



软件分析

# 过程内指针分析

熊英飞

北京大学



# 指向分析

- 每个指针变量可能指向的内存位置
- 通常是其他很多分析的基础
  
- 本节课先考虑流非敏感指向分析
- 先不考虑在堆上分配的内存，不考虑struct、数组等结构，不考虑指针运算（如 $*(p+1)$ ）
  - 内存位置==局部和全局变量在栈上的地址



# 指向分析——例子

```
o=&v;  
q=&p;  
if (a > b) {  
    p=*q;  
    p=o; }  
*q=&w;
```

- 指向分析结果

- $p = \{v, w\}$ ;
- $q = ?$
- $o = ?$

$p$ : 变量 $p$ 的地址（内存位置）

$p$ : 指针 $p$ 所指向的集合（指针变量）



# 指向分析——例子

```
o=&v;  
q=&p;  
if (a > b) {  
    p=*q;  
    p=o; }  
*q=&w;
```

- 指向分析结果
  - $p = \{v, w\}$ ;
  - $q = \{p\}$ ;
  - $o = \{v\}$ ;
- 问题：如何设计一个指向分析算法？



# 复习：方程求解

- 数据流分析的传递函数和 $\sqcap$ 操作定义了一组方程
  - $D_{v_1} = F_{v_1}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
  - $D_{v_2} = F_{v_2}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
  - ...
  - $D_{v_n} = F_{v_n}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
- 其中
  - $F_{v_1}(D_{v_1}, D_{v_2}, \dots, D_{v_n}) = f_{v_1}(I)$
  - $F_{v_i}(D_{v_1}, D_{v_2}, \dots, D_{v_n}) = f_{v_i}(\sqcap_{j \in pred(i)} D_{v_j})$
- 数据流分析即为求解该方程的最大解
  - 传递函数和 $\sqcap$ 操作表达了该分析的安全性条件，所以该方程的解都是安全的
  - 最大解是最有用的解



# 从不等式到方程组

- 有一个有用的解不等式的unification算法
  - 不等式
    - $D_{v_1} \sqsubseteq F_{v_1}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
    - $D_{v_2} \sqsubseteq F_{v_2}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
    - ...
    - $D_{v_n} \sqsubseteq F_{v_n}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
  - 可以通过转换成如下方程组求解
    - $D_{v_1} = D_{v_1} \sqcap F_{v_1}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
    - $D_{v_2} = D_{v_2} \sqcap F_{v_2}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
    - ...
    - $D_{v_n} = D_{v_n} \sqcap F_{v_n}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$




# Anderson指向分析算法

赋值语句	约束
$a = \&b$	$a \supseteq \{b\}$
$a = b$	$a \supseteq b$
$a = *b$	$\forall v \in b. a \supseteq v$
$*a = b$	$\forall v \in a. v \supseteq b$

$a$ : 变量 $a$ 的地址  
(内存位置)  
 $a$ : 指针 $a$ 所指向  
的集合 (指针变  
量)

其他语句可以转换成这四种基本形式

$*a = **b;$    $c = *b;$   
 $d = *c;$   
 $*a = d;$



# Anderson指向分析算法-例

```
o=&v;  
q=&p;  
if (a > b) {  
    p=*q;  
    p=o; }  
*q=&w;
```

- 产生约束
  - $o \supseteq \{v\}$
  - $q \supseteq \{p\}$
  - $\forall v \in q. p \supseteq v$
  - $p \supseteq o$
  - $\forall v \in q. v \supseteq \{w\}$
- 如何求解这些约束





# 约束求解方法—通用框架

- 将约束
  - $\mathbf{o} \supseteq \{v\}$
  - $\mathbf{q} \supseteq \{p\}$
  - $\forall v \in \mathbf{q}. \mathbf{p} \supseteq v$
  - $\mathbf{p} \supseteq \mathbf{o}$
  - $\forall v \in \mathbf{q}. v \supseteq \{w\}$
- 转换成标准形式
  - $\mathbf{p} = \mathbf{p} \cup \mathbf{o} \cup (\cup_{v \in \mathbf{q}} v) \cup (p \in \mathbf{q} ? \{w\} : \emptyset)$
  - $\mathbf{q} = \mathbf{q} \cup \{p\} \cup (q \in \mathbf{q} ? \{w\} : \emptyset)$
  - $\mathbf{o} = \mathbf{o} \cup \{v\} \cup (o \in \mathbf{q} ? \{w\} : \emptyset)$
- 等号右边都是递增函数



# 求解方程组

- $p = p \cup o \cup (\cup_{v \in q} v) \cup (p \in q ? \{w\} : \emptyset)$
- $q = q \cup \{p\} \cup (q \in q ? \{w\} : \emptyset)$
- $o = o \cup \{v\} \cup (o \in q ? \{w\} : \emptyset)$

$$\begin{aligned} p &= \{ \} \\ q &= \{ \} \\ o &= \{ \} \end{aligned}$$



$$\begin{aligned} p &= \{ \} \\ q &= \{p\} \\ o &= \{v\} \end{aligned}$$



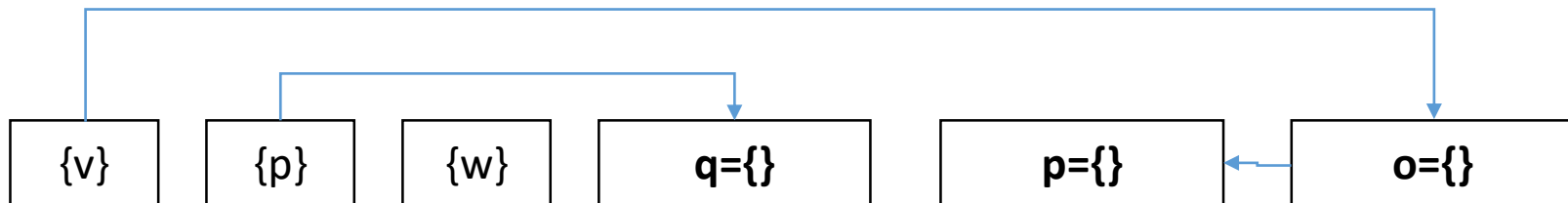
$$\begin{aligned} p &= \{v, w\} \\ q &= \{p\} \\ o &= \{v\} \end{aligned}$$





# 约束求解方法—直接计算

- $o \supseteq \{v\}$
- $q \supseteq \{p\}$
- $\forall v \in q. p \supseteq v$
- $p \supseteq o$
- $\forall v \in q. v \supseteq \{w\}$



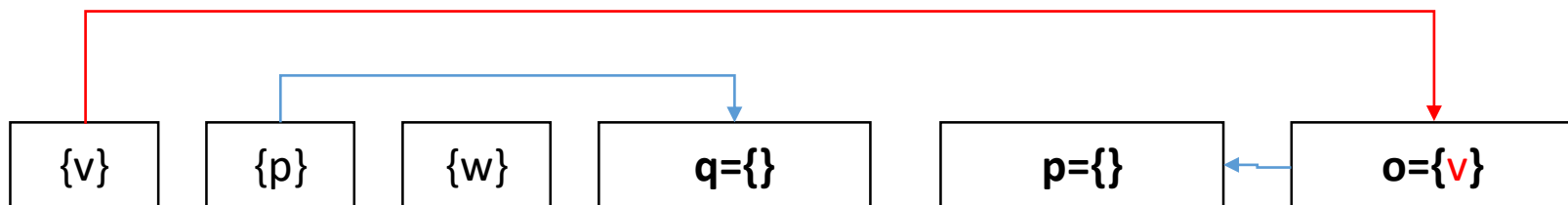
$$\forall v \in q. p \supseteq v$$

$$\forall v \in q. v \supseteq \{w\}$$



# 约束求解方法—直接计算

- $o \supseteq \{v\}$
- $q \supseteq \{p\}$
- $\forall v \in q. p \supseteq v$
- $p \supseteq o$
- $\forall v \in q. v \supseteq \{w\}$



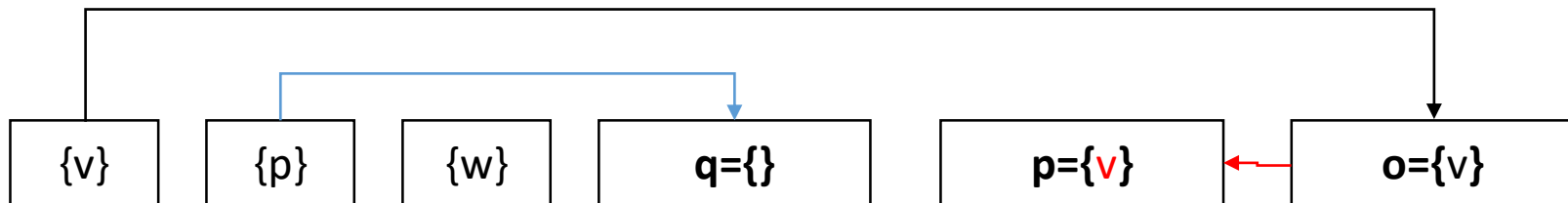
$$\forall v \in q. p \supseteq v$$

$$\forall v \in q. v \supseteq \{w\}$$



# 约束求解方法—直接计算

- $o \supseteq \{v\}$
- $q \supseteq \{p\}$
- $\forall v \in q. p \supseteq v$
- $p \supseteq o$
- $\forall v \in q. v \supseteq \{w\}$



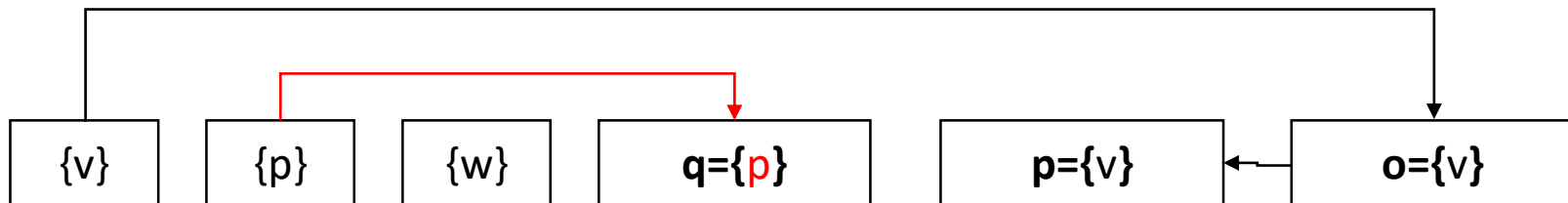
$$\forall v \in q. p \supseteq v$$

$$\forall v \in q. v \supseteq \{w\}$$



# 约束求解方法—直接计算

- $o \supseteq \{v\}$
- $q \supseteq \{p\}$
- $\forall v \in q. p \supseteq v$
- $p \supseteq o$
- $\forall v \in q. v \supseteq \{w\}$



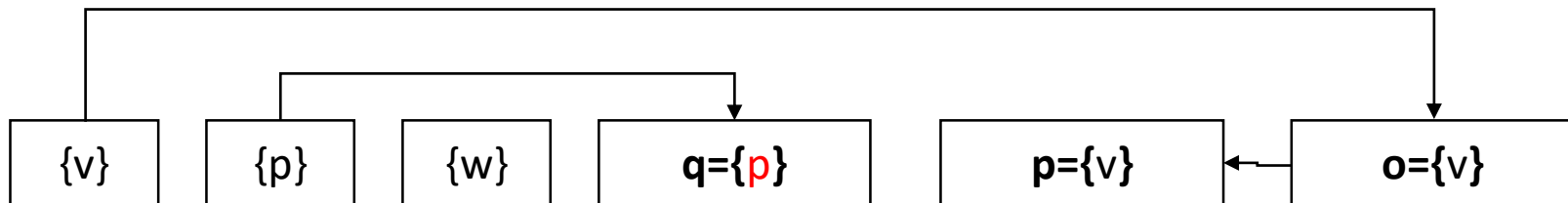
$$\forall v \in q. p \supseteq v$$

$$\forall v \in q. v \supseteq \{w\}$$



# 约束求解方法—直接计算

- $o \supseteq \{v\}$
- $q \supseteq \{p\}$
- $\forall v \in q. p \supseteq v$
- $p \supseteq o$
- $\forall v \in q. v \supseteq \{w\}$



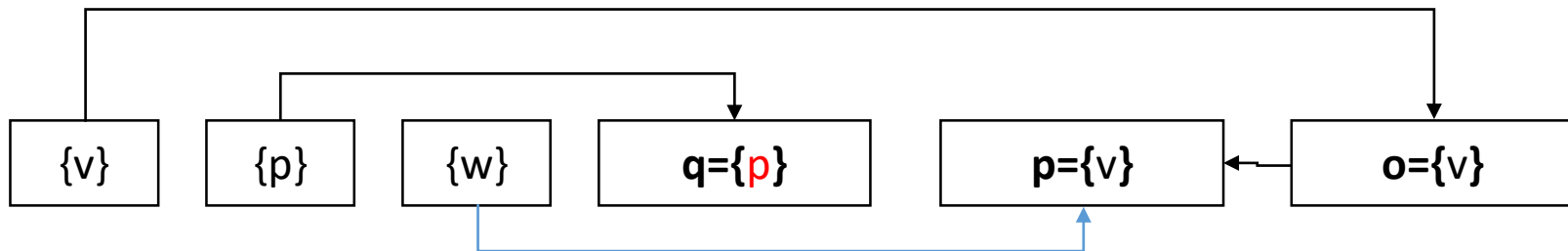
$$\forall v \in q. p \supseteq v$$

$$\forall v \in q. v \supseteq \{w\}$$



# 约束求解方法—直接计算

- $o \supseteq \{v\}$
- $q \supseteq \{p\}$
- $\forall v \in q. p \supseteq v$
- $p \supseteq o$
- $\forall v \in q. v \supseteq \{w\}$



$$\forall v \in q. p \supseteq v$$

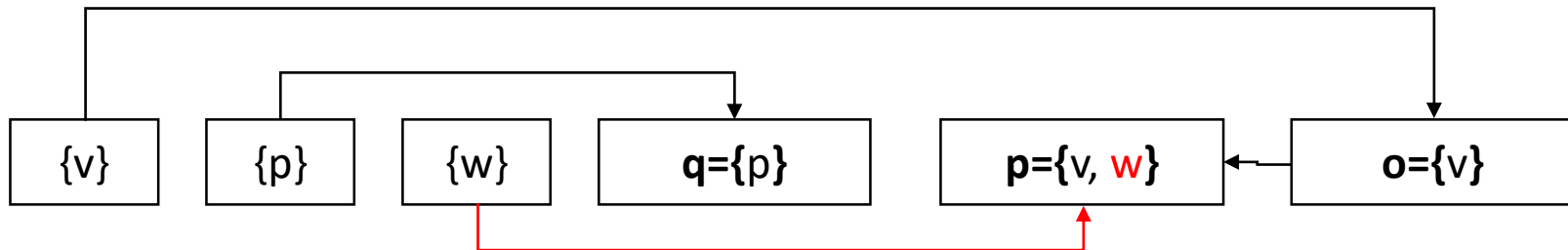
$$\forall v \in q. v \supseteq \{w\}$$





# 约束求解方法—直接计算

- $o \supseteq \{v\}$
- $q \supseteq \{p\}$
- $\forall v \in q. p \supseteq v$
- $p \supseteq o$
- $\forall v \in q. v \supseteq \{w\}$



$$\forall v \in q. p \supseteq v$$

$$\forall v \in q. v \supseteq \{w\}$$



# 复杂度分析

- 对于每条边来说，前驱集合新增元素的时候该边将被激活，激活后执行时间为 $O(m)$ ，其中 $m$ 为新增的元素数量
  - 应用均摊分析，每条边传递的总复杂度为 $O(n)$ ，其中 $n$ 为结点数量
- 边的数量为 $O(n^2)$
- 总复杂度为 $O(n^3)$



# 进一步优化

- 强连通子图中的每个集合必然相等
- 动态检测图中的强连通子图，并且合并成一个集合
- 参考论文：
  - The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code, Hardekopf and Lin, PLDI 2007



# 流敏感的指针分析算法

- 如何把Anderson算法转换成数据流分析?
  - 半格集合是什么?
    - 指针变量到内存位置集合的映射
  - 交汇操作是什么?
    - 对应内存位置集合取并
  - 四种基本操作对应的转换函数是什么?



# 流敏感的指针分析算法

赋值语句	转换函数
$a = \&b$	$f(V) = V[a \mapsto \{b\}]$
$a = b$	$f(V) = V[a \mapsto b]$
$a = *b$	$f(V) = V \left[ a \mapsto \bigcup_{\forall v \in b} v \right]$
$*a = b$	?



# 流敏感的指针分析算法

赋值语句	转换函数
$a = \&b$	$f(V) = V[a \mapsto \{b\}]$
$a = b$	$f(V) = V[a \mapsto b]$
$a = *b$	$f(V) = V \left[ a \mapsto \bigcup_{\forall v \in b} v \right]$
$*a = b$	$f(V) = \begin{cases} \forall v \in a. V[v \mapsto b] &  a  = 1 \\ \forall v \in a. V[v \mapsto v \cup b] &  a  > 1 \end{cases}$

Strong Update   Weak Update

注：这里不考虑变量没有被初始化的情况



# 流敏感的指针分析算法

- 传统流敏感的指针分析算法很慢
- 最新工作采用部分SSA来对流敏感进行加速，可以应用到百万量级的代码
- 参考论文：
  - Hardekopf B, Lin C. Flow-sensitive pointer analysis for millions of lines of code. CGO 2011:289-298.



# 堆上分配的内存

- `a=malloc();`
- `malloc()`语句每次执行创建一个内存位置
- 无法静态的知道`malloc`语句被执行多少次
  - 无法定义出有限半格
- 应用抽象的思想
  - 每个`malloc()`创建一个抽象内存位置
  - `a=malloc(); //1`
  - $f(V) = V[\mathbf{a} \mapsto \{1\}]$





# Struct

```
Struct Node {  
    int value;  
    Node* next;  
    Node* prev;  
};  
  
a = malloc();  
a->next = b;  
a->prev = c;
```

- 如何处理结构体的指针分析?
- 域非敏感Field-Insensitive分析
- 基于域的Field-Based分析
- 域敏感Field-sensitive分析

# 域非敏感Field-Insensitive分析



```
Struct Node {  
    int value;  
    Node* next;  
    Node* prev;  
};
```

```
a = malloc();  
a->next = b;  
a->prev = c;
```

- 把所有struct中的所有fields当成一个对象
- 原程序变为
  - $a' = \text{malloc}()$ ;
  - $a' = b$ ;
  - $a' = c$ ;
  - 其中 $a'$ 代表 $a$ ,  $a \rightarrow \text{next}$ ,  $a \rightarrow \text{prev}$
- 分析结果
  - $a$ ,  $a \rightarrow \text{next}$ ,  $a \rightarrow \text{prev}$ 都有可能指向 $\text{malloc}()$ ,  $b$ 和 $c$



# 基于域的Field-Based分析

```
Struct Node {  
    int value;  
    Node* next;  
    Node* prev;  
};  
a = malloc();  
a->next = b;  
a->prev = c;  
b = malloc();  
b->next = c;
```

- 把所有对象的特定域当成一个对象
- 原程序变为
  - a=malloc();
  - next=b;
  - prev=c;
  - b=malloc();
  - next = c;
- 分析结果
  - a和a->prev是精确的，但a->next和b->next都指向b和c



# 域敏感Field-sensitive分析

```
Struct Node {  
    int value;  
    Node* next;  
    Node* prev;  
};
```

```
a = malloc();  
a->next = b;  
a->prev = c;
```

- 对于Node类型的内存位置x，添加两个指针变量
  - x.next
  - x.prev
- 对于任何Node类型的内存位置x，拆分成四个内存位置
  - x
  - x.value
  - x.next
  - x.prev
- a->next = b转换成
  - $\forall x \in a, x.next \supseteq b$



# Java上的指向分析

- Java上的指向分析可以看成是C上的子集

Java	C
<code>A a = new A();</code>	<code>A* a = malloc(sizeof(A));</code>
<code>a.next = b</code>	<code>a-&gt;next = b</code>
<code>b = a.next</code>	<code>b = a-&gt;next</code>

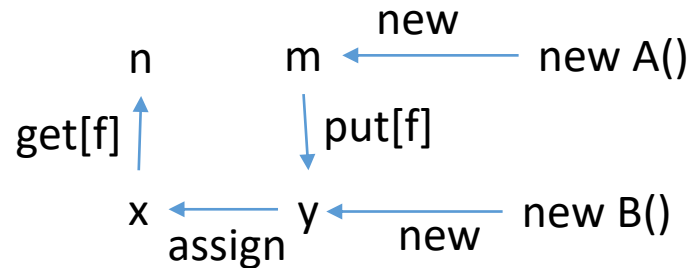


# 基于CFL可达性的域敏感分析

```

y = new B();
m=new A();
x=y;
y.f=m;
n=x.f;

```



图上的每条边f同时存在反向边f

```

FlowTo= new (assign | put[f] Alias get[f])*
PointsTo = (assign | get[f] Alias put[f])* new
Alias = PointsTo FlowTo

```

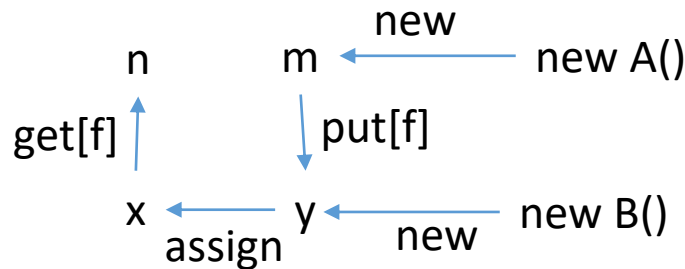


# 基于CFL可达性的域敏感分析

```

y = new B();
m=new A();
x=y;
y.f=m;
n=x.f;

```



图上的每条边f同时存在反向边f

```

FlowTo= new (assign | put[f] Alias get[f])*
PointsTo = (assign | get[f] Alias put[f])* new
Alias = PointsTo FlowTo

```

以下文法安全吗？精确吗？

- FlowTo = new FlowTo'  
FlowTo' = put[f] FlowTo' get[f]  
| FlowTo' FlowTo' | assign | ε
- FlowTo= new (Alias | put[f] Alias get[f])\*  
PointsTo = (Alias | get[f] Alias put[f])\* new  
Alias = PointsTo FlowTo | assign | assign



# 基于CFL和基于Anderson算法的域敏感分析等价性

基于CFL	基于Anderson算法
$\begin{array}{c} \text{PointsTo} \\ x \longrightarrow m \end{array}$	$m \in x$
$\begin{array}{c} \text{FlowsTo} \\ m \longrightarrow x \end{array}$	$m \in x$
$\begin{array}{c} \text{Alias} \\ x \longrightarrow y \end{array}$	$x \cap y \neq \emptyset$
$\exists y. y \xrightarrow{\text{PointsTo}} n \wedge y \xrightarrow{\text{puts}[f] \text{ PointsTo}} m$	$n \in m.f$

归纳证明 以上各行左右的等价性

- 从左边推出右边：在CFL的路径长度上做归纳
- 从右边推出左边：在集合的元素个数上做归纳





# 数组和指针运算

- 从本质上来讲都需要区分数组中的元素和分析下标的值
  - $p[i]$ ,  $*(p+i)$
- 大多数框架提供的指针分析算法不支持数组和指针运算
  - 一个数组被当成一个结点



# 参考文献

- Lecture Notes on Pointer Analysis
  - Jonathan Aldrich
  - <https://www.cs.cmu.edu/~aldrich/courses/15-8190-13sp/resources/pointer.pdf>
- 《编译原理》 12章