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, Spring 2021

中山大學
SUN YAT-SEN UNIVERSITY

任课教师

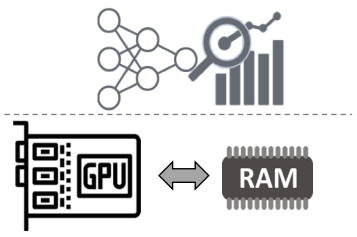
博士，2011 – 2017，University of Pittsburgh

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

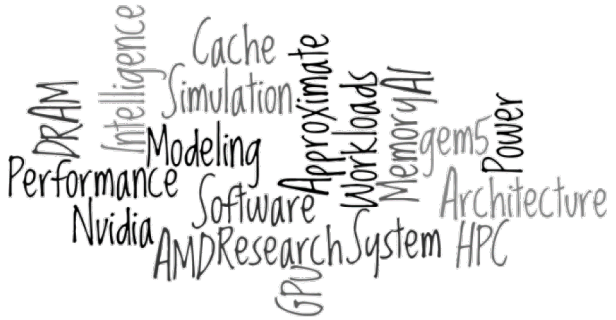
NVIDIA 实习研究员，2016.05 – 2016.08



计算机体系结构

高性能及智能计算

软硬件设计及优化



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

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 C303

- 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
  - 每次授课包括2个课时
    - 第七节：16:20 – 17:05，第八节：17:15 – 18:00
  - 地点：实验中心 E401

考核要求

- 编译原理
  - 课堂参与（10%） - 点名、提问、测试
  - 课程作业 （20%）- 4次左右，理论
  - 期中考查 （10%）- 课下习题
  - 期末考试 （60%）- 闭卷

- 编译器构造实验
  - Project 1（25%）- Lexical Analysis
  - Project 2（25%）- Syntax Analysis
  - Project 3（25%）- Semantic Analysis
  - Project 4（25%）- Code Generation

- 课堂
  - 随机点名
    - 缺席优先
  - 随机提问
    - 后排优先
  - 随机测试

- 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

课件及答疑

- 课件
  - xianweiz.github.io/teach/dcs290/s2021.html
  - 课后上传更新

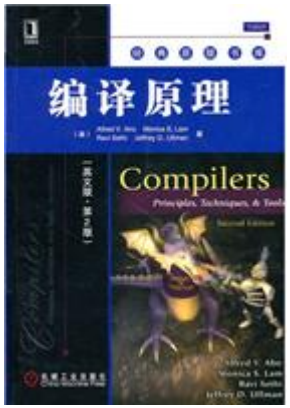
- 教师
  - 张献伟 （超算中心）
    - Email: zhangxw79@mail.sysu.edu.cn
    - 实验课答疑，其他时间需预约
- 助教
  - 理论：莫则威 （导师：杜云飞）
    - Email：mozw5@mail2.sysu.edu.cn
    - 研究方向：GCC/LLVM优化
  - 实验：王子彦（导师：郑子彬）
    - Email：wangzy75@mail2.sysu.edu.cn
    - 研究方向：区块链编译优化

关于课程

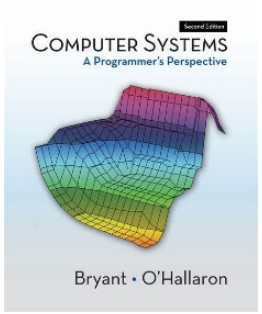
- 年级专业
  - 18级计科（专必，38人）
  - 18级保密管理 （专选，9人）
- 先修课程
  - 计算机体系结构/组成原理、汇编语言
  - 离散数学、数据结构、C/C++或其他编程语言
- 编译原理
  - 高级编程语言（如C）是如何转换为机器语言（0/1）的？
  - 介绍编译器设计与实现的主要理论和技术
    - 包括词法分析、语法分析、语义分析、代码生成、代码优化等
- 编译器构造实验
  - 单独课程，分阶段实现一个小型编译器
    - 词法分析、语法分析、语义分析等

课程教材

- 主要教材
  - 编译原理（Compilers: Principles, Techniques, and Tools, 2nd Edition）, By Alfred V. Aho, Monica S. Lam, and *et al.*



- 参考资料
  - Computer Systems: A Programmer's Perspective (CSAPP), Bryant and O'Hallaron
  - 编译原理，陈鄞（哈工大）
  - 编译原理，王挺（国防科大）
  - COMS 4115, Baishakhi Ray (Columbia U.)
  - CS 143, Fredrik Kjolstad (Stanford U.)
  - CS 411, Jan Hoffmann (Carnegie Mellon U.)
  - CS 2210, Wonsun Ahn (U. of Pittsburgh)



教学安排

周次	课程内容	周次	课程内容
第1周 (02.25)	四：课程介绍、编译概述	第11周 (05.06)	四：中间代码 – IR表示
第2周 (03.02/04)	二：词法分析 – 过程、正则表达 四：词法分析 – NFA、DFA （1）	第12周 (05.13)	四：中间代码 – 生成
第3周 (03.09/11)	二：词法分析 – NFA、DFA （2） 四：语法分析 – 语法、语言、CFG	第13周 (05.20)	四：运行时管理
第4周 (03.16/18)	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 (05.27)	四：运行时代码生成
第5周 (03.23/25)	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 (06.03)	四：代码优化
第6周 (03.30/04.01)	二：语法分析 – 更多LR分析 四：语法分析 – YACC工具	第16周 (06.10)	四：控制流、数据流分析
第7周 (04.06/08)	二：语义分析 – 符号表 四：语义分析 – 类型检查	第17周 (06.17)	四：目标代码生成
第8周 (04.13/15)	二：语法制导翻译 （1） 四：语法制导翻译（2）	第18周 (06.24)	四：前沿编译技术
第9周 (04.20/22)	二：回顾 – 编译前端 四：介绍 – 编译后端	第19周 (07.01)	四：总结 – 编译前后端
第10周	**（期中：课下考查）**	**第20周**	**（期末：闭卷考试）**

编译器的发展

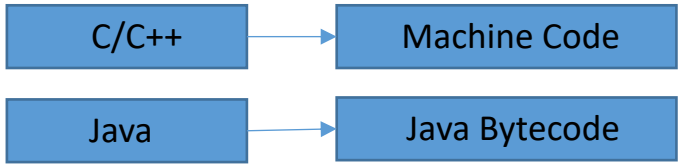
- Compiler History
  - 1952: A-0, term ‘compiler’ (Grace Hopper)
  - 1957: FORTRAN, first commercial compiler (John Backus)
  - 1962: LISP, self-hosting and GC (Tim Hart and Mike Levin)
  - 1984: GNU Compiler Collection (Stallman)
- Turing Awards (see link)
  - Compiler: 1966, 1987, 2006
  - Programming 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

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
- 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

什么是编译？

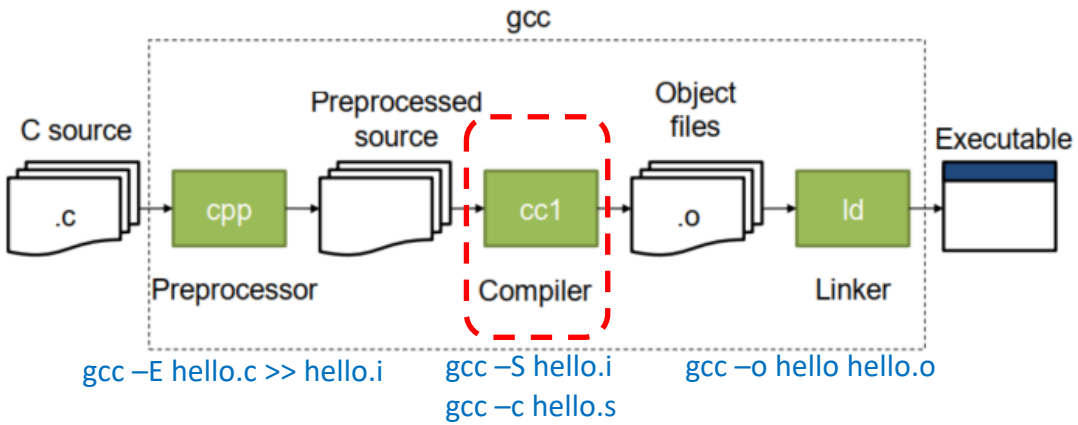
- 高级语言编写程序，但计算机只理解0/1
  - 自然语言翻译：”This is a sentence” → “这是一个句子”
  - 计算机语言翻译：源程序 → 目标程序
  - 编程人员专注于程序设计，无需过多考虑机器相关的细节
- 不同语言有不同的实现方式
  - “底层” 语言通常使用编译
    - C, C++
  - “高级” 语言通常是解释性的
    - Python, Ruby
  - 有些使用混合的方式
    - Java：编译 + 即时编译（JIT, Just-in-Time）



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）

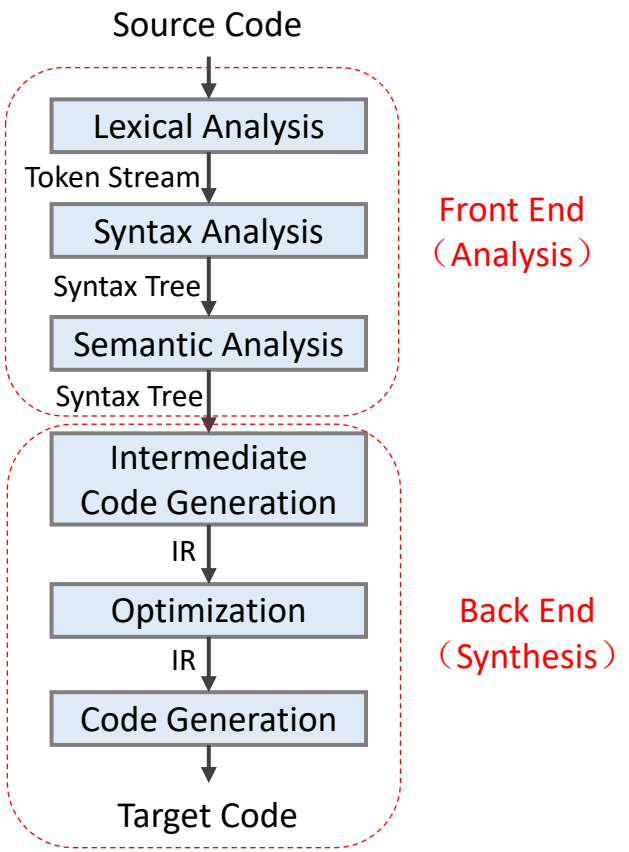
```c
#include <stdio.h>

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
```

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）

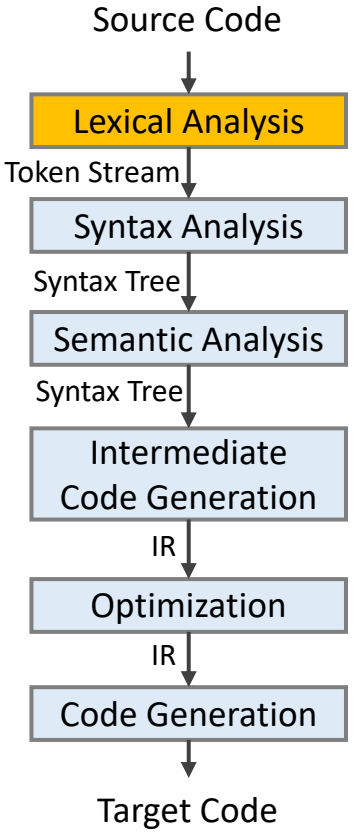


词法分析

- 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

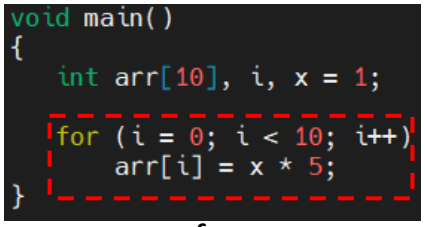
```
void main()
{
    int arr[10], i, x = 1;

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

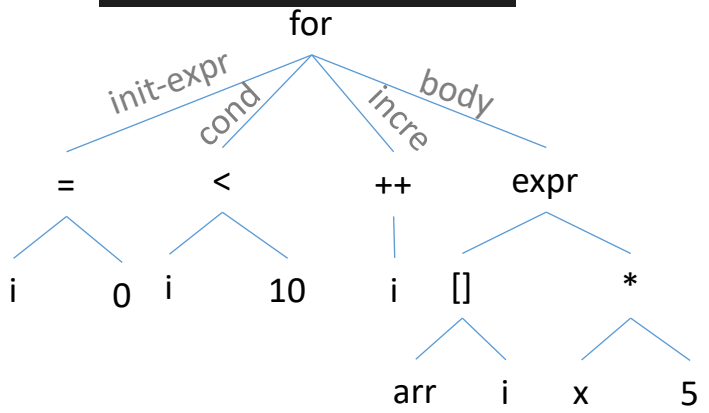
keyword(for) id(i) sym(=) num(0) sym(;) id(i) sym(<) num(10) sym(;) id(i) sym(++)
id(arr) sym([) id(i) sym(]) sym(=) id(x) sym(*) num(5) symbol(;)

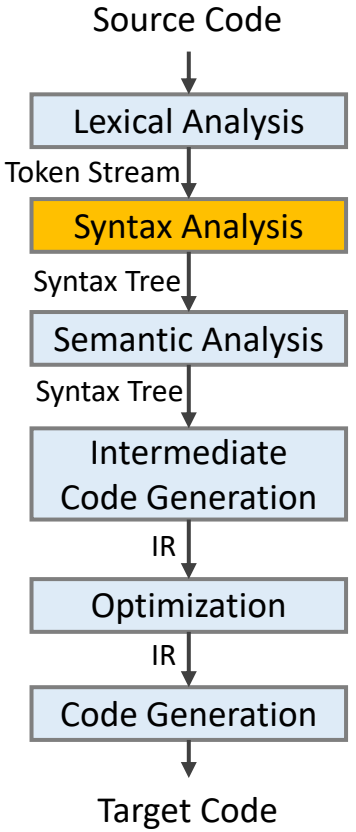


语法分析

- 解析源程序对应的token序列，生成语法分析结构（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;



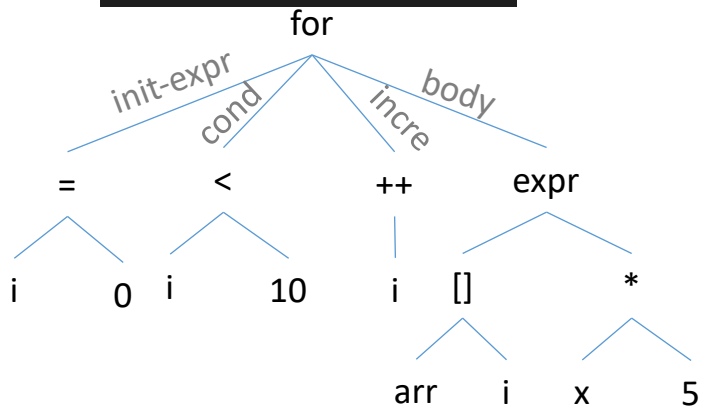


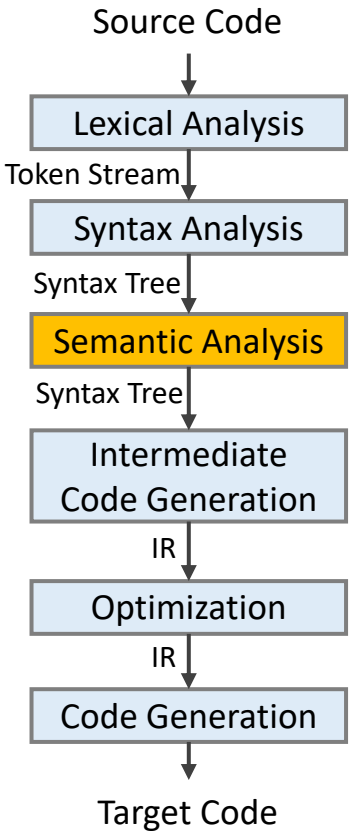


语义分析

- 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;
}
```



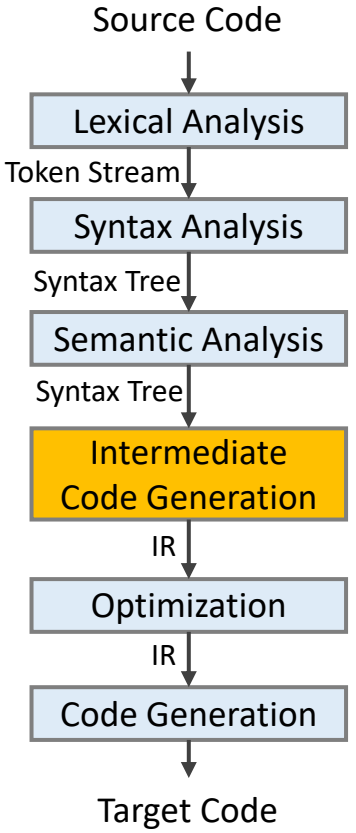


中间代码生成

- 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
```

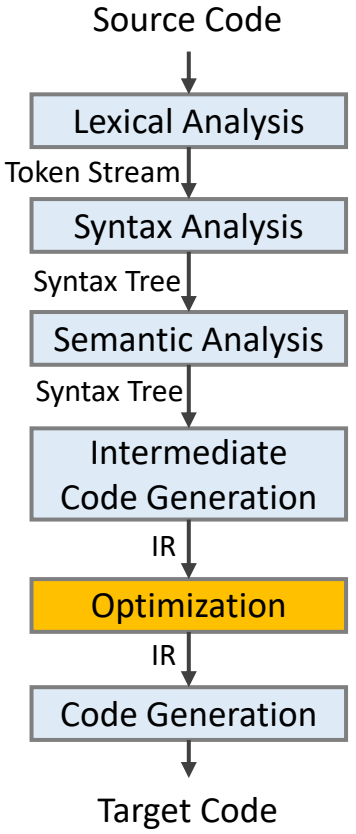


代码优化

- 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
```

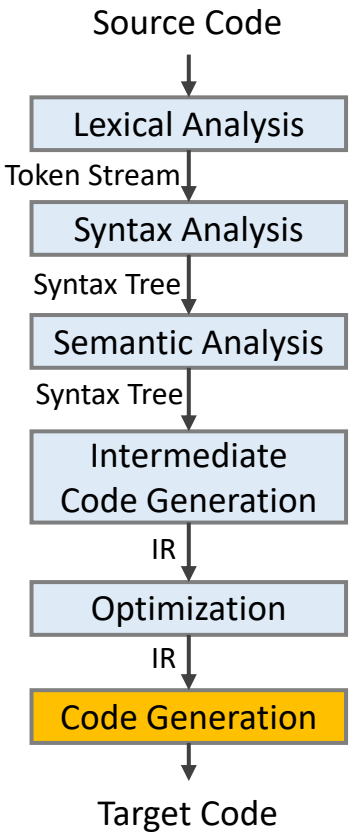


目标代码生成

- 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

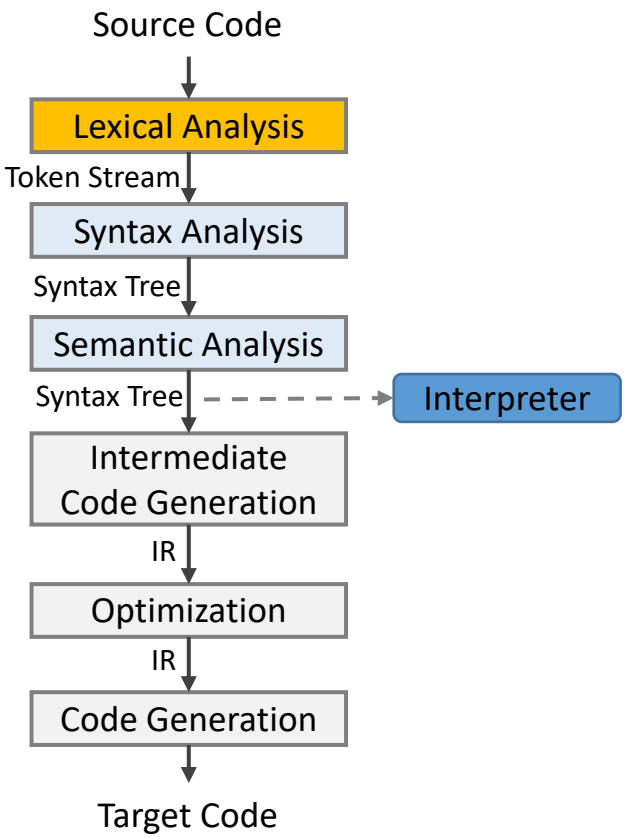
```
void main()
{
    int arr[10], i, x = 1;

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

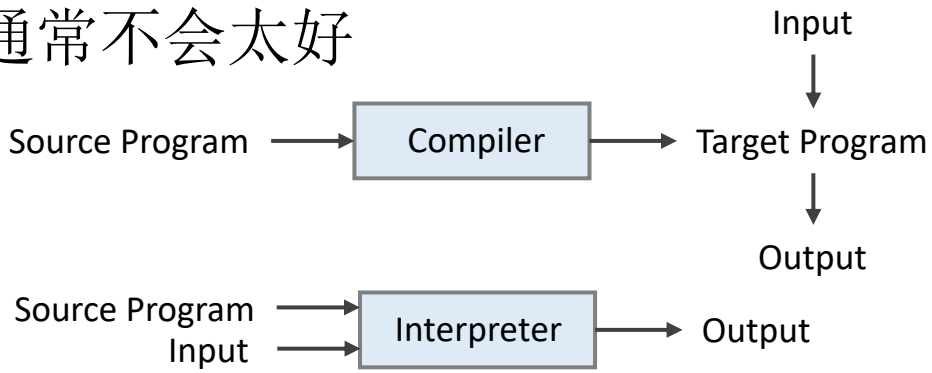
```
                                ... ...
14:   8b 55 f8          mov   -0x8(%rbp),%edx          // edx = x
17:   89 d0             mov   %edx,%eax                // eax = x
19:   c1 e0 02          shl   $0x2,%eax                // eax  = (x << 2)
1c:   01 c2             add   %eax,%edx           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



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（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





即时编译 (JIT)

- 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