1 课程介绍

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆ 理解学习编译原理的意义与价值
- ⋆⋆ 了解编译器的整体工作流程
- ⋆⋆ 掌握运算符优先级解析算法

1.1 为什么学习编译原理？

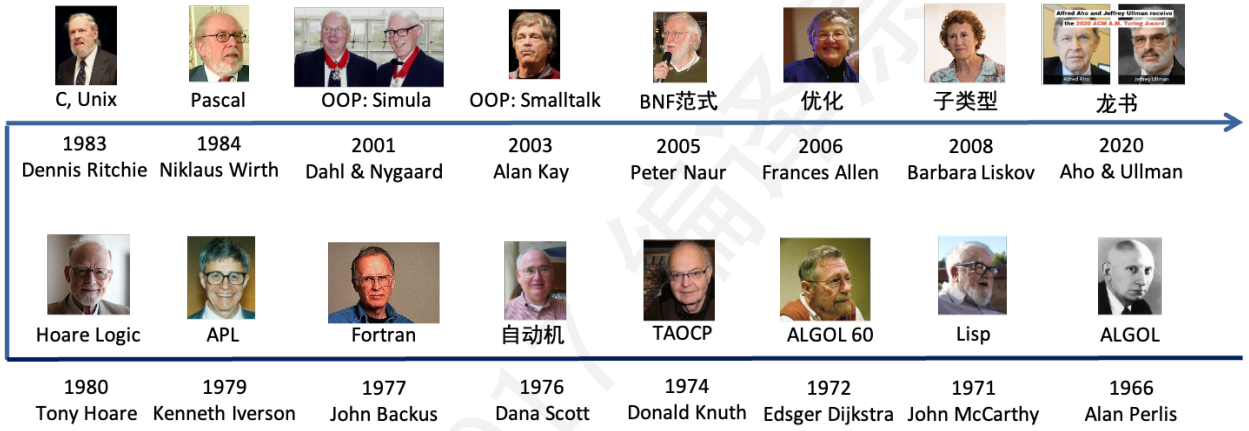

图 1.1: 主要成就与编程语言有关的图灵奖得主

编程语言与编译技术贯穿了计算机科学的发展历程，许多历届图灵奖得主的代表性工作都与该领域密切相关，如图 1.1 所示。其中，多位获奖者在早期参与了 ALGOL 语言及其编译器的设计，提出了具有奠基意义的理论、算法与技术，包括提出 BNF 范式的 John Backus 和 Peter Naur，人工智能的重要奠基者、Lisp 语言的主要设计者 John McCarthy，以及《The Art of Computer Programming》的作者 Donald Knuth 等。对这一领域感兴趣的同学可以访问 ACM 官方网站[1]，或阅读《图灵和 ACM 图灵奖》[1] 进一步查阅相关资料。

尽管编译原理是一门经典课程，在软硬件体系结构不断演进的背景下，编译技术至今仍然不可或缺。当现有工具无法满足新的需求时，我们往往需要自行开发新的“轮子”。例如，图灵奖得主 Leslie Lamport 为了获得一款高质量且易用的排版工具，设计并实现了 LaTeX；在操作系统领域，为了降低系统调用开销、提升性能，业界发展出了 eBPF 等关键技术；Mozilla 公司的程序员 Graydon Hoare 为构建安全、高效的浏览器引擎，设计了 Rust 编程语言。近年来，随着深度学习和大模型的快速发展，大量面向新计算范式的编程语言与编译技术不断涌现，典型代表包括 OpenAI 的 Triton，以及 NVIDIA 的 CUTLASS、cuTILE 等。此外，许多芯片厂商也需要开发自有编译器，以适配特定的指令集架构，并针对硬件特性进行深度优化。

[1] ACM 图灵奖得主：https://amturing.acm.org/byyear.cfm

1.2 初识编译：以计算器为例

计算器能够识别并计算算式，因此可以将其视为一种功能极其简化的编译器。本节将以实现一个简单计算器为例，介绍编译器的基本组成与实现思路。

1.2.1 功能需求

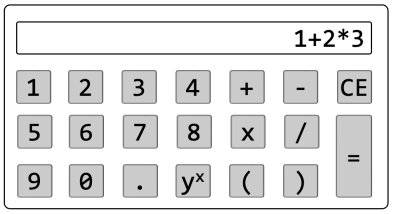

图 1.2: 目标计算器示例

假设目标计算器如图 1.2 所示，其主要功能需求如下：

- **操作数**：支持整数与小数。
- **运算符**：支持加、减、乘、除等四则运算，以及指数运算。
- **括号**：支持小括号，用于显式改变运算优先级。

1.2.2 实现思路

实现上述计算器通常需要经历以下几个基本阶段：

1) **词法分析**：对输入算式进行扫描，将操作数、运算符和括号等字符序列识别为词元（token）序列。

2) **句法解析**：根据运算符的优先级和结合性规则，对词元序列进行组织，构建语法解析树。

3) **解释执行**：遍历语法解析树并计算算式的最终结果。

词法分析：识别操作数和运算符

以算式 `123+456` 为例，词法分析过程需要按顺序识别操作数 `123`、运算符 `+` 以及操作数 `456`，并将其转换为词元序列：`<NUM(123)><ADD><NUM(456)>`。算法 1 描述了词元识别的基本思路。其关键在于使用一个缓冲区 `num` 来记录当前读取的数字字符。

算法 1 识别算式中的操作数和运算符

```
1: input: character stream
2: output: token stream
3: procedure TOKENIZE(charStream)
4:     let toks, num = ∅
5:     while ture do
6:         let cur = charStream.next();
7:         match cur :
8:             case '0'-'9' ⇒ num.append(cur); // insert at the beginning if num is empty
9:             case '+' ⇒ toks.add(num); toks.add(ADD); num.clear(); // add(num) do nothing if num is empty
10:            case '-' ⇒ toks.add(num); toks.add(SUB); num.clear();
11:            case '*' ⇒ toks.add(num); toks.add(MUL); num.clear();
12:            case '/' ⇒ toks.add(num); toks.add(DIV); num.clear();
13:            case '∧' ⇒ toks.add(num); toks.add(POW); num.clear();
14:            case '(' ⇒ toks.add(num); toks.add(LPAR); num.clear();
15:            case ')' ⇒ toks.add(num); toks.add(RPAR); num.clear();
16:            case _ ⇒ break; //EOF or an illegal character
17:        end match
18:    end while
19: end procedure
```

在此步骤中，我们暂不考虑算式的合法性问题（例如 `123++456`），因为判断合法性通常需要结合上下文信息，并建立相应规则。此外，在实际编译器中，词法分析阶段通常也不区分符号 `-` 是负号还是减号，这主要是因为判断是否为负号需要依赖上下文，例如仅当符号前面是运算符时，才能将 `-` 识别为负号。为便于后续讨论，本节暂不考虑负数和非法算式的情况。

句法分析：根据算式含义组织词元

算式解析是编译原理中一个经典问题。由于我们通常使用的算式表示形式是中缀（infix）表达式，因此在解析时必须遵循运算符的优先级和结合性规则。

- **优先级（precedence）**：指数运算符 > 乘除运算符 > 加减运算符。
- **结合性（associativity）**：加、减、乘、除运算符均为左结合；指数运算符为右结合。例如，`2^3^2` 应解析为 `2^(3^2)`，而不是 `(2^3)^2`。

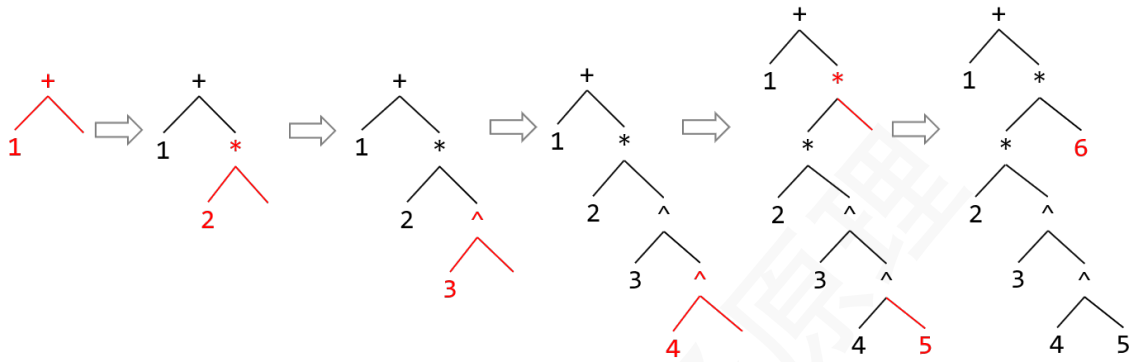

图 1.3: 算式 `1+2*3^4^5*6` 的解析过程

图 1.3 展示了算式 `1+2*3^4^5*6` 的解析过程。通过这个例子，我们可以梳理无括号算式的解析思路。解析结果形成的语法树为满二叉树：所有叶子节点表示操作数，非叶子节点表示运算符。每个运算符的计算顺序由语法树的后序遍历决定，从而体现了运算符的优先级和结合性。

该算式解析的基本思路是从左到右依次扫描，并使用栈记录已读取的运算符。解析过程中可归纳为以下三种情况：

- 当遇到左结合运算符，且其优先级高于栈顶运算符的优先级时，应将该运算符作为栈顶运算符的右子节点，此时栈顶运算符的左子节点已存在。
- 当遇到左结合运算符，且其优先级不高于栈顶运算符的优先级时，应将其作为栈顶运算符的父节点或更高祖先节点。具体做法是，从栈中依次弹出已读取的运算符，直到找到一个优先级低于当前运算符的运算符为止。
- 当遇到右结合运算符时，应将其作为栈顶运算符的右子节点。

Pratt 解析算法

Pratt 解析 [2] 是一种支持运算符优先级和结合性的解析算法。为了便于分析，该算法为每个运算符的左右两侧分别分配优先级数值，以同时反映运算符的优先级和结合性。对于左结合运算符，其左侧优先级低于右侧；而对于右结合运算符，其左侧优先级则高于右侧。以图 1.4 中的标注为例，运算符 “+” 和 “-” 的左右优先级分别为 1 和 2，运算符 “*” 和 “/” 的左右优先级分别为 3 和 4，而指数运算符 “^” 的左右优先级分别为 6 和 5。

算法 2 展示了 Pratt 算法的伪代码实现。该算法以词元序列作为输入，并根据运算符的优先级和结合性生成对应的语法解析树。假设初始优先级为 0，调用 `PrattParse` 函数即可从词元序列构建出图 1.3 所示的语法解析树。算法的核心思想是：从左到右依次读取词元，利用运算符的左右优先级决定运算符在解析树中的位置，从而正确体现算式的运算顺序。

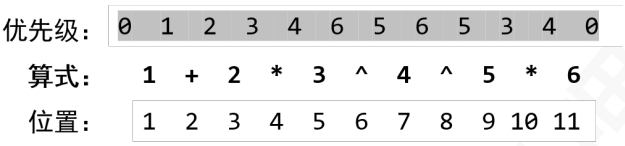

图 1.4: 算式 `1+2*3^4^5*6` 的运算符优先级标注

算法 2 Pratt 运算符优先级解析算法

```
输入: 词元序列 cur, 当前优先级 preced (初始为 0)
输出: 语法解析树（满二叉树）
初始化运算符优先级:
p[ADD] = (1,2), p[SUB] = (1,2), p[MUL] = (3,4), p[DIV] = (3,4), p[POW] = (6,5)
procedure PrattParse(cur, preced)
    l = cur.next();                          ▷ 获取当前词元并移动到下一个位置
    if l.type() ≠ TOK::NUM then
        return ERROR
    end if
    while true do                            ▷ 对应于从运算符栈中弹出运算符的操作
        op = cur.peek();                     ▷ 查看下一个词元，但不移动指针
        match op.type() :
            case TOK::NUM ⇒
                return ERROR
            case TOK::EOF ⇒
                return l
        end match
        (lp, rp) = p[op];
        if lp < preced then
            return l
        end if
        cur.next()
        r = PrattParse(cur, rp)
        l = CreateBinTree(op, l, r)
    end while
    return l
end procedure
```

以算式 `1+2*3^4^5*6` 为例，算法 2 的具体执行步骤及中间结果整理在表 1.1 中，可用于直观理解运算符优先级解析的过程。

表 1.1: 应用算法 2 解析算式 `1+2*3^4^5*6` 的步骤

cur	preced	l	op	lp	rp	操作
0	0	1	+	1	2	r = PrattParse(cur, rp); l = CreateBinTree(peek, l, r);
2	2	2	*	3	4	r = PrattParse(cur, rp); l = CreateBinTree(peek, l, r);
4	4	3	^	6	5	r = PrattParse(cur, rp); l = CreateBinTree(peek, l, r);
6	6	4	^	6	5	r = PrattParse(cur, rp); l = CreateBinTree(peek, l, r);
8	6	5	*	3	4	return l;
8	6	^(4,5)	*	3	4	return l;
8	4	^(3,^(4,5))	*	3	4	return l;
8	2	*(2,^(3,^(4,5)))	*	3	4	r = PrattParse(cur, rp); l = CreateBinTree(peek, l, r);
10	4	6	EOF	-	-	return l;
10	2	*(*(2,^(3,^(4,5)),6)	EOF	-	-	return l;
10	0	+(1,*(*(2,^(3,^(4,5)),6))	EOF	-	-	return l;

解释执行

基于语法解析树，我们可以通过后序遍历来计算算式的结果。对于计算器程序，另一种常用方法是将算式转换为逆波兰表达式（Reverse Polish Notation），它实际上就是语法解析树的后序遍历序列。例如，算式 `1+2*3^4^5*6` 的逆波兰表达式为：`1 2 3 4 5 ^ ^ * 6 * +`。

逆波兰表达式的计算过程非常直观：按顺序读取每个符号，若遇到操作数则将其压入栈中；若遇到运算符，则弹出栈顶的两个操作数进行计算，并将计算结果重新压入栈中。重复此过程直至读取完所有符号，此时栈顶元素即为算式的最终结果。

1.3 编译流程概览

与算式相比，编程语言的复杂度更高，因此真实的编译器也比计算器复杂得多。图 1.5 展示了编译的主要流程及相关技术分支。对于简单算式，由于复杂度较低，可以直接进行解释执行；而通用编程语言通常是图灵完备的，因此用其编写的程序需要通过通用图灵机来运行，这在实际应用中通常体现为虚拟机和实机两种方式。本学期后续课程将对这些过程和技术进行详细讲解。

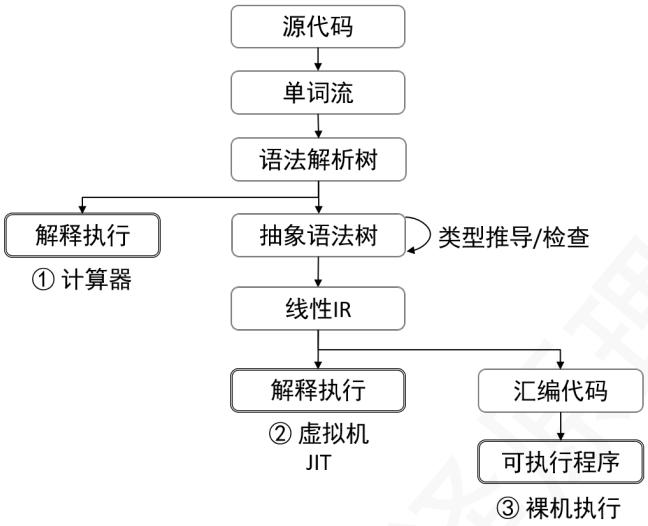

图 1.5: 编译流程和主要技术分支

练习

1) 如下图所示，在某些计算器中输入算式 `1+60%+60%` 后，计算结果为 `2.56`。请分析产生该结果的可能原因。

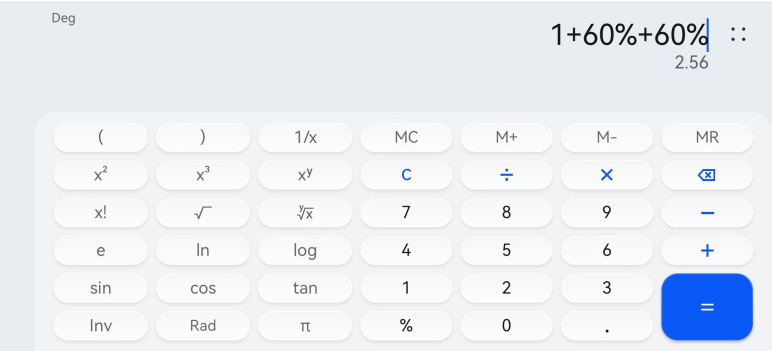

图 1.6: 某计算器软件界面

2) 实现 Pratt 算法并通过测试用例验证其正确性：

   a) 不考虑括号的情况；

   b) 支持小括号的情况。

3) 思考在你的学习、工作或实际问题中，哪些任务可以通过编译技术来解决？同时回顾日常使用的技术或工具，分析其中哪些与编译原理相关？

参考文献

[1] 吴鹤龄、崔林，《图灵和 ACM 图灵奖》（第 4 版），高等教育出版社，2012 年。

[2] Vaughan R. Pratt, “Top down operator precedence.” *In ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 1973.