



# 第一章 引论

**重点:** 教学目的, 教学要求, 学习方法, 课程的基本内容, 编译系统的结构, 编译程序的生成。

**难点:** 编译程序的生成。





# 第1章 引论

---

1.1 程序设计语言

1.2 程序设计语言的翻译

1.3 编译程序的总体结构

1.4 编译程序的组织

1.5 编译程序的生成

1.6 本章小结

**assume cs:code, ds:data  
data segment**

**dw 1234h,5678h**

**data ends**

**code segment**

**start: mov ax, data**

**mov ds, ax**

**mov ax, ds:[0]**

**mov bx, ds:[2]**

**mov cx, 0**

**add cx, ax**

**add cx, bx**

**mov cx, ds:[4]**

**mov ax, 4c00h**

**int 21h**

**code ends**

**end start**

**int main**

**{**

**int a,b,c;**

**a=1234h;**

**b=5678h;**

**c=a+b;**

**return 0;**

**}**

**A10**

**0000 1011 0001 1110 0000 0010**

**0000 0000 (8B1E0200)**

**0111 1001 0000 0000 0000 0000**

**B90000)**

**0000 0011 1100 1000 (03C8)**

**0000 0011 1100 1011 (03CB)**

**0000 1011 0000 1110 0000 0100**

**0000 0000 (8B0E0400)**

**1011 1000 0000 0000 0100 1100**

**(B8004C)**

**1100 1101 0010 0001 (CD21)**

■ 如UNIX上的sh



# 程序设计语言的分类

---

- **强制式（命令式）语言(Imperative Language)**
  - 通过指明一系列可执行的运算及运算的次序来描述计算过程的语言；
  - **FORTRAN(段结构)、BASIC、Pascal(嵌套结构)、C.....**
  - 程序的层次性和抽象性不高



# 程序设计语言的分类

---

- **申述式语言 (Declarative Language)**
  - 着重描述要处理什么，而非如何处理的非命令式语言
  - **函数(应用)式语言(Functional Language)**
    - 基本运算单位是函数，如LISP、ML.....
  - **逻辑式(基于规则)语言(Logical Language)**
    - 基本运算单位是谓词，如Prolog, Yacc.....



# 程序设计语言的分类

---

- 面向对象语言(Object-Oriented Language)
  - 以对象为核心，如Smalltalk、C++、Java、Ada(程序包).....
  - 具有识认性（对象）、类别性（类）、多态性和继承性

## 1.2 程序设计语言的翻译

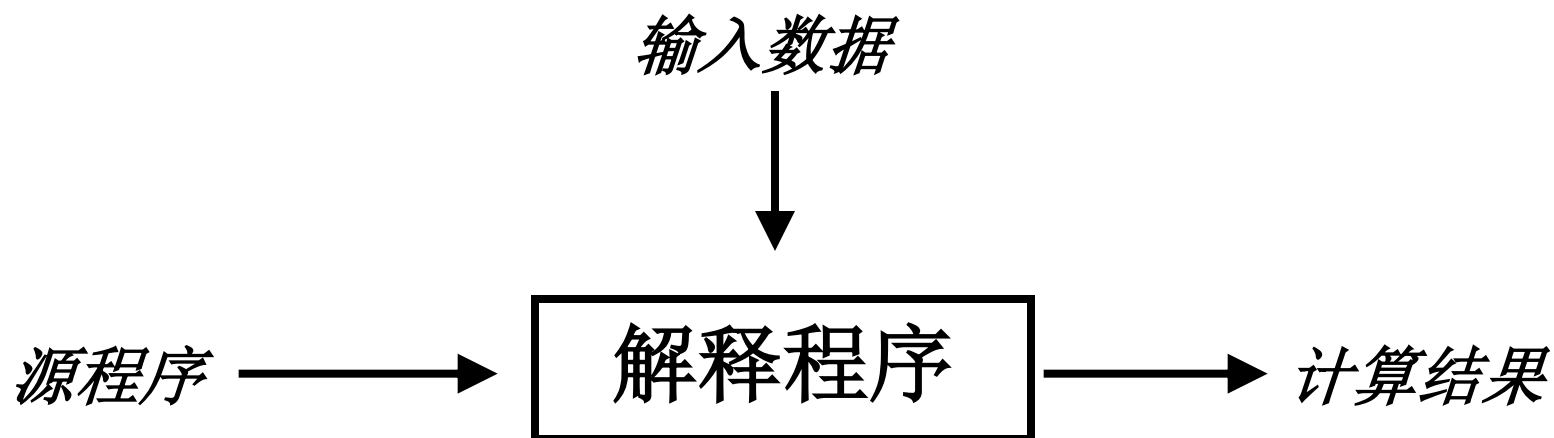
### ■ 翻译程序(Translator)

将某一种语言描述的程序(源程序——**Source Program**)翻译成等价的另一种语言描述的程序(目标程序——**Object Program**)的程序。



## 1.2 程序设计语言的翻译

- 解释程序(Interpreter)
  - 一边解释一边执行的翻译程序
  - 口译与笔译（单句提交与整篇提交）





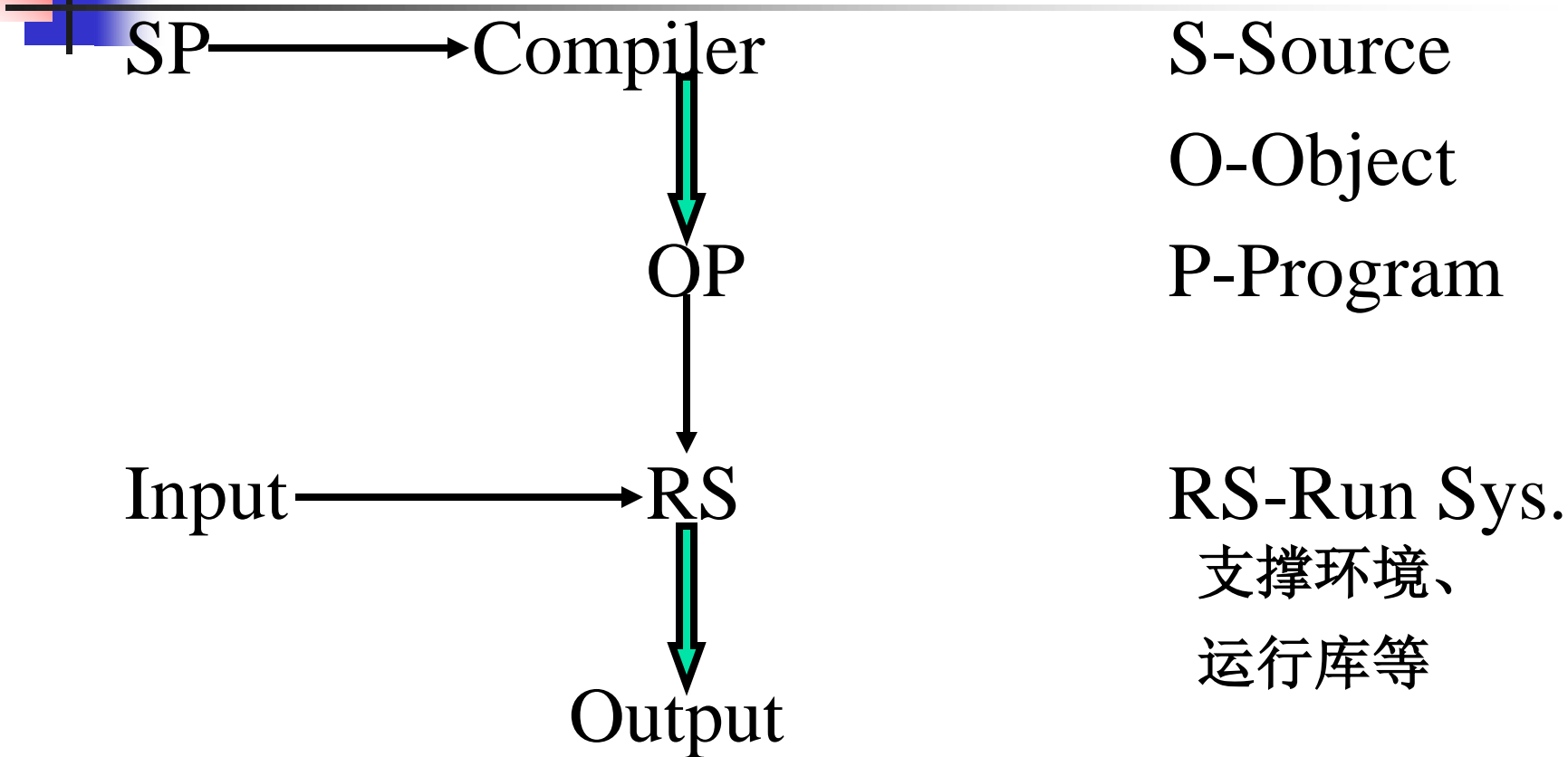
# 1.2 程序设计语言的翻译

## ■ 编译程序(Compiler)

- 将源程序完整地转换成机器语言程序或汇编语言程序，然后再处理、执行的翻译程序
- 高级语言程序→汇编/机器语言程序



# 1.2 程序设计语言的翻译



## ■ 编译系统(Compiling System)

◆ 编译系统=编译程序+运行系统



# 1.2 程序设计语言的翻译

- 其它翻译程序：
  - 汇编程序(Assembler)
  - 交叉汇编程序(Cross Assembler)
  - 反汇编程序 (Disassembler)
  - 交叉编译程序 (Cross Compiler)
  - 反编译程序 (Decompiler)
  - 可变目标编译程序 (Retargetable Compiler)
  - 并行编译程序 (Parallelizing Compiler)
  - 诊断编译程序 (Diagnostic Compiler)
  - 优化编译程序 (Optimizing Compiler)

# 1.2 程序设计语言的翻译—汇总

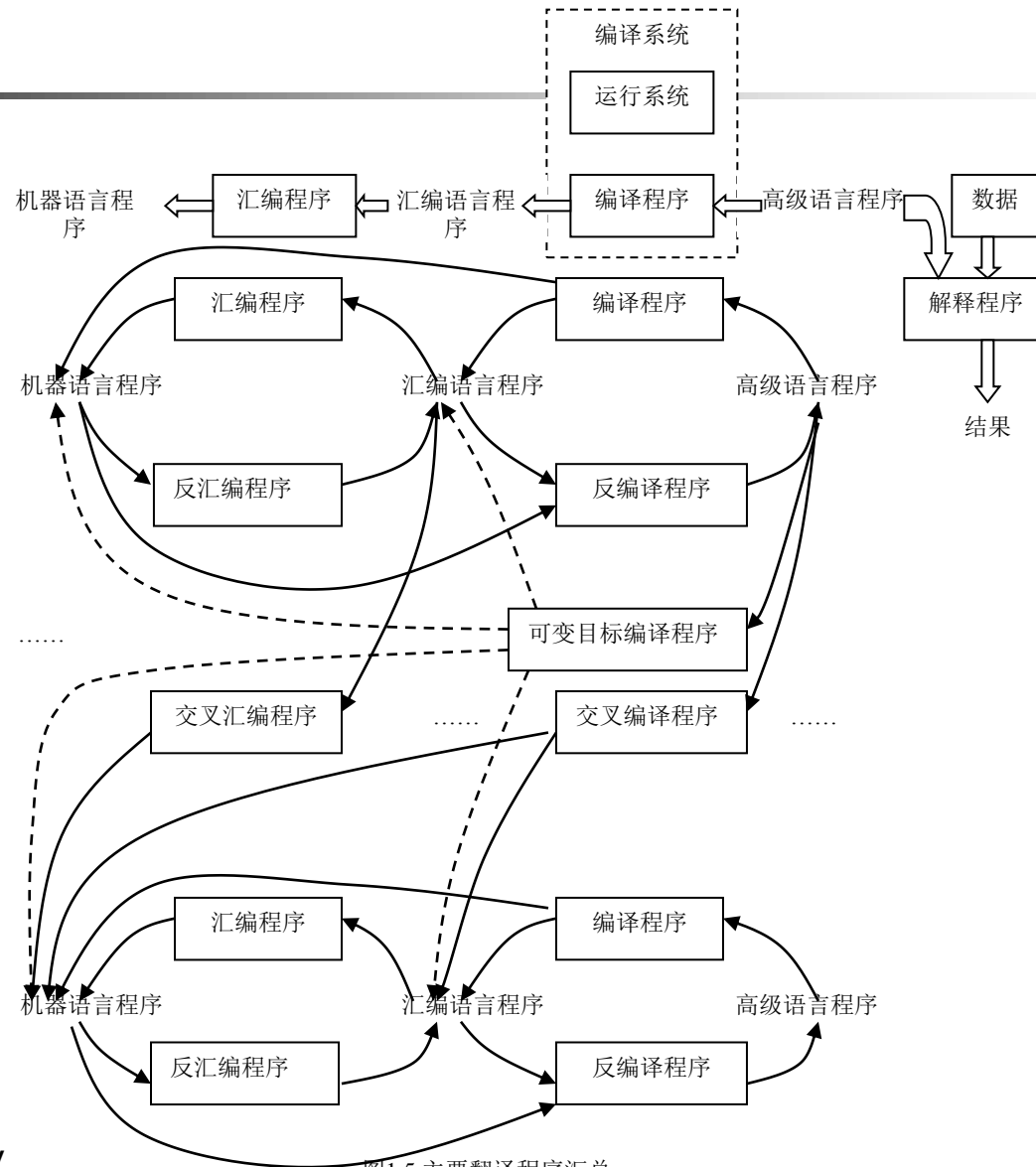


图1.5 主要翻译程序汇总

# 1.3 编译程序总体结构

编译程序的工作，从输入源程序开始，到输出目标程序结束，与自然语言之间的翻译有很多相似之处。

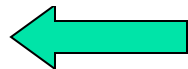
一段英文翻译成中文，

构成编译程序各个阶段

译程序

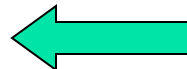
需经下列步骤：Main(){printf("hello")}

识别出句子中的单词



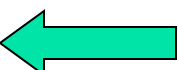
词法分析

分析句子的语法结构



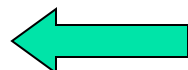
语法分析

根据句子的含义进行初步分析



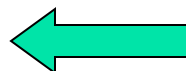
语义分析及中间代码生成

对译文进行修饰



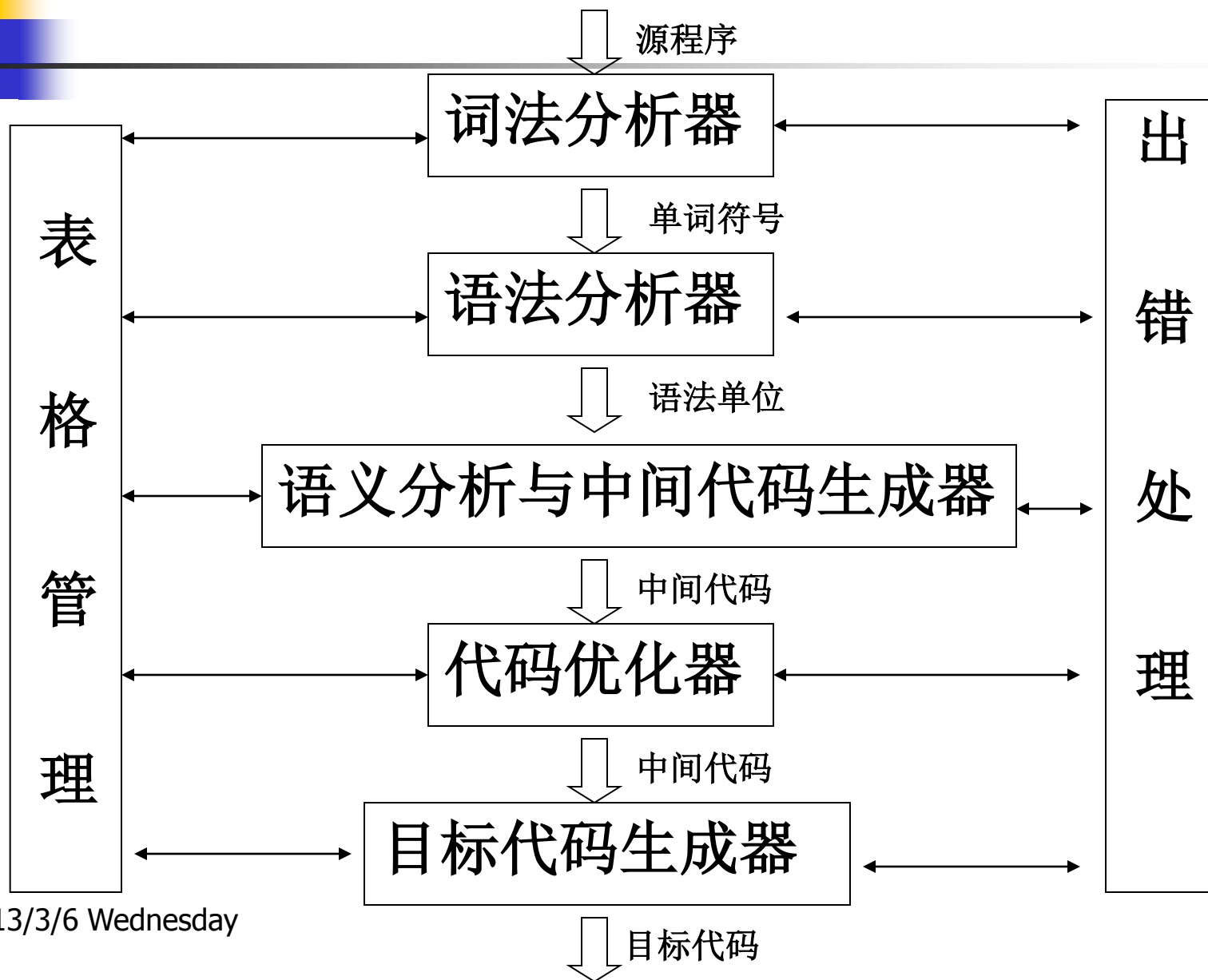
代码优化

写出最后的译文



目标代码生成

# 1.3 编译程序总体结构





# 1、词法分析

■ 例:

**sum=(10+20)\*(num+square);**

## 结果

- (标识符, **sum**)
- (赋值号, **=**)
- (左括号, **(**)
- (整常数, **10**)
- (加号, **+**)
- (整常数, **20**)
- (右括号, **)**)
- (乘号, **\***)
- (左括号, **(**)
- (标识符, **num**)
- (加号, **+**)
- (标识符, **square**)
- (右括号, **)**)
- (分号, **;**)



# 1、词法分析

- 词法分析由词法分析器(Lexical Analyzer)完成，词法分析器又称为扫描器(Scanner)
- 词法分析器从左到右扫描组成源程序的字符串，并将其转换成单词(记号—token)串；同时要：查词法错误，进行标识符登记——符号表管理。
- 输入：字符串
- 输出：(种别码，属性值)——序对
  - 属性值——token的机内表示





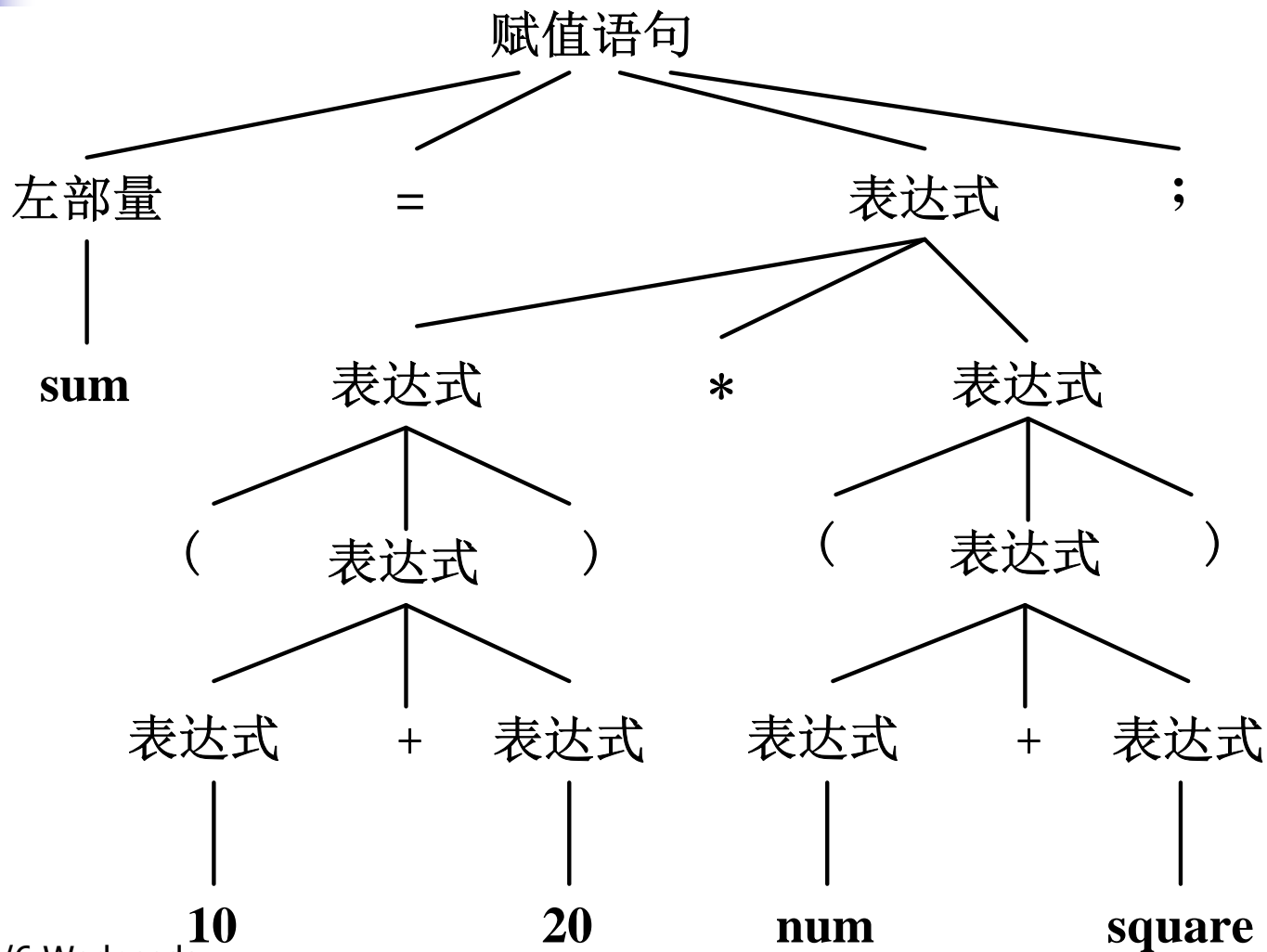
## 2、语法分析

---

- 语法分析由语法分析器(Syntax Analyzer)完成，语法分析器又叫Parser。
- 功能：
  - Parser实现“组词成句”
    - 将词组成各类语法成分：表达式、因子、项，语句，子程序...
  - 构造分析树
  - 指出语法错误
  - 指导翻译
- 输入：token序列
- 输出：语法成分

## 2、语法分析

`sum=(10+20)*(num+square);`





# 3、语义分析

- 语义分析(semantic analysis)一般和语法分析同时进行, 称为**语法制导翻译**(syntax-directed translation)
- 功能: 分析由语法分析器识别出来的语法成分的语义
  - 对于说明语句, 获取标识符的属性: 类型、作用域等
  - 对于可执行语句, 语义检查: 运算的合法性、取值范围等, 生成中间代码
  - 变量的静态绑定: 数据的相对地址



# 4、中间代码生成

---

- 中间代码(intermediate Code)
- 例: $\text{sum}=(10+20)*(\text{num}+\text{square});$

中间代码表示形式:

- $T_1=10+20$
- $T_2=\text{num}+\text{square}$
- $T_3=T_1*T_2$
- $\text{sum}=T_3$

# 4、中间代码生成

## 中间代码的常用表示形式

### 后缀表示(逆波兰Anti-Polish Notation)

sum 10 20 + num square +\*=

### 前缀表示(波兰Polish Notation)

= sum \*+10 20+num square

### 四元式表示

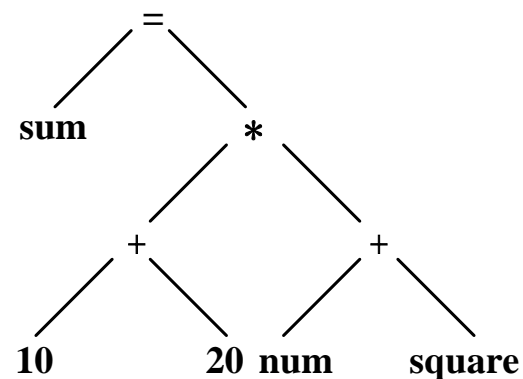
(三地址码)

(+, 10, 20,  $T_1$ )  
(+, num, square,  $T_2$ )  
(\*,  $T_1$ ,  $T_2$ ,  $T_3$ )  
(=,  $T_3$ , , sum)

### 三元式表示

(+, 10, 20)  
(+, num, square)  
(\*, (1), (2))  
(=, sum, (3))

### 语法树





# 波兰表示问题——Lukasiewicz 1929年发明

- 中缀表示(Infix notation):  $(a + ^{(1)}b) * (-c + ^{(2)}d) + ^{(3)}e / f$
- 波兰表示 (Polish / Prefix / Parenthesis-free / Lukasiewicz notation) ——也就是前缀表示
  - $+ ^{(3)} * + ^{(1)} a \ b + ^{(2)} @ c \ d / e f$
- 逆波兰表示(Reverse Polish / Suffix / Postfix notation) ——也就是后缀表示
  - $a \ b + ^{(1)} c @ \ d + ^{(2)} * e f / + ^{(3)}$  运算顺序从左向右

# 4、中间代码生成

## ■ 中间代码的特点

- 简单规范
- 与机器无关
- 易于优化与转换

三地址码的另一种表示形式:

- $T_1 = 10 + 20$
- $T_2 = \text{num} + \text{square}$
- $T_3 = T_1 * T_2$
- $\text{sum} = T_3$

其它类型的语句

例: **printf**(“hello”)

**x** := **s**            (赋值)

**param** **x**            (参数)

**call** **f**            (函数调用)

注释

**s** 是 **hello** 的地址

**f** 是函数 **printf** 的地址



# 5、代码优化

---

- 代码优化(optimization)是指对中间代码进行优化处理，使程序运行能够尽量节省存储空间，更有效地利用机器资源，使得程序的运行速度更快，效率更高。当然这种优化变换必须是等价的。
  - 与机器无关的优化
  - 与机器有关的优化





# 与机器无关的优化

---

## ■ 局部优化

- 常量合并：常数运算在编译期间完成，如 $8+9*4$
- 公共子表达式的提取：在基本块内进行的

## ■ 循环优化

- 强度削减
  - 用较快的操作代替较慢的操作
- 代码外提
  - 将循环不变计算移出循环



# 与机器有关的优化

- 寄存器的利用
  - 将常用量放入寄存器，以减少访问内存的次数
- 体系结构
  - **MIMD、SIMD、SPMD、向量机、流水机**
- 存储策略
  - 根据算法访存的要求安排：**Cache、并行存储体系——减少访问冲突**
- 任务划分
  - 按运行的算法及体系结构，划分子任务(**MPMD**)



# 6、目标代码生成

- 将中间代码转换成目标机上的机器指令代码或汇编代码
  - 确定源语言的各种语法成分的目标代码结构（机器指令组/汇编语句组）
  - 制定从中间代码到目标代码的翻译策略或算法
- 目标代码的形式
  - 具有绝对地址的机器指令
  - 汇编语言形式的目标程序
  - 模块结构的机器指令（需要链接程序）



# 7、表格管理

- 管理各种符号表(常数、标号、变量、过程、结构.....)，查、填（登记、查找）源程序中出现的符号和编译程序生成的符号，为编译的各个阶段提供信息。
  - 辅助语法检查、语义检查
  - 完成静态绑定、管理编译过程
- Hash表、链表等各种表的查、填技术
- “数据结构与算法”课程的应用



# 7、表格管理

符号表是一个数据结构。  
每个标识符在符号表中都有一条记录

例：int a, b;

名字	记号	类型	种属	.....	addr
a	id1(25)	int	简变		0
b	id2(25)	int	简变		4



# 8、错误处理

---

- 进行各种错误的检查、报告、纠正，以及相应的续编译处理(如：错误的定位与局部化)
  - 词法：拼写.....
  - 语法：语句结构、表达式结构.....
  - 语义：类型不匹配、参数不匹配.....

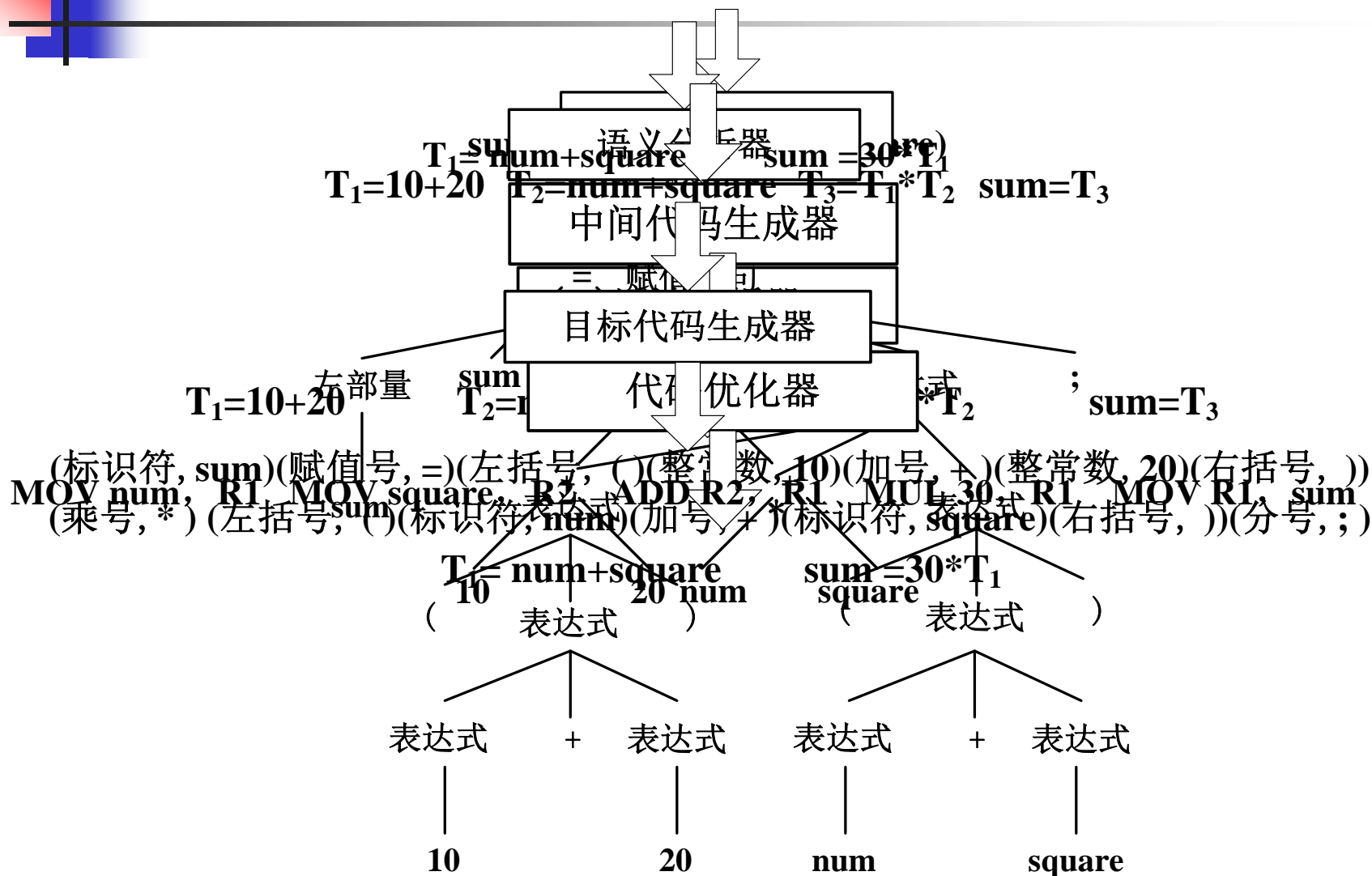


# 模块分类

---

- 分析：词法分析、语法分析、语义分析
- 综合：中间代码生成、代码优化、目标代码生成
- 辅助：符号表管理、出错处理
- 8项功能对应8个模块

# 语句 $\text{sum} = (10 + 20) * (\text{num} + \text{square});$ 的翻译过程







## 1.4 编译程序的组织

- 根据系统资源的状况、运行目标的要求……等，可以将一个编译程序设计成多遍（Pass）扫描的形式，在每一遍扫描中，完成不同的任务。
  - 如：首遍构造语法树，二遍处理中间表示，增加信息等。
- 遍可以和阶段相对应，也可以和阶段无关
- 单遍代码不太有效



# 1.4 编译程序的组织

---

- 编译程序的设计目标
  - 规模小、速度快、诊断能力强、可靠性高、可移植性好、可扩充性好
  - 目标程序也要规模小、执行速度快
- 编译系统规模较大，因此可移植性很重要
  - 为了提高可移植性，将编译程序划分为前端和后端



# 1.4 编译程序的组织

---

## ■ 前端

- 与源语言有关、与目标机无关的部分
- 词法分析、语法分析、语义分析与中间代码生成、与机器无关的代码优化

## ■ 后端

- 与目标机有关的部分
- 与机器有关的代码优化、目标代码生成



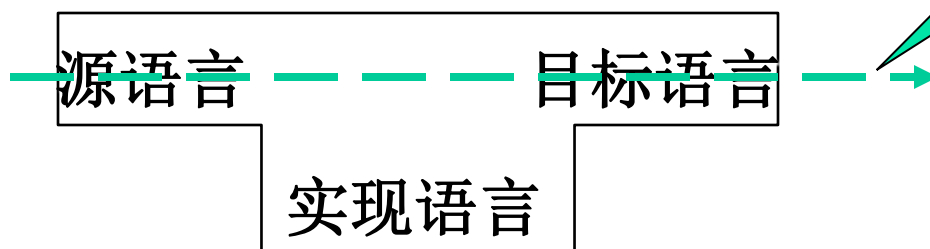
# 1.5 编译程序的生成

---

- 如何实现编译器？
  - 直接用可运行的代码编制——太费力！
  - 自举-使用语言提供的功能来编译该语言自身。
  - “第一个编译器是怎样被编译的？”

# 1. T形图

## ■ 表示语言翻译的 T 形图

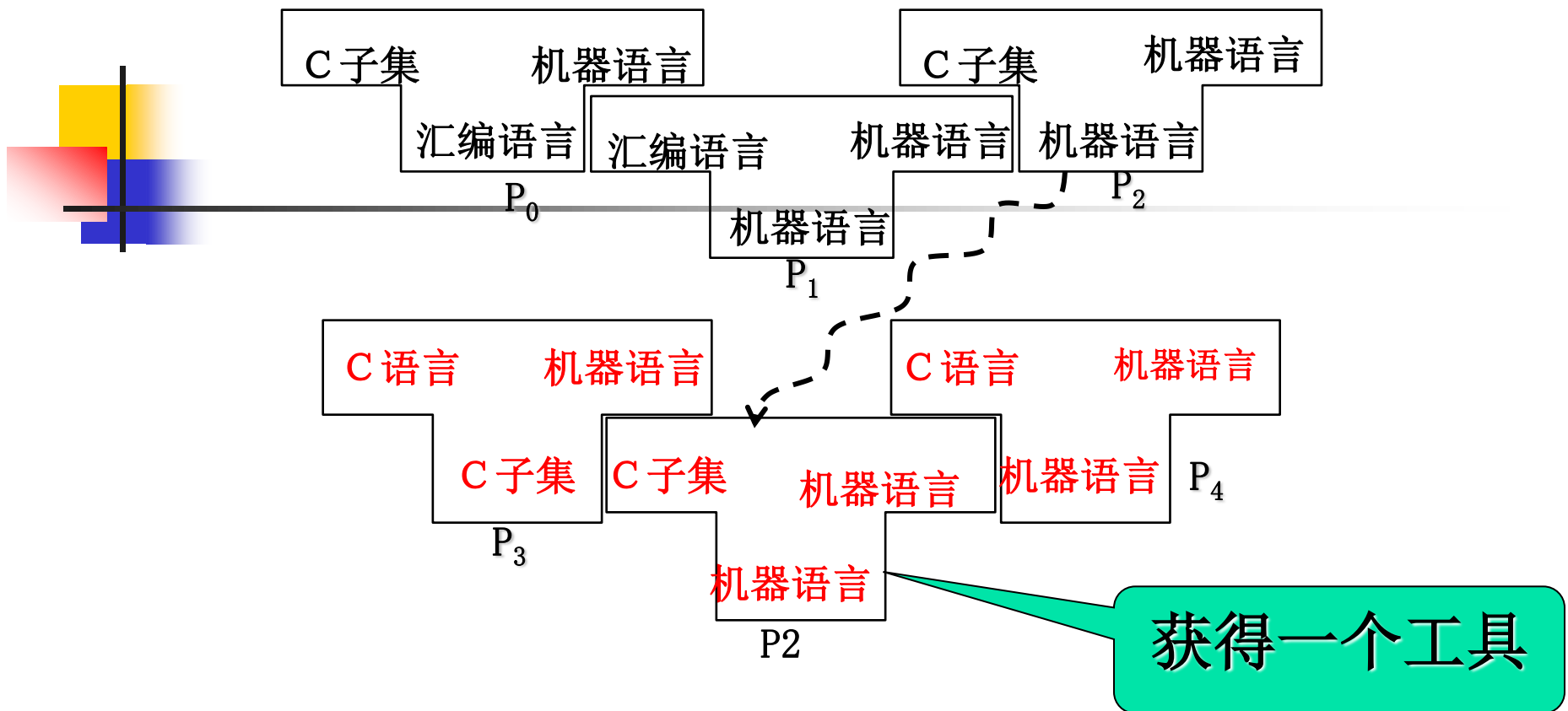




## 2. 自展

---

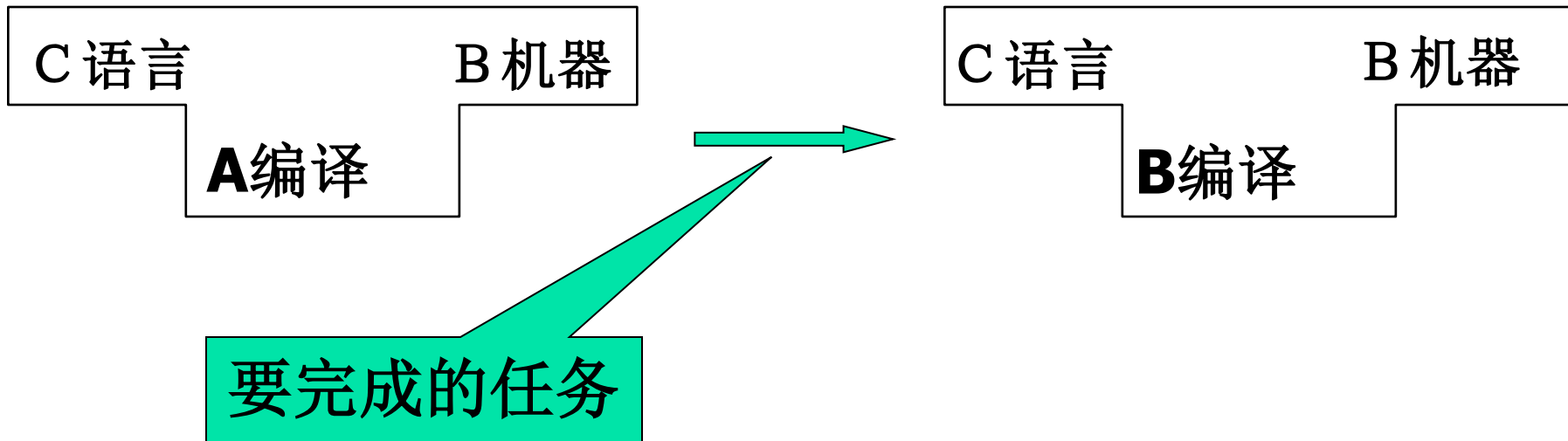
- 问题一：如何直接在一个机器上实现C语言编译器？
- 解决：
  - 用汇编语言实现一个C子集的编译程序( $P_0$ )
  - 用汇编程序处理该程序,得到( $P_2$ :可直接运行)
  - 用C子集编制C语言的编译程序( $P_3$ )
  - 用 $P_2$ 编译 $P_3$ , 得到 $P_4$



1. 用汇编语言实现一个 C 子集的编译程序(P<sub>0</sub>)
2. 用汇编程序(P<sub>1</sub>)处理该程序,得到(P<sub>2</sub>:可直接运行)
3. 用C子集编制 C 语言的编译程序(P<sub>3</sub>)
4. 用P<sub>2</sub>编译P<sub>3</sub>, 得到P<sub>4</sub>

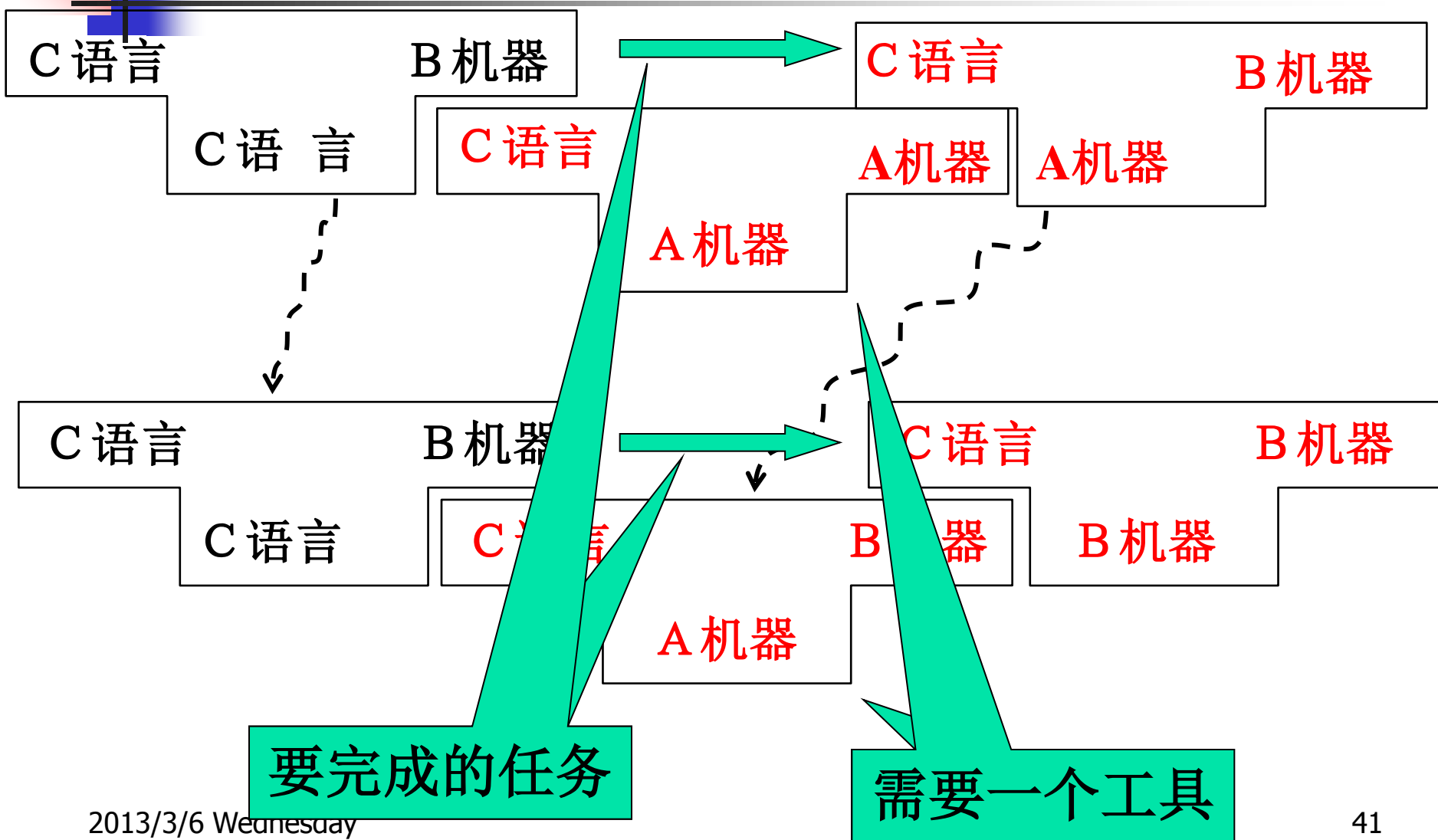
# 3.移植

- 问题二：A机上有一个C语言编译器，是否可利用此编译器实现B机上的C语言编译器？
  - 条件：A机有C语言的编译程序
  - 目的：实现B机的C语言的编译

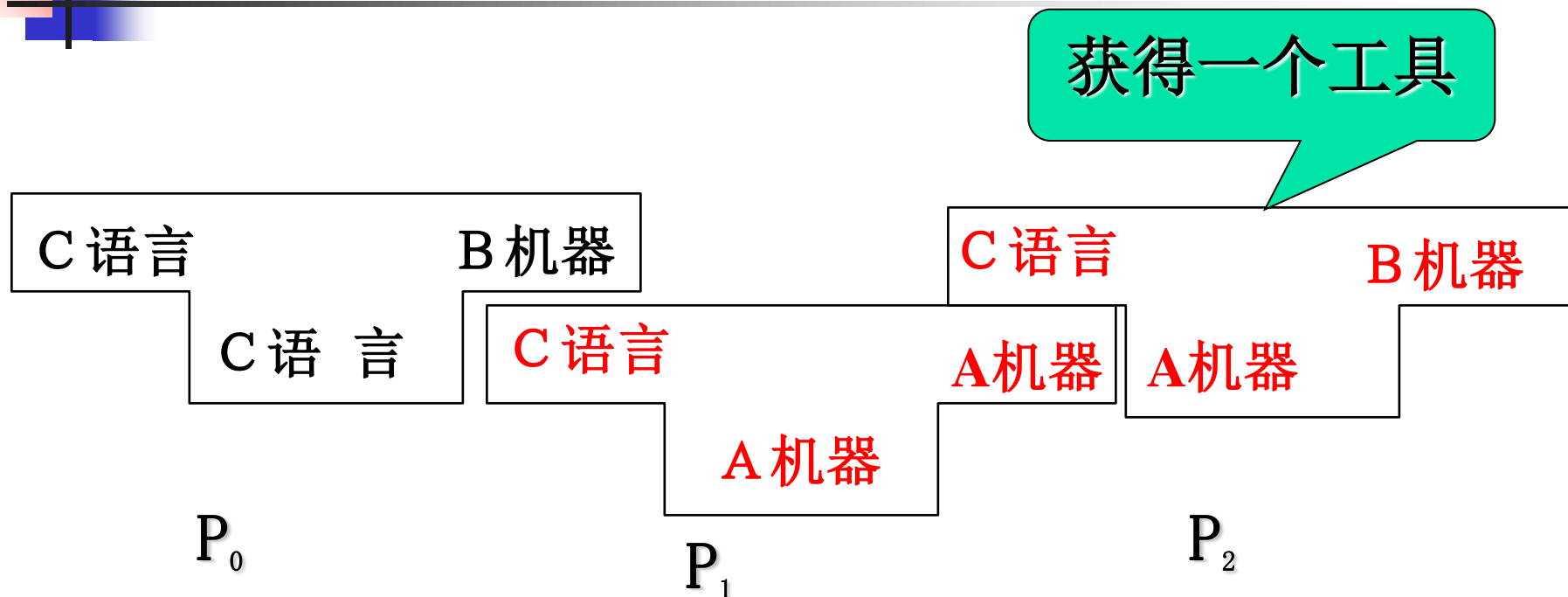




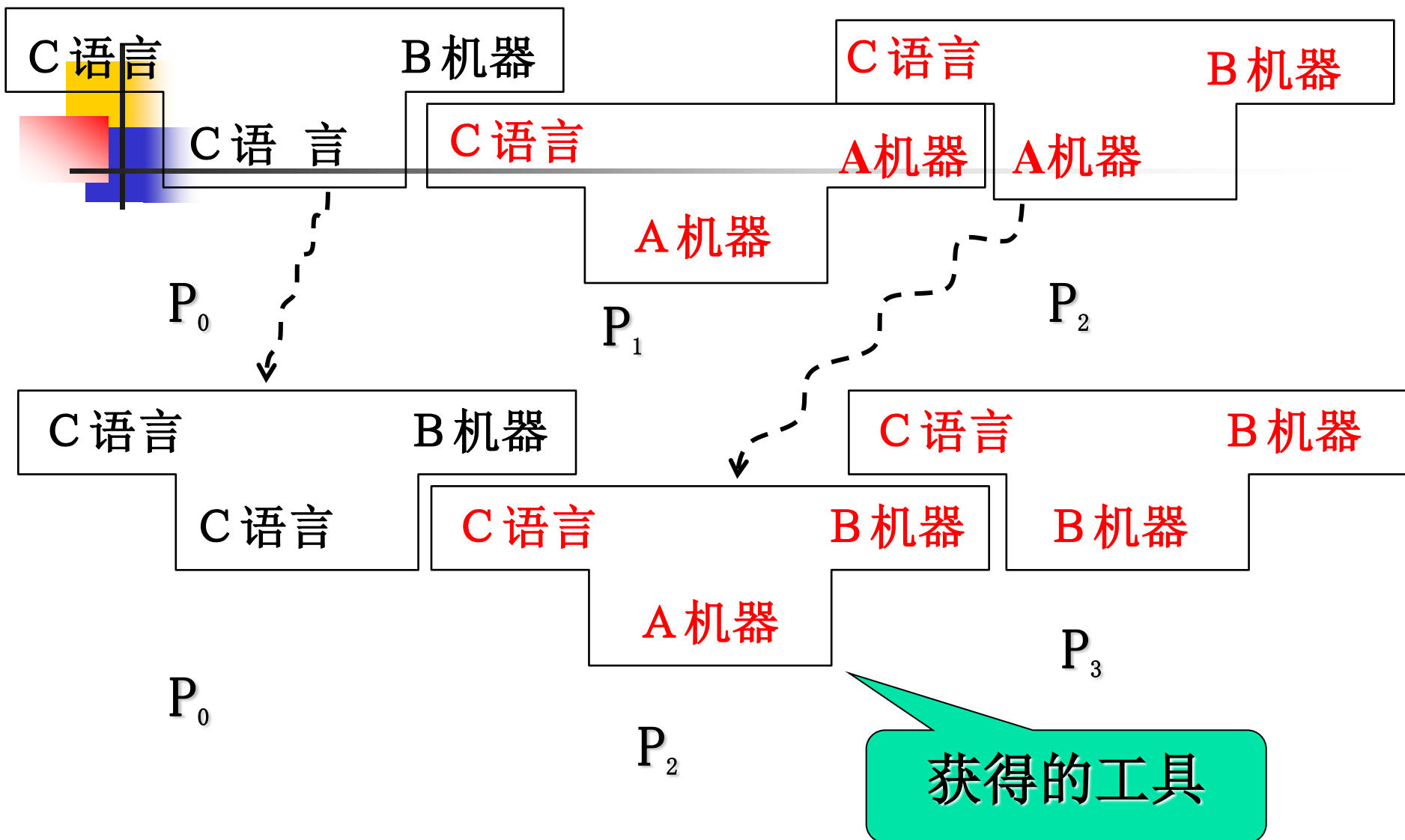
# 1)问题的分析



## 2)问题的解决办法



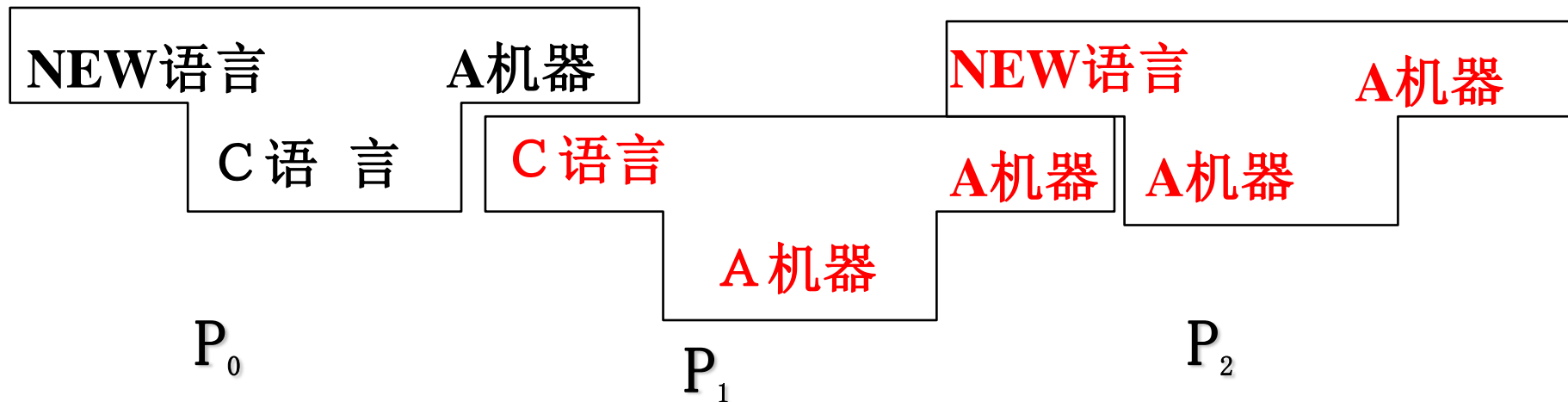
1. (人)用 C 语言编制B机的 C 编译程序 $P_0$  ( $C \rightarrow B$ )
2. (A 机的C编译 $P_1$ ) 编译 $P_0$ , 得到在A机上可运行的 $P_2$  ( $C \rightarrow B$ )



3. (A 机的P<sub>2</sub>) 编译P<sub>0</sub>, 得到在B机上可运行的P<sub>3</sub> (C→B)

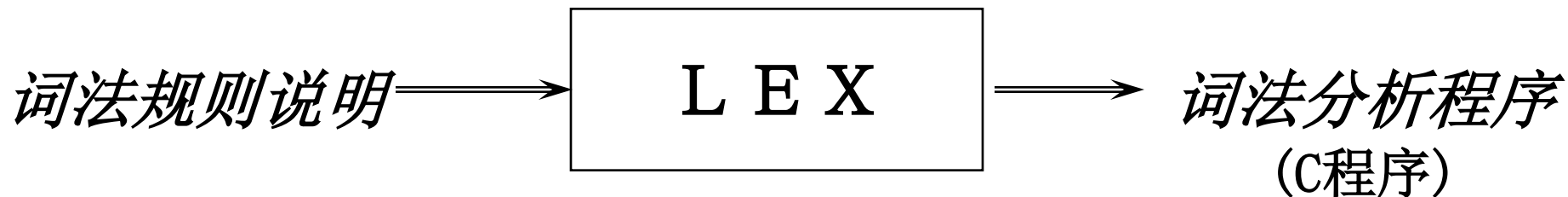
## 4.本机编译器的利用

- 问题三： A机上有一个C语言编译器，现要实现一个新语言NEW的编译器？能利用交叉编译技术么？
- 用C编写NEW的编译，并用C编译器编译它



# 5. 编译程序的自动生成

## 1) 词法分析器的自动生成程序



输入:

词法 (正规表达式)

识别动作 (C 程序段)

输出:

yylex( ) 函数

## 2) 语法分析器的自动生成程序



输入:

语法规则 (产生式)

语义动作 (C 程序段)

输出:

`yyparse()` 函数



# 1.6 本章小结

---

- 编译原理是一门非常好的课
- 程序设计语言及其发展
- 程序设计语言的翻译
- 编译程序的总体结构
- 编译程序的各个阶段
- 编译程序的组织与生成