北京大学信息科学技术学院

2022年春季学期《编译原理》

第1章 编译器概述

Introduction to Compiler

主要内容

- 编译的起源
  - 程序设计语言的发展
  - 编译的基本概念
  - 编译器 vs 解释器
- 编译器的结构
- 编译技术的应用
- 程序设计语言基础

程序设计语言的发展

第1代语言	第2代语言	第3代语言	第4代语言
机器语言	汇编语言	面向用户的语言	面向问题的语言
00011101	MOV x, 2	C/C++ Java	SQL Postscript

低级语言

高级语言

编程语言的演进

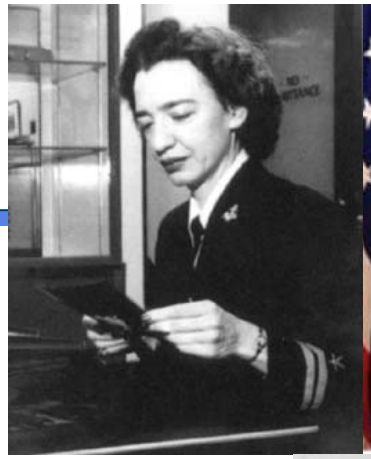
低级语言 vs. 高级语言

- 低级语言（Low level language)
  - 字位码、机器语言、汇编语言
  - 特点：与特定的机器有关，功效高，但使用复杂、繁琐、费时、易出错
- 高级语言（High level language）
  - Fortran、Pascal、C/C++、Java 语言等
  - 特点：不依赖具体机器，移植性好、对用户要求低、易使用、易维护等

用高级语言编写的程序，计算机不能立即执行，必须通过一个“翻译程序”进行加工，转化为与其等价的机器语言程序，机器才能执行。

这种翻译程序，被称为“编译程序”。

最早的编译器



- 第一个编译器（1952年）
  - A-0 System on UNIVAC I
  - 编写者：Grace Murray Hopper
  - 美国海军少将
  - COBOL发明人
  - 创造了术语“debug”
- 第一个完整的编译器 （1957年）
  - John Backus 领导的 FORTRAN编译器 (IBM)
- 第一个高级语言书写的编译器（1962年）
  - Lisp编译器（MIT, Tim Hart and Mike Levin）

如何构造第一个编译器？

- 在机器W上，如何为语言X构造一个用X编写的编译器？
  - 转化程序，X->W
- 假设机器W上已经存在某种语言Y的编译器
  - 首先用Y语言编写X语言的编译器，
  - 然后用该编译器编译用X语言编写X的编译器
- 假设机器W上不存在任何语言的编译器
  - **bootstrapping**

基本概念

- 源程序（Source Program）
  - 用高级语言编写的程序称为源程序。
- 目标程序（Object Program）
  - 用目标语言所表示的程序。
  - 目标语言：可以是介于源语言和机器语言之间的“中间语言”,可以是某种机器的机器语言,也可以是某机器的汇编语言。
- 翻译程序（Translator）
  - 将源程序转换为目标程序的程序称为翻译程序。
  - 它是指各种语言的翻译器，包括汇编程序和编译程序，是汇编程序、编译程序以及各种变换程序的总称。

编译的基本功能

- 编译（Compilation）就是把源程序翻译成与其等价的目标语言程序的过程。
  - 等价：针对所有相同输入会给出相同的输出
  - 在应用程序与硬件体系结构之间的接口
  - 应当对所生成的程序进行一定的优化

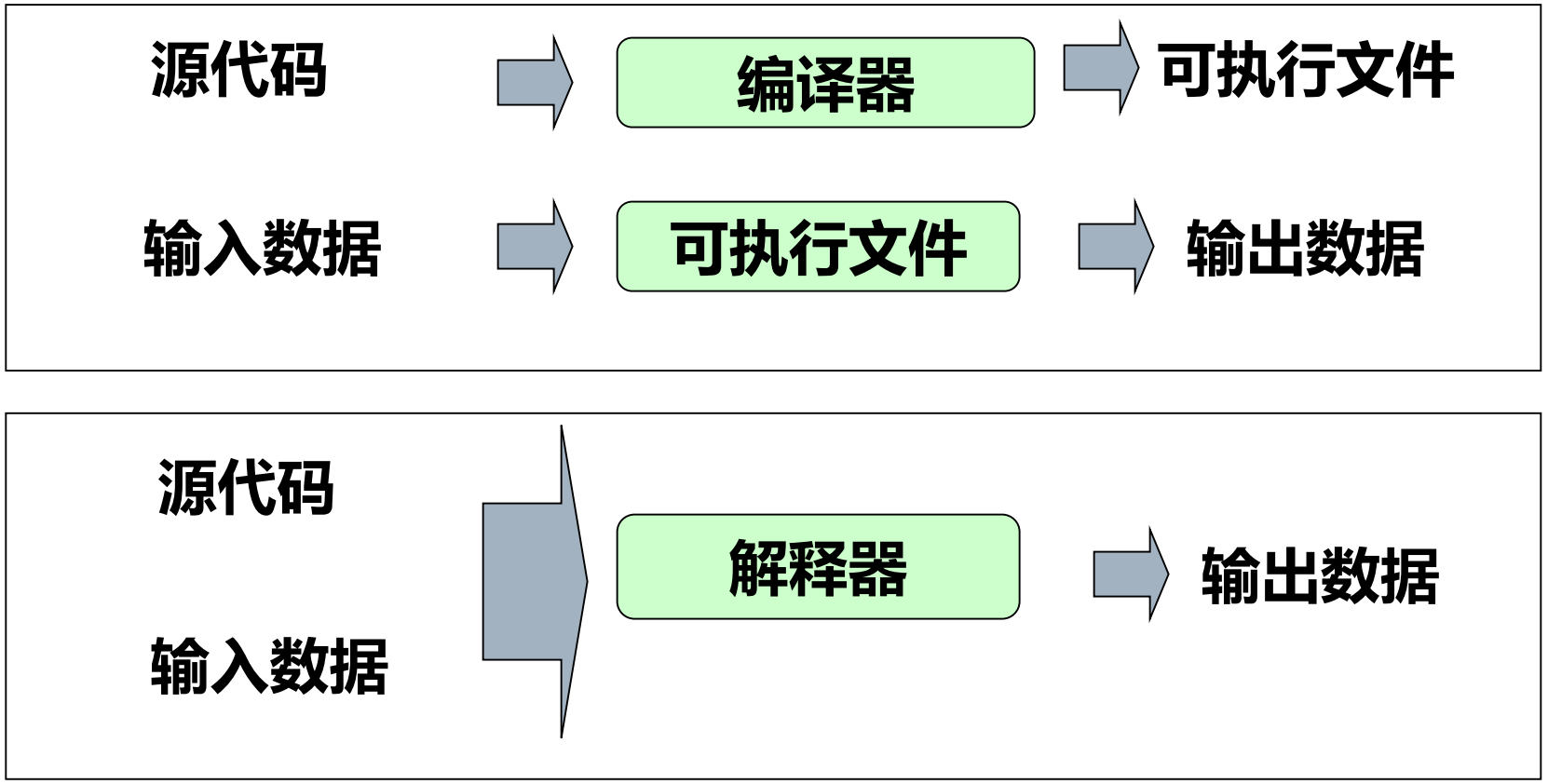
编译器 vs. 解释器 (1/5)

- 编译器（Compiler）：把（人编写的）源程序转化为（机器可读的）可执行程序，线下的过程(offline)

- 解释器（Interpreter）：在转换源程序的同时执行程序，线上的过程(online)
  - 不生成目标代码，而是直接执行源程序所指定的运算
  - 解释器也需要对源程序进行词法，语法和语义分析，中间代码生成

编译器 vs. 解释器 (2/5)

- 理想状态



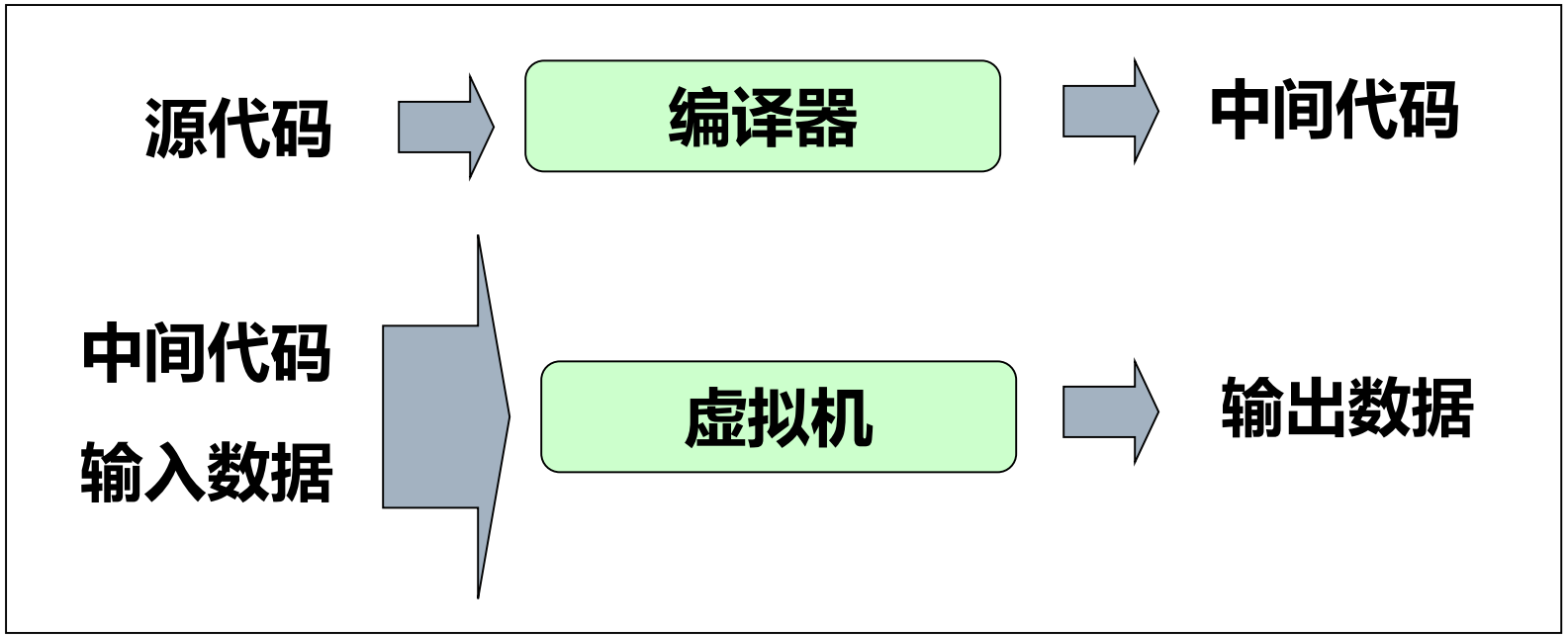
编译器 vs. 解释器 (3/5)

- 大多数语言通常属于二者之一：
  - 编译器：**FORTRAN, COBOL, C, C++, Pascal, PL/1**
  - 解释器：**Lisp, BASIC, APL, Perl, Python, Smalltalk**

- 但是：在现实中并不总是这样来区分的
  - **e.g., Java, JVM(Just-in-time, JIT Compiler)**

编译器 vs. 解释器 (4/5)

- 实际使用中，在二者之间并没有明确的边界
  - 常见的情形是二者混合起来



编译器 vs. 解释器 (5/5)

编译器

- 优点
  - 执行速度快
  - 占空间小
- 缺点
  - 调试困难
  - 目标代码不可移植
  - 额外的编译时间

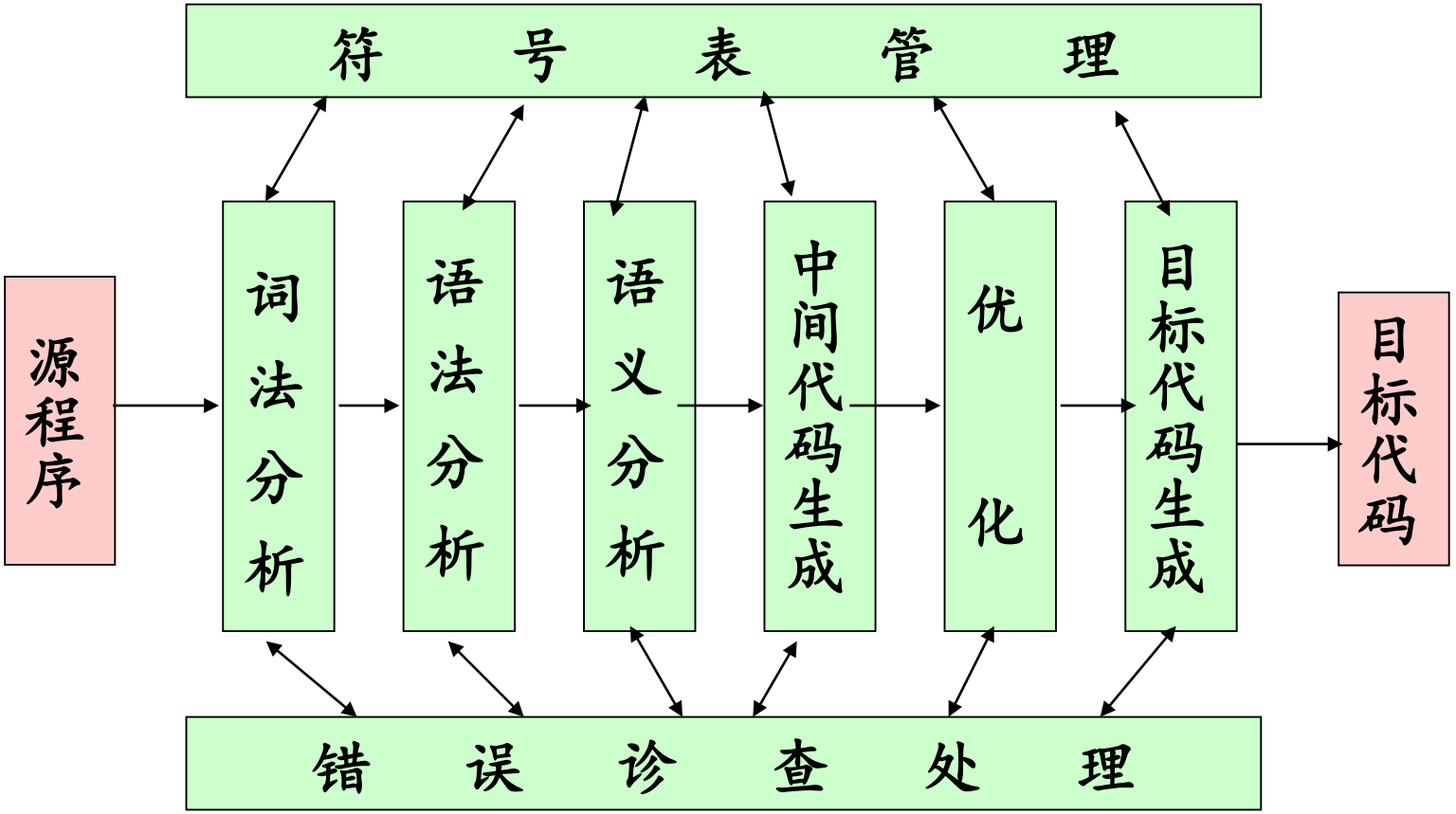
解释器

- 优点
  - 易于调试
  - 开发快捷
  - 便于移植
- 缺点
  - 执行速度较慢
  - 解释器必须常驻内存
  - 占用较多空间

主要内容

- 编译的起源
  - 程序设计语言的发展
  - 编译的基本概念
  - 编译器vs解释器
- **编译器的结构**
- 编译技术的应用
- 程序设计语言基础

编译器的结构



编译程序（器）的组织

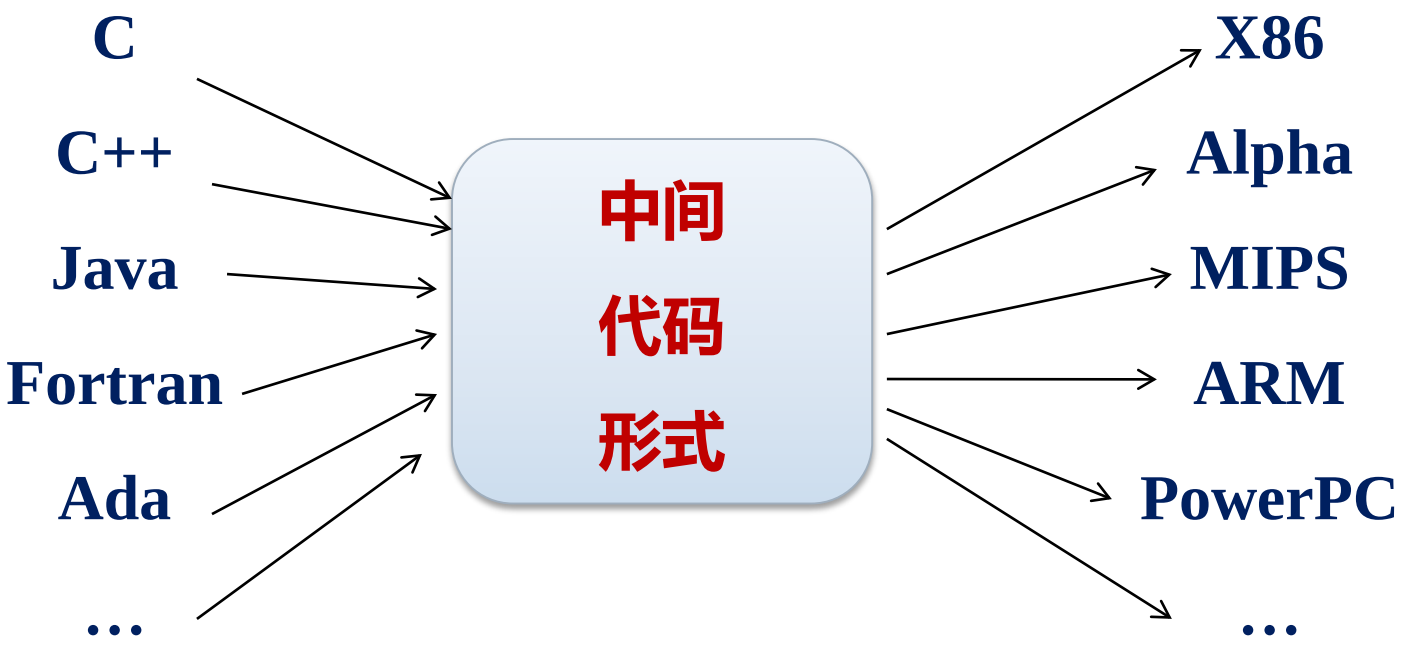
- 前端（frontend）：从源代码到中间代码的翻译，与具体机器无关
- 后端（backend）：把中间代码翻译为具体的机器指令（目标代码）

分析
（Analysis）

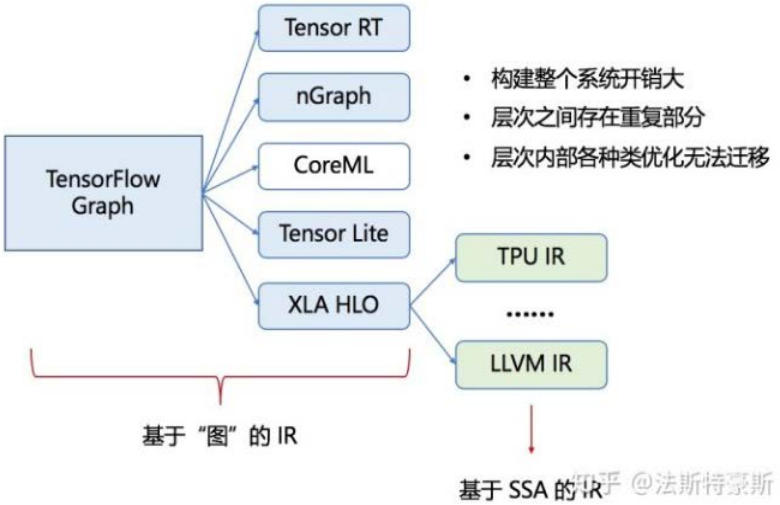
代码生成/综合
（Code generation/ Synthesis ）

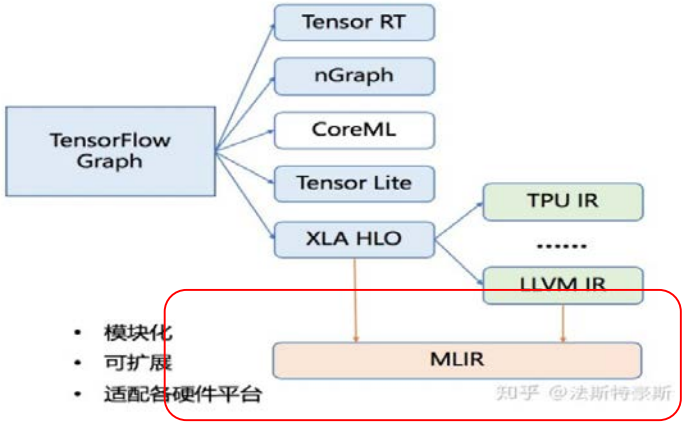
现代编译器(LLVM)

- 多种语言（C, C++, Java, Fortran, Ada…）
- 多种目标机器（x86, Alpha, MIPS, ARM…）



现代编译器(MLIR)





大量的IR各自有各自的目的，但是很多相似的功能被重复实现，比如CSE等

设计一个中心化的表示，集成所有的IR，统一地写各个pass，这样不同的IR都复用同一个实现

编译器流程示例

一个C语言源程序

```
void cal(float b,
         float c){
   float a;

   a = b+c*60;
   printf("%f\n", a);
}
```

编译器流程示例——词法分析

词法分析的结果是代表单词的token流

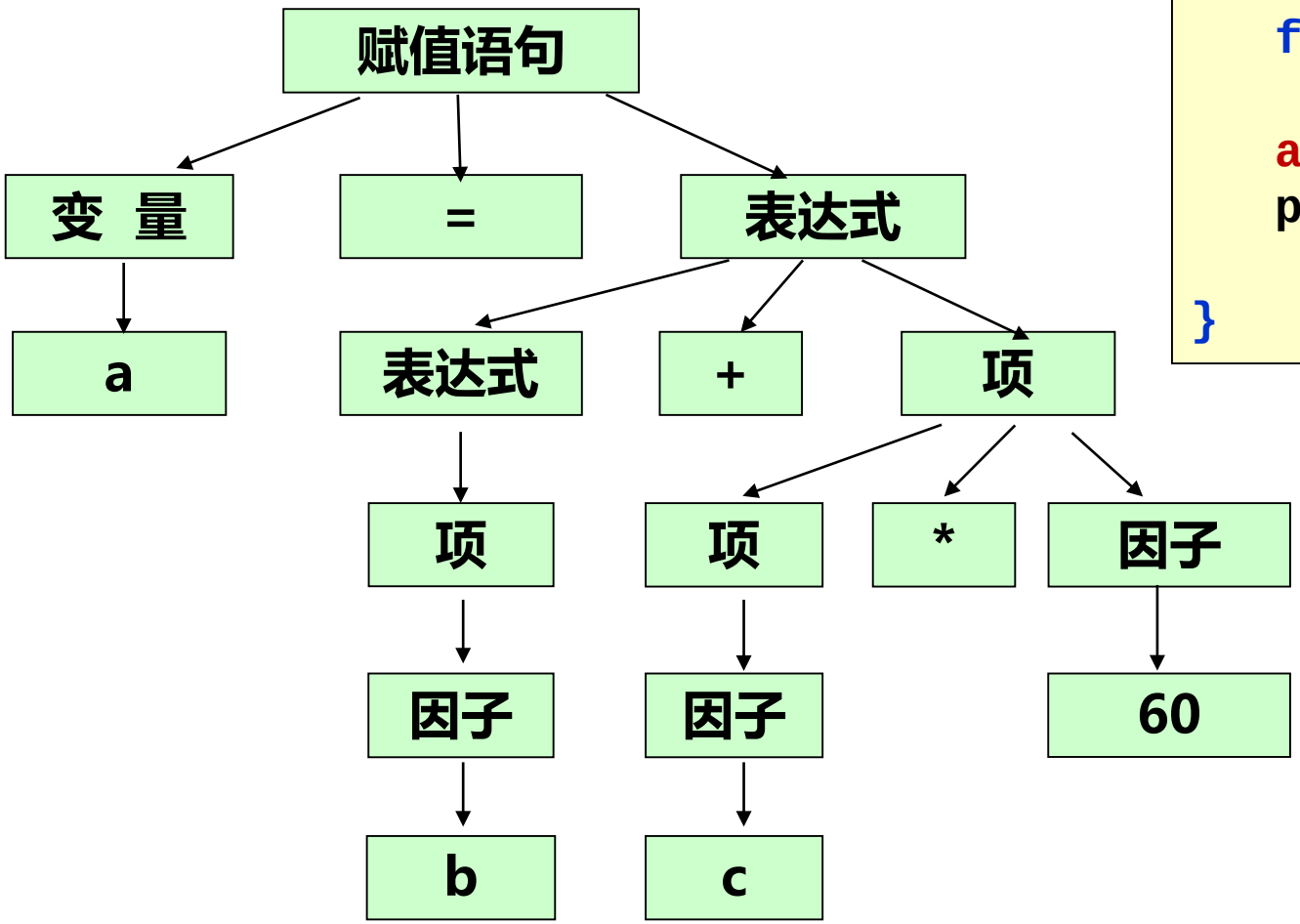
<保留字，void>
<标识符，cal>
<分隔符，(>
<保留字，float>
……
<标识符，a>
<算符，=><标识符，b> <算符，+>
<标识符，c> <算符，*> <常数，60>
……
<字符串，“%f\n”>
<分隔符，，>
<标识符，a>
……

```
void cal(float b,
         float c){
    float a;

    a = b+c*60;
    printf(“%f\n”,
           a);
}
```

编译器流程示例——语法分析

为程序中赋值语句生成的分析树

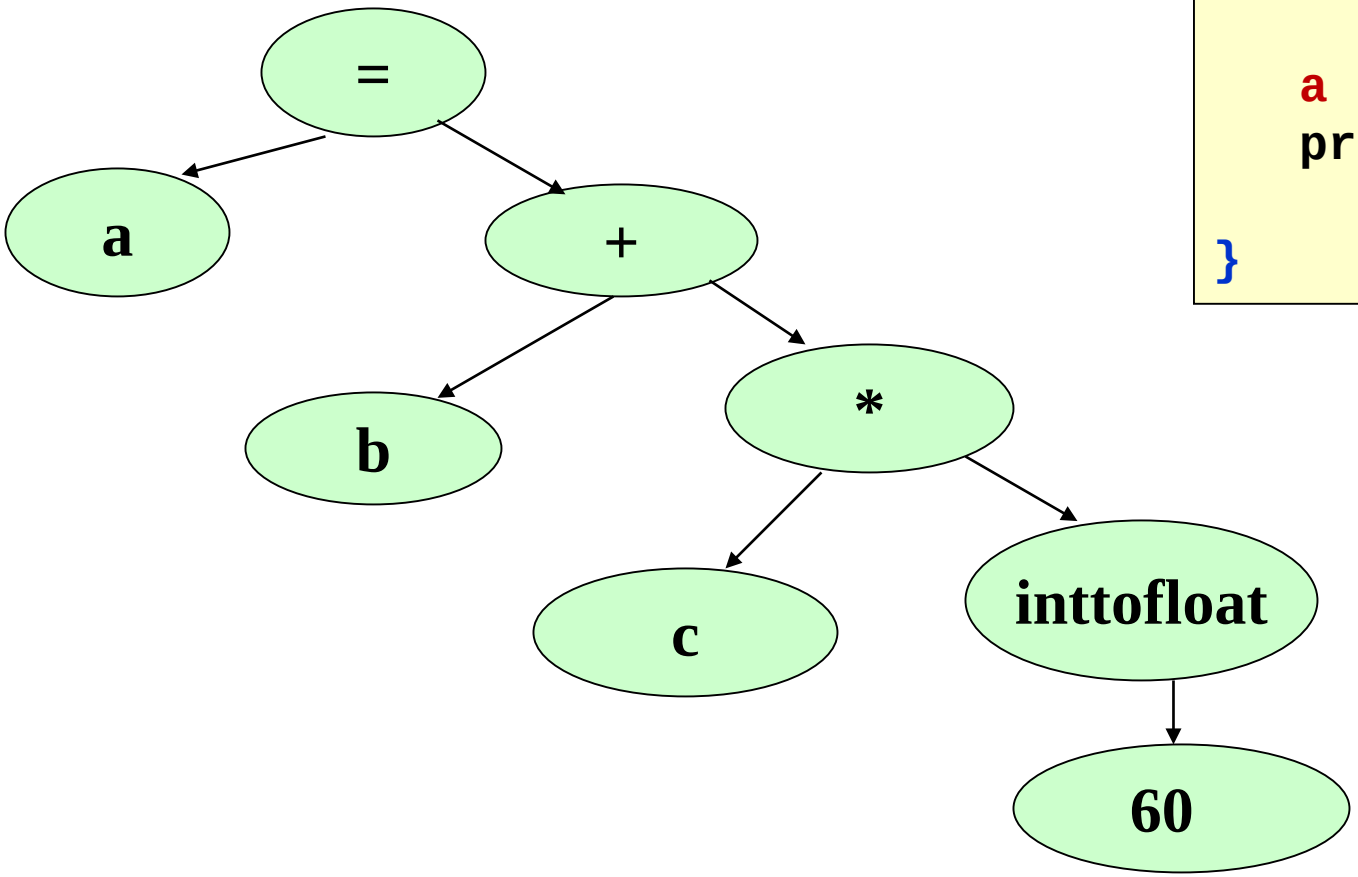



```
void cal(float b,
         float c){
   float a;

   a = b+c*60;
   printf("%f\n",
          a);
}
```

编译器流程示例——语义分析

为程序中赋值语句生成的抽象语法树

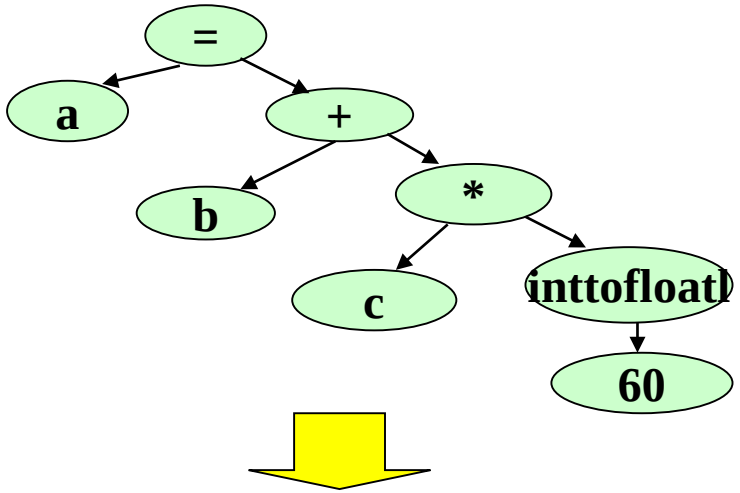



```
void cal(float b,
         float c){
   float a;

   a = b+c*60;
   printf("%f\n",
          a);
}
```

编译器流程示例——中间代码生成

为程序中赋值语句生成的中间代码表示




```
void cal(float b,
         float c){
   float a;

   a = b+c*60;
   printf("%f\n",
          a);
}
```

```
temp1 = inttofloat(60);
temp2 = c * temp1;
temp3 = b + temp2;
a = temp3;
```

编译器流程示例——代码优化

```
void cal(float b,
         float c){
   float a;

   a = b+c*60;
   printf("%f\n",
          a);
}
```

中间代码

```
temp1 = inttofloat(60);
temp2 = c * temp1;
temp3 = b + temp2;
a = temp3;
```

优　　化

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

编译器流程示例——目标代码生成

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

目标代码生成

```
LDF    R2, c
MULF   R2, R2, #60.0
LDF    R1, b
ADDF   R1, R1, R2
STF    a, R1
```

```
void cal(float b,
         float c){
   float a;

   a = b+c*60;
   printf("%f\n",
          a);
}
```

符号表

```
void cal(float b,
         float c){
   float a;

   a = b+c*60;
   printf("%f\n",
          a);
}
```

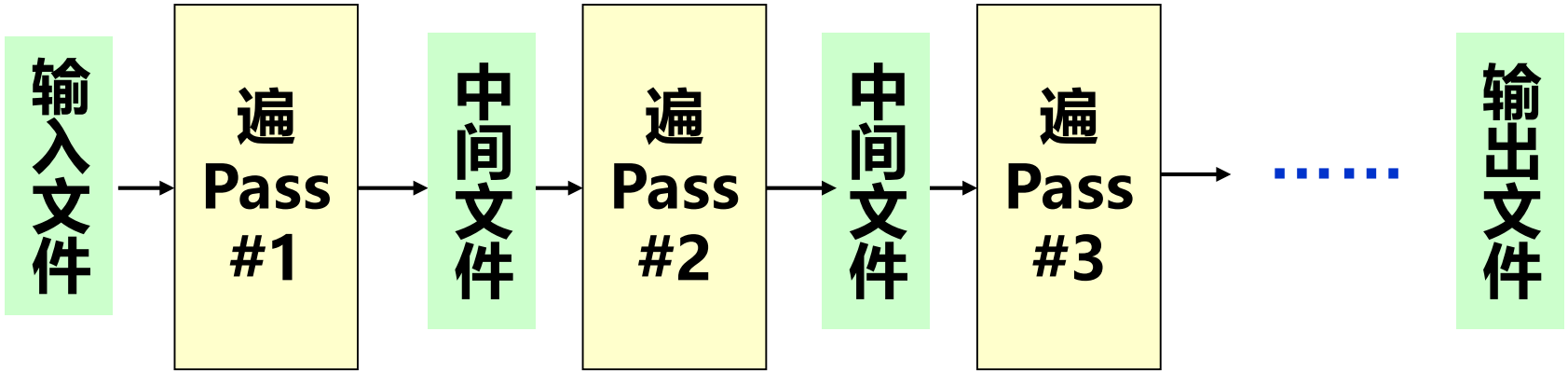
名 字	种 类	类 型	其他信息
cal	函 数		
a	变 量	float	
b	变 量	float	
c	变 量	float	
printf	函数		
…			

错误诊断与处理

- 编译程序在各个阶段应诊断和报告源程序中的错误
  - 词法错误
  - 语法错误
  - 语义错误
- 编译程序应报告出错位置（文件、行、列），并给出相应的纠错提示信息。
- 出错处理能力的强弱是衡量编译程序质量好坏的一个重要指标。

遍（PASS）

- 对源程序或其等价的中间形式（通常以文件形式存在）从头到尾扫视一次，完成预定的处理任务。



编译器构造工具

- 目标：辅助用户生成复杂的、容易出错的编译分析程序
- 词法分析工具
  - Lex, Flex
- 语法分析工具
  - YACC, Bison, ANTLR
- 研究/教学型编译器
  - SUIF, Soot, JikeRVM, LLVM, ……

主要内容

- 编译的起源
  - 程序设计语言的发展
  - 编译的基本概念
  - 编译器vs解释器
- 编译器的结构
- 编译技术的应用
- 程序设计语言基础

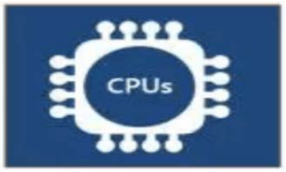
编译技术的应用（1/4）

- 新的编程语言的实现
  - 流行编程语言的大多数演变都是朝着提高抽象级别的方向，每一轮编程语言新特征的出现都刺激编译器优化的新研究
    - 支持用户定义的聚合数据类型和高级控制流，如数组和记录、循环和过程调用：C、Fortran
    - 面向对象的主要概念是数据抽象和性质继承，使得程序更加模块化并易于维护：C++、C#、Java
    - 类型安全的语言：Java没有指针，也不允许指针算术。它用无用单元收集机制来自动地释放那些不再使用的变量占据的内存
    - 面向大规模并行编程的语言与编译优化

编译技术的应用（2/4）

- 针对计算机体系结构的优化
  - 计算机体系结构的演化推动了新的编译器技术的研究
    - 多核、众核、专用架构(Domain Specific Architecture)

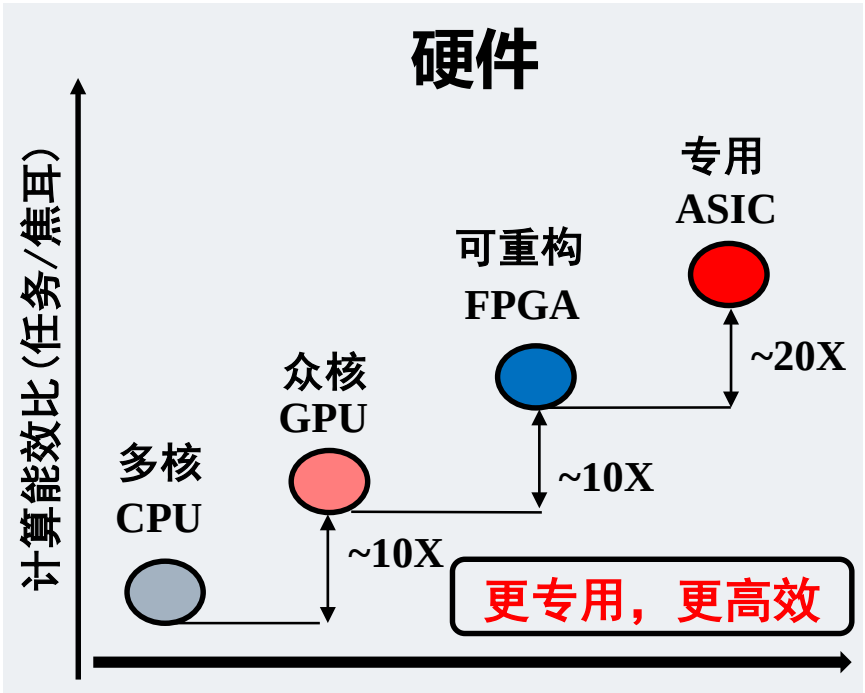
- 新型计算机体系结构的设计
  - 计算机系统的性能不仅仅取决于它的原始速度，还取决于编译器是否能生成充分利用其特征的代码

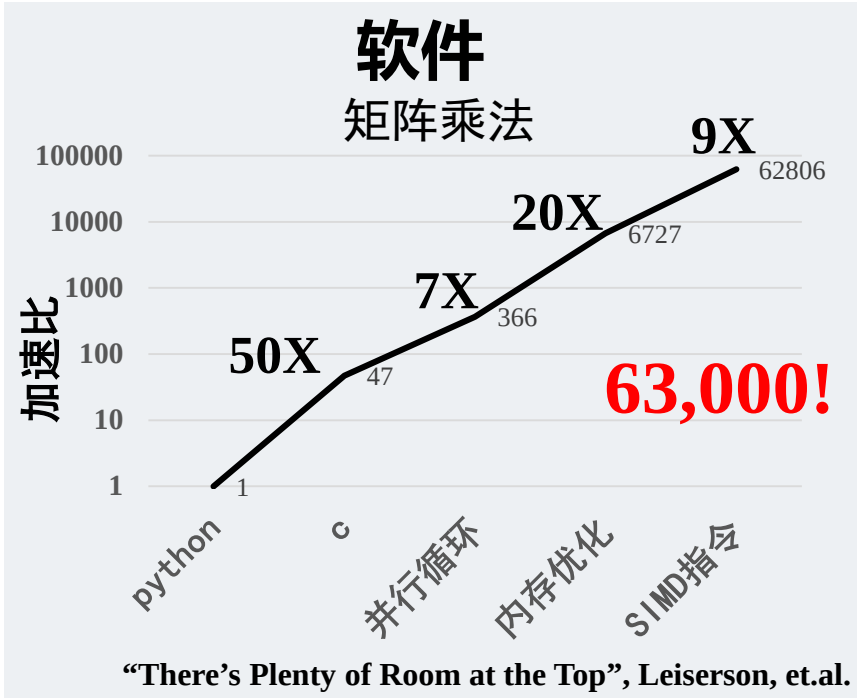


编译技术的应用（2/4）

软硬件协同设计

David Patterson
2018图灵奖





“There’s Plenty of Room at the Top”, Leiserson, et.al.

编译技术的应用（3/4）

- 提高软件开发效率的工具

  源于代码优化技术的程序分析一直在改进软件开发效率

  - 类型检查

    类型检查是一种捕捉程序中前后不一致的有效技术

  - 边界检查

    数据流分析技术可用来定位缓冲区溢出

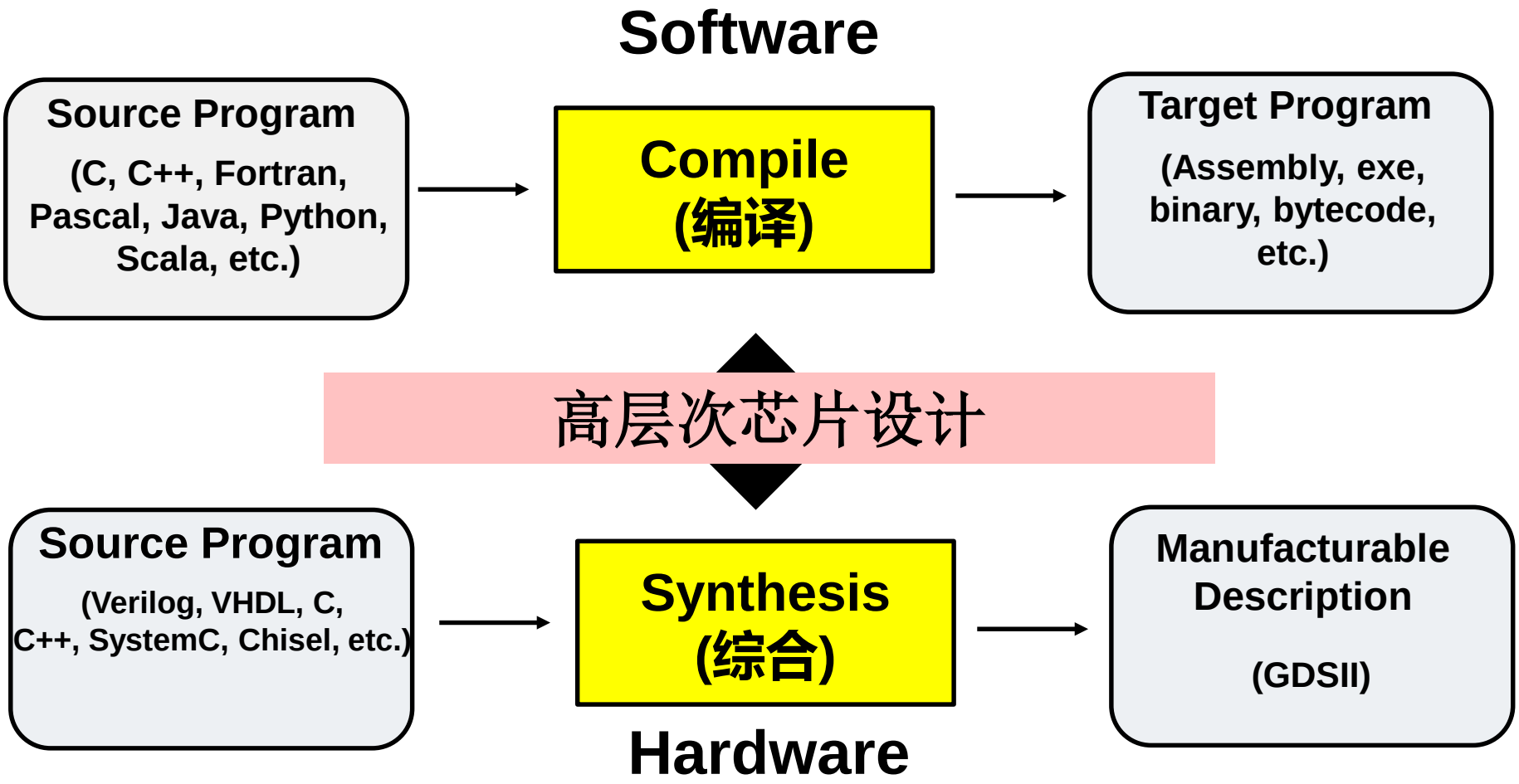
  - 内存管理

    自动的内存管理删除内存泄漏等内存管理错误

- 其他应用

  - 软件测试工具，程序理解工具
  - 程序变换：二进制翻译、硬件综合 (hardware synthesis)，等等

编译技术的应用（4/4）


Software
Source Program
(C, C++, Fortran, Pascal, Java, Python, Scala, etc.)
Compile
(编译)
Target Program
(Assembly, exe, binary, bytecode, etc.)
高层次芯片设计
Source Program
(Verilog, VHDL, C, C++, SystemC, Chisel, etc.)
Synthesis
(综合)
Manufacturable Description
(GDSII)
Hardware

主要内容

- 编译的起源
  - 程序设计语言的发展
  - 编译的基本概念
  - 编译器 vs 解释器
- 编译器的结构
- 编译技术的应用
- 程序设计语言基础

程序设计语言基础（1/6）

- 静态/动态（static vs dynamic）
  - 静态：语言策略支持编译器静态决定某个问题
  - 动态：只允许在程序运行时刻作出决定
  - 例：Java类声明中的static指明了变量的存放位置可静态确定。
- 作用域（scope）
  - x的一个声明的作用域指程序中的一个区域，其中对x的使用都指向这个声明。
  - 静态作用域：通过静态阅读程序即可决定作用域
  - 动态作用域：程序运行时决定作用域

程序设计语言基础（2/6）

- 环境与状态
  - **环境 (environment)：**是从名字到内存地址的映射
  - **状态 (state)：**从内存地址到它们的值的映射

```
...
int i;                          /* 全局i          */
...
void f(···) {
      int i;                    /* 局部 i         */
      ...
      i = 3;                    /* 对局部i的使用  */
      ...
}
...
      x = i + 1;                /* 对全局i 的使用 */
```

图 1-9　名字 i 的两个声明

程序设计语言基础（3/6）

- 静态作用域和块结构
  - C语言(以及C++/Java/C#等)使用静态作用域。
    - C语言程序由顶层的变量、函数声明组成
    - 函数内部可以声明变量（局部变量/参数），这些声明的作用域在它出现的函数内
    - 一个顶层声明的作用域包括其后的所有程序。
  - 作用域规则基于程序结构，声明的作用域由它在程序中的位置决定。
  - 也可以通过public、private、protected等限定词进行明确控制

程序设计语言基础（4/6）

```
main() {
    int a = 1;                          B1
    int b = 1;
    {
        int b = 2;                      B2
        {
            int a = 3;                  B3
            cout << a << b;
        }
        {
            int b = 4;                  B4
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

声　　明	作用域
int a = 1;	$B_1 - B_3$
int b = 1;	$B_1 - B_2$
int b = 2;	$B_2 - B_4$
int a = 3;	B_3
int b = 4;	B_4

- 块作用域的例子

程序设计语言基础（5/6）

- 参数传递机制
  - **传值调用 (call by value)：**对实在参数求值/拷贝，再存放到被调用过程的形参的内存位置上。
  - **引用调用 (call by reference)：**实际传递的是实在参数的地址。

程序设计语言基础（6/6）

- 别名(Alias)
  - 两个指针指向同一个位置的情况
  - 在函数调用时，看起来不同的形式参数实际上可能互为别名

- 如何发现别名？
  - Alias Analysis (或 Points-to Analysis)

本章小结

- 编译是把一种语言编写的源程序翻译成为另外一种语言的目标程序的过程
- 编译器具有非常良好的系统架构，对于大规模程序开发来说具有很好的借鉴意义
  - 前端、后端、遍
- 编译技术在计算科学的各个领域得到了广泛的应用

Next

- 第3章“词法分析”

- 自学：龙书第2章
  - “一个简单的语法制导翻译器”
  - 阅读本章内容，并对照附录代码学习