第6章　语义分析

重庆大学　葛亮

知识点：符号表

语义分析

6.1 语义分析概述

6.2 符号表

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

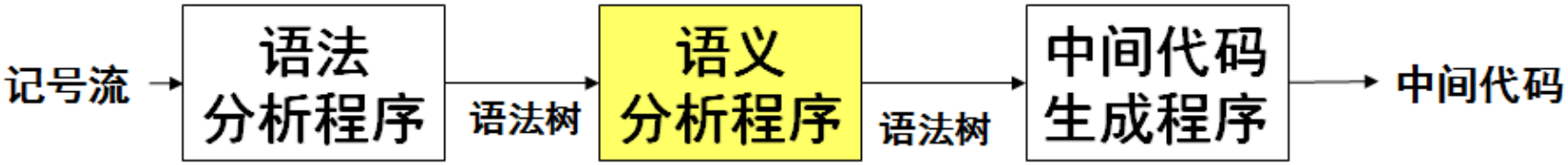
- 语义分析程序通过将变量的定义与变量的引用联系起来，对源程序的含义进行检查。
  检查每一个语法成分是否具有正确的语义。
- 语义分析的任务
  （1）收集并保存上下文有关的信息；
  （2）类型检查。
- 符号表的建立和管理
  - 在分析声明语句时，收集所声明标识符的有关信息，如类型、存储位置、作用域等，并记录在符号表中。
  - 只要编译时控制处于声明该标识符的程序块中，就可以从符号表中查到它的记录。

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：
    - 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

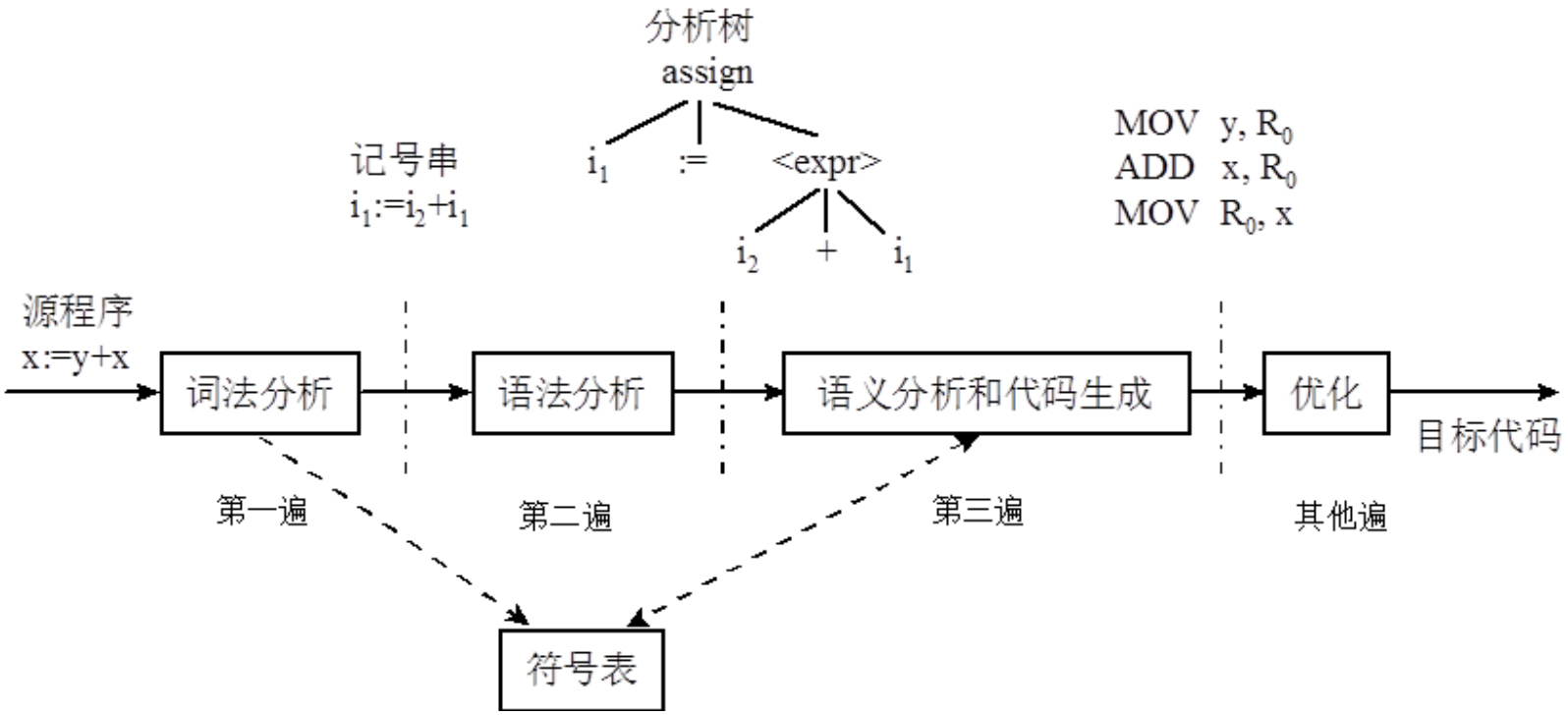
6.2.2 符号表内容

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

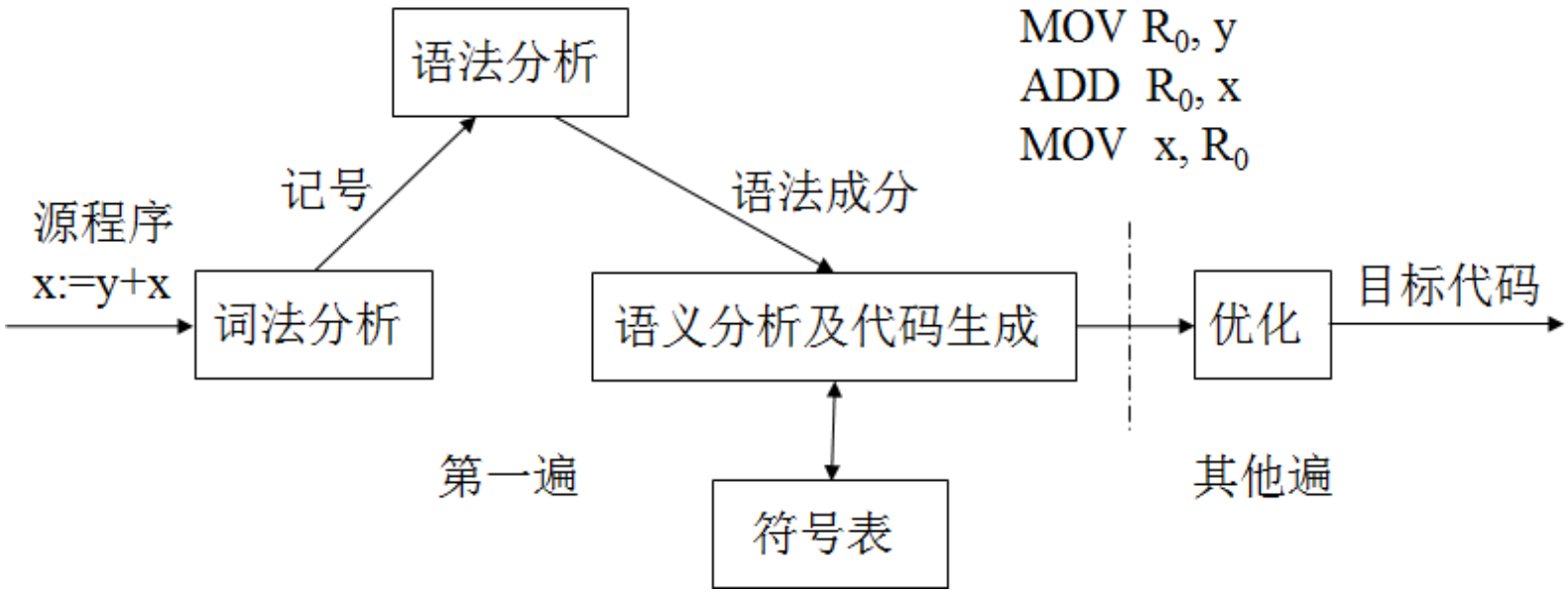
1. 多遍编译程序



- 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

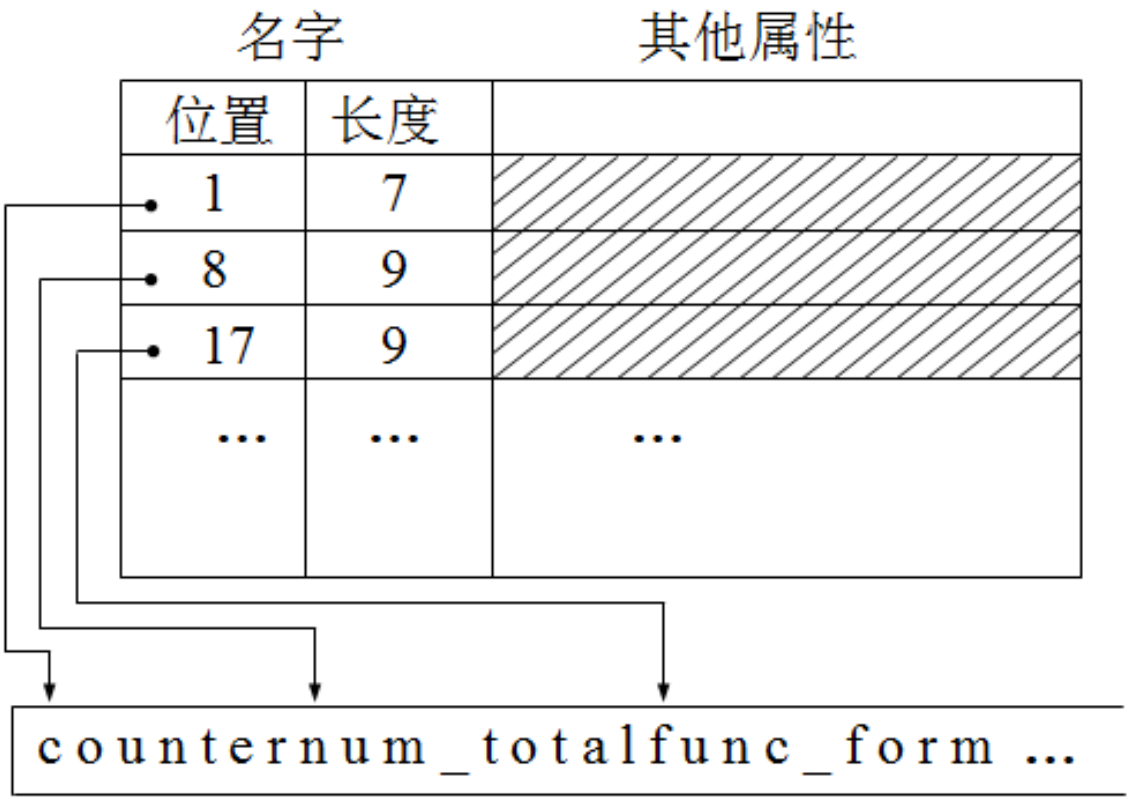
名字

- 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

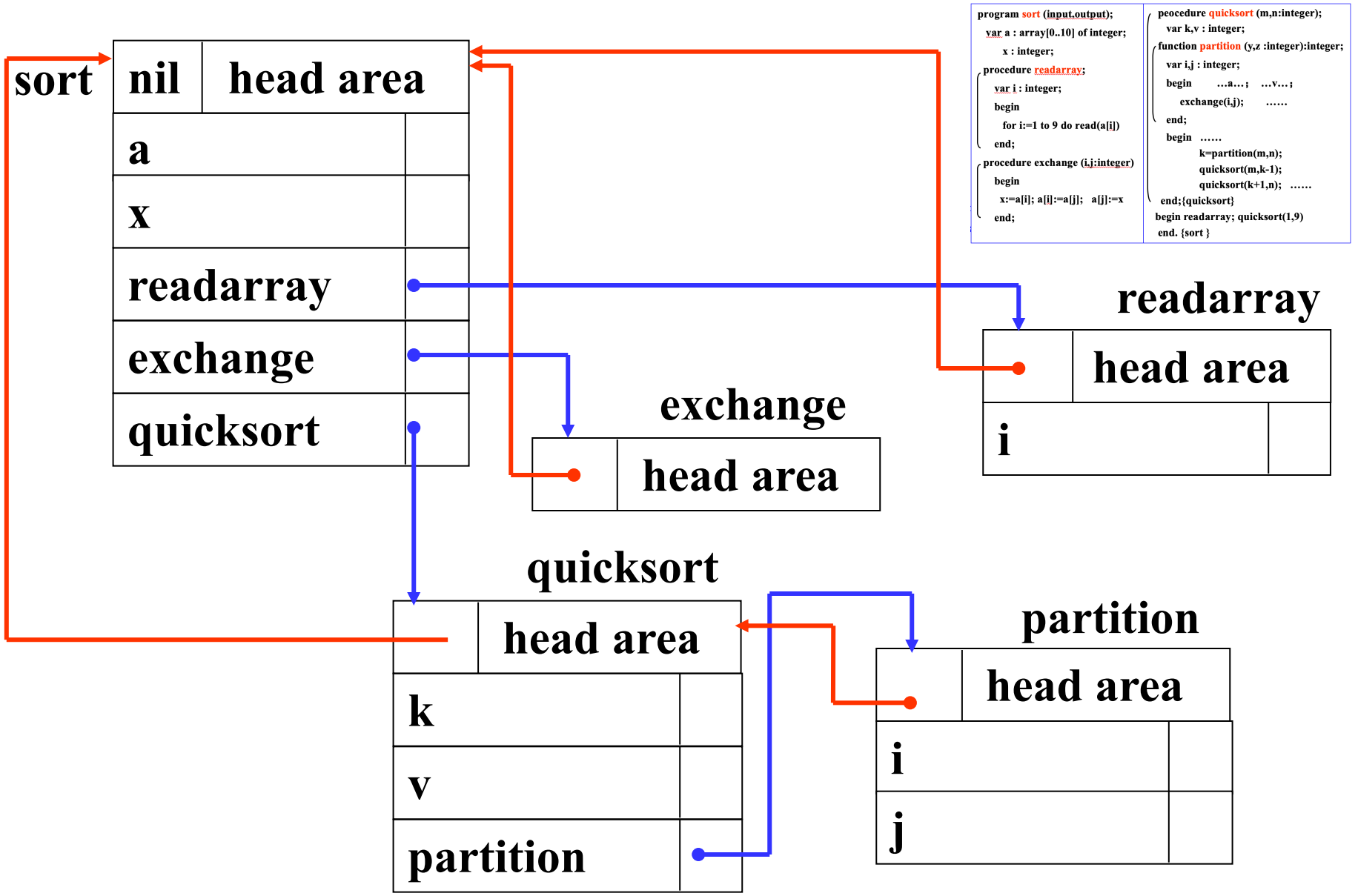
- 对于块结构的语言，在建立和删除符号表时还要使用两种附加的操作，即定位和重定位。
- 当编译程序识别出块开始时，执行定位操作。
- 当编译程序遇到块结束时，执行重定位操作。
- 定位操作：
  - 建立一个新的子表（包含于符号表中），在该块中声明的所有名字的属性都存放在此子表中。
- 重定位操作：
  - “删除”存放该块中局部名字的子表
  - 这些名字的作用域局部于该块，出了该块后不能再被引用。

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
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 }
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 }。

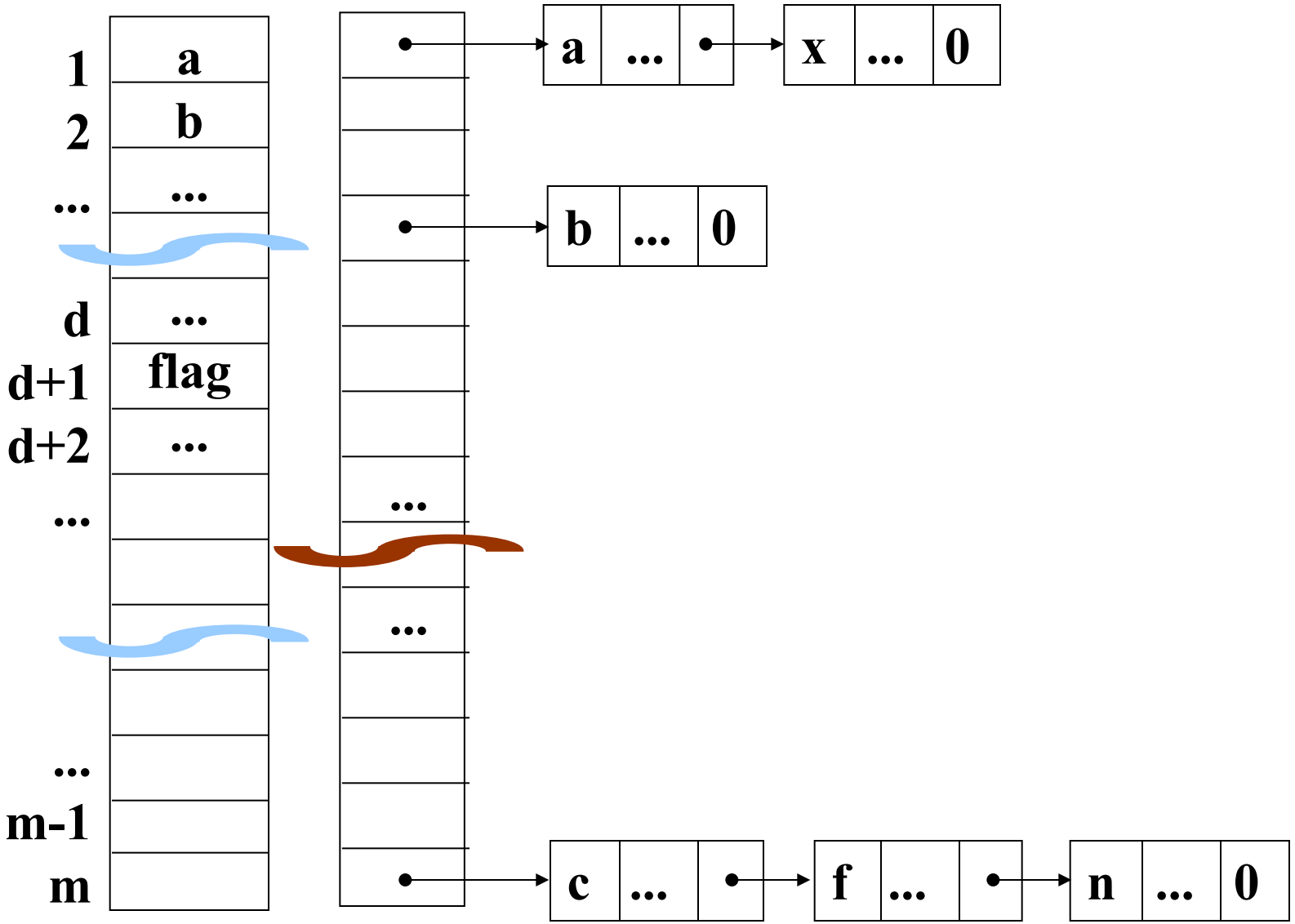
非块结构语言的符号表组织（续3）

- 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

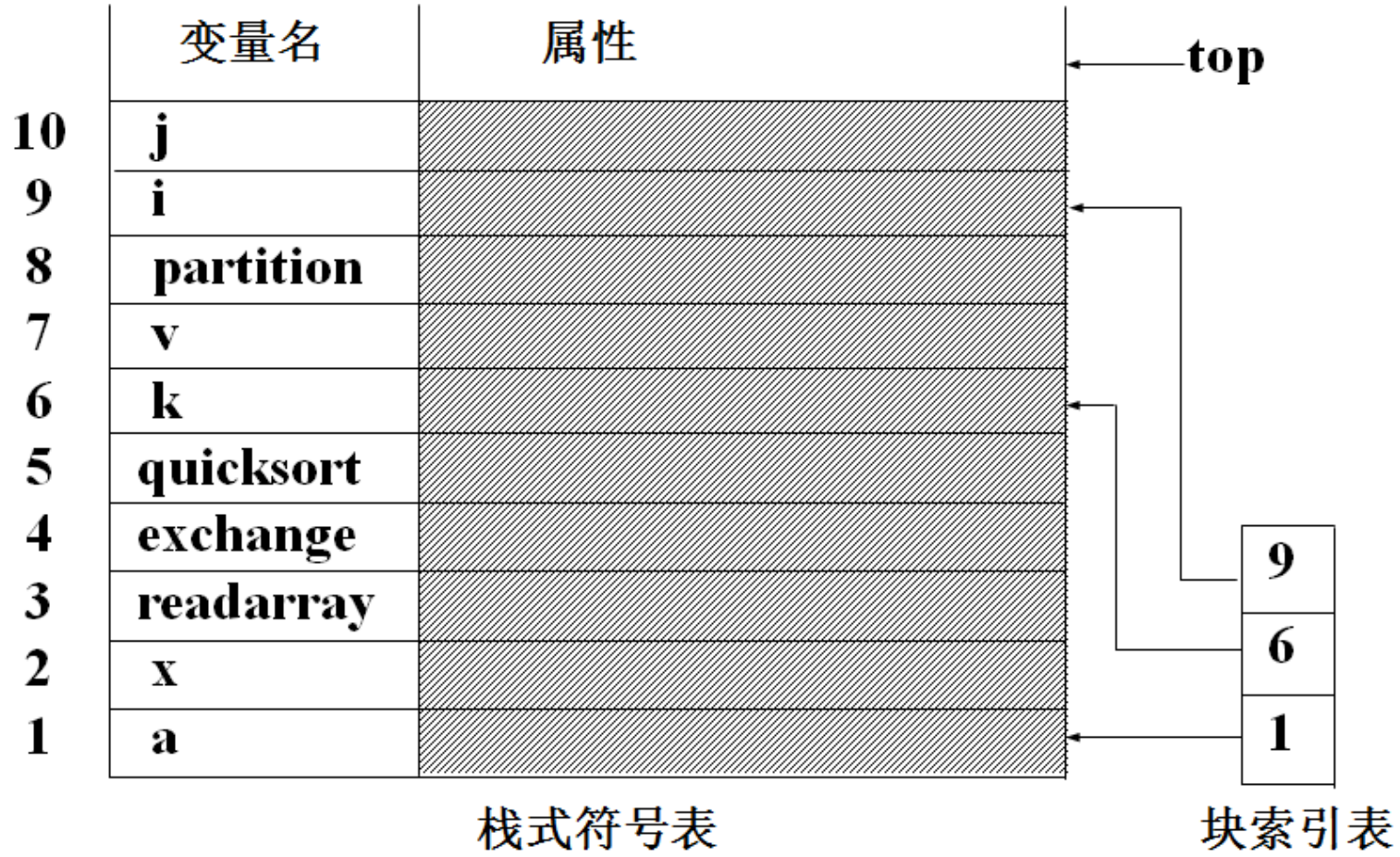


2. 块结构语言的符号表组织

- 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

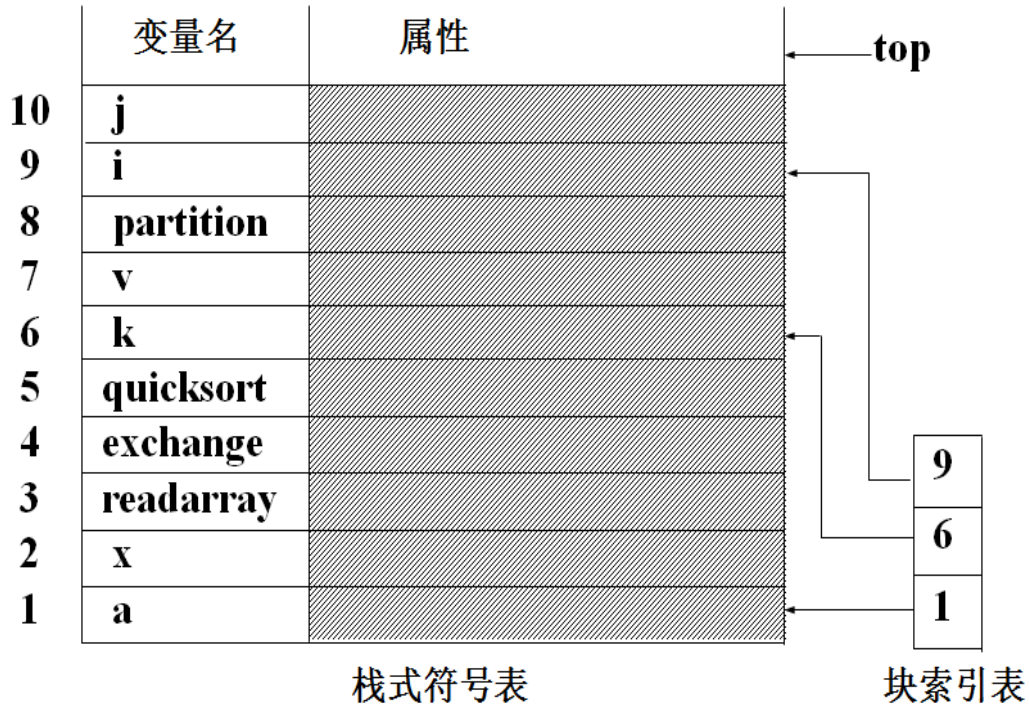
栈式符号表

- 当遇到变量声明时，将包含变量属性的记录入栈
- 当到达块结尾时，将该块中声明的所有变量的记录出栈



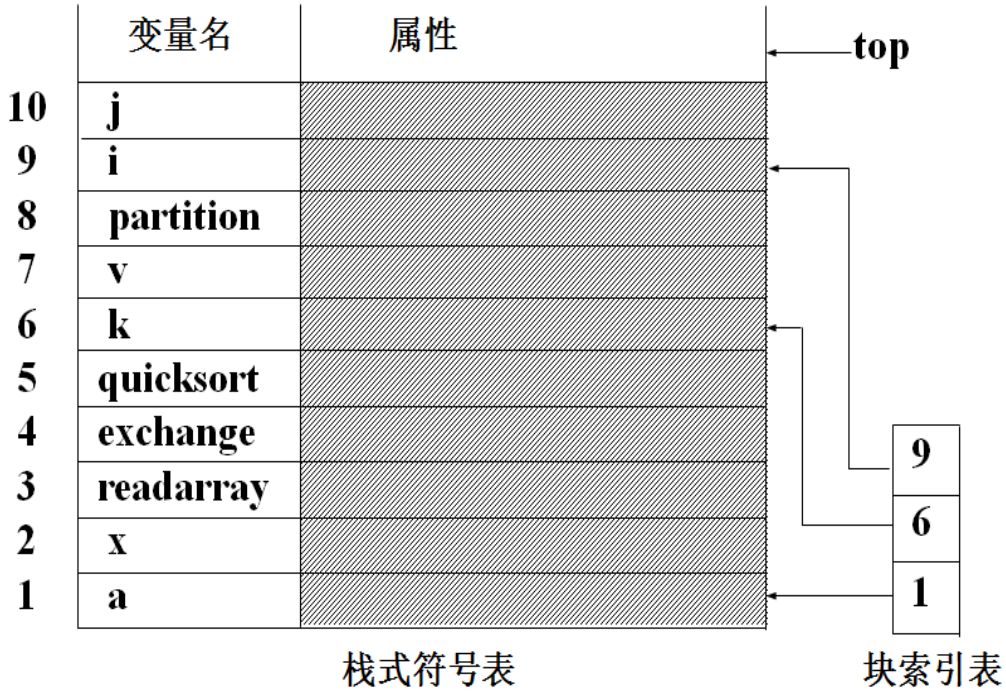
栈式符号表操作

- 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



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。
  - 有效地清除刚刚被编译完的块在栈式符号表中的所有记录。

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

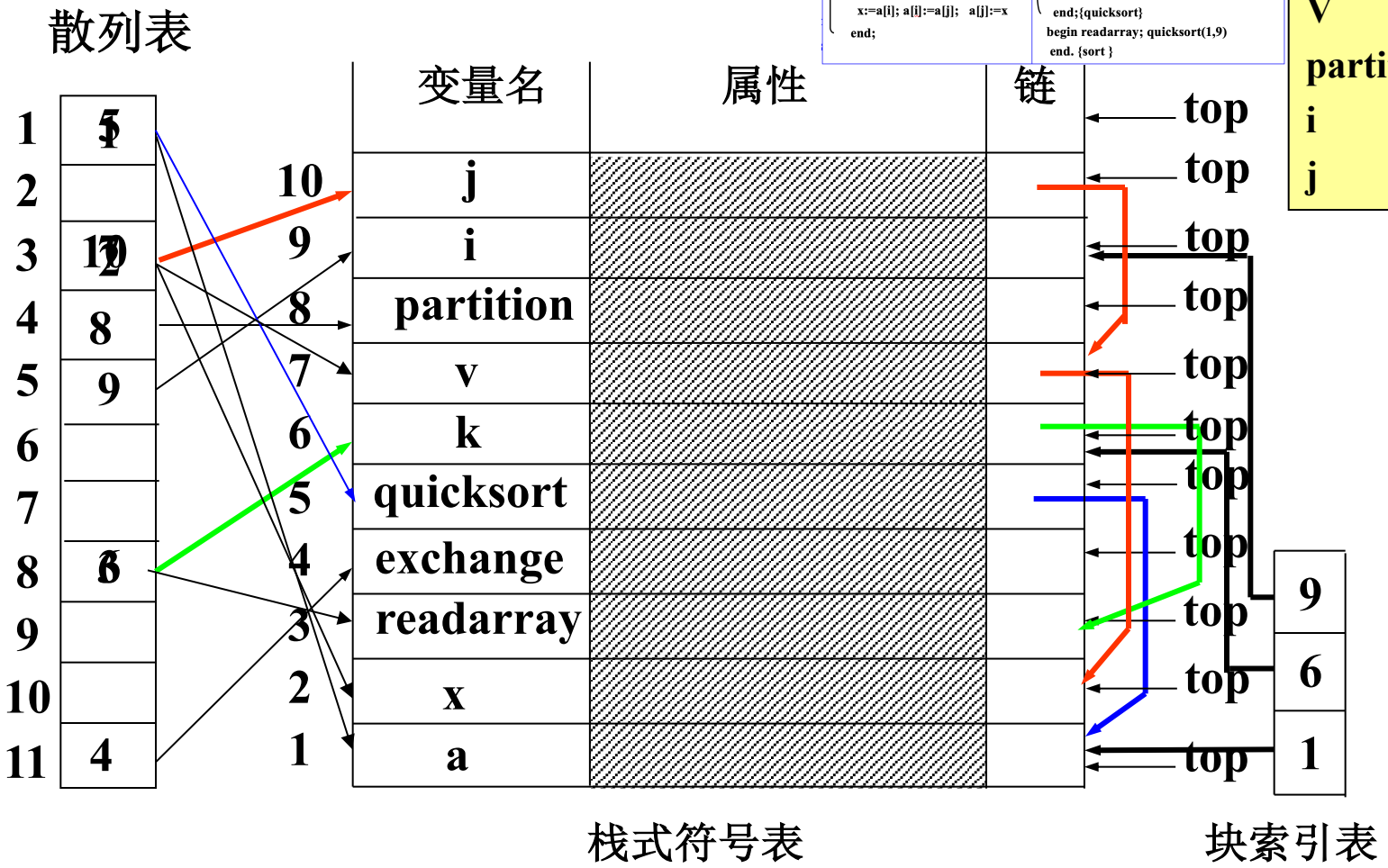
栈式散列符号表示意图

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

```
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 }
```

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。
    - 用块索引表顶端单元的值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。

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