中山大學
SUN YAT-SEN UNIVERSITY

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

DCS290

Compilation Principle

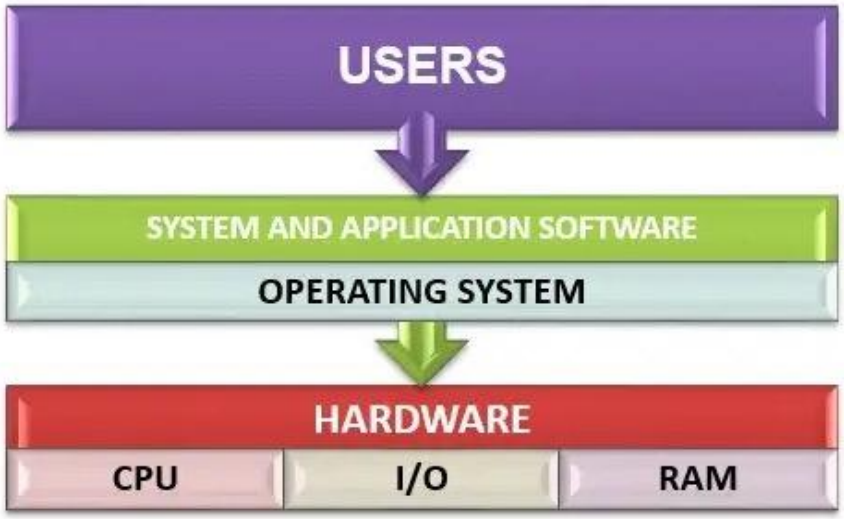
编 译 原 理

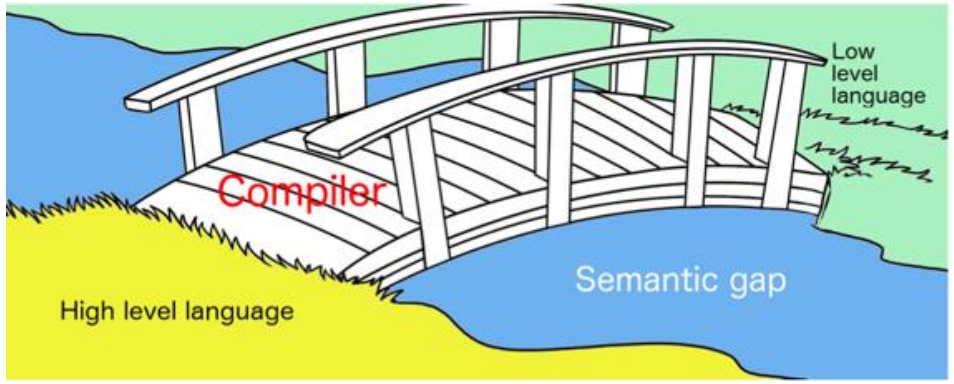
第一章：概述

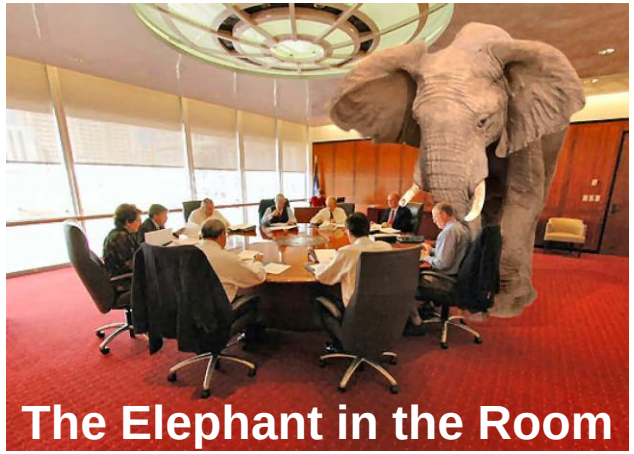
郑馥丹
zhengfd5@mail.sysu.edu.cn

1. Compiler[编译器]

- Compiler is a 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







2. Compiler History[编译器的发展]

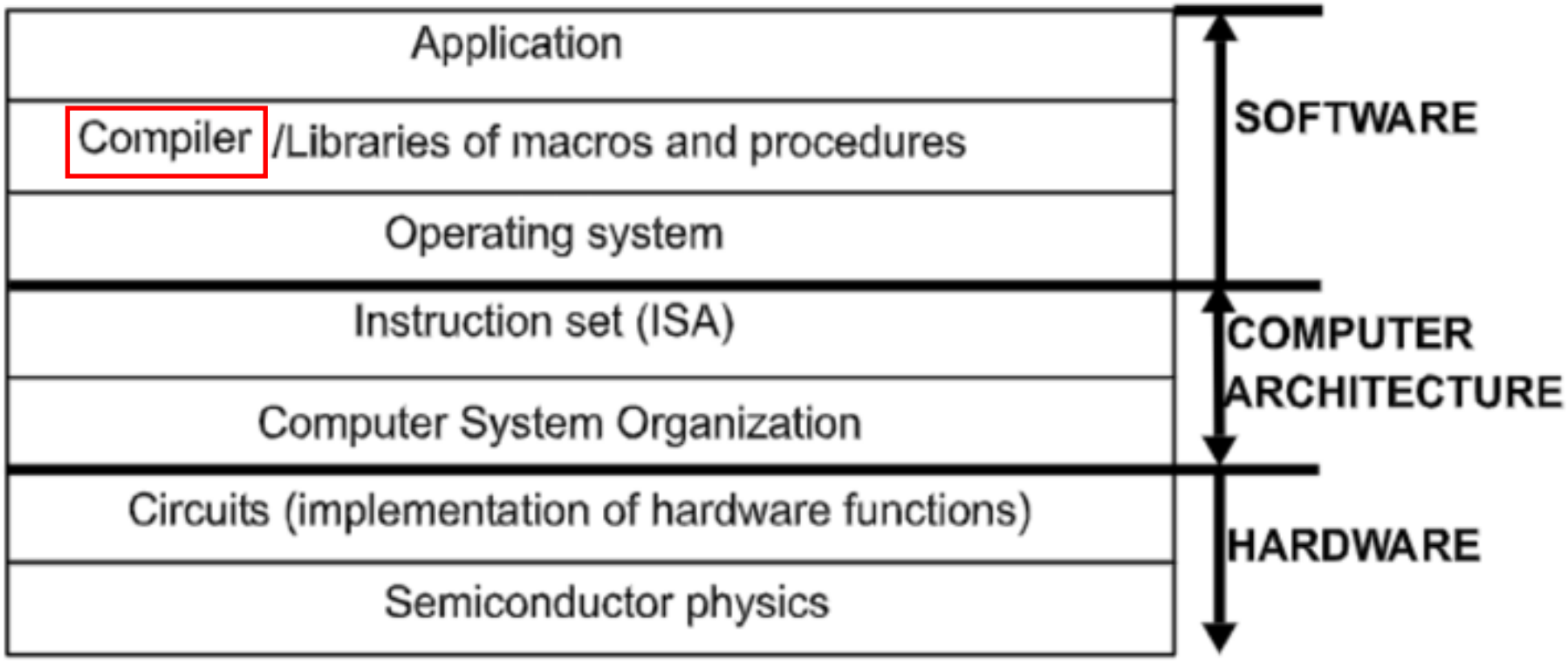
- Compiler origins
  - 1952: A-0, term ‘compiler’ (Grace Hopper)
  - 1957: FORTRAN, first commercial compiler (John Backus)
  - 1962: LISP, self-hosting and GC (Tim Hart and Mike Levin)
  - 1984: GNU Compiler Collection (Stallman)
  - 2000: LLVM (Vikram Adve and Chris Lattner)(2012ACM软件系统奖)
- Turing 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

3. Why Compilation?[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
  - 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

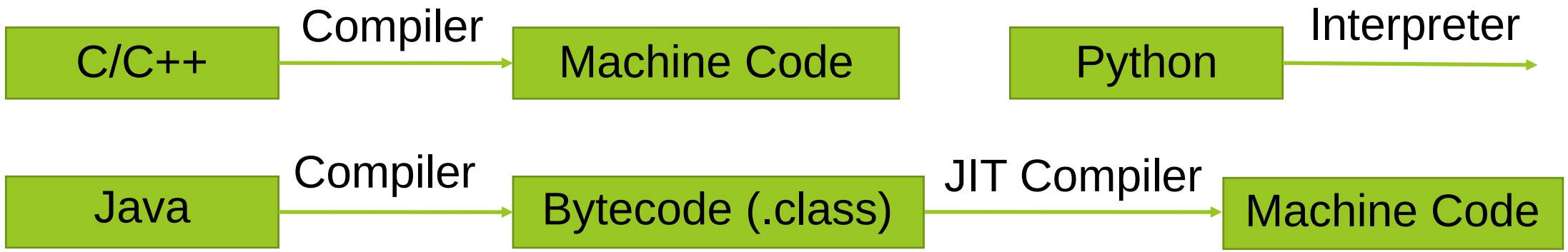
4. What is Compilation?[什么是编译?]

- 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



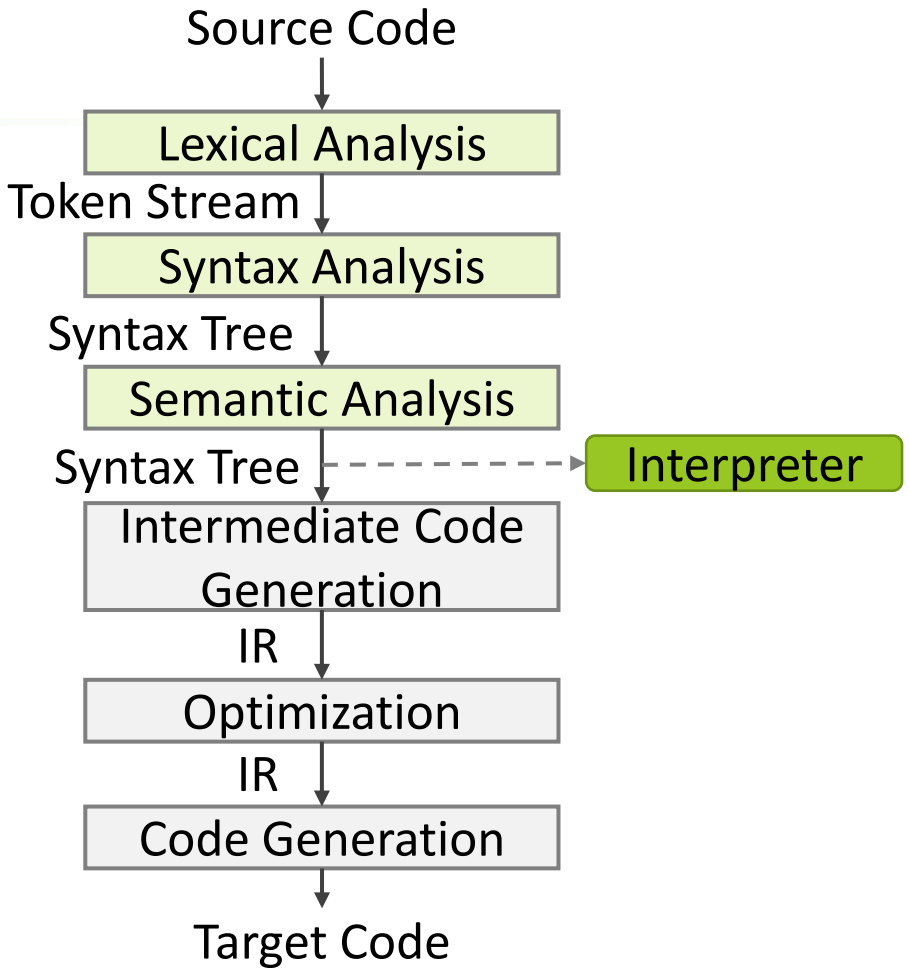
4. What is Compilation?[什么是编译?]

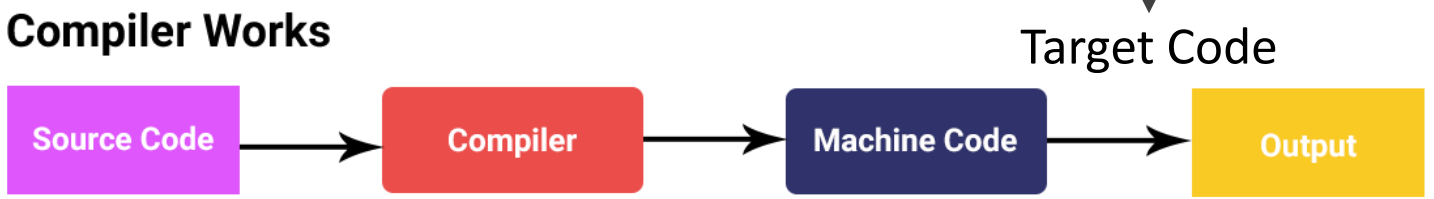
- 不同语言有不同的实现方式
  - “底层”语言通常使用编译
    - C, C++
  - “高级”语言通常是解释性
    - Python, Ruby
  - 有些使用混合的方式
    - Java：编译 + 即时编译（JIT, Just-in-Time）



4. What is Compilation?[什么是编译?]

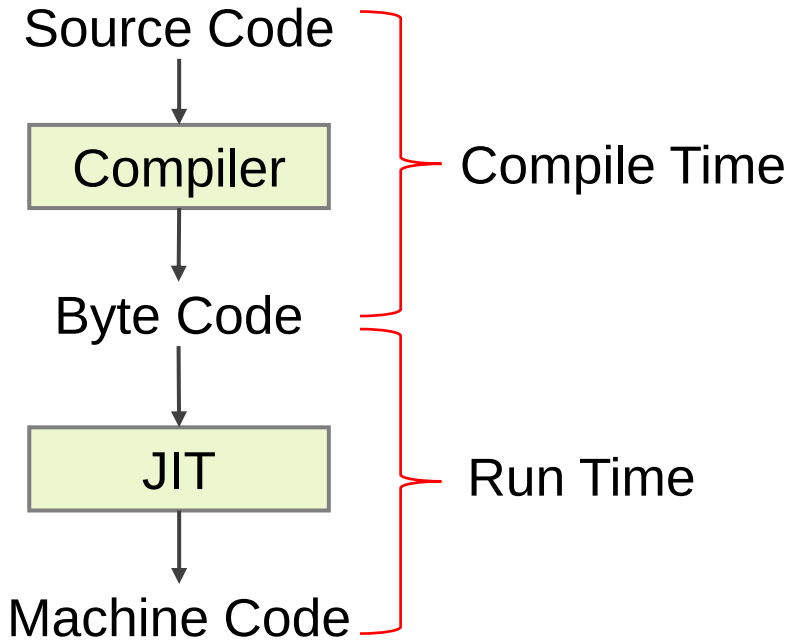
- 编译：翻译成机器语言后方能运行
  - 目标程序独立于源程序（修改→再编译→运行）
  - 分析程序上下文，易于整体性优化
  - 性能更好（因此，核心代码通常C/C++）
- 解释：源程序作为输入，边解释边执行
  - 不生成目标程序，可迁移性高
  - 逐句执行，很难进行优化
  - 性能通常不会太好





4. What is Compilation?[什么是编译?]

- 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



5. C Compilation[C语言编译]

- 源程序 (hello.c) → 可执行文件 (./hello)

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

① 预处理阶段（Preprocessor）

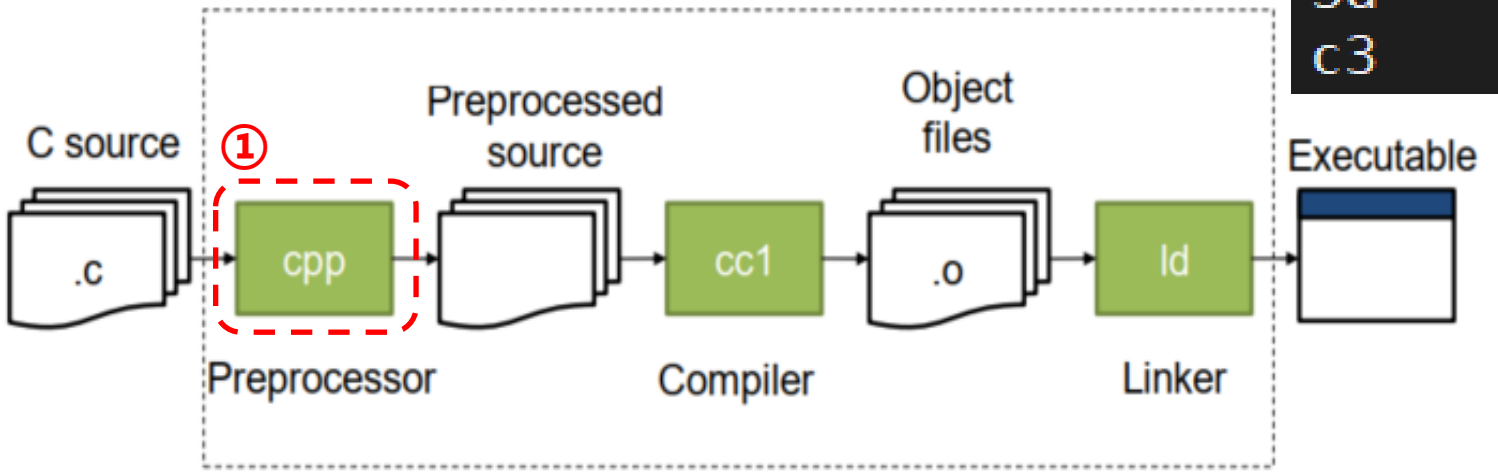
✓汇合源程序，展开宏定义，生成.i文件（另一个C文本文件）

✓hello.c → hello.i （源文件→处理后的源文件）

✓clang -E hello.c -o hello.i

```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
```



5. C Compilation[C语言编译]

- 源程序 (hello.c) → 可执行文件 (./hello)

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

② 编译阶段（Compiler）

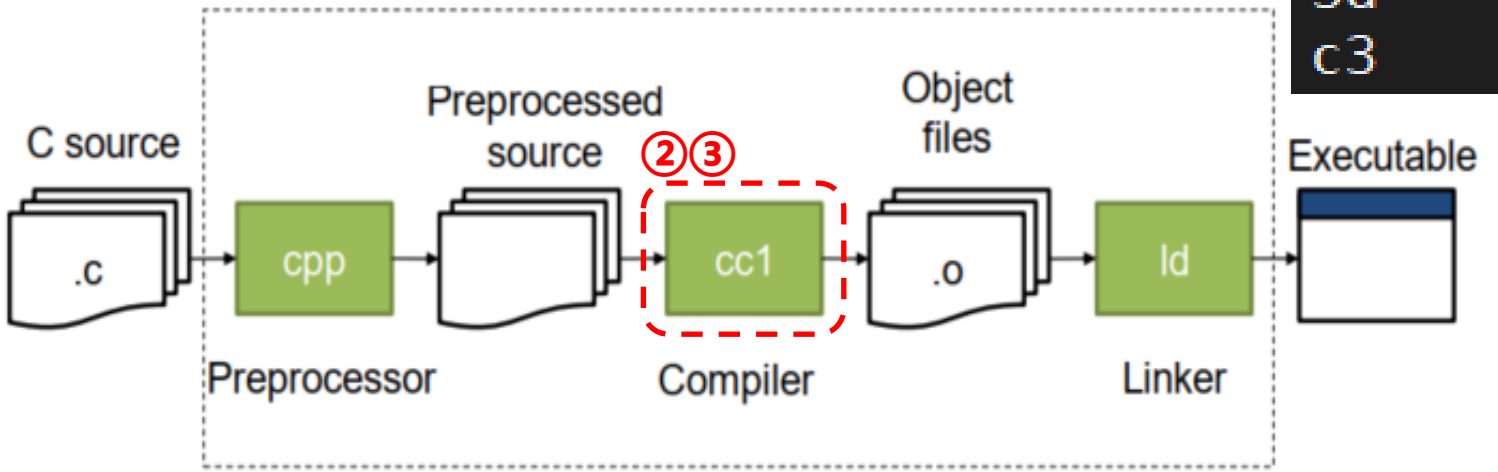
✓hello.i → hello.s （处理后的源文件→汇编代码文件）

✓clang -emit-llvm hello.i -S -o hello.ll

✓clang -S hello.ll -o hello.s

```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
```



5. C Compilation[C语言编译]

- 源程序 (hello.c) → 可执行文件 (./hello)

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

③ 汇编阶段（Assembler）

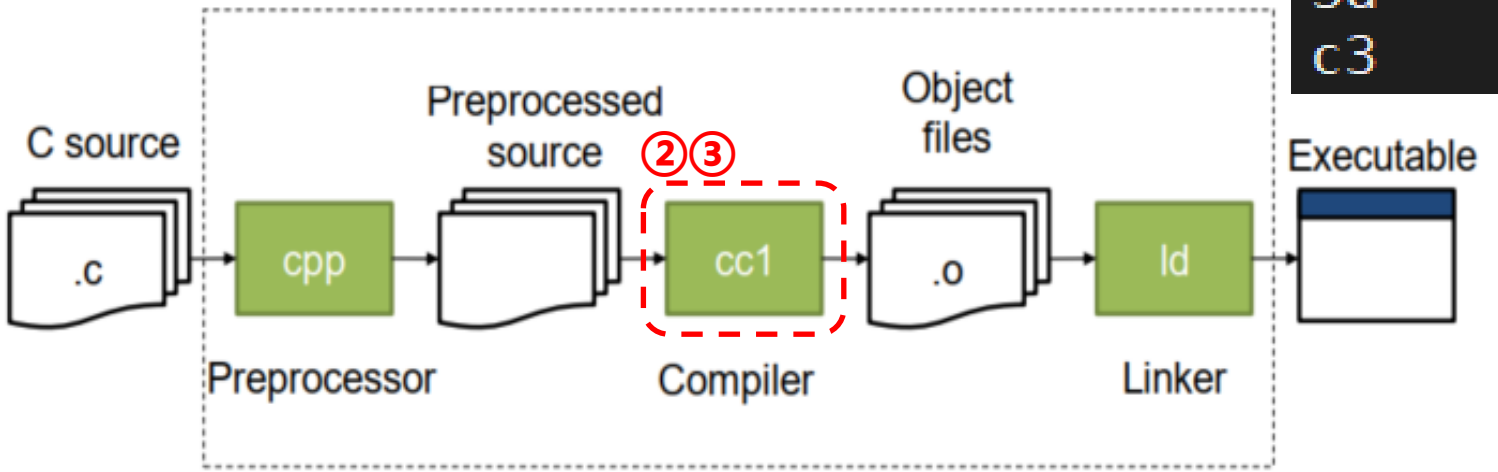
✓.s文件转为.o可重定位对象（relocatable object）文件 （机器指令）

✓hello.s → hello.o （汇编代码文件→目标文件/机器指令文件）

✓clang -c hello.s -o hello.o

```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
```



5. C Compilation[C语言编译]

- 源程序 (hello.c) → 可执行文件 (./hello)

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

④ 连接阶段（Linker/Loader）

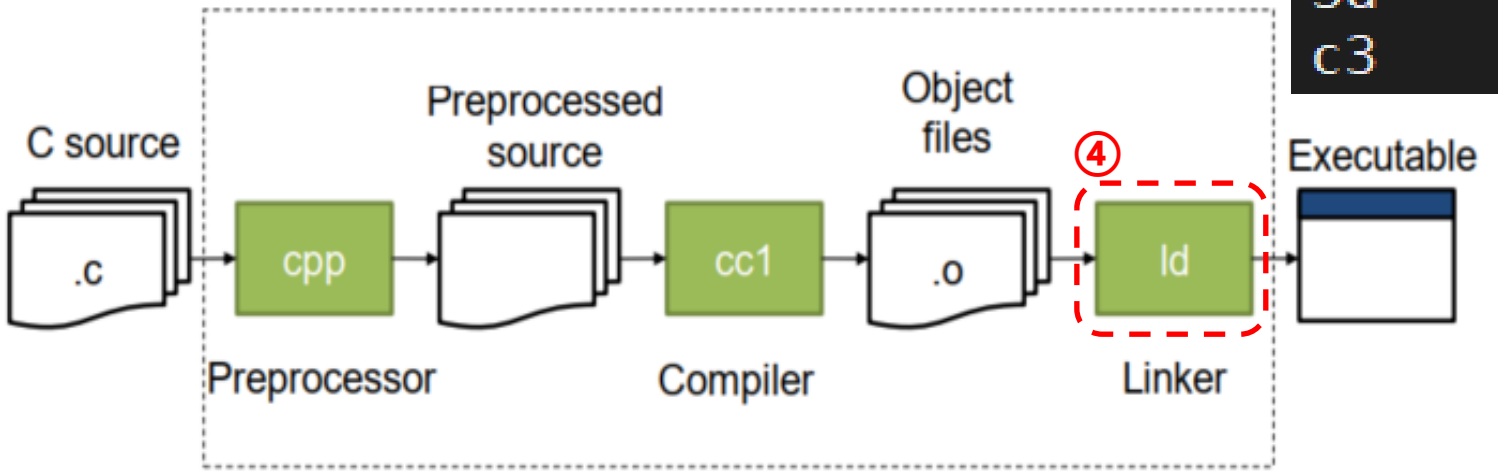
✓连接库代码从而生成可执行（executable）文件（机器指令）

✓hello.o → hello （目标文件→可执行文件）

✓clang hello.o -o hello

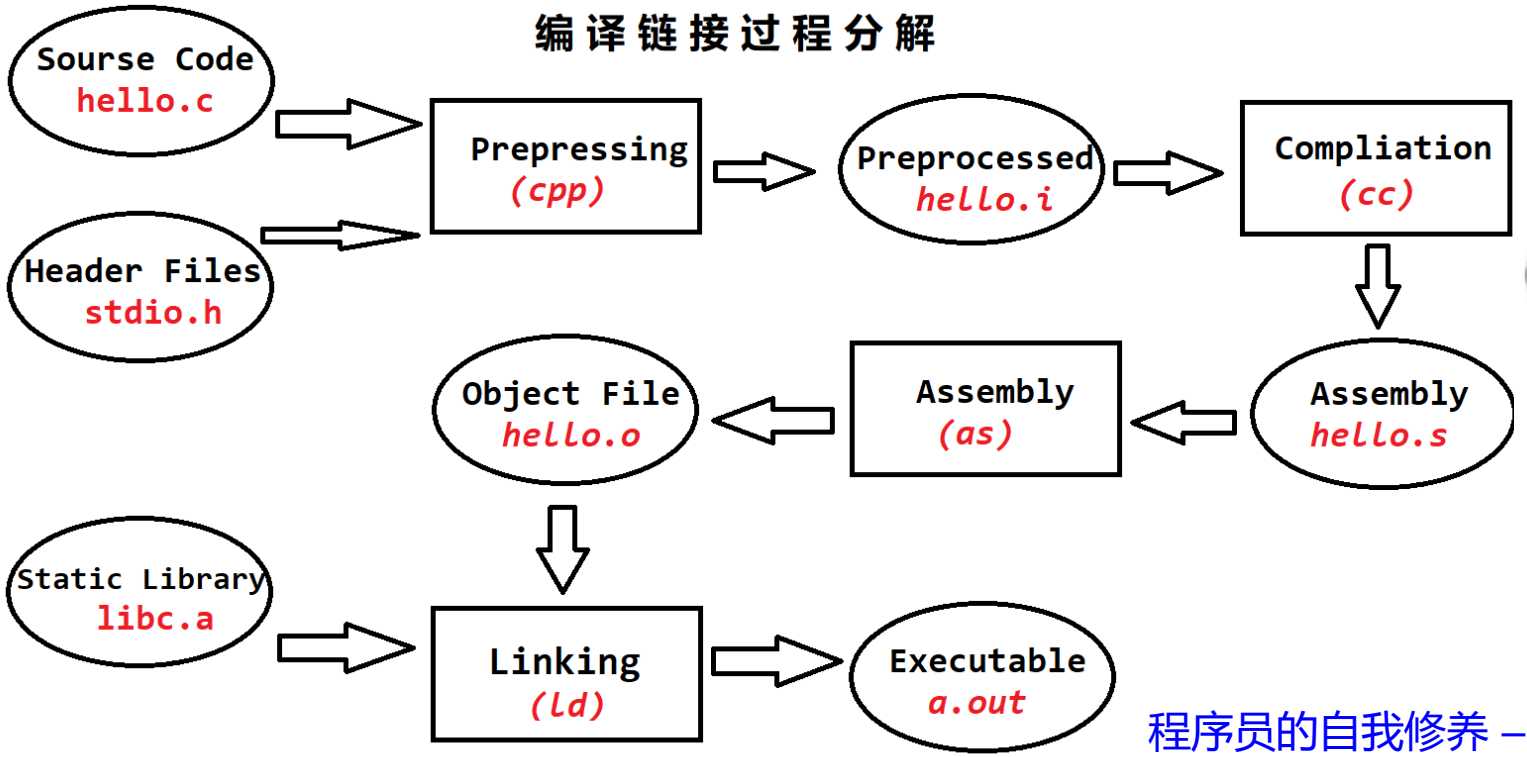
```c
#include <stdio.h>

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

   return 0;
}
```

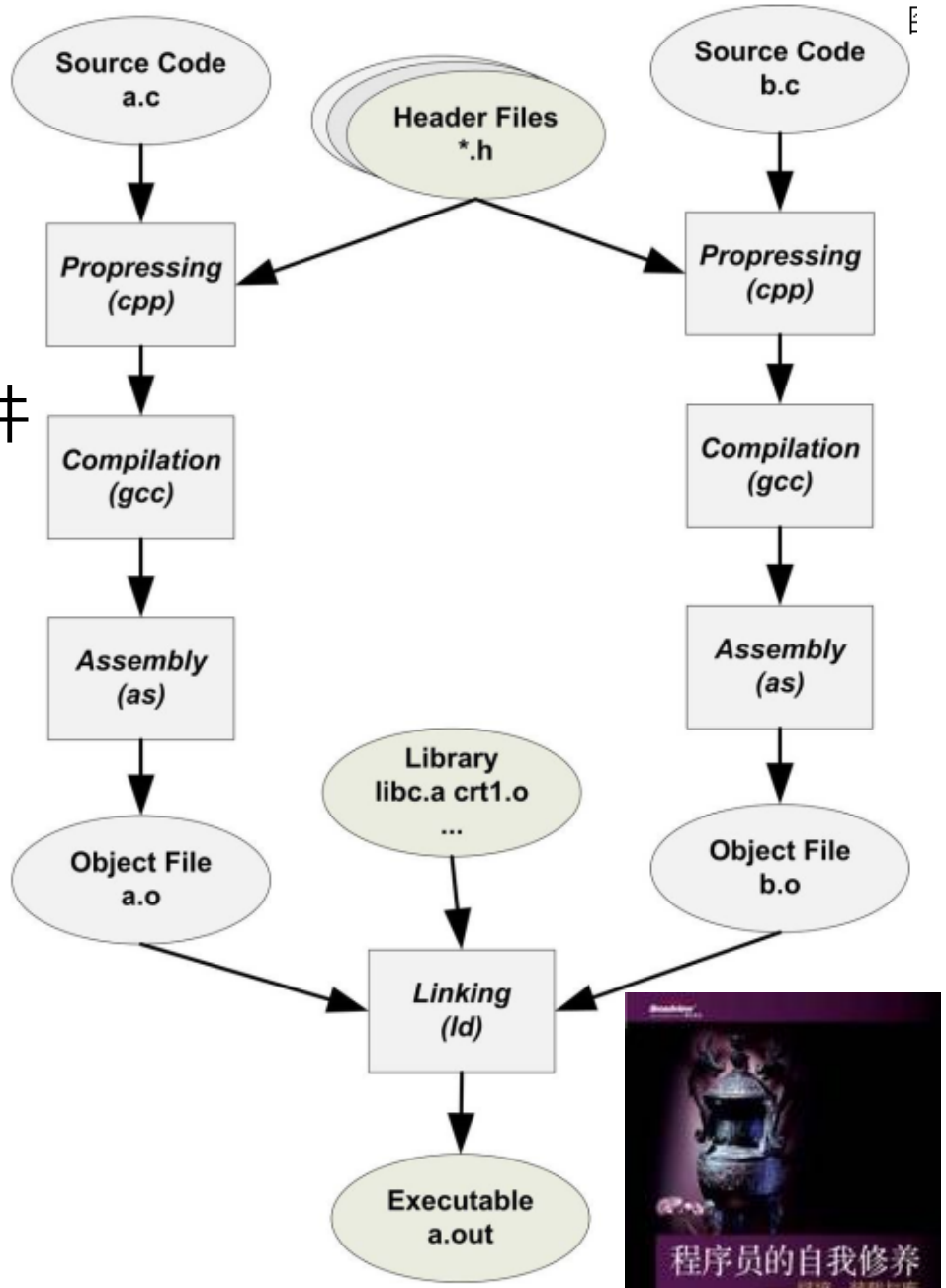
```
55
48 89 e5
bf d0 05 40 00
e8 d5 fe ff ff
b8 00 00 00 00
5d
c3
```



5. C Compilation[C语言编译]

- 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

5. C Compilation[C语言编译]

- $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.
```

5. C Compilation[C语言编译]

```
#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.
```

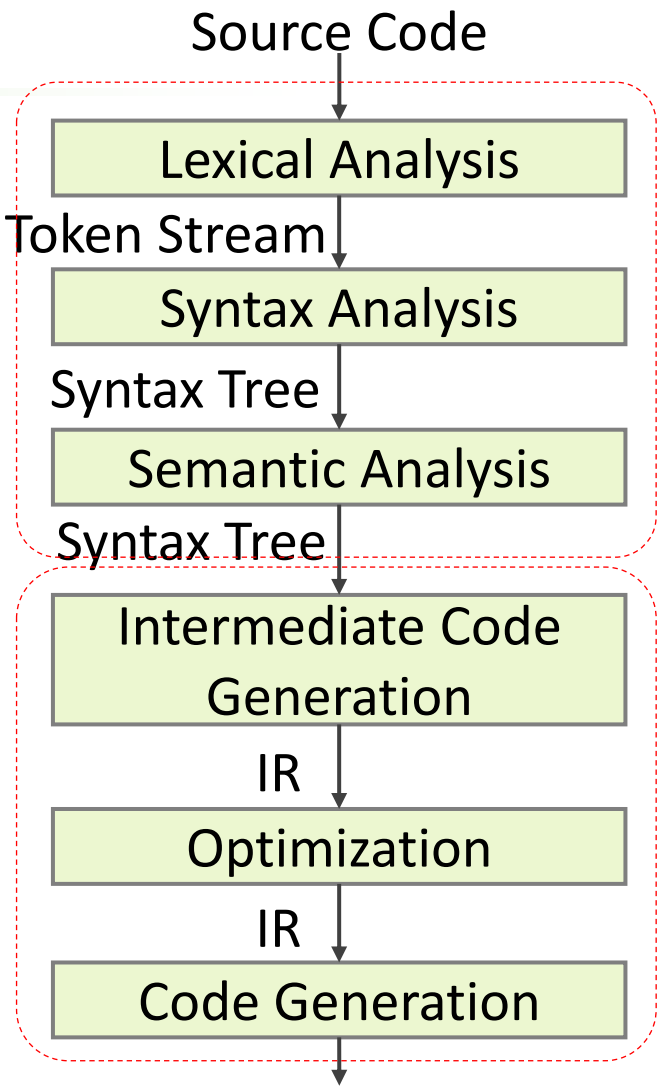
5. C Compilation[C语言编译]

```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:6:27: 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
```

6. 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



6. Compilation Procedure[编译过程]

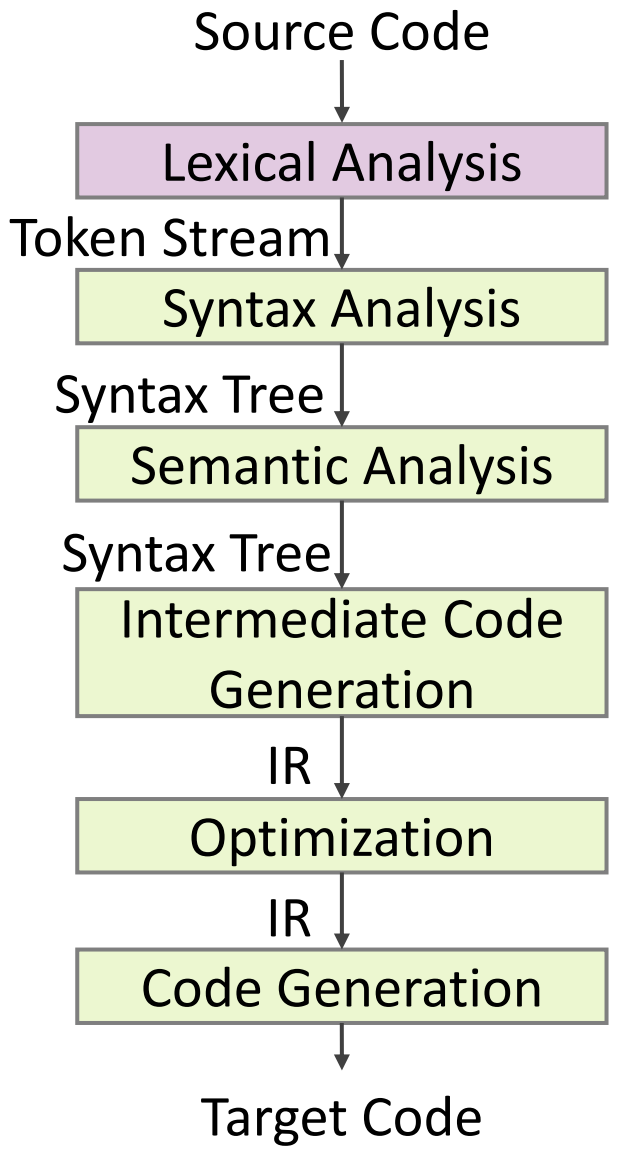
(1) Lexical Analysis[词法分析]

- 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)
symbol(()
id(i)
symbol(=)
num(0)
symbol(;)
id(i)
symbol(<)
num(10)
symbol(;)
id(i)
symbol(++)
symbol())

id(arr)
symbol([)
id(i)
symbol(])
symbol(=)
id(x)
symbol(*)
num(5)
symbol(;)



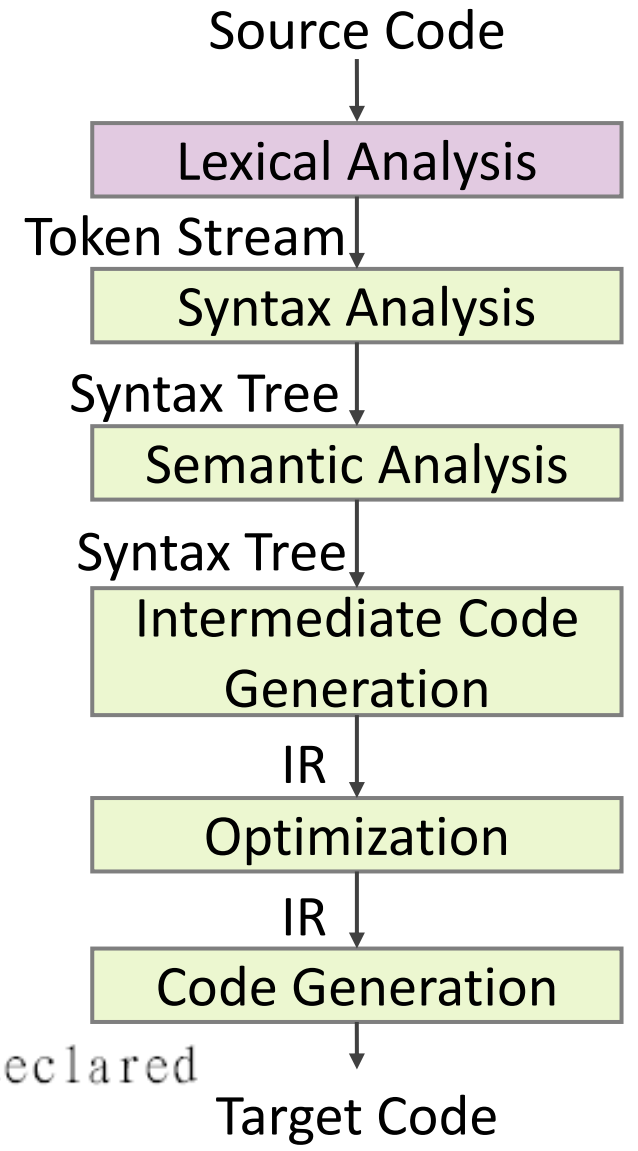
6. Compilation Procedure[编译过程]

(1) 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？

```
#include<iostream.h>

void main(){
    cout<<"Hello world!"<<endl;
    int 1a;
}
```

```
error C2059: syntax error : 'bad suffix on number'
warning C4091: '' : ignored on left of 'int' when no variable is declared
error C2143: syntax error : missing ';' before 'constant'
error C2146: syntax error : missing ';' before identifier 'a'
error C2065: 'a' : undeclared identifier
```

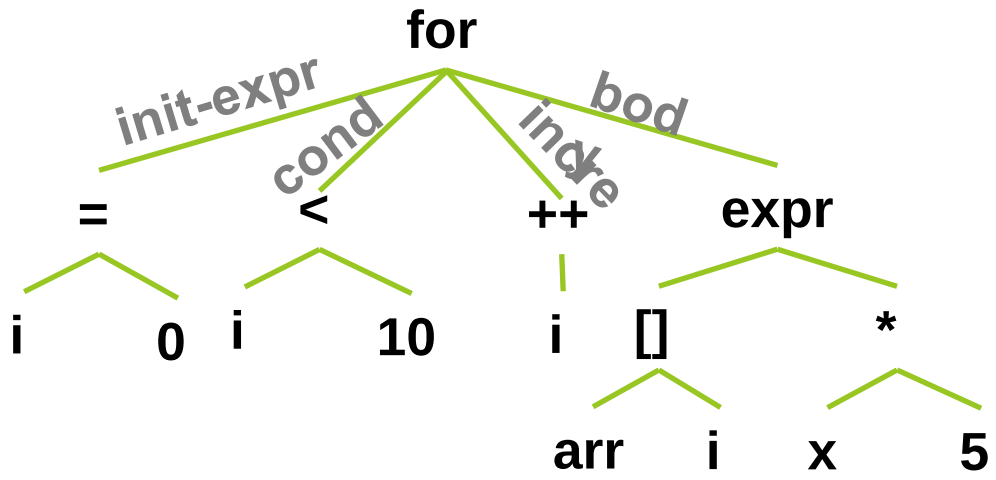


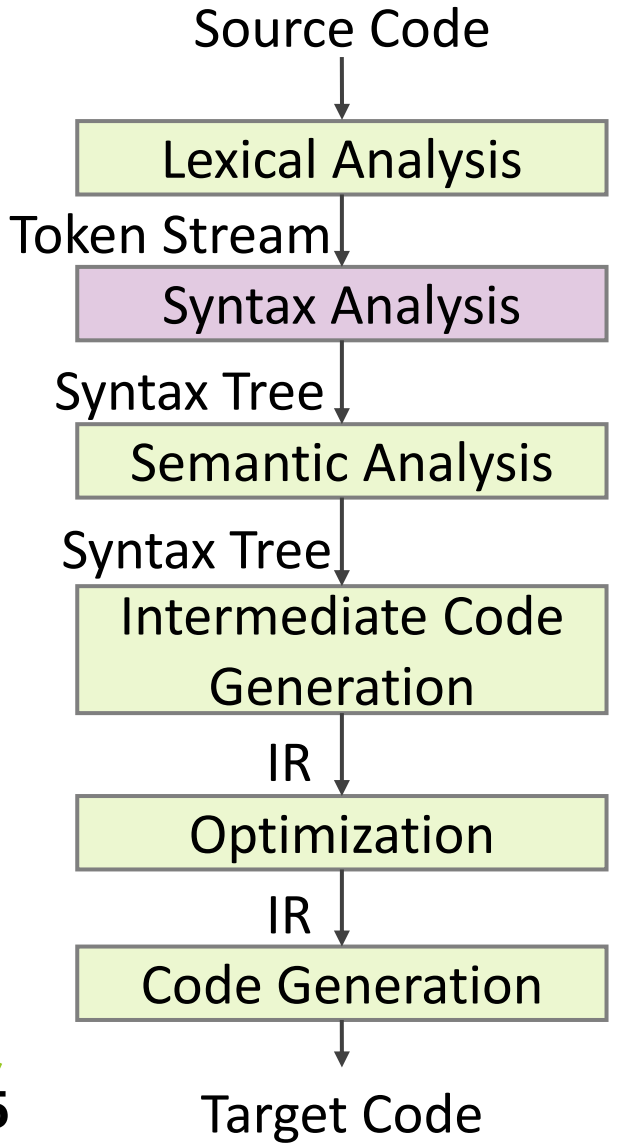
6. Compilation Procedure[编译过程]

(2) 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;
}
```





6. Compilation Procedure[编译过程]

(2) 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？

else没有匹配的if

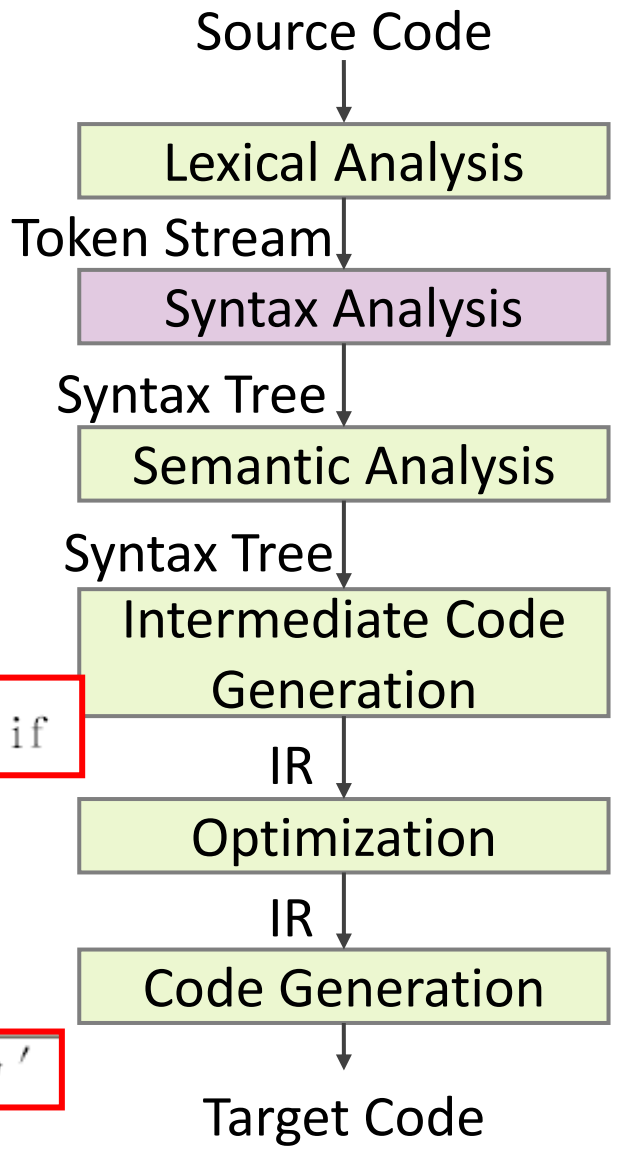
表达式缺少分号结尾

```
#include<iostream.h>

void main(){
    cout<<"Hello world!"<<endl;
    else cout<<"oh no!"<<endl;
}
```

```
error C2181: illegal else without matching if
```

```
#include<iostream.h>

void main(){
    cout<<"Hello world!"<<endl
}
```

```
syntax error : missing ';' before '}'
```

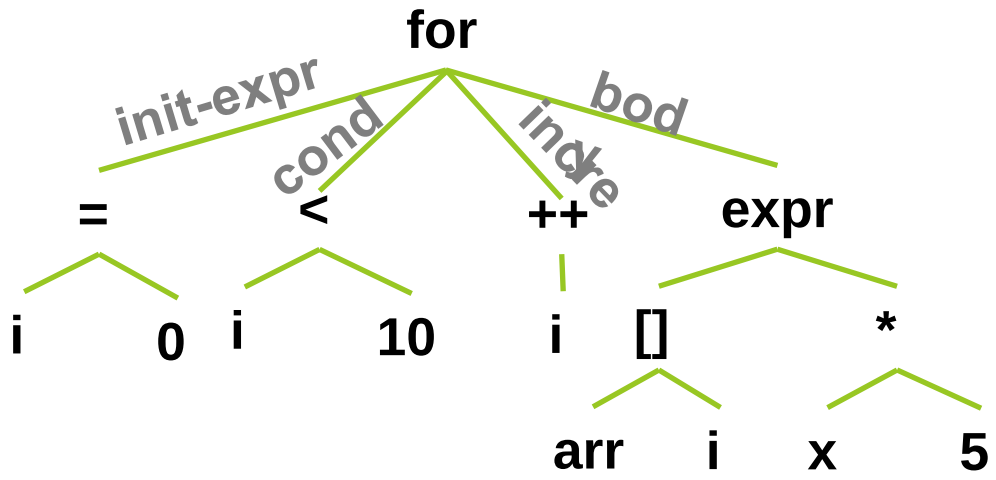


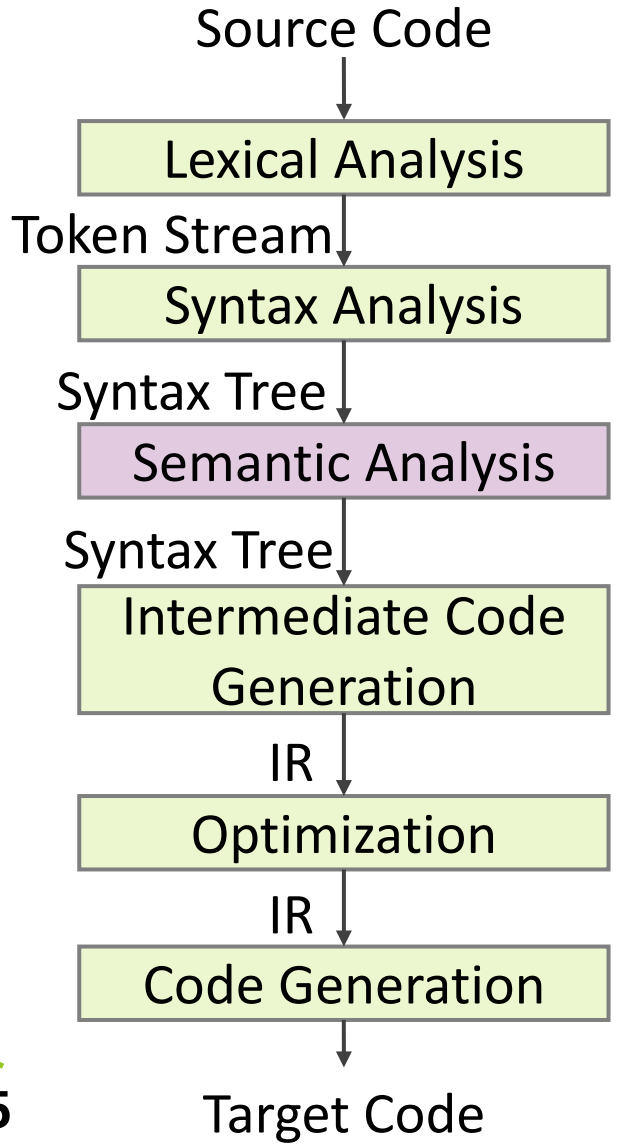
6. Compilation Procedure[编译过程]

(3) 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;
}
```





6. Compilation Procedure[编译过程]

(3) Semantic Analysis[语义分析]

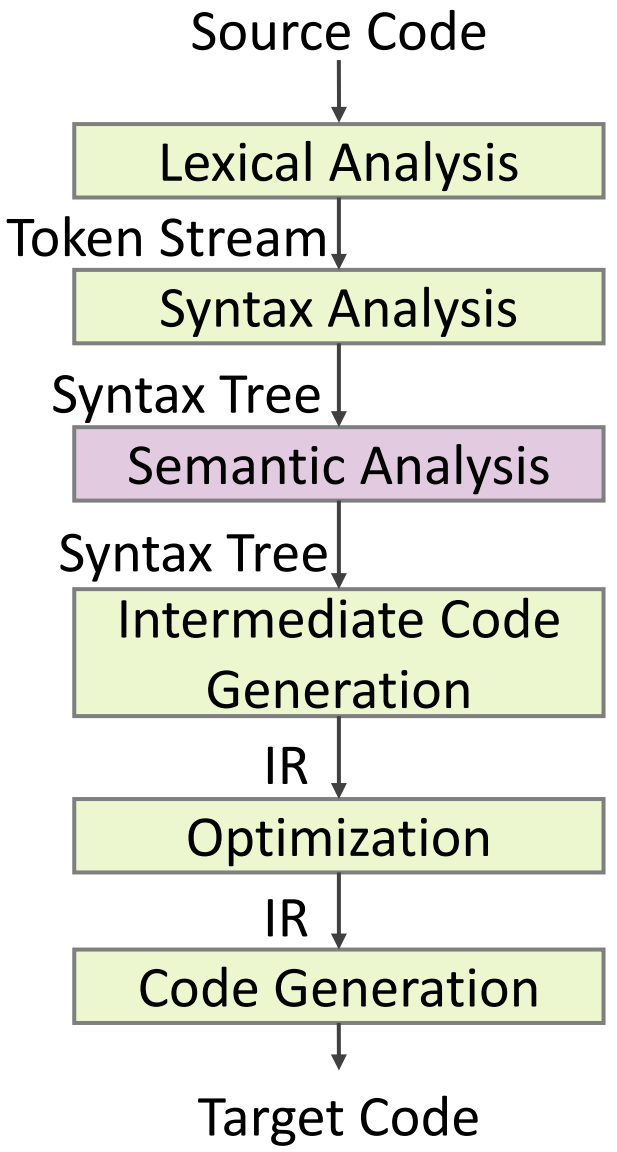
- 基于语法结果进一步分析语义
  - 输入：语法树，输出：语法树+符号表
  - 收集标识符的属性信息（type，scope等）
  - 输入程序是否符合语义规则？ 数组下标越界

    声明和使用的函数没有定义

    零作除数

```
#include<iostream.h>

void main() {
    cout<<"Hello world!"<<endl;
    int a=10/0;
}
```

```
error C2124: divide or mod by zero
```



6. Compilation Procedure[编译过程]

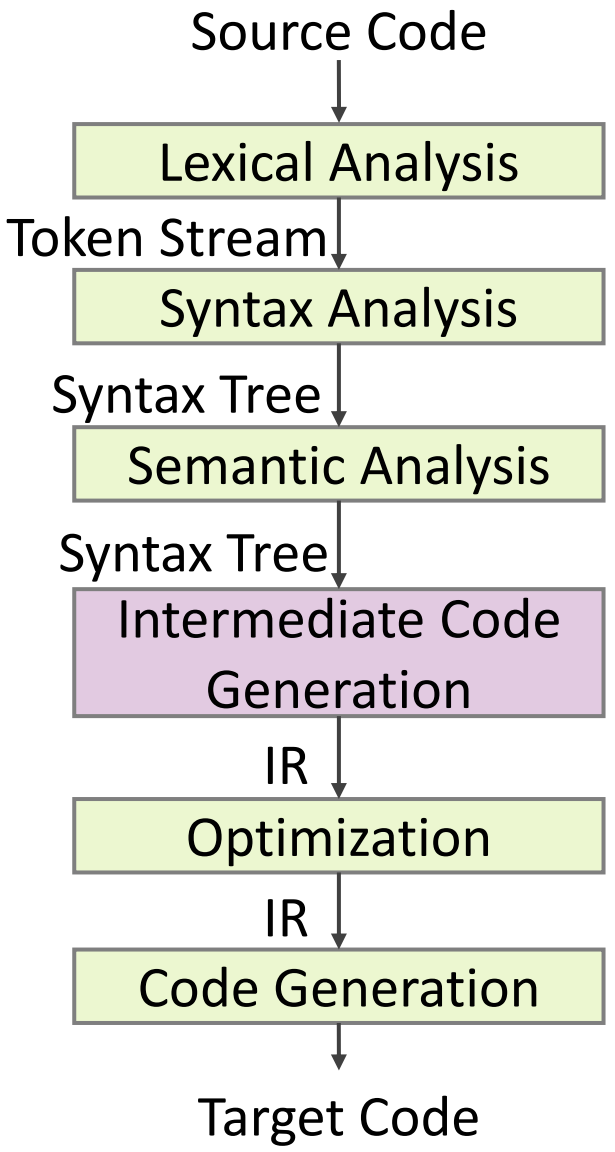
(4) Intermediate Code[中间代码生成]

——从这里开始真正做翻译工作

- 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
```



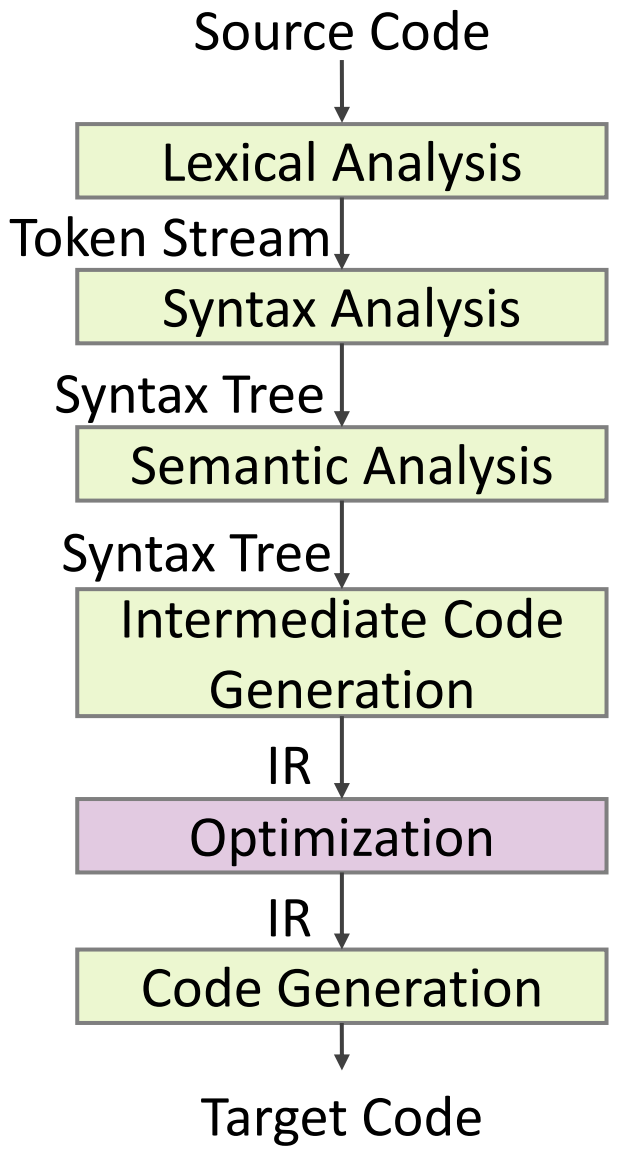
6. Compilation Procedure[编译过程]

(5) Code Optimization[代码优化]

- 加工变换中间代码使其更好（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
```



6. Compilation Procedure[编译过程]

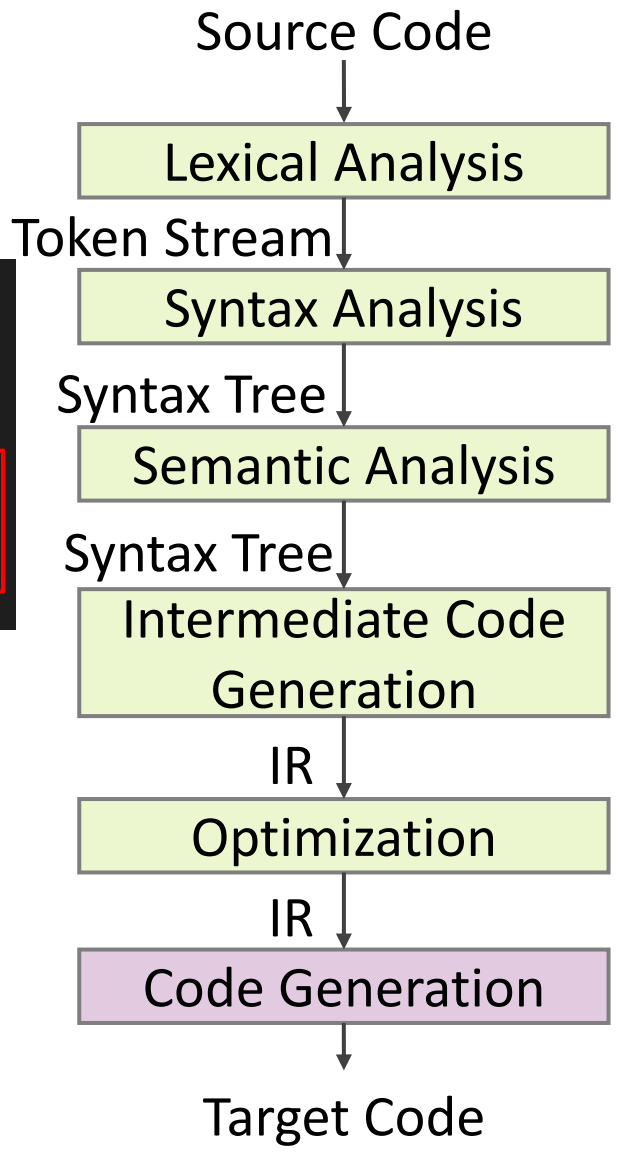
(6) Target Code[目标代码生成]

- 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

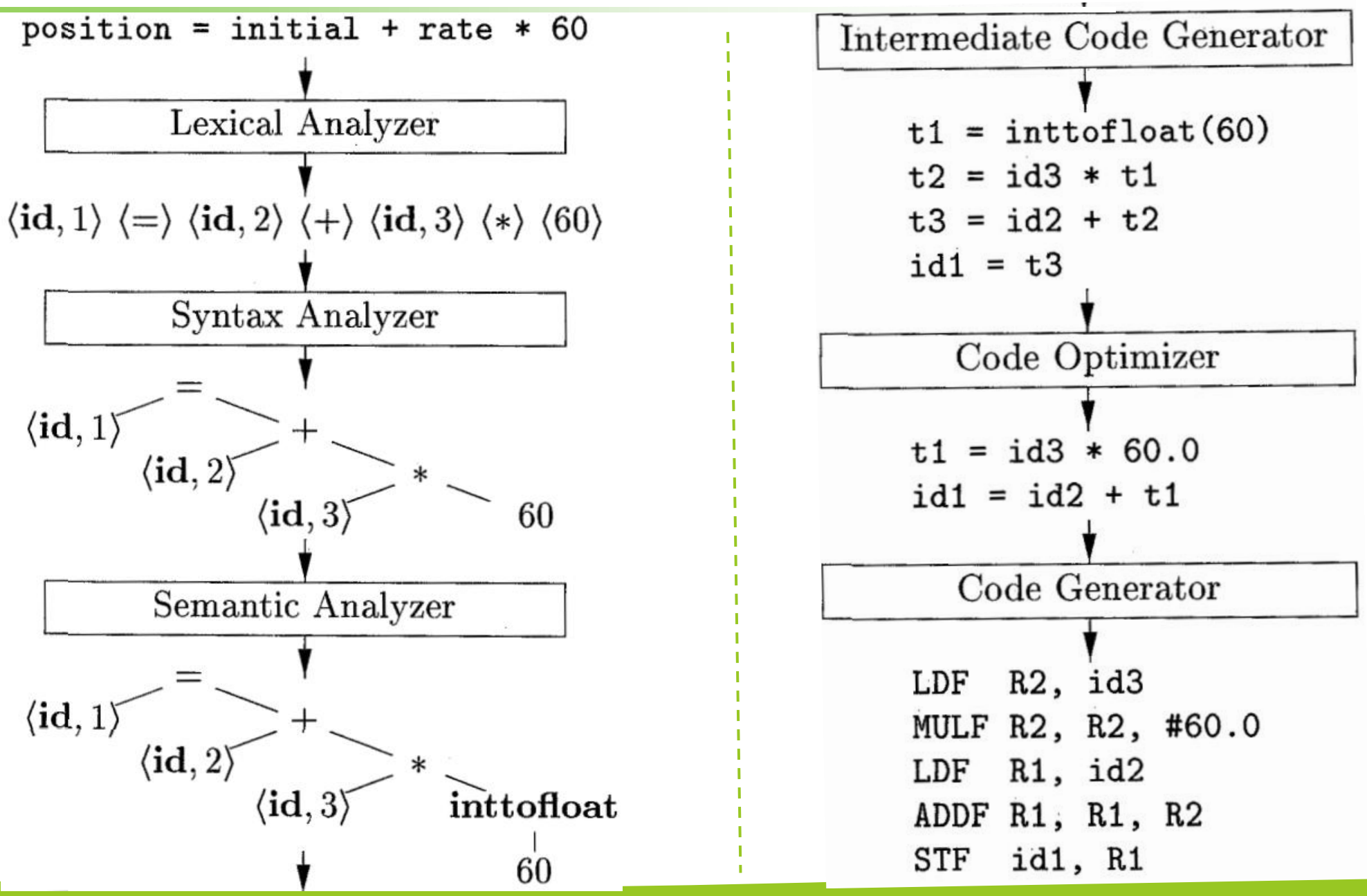
    例如：寄存器及访存优化

```
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?
```

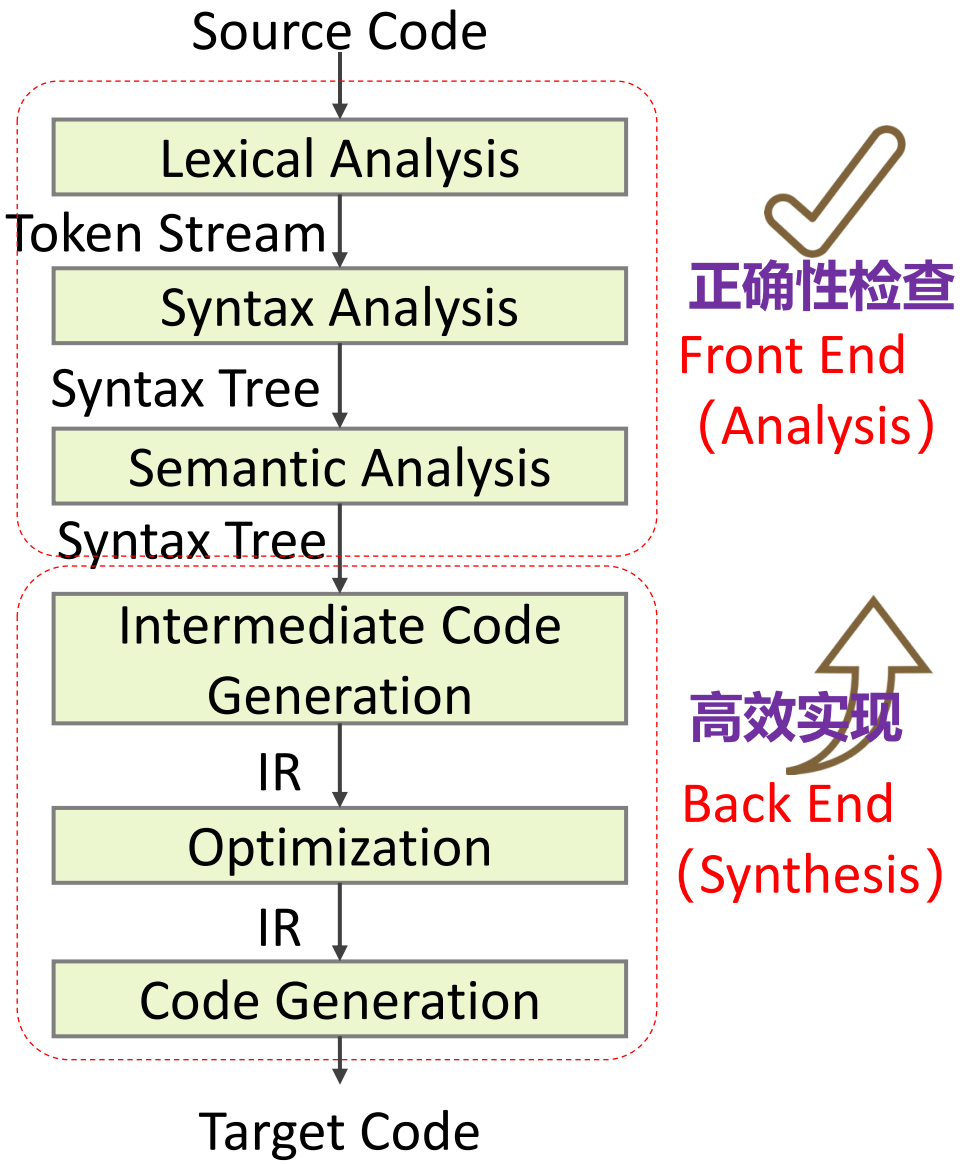


6. Compilation Procedure[编译过程]



小结

- 内容
  - 什么是编译程序
  - 编译的各个阶段
- 重点掌握
  - 编译的基本概念
  - 编译程序的功能和结构
  - 编译的完整流程的各个阶段
  - 以及他们怎样作为一个整体完成编译任务



随堂小练习

在词法分析的基础上，将单词序列分解成各类语法短语。（B）

把源程序翻译成中间代码。（D）

从左到右一个字符一个字符地读入源程序，对构成源程序的字符流进行扫描和分解，从而识别出一个个单词。（A）

把中间代码变换成特定机器上的绝对指令代码或可重定位的指令代码或汇编指令代码。（F）

对中间代码进行等价变换，以便生成更高效的目标代码。（E）

审查源程序有无语义错误，为代码生成阶段收集类型信息。（C）

A.词法分析

B.语法分析

C.语义分析

D.中间代码生成

E.代码优化

F.目标代码生成

随堂小练习

- 请说明以下错误是在编译的哪个阶段（词法分析、语法分析、语义分析、代码生成）报告的
  - （1）else没有匹配的if **语法分析**
  - （2）数组下标越界 **语义分析、代码生成**
  - （3）使用的函数没有定义 **语义分析**
  - （4）在数中出现非数字字符 **词法分析**