中山大學 SUN YAT-SEN UNIVERSITY 计算机学院（软件学院） SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle
编 译 原 理

第0讲：课程介绍、概述

张献伟

xianweiz.github.io

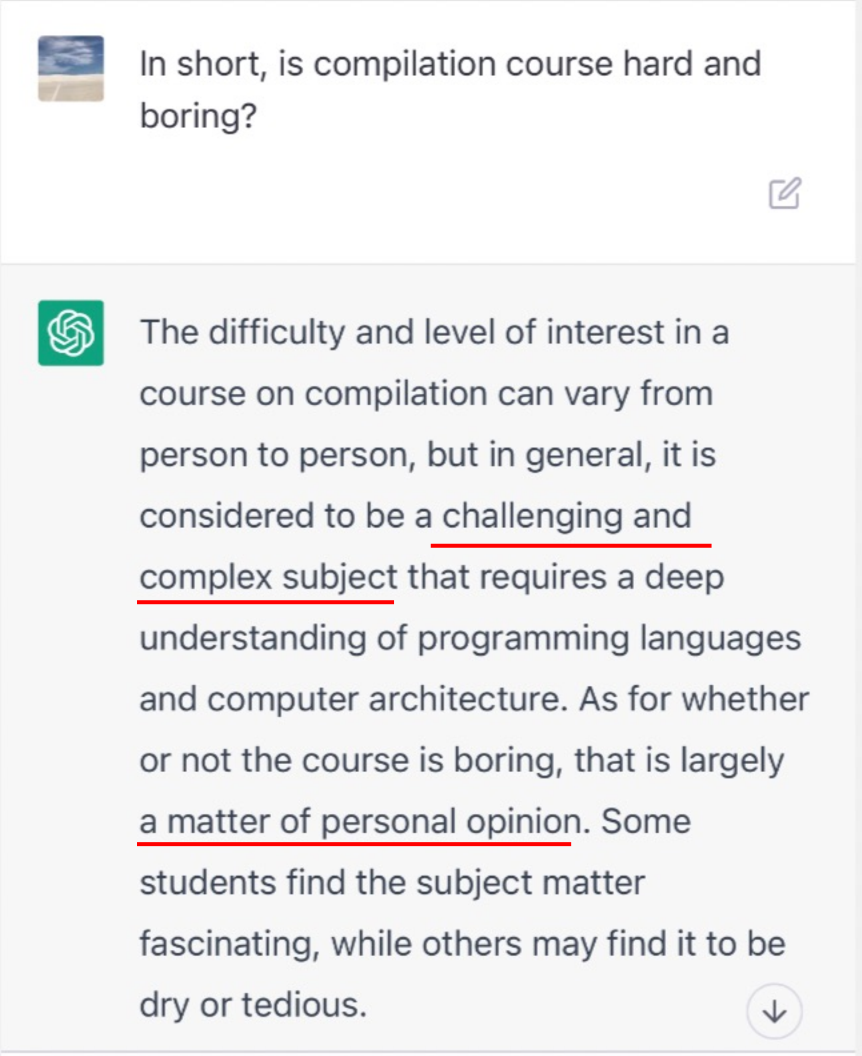
DCS290, 2/27/2024

sysu–编译(张)

群号: 189205980

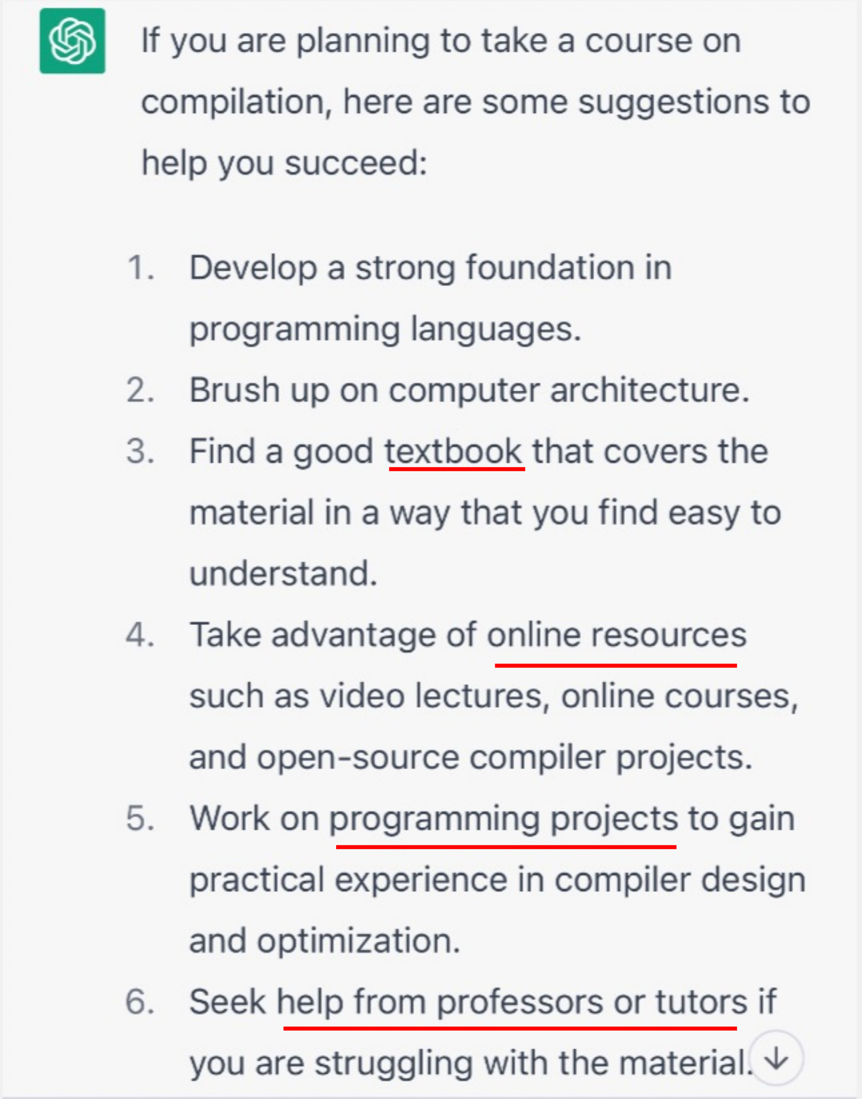
扫一扫二维码，加入群聊

Compilation, what does ChatGPT say?



In short, is compilation course hard and boring?

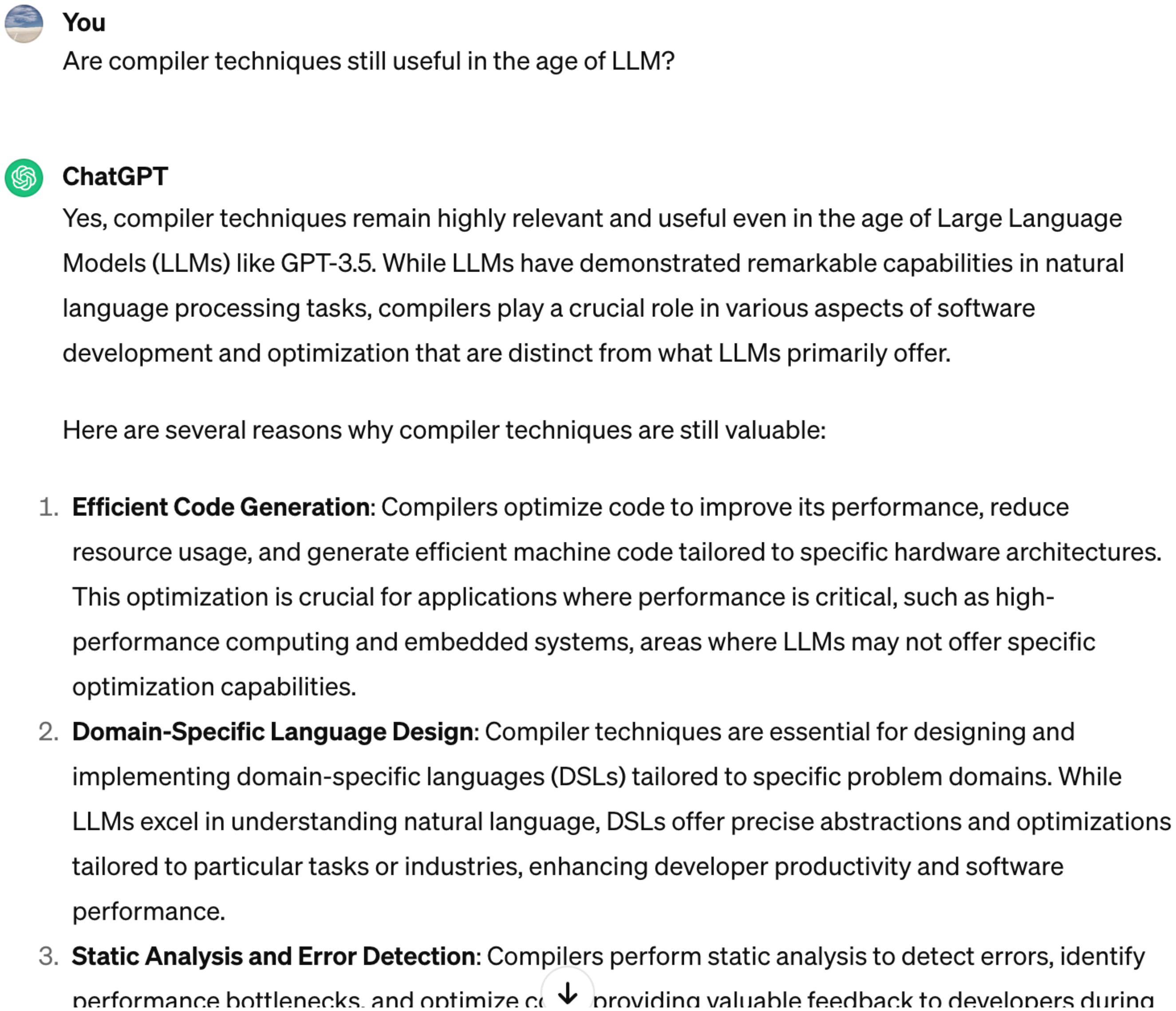
The difficulty and level of interest in a course on compilation can vary from person to person, but in general, it is considered to be a challenging and complex subject that requires a deep understanding of programming languages and computer architecture. As for whether or not the course is boring, that is largely a matter of personal opinion. Some students find the subject matter fascinating, while others may find it to be dry or tedious.



If you are planning to take a course on compilation, here are some suggestions to help you succeed:

1. Develop a strong foundation in programming languages.
2. Brush up on computer architecture.
3. Find a good textbook that covers the material in a way that you find easy to understand.
4. Take advantage of online resources such as video lectures, online courses, and open-source compiler projects.
5. Work on programming projects to gain practical experience in compiler design and optimization.
6. Seek help from professors or tutors if you are struggling with the material.

Compilation, what does ChatGPT say?



You

Are compiler techniques still useful in the age of LLM?

ChatGPT

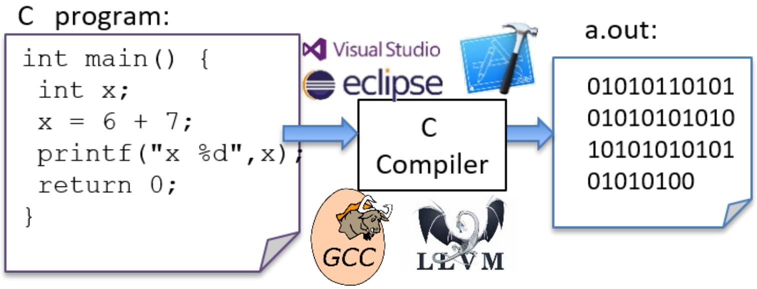
Yes, compiler techniques remain highly relevant and useful even in the age of Large Language Models (LLMs) like GPT-3.5. While LLMs have demonstrated remarkable capabilities in natural language processing tasks, compilers play a crucial role in various aspects of software development and optimization that are distinct from what LLMs primarily offer.

Here are several reasons why compiler techniques are still valuable:

1. **Efficient Code Generation**: Compilers optimize code to improve its performance, reduce resource usage, and generate efficient machine code tailored to specific hardware architectures. This optimization is crucial for applications where performance is critical, such as high-performance computing and embedded systems, areas where LLMs may not offer specific optimization capabilities.
2. **Domain-Specific Language Design**: Compiler techniques are essential for designing and implementing domain-specific languages (DSLs) tailored to specific problem domains. While LLMs excel in understanding natural language, DSLs offer precise abstractions and optimizations tailored to particular tasks or industries, enhancing developer productivity and software performance.
3. **Static Analysis and Error Detection:** Compilers perform static analysis to detect errors, identify performance bottlenecks, and optimize c providing valuable feedback to developers during

The Course[关于课程]

- 年级专业
  - 21级计科/1班（李文军）
  - 21级计科/2班（冯剑琳 + 林瀚）
  - 21级计科/3班（张献伟）
  - 21级计科/系统+人工大数据等 （赵帅）
- 先修课程
  - C/C++或其他编程语言、数据结构
  - 计算机组成原理/体系结构、离散数学
- 编译原理
  - 高级编程语言（如C）是如何转换为机器语言（0/1）的？
  - 介绍编译器设计与实现的主要理论和技术
    - 包括词法分析、语法分析、语义分析、代码生成、代码优化等
- 编译器构造实验
  - 独立课程，分阶段实现小型编译器



```
int main() {
 int x;
 x = 6 + 7;
 printf("x %d",x);
 return 0;
}
```

```
01010110101
01010101010
10101010101
01010100
```

a.out
(a binary executable)

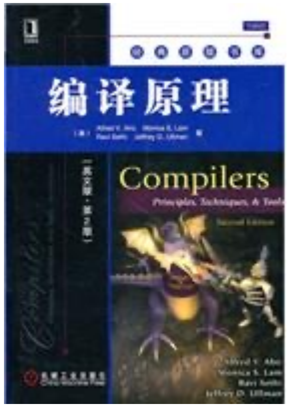
Operating System (OS)

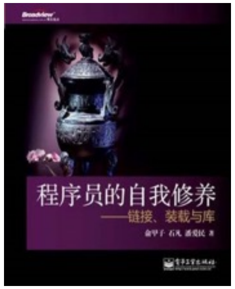
Computer Hardware (HW)

C: First compiled into a.out
Then direct execution of a.out

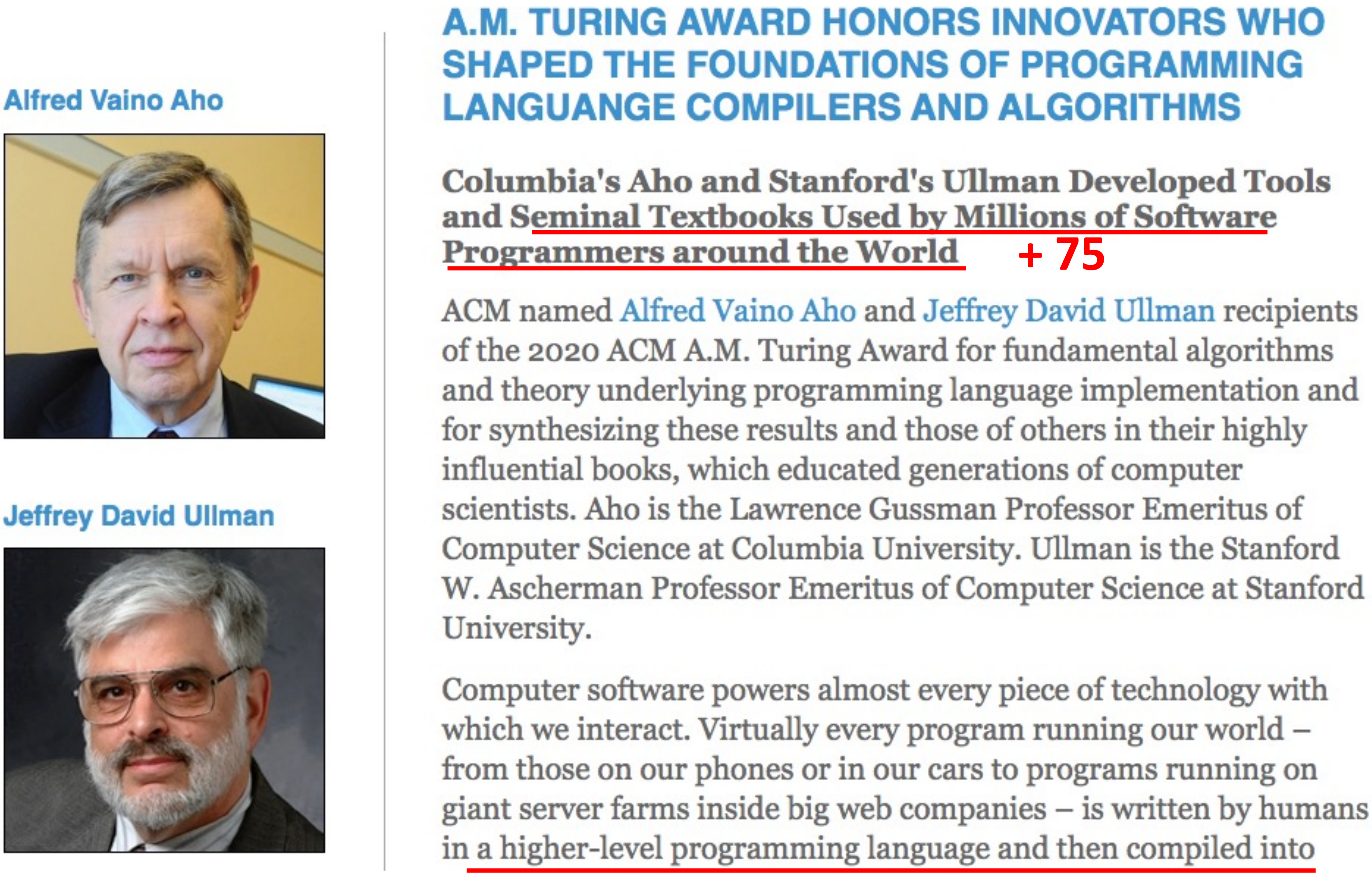
Textbook & Materials[教材]

- 主要教材
  - 编译原理（Compilers: Principles, Techniques, and Tools, 2nd Edition）, By Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
  - 《编译原理与技术》，张莉等（101计划教材）
  - LLVM编译器实战教程（Getting Started with LLVM Core Libraries）, Lopes and Auler
- 参考材料
  - 编译原理，陈鄞（哈工大）
  - CS 143, Fredrik Kjolstad (Stanford U.)
  - CS 411, Jan Hoffmann (Carnegie Mellon U.)
  - COMS 4115, Baishakhi Ray (Columbia U.)
  - CS 2210, Wonsun Ahn (U. of Pittsburgh)
  - Compilers and Computer Architecture, Martin Berger (U. of Sussex)
  - 程序员的自我修养 – 链接、装载与库





Turing Award[图灵奖’2020]



Alfred Vaino Aho

Jeffrey David Ullman

A.M. TURING AWARD HONORS INNOVATORS WHO SHAPED THE FOUNDATIONS OF PROGRAMMING LANGUANGE COMPILERS AND ALGORITHMS

Columbia's Aho and Stanford's Ullman Developed Tools and Seminal Textbooks Used by Millions of Software Programmers around the World

+ 75

ACM named Alfred Vaino Aho and Jeffrey David Ullman recipients of the 2020 ACM A.M. Turing Award for fundamental algorithms and theory underlying programming language implementation and for synthesizing these results and those of others in their highly influential books, which educated generations of computer scientists. Aho is the Lawrence Gussman Professor Emeritus of Computer Science at Columbia University. Ullman is the Stanford W. Ascherman Professor Emeritus of Computer Science at Stanford University.

Computer software powers almost every piece of technology with which we interact. Virtually every program running our world – from those on our phones or in our cars to programs running on giant server farms inside big web companies – is written by humans in a higher-level programming language and then compiled into

https://awards.acm.org/about/2020-turing

Instructor[任课教师]

博士，2011 – 2017，University of Pittsburgh
学士，2007 – 2011，西北工业大学

副教授，2020.10 – 今
工程师/研究员，2017.08 – 2020.09
实习研究员，2016.05 – 2016.08

计算机体系结构
高性能及智能计算
编程及编译优化

本科：编译原理/实验（21-23s）
本科：计算机体系结构（22f）
研究生：高级计算机体系结构（21-23f）

学院个人主页：http://sdcs.sysu.edu.cn/content/5592

Teaching Assistants[助教]

群聊名称 编译课-2024 >

席梦悦

陈俊瑜

黄瀚#

郑中淳#

潘文轩#

顾宇浩*

单招文*

郑腾扬^

孙高锦^

?

And more …

#：2023编译系统比赛
*：2023编译实验助教
^：2023编译实验课优秀

Time/Location[课时安排]

- **编译原理**（3学分，54学时）
  - 排课：1-18周
    - 周二：1-9周
    - 周四：1-18周
  - 每次授课包括2个课时
    - 第五节：14:20 - 15:05，第六节：15:15 - 16:00
  - 地点：教学大楼 C104

- **编译器构造实验**（1学分，36学时）
  - 排课：1-18周
    - 周四：1-18周
  - 每次实验包括2个课时
    - 第七节：16:30 - 17:15，第八节：17:25 - 18:10
  - 地点：实验中心 B202

Slides/Office Hours[课件及答疑]

- 课件
  - 英文为主，术语中文标注
    - 课后或课前上传
  - 主页：https://arcsysu.github.io/teach/dcs290/s2024.html
- 作业及实验提交
  - 超算习堂：https://easyhpc.net/course/164
- **课程QQ群： 189 205 980**
  - 通知提醒、答疑讨论
- 答疑
  - 理论课前课间，或实验课期间
    - 其他时间需预约
  - Email: zhangxw79@mail.sysu.edu.cn

sysu–编译(张)

群号: 189205980

扫一扫二维码，加入群聊

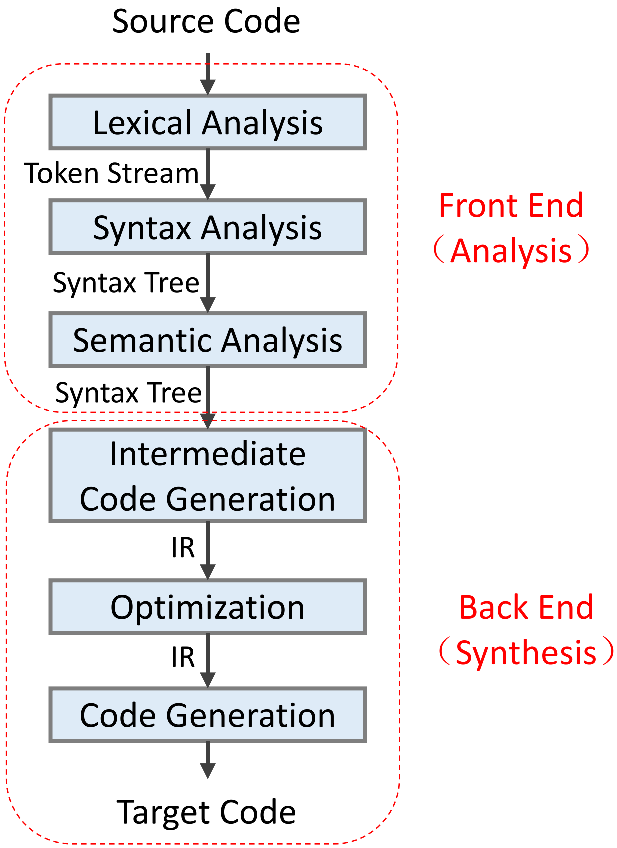
Grading[考核标准]

- **编译原理**
  - 课堂参与（15%）- 点名、提问、测试
  - 课程作业（25%）- 5次左右，理论
  - 期末考试（60%）- 闭卷
- **编译器构造实验**
  - 课堂参与（10%）- 签到、练习等
  - Project 1（20%）- Lexical Analysis
  - Project 2（20%）- Syntax/Semantic Analysis
  - Project 3（20%）- IR Generation
  - Project 4（30%）- Code Optimization

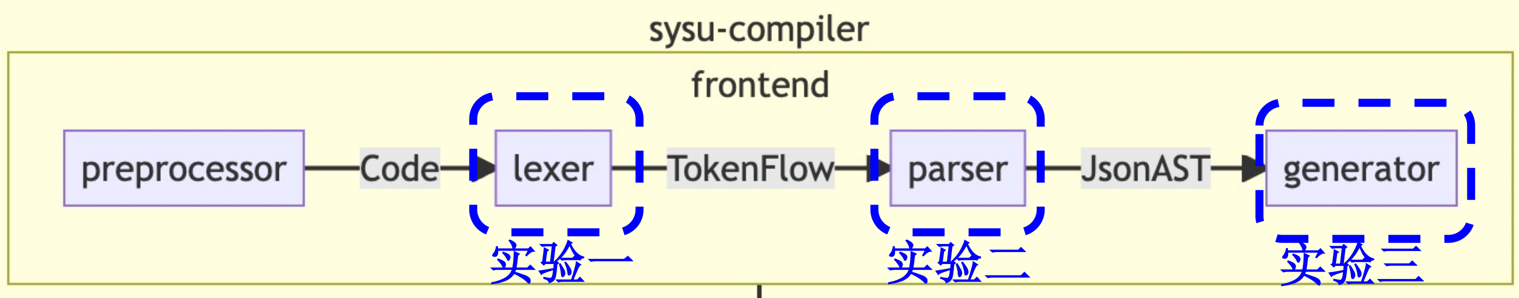
- 理论
  - 随机点名
    - 缺席优先
  - 随机提问
    - 后排优先
  - 随机测试
    - 不定时间
- 实验
  - 个人完成
    - 杜绝抄袭
  - 按时提交
    - 硬性截止
  - 侧重代码实现
    - 简略报告

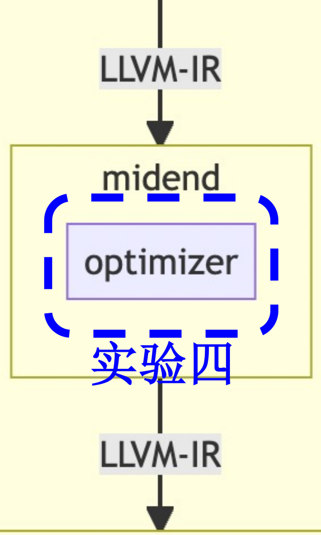
Schedule-Lec[理论安排]

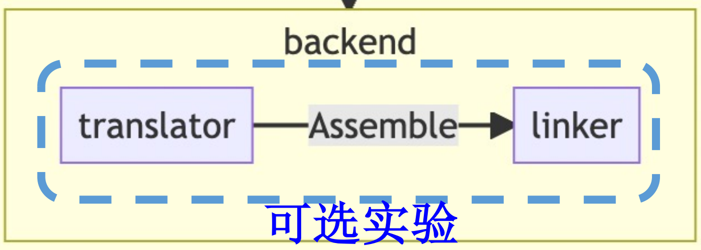
- **Lexical**: source code → tokens
  - RE, NFA, DFA, …
- **Syntax**: tokens → AST or parse tree
  - CFG, LL(1), LALR(1), …
- **Semantic**: AST → AST + symbol table
  - SDD, SDT, typing, scoping, …
- **Int. Code Generation**: AST → IR
  - TAC, offset, CodeGen, …
- **Optimization**: IR → (optimized) IR
  - BB, CFG, DAG, …
- **Code Generation**: IR → Instructions
  - Instruction, register, stack, …

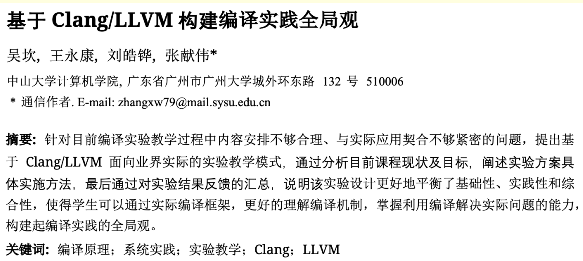


Schedule-Lab[实验安排]









基于 Clang/LLVM 构建编译实践全局观

吴坎，王永康，刘皓铧，张献伟*

中山大学计算机学院，广东省广州市广州大学城外环东路 132 号 510006

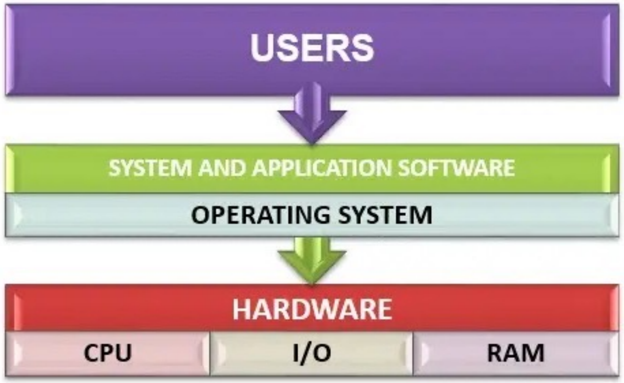
* 通信作者. E-mail: zhangxw79@mail.sysu.edu.cn

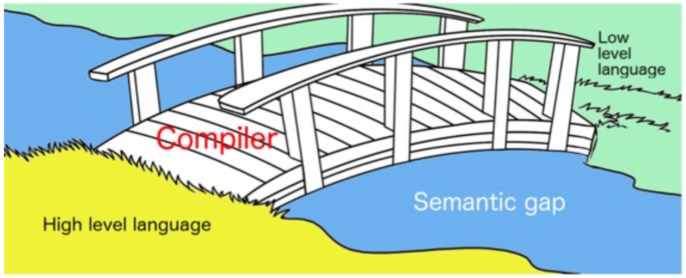
摘要：针对目前编译实验教学过程中内容安排不够合理、与实际应用契合不够紧密的问题，提出基于 Clang/LLVM 面向业界实际的实验教学模式，通过分析目前课程现状及目标，阐述实验方案具体实施方法，最后通过对实验结果反馈的汇总，说明该实验设计更好地平衡了基础性、实践性和综合性，使得学生可以通过实际编译框架，更好的理解编译机制，掌握利用编译解决实际问题的能力，构建起编译实践的全局观。

关键词：编译原理；系统实践；实验教学；Clang；LLVM

Compiler

- A compiler is system software
  - System software provides platforms for other software
- The elephant in the room
  - People are always use the compiler, but very few are paying much attention to it





The Elephant in the Room

https://clang.llvm.org/, https://clang.llvm.org/get_started.html

Compiler History[编译器的发展]

- Compiler origins
  - 1952: A-0, term ‘compiler’ (Grace Hopper)
  - 1957: FORTRAN, first commercial compiler (John Backus)
  - 1962: LISP, self-hosting and GC (Tim Hart and Mike Levin)
  - 1984: GNU Compiler Collection (Stallman)
  - 2000: LLVM (Vikram Adve and Chris Lattner)
- Turing awards (see link)
  - Compiler: 1966, 1987, 2006, 2020
  - Programming Language: 1972, 1974, 1977-1981, 1984, 2001, 2003, 2005, 2008
- Compilers today
  - Modern compilers are complex (gcc has 7M+ LOC)
  - There is still a lot of compiler research (LLVM, Pytorch, TVM, ...)
  - There are emerging compiler developments in industry

Why Compiler?[为什么要学习编译？]

- 计算机生态一直在改变
  - 新的硬件架构（通用GPU、AI加速器等）
  - 新的程序语言（Rust、Go等）
  - 新的应用场景（ML、IoT等）
- 了解编译程序的实现原理与技术
  - 掌握编译程序/系统设计的基本原理
  - 理解高级语言程序的内部运行机制
    - 提高编写和调试程序的能力
  - 培养形式化描述和抽象思维能力
- 大量专业工作与编译技术相关
  - 高级语言实现、软硬件设计与优化、软件缺陷分析
- 硕博士阶段从事与编译相关的研究
  - 尽管可能并不是直接的编译或程序设计方向

clang

GCC

MLIR

tvm

方舟编译器
多端多语言，轻量低开销

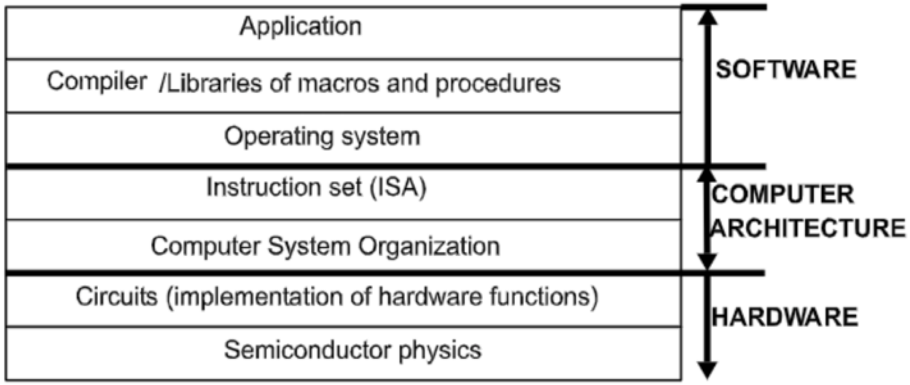
PyTorch

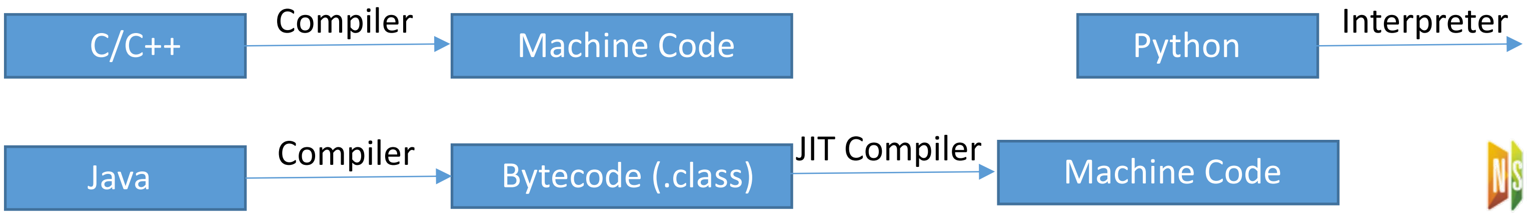
中山大学
SUN YAT-SEN UNIVERSITY

NSCC GZ

What is Compilation?[什么是编译？]

- 高级语言编写程序，但计算机只理解0/1
  - 自然语言翻译：”This is a sentence” → “这是一个句子”
  - 计算机语言翻译：源程序 → 目标程序
  - 编程人员专注于程序设计，无需过多考虑机器相关的细节
- 不同语言有不同的实现方式
  - “底层” 语言通常使用编译
    - □ C, C++
  - “高级” 语言通常是解释性
    - □ Python, Ruby
  - 有些使用混合的方式
    - □ Java：编译 + 即时编译（JIT, Just-in-Time）

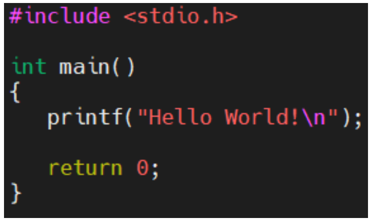

Application
Compiler /Libraries of macros and procedures
Operating system
Instruction set (ISA)
Computer System Organization
Circuits (implementation of hardware functions)
Semiconductor physics
SOFTWARE
COMPUTER ARCHITECTURE
HARDWARE



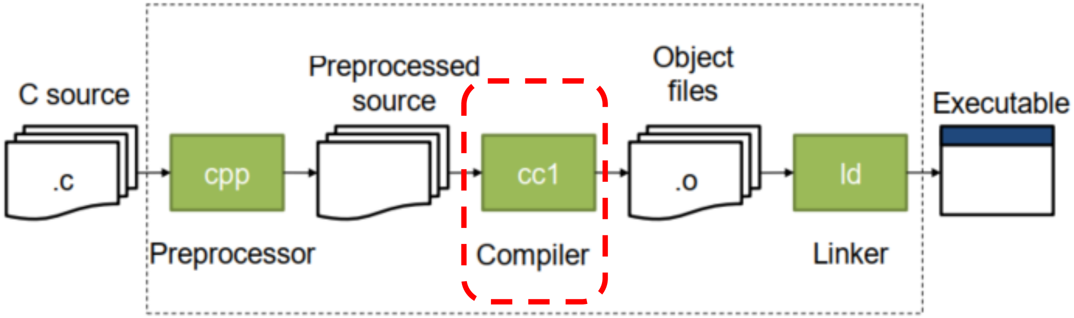
C Compilation[C语言编译]

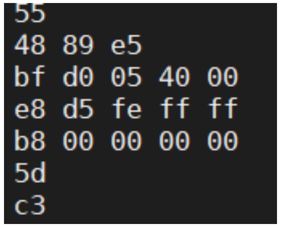
```
$ clang hello.c -o hello
$ ./hello
```

- 源程序 (hello.c) → 可执行文件 (./hello)
  - 预处理阶段（preprocessor）
    - 汇合源程序，展开宏定义，生成*.i*文件（另一个C文本文件）
  - 编译阶段（compiler）
    - *.i*文件翻译为*.s*文件（汇编代码）
  - 汇编阶段（assembler）
    - *.s*文件转为*.o*可重定位对象（relocatable object）文件 （机器指令）
  - 连接阶段（linker/loader）
    - 连接库代码从而生成可执行（executable）文件（机器指令）



```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");

    return 0;
}
```



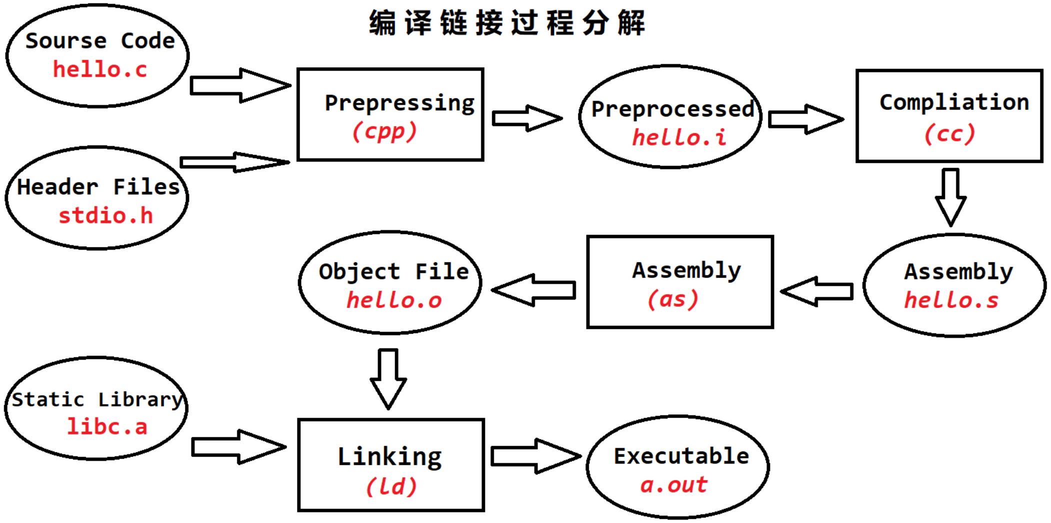


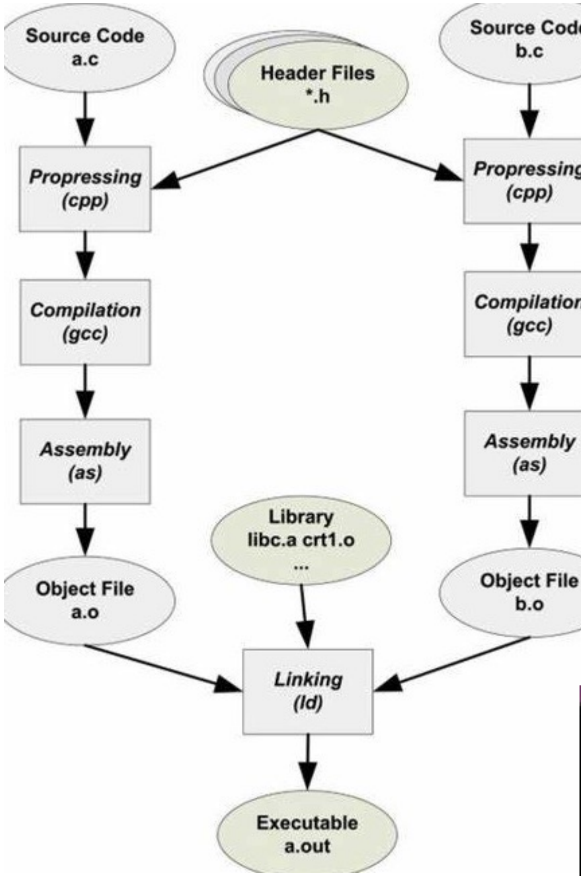
```
55
48 89 e5
bf d0 05 40 00
e8 d5 fe ff ff
b8 00 00 00 00
5d
c3
```

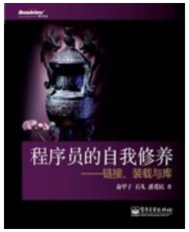
```
clang -E hello.c -o hello.i
clang -emit-llvm hello.i -S -o hello.ll
clang -S hello.ll -o hello.s
clang -c hello.s -o hello.o
clang hello.o -o hello
```

C Compilation (cont.)

- Preprocessing: 源文件→处理后的源文件
- Compiling: 处理后的源文件→汇编代码文件
- Assembling: 汇编代码文件→目标文件/机器指令文件
- Linking: 目标文件→可执行文件







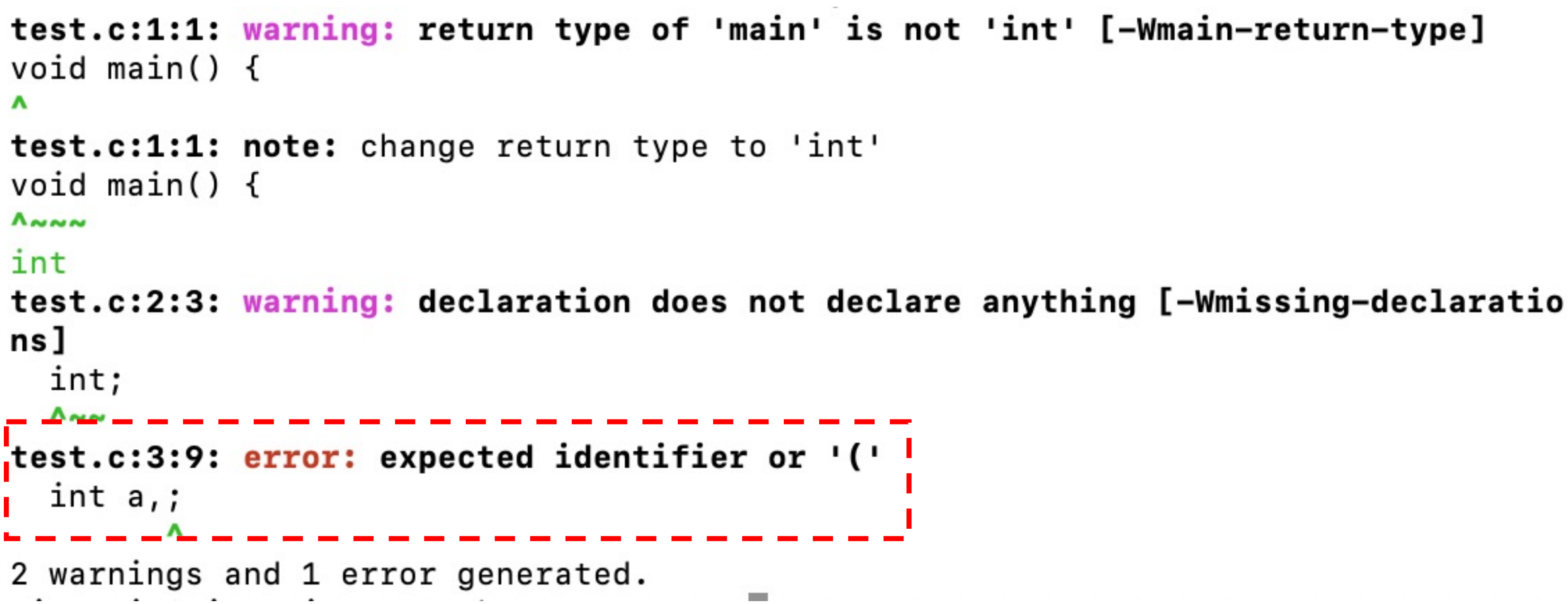
程序员的自我修养 – 链接、装载与库，第2章

Example

- $vim test.c

```
void main() {
  int;
  int a,;
  int b, c;
}
```

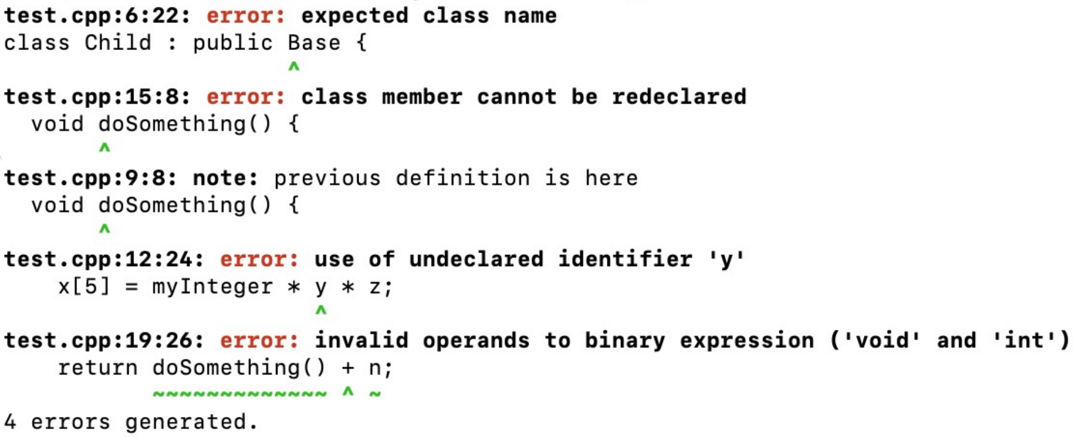
- $clang -o test test.c



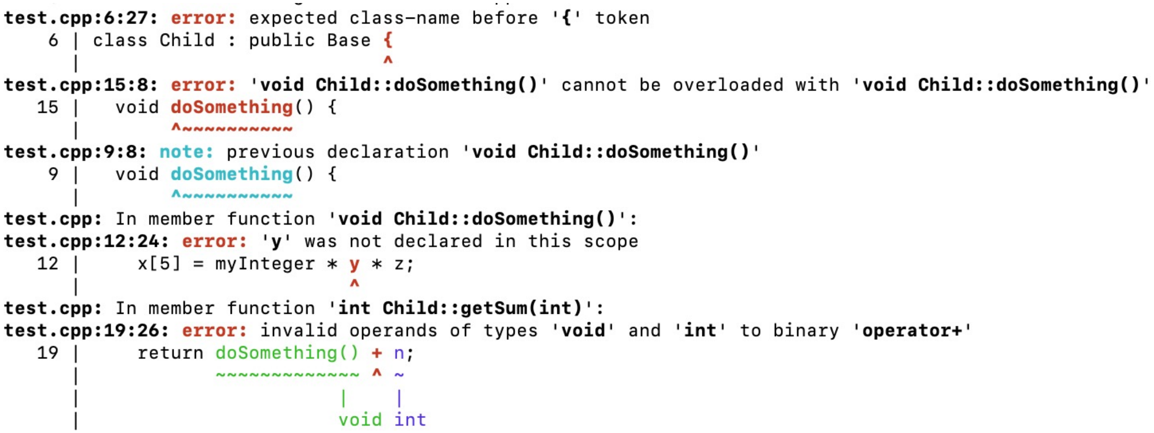
```
test.c:1:1: warning: return type of 'main' is not 'int' [-Wmain-return-type]
void main() {
^
test.c:1:1: note: change return type to 'int'
void main() {
^~~~
int
test.c:2:3: warning: declaration does not declare anything [-Wmissing-declaratio
ns]
  int;
  ^~~
test.c:3:9: error: expected identifier or '('
  int a,;
        ^
2 warnings and 1 error generated.
```

Example (cont.)

```cpp
#include <iostream>

using namespace std;

//Derived class
class Child : public Base {
  string myInteger;

  void doSomething() {
    int x[] = {0, 1, 2, 3, 4};
    int z = 'a';
    x[5] = myInteger * y * z;
  }

  void doSomething() {
  }

  int getSum(int n) {
    return doSomething() + n;
  }
};
```



```
test.cpp:6:22: error: expected class name
class Child : public Base {
                     ^
test.cpp:15:8: error: class member cannot be redeclared
  void doSomething() {
       ^
test.cpp:9:8: note: previous definition is here
  void doSomething() {
       ^
test.cpp:12:24: error: use of undeclared identifier 'y'
    x[5] = myInteger * y * z;
                       ^
test.cpp:19:26: error: invalid operands to binary expression ('void' and 'int')
    return doSomething() + n;
           ~~~~~~~~~~~~~ ^ ~
4 errors generated.
```

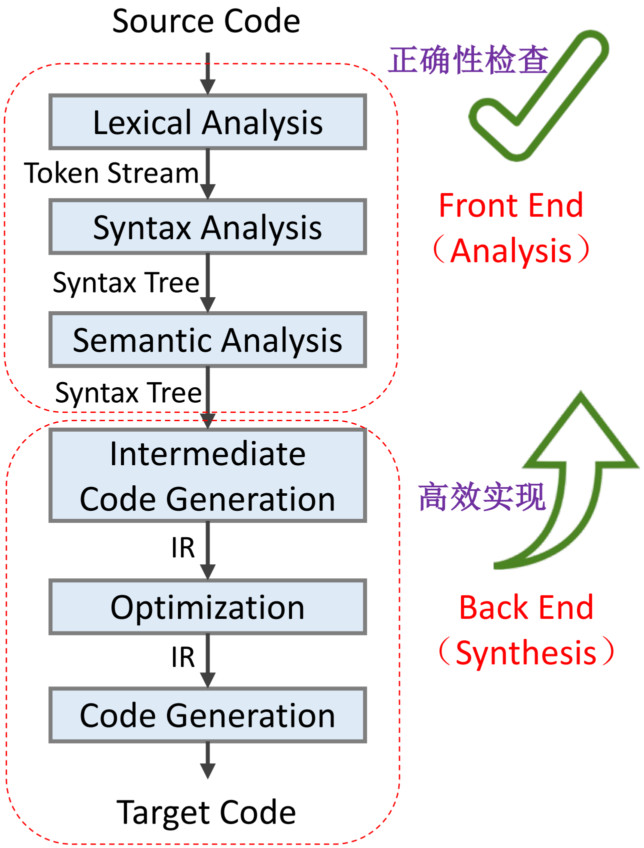


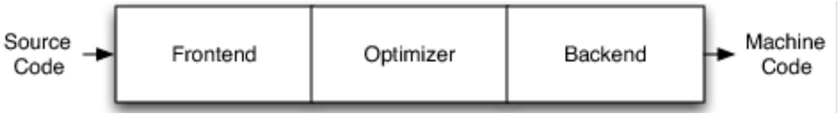
```
test.cpp:6:27: error: expected class-name before '{' token
    6 | class Child : public Base {
      |                           ^
test.cpp:15:8: error: 'void Child::doSomething()' cannot be overloaded with 'void Child::doSomething()'
   15 |   void doSomething() {
      |        ^~~~~~~~~~~
test.cpp:9:8: note: previous declaration 'void Child::doSomething()'
    9 |   void doSomething() {
      |        ^~~~~~~~~~~
test.cpp: In member function 'void Child::doSomething()':
test.cpp:12:24: error: 'y' was not declared in this scope
   12 |     x[5] = myInteger * y * z;
      |                        ^
test.cpp: In member function 'int Child::getSum(int)':
test.cpp:19:26: error: invalid operands of types 'void' and 'int' to binary 'operator+'
   19 |     return doSomething() + n;
      |            ~~~~~~~~~~~~~ ^ ~
      |                      |    |
      |                      void int
```

https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/08/Slides08.pdf

Compilation Procedure[编译过程]

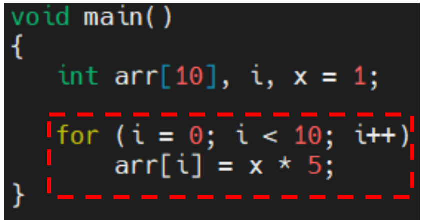
- **前端**（分析）：对源程序，识别语法结构信息，理解语义信息，反馈出错信息
  - 词法分析（Lexical Analysis）
  - 语法分析（Syntax Analysis）
  - 语义分析（Semantic Analysis）
- **后端**（综合）：综合分析结果，生成语义上等价于源程序的目标程序
  - 中间代码生成（Intermediate Code Generation）
    - Intermediate representation (IR)
  - 代码优化（Code Optimization）
  - 目标代码生成（Code Generation）


Source Code
Lexical Analysis
Token Stream
Syntax Analysis
Syntax Tree
Semantic Analysis
Syntax Tree
Intermediate Code Generation
IR
Optimization
IR
Code Generation
Target Code
正确性检查
Front End（Analysis）
高效实现
Back End（Synthesis）


Source Code
Frontend
Optimizer
Backend
Machine Code

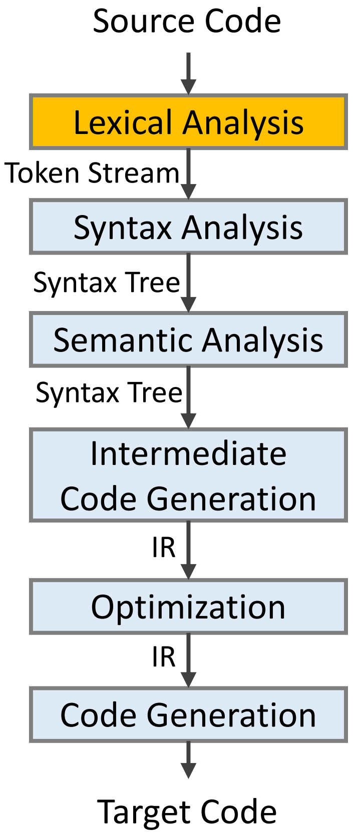
Lexical Analysis[词法分析]

- 扫描源程序字符流，识别并分解出有词法意义的单词或符号（token）
  - 输入：源程序，输出：token序列
  - token表示：<类别，属性值>
    - 保留字、标示符、常量、运算符等
  - token是否符合词法规则？
    - 0var, $num



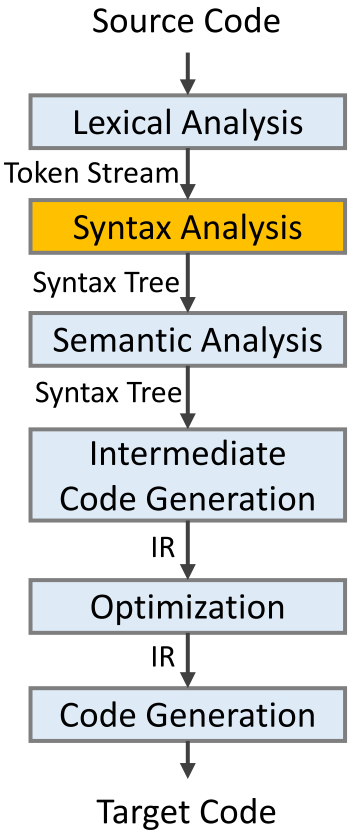
```
void main()
{
    int arr[10], i, x = 1;
    for (i = 0; i < 10; i++)
        arr[i] = x * 5;
}
```

keyword(for)
sym(()
id(i)
sym(=)
num(0)
sym(;)
id(i)
sym(<)
num(10)
sym(;)
id(i)
sym(++)
sym())
id(arr)
sym([)
id(i)
sym(])
sym(=)
id(x)
sym(*)
num(5)
symbol(;)



Syntax Analysis[语法分析]

- 解析源程序对应的token序列，生成语法分析结构（syntax tree, 语法分析树）
  - 输入：单词流，输出：语法树
  - 输入程序是否符合语法规则？
    - x*+
    - a += 5~~;~~



Source Code

Lexical Analysis

Token Stream

Syntax Analysis

Syntax Tree

Semantic Analysis

Syntax Tree

Intermediate Code Generation

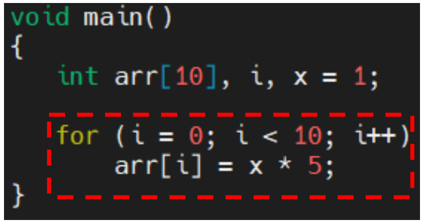
IR

Optimization

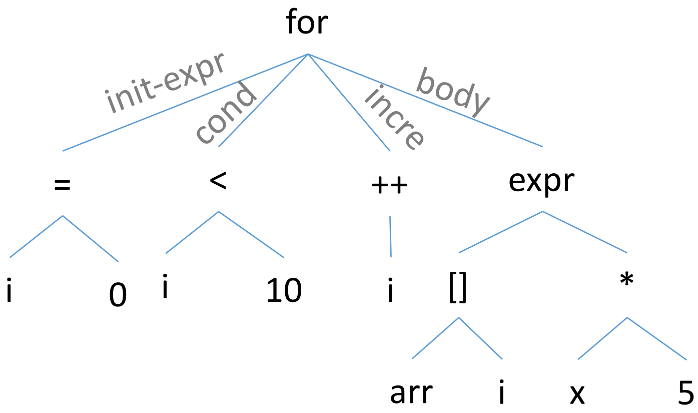
IR

Code Generation

Target Code

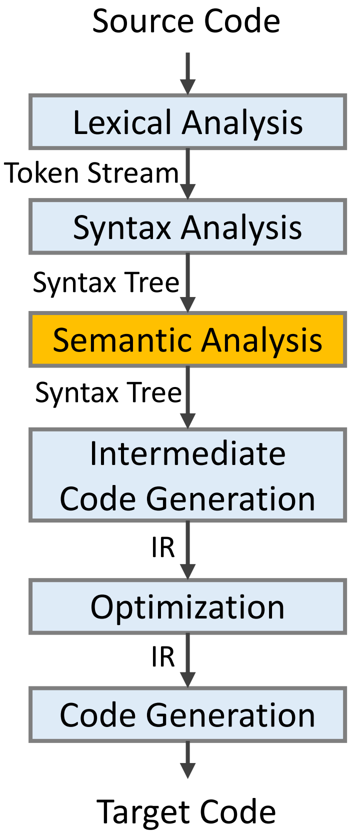


```
void main()
{
    int arr[10], i, x = 1;

    for (i = 0; i < 10; i++)
        arr[i] = x * 5;
}
```



Semantic Analysis[语义分析]

- 基于语法结果进一步分析语义
  - 输入：语法树，输出：语法树+符号表
  - 收集标识符的属性信息（type, scope等）
  - 输入程序是否符合语义规则？
    - 变量未声明即使用；重复声明
    - int x; y = x(3);


Source Code

Lexical Analysis

Token Stream

Syntax Analysis

Syntax Tree

Semantic Analysis

Syntax Tree

Intermediate Code Generation

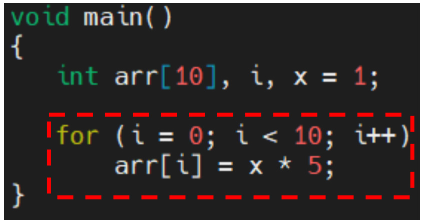
IR

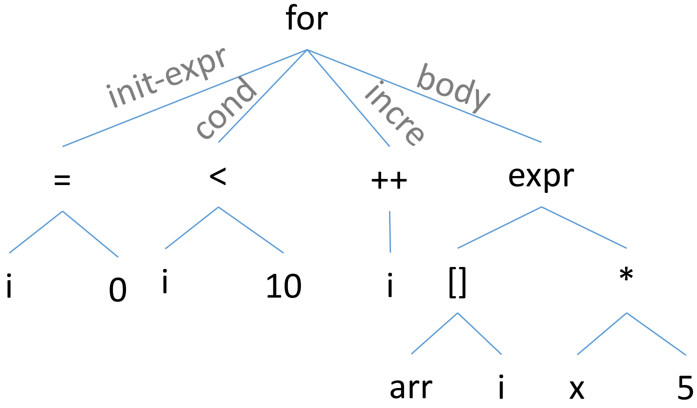
Optimization

IR

Code Generation

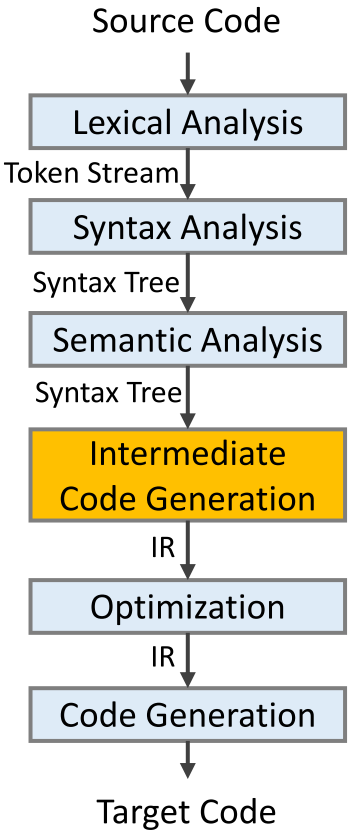
Target Code

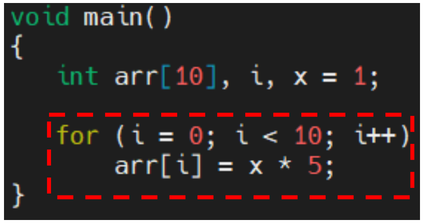

```
void main()
{
    int arr[10], i, x = 1;
    for (i = 0; i < 10; i++)
        arr[i] = x * 5;
}
```



Intermediate Code[中间代码生成]

- 初步翻译，生成等价于源程序的中间表示（IR）
  - 输入：语法树，输出：IR
  - 建立源和目标语言的桥梁，易于翻译过程的实现， 利于实现某些优化算法
  - IR形式：通常三地址码（TAC）


Source Code
Lexical Analysis
Token Stream
Syntax Analysis
Syntax Tree
Semantic Analysis
Syntax Tree
Intermediate Code Generation
IR
Optimization
IR
Code Generation
Target Code

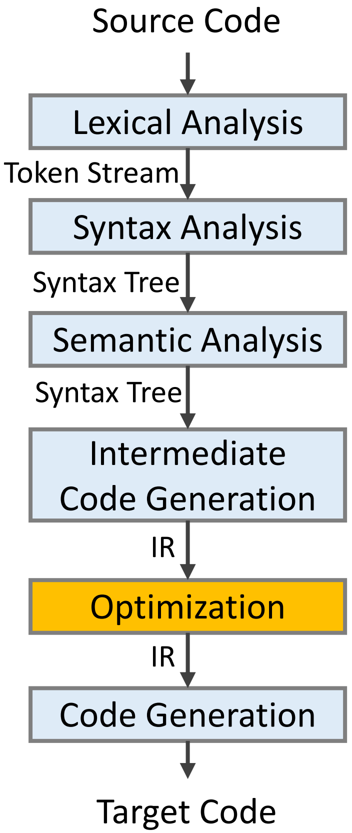


```
void main()
{
    int arr[10], i, x = 1;

    for (i = 0; i < 10; i++)
        arr[i] = x * 5;
}
```

```
i := 0
loop:
    t1 := x * 5
    t2 := &arr
    t3 := sizeof(int)
    t4 := t3 * i
    t5 := t2 + t4
    *t5 := t1
    i := i + 1
    if i < 10 goto loop
```

Code Optimization[代码优化]

- 加工变换中间代码使其更好（例如，代码更短、性能更高、内存使用更少）
  - 输入：IR，输出：（优化的）IR
  - 机器无关（machine independent）
  - 例如：设别重复运算并删除；运算操作替换；使用已知量



Source Code

Lexical Analysis

Token Stream

Syntax Analysis

Syntax Tree

Semantic Analysis

Syntax Tree

Intermediate Code Generation

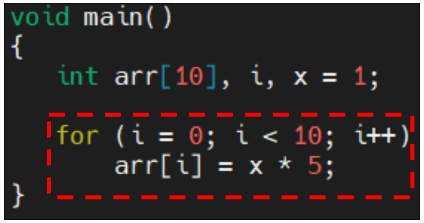
IR

Optimization

IR

Code Generation

Target Code

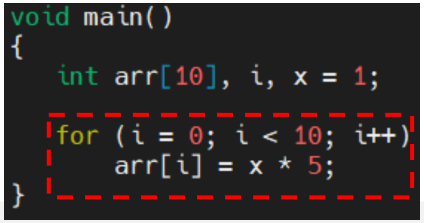


```
void main()
{
    int arr[10], i, x = 1;

    for (i = 0; i < 10; i++)
        arr[i] = x * 5;
}
```

```
i := 0
loop:
    t1 := x * 5
    t2 := &arr
    ~~t3 := sizeof(int)~~
    t4 := t3 * i
    t5 := t2 + t4
    *t5 := t1
    i := i + 1
    if i < 10 goto loop
```

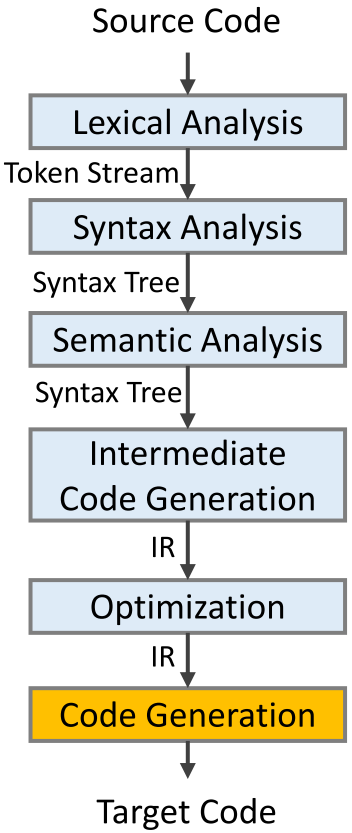
Target Code[目标代码生成]

- 为特定机器产生目标代码（e.g., 汇编）
  - 输入：（优化的）IR，输出：目标代码
  - 寄存器分配：放置频繁访问数据
  - 指令选取：确定机器指令实现IR操作
  - 进一步的机器有关优化
    - 例如：寄存器及访存优化



```c
void main()
{
    int arr[10], i, x = 1;

    for (i = 0; i < 10; i++)
        arr[i] = x * 5;
}
```

```
… …
14:  8b 55 f8        mov   -0x8(%rbp),%edx          // edx = x
17:  89 d0           mov   %edx,%eax                // eax = x
19:  c1 e0 02        shl   $0x2,%eax                // eax  = (x << 2)
1c:  01 c2           add   %eax,%edx                // edx = (x << 2) + x
1e:  8b 45 fc        mov   -0x4(%rbp),%eax          // eax = i
21:  48 98           cltq
23:  89 54 85 d0     mov   %edx,-0x30(%rbp,%rax,4)  // arr[i] = 5x
27:  83 45 fc 01     addl  $0x1,-0x4(%rbp)          // i++
2b:  83 7d fc 09     cmpl  $0x9,-0x4(%rbp)          // i <= 9
2f:  7e e3           jle   14 <main+0x14>           // loop end?
… …
```



Source Code

Lexical Analysis

Token Stream

Syntax Analysis

Syntax Tree

Semantic Analysis

Syntax Tree

Intermediate Code Generation

IR

Optimization

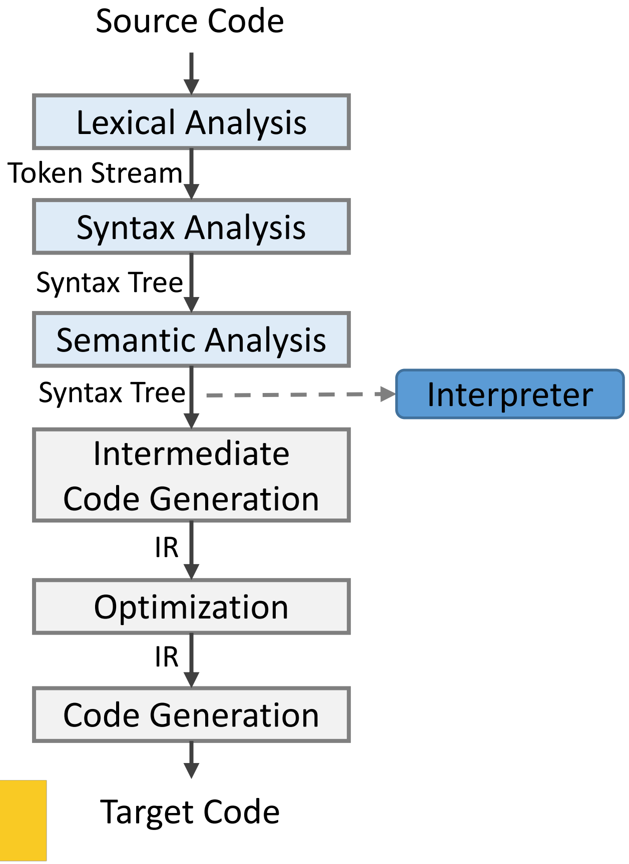
IR

Code Generation

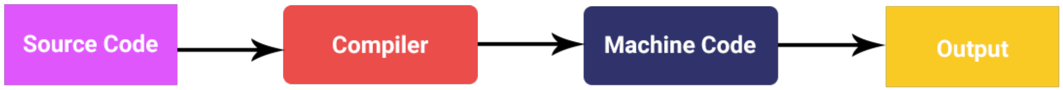
Target Code

Interpret vs Compile[解释 vs. 编译]

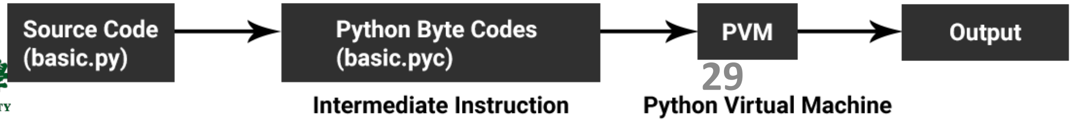
- **编译：** 翻译成机器语言后方能运行
  - 目标程序独立于源程序（修改 → 再编译 → 运行）
  - 分析程序上下文，易于整体性优化
  - 性能更好（因此，核心代码通常C/C++）
- **解释：** 源程序作为输入，边解释边执行
  - 不生成目标程序，可迁移性高
  - 逐句执行，很难进行优化
  - 性能通常不会太好


Source Code
Lexical Analysis
Token Stream
Syntax Analysis
Syntax Tree
Semantic Analysis
Syntax Tree
Interpreter
Intermediate Code Generation
IR
Optimization
IR
Code Generation
Target Code

Compiler Works

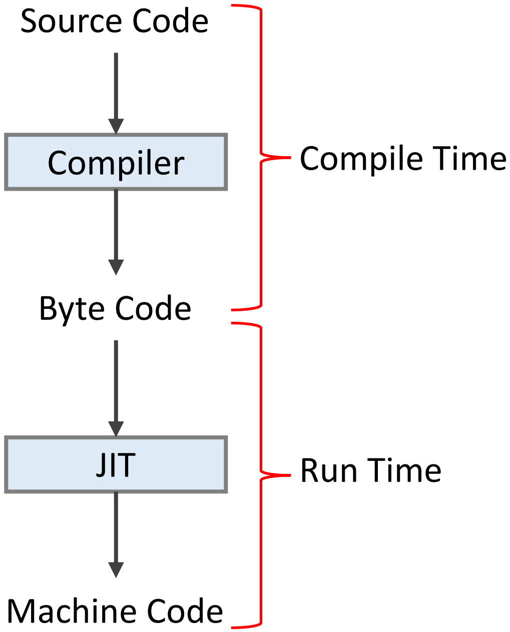

Source Code
Compiler
Machine Code
Output

Interpreter Works


Source Code (basic.py)
Python Byte Codes (basic.pyc)
Intermediate Instruction
PVM
Python Virtual Machine
Output

JIT[即时编译]

- 即时编译（Just-In-Time Compiler）：运行时执行程序编译操作
  - 弥补解释执行的不足
    - 把翻译过的机器代码保存起来，以备下次使用
  - 传统编译（AOT, Ahead-Of-Time）：先编译后运行
- JIT vs. AOT
  - JIT具备解释器的灵活性
    - 只要有JIT编译器，代码即可运行
  - 性能上基本和AOT等同
    - 运行时编译操作带来一些性能上的损失
    - 但可以利用程序运行特征进行动态优化


Source Code

Compiler

Byte Code

JIT

Machine Code

Compile Time

Run Time