



# 《编译原理》课程简介

---

Course Introduction



# 基本信息

- 课程网页
  - ❖ 北京大学教学网 (<https://course.pku.edu.cn/>)
- 主讲教师：
  - 张路、刘先华、王迪
  - ❖ Office：燕园大厦 520 室
  - ❖ Tel：13882266228
  - ❖ Email：[wangdi95@pku.edu.cn](mailto:wangdi95@pku.edu.cn)
- 助教
  - ❖ 李天驰、张钊
- 交流：微信群



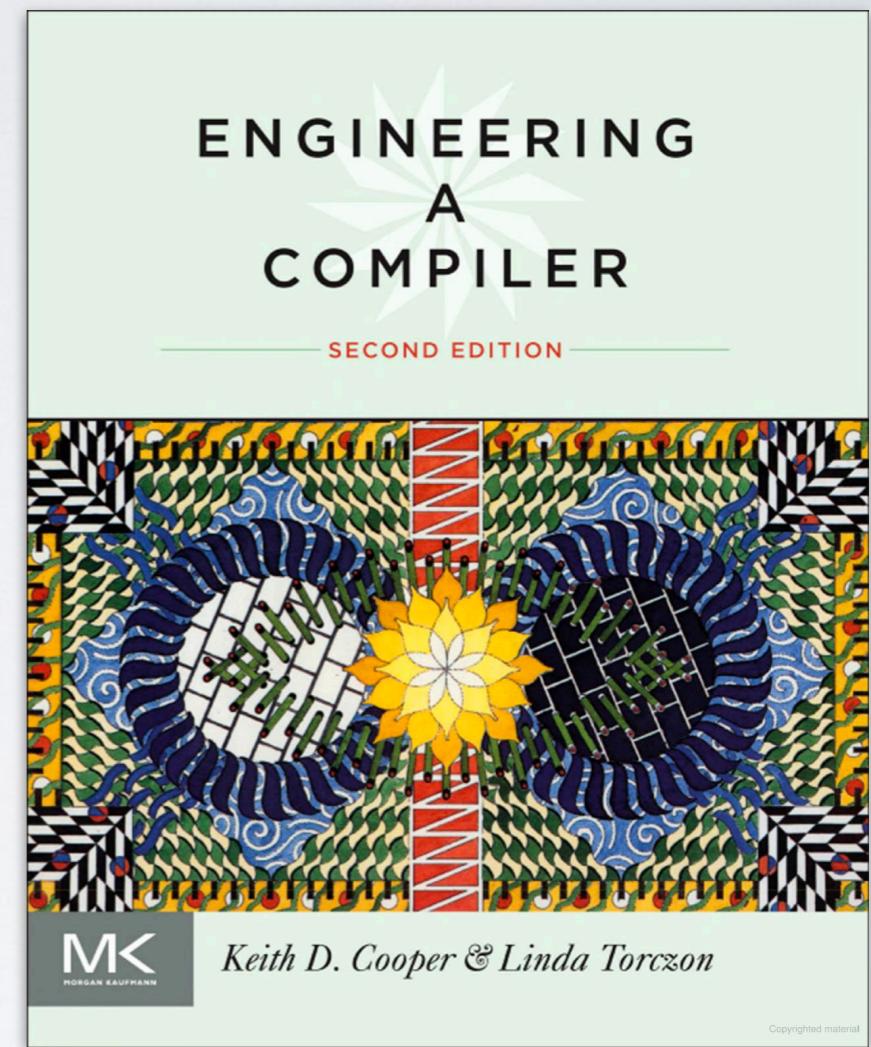
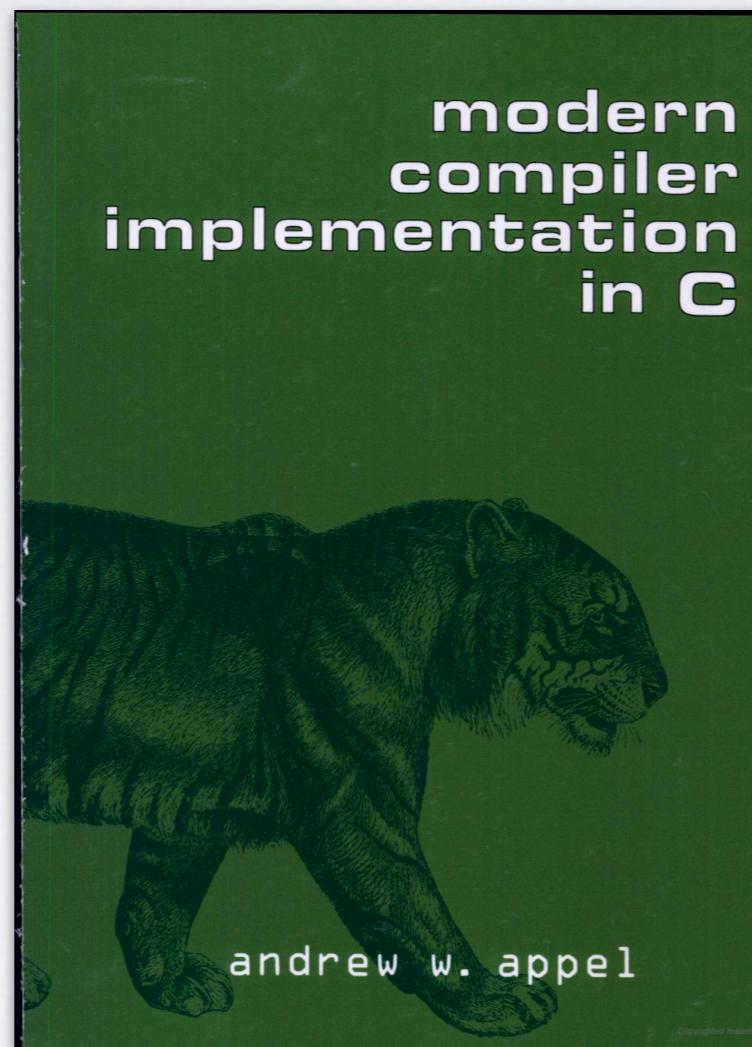
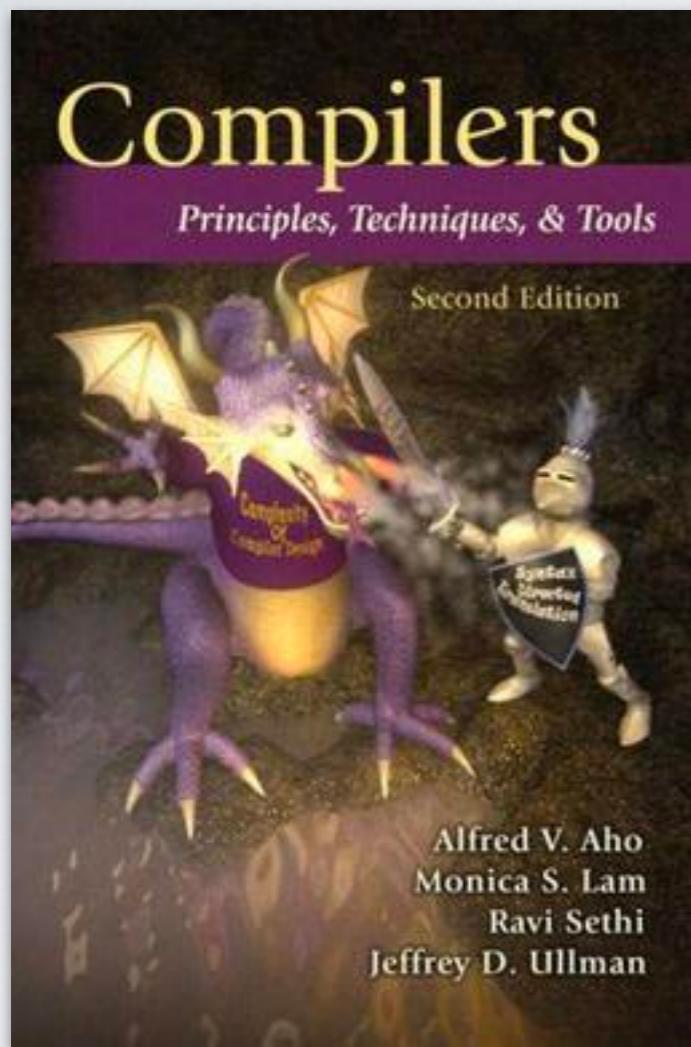
# 教材及参考书

## ◎ 教材:「龙书」

- ❖ Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. **Compilers: Principles, Techniques, and Tools** (Second Edition). Addison-Wesley, 2007.
- ❖ 中文翻译版或本科教学版:《编译原理(第2版)》, 赵建华、郑滔、戴新宇译, 机械工业出版社, 2009.

## ◎ 参考书

- ❖ 「虎书」: Andrew W. Appel and Maia Ginsburg. **Modern Compiler Implementation in C**. Cambridge University Press, 1998.
- ❖ Keith D. Cooper and Linda Torczon. **Engineering a Compiler** (Second Edition). Morgan Kaufmann, 2011.
- ❖ 中文翻译版:《编译器设计(第2版)》, 郭旭译, 人民邮电出版社, 2013.





# 评分标准

- 期中考试 (20%)
- 期末考试 (30%)
- 课程实践 (40%)
  - ❖ 实现 SysY 语言 (C 语言的子集) 到 RISC-V 汇编的编译器
- 书面作业 (10%)
- 注：成绩比例可能按需调整



# 课程要求

## ○ 课堂

- ❖ 有问题请随时打断老师
- ❖ 积极参与随堂小测
  - ❖ 在第 2 节课末尾，通过教学网进行

## ○ 考试：鼓励知识的灵活运用，不提倡死记硬背

## ○ 实践：工作量大，建议早做规划，遇到问题及时反馈

- ❖ 在线文档：<https://pku-minic.github.io/online-doc/>

## ○ 其它

- ❖ 如果课堂、考试需要请假，需提前向老师提出，事后请假无效
- ❖ 重修、旁听、留学生或有特殊情况的同学请课间与我交流一下



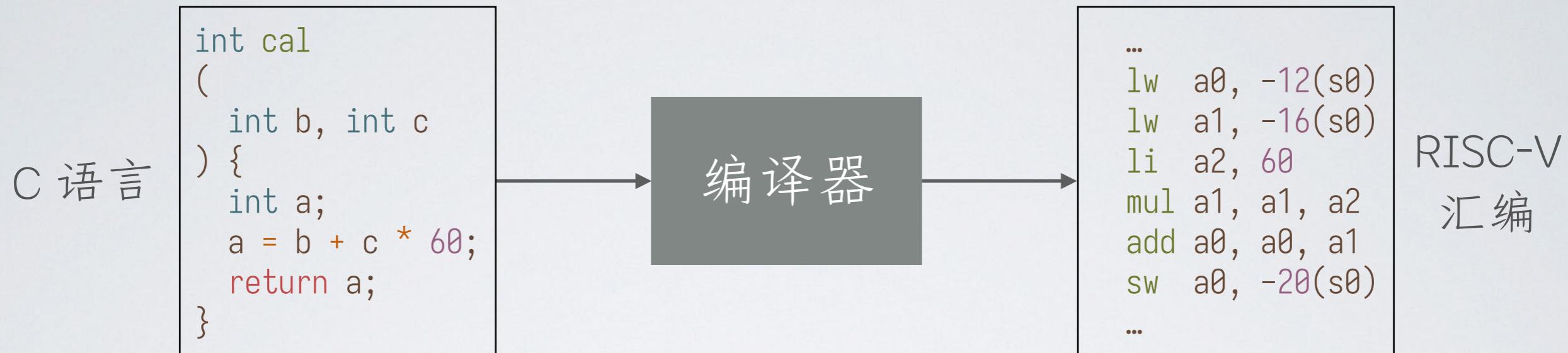
# 为什么要学编译原理？

除了用来满足专业核心课学分之外？



# 编译技术有什么应用？

# 应用1：实现编译器



- 编译器是一个程序，它的功能是**翻译**别的程序
  - ❖ 源程序 (source program): 通常使用一个**编程语言**来编写
  - ❖ 目标程序 (target program): 通常表示为一个处理器的**指令集**
- 从「方便程序员表达的」翻译为「方便机器运行的」
- 从「做什么」翻译为「怎么做」



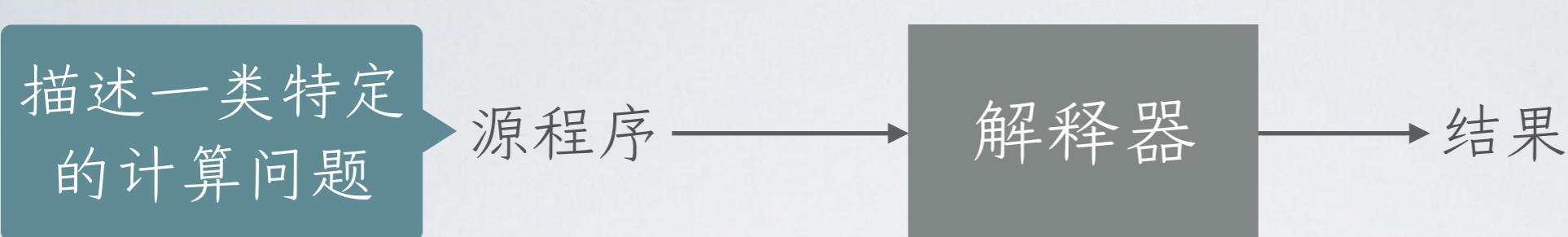
# 应用2：实现语言间的翻译



- 高级编程语言可以先翻译到 C 语言，再调用 C 的编译器
- 二进制翻译：不同指令集上的程序之间的转换
  - ❖ 苹果的 Rosetta：在 ARM 芯片上运行基于 x86 的软件
- 排版程序：把「描述排版的语言」翻译为「描述图像的语言」
  - ❖ LaTeX：先翻译到 DVI，再进一步翻译到 PostScript
- 硬件合成：把硬件描述翻译为门电路，再进一步翻译为物理布局
  - ❖ 芯片设计涉及的 EDA 软件中的重要部分



# 应用3：实现解释器



- 解释器是一个程序，它的功能是计算源程序的执行结果
- 很多语言采取解释器实现，如 Python、Ruby、JavaScript 等
- PostScript 程序转换为图片像素的过程是一个解释器
- 数据库对查询语言(如 SQL)的处理是通过解释进行的
- 一个解释器为一类特定的计算问题提供了解决方案

# 应用4：实现领域特定语言

- **HTML 和 CSS:** 描述网页的结构和样式的语言
  - ❖ 通过浏览器进行解释，渲染页面

```

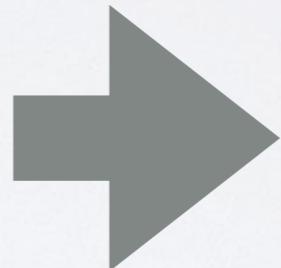
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <link rel="stylesheet" type="text/css" href="css_sample_file.css">
5      <!-- The value of 'href' should be a file path to your chosen css file --&gt;
6  &lt;/head&gt;
7  &lt;body&gt;
8      &lt;h1&gt;Hey there! I'm a header!&lt;/h1&gt;
9      &lt;button class="button"&gt; I'm a button! &lt;/button&gt;
10 &lt;/body&gt;
11 &lt;/html&gt;
</pre>

```

```

11 button {
12     background-color:#e30074;
13     font-family: 'Fredoka One';
14     font-size: 20px;
15     color: white;
16     height: 300px;
17     width: 300px;
18     box-shadow: none;
19     border-width: 4px;
20     border-color: white;
21     border-radius: 150px;
22 }

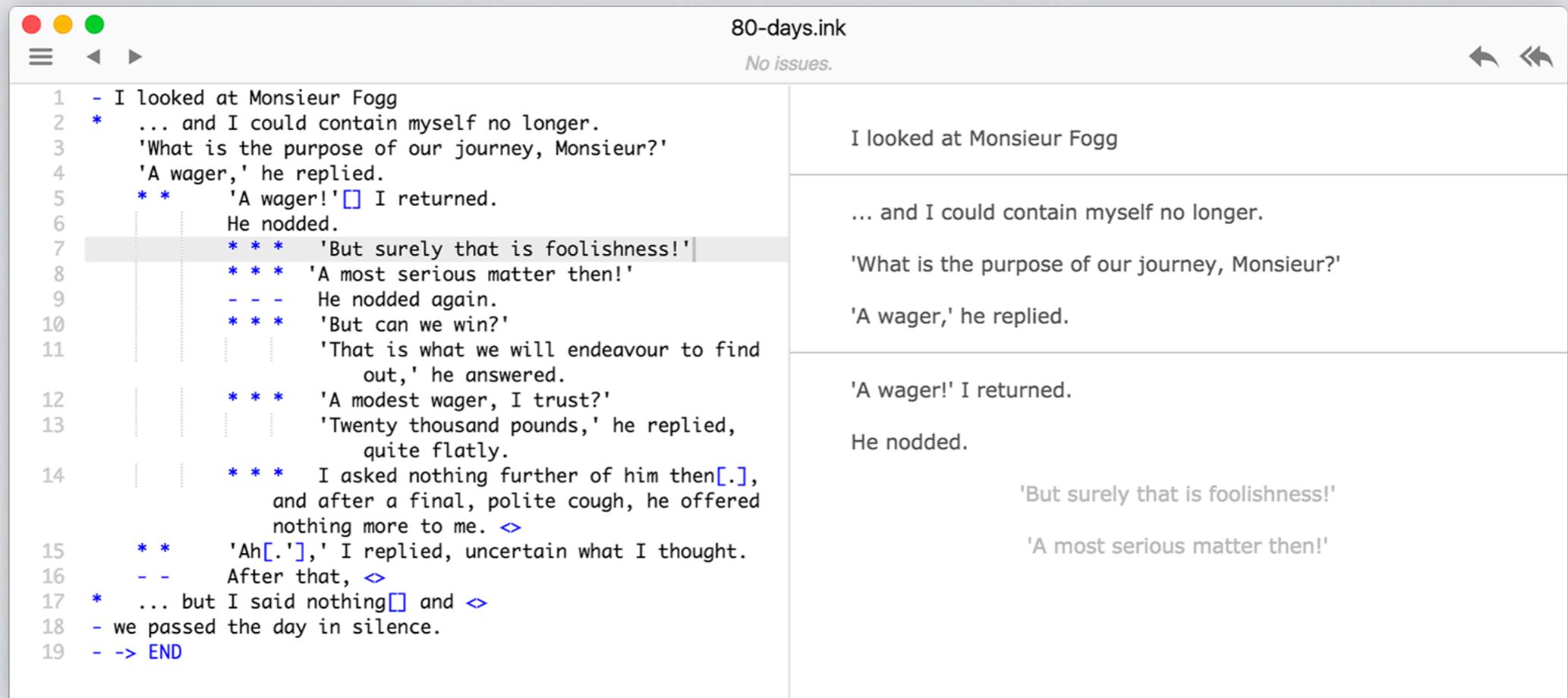
```



图片来源：<https://medium.com/@nikila.saravanan/html-css-javascript-101-f1af10e2f154>.

# 应用4：实现领域特定语言

- **ink:** 用于交互式故事和剧情设计的语言
  - ❖ 用于简化游戏开发，可通过插件在 Unity 游戏引擎中使用
  - ❖ 支持丰富的控制逻辑，包括条件分支和循环



The screenshot shows a Mac OS X-style window titled "80-days.ink". The left pane contains the Ink source code, and the right pane shows the generated story text.

```

1 - I looked at Monsieur Fogg
2 * ... and I could contain myself no longer.
3 'What is the purpose of our journey, Monsieur?'
4 'A wager,' he replied.
5 ** 'A wager!' □ I returned.
6 He nodded.
7 * * * 'But surely that is foolishness!'
8 * * * 'A most serious matter then!'
9 - - - He nodded again.
10 * * * 'But can we win?'
11     'That is what we will endeavour to find
12         out,' he answered.
13 * * * 'A modest wager, I trust?'
14     'Twenty thousand pounds,' he replied,
15         quite flatly.
16 * * * I asked nothing further of him then[.],
17     and after a final, polite cough, he offered
18         nothing more to me. <>
19 * * 'Ah[.]', I replied, uncertain what I thought.
20 - - After that, <>
21 * ... but I said nothing□ and <>
22 - we passed the day in silence.
23 - -> END

```

The right pane displays the generated story text:

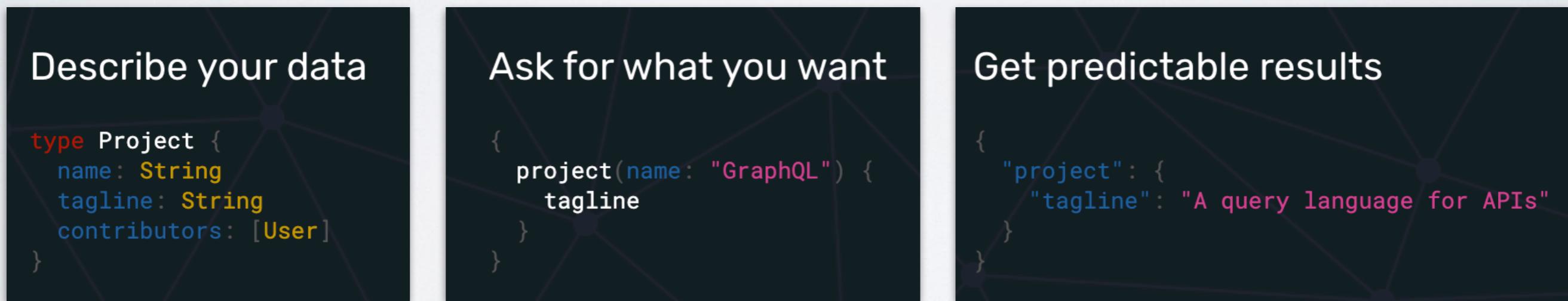
I looked at Monsieur Fogg  
... and I could contain myself no longer.  
'What is the purpose of our journey, Monsieur?'  
'A wager,' he replied.  
'A wager!' I returned.  
He nodded.  
'But surely that is foolishness!'  
'A most serious matter then!'

图片来源: <https://www.inklestudios.com/ink/>.

# 应用4：实现领域特定语言

## ● GraphQL：图状数据查询语言

- ❖ 定义数据关联的语言 + 嵌套查询语言
- ❖ 可以与不同的数据库后端对接
- ❖ 通过解释从数据图中查询信息

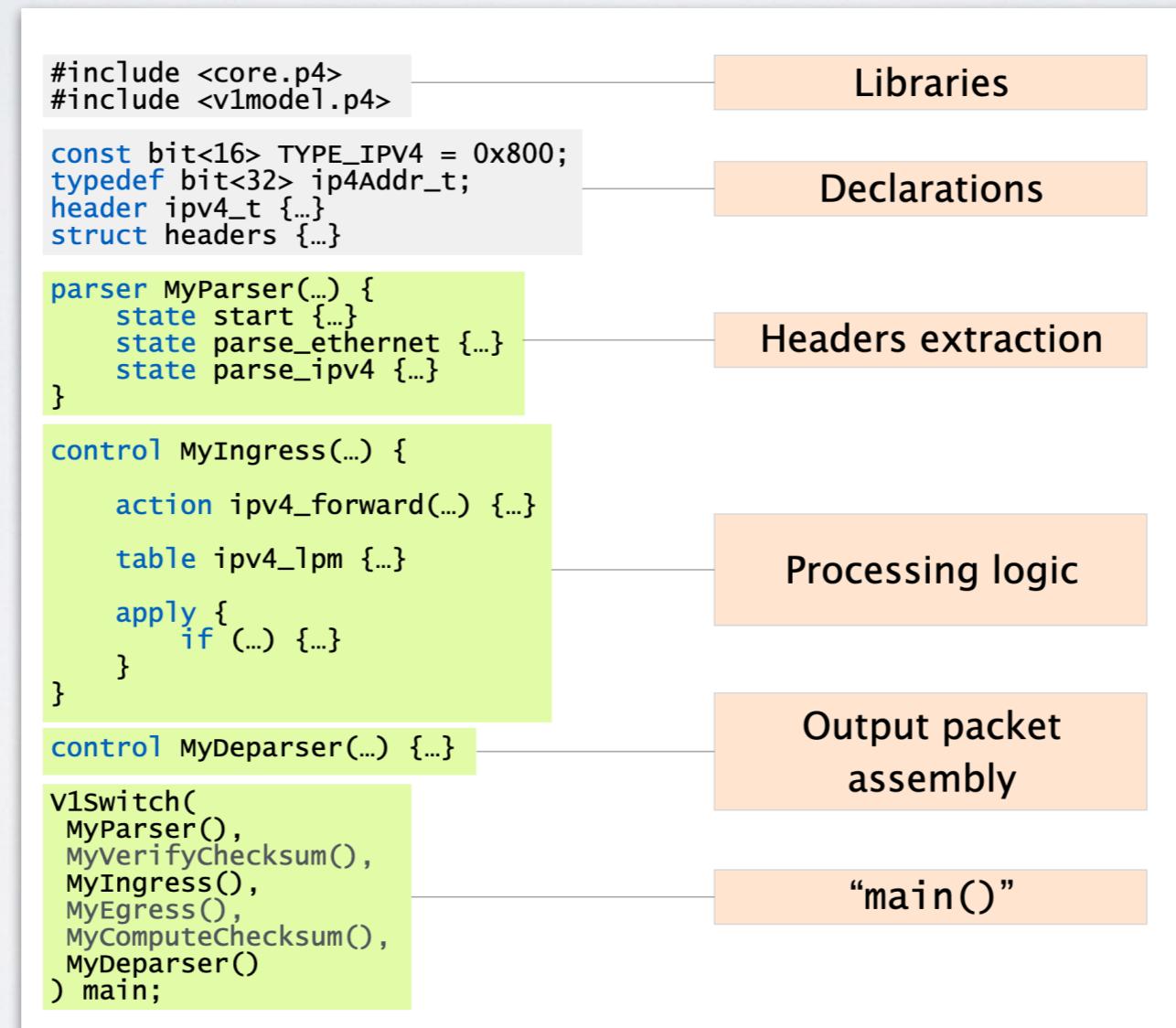


图片来源：<https://graphql.org/>.

# 应用4：实现领域特定语言

- p4: 针对网络设备的编程语言

- ❖ 可以编译到不同的硬件(如 FPGA)或软件目标

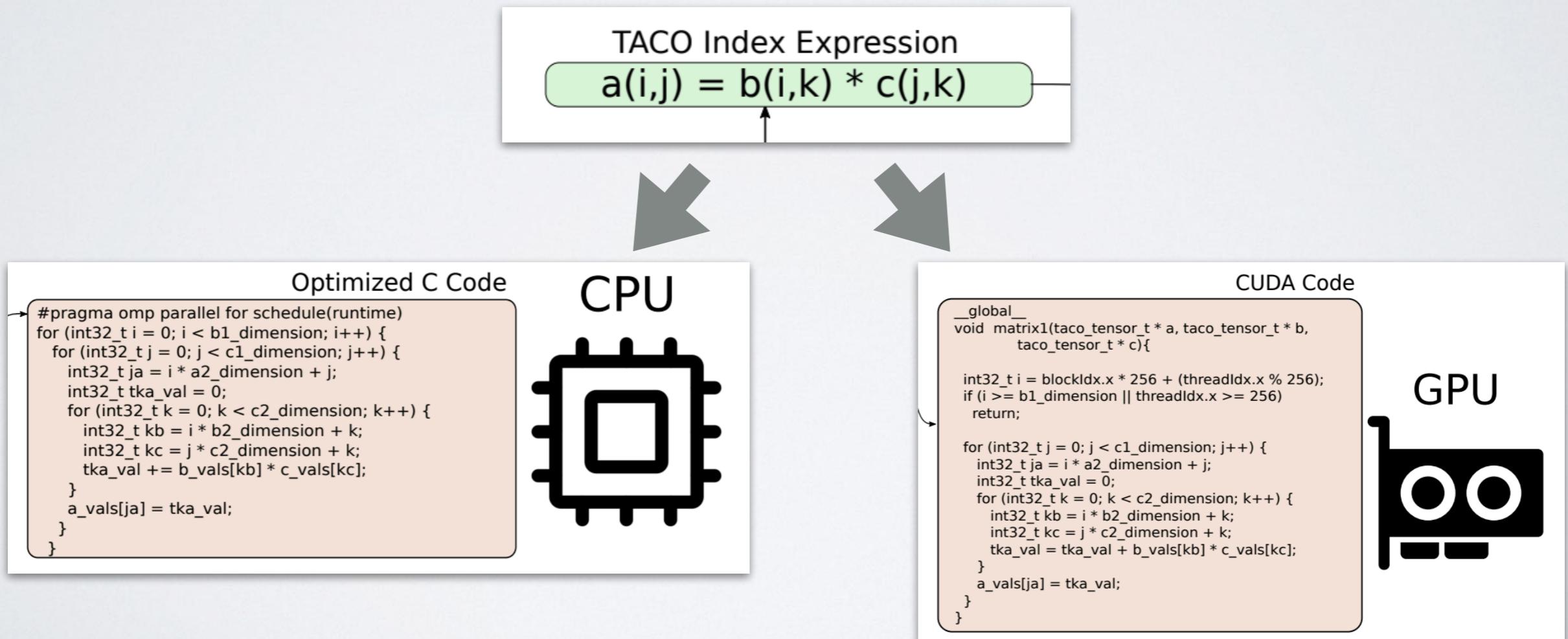


图片来源: <https://adv-net.ethz.ch/2022/>, Lecture 1.

# 应用4：实现领域特定语言

- **TACO**: 针对稀疏张量计算的编译器

- ❖ 从张量计算的式子编译为高效的 CPU/GPU 代码
- ❖ 应用于数据分析、科学计算、机器学习等领域



# 应用4：实现领域特定语言

- **Taichi**: 高性能并行计算语言

- ❖ 最初是面向图形学领域设计的
- ❖ 嵌入在 Python 中并在编译为 CPU/GPU 代码时实现了大量优化

```

import taichi as ti

ti.init(arch=ti.gpu) # Run on GPU by default

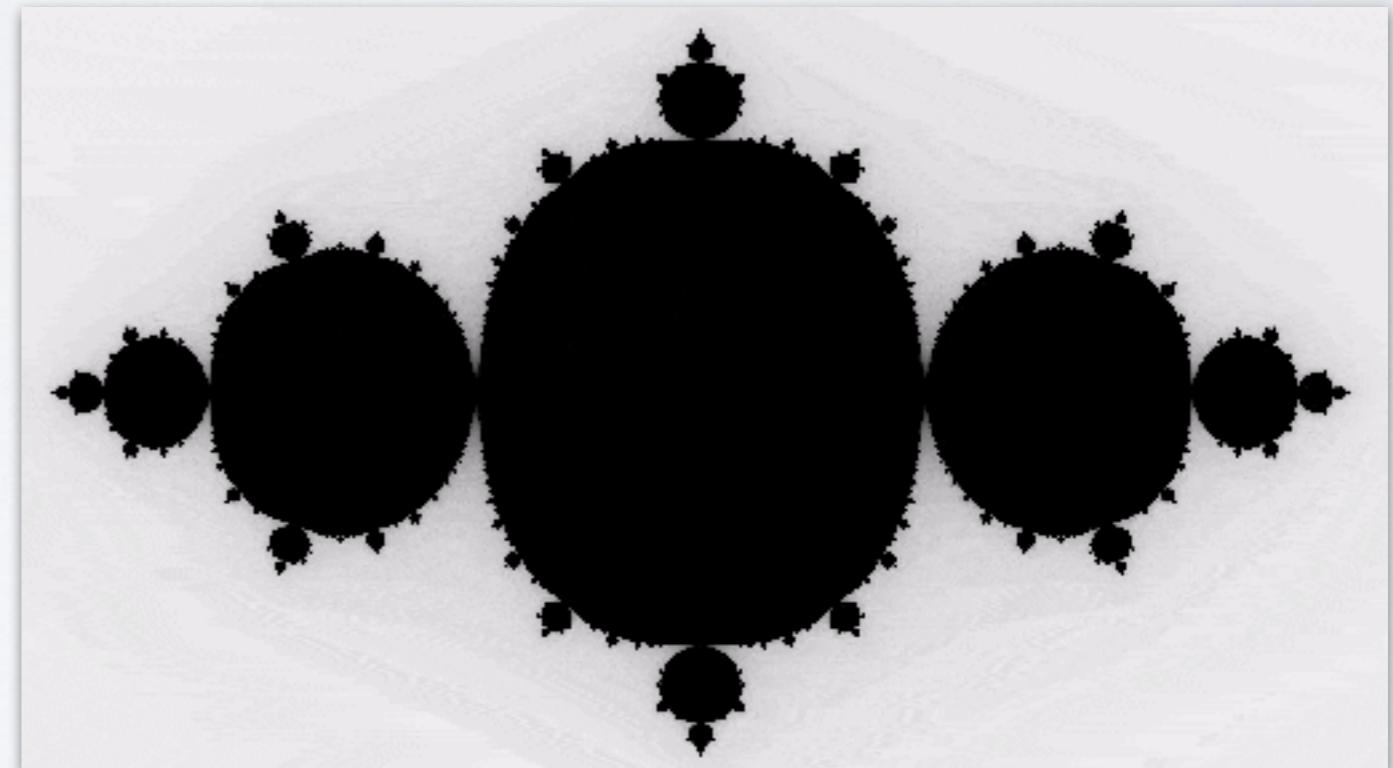
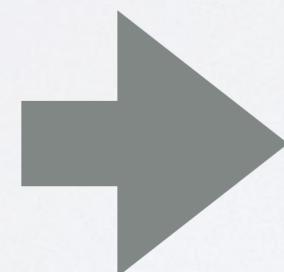
n = 320
pixels = ti.field(dtype=float, shape=(n * 2, n))

@ti.func
def complex_sqrt(z):
    return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])

@ti.kernel
def paint(t: float):
    for i, j in pixels: # Parallelized over all pixels
        c = ti.Vector([-0.8, ti.cos(t) * 0.2])
        z = ti.Vector([i / n - 1, j / n - 0.5]) * 2
        iterations = 0
        while z.norm() < 20 and iterations < 50:
            z = complex_sqrt(z) + c
            iterations += 1
        pixels[i, j] = 1 - iterations * 0.02

gui = ti.GUI("Julia Set", res=(n * 2, n))

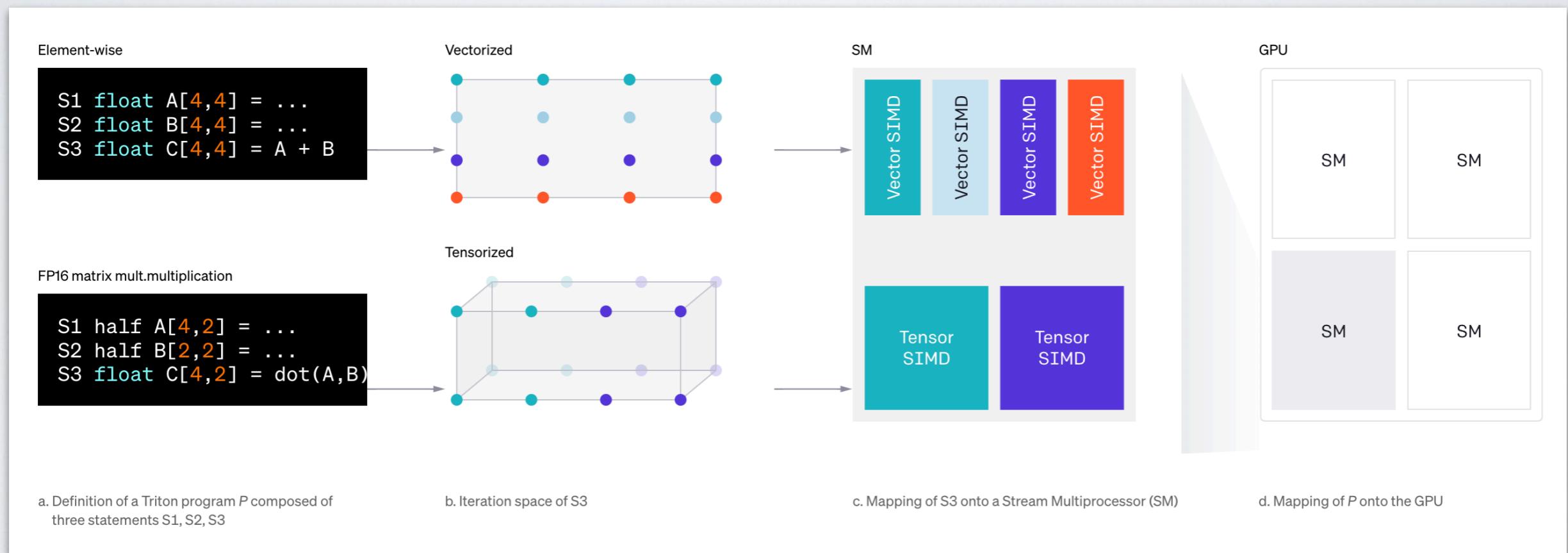
for i in range(1000000):
    paint(i * 0.03)
    gui.set_image(pixels)
    gui.show()
    
```



图片来源: <https://github.com/taichi-dev/taichi>.

# 应用4：实现领域特定语言

- **Triton**: 为神经网络设计的面向 GPU 的语言
  - ❖ 在编译过程中实现了大量针对神经网络的优化
  - ❖ 在 PyTorch 2.0 的默认深度学习编译器中使用



图片来源: <https://openai.com/research/triton>.

信息来源: <https://pytorch.org/docs/stable/torch.compiler.html>.

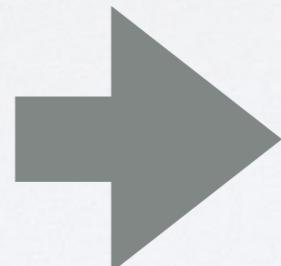
# 应用4：实现领域特定语言

- **Scenic**: 自动驾驶场景建模语言

- ❖ 描述带有不确定性的驾驶场景
- ❖ 通过解释执行采样，再通过其它工具(如 GTA 游戏)进行渲染

```

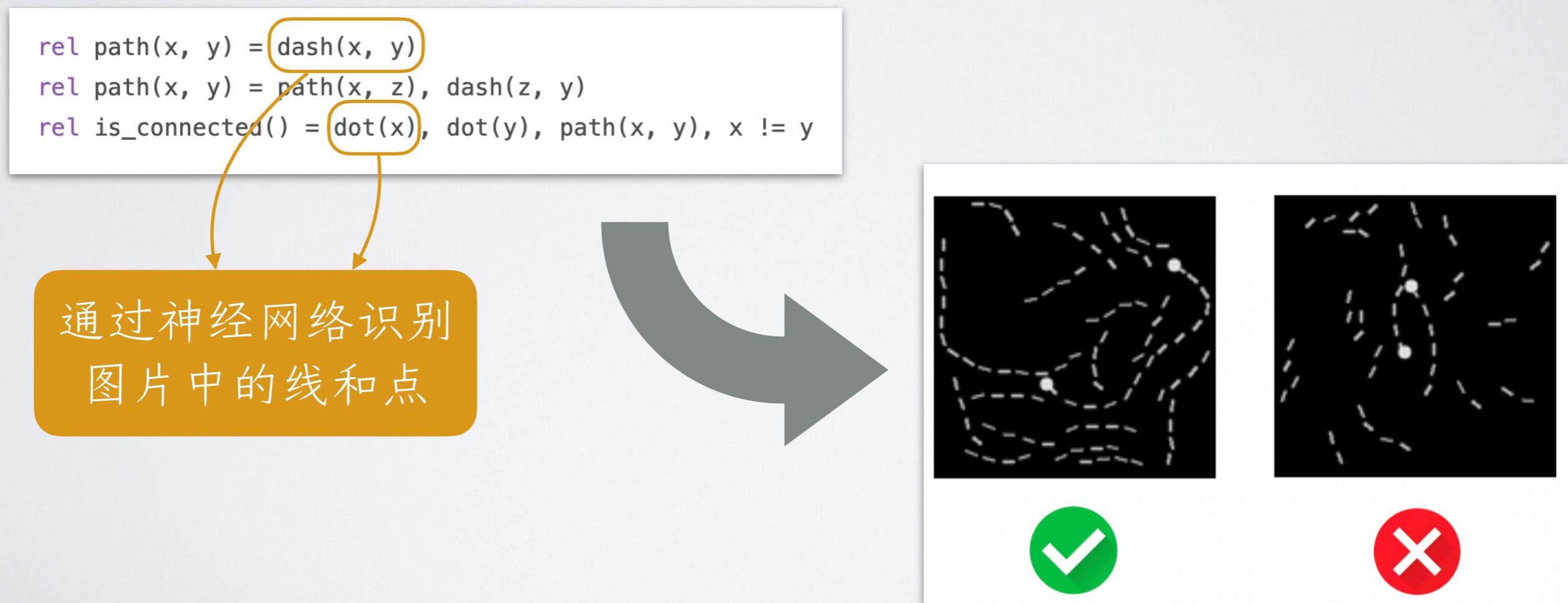
spot = OrientedPoint on visible curb
badAngle = Uniform(1.0, -1.0) *
           Range(10, 20) deg
Car left of spot by 0.5,
facing badAngle relative to
roadDirection
    
```



# 应用4：实现领域特定语言

- ◎ **Scallop**: 声明式神经符号编程语言

- ❖ 描述人工智能任务中的符号或逻辑规则
- ❖ 通过解释执行与机器学习模块协同工作

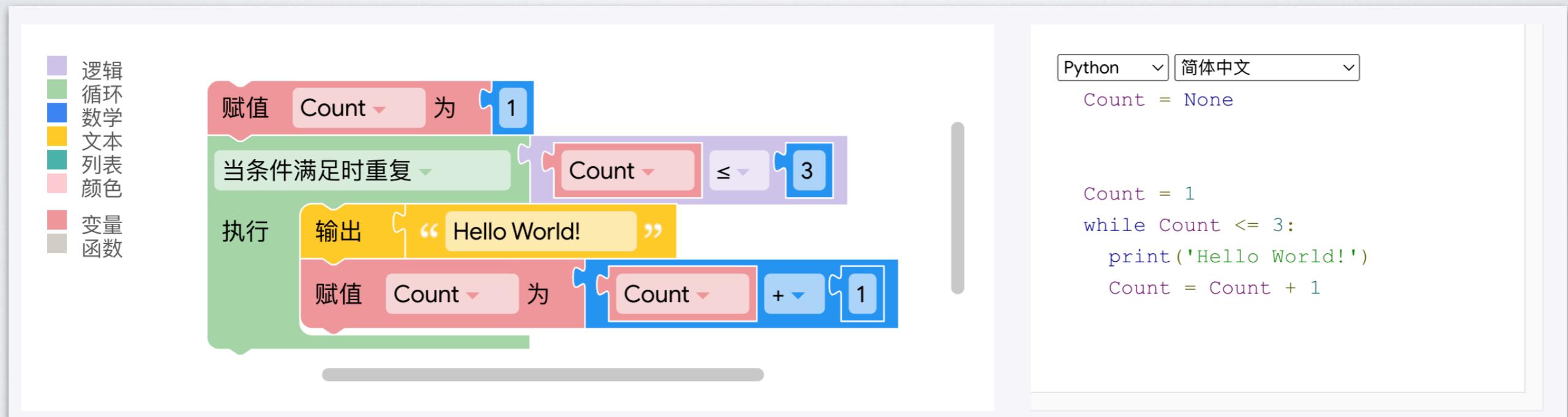


图片来源: <https://www.scallop-lang.org/>.

# 应用4：实现领域特定语言

- **Blockly**: 可视化编程工具

- ❖ 源程序: 可视化、图形化的程序
- ❖ 目标程序: 某种编程语言(如 Python)的程序

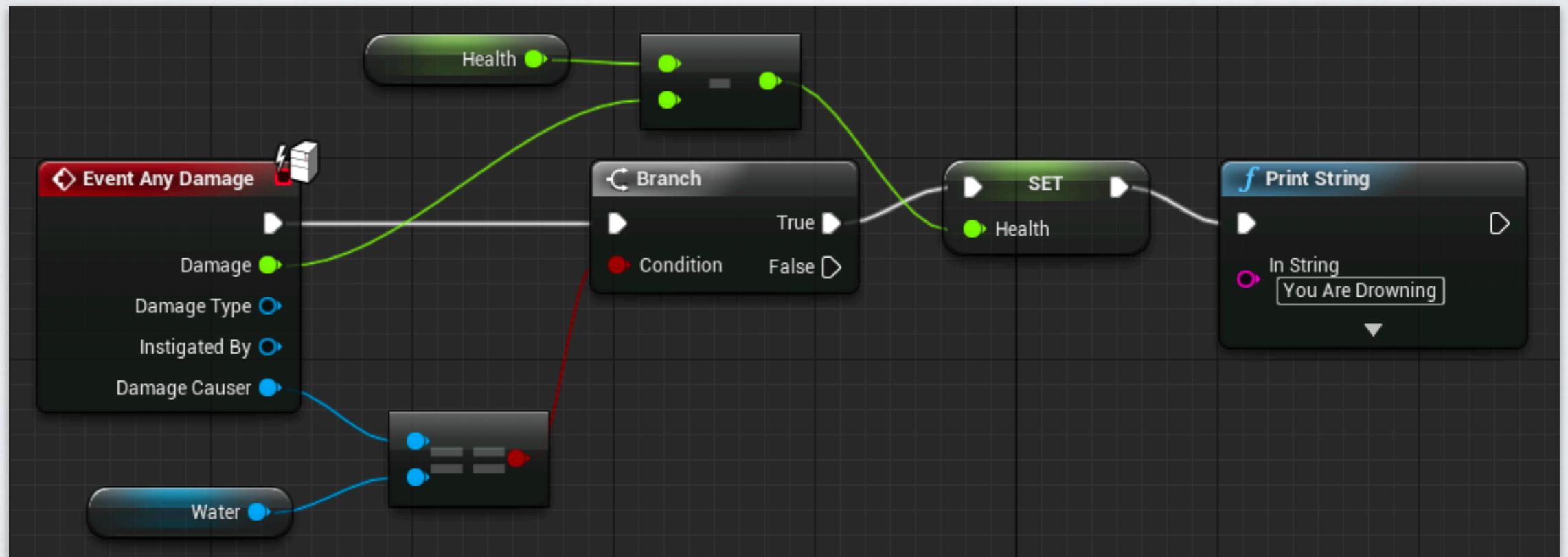


图片来源: <https://developers.google.com/blockly>.

# 应用4：实现领域特定语言

- **Blueprint:** 可视化游戏编程系统

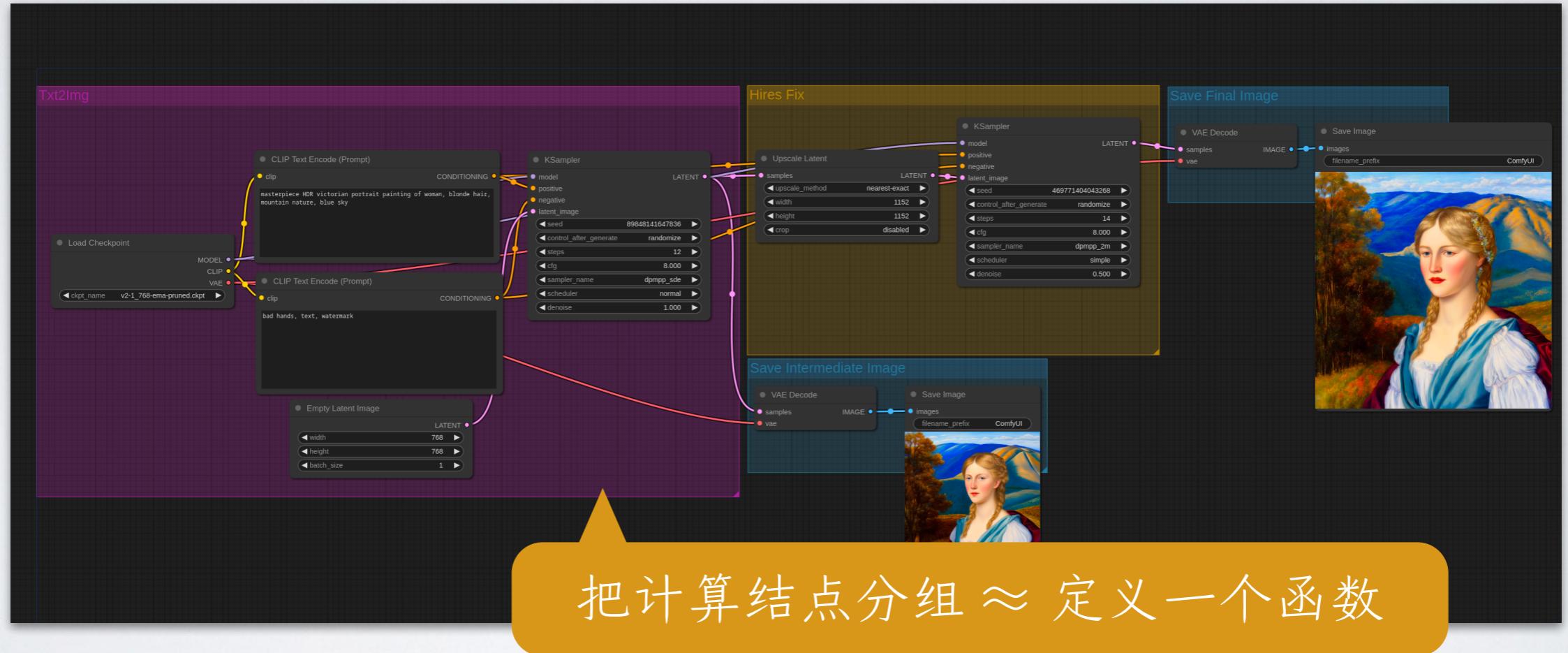
- ❖ 来自虚幻引擎，用于描述游戏中的逻辑和行为
- ❖ 编译为 C++ 代码或者虚幻引擎支持的代码形式
- ❖ 支持丰富的控制逻辑，包括条件分支和循环



图片来源: <https://docs.unrealengine.com/5.0/en-US/events-in-unreal-engine/>.

# 应用4：实现领域特定语言

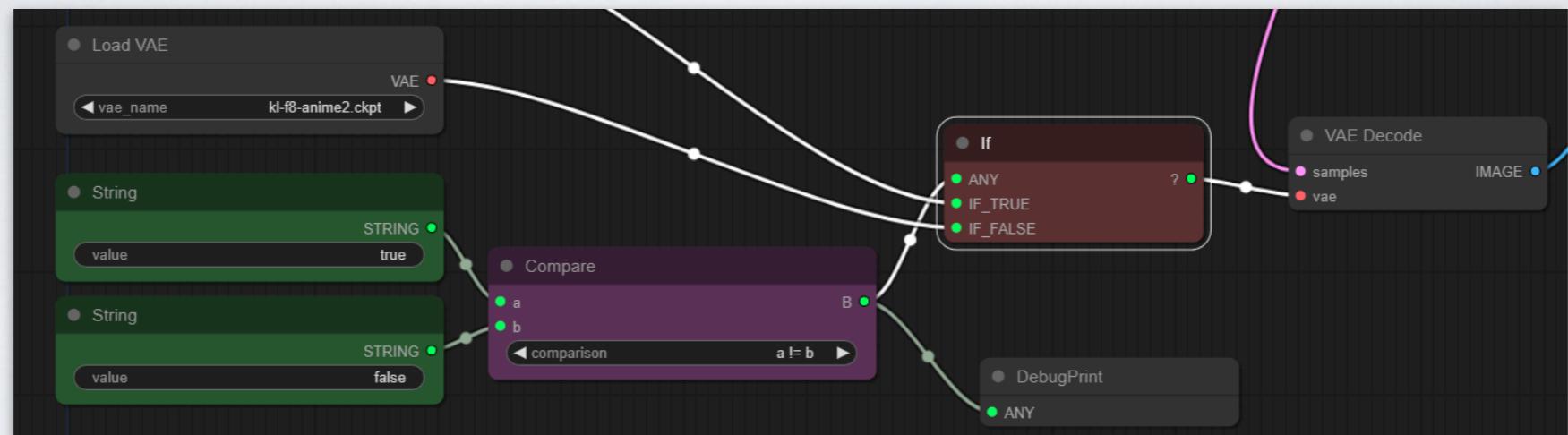
- **ComfyUI**: 针对 AI 绘画的流图式界面
  - ❖ 对可视化编辑的流程进行「解释」，计算结果为图片



- 但这真的算编程吗？哪里用到了编译技术？

图片来源: [https://comfyanonymous.github.io/ComfyUI\\_examples/2\\_pass\\_txt2img/](https://comfyanonymous.github.io/ComfyUI_examples/2_pass_txt2img/).

- 人们希望给 ComfyUI 添加更多的控制逻辑



- 人们希望把 ComfyUI 中的工作流翻译为 Python 代码

### ComfyUI-to-Python-Extension

The [ComfyUI-to-Python-Extension](#) is a powerful tool that translates [ComfyUI](#) workflows into executable Python code. Designed to bridge the gap between ComfyUI's visual interface and Python's programming environment,

图片来源: <https://github.com/theUpsider/ComfyUI-Logic>.

图片来源: <https://github.com/pydn/ComfyUI-to-Python-Extension>.



# 应用4：实现领域特定语言

- Domain-Specific Language, DSL
- 用程序来描述**领域特定问题**
  - ❖ 可以是图形化的程序！
- 通过编译技术，**自动**进行：
  - ❖ 编译到 CPU/GPU/FPGA 等硬件平台
  - ❖ 翻译到其它编程语言的程序
  - ❖ 直接通过**解释**执行，计算出结果
    - ❖ 结果可以是图片、网页等形式！
- 面向语言的程序设计
  - ❖ Language-Oriented Programming, LOP



# 应用5：新计算机体系结构的设计

- 编译器在处理器设计阶段就进行开发
  - ❖ 利用编译的效果来评价体系结构特征
- Reduced Instruction-Set Computer, RISC
  - ❖ RISC 之前，指令集的设计目标是让汇编编程更容易
  - ❖ 编译器优化常常能够消除复杂指令
  - ❖ RISC 体系结构中，编译器要能够有效地使用简单指令
  - ❖ 使用简单指令集使得硬件也更容易优化
  - ❖ 但是，代价是什么呢？
- RISC 体系结构对编译器的要求更高
- 挑战与机遇并存！



# 应用6：提高软件开发效率的工具

- 源于编译优化技术的程序分析一直在改进软件开发效率
  - ❖ 典型应用：**Language Servers**
- 类型检查
  - ❖ 捕捉程序中的前后不一致
- 边界检查
  - ❖ 数据流分析技术可用来定位缓冲区溢出
- 内存管理工具
  - ❖ 动态地捕捉内存管理错误
- 软件测试工具
  - ❖ 自动生成高覆盖率的测试用例



# 当前是学习编译技术的好时代

- 人们希望用计算机解决越来越广泛的问题
  - ❖ 领域特定语言
- 硬件体系结构正在多样化和异构化
  - ❖ CPU, GPU, TPU, FPGA, .....
  - ❖ 国产处理器的需求也催生了国产编译器的需求
- 编译器/解释器的设计目标也变得多元
  - ❖ 以前基本是追求运行速度
  - ❖ 现在：代码大小，能耗高低，产生的缺页异常多少，.....



# 为什么要学编译原理?

如果编译器是程序员的「武器」，  
那么编译原理将告诉你「如何打造武器」



# 编译原理中的科学

- 一个好的编译器是计算机科学的一个**缩影**
  - ❖ 贪心算法(寄存器分配)
  - ❖ 启发式搜索(指令排序)
  - ❖ 图算法(死代码消除)
  - ❖ 动态规划(指令选择)
  - ❖ 文法和自动机(词法分析、语法分析)
  - ❖ 不动点算法(数据流分析)
  - ❖ .....
- 编译是把理论应用到实际问题的**范例**
- 编译领域仍有很多**开放性问题**
  - ❖ 一些问题本身就是 NP 难, 甚至图灵不可判定的!



# 实践是检验真理的唯一标准

- 编译器是个不错的练习软件工程的实践项目
  - ❖ 编译器是庞大而复杂的程序
  - ❖ 实现编译器需要对相关算法的掌握、良好的工程能力以及项目规划
- 编译实践能帮助你提高编程技能
  - ❖ 理解计算机的底层原理
  - ❖ 理解编程语言、编译器、操作系统间的交互
  - ❖ 理解编译器如何优化代码，从而编写更容易被优化的代码
  - ❖ 使得学习新的编程语言更加容易
- 纸上得来终觉浅，绝知此事要躬行



WA

**You**

为什么不学编译原理?

**ChatGPT**

不学习编译原理可能有多种原因，这取决于个人的兴趣、目标、时间、能力等因素。以下是一些可能的原因：

- 兴趣和目标不匹配：**某些人可能对编译原理不感兴趣，或者他们的职业目标与编译原理无关。如果一个人更喜欢其他领域，如Web开发、数据科学、人工智能等，他可能会选择投入时间学习这些领域而不是编译原理。
- 专注于实用技能：**有些人可能更倾向于学习那些可以直接应用于实际工作中的技能，而编译原理可能被认为是一种较为理论性的学科，他们可能更愿意学习与实际项目更直接相关的技能。
- 时间和精力限制：**学习编译原理需要大量的时间和精力，包括理论学习、实践编程、实验和项目等。如果一个人的时间和精力有限，他可能会选择投入到其他更紧迫或更感兴趣的事务中。
- 其他重要技能优先：**在某些情况下，个人可能认为其他技能对他们的职业发展更为重要，因此他们会优先选择学习这些技能，而将编译原理放在次要位置。

总的来说，不学习编译原理可能是基于个人的兴趣、目标和优先级考虑。虽然学习编译原理具有其独特的好处，但并不是每个人都需要或愿意投入时间和精力来学习这个领域。



# 课程目标

- 掌握编译的基本理论
  - ❖ 强调形式化描述技术和自动生成技术
- 掌握常用的编译技术
  - ❖ 不过于偏向任何源语言或目标机器
    - ❖ 侧重命令式编程语言和 RISC 体系结构
  - ❖ 不过于强调算法细节
- 掌握编译器的构造技术
  - ❖ 通过实践来达成
- 进一步培养计算机思维能力



# 第一讲 编译概述

---

Introduction to Compilation



# 主要内容

- 编译的基本概念和原则
- 编译器的结构
- 翻译过程示例
- 对应章节：第 1 章、第 2 章



# 主要内容

- 编译的基本概念和原则
- 编译器的结构
- 翻译过程示例

# 基本概念

C 语 言

```
int cal
(
    int b, int c
) {
    int a;
    a = b + c * 60;
    return a;
}
```

编译器

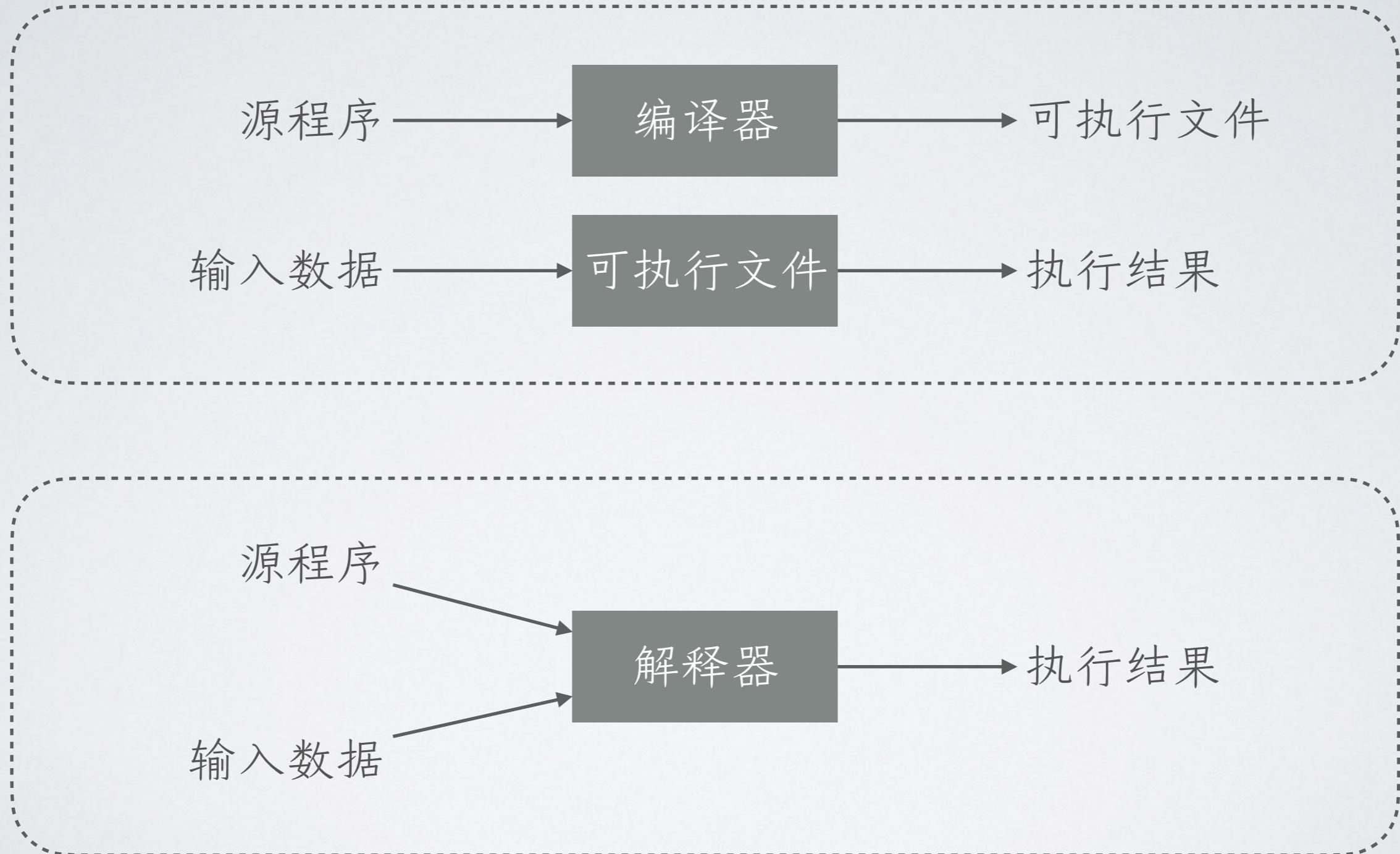
```
...
lw  a0, -12($0)
lw  a1, -16($0)
li  a2, 60
mul a1, a1, a2
add a0, a0, a1
sw  a0, -20($0)
...
```

RISC-V  
汇 编

- 编译器是一个程序，它的功能是**翻译**别的程序
  - ❖ 源程序 (source program): 通常使用一个**编程语言**来编写
  - ❖ 目标程序 (target program): 通常表示为一个处理器的**指令集**
- 目标程序可以是另一个编程语言的程序
- **解释器**不生成目标程序，而是直接计算源程序的**执行结果**



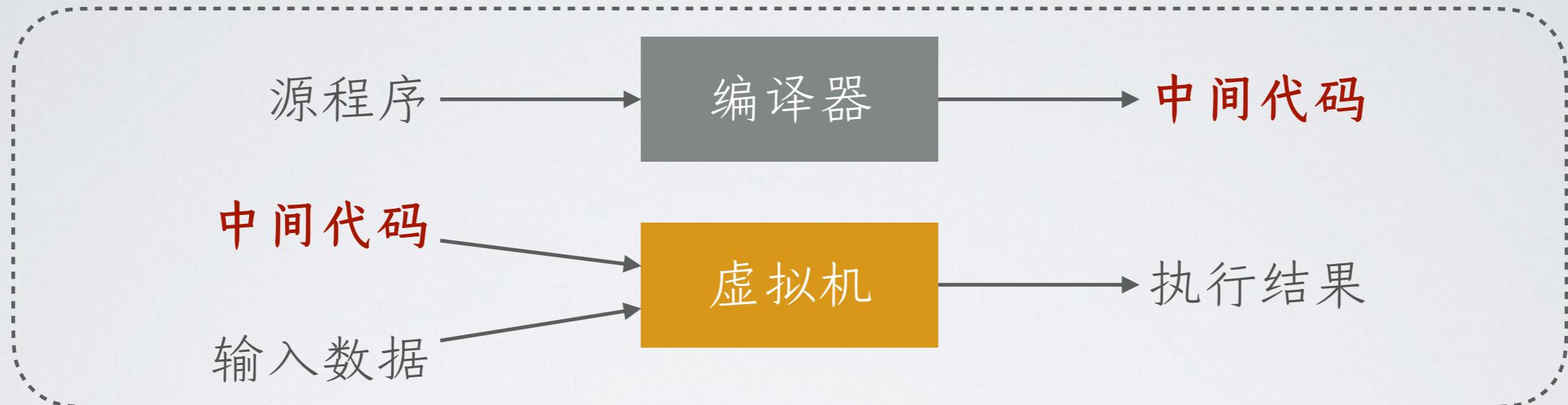
# 编译器与解释器





# 编译器与解释器

- 实现编程语言时，两者没有明确的边界，可以混合起来



- 虚拟机：中间代码的解释器**

- ❖ 虚拟机接受的中间代码通常为字节码(bytecode)形式

- 例子：

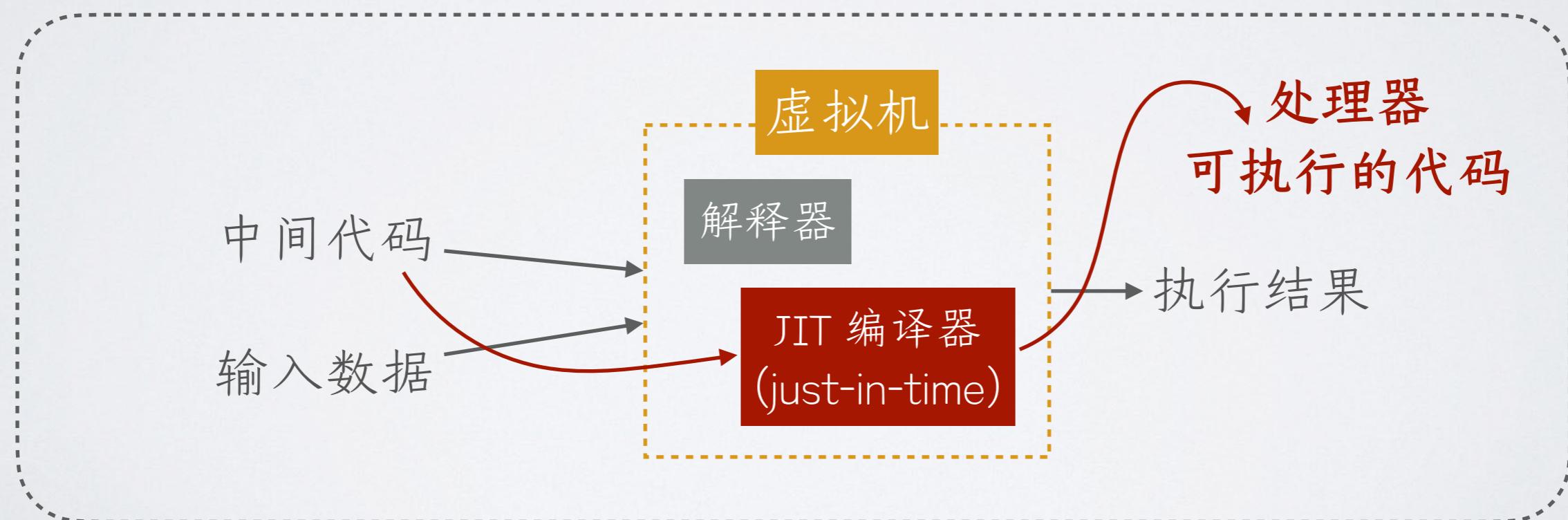
- ❖ Java 的虚拟机(如 JVM)、JavaScript 的虚拟机(如 V8)
- ❖ WebAssembly(Wasm)是目前流行的面向浏览器的中间代码形式

# 编译器与解释器

## ● 编译和解释两种方案各自有什么优缺点？

- ❖ 编译后程序的执行速度快；运行时环境不需要有编译器
- ❖ 解释器的开发流程快捷；解释执行的程序易于调试

## ● 虚拟机方案还可以怎么改进？





# 基本原则

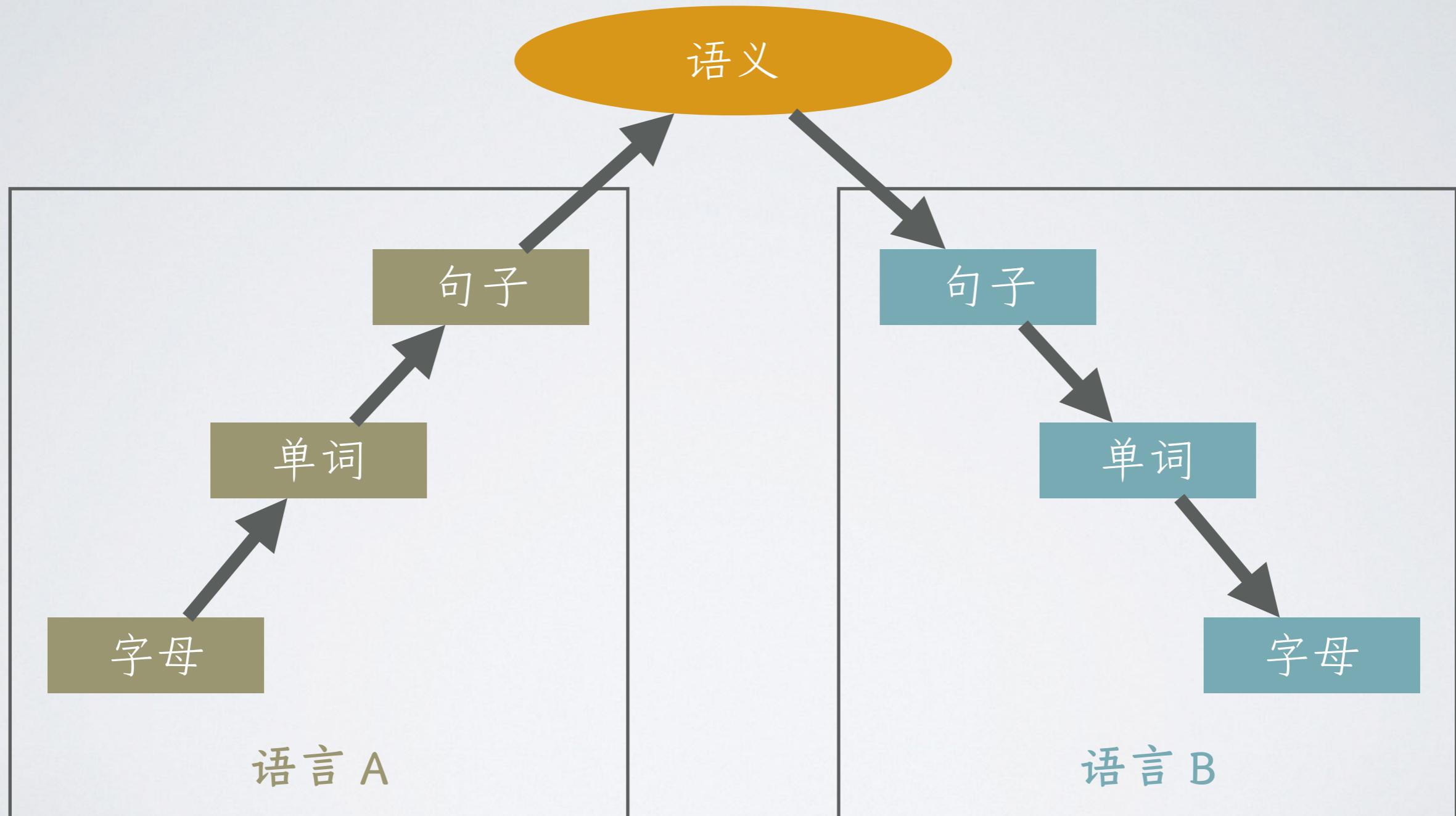
- 编译器必须保证目标程序和源程序有**等价的语义**
  - ❖ 等价: 针对相同的输入数据, 给出相同的执行结果
- 编译器必须以某种合理的方式**改进源程序**
  - ❖ 传统编译器: 翻译为可以在机器上直接执行的程序
  - ❖ 其它语言到 C 语言的编译器: C 有成熟的编译器、可移植性高
  - ❖ LaTeX: 从排版的描述生成可阅读、打印的文档
  - ❖ 同一语言上的转换器: 通过各种优化生成更好的代码
  - ❖ .....



# 主要内容

- 编译的基本概念和原则
- 编译器的结构
- 翻译过程示例

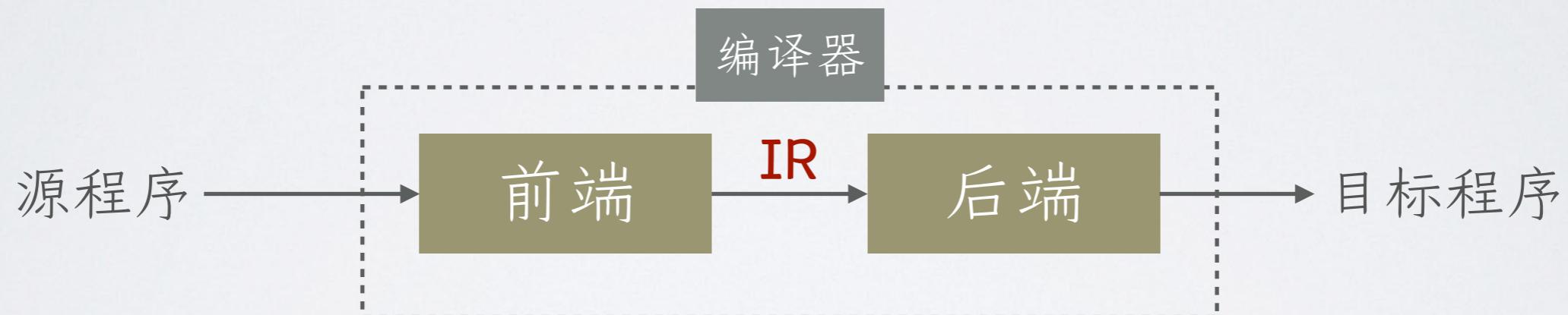
# 翻译



# 基本结构



- 翻译过程要求编译器既要理解源程序，也要理解目标机器
- 两个不同的任务驱动了基于**前端**和**后端**的设计

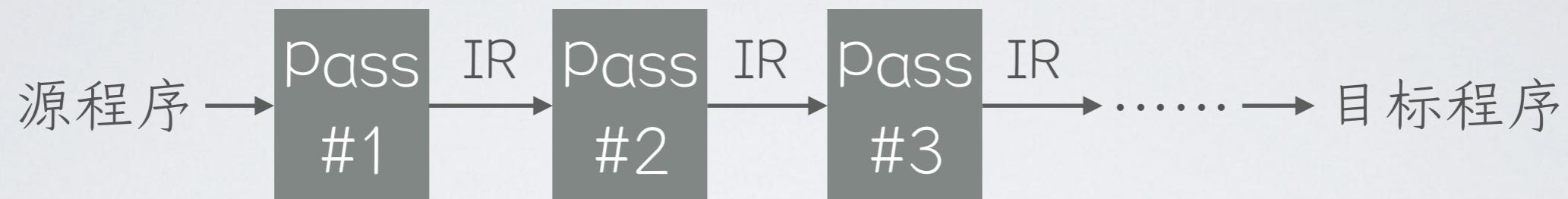


- ❖ **前端**: 理解源程序，得到语义
- ❖ **后端**: 把语义翻译为目标程序
- ❖ **中间表示**(intermediate representation, IR): 编译器对语义的表示形式



# 多趟处理

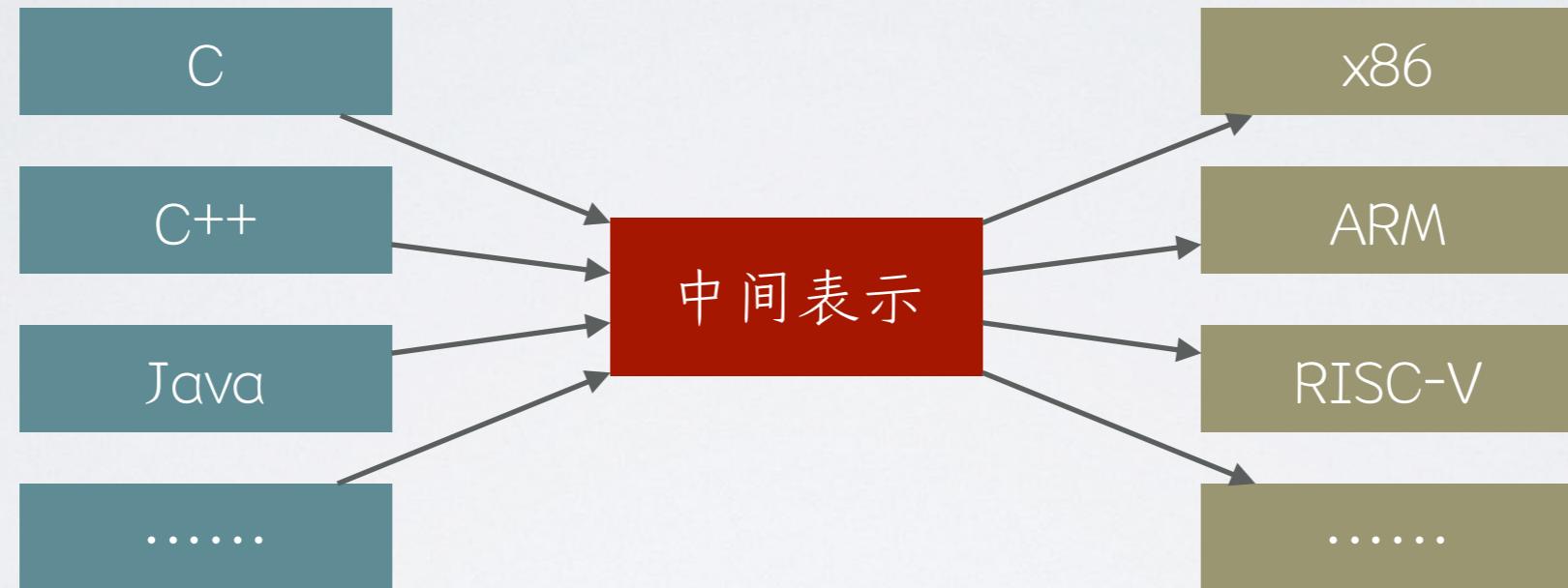
- 一趟 (pass) 通常指对程序从头到尾扫描一次



- 编译过程中可以使用不同的中间表示 (IR)
- 后面的趟可以利用前面的趟分析程序得出的信息
  - 多趟处理可能生成更好的代码
- 每一趟可以只做一些简单的任务
  - 多趟处理更好实现和维护

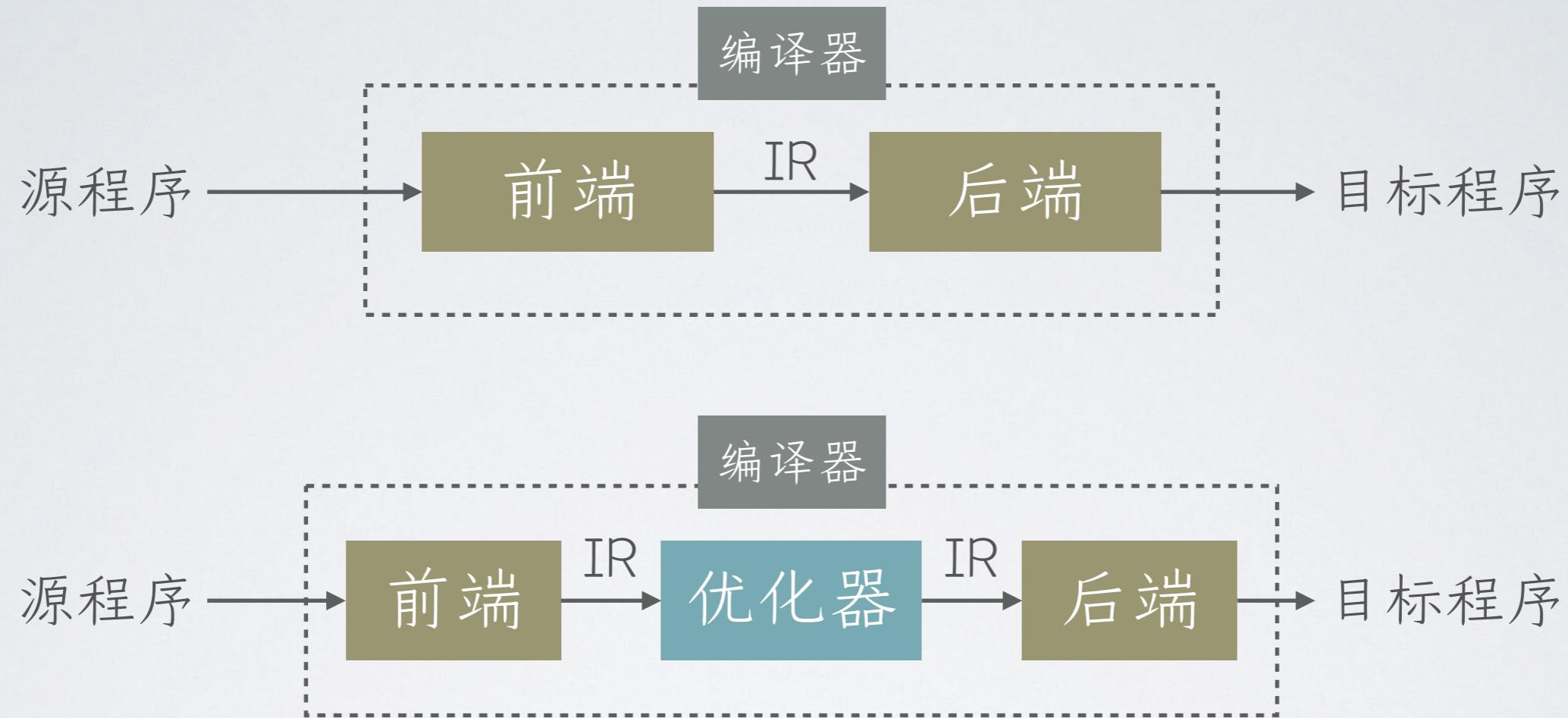
# 重定目标 (Retargeting)

- 重定目标一般指改变编译器使得其为别的处理器生成代码
- 现代编译器构造同时面对多种源语言和多种目标机器



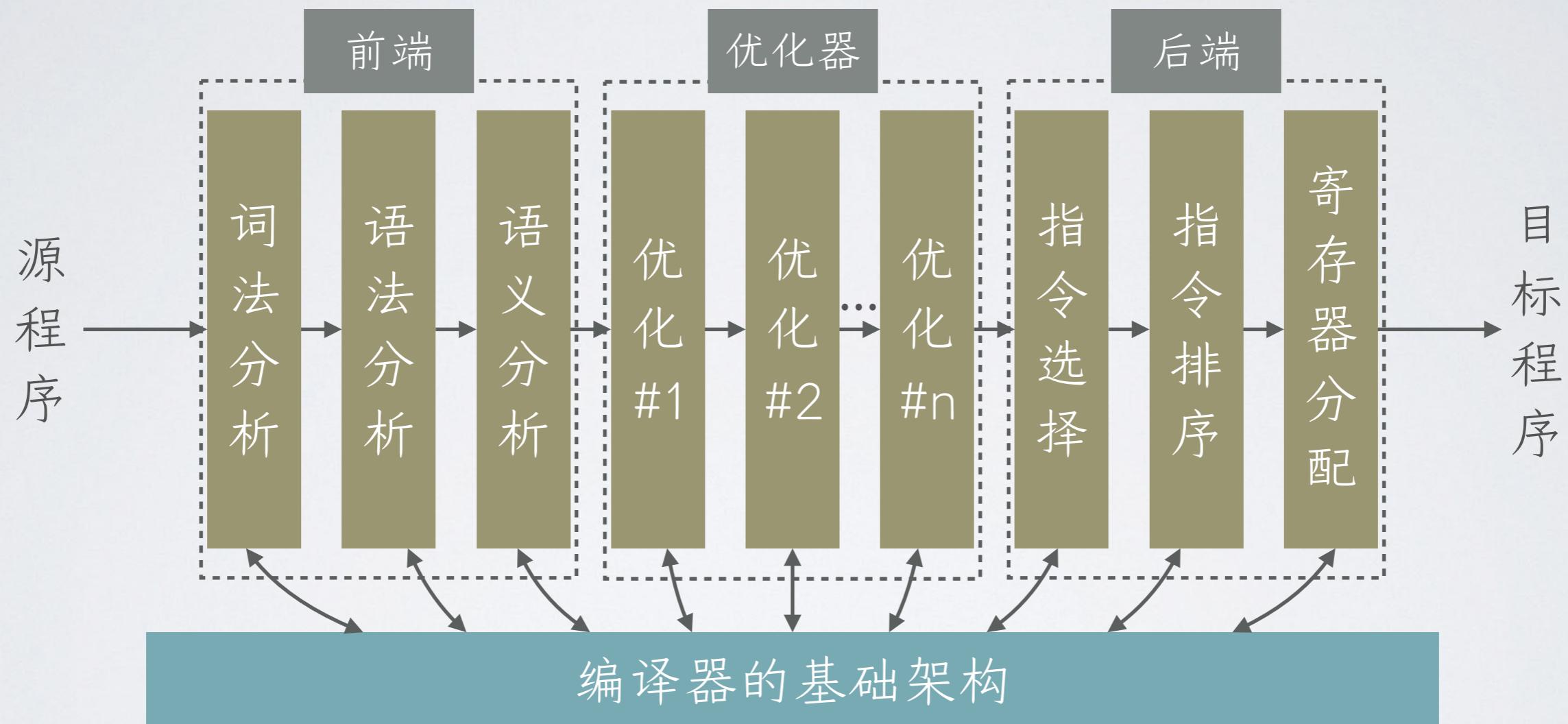
- 通过引入中间表示(IR)，可以在不同的编译器间复用代码
- **LLVM**: 一种流行的编译器中间表示形式

# 从两阶段到三阶段



- **优化器:** 负责分析、改进、转换中间表示(IR)
- 现代编译器中，优化是至关重要的阶段

# 典型编译器的结构





# 主要内容

- 编译的基本概念和原则
- 编译器的结构
- 翻译过程示例



# 翻译过程示例

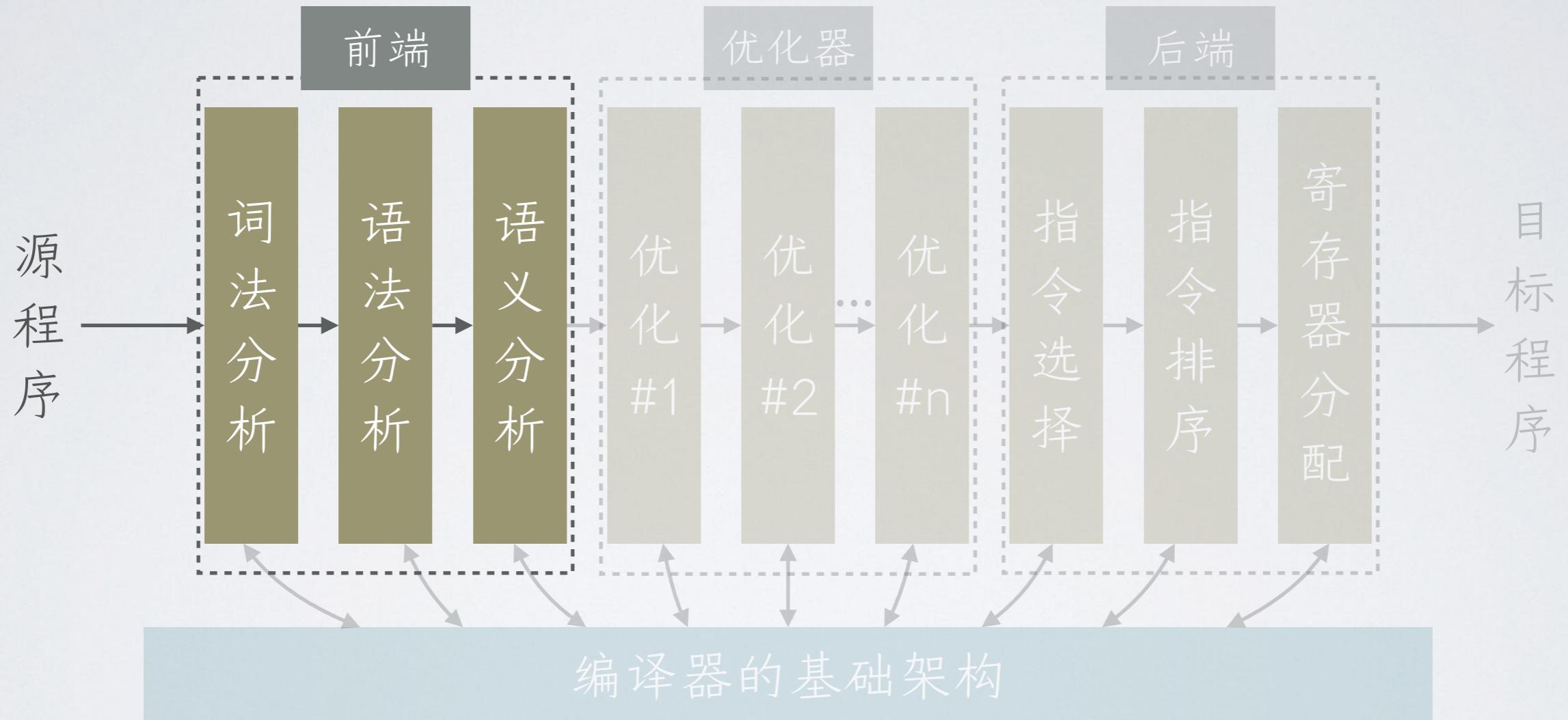
- 考虑如下这个赋值语句：

```
a = a * 2 * b * c * d;
```

- 我们将看到把它翻译为如下类 RISC 汇编代码的过程：

```
LD  R1, a          // 加载变量 a 到寄存器 R1
LD  R2, b          // 加载变量 b 到寄存器 R2
LD  R3, c          // 加载变量 c 到寄存器 R3
ADD R1, R1, R1    // R1 ← a * 2
MUL R1, R1, R2    // R1 ← (a * 2) * b
LD  R2, d          // 加载变量 d 到寄存器 R2
MUL R1, R1, R3    // R1 ← (a * 2 * b) * c
MUL R1, R1, R2    // R1 ← (a * 2 * b * c) * d
ST  a, R1          // 保存寄存器 R1 到变量 a
```

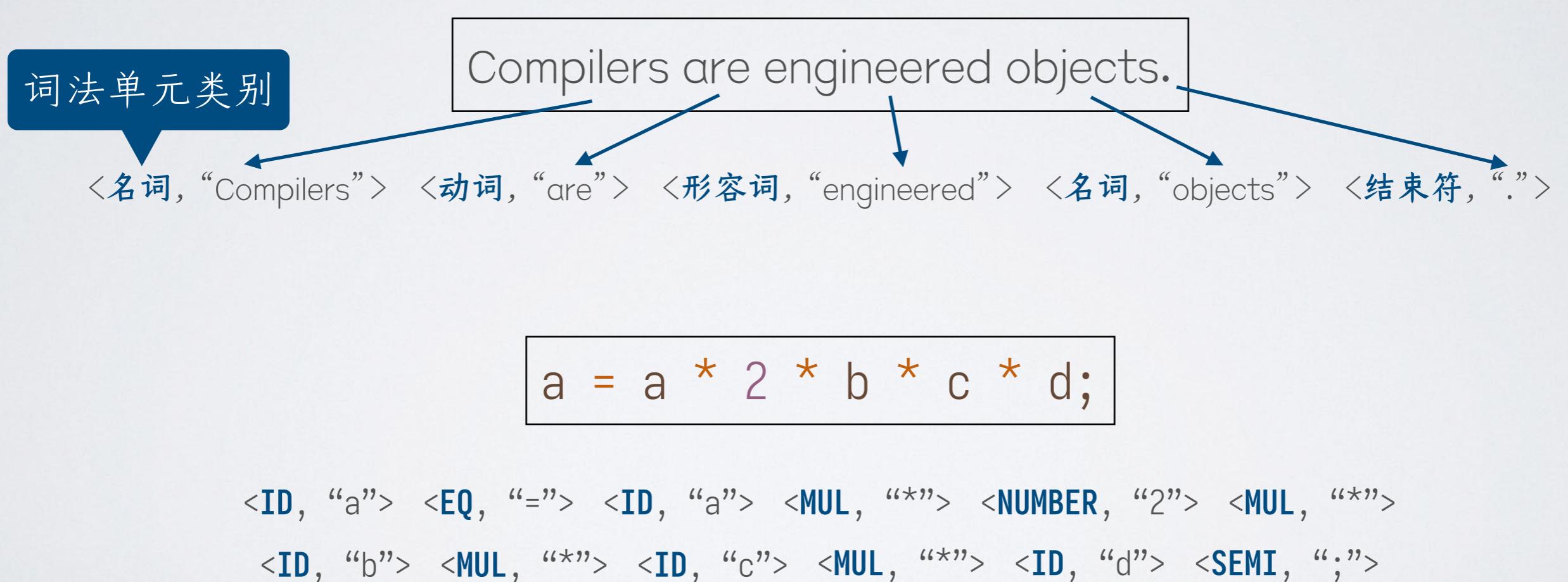
# 前端





# 词法分析

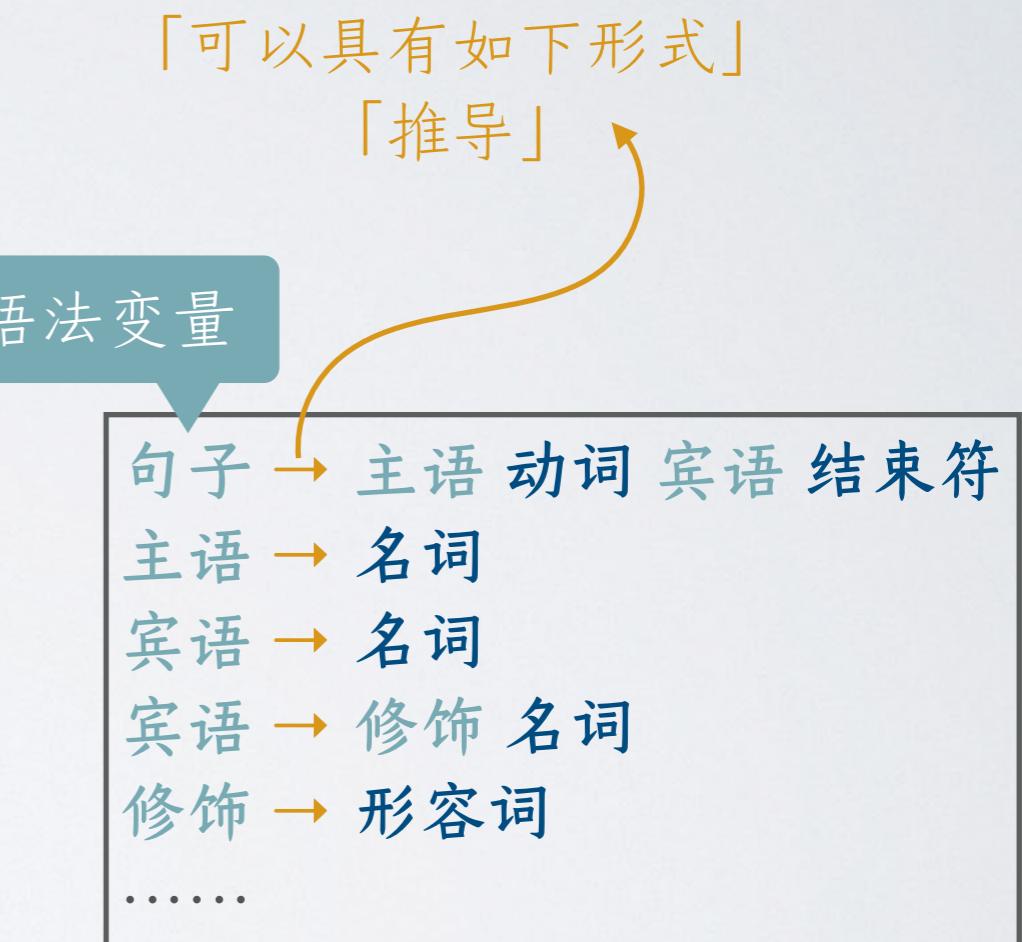
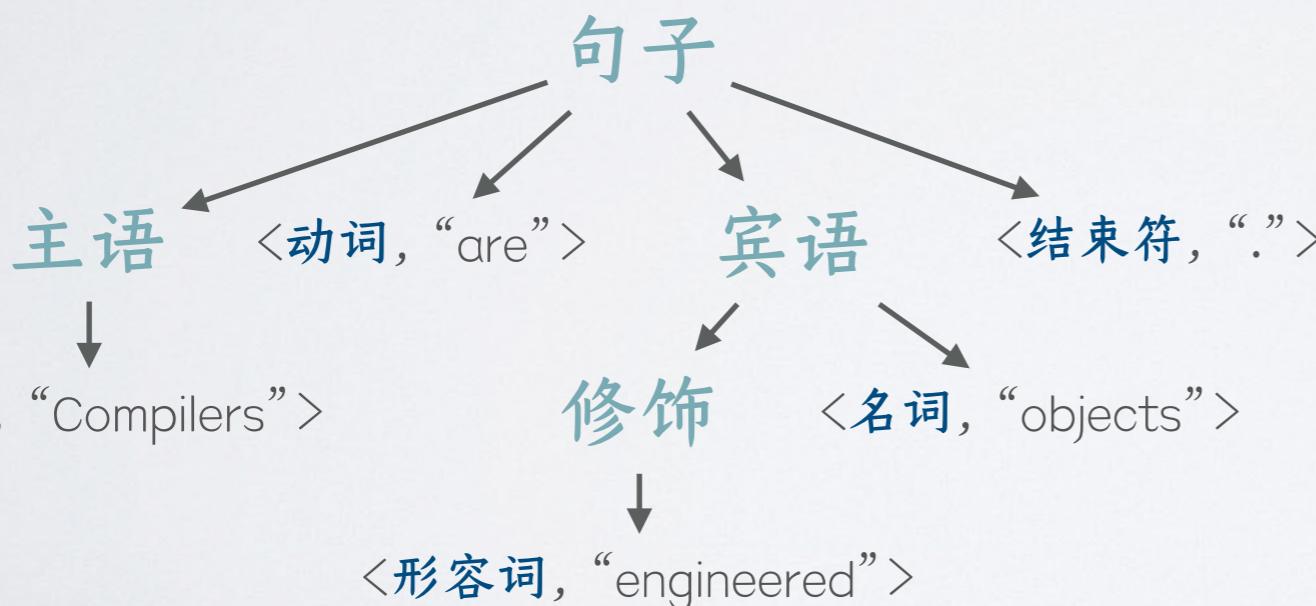
- 把「字母」组合成「单词」
- 从程序中提取的「单词」被称为 **词法单元**(token)



# 语法分析

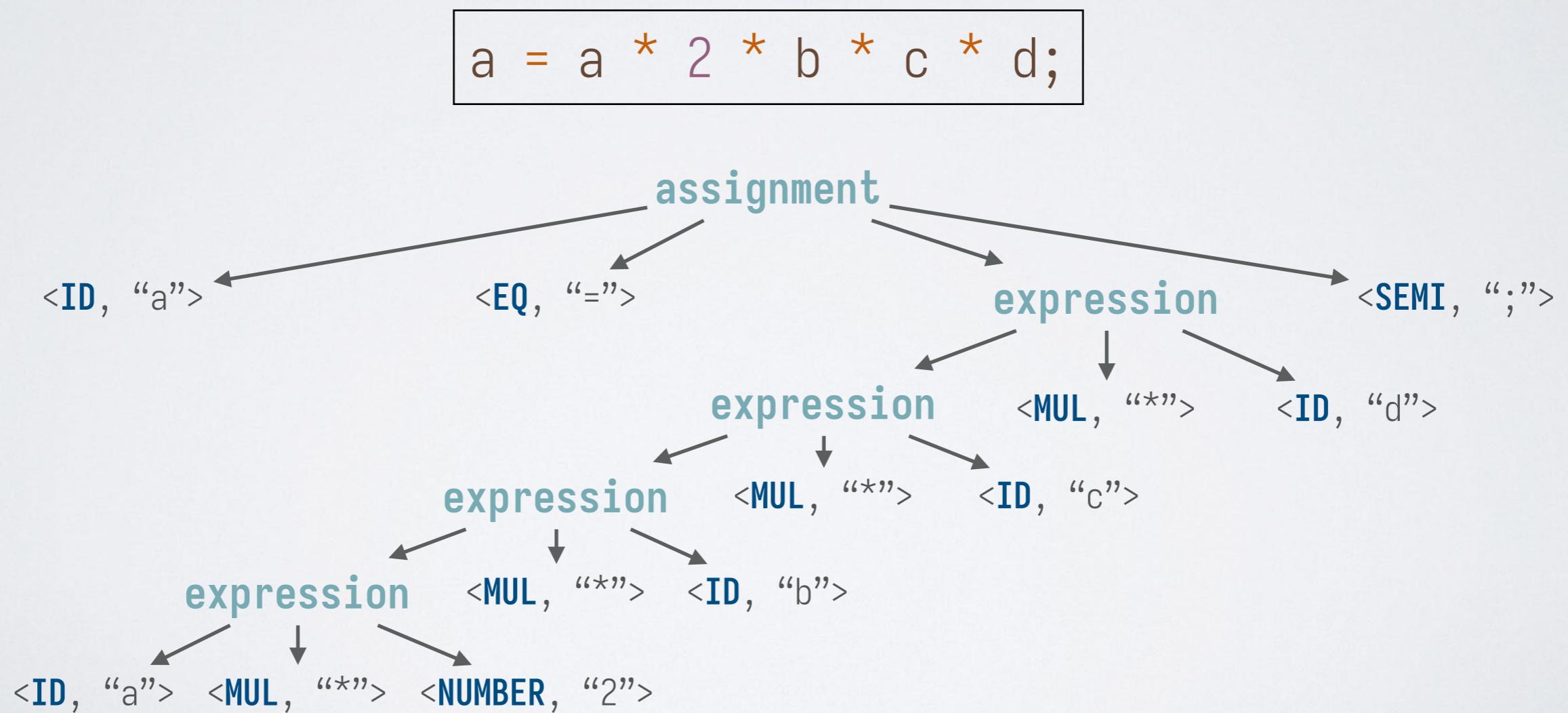
- 把「单词」组合成「句子」
- 组合的过程形成了**语法分析树**

Compilers are engineered objects.



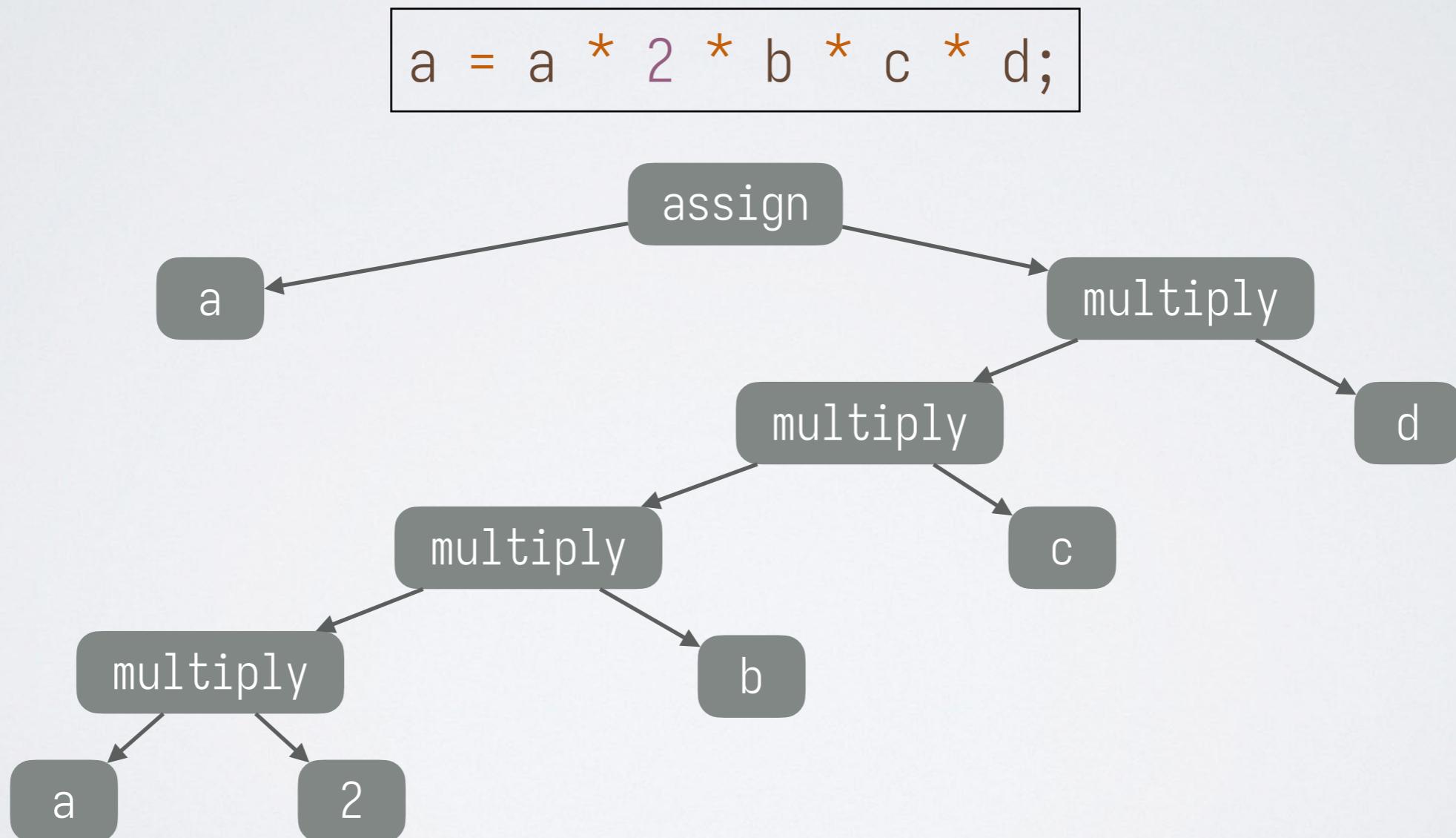
# 语法分析

- 把「单词」组合成「句子」
- 组合的过程形成了语法分析树



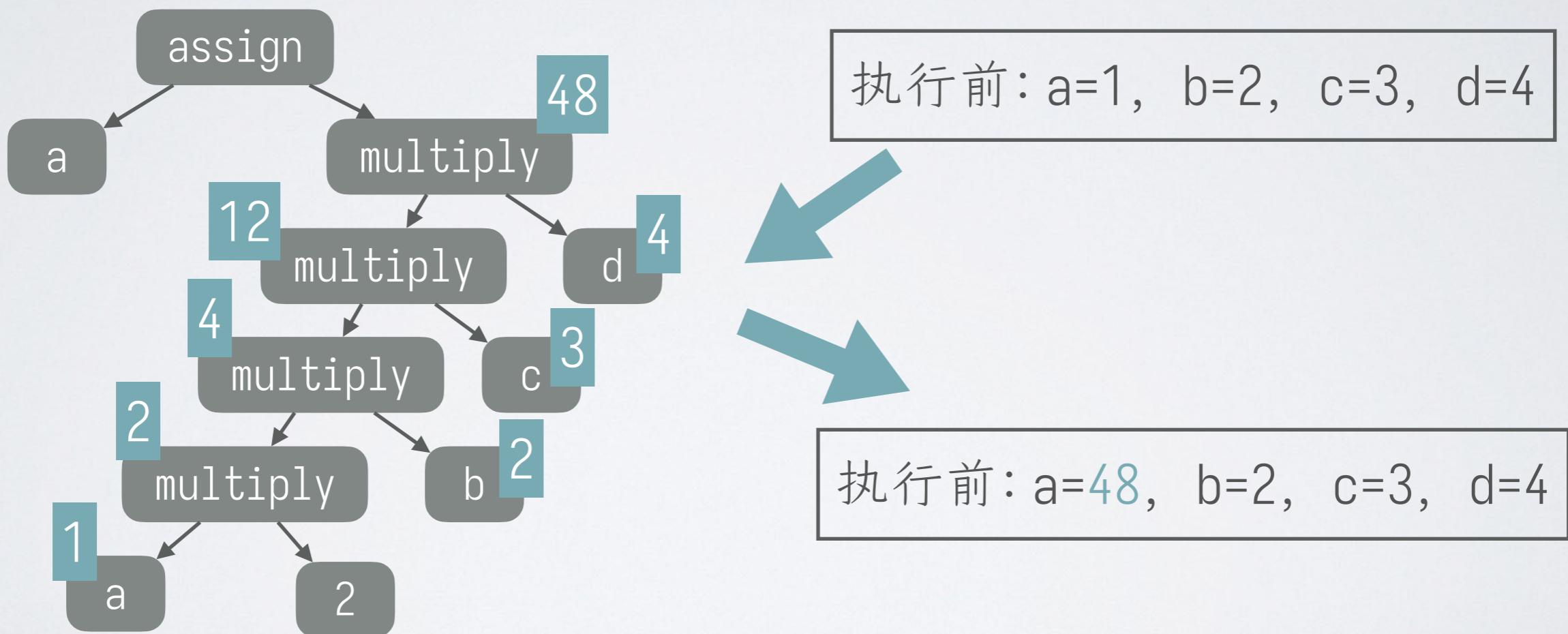
# 语义分析

- 语义分析可以为程序构造**抽象语法树**
  - 与语法分析树相比，记录了语义相关的核心信息



# 解释器

- 解释器可以在**抽象语法树**上进行解释执行
  - 解释器与编译器都需要的技术：词法分析，语法分析，语义分析





# 语义分析

- 并非所有满足语法的「句子」都有合理的「语义」

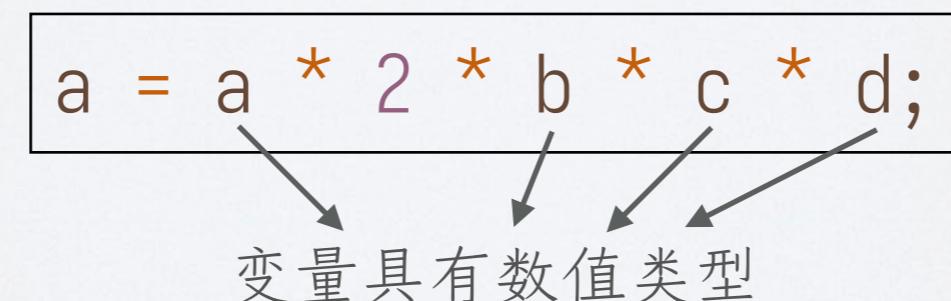
- ❖ “Rocks are green vegetables.”
  - ❖ “PKU eats THU.”

句子 → 主语 动词 宾语 结束符

- 语义分析检查程序是否合理

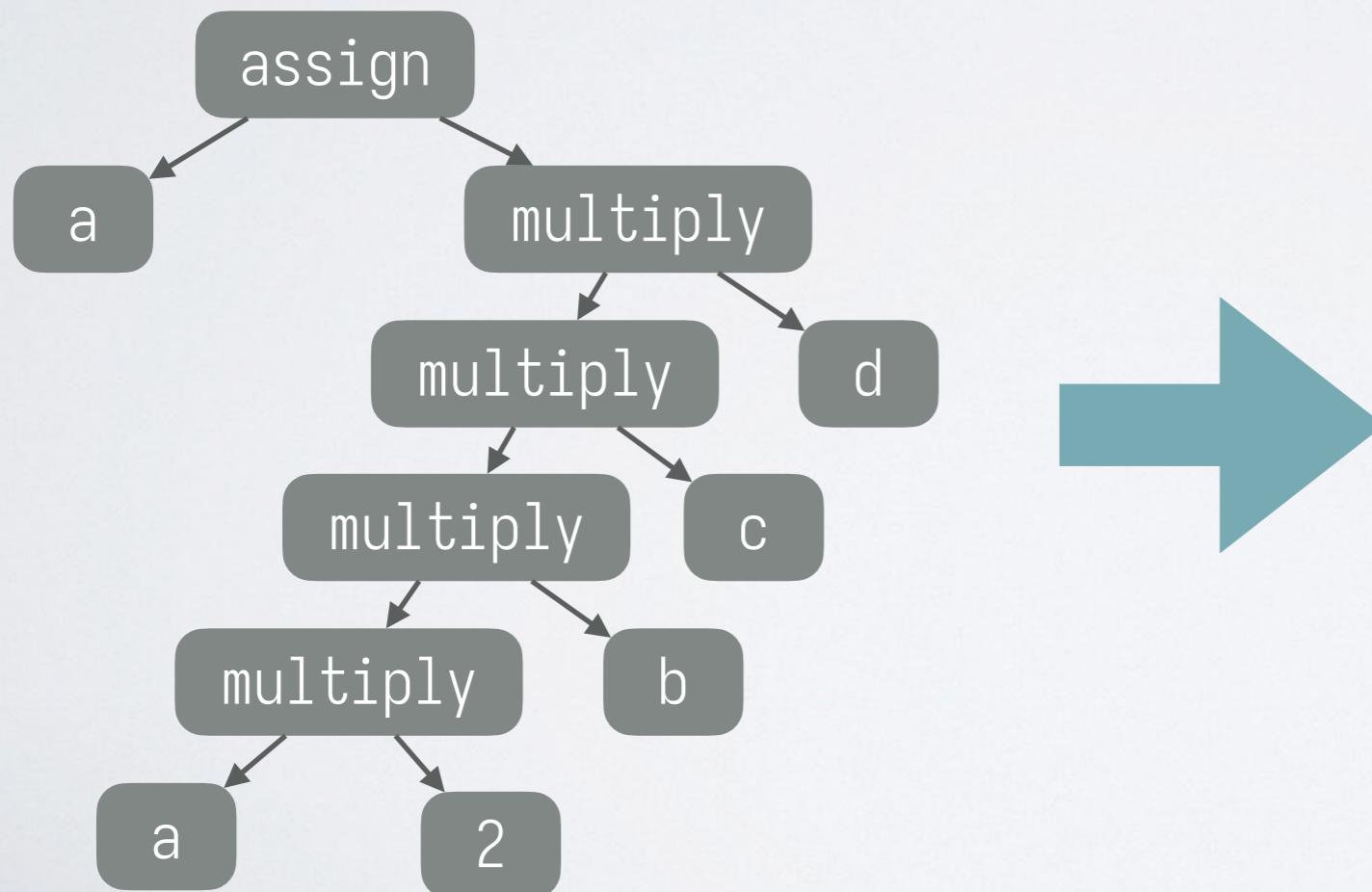
- ❖ 编程语言的语义模型比自然语言简单

- **类型检查：**检查程序中前后的类型使用是否一致



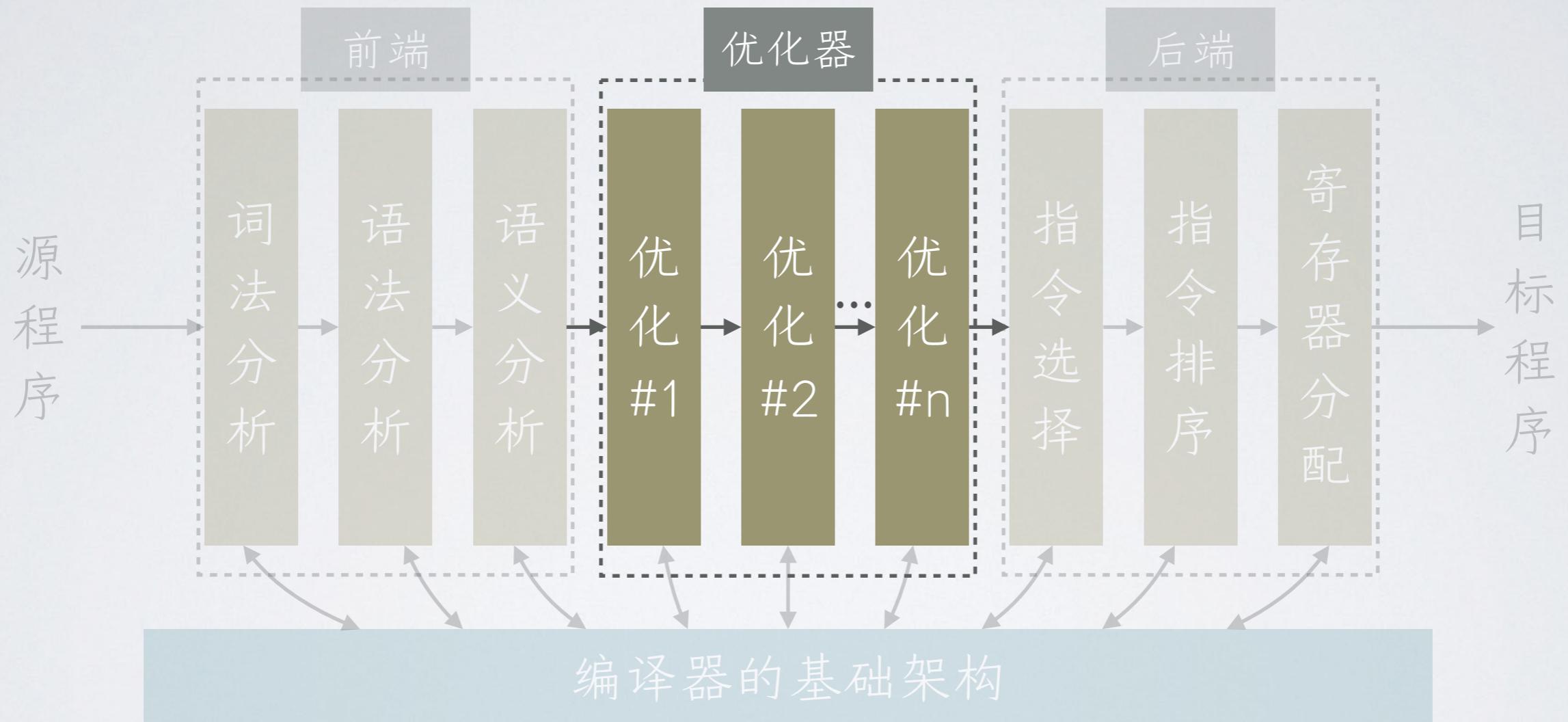
# 中间表示

- 编译器可以使用各种不同形式的中间表示(IR)
  - ❖ **图状 IR:** 用树、图等结构表示
    - ❖ 抽象语法树
  - ❖ **线性 IR:** 类似汇编的列表形式



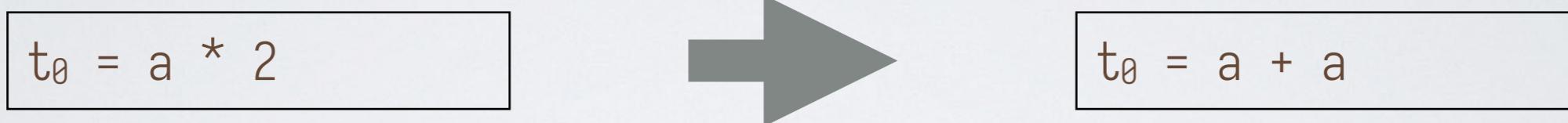
$t_0 = a * 2$   
 $t_1 = t_0 * b$   
 $t_2 = t_1 * c$   
 $t_3 = t_2 * d$   
 $a = t_3$

# 优化器

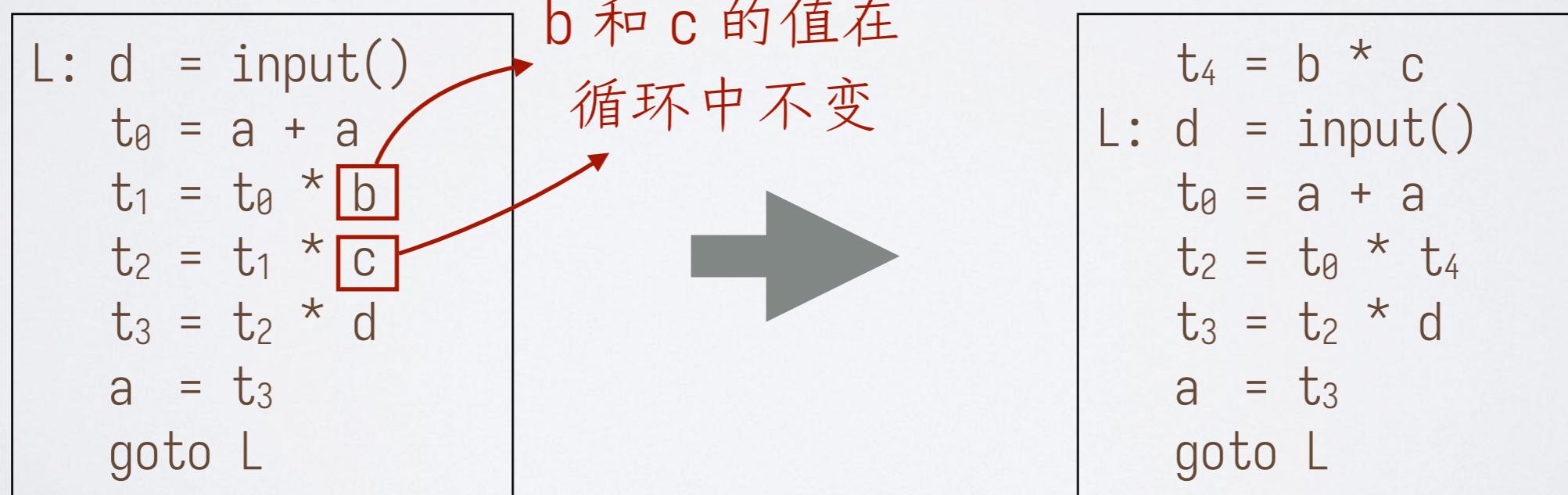


# 机器无关优化

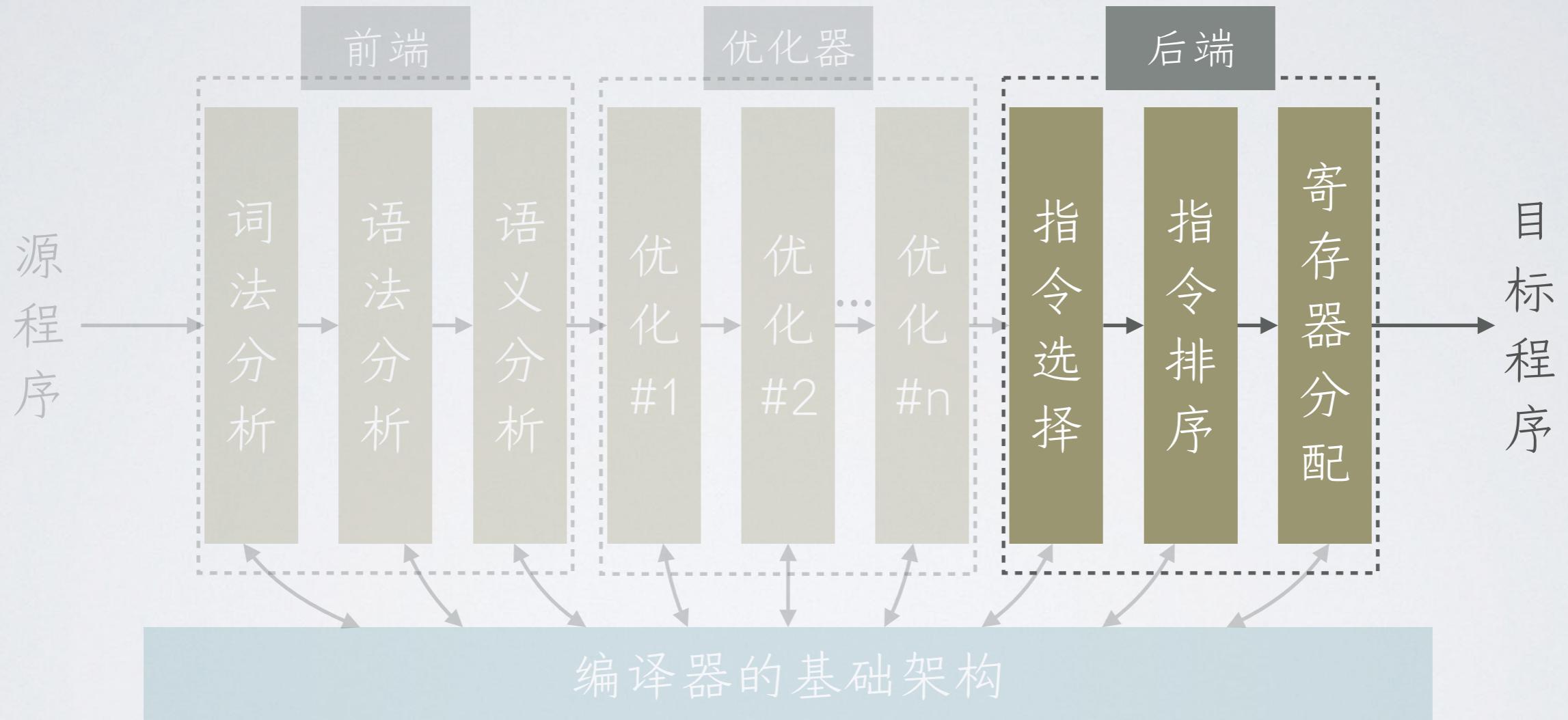
- 中间表示通常与具体目标机器无关，适合进行通用的优化
- 优化的目标通常是提高代码的执行效率



- 优化通常包括分析和转换两个部分



# 后端

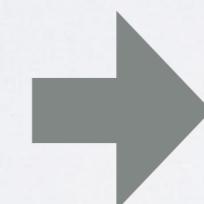




# 指令选择

- 把中间代码重写为目标机器上的指令
- 一条中间代码语句被转换为一个或多个目标机操作

```
t0 = a + a  
t1 = t0 * b  
t2 = t1 * c  
t3 = t2 * d  
a = t3
```



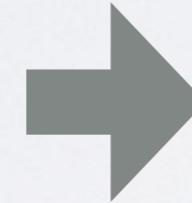
LD R1, a	// 加载变量 a 到寄存器 R1
LD R2, b	// 加载变量 b 到寄存器 R2
LD R3, c	// 加载变量 c 到寄存器 R3
LD R4, d	// 加载变量 d 到寄存器 R4
ADD R1, R1, R1	// R1 ← a * 2
MUL R1, R1, R2	// R1 ← (a * 2) * b
MUL R1, R1, R3	// R1 ← (a * 2 * b) * c
MUL R1, R1, R4	// R1 ← (a * 2 * b * c) * d
ST a, R1	// 保存寄存器 R1 到变量 a



# 寄存器分配

- 在指令选择期间，编译器通常不考虑寄存器数目有限的现实
- 寄存器分配算法需要决定在每个时刻，哪些值存于寄存器中

```
LD R1, a  
LD R2, b  
LD R3, c  
LD R4, d  
ADD R1, R1, R1  
MUL R1, R1, R2  
MUL R1, R1, R3  
MUL R1, R1, R4  
ST a, R1
```



```
LD R1, a          // 加载变量 a 到寄存器 R1  
ADD R1, R1, R1    // R1 ← a * 2  
LD R2, b          // 加载变量 b 到寄存器 R2  
MUL R1, R1, R2    // R1 ← (a * 2) * b  
LD R2, c          // 加载变量 c 到寄存器 R2  
MUL R1, R1, R2    // R1 ← (a * 2 * b) * c  
LD R2, d          // 加载变量 d 到寄存器 R2  
MUL R1, R1, R2    // R1 ← (a * 2 * b * c) * d  
ST a, R1          // 保存寄存器 R1 到变量 a
```

4 个寄存器

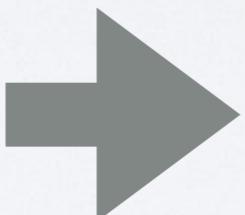
2 个寄存器

# 指令排序

- 指令排序是一种**机器有关优化**, 将目标机特性纳入考量
  - ❖ 不同操作的执行时间(时钟周期数)不同
  - ❖ 很多处理器支持在长延迟操作执行期间开始下一个操作
- 假设 LD、ST 要 3 个周期, MUL 要 2 个周期, ADD 要 1 个周期

```

LD R1, a          // 开始: 1, 结束: 3
ADD R1, R1, R1    // 开始: 4, 结束: 4
LD R2, b          // 开始: 5, 结束: 7
MUL R1, R1, R2    // 开始: 8, 结束: 9
LD R2, c          // 开始: 9, 结束: 11
MUL R1, R1, R2    // 开始: 12, 结束: 13
LD R2, d          // 开始: 13, 结束: 15
MUL R1, R1, R2    // 开始: 16, 结束: 17
ST a, R1          // 开始: 18, 结束: 20
    
```



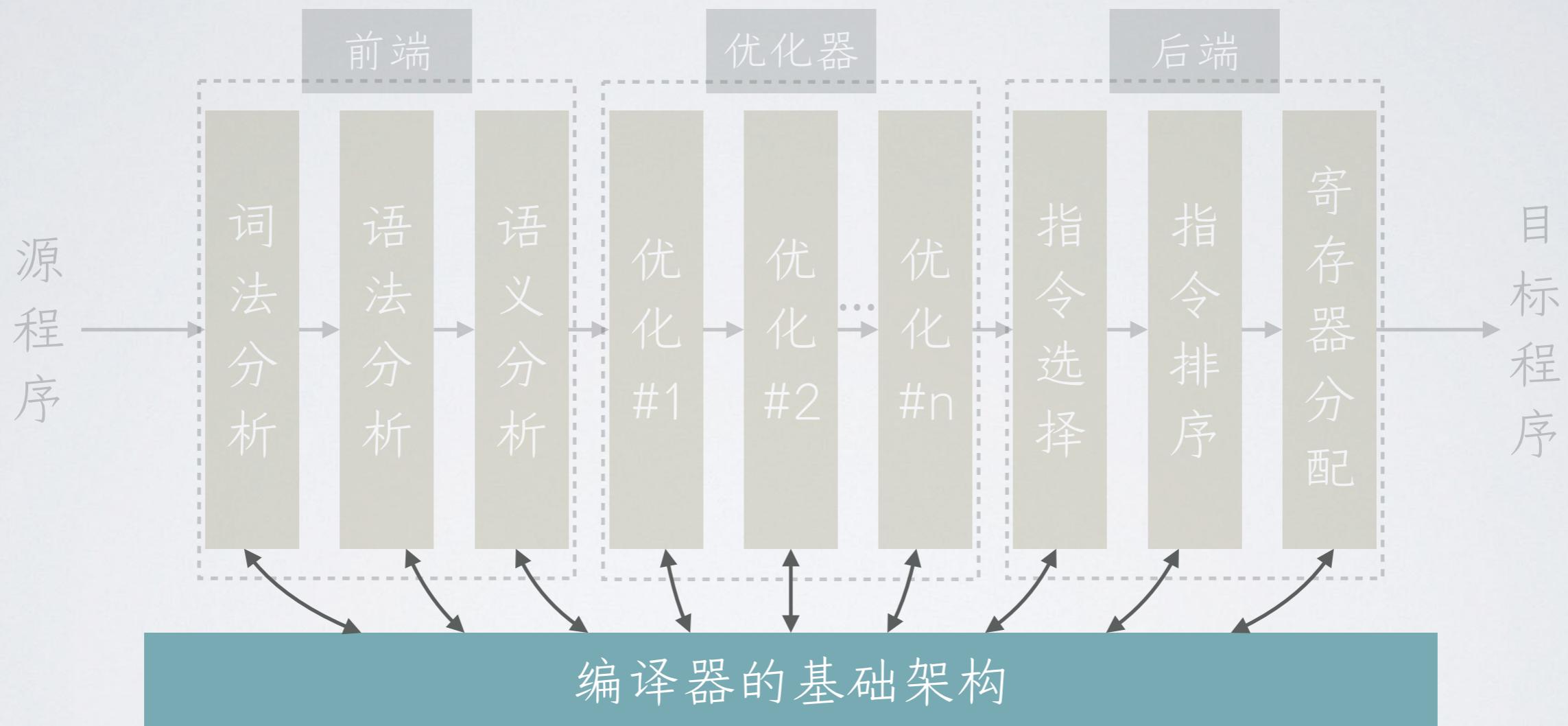
20 个时钟周期

```

LD R1, a          // 开始: 1, 结束: 3
LD R2, b          // 开始: 2, 结束: 4
LD R3, c          // 开始: 3, 结束: 5
ADD R1, R1, R1    // 开始: 4, 结束: 4
MUL R1, R1, R2    // 开始: 5, 结束: 6
LD R2, d          // 开始: 6, 结束: 8
MUL R1, R1, R3    // 开始: 7, 结束: 8
MUL R1, R1, R2    // 开始: 9, 结束: 10
ST a, R1          // 开始: 11, 结束: 13
    
```

13 个时钟周期

# 编译器的基础架构





# 符号表

- 编译器架构中最常见的一种数据结构
- 进行类型检查时，怎么知道每个变量的类型？
- 怎么记录不同的分析得出的关于程序的信息？
- 通过全局的符号表 (symbol table) 来汇总不同名字的信息

名字	种类	类型	其它信息
a	变量	int	
b	变量	int	
c	变量	int	
d	变量	int	
printf	函数		
.....			

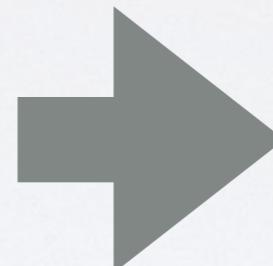


# 翻译过程示例

- 再来看从一个语言到另一个语言的翻译过程示例

文言 / wenyang-lang

吾有一術。名之曰「階乘」。  
欲行是術。必先得一數。曰「甲」。  
乃行是術曰。  
若「甲」等於一者。  
乃得「甲」。  
若非。  
減「甲」以一。名之曰「乙」。  
施「階乘」於「乙」。名之曰「丙」。  
乘「丙」以「甲」。名之曰「丁」。  
乃得「丁」  
也。  
是謂「階乘」之術也。  
  
施「階乘」於五。書之。



JavaScript

```
var 階乘 = _ => {}
階乘 = 甲 => {
  if (甲 == 1) {
    return 甲
  } else {
    const _ans1 = 甲 - 1
    var 乙 = _ans1
    const _ans2 = 階乘(乙)
    var 丙 = _ans2
    const _ans3 = 丙 * 甲
    var 丁 = _ans3
    return 丁
  }
}
const _ans4 = 階乘(5)
console.log(_ans4)
```

例子来源: <https://ide.wy-lang.org/?file=factorial>.



# 词法分析

## 句号视作空格

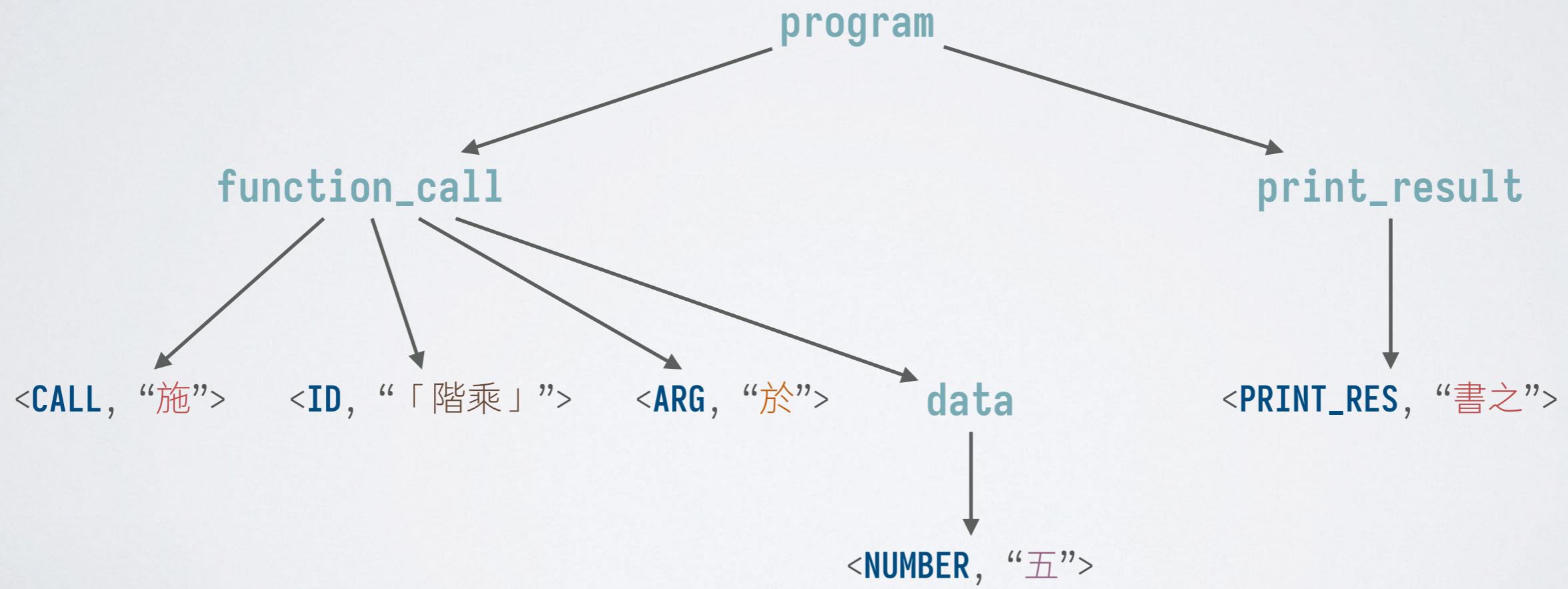
吾有一術。名之曰「階乘」。  
欲行是術。必先得一數。曰「甲」。  
乃行是術曰。  
若「甲」等於一者。  
乃得「甲」。  
若非。  
減「甲」以一。名之曰「乙」。  
施「階乘」於「乙」。名之曰「丙」。  
乘「丙」以「甲」。名之曰「丁」。  
乃得「丁」  
也。  
是謂「階乘」之術也。  
  
施「階乘」於五。書之。

<DEFN, “吾有”> <NUMBER, “—”> <FUNC, “術”>  
<NAME, “名之”> <IS, “曰”> <ID, “「階乘」”>  
<NEED, “欲行是術”> <PARAM, “必先得”> <NUMBER, “—”>  
<NUMBER\_TYPE, “數”> <IS, “曰”> <ID, “「甲」”>  
<BEGIN, “乃行是術曰”> <IF, “若”> <ID, “「甲」”>  
.....  
<END, “是謂”> <ID, “「階乘」”> <END\_DEFN, “之術也”>  
  
<CALL, “施”> <ID, “「階乘」”> <ARG, “於”>  
<NUMBER, “五”> <PRINT\_RES, “書之”>

# 语法分析

施「階乘」於五。書之。

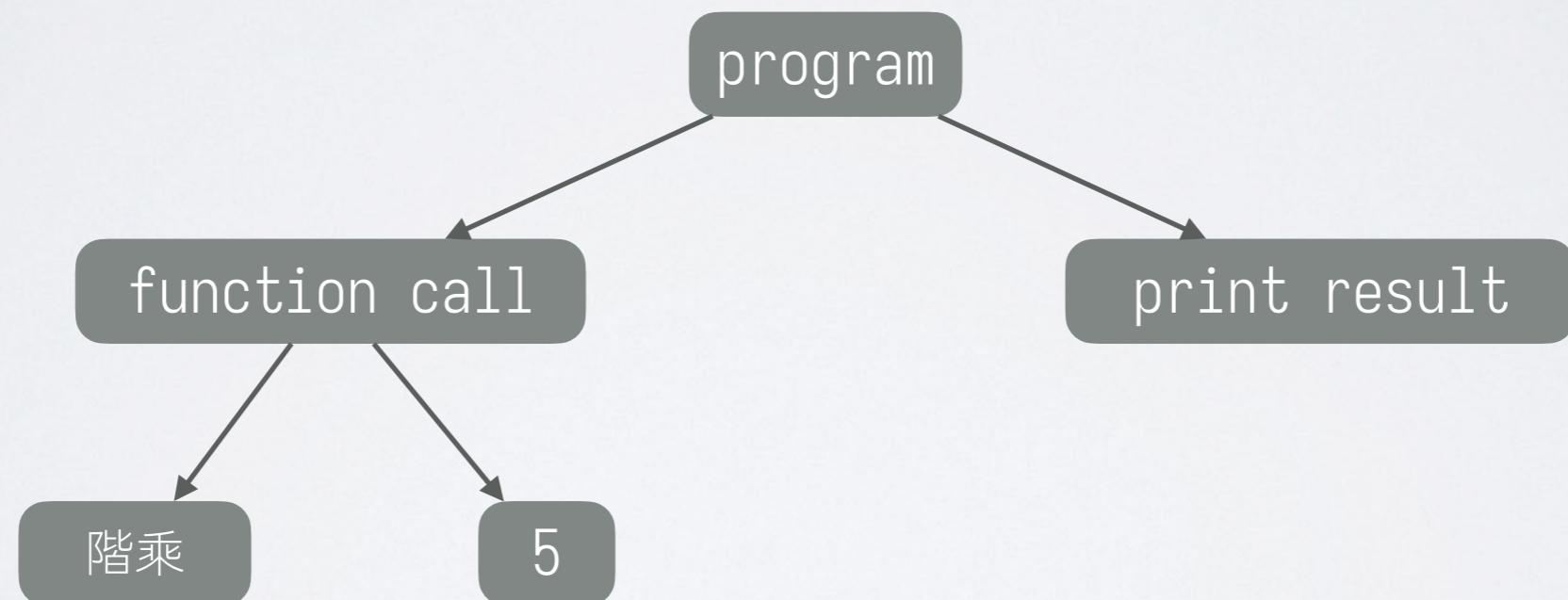
<CALL, “施”>    <ID, “「階乘」”>    <ARG, “於”>  
 <NUMBER, “五”>    <PRINT\_RES, “書之”>



# 语义分析

施「階乘」於五。書之。

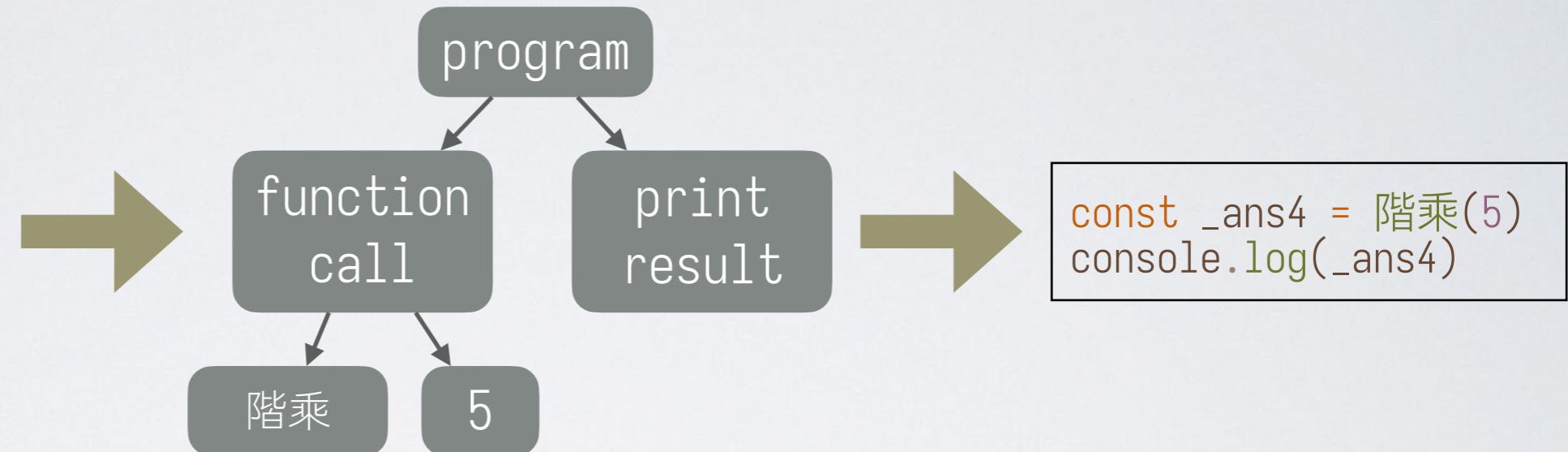
- 检查程序的合法性，如書之必须在计算了值的语句后面
- 生成抽象语法树



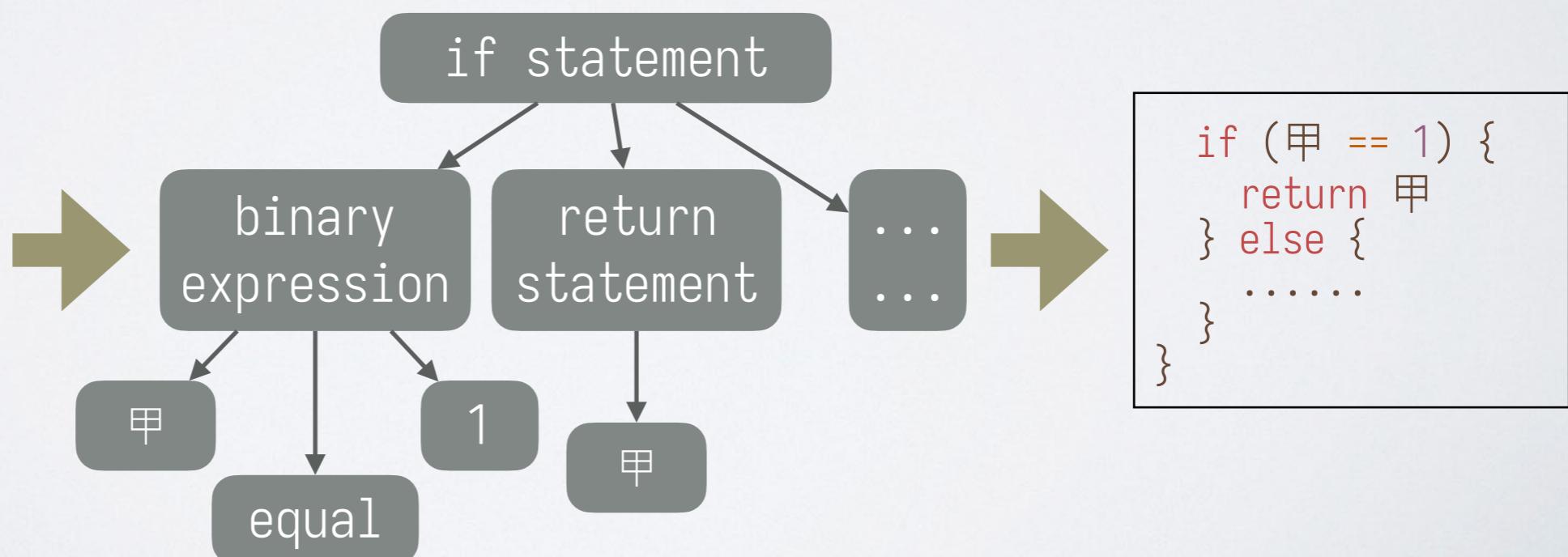
# 目标代码生成

- 因为是从文言到 JavaScript 的翻译，所以输出 JavaScript 代码

施「階乘」於五。書之。



若「甲」等於一者。  
乃得「甲」。  
若非。  
.....  
也。





# 本讲小结

- 编译是把源程序翻译为目标程序的过程
  - ❖ 编译器与解释器的区别和联系
- 编译器具有良好的结构，是很好的软件工程实践项目
  - ❖ 编译器的阶段划分
  - ❖ 前端、优化器、后端
  - ❖ 多趟处理
- 编译技术涉及了计算机科学不同领域的理论和算法
- 编译技术在多种领域有着广泛的应用