

软件安全测评

北京邮电大学计算机学院
信息安全系

张淼



火龙果•整理
uml.org.cn



第五讲 源代码安全审查

- 1. 编译原理的基本概念
- 2. 静态代码分析技术
- 3. 常用静态代码分析工具
- 4. 源代码审查实施步骤与策略



1. 编译原理的基本概念

- 介绍语言翻译的基本概念
 - 程序和语言
 - 翻译和解释
- 介绍翻译的步骤和相关的活动
- 介绍编译器的开发方法



基本观念

——程序、语言

- 计算机、程序、语言
 - 计算机接受指令，然后执行指令
 - 指令组成的序列，称为程序
 - 符合一定规则（文法）的程序的集合，称为语言
 - 语法（形式）
 - 语义（意义） //形式与意义之间的对应关系？
- 讨论：C语言与C语言程序之间的关系



基本观念

——语言与程序

- 语言的作用
 - 设计程序(选出特定的程序——构造程序)
- 程序的作用
 - 由计算机执行
 - 在人之间的交流想法，由于程序没有歧义
- 如何定义语言——**编译原理涉及的内容**
 - 以有限的规则，定义无限多的程序。



基本观念

——语言的多样性

- 语言的多样性
 - 方便在特定领域的应用
 - 交流障碍
- 解决方案
 - 统一语言，一个梦想
 - 没有个性的语言
 - 语言的不断发展
 - 翻译



基本观念——语言之间的翻译

- 对翻译的要求
 - 保持程序的意义，即功能不变。
- 翻译的可能性
 - Church猜想与通用计算机（语言的等价性）
 - ++、--、JNZ
- 翻译的策略——软件开发过程的策略
 - 编译（整体翻译）
 - 解释（逐句的翻译、执行）



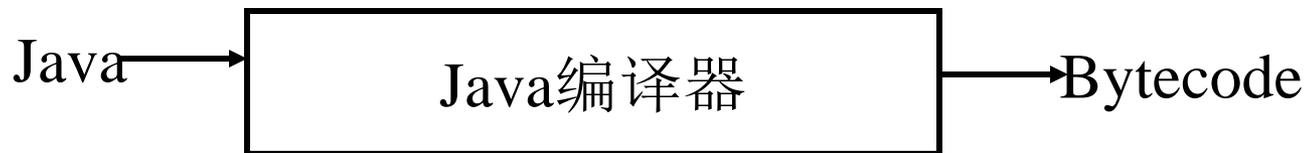
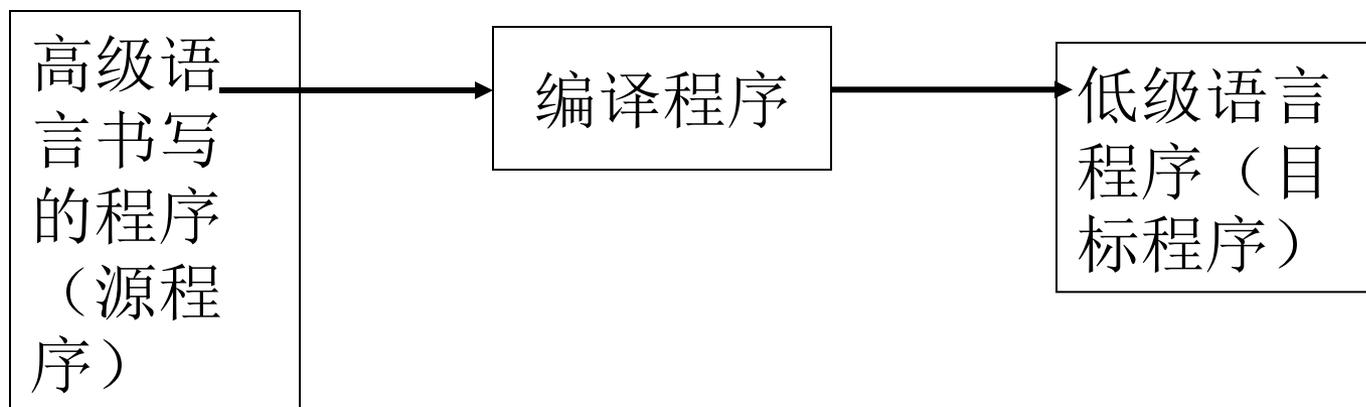
程序的执行方式

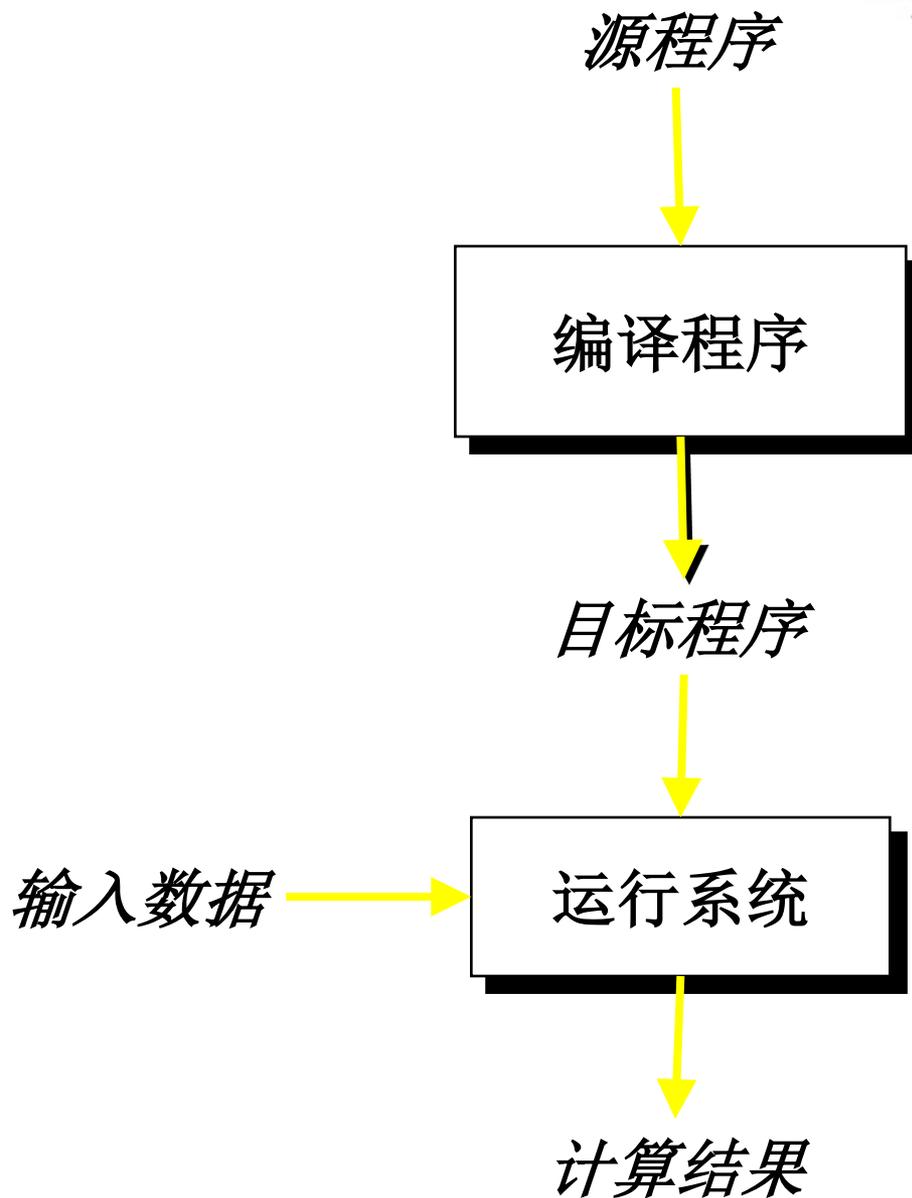
- 高级语言程序通常采用两种方式执行：解释方式和翻译方式
- 解释方式：逐个语句地分析和执行，如**Basic**，**Prolog**
 - 优点：易于查错
 - 缺点：效率低，运行速度慢
- 翻译方式：对整个程序进行分析，翻译成等价机器语言程序后执行，如**Pascal**，**Fortran**，**C**
 - 优点：只需分析和翻译一次，
 - 缺点：在运行中发现的错误必须在源程序中查找



什么是编译程序？

- 定义：是一种语言转换系统







需预处理的源程序

预处理程序

源程序

编译程序

目标汇编程序

汇编程序

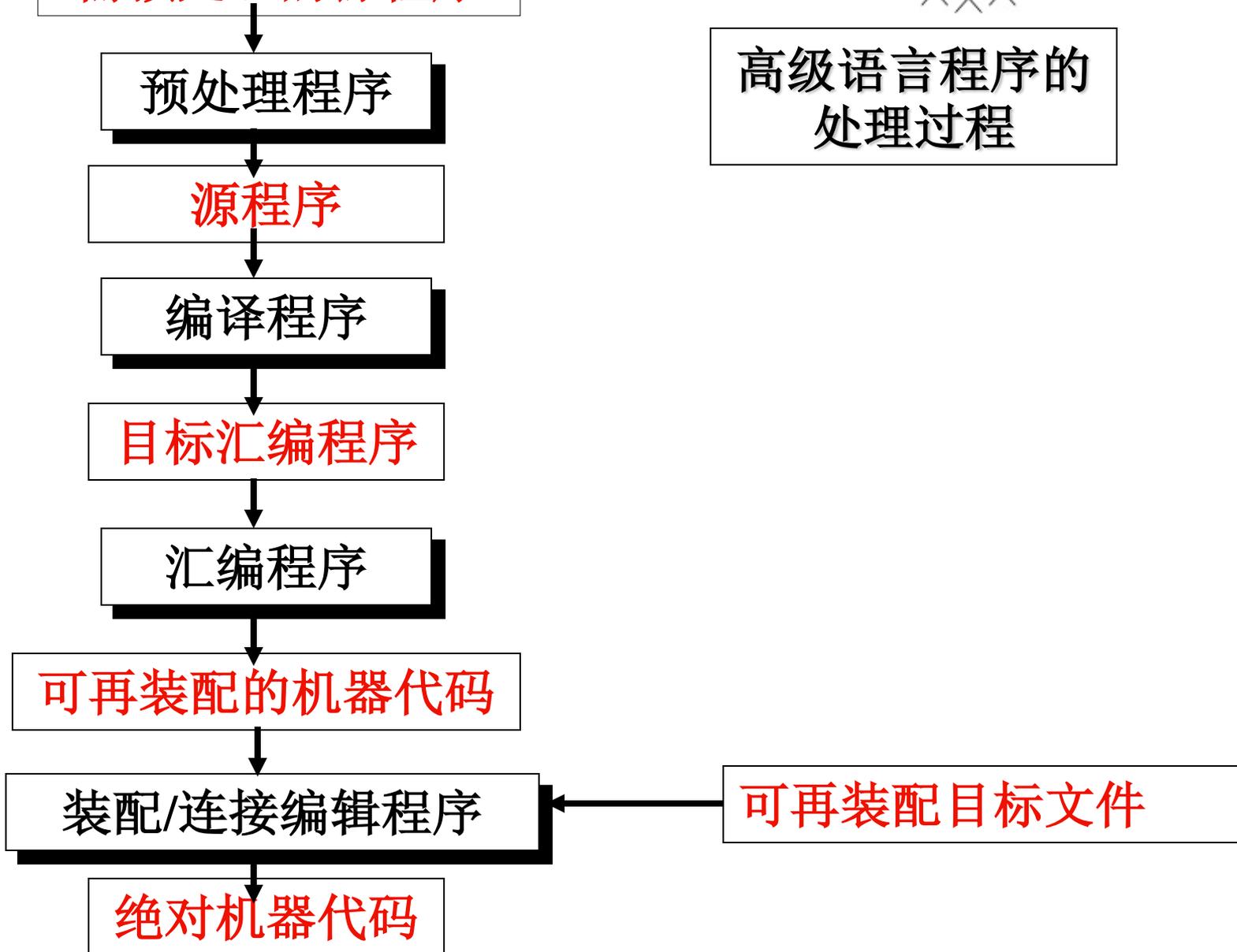
可再装配的机器代码

装配/连接编辑程序

绝对机器代码

高级语言程序的
处理过程

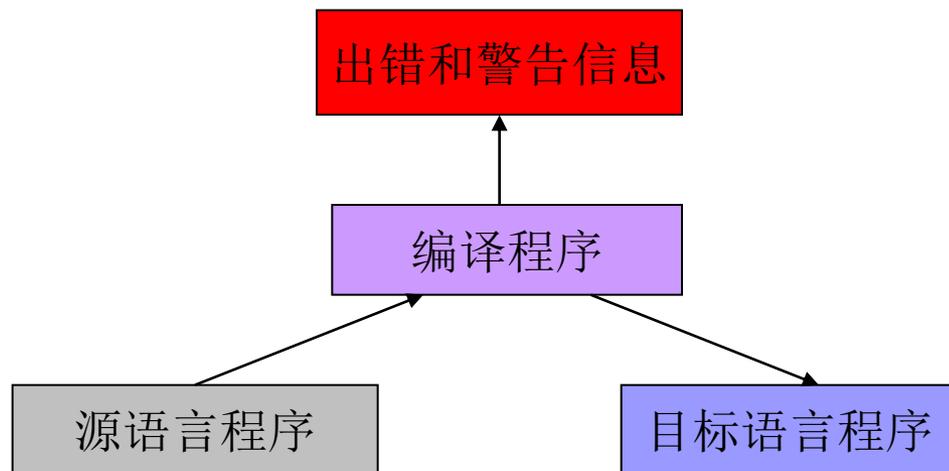
可再装配目标文件





编译程序的功能

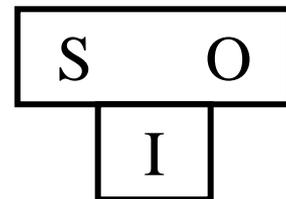
- 从功能上看，一个编译程序就是一个语言翻译程序。
- 源语言通常是一个高级语言，如 FORTRAN，C 或 Pascal。
- 目标语言通常是一个低级语言，如汇编或机器语言。





T形图

常用T形图来表示编译程序涉及的三个语言：



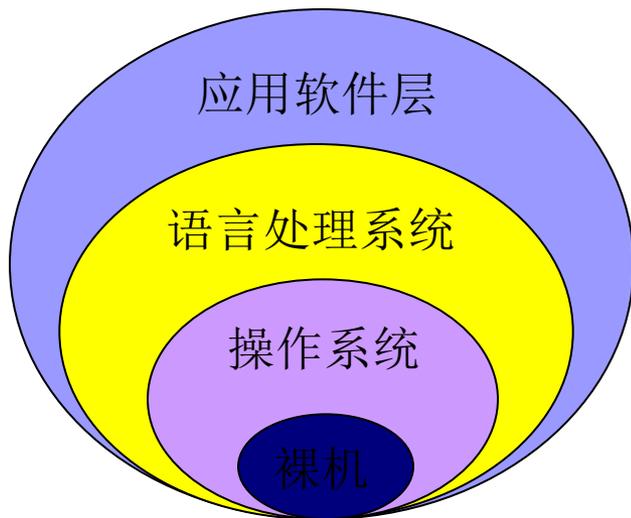
其中：

- S:源语言(程序), Source language(program)
- O:目标语言(程序), target/object language(program)
- I:实现语言, implementation language



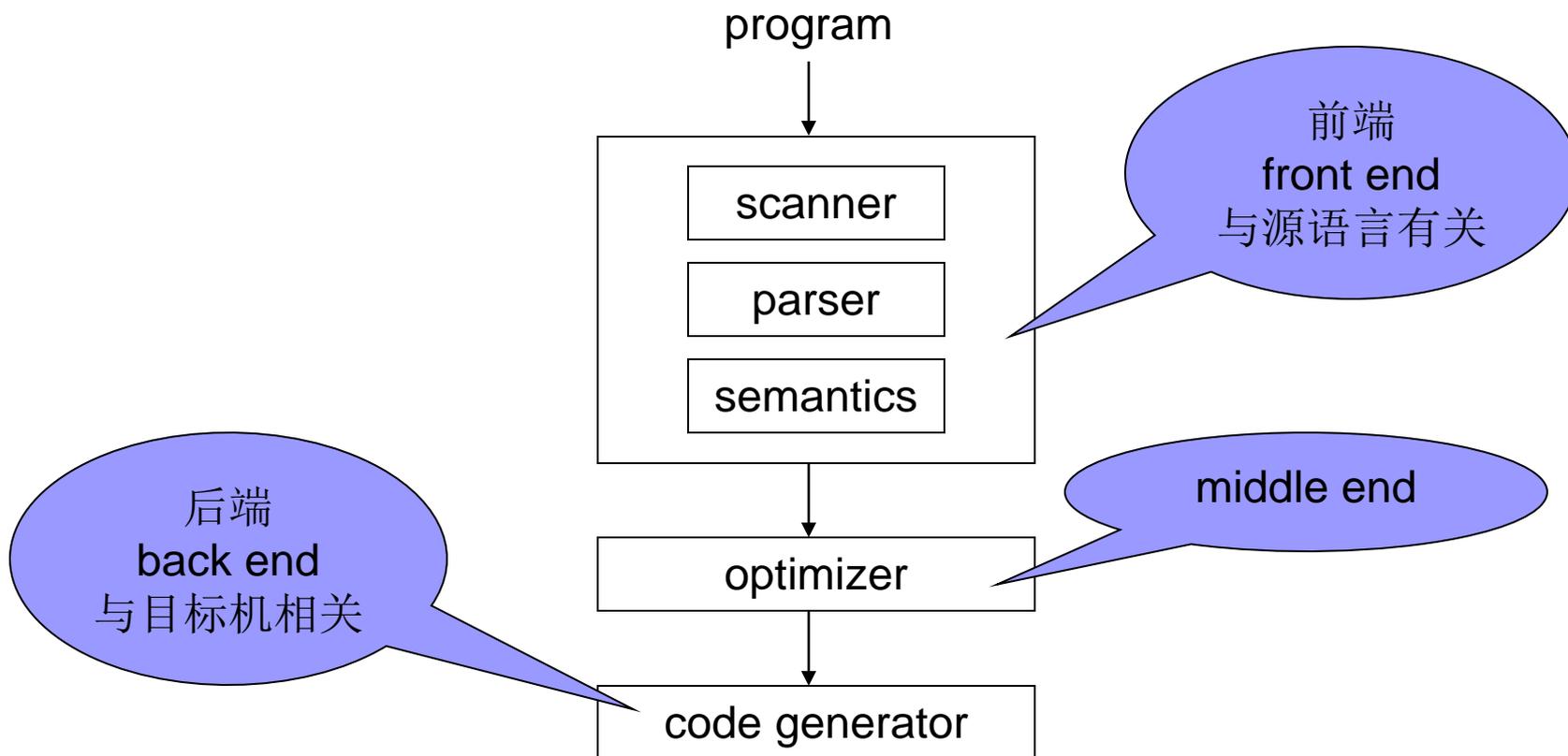
编译程序在计算机系统中的作用

- 编译系统是一种软件，一种系统软件。
 - 软件：计算机系统程序及其文档。
 - 系统软件：居于计算机系统最靠近硬件的一层，其他软件一般通过系统软件发挥作用。和具体的应用领域无关，如**编译系统**和**操作系统**等。
 - 语言处理系统：把软件语言书写的各种程序处理成可在计算机上执行的程序，如编译系统。





编译程序的组成结构





编译程序的结构

翻译外文资料与编译源程序进行类比

	翻译外文资料	编译源程序
分析	阅读原文 识别单词 分析句子	输入并扫描源程序 词法分析 语法分析
综合	修辞加工 写出译文	代码优化 目标代码生成



翻译之前的准备

■ 必须了解的

- 源语言（翻译谁——输入）
- 目标语言（翻译成谁——输出）
- 翻译方法（如何实现翻译）

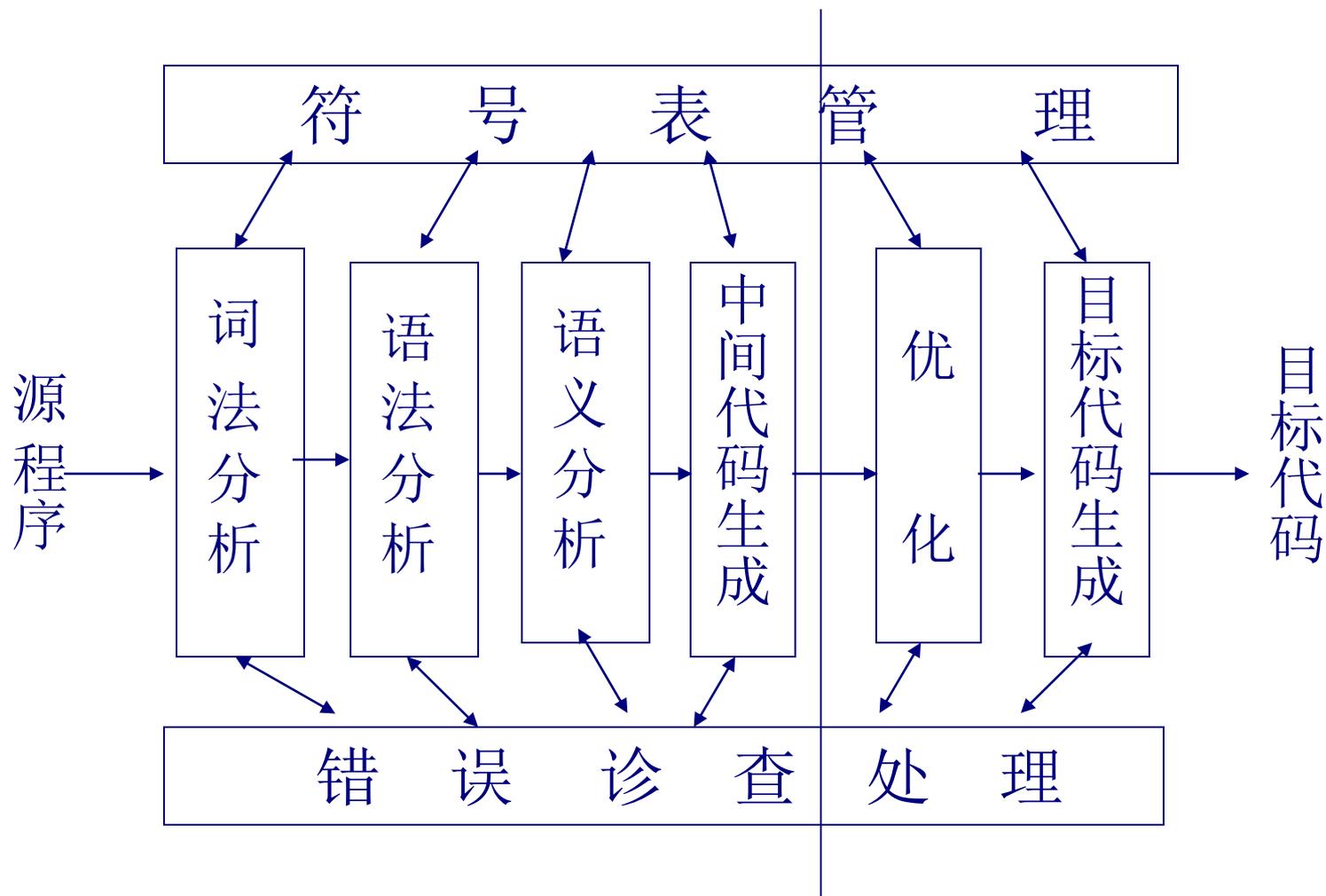
先定义
接口，
然后再
实现

■ 如何定义源语言和目标语言

- 程序的基本组成是字符，程序是一个字符串
- 由字符组成程序的方法



编译程序的逻辑结构





源程序

```
■ PROGRAM m;  
  ■ VAR a,b,c:real;  
  ■ BEGIN  
    ■ read(b,c);  
    ■ a:=b+c*60;  
    ■ write(a)  
  ■ END.
```



(1) 词法分析(Lexical analysis)

- 词法分析程序又称**扫描程序**。
- 是编译过程的第一个阶段，其任务是：读源程序的字符流、识别单词（如标识符、整数、界限符等），并转换成内部形式。
 - 输入字符串（即源程序）
 - 输出单词符号（最基本的语法单位）。



词法分析举例

- 一个C源程序片段：

```
int a;
```

```
a=a+2;
```

词法分析后返回(如右图):

单词类型

单词值

保留字

int

标识符

a

界符

;

标识符

a

算符(赋值)

=

标识符

a

算符(加)

+

整数

2

界符

;



经词法分析源程序被加工成单词流

- <保留字, PROGRAM> <标识符, m>
- <分隔符, ;> <保留字, VAR> <标识符, a>
- <标识符, b> <标识符, c> <分隔符, :>
- <标识符, real> <分隔符, ;>
- <保留字, BEGIN> <标识符, a>
- <算符, :=> <标识符, b> <算符, +>
- <标识符, c> <算符, *> <常数, 60>.....
- <保留字, END> <分隔符, .>



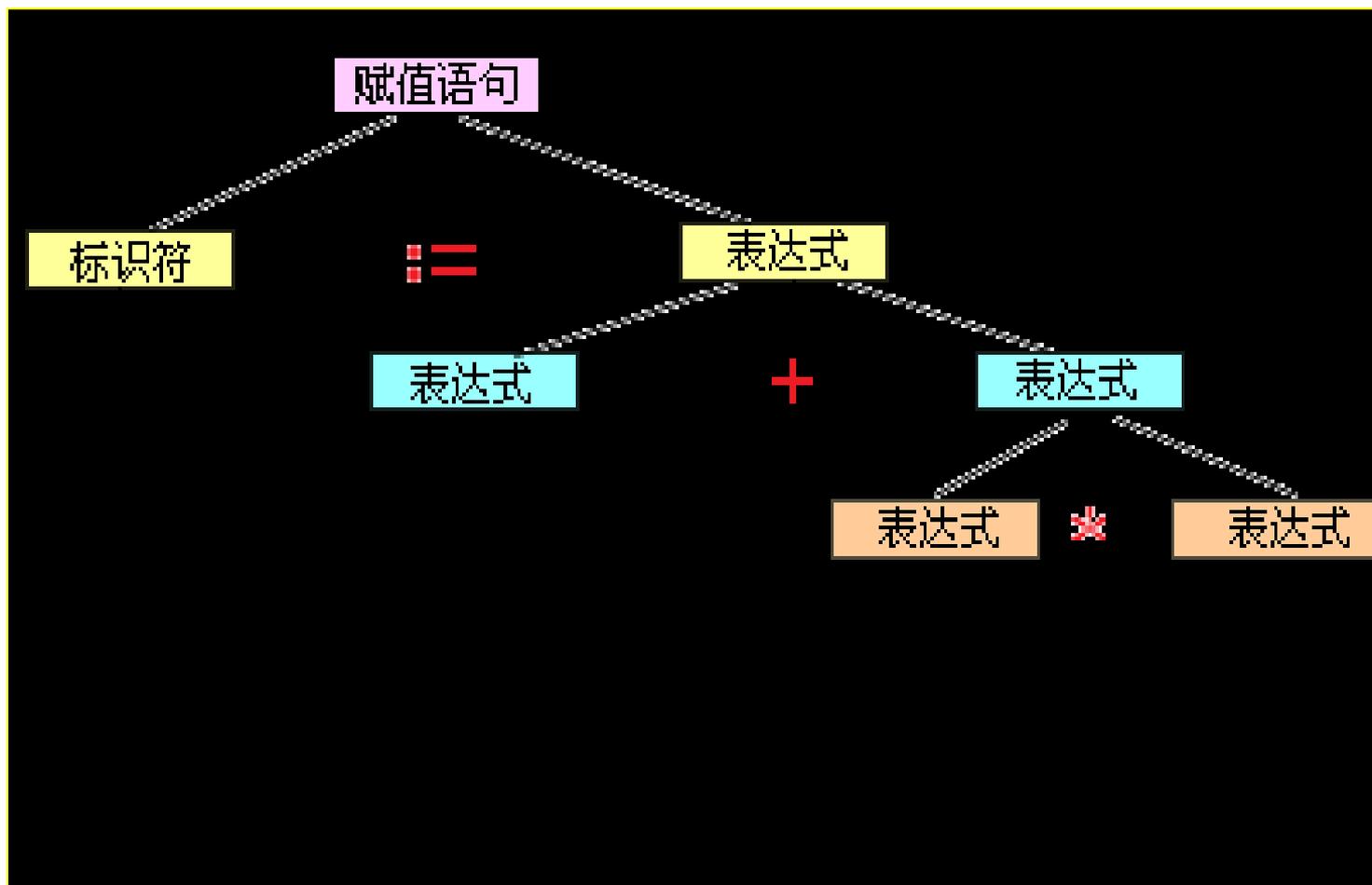
(2) 语法分析 (Syntax analysis)

■ 语法分析

- 语法定义如何由单词符号组成更大的语法单位
- 输入单词符号
- 输出语法单位及其之间的关系，通常是语法树
- 表达语法规则的主要工具为上下文无关文法

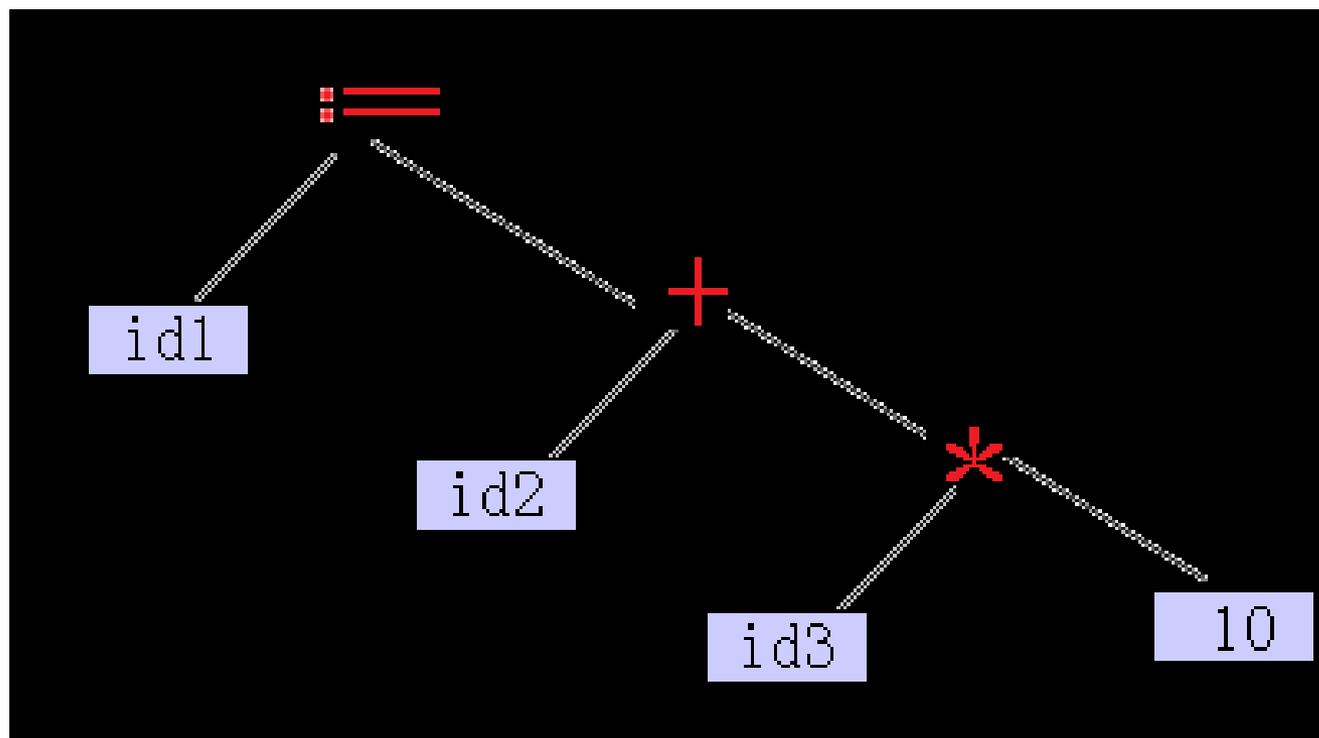


id1:=id2+id3*10 的语法树

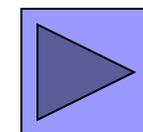
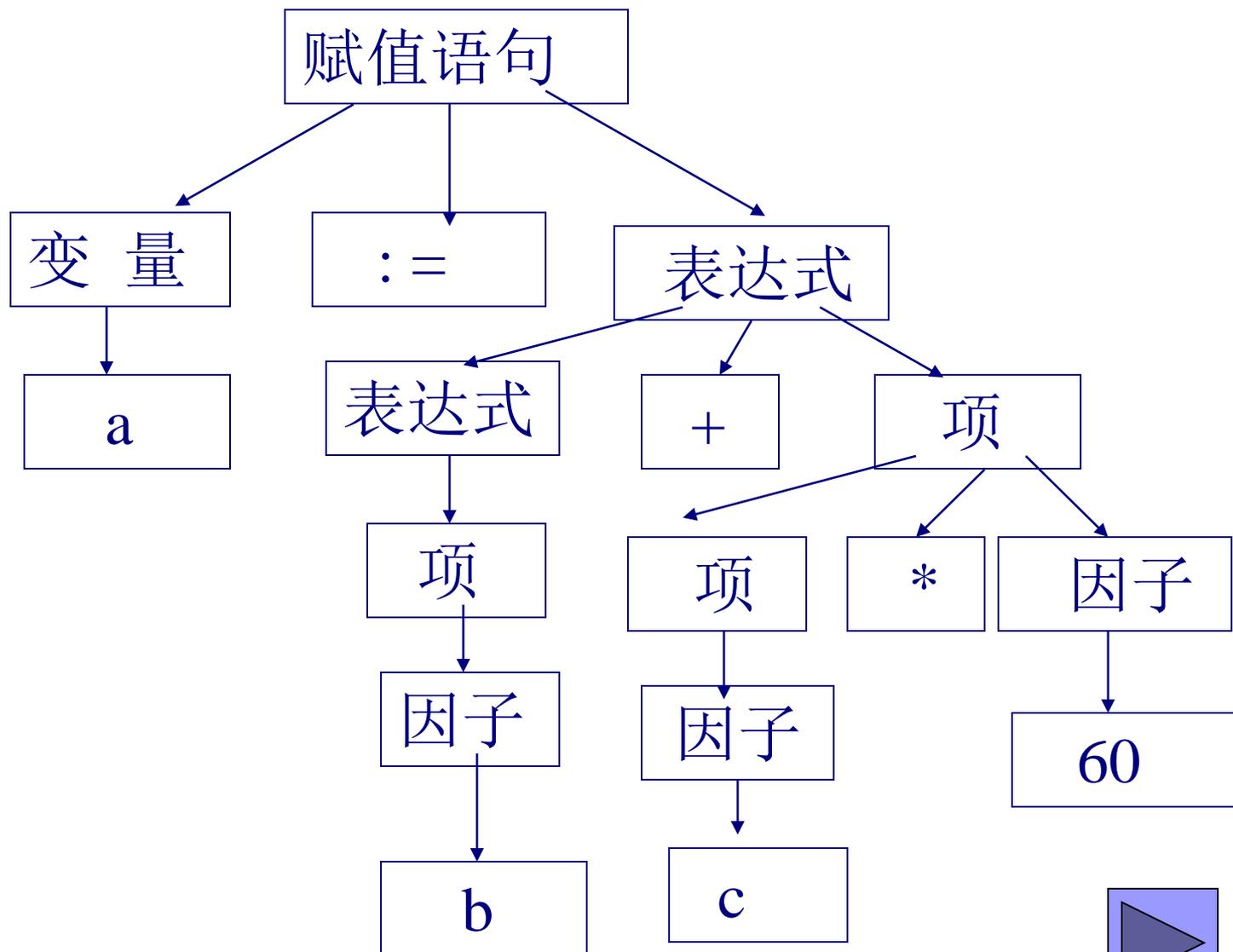




id1:=id2+id3*10的语法树的另一种形式



赋值语句经语法分析生成分析树



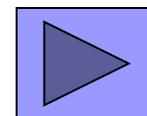
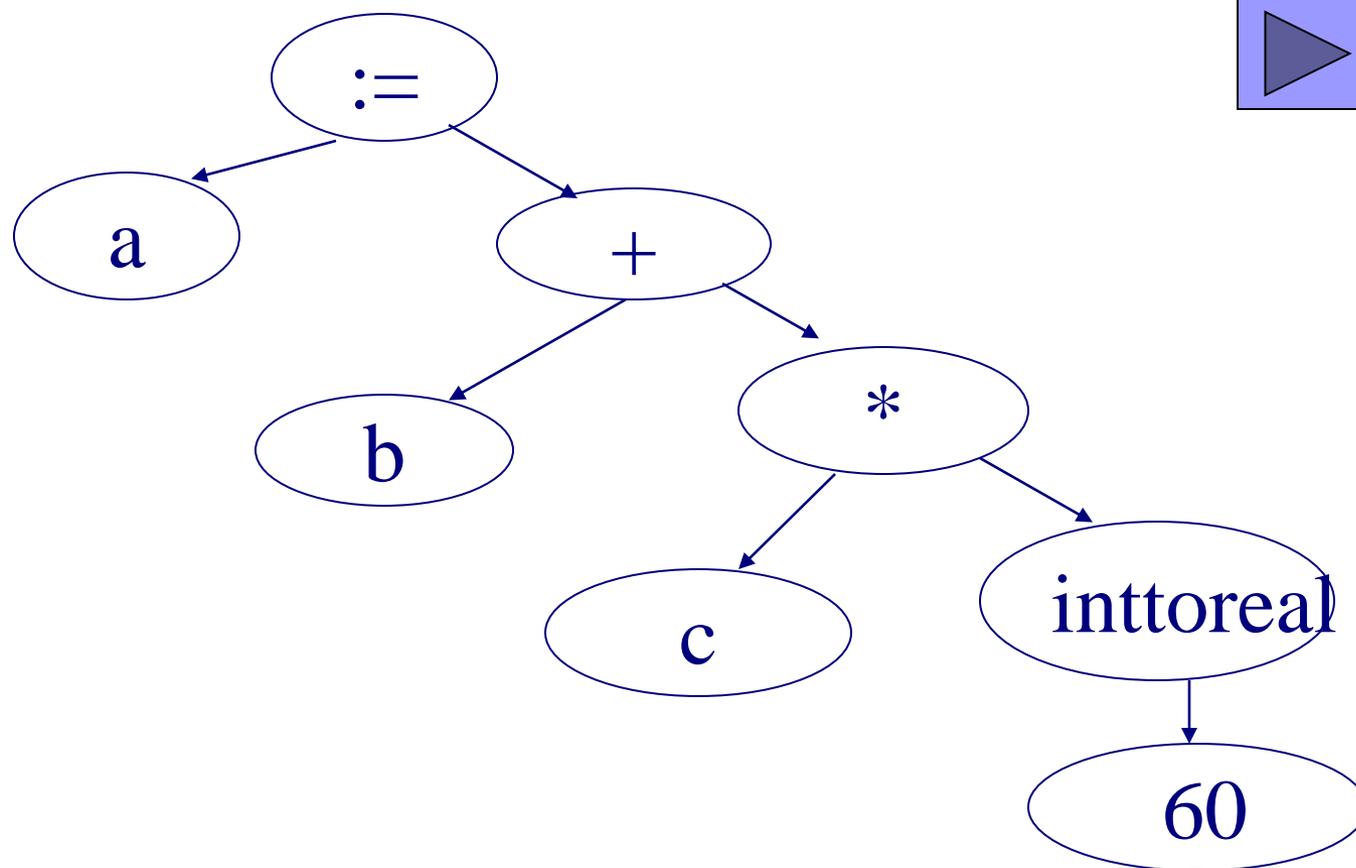


(3) 语义分析(Semantic analysis)和 中间代码生成

- 语义分析和中间代码生成
 - 分析（检查和计算）程序中各个语法单位的语义，翻译为中间代码形式。
 - 输入语法单位及其之间的关系（语法树）
 - 输出中间代码
 - 表达语义规则的主要工具是属性文法
 - 表达中间代码的主要工具是四元式
 - 主要使用语法制导的翻译方法



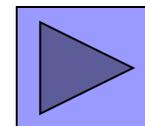
赋值语句经语义分析生成语法树





生成中间代码

- `temp1:=inttoreal(60);`
- `temp2:=c * temp1;`
- `temp3:=b +temp2;`
- `a :=temp3;`





(4) 代码优化

- 优化（在理论上不是必须的）
 - 输入中间代码
 - 输出优化后的中间代码
 - 对中间代码进行处理，期望得到高效的代码
 - 删除无用代码
 - 减少冗余
 - 目前不要对优化寄予过高的期望
 - 现在不能优化算法
 - 将来也许能在一定范围内自动选择算法或重构



(5) 目标代码生成

- 目标代码生成
 - 输入中间代码
 - 输出目标代码
 - 涉及的知识包括目标机器指令的选取、寄存器的分配、运行时存储空间组织等
 - 可以根据目标机的模型，对目标代码进一步进行优化

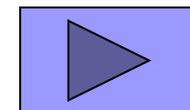


优 化

```
Temp1 :=c * 60.0  
a := b+temp1
```

生成目标代码

```
movf  c  ,  r2  ;  
mulf  #60.0 , r2  ;  
movf  b  ,  r1  ;  
addf  r2  ,  r1  ;  
movf  r1  ,  a  ;
```





符号表管理

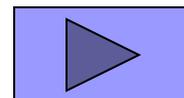
- 记录源程序中使用的名字(标识符)
- 收集每个名字的各种属性信息
 - 类型、作用域、分配存储信息

名字	信息
...	...



符号表

名字	种类	类型	层次	偏移量
m	过程		0	
a	变量	real	1	d
b	变量	real	1	d+4
c	变量	real	1	d+8





出错处理

- 检查错误、报告出错信息、排错、恢复编译工作
- 词法错误和语法错误可由编译程序在编译时刻查出。
- 语义错误常采用下列方式查出：
 - 静态模拟检查：
 - 动态调试检查：

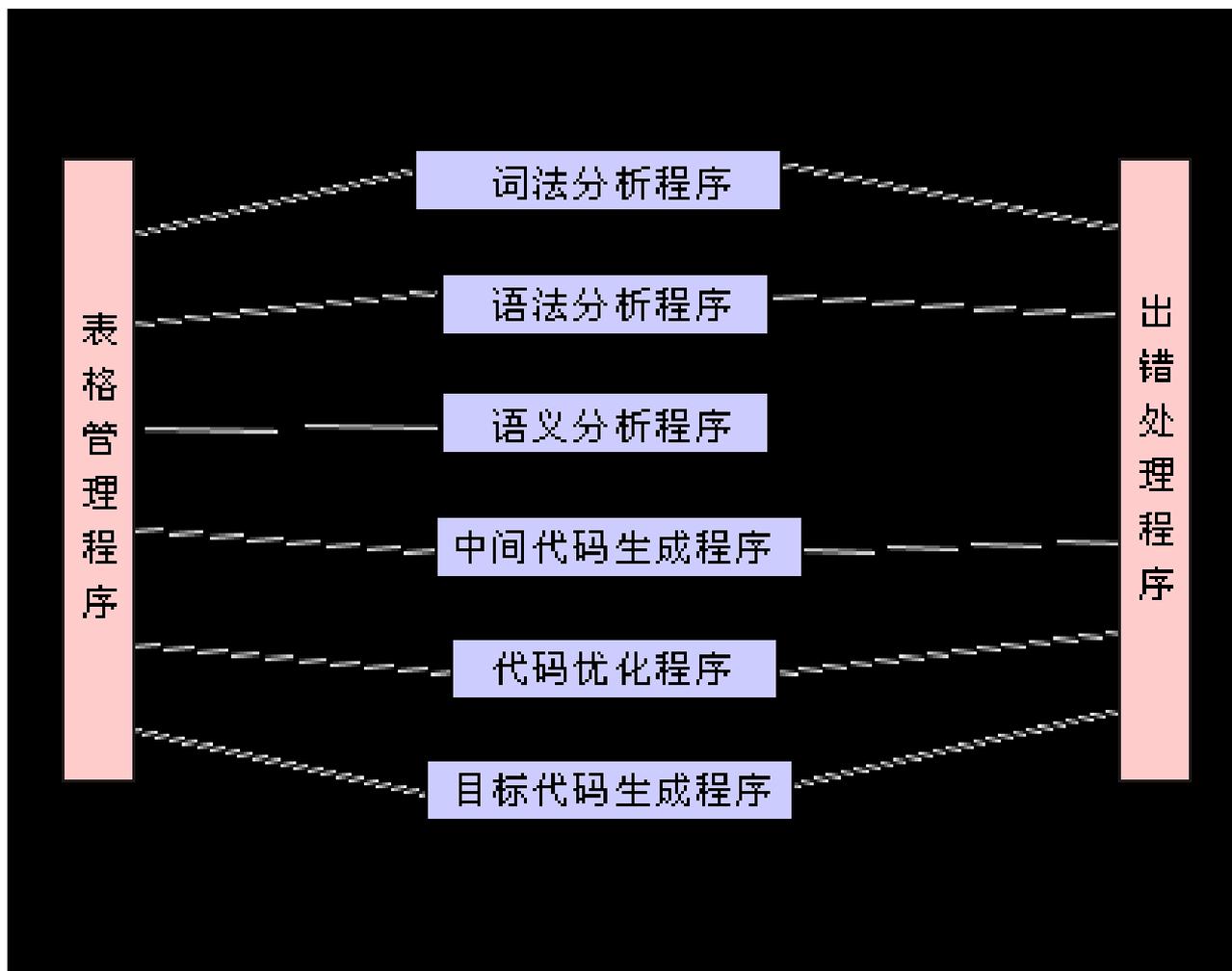


编译程序的结构

- 词法分析程序
- 语法分析程序
- 语义分析程序
- 中间代码生成程序
- 代码优化程序
- 目标代码生成程序
- 符号表管理程序
- 出错管理程序

编译的各个阶段

编译程序的结构框图





编译程序（器）的组织

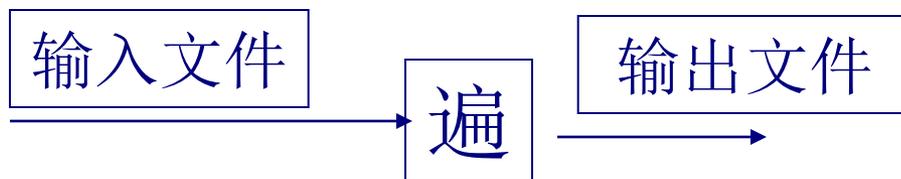
■ 前端和后端



■ 仅依赖源程序

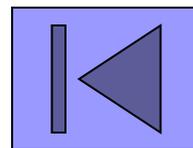
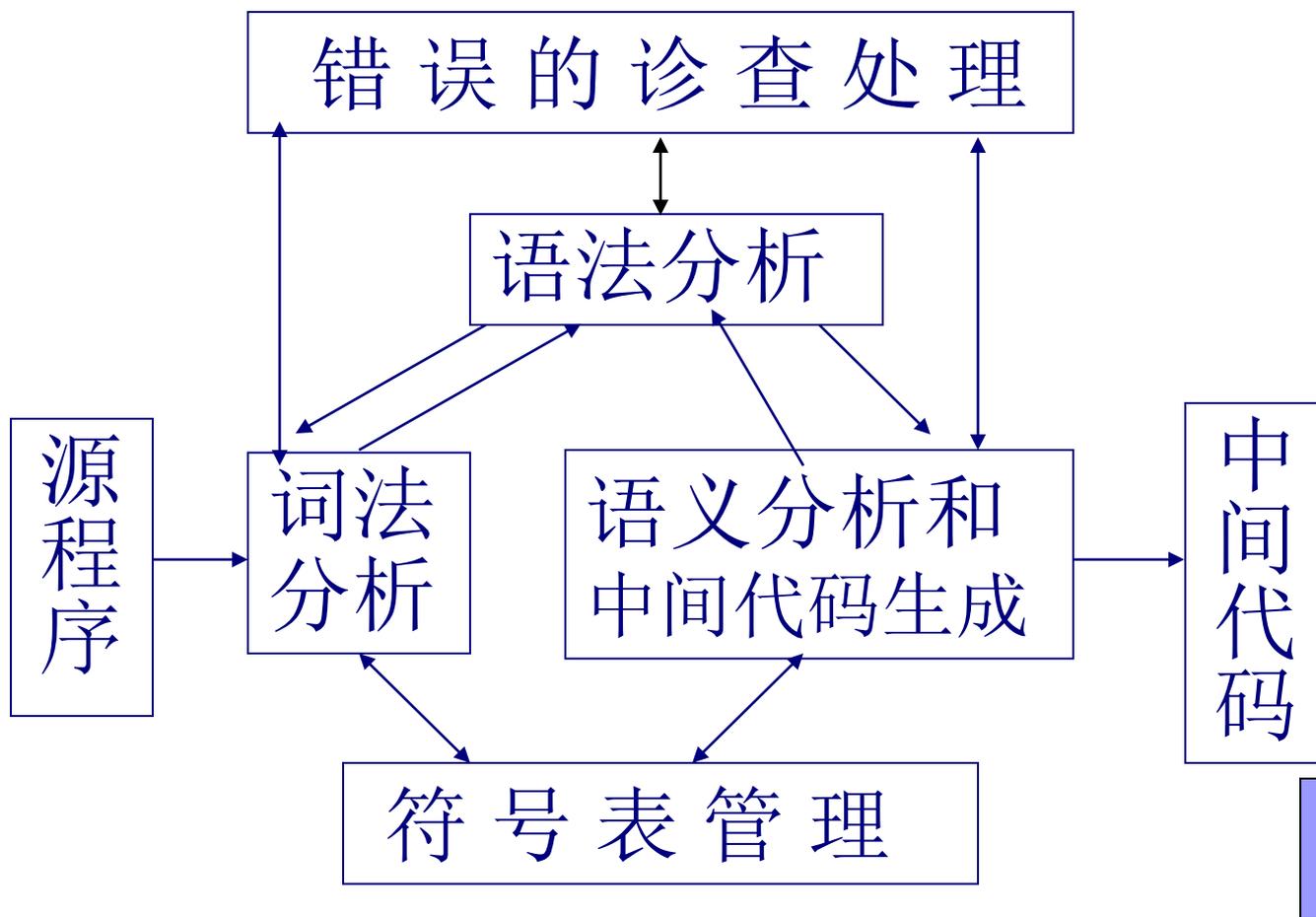
仅依赖目标计算机

■ 遍（PASS）：对输入文件（源程序或其等价的中间形式）从头到尾扫视，完成预定的处理。



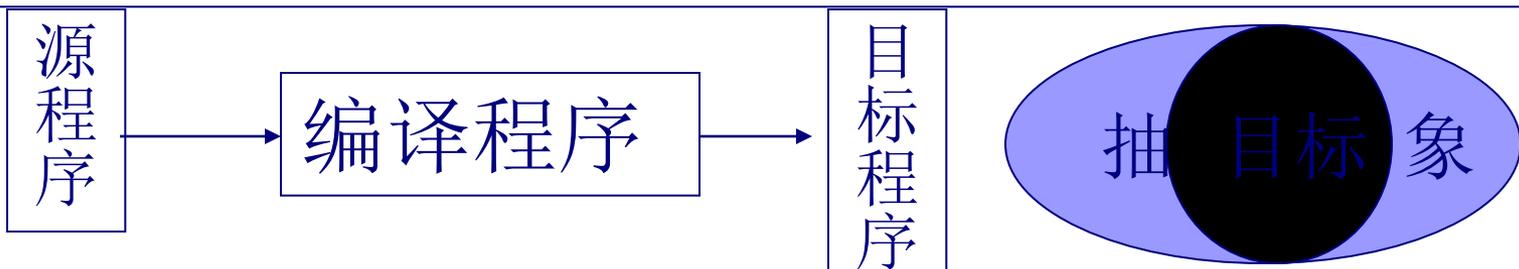


把前端组织成一遍扫描





设计编译程序应首先研究的问题



- 首先研究源程序的语法和语义及运行模型，源是设计编译程序的出发点。
- 研究目标计算机，设计目标代码的指令系统，它是由目标计算机扩充而成，扩充后的计算机称作抽象计算机。目前的通用计算机往往和源语言执行模型不一致。。



编译技术的发展

- 第一个编译程序出现在20世纪50年代早期，多是将算术公式翻译成机器代码
- 20世纪50年代末，提出并研制编译程序的编译程序，
- 20世纪60年代起，出现**自展技术**（用被编译的语言来书写该语言自身的编译程序）。



编译实现方式的发展

■ 手工

- 机器语言
- 汇编语言
- 系统程序设计语言

■ 自动构造工具

- lex
- yacc
- ANTLR



编译技术的应用

- 语言的结构化编辑器
- 语言程序的调试工具
- 预处理程序
- 高级语言之间的转换工具
- 静态代码分析



小结

- 高级程序设计语言通常有解释和翻译两种执行方式。
- 从功能上说,编译程序是一个翻译程序,将高级语言的程序翻译成低级语言的程序。
- 一个编译过程可划分成词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成六个阶段。
- 编译技术会应用在很多领域。



2. 静态代码分析技术介绍

- 2.1 静态代码分析简介
- 2.2 静态代码分析功能
- 2.3 静态代码分析技术
- 2.4 静态代码分析算法



2.1 静态代码分析简介

■ 含义

静态代码分析是指不编译运行程序，只通过对程序源代码进行分析来发现其中的错误。它从语法、语义上理解程序行为，直接分析被测程序特征，寻找可能导致错误的异常。

- 静态代码分析是相对于程序动态分析而言的，这两者的显著区别就是，程序动态分析需要实际执行程序。



- 相比较传统的测试，静态分析有两大优点：

1. 它能检查更多的执行路径。

传统的测试只能检查执行过的代码，但在现实的系统中，测试所能覆盖的路径只占程序中总的路径数的一小部分。测试用例集可能会达到100%的语句覆盖，但大部分的路径还没有执行到。相比之下，静态分析工具可以检查测试用例没有覆盖到的路径，这就是为什么静态分析工具在即便是已经做了很多次测试的软件中仍然可以发现很多的bug。



■ 2. 可以用在开发过程中更早的阶段

攻击者经常利用极端状况下程序的行为来发动攻击，特别是针对测试用例容易遗漏的路径。静态分析工具则更擅长捕获这种漏洞，它可以在开发活动的早期就发现bug，从而节省时间降低成本。根据 **US National Institute of Standards and Technology** 在2002年的一项研究表明，同样一个缺陷，在产品现场发现后修复要比在编码阶段耗费10-30倍的成本，所以发现bug要越早越好——编码完成后，系统测试前——因为越往后发现意味着修复成本越高。



2. 2静态代码分析功能

1. 类型检查

2. 风格检查

3. 程序理解

4. 程序验证和属性检查

5. Bug查找

6. 安全审查



1. 类型检查

- 类型检查主要是指验证变量和表达式的类型是否兼容，检查变量指定的类型是否与赋给它的值相匹配，
- 例6.1 类型检查将不会让整型值赋予给对象变量

```
int i = 0;
```

```
object[] s = new object;
```

```
s = i; /*不允许整型值赋予给对象变量*/
```



- 就像所有的静态代码分析工具和技术一样，类型检查也存在着漏报和误报的问题。
- 例6.2 类型检查的漏报：

```
String[] sa = new String[100];
```

```
sa[0] = "hello world";
```

```
Object[] oa = sa;
```

```
oa[0] = new Integer(1);
```

```
System.out.println(sa[0]);
```

只有只读的**Object[]** 才能是**String[]**的父类型。
但**Java**里并没有只读数组这么个类型，于是错误发生了。



■ 例6.3 类型检查的误报

```
short s = 1;
```

```
int i =s;
```

```
short r =i;
```

本意是想要将一个类型为`int`的表达式赋予给一个类型为`short`的变量，但是却无法通过类型检查。可以通过一个显式的类型转换来解决这个问题



2. 风格检查

- 它主要针对代码规范，与标题、空格、命名、否决函数、注释、程序结构这些东西有关，总的来说，比起类型检查而言，风格检查更加挑剔，也更加表面化。但风格检查并不会指出程序运行时会发生哪种错误，因为它所查出来的错误都是与代码的可读性以及维护性相关的。并且由于编程风格的多样性，风格检查所依据的规则也并不是唯一的，它具有灵活多样性。



- 现在有很多开源和商业的程序都拥有风格检查功能，最古老以及著名的就是**lint**。而对于**java**开发者来说，**PMD**则是一个很好用的工具，它容易添加判定规则，配置自己想要的风格规范



■ 例6.4 PMD的风格检查代码示例

```
import java.util.*;
public class Test {
    public static void main(String[] args) {
        try{
            if(true) {}
            System.out.println("Hello World!");
        } catch(Exception e) {
        }
    }
}
```

PMD会检查出：**catch**块中没有内容、**if**判断块中没有内容、代码中出现**System.out.println**等警告描述



3. 程序理解

- 程序理解最主要的用途在于帮助理解程序，搞懂代码库中的大量代码。它是一个从计算机程序中获取知识信息的过程，这些知识信息可以用于程序排错、增强程序、重用程序和整理文档等工作。
- 在很多集成开发环境（**IDE**）中，都包括了一些初级的程序理解功能，比如：查找本变量的声明和使用位置。更高级一点的还能帮助查找类之间的关联关系等。



4. 程序验证和属性检查

- 程序验证是对源代码进行分析，如果源代码符合预先专门制定的一份描述程序行为的规格说明，那么则说明该程序完好。
- 这里存在的最大的问题就是，程序验证依靠的是预先制定的规格说明，而要想程序验证的结果有足够的说服力，这份规格说明务必需要完整而详细。但是通常没有办法达到这种程度。因为为了编写达到这种程度规格说明书，可能需要花费比编这个程序更多的时间。



- 作为替代的方法，更多的是使用属性检查。
- 属性检查的方法跟程序验证的方法从本质上是一样的，它们的区别在于，属性检查依据的只是描述部分程序行为的部分规格说明，而程序验证依据的是描述所有程序行为的规格说明。



- 目前属性检查多通过以下三种技术来实现程序验证：

(1) 模型检测

模型检测对有限状态的程序构造出状态或有向图等抽象模型，再对模型遍历以验证系统特性。模型检测需要列举所有可能的状态。由于软件本身的高复杂度，对所有程序点进行建模可能会使模型规模庞大，因此一般只针对程序中某一方面的属性构造抽象模型。近期出现的一种模型检测方法是通过对内存状态的建模，使原先主要检测时序相关漏洞的模型检测方法还可用来对内存相关漏洞进行相关检测。



(2) 定理证明

定理证明比模型检测的形式化方法更加严格，通过使用各种判定过程来验证程序抽象公式是否为真。判别的方法取决于公式的形式，如不等式的合取：首先由合取式构造一个图，合取式中每个条件对应于图中的一个节点，然后利用给出的等式将对应的顶点合并，在顶点合并的过程中对合取式的不等式进行检查，如发现存在不成立，则该合取式不可满足。



(3) 符号执行

符号执行的基本思想是将程序中变量的值逻辑转换成抽象符号，模拟路径敏感的程序控制流，通过约束求解的方法检测是否有发生错误的可能。



- 大多数的属性检查工具都将检查的重点放在临时性安全属性上，临时性安全属性规定了一系列在系统中不能发生的有序的事件。
- 大多数的属性检查工具都支持自己定义的临时性安全属性，如下例，用无符号整数识别负数

```
BOOL fun(size_t cbSize)
{
    if(cbSize > 1024)
        rerurn FALSE;
    char *pBuf = new char[cbSize - 1];
    //未对new 的返回值进行检查
    memset(pBuf, 0x90, cbSize - 1);
    .....
    return TRUE;
}
```



- 在上面代码中，在调用**new**分配内存后，程序未对调用结果的正确性进行检测。如果**cbSize**为0的话，则（**cbSize - 1**）为-1。但是**Memset** 中第3个参数本身是无符号数，因此会将-1视为正的**0xffffffff**，函数执行之后程序当然就只有崩溃了。



- 由于属性检查依靠事先制定的规则，因此事先制定的规则越多，属性检查越“健全”。大部分时候会在追求健全性和复杂度之间取一个平衡，因为规则制定的越健全，属性检查越复杂，越有可能出现误报。需要注意的是，属性检查不会出现漏报，这是因为只要制定了相应规则，属性检查就会报告相应错误，而不制定，则不报告。
- 目前关于程序验证的工具具有**Praxis High Integrity Systems**，它是一个专门面向Ada编程语言子集的程序验证工具。



5. Bug查找

- Bug查找是指按照预先制定的一些共同认同的规则，来查找程序中存在的一些bug。这些所谓的“坏代码”也许是效率问题（例如创建不必要的对象），也有可能是安全性或稳定性方面的隐患（例如不恰当的同步、未关闭的文件句柄）。



■ 例6.6 在构造函数中读取未初始化的字段

```
public class Thing {  
    private List actions;  
    public Thing(String startingActions) {  
        StringTokenizer tokenizer = new  
            StringTokenizer(startingActions);  
        while (tokenizer.hasMoreTokens()) {  
            actions.add(tokenizer.nextToken());  
        }  
    }  
}
```

在这个例子中，最后一行将产生一个 `null` 指针异常，因为变量 `actions` 还没有初始化。



■ 例6.7 Null 指针示例

```
1 Person person = aMap.get("bob");  
2 if (person != null) {  
3     person.updateAccessTime();  
4 }  
5 String name = person.getName();
```

如果第 1 行的 Map 不包括一个名为 “bob” 的人，那么在第 5 行询问 person 的名字时就会出现 null 指针异常。



6. 安全审查

- 以安全为中心的静态分析工具使用了很多在其他工具中使用的技术，但是由于它对识别安全问题的关注点的不同，因此意味着会以不同的方式来使用这些技术。
- 静态分析工具总是在漏报和误报上有此消彼长的情况，显然安全工具需要更全面的覆盖率而不是为了减少误报而增加漏报，所以，安全人工审查在对待其分析结果的时候就显得更加有必要。关于安全审查，将在后面再详细的讨论。



2.3 静态代码分析技术

- 词法分析
- 语法解析
- 语法抽象
- 语义分析
- 污染传播分析
- 指针别名分析



1. 词法分析

- 词法分析是编译器实现的第一步。它的任务是把源文件的字符流转换成记号流，即从左到右逐个字符对构成源程序的字符串进行扫描，依据语法规则，识别出一个一个的标记（**token**），把源程序变为等价的标记串序列。例如：`int a = 3 + 5;`经过词法分析会输出 `int,a,=,3,+,5`和`;`这七个单词。然后比较记号流中的标识符和预先定义的安全性漏洞字典，如果匹配就发出警告。例如：一旦发现C源程序中存在`strcpy`、`strcat`等字符串操作函数即认为存在缓冲区溢出这种安全性漏洞，因为这些函数可能引起缓冲区溢出，此时的安全性漏洞字典包含`strcpy`、`strcat`等。



2. 语法解析

- 解析器采用了上下文无关语法（**CFG**）来匹配记号流。这些语法组成了一套用来描述语言中的符号（元素）的规程。

- 例6.8 解析记号流规程举例

`stmt := if_stmt | assign_stmt`

`if_stmt := IF LPAREN expr RPAREN stmt`

`expr := lval`

`assign_stmt := lval EQUAL expr SEMI`

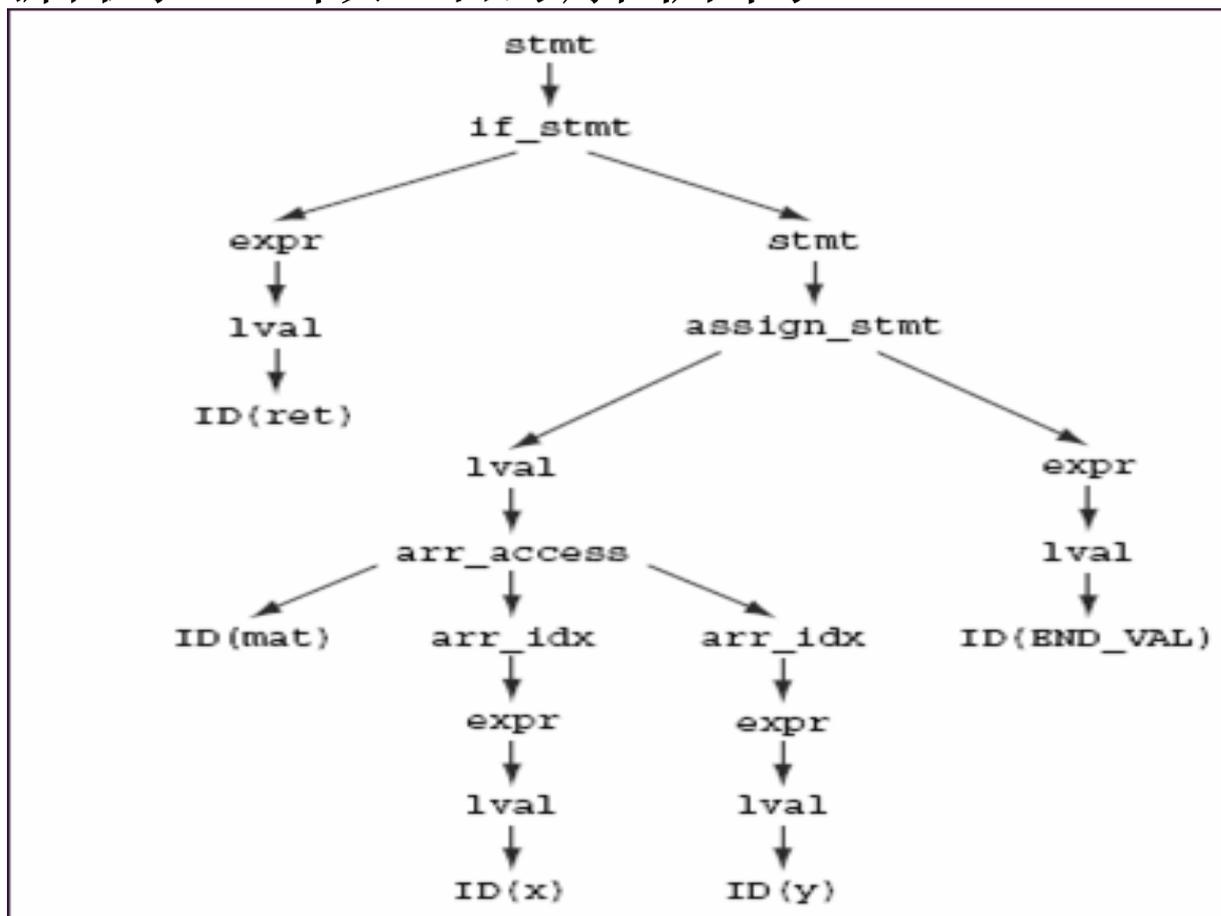
`lval = ID | arr_access`

`arr_access := ID arr_index+`

`arr_idx := LBRACKET expr RBRACKET`



■ 根据例6.8做出的解析树



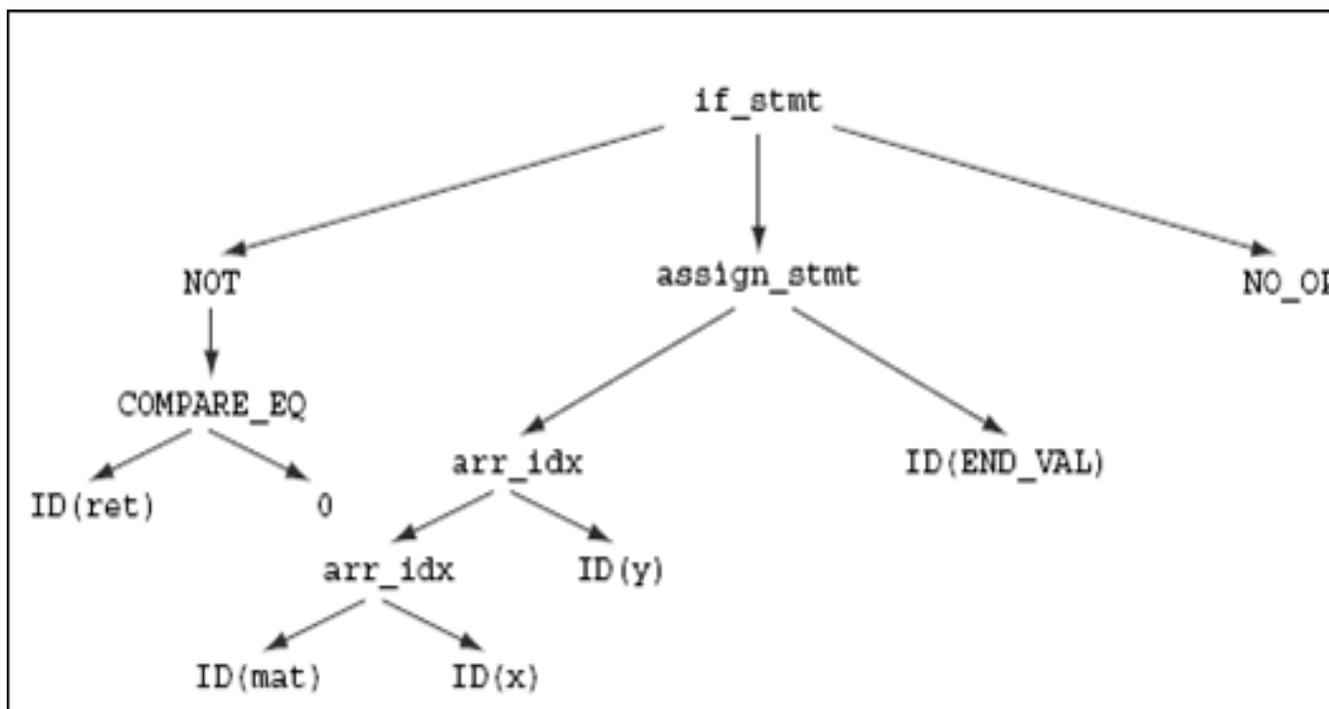


3. 语法抽象

- 通过解析树可以完成很多重要类型的检查和分析，因为解析树是包含程序员初始想法的代码的最直接代表。但是解析树无法完成复杂的分析，因为在解析工程中，经常会对语法进行等价的转换，这样会给语法引入一些多余的成分，对后续阶段造成不利影响，甚至会使各阶段变得混乱。因此，很多编译器（包括GJC）经常要独立地构造解析树，为前、后端建立一个清晰的接口。这时就需要使用抽象语法树（AST）。
- 抽象语法树的结构不依赖于源文件的文法，也就是解析阶段所采用的上下文无关语法，因此它能够为后来的分析提供一个更加标准方便的版本。



■ 对例6.8所做的抽象语法树





4. 语义分析

- 源程序如果通过了词法分析和解析，就表明该源程序书写正确、符合程序语言所规定的语法。但语法分析并未对程序内部的逻辑含义加以分析，因此接着进行语义分析。
- 语义分析，即审查每个语法成分的静态语义。如果静态语义正确，则生成与该语言成分等效的中间代码，或直接生成目标代码。



- 静态语义检查涉及到以下几个方面：
 - (1) 类型检查，如运算操作数的类型应相容。
 - (2) 控制流检查，用以保证控制语句有合法的转向点。如C语言中不允许goto语句转入case语句流；break语句需寻找包含它的最小switch、while或for语句方可找到转向点。
 - (3) 一致性检查，如在相同作用域中标识符只能说明一次、case语句的标号不能相同等。



- 基于语法和简单语义分析的安全性检查的工作原理非常类似于编译器系统，它以语法分析和语义规则为基础，同时加入简单的控制流分析和数据流分析。因此这种方法具有较高的分析效率和可扩展性，并且可以通过向程序中加入面向对象程序切片中的数据流分析注释信息的方式发现软件中广泛存在的安全性漏洞，如程序中出现机率最多的内存访问漏洞，包括存储区的非法使用、空指针的引用、缓冲区溢出等等。它的另一个优点是可适用于对大规模程序的分析。



5. 污染传播

- 通过污染传播测试能知道攻击者可能潜在地控制程序中的哪些值，确定污染数据的来源，找出所有外部数据进入程序的入口代码以及它在程序中是如何移动的。
- 污染传播是大多数输入验证和其他代表性缺陷产生的关键。例如，包含了一个可被利用的缓冲区溢出漏洞的程序几乎都有一个从输入函数到一个脆弱操作的数据流路径。



实例

- 实例 1:下面的代码动态构建和执行一个SQL查询，查找与给定名称匹配的item。查询限定只有当当前用户名与item的所有者名称匹配时，才向当前用户显示item。

```
...  
String userName = tx.getAuthenticatedUserName();  
String itemName = request.getParameter("itemName");  
String query = "SELECT * FROM items WHERE owner = '"  
+ userName + "' AND itemname = '" + itemName + "'";  
ResultSet rs = stmt.execute(query);  
...  
...
```

6. 指针别名分析

- 指针别名分析是另一个问题数据流问题。别名分析的目的是要了解哪些指针可能是指向相同的内存位置。别名分析算法用诸如“必须的别名”，“可能的别名”和“不能的别名”来描述指针的关系。许多编译器优化都需要某种形式的别名分析的正确性。
- 例如，只有当指针p1和p2不指向内存的相同位置时，编译器才会记录下面两个声明：

```
*p1 = 1;
```

```
*p2 = 2;
```



- 对于安全工具来说，别名分析在执行污染传播测试方面很重要。一个流动的敏感的污点-跟踪算法需要执行别名分析，以了解下面代码中从 `getUserInput()` 到 `processInput()` 的数据流：

```
p1 = p2;
```

```
*p1 = getUserInput();
```

```
processInput(*p2);
```

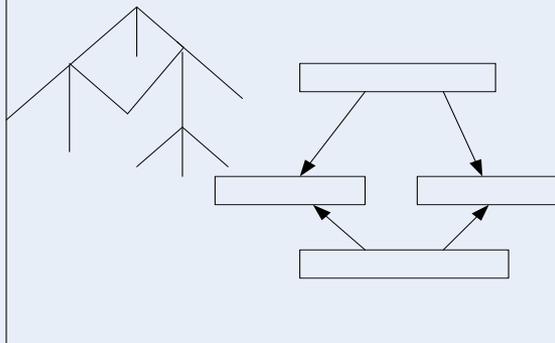


- 使用静态分析算法的目的是要提高上下文的敏感性（以确定一段特殊代码运行的环境和条件），同时更好的上下文敏感性，能够更好地评估代码的危险性。
- 任何优秀的分析策略都至少包括两个组成部分：分析每个单独的函数的程序内分析（本地分析）和分析函数之间的关系的程序间分析（全局分析）。下图显示了这两个分析的结构组成以及每个分析中使用的数据的关系。

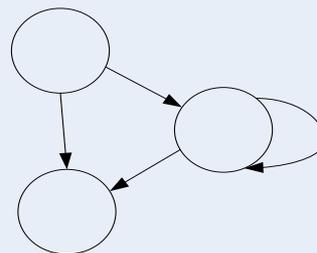


分析算法

本地分析
抽象语法树
(AST)
控制流图表



全局分析
调用图表





2.4 静态代码分析算法

算法1：断言检查

- Floyd提出的"用断言式方法"证明程序的正确性的方法,是通过一个断言发现工具从程序中发现该程序断言,然后与程序要求满足的断言条件比较,从而判断其正确性。该工具在复杂条件下对程序正确性判断和大量重复程序检测上能发挥重要的作用。



算法2：本地分析

- 这部分的分析主要针对于分支和循环，所以要求静态分析方法不必那么精确，但是一定要可靠。本地分析包括以下几个方面：
 - (1) 抽象解释
 - (2) 谓词转换器
 - (3) 模型检验



算法3：全局分析

- 全局分析的作用是用来分析程序中各个函数之间的关系。它是一种封闭式环境下的假设，即对整个程序来进行检测。全局分析被采用的实质原因有两个方面：首先，它可以保证汇编系统的一致性；其次，它通过提供自动优化，减轻了程序员的负担。
- 它的做法如下：将整个系统,按可重用的程度划分为若干个不同的基本包。它表示了系统的大的逻辑框架,然后测试包与包之间的联系的安全性。



3 常用静态代码分析工具介绍

3.1 PMD

3.2 PC-Lint

3.3 CheckStyle

3.4 Jtest

3.5 其他静态分析工具简介



3.1 PMD

1.PMD简介

- **PMD**是一种静态分析**Java**代码错误的工具。**PMD**附带了许多可以直接使用的规则，利用这些规则可以找出**Java**源程序的许多问题，例如没有用到的变量、多余的变量创建操作、空的**catch**块等等。此外，用户还可以自己定义规则，检查**Java**代码是否符合某些特定的编码规范。例如，可以编写一个规则，要求**PMD**找出所有创建**Thread**和**Socket**对象的操作。



- 如果要在一个Java源代码目录中运行PMD，只需直接在命令行上运行下面的命令：

```
c:\data\pmd\pmd>java -jar lib\pmd-1.02.jar
```

```
c:\j2sdk1.4.1_01\src\java\util
```

```
text rulesets/unusedcode.xml
```

- 输出结果类如：

```
c:\j2sdk1.4.1_01\src\java\util\AbstractMap.java
```

```
650    Avoid unused local variables such as 'v'
```

```
c:\j2sdk1.4.1_01\src\java\util\Date.java      438
```

```
Avoid unused local variables such as 'millis'
```



- 除了直接在命令行上运行PMD之外，还可以通过Ant、Maven或者各种集成开发环境（IDE）运行PMD，例如jEdit、Netbeans、Eclipse、Emacs、IDEAJ和JBuilder等。



工作原理

- PMD的核心是JavaCC解析器生成器。
PMD结合运用JavaCC和EBNF（扩展巴科斯-诺尔范式，Extended Backus-Naur Formal）语法，再加上JJTree，把Java源代码解析成抽象语法树（AST，Abstract Syntax Tree）。





下面是一段简单的Java代码以及与之对应的AST

■ Java源代码:

```
public class Foo {  
    public void bar() {  
        System.out.println("hello world");  
    }  
}
```



- 对应的抽象语法树

CompilationUnit

TypeDeclaration

ClassDeclaration

UnmodifiedClassDeclaration

ClassBody

ClassBodyDeclaration

MethodDeclaration

ResultType

MethodDeclarator

FormalParameters



接上页 Block

BlockStatement

Statement

StatementExpression

PrimaryExpression

PrimaryPrefix

Name

PrimarySuffix

Arguments

ArgumentList

Expression

PrimaryExpression

PrimaryPrefix

Literal



3.2 PC-Lint

1. PC-Lint简介

- PC-Lint是一个历史悠久，功能异常强劲的静态代码检测工具，它不但能够监测出许多语法逻辑上的隐患，而且也能够有效地提出许多程序在空间利用、运行效率上的改进点
- PC-Lint能够识别并报告C语言中的编程陷阱和格式缺陷的发生。它进行程序的全局分析，能识别没有被适当检验的数组下标，报告未被初始化的变量，警告使用空指针，冗余的代码等等



2.PC-Lint的功能

- 1) **PC-Lint**是一种静态代码检测工具，也是一种更加严格的编译器
- 2) **PC-lint**不但可以检测单个文件，也可以从整个项目的角度来检测问题
- 3) **PC-Lint**支持几乎所有流行的编辑环境和编译器
- 4) **PC-Lint**支持**Scott Meyes**的名著（**Effective C++/More Effective C++**）中所描述的各种提高效率 and 防止错误的方法



3.PC-Lint的使用

- 第一步:安装&设置
- 第二步:整合PC-Lint到选定的编译环境
- 第三步:Lint单个C文件
- 第四步:Lint多个C文件



3.3 CheckStyle

1. CheckStyle简介

- **CheckStyle**是目前最广泛使用的代码检查工具,功能强大,操作简单,可以和**Ant**结合使用,最重要的是它是开源的。
- **Checkstyle**已经成了加强编码规范的首选工具。**NtCheckstyle** 是一个可安装的模块,它自动完成**Checkstyle**和**Netbeans** 环境的集成。集成后**checkstyle**的使用非常的方便。



2. 安装及使用

- 可以在Netbeans中直接安装NbCheckstyle。
- 安装方法是：
 - (1) 选择菜单“Tools/Update Center”。
 - (2) 选择“install manually downloaded module”单选按钮。
 - (3) 按照向导添加NbCheckstyle 模块，然后选择
“install”



3. CheckStyle的结果输出

- Checkstyle 工具中使用的规范主要参考sun公司的编码规范，缺省值均是最常用的情况。但是，也可以根据公司的需要或开发人员的习惯方便地定制适合自己团队的规范。可以在netbeans中options窗口修改相应的设置。
- Checkstyle 会将检查的结果输出到netbeans的标准窗口中



3.4 Jtest

1. Jtest简介

- **Jtest**是parasoft公司推出的一款针对java语言的自动化白盒测试工具,它通过自动实现java的单元测试和代码标准校验,来提高代码的可靠性。
- **Jtest**先分析每个java类,然后自动生成junit测试用例并执行用例,从而实现代码的最大覆盖,并将代码运行时未处理的异常暴露出来;另外,它还可以检查以**DbC (Design by Contract)**规范开发的代码的正确性。



- 用户可以通过扩展测试用例的自动生成器来添加更多的junit用例。
- **Jtest**还能按照现有的超过**350**个编码标准来检查并自动纠正大多数常见的编码规则上的偏差，用户可自定义这些标准，通过简单的几个点击，就能预防类似于未处理异常、函数错误、内存泄漏、性能问题、安全隐患这样的代码问题。



2.Jtest的优势和特征

■ 优势:

- 1) 使预防代码错误成为可能，从而大大节约成本，提高软件质量和开发效率
- 2) 使单元测试——包括白盒、黑盒以及回归测试成为可能
- 3) 使代码规范检查和自动纠正成为可能
- 4) 鼓励开发团队横向协作来预防代码错误



■ 特征:

- 1) 通过简单的点击, 自动实现代码基本错误的预防, 这包括单元测试和代码规范的检查
- 2) 生成并执行junit单元测试用例, 对代码进行即时检查
- 3) 提供了进行黑盒测试、模型测试和系统测试的快速途径
- 4) 确认并阻止代码中不可捕获的异常、函数错误、内存泄漏、性能问题、安全弱点的问题
- 5) 监视测试的覆盖范围
- 6) 自动执行回归测试
- 7) 支持DbC编码规范



■ 特征:

- 8) 检验超过350个来自java专家的开发规范
- 9) 自动纠正违反超过160个编码规范的错误
- 10) 允许用户通过图形方式或自动创建方式来自定义编码规范
- 11) 支持大型团队开发中测试设置和测试文件的共享
- 12) 实现和IBM Websphere Studio /Eclipse IDE 的安全集成



3.5 其他静态分析工具简介

1. BOON

- BOON是一个用来检查C语言源代码缓冲区溢出漏洞的静态代码分析工具。
- BOON可以发现许多其他分析工具所不能发现的错误，但是同时，它也对一些漏洞没有办法，比如：**statement order, pointer aliasing**,程序间的交互依赖。



BOON规则

Code	Interpretation
Char s [n]	alloc(s) = n
Strlen(s)	Len(n)-1
Strcpy(dst,src)	Len(src) <=len(dst)
S = "foo"	4 len(s) 4 alloc(s)
P = malloc(n)	Alloc(p) = n
Strcat(s,suffix)	Len(s) +len(suffix)- 1<=alloc(s)
Strncat(s,suffix,n)	Len(s)+min(len(suffix)- 1,n) <= alloc(s)



2.MOPS

■ 模型检验工具的典型MOPS

（**modelchecking programs for security properties**）能够发现程序中存在复杂语义上的错误，进而准确发现程序中潜在的安全性漏洞，它以程序控制流为输入，绘制系统状态机，与安全数据库中规则作比较分析。



3.ITS4

- **ITS4** 是一种用于静态检测**C**和**C++**源代码安全漏洞的工具。同其它类似技术相比，**ITS4** 的准确性更高，能够在编程过程中把检测结果实时反馈给开发人员。
- **ITS4**同时能够轻松的支持**C++**代码的检测，还支持命令行格式，可运行于**Windows**和**Unix**平台。**ITS4**在**C**或**C++**源代码中寻找危险的函数调用。对于某些调用，**ITS4**会加以分析以确定其危险程度。



- ITS4会提供分析报告，包括漏洞的简单描述和改进方法。

ITS4维护的安全漏洞词典为如下形式：

函数	严重性	解决方案
Gets	最危险	使用 fgets(buf,sizestdin)
Strcpy	很危险	使用 strncpy
Strcat	很危险	使用 strncat
Sprintf	很危险	使用 snprintf ，或者使用精度说明符
Scanf	很危险	使用精度说明符或者自己进行解析
Strtrns	危险	用手工检查来查看目的地大小是否至少与原字符串相等
realpath	危险	分配缓冲区大小为 MAXPATHLEN ,同样保证输入参数不超过 MATHPATHLEN



4.RATS

- **RATS**是用于C、C++、Python、Perl和PHP代码的安全审计工具。它能够对源代码进行扫描，找出潜在的危险函数调用。该工具的最终目标并不是找出代码漏洞，而是为人工安全审计提供一个方便合理的起点。
- **RATS**结合了ITS4的静态检查技术和MOPS的深度语义分析技术来检查缓冲区溢出漏洞。**RATS**遵守**GPL**。同ITS4相比，**RATS**可以对整个工程代码进行检查，而不是单一文件。同时，**RATS**还可以检查数组的边界。



5.Prefix

- **Prefix**使用符号执行及约束求解的方法对C/C++程序进行静态分析测试。
- **Prefix**的工作流程是：首先分析源代码，将其转换为抽象的语法树，然后对过程依照调用关系进行拓扑排序，再为每个过程生成相应的抽象模型，最后静态模拟执行路径并用约束求解的方法对约束集合进行检测。
- 为了解决路径空间爆炸的问题，**prefix**选择了一定数量具有代表性的路径进行分析。用**prefix**对**apache**，**mozilla**等程序进行检查，发现了其中存在的数百个错误和安全隐患。



6.BANE

- **BANE**是一个用于构造程序分析工具的工具集，它提供了类型推导的接口并内嵌了多种不同的约束求解器。**Cqual**是以**BANE**为基础构造的分析检查工具。用**Cqual**对**Linux**内核进行分析，发现了其中关于加锁动作的错误。



4. 源代码审查实施步骤与策略

4.1 代码审查及实施步骤

4.2 集成代码审查到软件开发

4.3 静态分析工具结果管理



4.1 代码审查及实施步骤

- 源代码审查(**code review**)是软件开发过程的一个阶段，在这个阶段中，代码创造者和审查人员，可能还有质量保证(QA)测试人员，一起进行代码审查。
- 能在该阶段中就找出并更正存在的错误，相对来说比较合理，因为如果在开发软件后面的阶段或者软件交付给用户后才来处理、查找和修改程序缺陷的话，会花费更多的成本。



- 审查人员需要仔细检查的代码问题包括：
 1. 缺陷或者潜在缺陷
 2. 和整个程序设计的一致性
 3. 注释的质量
 4. 遵守编码规范情况



■ 常见的成熟的代码审查的过程：

(1)代码编写者和代码审核者坐在一起，由代码编写者讲解自己负责的代码和相关逻辑。

(2)代码审核者在此过程中可以随时提出自己的疑问，同时积极发现隐藏的bug;对这些bug记录在案。

(3)代码讲解完毕后，代码审核者给自己安排几个小时再对代码审核一遍。代码需要一行一行静下心来。同时代码又要全面的看，以确保代码整体上设计优良。



■ 常见的成熟的代码审查的过程：

(4)代码审核者根据审核的结果编写“代码审核报告”，“审核报告”中记录发现的问题及修改建议，然后把“审核报告”发送给相关人员。

(5)代码编写者根据“代码审核报告”给出的修改意见，修改好代码，有不清楚的地方可积极向代码审核者提出。

(6)代码编写者在修补bug完毕之后给出反馈。

(7)代码审核者把审查中发现的有价值的问题更新到“代码审核规范”的文档中，对于特别值得提醒的问题可群发email给所有技术人员。



- 按照审查内容侧重点的不同，代码审查可以分为三类：

第一类：主要审查代码风格、程序逻辑或者某个特定的目标，比如安全。

第二类：以审查代码风格为主的审查，适合周期性举行。

第三类：以审查程序逻辑为主的审查，一般在设计之初，实现中，结尾时各做一次。

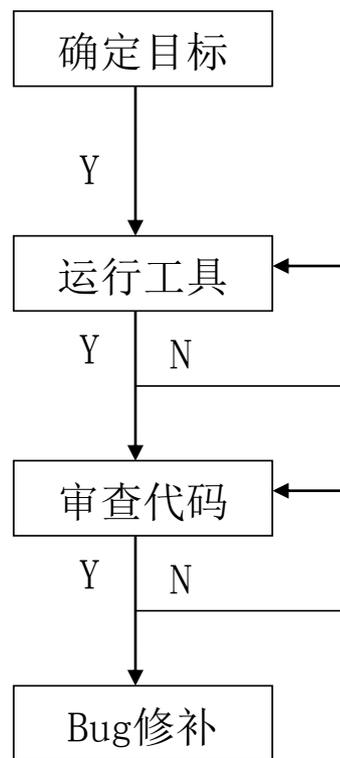


- **Fortify**案例：
- 一个例子是一个代码审查团队在采用工具之前，每年审查1千万行代码。在使用了**Fortify**的工具之后，可以每年审查2千万行——3千万行代码。
- 另一个例子是负责在程序变成产品之前，对所有面向**Internet**的应用程序进行审查。在过去，执行一次审查要花掉这个团队**3-4**周的时间。使用静态分析工具之后，这个安全团队现在将审查控制在**1-2**周的时间。



- 当将静态代码分析工具纳入到代码审查中时，可以将代码审查过程大致分为四个阶段（代码审查周期）：

- （1）确定目标
- （2）运行静态分析工具
- （3）审查代码（使用静态分析工具的输出）
- （4）Bug修补





(1)确定目标

- 在代码审查之前要制定出一组完善的安全目标，这些目标应该来自于对软件风险的评估
- 明确为什么进行代码审查，不能毫无目的或者随波逐流
- 要确保审查人员了解所要审查的代码的功能和用途



(2)运行静态分析工具

- 首先需要围绕审查目标对工具进行配置
- 默认的情况下，静态代码分析不会“主动”知道代码中哪些是违规的或者是违反程序编码意愿的，这样的话，就需要根据实际环境和情况来定制规则



(3)审查代码

- 人工审查代码的目的是对静态分析结果进行验证
- 通常在审查非自己编写的代码时，需要与代码编写者合作来进行代码审查
- 代码审查的结果可以用很多种形式展现，但无论是什么形式，都需要确保能够得到永久保存，以便在下一次代码审查期间可以使用这些结果



(4)Bug修补

- 一般bug修补是属于开发人员的事情，所以要尽早的明白以下两点，否则这有可能会影响bug修补：
 - (1) 安全是开发人员的事情，代码审查人员只负责代码审查
 - (2) 尽量让安全分析结果能被开发人员读懂。审查人员应该以一种尽可能容易理解的方式来编写这份结果报告，并培训开发人员以使得他们能读懂报告



4.2 集成代码审查到软件开发

- 好的代码审查都是开始于项目初期，与软件开发集成在一起，这是为了避免项目到了后期，所有的问题集结到一起，容易使问题的复杂度和解决问题的难度成倍的增加
- 一个优秀的静态分析工具能不能取得好的效果，主要取决于三个前提条件：
 - (1) 谁来用这个工具
 - (2) 什么时候用这个工具
 - (3) 分析结果怎么样



(1) 谁来用这个工具

- 代码审查人员
- 开发人员
- 以上两类人员



(2) 什么时候用这个工具

- 代码编写期间
- 程序生成时
- 到达里程碑时



(3) 分析结果怎么样

■ 分析结果会影响软件发布

- 通常在项目的里程碑到来之时，安全团队将分析工具的输出作为检查点的一部分进行处理，并对其进行优先级排序。开发团队收到排序结果以及安全团队关于应进行修补部分的建议之后，将决定哪些问题可以解决，哪些问题归类为“可接受的风险”。安全团队应对开发团队的决定进行审查，并且对于那些看上去开发团队所承担的比本应承担的风险要高的地方，应逐步加强用例。很显然，如果这种审查会妨碍项目按照预期实现里程碑的目标，那么，程序自然无法按照预定的时间变成产品。



4.3 静态分析工具结果管理

- 当静态分析工具输出结果之后，对比审查目标，如果能以一种灵活的方式对结果进行分组和分类，那么这将非常有助于消除大量的非预期的结果，以便于提高审查的效率
- 除了依照类型对结果分组之外，通常还会将结果以风险级别进行一个分类，例如：**Hot**（热点）、**Warning**（警告）、**Info**（提示）和**All**（所有）
- 现在的高级静态代码分析工具都支持对结果进行剔除，如果没有事先制定好规则来消除不需要的结果，那么事后也可以对结果进行消除以使得审查人员不需要再对非预期的结果进行审查



■ 本节结束