北京大学计算机学院

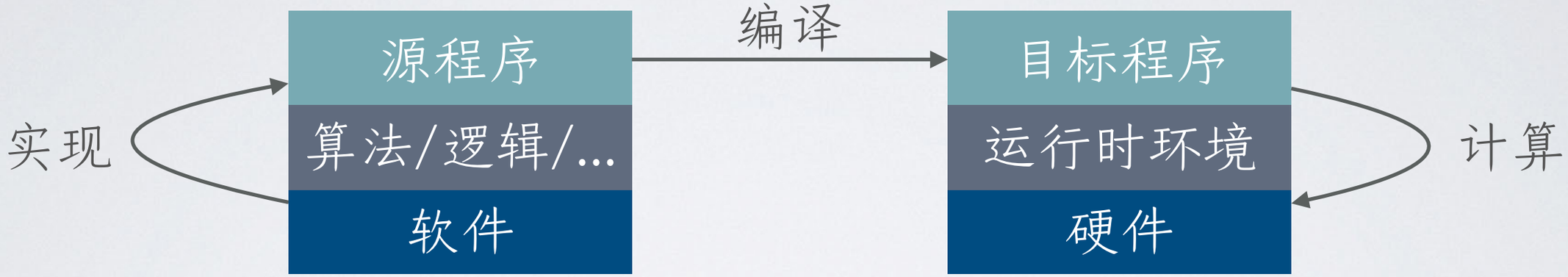
2026 年春季学期　　《编译原理(实验班)》

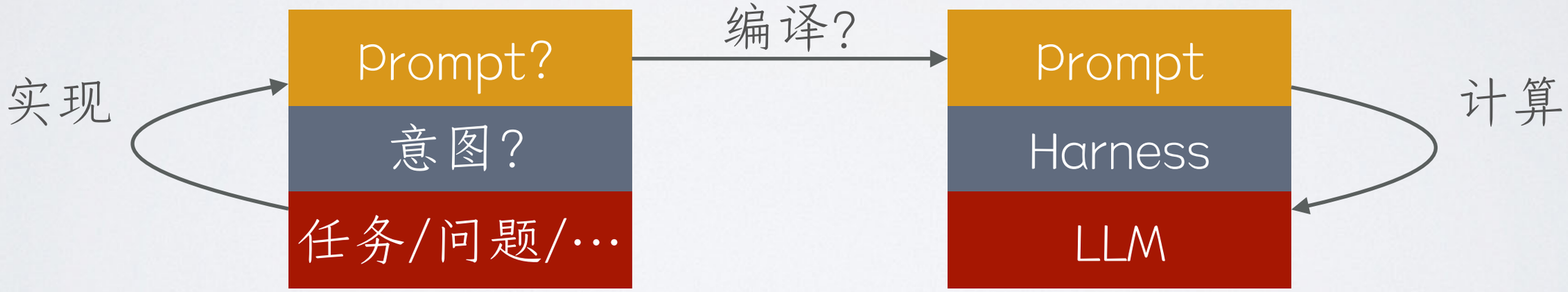
《编译原理（实验班）》内容回顾

Course Review

编译是什么？

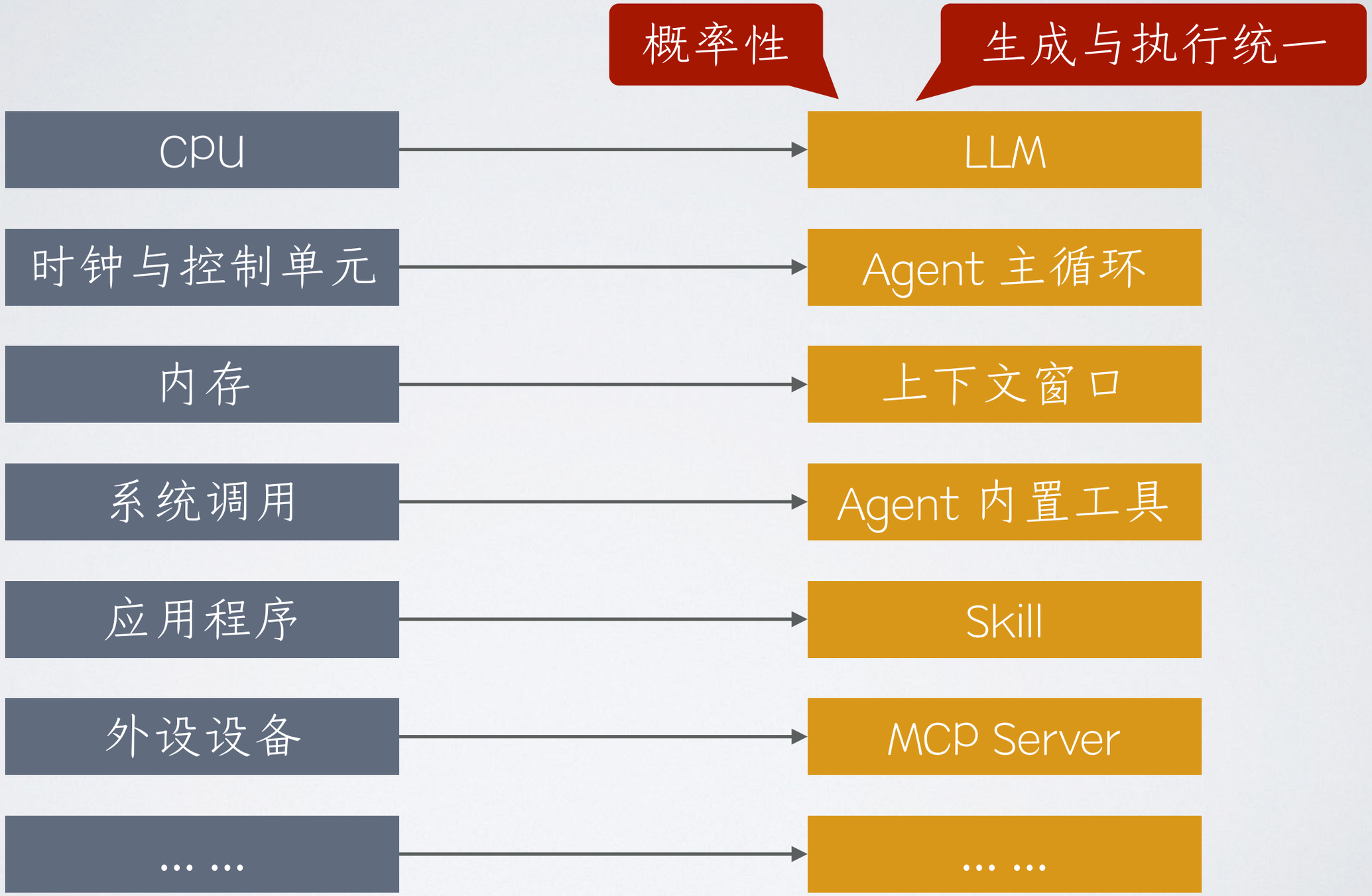
- **编译(Compiling)**就是把**高级语言程序**翻译成与其等价的**目标语言程序**的过程





- 编译的**科学**：如何**正确**、**高效**地实现翻译？
- 编译的**艺术**：如何**设计**过程中所需的各种中间表示？

一坨不成熟的类比


概率性
生成与执行统一
CPU
LLM
时钟与控制单元
Agent 主循环
内存
上下文窗口
系统调用
Agent 内置工具
应用程序
Skill
外设设备
MCP Server
…… ……

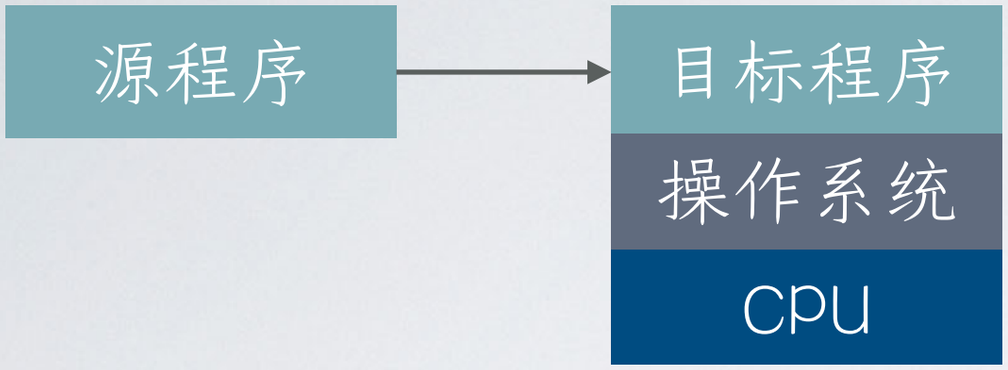
Harness ≈ 操作系统

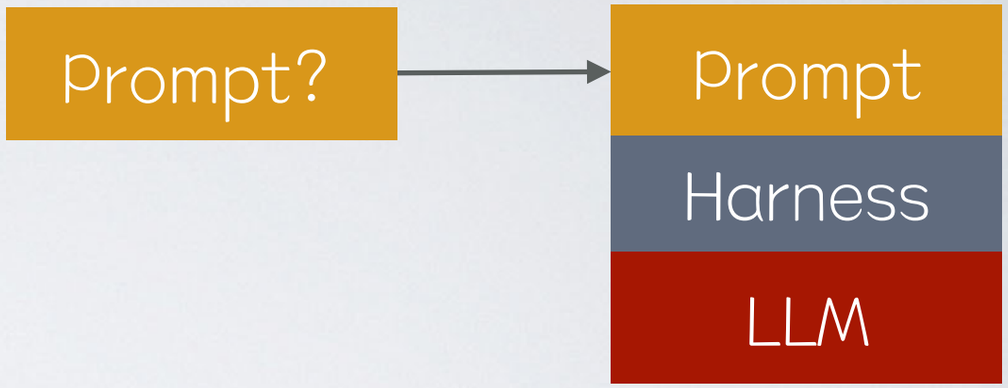
- 回顾：操作系统就是目标程序的运行时环境

- **问：Harness 中有哪些与操作系统神似的概念？**

进程管理	会话与 Agent 生命周期
内存管理	上下文压缩与窗口管理
设备驱动	MCP 注册与动态发现
进程间通信	多 Agent 协作
权限与安全	沙盒隔离

什么东西不见了？





- **编程语言不见了！编译器不见了！**

- **我们手写 Prompt 类比直接手写汇编？**
  - 由于 LLM 是「生成与执行统一」的，Prompt 可以写得很抽象
- 强行类比，Prompt 是否有「High Level」和「Low Level」之分？
- 如果有的话，**编译**代表啥？**编译器**是啥？

在这之前，先来看看汇编

- 不同的 LLM ≈ 不同的 CPU
- 不同的 Harness ≈ 不同的操作系统
- **但是 Skill 为啥能通用？**
  - 回忆：Skill 是「以自然语言为汇编直接写的应用程序」
- **通用不是必然的**
  - 思想实验：假设 LLM A 只懂英文，LLM B 只懂中文，还能通用吗？
- 再次类比：Apple Rosetta，把 Intel 应用转译到 Arm 架构上
- SkVM: Revisiting Language VM for Skills across Heterogenous LLMs and Harnesses

SkVM ≈ JVM

- 虽然 Skill 确实通用，但是效果也确实受 LLM + Harness 影响

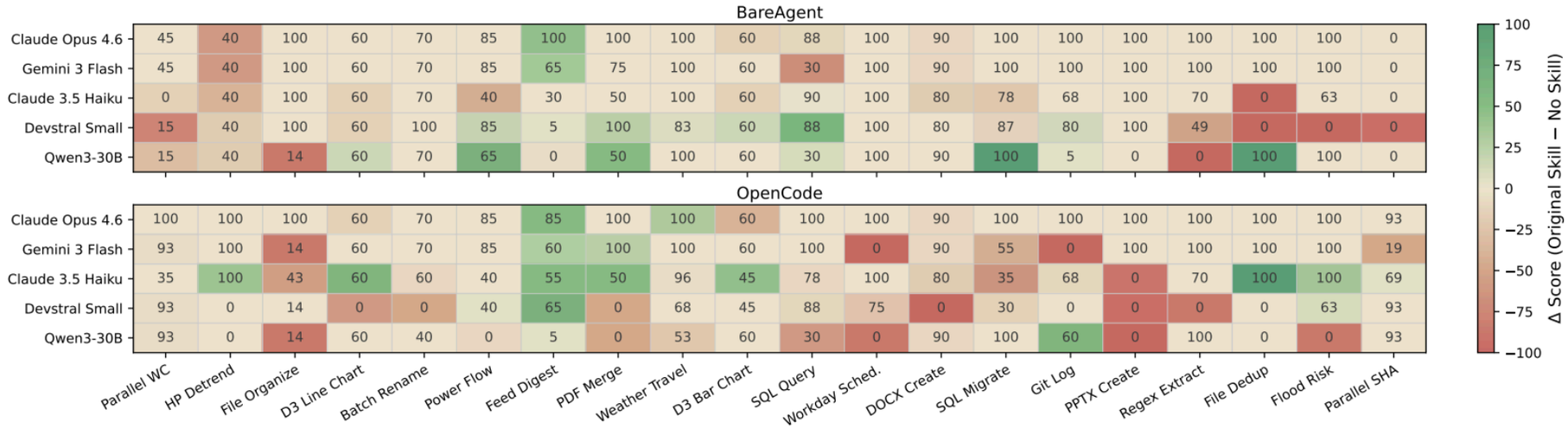
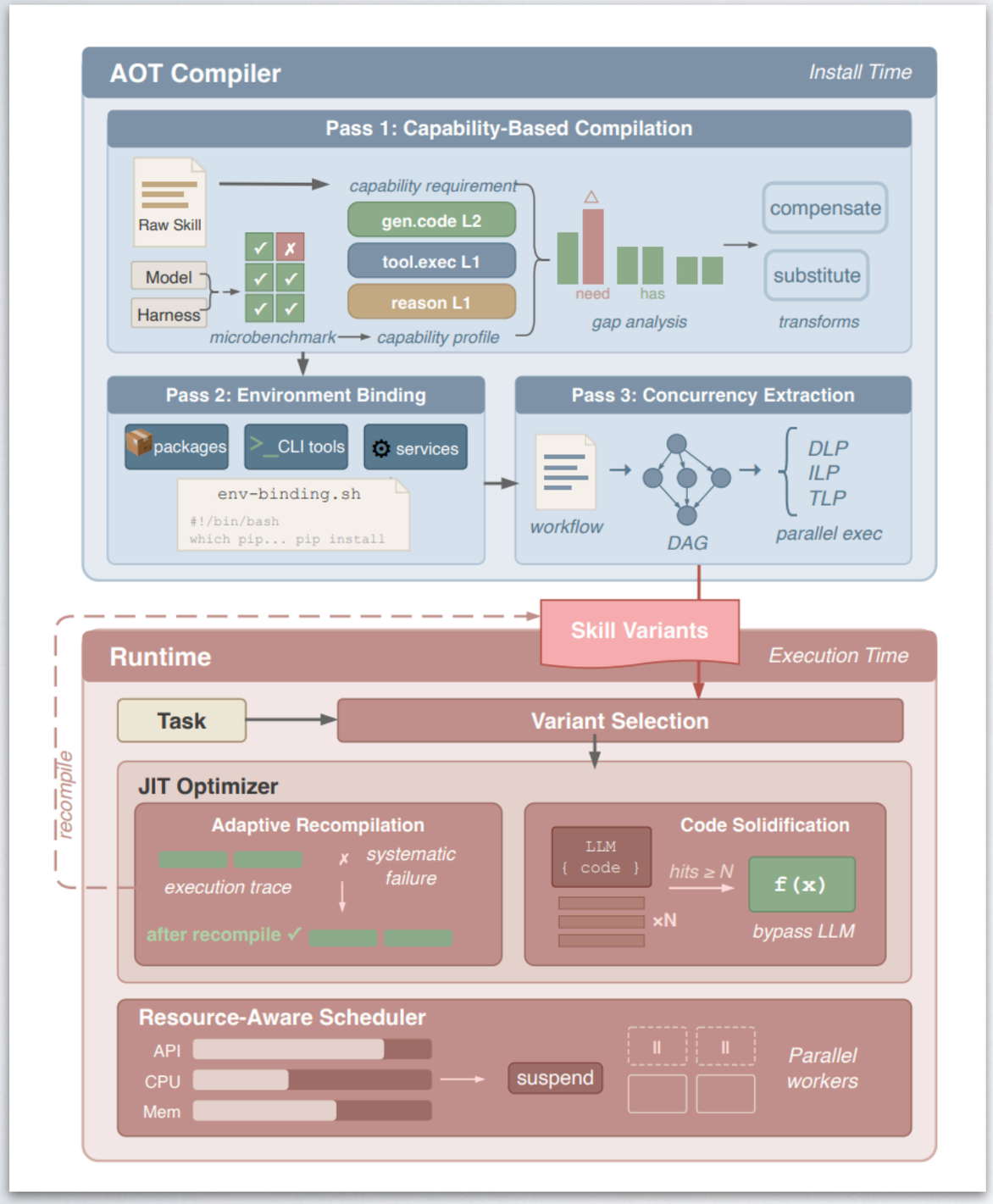


Figure 2. Skill performance across models and harnesses. Columns are tasks, rows are models, and each subplot corresponds to one harness. Cell colors indicate score deltas relative to the no-skill baseline, and cell labels show absolute task scores. Both model identity and harness choice significantly affect outcomes.

- **问：JVM 做了什么？**
  - AOT(Ahead-of-Time)：编译时翻译到**平台无关**的字节码
  - JIT(Just-in-Time)：运行时为**热点代码**实时生成机器码
  - **可以怎么类比？**

SkVM ≈ JVM



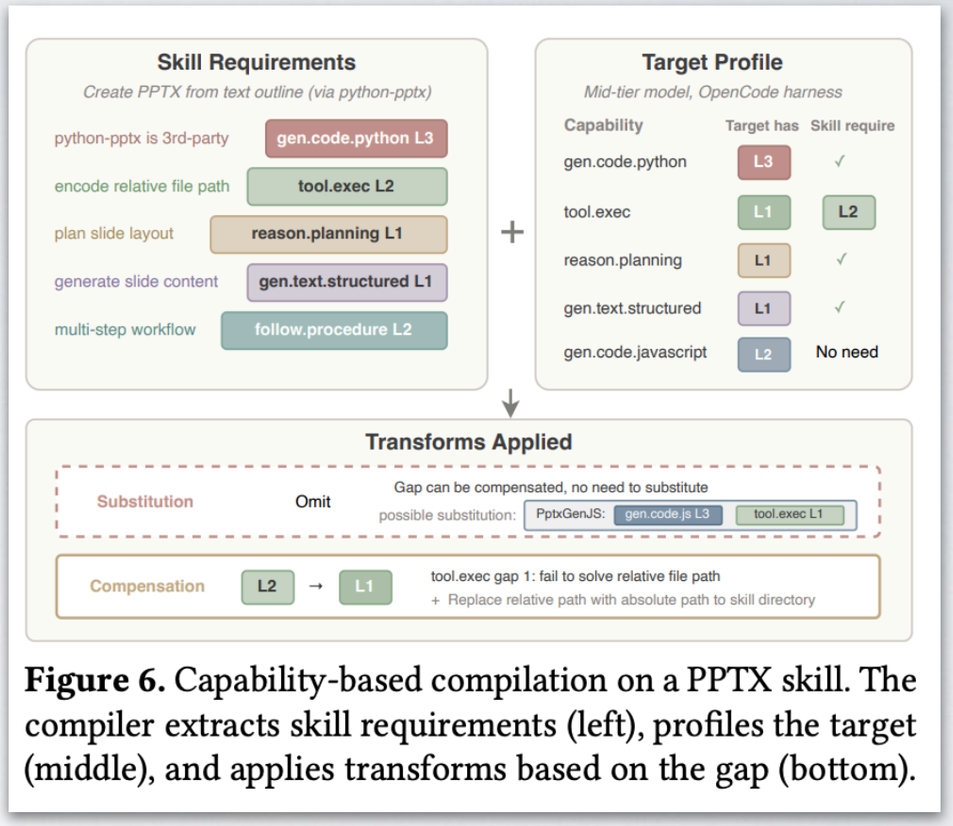
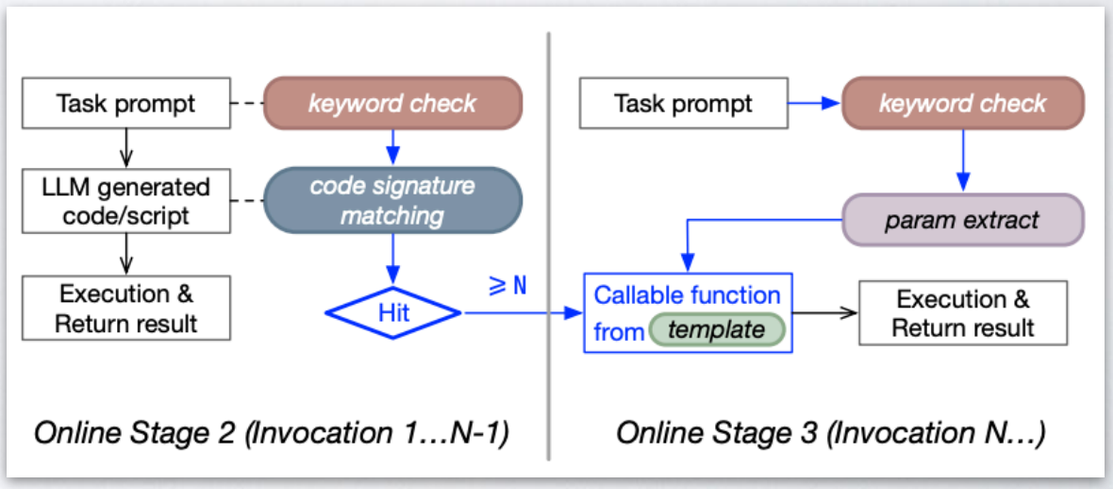
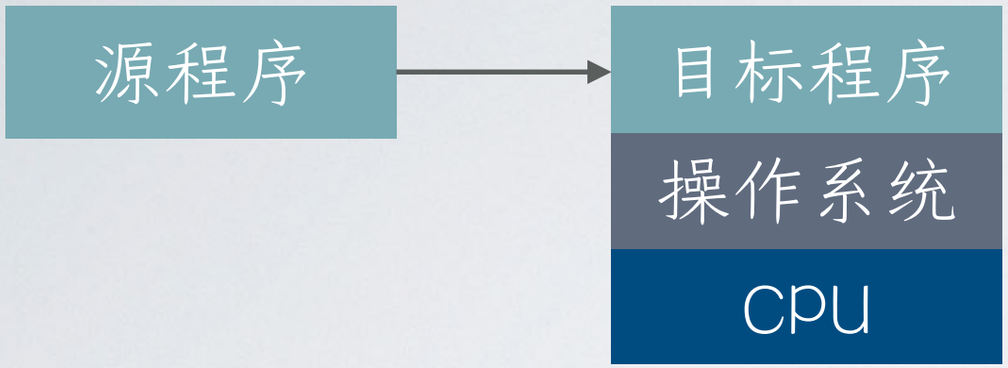
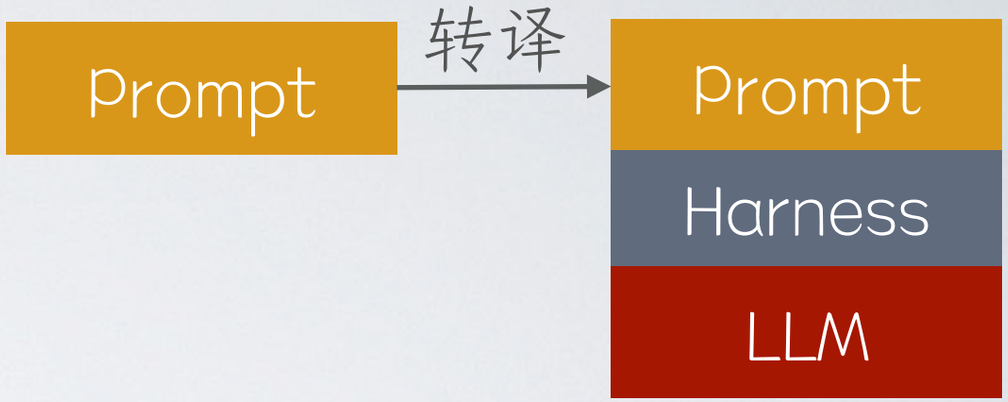


Figure 6. Capability-based compilation on a PPTX skill. The compiler extracts skill requirements (left), profiles the target (middle), and applies transforms based on the gap (bottom).



什么东西不见了？



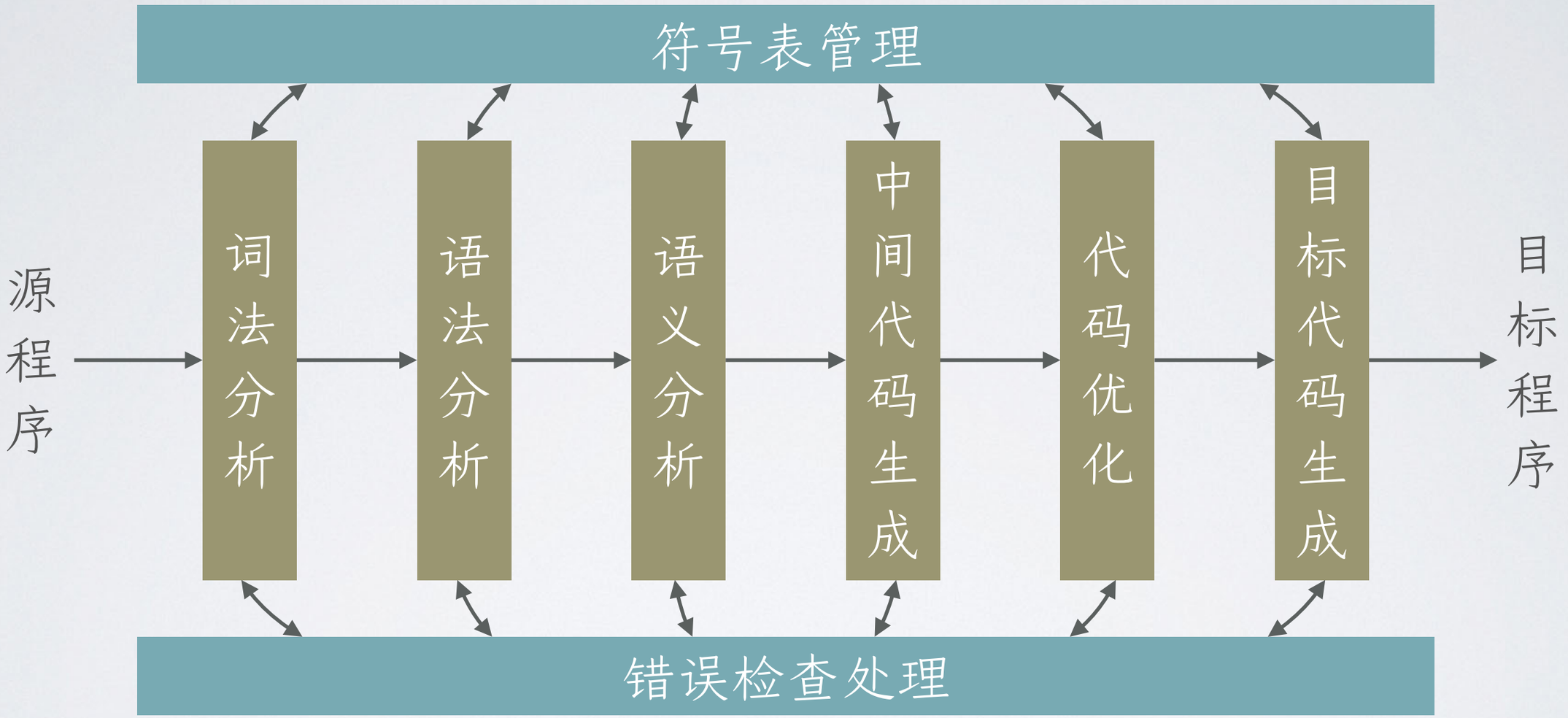


- SkVM 是一套「编译器」技术
  - 在自然语言 Prompt 层面的「源源编译」
  - 还有很多潜力：并行化、技能内联、工具固化、死上下文消除……

- **问：还有可能从「汇编语言」走向「高级语言」吗？**
- **会是自然语言、形式语言，还是两者的混合？**

编译器是什么？

- **编译器(Compiler)**：实现源程序编译到目标程序的**软件**



- 编译的**工程**：如何**架构**编译器使得开发更加可控？

词法分析：源代码 -> Token 流

- 词法分析的工作流程(以 Lex 为例)
  - 正则表达式 => NFA => DFA => 最小化 DFA
- **关键技术**
  - 正则表达式的表示
    - 正则语言
  - 正则表达式转换为 DFA
    - 求导法

例题

- 下列哪个正则表达式与 $(0^*10^*1)^*0^*10^*$ 表示的语言相同：

  (A) $0^*1(0 \mid 10^*1)^*$

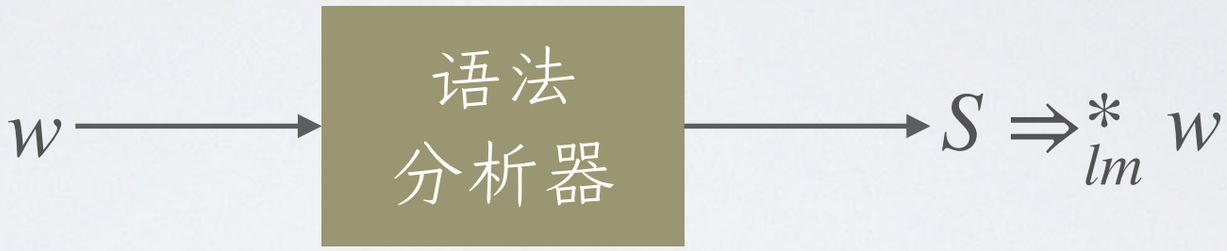
  (B) $(0 \mid 10^*1)^*$

  (C) $1^*0(1 \mid 01^*0)^*$

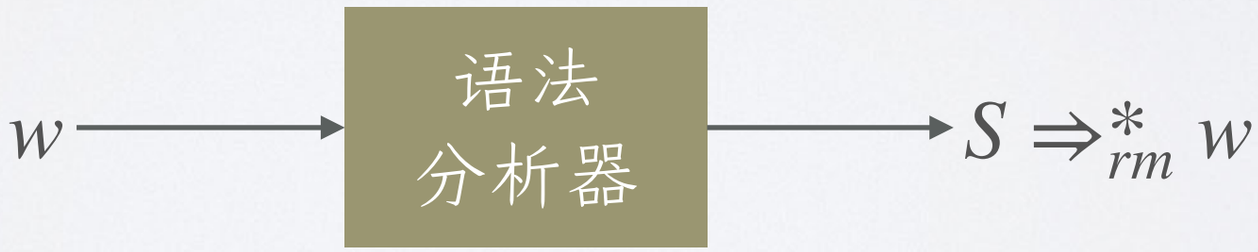
  (D) $(1 \mid 01^*0)^*$

语法分析：Token 流 -> 分析树

- 上下文无关文法
- 自顶向下分析



- 自底向上分析



自顶向下分析

- 自顶向下分析算法的基本思想：
  - 对于文法 $G[S]$：若 $S \Rightarrow^{+} w$ 则 $w \in L(G[S])$，否则 $w \notin L(G[S])$
- **主要问题：**
  - 左递归问题
  - 回溯问题
- **主要方法：**
  - 递归下降分析法
  - LL 分析法
  - PEG 和 Packrat 算法
  - Parser Combinators

自底向上分析

- 自底向上分析算法的基本思想：
  - 对于文法 $G[S]$：若 $S \Leftarrow^{+} w$ 则 $w \in L(G[S])$，否则 $w \notin L(G[S])$
- **主要问题：**
  - 分析栈状态的识别问题
- **主要方法：**
  - 移进-归约分析法
  - LR 分析法

语法分析的关键技术

- **文法的构造与转换**
  - 消除左递归
  - 提取左公因子
- **自顶向下的语法分析**
  - 递归下降分析
  - LL(1) 分析：FIRST、FOLLOW 集合，LL(1) 分析表
- **自底向上的语法分析**
  - 移进-归约分析
  - LR 分析：LR(0)，SLR(1)，LR(1)，LALR(1)
  - LR 项和 LR 项集的构造
    - 识别分析栈的 DFA

例题

- 下列哪些是上下文无关语言：
  - (A) 所有含偶数个 a 和偶数个 b 的由小写字母组成的串
  - (B) 所有形如 ca^nb^m 的串，其中 $n > 2m$
  - (C) 所有能通过 C 语言编译器编译的串
  - (D) 所有由 1、2、3、4 组成且所表示的十进制数大于 1234 的串

例题

- 若上下文无关文法 G 定义的语言是一个有限集合，则下列哪些性质 G 必然满足：

(A) G 无二义性

(B) G 是正则文法

(C) 存在无二义性上下文无关文法 G'，使得 $L(G') = L(G)$

(D) 存在正则文法 G'，使得 $L(G') = L(G)$

例题

- 考虑如下文法，回答下列问题：

$$S \to AB$$

$$A \to B + A \mid \mathrm{a}$$

$$B \to C * B \mid \mathrm{b}$$

$$C \to (A) \mid \varepsilon$$

(A) C 的 FIRST 集合是什么？

(B) A 的 FOLLOW 集合是什么？

(C) 这是 LL(1) 文法吗？

(D) 文法的预测分析表中，输入符号为 * 时 A 对应的表项是什么？

语义分析：分析树 -> 「语义」

- **语法制导定义(SDD)，也称为属性文法**
  - 综合属性
  - 继承属性
  - 属性求值

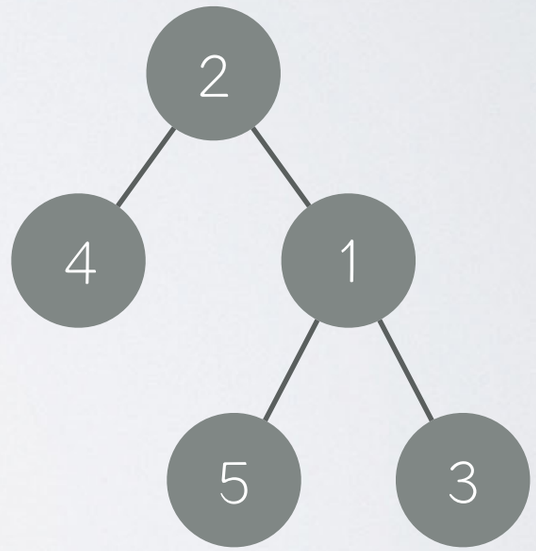
- **语法制导的翻译方案(SDT)**
  - 自顶向下翻译
    - 消除翻译方案中的左递归
  - 自底向上翻译
    - 消除嵌入在产生式中间的动作

- **解释执行、类型检查、中间表示生成**

例题

- 考虑上下文无关文法 $T \rightarrow$ **uint** | **uint** (T) (T), 其中 **uint** 是一个表示非负整数的终结符号。该文法可以用来描述二叉树，且每个结点有一个非负整数作为这个结点的权重。例如，2(4)(1(5)(3)) 表示了右下图中的树。定义路径的权重是路径上所有结点的权重之和。

- 设计只包含非负整数属性的 S 型 SDD(只包含综合属性)，计算树上权重最大的路径的值，并保存在综合属性 *maxlen* 中。**uint** 的数值可以通过 **uint**.*val* 获得。例如，右图中根结点的 *maxlen* 值为 12 (4+2+1+5)。



例题

- 以下是上下文无关文法 $G[A]$ 的自底向上的 SDT，其中 a、b、c 为终结符，A、B 为非终结符：

  $A \to \mathrm{a}\ B$ { print "0" }

  $A \to \mathrm{c}$ { print "1" }

  $B \to A\ \mathrm{b}$ { print "2" }

- 若输入串 w 能被 G 识别，则 SDT 将打印出输出串 w'。判断所有可能的 w' 构成的语言 L' 是否为正则语言或上下文无关语言，并对应给出正则表达式或上下文无关文法。

中间代码生成：分析树 -> 三地址代码

- **中间代码的常用形式**
  - 抽象语法树、有向无环图、**三地址代码**、静态单赋值、LLVM、MLIR
  - 函数式语言：KNF、ANF、CPS、闭包形式
- **通过语法制导翻译实现中间代码生成**
  - 为每种语法结构构造中间代码的表示形式
  - 构造生成上述中间代码的 SDD/SDT
    - 表达式和赋值语句
    - 类型和声明、数组的处理、类型检查
    - **布尔表达式和控制流语句**
      - if, while, do-while, for, switch
      - 回填

例题

- 考虑 Pascal 语言中的 FOR 循环语句：
  - $S \rightarrow$ FOR **id** := E_1 TO E_2 DO S_1
  - 其语义为：先计算初始值 E_1 和结束值 E_2，再循环执行 S_1，循环变量从初始值变化到结束值，每次执行完 S_1 后对循环变量加一

- 为如下中间代码形式设计 SDD（继承属性 $S.next$ 记录紧跟在 S 之后的语句标号）

```
        E1.code
        id.addr = E1.addr
        E2.code
(loop)  if id.addr > E2.addr goto (next)
        S1.code
        id.addr = id.addr + 1
        goto (loop)
(next)
```

例题

- 考虑 Pascal 语言中的 FOR 循环语句：
  - $S \rightarrow$ FOR **id** := E_1 TO E_2 DO S_1
  - 其语义为：先计算初始值 E_1 和结束值 E_2，再循环执行 S_1，循环变量从初始值变化到结束值，每次执行完 S_1 是对循环变量加一
- 为上一页的 SDD 设计自顶向下的 SDT
- 把上面的 SDT 改造成边扫描边生成代码
- 继续改造，使用回填技术生成代码
  - 综合属性 $S.nextlist$ 中的跳转语句的目标是紧接着 S 的位置
  - 变量 *nextinstr* 保存了生成过程中紧跟着的下一条语句的标号
- 继续改造成自底向上的 SDT
  - 如果要求显式操作语法分析栈呢？

机器无关优化：在三地址代码上进行

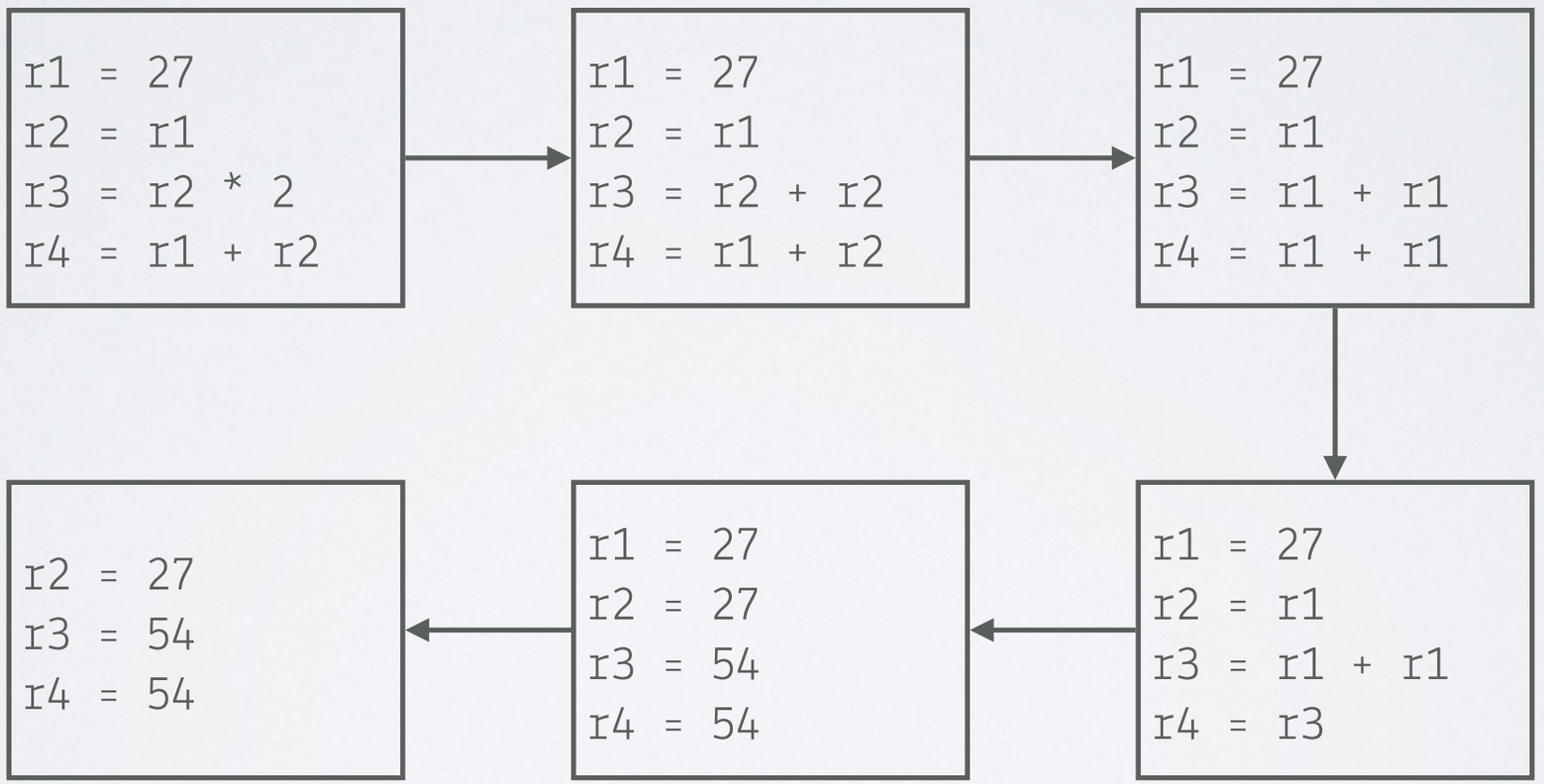
- 了解编译优化的基本概念
- **局部优化**
  - **基本块的划分、流图的构造、循环的识别**
  - 基本块内的活跃变量分析
  - 基本块内的优化
- **全局优化**
  - 常用方法：常量折叠、公共子表达式消除、复写传播、死代码消除、循环优化(代码外提、归纳变量消除、强度消减)
  - **简单的数据流分析**
    - 活跃变量、可达定值、可用表达式

例题

- 代码优化(不局限于编译优化)的目的包括以下哪些内容:
  - (A) 提高运行速度
  - (B) 优化能耗
  - (C) 减少资源消耗
  - (D) 提高开发速度
  - (E) 提高代码质量和可维护性
  - (F) 更改代码的功能

例题

- 下列代码变换中所用的优化技术是什么？
  - 可选项：强度消减、复写传播、常量折叠/传播、死代码消除、公共子表达式消除



例题

- 考虑右边的三地址代码序列，其中 ret s 语句表示从函数中返回变量 s
- 把这段代码序列划分为基本块
- 为这段代码构造流图，并额外添加入口结点 ENTRY 和出口结点 EXIT
- 找出流图中的所有循环
- 在流图上进行可达定值分析和活跃变量分析
- 给出流图中可以进行的优化
  - 在进行一些优化后，是否会产生新的优化机会？

```
1)  s = 0
2)  i = 0
3)  t1 = s % 19
4)  if t1 == 1 goto (19)
5)  j = 0
6)  if i < 50 goto (8)
7)  ret s
8)  if j >= 100 goto (17)
9)  if i == j goto (15)
10) t2 = 100 * i
11) t3 = t2 + j
12) t4 = 4 * t3
13) t5 = a[t4]
14) s = s + t5
15) j = j + 1
16) goto (8)
17) i = i + 1
18) goto (3)
19) ret s
```

运行时环境：为生成目标代码做准备

- **存储分配策略**
  - 静态分配
  - 栈式分配
  - 堆式分配

- **栈式存储管理**
  - **活动记录、控制链、访问链、显示表**

- **堆存储管理**
  - 垃圾回收的基本原理和方法
    - 引用计数、标记-清扫

例题

- 考虑右下图中两个函数 f、g 在运行过程中的栈空间的内容片段，判断下列说法是否正确：

(A) (4)(6) 由 f 写入值

(B) (1)(2)(3)(4)(5) 属于 f 的活动记录

(C) (7)(8) 由 g 写入值

(D) (5) 由 g 在函数返回时由 g 写入值

```
void f() { int f1, f2; … g(5); … }
int g(int g1) { int g2; float g3; … }
```

例题

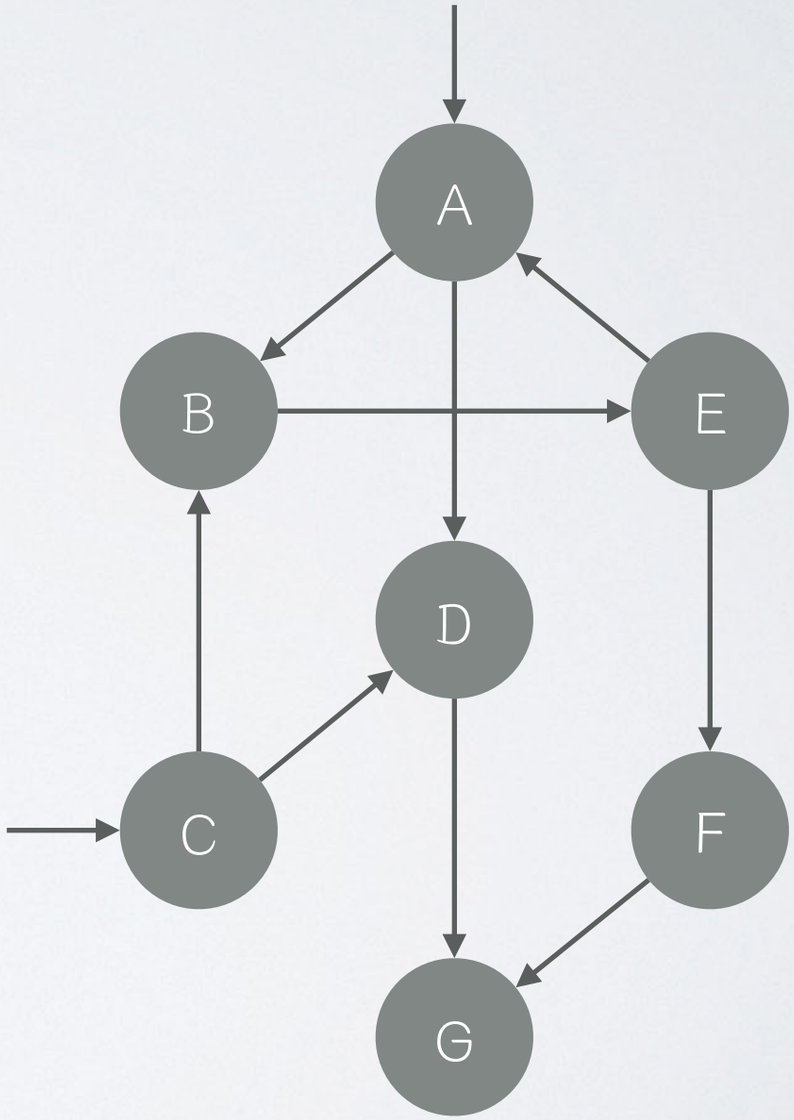
- 使用引用计数进行带垃圾回收的堆存储管理，对于右下的指向关系图，删除哪些边，会导致对象 G 被回收：

(A) D -> G, B -> E

(B) C -> B, C -> D

(C) D -> G, C -> B

(D) E -> F, A -> D



例题

- 考虑如下嵌套定义的过程：

```
procedure f;
    procedure f1;
        procedure f11;
            procedure f111;
        procedure f12;
    procedure f2;
```

- f 可以调用 f111 吗？反过来，f111 可以调用 f 吗？
- 考虑过程调用序列 f->f2->f->f1->f12->f11->f111->f2，画出栈中的过程活动记录，只需要标明过程名和访问链

目标代码生成：三地址代码 -> 目标代码

- **目标代码生成的基本任务**
  - 指令选择
  - **寄存器分配和指派**
    - 局部寄存器分配：寄存器描述符、地址描述符
    - 全局寄存器分配：冲突图构造、图着色算法、溢出
- **简单优化**
  - 窥孔优化

例题

- 考虑右边的三地址代码，为其生成目标代码，可用的指令为 LD R, x、LD R_1, R_2、ST x, R、ADD R_1, R_2, R_3、SUB R_1, R_2, R_3
- 假设可以使用任意多个寄存器，基本块入口处所有寄存器都没有存储有效值，而生成代码要保证出口处所有变量的内存位置都有它们的最新值
- 给出出口处的寄存器描述符和地址描述符
- 不额外生成指令的前提下，至少需要几个寄存器？

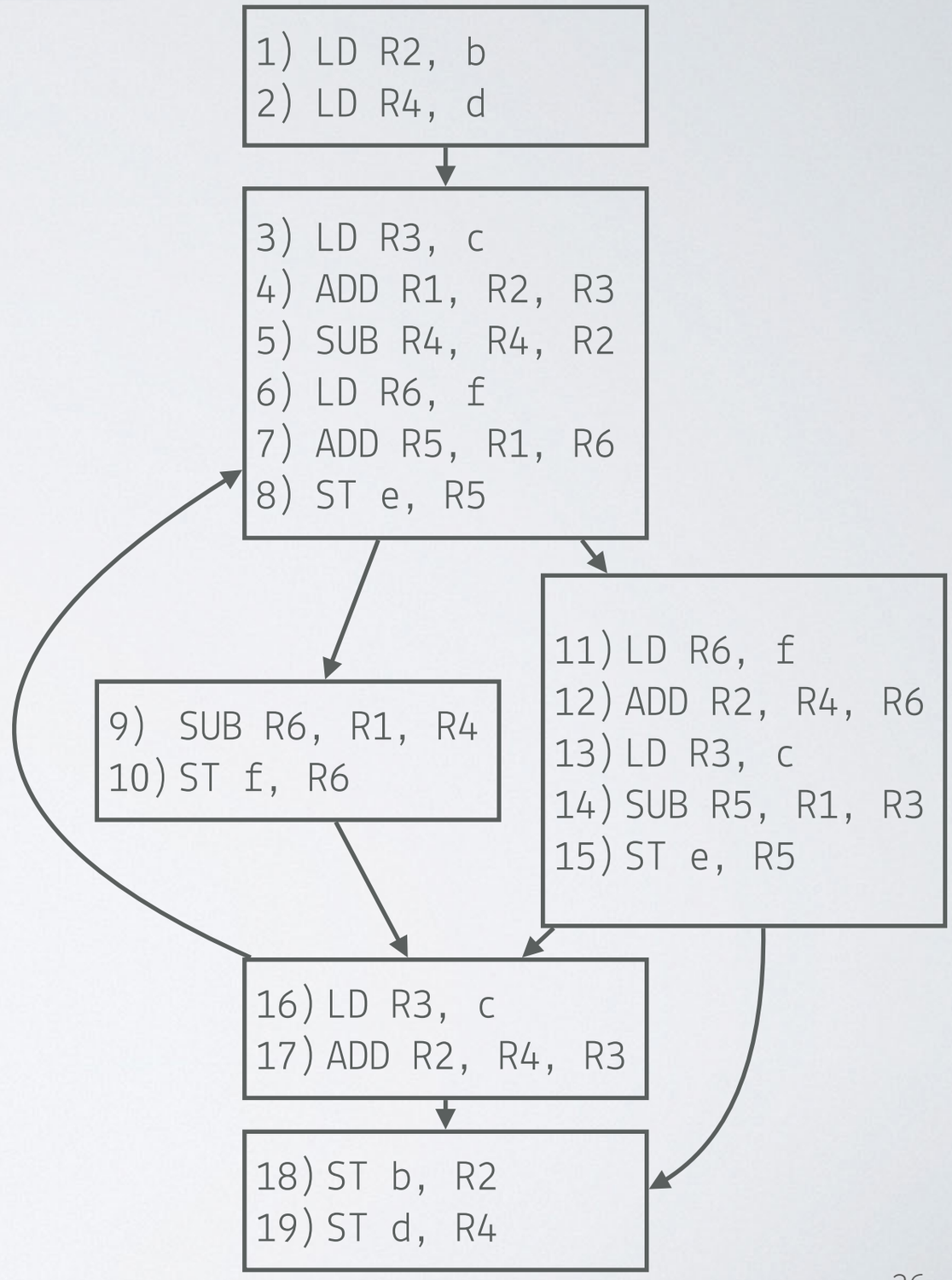
```
a = b + c
d = d - b
e = a + f
f = a - d
b = d + c
```

R1	R2	R3	R4	R5	R6

a	b	c	d	e	f

例题

- 考虑右边的流图(ENTRY 和 EXIT 画不下了)
- 给出各寄存器的活跃范围，用数字表示指令标号，l 表示在第 l 条指令后活跃
- 画出寄存器冲突图
- 假设有 4 个物理寄存器，采用图着色法进行寄存器分配
- 假设有 3 个物理寄存器，给出分配和溢出方案，并尽可能减少溢出代价(假设 load 和 store 的开销相同，循环执行 10 次)



期末考试形式

- **以大题为形式，分为必答题和选答题**

- **必答题：**
  - 自底向上语法分析：LR、SLR、LALR 分析(参考作业的形式)
  - 语法制导的翻译方案：三地址代码生成(回填思想？)
  - 运行时环境：非局部数据访问、垃圾回收
  - 代码优化：公共子表达式提取、死代码消除……
  - ？？？

- **选答题：**
  - 静态单赋值(Koopa IR)的生成、优化
  - 函数式语言的编译
  - 类型系统和类型检查
  - ？？？

去年的期末考试结构

- **【必答】语法分析(20分)**
  - 计算 FOLLOW 集合
  - 补全 LR(0) 自动机
  - 补全 SLR(1) 的 ACTION 和 GOTO 表
- **【必答】中间代码生成(20分)**
  - 补全 SDT
- **【必答】简答题(20分)**
  - 嵌套函数、静态作用域
  - 自举
  - 对编译课的建议

去年的期末考试结构

- **【选做】静态单赋值(20分)**
  - 描述 SSA
  - 转换 CFG、转换 SSA
  - 转换为非 SSA 形式的三地址代码
- **【选做】运行时环境(20分)**
  - 尾递归优化
  - 改写函数为尾递归版本
  - 说明尾递归和循环的关系
- **【选做】函数式语言(20分，作业中做过了)**
- **【选做】类型系统(20分，作业中做过了)**