



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

[yatcc-ai.com](http://yatcc-ai.com)

DCS290, 3/2/2026



# Happy New Year[新年快乐]

**千问一跃成为国民级AI助手**

**1.3亿** 人第一次体验AI购物

**50亿** 次自然语言输入 “千问帮我”

**Qwen3.5**

除夕发布 登顶全球最强开源模型

追求星辰大海 呵护人间烟火

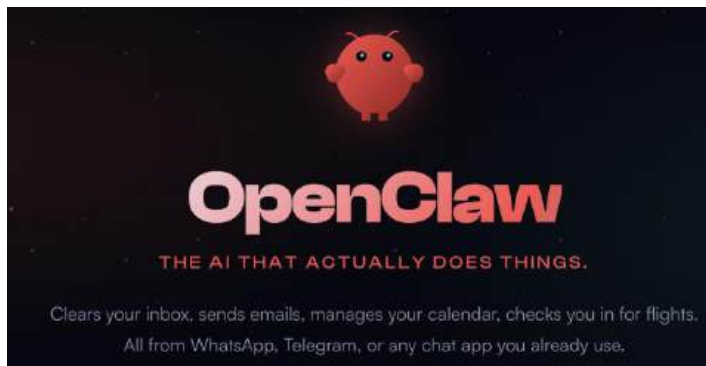
**2026 豆包过年**

除夕AI总互动 **19亿**次

豆包过年活动 帮用户生成超 **5000** 万张新春头像

帮用户写下超 **1** 亿条拜年祝福语

豆包大模型 TPM (每分种Token数) 最高 **633** 亿



**OpenClaw**

THE AI THAT ACTUALLY DOES THINGS.

Clears your inbox, sends emails, manages your calendar, checks you in for flights. All from WhatsApp, Telegram, or any chat app you already use.



即梦

Seedance 2.0

Claude 4 AI

千问超级免单卡 开启AI生活新方式

<p>· 电影票 ·</p>  <p>去买票</p>	<p>· 零食百货 ·</p>  <p>去下单</p>	<p>· 酒店 ·</p>  <p>选酒店</p>
<p>· 餐饮 ·</p>  <p>去下单</p>	<p>· 机票/火车票 ·</p>  <p>去订票</p>	<p>· 景点 ·</p>  <p>去订票</p>
<p>· 购物 ·</p>  <p>敬请期待</p>	<p>· 打车 ·</p>  <p>敬请期待</p>	<p>· 扫街榜团购 ·</p>  <p>敬请期待</p>

更多AI生活服务办事能力 持续接入中



Nano Banana



## YatCC-AI: The Future of Compiler Construction

A Modern Experimental Framework built on open-source LLVM

DeepSeek-powered, web-based experience

Unit Testing Local/Online Evaluation

Front-End Middle-End

MODULAR DESIGN, INTELLIGENT COMPILERS, HANDS-ON LEARNING

**YatCC-AI**  
ATTRACTING NEW DEVELOPERS & CONTRIBUTORS NOW

DeepSeek-Powered, Web-Based Experience. Instant Access, No Setup Required.

BUILD FULL-FLEDGED COMPILERS. EXPLORE CUTTING-EDGE OPTIMIZATION. JOIN THE EVOLUTION.

## YatCC-AI

DEVELOPER-FRIENDLY COMPILER CONSTRUCTION

DeepSeek LLM Integration

INTELLIGENT SUPPORT

UNIT TESTING

AUTOMATED EVALUATION  ★

LOCAL & ONLINE

FRONT-END MIDDLE-END

1011 00000 111001 111111

FRONT-END MIDDLE-END BACK-END

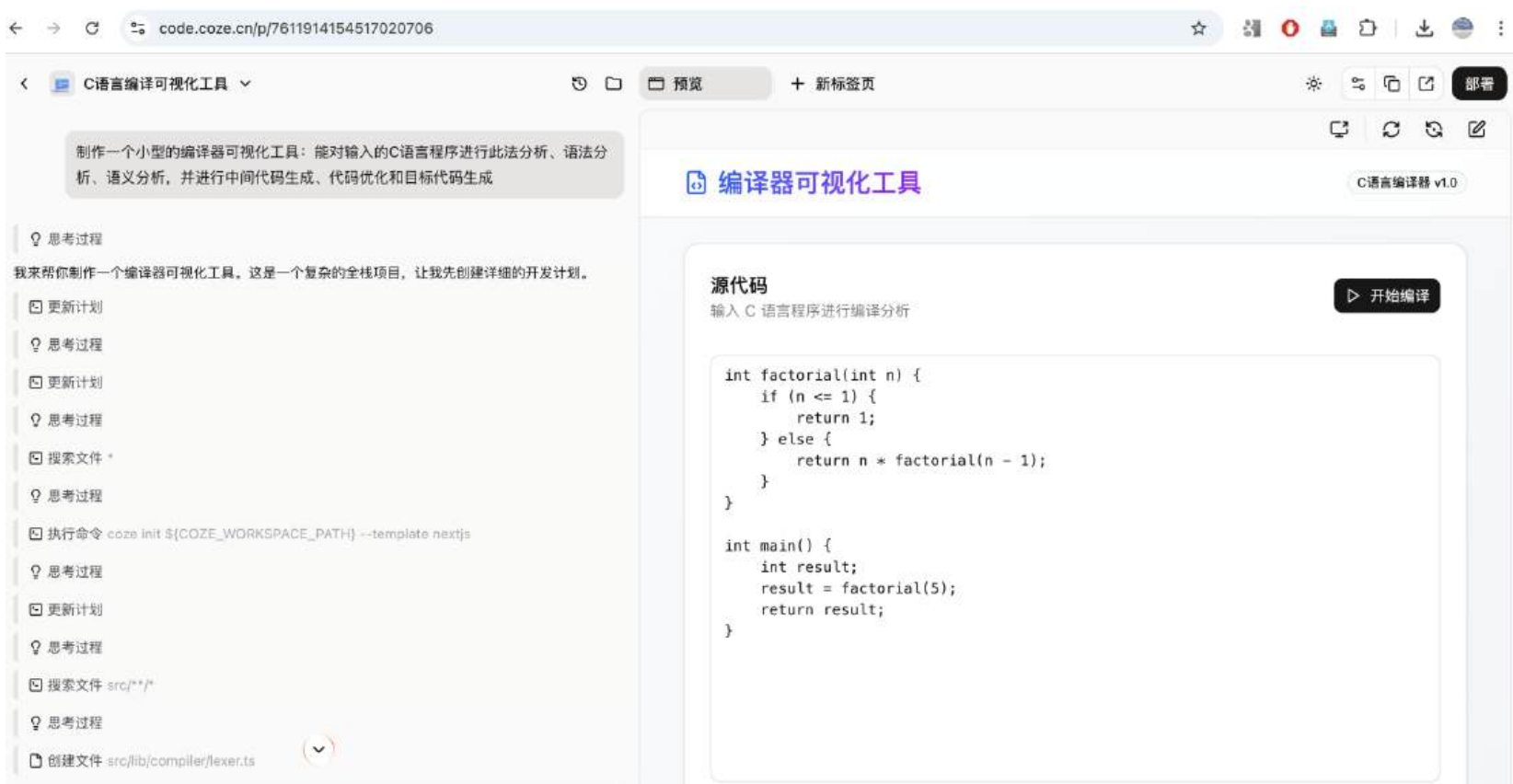
COMPILER PIPELINE

LLVM FOUNDATION

# Compiler Tool[编译小工具]

- Prompt

- 制作一个小型的编译器可视化工具：能对输入的C语言程序进行此法分析、语法分析、语义分析，并进行中间代码生成、代码优化和目标代码生成



# Front End[词法/语法/语义分析]

## 编译器可视化工具

C语言编译器 v1.0

### 编译分析结果

查看编译器各个阶段的处理结果

词法分析 语法分析 语义分析 中间代码 代码优化 目标代码

### 词法分析结果

共 29 个 Token

序号	类型	值	分类	位置
1	INT	int	Keyword	行 1, 列 1
2	IDENTIFIER	main	Identifier	行 1, 列 5
3	LPAREN	(	Separator	行 1, 列 9
4	RPAREN	)	Separator	行 1, 列 10
5	LBRACE	{	Separator	行 1, 列 12
6	INT	int	Keyword	行 2, 列 5
7	IDENTIFIER	val	Identifier	行 2, 列 9
8	ASSIGN	=	Operator	行 2, 列 13

词法分析 语法分析 语义分析 中间代码 代码优化 目标代码

### 语法分析结果 (AST)

```
程序
├── 函数声明: main (int)
│   └── 复合语句
│       ├── 变量声明: val: int
│       ├── 变量声明: rst: int
│       ├── if 语句
│       └── 返回语句
```

词法分析 语法分析 语义分析 中间代码 代码优化 目标代码

### 语义分析结果

错误 (0) 警告 (0) 符号表

没有语义错误

# Middle End[中间代码生成/优化]

词法分析 语法分析 语义分析 中间代码 代码优化 目标代码

## 中间代码 (三地址码)

共 12 条指令

序号	操作	指令
1		main:
2	func_begin	func_begin main
3	decl	decl undefined int
4	=	val = 3
5	decl	decl undefined int
6	=	rst = 0
7	>	> val 1 -> t0
8	ifFalse	ifFalse t0 goto L0

词法分析 语法分析 语义分析 中间代码 代码优化 目标代码

## 代码优化

无优化

优化信息 优化后代码 完整对比

优化后的中间代码 12 条指令

```
1. main:
2. func_begin main
3. decl undefined int
4. val = 3
5. decl undefined int
6. rst = 0
7. > 3 1 -> t0
8. ifFalse t0 goto L0
9. rst = 1
10. L0:
11. return 0
12. func_end main
```

## 代码优化

无优化

优化信息 优化后代码 完整对比

没有应用优化

### 应用的优化技术:

- 常量折叠: 计算常量表达式
- 常量传播: 用常量替换变量
- 死代码消除: 删除未使用的临时变量
- 冗余消除: 重复表达式消除

# Back End[后端代码生成]

词法分析   语法分析   语义分析   中间代码   代码优化   目标代码

目标代码 (汇编) 共 18 行

```
1  main:
2
3  ; Function main
4  PUSH rbp
5  MOV rbp, rsp
6  ; declare undefined as int
7  MOV val, 3
8  ; declare undefined as int
9  MOV rst, 0
10 ; > 3 1 -> t0
11 CMP t0, 0
12 JEQ L0
13 L0:
14 MOV r0, 0
15 RET
16 MOV rsp, rbp
17 POP rbp
18 RET
```

伪汇编代码   目标架构: 简化版 x86-like

- MOV r0, value – 将值加载到寄存器
- ADD/SUB/MUL/DIV – 算术运算
- CMP/JEQ/JMP – 比较和跳转指令
- CALL/RET – 函数调用和返回
- PUSH/POP – 栈操作

# Deploy[服务部署]

☰ 预览    部署    + 新标签页    设置    刷新    分享    部署

总览   日志   分析   域名

< 返回总览

部署中   取消

☑ 打包    构建    3 部署

```
2026-02-28T22:47:37+08:00 info: [launch] Create deploy history: 7611927511424827444
2026-02-28T22:47:38+08:00 info: [launch] Deploy logging init success, logid: 20260228224737998673F123B471320F87
2026-02-28T22:47:38+08:00 info: [launch] Initializing deployment task
2026-02-28T22:47:38+08:00 info: [launch] Loading pipeline run history
2026-02-28T22:47:38+08:00 info: [launch] Creating new pipeline run history
2026-02-28T22:47:38+08:00 info: [launch] Starting deployment
2026-02-28T22:47:38+08:00 info: [package] [code] Pipeline run start
2026-02-28T22:47:40+08:00 info: [package] [code] Pipeline run success
2026-02-28T22:47:40+08:00 info: [build] [runtime] Pipeline run start
```

<https://c4yx2w774v.coze.site/>



# Compiler is DEAD?[还需要编译器吗?]

“代码 + 编译器”要消失了? 马斯克在 xAI 全员会上放话: 到今年年底, AI 或将直接生成二进制

Original Tina InfoQ 2026年2月12日 13:17 辽宁



<https://mp.weixin.qq.com/s/-OTmOOQFvQlflHBnHBEmuDg>

“16个Agent组队, 两周干翻37年GCC!”最强编码模型Claude Opus 4.6首秀, 10万行Rust版C编译器震撼业界

Original Tina InfoQ 2026年2月6日 10:12 浙江



Anthropic had 16 AI agents build a C compiler from scratch. 100k lines, compiles the Linux kernel, \$20k, 2 weeks.

To put that in perspective GCC took thousands of engineers over 37 years to build. (Granted from 1987 - however) One researcher and 16 AI agents just built a compiler that passes 99% of GCC's own torture test suite, compiles FFmpeg, Redis, PostgreSQL, QEMU and runs Doom.

They say they "(mostly) walked away." But that "mostly" is doing heavy lifting.

[https://mp.weixin.qq.com/s/\\_WNbBA6ThPXHw-2GJhzuEA](https://mp.weixin.qq.com/s/_WNbBA6ThPXHw-2GJhzuEA)

LLVM之父拆解Claude构建的C编译器: 当 AI 把几十年工程实践变成默认能力, 软件工程开始换玩法

Original Chris Lattner InfoQ 2026年2月21日 10:15 辽宁

FEBRUARY 18, 2026

## The Claude C Compiler: What It Reveals About the Future of Software

CHRIS LATTNER ENGINEERING

[https://mp.weixin.qq.com/s/NYKDLw\\_yVOIEvrobNlfGgg](https://mp.weixin.qq.com/s/NYKDLw_yVOIEvrobNlfGgg)

<https://www.modular.com/blog/the-claude-c-compiler-what-it-reveals-about-the-future-of-software>

# Compilation, what does ChatGPT say? [2024]

---



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 code, providing valuable feedback to developers during

# Compilation, what does LLM say? [2026]



are compiler techniques still useful in the age of LLM



Short answer: **yes** — **arguably more than ever**.

But their role is evolving.

Let's look at this from a systems perspective.

## 1 LLMs Generate Code — They Don't Optimize Machines

Large language models like [ChatGPT](#) or [GitHub Copilot](#) can:

- Generate syntactically valid code
- Suggest refactorings
- Translate between languages

But they do *not*:

- Perform alias analysis
- Guarantee memory safety
- Optimize register allocation
- Schedule instructions for a specific microarchitecture

## 7 Strategic Takeaway

Compiler techniques are:

- Not obsolete
- Not replaced
- Not diminished

They are becoming **infrastructure for AI itself**.

The age of LLMs does not eliminate compilers.

It pushes them upward — toward cross-layer intelligent orchestration.



are compiler techniques still useful in the age of LLM



Excellent question. The short answer is a resounding **yes**, **compiler techniques are not just still useful, but are arguably becoming more critical than ever**.

The relationship isn't one of replacement, but of powerful synergy. Large Language Models (LLMs) are changing *how* we interact with computers and generate code, while compiler techniques are the foundational tools that make that code efficient, secure, and runnable on actual hardware.

Here's a breakdown of why compiler techniques remain indispensable in the age of LLMs, looking at three key areas: **Generating Code, Running Code, and Building the LLMs Themselves**.

## 1. For the Code that LLMs Generate (The "What")

LLMs are brilliant at producing code that *looks* correct. However, the code they generate often

### Conclusion

**Compiler techniques are not dying; they are evolving.** They have moved from being a quiet, final step in the software development process to becoming a critical component at every level of the AI revolution. They are the tools that:

1. **Polish** the raw code generated by LLMs.
2. **Bridge** that code to the diverse and powerful hardware it needs to run on.
3. **Build** the very frameworks that make LLMs possible.

So, far from being obsolete, a deep understanding of compilers is more valuable than ever for anyone looking to build efficient, scalable, and robust AI-powered systems.

# Guides from ChatGPT[怎么学习编译? ]

- “in the age of widely used vibe coding and AI agent, for undergraduates of computer science, how to learn compilation and grasp compiler techniques to prepare for future career path. list key takeaways”
  - 1 Learn machine-level thinking, not just code writing
  - 2 Strengthen core foundations early
  - 3 **Focus on IR and optimization, not only parsing**
  - 4 **Use AI as an accelerator, not a shortcut**
  - 5 **Build real systems incrementally**
  - 6 Connect compiler skills to high-demand fields
  - 7 Develop cross-layer systems thinking
  - 8 Treat compilation as career leverage
- **Final Takeaway**

Vibe coding democratizes programming.

Compiler knowledge differentiates engineers.

Students who master compiler techniques are not competing with AI — they are positioning themselves to build, optimize, and scale the intelligent systems that power the future.

# Compiler is DEAD?[还需要编译器吗?]

“代码 + 编译器”要消失了? 马斯克在 xAI 全员会上放话: 到今年年底, AI 或将直接生成二进制

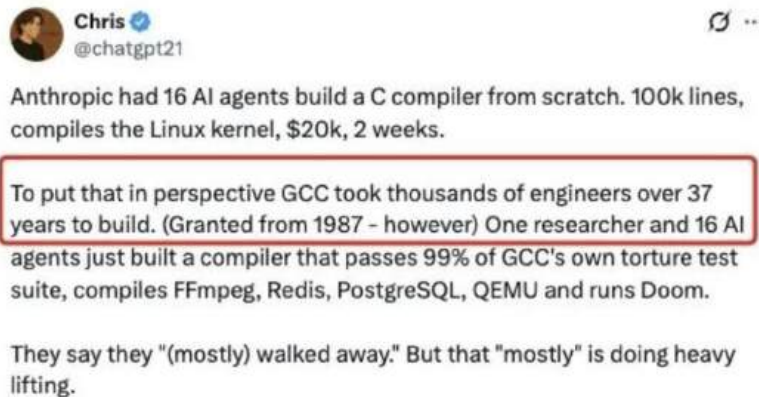
Original Tina InfoQ 2026年2月12日 13:17 辽宁



<https://mp.weixin.qq.com/s/-OTmOOQFvQlflHBnHBEmuDg>

“16个Agent组队, 两周干翻37年GCC!”最强编码模型Claude Opus 4.6首秀, 10万行Rust版C编译器震撼业界

Original Tina InfoQ 2026年2月6日 10:12 浙江



[https://mp.weixin.qq.com/s/\\_WNbBA6ThPXHw-2GJhzuEA](https://mp.weixin.qq.com/s/_WNbBA6ThPXHw-2GJhzuEA)

LLVM之父拆解Claude构建的C编译器: 当 AI 把几十年工程实践变成默认能力, 软件工程开始换玩法

Original Chris Lattner InfoQ 2026年2月21日 10:15 辽宁

FEBRUARY 18, 2026

## The Claude C Compiler: What It Reveals About the Future of Software

CHRIS LATTNER ENGINEERING

[https://mp.weixin.qq.com/s/NYKDLw\\_yVOIEvrobNlfGgg](https://mp.weixin.qq.com/s/NYKDLw_yVOIEvrobNlfGgg)

<https://www.modular.com/blog/the-claude-c-compiler-what-it-reveals-about-the-future-of-software>

# Compiler is DEAD?[还需要编译器吗?]

Chris Lattner

Created LLVM  
Apple's Director of Developer Tools  
Worked 4 years in creating Swift  
He makes your IDE, Xcode!



## Clang



“16个Agent组队，两周干翻37年GCC!”最强编码模型Claude Opus 4.6首秀，10万行Rust版C编译器震撼业界

Original Tina InfoQ 2026年2月6日 10:12 浙江



Anthropic had 16 AI agents build a C compiler from scratch. 100k lines, compiles the Linux kernel, \$20k, 2 weeks.

To put that in perspective GCC took thousands of engineers over 37 years to build. (Granted from 1987 - however) One researcher and 16 AI agents just built a compiler that passes 99% of GCC's own torture test suite, compiles FFmpeg, Redis, PostgreSQL, QEMU and runs Doom.

They say they "(mostly) walked away." But that "mostly" is doing heavy lifting.

[https://mp.weixin.qq.com/s/\\_WNbBA6ThPXHw-2GJhzuEA](https://mp.weixin.qq.com/s/_WNbBA6ThPXHw-2GJhzuEA)

LLVM之父拆解Claude构建的C编译器：当 AI 把几十年工程实践变成默认能力，软件工程开始换玩法

Original Chris Lattner InfoQ 2026年2月21日 10:15 辽宁

FEBRUARY 18, 2026

## The Claude C Compiler: What It Reveals About the Future of Software



[https://mp.weixin.qq.com/s/NYKDLw\\_yVOIEvrobNlfGgg](https://mp.weixin.qq.com/s/NYKDLw_yVOIEvrobNlfGgg)

<https://www.modular.com/blog/the-claude-c-compiler-what-it-reveals-about-the-future-of-software>

# Compiler is DEAD?[还需要编译器吗?]



## Clang



“16个Agent组队，两周干翻37年GCC！”最强编码模型Claude Opus 4.6首秀，10万行Rust版C编译器震撼业界

Original Tina InfoQ 2026年2月6日 10:12 浙江



Anthropic had 16 AI agents build a C compiler from scratch. 100k lines, compiles the Linux kernel, \$20k, 2 weeks.

To put that in perspective GCC took thousands of engineers over 37 years to build. (Granted from 1987 - however) One researcher and 16 AI agents just built a compiler that passes 99% of GCC's own torture test suite, compiles FFmpeg, Redis, PostgreSQL, QEMU and runs Doom.

They say they "(mostly) walked away." But that "mostly" is doing heavy lifting.

[https://mp.weixin.qq.com/s/\\_WNbBA6ThPXHw-2GJhzuEA](https://mp.weixin.qq.com/s/_WNbBA6ThPXHw-2GJhzuEA)

LLVM之父拆解Claude构建的C编译器：当 AI 把几十年工程实践变成默认能力，软件工程开始换玩法

Original Chris Lattner InfoQ 2026年2月21日 10:15 辽宁

FEBRUARY 18, 2026

## The Claude C Compiler: What It Reveals About the Future of Software



[https://mp.weixin.qq.com/s/NYKDLw\\_yVOIEvrobNlfGgg](https://mp.weixin.qq.com/s/NYKDLw_yVOIEvrobNlfGgg)

<https://www.modular.com/blog/the-claude-c-compiler-what-it-reveals-about-the-future-of-software>

# Take-aways from Chris

---

- AI has moved beyond writing small snippets of code and is beginning to participate in **engineering large systems**. 【系统工程】
- AI is crossing from local code generation into **global engineering participation**: CCC maintains architecture across subsystems, not just functions. 【全局参与】
- CCC has an “LLVM-like” design (as expected): **training on decades of compiler engineering** produces compiler architectures shaped by that history. 【经验训练】
- Our legal apparatus frequently lags behind technology progress, and AI is pushing legal boundaries. Is proprietary software cooked? 【法律滞后】
- Good software depends on **judgment, communication, and clear abstraction**. AI has amplified this. 【抽象强化】
- AI coding is automation of implementation, so **design and stewardship** become more important. 【设计维护】
- Manual rewrites and translation work are becoming **AI-native tasks**, automating a large category of engineering effort. 【AI原生任务】
- **AI, used right**, should produce better software, provided humans actually spend more energy on architecture, design, and innovation. 【正确使用AI】
- **Architecture documentation** has become infrastructure as AI systems amplify well-structured knowledge while punishing undocumented systems. 【架构文档】

<https://www.modular.com/blog/the-claude-c-compiler-what-it-reveals-about-the-future-of-software>

# LLM for ...?

---

LLM4Chatting: 知识问答推理



面向所有课



LLM4Coding: 代码开发调试



面向有编程实践的课



LLM4Compiling: 编译前沿探索

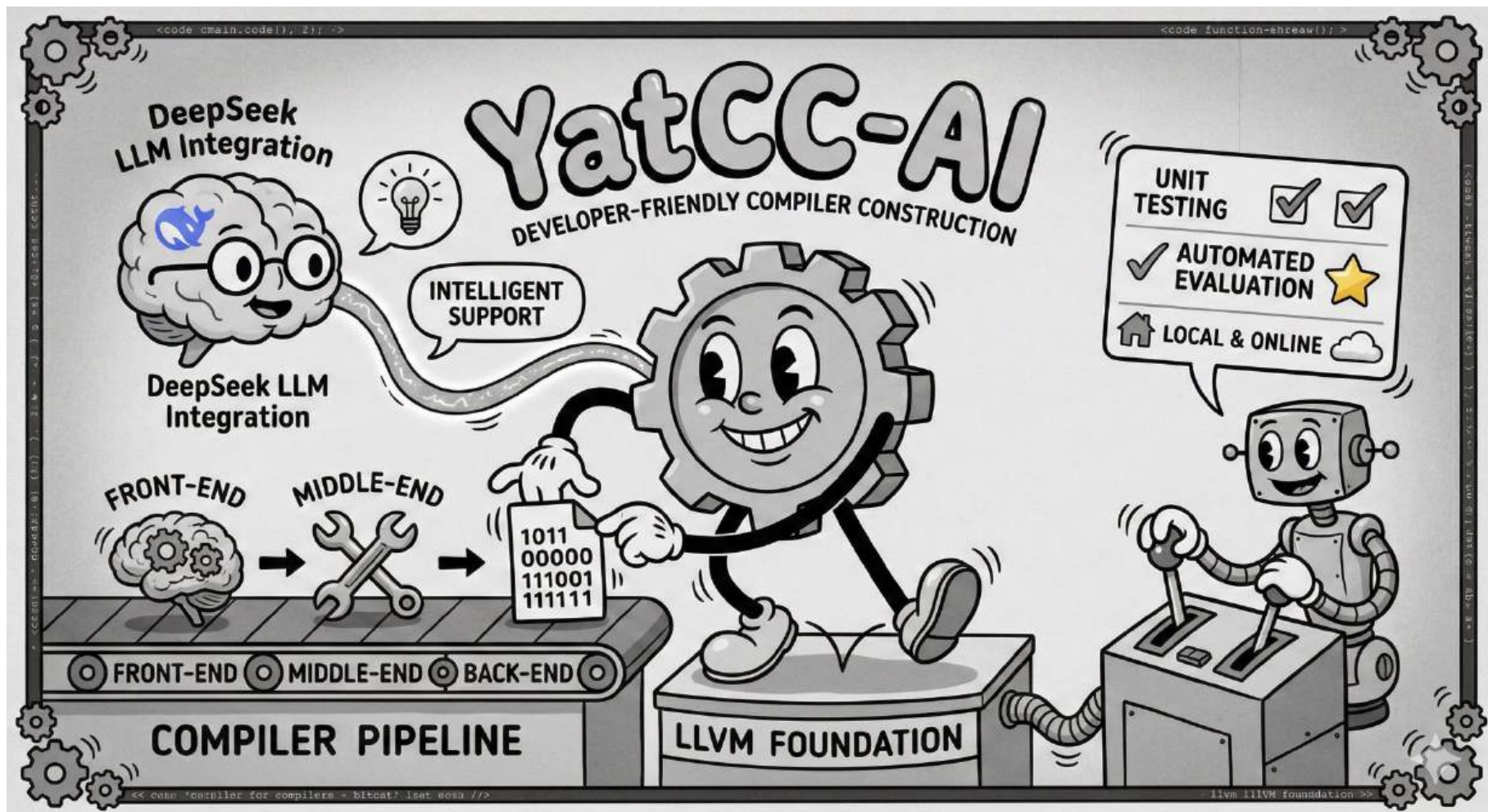


面向编译课



# YatCC-AI [[yatcc-ai.com](https://yatcc-ai.com)]

<https://yatcc-ai.com/>



# Courses on AI Coding

## CS146S: The Modern Software Developer

Stanford University • Fall 2025

Overview

Syllabus

FAQ



## CMU 17-316/616: AI Tools for Software Development

Home

Overview

Syllabus

Teamwork Practices

Staff

Assignments

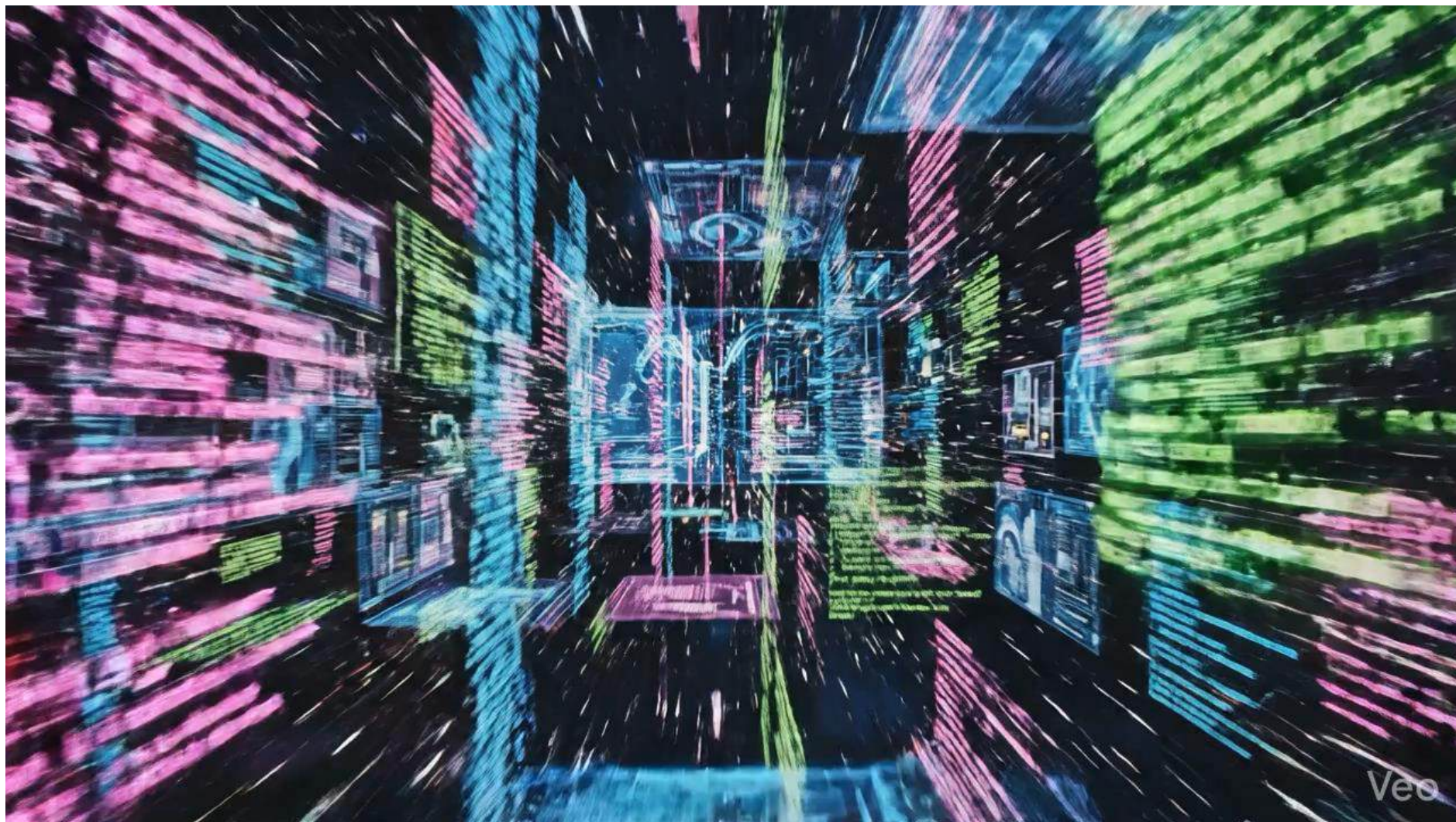
Project

MLiP / AI Engineering Schedule Syllabus Canvas

Machine Learning in Production (17-445/17-645/17-745) / AI Engineering (11-695)

“Wake up from the dream of LLM vibe coding to the reality of boring class”

---



# The Course[关于课程]

## • 年级专业

- 23级计科/1班（赵帅）
- 23级计科/2班（李文军）
- 23级计科/3班（郑馥丹）
- 23级计科/系统+AI大数据等（郑馥丹+张献伟）

## • 先修课程

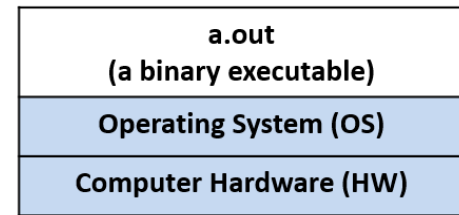
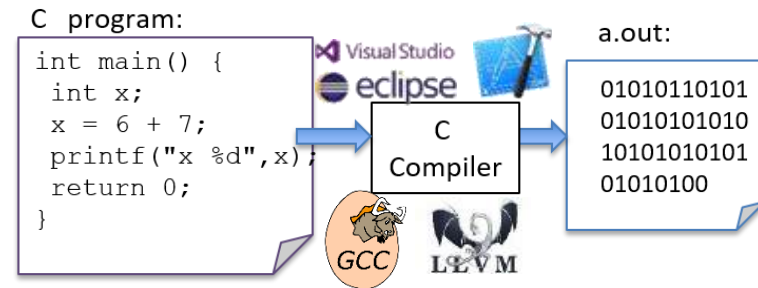
- C/C++或其他编程语言、数据结构
- 计算机组成原理/体系结构、离散数学

## • 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



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



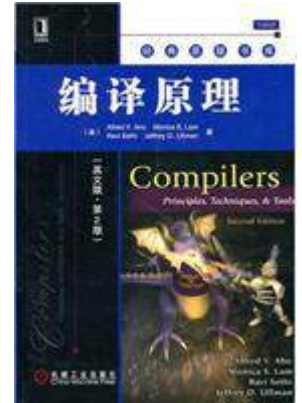
# Textbook & Materials[教材]

## • 主要教材

- 编译原理（Compilers: Principles, Techniques, and Tools, 2<sup>nd</sup> Edition），By Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
- 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 LANGUAGE COMPILERS AND ALGORITHMS

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

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



2025.03-今， on leave



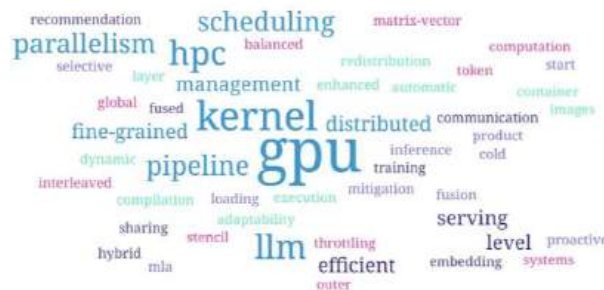
工程师/研究员，2017.08 – 2020.09



高性能及智能计算

编程及编译优化

计算机体系结构



本科：编译原理/实验（21-25s）

本科：计算机体系结构（22f）

研究生：高级计算机体系结构（21-23f）





**arcSYSu**  
A-R-X ARChitecture and system eXploration

**JOIN US!**

**S3**  
Speedy, Scalable, Smart

SUN YAT-SEN UNIVERSITY | SYSU COMPUTER SCIENCE

**FOR PASSIONATE RESEARCHERS!**  
PhD / MS / POSTDOCS / UNDERGRADS

**RESEARCH FRONTIERS**

- HIGH-PERFORMANCE COMPUTING
- COMPILATION & RUNTIME
- INTELLIGENT COMPUTING

**OUR PARTNERS**  
TENCENT | HUAWEI | PHYTIUM

**FIND OUR FAQs**  
arcsysu.edu/join

**arcSYSu**  
— ARX —

### JOIN THE ARCSYSU LAB!

**ARCSYSU** | Architecture and System Upscaling  
@ Sun Yat-sen University

Architecture Research led by Xianwei (ARX)

#### ABOUT US

- Led by Associate Professor Xianwei Zhang, School of Computer Science & Engineering. We focus on advancing **COMPUTER ARCHITECTURE, COMPILATION, and RUNTIME SYSTEMS** for high-performance and intelligent computing.
- Our Mission:** Design speedy, scalable and smart (S3) systems for next-generation computational paradigms.



#### RESEARCH & COLLABORATIONS

- Part of Interdisciplinary Research Center (xRC), directed by Prof. Yutong Lu & Prof. Nong Xiao.
  - Work closely with teams on HPC+AI design and applications.
  - Supported by National Key R&D Program, NSFC
- TENCENT®, HUAWEI®, PHYTIUM®



#### WHO WE SEEK

- **PASSIONATE PHD/MS STUDENTS**
  - **POSTDOCTORAL RESEARCHERS**
  - **MOTIVATED UNDERGRADUATES**
- EXPLORE FRONTIERS OF COMPUTING!**

READY TO ADVANCE COMPUTING?  
**APPLY NOW!**

Visit: [bit.ly/arcsysu-recruit](http://bit.ly/arcsysu-recruit)  
(or refer to FAQs for more details)



SUN YAT-SEN UNIVERSITY

# Teaching Staff[教学团队]

**Instructor:** 郑馥丹 张献伟  
**TA:** DeepSeek 黄鑫 梁铭恩 郑中淳 韩云昊  
陈俊儒 陈贤杰 孙高锦 吴露 郑腾扬  
顾宇浩 潘文轩 郭天宇 欧阳易芃 吕梓文  
郭盈盈 肖懿



xianweiz



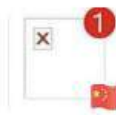
顾宇浩



郑腾扬



韩云昊



郑中淳



陈贤杰



陈俊儒



黄鑫



梁铭恩



吕梓文



郑馥丹



吴露



郭盈盈



肖懿



欧阳易芃



孙高锦

# Time/Location[课时安排]

---

- 编译原理（3学分，54学时）
  - 排课：1-17周
    - 周一：1-9周，第7/8节（16:30-18:10）
    - 周三：1-17周，第3/4节（10:10-11:50）
  - 地点：教学大楼 D402
  
- 编译器构造实验（1学分，36学时）
  - 排课：1-17周
    - 周三：1-17周，第9/10节（19:00-20:40）
  - 地点：实验中心 D501

# 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://yatcc-ai.com/teach/s2026.html>



## • 作业及实验提交

- 超算习堂

## • 课程QQ群: 905 211 783

- 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



# 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（15%）- Lexical Analysis
- Project 2（15%）- Syntax/Semantic Analysis
- Project 3（25%）- IR Generation
- Project 4（30%）- Code Optimization
- Project 5（5%）- Code Generation

## • 理论

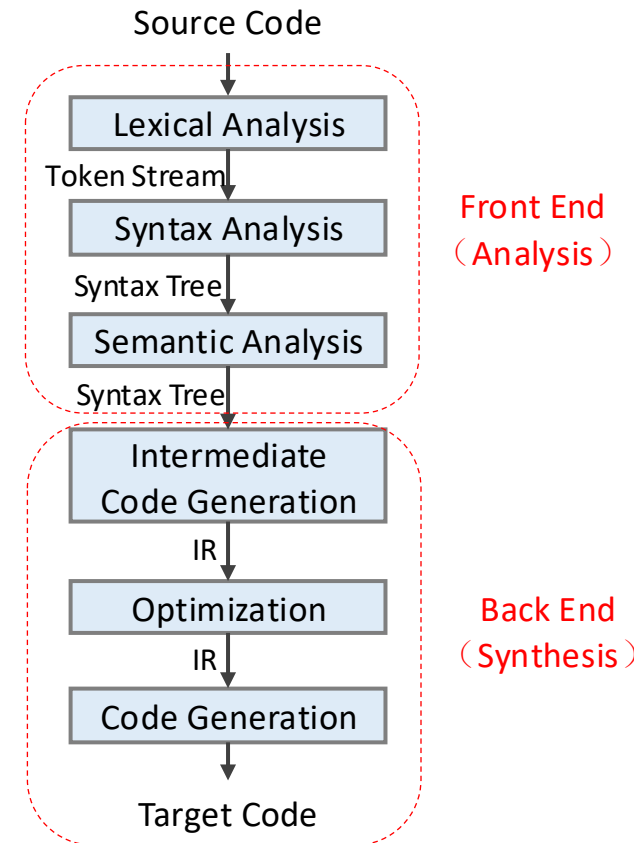
- 随机点名
  - 缺席优先
- 随机提问
  - 后排优先
- 随机测试
  - 不定时间

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





# What Mentioned[提到过的]

---

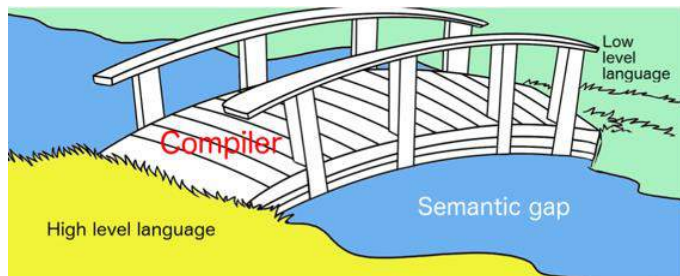
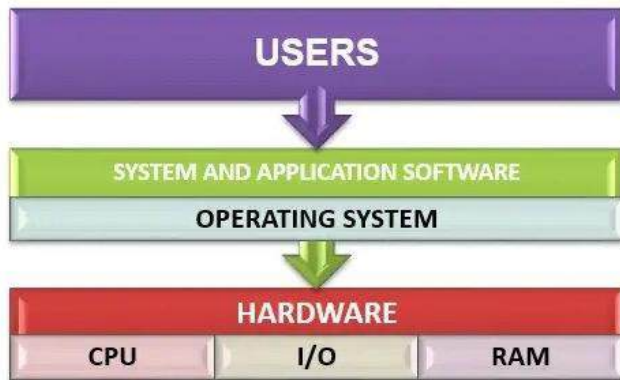


**Clang**

 **Claude**

# 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



# 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)
- 2026: **CCC — Claude's C Compiler** (Anthropic)



- **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



方舟编译器  
多端多语言，轻量低开销



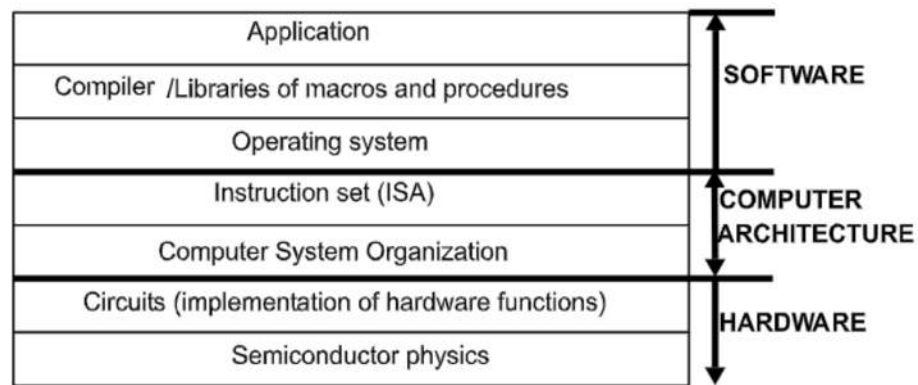
# What is Compilation? [什么是编译?]

- 高级语言编写程序，但计算机只理解**0/1**

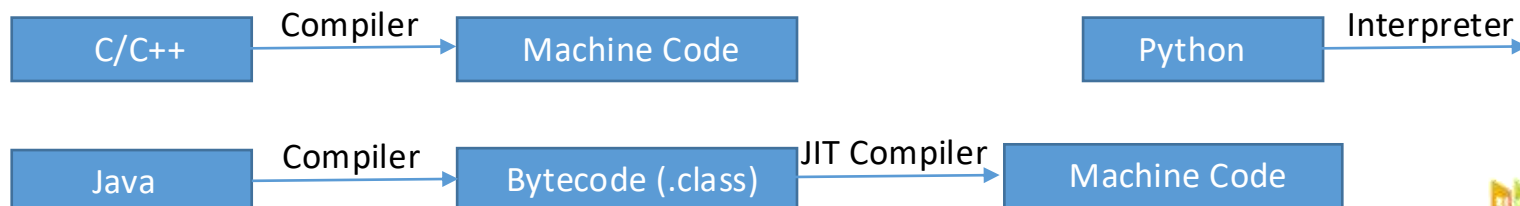
- 自然语言翻译: "This is a sentence" → "这是一个句子"
- 计算机语言翻译: 源程序 → 目标程序
- 编程人员专注于程序设计，无需过多考虑机器相关的细节

- 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)



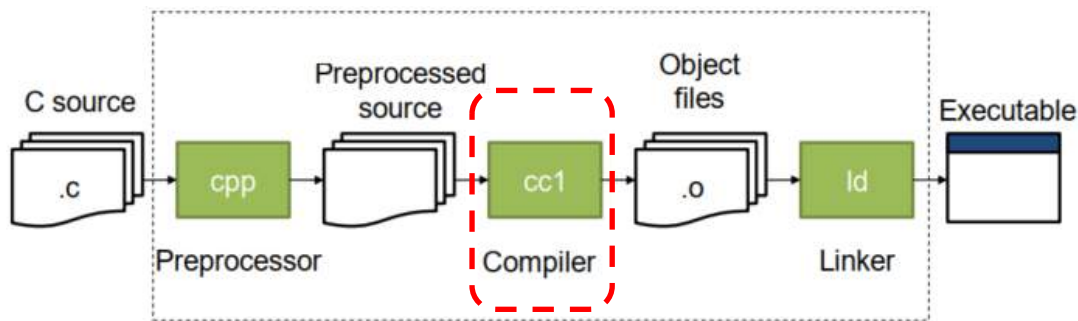
# C Compilation[c语言编译]

- 源程序 (hello.c) → 可执行文件 (./hello)

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



```
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
```

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

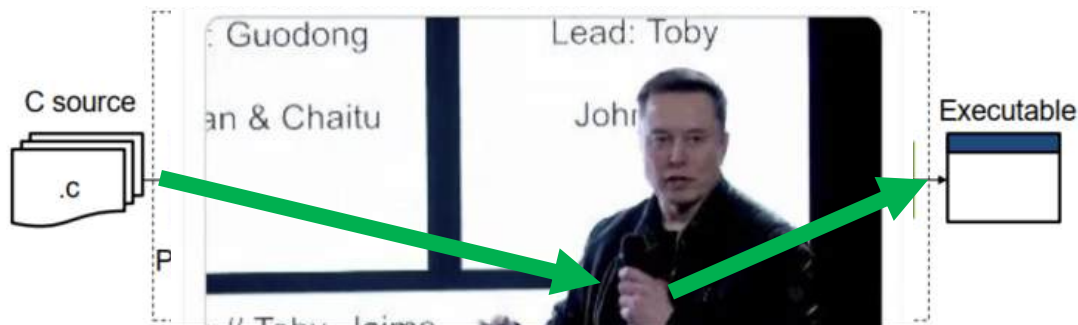
# C Compilation[c语言编译]

- 源程序 (hello.c) → 可执行文件 (./hello)

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



```
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
```

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

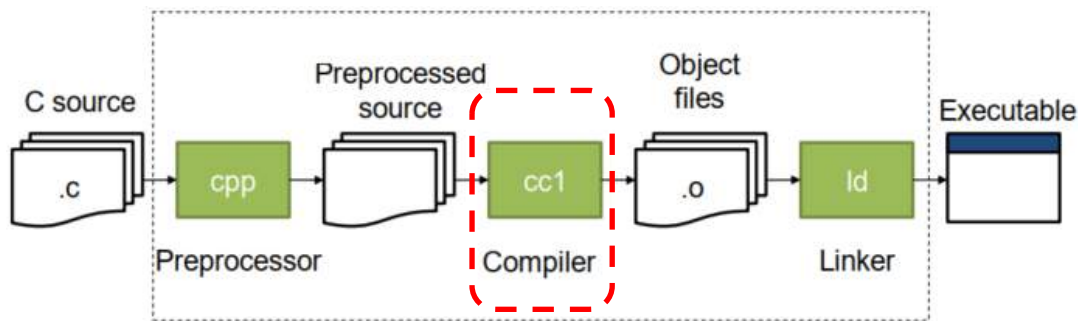
# C Compilation[c语言编译]

- 源程序 (hello.c) → 可执行文件 (./hello)

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

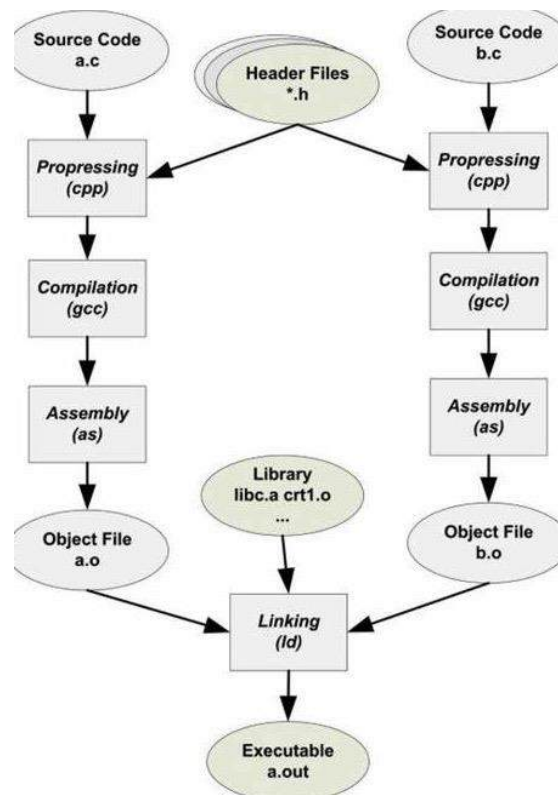
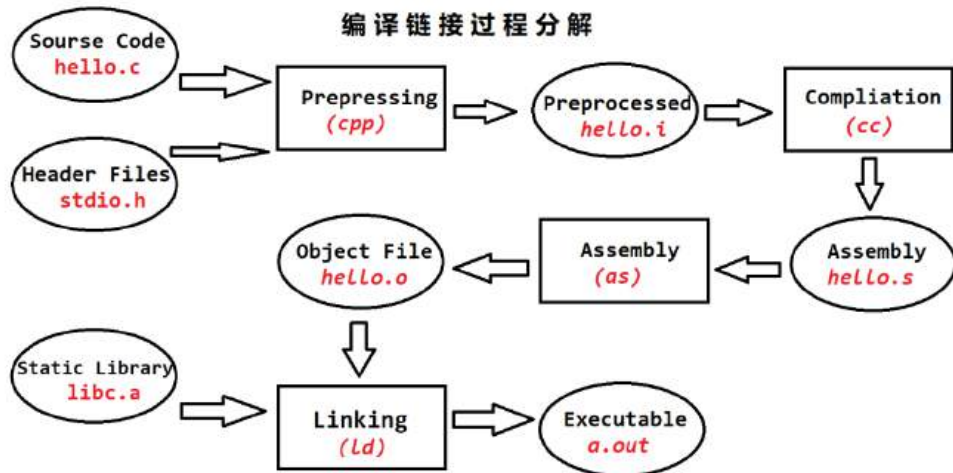
```
clang -S hello.ll -o hello.s
```

```
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



# 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.)

```
#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
```

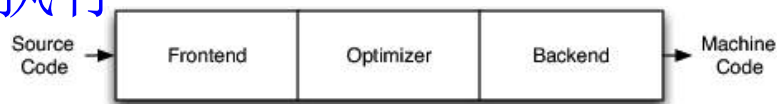
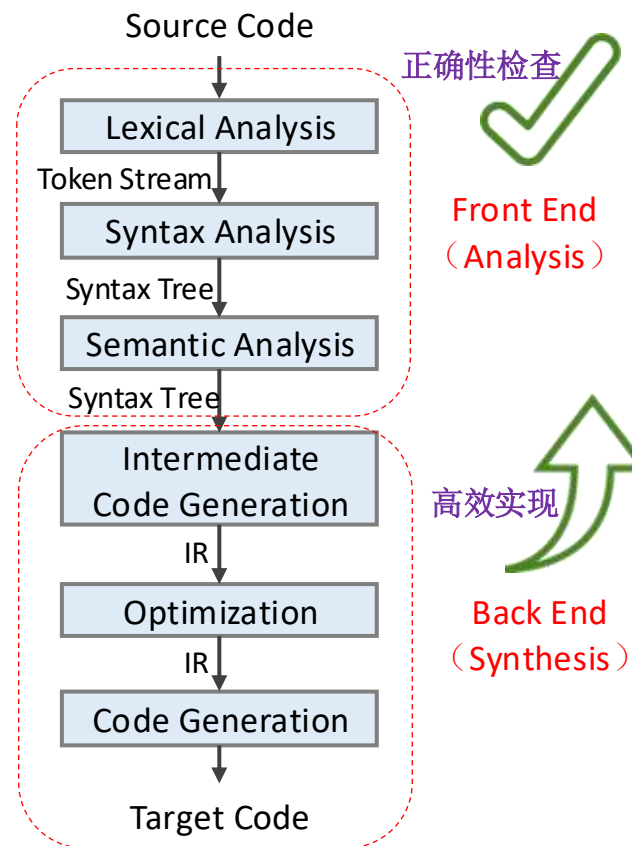
# Compilation Procedure[编译过程]

- **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） **词**
- 语法分析（Syntax Analysis） **语句**
- 语义分析（Semantic Analysis） **上下文**

- **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) **转换**
- 代码优化（Code Optimization） **更好**
- 目标代码生成（Code Generation） **可执行**



# Lexical Analysis[词法分析]

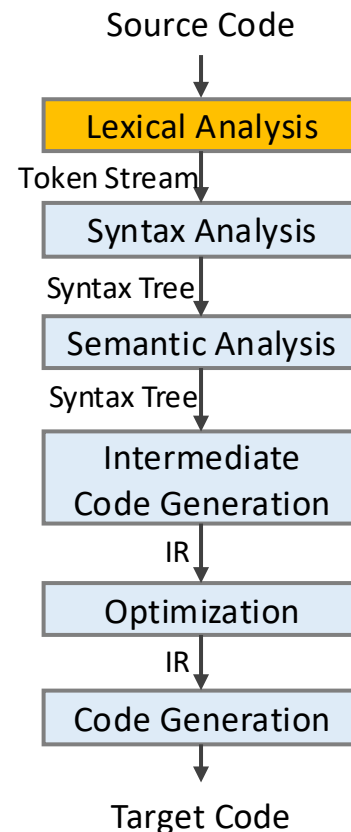
- 扫描源程序字符流，识别并分解出有词法意义的单词或符号（token）

- 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是否符合语法规则？

- Ovar, \$num

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

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



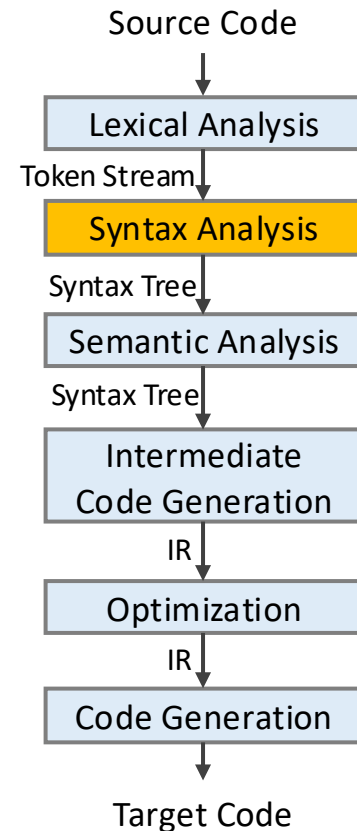
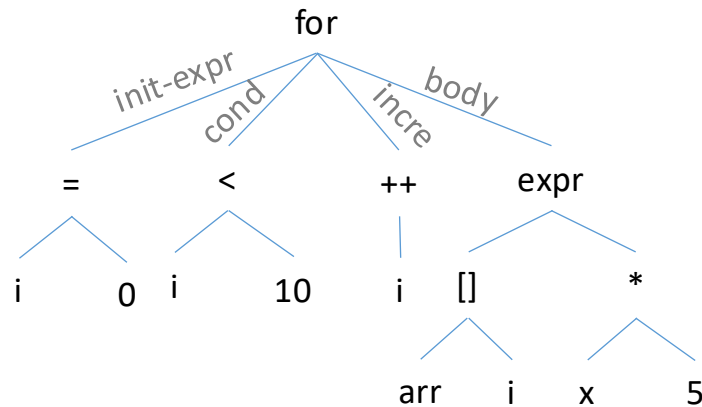
# Syntax Analysis[语法分析]

- **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？

- $x^{*+}$
- $a += 5;$

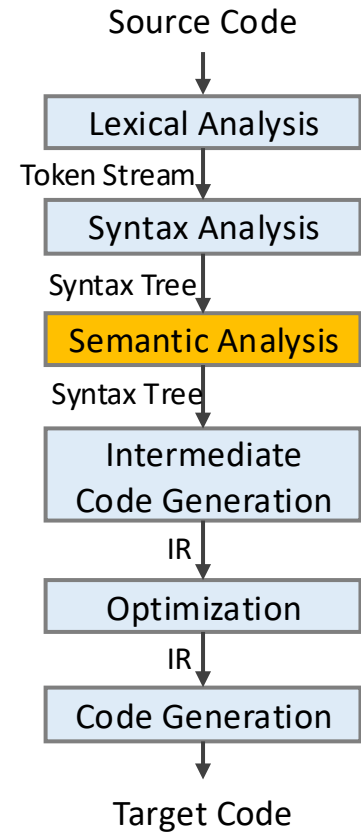
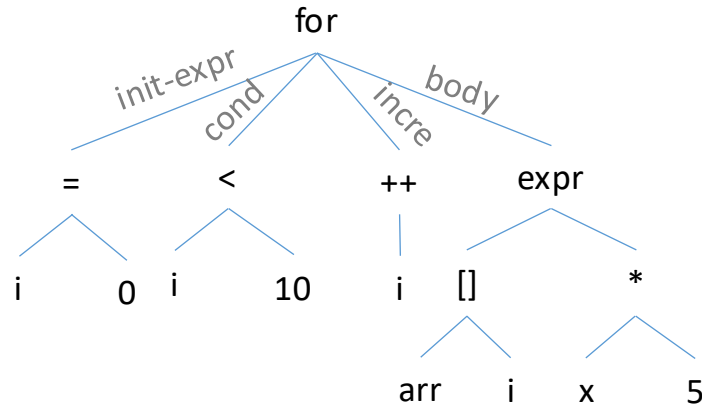
```
void main()  
{  
    int arr[10], i, x = 1;  
    for (i = 0; i < 10; i++)  
        arr[i] = x * 5;  
}
```



# 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);`

```
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]

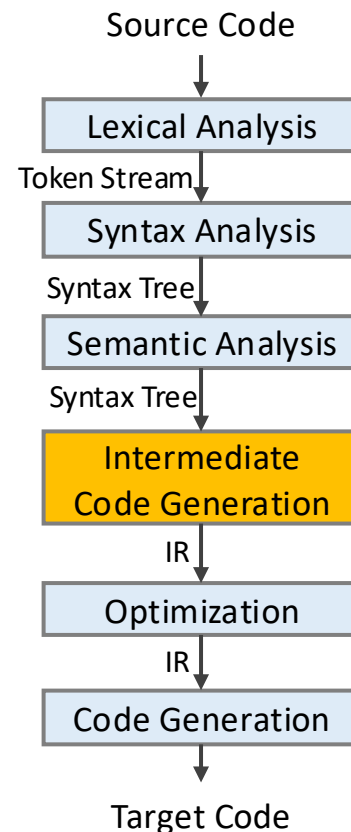
- 初步翻译，**生成**等价于源程序的中间表示

(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）

```
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
```

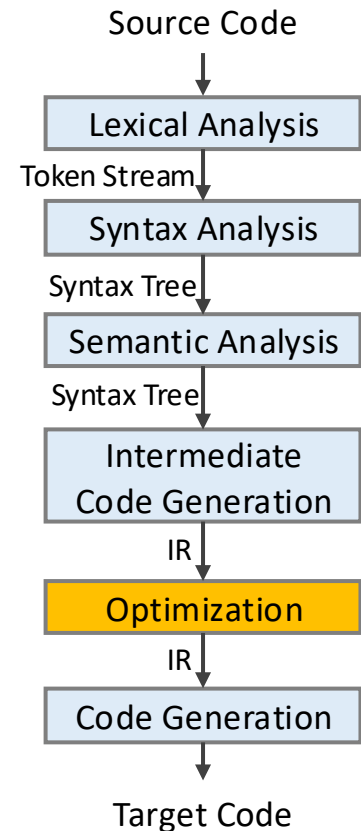


# Code Optimization[代码优化]

- **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

```
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?  
... ..
```

