



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

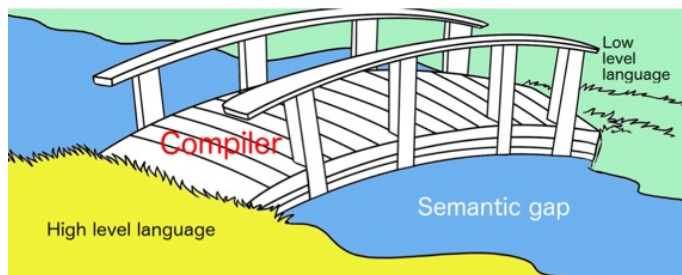
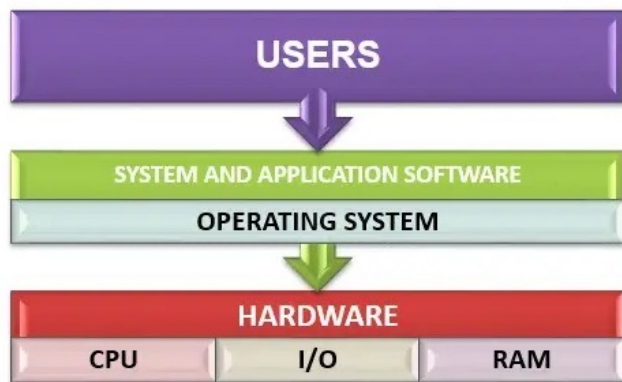


中山大學
SUN YAT-SEN UNIVERSITY



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



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



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

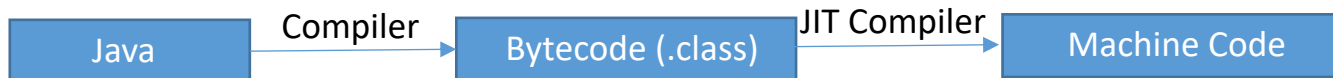
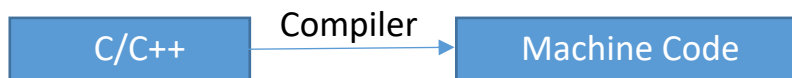
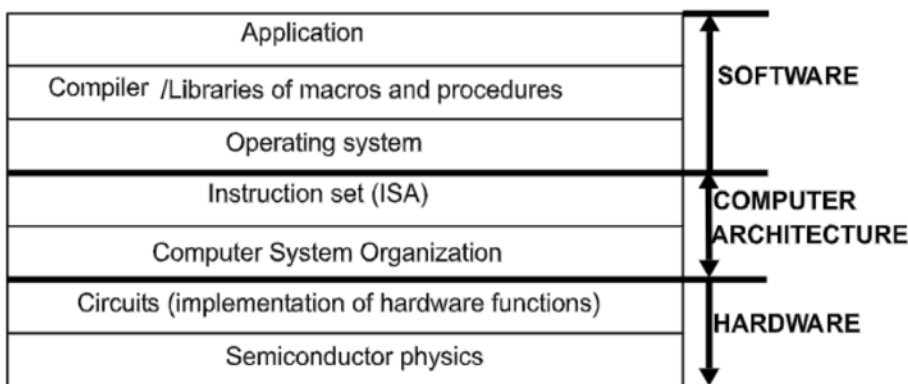
- C, C++

- “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)

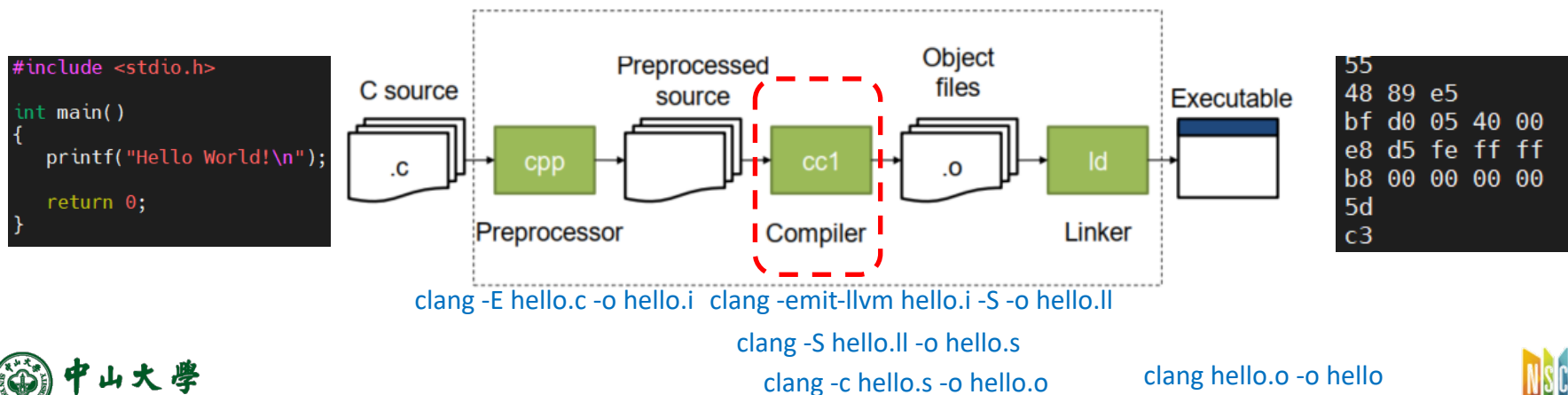
- *.i*文件翻译为*.s*文件（汇编代码）

- 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）

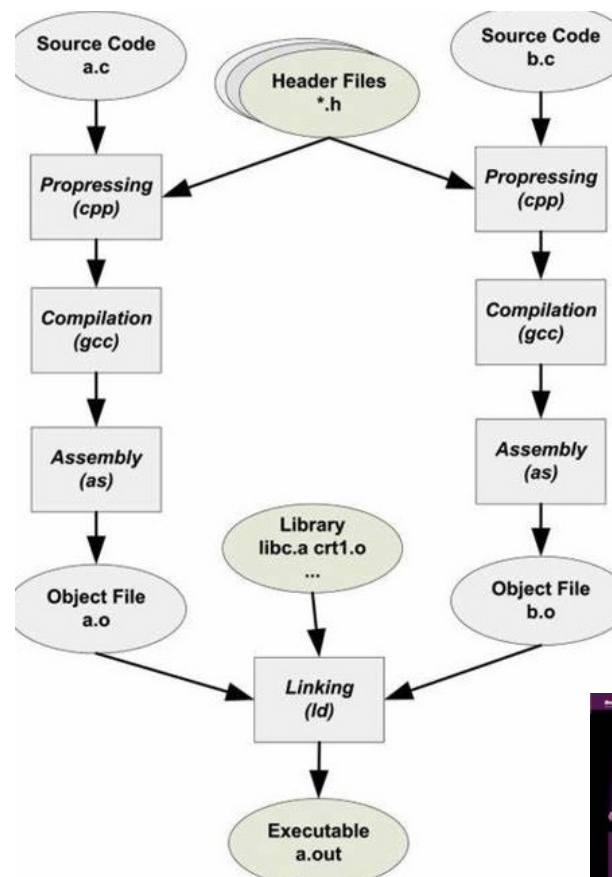
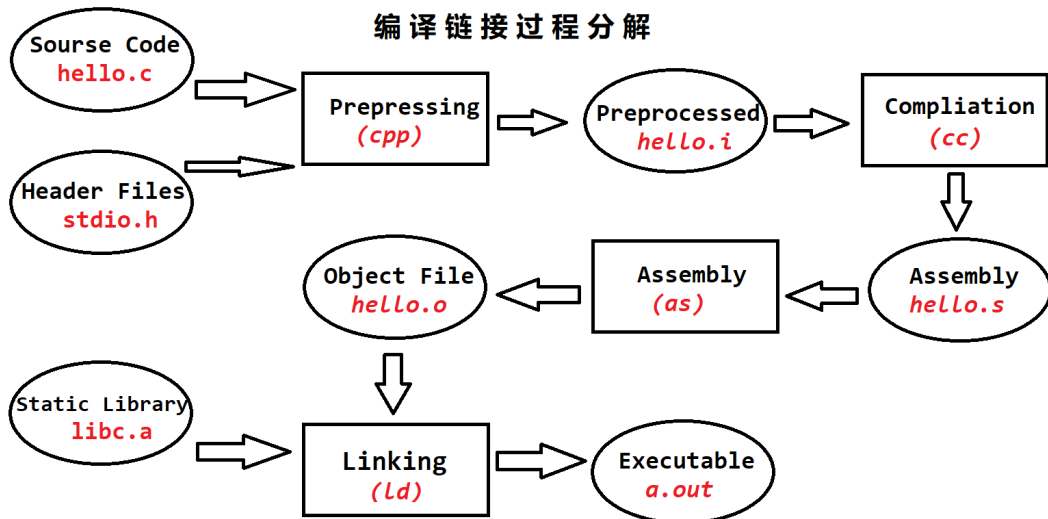
- 连接阶段 (linker/loader)

- 连接库代码从而生成可执行（executable）文件（机器指令）



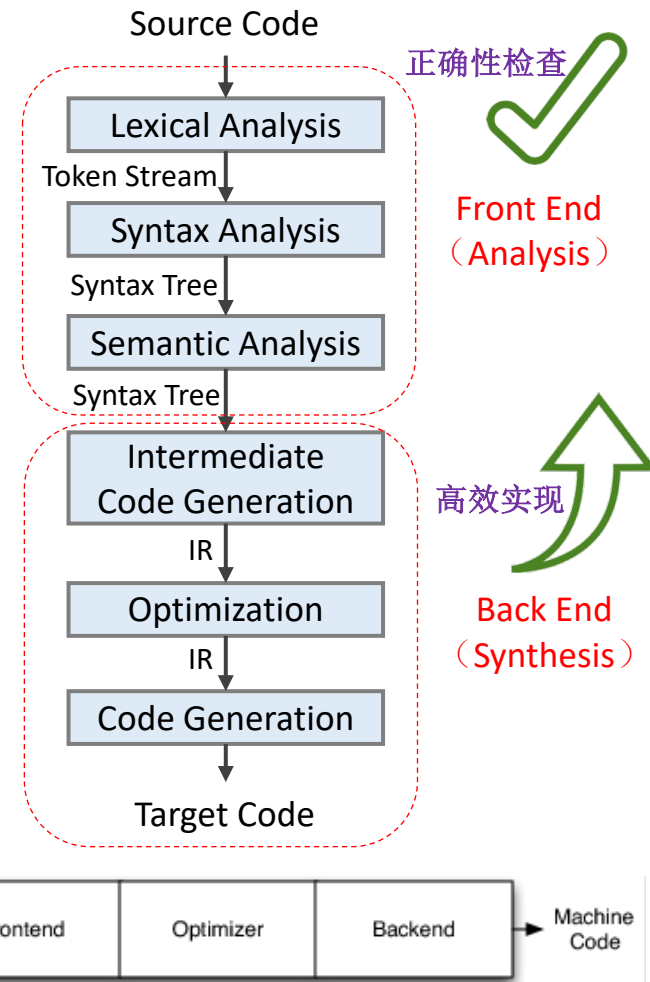
C Compilation (cont.)

- 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>

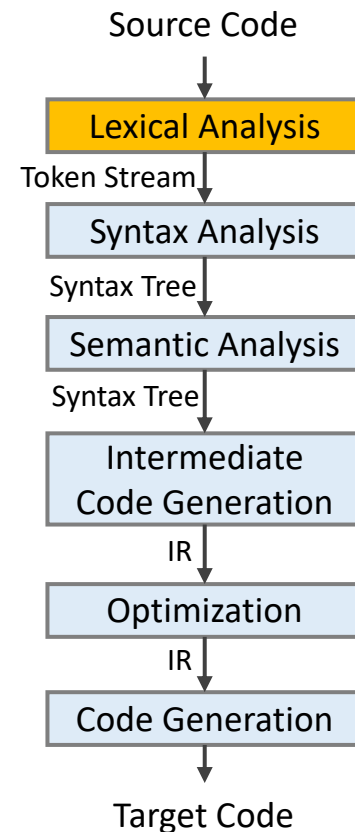
- 保留字、标示符、常量、运算符等

- **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(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]

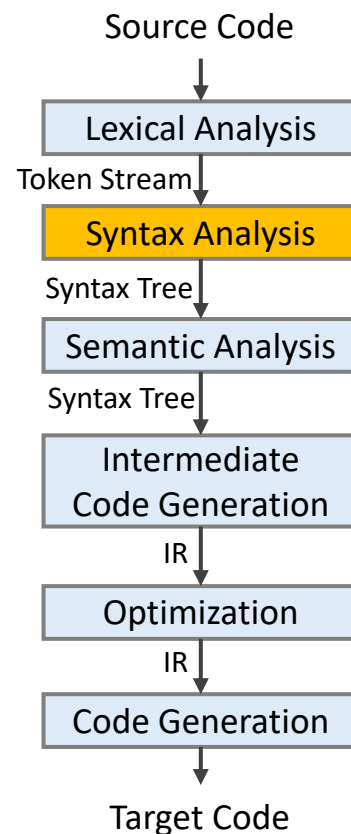
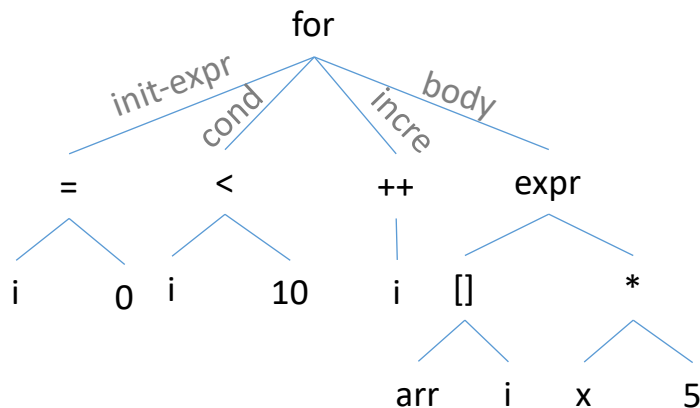
- 解析源程序对应的token序列，生成语法分析结构（**syntax tree**, 语法分析树）

- 输入：单词流，输出：语法树
- 输入程序是否符合语法规则？

□ x^*+

□ $a += 5;$

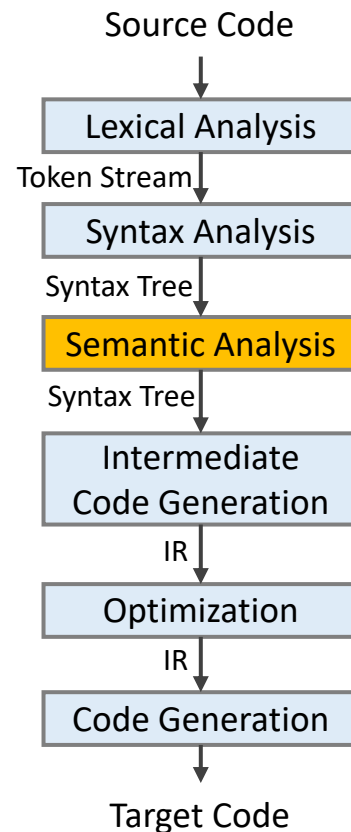
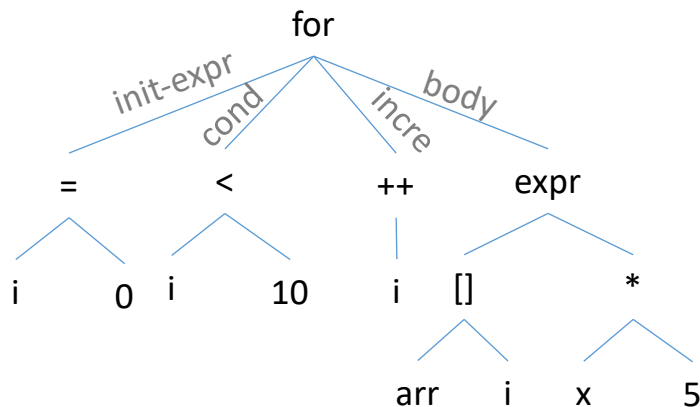
```
void main()  
{  
    int arr[10], i, x = 1;  
    for (i = 0; i < 10; i++)  
        arr[i] = x * 5;  
}
```



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;  
}
```



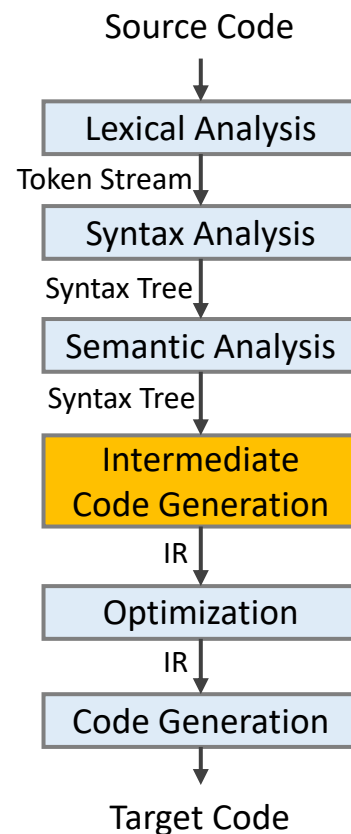
Intermediate Code[中间代码生成]

- 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
```

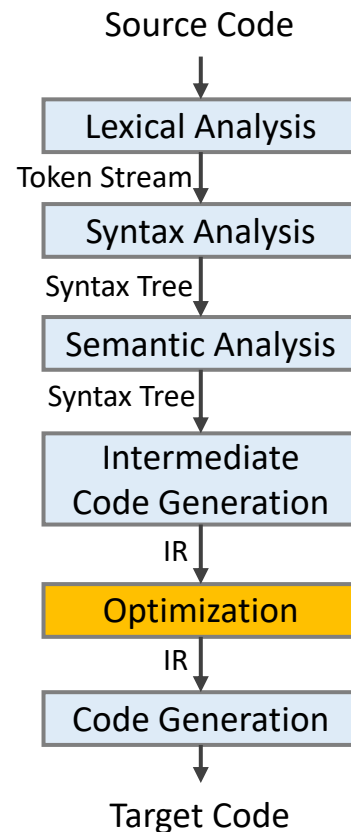


Code Optimization[代码优化]

- 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
```



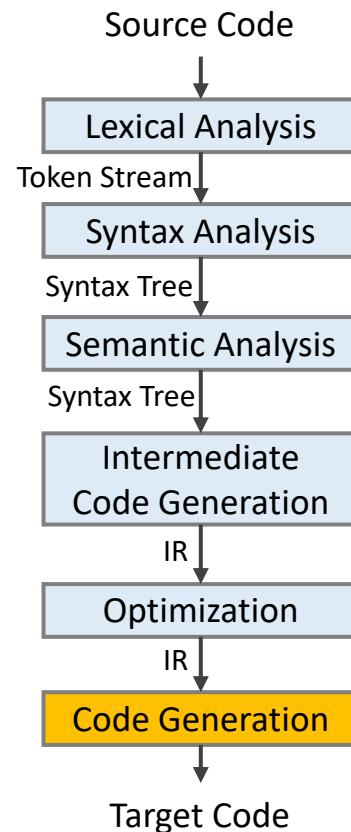
Target Code[目标代码生成]

- 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

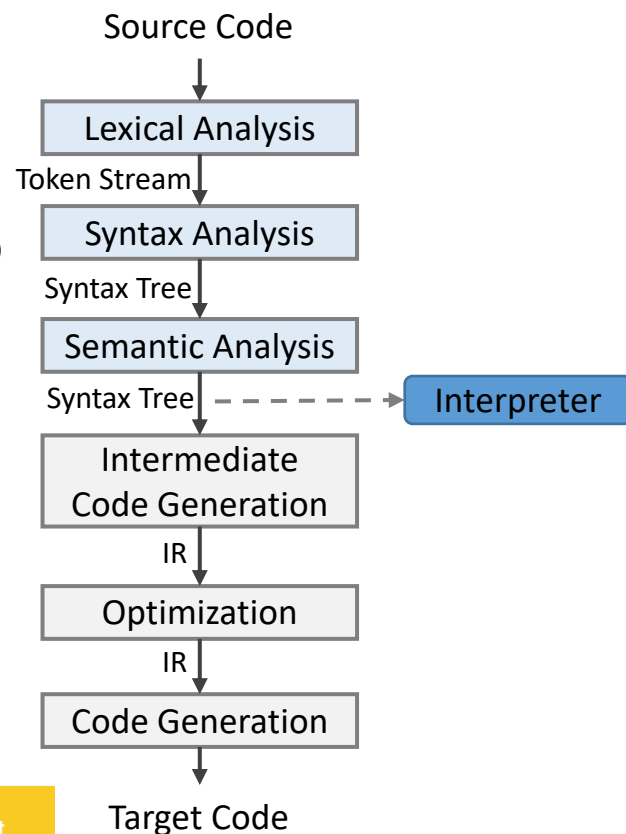
```
void main()  
{  
    int arr[10], i, x = 1;  
    for (i = 0; i < 10; i++)  
        arr[i] = x * 5;  
}
```

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?

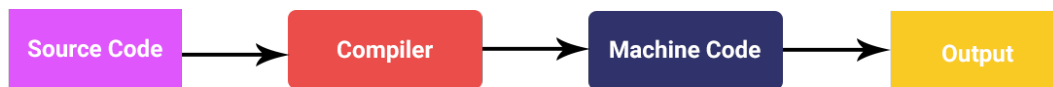


Interpret vs Compile[解释 vs. 编译]

- **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



Compiler Works

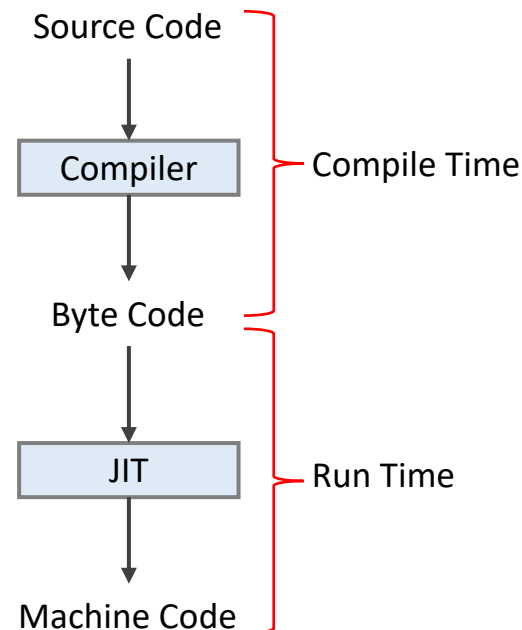


Interpreter Works



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



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**

