2 编译过程概述

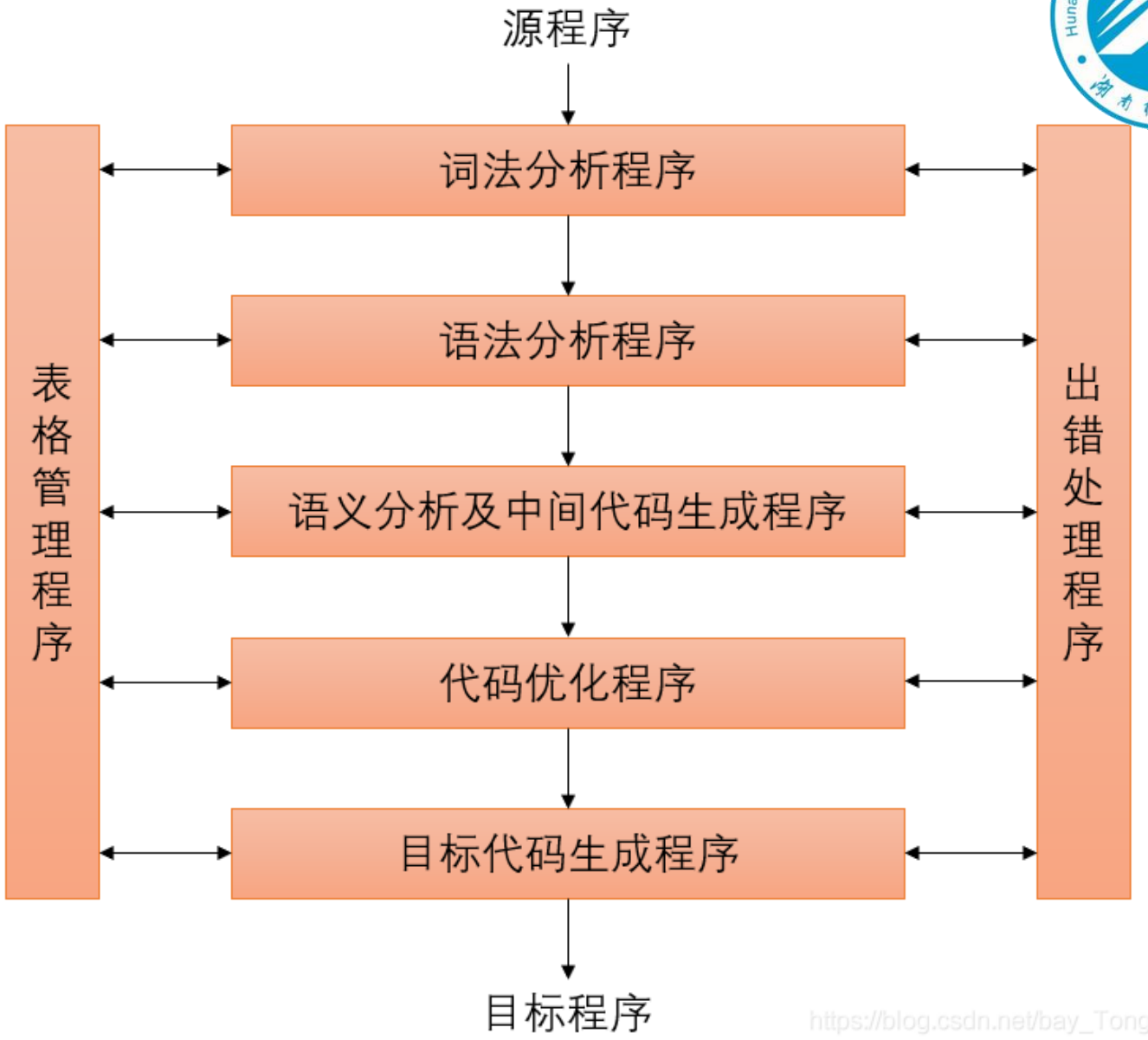
杨策

2 编译过程概述

- 编译过程简述
- 编程语言简述

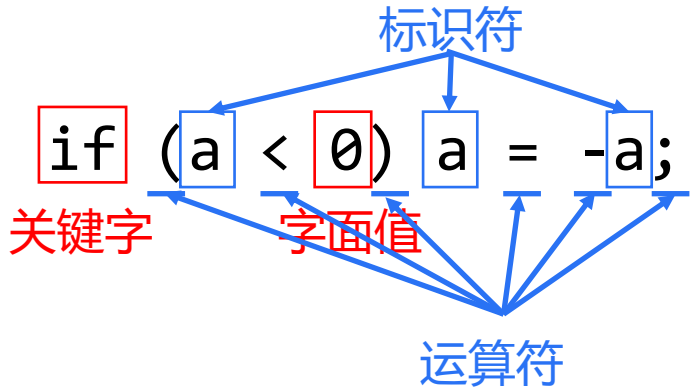
编译过程

- 输入
  - 高级语言源程序
- 输出
  - 低级语言/机器语言
- 实用模块
  - 表管理
  - 错误处理



词法分析

- 输入源程序，识别出单词符号
- 单词符号：关键字、变量、运算符
- 依据规则：语言的词法
- 有效工具：正则表达式、有限自动机
- 线性分析



空白一般不解析出词法符号

语法分析

- 任务：在词法分析的基础上，根据语言的语法规则，把单词符号串分解成各类语法单位（语法范畴），如短句、子句、句子、程序段和程序等。
- 通过语法分析，确定整个输入串是否构成语法上正确的“程序”
- 语法分析依循的是语言的语法规则
- 语法规则通常用上下文无关文法描述

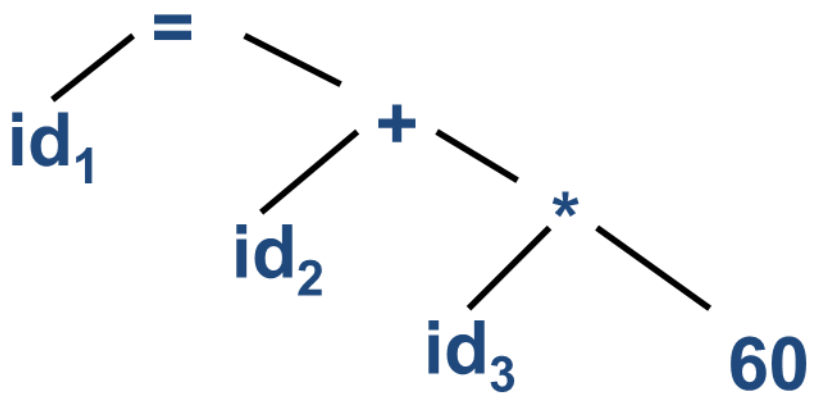
语法分析

输入：词法符号串

id1 = id2 + id3 * 60

算术表达式，赋值语句

输出：抽象语法树



语义分析和中间代码生成

- 对各种语法范畴进行静态语义检查
- 如果语义正确，则进行中间代码的翻译
  - 中间代码：一种含义明确、便于处理的记号系统，通常独立于具体的硬件
- 依据语言的语义规则，通常使用属性文法描述语义规则
-

语义分析和中间代码生成

- 中间代码可采用四元式、三元式、间接三元式、逆波兰记号和树形表示等等。

算符	左操作数	右操作数	结果

- `Z = ( X + 0.418 ) * Y / w` 翻译成四元式为

序号	算符	左操作数	右操作数	结果
(1)	+	X	0.418	T1
(2)	*	T1	Y	T2
(3)	/	T2	W	Z

优化

- 任务：对前段产生的中间代码进行加工变换，以期在最后阶段能产生更为高效（省时间和空间）的目标代码
- 优化的主要方面：公共子表达式的提取、循环优化、删除无用代码等等
- 优化所依循的原则是程序的等价变换规则

优化

```
for (k=1; k<=100;
   m = i + 10 * k;
   n = j + 10 * k;
}
```

300次加法
200次乘法

序号	OP	ARG1	ARG2	RESULT	注释
(1)	=	1		K	K=1
(2)	j<	100	K	(9)	if (100<K) goto (9)
(3)	*	10	K	T1	T1=10*K
(4)	+	I	T1	M	M=I+T1
(5)	*	10	K	T2	T2=10*K
(6)	+	J	T2	N	N=J+T2
(7)	+	K	1	K	K=K+1
(8)	j			(2)	goto (2)
(9)					

优化

```
for (k=1; k<=100; k
   m = i + 10 * k;
   n = j + 10 * k;
}
```

300次加法

序号	OP	ARG1	ARG2	RESULT	注释
(1)	=	I		M	M=I
(2)	=	J		N	N=J
(3)	=	1		K	K=1
(4)	j<	100	K	(9)	if (100<K) goto (9)
(5)	+	M	10	M	M=M+10
(6)	+	N	10	N	N=N+10
(7)	+	K	1	K	K=K+1
(8)	j			(4)	goto (4)
(9)					

优化

- AlphaDev
  - 强化学习

b Original

```
Memory[0] = A
Memory[1] = B
Memory[2] = C

mov Memory[0] P  // P = A
mov Memory[1] Q  // Q = B
mov Memory[2] R  // R = C

mov R S
cmp P R
cmovg P R  // R = max(A, C)
cmovl P S  // S = min(A, C)
mov S P    // P = min(A, C)
cmp S Q
cmovg Q P  // P = min(A, B, C)
cmovg S Q  // Q = max(min(A, C), B)


mov P Memory[0]  // = min(A, B, C)
mov Q Memory[1]  // = max(min(A, C), B)
mov R Memory[2]  // = max(A, C)
```

c AlphaDev

```
Memory[0] = A
Memory[1] = B
Memory[2] = C

mov Memory[0] P  // P = A
mov Memory[1] Q  // Q = B
mov Memory[2] R  // R = C

mov R S
cmp P R
cmovg P R  // R = max(A, C)
cmovl P S  // S = min(A, C)

cmp S Q
cmovg Q P  // P = min(A, B)
cmovg S Q  // Q = max(min(A, C), B)


mov P Memory[0]  // = min(A, B)
mov Q Memory[1]  // = max(min(A, C), B)
mov R Memory[2]  // = max(A, C)
```

目标代码生成

- 任务：把中间代码变换成特定机器上的低级语言代码
- 最后的翻译，工作有赖于硬件系统结构和机器指令含义
- 目标代码形式
  - 绝对指令代码：目标代码可立即执行
  - 可重定位的指令代码：目标代码在运行前必须借助于一个连接装配程序把各个目标模块（包括系统提供的库模块）连接在一起，确定程序变量（或常数）在主存中的位置，装入内存中指定的起始地址，使之成为一个可以运行的绝对指令代码程序
  - 汇编指令代码：需汇编器汇编之后才能运行

目标代码生成

- 目标代码示例 b=a+2

汇编指令代码

```
MOV R1, a
ADD R1, 2
MOV b, R1
```

可重定位指令

```
0001   01   00   00000000*
0011   01   10   00000010
0100   01   00   00000100*
```

绝对指令

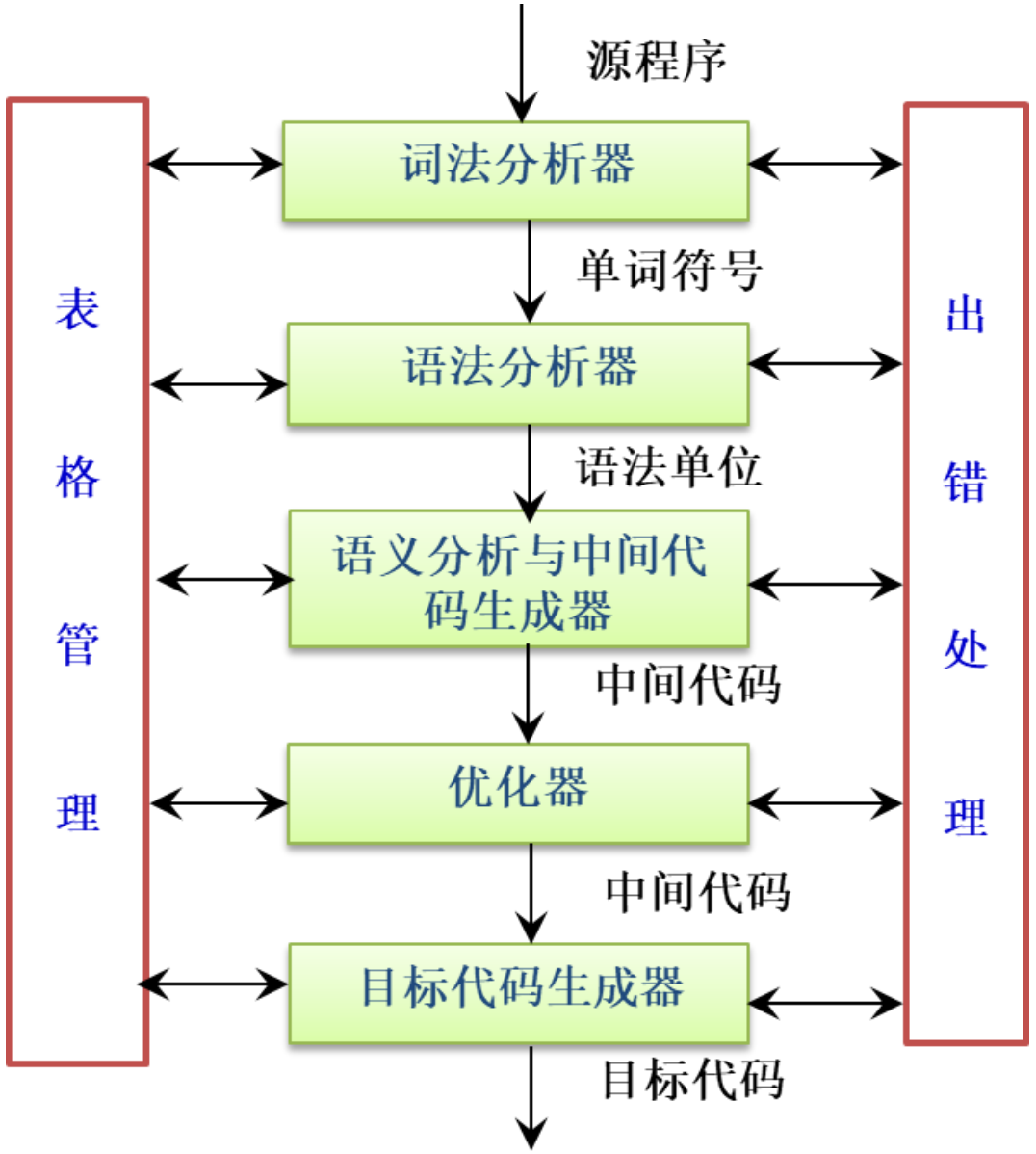
```
0001   01   00   00001111
0011   01   10   00000010
0100   01   00   00010011
```

编译程序总框

- 词法分析器（扫描器）：输入源程序，进行词法分析，输出单词符号。
- 语法分析器（分析器）：对单词符号串进行语法分析，识别出各类语法单位，判断输入串是否构成语法上正确的“程序”。
- 语义分析与中间代码产生器：按照语义规则对语法分析器归约出的语法单位进行语义分析并翻译成一定形式的中间代码。
- 优化器：对中间代码进行优化处理。
- 目标代码生成器：把中间代码翻译成目标程序。

编译程序总框

- 前端：源代码->中间代码
- 中端：机器无关优化
- 后端：中间代码->目标代码



表格与表格管理

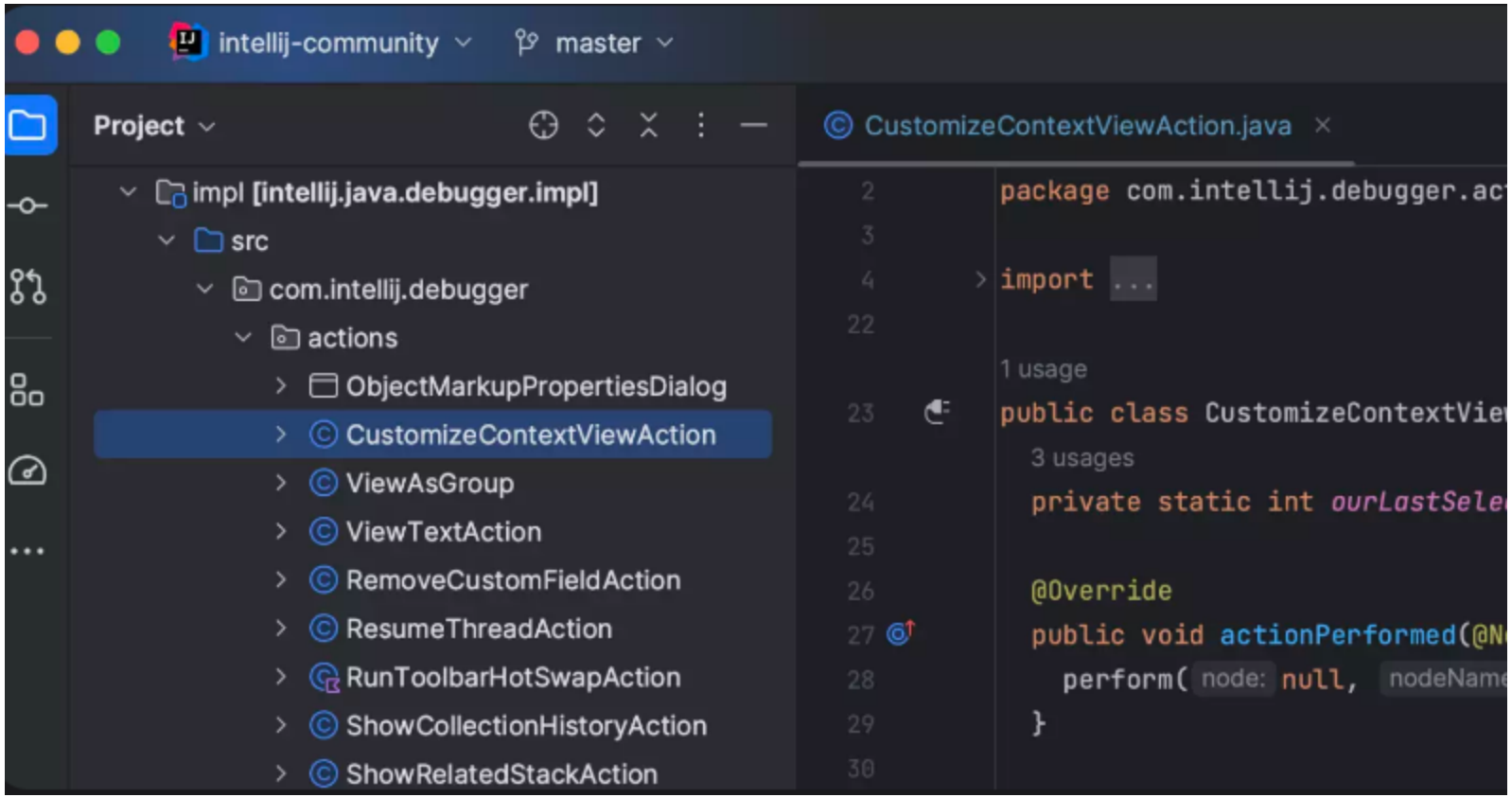
- 编译程序在工作过程中保持一系列的表格，以登记源程序的各类信息和编译各阶段的进展状况。
- 常见的表格：符号名表，常数表，标号表，入口名表，过程引用表。
  - 符号表：登记源程序中出现的各个名字以及名字的各种属性。
- 编译各阶段都涉及到构造、查找或更新有关的表格。
-

出错处理

- 如果源程序有错误，编译程序应设法发现错误，把有关错误信息报告给用户。这部分工作是由专门的一组程序（叫做出错处理程序）完成的。
- 源程序中的错误通常分语法错误和语义错误
  - 语法错误指源程序中不符合语法（或词法）规则的错误，可在词法分析或语法分析时检测出来。
    - 词法分析：“非法字符”等；语法分析：“括号不匹配”、“缺少；”等。
  - 语义错误指源程序中不符合语义规则的错误。一般在语义分析时检测出来，有的语义错误要在运行时才能检测出来。
    - 语义错误通常包括：说明错误、作用域错误、类型不一致等等。

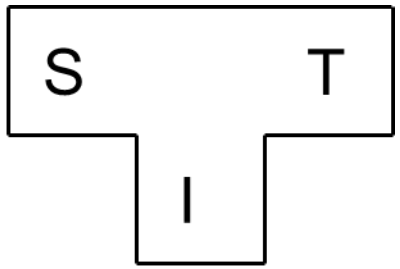
编译程序的产生

- 集成开发环境（IDE）
  - Intellij IDEA
  - Visual Studio
  - VSCode
  - Pycharm



编译程序的产生

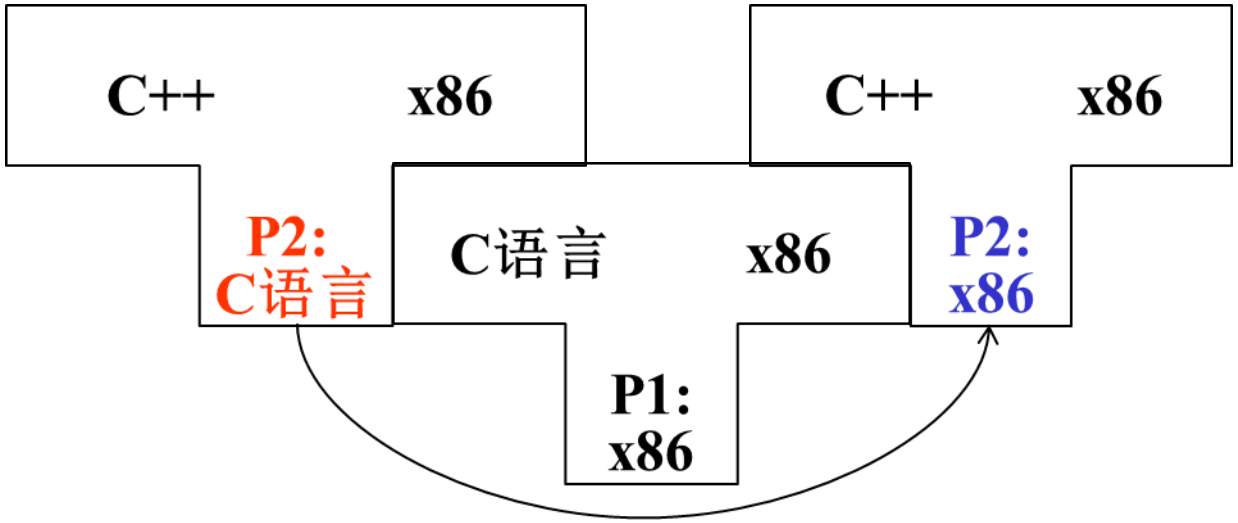
- 汇编语言和机器语言编写
- 高级语言编写
- 各有优劣，现代一般用高级语言



S 源程序 T 目标程序 I 实现语言
S：Python语言
T：机器语言
I：C语言
用C语言写的Python编译器

编译程序的产生

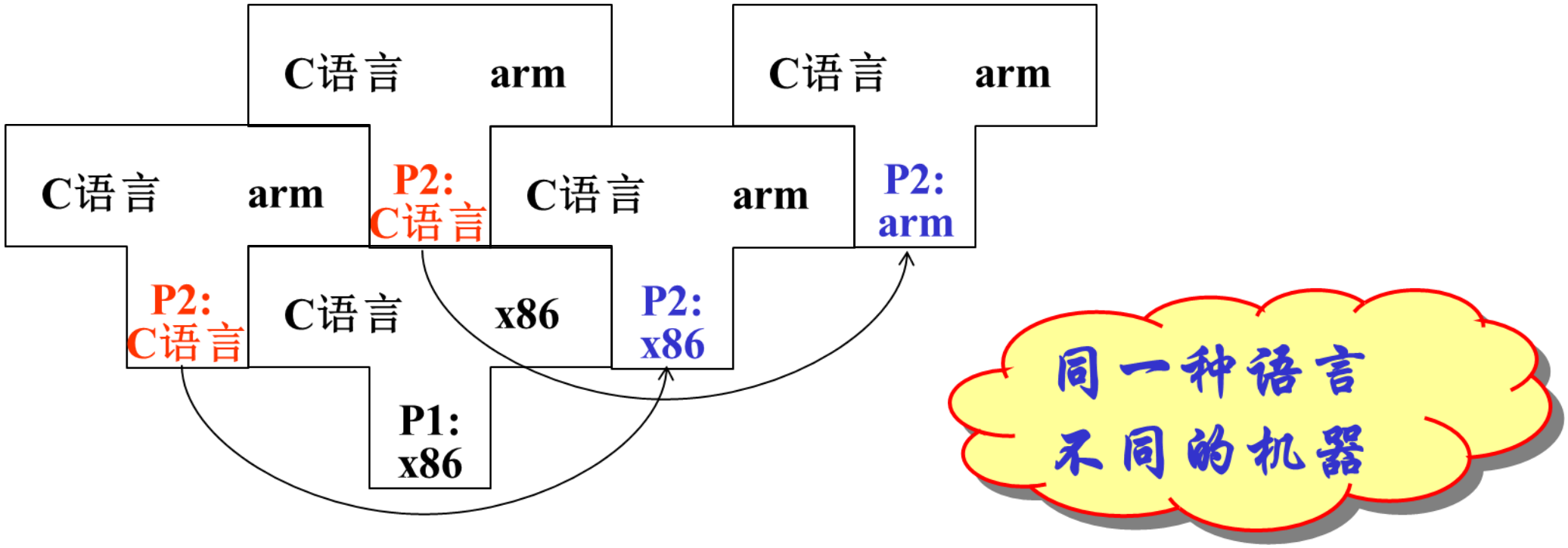
- 利用已有的某种语言编译程序实现另一语言的编译程序



同一台机器
不同的语言

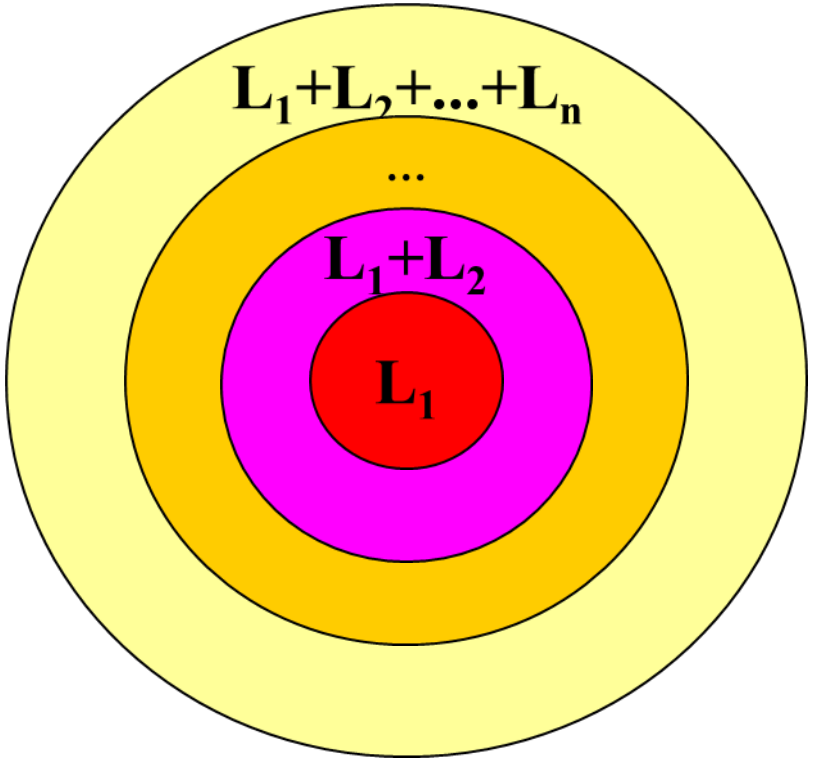
编译程序的产生

- 移植方法



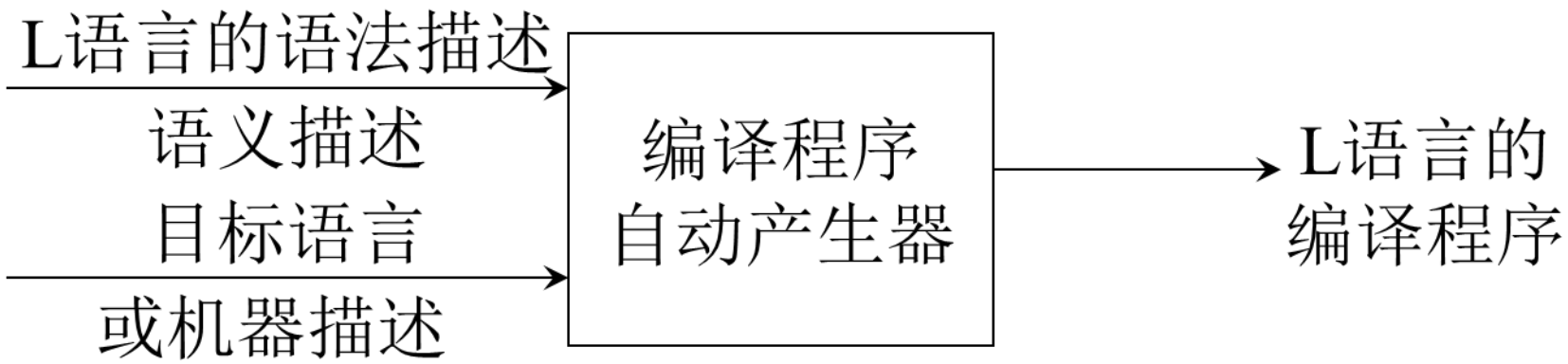
编译程序的产生

- 自展技术



编译程序的产生

- 使用工具自动化生成
  - LEX 词法分析程序产生器
  - YACC 语法分析程序产生器
  -



编程语言发展

- 早期计算机：硬编程
- 冯诺依曼架构：存储程序
- IBM 704/FORTRAN
- UNIX/C
- JAVA/C++/PYTHON
- LISP/PROLOG

函数/子函数

- 过程抽象
  - 函数名
  - 调用函数
  - 返回值
- 信息隐藏
  - 内部实现
  - 内部变量

函数/子函数

- IBM 704/FORTRAN
  - 汇编不支持函数调用
  - 固定地址调用

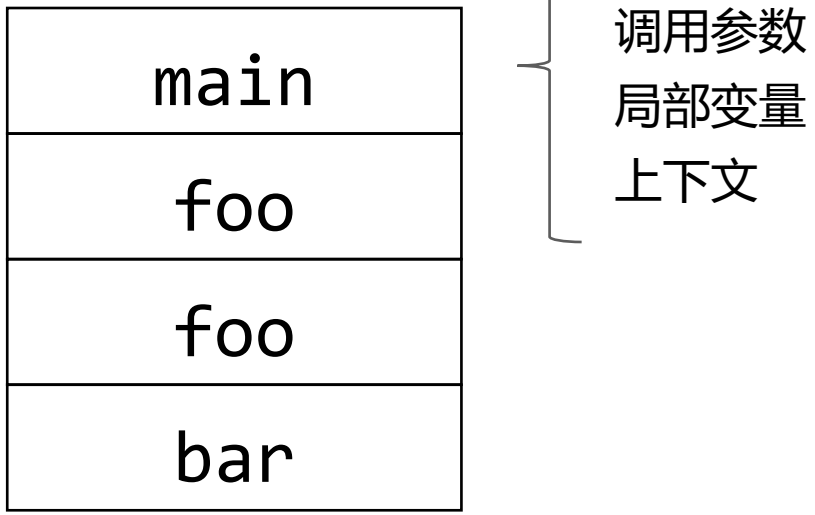
```
call foo(a)

200: xxx // foo
204: add b, a // b=b+a
...
280: jump r // get return addr

700: store 708 r
704: jump 200
```

递归调用不好处理

call stack



编译时多态-泛型

- 快速排序

```
void sort(int32_s[] s, size_t size) {
  ...
  if (s[i] < s[j]) {
    ...
  }
  ...
}
```

为什么数值类型有不同长度?

int32_s：4个字节

int64_s：8个字节

对应的汇编指令不一样

统一的函数实现，只是参数类型不一样

泛型

```
template <typename T>
void sort(T[] s, size_t size) {
  ...
  if (s[i] < s[j]) {
    ...
  }
  ...
}
```

编译时根据不同T生成不同代码

抽象数据类型

- 复数 complex
  - 实部 real
  - 虚部 image
  - 求模长 modulas
  - 求幅角 angle
  - 加法 add
  - 乘法 multiply

```
class complex {
public:
  double real;
  double image;
  double modulas();
  double angle();
  complex add(complex c);
  complex multiply(complex c);
};
```

数据与对应的操作相组合

实现隐藏

运行时多态-类与继承

- 1亿人的出生年月日，用一个数组存储，YYYYMMDD
- 统计各个月出生数量

导入数据库，执行sql查询：

```
select (extract month from birthday) as month, count(*)
  from people group by month
```

使用数组存储每个月人数：

```
int count[13] = {0};
for (i=0; i<n; i++) {
  int month = (birthday[i] / 100) % 100;
  count[month]++;
}
```

运行时多态-类与继承

- 1亿人的出生年月日，用一个数组存储，YYYYMMDD
- 统计每天出生数量

直接使用数组使用数组存储每个月人数：

```
count[birthday]++;
```

占用空间：10^8

存储空间严重冗余！可以不考虑大于100岁的人！

```
index = birthday % 1000000
```

占用空间 10^8->10^6

进一步优化！一年12个月，每个月最多31天！

```
year = birthday % 1000000 / 10000
month = birthday % 10000 / 100
day = birthday % 100
index = year * 12 * 31 +
  (month - 1) * 31 + (day - 1)
```

占用空间：10^6->37200

Hash函数

运行时多态-类与继承

- 哈希表
  - 键值对，每个键对应一个值
  - 根据键 访问/修改 值
  - 添加/删除 键
  - 实现方式要求：键->数字 映射

```
HashTable(T1 key, T2 value, Func hash);
```

是否可hash应该是T1类型的属性（封装）

```
class Hashable {
public:
  virtual int32_t hash() = 0;
};
```

```
HashTable(Hashable key, T value);
```

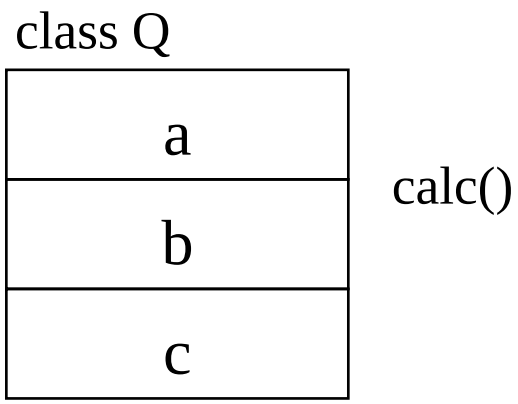
Hashable的每个子类中都存一下hash函数的指针（虚表）
调用hash函数的时候按指针来实际调用
key.hash()

运行时多态-类与继承

- 实现继承

```
class P {
public:
  int a;
  int b;
  int calc();
};

class Q: public P {
public:
  int c;
}
```



组合

```
public P {
public:
  Q q;
  int c;
}
```

面向对象

- C++
  - 多继承
  - 构造函数
  - 析构函数
- Java
  - 实现继承-单继承 extends
  - 接口 interface
  - 接口继承 implements

脚本语言-python

- 所有成员函数和变量都用一张表来存
- 函数是对象

```
class Complex:
    def __init__(self):
        self.real = 0
        self.image = 0

    def modulas(self):
        m = self.real * self.real + self.image * self.image
        return m

c = Complex()
c.real = 1
c.image = 2
print(c.modulas)
print(c.modulas())
```

```
<bound method Complex.modulas of <__main__.Complex object at (
5
```

```
dir(c)
```

```
['__class__',
 '__delattr__',
 '__weakref__',
 'image',
 'modulas',
 'real']
```

```
from types import MethodType

def modulas(self):
    m = self.real + self.image
    return m

c.modulas = MethodType(modulas, c)
c.real = 1
c.image = 2
c.modulas()
```

```
3
```

并发

- 资源竞争
- 死锁
- 信号量
- sychronized
- happens before

异常

- 错误码
  - 返回值表示程序运行情况
- 异常
  - 单独异常处理流
- Java
  - Checked Exception