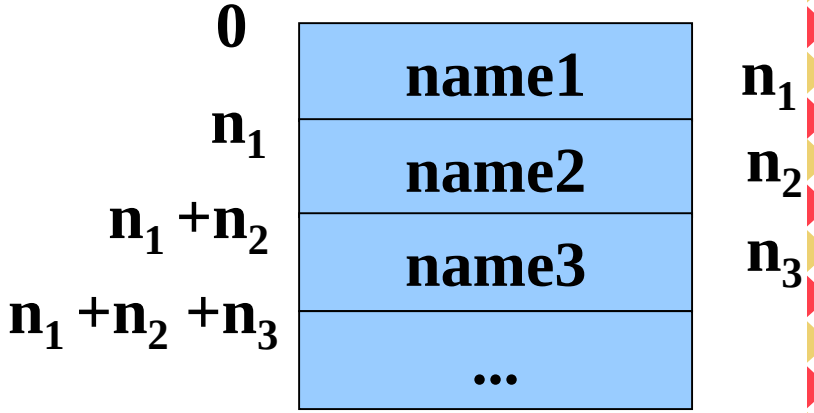
- 考虑：最内层过程
  - 暂不考虑记录结构的声明
  - 只涉及变量的声明
  - 变量作用域：该过程内。
- 声明语句的文法：
- 设计翻译方案的目的
  - 识别出被声明的实体
  - 记录实体的名字、类型、在数据区中的位置
- 引入：
  - 全局变量offset，记录相对地址
  - T.width：记录实体的域宽
    单位是字节
  - T.type：记录实体的类型
  - enter(id.name，T.type，offset)

$P \rightarrow D; S$

$D \rightarrow D; D$

$D \rightarrow id: T$

$T \rightarrow char \mid integer \mid real \mid boolean$
$\mid array\ [num]\ of\ T_1 \mid \uparrow T_1$



翻译方案 6.1

{ offset=0 }

P→ D;S

D→D;D

P → MD;S

M →ε { offset=0 }

练习：

i: integer;

x: real;

A: array[10] of char

D→id:T { enter(id.name, T.type, offset); offset=offset+T.width }

T→char { T.type=char; T.width=1 }

T→integer { T.type=integer; T.width=4 }

T→real { T.type=real; T.width=8 }

T→boolean { T.type=boolean; T.width=1 }

T→array [num] of T_1 { T.type=array(num.val, T_1.type); T.width=num.val×T_1.width }

T→↑T_1 { T.type=pointer(T_1.type); T.width=4 }

2. 过程定义的处理

作用域？

- 作用域信息的保存
- 文法产生式：

P→ M D;N S

D→ D；D

D→ id：T

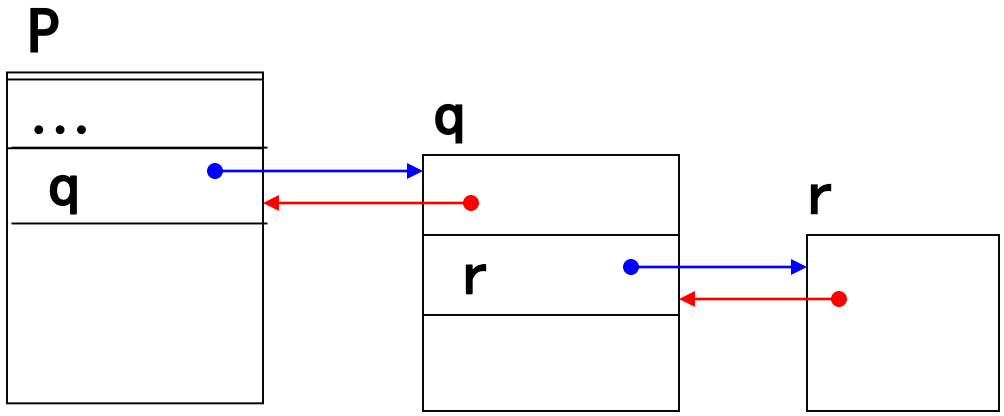
D→ proc id(A); M D;S

D→ fun id(A):T; M D;S

A→ε | paramlist

paramlist→id:T

| paramlist, id: T



创建主程序的符号表、初始化

记录主程序中声明的变量所需要的空间

重定位操作：记录子过程中声明的局部变量所需要的空间，返回到外围过程

定位操作：为子过程id创建符号子表，并初始化

过程定义的处理（续）

函数/过程参数？

- 作用域信息的保存
- 文法产生式：

P→ D; S

D→ D； D

D→ id： T

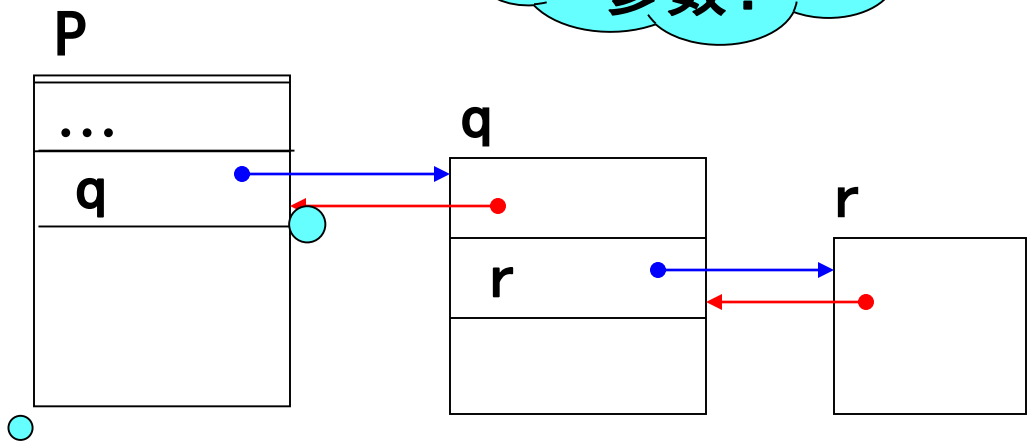
D→ proc id(A); D; S

D→ fun id(A):T;D;S

A→ε | paramlist

paramlist→id:T

| paramlist, id: T



创建主程序的符号表、初始化

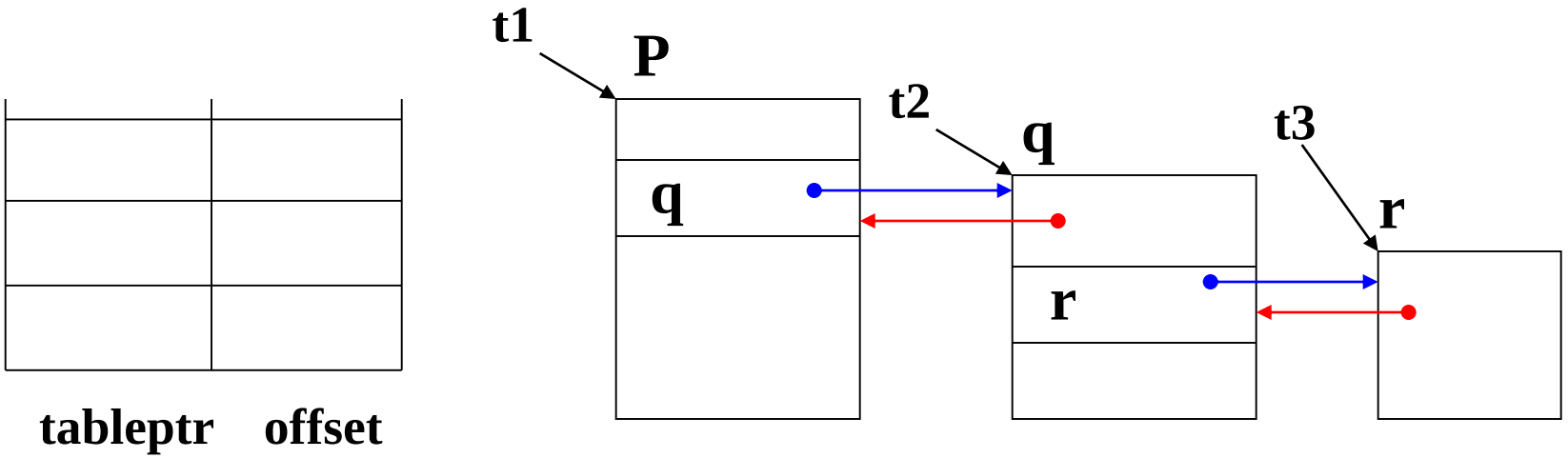
记录主程序中声明的变量所需要的空间

重定位操作：记录子过程中声明的局部变量所需要的空间，返回到外围过程

定位操作：为子过程id创建符号子表，并初始化

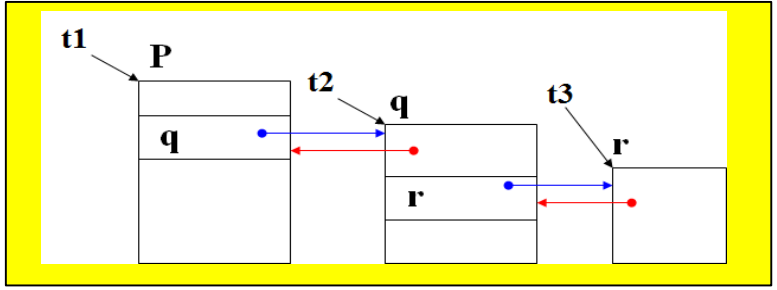
数据结构及过程

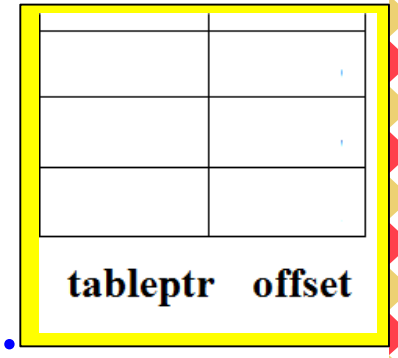
- 记录符号表嵌套关系的、便于操作的结构：栈



- 过程：
  - maketable(previous)
  - enter(table, name, type, offset)
  - addtheader(table, num, pwth, type, width)
  - addwidth(table, width)
  - enterproc(table, name, type, newtable)

翻译方案6.2





P→ M D；S { addwidth(top(tableptr), top(offset));
pop(tableptr); pop(offset); }

M→ε { t= maketable(nil); push(t, tableptr); push(0, offset); }

D→ D；D

D→ id：T { enter(top(tableptr), id.name, T.type, top(offset));
top(offset)=top(offset)+T.width; }

D→ proc id N(A); D;S {
t=top(tableptr);
addtheader(t, A.num, A.pwth, void, top(offset));
pop(tableptr); pop(offset);
enterproc(top(tableptr), id.name, proc, t); }

N→ε { t=maketable(top(tableptr));
push(t, tableptr); push(0, offset); }

翻译方案6.2(续)

D→ fun id N (A):T; D; S {
t=top(tableptr);
addtheader(t, A.num, A.pwth, T.type, top(offset));
pop(tableptr); pop(offset);
enterproc(top(tableptr), id.name, fun, t); }

A→ε { A.num=0; A.pwth=0; }

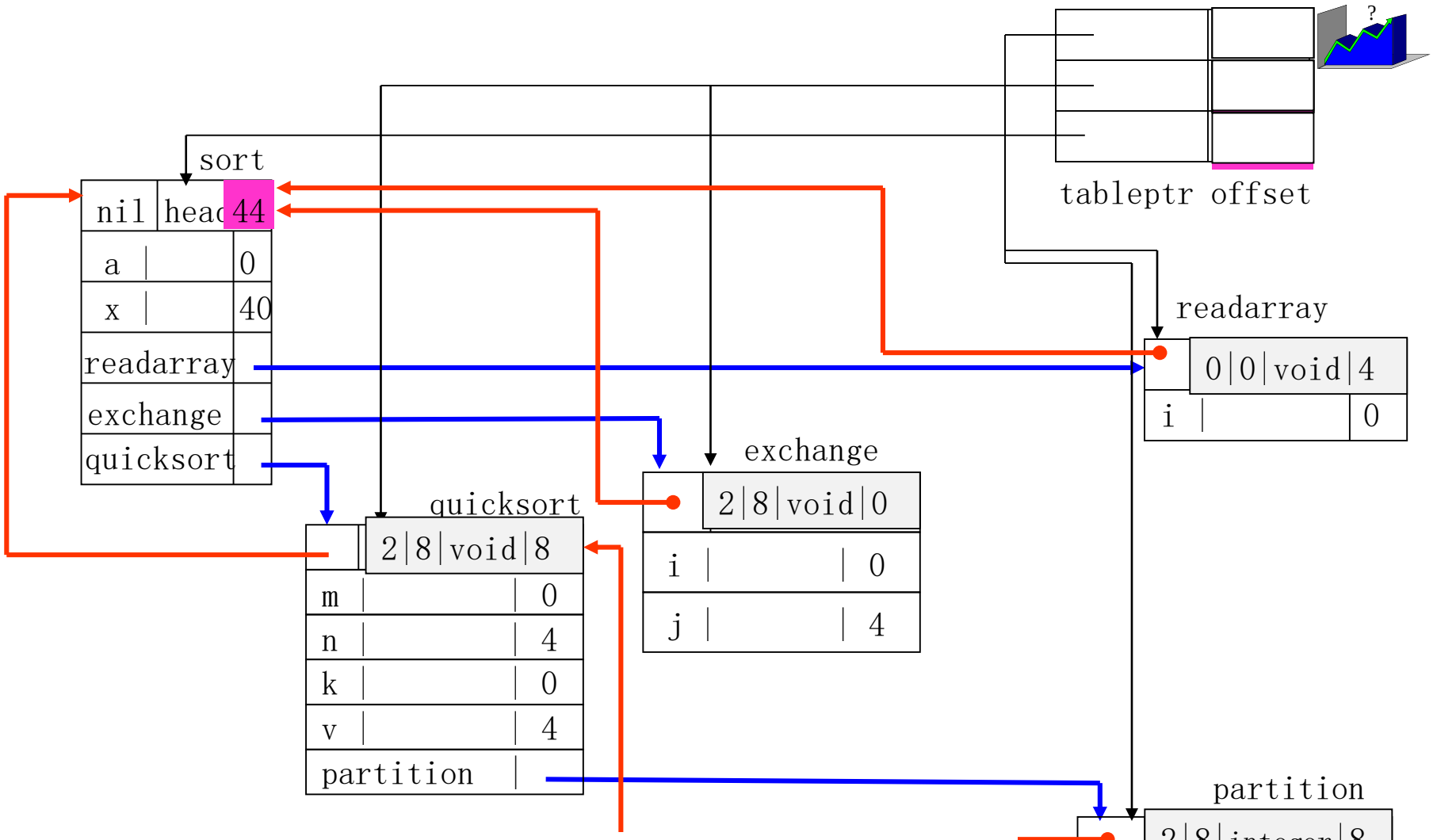
A→paramlist { A.num=paramlist.num; A.pwth=paramlist.pwth; }

paramlist→id:T { enter(top(tableptr), id.name, T.type, 0);
paramlist.num=1; paramlist.pwth=T.width; }

paramlist→ $paramlist_1$, id:T {
enter(top(tableptr), id.name, T.type, $paramlist_1$.pwth);
paramlist.num=$paramlist_1$.num+1;
paramlist.pwth=$paramlist_1$.pwth+T.width; }

例子

```
a:array[10] of integer;
x:integer;
proc readarray();
  i:integer;
  begin  …a …end;
proc exchange (i:integer, j:integer );
  begin …a … end;
proc quicksort (m:integer, n:integer );
  k:integer ;
  v:integer ;
  fun partition(y:integer , z:integer ):integer;
     i:integer ;
     j:integer ;
     begin
       …exchange(i, j); …
     end;
  begin  …  end
begin  readarray; quicksort(1, 9)  end
```



A→ paramlist

{A.num=paramlist.num; A.pwth=paramlist.pwth; }

3. 记录声明的处理

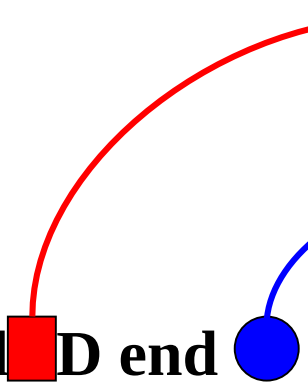
- 文法

  P→ D；S

  D→ D；D

  D→ id：T

  T→ record D end



定位操作：
创建符号表
保存记录中各域的信息

重定位操作：
将各域的总域宽作为
该记录型的域宽，
返回记录声明所在过程
的符号表

翻译方案6.3

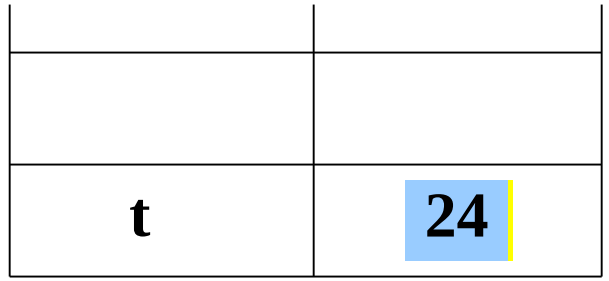
T→ record L D end { T.type=record(top(tableptr));
T.width=top(offset);
pop(tableptr); pop(offset) }

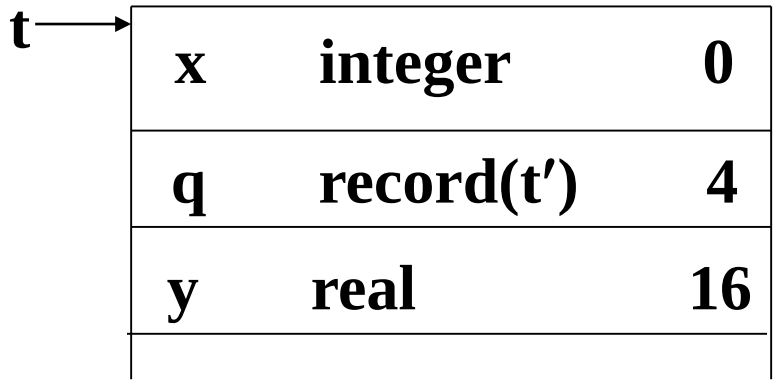
L→ε { t=mktable(nil);
push(t, tableptr); push(0, offset) }

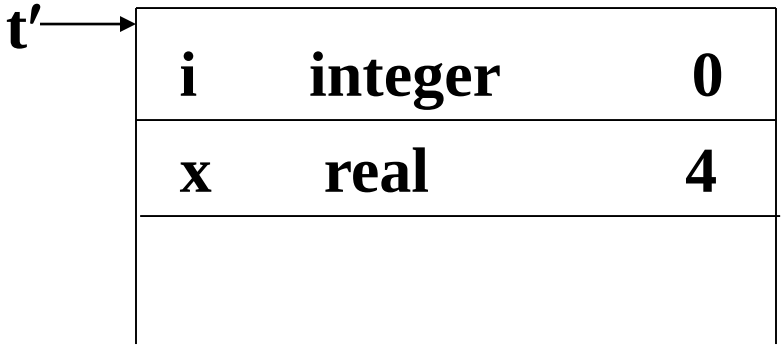
示例：

- 声明

```
x : integer;
q : record
      i: integer;
      x: real
    end;
y: real;
```







T.type=record(t′)
T.width=12

6.4.3 表达式的类型检查

综合属性E.type，类型体制指派给E产生的表达式的类型表达式

E→literal { E.type=char }

E→num { E.type=integer }

E→num.num { E.type=real }

E→id { E.type=lookup(id.entry) }

E→id_1<id_2 { if (lookup(id_1.entry)==char) &&
(lookup(id_2.entry)==char) ||
(lookup(id_1.entry)==integer) &&
(lookup(id_2.entry)==integer) ||
(lookup(id_1.entry)==real) &&
(lookup(id_2.entry)==real)
E.type=boolean;
else E.type=type_error; }

表达式的类型检查(续1)

$E \rightarrow E_1 + E_2$ { if ((E_1.type==integer)&&(E_2.type==integer)) E.type=integer;
else if ((E_1.type==real)&&(E_2.type==real)) E.type=real;
else if ((E_1.type==real)&&(E_2.type==integer)) E.type=real;
else if ((E_1.type==integer)&&(E_2.type==real)) E.type=real;
else E.type=type_error; }

$E \rightarrow E_1 * E_2$ { if ((E_1.type==integer)&&(E_2.type==integer)) E.type=integer;
else if ((E_1.type==real)&&(E_2.type==real)) E.type=real;
else if ((E_1.type==real)&&(E_2.type==integer)) E.type=real;
else if ((E_1.type==integer)&&(E_2.type==real)) E.type=real;
else E.type=type_error; }

$E \rightarrow -E_1$ { E.type= E_1.type; }

$E \rightarrow (E_1)$ { E.type= E_1.type; }

表达式的类型检查(续2)

$E \rightarrow E_1$ and E_2 { if (E_1.type==boolean) && (E_2.type==boolean)
E.type=boolean;
else E.type=type_error; }

$E \rightarrow E_1$ mod E_2 { if (E_1.type==integer) && (E_2.type==integer)
E.type=integer;
else E.type=type_error; }

$E \rightarrow$ id[E_1] { if ((gettype(id.entry)==array(s, t))
&& (E_1.type==integer))
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;}

表达式的类型检查(续3)

E→id(rparam) {
if ((gettype(id.entry)==fun)&&(rparam.num==0))
E.type=getrettype (id.entry);
else if ((gettype(id.entry)==fun) &&
(rparam.num==getpnum(id.entry)) &&
(checkptype(id.entry, rparam.type)))
E.type=getrettype (id.entry);
else E.type=type_error; }

rparam→ε { rparam.num=0; rparam.type=void; }

rparam→rplist { rparam.num=rplist.num;
rparam.type=rplist.type; }

rplist→E { rplist.num=1; rplist.type=E.type; }

rplist→rplist$_1$,E { rplist.num=rplist$_1$.num+1;
rplist.type=rplist$_1$.type×E.type; }

6.4.4 语句的类型检查

综合属性S.type，类型体制指派给语句的类型表达式。
语句中没有类型错误，则指派 void; 否则，指派type_error。

S→id:=E { if (gettype(id.entry) ==E.type) S.type=void;
else S.type=type_error; }

S→if E then S_1 { if (E.type==boolean) S.type=S_1.type;
else S.type=type_error; }

S→while E do S_1 { if (E.type==boolean) S.type=S_1.type;
else S.type=type_error; }

语句的类型检查（续）

S→id(rparam) {

if ((gettype(id.entry)==proc)&&(rparam.num==0))

S.type=void;

else if ((gettype(id.entry)==proc) &&

(rparam.num==getpnum(id.entry)) &&

(checkptype(id.entry, rparam.type)))

S.type=void;

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 ; }

P→D;S { if (S.type==void) P.type=void;

else P.type=type_error ; }

6.4.5 类型转换

- 不同类型的数据对象在计算机中的表示形式不同。
- 用于整型运算和实型运算的机器指令不同。
- 当不同类型的数据对象出现在同一表达式中时，编译程序必须首先对其中的一个操作数的类型进行转换，以保证在运算时两个操作数的类型是相同的。
- 语言定义指出什么转换是必需的
  - 赋值语句：把赋值号右边的对象转换成左边对象的类型
  - 表达式：把整数转换成实数，然后在这一对实数对象上进行实数运算
- 如果类型转换构建在类型体制中，由编译程序完成，这种类型转换是隐式的，称做强制转换。
  - 一般要求隐式转换原则上不丢失信息，如C语言。
  - 例如：int x=5;
    x=2.1+x/2;
  - 等价于：x=int(2.1+double(x/2))，结果：x=4
  - 优点：编程时无须考虑类型转换
  - 缺点：削弱类型检查，类型错误可能无法检出，导致不可预料的执行时错误。

类型转换（续）

- 如果类型转换必须由程序员显式地写在源程序中，则这种转换叫做显式转换。
- 显式类型转换的语法形式
  - 把希望的结果类型用括号括起来放在表达式之前，在C和Java中使用的形式。
    - 如C语言的语句：x=(int)(2.1+(double)(x/2));
  - 函数调用形式，即将表达式作为类型转换函数的参数。在Pascal、Ada、C++中使用的形式。
    - 如Pascal语言中：
      内部函数 ord 将字符转换为整数，如 ord('A')
      内部函数 chr 将整数转换为字符，如 chr(45)
  - 优点：很少产生不可预料的行为；
  - 缺点：编程时，写类型转换的代码。
- 折中方法：只允许保证不破坏数据的强制类型转换。
  - Java语言，只允许数学运算类型的宽隐式转换。

常数的隐式转换

- 常数的类型转换可以在编译时完成，常常可以使目标程序的运行时间大大改进。
- 如：float x;
  x=1 的执行时间比 x=1.0 的要长。
- 再如：double x[10]; int I;
  - for(I:=1; I<10; I++) X[I]:=1
  - for(I:=1; I<10; I++) X[I]:=1.0
  - 编译程序为第一个语句产生的目标代码含运行时的函数调用，该函数把1的整型表示转换为实型表示。
- 多数编译程序都在编译时完成常数的类型转换。

小　结

- 语义分析的概念
  - 编译的一个重要任务、检查语义的合法性
  - 符号表的建立和管理
  - 语义检查
- 符号表
  - 何时创建
  - 内容
  - 操作
    - 检索、插入
    - 定位、重定位
  - 组织形式
    - 非块结构语言
    - 块结构语言

小　结（续）

- 静态语义检查
  - 类型检查
  - 控制流合法性检查
  - 同名变量检查
  - 关联名字检查
  - 利用语法制导的翻译技术实现
- 类型体制
  - 语法结构、类型概念、把类型指派给语言结构的规则
  - 类型表达式的定义
  - 类型表达式的等价
    - 结构等价
    - 名字等价
    - 类型表达式结构等价的测试算法

小　结（续）

- 简单类型检查程序的说明
  - 声明语句的类型检查
  - 表达式的类型检查
  - 语句的类型检查
  - 类型转换