中山大學
SUN YAT-SEN UNIVERSITY

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Compilation Principle
编 译 原 理

第0讲：课程介绍、概述

张献伟

xianweiz.github.io

DCS290, 2/17/2022

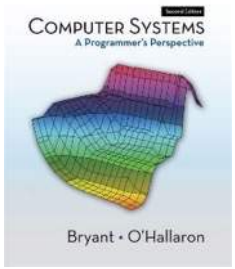
The Course[关于课程]

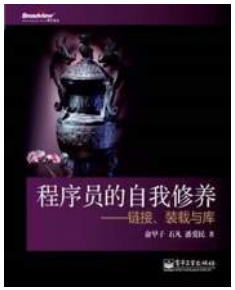
- 年级专业
  - 19级计科/1班（李文军，62人）
  - 19级计科/2班（张献伟，57人）
  - 19级计科/超算+人工大数据（沈明华，55人）
- 先修课程
  - 计算机组成原理/体系结构、汇编语言
  - 离散数学、数据结构、C/C++或其他编程语言
- 编译原理
  - 高级编程语言（如C）是如何转换为机器语言（0/1）的？
  - 介绍编译器设计与实现的主要理论和技术
    - 包括词法分析、语法分析、语义分析、代码生成、代码优化等
- 编译器构造实验
  - 单独课程，分阶段实现一个小型编译器
    - 词法分析、语法分析、语义分析等

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

- 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)
  - Computer Systems: A Programmer's Perspective (CSAPP), Bryant and O'Hallaron
  - 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 + 57

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

Instructor[任课教师]

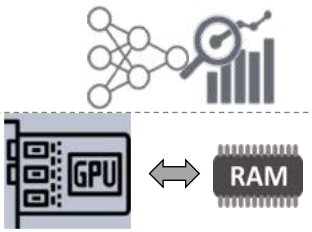
博士，2011 – 2017， University of Pittsburgh

学士，2007 – 2011，西北工业大学

中山大學 副教授，2020.10 – 今

AMD 工程师/研究员，2017.08 – 2020.09

NVIDIA 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（2021春）

研究生：高级计算机体系结构（2021秋）

Slides/Office Hours[课件及答疑]

- 课件（英文为主，术语中文标注）
  - 主页：xianweiz.github.io/teach/dcs290/s2022.html
  - 课后或课前上传
- 作业提交
  - 超算习堂：https://easyhpc.net/course/144
- **课程QQ群：687 642 753**
- 教师
  - 张献伟 （超算中心）
    - Email: zhangxw79@mail.sysu.edu.cn
    - 实验课答疑，其他时间需预约
- 助教
  - 理论：邓志涛（博士，超算中心）
    - Email：dengzht6@mail2.sysu.edu.cn
  - 实验：吴坎（学硕，超算中心）
    - Email：wukan3@mail2.sysu.edu.cn

Time/Location[课时安排]

- **编译原理**（3学分，54学时）
  - 排课：1-19周
    - 周二：2-9，11-19周
    - 周四：1-9，11-19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 C203

- **编译器构造实验**（1学分，36学时）
  - 排课：1-19周
    - 周四：1-9，11-19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 D402

Grading[考核标准]

- **编译原理**
  - 课堂参与（10%） - 点名、提问、测试
  - 课程作业 （30%）- 4次左右，理论
  - 期末考试 （60%）- 闭卷

- **编译器构造实验**
  - 课堂参与（12%）- 签到、练习等
  - Project 1（22%）- Lexical Analysis
  - Project 2（22%）- Syntax Analysis
  - Project 3（22%）- Semantic Analysis
  - Project 4（22%）- Code Generation

- 理论
  - 随机点名
    - 缺席优先
  - 随机提问
    - 后排优先
  - 随机提问
    - 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
    - 无需报告

Schedule[内容安排]

周次	课程内容	周次	课程内容
第1周 (02.17)	四：课程介绍、编译概述	第11周 (04.28)	四：中间代码 – IR表示
第2周 (02.22/24)	二：词法分析 – 过程、正则表达 四：词法分析 – NFA、DFA （1）	第12周 (05.05)	四：中间代码 – 生成
第3周 (03.01/03)	二：词法分析 – NFA、DFA （2） 四：语法分析 – 语法、语言、CFG	第13周 (05.12)	四：运行时管理
第4周 (03.08/10)	二：语法分析 – 自顶向下、文法 （1） 四：语法分析 – 自顶向下、文法 （2）	第14周 (05.19)	四：目标代码
第5周 (03.15/17)	二：语法分析 – 自底向上、LR分析 （1） 四：语法分析 – 自底向上、LR分析 （2）	第15周 (05.26)	四：目标代码生成
第6周 (03.22/24)	二：语法分析 – LR(0)、SLR 四：语法分析 – LALR（1）	第16周 (06.02)	四：代码优化（1）
第7周 (03.29/31)	二：语法分析 – LALR（2） 四：语法制导翻译 （1）	第17周 (06.09)	四：代码优化（2）
第8周 (04.05/07)	二：语法制导翻译 （1） 四：语义分析 – 符号表	第18周 (06.16)	四：前沿编译技术
第9周 (04.12/14)	二：语义分析 – 类型检查 四：回顾 – 编译前端	第19周 (06.23)	四：复习
第10周	**（期中：课下考查）**	**第20周**	**（期末：闭卷考试）**

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)
- 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, ...)
  - There is still a lot of compiler development 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（DL、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

clang

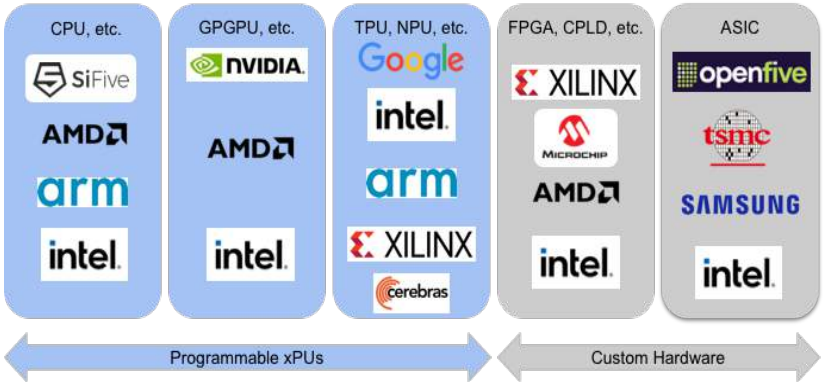
tvm

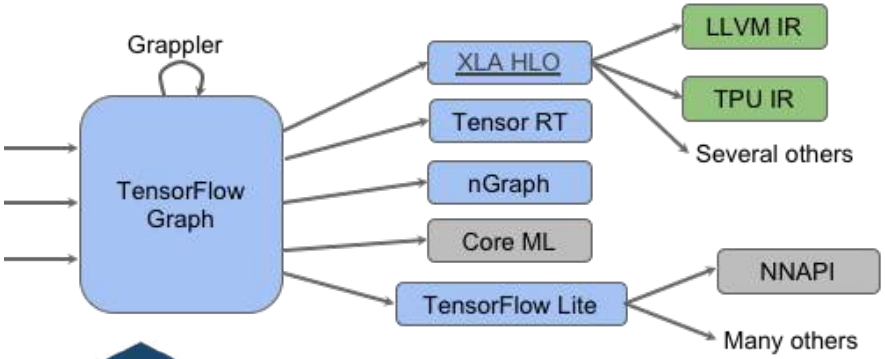
方舟编译器

多端多语言，轻量低开销

- 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

Golden Age of Compilers[黄金时代]





https://tenstorrent.com/research/asplos-keynote-the-golden-age-of-compiler-design-in-an-era-of-hw-sw-co-design-by-dr-chris-lattner/

Jobs[职位]

HUAWEI

编译器研究员- 校园招聘

Huawei · Hangzhou, Zhejiang, China (On-site) 1 month ago

About the job

Job Responsibilities

加入全球最领先的编译器团队，成为静态化编译、异构编译、GPU编译、AI编译技术的引领者和开创者，创造性地解决终端、云计算等软件的跨平台、性能等难点和痛点问题： 1、编译器设计与开发，编译优化和内存管理算法研究，解决终端产品、服务器GPU等性能挑战问题，通过编译优化提升终端系统和GPU极致能效比； 2、负责毕昇编译器、方舟编译器中后端算法优化及后端架构设计与开发； 3、负责人工智能/计算机视觉领域编译技术的应用场景分析、构架设计与技术落地； 4、面向开放场景通用CPU（RISC-V）的编译器技术和新的编译技术研究。

Position Requirements

1、具有扎实的计算机基础知识，有较强的软件编码功底（C/C++、体系结构、数据结构及算法等）； 2、了解编译器、编译优化、工具链、调试工具、安卓系统、JDK、JVM等任意一点相关的基本知识； 3、熟悉LLVM、GCC等编译器源码，熟悉编译器IR、Pass或性能调优者优先； 4、具有Profile、仿真、处理器建模、二进制翻译等工具的使用及开发经验优先。

15851–游戏编译器工程师

Tencent 腾讯招聘

IEG | 深圳 | 技术 | 2022年01月05日

工作职责

针对大型工程的代码生成以及编译器优化分析调优；

针对游戏GPU Shader编译进行优化；

针对对编译器中间语言表达机制有深入的理解，有相关设计与优化经验；

具有较强的C/C++开发能力，有丰富的问题分析定位与调试经验；

工作要求

熟悉编译器架构与算法，有设计与实现编译各阶段的经验，包括语言处理，编译器优化和代码生成等；

熟悉业界主流的编译器技术与框架，如LLVM，GCC；

对编译器中间语言表达机制有深入的理解，有相关设计与优化经验；

较强的C/C++开发能力，有丰富的问题分析定位与调试经验；

熟悉GPU Shader compiler开发与调优经验者优先；

熟悉OpenGL(ES)/Vulkan/Metal设计规范经验者优先；

有责任心，良好的团队合作能力和沟通能力。

NVIDIA

LLVM Compiler Optimization Engineer - New College Grad

NVIDIA · Shanghai, Shanghai, China 2 weeks ago · 2 applicants

About the job

We are looking to hire a LLVM Compiler Engineer for an exciting role developing compilers for our world leading GPUs. We craft outstanding compilers that realize the potential of NVIDIA GPU hardware for growing range of computational workloads, ranging from next generate graphics, deep learning, scientific computing, and self-driving cars. Our compiler organization makes its mark on every GPU and SoC product that NVIDIA builds. Would you like to be part of this best-in-class organization and lead the charge?

What You'll Be Doing

- Understand, modify, and improve an NVIDIA proprietary GPU compiler backend written in C++
- Contribute to compiler optimizations to produce best-in-class, robust, supportable compiler and tools
- Work on challenging problems in register allocation, instruction scheduling, synchronization, loop optimizations etc.
- Partner with global compiler, hardware and application teams to oversee improvements and problem resolutions
- Be part of a team that is at the center of deep-learning compiler technology spanning architecture design and support through functional languages

What We Need To See

- B.S. or higher degree in Computer Science/Engineering with significant compiler related project or thesis work
- Excellent C and C++ programming skills,

Ways For You To Stand Out From The Crowd

- Experience developing CUDA, DirectX, OpenGL/Vulkan applications

NVIDIA is committed to developing a diverse work environment and proud to be an equal opportunity employer. As we highly value diversity in our current and future employees, we do not discriminate (including in our hiring and promotion practices) on the basis of race, religion, color, national origin, gender, gender expression, sexual orientation, age, marital status, veteran status, disability status or any other characteristic protected by law.

Meta

Research Scientist, PyTorch Compiler (University Grad)

Meta · Menlo Park, CA

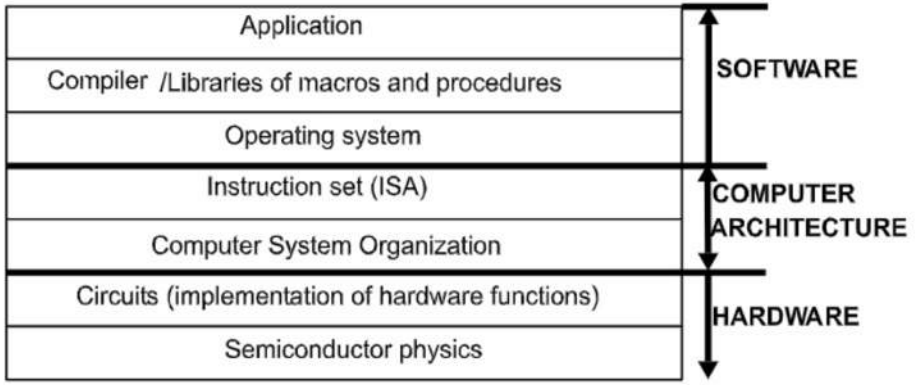
amazon

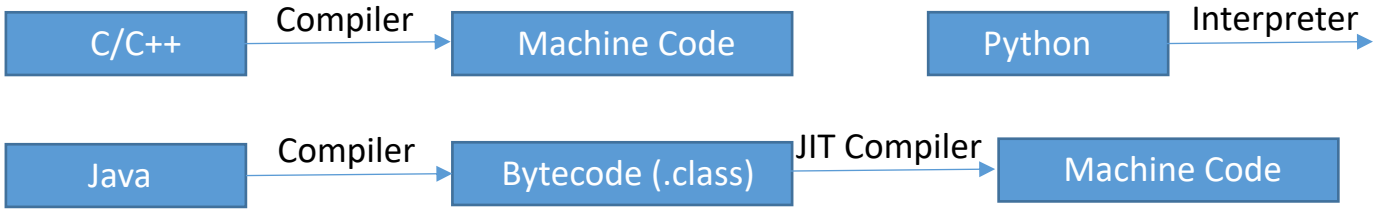
Deep Learning Compiler Engineer

Job ID: 1735730 | Annapurna Labs (U.S.) Inc.

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）



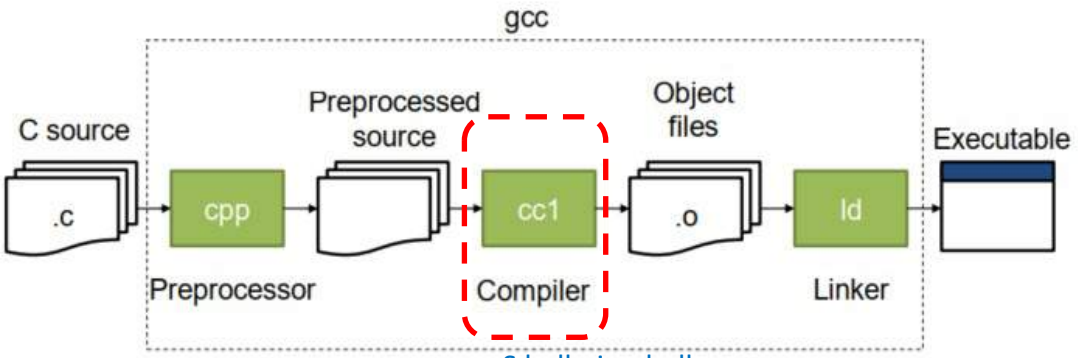


C Compilation[C语言编译]

- 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）

```
$ gcc hello.c –o hello
$ ./hello
```

```c
#include <stdio.h>

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

   return 0;
}
```



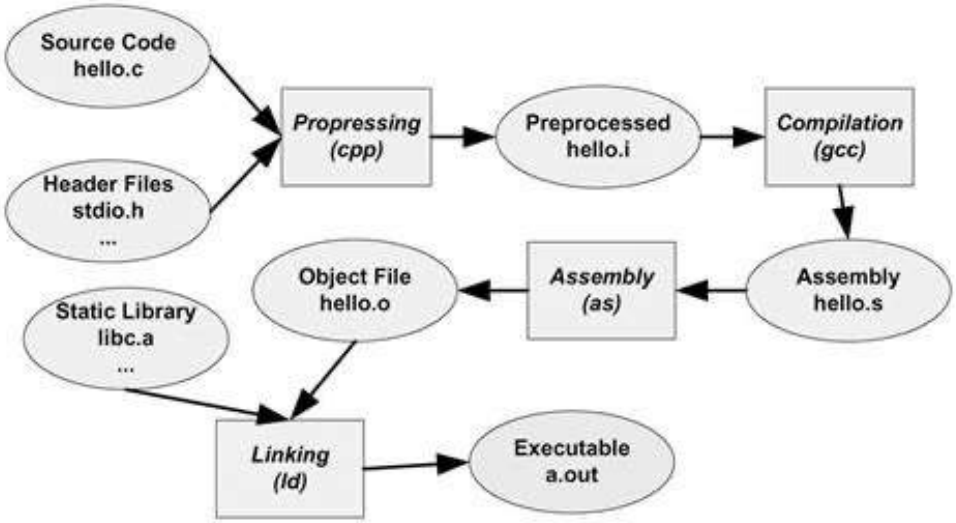

```
55
48 89 e5
bf d0 05 40 00
e8 d5 fe ff ff
b8 00 00 00 00
5d
c3
```

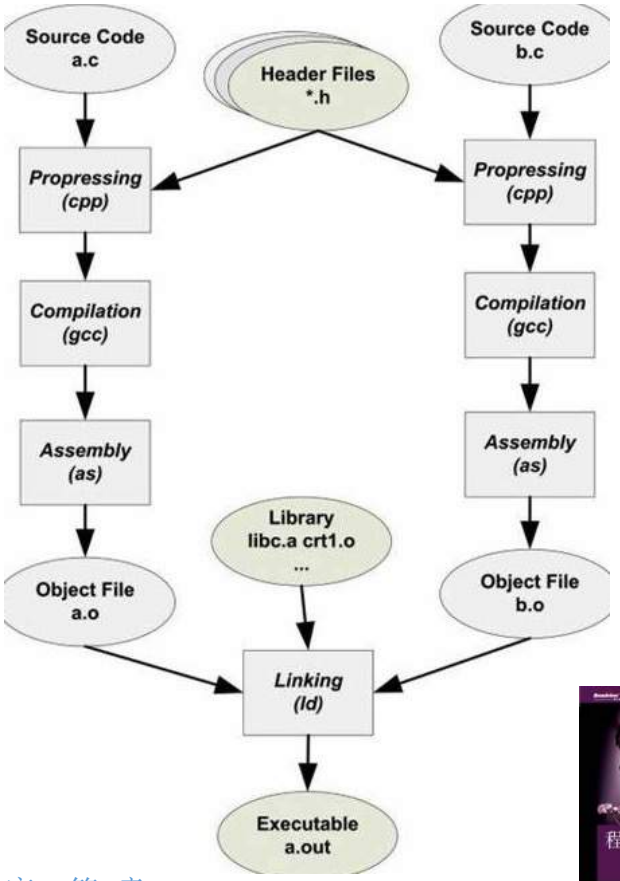
gcc –E hello.c -o hello.i　　gcc -S hello.i -o hello.s　　gcc hello.o –o hello

gcc –c hello.s –o hello.o

C Compilation (cont.)

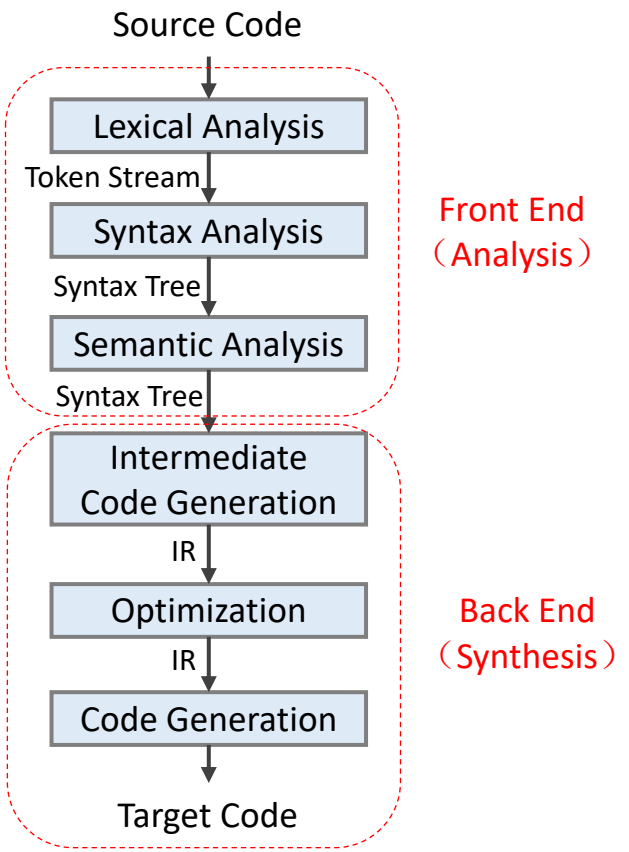
- Preprocessing: 源文件→处理后的源文件
- Compiling:处理后的源文件→汇编代码文件
- Assembling: 汇编代码文件→目标文件/机器指令文件
- Linking: 目标文件→可执行文件





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）

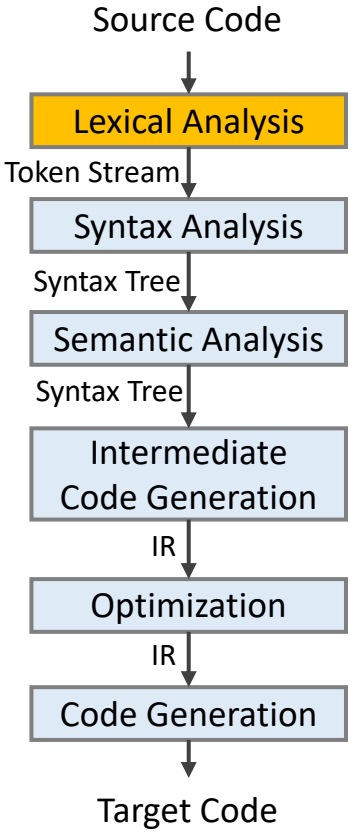


Lexical Analysis[词法分析]

- 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

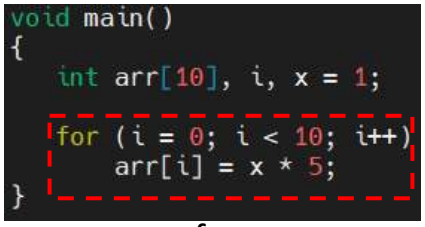
```
void main()
{
    int arr[10], i, x = 1;

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

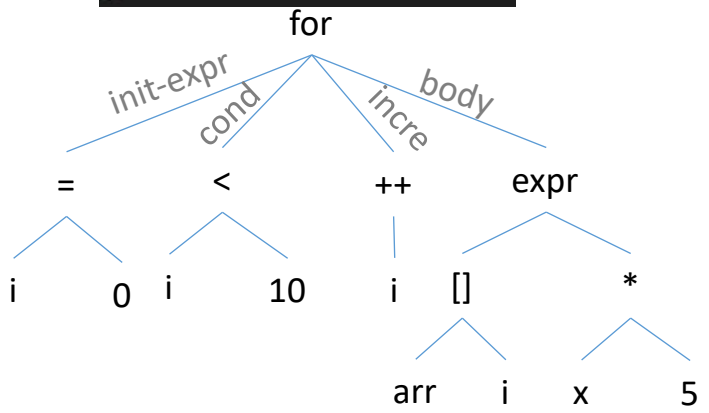
keyword(for) id(i) sym(=) num(0) sym(;) id(i) sym(<) num(10) sym(;) id(i) sym(++)
id(arr) sym([) id(i) sym(]) sym(=) id(x) sym(*) num(5) symbol(;)

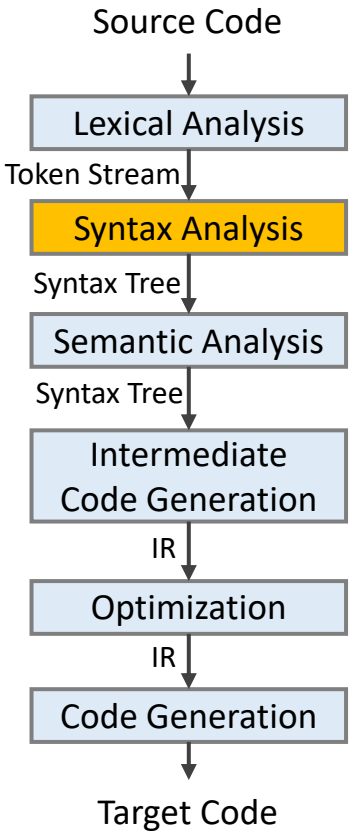


Syntax Analysis[语法分析]

- 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;

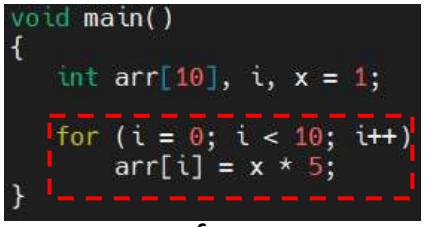


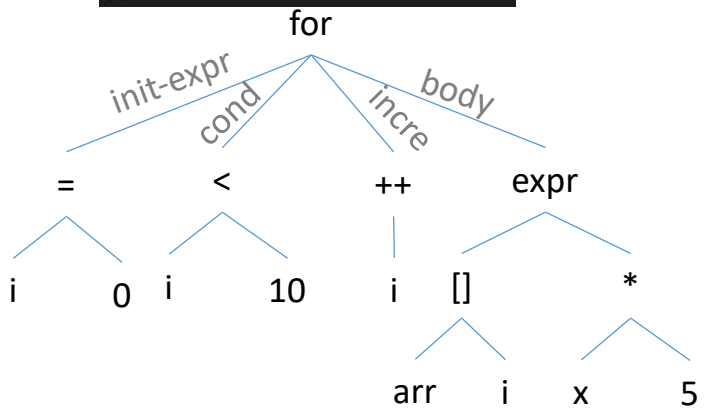


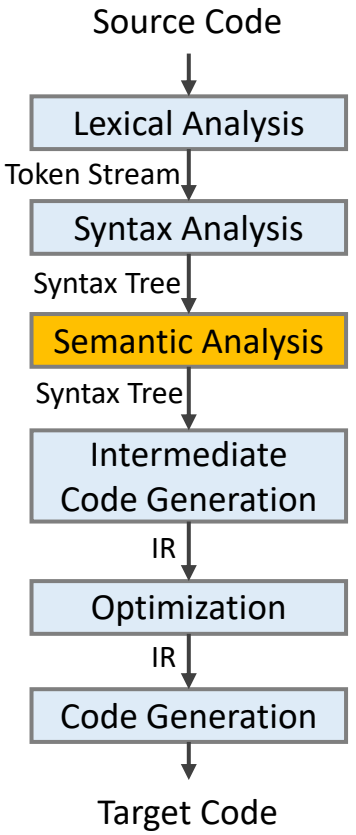


Semantic Analysis[语义分析]

- 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);





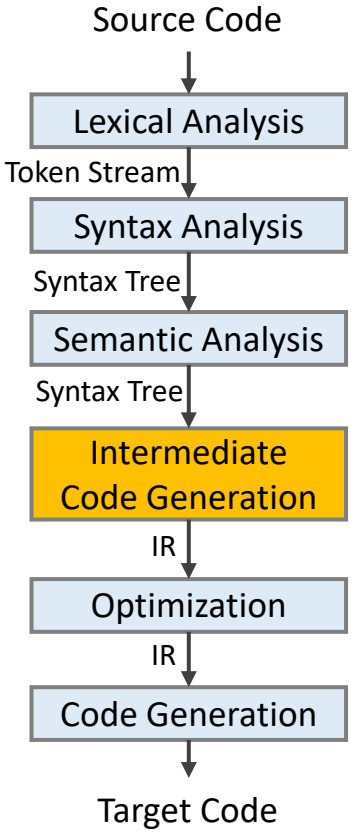


Intermediate Code[中间代码生成]

- 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：例如三地址码（TAC, Three-Address 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
```

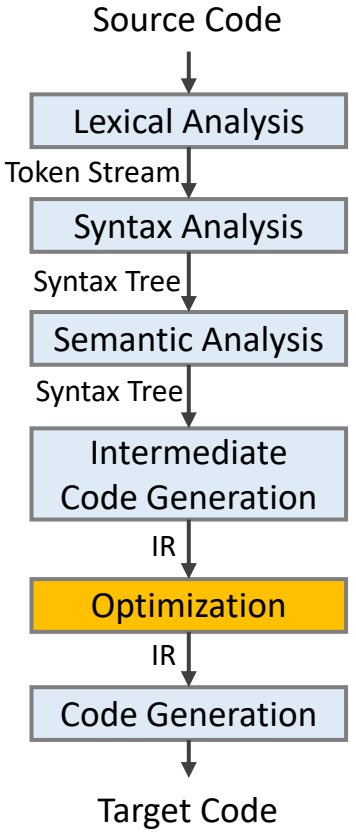


Code Optimization[代码优化]

- 加工变换中间代码使其更好（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

```
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
```

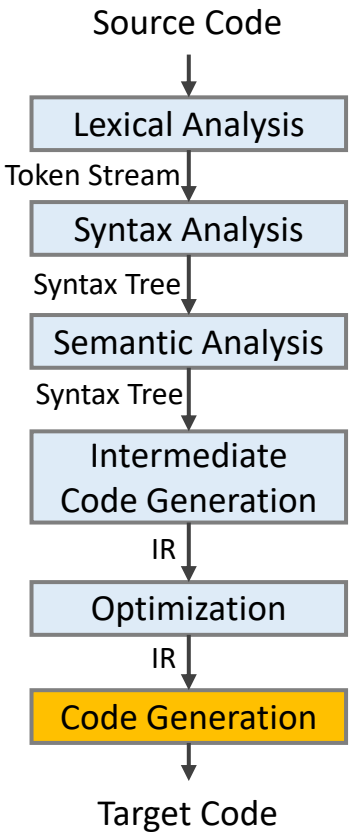


Target Code[目标代码生成]

- 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

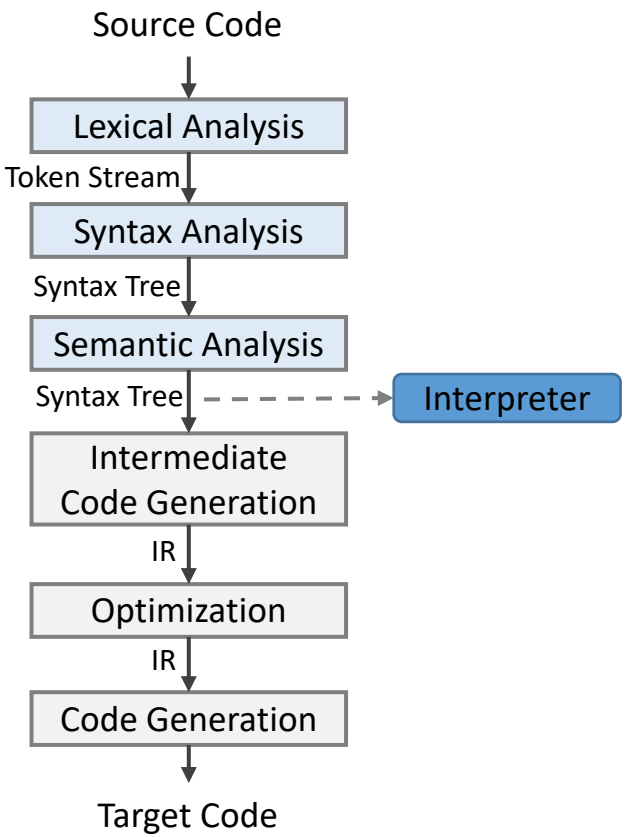
```
void main()
{
    int arr[10], i, x = 1;

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

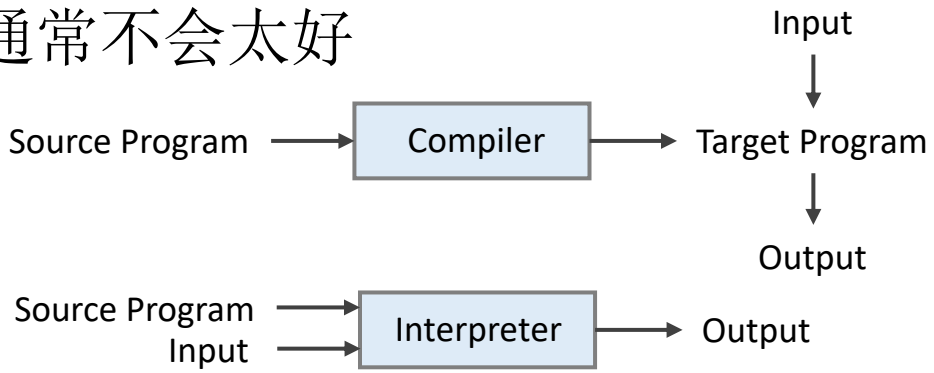
```
... ...
14:   8b 55 f8        mov   -0x8(%rbp),%edx          // edx = x
17:   89 d0           mov   %edx,%eax                // eax = x
19:   c1 e0 02        shl   $0x2,%eax                // eax  = (x << 2)
1c:   01 c2           add   %eax,%edx                // edx = (x << 2) + x
1e:   8b 45 fc        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?
... ...
```



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





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