编译原理与技术

第6章：语义分析

王吴凡

北京邮电大学计算机学院

主页：cswwf.github.io

邮箱：wufanwang@bupt.edu.cn

教学内容、目标与要求

- 教学内容
  - 语义分析程序的地位和作用
  - 符号表的组织与管理
  - 类型表达式及类型等价
  - 一个简单类型检查程序
- 教学目标与要求
  - 掌握块结构语言的符号表组织方式；
  - 掌握数据对象的类型表示及等价性。
  - 能够根据程序设计语言的语法结构和语义规则
    - 理解并分析对源程序进行语义分析的需求；
    - 掌握利用符号表检查名字的作用域；
    - 设计对源程序进行类型检查的语法制导翻译方案；

内容目录

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。
- 程序正确与否与结构的上下文有关，如：
  - 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
- 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？
- 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

```
① main() {
②   int i, j;
③   i=0;  j=1;
④   {
⑤     double i, k;
⑥     k=10;
⑦     j=k+m;
⑧   };
⑨   i=j*k;
⑩ }
```

1. 语义分析的任务

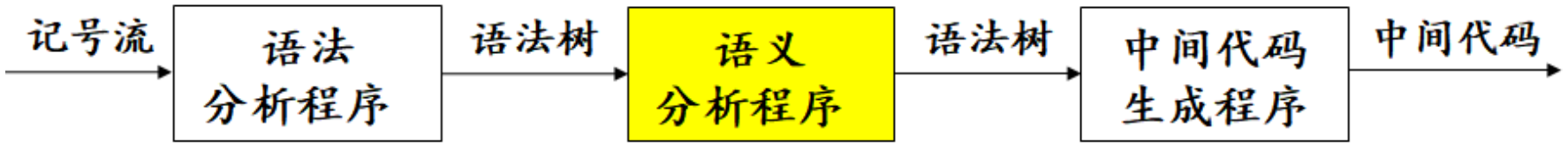
- 语义分析程序通过将变量的定义与变量的引用联系起来，对源程序的含义进行检查。
  检查每一个语法成分是否具有正确的语义。
- 语义分析的任务
  （1）收集并保存上下文有关的信息；
  （2）类型检查。
- 符号表
  - 在分析声明语句时，收集所声明标识符的有关信息，如类型、存储位置、作用域等，并记录在符号表中。
  - 只要编译时控制处于声明该标识符的程序块中，就可以从符号表中查到它的记录。

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声明一次
    - 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。

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：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：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

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。

```
①  main() {
②      int i, j;
③      float x;
④      i=0;  j=1;
⑤      x=2;
⑥      {
⑦          int i, k;
⑧          float k;
⑨          k=10;
⑩      };
⑪      i=j*k;
⑫      j=i+x;
⑬  }
```

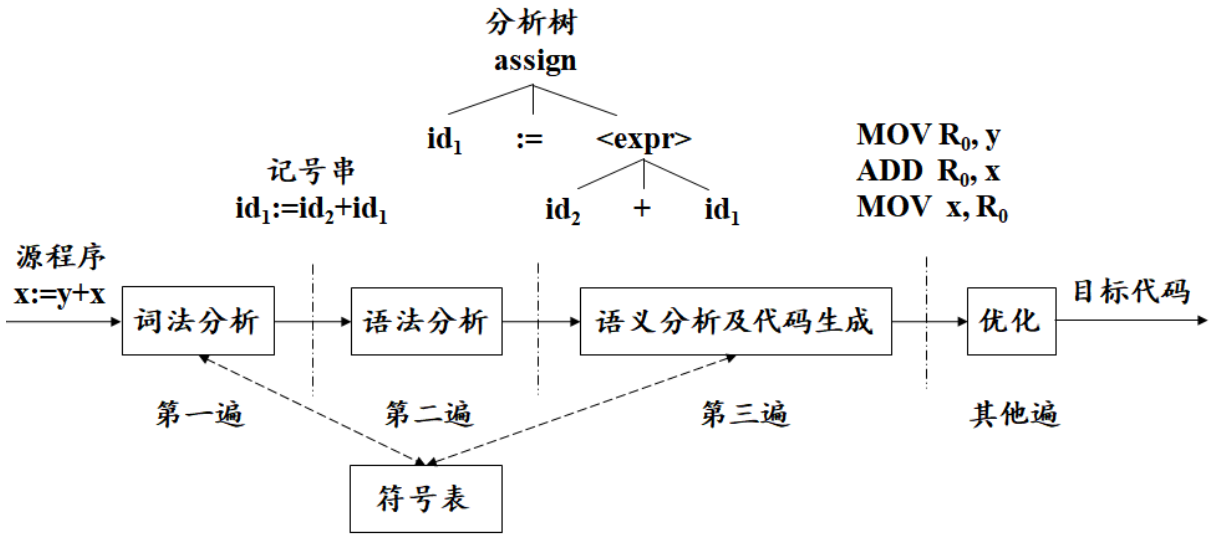
6.2 符号表

- 符号表的作用：
  - 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
- 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访问符号表，其效率直接影响编译程序的效率。

- 本节内容
  1. 符号表的建立和访问时机
  2. 符号表内容
  3. 符号表操作
  4. 符号表组织

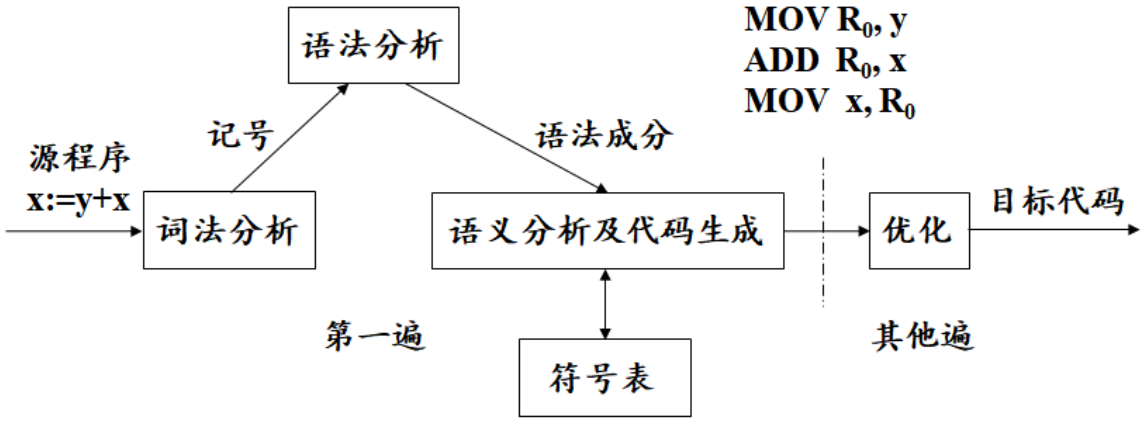
1. 符号表的建立和访问时机

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

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。
- 适用于块结构语言，如C、Pascal等

2. 符号表内容

- 记录标识符的相关属性
- 属性种类，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

3. 符号表操作

名字的作用域

- 基本操作：插入和检索
- 要求标识符显式声明的语言：
  - 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
  - 处理名字引用时，调用检索操作
    - 查找信息，进行语义分析、代码生成
    - 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，从其上下文推测出该变量的相关属性。
- 对于块结构语言，还需要两种操作，即定位和重定位。
  - 识别出块开始时，执行定位操作。
  - 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，在该块中声明的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。

4. 符号表组织　　① 非块结构语言的符号表组织

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

- 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
    - 按变量声明/出现的先后顺序填入表中
    - 适用于程序中变量很少的情况
  - 有序线性表
    - 按字母顺序对变量名排序的表
  - 哈希/散列表
    - 查找时间与表中记录数无关
    - 哈希函数

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
  - 栈式哈希符号表

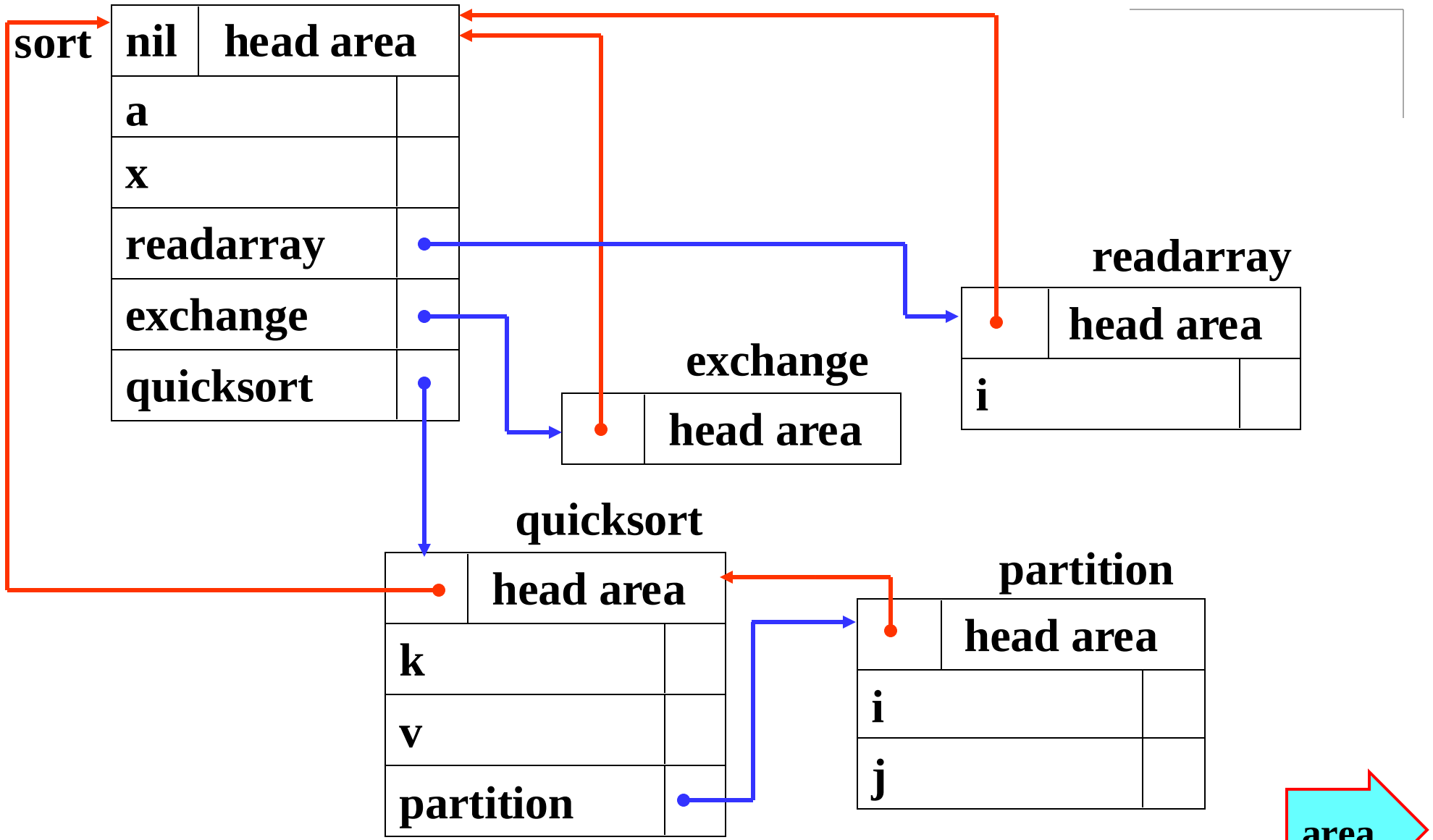
读入数据，并进行排序的PASCAL程序

```
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;
```

```
procedure 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 }
```

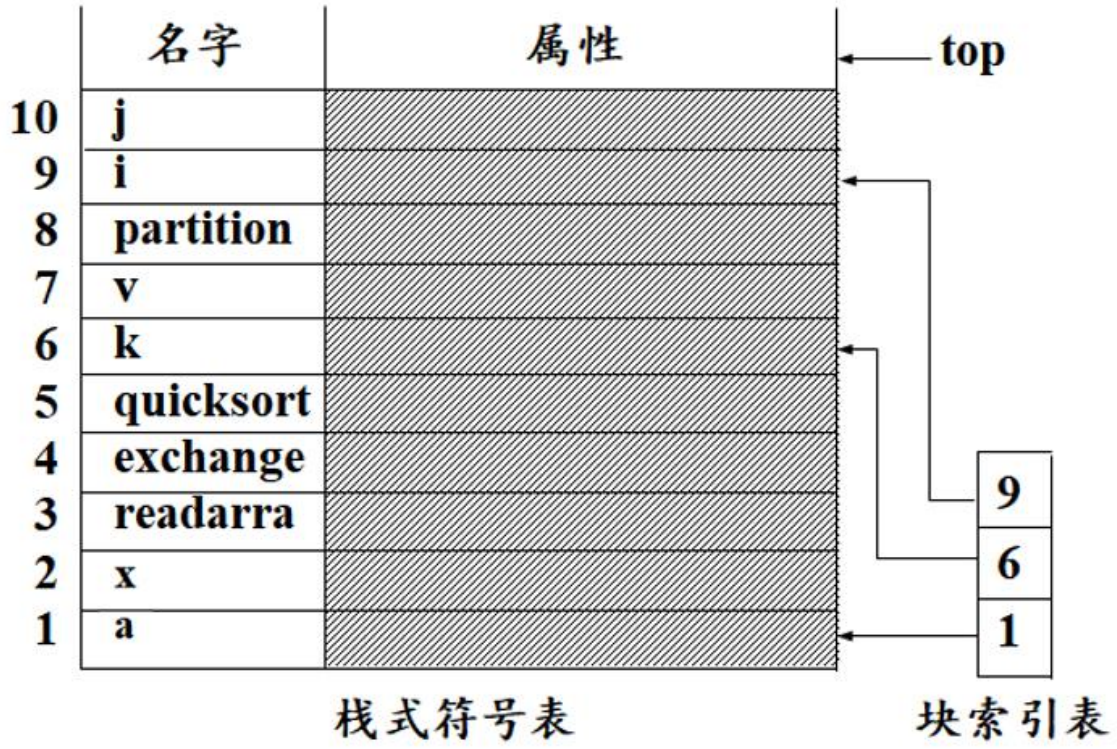
符号表的逻辑结构

```
sort:
    a
    x
    readarray:
        i
    exchange:
    quicksort:
        k
        v
        partation:
            i
            j
```



栈式符号表及操作

- 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。



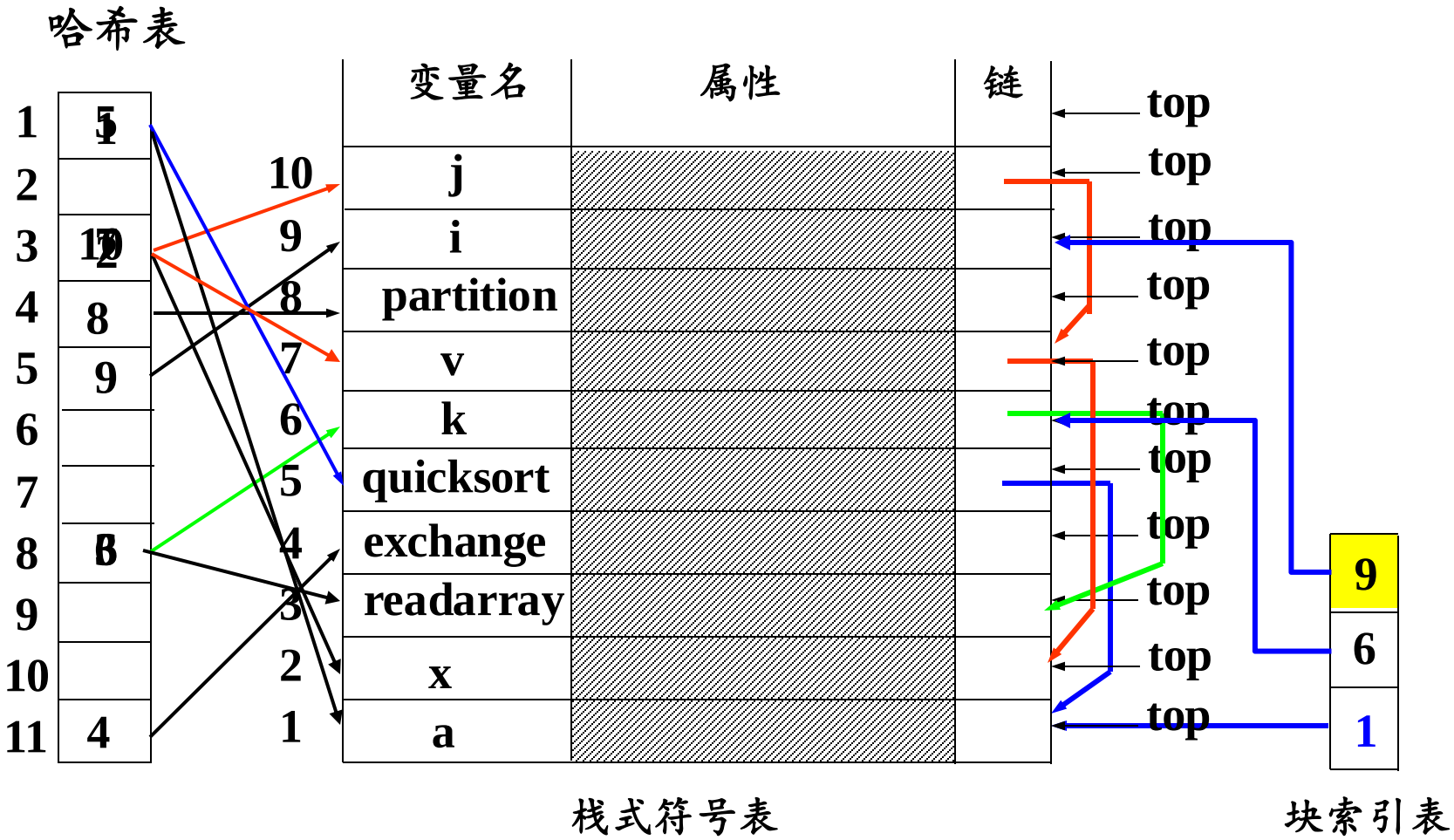
- **插入**
  - 检查子表中是否有重名变量
    - 无，新记录压入栈顶
    - 有，报告错误
- **检索**
  - 从栈顶到栈底线性检索
    - 在当前子表中找到，局部变量
    - 在其他子表中找到，非局部名字
  - 实现了最近嵌套作用域原则
- **定位**
  - 将栈顶指针top的值压入块索引表顶端。
  - 块索引表的元素是指针，指向相应块的子表中第一个记录在栈中的位置。
- **重定位**
  - 用块索引表顶端元素的值恢复栈顶指针top，完成重定位操作。
  - 有效地清除刚刚被编译完的块在栈式符号表中的所有记录。

栈式哈希符号表及操作

插入？
检索？
定位？
重定位？

- 假设哈希表的大小为11，哈希函数执行如下变换：

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



6.3 类型检查

- 不同的观点
  - 强调最大程度限制，严格类型检查。
    - 确保大多数不安全的程序在编译阶段被检出。
    - Ada语言，要求显式声明类型；
    - 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，隐式类型语言，其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。
- Pascal，被认为强类型的语言
  - 给程序员带来了额外的负担
- C，有时被称为弱类型语言
- 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。
  - 所有安全检查都是在程序执行期间进行的

类型检查

- 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。
  - 去除运算符的重载。
- 静态类型信息
  - 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
  - 数据类型
  - 类型体制，即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。
- 隐含的概念：
  - 每一个表达式有一个类型
  - 类型有结构

数据类型

- 基本类型，没有内部结构的类型，如integer、char、boolean等。
- 类型构造器，即类型运算符。
  - 利用类型构造器可以由基本类型构造新的有结构的类型，称为构造类型。
  - 可以由具有简单结构的构造类型构造新的具有更复杂结构的构造类型。

    如：

    a: array[1..10] of integer; // Pascal

    int a[10]; // C
- 新构造类型的名字由类型声明创建。如：

  typedef int array_integer_ten[10];

  array_integer_ten a;

类型体制

- 把类型表达式指派到程序各语法结构的一组规则。
- 由类型检查程序实现。
- 同一语言的不同编译程序使用的类型体制可能不同
  - 数组作为参数传递时，数组的下标集合可以指明，也可以不指明。
  - 原因：不同的Pascal语言编译程序实现的类型体制不同
  - UNIX系统中，lint命令实现的类型体制比C语言编译程序本身所实现的更详细。

类型表达式的递归定义

(1) 基本类型是类型表达式

boolean、char、integer、和 real

错误类型 type_error、回避类型 void

```
typedef double real;
real x, y;
```

```
typedef int arr[10];
arr a, b;
```

(2) 类型名是类型表达式，因为类型表达式可以命名。

(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) 是元素类型为T和下标集合为I的数组的类型表达式，I通常是一个整数域。

A:array[1..10] of integer;

A的类型表达式为：array(1..10, integer)

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

类型表达式的递归定义

③ 记录：把类型构造器 record 作用于记录中各域类型的笛卡儿积上，就形成了记录的类型表达式。

域类型是由域名和域的类型表达式组成的二元组。

```
B：  record
       i: integer;
       c: char;
       r: real
end;
```

```
struct {
    int i;
    char c;
    float r;
} B;
```

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

B的类型表达式：record((*i*×integer)×(*c*×char)×(*r*×real))

row 的类型表达式：record((*age*×integer)×(*name*×array(0..9 ，char)))

table 的类型表达式：array(0..9 ，row)

类型表达式的递归定义

④ 指针：如果T是类型表达式，那么 pointer(T) 也是类型表达式，表示类型“指向类型T的对象的指针”。

如：p:↑row;　　row *p;

P的类型表达式为：pointer(row)

⑤ 函数：从定义域类型D到值域类型R的映射。类型表达式 D→R

如：Pascal的内部函数 mod 的类型表达式：integer×integer→integer

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

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

形参名字?

如：C 语言函数声明 int square(int x) { return x*x };

函数*square*的类型表达式为：integer→integer

(4) 类型表达式中可以包含变量（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？
  - 新声明的类型名是一个类型表达式，它与该名字所代表的类型表达式是否总是等价的？

1. 结构等价(structure type equivalence)

- 两个类型表达式结构等价：
  - 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)
- 类型等价的变量，它们的类型具有相同的结构。

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：

```
bool  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

```
if (s==array(s1, s2))&&(t==array(t1, t2))
    return seqtest(s2, t2);
```

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。

- 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

- 如：

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

2. 名字等价(name type equivalence)

- 多数语言(如Pascal、C等)允许用户定义类型名

```
typedef int int_var;
int a;
int_var b;
```

- 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;
```

- 问题：a、b、c、d、e是否具有相同的类型?

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

- 关键： recP 和 pointer(recA) 是否等价?
- 回答：依赖于具体的实现系统

名字等价

- 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，当且仅当它们完全相同。
- 名字等价的变量，或者它们在同一个声明语句中定义，或者使用相同类型名声明。
- 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。
- 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

名字等价

名字等价

名字不等价

类型等价实例：C语言

- 使用名字等价和结构等价
  - 对于struct、enum和union的类型，使用名字等价
    - 例外：在不同的文件中定义，使用结构等价
- 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 采用了与 C语言类似的规则：

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。

```
struct rec1{
    short j;
    char c;
};
rec1 x, y;
struct rec2{
    short j;
    char c;
};
rec2 b;
```

与声明6.1等效的声明6.2：

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

```
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 的类型：recP 等价
  - c 和 d的类型：recD 等价
  - e的类型：recE
  - a、c、e不等价

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。
- 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。
    - 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，许多Pascal语言的编译程序允许不指明数组的下标集合，只检查元素的类型。
- lint，Unix系统的静态代码分析程序。
  - lint所实现的类型体制更详细
  - 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
  - 可以对源程序进行更加广泛的错误分析

1. 语言说明

- 假设语言文法如下：

$P \rightarrow D; S$

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

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

$A \rightarrow \varepsilon \mid paramlist$

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

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

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

$rparam \rightarrow \varepsilon \mid rplist$

$rplist \rightarrow rplist, E \mid E$

说明：
① 名字必须先声明后引用
② 每个声明语句声明一个名字
③ 过程声明允许嵌套
④ 数组下标从1开始

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

2. 符号表的建立

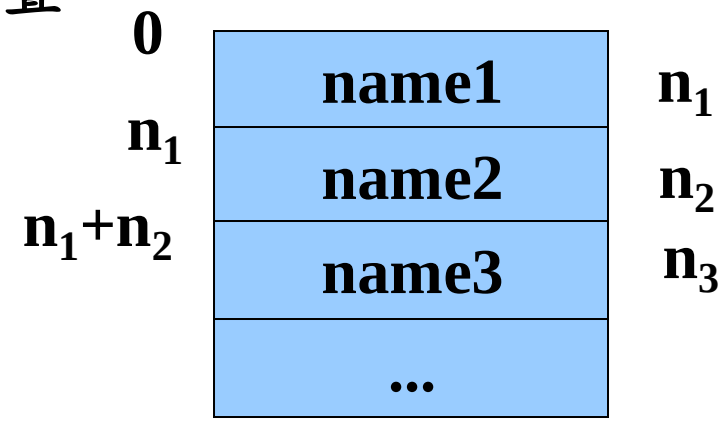
- 处理声明语句时，编译程序的任务：
  - 分离出每一个被声明的实体；
  - 尽可能多地将该实体的信息填入符号表，名字、类型、存储地址等。
- 声明语句的形式
  - 类型关键字的位置
    - 标识符表的前面，如 int a, b;
    - 标识符表的后面，如 a, b: integer;
  - 标识符表
    - 单一实体，如 Ada
    - 多个同类型的实体，如 Pascal、C、Java
    - 不同种类的实体，如 FORTRAN

(1) 过程中声明语句的处理

- 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：

$P \rightarrow D; S$
$D \rightarrow D; D$
$D \rightarrow id: T$
$T \rightarrow char \mid integer \mid real \mid boolean$
$\mid array\ [num]\ of\ T_1 \mid \uparrow T_1$

- 翻译目标
  - 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
  - T.type：记录实体的类型
  - enter(id.name，T.type，offset)

翻译方案 6.1

{ offset=0 }

P→ D;S

P → MD;S

M →ε { offset=0 }

D→D;D

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; }

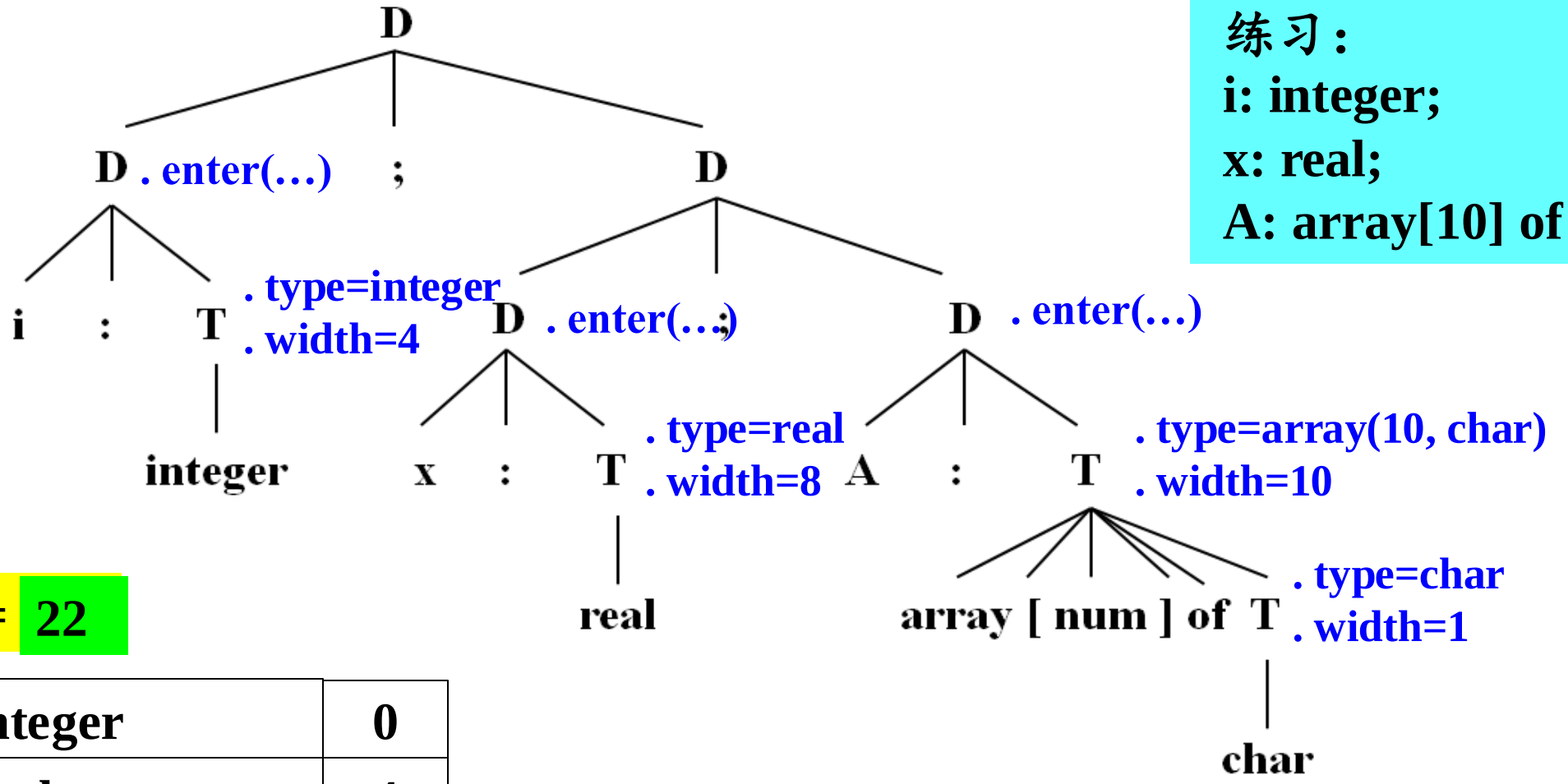
练习：

i: integer;

x: real;

A: array[10] of char

翻译方案 6.1应用示例



练习：
i: integer;
x: real;
A: array[10] of char

offset = 22

i	integer	0
x	real	4
A	array(10,char)	12

(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; S

D→ D； D

D→ id： T

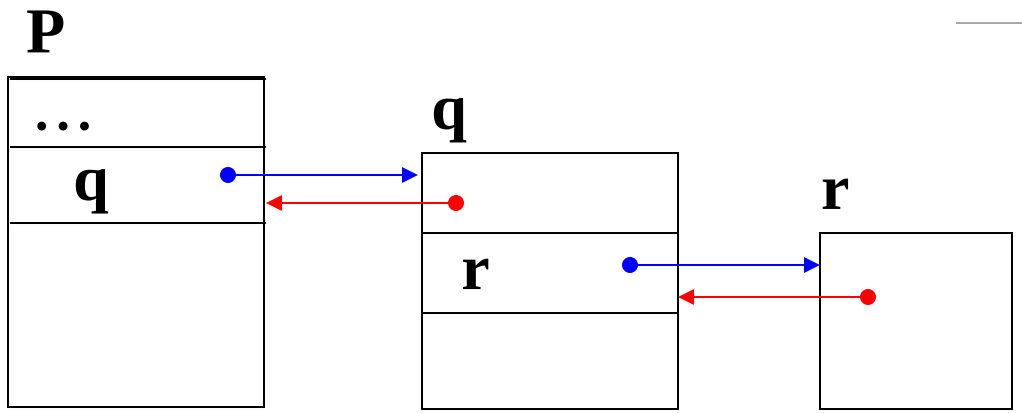
D→ proc id(A); N D;S

D→ fun id(A):T; N D;S

A→ε | paramlist

paramlist→id:T

| paramlist, id: T



M：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

N：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

函数形参?

- 作用域信息的保存
- 文法产生式:

P→ M D; S

D→ D; D

D→ id: T

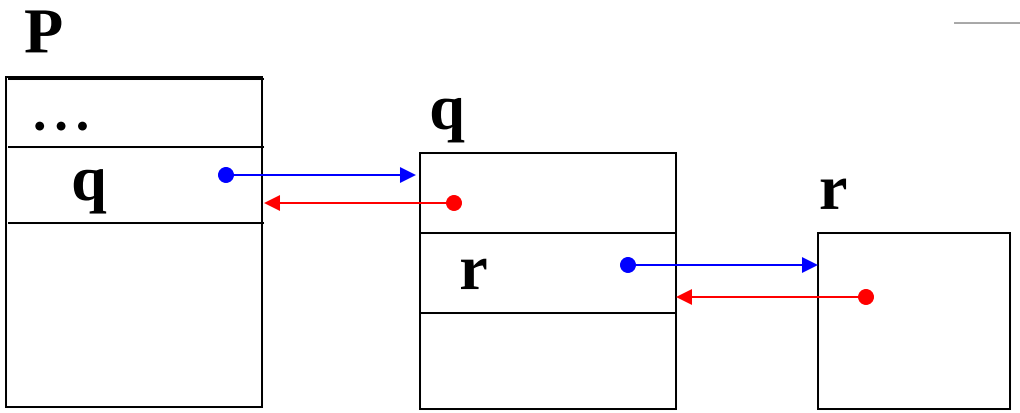
D→ proc id M (A); D; S

D→ fun id M (A):T;D;S

A→ε | paramlist

paramlist→id:T

| paramlist, id: T



M: 创建主程序的符号表、初始化

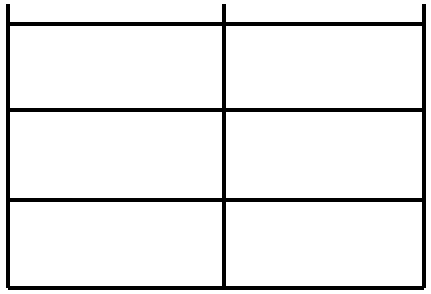
M: 记录主程序中声明的变量所需要的空间

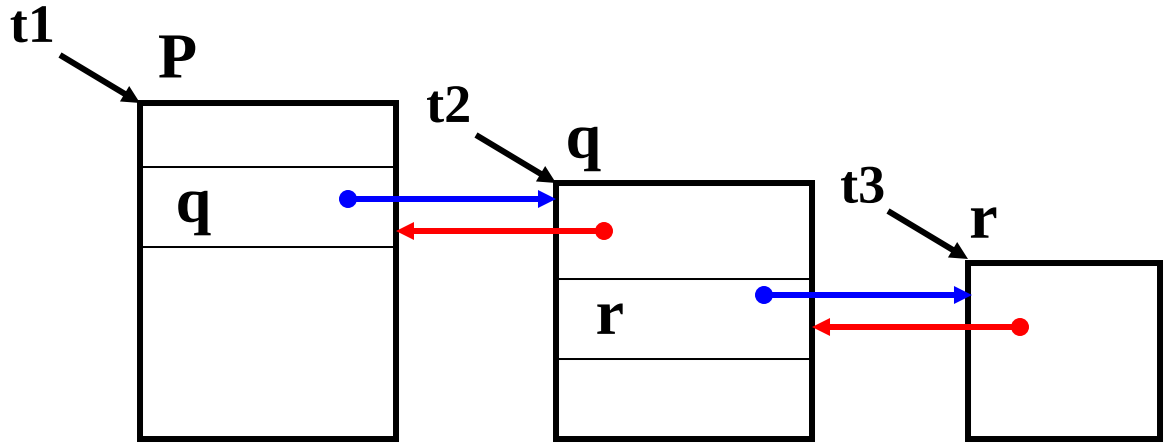
M: 重定位操作:
记录子过程中声明的局部变量所需要的空间,
返回到外围过程

M: 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

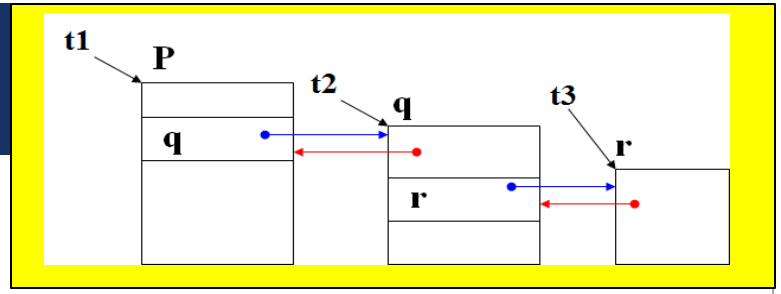


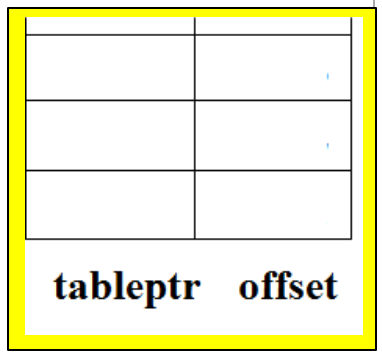


- 过程：

maketable(previous)
enter(table, name, type, offset)
addtheader(table, param_num, param_wth, ret_type, loc_var_width)
addwidth(table, glo_var_width)
enterproc(table, name, ‘proc’/‘fun’, 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

练习：
利用翻译方案6.2处理 sort 程序

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.type=void; 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; }

(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

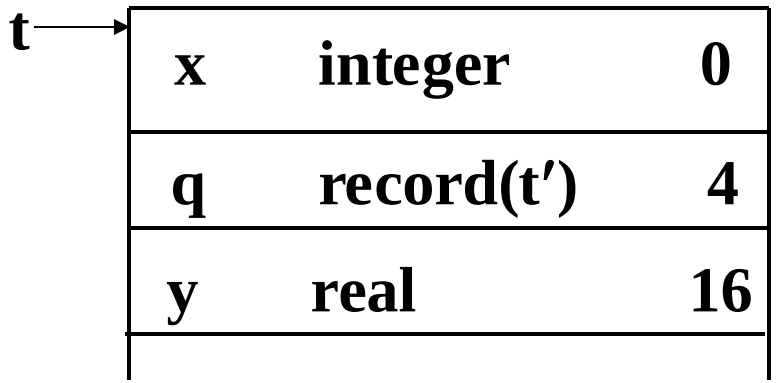
T→ record L D end { T.type=record(top(tableptr));
T.width=top(offset);
pop(tableptr); pop(offset); }

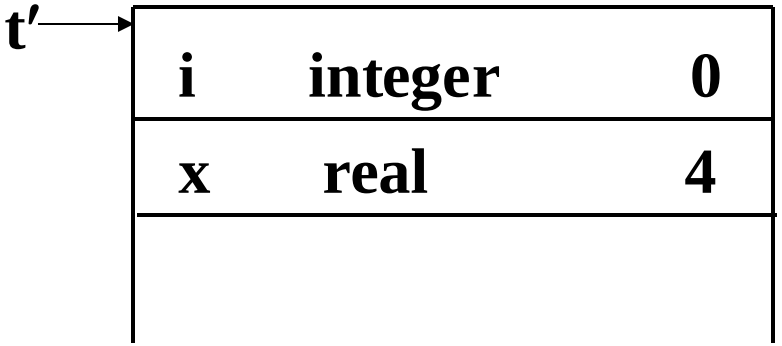
L→ε { t=mktable(nil);
push(t, tableptr); push(0, offset); }

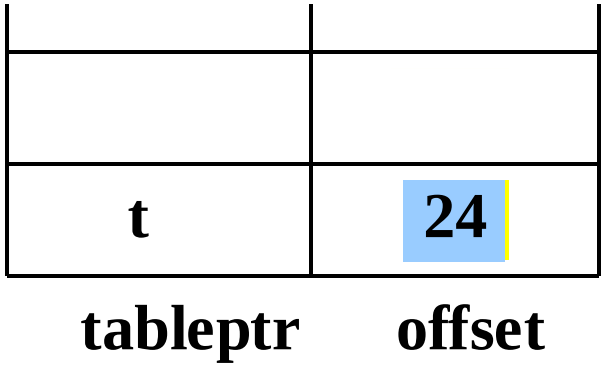
示例

- 声明

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







T.type=record(t′)
T.width=12

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 { *p*=*lookup*(id.*name*);
if (*p*!=nil) E.type=*gettype*(*p*);
else E.type = type_error; }

$E \rightarrow id_1 < id_2$ { p_1=*lookup*(id_1.*name*); p_2=*lookup*(id_2.*name*);
if ((p_1==nil) || (p_2==nil)) *E.type*= type_error;
else if (((*gettype*(p_1)==char) && (*gettype*(p_2)==char)) ||
((*gettype*(p_1)==integer) && (*gettype*(p_2)==integer)) ||
((*gettype*(p_1)==real) && (*gettype*(p_2)==real)))
E.type = boolean;
else E.type = type_error; }

表达式的类型检查

$E \rightarrow E_1 + E_2$ { 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

$E \rightarrow E_1$ 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; }

$E \rightarrow id[E_1]$ { *p*=*lookup*(id.*name*);
if (*p*==nil) E.type= type_error;
else if ((*gettype*(*p*)==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

E→id(rparam) { *p*=*lookup*(id.*name*);
if (*p*==nil) E.type= type_error;
else if ((gettype(*p*)==‘fun’) &&
(rparam.num==*getpnum*(*p*)) &&
(*checkptype*(*p*, rparam.type)))
E.type=*getrettype (p);*
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; }

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 { *p*=*lookup*(id.*name*);
if (*p*==nil) *S*.type = type_error;
else if (*gettype*(*p*) ==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

```
S→id(rparam) { p=lookup(id.name);
               if (p==nil)  S.type = type_error;
               else if ((gettype(p )==‘proc’) &&
                        ( rparam.num==getpnum(p )) &&
                        (checkptype(p , rparam.type)))
                    S.type=void;
               else S.type=type_error;    }

S→S1;S2  {   if (S1.type==void)&&(S2.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 ;   }
```

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。
- 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**。
  - 允许不丢失信息的隐式转换，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。

```
void foo(void) {
   unsigned int a = 8;
   int b = -20;
   (a+b > 8) ? puts("> 8") : puts("<= 8");
}
```

类型转换

- 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，并且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。

本章小结

- 语义分析的概念
  - 检查语义的合法性
  - 符号表的建立和管理
- 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
- 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
- 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

学习任务

- 作业要求
  - 掌握数据对象的类型表示
  - 数据对象类型等价判断
- 研究性学习
  - 分析源语言的特性对符号表组织的影响。