编译授课笔记

徐辉

复旦大学　计算与智能创新学院

2026 年 5 月 7 日

说明

编译原理历来被认为是计算机专业课程中较难学习的一门，涉及复杂的文法理论，内容繁杂，而这些理论在实际编译器开发中并非总能直接派上用场。本教材的设计初衷是以“从零开始构建一个编译器”目标，重新梳理必要的知识点。针对课程内容的取舍，我们一方面弱化了语法解析的部分，主要因为这类技术已非常成熟；另一方面，增加了更多关于中间代码分析与优化的内容，这部分技术对现代编译器至关重要，且仍在不断发展。具体而言，本教材中的编译器实现主要面向 TeaLang：一门专为教学设计的语法简洁且时髦的语言。编译器的中间代码采用 LLVM IR，以实现与 LLVM 工具链的兼容性，而后端则主要面向 ARM 指令集架构。

本教材共包含 16 章：

- 第 1 至第 5 章：涵盖编译器前端，包括词法分析和句法分析，重点讲解 Tea 语言的语法规则设计及语法解析器的实现。其中，第 2 至第 3 章内容最为关键。
- 第 6 至第 10 章：涉及编译器的中间层，包括类型检查和优化，核心目标是将合法的 Tea 语言代码翻译为 LLVM IR。其中，第 6 至第 7 章为必要章节。
- 第 11 至第 13 章：编译器后端，涉及指令选择和寄存器分配等内容，重点讲解如何将 LLVM IR 翻译为 ARM 汇编代码并进行优化。

本教材适用于一个学期的课程安排，建议每周安排 3 节课进行讲授。各章的学习目标中，通过 $\star$ 标记知识点的重要性或实用性，最高级别为 $\star\star\star$。此外，本教材配套 1 个前端实验、3 个中间层实验、1 个后端实验，在理论学习的同时，强化实践能力培养。本教材区别于传统编译原理教材的核心特点在于内容简洁连贯，强调实用性。它的目标是在有限的篇幅和时间内，清晰呈现完整的编译流程，而非广泛对比不同方法。不仅降低了学习门槛，也更贴合实际编译器的开发路径和技术体系。

目录

Part I

前端

1 课程介绍

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？

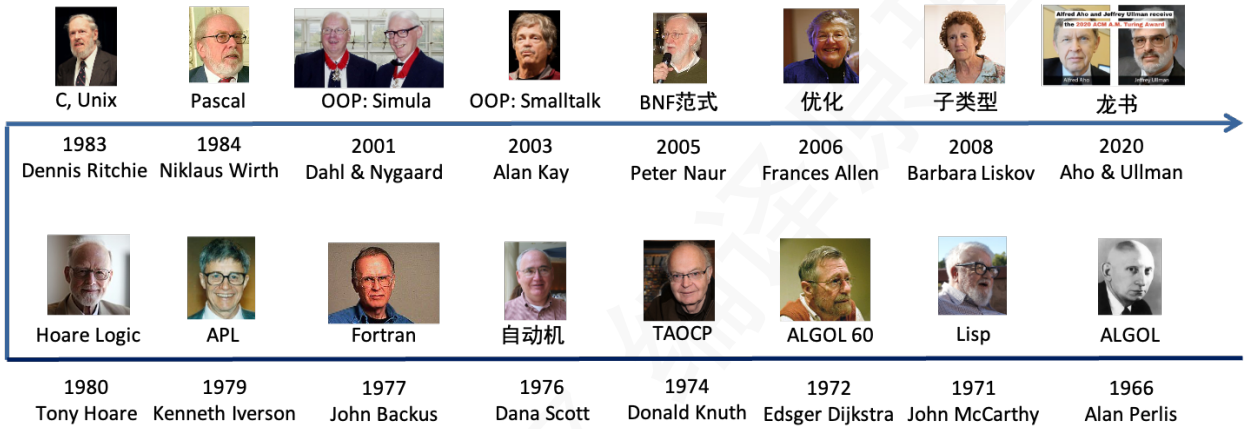


图 1.1: 主要成就与编程语言有关的图灵奖得主

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

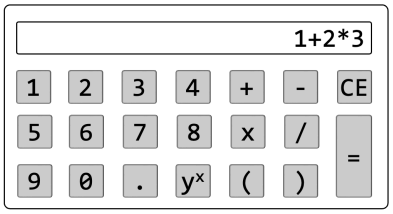


图 1.2: 目标计算器示例

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`。

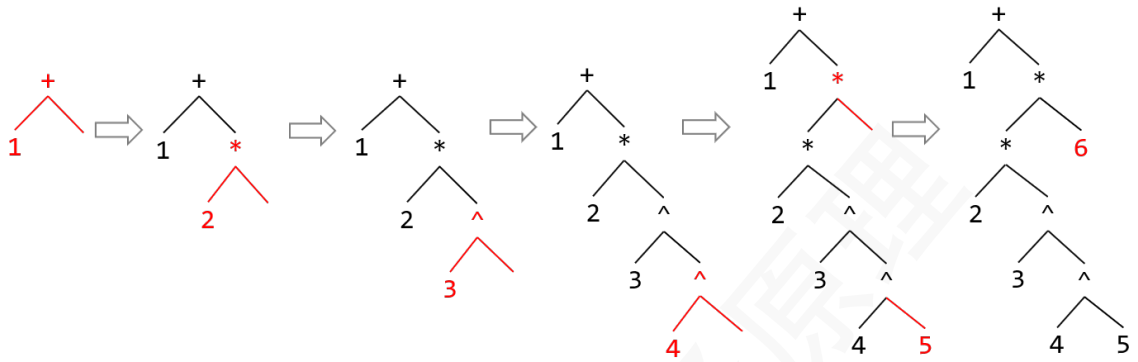


图 1.3: 算式 `1+2*3^4^5*6` 的解析过程

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。

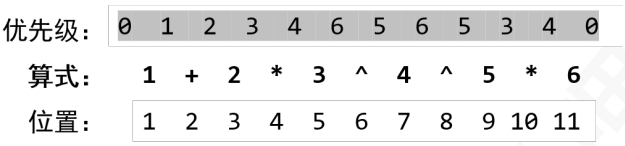


图 1.4: 算式 `1+2*3^4^5*6` 的运算符优先级标注

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。

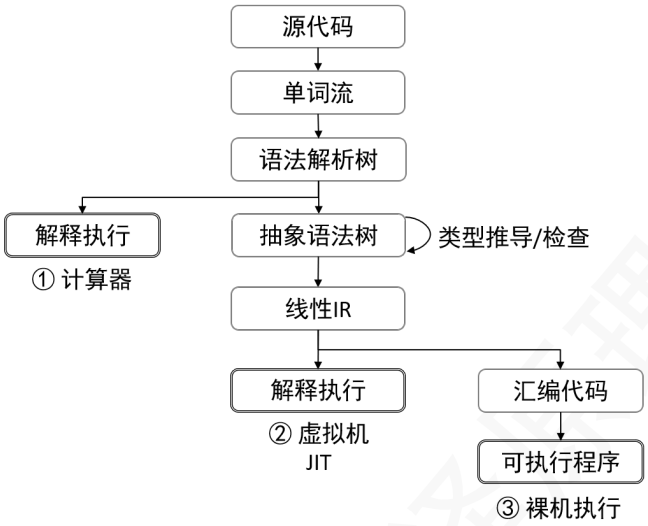


图 1.5: 编译流程和主要技术分支

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.

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。

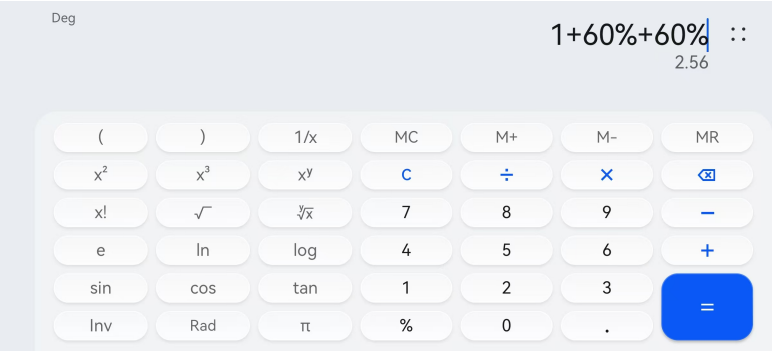


图 1.6: 某计算器软件界面

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？

2 词法分析

本章学习目标：

- ⋆⋆⋆ 掌握正则表达式的表示方法及其应用，包括字符表示、构造方式和常见扩展符号
- ⋆ 理解 Thompson 构造法及其实现原理，能够将正则表达式转换为 NFA
- ⋆ 理解子集构造法，能够将 NFA 转换为 DFA

2.1 词法声明：正则表达式

正则表达式（Regular Expression，简称 Regex）是一种形式化工具，用于描述字母表 Σ 上的字符串集合。它由基本字符元素及一系列构造规则组成，能够精确地定义文本模式，因此在词法分析、文本搜索以及其他各种字符串匹配任务中得到广泛应用。

2.1.1 字符表示

单个字符元素是正则表达式的最基本组成单位，用于目标字符串中的单个符号。表 2.1 列出了常用的单个字符元素的表示方法及其对应的含义。

表 2.1: 单个字符元素正则表示方法

字符元素	表述形式	含义
特定字符	a	$x = a$
字符范围	$[ab]$	$x \in \{a, b\}$
字符范围	$[a-z]$	$x \in \{a, ..., z\}$
字符范围	$[a-zA-Z]$	$x \in \{a, ..., z, A, ..., Z\}$
通配符	.	$x \in \Sigma$
排除特定字符	$\hat{}a$	$x \in \Sigma \backslash \{a\}$
空字符	ϵ	$x \in \emptyset$
特定字符或空	$a?$	$x = a$ or $x = \epsilon$

2.1.2 构造方式

字符元素可以通过选择（Union）、连接（Concatenation）和闭包（Kleene Closure）三种基本方式组合，形成更复杂的正则表达式。表 2.2列出了常用的构造方法及其优先级。为了保持一般性，这些构造方式通常采用递归定义，即复杂正则表达式可以由子正则表达式逐层组合而成。

表 2.2: 正则表达式构造方法，其中 S 和 T 为子正则表达式或单个字符

构造方式	符号	优先级	示例	含义
选择	|	1	$S\|T$	$x \in \{S \cup T\}$
连接		2	ST	$x \in \{st \mid s \in S, t \in T\}$
闭包	$*$	3	S^*	$x \in \{s_1..s_n \mid \forall 1 \le i \le n, s_i \in S, 0 \le n < \infty\}$
正闭包（扩展）	$+$	3	S^+	$x \in \{s_1..s_n \mid \forall 1 \le i \le n, s_i \in S, 1 \le n < \infty\}$
区间闭包（扩展）		3	$S^{\{min,max\}}$	$x \in \{s_1..s_n \mid \forall 1 \le i \le n, s_i \in S, min \le n \le max\}$

在解析正则表达式时，需要遵循一定的优先级规则：闭包 > 连接 > 选择。例如，对于正则表达式“a|bc*”，按照优先级进行解析后，其对应的字符串集合（亦称为正则集）为 {a, bc, bcc, ...}。需要注意的是，字符串“abc”并不属于该集合。在定义正则表达式时，还可以通过使用括号来改变运算的结合方式。例如，正则表达式“(a|b)c*”所表示的正则集为 {ac, bc, acc, bcc, ...}。

下面我们使用正则表达式来设计一个简单计算器的词法规则。首先，计算器的输入符号来自一个有限符号集：

$$\Sigma = \{0, 1, 3, 4, 5, 6, 7, 8, 9, \ . \ , +, -, *, /, \hat{}, (,)\}$$

表 2.3 给出了该计算器的词法规则定义。

表 2.3: 计算器词法定义

词元类型	含义	定义
<UNUM>	无符号数字	[0-9]$^+$(.[0-9]$^+$|ϵ)
<ADD>	加号	+
<SUB>	减号	−
<MUL>	乘号	*
<DIV>	除号	/
<EXP>	指数运算	^
<LPAR>	左括号	(
<RPAR>	右括号	)

需要注意的是，当一组正则表达式规则同时使用时，可能会产生二义性。具体而言，不同词元类型的规则之间可能存在交集，从而使某些字符串同时符合多种词元类型的定义。例如，在编程语言中，标识符与许多关键字之间往往存在交集。在这种情况下，字符串“iffy”（本应被解析为一个标识符）可能会被错误地识别为关键字 if 与标识符 fy 的组合。

在实际的词法分析工具中，通常可以通过以下两种方式解决二义性问题：

- 通过约定规则顺序消除二义性，即优先匹配先定义的规则；
- 通过为关键字设置边界约束来排除非关键字的情况，例如规定关键字之后只能出现特定的分隔符（如空格、换行或括号等）。

此外，在使用正则表达式扫描字符串时默认采用最长匹配原则：若已匹配到某个词元且仍可继续向前匹配，则应尽可能匹配更长的字符串。例如，字符串“iffy”应被识别为一个标识符，而不是四个标识符。

2.2 词法解析：有穷自动机及其构造

本节将解决一个重要问题：给定一组由正则表达式定义的词元规则，如何自动生成相应的词元识别程序。从理论上讲，所有正则表达式都可以用确定性有穷自动机（DFA：Deterministic Finite Automaton）表示，而 DFA 又可以进一步转换为等价的程序实现。

定义 1 (有穷自动机（FA: Finite Automaton）)**.** 有穷自动机是一个五元组：$(S, s_0, T, \Sigma, \Delta)$，其中：

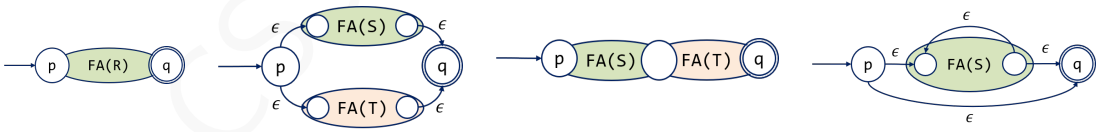
- S 是有限状态集合；
- $s_0 \in S$ 是初始状态；
- $T \subseteq S$ 是结束状态集合；
- Σ 是有限字符集合；
- $\Delta \subseteq S \times \Sigma \times S$ 是边的集合，表示状态转移关系，其中每个转移对应一个输入字符和两个状态。若对于任意状态 $s_i \in S$ 和输入字符，至多存在一个转移目标状态，则该有穷自动机称为确定性有穷自动机；否则称为非确定性有穷自动机（NFA: Non-Deterministic Finite Automaton）。

将一组正则表达式转换为 DFA 的过程大致可以分为以下三个步骤：

1) 根据正则表达式构造 NFA；
2) 将 NFA 转换为 DFA；
3) 对 DFA 进行优化。

2.2.1 根据 Regex 构造 NFA

本节介绍一种经典的 NFA 构造算法：McNaughton–Yamada–Thompson 构造法（简称 Thompson 构造法）[1, 3]。该方法的基本思想是：针对正则表达式中的三种基本构造方式，分别设计相应的 NFA 构造规则。具体而言，从初始 NFA 出发，按照运算符的优先顺序（逆序）递归展开正则表达式，并根据当前子表达式的构造方式选择相应的 NFA 构造规则进行组合，从而最终得到与该正则表达式等价的 NFA。



(a) 初始 FA　(b) 展开选择：$S|T$　(c) 展开连接：ST　(d) 展开闭包：S^*

图 2.1: Thompson 构造法

图 2.1 展示了初始 NFA 以及表 2.2 中三种主要正则构造方式对应的 NFA 构造方法。从图 2.1a 中的初始 NFA 出发，该 NFA 的边表示目标正则表达式 R。若 R 的最外层构造为选择 $S|T$，则按照图 2.1b 展开；若为连接 ST，则根据图 2.1c 展开；若为闭包 S^*，则采用图 2.1d 的方式展开。

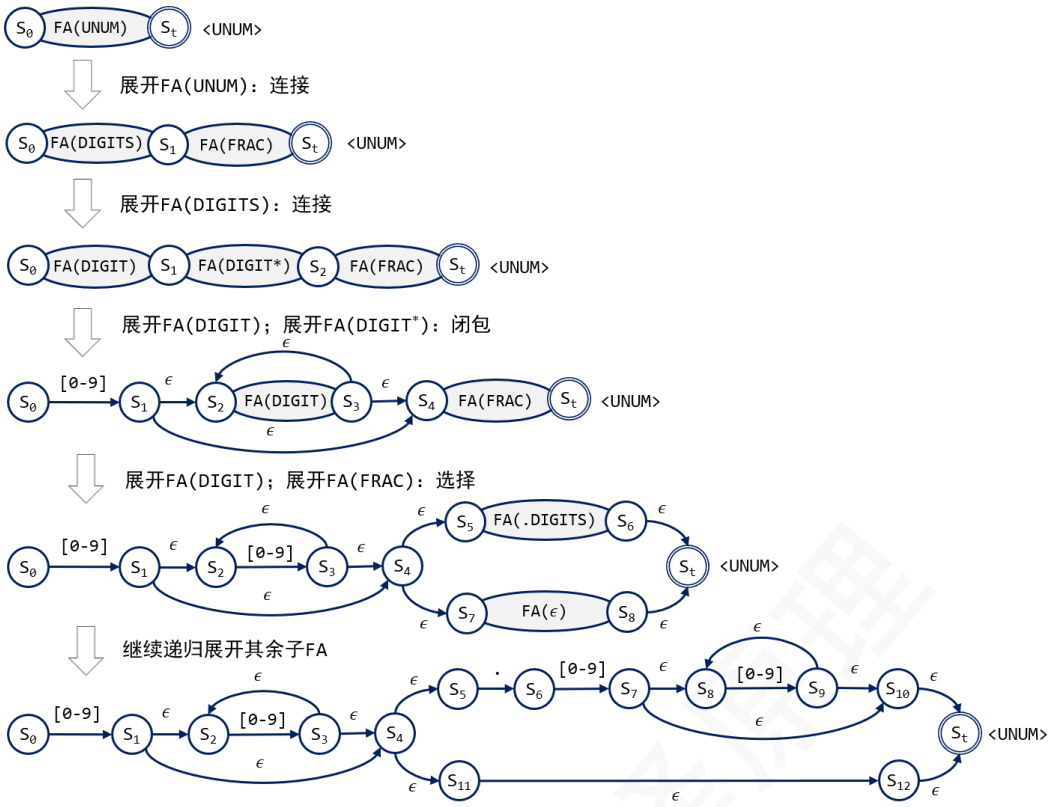


图 2.2: 应用 Thompson 构造法将词元 `<UNUM>` 对应的正则表达式转换为 NFA

图 2.2 以表 2.3 中 `<UNUM>` 词元对应的正则表达式为例，详细展示了该递归构造过程。在此之前，需要将该表达式中的正闭包改写为普通闭包形式：`[0-9][0-9]*(.[0-9][0-9]*|ϵ)`。

在构造出每种词元类型对应的 NFA 后，可以进一步通过 ϵ 转移将它们合并为一个整体 NFA。图 2.3a 展示了表 2.3 中所有词元类型对应的 NFA。

2.2.2 将 NFA 转换为 DFA

所有 NFA 都可以转换为 DFA。本节将介绍一种基于子集构造法（powerset）的 DFA 构建方法。在介绍具体方法之前，我们首先定义两个基本概念：ϵ 闭包（ϵ-closure）和 α 转移（α-transition）。

对于 NFA 中的单个状态 s_i，其 ϵ 闭包指通过 ϵ 转移能够到达的所有状态的集合：

$$Cl^{\epsilon}(s_i) = \{s_j \mid (s_i, \epsilon) \rightarrow^* (s_j, \epsilon)\}$$

对于 NFA 中的状态集合 S，其 ϵ 闭包定义为 S 中所有状态的 ϵ 闭包的并集：

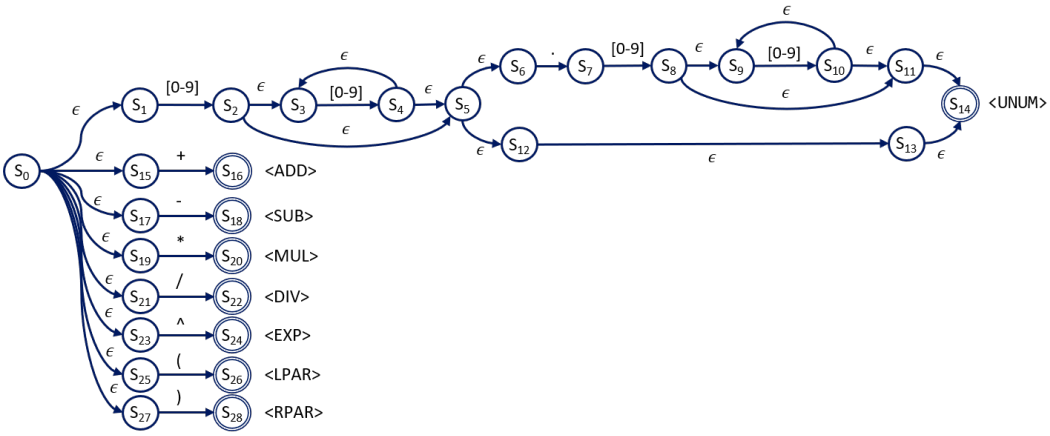
$$Cl^{\epsilon}(S) = \{q' \mid \forall q \in S, (q, \epsilon) \rightarrow^* (q', \epsilon)\}$$

对于状态集合 S 及输入字符 α，其 α 转移指：集合 S 在读取字符 α 后，所有可达状态的 ϵ 闭包的并集：

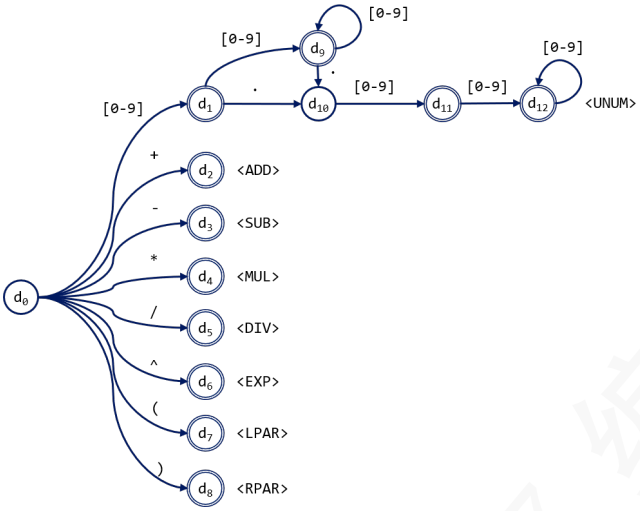
$$\Delta(S, \alpha) = Cl^{\epsilon}(\{q' \mid \forall q \in S, (q, a) \rightarrow q'\})$$

基于上述概念，我们可以定义 NFA 到 DFA 的构造方法

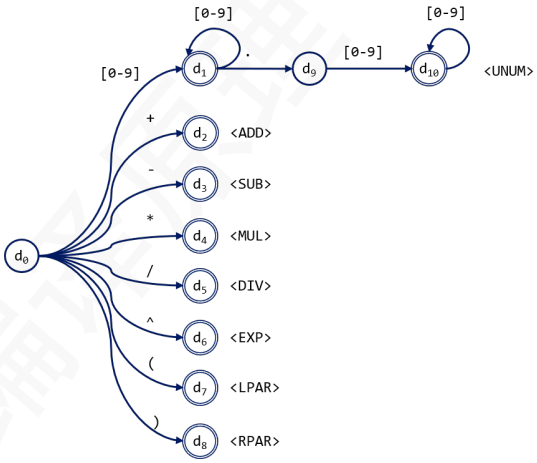
定义 2 (NFA→DFA)**.** 给定一个 NFA $\{N, n_0, N_f, \Sigma, \Delta\}$，其对应的 DFA 可表示为 $\{D, d_0, D_f, \Sigma, \Theta\}$，其中：



(a) 合并所有词元后的 NFA



(b) NFA 转换后的 DFA



(c) 优化后的 DFA

图 2.3: NFA 转换 DFA

- D 为 DFA 的状态集合，每个状态 d_i 是若干 NFA 状态的集合；
- d_0 为 DFA 的初始状态，且 d_0 为 n_0 的 ϵ 闭包，即 $d_0 = Cl^{\epsilon}(n_0)$；
- D_f 为 DFA 的结束状态集合，且每个结束状态 $d_f \in D_f$ 都满足 $d_f \cap N_f \neq \emptyset$；
- Θ 为 DFA 的状态转移集合，对应 NFA 中状态集合 S 的 α 转移：$\Theta = \{(S, a, \Delta(S, \alpha)), \alpha \in \Sigma\}$。

算法 3 描述了将 NFA 转换为 DFA 的具体步骤。首先，将初始状态 d_0 放入工作列表 $worklist$。然后，对于 d_0 的每个输入字符 α，计算其 α 转移，得到一组新的状态集合，并将这些状态加入 $worklist$。接下来，对 $worklist$ 中新加入的 DFA 状态重复上述过程，直到不再产生新的状态为止。

算法 3 NFA 转换为 DFA

```
1: procedure NFAtoDFA({N, n_0, N_f, Σ, Δ})
2:     let d_0 = Cl^ϵ(n_0)
3:     let D = {d_0}
4:     let worklist = {d_0}
5:     while worklist ≠ NULL do
6:         d = worklist.pop()
7:         for each α ∈ Σ do:
8:             t = Δ(d, α)
9:             if !D.find(t) then:
10:                worklist.add(t)
11:                D.add(t)
12:            end if
13:        end for
14:    end while
15: end procedure
```

应用算法 3 ，我们可以将图 2.3a 中的 NFA 转换为等价的 DFA。表 2.4 给出了具体的计算过程，最终得到的 DFA 如图 2.3b 所示。从图中可以看出，该 DFA 中仍存在一些冗余的 ϵ 转移，前后状态之间可以合并，因此该 DFA 可以进一步优化。

表 2.4: 应用子集构造法将图 2.3a 中的 NFA 转换为 DFA

DFA 状态	**NFA 状态集合**	**0-9**	**.**	+	**-**	*****	/	^	(	)
d_0	$\{s_0, s_1, s_{15}, s_{17}, s_{19}, s_{21}, s_{23}, s_{25}, s_{27}\}$	d_1	-	d_2	d_3	d_4	d_5	d_6	d_7	d_8
d_1	$\{s_2, s_3, s_5, s_6, s_{12}, s_{13}, s_{14}\}$	d_9	d_{10}	-	-	-	-	-	-	-
d_2	$\{s_{16}\}$	-	-	-	-	-	-	-	-	-
d_3	$\{s_{18}\}$	-	-	-	-	-	-	-	-	-
d_4	$\{s_{20}\}$	-	-	-	-	-	-	-	-	-
d_5	$\{s_{22}\}$	-	-	-	-	-	-	-	-	-
d_6	$\{s_{24}\}$	-	-	-	-	-	-	-	-	-
d_7	$\{s_{26}\}$	-	-	-	-	-	-	-	-	-
d_8	$\{s_{28}\}$	-	-	-	-	-	-	-	-	-
d_9	$\{s_3, s_4, s_5, s_6, s_{12}, s_{13}, s_{14}\}$	d_9	d_{10}	-	-	-	-	-	-	-
d_{10}	$\{s_7\}$	d_{11}	-	-	-	-	-	-	-	-
d_{11}	$\{s_8, s_9, s_{11}, s_{14}\}$	d_{12}	-	-	-	-	-	-	-	-
d_{12}	$\{s_9, s_{10}, s_{11}, s_{14}\}$	d_{12}	-	-	-	-	-	-	-	-

2.2.3 优化 DFA

DFA 优化的核心思想是合并可以合并的状态节点。例如，对于两个状态节点 d_i 和 d_j，如果满足以下条件，则可以将它们合并：

$$\forall \alpha \in \Sigma, \Theta(d_i, \alpha) = \Theta(d_j, \alpha)$$

基于这一思想，最经典的方法是 Hopcroft 分割算法 [4]，其核心在于利用状态集合的划分来实现 DFA 的最小化。如算法 4 所示，具体步骤如下：

1) 将 DFA 的状态集合 D 初始化为两个子集：结束状态 D_f 和其他状态 $D \backslash D_f$；

2) 对每个子集进行检查，判断是否需要进一步划分：即是否存在某个输入字符，使得子集内的状态转移指向不同的子集；

3) 重复上述过程，直到无法再进行划分为止，从而得到最优 DFA。

算法 4 Hopcroft 分割算法（DFA 最小化）

procedure OptDFA($\{D, d_0, D_f, \Sigma, \Theta\}$)
  let $R = \{D_f, D \backslash D_f\}$ ▷ 将状态集合划分为结束状态与非结束状态
  let $S = \{\}$ ▷ 初始化上一轮划分集合为空
  while $S \neq R$ **do**
    $S \leftarrow R$ ▷ 保存当前划分作为上一轮划分
    $R \leftarrow \{\}$ ▷ 准备存放新的划分结果
    for each $s_i \in S$ **do**:
      $R \leftarrow R \cup$ Split(s_i) ▷ 调用 Split 函数，对子集 s_i 根据输入字符划分状态
    end for
  end while
end procedure
procedure Split(S)
  for each $\alpha \in \Sigma$ **do**
    if α splits S into $\{s_1, s_2\}$ **then**
      return $\{s_1, s_2\}$ ▷ 如果输入字符 α 能将 S 划分为两个子集，返回划分结果
    end if
  end for
  return $\{S\}$ ▷ 如果没有划分，返回原集合
end procedure

图 2.3c 展示了对图 2.3b 优化后的最终 DFA。除了上述“先构造 NFA，再转换为 DFA 并优化”的方法外，还有一种直接构造最优 DFA 的方法，即 Brzozowski 算法 [2]。有兴趣的同学可以查阅相关资料，进一步了解其原理和实现。

本章介绍的正则表达式解析技术已经相当成熟，并被广泛应用于实际工程中。许多强大且易用的工具，如传统的 C 语言工具 Flex[1]、Python 的 `re` 库[2]，以及 Rust 的 pest[3] 和 lalrpop[4]，可以直接用于处理各种词法分析任务，从而大幅提升相关软件原型的开发效率。

参考文献

[1] Robert McNaughton, and Hisao Yamada. “Regular expressions and state graphs for automata.” *IRE Transactions on Electronic Computers*, 1960.

[2] Janusz A. Brzozowski, “Canonical regular expressions and minimal state graphs for definite events.” *In Symposium of Mathematical Theory of Automata*, 1962.

[3] Ken Thompson. “Programming techniques: Regular expression search algorithm.” *Communications of the ACM*, 1968.

[4] John E. Hopcroft, “An nlogn algorithm for minimizing the states in a finite automaton.” *The Theory of Machines and Computation*, 1970.

[1] Flex：`https://github.com/westes/flex`

[2] `re`：`https://docs.python.org/3/library/re.html`

[3] pest：`https://github.com/pest-parser/pest`

[4] lalrpop：`https://github.com/lalrpop/lalrpop`

练习

1) 假设某编程语言在词法分析环节需要支持日期识别，日期的格式为 `YYYY-MM-DD`，例如 `1980-01-12` 或 `0001-01-02`。回答以下两个问题：

   a) 写出正则表达式，要求合法日期格式中月份 MM 必须是 01-12，日期 DD 必须是 01-31。

   b) 改进正则表达式，使 DD 满足下列条件：

      - 如果月份为 01, 03, 05, 07, 08, 10, 12，则日期范围为 01-31；
      - 如果月份为 04, 06, 09, 11，则日期范围为 01-30；
      - 如果月份为 02，则日期范围为 01-28。

2) 选择一门你熟悉的语言（如 Markdown、LaTeX、HTML、Python 或 Golang），并按以下步骤进行词法分析：

   a) 找出该语言中的词法词元；

   b) 使用正则表达式设计词法规则；

   c) 将正则表达式转换为 NFA；

   d) 将 NFA 转换为 DFA 并进行优化。

3) 在 RESTful API 中，API 路径由字符串常量和变量拼接而成，其中变量以 `:` 开头。例如，以下三个 API 中的 `:id`、`:branch` 和 `:sha` 都是变量：

```
API-1: GET /projects/:id/repository/branches
API-2: GET /projects/:id/repository/branches/:branch
API-3: GET /projects/:id/repository/commits/:sha
```

   调用 API 时，变量会被替换为具体的参数值，例如下面的调用日志对应 API-1 和 API-3：

```
2021-07-04 16:43:47.193: Sending:
 'GET /projects/MyProject/repository/branches?'
2021-07-04 16:43:49.761: Sending:
 'GET /projects/MyProject/repository/commits/ed899a2f?'
```

   问：给定多个 API 定义和一组访问日志，如何识别每条日志属于哪个 API？该问题是否可以用正则表达式解决？

3 上下文无关文法

本章学习目标：

- ⋆⋆⋆ 掌握上下文无关文法的基本概念；
- ⋆⋆⋆ 理解上下文无关文法的二义性问题及其消除方法；
- ⋆⋆⋆ 学会使用 EBNF 范式定义上下文无关语言的语法规则；
- ⋆⋆⋆ 理解 Tea 语言的语法规则；
- ⋆ 了解 Chomsky 文法的分类。

3.1 上下文无关文法

在上一节中，我们已经使用正则表达式定义了计算器中的标签类型。然而，正则表达式无法进一步刻画计算器表达式的结构，尤其难以处理括号匹配等嵌套结构问题。本节将介绍上下文无关文法（Context-Free Grammar, CFG）。与正则文法相比，CFG 具有更强的表达能力，能够描述更加复杂的语法结构。

定义 3 (上下文无关文法)**.** 上下文无关文法由一组产生式（或规则）构成。每个产生式的形式为 $X \mapsto \gamma$，其中 X 是一个非终结符，γ 是由终结符和非终结符组成的字符串。

规则 3.1 给出了一个使用 CFG 描述合法计算器表达式的示例。该规则从非终结符 `E` 出发，通过不断应用产生式进行展开，能够生成所有合法的计算器表达式，同时不会生成任何非法表达式。

```
[1] E ↦ E '+' E
[2] E ↦ E '-' E
[3] E ↦ E '*' E
[4] E ↦ E '/' E
[5] E ↦ E '^' E                    (3.1)
[6] E ↦ '(' E ')'
[7] E ↦ NUM
[8] NUM ↦ <UNUM>
[9] NUM ↦ '-' <UNUM>
```

需要注意的是，在 CFG 中，每条产生式的左侧必须且只能是一个非终结符，不能包含额外的上下文约束。例如，$aX \mapsto ab$ 和 $bX \mapsto bc$ 的左侧对 X 的展开施加了上下文条件，因此不符合上下文无关文法的定义。

3.2 二义性问题和消除

虽然语法规则 3.1 可以覆盖所有合法的计算器表达式，但对于某些表达式，可能存在多种解析方式，从而产生二义性。以算式 `1+2*3` 为例，图 3.1 展示了两种不同的解析方式。这两棵语法解析树对应的计算结果并不相同，其中仅解析树 1 是符合通常算术语义的正确结果。

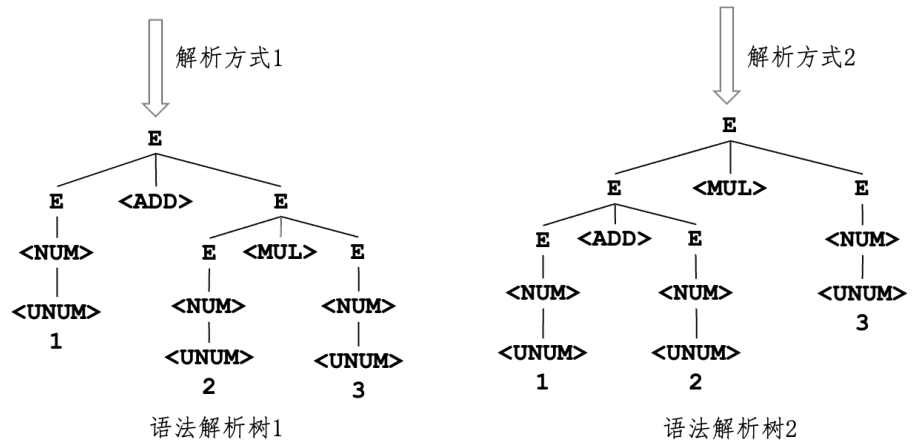


图 3.1: 算式 1+2*3 的两种解析方式

语法规则 3.1 存在二义性的主要原因有两点。首先，该规则未体现运算符的优先级关系；其次，未规定运算符的结合性，从而在处理诸如 “2^3^4” 这样的连续指数运算时可能产生错误的解析。

为消除由优先级引起的二义性，应在语法设计中显式引入运算符优先级，使高优先级运算的结果只能作为低优先级运算的操作数。为解决由结合性引起的二义性，则需要在文法中加以约束：对于左结合运算，递归应作用于左操作数；对于右结合运算，递归应作用于右操作数。

基于上述思路，对规则 3.1 进行改写，得到无二义性的文法规则 3.2 。

```
[1] E  ↦  E OP1 E1
[2] E  ↦  E1
[3] E1  ↦  E1 OP2 E2
[4] E1  ↦  E2
[5] E2  ↦  E3 OP3 E2
[6] E2  ↦  E3
[7] E3  ↦  NUM
[8] E3  ↦  '('  E  ')'
[9] NUM  ↦  <UNUM>
[10] NUM  ↦  '-'  <UNUM>
[11] OP1  ↦  '+'
[12] OP1  ↦  '-'
[13] OP2  ↦  '*'
[14] OP2  ↦  '/'
[15] OP3  ↦  '^'
```
(3.2)

该改写主要体现在以下几个方面：

- **区分运算符优先级**：通过引入不同层级的运算符集合，明确优先级关系。其中，OP1 表示优先级最低的加减运算，OP2 表示乘除运算，OP3 表示优先级最高的指数运算。
- **分层表示表达式结构**：通过引入分层非终结符（如 E、E1、E2、E3），将不同优先级的表达式逐层刻

画。例如，E 表示加减表达式，其操作数为更高优先级的乘除表达式 E1；同时，通过规则 E $\mapsto$ E1 保证文法的完备性与等价性。

- **显式刻画运算符结合性**：在文法中通过递归方向体现结合性。对于左结合运算（如 OP1 和 OP2），采用左递归形式（如 E $\mapsto$ E OP1 E1）；对于右结合运算（如指数运算 OP3），采用右递归形式（如 E2 $\mapsto$ E3 OP3 E2）。

3.3 扩展 BNF 范式

由于上下文无关文法的规则形式相对繁琐，在实际应用中通常采用扩展 BNF 范式（EBNF: Extended Backus–Naur Form）[2] 来描述具体的语言语法。EBNF 在基本 CFG 的基础上，引入了可选和闭包等构造，使语法规则更加紧凑且易于理解。

为了与上一节中使用的正则表达式符号保持一致，并提高书写的直观性与便捷性，本文采用表 3.1 所示的符号体系来构造 EBNF 表达式。该表示方法在一定程度上借鉴了 PEG 文法（Parsing Expression Grammar）[3] 中的符号设计，使语法描述更加统一和清晰。

表 3.1: 本文采用的 EBNF 文法构造符号

构造方式	符号	优先级	示例	含义
特定字符串	`' '`	5	`'ab'`	匹配字符串 “`ab`”
可选匹配	`?`	4	$\alpha?$	匹配 α 或空串 ϵ
闭包	`*`	4	$\alpha*$	匹配 α 的零次或多次重复（≥ 0）
正闭包	`+`	4	$\alpha+$	匹配 α 的一次或多次重复（≥ 1）
排除	`-`	3	$\alpha - \beta$	匹配 α 但不匹配 β（α, β 为正则表达式）
连接		2	$\alpha\beta$	依次匹配 α 和 β
选择	`|`	1	$\alpha\|\beta$	匹配 α 或 β

基于上述表示方法，语法规则 3.3 使用 EBNF 对规则 3.2 进行了等价改写。相比原始 CFG，该表示形式更加简洁，并显著提升了语法规则的可读性与表达能力。因此，在实际编程语言的语法定义中，EBNF 是更为常用的描述方式。

```
    E ↦ Factor (('+'|'-') Factor)*
Factor ↦ Power (('*'|'/') Power)*
 Power ↦ Value ('^ Power)?
 Value ↦ <UNUM> | '-' <UNUM> | '(' E ')'
```
(3.3)

需要注意的是，上述改写不仅简化了规则形式，同时仍然保持了运算符的优先级与结合性：加减运算为最低优先级且左结合，乘除运算次之且左结合，指数运算优先级最高且为右结合。

上下文无关文法相关技术已经发展得较为成熟，存在许多实用的语法分析工具，例如 Bison[1]、ANTLR[2] 以及 Pest[3]。其中，Bison 通常与词法分析器（如 Flex）配合使用，ANTLR 提供了较为完善的语法分析框架，而 Pest 则是一种基于 PEG 文法的解析器生成工具，语法形式与本文所采用的 EBNF 表示方法较为接近。在这些工具中，用户通常可以通过单独声明运算符的优先级和结合性来消解二义性，而无需在文法规则中显式分层，从而降低语法描述的复杂性，使语法描述更加直观、紧凑。

[1] Bison: `https://www.gnu.org/software/bison/`

[2] ANTLR: `https://github.com/antlr/antlr4`

[3] Pest: `https://pest.rs/`

3.4 Tea 语法规则

Tea 语言（Teaching Programming Language）是一门面向编译原理课程设计的教学语言。其语法设计可视为 Rust 的子集，支持类型缺省推导，但不包含 Rust 的所有权和借用检查等高级语义特性。

图 3.2 展示了一段使用 Tea 编写的阶乘函数示例代码。可以看到，该语言在函数定义、条件语句以及表达式结构等方面具有较强的直观性，同时保持了良好的结构化特征，便于进行语法分析。

```
fn factorial(n:i32) -> i32 {
    if n == 0 || n == 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

fn main() -> i32 {
    let r = factorial(n);
    return r;
}
```

图 3.2: Tea 代码样例：阶乘函数

接下来，我们将在上述示例的基础上，使用扩展 BNF 范式（EBNF）对 Tea 语言的核心语法规则进行形式化定义，从而为后续的语法分析与编译实现奠定基础。

3.4.1 运算符与优先级

Tea 语言中的运算符及其特性如表 3.2 所示。为了保持与常见语言的兼容性，这些运算符的优先级和结合性设置与 Rust 官方规范一致[4]，同时与 C 语言标准一致[5]。

表 3.2: Tea 中的运算符及优先级

优先级（C）	运算符	描述	结合性（C）	Tea 使用限制
8	`-`, `!`	单目运算符：负号、逻辑非	右	`-` 后仅允许跟数字，`!` 后仅允许跟括号
7	`*`, `/`	双目运算符：乘除法	左	—
6	`+`, `-`	双目运算符：加减法	左	—
5	`>`, `>=`, `<`, `<=`	比较运算符：大小比较	左	不支持连续比较
4	`==`, `!=`	比较运算符：等值判断	左	不支持连续比较
3	`&&`	逻辑与运算	左	—
2	`||`	逻辑或运算	左	—
1	`=`	赋值运算	右	不支持连续赋值

为了实现简化与便于解析，Tea 在运算符支持和使用方式上相比 Rust 做了一些限制。例如，Tea 不支持位运算，也不允许连续赋值（如 `a = b = c`）。这些限制将在后续的具体语法规则定义中体现。

[4]Rust 运算符优先级与结合性：https://doc.rust-lang.org/reference/expressions.html#expression-precedence

[5]C 语言运算符优先级：https://c-cpp.com/c/language/operator_precedence

3.4.2 代码基本组成

Tea 程序由若干语句和定义组成，包括模块引用、变量声明、函数声明与定义、结构体定义以及注释等。语法规则如下：

$$\texttt{program} \mapsto (\texttt{useStmt} \mid \texttt{varDeclStmt} \mid \texttt{fnDeclStmt} \mid \texttt{fnDef} \mid \texttt{structDef} \mid \texttt{comment})^* \tag{3.4}$$

3.4.3 代码模块引用

模块引用用于导入其它模块的定义，语法规则如下：

$$\begin{aligned} \texttt{useStmt} &\mapsto \texttt{'use'}\ \texttt{modPath}\ \texttt{';'} \\ \texttt{modPath} &\mapsto \texttt{id}\ (\texttt{'::'}\ \texttt{id})^* \end{aligned} \tag{3.5}$$

3.4.4 变量声明和定义

变量声明与定义是程序中存储数据的基础。Tea 语言支持标量和数组类型的声明，类型可以缺省或显式指定，并且可以在声明时进行初始化。但不允许在同一语句中声明多个变量或数组对象。

$$\begin{aligned} \texttt{varDeclStmt} &\mapsto \texttt{'let'}\ (\texttt{varDecl} \mid \texttt{varDef})\ \texttt{';'} \\ \texttt{varDecl} &\mapsto \texttt{id} \mid \texttt{typedDecl} \\ \texttt{typedDecl} &\mapsto \texttt{typedScalarDecl} \mid \texttt{typedArrayDecl} \\ \texttt{typedScalarDecl} &\mapsto \texttt{id}\ \texttt{':'}\ \texttt{type} \\ \texttt{typedArrayDecl} &\mapsto \texttt{id}\ \texttt{':'}\ \texttt{'['}\ \texttt{type}\ \texttt{';'}\ \texttt{num}\ \texttt{']'} \\ \texttt{varDef} &\mapsto \texttt{id}\ \texttt{'='}\ (\texttt{rValue} \mid \texttt{arrayInit}) \\ &\quad \mid \texttt{typedScalarDecl}\ \texttt{'='}\ \texttt{rValue} \\ &\quad \mid \texttt{typedArrayDecl}\ \texttt{'='}\ \texttt{arrayInit} \\ \texttt{arrayInit} &\mapsto \texttt{'['}\ \texttt{rValue}\ (\texttt{','}\ \texttt{rValue})^*\ \texttt{']'} \\ &\quad \mid \texttt{'['}\ \texttt{rValue}\ \texttt{';'}\ \texttt{num}\ \texttt{']'} \end{aligned} \tag{3.6}$$

由于标量类型和结构体类型在 Tea 语言中的语法规则相似，这里统一使用 `typedScalarDecl` 来表示标量或结构体类型的变量声明。

3.4.5 类型

Tea 语言支持原始类型和结构体类型，结构体可包含标量或数组字段：

$$\begin{aligned} \texttt{type} &\mapsto \texttt{primitiveType} \mid \texttt{structType} \mid \texttt{sliceType} \\ \texttt{primitiveType} &\mapsto \texttt{i32} \\ \texttt{structType} &\mapsto \texttt{id} \\ \texttt{sliceType} &\mapsto \texttt{'\&'}\ \texttt{'['}\ \texttt{type}\ \texttt{']'} \\ \texttt{structDef} &\mapsto \texttt{'struct'}\ \texttt{id}\ \texttt{'\{'}\ \texttt{varDeclList}\ \texttt{'\}'} \\ \texttt{varDeclList} &\mapsto \texttt{typedDecl}\ (\texttt{','}\ \texttt{typedDecl})^* \end{aligned} \tag{3.7}$$

由于指针和引用的复杂性，Tea 语言编译器暂未考虑在变量声明时直接使用引用类型。上述定义中的 `sliceType` 仅用于函数参数传递。另外，当前语法未考虑结构体构造器的设计，因此在使用结构体变量时，不能通过整体初始化的方式赋值，只能对各个字段分别进行赋值。

3.4.6 左右值

左值表示可以被赋值的对象或可寻址实体，例如变量、数组元素或结构体字段；右值表示表达式计算得到的值，可以用于运算、函数调用参数或布尔判断。所有左值都是右值的一种特殊情况，但并非所有右值都是左值。形式化定义如下：

```
    lValue ↦ id exprSuffix*
exprSuffix ↦ '.' id | '[' (id | num) ']'
    rValue ↦ arithExpr | boolExpr
 arithExpr ↦ (arithExpr ('+' | '-'))? factor
    factor ↦ (factor ('*' | '/'))? exprUnit
  exprUnit ↦ num | lValue | fnCall | '&' id | '(' arithExpr ')'
  boolExpr ↦ (boolExpr '||')? andExpr
   andExpr ↦ (andExpr '&&')? boolUnit
  boolUnit ↦ cmpExpr | '!'? '(' boolExpr ')'
   cmpExpr ↦ exprUnit ('==' | '!=' | '>' | '>=' | '<' | '<=') exprUnit
```
(3.8)

3.4.7 函数声明和调用

函数声明用于标明函数签名，函数调用用于执行函数逻辑。在 Tea 语言中，函数参数支持数组引用类型的传递。规则定义如下：

```
   fnDeclStmt ↦ 'fn' fnSign ';'
       fnSign ↦ id '(' paramList? ')'
              | id '(' paramList? ')' '->' type
    paramList ↦ varDeclList
       fnCall ↦ localCall | modPrefixCall
    localCall ↦ id '(' argList? ')'
modPrefixCall ↦ modPath '::' id '(' argList? ')'
      argList ↦ rValue (',' rValue)*
```
(3.9)

3.4.8 函数定义

函数定义包含函数体及语句列表：

$$
\begin{aligned}
\texttt{fnDef} &\mapsto \texttt{'fn' fnSign '\{' stmt}^{*}\texttt{ '\}'} \\
\texttt{stmt} &\mapsto \texttt{varDeclStmt | assignStmt | callStmt | retStmt | ifStmt} \\
&\quad \texttt{| whileStmt | breakStmt | continueStmt | ';'} \\
\texttt{assignStmt} &\mapsto \texttt{lValue '=' rValue ';'} \\
\texttt{callStmt} &\mapsto \texttt{fnCall ';'} \\
\texttt{retStmt} &\mapsto \texttt{'return' rValue? ';'} \\
\texttt{ifStmt} &\mapsto \texttt{'if' boolExpr '\{' stmt}^{*}\texttt{ '\}' ('else' '\{' stmt}^{*}\texttt{ '\}')?} \\
\texttt{whileStmt} &\mapsto \texttt{'while' boolExpr '\{' stmt}^{*}\texttt{ '\}'} \\
\texttt{breakStmt} &\mapsto \texttt{'break' ';'} \\
\texttt{continueStmt} &\mapsto \texttt{'continue' ';'}
\end{aligned}
\tag{3.10}
$$

3.4.9 标识符和数字

我们用正则表达式定义标识符和数字。

$$
\begin{aligned}
\texttt{id} &\mapsto \texttt{[a-z\_A-Z][a-z\_A-Z0-9]*} \\
\texttt{num} &\mapsto \texttt{unum | ('-' unum)} \\
\texttt{unum} &\mapsto \texttt{[1-9][0-9]* | 0}
\end{aligned}
\tag{3.11}
$$

3.4.10 代码注释

Tea 支持单行和多行注释，形式化定义如下：

$$
\begin{aligned}
\texttt{comment} &\mapsto \texttt{lineComment | blockComment} \\
\texttt{lineComment} &\mapsto \texttt{'//' [\^{}\textbackslash n]}^{*}\texttt{ '\textbackslash n'} \\
\texttt{blockComment} &\mapsto \texttt{'/*' ([\^{}*] | '*' [\^{}/])}^{*}\texttt{ '*/'}
\end{aligned}
\tag{3.12}
$$

3.5 文法能力分类

根据表示能力的不同，[1] 将文法分为四个等级。如表 3.3 所示，正则文法（3 型）的表示能力最弱，无法表示像 $a^n b^n$ $(n \in N)$ 这类要求字符出现次数相同的语言。所有正则语言都可以通过上下文无关文法表示，即产生式右侧允许包含非终结符。由于 CFG 不考虑上下文，它在编程语言语法设计中无法直接保证变量定义与使用的类型一致性。因此，类型检查等语义规则至少需要借助上下文敏感文法（1 型文法）来描述。而常见通用编程语言，如 C、Rust 等，其完整语法与语义规则属于 0 型文法（递归可枚举文法）。

表 3.3: Chomsky 文法分类

类型	计算模型	规则形式	语言示例
0 型：递归枚举	图灵机	-	普通程序
1 型：上下文敏感	线性有界图灵机	$\alpha S \to \beta$	$a^n b^n c^n,\ n \in N$
2 型：上下文无关	下推自动机	$S \to \beta$	$a^n b^n,\ n \in N$
3 型：正则	有穷自动机	$S \to a \mid b$	$a^n,\ n \in N$

理论上，每一级文法都可以用来描述下一级文法的规则。例如，CFG 可以用来定义正则表达式的语法，并进一步解析任意正则表达式；同理，0 型文法可以用于描述 1 型文法的规则，从而应用于类型推导或类型检查等任务。

参考文献

[1] Noam Chomsky. “On certain formal properties of grammars.” *Information and Control 2*, 1959.

[2] ISO/IEC 14977:1996 Information Technology-Syntactic Metalanguage-Extended BNF.

[3] Bryan Ford. “Parsing expression grammars: a recognition-based syntactic foundation.” *In Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2004.

练习

1) 判断下列两组 CFG 规则是否属于正则文法：

   (a) S $\mapsto$ 0S1S|1S0S|ϵ

   (b) S $\mapsto$ aT|b, T $\mapsto$ c|ϵ

2) 以开发一个能够接收任意正则表达式并生成对应正则匹配器的工具为目标，设计用于解析正则表达式的语法规则。

3) Scheme 是一种函数式编程语言，属于 Lisp 语言家族的分支。部分语法规则如下表所示。请以编写一个 Scheme 语法解析器为目标，设计“函数定义”的上下文无关语法规则。只需覆盖 factorial 函数示例，并保证规则逻辑自洽，无需全面实现全部 Scheme 特性。

表 3.4: Scheme 部分语法规则

规则	Scheme 代码示例	规则描述	含义
算数运算	`(+ x 1)`	支持 +、-、*、/	x + 1
变量赋值	`(define y (+ x 1))`	`define` 变量名 右值表达式	y = x+1

函数定义	`(define [foo x y]`
	`  (define z 2)`
	`  (- (+ x y) z))`
		`define` [函数名 参数] 函数体	定义函数 foo(x,y)
			返回值：x + y - z

函数调用	`(foo 1 2)`	函数名 参数 1 参数 2 ..	foo(1,2)

条件语句	`(if (= n 1)`
	`  1`
	`  0)`
		`if` 条件 分支 1 分支 2	if(n==1) 1 else 0

```
(define [factorial n]
    (if (= n 1)
        n
        (* n (factorial (- n 1)))
    )
)
```

代码 3.1: Scheme 代码示例：阶乘函数

4 自顶向下解析

本章学习目标：

- ⋆⋆ 掌握 LL(1) 文法及其解析算法
- ⋆⋆⋆ 掌握 PEG 文法的 Packrat 解析算法
- 了解 Earley 解析算法

4.1 自顶向下解析

给定一套语法规则 G 和句子 s，寻找由 G 推导出 s 的过程称为解析。本章将介绍一种自顶向下的解析方法，即从 G 的起始符号出发，逐步展开非终结符，直到生成的终结符序列与目标句子完全匹配。具体而言，本章采用最左推导策略，即在每一步中选择当前句型中最左侧的非终结符进行展开。对于无二义性的语法 G，若 $s \in L(G)$，则存在且仅存在一种最左推导；若 $s \notin L(G)$，则不存在任何有效的解析过程。

语法解析的难点在于：在每一步推导中，一个非终结符可能对应多条产生式，如何选择合适的产生式是关键问题；此外，某些文法中可能存在递归结构，从而导致解析过程无法终止。一种基本思路是对上下文无关文法施加一定限制，以避免复杂情况。接下来，将介绍几种在不同限制条件下的解析方法：

- LL(1) 文法：不允许左递归，且不允许回溯
- PEG 文法：不允许左递归
- Earley 算法：对上下文无关文法不作限制

4.2 LL(1) 文法和解析

4.2.1 LL(1) 文法

为了降低解析算法的复杂度，可以对上下文无关文法施加一定的限制。LL(1) 文法（Left-to-right, Leftmost, 前瞻 1 个字符）有两个基本要求：一是不含左递归，二是在解析过程中不需要回溯。接下来将分别讨论这两个性质。

左递归与消除

对于一条产生式，若其右侧推导出的句型的最左符号可以为该产生式左侧的非终结符，则称该文法存在左递归。例如：`E` $\mapsto$ `E OP1 E1`。左递归会导致自顶向下解析过程中出现无限递归，从而使解析无法终止。通常可以通过对文法进行等价变换来消除左递归。

一般地，对于形如：

$$X \mapsto X\alpha_1 \mid X\alpha_2 \mid \cdots \mid \beta_1 \mid \beta_2$$

的产生式（其中 β_i 不以 X 开头），可以改写为：

$$\begin{aligned} X &\mapsto \beta_1 Y \mid \beta_2 Y \\ Y &\mapsto \alpha_1 Y \mid \alpha_2 Y \mid \epsilon \end{aligned}$$

上一章定义的计算器语法中，第 [1] 条和第 [3] 条产生式存在左递归。通过应用上述方法，可以消除这些左递归。修改后的语法规则如式 4.1 所示。

```
[1] E  ↦  E1 E'
[2] E'  ↦  OP1 E1 E'
[3] E'  ↦  ϵ
[4] E1  ↦  E2 E1'
[5] E1'  ↦  OP2 E2 E1'
[6] E1'  ↦  ϵ
[7] E2  ↦  E3 OP3 E2
[8] E2  ↦  E3
[9] E3  ↦  NUM
[10] E3  ↦  '('  E  ')'
[11] NUM  ↦  <UNUM>
[12] NUM  ↦  '-'  <UNUM>
[13] OP1  ↦  '+'
[14] OP1  ↦  '-'
[15] OP2  ↦  '*'
[16] OP2  ↦  '/'
[17] OP3  ↦  '^'
```
(4.1)

无回溯语法

对于同一非终结符的任意两条产生式，若它们可能推导出的串的首个终结符不同，则可以通过前瞻一个终结符来唯一确定所使用的产生式。当产生式右侧的首符号为非终结符时，需要递归考察该非终结符所能推导出的首终结符。

以规则 4.2 为例，若当前待展开的非终结符为 X，则根据下一个输入符号为 'a'、'b' 或 'c'，总可以选择出唯一的产生式，从而无需回溯。

```
[i] X ↦ 'a'...
[j] X ↦ 'b'...
[k] X ↦ Y...
[p] Y ↦ 'c'...
...
```
(4.2)

当语法存在回溯时，可以通过提取左公因子对文法进行等价变换，从而消除回溯。例如：

```
X ↦ 'a' A | 'a' B | 'b'
        ⇓
X ↦ 'a' Y | 'b'
Y ↦ A | B
```
(4.3)

语法规则 4.1 中，第 [7] 条和第 [8] 条产生式同属于非终结符 E2，且右侧的首个符号都是 E3，因此在某些情况下可能导致回溯问题。通过应用上述左公因子提取方法，可以对该部分语法进行改写，从而消除回溯。修改后的结果见语法规则 4.4 。

```
[1] E ↦ E1 E'
[2] E' ↦ OP1 E1 E'
[3] E' ↦ ϵ
[4] E1 ↦ E2 E1'
[5] E1' ↦ OP2 E2 E1'
[6] E1' ↦ ϵ
[7] E2 ↦ E3 E2'
[8] E2' ↦ OP3 E2
[9] E2 ↦ ϵ
[10] E3 ↦ NUM
[11] E3 ↦ '(' E ')'
[12] NUM ↦ <UNUM>
[13] NUM ↦ '-' <UNUM>
[14] OP1 ↦ '+'
[15] OP1 ↦ '-'
[16] OP2 ↦ '*'
[17] OP2 ↦ '/'
[18] OP3 ↦ '^'
```
(4.4)

4.2.2 LL(1) 文法解析

接下来，我们利用 LL(1) 文法的无回溯特性，构建预测分析表，用于记录每个非终结符可产生的首个终结符及其对应产生式。

我们定义

$$First\big(X \xrightarrow{[i]} \beta_1\beta_2\dots\beta_n\big)$$

表示应用 X 的第 i 条产生式所能产生的首终结符集合。计算规则如下：

- 如果 $\epsilon \notin First(\beta_1)$，则
$$First\big(X \xrightarrow{[i]} \beta_1\beta_2\dots\beta_n\big) = First(\beta_1)$$
- 如果 $\beta_1,\dots,\beta_k$ 都可能产生 ϵ，则
$$First\big(X \xrightarrow{[i]} \beta_1\beta_2\dots\beta_n\big) = \bigcup_{j=1}^{k+1} First(\beta_j)$$

其中 k 是使 β_{k+1} 首次不能产生 ϵ 的索引；如果所有 β_j 都可能产生 ϵ，则 $First$ 集还需包含 ϵ。

表 4.1 展示了语法 4.4 中每条产生式对应的 $First$ 集。

表 4.1: 语法 4.4 的 $First$ 集合

	<UNUM>	'+'	'-'	'*'	'/'	'^'	'('	')'	ϵ
E	[1]		[1]				[1]		
E'		[2]	[2]						[3]
E1	[4]		[4]				[4]		
E1'				[5]	[5]				[6]
E2	[7]		[7]				[7]		
E2'						[8]			[9]
E3	[10]		[10]				[11]		
NUM	[12]		[13]						
OP1		[14]	[15]						
OP2				[16]	[17]				
OP3						[18]			

说明：行表示非终结符，列表示终结符，单元格中的值表示对应的规则编号。

由于句子词元序列中不包含 ϵ，$\epsilon \in First(X \to \beta)$ 对规则选择的帮助有限。因此需要对表 4.1 中的 ϵ 一列进行特殊处理。主要思路是：当 $\epsilon \in First(X \to \beta)$ 时，进一步考虑 X 之后可能出现的字符 $Follow(X \xrightarrow{[i]} ...)$，并据此决定是否采用规则 $X \to \epsilon$。为此，我们使用 $First^+(X \xrightarrow{[i]} \beta)$ 表示应用 X 的第 i 条规则所能产生的首个终结符集合（不含 ϵ）。

$$First^+(X \mapsto \beta) = \begin{cases} First(\beta), & \text{if } \epsilon \notin First(\beta) \\ First(\beta) \cup Follow(X), & \text{otherwise} \end{cases}$$

基于 $First^+$ 集的定义，可以严格表述 LL(1) 文法的无回溯性质：

$$\forall i \neq j, \quad First^+(X \to \beta_i) \cap First^+(X \to \beta_j) = \emptyset.$$

根据 $First^+$ 集合的计算方法，我们更新表 4.1 并消除其中的 ϵ 列，最终得到 LL(1) 解析表。结果如表 4.2 所示，可以看出该表的所有单元格至多包含一条规则，符合无回溯文法的特性。

表 4.2: LL(1) 解析表：记录每条规则的 $First^+$ 集合

	<UNUM>	'+'	'-'	'*'	'/'	'^'	'('	')'
E	[1]		[1]				[1]	
E'		[2]	[2]					[3]
E1	[4]		[4]				[4]	
E1'		[6]	[6]	[5]	[5]			[6]
E2	[7]		[7]				[7]	
E2'		[9]	[9]	[9]	[9]	[8]		[9]
E3	[10]		[10]				[11]	
NUM	[12]		[13]					
OP1		[14]	[15]					
OP2				[16]	[17]			
OP3						[18]		

通过查询 LL(1) 解析表，可以实现精准快速的解析。图 4.1 展示了使用表 4.2 解析算式 <UNUM(1)> '+' <UNUM(2)> '*' <UNUM(3)> 的过程及最终结果。图中每个节点的属性 m:[n] 表示在第 m 步应用规则 n 进行展开。

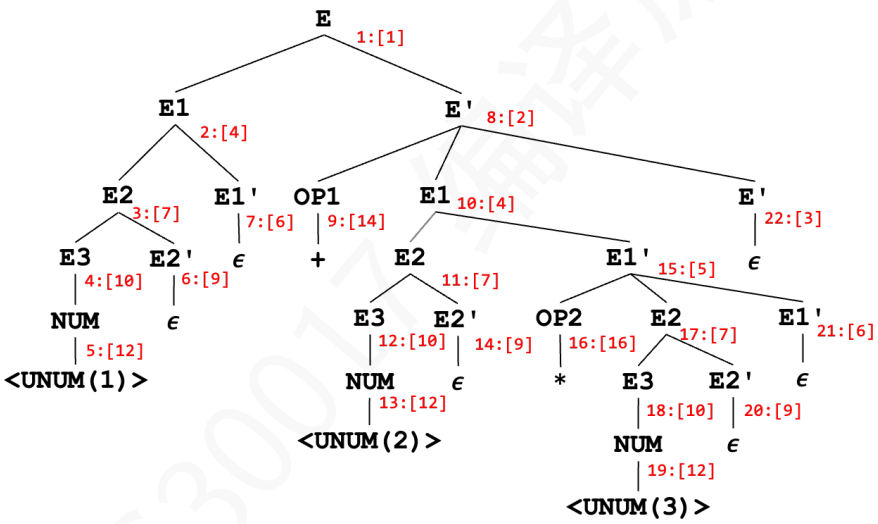


图 4.1: 应用表 4.2 解析 <UNUM(1)> '+' <UNUM(2)> '*' <UNUM(3)

4.3 PEG 文法与解析

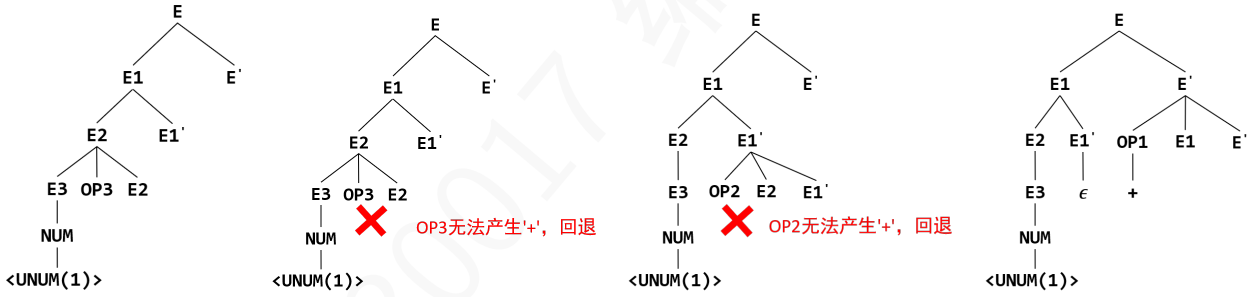
由于 LL(1) 文法在实际应用中受限，改写语法规则往往冗长。相比之下，PEG（Parsing Expression Grammar, PEG）[1] 允许有限回溯，能够更灵活地定义语言规则，因此应用场景更广泛。需要注意的是，PEG 同样不允许左递归，并通过有序选择（运算符：`/`）避免解析时的二义性。

对消除左递归后的文法 4.1进行适当改写，并引入条件选择，即可得到如下 PEG 文法：

```
E  ↦  E1 E'
E'  ↦  OP1 E1 E' / ϵ
E1  ↦  E2 E1'
E1'  ↦  OP2 E2 E1' / ϵ
E2  ↦  E3 OP3 E2 / E3
E3  ↦  NUM / '(' E ')'
NUM  ↦  <UNUM> / '-'  <UNUM>
OP1  ↦  '+' / '-'
OP2  ↦  '*' / '/'
OP3  ↦  '^'
```

(4.5)

由于 PEG 的递归下降解析可能导致指数级回溯，Packrat 解析方法 [2] 通过在每个输入位置缓存每条规则的解析结果，实现记忆化。这样既保持了 PEG 的完整回溯能力，又将解析时间复杂度降低到线性。



(a) `1 ∘ '+' 2 '*' 3`　(b) `1 ∘ '+' 2 '*' 3`　(c) `1 ∘ '+' 2 '*' 3`　(d) `1 '+' ∘ 2 '*' 3`

图 4.2: 使用 PEG 文法解析算式 `1 '+' 2 '*' 3` 的步骤示意图。游标 ∘ 左侧表示当前已处理字符。

Packrat 解析算法如代码 5 所示，其中每条规则的解析结果都会存入记忆表，从而避免重复解析并保证解析效率。图 4.2以算式 `1 '+' 2 '*' 3` 为例展示了 PEG 文法的解析过程。其中，在图 4.2b 中，当解析无法继续进行时，对应的记忆表状态如表 **??** 所示，此时触发回溯。具体而言，解析器由产生式 `E2 ↦ E3 OP3 E2` 回退并切换至 `E2 ↦ E3` 。由于此前在相同输入位置已经成功解析过 `E3`，因此可以直接复用其解析结果，而无需重新计算。

表 4.3: PEG 文法解析过程中记忆表的变化：左表为图 4.2b回溯发生时的中间状态，右图为最终解析完成后的状态。表中列 1 ∼ 5 表示输入串 `1 '+' 2 '*' 3` 的位置，行表示各非终结符。单元格中的数字表示在对应输入位置从非终结符解析成功的字符数。

(a) 回溯时的记忆表状态

	1	2	3	4	5
E					
E'					
E1					
E1'					
E2					
E3	1（复用）				
NUM	1				
OP1					
OP2					
OP3					
	1	+	2	*	3

(b) 最终解析完成后的记忆表

	1	2	3	4	5
E	5				
E'		4			
E1	1		3		
E1'				2	
E2	1		1		1
E3	1		1		1
NUM	1		1		1
OP1		1			
OP2				1	
OP3					
	1	+	2	*	3

算法 5 Packrat 解析算法

input: G: PEG grammar with a start symbol P; ts: token stream;
output: a parse result (success / failure) with parse tree;

procedure PackratParse(ts, G)
  初始化记忆表 M // $M[X, i]$ 记录非终结符 X 在位置 i 的解析结果
  return Parse(P, 0)
end procedure
procedure Parse(X, i)
  if $M[X, i]$ 已存在 **then**
    return $M[X, i]$ // 直接返回缓存结果（避免重复解析）
  end if
  for each $X \leftarrow \alpha \in G$ **do** // 按顺序尝试 PEG 产生式
    $(success, j) \leftarrow$ ParseSeq(α, i)
    if $success$ **then**
      $M[X, i] \leftarrow (true, j)$
      return $(true, j)$ // PEG 选择第一个成功的分支
    end if
  end for
  $M[X, i] \leftarrow (false, i)$
  return $(false, i)$
end procedure
procedure ParseSeq(α, i)
  $j \leftarrow i$
  for each 符号 s in α **do** // 依次匹配序列中的符号
    if s 是终结符 **then**
      if $ts[j] == s$ **then**
        $j \leftarrow j + 1$
      else
        return $(false, i)$
      end if
    else// s 是非终结符
      $(success, j') \leftarrow$ Parse(s, j)
      if not $success$ **then**
        return $(false, i)$
      end if
      $j \leftarrow j'$
    end if
  end for
  return $(true, j)$
end procedure

4.4 Earley 解析算法

Earley 解析算法 [3] 是一种通用的 CFG 解析算法，不对具体的语法规则做任何限制。Earley 算法包括以下三种基本操作：

- **预测**：对于规范项 $X \to \alpha \circ Y\beta$（符号 $\circ$ 表示当前解析位置），根据语法规则推导 $Y \to \circ\gamma$；
- **扫描**：如果下一个待解析的终结符是 a，且存在状态 $X \to \alpha \circ a\beta$，则移进终结符 a，并更新规范项为 $X \to \alpha a \circ \beta$；
- **完成**：如果规范项为 $Y \to \gamma \circ$，即完成了对非终结符 Y 的展开，则将所有关联状态 $X \to \alpha \circ Y\beta$ 更新为 $X \to \alpha Y \circ \beta$。

算法 6 详细描述了 Earley 算法的解析过程。该算法在符号展开时能够有效避免无限递归和路径爆炸问题，提高解析效率。

算法 6 Earley 解析算法

```
input: G: context-free grammar with a start symbol P; ts: token stream;
output: a parse tree;
procedure EarleyParse(ts, G)
    for each P → γ ∈ G do // 初始化，选取 G 中每一条以 P 开头的规则
        S[0].add((P → ∘γ, 0)) // 添加到 Earley 解析状态 S[0] 中，第二个参数 0 表示规则起源于 Earley 解析状态 S[0]
    end for
    for each i in 0..ts.len() do // 遍历每一个终结符
        for each item in S[i] do // 遍历 S[i] 中的每一条规则状态
            match NextSymbol(item) :
                case END ⇒ Complete(item, i) // 当前规则右侧字符串已经全部匹配，则执行完成操作
                case ts[i] ⇒ Scan(item, i, ts) // 当前规则状态的下一个字符为终结符，且恰好是目标终结符，执行扫描操作
                case NON-TERMINAL ⇒
                    Predict(item, i, G) // 当前规则状态的下一个字符为非终结符，执行预测操作
            end match
        end for
    end for
end procedure
procedure Complete((A → β∘, j), i) // 完成操作：j 表示此条规则的起始 Earley 解析状态，i 是当前 Earley 解析状态
    for each (B → α∘Aδ, k)∈ S[j] do // 完成操作只会更新 S[j] 中的相关的规则状态
        S[i].add((B → αA∘δ, k)) // 移进完成的非终结符 A
        if δ == ϵ then // 规则 B → αA∘ 已经扫描完成
            Complete((B → αA∘, k), i) // 继续对 S[k] 中相关的规则执行完成操作
        end if
    end for
end procedure
procedure Predict((A → α∘Bβ, j), i) // 预测操作
    for each B → γ in G do
        S[i].add((B → γ, i))
    end for
end procedure
procedure Scan((A → α∘aβ, j), i) // 扫描操作
    if a == ts[i] then
        S[i + 1].add((A → αa∘β, j)) // 移进终结符，并将新的规则状态添加到下一个 Earley 解析状态 S[i+1] 中
    end if
end procedure
```

下面以词元序列 `<UNUM(1)> '+' <UNUM(2)> '*' <UNUM(3)>` 为例[1]，演示 Earley 算法的解析过程，其具体步骤如表 4.4–4.9 所示。

表 4.4: 状态 $S[0]$：∘ `<UNUM(1)> '+' <UNUM(2)> '*' <UNUM(3)>`

序号	操作	条目	
		规范项	起源
1	初始化	E → ∘ E OP1 E1	S[0]
2	初始化	E → ∘ E1	S[0]
3	预测 2	E1 → ∘ E1 OP2 E2	S[0]
4	预测 2	E1 → ∘ E2	S[0]
5	预测 4	E2 → ∘ E3 OP3 E2	S[0]
6	预测 4	E2 → ∘ E3	S[0]
7	预测 5	E3 → ∘ NUM	S[0]
8	预测 5	E3 → ∘ '(' E ')'	S[0]
9	预测 7	NUM → ∘ <UNUM>	S[0]
10	预测 7	NUM → ∘ '-' <UNUM>	S[0]
11	扫描 9	-	-

表 4.5: 状态 $S[1]$：`<UNUM(1)>` ∘ `'+' <UNUM(2)> '*' <UNUM(3)>`

序号	操作	条目	
		规范项	起源
1	扫描 s[0]-9	NUM → <UNUM> ∘	S[0]
2	完成：基于 1 更新 s[0]-7	E3 → NUM ∘	S[0]
3	完成：基于 2 更新 s[0]-5	E2 → E3 ∘ OP3 E2	S[0]
4	完成：基于 2 更新 s[0]-6	E2 → E3 ∘	S[0]
5	完成：基于 4 更新 s[0]-4	E1 → E2 ∘	S[0]
6	完成：基于 5 更新 s[0]-2	E → E1 ∘	S[0]
7	完成：基于 5 更新 s[0]-3	E1 → E1 ∘ OP2 E2	S[0]
8	完成：基于 6 更新 s[0]-1	E → E ∘ OP1 E1	S[0]
9	预测 3	OP3 → ∘ '^'	S[1]
10	预测 7	OP2 → ∘ '*'	S[1]
11	预测 7	OP2 → ∘ '/'	S[1]
12	预测 8	OP1 → ∘ '+'	S[1]
13	预测 8	OP1 → ∘ '-'	S[1]
14	扫描 12	-	-

[1]符号说明：`'+'` 表示词元 `<ADD>`，`'*'` 表示词元 `<MUL>`。

表 4.6: 状态 $S[2]$：<UNUM(1)> '+' ∘ <UNUM(2)> '*' <UNUM(3)>

序号	操作	条目	
		规范项	起源
1	扫描 s[1]-12	OP1 → '+' ∘	S[1]
2	完成：基于 1 更新 s[1]-8	E → E OP1 ∘ E1	S[0]
3	预测 2	E1 → ∘ E1 OP2 E2	S[2]
4	预测 2	E1 → ∘ E2	S[2]
5	预测 4	E2 → ∘ E3 OP3 E2	S[2]
6	预测 5	E2 → ∘ E3	S[2]
7	预测 5	E3 → ∘ NUM	S[2]
8	预测 5	E3 → ∘ '(' E ')'	S[2]
9	预测 7	NUM → ∘ <UNUM>	S[2]
10	预测 7	NUM → ∘ '-' <UNUM>	S[2]
11	扫描 9	-	-

表 4.7: 状态 $S[3]$：<UNUM(1)> '+' <UNUM(2)> ∘ '*' <UNUM(3)>

序号	操作	条目	
		规范项	起源
1	扫描 s[2]-9	NUM → <UNUM> ∘	S[2]
2	完成：基于 1 更新 s[2]-7	E3 → NUM ∘	S[2]
3	完成：基于 2 更新 s[2]-5	E2 → E3 ∘ OP3 E2	S[2]
4	完成：基于 2 更新 s[2]-6	E2 → E3 ∘	S[2]
5	完成：基于 4 更新 s[2]-4	E1 → E2 ∘	S[2]
6	完成：基于 5 更新 s[2]-2	E → E OP1 E1 ∘	S[0]
7	完成：基于 5 更新 s[2]-3	E1 → E1 ∘ OP2 E2	S[2]
8	预测 3	OP3 → ∘ '^'	S[3]
9	预测 7	OP2 → ∘ '*'	S[3]
10	预测 7	OP2 → ∘ '/'	S[3]
11	扫描 9	-	-

说明：S[3]-6 还会导致完成和更新 S[0]-1 的操作，如果下一个词元是'+' 或'-' 时有用；为节约空间，此处未列出相关规范项。

表 4.8: 状态 $S[4]$：<UNUM(1)> '+' <UNUM(2)> '*' ∘ <UNUM(3)>

序号	操作	条目	
		规范项	起源
1	扫描 s[3]-9	OP2 → '*' ∘	S[3]
2	完成：基于 1 更新 s[3]-7	E1 → E1 OP2 ∘ E2	S[2]
3	预测 2	E2 → ∘ E3 OP3 E2	S[4]
4	预测 2	E2 → ∘ E3	S[4]
5	预测 3	E3 → ∘ NUM	S[4]
6	预测 3	E3 → ∘ '(' E ')'	S[4]
7	预测 5	NUM → ∘ <UNUM>	S[4]
8	预测 5	NUM → ∘ '-' <UNUM>	S[4]
11	扫描 7	-	-

表 4.9: 状态 $S[5]$：`<UNUM(1)> '+' <UNUM(2)> '*' <UNUM(3)>` ∘

序号	操作	条目	
		规范项	起源
1	扫描 s[4]-7	`NUM` → `<UNUM>` ∘	S[4]
2	完成：基于 1 更新 s[4]-5	`E3` → `NUM` ∘	S[4]
3	完成：基于 2 更新 s[4]-3	`E2` → `E3` ∘ `OP3 E2`	S[4]
4	完成：基于 2 更新 s[4]-4	`E2` → `E3` ∘	S[4]
5	完成：基于 4 更新 s[4]-2	`E1` → `E1 OP2 E2` ∘	S[2]
6	完成：基于 5 更新 s[2]-2	`E` → `E OP1 E1` ∘	S[0]

Earley 本质上是一种动态规划方法，可以有效支持左递归。具体而言，在处理左递归产生式（如 `E` ↦ `E OP1 E1`）时，预测步骤会将该产生式加入当前状态集，但由于每个项目由产生式、游标位置和起始位置唯一确定，已经出现过的项目不会被重复加入，从而避免了无限递归。

参考文献

[1] Bryan Ford “Parsing expression grammars: a recognition-based syntactic foundation.” In Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2004.

[2] Bryan Ford. “Packrat parsing: simple, powerful, lazy, linear time, functional pearl.” ACM SIGPLAN Notices 37, no. 9 (2002): 36-47.

[3] Jay Earley. “An efficient context-free parsing algorithm.” *Communications of the ACM*, 1970.

练习

1) 已知如下正则表达式的语法规则：

```
[1] Regex  ↦  Regex '|' Concat
[2] Regex  ↦  Concat
[3] Concat  ↦  Concat Closure
[4] Concat  ↦  Closure
[5] Closure  ↦  Closure '*'
[6] Closure  ↦  Item
[7] Item  ↦  '(' Regex ')'
[8] Item  ↦  <Char>
```

(4.6)

a) 判断上述语法是否为 LL(1) 文法；若不是，请将其改写为 LL(1) 文法，并构造对应的预测分析表。

b) 判断上述语法是否符合 PEG 文法的要求；若不符合，请将其改写为等价的 PEG 文法，并给出对正则表达式 `ab*|c` 的解析过程。

c) 使用 Earley 算法对正则表达式 `ab*|c` 进行解析。

d) 分析 LL(1)、PEG（Packrat）和 Earley 三种解析方法在该问题上的时间复杂度及其差异。

5 自底向上解析

本章学习目标：

- ⋆⋆ 理解自底向上的语法分析思想，掌握移进-归约分析的基本过程
- ⋆⋆ 掌握 SLR 文法的解析方法，包括 SLR 分析表的构造与使用
- ⋆ 了解 LALR 和 LR(1) 文法的基本思想及其与 SLR 文法的区别

5.1 自底向上解析思想

自底向上解析是从输入串开始，通过不断进行规约操作，最终将其归约为文法开始符号的过程。本章主要介绍 LR（Left-to-right, Rightmost）自底向上解析方法，即从左到右扫描输入，并构造最右推导的逆过程。

LR 解析的基本操作包括：

- 移进（shift）：将输入串中的下一个词元读入解析栈；
- 规约（reduce）：根据产生式 $X \mapsto \beta$，将解析栈顶的 β 规约为 X。

该问题的难点在于：在解析过程中，某些状态下可能存在多种可选操作，例如既可以移进又可以规约，或存在多个规约选择，必须通过合适的分析方法进行选择。本章将以 SLR（Simple LR）文法为基础，详细介绍 LR 解析方法，并在此基础上进一步讨论 LALR 和 LR(1) 等扩展方法。

5.2 SLR 文法和解析

SLR 文法是指其对应的上下文无关文法在构造 SLR 分析表时不存在冲突，即解析表的每个单元格中至多包含一个分析动作。

对于一套 SLR 文法，其解析表的构建通常包括两个步骤：

- 构造 LR(0) 项目集规范族（canonical collection of LR(0) items）及其对应的有穷自动机；
  - LR(0) 项目（item）：在产生式右部插入一个点（$\circ$）得到的形式，如 $\texttt{A} \mapsto \alpha \circ \beta$，用于表示产生式右部的识别进度；
  - LR(0) 项目集（item set）：若干 LR(0) 项目构成的集合；在 LR 分析中，通常从某一项目出发，对待识别非终结符进行闭包展开，从而得到项目集。具体而言，若项目中存在 $\texttt{A} \mapsto \alpha \circ \texttt{B}\beta$，则需将所有形如 $\texttt{B} \mapsto \circ\gamma$ 的项目加入该项目集。
  - LR(0) 项目集规范族：从初始 LR(0) 项目进行闭包展开，并结合转移操作（Goto）逐步构造得到的全部项目集的集合。其中每一个 LR(0) 项目集对应一个 LR(0) 有穷自动机状态，每一个 Goto 操作表示状态转移关系。
- 根据 LR(0) 有穷自动机构造 SLR 解析表，即 Action-Goto 表。

下面以计算器文法为例，详细讲解 SLR 上述过程。

5.2.1 构造 LR(0) 有穷自动机

由于计算器语法中开始符号对应的产生式不唯一，为便于后续分析，我们在原文法基础上增加一条新的产生式 $\texttt{S} \mapsto \texttt{E}$。更新后的规则如语法 5.1 所示。

```
[0] S  ↦  E
[1] E  ↦  E OP1 E1
[2] E  ↦  E1
[3] E1  ↦  E1 OP2 E2
[4] E1  ↦  E2
[5] E2  ↦  E3 OP3 E2
[6] E2  ↦  E3
[7] E3  ↦  NUM
[8] E3  ↦  '(' E ')'
[9] NUM  ↦  <UNUM>
[10] NUM  ↦  '-' <UNUM>
[11] OP1  ↦  '+'
[12] OP1  ↦  '-'
[13] OP2  ↦  '*'
[14] OP2  ↦  '/'
[15] OP3  ↦  '^'
```
(5.1)

从初始 LR(0) 项目 $\texttt{S} \mapsto \circ\ \texttt{E}$ 开始，我们对其产生的符号进行闭包展开，可以得到一个初始的项目集，如表达式 5.2 所示，即 LR(0) 有穷自动机的初始状态。

$$
\begin{aligned}
\texttt{S} &\mapsto \circ\ \texttt{E} \\
\texttt{E} &\mapsto \circ\ \texttt{E OP1 E1} \\
\texttt{E} &\mapsto \circ\ \texttt{E1} \\
\texttt{E1} &\mapsto \circ\ \texttt{E1 OP2 E2} \\
\texttt{E1} &\mapsto \circ\ \texttt{E2} \\
\texttt{E2} &\mapsto \circ\ \texttt{E3 OP3 E2} \\
\texttt{E2} &\mapsto \circ\ \texttt{E3} \\
\texttt{E3} &\mapsto \circ\ \texttt{NUM} \\
\texttt{E3} &\mapsto \circ\ \texttt{'(' E ')'} \\
\texttt{NUM} &\mapsto \circ\ \texttt{<UNUM>} \\
\texttt{NUM} &\mapsto \circ\ \texttt{'-' <UNUM>}
\end{aligned}
\tag{5.2}
$$

算法 7 给出了上述 LR(0) 项目集的闭包构造过程。其中，输入集合 Q 表示一组核心项目（kernel items），通过不断展开 $\circ$ 后为非终结符的项目，最终得到完整的项目集。对于上述初始项目集来说，其核心项目是 $\texttt{S} \mapsto \circ\ \texttt{E}$。

算法 7 LR(0) 规范项集生成算法

```
procedure RegularSet(Q)
    hasChanged ← TRUE
    while hasChanged do
        hasChanged ← FALSE
        for each A ↦ β ∘ Cδ ∈ Q do
            for each C ↦ λ ∈ G do
                if C ↦ ∘λ ∉ Q then
                    Q ← Q ∪ {C ↦ ∘λ}
                    hasChanged ← TRUE
                end if
            end for
        end for
    end while
end procedure
```

LR(0) 有穷自动机中的每个状态对应一个项目集，边表示在文法符号上的状态转移关系。该自动机刻画了项目集之间在读入一个符号后的转移，即 Goto 操作。其构造过程从初始状态 S_0 开始，对每个项目集和可能的文法符号应用 Goto 操作，生成新的项目集；不断迭代该过程，直到不再产生新的项目集和状态转移关系为止。图 5.1 展示了语法规则 5.1 对应的 LR(0) 有穷自动机。

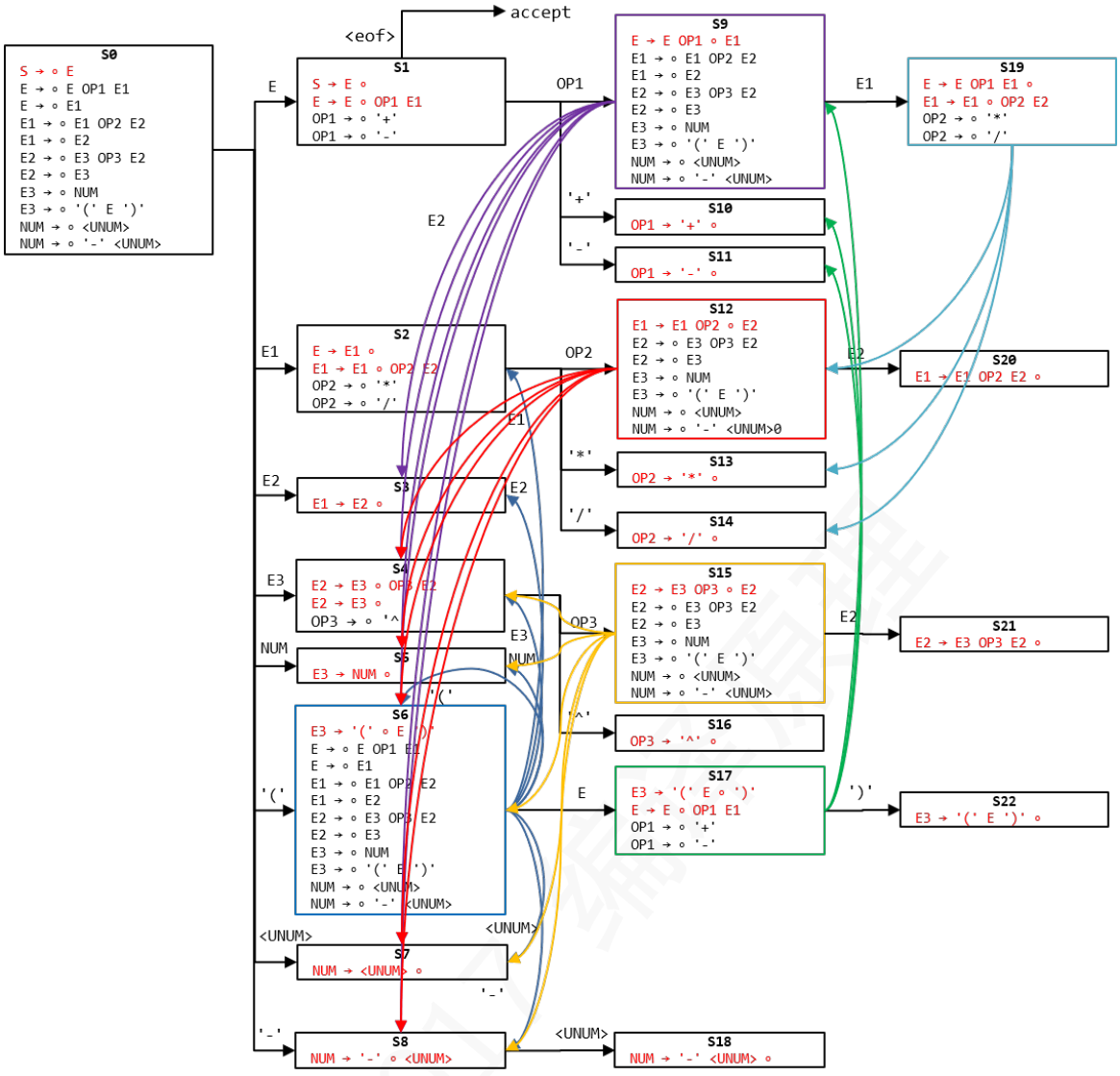


图 5.1: 语法规则 5.1 对应的 LR(0) 有穷自动机

5.2.2 创建 SLR 解析表

将 LR(0) 有穷自动机的状态转移关系转化为表格形式，即可得到 SLR 解析表。如表 5.1 所示，每一行对应一个 LR(0) 有穷自动机状态（即一个项目集），每一列对应一个文法符号，每个单元格表示在读入相应符号后转移到的目标状态。此外，对于包含形如 $\mathtt{A} \mapsto \alpha\circ$ 的项目的状态，还需要在表中填入规约动作。例如，R[2] 表示根据语法规则 [2] 进行规约，即将 $\mathtt{E} \mapsto \mathtt{E1}\circ$ 规约为 E。需要注意的是，规约操作并非在所有输入符号下均可执行，而仅在下一个输入符号属于 Follow(A) 时才允许。因此，每个状态的规约动作仅填入 SLR 解析表中对应的 Follow 集合的列中。

通常，根据文法符号是否为终结符，SLR 解析表分为两部分：Action 表（对应终结符）和 Goto 表（对应非终结符），其中规约操作仅出现在 Action 表中。

表 5.1: 语法规则 5.1 对应的 SLR 解析表

LR(0) 项目集	Goto								Action (Shift-Reduce)								
	E	E1	E2	E3	OP1	OP2	OP3	NUM	<UNUM>	'+'	'-'	'*'	'/'	'^'	'('	')'	eof
S0	S1	S2	S3	S4				S5	S7		S8				S6		
S1					S9					S10	S11						accept
S2						S12				R[2]	R[2]	S13	S14		R[2]	R[2]	
S3										R[4]	R[4]	R[4]	R[4]			R[4]	R[4]
S4							S15			R[6]	R[6]	R[6]	R[6]	S16		R[6]	R[6]
S5										R[7]	R[7]	R[7]	R[7]	R[7]		R[7]	R[7]
S6	S17	S2	S3	S4				S5	S7		S8				S6		
S7										R[9]	R[9]	R[9]	R[9]	R[9]		R[9]	R[9]
S8									S18								
S9		S19	S3	S4				S5	S7		S8				S6		
S10									R[11]		R[11]				R[11]		
S11									R[12]		R[12]				R[12]		
S12			S20	S4				S5	S7		S8				S6		
S13									R[13]		R[13]				R[13]		
S14									R[14]		R[14]				R[14]		
S15			S21	S4				S5	S7		S8				S6		
S16									R[15]		R[15]				R[15]		
S17					S9					S10	S11					S22	
S18										R[10]	R[10]			R[10]		R[10]	R[10]
S19						S12				R[1]	R[1]	S13	S14			R[1]	R[1]
S20										R[3]	R[3]	R[3]	R[3]			R[3]	R[3]
S21										R[5]	R[5]					R[5]	R[5]
S22										R[8]	R[8]	R[8]	R[8]	R[8]		R[8]	R[8]

5.2.3 应用 SLR 解析表

本节以算式 <UNUM(1)> '*' <UNUM(2)> 为例，演示 SLR 解析方法。解析过程中使用两个栈：状态栈和符号栈，分别用于记录当前分析状态和已识别的符号。在每一步中，根据状态栈栈顶状态以及当前输入符号，查 SLR 解析表以确定下一步操作。具体的解析过程如表 5.2 所示。

表 5.2: 应用 SLR 解析表 5.1 解析乘法算式 <UNUM(1)> '*' <UNUM(2)>。

状态栈	符号栈	待读入词元	操作
S0		<UNUM(1)> '*' <UNUM(2)> <eof>	shift <UNUM(1)>, Goto S7
S0,S7	<UNUM(1)>	'*' <UNUM(2)> <eof>	Reduce [9], back to S0, Goto S5
S0,S5	NUM	'*' <UNUM(2)> <eof>	Reduce [7], back to S0, Goto S4
S0,S4	E3	'*' <UNUM(2)> <eof>	Reduce [6], back to S0, Goto S3
S0,S3	E2	'*' <UNUM(2)> <eof>	Reduce [4], back to S0, Goto S2
S0,S2	E1	'*' <UNUM(2)> <eof>	Shift '*', Goto S13
S0,S2,S13	E1 '*'	<UNUM(2)> <eof>	Reduce [13], back to S2, Goto S12
S0,S2,S12	E1 OP2	<UNUM(2)> <eof>	Shift <UNUM(2)>, Goto S7
S0,S2,S12,S7	E1 OP2 <UNUM(2)>	<eof>	Reduce [9], back to S12, Goto S5
S0,S2,S12,S5	E1 OP2 NUM	<eof>	Reduce [7], back to S12, Goto S4
S0,S2,S12,S4	E1 OP2 E3	<eof>	Reduce [6], back to S12, Goto S20
S0,S2,S12,S20	E1 OP2 E2	<eof>	Reduce [3], back to S0, Goto S2
S0,S2	E1	<eof>	Reduce [2], back to S0, Goto S1
S0,S1	E	<eof>	accept

5.3 更多文法

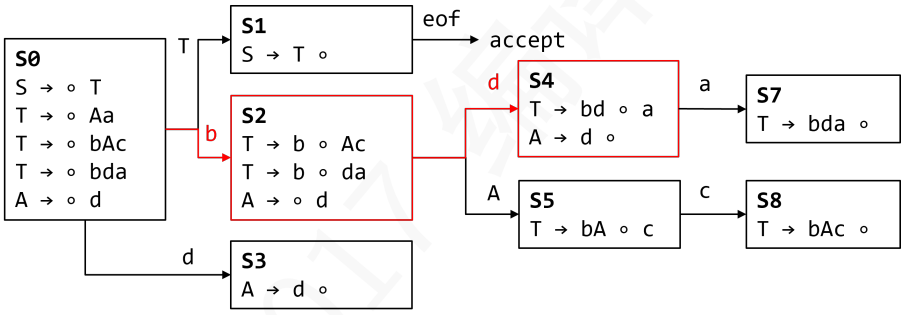
5.3.1 LR(1) 文法

SLR 方法的表达能力较为有限。当构造的 SLR 解析表中某些单元格存在多个分析动作（即出现移进-规约冲突或规约-规约冲突）时，对应的文法就不是 SLR 文法。例如，下列语法规则（文法 5.3）不属于 SLR 文法。

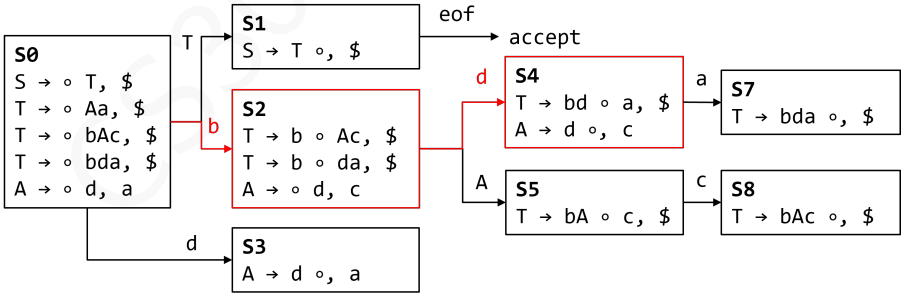
$$\begin{aligned} \texttt{T} &\mapsto \circ\ \texttt{A}\ \texttt{a} \\ \texttt{T} &\mapsto \circ\ \texttt{b}\ \texttt{A}\ \texttt{c} \\ \texttt{T} &\mapsto \circ\ \texttt{b}\ \texttt{d}\ \texttt{a} \\ \texttt{A} &\mapsto \circ\ \texttt{d} \end{aligned} \tag{5.3}$$

图 5.2a 构造了文法 5.3 对应的 LR(0) 有穷自动机。可以看到，在状态 S_4 中（已移进符号 b 和 d），当下一个输入符号为 a 时，同时存在移进和规约两种可能的分析动作，从而产生移进-规约冲突。

对该情况进行分析可以发现，应选择移进操作作为正确的分析动作。这是因为，若按照项目 $\texttt{A} \mapsto \texttt{d}\circ$ 进行规约，则该产生式在原文法中的使用场景要求其后继符号为 c，而不是 a。然而，SLR 方法在确定规约操作时仅依赖于 Follow(A) 集合，其粒度较粗，无法区分当前项目的具体上下文来源，从而导致冲突无法消除。



(a) LR(0) 有穷自动机：解析输入串 “bda” 时存在移进规约冲突



(b) LR(1) 有穷自动机：可以有效解析输入串 “bda”

图 5.2: LR(1)，但非 SLR 文法举例

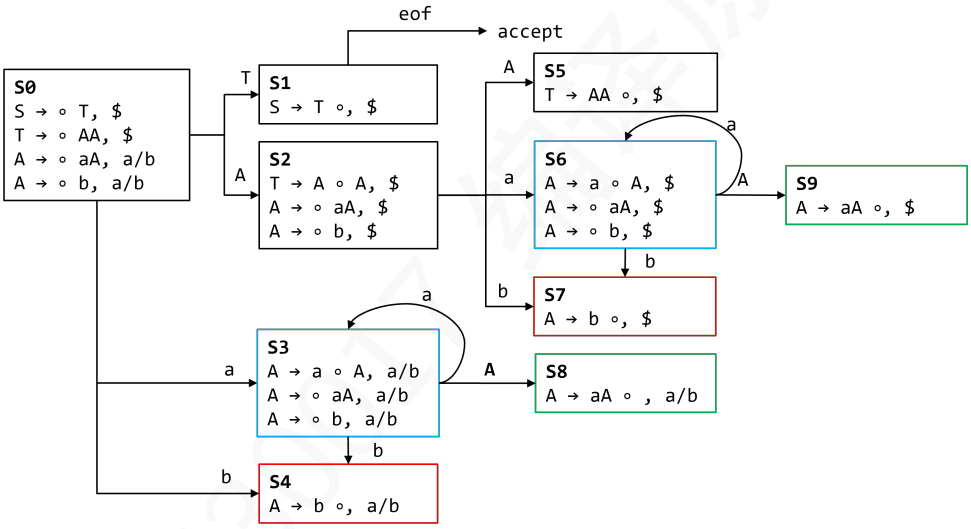
LR(1) 分析方法能够解决上述问题。相比于 SLR 方法，其改进之处在于在构造 LR(1) 有穷自动机时，为每个项目引入一个向前看符号，从而携带更精确的上下文信息。如图 5.2b所示，LR(1) 项目形如 $[\texttt{A} \mapsto \alpha \circ \beta, \texttt{a}]$，其中 a 表示当前项目在后续规约时所允许的输入符号。通过这种方式，可以更精确地限定规约操作的适用范围，从而避免 SLR 方法中由于 Follow 集合过于粗糙而引发的冲突。在构造有穷自动机时，若两个项目集中的项目在向前看符号上存在差异，则将其视为不同的状态，从而避免潜在的分析冲突。

5.3.2 LALR 文法

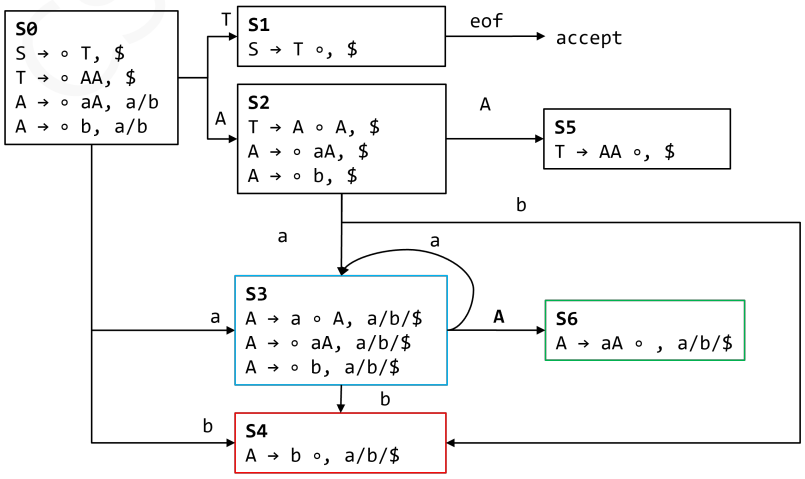
LR(1) 分析方法 [1] 通过在项目中引入前瞻符号，从而避免解析表中的冲突。然而，其缺点是构造得到的项目集族规模较大，相比 SLR 可能会显著增加，尤其在文法较复杂时更为明显。为减小 LR(1) 项目集族规模增长带来的开销，LALR 方法将具有相同核心项的 LR(1) 项目集合并，即在保持项目集内容相同的前提下，合并其前瞻符号集合。因此，LALR 在分析能力上接近 LR(1)，但状态数量大大减少，同时相比 SLR 在上下文信息的利用上更加精细。

$$
\begin{aligned}
\mathtt{S} &\mapsto \circ\ \mathtt{T} \\
\mathtt{T} &\mapsto \circ\ \mathtt{A}\ \mathtt{A} \\
\mathtt{A} &\mapsto \circ\ \mathtt{a}\ \mathtt{A} \\
\mathtt{A} &\mapsto \circ\ \mathtt{b}
\end{aligned}
\tag{5.4}
$$

文法 5.4 展示了一套语法规则。按照 LR(1) 解析表的构造过程，可以得到图 5.3a 所示的 LR(1) 项目集族及其对应的有穷自动机。可以看到，其中存在三对具有相同核心项的项目集：S_3 和 S_6，S_4 和 S_7，以及 S_8 和 S_9。将这些核心项目相同的项目集进行合并，即可得到图 5.3b 所示的 LALR 项目集族。



(a) LR(1) 项目集族对应的有穷自动机



(b) LALR 项目集族对应的自动机

图 5.3: LALR 文法举例

什么情况下 LR(1) 项目集可以合并，即合并后构造的语法分析表仍不存在冲突？通过对该案例的分析可以发现，其关键在于避免合并后引入新的分析冲突。具体而言，首先，合并后的项目集中不存在两个不同的规约项目，从而避免规约-规约冲突；其次，对于包含规约项目的状态，要么不存在移进操作，要么其规约项目的前瞻符号集合与移进符号集合互不相交，从而避免移进-规约冲突。

5.3.3 应用探讨

在实际应用中，LR(1) 文法已经具有较强的表达能力，许多语法分析工具基于 LR(1) 或其变种进行实现。例如，Rust 中的 lalrpop 库[1]主要采用 LR(1) 分析方法。尽管 LR(1) 分析方法能力较强，但仍不能处理所有上下文无关文法。在需要更高通用性的场景中，可以采用 GLR（Generalized LR）方法 [2]。该方法在出现分析冲突时同时保留多个可能的分析路径，并进行并行推进，从而能够处理任意上下文无关文法。GLR 方法可以视为对 LR 分析方法的推广，并可与 LALR、LR(1) 等分析框架结合使用。此外，还存在其它提升分析能力的思路。例如，通过增加向前看符号的数量可以得到 LR(k) 文法 [3]，但其实现复杂度也随之显著增加；或者采用基于动态规划思想的 CYK 算法 [4]。

参考文献

[1] Frank DeRemer, and Thomas Pennello. ”Efficient computation of LALR (1) look-ahead sets.” ACM Transactions on Programming Languages and Systems (TOPLAS), 1982.

[2] Masaru Tomita. “An efficient context-free parsing algorithm for natural languages.” *In International Joint Conference on Artificial Intelligence (IJCAI)*, 1985.

[3] Donald E. Knuth. “On the translation of languages from left to right.” *Information and Control*, 1965.

[4] Daniel H. Younger. “Recognition and parsing of context-free languages in time n3.” *Information and Control*, 1967.

[1] lalrpop: https://lalrpop.github.io/lalrpop/

练习

1) (多选题) 下列文法属于:

(a) LL(1)

(b) SLR

$$
\begin{aligned}
\texttt{S} &\mapsto \texttt{S A} \\
\texttt{S} &\mapsto \texttt{A} \\
\texttt{A} &\mapsto \texttt{a}
\end{aligned}
\tag{5.5}
$$

2) 已知以下上下文无关文法规则，分析该文法是否为 SLR 文法。如果是，请为其构造 SLR 解析表。

$$
\begin{aligned}
\texttt{Regex} &\mapsto \texttt{Regex '|' Concat} \\
\texttt{Regex} &\mapsto \texttt{Concat} \\
\texttt{Concat} &\mapsto \texttt{Concat Closure} \\
\texttt{Concat} &\mapsto \texttt{Closure} \\
\texttt{Closure} &\mapsto \texttt{Closure '*'} \\
\texttt{Closure} &\mapsto \texttt{Item} \\
\texttt{Item} &\mapsto \texttt{'(' Regex ')'} \\
\texttt{Item} &\mapsto \texttt{<Char>}
\end{aligned}
\tag{5.6}
$$

3) 修改文法 5.3，分别达到以下几个目标:

1) 在 SLR 解析表中引入规约-规约冲突;

2) 在 LR(1) 解析表中引入移进-规约冲突;

3) 在 LR(1) 解析表中引入规约-规约冲突。

4) LL(1) 文法一定符合 SLR 文法吗?

Part II

中间层

6 类型推导

本章学习目标：

- ⋆⋆ 了解抽象语法树的概念
- ⋆⋆⋆ 掌握标识符索引化方法
- ⋆⋆⋆ 掌握 Hindley-Milner 类型规则设计与应用方法

6.1 类型系统

类型系统由类型和使用规则组成，用于保证程序在运行过程中不会发生类型错误。

- 类型：指程序中数据的分类，用于描述数据的取值范围及其可执行的操作，主要包括基础类型（如 i32、bool）、复合类型（如数组、结构体）以及函数类型（用于描述函数的参数类型和返回值类型）。
- 规则：指对类型的使用方式所施加的约束，包括类型检查、类型推导以及类型之间的兼容性与转换规则，用于确保不同类型的数据被正确、安全地使用。

根据类型检查发生的时间，类型系统可分为静态类型系统和动态类型系统：静态类型系统通常在编译阶段确定标识符类型并完成类型检查；动态类型系统则在程序运行过程中确定类型并进行类型检查。根据类型约束的严格程度或是否允许隐式类型转换，类型系统还可以分为强类型系统和弱类型系统。

Tea 语言是一种编译型语言，其类型系统设计为静态类型系统，支持的类型如表 6.1 所示。

表 6.1: Tea 语言类型分类

类型类别	具体类型	说明
基础类型	i32	整型标量类型
	bool	仅作为中间计算结果存在
复合类型	数组	同类型元素的集合
	结构体	用户自定义数据类型
函数类型	$T_1, T_2 \to T$	描述参数类型与返回类型

为了提高编程的简洁性，Tea 语言允许在部分场景中省略标识符类型标注，并由编译器自动进行类型推导。主要规则如下：

- 局部变量声明中允许省略类型，由编译器进行类型推导；
- 函数声明中，参数类型与返回值类型必须显式指定；
- 结构体字段类型必须显式声明。

类型推导一般是基于抽象语法树进行的，包括两个步骤：1）标识符索引化；2）根据类型规则从抽象语法树中提取类型约束并求解。下面分别进行讲解。

6.2 抽象语法树

语法解析的结果是语法解析树 (Parse Tree 或 Concrete Syntax Tree)。以代码 6.1为例，其对应的语法解析树如图 6.1所示。可以看出，语法解析树中包含了大量对后续分析和编译的冗余信息，因此可以对其进行进一步的抽象与化简。

```
let g:i32 = 10;
fn fib(x:i32) -> i32 {
    if (x <= 1) {
        return x;
    }
    let a = fib(x - 1);
    let b = fib(x - 2);
    let r = a + b;
    return r;
}
fn main() {
    let r = fib(10) + g;
}
```

代码 6.1: Tea 语言代码示例

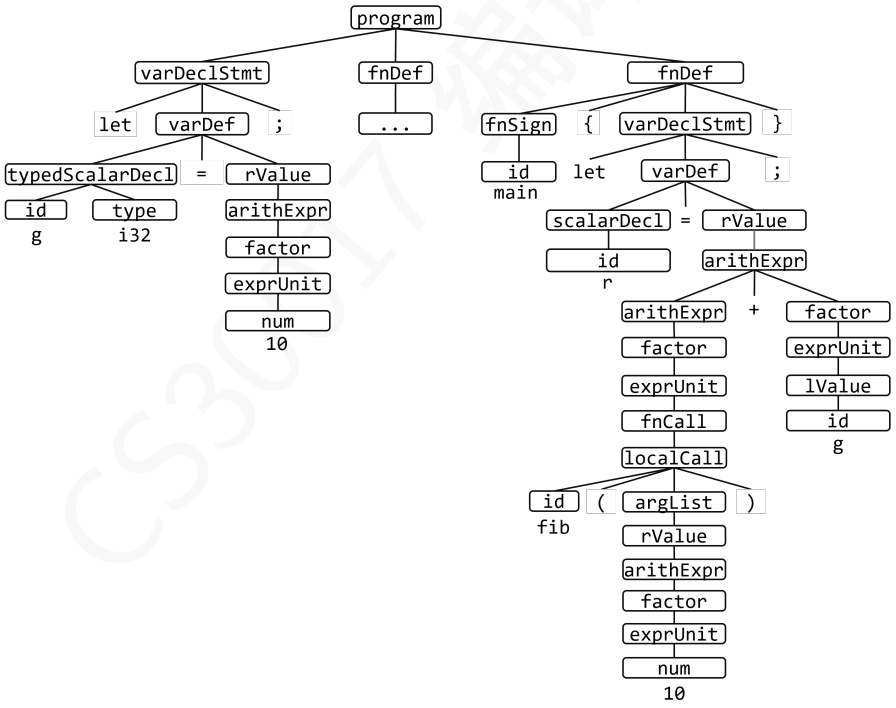


图 6.1: 代码 6.1对应的语法解析树示例（省略了 fib 函数体）

抽象语法树（Abstract Syntax Tree，AST）是语法解析树化简后的树形中间代码表示形式，在整个编译过程中可能会被多次编辑，记录代码分析和编译过程的中间结果。图 6.2展示了代码 6.1对应的一种 AST 表示。该 AST 去除了语法解析树中的括号和分号等冗余节点，并且对单一展开形式（只有一个孩子节点）的情况进行了合并处理，如将 `rValue->arithExpr->factor->exprUnit->num` 缩短为 `num`。

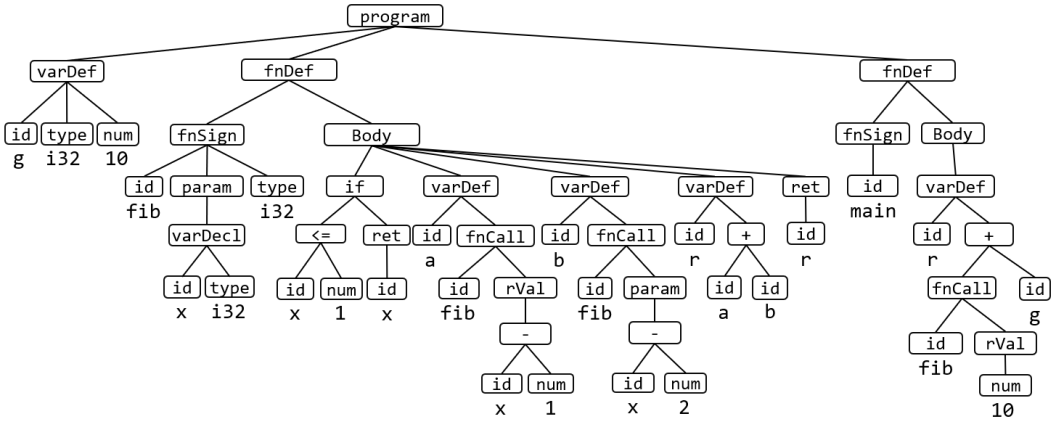


图 6.2: 代码 6.1对应的抽象语法树示例

6.3 标识符索引化

标识符索引化的目的是解决代码中的标识符指代问题。当标识符的指代对象唯一时，该问题较易处理，可以直接将其声明与使用建立对应关系；当标识符的指代对象不唯一时，则需要结合作用域等代码上下文信息进行消歧，并通过符号解析确定其具体指代对象。

Tea 语言对全局标识符在文件中的声明与使用顺序不作限制，但对象标识符的命名需遵循如下规则：

- 两个同名局部变量的作用域不能存在交集；
- 局部变量可以与全局标识符（包括变量和函数）同名。

标识符索引化的输出结果包括若干符号表以及索引化后的抽象语法树。需要注意的是，该符号表可能不包含完整的类型信息；从本质上看，类型推导过程可以视为作为标识符补充类型标注的过程。

6.3.1 创建符号表

符号表记录所有标识符的作用域和已知类型信息，每一行对应一个索引项。通过扫描 AST 中的变量和函数声明（或定义）节点，可以直接得到符号表。

```
let g:i32 = 10;
fn fib(x:i32) -> i32 { // scope fib
    if (x <= 1) {
        return x;
    }
    let a = fib(x - 1); // {  scope 1
        let b = fib(x - 2); // { scope 2
            let r = a + b; // { scope 3
                return r;
            // }
        // }
    // }
}
fn main() { // scope main
    let r = fib(10) + g; // { scope 1
    // }
}
```

代码 6.2: Tea 语言代码示例：补充了标识符作用域标注

符号表通常分为全局符号表和局部变量符号表。以代码 6.2 为例，其符号表包括一个全局符号表（见表 6.2 ）以及两个函数对应的局部变量符号表（见表 6.3 和表 6.4 ）。需要注意的是，函数参数本质上也属于局部变量的范畴。此外，在构建符号表时，无需考虑变量或函数的使用节点，仅需处理其声明信息。

表 6.2: 代码 6.2 对应的全局符号表

标识符	作用域 (辅助信息)	索引	类型
g	global	0x0000	i32
fib	global	0x0100	(i32) → i32
main	global	0x0101	(void) → void

表 6.3: 代码 6.2 中 main 函数对应的局部变量符号表

标识符	作用域 (辅助信息)	索引	类型
r	main:scope1	0x1000	未知

表 6.4: 代码 6.2 中 fib 函数对应的局部变量符号表

标识符	作用域 (辅助信息)	索引	类型
x	fib	0x1100	i32
a	fib:scope1	0x1101	未知
b	fib:scope2	0x1102	未知
r	fib:scope3	0x1103	未知

6.3.2 添加标识符索引

该步骤为 AST 中的每个标识符添加索引信息。在实际编译器实现中，该过程通常与符号表的构建同步进行：在遇到标识符声明时创建新的索引，在遇到标识符引用时则关联已有索引。

图 6.3 对该问题进行了抽象表示，其中红色节点表示局部变量的声明，蓝色节点表示标识符的引用。索引化的难点在于需要严格结合标识符的作用域进行分析。

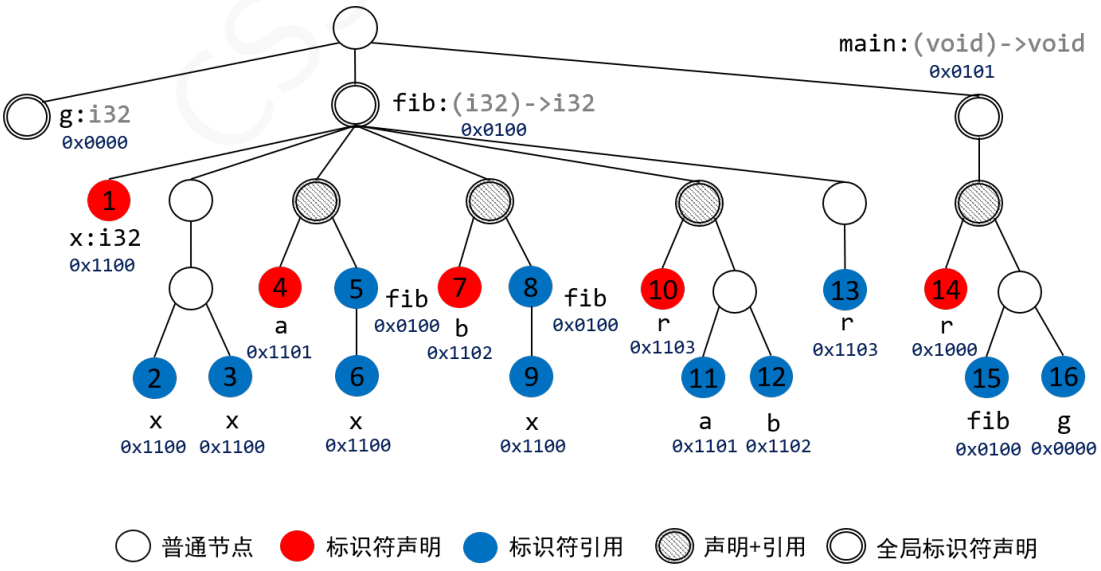


图 6.3: 标识符索引化问题

假设全局符号表已经创建完毕，算法 8 给出了对于函数内部的标识符进行索引化的具体思路。其核心思想是为每个函数维护一个标识符字典 `dict`，用于记录当前作用域中可见的标识符。在遍历 AST 时，对每个中间节点维护一个局部字典 `subdict`，用于记录该子树中新声明的标识符；当遍历退出该节点作用域时，需要将 `subdict` 中的标识符从 `dict` 中移除，从而恢复外层作用域状态。

算法 8 函数局部变量标识符索引化算法

Input: AST root of a function; Global symbol table: *gdict*

```
let dict = gdict // all usable identifiers of the function
procedure INDEXING(cur)
    subdict = ∅; // identifiers defined in the current subtree;
    for each child ∈ cur.children do // left to right visit in order;
        match child.type :
            case VarDecl ⇒ // declaration node
                dict.add(child.id); // add to the dictionary; If already existed, report error;
                subdict.add(child.id); // add to the sub dictionary;
            case VarRef ⇒ // reference node
                child.refid.index = dict.getIndex(child.refid); //this step may fail; or return none if not existed;
            case VarDeclRef ⇒ // declaration and reference that may reference multiple vars, e.g., d = a + b;
                for refid ∈ child.refids do
                    refid.index = dict.getIndex(refid); //this step may fail; or return none if not existed;
                end for
                dict.add(child.id); // add to the dictionary; If already existed, report an error;
                subdict.add(child.id); // add to the sub dictionary;
            case OtherLeafNode ⇒ // other leaf node that has no identifiers
                Continue;
            case NonLeafNode ⇒ // for intermediate nodes: recursively indexing the subtree;
                Indexing(child);
        end match
    end for
    for each entry ∈ subdict do // remove the identifiers defined in the current subtree;
        dict.remove(entry);
    end for
end procedure
```

6.4 类型约束和求解

接下来，需要为符号表中缺省类型的标识符确定具体类型。常用的方法是基于约束求解的 Hindley-Milner 类型推导方法 [1]。该方法主要包括以下三个步骤：

1) 为不同语句类型定义相应的类型约束规则；

2) 根据上述规则从程序中提取类型约束；

3) 对类型约束进行求解，从而确定标识符的具体类型。

6.4.1 类型约束规则

表 6.5 定义了 Tea 语言语言的主要类型约束规则。

表 6.5: Tea 语言中的主要类型约束规则

代码模式	**类型约束**	**含义**
`X: Ty`	$[\![X]\!] = \text{Ty}$	声明 X 的类型为 Ty
`I`	$[\![I]\!] = \text{i32}$	数字类型为 i32
`X[I]: Ty`	$[\![X]\!] = \&\text{Ty}$, $[\![I]\!] = \text{i32}$	声明 X 数组的类型为 &Ty
`{I1,...,In}`	$[\![I_1, ..., I_n]\!] = \&\text{i32}$	数组类型为 &i32
`{I; N}`	$[\![I; N]\!] = \&\text{i32}$	数组类型为 &i32
`X = Y`	$[\![X]\!] = [\![Y]\!]$	等号左右节点类型相同
`X = Y[Z]`	$[\![Z]\!] = \text{i32}$, $[\![X]\!]=[\![*Y]\!]$, $[\![Y]\!]=\&[\![*Y]\!]$	数组解引用作为右值
`X[Z] = Y`	$[\![Z]\!] = \text{i32}$, $[\![X]\!]= \&[\![Y]\!]$	数组解引用作为左值
`X bArithOp Y`	$[\![X]\!] = [\![Y]\!] = [\![X\ bArithOp\ Y]\!]$	二元算数运算操作数和运算结果同类型
`X bRelOp Y`	$[\![X]\!] = [\![Y]\!]$, $[\![X\ bRelOp\ Y]\!] = \text{bool}$	二元关系运算操作数同类型，结果为 bool
`if(X) {...}`	$[\![X]\!] = \text{bool}$	条件为 bool
`while(X) {...}`	$[\![X]\!] = \text{bool}$	条件为 bool
`X bLogOp Y`	$[\![X]\!] = [\![Y]\!] = [\![X\ bLogOp\ Y]\!] = \text{bool}$	二元逻辑运算操作数和结果均为 bool
`uLogOp X`	$[\![X]\!] = [\![uLogOp\ X]\!] = \text{bool}$	一元逻辑运算操作数和结果均为 bool

`F(X: Ty1)-> Ty2 {`
`  return Y;`
`}`
	$[\![F]\!] = (\text{Ty1}) \rightarrow \text{Ty2}$, $[\![Y]\!] = \text{Ty2}$	函数声明/定义涉及的类型约束

`F(X)`	$[\![F]\!] = ([\![X]\!]) \rightarrow [\![F(X)]\!]$	函数调用的类型约束

`struct ST {`
`  A:Ty1,`
`  B:Ty2`
`}`
	$[\![ST]\!] = (\text{Ty1, Ty2})$	结构体类型

`X.A = Y`	$[\![X.A]\!] = [\![Y]\!]$, $[\![X]\!] = [\![X.A, \_]\!]$	结构体 field 类型

注：符号 $[\![X]\!]$ 表示标识符 X 的类型

将上述规则应用到代码 6.1 的 AST 中，可以得到类型约束。

$$
\begin{aligned}
&\texttt{let g:i32 = 10}\\
&\Rightarrow [\![0x0000]\!] = [\![10]\!],\ [\![0x0000]\!] = i32,\ [\![10]\!] = i32\\
&\texttt{fn fib(x:i32) -> i32}\\
&\Rightarrow [\![0x0100]\!] = (i32) \to i32,\ [\![0x1100]\!] = i32\\
&\texttt{if (x <= 1)}\\
&\Rightarrow [\![0x1100 <= 1]\!] = bool,\ [\![0x1100]\!] = [\![1]\!],\ [\![1]\!] = i32\\
&\texttt{return x}\\
&\Rightarrow [\![0x1100]\!] = i32\\
&\texttt{let a = fib(x - 1)}\\
&\Rightarrow [\![0x1101]\!] = [\![0x0100(0x1100 - 1)]\!]\\
&\Rightarrow [\![0x0100]\!] = ([\![0x1100 - 1]\!]) \to [\![0x0100(0x1100 - 1)]\!]\\
&\Rightarrow [\![0x1100 - 1]\!] = [\![0x1100]\!] = [\![1]\!]\\
&\texttt{let b = fib(x - 2)}\\
&\Rightarrow [\![0x1102]\!] = [\![0x0100(0x1100 - 2)]\!]\\
&\Rightarrow [\![0x0100]\!] = ([\![0x1100 - 2]\!]) \to [\![0x0100(0x1100 - 2)]\!]\\
&\Rightarrow [\![0x1100 - 2]\!] = [\![0x1100]\!] = [\![2]\!]\\
&\texttt{let r = a + b}\\
&\Rightarrow [\![0x1103]\!] = [\![0x1101 + 0x1102]\!]\\
&\Rightarrow [\![0x1101 + 0x1102]\!] = [\![0x1101]\!] = [\![0x1102]\!]\\
&\texttt{return r}\\
&\Rightarrow [\![0x1103]\!] = i32\\
&\texttt{fn main()}\\
&\Rightarrow [\![0x0101]\!] = (void) \to void\\
&\texttt{let r = fib(10) + g}\\
&\Rightarrow [\![0x1000]\!] = [\![0x0100(10) + 0x0000]\!]\\
&\Rightarrow [\![0x0100(10) + 0x0000]\!] = [\![0x0100(10)]\!] = [\![0x0000]\!]\\
&\Rightarrow [\![0x0100]\!] = ([\![10]\!]) \to [\![0x0100(10)]\!]
\end{aligned}
\tag{6.1}
$$

由于上述类型约束均为等价关系，可采用并查集方法进行求解，从而得到：$[\![0x1000]\!] = i32$, $[\![0x1101]\!] = i32$, $[\![0x1102]\!] = i32$, $[\![0x1103]\!] = i32$。若类型系统中引入子类型或范型机制，则类型约束关系将由等价关系扩展为包含关系，此时需要采用更一般的约束求解方法。

需要注意的是，类型推导不一定总是存在解。例如，当代码中出现结构体递归定义时，通常会导致约束不可解，此时应报告类型推导错误。若类型约束集合存在解，则说明程序是可类型的（typable）；否则，编译器应报告类型错误，或在语言允许的情况下通过隐式类型转换消解部分不一致。因此，类型推导在一定程度上也实现了类型检查；而类型检查可以视为在类型已知前提下的类型推导特例。

参考文献

[1] Robin Milner. “A theory of type polymorphism in programming.” *Journal of Computer and System Sciences*, 1978.

练习

1) 为什么要基于 AST 而非源代码进行类型推导或检查?

2) 按步骤为下列 Tea 语言代码进行类型推导:

   (a) 画出 AST;

   (b) 创建符号表;

   (c) 提取类型约束并求解。

```
fn fac(n:i32) -> i32 {
    let r = 1;
    while (n>0) {
        r = r * n;
        n = n-1;
    }
    return r;
}
```

代码 6.3: Tea 语言代码

3) 思考: 若对 Tea 语言的类型系统规则进行扩展，以支持以下特性，应如何相应地修改类型推导方法?

- 允许同名局部变量的作用域发生重叠，并在标识符引用时优先选择作用域最内层的定义;
- 允许函数重名（函数重载），但要求其函数签名互不相同。

7 线性 IR

本章学习目标：

- ⋆⋆⋆ 熟悉 LLVM IR
- ⋆⋆⋆ 能够将 Tea 语言代码翻译为 LLVM IR
- ⋆ 了解解释执行

7.1 线性 IR

本章将系统介绍一种线性中间表示（Intermediate Representation, IR）的定义方式及其基本使用方法。这里所采用的 IR 是 LLVM IR [1] 的一个简化子集，专门为 Tea 语言的编译过程设计。选择 LLVM IR 作为中间表示具有多方面优势。首先，LLVM 生态成熟，拥有完善的工具链支持，使得 IR 不仅是中间产物，同时也是一个可观察、可执行、可优化的程序表示形式。例如，可以通过现有的 C 语言编译器 `clang` 将高级语言程序转换为 LLVM IR，从而直观理解语言结构与底层表示之间的映射关系；同时，还可以借助解释器 `lli` 直接执行 IR 代码，方便调试和验证编译结果。这种可执行的中间表示大大降低了编译器开发与教学的门槛。

```
; 生成中间代码hello.ll
clang -emit-llvm -S hello.c
; 执行中间代码
lli hello.ll
```

代码 7.1: 生成与执行 LLVM IR 的基本命令

代码 7.2 给出了一个简单的 LLVM IR 示例。该示例包含一个全局变量声明 `@g`，以及两个函数定义 `@foo` 和 `@main`。通过这个例子，可以初步观察 LLVM IR 的基本结构：包括全局区、函数定义、指令序列以及显式的内存操作。

```
@g = global i32 10                ; 声明全局变量@g：类型为i32，初始值为10
define i32 @foo(i32 %0) {         ; 定义函数foo：类型为i32->i32，参数为%0
    %x = alloca i32               ; 在栈上分配i32空间，返回指针%x
    store i32 %0, ptr %x          ; 将%0写入%x指向的内存
    %g0 = load i32, ptr @g        ; 加载全局变量@g的内容到%g0
    ret i32 %g0                   ; 返回%g0
}
define i32 @main() {              ; 定义函数foo：类型为 void->i32
    %r0 = call i32 @foo(i32 1)    ; 调用函数foo，返回值保存为%r0
    ret i32 %r0                   ; 返回%r0
}
```

代码 7.2: LLVM IR 代码示例

接下来，我们将围绕 Tea 语言所使用的 IR 子集，对相关指令和核心概念进行逐一讲解。

7.1.1 类型

LLVM IR 是一种强类型语言，每一条指令都需要显式标注数据类型。Tea 语言中涉及的主要类型包括以下几类：

- **标量类型**：表示基本数值数据，主要是不同位宽的有符号整数，如`i32`（32 位整数）、`i8`（8 位整数）和`i1`（1 位布尔值）。
- **指针类型**：用于表示内存地址。在 LLVM 17 之后，引入了统一的`ptr`类型，不再需要区分具体指向的数据类型（例如`i32*`）。
- **数组类型**：用于表示定长序列，例如`[2 x i32]`表示包含两个`i32`元素的数组。
- **自定义类型**：可以通过`type`关键字定义结构体类型，例如`%mytype = type {i32, i32}`。

7.1.2 标识符

在 LLVM IR 中，变量名、函数名以及基本块名称统称为标识符，并分为两类：

- **局部标识符**：以`%`开头，仅在当前函数内部有效，例如`%r1`或`%0`。局部标识符既可以表示临时变量，也可以表示基本块标签（如`%bb1`）。

  需要特别注意的是，当局部标识符采用纯数字形式（如`%0`、`%1`等）时，必须满足以下约束：

  – 编号必须从`%0`开始；
  – 编号必须连续递增，不能跳号或重复；
  – 所有局部标识符（包括变量和基本块标签）共享同一编号空间，即它们的编号是统一递增的，而不是分别独立编号。

  如果违反上述规则，解释器 `lli` 可能无法正确解析或执行 IR 代码。

- **全局标识符**：以`@`开头，在整个程序范围内可见，例如`@g`或`@main`。全局标识符用于表示全局变量或函数名称，其命名不受局部编号规则的限制。

LLVM IR 采用一种重要的设计原则：静态单赋值形式（SSA, Static Single Assignment）。在这种形式下，每个变量只能被赋值一次，即每个标识符只会出现在一条定义语句的左侧。这种约束使得数据流关系更加清晰，也为后续优化提供了便利。关于 SSA 的详细内容将在后续章节展开。

7.1.3 内存分配和数据存取

本节主要讨论函数栈帧上的内存分配与访问机制。在 LLVM IR 中，内存操作是显式的：与高级语言中“变量即存储”的抽象不同，IR 不会隐式为变量分配空间，而是需要通过指令明确地完成“分配—写入—读取”的全过程。

基本机制 局部变量通常分配在函数栈帧中，其内存空间通过`alloca`指令申请。该指令返回一个指向所分配内存的指针。对内存的访问则通过`store`和`load`指令完成：前者用于写入数据，后者用于读取数据。需要注意的是`alloca`按字节分配内存，因此通常不会用于`i1`这样的位级类型。

基本示例 代码 7.3 展示了典型的“分配—写入—读取”流程：

```
; <result> = alloca <type> [, <num_elements>] [, align <alignment>]
; 在栈上分配内存，返回指向该内存的指针
%x = alloca i32               ; 分配一个 i32 空间，返回指针 %x

; store <type> <value>, ptr <pointer> [, align <alignment>]
; 将 value 写入 pointer 指向的内存
store i32 1, ptr %x           ; 向 %x 指向的地址写入整数 1

; <result> = load <type>, ptr <pointer> [, align <alignment>]
; 从 pointer 指向的内存读取数据
%t1 = load i32, ptr %x        ; 从 %x 读取 i32，结果存入 %t1
```

代码 7.3: LLVM IR 代码示例：内存分配和数据存取

变量与临时值 在 Tea 语言的源程序中，每个变量在 IR 中通常对应一块内存单元，需要通过`load/store`访问。而在 LLVM IR 中，还会引入大量临时变量（如`%t1`、`%t2`），这些变量不对应具体内存位置，而更类似于“虚拟寄存器”，用于保存中间计算结果。

类型转换 LLVM IR 提供了显式的类型转换指令，用于在不同位宽之间转换数据。例如：

- `zext`（zero extension）：将小类型扩展为大类型，高位补零；
- `trunc`（truncate）：将大类型截断为小类型，仅保留低位。

```
; <result> = zext <from_type> <value> to <to_type>
; 零扩展（zero extend）：将较小整数类型扩展为较大整数类型，高位补 0
%t2 = zext i1 %t1 to i32          ; 将 i1 扩展为 i32，高位补 0，结果存入 %t2

; <result> = trunc <from_type> <value> to <to_type>
; 截断（truncate）：将较大整数类型转换为较小整数类型，保留低位，丢弃高位
%t3 = trunc i32 %t2 to i8         ; 将 i32 截断为 i8，仅保留低 8 位
```

代码 7.4: LLVM IR 代码示例：类型转换

复合类型的访问 对于数组和结构体等复合类型，内存访问不仅仅是“读/写”，还需要先进行地址计算。在 LLVM IR 中，这一过程通过`getelementptr`（GEP）指令完成，然后再结合`load/store`进行数据访问。可以将`getelementptr`理解为一种“基于类型信息的地址偏移计算”：它不会访问内存，而只是根据给定的类型结构和索引，计算出目标元素的地址。

代码 7.5展示了数组元素访问的两种典型情况。一种是基于元素指针的线性访问，另一种是基于数组类型的分层访问。

```
; <result> = getelementptr <ty>, ptr <ptrval>{, <idx_ty> <idx>}*
; ===== 情况1：线性内存（指向单一元素类型）=====
; 在%p的基础上按i32大小偏移1个元素，得到下一个元素地址
%t1 = getelementptr i32, ptr %p, i32 1
; 读取该地址中的值
%t2 = load i32, ptr %t1

; ===== 情况2：数组对象访问 =====
; 在栈上分配一个长度为10的i32数组
%a = alloca [10 x i32]
; 第一个0：从数组对象本身开始（不跳过数组）
; 第二个1：访问数组下标为1的元素（即第2个元素）
%t3 = getelementptr [10 x i32], ptr %a, i32 0, i32 1
; 将99写入该元素位置
store i32 99, ptr %t3
```

代码 7.5: LLVM IR 代码示例：数组元素存取

当 GEP 操作的是“指向元素类型的指针”（如`ptr %p`指向`i32`）时，只需要一个索引，表示按元素大小进行线性偏移。当操作的是“数组类型指针”（如`ptr %a`指向`[10 x i32]`）时，需要两个索引：第一个索引通常为`0`，表示从当前数组对象本身开始；第二个索引表示数组中具体元素的下标。因此，`getelementptr [10 x i32], ptr %a, i32 0, i32 1`的含义是：在数组`%a`中，取第 1 号元素（即第二个元素）的地址。

结构体的访问方式与数组类似，同样通过 GEP 进行逐层定位。代码 7.6给出了示例。其中，第一个索引`0`表示从结构体对象本身开始；第二个`0`表示访问结构体的第 0 个字段。

```
; 定义结构体类型：包含两个i32字段
%mystruct = type { i32, i32 }
; 在栈上分配一个结构体对象，返回指针%st
%st = alloca %mystruct
; 第一个0：从结构体对象本身开始
; 第二个0：访问第0个字段（第一个成员）
%t1 = getelementptr %mystruct, ptr %st, i32 0, i32 0
; 将整数1写入该字段
store i32 1, ptr %t1
```

代码 7.6: LLVM IR 代码示例：结构体域数据存取

可以看出，数组和结构体在 GEP 中的访问方式本质是一致的：都是通过“逐层索引”的方式，在类型结构中逐级定位目标元素。GEP 支持多层级的嵌套索引，可用于访问多维数组或嵌套结构体等复杂数据结构。更详细的说明可参考官方文档[1]。

[1] https://llvm.org/docs/GetElementPtr.html

7.1.4 算数运算

Tea 语言使用的 LLVM IR 中的算数运算指令均为有符号数运算，包括add、sub、mul 和 sdiv，不涉及无符号数运算。为了简化讨论，本教材暂不考虑整数运算溢出的情况。

```
; <res> = add <res type> <oprand 1>, <oprand 2>
; oprand1和oprand2也必须和<res type>一致
%t3 = add i32 %t1, %t2      ; 加法运算: %t3 = %t1 + %t2
%t4 = sub i32 %t1, %t2      ; 减法运算: %t4 = %t1 - %t2
%t5 = mul i32 %t1, %t2      ; 乘法运算: %t5 = %t1 * %t2
%t6 = sdiv i32 %t1, %t2     ; 有符号的除法运算: : %t6 = %t1 / %t2
```

代码 7.7: LLVM IR 代码示例：算数运算

7.1.5 关系运算

IR 中支持的关系运算指令是icmp，可通过参数设置区分不同的比较模式。

```
; <res> = icmp <mod> <operand type> <oprand1>, <oprand2>
; <mod>是比较模式，包括: eq, neq, sgt, sge, slt, sle
%t3 = icmp eq i32 %t1, %t2      ; 等于
%t4 = icmp neq i32 %t1, %t2     ; 不等于
%t5 = icmp sgt i32 %t1, %t2     ; 大于
%t6 = icmp sge i32 %t1, %t2     ; 大于等于
%t7 = icmp slt i32 %t1, %t2     ; 小于
%t8 = icmp sle i32 %t1, %t2     ; 小于等于
```

代码 7.8: LLVM IR 代码示例：比较运算

7.1.6 控制流

控制流描述程序执行过程中基本块（basic block）之间的跳转关系。在 LLVM IR 中，基本块以标识符加冒号定义，例如“bb1:”。每个基本块必须以一条“终结指令”（terminator）结束，例如 br 或 ret，用于显式指定控制流的去向。

LLVM IR 中最常见的跳转指令是 br，分为无条件跳转和条件跳转两种形式：

```
bb0: ; 定义代码块 bb0
    ; br label <dst block> ; 无条件跳转
    br label %bb1

bb1: ; 定义代码块 bb1
    %t3 = icmp eq i32 %t1, %t2
    ; br i1 <cond>, label <true block>, label <false block> ; 条件跳转
    br i1 %t3, label %bb0, label %bb2

bb2: ; 定义代码块 bb2
    ...
```

代码 7.9: LLVM IR 代码示例：控制流

此外，LLVM IR 中还有一条与控制流密切相关的指令 phi，用于在不同控制流路径汇合时选择变量的值。该指令将在下一章介绍静态单赋值（SSA）形式时详细讲解，这里仅给出基本形式：

```
; <res> = phi <type> [<value 1>, <label 1>], [<value 2>, <label 2>], ...
; 若来自 <label 1>, 则取 <value 1>; 若来自 <label 2>, 则取 <value 2>
%t3 = phi i32 [%t1, %bb1], [%t2, %bb2]
```

代码 7.10: LLVM IR 代码示例：phi 指令

7.1.7 逻辑运算

LLVM IR 中没有专门的逻辑运算指令。逻辑运算可以通过位运算指令xor、and和or来实现。

```
; 实现逻辑非运算：%b = !%a
; <res> = xor <type> <operand 1> <operand 2>
%b = xor i1 %a, true ;
; 实现逻辑与运算：%r = %b && %a
; <res> = and <type> <operand 1> <operand 2>
%r = and i1 %a, %b
; 实现逻辑或运算：%r = %b || %a
; <res> = or <type> <operand 1> <operand 2>
%r = or i1 %a, %b
```

代码 7.11: LLVM IR 代码示例：通过位运算实现逻辑运算

此外，逻辑“与”和“或”运算通常通过控制流指令以短路方式实现等效功能。

```
bb1:
    br i1 %a, label %bb2, label %bb3
bb2:
    br label %bb3
bb3:
    %r = phi i1 [false, %bb1], [%b, %bb2]
```

代码 7.12: LLVM IR 代码示例：通过控制流指令实现%a && %b

```
bb1:
    br i1 %a, label %bb3, label %bb2
bb2:
    br label %bb3
bb3:
    %r = phi i1 [true, %bb1], [%b, %bb2]
```

代码 7.13: LLVM IR 代码示例：通过控制流指令实现%a || %b

7.1.8 函数

在 LLVM IR 中，定义函数使用`define`语句；如果仅声明该函数，则使用`declare`语句。在同一个 LLVM IR 文件中，不允许对同一个函数同时进行声明和定义。如果需要在一个 IR 文件中调用另一个 IR 文件中定义的函数，应先在当前 IR 文件中进行声明，并使用 llvm-link 工具进行链接。函数调用相关的指令主要包括调用指令`call`和返回指令`ret`。

```
; define <return type> <function ID> (<arg1 type> <arg1>, <arg2 type> <arg2>) {...}
define i32 @foo(i32 %0) {    ; 定义函数foo：类型是i32->i32
    ret i32 %0
}
; declare <return type> <function ID> (<arg1 type> <arg1>, <arg2 type> <arg2>)
declare void @bar(i32 %0)    ; 声明函数bar：类型是i32->void
define i32 @main() {         ; 定义函数main：类型是void->i32
    ; <return value> = call <return type> <function ID>(<arg type> <arg value>)
    %r0 = call i32 @foo(i32 1)
    ; ret <return type> <return value>
    ret i32 %r0
}
```

代码 7.14: LLVM IR 代码示例：函数声明、定义和调用

7.2 AST 翻译线性 IR

将 AST 翻译为 IR 代码的整体思路如下：

1) 遍历顶层 AST，生成函数及全局变量的 IR 表示；

2) 对每个函数的 AST 进行递归下降遍历，构建基本块（代码块）及其跳转关系；

3) 遍历各个基本块，逐条翻译其中的语句为 IR 指令。

该翻译过程的主要难点体现在两个方面：其一是基本块的划分及其控制流关系的构建，其二是指令参数的定义-使用（def-use）关系的处理。

7.2.1 基本块的构建及跳转关系

在 LLVM IR 中，每个基本块必须以终结指令（如`br`或`ret`）结束。因此，在递归下降遍历 AST 的过程中，需要在控制流发生分叉或回跳的位置显式划分基本块，并建立清晰的跳转关系。主要包括以下几种结构：

- **函数定义**：创建入口基本块`%bb0`，并预先插入返回指令 `ret <type> %tobeDetermined` 作为占位。
- **if 语句**：创建三个基本块：`%bb-true`、`%bb-false` 以及后继块`%bb-after`。在当前基本块中插入条件分支指令：`br i1 %tobeDetermined, label %bb-true, label %bb-false`。同时，将当前基本块中原有的终结指令移动至`%bb-after`，作为其终结指令。在`%bb-true` 和`%bb-false` 中分别添加无条件跳转至`%bb-after` 的指令。
- **while 语句**：创建三个基本块：`%bb-cond`（条件判断块）、`%bb-body`（循环体）以及后继块`%bb-after`。在当前基本块中插入跳转至`%bb-cond` 的指令，并将原有终结指令移动至`%bb-after`。在`%bb-cond` 中生成条件分支指令：`br i1 %tobeDetermined, label %bb-body, label %bb-after`；在`%bb-body` 中添加回跳至`%bb-cond` 的指令以形成循环。

在 `while` 结构中，还需要额外考虑 `break` 和 `continue` 语句对控制流的影响。`continue` 语句应直接跳转至`%bb-cond`，以开始下一轮循环；`break` 语句则跳转至后继块`%bb-after`，从而退出循环。因此，在构建循环体内部基本块时，需要根据语句类型动态确定其后继基本块。

需要注意的是，`break` 和 `continue` 的影响范围取决于其所在位置：当它们出现在嵌套的内层 `while` 中时，仅影响该内层循环的控制流，对外层结构没有影响；而当它们出现在当前 `while` 内部的 `if` 分支中时，会改变该 `if` 语句后续基本块的控制流走向（例如提前跳转至`%bb-cond` 或`%bb-after`），从而影响后续基本块的构建。因此，在实现时需要结合语法结构，对不同层级的跳转目标进行区分和维护。

伪代码 9展示了函数内部的代码块生成逻辑。该设计能够正确表达 Tea 语言中的控制流结构，包括 `if-else` 与 `while` 的嵌套情况。在实现时，基本块编号建议采用`%bb` 后接递增数字的形式；不推荐使用纯数字编号，以避免编号不连续导致的问题，从而影响 `lli` 的正确执行。

算法 9 基于 AST 的基本块构建与 IR 生成

input: AST statement $stmt$
output: IR in basic blocks
procedure GenIR($stmt, curBB, breakTarget, continueTarget$)　▷ 在当前基本块 $curBB$ 中为语句 $stmt$ 生成 IR
    match $stmt$:
        case If($cond, thenStmt, elseStmt$) ⇒
            $trueBB \leftarrow$ newBB()　▷ then 分支
            $falseBB \leftarrow$ newBB()　▷ else 分支
            $afterBB \leftarrow$ newBB()　▷ 汇合块
            $v \leftarrow$ GenExpr($cond$, $curBB$)　▷ 计算条件表达式，得到 i1 值
            emit($curBB$, br(v, $trueBB$, $falseBB$))　▷ 根据条件跳转
            GenIR($thenStmt, trueBB, breakTarget, continueTarget$)　▷ 生成 then 分支
            emit_if_no_term($trueBB$, br($afterBB$))　▷ 若未终结，则跳转到汇合块
            if $elseStmt \neq$ null **then**
                GenIR($elseStmt$, $falseBB, breakTarget, continueTarget$)　▷ 生成 else 分支
                emit_if_no_term($falseBB$, br($afterBB$))　▷ 保证 else 分支正确终结
            else
                emit($falseBB$, br($afterBB$))　▷ 无 else 时直接跳转
            end if
        case While($cond, body$) ⇒
            $condBB \leftarrow$ newBB()　▷ 条件判断块
            $bodyBB \leftarrow$ newBB()　▷ 循环体
            $afterBB \leftarrow$ newBB()　▷ 循环结束后
            emit($curBB$, br($condBB$))　▷ 进入循环前跳转到条件判断
            $v \leftarrow$ GenExpr($cond$, $condBB$)　▷ 在条件块中计算条件
            emit($condBB$, br(v, $bodyBB$, $afterBB$))　▷ 根据条件进入循环体或退出
            GenIR($body, bodyBB, afterBB, condBB$)　▷ 更新 break/continue 目标
            emit_if_no_term($bodyBB$, br($condBB$))　▷ 若未终结，则回跳形成循环
        case Break ⇒
            emit($curBB$, br($breakTarget$))　▷ 跳转到当前循环的退出块
        case Continue ⇒
            emit($curBB$, br($continueTarget$))　▷ 跳转到当前循环的条件判断块
        case _ ⇒
            translate($stmt$, $curBB$)　▷ 普通语句在线性附加到当前基本块
    end match
end procedure

7.2.2 指令参数的定义和使用

在翻译每条 IR 指令时，首先需要明确其操作数的来源。理想情况下，应尽量复用已经保存在寄存器中的计算结果，而不是频繁地从局部变量中通过 load 指令重新读取。然而，实际情况中，参数往往定义于不同的基本块，且可能存在多重定义（如控制流汇合处），若在 IR 翻译阶段直接处理这些问题，将显著增加实现复杂度。

因此，在当前阶段的翻译过程中，我们暂不考虑跨基本块的优化，而是将参数的定义与使用关系限制在单个基本块内部，避免直接依赖来自其它基本块的寄存器值。具体策略如下：

- 所有局部变量在使用前必须先通过 load 指令读入寄存器；
- 所有局部变量在更新后立即通过 store 写回内存。

代码 7.15 与 7.16 分别给出了阶乘函数的 Tea 语言源代码及其对应的 IR 表示。这种方式虽然牺牲了一定的执行效率，但能够有效简化实现逻辑，并保证语义的正确性。

```
fn fac(n:i32) -> i32 {
    let r = 1;
    while (n>0) {
        r = r * n;
        n = n-1;
    }
    return r;
}
```

代码 7.15: Tea 语言代码

```
define i32 @foo(i32 %0) {
bb0:
    %n = alloca i32 ; 参数内存单元
    %r = alloca i32
    store i32 %0, ptr %n ; 保存参数值
    store i32 1, ptr %r
    br label %bb1
bb1:
    %t1 = load i32, ptr %n ; 使用变量的值前先load，限制临时变量%t1仅在当前代码块使用
    %t2 = icmp sgt i32 %t1, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t3 = load i32, ptr %r ; 使用变量的值前先load，避免与其它代码块中的%r值耦合
    %t4 = load i32, ptr %n ; 使用变量的值前先load，避免与其它代码块中的%n值耦合
    %t5 = mul i32 %t3, %t4 ; 限制临时变量%t5仅在当前代码块使用
    store i32 %t5, ptr %r ; 立即更新%r的内存单元，保证后续指令可以load到最新的数值
    %t6 = load i32, ptr %n
    %t7 = sub i32 %t6, 1
    store i32 %t7, ptr %n ; 立即更新%n的内存单元，保证后续指令可以load到最新的数值
    br label %bb1
bb3:
    %t8 = load i32, ptr %r
    ret i32 %t8
}
```

代码 7.16: 代码 7.15 对应的 IR

7.3 解释执行

线性 IR 通过消除 `if-else` 和 `while` 等语法结构，将程序转换为由基本块（Basic Block）和显式跳转指令组成的形式，其结构已经非常接近底层汇编代码。因此，可以从程序入口函数开始，按照控制流顺序逐条解释执行 IR 指令。

指令加载与解析 在执行之前，需要将文本形式的 LLVM IR 加载并解析为内部数据结构。解析过程首先对 IR 进行词法与语法分析，识别每条指令的组成部分。例如，`%t5 = mul i32 %t3, %t4` 可以解析为：

- 指令（opcode）：`mul`
- 类型信息：`i32`
- 操作数：`%t3`、`%t4`
- 结果寄存器：`%t5`

在此基础上，需要进一步按基本块对指令进行组织。每个基本块包含一段顺序执行的指令序列，并以终结指令（如 `br`、`ret`）结束。从而方便解释执行过程中根据标签（如 `bb1`、`bb2`）进行跳转寻址。

虚拟机 解释执行的核心问题在于：如何保存并传递指令执行过程中产生的中间结果。为此，需要引入虚拟机来模拟程序运行环境。根据执行模型的不同，虚拟机通常可以分为两类：

- **栈虚拟机（Stack-based VM）**：操作数通过操作数栈隐式传递，指令通常不显式给出操作数位置。例如在 `Java Bytecode` 或 `Python` 字节码中，`add` 指令会从栈顶弹出两个操作数并将结果压回栈中。这类模型实现简单，但指令间的数据依赖较为隐式。
- **寄存器虚拟机（Register-based VM）**：操作数显式存储在“寄存器”中，指令直接引用这些寄存器。LLVM IR 本质上是一种寄存器虚拟机模型，其 SSA 变量（`%t1`, `%t2` 等）可以看作数量无限的虚拟寄存器。因此，在实现解释器时，更自然的方式是采用寄存器模型，而非栈模型。

在解释执行过程中，虚拟机通常需要维护以下几类状态：

- **寄存器**：用于存储 SSA 变量的当前值，可实现为“变量名 → 数值”的映射表。
- **内存模型**：对应 `alloca` 分配的栈空间，可以用线性内存或哈希表模拟地址到数值的映射。
- **函数栈桢**：每次函数调用时创建新的栈桢，包含局部变量、参数以及返回地址等信息。
- **程序计数器**：指示当前执行的基本块及其内部指令位置。

基于上述结构，解释执行的基本流程为：从入口基本块开始，顺序执行指令；遇到普通指令时更新寄存器或内存状态；遇到跳转指令时根据条件切换基本块；遇到函数调用指令时创建新的栈桢并进入被调函数执行；遇到返回指令时结束当前函数执行并将结果返回给调用者。该过程在语义上与真实机器执行类似，但所有状态均由虚拟机在软件中维护。

参考文献

[1] LLVM 语言参考文档-指令部分, `https://llvm.org/docs/LangRef.html#instruction-reference`.

练习

1) 使用控制流指令，通过短路求值方式改写以下代码中的 and 指令，并使用 lli 工具验证其执行结果。

```
define i32 @foo(i32 %0, i32 %1) {
    %t1 = alloca i32
    %t2 = alloca i32
    store i32 %0, ptr %t1
    store i32 %1, ptr %t2
    %t3 = load i32, ptr %t1
    %t4 = load i32, ptr %t2
    %t5 = icmp sgt i32 %t3, %t4
    %t6 = load i32, ptr %t1
    %t7 = icmp ne i32 %t6, 0
    %t8 = and i1 %t5, %t7
    %t9 = zext i1 %t8 to i32
    ret i32 %t9
}
define i32 @main() {
    %1 = call i32 @foo(i32 2, i32 1)
    ret i32 %1
}
```

代码 7.17: LLVM IR 代码片段

2) 将下列 Tea 语言代码翻译为线性 IR，并使用 lli 工具进行测试。

```
let a[i32;10] = {1,3,5,7,9,2,4,6,8,10};
fn binsearch(x:i32) -> i32 {
    let high:i32 = 9;
    let low:i32 = 0;
    let mid:i32 = (high + low)/2;
    while a[mid] != x && low < high {
        mid = (high + low) / 2;
        if x < a[mid] {
            high = mid-1;
        } else {
            low = mid +1;
        }
    }
    if x == a[mid] {
        return mid;
    }
    else {
        return -1;
    }
}
fn main() -> i32 {
    let r = binsearch(2);
    return r;
}
```

代码 7.18: Tea 语言代码片段

8 静态单赋值

本章学习目标：

- ⋆⋆ 了解静态单赋值形式
- ⋆⋆⋆ 掌握基于混沌迭代的数据流分析方法
- ⋆⋆⋆ 掌握静态单赋值形式的构造方法

8.1 静态单赋值

静态单赋值（SSA, Static Single Assignment） [1] 是一种特殊的线性中间表示（IR）。其核心目标是更清晰地刻画变量的定义—使用（def-use）关系，从而便于后续的程序分析与代码优化。SSA 通常满足以下要求：

- **单次定义：**每个标识符对应一个虚拟寄存器，且只能被定义（def）或赋值一次；若变量值发生变化，则必须重新定义新的标识符以表示更新后的值。
- **Phi 指令：**当控制流汇合导致某变量在同一使用点（use）可能对应多个不同来源的定义时，需要使用 phi 指令统一表示这些可能的取值来源。
- **最优化：**应尽可能减少 phi 指令的数量，以简化数据流关系并降低 IR 的冗余。

上一章中使用的 LLVM IR 已经满足“标识符仅定义一次”的要求，但尚未引入 phi 指令。在不同控制流路径产生不同变量值时，我们采用的是 `store-load` 机制进行处理，而非使用 phi 指令。接下来，我们将讨论如何将上一章 LLVM IR 中基于 `load/store` 的表示方式逐步转换为基于 phi 指令的表示，并最终构造出最优的 SSA 形式。

8.2 基于冗余消除的 SSA 构造方法

8.2.1 消除 IR 中冗余的 load/store

在 AST 翻译为 IR 的过程中，为了降低 def-use 关系分析的复杂度，我们要求变量在使用前必须先执行 load，并在变量值更新后立即执行 store。这种策略虽然简化了前端生成逻辑，但也会引入大量冗余的 load/store 指令。本节将采用基于混沌迭代（Chaotic Iteration）的数据流分析方法，对 IR 中冗余的 load/store 操作进行消除，从而为后续构造 SSA 形式奠定基础。

```
define i32 @fac(i32 %0) {
bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %0, i32* %n
    store i32 1, i32* %r
    br label %bb1
bb1:
    %t1 = load i32, i32* %n
    %t2 = icmp sgt i32 %t1, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t3 = load i32, i32* %r
    %t4 = load i32, i32* %n
    %t5 = mul i32 %t3, %t4
    store i32 %t5, i32* %r
    %t6 = load i32, i32* %n
    %t7 = sub i32 %t6, 1
    store i32 %t7, i32* %n
    br label %bb1
bb3:
    %t8 = load i32, i32* %r
    ret i32 %t8
}
```

代码 8.1: IR 代码示例：阶乘函数

消除冗余 load

代码 ?? 展示了一段 IR。由于变量 n 的值在代码块 bb1 中已经被加载到虚拟寄存器 %t1，且在到达代码块 bb2 的过程中没有发生对 n 的更新，因此 bb2 中再次将 n 加载到 %t4 和 %t6 的操作是冗余的，可以直接复用 %t1。其基本规律是：对于同一变量，若两次 load 之间不存在对应的 store 操作，则说明该变量的值未发生变化，因此后一次 load 为冗余操作，可以直接使用前一次 load 得到的虚拟寄存器；反之，若两次 load 之间存在 store，则说明变量值已经被更新，之前的 load 结果失效，必须重新加载。

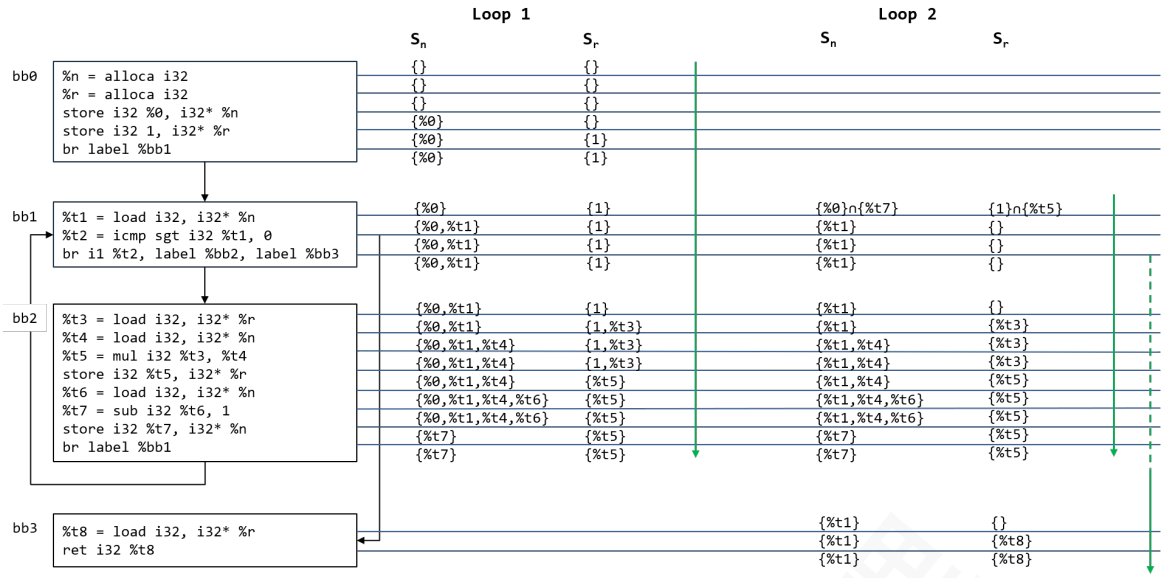


图 8.1: 冗余 load 指令分析

基于上述分析，我们可以总结出与该优化相关的指令及其影响，即表 8.1 中定义的 transfer 函数。对于线性的指令序列，可以直接将这些 transfer 函数依次作用于代码块中的各条指令；然而，当程序中存在控制流分支或循环时，还需要额外的数据流分析机制进行处理。

表 8.1: Transfer 函数定义：可用 load 指令分析

IR 指令	举例	Transfer 函数
`load`	`%t = load i32, i32* %x`	$S_x = S_x \cup \{t\}$
`store`	`store i32 %t, i32* %x`	$S_x = \{t\}$

混沌迭代（Chaotic Iteration）算法是一种处理控制流数据流分析的经典框架，可以根据具体分析任务设计不同的数据流状态与 transfer 操作。如算法 10 所示，该算法对每条指令 `i` 进行分析并计算其对应的 `OUT[i]`。若某程序点存在多个前驱节点，则需要对各前驱的分析结果进行合并。由于本节进行的是可用表达式类型的 Must Analysis，因此只有当某个变量对应的虚拟寄存器在所有前驱路径上均一致可用时，才能认为其在当前程序点可用，因此这里采用交集操作进行合并。将该算法应用于图 8.1 后，便可以求得每个程序节点处各变量对应的可用虚拟寄存器，从而为后续消除冗余 `load` 指令提供依据。

算法 10 混沌迭代算法：可用 load 指令分析

Require: IR and variables of a target function

for each $i \in irs$ **do**
  $IN[i] \leftarrow \{S_v = \emptyset \mid v \in Var\}$;
  $OUT[i] \leftarrow \{S_v = \emptyset \mid v \in Var\}$;
end for
repeat
  for each $i \in irs$ **do**
    for each $p \in \text{Predecessor}(i)$ **do**
      $IN[i] \leftarrow IN[i] \cap OUT[p]$;
    end for
    $OUT[i] \leftarrow \text{Transfer}(i)$;
  end for
until $IN[i]$ and $OUT[i]$ stop changing for all i

消除冗余 store

如果同一变量的两条 `store` 指令之间不存在对应的 `load` 操作，则前一条 `store` 的结果不会被读取，因此该 `store` 属于冗余操作，可以直接删除。与冗余 `load` 分析不同，该分析仅需维护一个“已经被 `store` 但尚未被 `load`”的变量集合，而无需为每个变量分别维护对应的可用虚拟寄存器集合。表 8.2 总结了不同指令对应的 transfer 函数。

表 8.2: Transfer 函数定义：可用 store 分析

IR 指令	举例	Transfer 函数
`store`	`store i32 %t, i32* %x`	$S = S \cup \{x\}$
`load`	`%t = load i32, i32* %x`	$S = S \setminus \{x\}$
`alloca`	`%x = alloca i32`	$S = S \setminus \{x\}$

算法 11 混沌迭代算法：可用 store 分析

Require: IR and variables of a target function

1: **for each** $i \in irs$ **do**
2: $IN[i] \leftarrow \emptyset$;
3: $OUT[i] \leftarrow \emptyset$;
4: **end for**
5: **repeat**
6: **for each** $i \in irs$ **do**
7: **for each** $s \in$Successor(i) **do**
8: $OUT[i] \leftarrow OUT[i] \cap IN[s]$;
9: **end for**
10: $IN[i] \leftarrow$ Transfer(i);
11: **end for**
12: **until** $IN[i]$ and $OUT[i]$ stop changing for all i

根据算法 11 对 IR 控制流图进行逆向遍历，即可识别所有满足条件的冗余 `store` 操作。由于图 8.2 中不存在“store 后未被读取便再次 store”的情况，因此其中不包含冗余 `store` 指令。

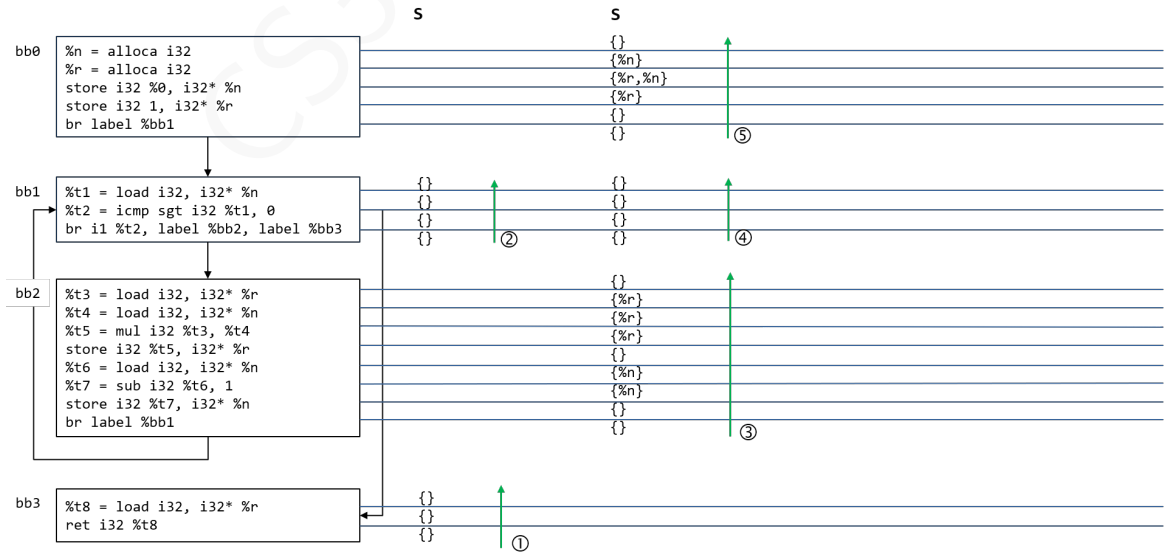


图 8.2: 冗余 load 指令分析

8.2.2 转换为静态单赋值形式

这一步的目标是消除 IR 中所有针对局部变量的 `store/load` 指令，使变量值完全通过虚拟寄存器传递，从而构造纯 SSA 形式的 IR。其中的关键问题在于：某些 `load` 指令可能对应来自不同控制流路径的多个定义，此时无法直接使用单一虚拟寄存器替换，而需要引入 phi 指令对不同来源的值进行统一表示。

以图 8.3 为例，代码块 `bb1` 中的 `load`（`%n`）可能对应两种不同的定义来源：

- 若控制流路径为 `bb0` → `bb1`，则其值来源于 `bb0` 中的 `%0`；
- 若控制流路径为 `bb2` → `bb1`，则其值来源于 `bb2` 中的 `%t7`。

由于该 `load` 的取值依赖于控制流路径，因此需要使用 phi 指令对这些可能的定义进行合并。下面介绍从 LLVM IR 转换到 SSA 形式的方法，其主要分为两个步骤：

1) 值流（value-flow）分析：识别每个 `load` 指令在不同控制流路径下可能对应的变量定义来源；

2) 使用 phi 指令替换 `store/load`。

值流分析

对于值流分析，我们同样采用混沌迭代（Chaotic Iteration）方法分析 `store` 指令对 def-use 关系的影响。具体而言，通过正向遍历控制流图，在遇到 `store` 指令时应用表 8.3 中定义的 transfer 函数；当遇到控制流合并节点时，则对来自不同前驱路径的分析结果取并集。经过迭代直至达到不动点后，便可以得到各程序点处变量可能对应的定义来源。图 8.3 展示了该分析的结果。

表 8.3: Transfer 函数定义：值流分析

IR 指令	举例	Transfer 函数
`store`	`store i32 %t, i32* %x`	$S_x = \{t\}$

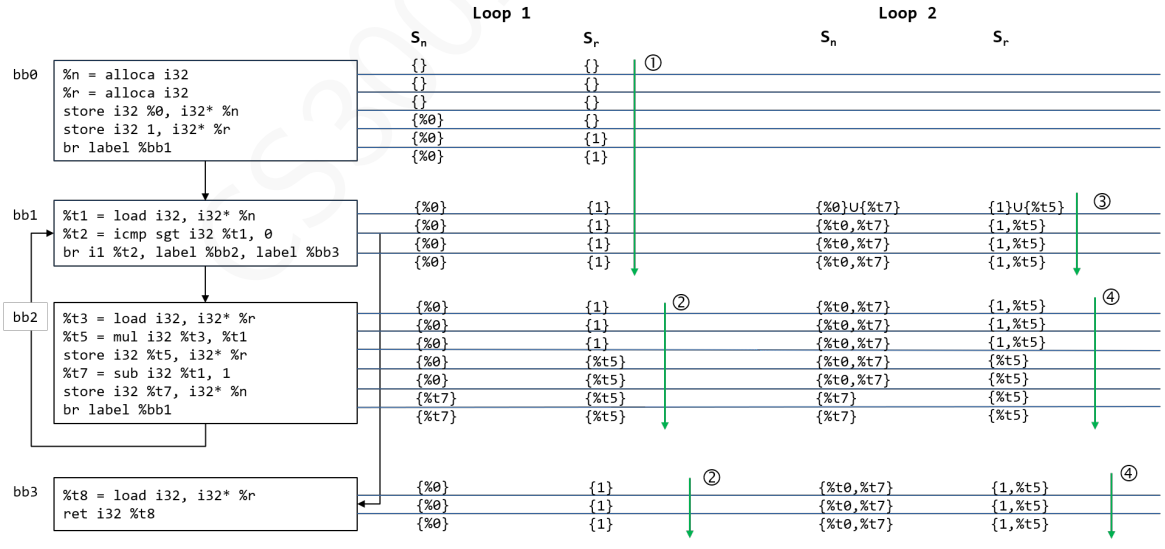


图 8.3: 数值流分析

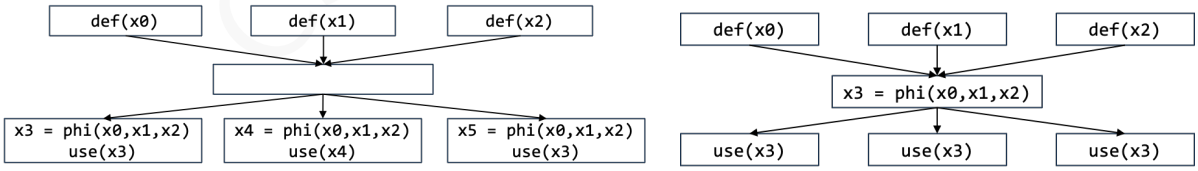
使用 phi 指令替换 store-load

确定了每个程序节点可能对应的变量数值定义后，只需在存在多个定义来源的程序点插入 phi 指令，即可使用虚拟寄存器替换原有的 `store/load` 操作。对于图 8.3 而言，变量 `n` 在代码块 `bb1` 中同时可能来源于 `bb0` 和 `bb2`，因此必须在 `bb1` 中插入 `phi(n)`。变量 `r` 在代码块 `bb1` 中也存在不同来源，但在 `bb1` 中并未被使用，因此其对应的 phi 指令既可以在 `bb1` 中插入，也可以延迟到 `bb2`, `bb3` 再插入。显然，在 `bb1` 中同时插入 `phi(n)` 和 `phi(r)` 更为理想。这种方式能够统一循环入口处的变量定义关系，使 IR 结构更加规整。最终得到的 SSA 形式如代码 8.2 所示。

```
define i32 @fac(i32 %0) {
bb0:
    br label %bb1
bb1:
    %n0 = phi i32 [%0 %bb0], [%t7:%bb2];
    %r0 = phi i32 [1 %bb0], [%t5:%bb2];
    %t2 = icmp sgt i32 %n0, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t5 = mul i32 %r0, %n0
    store i32 %t5, i32* %r
    %t7 = sub i32 %n0, 1
    store i32 %t7, i32* %n
    br label %bb1
bb3:
    ret i32 %r0
}
```

代码 8.2: 代码 8.1的静态单赋值形式

值得注意的是，虽然纯寄存器形式的 IR 能够将变量的 def-use 关系显式表示出来，但这并不意味着其 def-use 关系一定足够简洁。若 phi 指令插入位置不合理，仍可能导致 def-use 关系数量快速增长。为了获得更优的 phi 指令组织形式，应尽量在靠近控制流汇合起点的位置提前插入 phi 指令。以图 8.4a 为例，在直接使用寄存器表示后，其 def-use 关系数量达到 3×3，并且会随着控制流深度增加呈指数级增长。若将 phi 指令前移，如图 8.4b 所示，则 def-use 关系数量可降低为 3+3，从而有效避免 def-use 关系的指数爆炸问题。



(a) 非最优 phi 指令放置　　(b) 最优 phi 指令放置

图 8.4: Phi 指令放置位置与 def-use 关系的优化举例

上述基于冗余消除的 SSA 构造方法虽然可行，但在实际编译器中较少直接使用。其主要原因在于，该方法难以自然地得到较优的 phi 指令布局，尤其是在复杂控制流下，phi 指令的数量与位置往往不够理想，容易导致冗余的 def-use 关系和额外的优化开销。因此，当函数控制流较为复杂时，通常还需要进一步设计专门的 phi 优化机制，以获得更加紧凑和高效的 SSA 形式。

8.3 基于支配边界的 SSA 构造方法

对于 phi 指令放置位置的确定，实际编译器中更常采用基于支配边界（Dominance Frontier）的 SSA 构造方法。与前述“先进行冗余消除、再逐步引入 phi 指令”的思路不同，该方法首先根据控制流图的支配关系确定 phi 指令的插入位置，再通过数据流分析与变量重命名过程更新 IR 中的虚拟寄存器使用关系，从而直接构造出较为规范且高效的 SSA 形式。

8.3.1 支配边界

定义 4（支配）**.** 给定有向图 $G(V,E)$ 与起始节点 v_0，若从 v_0 到节点 v_j 的所有路径都必须经过节点 v_i，则称 v_i 支配（dominate）v_j，记作 $v_i \in Dom(v_j)$。若进一步满足 $v_i \neq v_j$，则称 v_i 严格支配 v_j，记作 $v_i \in IDom(v_j)$。

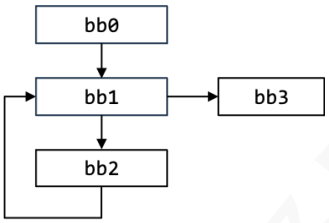


图 8.5: 控制流图举例

支配关系可以通过基于混沌迭代的数据流分析方法计算得到。具体而言，通过正向遍历控制流图，并维护从起始节点到当前节点路径上所有必经的代码块集合；当遇到具有多个前驱的控制流合并节点时，对各前驱路径的结果取交集。

以图 8.5 为例，各节点对应的支配节点集合如下：

$$Dom(bb_0) = \{bb_0\}$$

$$Dom(bb_1) = \{bb_0, bb_1\}$$

$$Dom(bb_2) = \{bb_0, bb_1, bb_2\}$$

$$Dom(bb_3) = \{bb_0, bb_1, bb_3\}$$

定义 5（支配边界）**.** 节点 v_i 的支配边界（Dominance Frontier）定义为所有满足以下条件的节点 v_j 的集合：

- v_i 支配 v_j 的某个前驱节点；
- v_i 不严格支配 v_j。

设节点 $v_j \in V$ 的前驱节点集合为 P_j，支配节点集合为 $Dom(v_j)$，则支配边界可通过集合关系直接计算得到。即对于任意 $v_p \in P_j$，以及任意：

$$v_i \in Dom(v_p) \backslash IDom(v_j)$$

都有：

$$v_j \in DF(v_i)$$

图 8.5 中各节点对应的支配边界如下：

$$DF(bb_0) = \emptyset$$

$$DF(bb_1) = \{bb_1\}$$

$$DF(bb_2) = \{bb_1\}$$

$$DF(bb_3) = \emptyset$$

在 SSA 构造中，若某节点对变量 x 进行了赋值，则需要在该节点的支配边界上插入 $\phi(x)$。以代码 8.1 为例，由于 bb_2 的支配边界为 bb_1，且 bb_2 中对变量 r 和 n 进行了赋值，因此应在 bb_1 中插入 $\phi(r)$ 与 $\phi(n)$。

8.3.2 更新 def-use 关系

在确定了各程序点需要插入的 phi 指令之后，需要进一步更新 IR 中的 def-use 关系，从而完成 SSA 形式的构造。该过程可以视为一种基于控制流图的值传播与重命名过程：沿控制流图进行遍历，为每个变量维护其当前对应的虚拟寄存器。

具体过程如下：

- 进入基本块时，若该块包含 phi 指令（位于控制流合并处），则根据各前驱路径上的变量取值，为 phi 指令补充对应的参数；
- 在基本块内部，当遇到新的定义（包括普通指令和 phi 指令）时，更新该变量当前对应的虚拟寄存器；在该过程中，每个程序点上每个变量至多对应一个虚拟寄存器，从而保证单赋值性质；
- 在使用变量时，直接引用当前记录的虚拟寄存器；

最终，通过上述过程可以完全消除原有的 `load/store` 指令，并建立起基于虚拟寄存器的显式 def-use 关系，从而得到规范的 SSA 形式。

参考文献

[1] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. “An efficient method of computing static single assignment form.” *In ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1989.

练习

1) 分析图 8.6 中各基本块的支配边界。

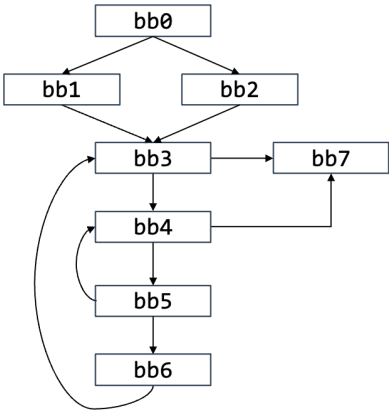


图 8.6: 控制流图

2) 代码 8.7 给出了 Eratosthenes 质数筛选算法的 IR。请按照以下步骤将其转换为 SSA 形式：

   1) 计算各基本块的支配边界；

   2) 在合适位置插入 phi 指令；

   3) 更新虚拟寄存器的 def-use 关系。

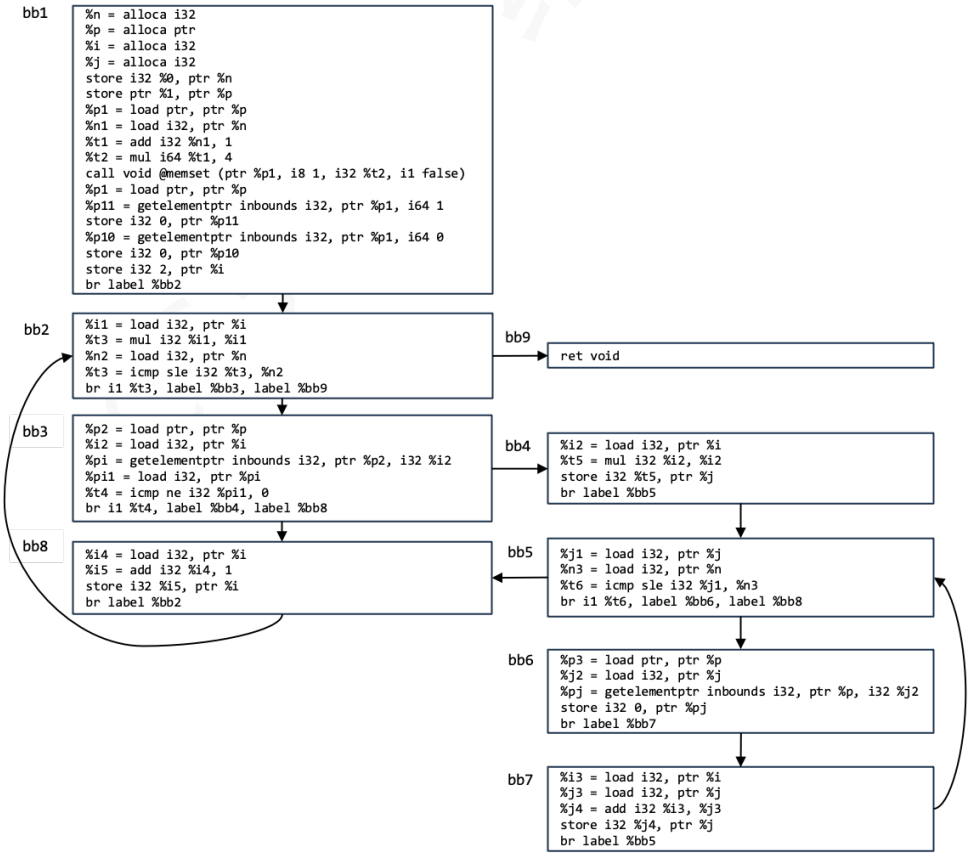


图 8.7: 质数筛选算法对应的 IR

9 过程内优化

本章学习目标：

- ⋆⋆ 掌握常量分析优化方法
- ⋆⋆ 掌握冗余代码优化方法
- ⋆⋆ 掌握循环优化方法

9.1 概述

代码优化是一个复杂的过程，由面向特定优化模式的若干个编译流程（pass）组成。这些流程有的工作在 IR 层面，有的则是工作在汇编代码层面。其中工作在 IR 层面的优化方法是与具体指令集无关的通用优化方法，具有更好的普适性。LLVM 编译器中提供了很多 IR 层面的优化流程 [1]。本章内容探讨其中常见的一些针对单个函数的代码优化模式，暂不考虑跨函数的情况。

9.2 基于常量分析的优化

9.2.1 常量分析

常量分析的目的是找出在某一程序节点，某一变量或寄存器是否为固定不变的特定值。该分析任务可以通过上一章学习的循环迭代分析方法完成，即前向遍历控制流图中的每条语句，并维护已识别的常量标识符信息；如果遇到合并节点取交集即可。

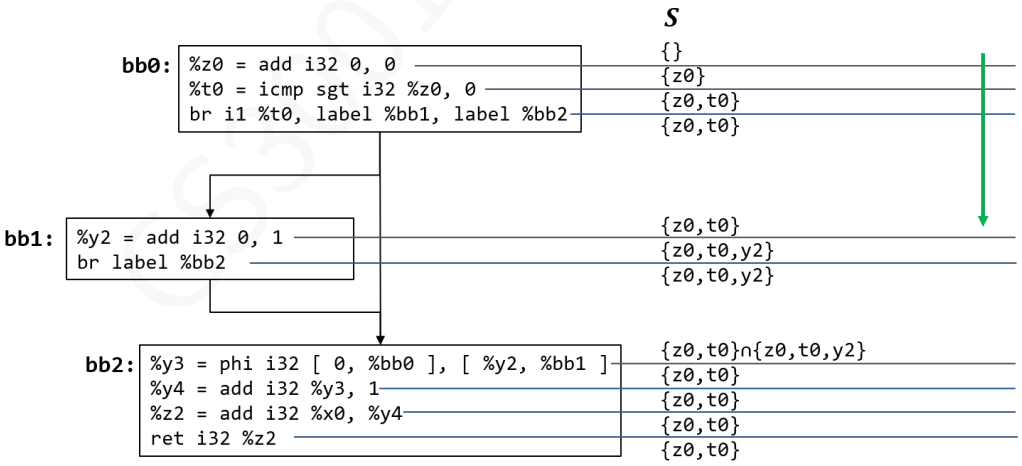


图 9.1: 面向 LLVM IR 的常量分析

以图 9.1 中 SSA 形式的 LLVM IR 为例，我们可以采用表 9.1 中定义的 Transfer 函数分析每一个程序节点的常量标识符集合。SSA 形式的常量分析比较简单，一个虚拟寄存器一旦被识别为常量，则不会存在因其值发生变化而成为变量的情况。非 SSA 形式的代码还需要考虑该虚拟寄存器或变量是否会被重新赋值。

表 9.1: Transfer 函数定义：常量分析

IR 指令	举例	Transfer 函数
add/sub/mul/sdiv	%r = add i32, op1, op2	$S = S \cup \{r\},\ op1 \in S \cup N$ and $op2 \in S \cup N$
xor/and/or	%r = xor i32, op1, op2	$S = S \cup \{r\},\ op1 \in S \cup N$ and $op2 \in S \cup N$
icmp	%r = icmp sgt i32, op1, op2	$S = S \cup \{r\},\ op1 \in S \cup N$ and $op2 \in S \cup N$
zext/trunc	%r = zext i8 op1 to i32	$S = S \cup \{r\},\ op1 \in S \cup N$

注：S 为常量标识符集合；N 为常数。

9.2.2 常量分析应用

常量分析的直接应用是通过使用具体数值替换虚拟寄存器（常量传播），从而在编译时完成某些计算（常量折叠），以减少运行时开销。常量传播算法可以在常量分析算法的基础上进行适当改进实现，具体方法是，在维护常量标识符集合的同时，记录每个标识符对应的具体数值或计算方式。

当二元运算中仅有一个操作数为常量时，虽然不能进行常量折叠，但可以考虑是否存在指令合并的可能性。常见的指令合并情况是：指令 I_1 的一个操作数为常量，另一个为变量；指令 I_2 的一个操作数为常量，另一个为指令 I_1 的运算结果。此时，可以对指令 I_2 的操作数进行优化。代码 9.1 展示了一个指令合并样例。

```
%x1 = add i32 %x0, 1 ; 如果%x1没有被使用，则可以删除该指令
%x2 = add i32 %x1, 2 ; 优化结果：%x2 = add i32 %x0, 3
```

代码 9.1: 指令合并示例

9.3 冗余代码优化

9.3.1 无效代码优化

程序 IR 中可能包含一些指令或虚拟寄存器，其运行结果不会被后续指令使用，则这些指令都是冗余的，带来无谓的运行时开销，应当将其删除。以代码 9.1 为例，对其进行指令合并优化后，计算%x1 的指令很可能成为多余的。这类无效代码优化可以通过活跃性分析实现，即后向分析控制流图，如果遇到 IR 指令将某虚拟寄存器作为其操作数，则将该虚拟寄存器标记为活跃，直至其被定义为止。如果一个虚拟寄存器在不活跃状态下被定义，则定义该寄存器的 IR 指令很可能是冗余的。例外情况是该虚拟寄存器作为函数调用返回值时，由于函数调用会有副作用，不能将其删除。

9.3.2 死代码优化

比较典型的死代码是不可达代码块，即条件恒为真或假的条件跳转语句。以图 9.1 为例，对其进行常量传播优化后，发现 bb1 是不可达的，应当删除。

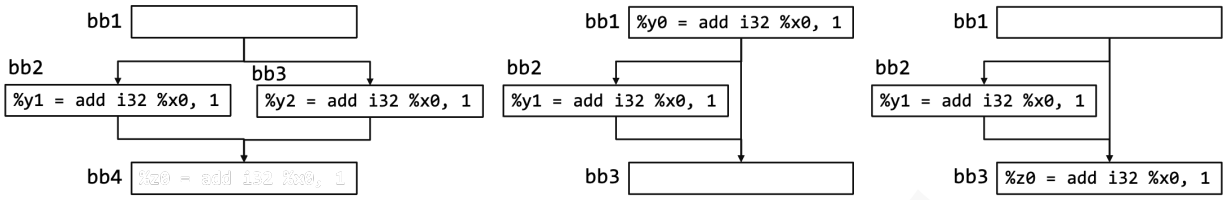
9.3.3 全局值编号

如果 IR 中操作数数值相同的同名指令出现多次，只需保留一个副本或计算一次即可。分析同名指令操作数数值是否相同的过程称为全局值编号（GVN：Global Value Numbering）。GVN 的分析一般在常量传播优化之后进行，其主要思路是为操作数数值相同的同名指令维护一个集合。由于 SSA 形式的 IR 无须考虑操作数是否被重新赋值的情况，只需通过操作数标识符是否相同便可识别出很大一部分操作数数值相同的指令，大大简化了 GVN 的分析和优化过程。如果两条同名指令的某个操作数标识符不相同，但在一个集合中，也应将其认定为同一编号，即表格 9.2 定义了 GVN 分析对应的 Transfer 函数。因此，我们可以沿用循环迭代的分析方法直至 GVN 集合不再更新为止。

表 9.2: Transfer 函数定义：全局值编号

IR 指令	举例	Transfer 函数
add/sub/mul/sdiv	%r = add i32, op1, op2	$S_r = S \cup r,\ S = \text{Find}(\text{add}, S_{op1}, S_{op2}))$
xor/and/or	%r = xor i32, op1, op2	$S_r = S \cup r,\ S = \text{Find}(\text{xor}, S_{op1}, S_{op2}))$
icmp	%r = icmp sgt i32, op1, op2	$S_r = S \cup r,\ S = \text{Find}(\text{icmp, sgt}, S_{op1}, S_{op2}))$
zext/trunc	%r = zext i8 op1 to i32	$S = S \cup r,\ S_r = \text{Find}(\text{zext, i32}, S_{op1})$

注：S_i 是包含 i 的与 i 值相同的标识符集合。



(a) 繁忙表达式　(b) 公共表达式　(c) 弱公共表达式

图 9.2: 基于 GVN 的优化

根据重复计算指令出现位置的不同，GVN 具体可以优化的情况又分为繁忙表达式、公共表达式、和弱公共表达式。

- 繁忙表达式：操作数相同的指令出现在不同的分支。例如图 9.2a 中的 bb2 和 bb3 都包含 add i32 %x0, 1，可将其提前到条件跳转指令之前，从而优化代码体积。
- 公共表达式：操作数相同的指令具有支配关系。例如图 9.2b 中的 bb1 和 bb2 都包含 add i32 %x0, 1，且 bb1 支配 bb2。在这种情况下，bb2 中的对%y1 的求值运算是多余的，可以将其删除，使用%y0 代替%y1。
- 弱公共表达式：两条操作数相同的指令不存在支配关系也存在可以优化的情况。例如图 9.2c 中 bb2 和 bb3 中都包含 add i32 %x0, 1，将该表达式提前至 bb1 则可以避免走左侧分支时的重复计算。

9.4 循环优化

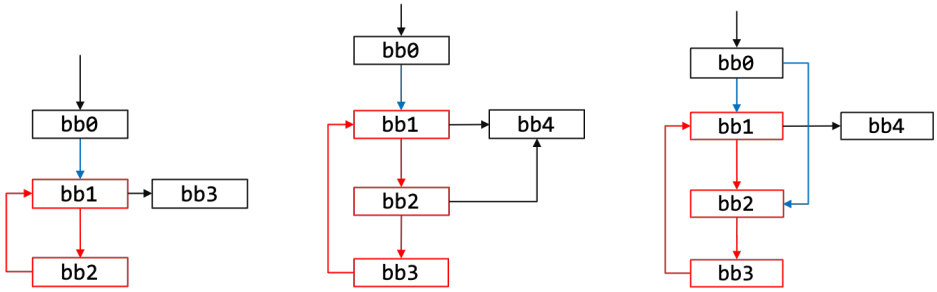
循环优化是对于提升代码运行效率效果最为显著的优化手段之一，其核心思想是将循环内重复执行的代码提前至循环外的支配节点，避免代码被重复执行。下面先介绍面向代码控制流图中循环路经的检测方法，再介绍具体的循环优化技巧。

9.4.1 循环检测算法

由于 TeaPL 中只有 if-else 和 while 语句会引入控制流，使得其对应 IR 中的每个循环都只会有一个入口节点，该节点支配循环中的所有其它节点，这种循环称为自然循环。因此，TeaPL 对应的控制流图中的循环都是自然循环，这种都是自然循环的图是可规约图。

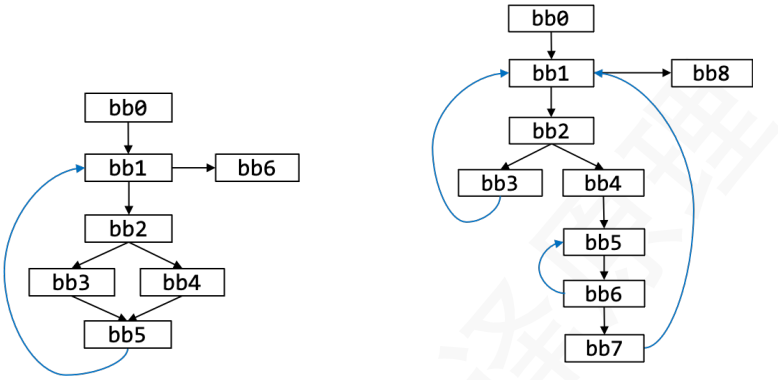
图 9.3 展示了几个控制流图的例子。其中，图 9.3a 是 while 循环的控制流图，很明显 bb1->bb2->bb1 是一个自然循环；图 9.3b 中的 bb1->bb2->bb3->bb1 也是自然循环，通过在循环体内部加入一个条件跳转分支和 break 语句可以达到此效果；图 9.3c 中的循环存在两个入口，因此是非自然循环。

自然循环中的入口节点是唯一的，因此可以采用入口节点标识一个自然循环；但不同的循环可能出现入口节点相同的情况，所以我们采用更细粒度的返回边唯一标识一个循环。以图 9.4a 为例，其中包含一个自然循环：bb5->bb1。图 9.4b 则包含三个自然循环：bb3->bb1、bb6->bb5、和 bb7->bb1。其中，循环 bb7->bb1 包含循环 bb6->bb5 的所有节点，这两个循环为嵌套关系；循环 bb7->bb1 和循环 bb3->bb1



(a) 自然循环：while　　(b) while+break　　(c) 非自然循环

图 9.3: 自然循环



(a) 控制流图示例：1 个循环　　(b) 控制流图示例：3 个循环

图 9.4: 通过返回边标识自然循环

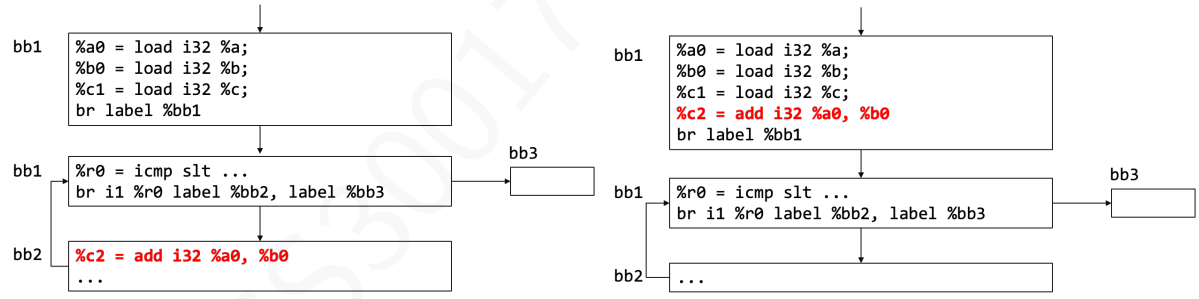
有两个公共节点 `bb1` 和 `bb2`，以及一条公共边 `bb1->bb2`，这两个循环为相切的关系。存在嵌套或相切关系的两个循环不需要经过其它节点可以组成一个大的强联通分量。自然循环之间不会出现相交，即存在多个入口节点的情况。

基于上述分析，我们可以设计出算法 12 ，用于检测 IR 中的自然循环。

算法 12 自然循环搜索算法

```
1: s ← ∅; // 栈，用于记录访问过的节点
2: Loop ← ∅; // 记录识别出的循环，使用返回边作为唯一标识
3: procedure FindLoops(v) // 从控制流图入口开始搜索其中的循环
4:     s.push(v);
5:     for each w in v.next() do
6:         if s.contains(w) then // 已经访问过该节点，说明找到循环
7:             AddLoop(w, v);
8:         else
9:             FindLoops(w); // 深度优先递归搜索
10:        end if
11:    end for
12:    s.pop(v);
13: end procedure
14: procedure AddLoop({v, w}) // 将识别到的循环添加到结果中
15:    if !Loop.exists(v, w) then
16:        l ← CreateLoop(top n items of s until w);
17:        Loop.add((v, w), l);
18:    else// 循环已经出现过：以图 9.4a 为例，由于循环内部的条件分支导致其再次被检测到
19:        l ← CreateLoop(top n items of s until w);
20:        Loop.merge((v, w), l);
21:    end if
22: end procedure
```

9.4.2 循环优化应用

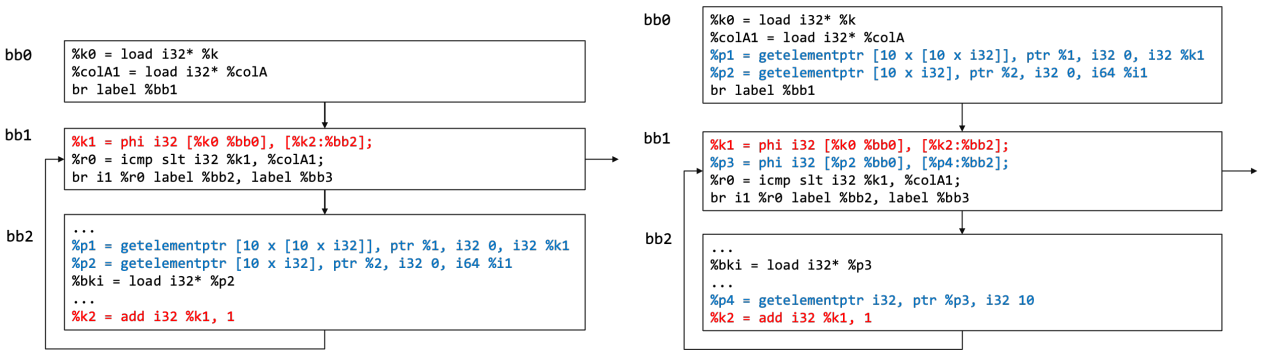


(a) 原始 IR　　(b) 优化后的 IR

图 9.5: 循环不变代码优化

```
while (i<rowA) {
    while (j<colB) {
        while (k<colA) {
            R[i][j] = R[i][j] + A[i][k]*B[k][j];
            //优化：改为t = t + A[i][k]*B[k][j];
            k = k+1;
        }
        j = j+1;
    }
    i = i+1;
}
```

代码 9.2: 标量替换优化示例：TeaPL 实现矩阵乘法代码片段



(a) 原始 IR　　(b) 优化后的 IR

图 9.6: 归纳变量代码优化

本节介绍三种典型的循环优化应用：

- 循环不变代码：如果一条指令在循环体内被多次执行，但其操作数未发生变化，则应将这条指令前移至循环外部，避免重复执行。图 9.5 展示了一个示例，bb3 中指令`%c2 = add i32 %a0, %b0` 的操作数`%a0` 和`%b0` 均定义自循环外部，因此可将这条指令前移至 bb0，从而避免重复计算。
- 标量替换：如果由于循环导致需要多次仿存同一内存地址上的标量数据，应当使用寄存器替换该仿存操作。代码 9.2 展示了一个典型的矩阵乘法案例，将其中的 R[i][j] 替换为临时变量 t 则可以避免在循环体内对 R[i][j] 的重复仿存操作。需要注意的是，该替换是有风险的。例如，当数组 R 和数组 A 或 B 存在别名关系时，其部分元素会共用同一块内存地址，更新 R[i][j] 的同时也更新了 A 或 B 的某个元素值，使用标量替换后则无法保持一致的计算结果。
- 归纳变量优化：这种优化一般与循环的条件变量相关。以代码 9.2 中最内层的循环为例，我们可以将其控制流图表示为图 9.6a 的形式。其中，`%k1` 是循环的条件变量，每一轮循环增加 1，直至等于`%colA1` 时退出循环。我们可以将 bb2 中的以`%k1` 作为操作数的相关指令进行优化，如将数组 `B[k+1][j]` 的寻址方式转化 `&B[k][j]+%colB1` 或 `&B[k][j]+10`（数组的大小为 10x10）的形式，从而优化寻址过程。由于 LLVM IR 的寻址指令以类型而非字节为基本单位，这种归纳变量优化方法在翻译汇编代码后可以再次发挥作用。

练习

1) 如果允许 LLVM IR 中的标识符被多次赋值，修改常量分析算法中的 Transfer 函数。

2) 代码 9.3 是 Collatz 函数对应的 IR，分析这段 IR 是否可以被优化？如何设计相应的优化算法？

```
define void @collatz(i32 %0) {
bb1:
    br label %bb2
bb2:
    %t0 = phi i32 [ %0, %1 ], [ %t1, %12 ]
    %b0 = icmp ne i32 %t0, 1
    br i1 %b0, label %bb3, label %bb7
bb3:
    %t2 = srem i32 %t0, 2 ; srem指令：取余数
    %b1 = icmp eq i32 %t2, 0
    br i1 %b1, label %bb4, label %bb5
bb4:
    %t3 = sdiv i32 %t0, 2
```

```
    br label %bb6
bb5:    %t4 = mul i32 3, %t0
    %t5 = add i32 %t4, 1
    br label %bb6
bb6:
    %t1 = phi i32 [ %t3, %bb4 ], [ %t5, %bb5 ]
    br label %bb2
bb7:
    ret void
}
```

代码 9.3: IR 代码：Collatz 函数

参考文献

[1] LLVM's Analysis and Transform Passes, https://llvm.org/docs/Passes.html

10 过程间优化

本章学习目标：

- ⋆ 掌握内联优化
- ⋆⋆ 掌握尾递归优化

过程间优化指的是利用函数调用的上下文信息优化函数调用和代码运行。在现代编程语言中，程序员可以对需要内联的函数手动标注，编译器会根据标注情况进行内联。编译器也支持一些自动化的内联优化分析。本节主要介绍一种通用的过程间优化问题：内联优化，以及一类特殊的内联优化问题：尾递归。

10.1 内联优化分析

内联指的是将 callee 的函数体复制到 caller 中，并使得程序等价的代码转换方法。并非所有的内联都是有优化收益的，下面我们对影响内联优化收益的因素进行讨论。

10.1.1 收益分析

内联产生优化收益的主要原因有两点：

- 减少函数调用开销;
- 带来新的过程内优化可能。

函数调用开销的优化效果与参数个数以及是否有返回值正相关，具体可表示为以下形式：

$$Cost = C(jmp_1) + \sum_{i}^{n} C(par_i) + C(jmp_2) + C(retval)$$

其中，$C(jmp_1)$ 和 $C(jmp_2)$ 分别为跳转到 callee 以及跳回到 caller 的开销；$\sum_i^n C(par_i)$ 为参数传递开销，n 为参数的个数；$C(retval)$ 为返回值传递开销。因此，如果 callee 没有参数和返回值，则内联优化收益比较一般。另外，如果 callsites 在循环内部，则该内联会在运行时带来更为明显的优化效果。

只有当函数有参数或返回值时内联才能带来新的优化可能。具体来说，参数可以为 callee 优化带来可能，例如参数为常量时通过常量传播进行优化；类似的，返回值为常量时可以为 caller 中 callsites 后续的代码带来优化可能。

10.1.2 副作用分析

函数内联可能的副作用有三点，其中前两点最为关键：

- 增大代码体积：如果同一个 callee 在多个 callsites 出现，内联该函数会导致代码体积膨胀。这是因为 callee 的代码会被复制到多个调用点，增加了重复代码。另外，如果一个 callee 为 public API，内联该 callee 后不能将该 callee 删除，因此会增加重复代码。
- 增加指令读取开销：函数内联将 callee 的代码复制到 caller 中，通常会导致 caller 体积增大，可能引发不同程度的 cache miss 问题。尤其是当 callee 中包含循环时，内联使得原来的循环被拆分到不同的 cache line 中，增加引发 cache miss 的频率。

- 加剧寄存器分配负担：一般认为，将 callee 的寄存器值复制到 caller 内部会导致更复杂的寄存器干扰，需要使用更多寄存器。但实际上，如果处理得当，这种影响几乎可以忽略不计。具体来说，如果 callsite 之前赋值的寄存器在调用后仍需使用，可能会导致被调用方缺少可用寄存器，从而需要将寄存器值溢出到内存。此时，只需按照函数调用约定选择相同的寄存器溢出即可。

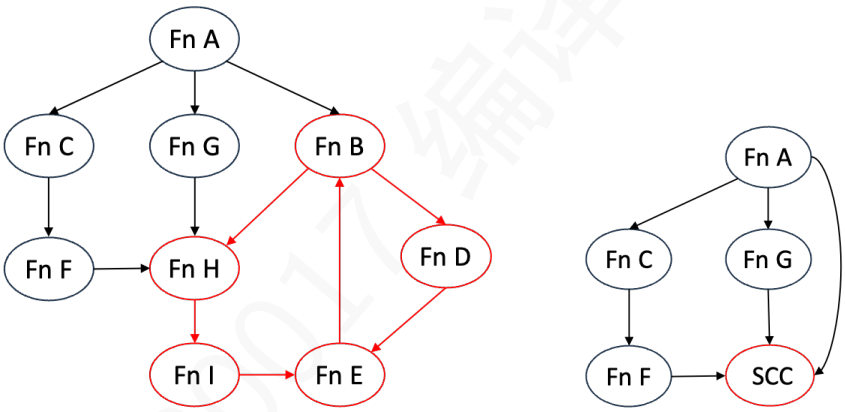
结合上述因素，一个函数越小，则将其内联带来副作用的概率越小。其它情况下，内联是否会带来正优化收益取决于很多因素，难以给出确切的建议 [1]。编译器一般不会主动对循环和递归调用（尾递归除外）进行内联。

10.2 内联优化算法

10.2.1 问题建模

本节讨论一种自动内联问题，即编译器如何在没有内联标注的情况下自动选取内联的 callsites，达到最优的内联优化效果。由于函数的调用关系可以表示为有向有环图，该问题相当于如何在图上选取特定的边进行内联，使得在预算有限的情况下收益最大。

10.2.2 调用图预处理



(a) 包含强连通分量的函数调用图　　(b) 缩环后的调用图

图 10.1: 调用图预处理

函数调用图与控制流图不同，其中的循环不一定是自然循环。解决内联问题一般先需要对函数调用图进行预处理，消除其中递归调用引入的环或强连通分量。以图 10.1a 为例，其中包含一个强连通分量 B-D-E-H-I。由于内联涉及选取顺序（bottom-up 或 top-down）问题，一般不对强连通分量进行内联或对其进行单独内联。

有向图的强连通分量检测可以采用经典的 Tarjan 算法 [2]（算法 13 ）。下面以图 10.1a 为例分析 Tarjan 算法的运算过程。

表 10.1: 应用 Tarjan 算法检测图 10.1a 中的强联通分量。

步骤	Stack	Time	ArriveTime, NextArriveTime									SCC
			A	B	C	D	E	F	G	H	I	
1	A	1	1,1									
2	A, C	2	1,1		2,2							
3	A, C, F	3	1,1		2,2			3,3			*	
4	A, C, F, H	4	1,1		2,2			3,3		4,4		
5	A, C, F, H, I	5	1,1		2,2			3,3		4,4	5,5	
6	A, C, F, H, I, E	6	1,1		2,2		6,6	3,3		4,4	5,5	
7	A, C, F, H, I, E, B	7	1,1	7,7	2,2		6,6	3,3		4,4	5,5	
8	A, C, F, H, I, E, B, H	8	1,1	7,4	2,2		6,6	3,3		4,4	5,5	
9	A, C, F, H, I, E, B, D	8	1,1	7,4	2,2	8,8	6,6	3,3		4,4	5,5	
10	A, C, F, H, I, E, B, D, E	8	1,1	7,4	2,2	8,6	6,6	3,3		4,4	5,5	
11	A, C, F, H, I, E, B	8	1,1	7,4	2,2	8,6	6,6	3,3		4,4	5,5	
12	A, C, F, H, I, E	8	1,1	7,4	2,2	8,6	6,4	3,3		4,4	5,5	
13	A, C, F, H, I	8	1,1	7,4	2,2	8,6	6,4	3,3		4,4	5,4	
14	A, C, F, H	8	1,1	7,4	2,2	8,6	6,4	3,3		4,4	5,4	H-I-E-B-D
15	A, C, F	8	1,1	7,4	2,2	8,6	6,4	3,3		4,4	5,4	H-I-E-B-D, F
16	A, C	8	1,1	7,4	2,2	8,6	6,4	3,3		4,4	5,4	H-I-E-B-D, F, C
17	A, G	2	1,1	7,4	2,2	8,6	6,4	3,3	2,2	4,4	5,4	H-I-E-B-D, F, C
18	A, G, H	2	1,1	7,4	2,2	8,6	6,4	3,3	2,2	4,4	5,4	H-I-E-B-D, F, C
19	A, G	2	1,1	7,4	2,2	8,6	6,4	3,3	2,2	4,4	5,4	H-I-E-B-D, F, C, G
20	A, B	2	1,1	7,4	2,2	8,6	6,4	3,3	2,2	4,4	5,4	H-I-E-B-D, F, C, G
21	A	2	1,1	7,4	2,2	8,6	6,4	3,3	2,2	4,4	5,4	H-I-E-B-D, F, C, G, A

算法 13 Tarjan 强联通分量检测算法

```
1: t ← 0; // time of arrival
2: procedure VISIT(v)
3:     Arrive[v] ← t; // 记录每个节点的到达时间
4:     NextArrive[v] ← t; // 记录下一跳的最早到达时间
5:     t ← t + 1;
6:     S.push(v);
7:     for each n in v.next() do
8:         if Arrive[n] == 0 then
9:             Visit(n);
10:            NextArrive[v] ← min(NextArrive[v], NextArrive[n]);
11:        else if s.contains(n) then
12:            NextArrive[v] ← min(NextArrive[v], Arrive[n]);
13:        end if
14:    end for
15:    if NextArrive[v] == Arrive[v] then // 找到强联通分量
16:        scc ← pop S until v;
17:        SCC.add(scc);
18:    end if
19: end procedure
```

10.2.3 贪心式内联优化算法

从有向无环图中选取边进行内联的问题可建模为背包问题，属于 NP-hard 问题。算法 14 介绍了一种基于贪心方法求解的思路。该方法首先对所有边的内联收益进行排序，然后优先选取收益最大、且不超过总预算的边进行内联。具体的收益和开销可以通过启发式算法计算获得，也可以针对不同 callsites 内联后实际的收益评测获得。

算法 14 贪心式内联优化算法

```
1: S ← ∅; // 记录可以被内联的函数调用
2: C ← 0; // 记录内联代价
3: procedure SearchInline(v)
4:     for each e in E do
5:         if inlineable(w) then // 排除不可内联的函数调用，如间接调用
6:             BenefitEstimation(e);
7:             S.insert(e); // 基于收益排序
8:         end if
9:     end for
10:    for each e in S do
11:        cost ← CostEstimation(e);
12:        C ← C + cost;
13:        if C > budget then // 排除不可内联的函数调用，如间接调用
14:            S.remove(e); // 基于收益排序
15:        end if
16:    end for
17: end procedure
```

10.3 尾递归优化

如果函数返回语句之前的最后一条指令是调用自己，则称为尾递归调用。以阶乘为例，代码 10.1 实现了一个尾递归版本，而代码 10.2 则非尾递归。尾递归是一种特殊的递归调用，可以专门对其进行内联优化。

```
fn fac(n:int, r:int) -> int {
    if (n < 2) {
        ret r;
    }
    else {
        ret fac(n-1, n*r);
    }
}
```

代码 10.1: TeaPL 代码：尾递归形式的阶乘算法

```
fn fac(n:int) -> int {
    if (n < 2) {
        ret 1;
    }
    else {
        ret n * fac(n-1);
    }
}
```

代码 10.2: TeaPL 代码：非尾递归形式的阶乘算法

尾递归调用内联的过程称为尾递归消除。与一般内联不同，该过程无需拷贝函数体，在递归调用处将参数保存到原参数变量中并且跳转到函数入口即可。代码 10.3 以 IR 形式展示了尾递归调用代码 10.1 的递归调用消除方式。该方法的本质是使用循环替换递归。由于无 phi 指令形式的 IR 更容易反应出最终的优化效果，此处我们采用非完全 SSA 形式进行演示。

```
define i32 @fac(i32 %n0, i32 %r0) {
bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %n0, i32* %n
    store i32 %r0, i32* %r
    br label %bb1
bb1:
    %n1 = load i32, i32* %n
    %t0 = icmp slt i32 %n1, 2
    br i1 %t0, label %bb2, label %bb3
bb2:
    %r1 = load i32, i32* %r
    ret i32 %r1
bb3:
    %n2 = load i32, i32* %n
    %r2 = load i32, i32* %r
    %n3 = sub i32 %n2, 1
    %r3 = mul i32 %n2, %r2
    ; 优化前：%t1 = call i32 @fac(i32 %n3, i32 %r3)
    ; 优化前：ret i32 %t1
    store i32 %n3, %n
    store i32 %n2, %r
    br %bb1
}
```

代码 10.3: 尾递归消除

将尾递归转化为循环后，可以进一步设计算法对这种循环进行优化。例如，代码 10.4 将代码 10.3 中 bb1 的内容复制到 bb3 中，从而消除循环体内针对变量%n 的一次冗余的 store 和 load。

```
define i32 @fac(i32 %n0, i32 %r0) {
bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %n0, i32* %n
    store i32 %r0, i32* %r
    br label %bb1
bb1:
    %n1 = load i32, i32* %n
    %t0 = icmp slt i32 %n1, 2
    br i1 %t0, label %bb2, label %bb3
bb2:
    %r1 = load i32, i32* %r
    ret i32 %r1
```

```
bb3:
%n2 = load i32, i32* %n     %r2 = load i32, i32* %r
    %n3 = sub i32 %n2, 1
    %r3 = mul i32 %n2, %r2
    store i32 %r3, %r
    %t1 = icmp lt i32 %n3, 2;
    br i1 %t1 label %bb2, label %bb3
}
```

代码 10.4: 尾递归优化

练习

1) 通过实验对比下列代码在内联与非内联情况下的性能表现，并分析原因。可以手动编写 IR 代码并编译为可执行文件，或先将其翻译为 C 或 Rust 语言，再借助相应的编译器进行实验。

```
fn callee(a[]: int, l:int) {
    let i = 0;
    while (i<l) {
        a[i] = a[i] + 1;
        i = i + 1;
    }
}
fn caller() {
    let i = 0;
    let a[1000]:int = {0};
    while (i<1000) {
        callee(a);
    }
}
```

代码 10.5: TeaPL 代码：内联实验

2) 宏的实现方式与内联有许多相似之处，但宏是在编译预处理阶段进行字符串替换。请对比分析这两种方式的区别，并举例说明。

```
macro_rules! foo(i)
    i + i;
}
fn main() {
    foo!(10); //替换为 10 + 10;
    foo!(f(x)); // 替换为 ?
}
```

代码 10.6: TeaPL 代码：宏

参考文献

[1] Inline Functions, `https://isocpp.org/wiki/faq/inline-functions#inline-and-perf`.

[2] Robert Tarjan. "Depth-first search and linear graph algorithms." *SIAM Journal on Computing*, 1972.

Part III

后端-ARM 版

11 指令选择-ARM

本章学习目标：

- ⋆⋆ 掌握 ARMv8-A（AArch64）指令集的基础指令
- ⋆⋆⋆ 掌握从 LLVM IR 到目标汇编代码的指令选择方法及其建模优化技术

11.1 ARMv8-A 指令集

本章介绍如何将 LLVM IR 翻译为 AArch64 汇编代码，目标体系结构为 ARMv8-A。我们首先讨论单条 LLVM IR 指令到 AArch64 指令的基本翻译方法，然后进一步介绍针对整个代码块的指令选择问题及其优化方法。为简化讨论，本章暂不考虑具体物理寄存器的分配问题，而统一使用虚拟寄存器表示操作数。其中，`w0-wn` 表示 32 位寄存器，`x0-xn` 表示 64 位寄存器。

ARMv8-A 属于典型的精简指令集（RISC）体系结构，其主要特点之一是访存操作与算术运算相互分离：数据必须先通过加载指令从内存读入寄存器，随后才能参与运算。代码 11.1 给出了一个简单的 Hello World 汇编程序。该程序首先使用 `str` 指令将返回地址寄存器 `x30` 保存到栈中；随后通过 `adrp` 与 `add` 指令构造字符串`"Hello World!"` 的地址。其中，`adrp` 用于获取目标符号所在内存页的基地址，`add` 进一步补充页内偏移。最后，程序通过 `bl` 指令调用 `puts` 函数输出字符串，并在返回前恢复寄存器 `x30`。

```
    .text                    ; 代码段
    .global main             ; 导出符号 main

main:
    str     x30, [sp, -16]! ; 保存返回地址，并为栈帧分配16字节空间
    adrp    x0, .LC0         ; 将字符串所在页的基地址加载到 x0
    add     x0, x0, :lo12:.LC0 ; 加上页内偏移，得到字符串完整地址
    bl      puts             ; 调用 puts(x0)
    ldr     x30, [sp], 16    ; 恢复返回地址，并回收栈空间
    ret                      ; 返回调用者

.LC0:
    .string "Hello World!"   ; 字符串常量
```

代码 11.1: ARM-v8A 指令示例：Hello World 程序

不同平台可能采用不同的汇编语法，本章统一采用 Linux/LLVM/GCC 工具链常见的 GNU 汇编语法。对于大多数数据处理指令，其操作数一般采用：

```
opcode destination, source1, source2
```

的形式，即结果寄存器通常写在最前面。例如：`add x0, x1, x2` 表示 `x0 = x1 + x2`。

11.1.1 寻址模式

在 ARMv8-A 架构中，变量可能位于不同的存储区域，因此需要采用不同的寻址方式。根据变量的作用域和存储位置，寻址通常可以分为全局变量寻址和局部变量寻址两类。

全局变量通常位于数据段或只读段，其地址在程序运行期间相对固定。由于 AArch64 指令长度固定为 32 位，无法在单条指令中直接编码完整的 64 位地址，因此通常采用“页基址 + 页内偏移”的方式构造全局地址。一般先通过 `adrp` 指令获取目标符号所在内存页的基地址（4KB 对齐），再通过 `add` 或 `ldr` 补充低 12 位偏移量。

```
adrp x0, g                  ; 加载全局变量 g 所在页的基地址
ldr  w1, [x0, :lo12:g]      ; 读取全局变量 g; :lo12:g表示符号g在页内的低12位偏移量
```

局部变量通常分配在栈空间中，并通过栈指针寄存器 `sp` 访问。ARMv8-A 支持多种灵活的访存寻址模式，主要包括以下几种：

- **立即偏移寻址模式**：使用基地址寄存器和立即数偏移量计算目标地址。例如：

  ```
  ldr x2, [x1]          ; 加载地址 x1 中的数据到 x2
  ldr x2, [x1, #10]     ; 加载地址 x1+10 中的数据到 x2
  ```

- **寄存器偏移寻址模式**：偏移量由另一个寄存器提供，可用于数组访问等场景。例如：

  ```
  ldr x2, [x1, x0]              ; 加载地址 x1+x0 中的数据
  ldr x2, [x1, x0, lsl #3]      ; 加载地址 x1+x0*8 中的数据
  ```

  其中，`lsl #3` 表示将寄存器 `x0` 左移 3 位，相当于乘以 8，因此该寻址模式常用于访问 8 字节元素（如 `i64` 或指针类型）构成的数组。虽然左移操作在硬件上可能发生整数回绕，但若数组下标已经大到导致地址计算溢出，则对应数组所需的内存规模通常已经超过机器实际可分配的地址空间，因此这种情况不会出现在正常的数组访问中。

- **预索引寻址模式**：在访存之前先更新基地址寄存器。例如：

  ```
  ldr x2, [x1, #10]! ; 先将x1更新为x1+10，再读取该地址中的数据到x2
  ```

  这种形式常用于栈帧调整。

- **后索引寻址模式**：先完成访存，再更新基地址寄存器。例如：

  ```
  ldr x2, [x1], #10 ; 先读取地址x1中的数据到x2，随后再将x1更新为x1+10
  ```

- **栈寻址模式**：通过栈指针寄存器 `sp` 访问局部变量。例如：

  ```
  ldr x2, [sp, #8] ; 读取地址sp+8中的数据到x2。
  ```

11.1.2 立即数支持

由于 AArch64 指令采用 32 位定长编码，因此一条指令中能够用于表示立即数的位数较为有限。不同类型的指令采用不同的立即数编码方式，其中算术运算指令和逻辑运算指令的支持方式并不相同。

大多数算术数据处理指令（如 `add`、`sub`）支持 12 位无符号立即数，即：

$$0 \leq x < 2^{12}$$

此外，这类立即数还支持额外左移 12 位，即：

$$x \times 2^{12}$$

因此，除了 `0-4095` 外，还可以直接表示如 `4096`、`8192` 等数值。例如：

```
add x0, x1, #100
add x0, x1, #4096
```

当立即数无法直接编码时，通常需要通过多条指令构造完整常数。AArch64 提供了 `movz`、`movk` 等指令用于分段构造 64 位立即数。其中：

- `movz`：将寄存器其它位清零，并写入 16 位立即数；
- `movk`：保持寄存器原有位不变，仅修改指定的 16 位字段。

例如，立即数 65539（即 `0x00010003`）可以通过如下方式构造：

```
movz x8, #3                 ; x8 = 0x0000000000000003
movk x8, #1, lsl #16        ; x8 = 0x0000000000010003
```

代码 11.2: ARM-v8A 指令：立即数示例

其中，`lsl #16` 表示将立即数左移 16 位后写入寄存器对应位置，即修改寄存器的 16-31 位。

逻辑运算指令（如 `and`、`orr`、`eor`）采用与算术指令不同的立即数编码方式。它并不是简单支持固定宽度的整数，而是支持一类特殊的位掩码模式（bitmask immediate）。这种设计来源于位运算的常见使用场景：程序通常更关注“哪些位为 1”，而不是立即数本身的数值大小。

例如，下列连续 1 位掩码可以直接编码：

```
and x0, x1, #0xFF           ; 保留低8位
and x0, x1, #0xFFFF         ; 保留低16位
```

上述指令常用于整数截断、类型转换以及提取某些字段。

此外，ARMv8-A 还支持周期性重复的位模式。例如：

```
orr x0, xzr, #0xAAAAAAAAAAAAAAAA
```

该指令会生成如下二进制模式：

```
1010101010101010...
```

需要注意的是，这里的 `0xAAAAAAAAAAAAAAAA` 并不是作为普通 64 位整数直接编码到指令中，而是属于 ARMv8-A 逻辑立即数支持的一类特殊位掩码模式。ARM 会使用专门的压缩编码方式描述位模式的重复结构，并由硬件自动生成完整的 64 位掩码。例如，`0xAAAAAAAAAAAAAAAA` 和 `0x3333333333333333` 都属于这种可直接编码的周期性位模式。这种模式主要用于高效支持位掩码生成、SIMD 向量计算以及各类底层位运算优化。

“‘latex

11.1.3 主要指令

下面以单条 LLVM IR 指令的翻译为例，介绍常用的 ARMv8-A（AArch64）指令，见表 11.1 。实际 ARMv8-A 架构包含数百条指令，并提供了丰富的寻址模式与硬件机制。本节仅介绍编译器后端代码生成过程中最常见的一部分指令。如需进一步了解完整指令系统，可参考 ARM 官方手册 [1]。

表 11.1: LLVM IR 及其对应的 ARM-v8A 指令

IR 指令	ARM-v8A 指令	说明
`%a = alloca i32`	`sub sp, sp, 16`	为局部变量分配栈空间；AArch64 要求栈指针保持 16 字节对齐
`store i32 %0, ptr %a`	`str w0, [sp, 12]`	将寄存器中的 32 位整数保存到栈空间
`store i32 1, ptr %a`	`mov w0, #1` `str w0, [sp, 12]`	先构造立即数，再写入内存；`str` 不支持立即数操作数
`%a0 = load i32, ptr %a`	`ldr w0, [sp, 12]`	将栈空间中的 32 位整数加载到寄存器
`%g0 = load i32, ptr @g`	`adrp x8, g` `ldr w0, [x8, :lo12:g]`	先加载全局变量所在内存页地址，再加低 12 位偏移访问全局变量
`%r = add i32 %a, %b`	`add w0, w1, w2`	两个寄存器中的整数相加，结果保存到 `w0`
`%r = add i32 %a, 4095`	`add w0, w1, #4095`	算术指令支持 12 位立即数，以及左移 12 位后的形式
`%r = sub i32 %a, %b`	`sub w0, w1, w2`	两个寄存器中的整数相减；也支持立即数减法
`%r = mul i32 %a, %b`	`mul w0, w1, w2`	整数乘法；不支持立即数操作数
`%r = sdiv i32 %a, %b`	`sdiv w0, w1, w2`	有符号整数除法；不支持立即数操作数
`%r = icmp sgt i32 %a, %b` `br i1 %r, label %bb1, label %bb2`	`cmp w0, w1` `b.le .LBB2`	比较结果会更新条件标志位（NZCV），随后执行条件跳转
`%r = xor i32 %a, %b`	`eor w0, w1, w2`	按位异或运算；支持逻辑立即数
`%r = and i32 %a, %b`	`and w0, w1, w2`	按位与运算；支持逻辑立即数
`%r = or i32 %a, %b`	`orr w0, w1, w2`	按位或运算；支持逻辑立即数
`call void @foo()`	`bl foo`	函数调用，并将返回地址保存到 `x30` 寄存器
`ret i32 %r`	`ldr x30, [sp], 16` `ret`	从栈中恢复返回地址寄存器 `x30`，还原栈顶指针并返回

本节中的映射关系主要用于帮助理解 LLVM IR 与目标汇编之间的基本对应关系。需要注意的是，LLVM IR 中的单条指令通常并不一定严格对应一条 ARM 汇编指令，可能存在“一对多”或“多对一”的情况。

一方面，一条 IR 指令可能需要多条 ARM 指令共同完成。例如，ARMv8-A 中的许多运算指令并不支持立即数操作数，而较大的立即数通常也无法直接编码到单条指令中，因此编译器需要先构造常数，再执行对应运算。例如：

```
mul w0, w1, w2
sdiv w0, w1, w2
```

均只能使用寄存器作为输入操作数。

另一方面，一条 ARM 指令也可能同时对应多条 IR 指令。例如，ARMv8-A 提供了乘加指令：

```
madd w0, w1, w2, w3
```

其语义为 `w0 = w1 * w2 + w3`，因此可以同时对应 LLVM IR 中的乘法与加法两条指令。编译器还会主动利用 ARMv8-A 提供的特殊指令与寻址模式减少指令数量。例如，乘以 2 的幂通常会被优化为移位操作：

```
lsl w0, w1, #3      ; 等价于 w0 = w1 * 8
```

类似地，数组访问中的地址计算也常使用带缩放的寄存器偏移寻址模式：

```
ldr x0, [x1, x2, lsl #3]
```

其中，`x1` 为数组基地址，`x2` 为数组下标，`lsl #3` 表示元素大小为 8 字节。

11.2 消除 phi 指令

LLVM IR 在转换为 SSA（Static Single Assignment）形式后，会引入大量 phi 指令用于表示控制流汇合点上的变量选择。然而，ARM 等目标体系结构并不存在与 phi 指令直接对应的机器指令，因此在代码生成阶段必须消除 phi 指令。以代码 11.3 为例，可以通过两种方式完成 phi 指令的消除。

```
bb1:
%r1 = icmp eq i32 %a1, 0
; 方式一: store i32 %a1, ptr %a
; 方式二: %a3 = %a1
br i1 %r1, label %bb2, label %bb3

bb2:
%a2 = add i32 %a1, %b1
; 方式一: store i32 %a2, ptr %a
; 方式二: %a3 = %a2
br label %bb3

bb3:
%a3 = phi i32 [%a1, %bb1], [%a2, %bb2]
; 方式一: %a3 = load i32, ptr %a
%r1 = add i32 %a3, %b1
```

代码 11.3: LLVM IR 代码：消除 phi 指令的例子

方式一：使用 store-load 替换 phi ：该方法通过内存中转的方式实现 phi 指令的语义。在 phi 指令所有前驱基本块的跳转指令之前插入 store 指令，将对应的源操作数写入同一内存位置；随后在包含 phi 指令的基本块开头使用 load 指令读取该值，并以读取结果替代原有 phi 指令。

以代码 11.3 为例，在基本块 %bb1 和 %bb2 中分别插入：

```
store i32 %a1, ptr %a
store i32 %a2, ptr %a
```

同时将基本块 %bb3 中的 phi 指令替换为：

```
%a3 = load i32, ptr %a
```

这种方式与现有 LLVM IR 兼容可以直接使用 LLVM 解释器执行。缺点是会引入额外的访存开销。

方式二：使用伪赋值指令替换 phi ：在每个前驱基本块中直接对 phi 指令的目标寄存器进行赋值。具体而言，可将：

```
%a3 = phi i32 [%a1, %bb1], [%a2, %bb2]
```

转换为在前驱基本块中的伪赋值操作：

```
; bb1
%a3 = %a1
; bb2
%a3 = %a2
```

该方法本质上模拟了后端汇编中的寄存器移动操作。由于数据始终保存在寄存器中，因此能够避免额外的访存开销。缺点是 LLVM IR 本身并不支持这种“寄存器直接赋值”的语法形式，因此这种修改后的 IR 无法被标准 LLVM 解释器直接执行。

11.3 指令选择问题

通过前面的介绍，我们可以较为直接地为单条 LLVM IR 指令找到对应的 ARM 汇编翻译方式。例如，IR 中的加法、减法或乘法指令通常都能映射为对应的 ARM 算术指令。然而，仅采用“单条 IR 指令对应单条汇编指令”的翻译策略通常并不是最优的，因为现代处理器提供了大量能够同时完成多个操作的复合指令。如果忽略这些复合指令的使用，生成的目标代码往往会包含更多指令数量和额外的寄存器读写，从而影响程序执行效率。

例如，ARM-v8A 提供了乘法累加和乘法减法等复合算术指令，它们能够在一次指令执行过程中同时完成乘法与加法（或减法）运算。例如：

```
%t1 = mul i32 %a, %b
%t2 = add i32 %t1, %c
```

可以直接翻译为：

```
madd w0, w1, w2, w3
```

而无需先生成：

```
mul w4, w1, w2
add w0, w4, w3
```

上述过程体现的正是指令选择问题：即如何从 IR 表达的计算语义中，选择最适合目标体系结构的机器指令组合，使生成代码在指令数目和执行效率等方面达到更优效果。从更一般的角度来看，指令选择并不是简单的“翻译”问题，而是一个“模式匹配”问题。编译器需要识别 IR 中能够被目标机器复杂指令覆盖的计算模式，并使用代价更低的机器指令进行替换。例如：

- 将“乘法 + 加法”识别为 `madd`
- 将“乘法 + 减法”识别为 `msub`
- 将数组地址计算识别为 ARM 的地址偏移寻址模式

因此，高质量的指令选择算法往往需要在 IR 上进行模式分析与匹配，以最大程度利用目标体系结构提供的复杂指令能力。为了降低问题复杂度，下面我们将重点讨论单个基本块上的指令选择问题。由于函数控制流可以被划分为多个相互独立的基本块，因此编译器通常可以分别对每个基本块进行指令选择。

11.3.1 指令选择图

为了系统化地研究单个基本块上的指令选择问题，我们通常将基本块中的 LLVM IR 表示为一种特殊的有向图结构，从而将“生成目标机器指令”的过程转化为图上的模式匹配与覆盖问题。

定义 6（指令选择图）**.** 指令选择图（Instruction Selection Graph）是一个有向无环图（Directed Acyclic Graph, DAG），图中包含两类节点：

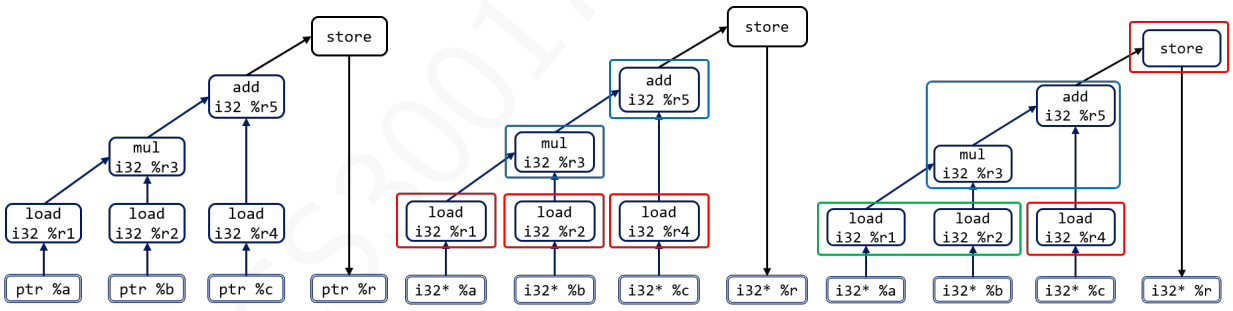
- **指令节点**：表示 IR 中的运算操作，例如 `add`、`mul`、`load` 等；
- **数据存储节点**：表示变量、寄存器或内存中的数据值。

图中的有向边表示数据依赖关系，即某条指令执行时需要使用的输入操作数。

由于 LLVM IR 处于 SSA 形式，每个虚拟寄存器只会被定义一次，因此普通算术数据依赖天然构成有向无环图。以代码 11.4 为例，该代码首先从内存中读取变量 `a` 和 `b`，随后执行乘法运算；接着读取变量 `c`，并将其与乘法结果相加；最后将结果写回内存。根据这些数据依赖关系，可以构建出图 11.1a 所示的指令选择图。若某条指令的输出被另一条指令使用，则在两者之间建立一条有向边。只要最终生成的机器指令序列满足该 DAG 的拓扑顺序，就能够保证程序语义正确。

```
%r1 = load i32 %a;
%r2 = load i32 %b;
%r3 = mul i32 %r1, %r2;
%r4 = load i32 %c;
%r5 = add i32 %r3, %r4;
store i32 %r5, %r;
```

代码 11.4: LLVM IR 代码



(a) 指令选择图　(b) 指令选择方案 1　(c) 指令选择方案 2

图 11.1: 指令选择问题举例

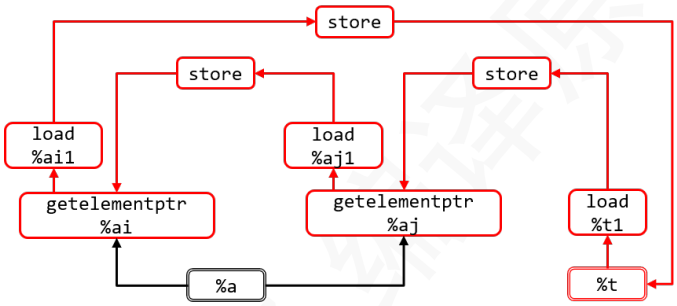
值得注意的是，当代码块中出现 `store` 等涉及内存写入的指令时，不能简单地仅依据寄存器数据依赖将整个基本块构造成一个指令选择 DAG。虽然 SSA 形式下的普通算术数据依赖天然构成无环图，但访存操作之间还存在额外的内存依赖关系。在某些情况下，这些内存依赖可能破坏原有 DAG 结构，从而无法仅通过普通数据流图正确表达程序语义。若忽略这些内存依赖关系，则后续拓扑排序与指令重排过程中可能生成违反原始 LLVM IR 语义的代码。

以代码11.5 为例，该代码实现了三个内存位置之间的数据交换操作，其语义高度依赖于 `store` 指令之间的执行顺序。若任意改变这些内存访问顺序，都可能导致读取到被覆盖后的值，从而破坏程序语义。因此，如果直接为整个基本块构建图，则不能有效表达内存读写之间的顺序约束，从而在进行拓扑排序时，可能得到错误的程序。

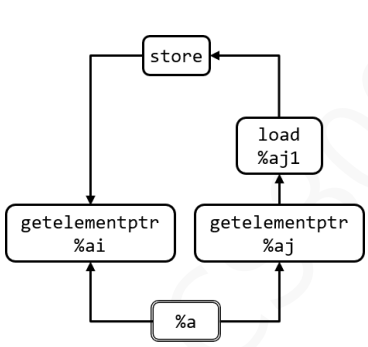
```
%ai = getelementptr [10 x i32], [10 x i32]* %a, i32 0, i32 i
%aj = getelementptr [10 x i32], [10 x i32]* %a, i32 0, i32 j
%aj1 = load i32, ptr %aj
store i32 %aj1, ptr %ai
%t1 = load i32, ptr %t
store i32 %t1, ptr %aj
%ai1 = load i32, ptr %ai
store i32 %ai1, ptr %t
```

代码 11.5: LLVM IR 代码：内存同步问题

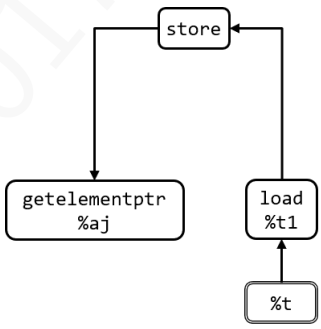
为解决这一问题，可以将 `store` 等具有副作用的指令作为边界，对基本块进行划分，将整个基本块拆分为多个具有顺序关系的指令选择子图。例如，图 11.2a可以拆分为图11.2b、11.2c和11.2d。



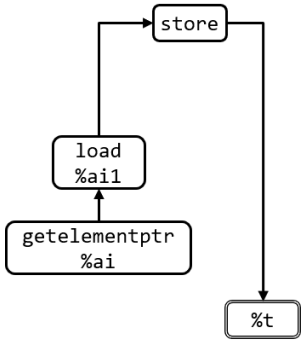
(a) 直接为代码 11.5 绘制指令选择图



(b) 指令选择子图 1



(c) 指令选择子图 2



(d) 指令选择子图 3

图 11.2: 代码块涉及内存同步问题时的指令选择子图分割

11.3.2 图覆盖问题

通过指令选择图，我们可以将指令选择问题转化为图覆盖问题，即如何使用目标机器指令覆盖指令选择图中的所有节点，并使生成代码的代价最小。这里的代价通常包括指令数量和执行时间。

以图 11.1a 为例，其至少存在图 11.1b 和图 11.1c 两种铺树方案，对应的汇编代码分别如代码 11.6 和代码 11.7 所示。

```
ldr w1, [sp, .a]
ldr w2, [sp, .b]
ldr w3, [sp, .c]
mul w3, w1, w2
add w5, w3, w4
str w5, [sp, .r]
```

代码 11.6: 图 11.1b 对应的汇编代码

```
ldp w1, w2, [sp, .a]
ldr w3, [sp, .c]
madd w5, w1, w2, w4
str w5, [sp, .r]
```

代码 11.7: 图 11.1c 对应的汇编代码

其中，方案一采用逐条指令翻译方式，而方案二使用了 ARM-v8A 的复合指令：

- `ldp` 同时完成两个连续变量加载；
- `madd` 同时完成乘法与加法。

因此，方案二的指令数量更少，寄存器使用也更少。若进一步假设 `ldr` 与 `ldp`、`mul` 与 `madd` 的执行代价相同，则图 11.1c 对应的铺树方案更优。

一般而言，铺树问题属于 NP-hard 问题，常见求解方法包括贪心算法与动态规划。其中，一种经典的贪心策略称为 Maximal Munch，其基本思想是：从 DAG 的根节点开始，每一步优先选择能够匹配且覆盖节点数最多的指令模式，并递归处理剩余未覆盖部分。例如，当编译器发现：

```
%r3 = mul i32 %r1, %r2
%r5 = add i32 %r3, %4
```

时，会优先匹配覆盖范围更大的 `madd` 模式，而不是分别选择 `mul` 与 `add` 两个较小模式。Maximal Munch 实现简单、效率较高，因此被许多编译器采用。不过，由于其属于局部贪心策略，因此并不一定能够得到全局最优解。关于其具体实现细节与优化效果，同学们可以进一步自行探索。

参考文献

[1] Arm Architecture Reference Manual for A-profile architecture. `https://developer.arm.com/documentation/ddi0487/latest/`, 2023.

练习

1) 将下列 IR 代码翻译为 ARMv8-A 汇编代码。

```
define i32 @collatz(i32 %x0) {
bb0:
    br label %bb1
bb1:
    %x1 = phi i32 [ %x0, %bb0 ], [ %x6, %bb5 ]
    %r0 = icmp ne i32 %x1, 1
    br i1 %r0, label %bb2, label %bb6
bb2:
    %x2 = srem i32 %x1, 2
    %r1 = icmp eq i32 %x2, 0
    br i1 %r1, label %bb3, label %bb4
bb3:
    %x3 = sdiv i32 %x1, 2
    br label %bb5
bb4:
    %x4 = mul i32 %x1, 3
    %x5 = add i32 %x4, 1
    br label %bb5
bb5:
    %x6 = phi i32 [ %x3, %bb3 ], [ %x5, %bb4 ]
    br label %bb1
bb6:
    ret i32 %x1
}
```

代码 11.8: LLVM IR 代码

12 寄存器分配-ARM

本章学习目标：

- ⋆ 了解寄存器分配问题
- ⋆ 记住 ARM-v8a 中的主要寄存器用法
- ⋆⋆⋆ 理解基于图着色的寄存器分配问题建模方法
- ⋆⋆⋆ 掌握基于单纯消除序列的寄存器分配算法

12.1 ARM-v8A 中的寄存器

ARM-v8A 指令集有 31 个 64-bit 通用寄存器 `x0-x30`，如果仅使用低 32-bit 则使用标识符 `w0-w30`。虽称为通用寄存器，但是这些寄存器在函数调用场景应当满足一些使用约定。如表 12.1 所示，寄存器 x0-x1 用于参数和返回值传递，`x30` 用于保存函数返回地址，`x29` 一般用于保存栈帧基地址。另外，`x9-x15` 和 `x19-x28` 寄存器用途不限，但前者是 Caller-saved，后者是 Callee-saved。Caller-saved 指的是 Caller 负责保障函数调用前后寄存器的值不变，即调用前备份，调用后还原；Callee-saved 则是由 Callee 负责，即使用前备份，使用后还原。

表 12.1: AArch64 中的寄存器用法约定

寄存器名称	调用规约	注释
x0-x1	参数 1-2/返回值	
x2-x7	参数 3-8	
x8	特殊用途：间接调用返回地址	
x9-x15	普通寄存器	Caller-saved
x16-x18	特殊用途	
x19-x28	普通寄存器	Callee-saved
x29	栈帧基地址	
x30	返回地址	

寄存器分配指的是将汇编代码中的虚拟寄存器转化为物理寄存器，使得汇编代码可在目标 CPU 上运行。该过程涉及寄存器预分配、通用寄存器分配和寄存器溢出环节。其中，预分配指的是寄存器的用法应满足该指令集架构下的一些使用约定，可概括为以下几点：

- **参数和返回值传递**：LLVM IR 中的函数调用和返回均是由单条 IR 指令完成的，参数和返回值传递未考虑调用规约的问题，将其翻译为汇编代码时应：函数调用前先将参数按照顺序依次拷贝到 `x0-x7` 中；函数返回前将返回值拷贝到 `x0-x1` 中。
- **返回地址**：如果当前函数涉及一处或多处函数调用，应在函数入口处将 `x30` 寄存器内容保存到栈上，函数返回前将其还原。否则，Callee 会改写 `x30` 的值，导致无法正常返回。
- **其它调用规约**：如果使用 `x9-x15` 这类 Caller-saved 寄存器并涉及函数调用，应当在函数调用前将其备份，调用后还原；对于 `x19-x28` 这类 Callee-saved 寄存器，应当在使用前将其备份，函数返回前还原。

12.2 通用寄存器分配

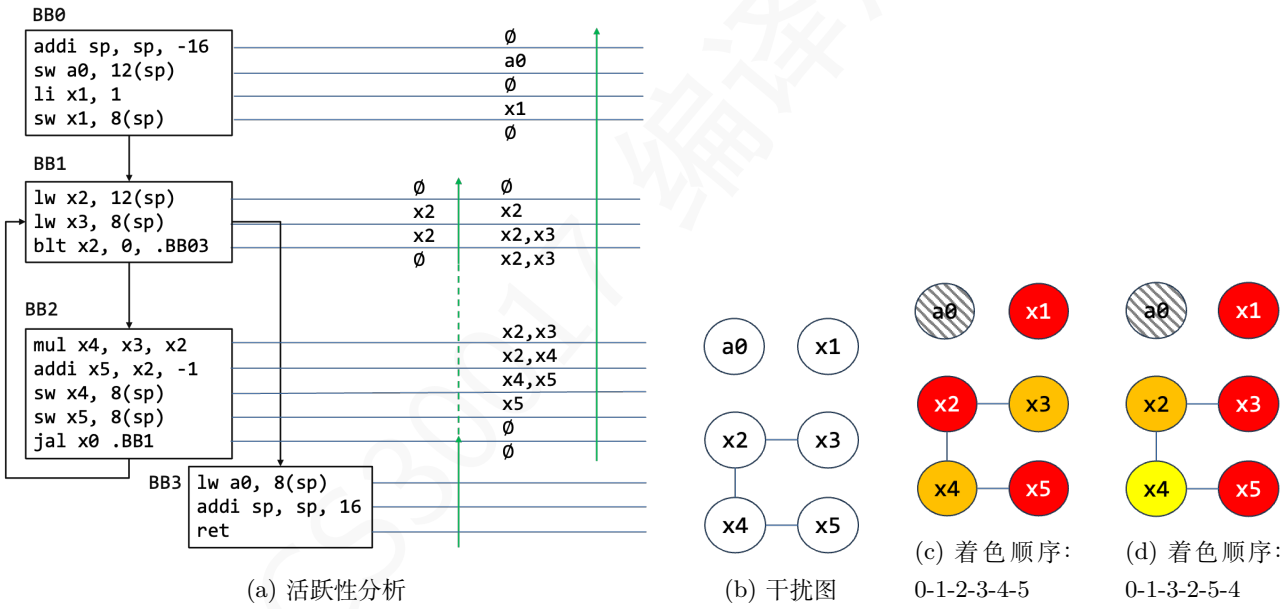
指令翻译时无需考虑虚拟寄存器数量限制，然而实际的物理寄存器数量是有限的。例如，AArch64 架构中一般使用 `x9-x15` 寄存器或 `x19-x28` 寄存器；X86 架构可用的寄存器则更少。因此，如何将虚拟寄存器翻译为物理寄存器非常关键。

下面我们使用干扰图对寄存器分配问题进行建模，并将其转化为着色问题。

12.2.1 干扰图构建

定义 7 (RIG)**.** 寄存器干扰图（Register Interference Graph）$\{V, E\}$ 是一个无向图，其中每一个点 $v_i \in V$ 表示一个虚拟寄存器，如果 $v_i, v_j \in V$ 同时活跃，则存在边 e_{ij}。存在干扰关系的虚拟寄存器应分配不同的物理寄存器。

干扰图是基于活跃性分析构建获得的，即分析每个虚拟寄存器的活跃区间。我们可以采用循环迭代数据流分析方法：对控制流图进行逆序分析；遇到 use(v_i) 则认为虚拟寄存器 v_i 在此之前都必须是活跃的，遇到 def(v_i)，则认为该虚拟寄存器最早活跃至此；遇到合并节点取并集即可。将同时存在活跃关系的虚拟寄存器两两相连便得到了干扰图。图 12.1a 和 12.1b 分别展示了一个活跃性分析示例及其干扰图构建结果。



(a) 活跃性分析　(b) 干扰图　(c) 着色顺序：0-1-2-3-4-5　(d) 着色顺序：0-1-3-2-5-4

图 12.1: 干扰图构建和着色示例

12.2.2 着色问题

基于干扰图的寄存器分配问题是一个着色问题。

定义 8 (着色问题)**.** 对无向图 $\{V, E\}$ 的所有点 $v_1, ..., v_n \in V$ 进行着色，要求相邻的点不能采用同样的颜色，请问至少需要多少个颜色？或是否存在至多使用 K 个颜色的着色方案？该问题又称为 K-colorable 问题，其中 K 个颜色代表 K 个物理寄存器。

K-colorable 问题在 $K \geq 3$ 时是 NP-Complete 的问题。

12.2.3 着色算法

我们假设存在颜色数组 Color = [红, 澄, 黄, 绿, 青, 蓝, 紫]，每次着色均采用当前编号最低的可能的颜色，即算法 15 。则着色问题本质上是着色顺序选取的问题。如图 12.1c 和12.1d 分别对应两种着色顺序，一个需要使用两个颜色，另外一个则需要使用三个颜色。

算法 15 颜色选取方法

```
1: procedure GreedyColoring(G(V, E))
2:     let C = {c_0, ..., c_k} be K colors
3:     for each v_i  s ∈ V do:
4:         let c_j be the lowest color not used in Adj(v_i)
5:         Col(v_i) = c_j
6:     end for
7:     Return G(V, C, E)
8: end procedure
```

针对着色问题有大量的贪心算法研究。比如线性扫描算法采用先到先得的思想，对于先遇到的寄存器先分配颜色。该方法的有点是非常块，无需维护干扰图的边数信息。另外还有一些基于干扰图边数选取着色顺序的启发式算法，下面介绍一种经典的 RLF（Recursive Largest First）算法 [1]。如算法 16 所示，该方法分为多轮次递归进行。每一轮次从图中选取度数最大（边数最的）的点进行着色；如果其余点中存在与该点不相连的点，则继续从中选取度数最大的点并采用相同的颜色着色，直至选择不出不相连的点为止。重复该过程直至全部点都被着色。

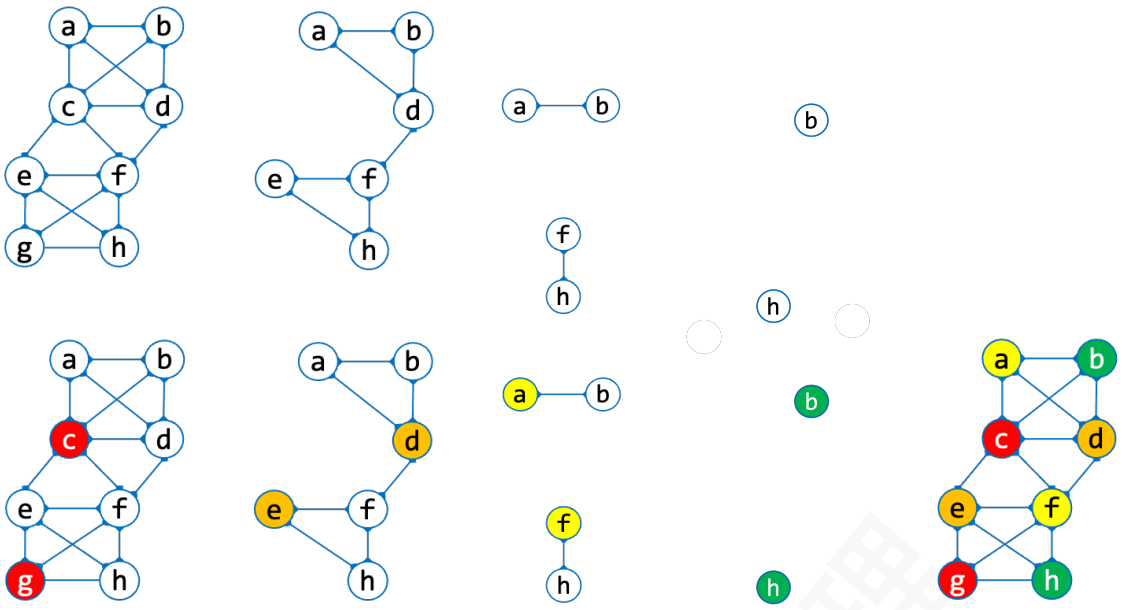
算法 16 Recursive largest first 算法

```
1: let S be the stack of nodes to be colored in one round; init S with empty.
2: let C = {c_0, ..., c_k} be K colors
3: procedure RLF(G(V, E))
4:     Find v_i ∈ G with the max degree
5:     S.push(v_i)
6:     Let T be the rest nodes in G non-adjacent to any node in S
7:     while T is not NULL do
8:         Find v_j ∈ T with the max degree
9:         S.push(v_j)
10:        Update T
11:    end while
12:    GreedyColoring(S)
13:    Remove S from G
14:    Set S to empty
15:    RLF(G)
16: end procedure
```

12.2.4 基于单纯消除序列着色

下面介绍一类特殊的着色问题：即当虚拟寄存器满足 SSA 形式时，则该着色问题不是 NP-hard 问题。我们可以采用基于单纯消除序列的方法选取最优着色顺序，确定最优着色方案 [2]。

下面首先定义单纯消除序列的相关概念。



(a) 第 1 轮　(b) 第 2 轮　(c) 第 3 轮　(d) 第 4 轮　(e) 结果

图 12.2: 应用 RLF 算法进行着色示例

定义 9 (单纯点). 无向图 $\{V, E\}$ 中的如果点 v_i 的所有邻居的邻居节点组成一个团（clique），则 v_i 是一个单纯点。

定义 10 (完美消除序列). 按照该序列消除的每一个点都是单纯点

定义 11 (单纯消除序列). 完美消除序列的逆序

定义 12 (弦图（Chordal Graph）). 无向图 $\{V, E\}$ 中的任意长度大于 3 的环都有弦。

当虚拟寄存器满足 SSA 形式时，其干扰图为弦图。弦图一定存在单纯消除序列。给定一个弦图，找单纯消除序列可以采用最大势算法，具体可参考算法 17 。表 12.2 展示了应用最大式算法找图 12.2e 的单纯消除序列的过程。

算法 17 最大势（Maximum Cardinality Search）算法

```
1:  procedure MCS(G(V, E))
2:      for each v_i ∈ V do
3:          w(v_i) = 0;
4:      end for
5:      W = V;
6:      for each i ∈ [1..n] do
7:          Let v be a node with max weight in W
8:          for each u ∈ Neighbor(v) do
9:              w(u) = w(u) + 1
10:         end for
11:         W = W - v;
12:     end for
13: end procedure
```

表 12.2: 应用最大势算法找图 12.2e 的单纯消除序列。

步骤		a	b	c	d	e	f	g	h
0	初始化	0	0	0	0	0	0	0	0
1	选取 a		1	1	1	0	0	0	0
2	选取 b			2	2	0	0	0	0
3	选取 c				3	1	1	0	0
4	选取 d					1	2	0	0
5	选取 f					2		1	1
6	选取 e							2	2
7	选取 g								3
8	选取 h								

12.3 寄存器溢出

当干扰图所需的颜色数量超过实际可用的物理寄存器时，部分寄存器的值需要溢出（spill）到内存中，以释放寄存器供其他用途。在值被溢出后，若再次需要使用，必须将其从内存重新加载到寄存器中。由于寄存器溢出会增加程序的运行开销，关键在于合理选择溢出的寄存器，以尽可能降低溢出代价。具体的寄存器选择通常与着色算法相关，同时也可以采用一些启发式方法，例如优先选择干扰图中最大团的顶点或度数最高的顶点进行溢出。

练习

1) 构造一个非弦图，使得 RLF 算法无法找到最优解。

2) 如果一个图是弦图，RLF 算法是否一定能找到最优解?

参考文献

[1] Frank Thomson Leighton, “A graph coloring algorithm for large scheduling problems.” Journal of Research of the National Bureau of Standards, 1979.

[2] Fernando Magno Quintao Pereira, and Jens Palsberg. “Register allocation via coloring of chordal graphs.” Asian Symposium on Programming Languages and Systems. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

13 后端优化-ARM

本章学习目标:

- ⋆⋆ 掌握指令调度问题和方法
- ⋆ 掌握窥孔优化方法
- 了解并行优化

13.1 指令调度

13.1.1 指令调度问题

由于 CPU 的流水线特性，指令的执行并非完全串行的，而是具有一定的并行度的。一条指令的执行通常包括指令读取、解码、执行、回写等步骤，这些步骤由 CPU 中不同的组件负责，每个组件无需等待本条指令完全执行完毕即可响应下一条指令的处理请求。经典的指令调度问题指的是：“一个程序中不同的指令执行效率存在巨大差异，且指令之间存在依赖性，应如何调整指令顺序才可以在不违背依赖性的前提下使总体程序执行时间最短?”随着超标量、乱序执行、分支预测等技术的发展，该问题实际上变得更为复杂。

以 ARM-A72 官方软件优化手册给出的性能数据为参照 [1]，我们得到常用指令的运行时延和吞吐大致如表 13.1 所示。其中，数据加载和乘除运算的时延较高，加载和乘法的吞吐基本不受时延高的影响，但除法的吞吐非常低。加法和减法指令具有两个执行单元 I0 和 I1，因此吞吐可以翻倍；A72 中其它执行单元只有一个。在新的 ARM CPU 型号中，上述执行单元都有所增加，如 A77 的加载、存储、乘法、跳转单元都是两个。

表 13.1: ARM-A72 指令开销。

指令	时延	吞吐	执行单元
`ldr`	4	1	L
`str`	1	1	S
`add`	1	2	I0/I1
`sub`	1	2	I0/I1
`mul`	3	1	M
`madd`	3	1	M
`sdiv`	7	1/7	M
`mov`	1	2	I0/I1
`adr`	1	2	I0/I1
`b`	1	1	B
`bl`	1	1	B, I0/I1
`ret`	1	1	B

13.1.2 指令重排

本节的指令调度优化问题主要考虑数据依赖因素带来的影响。假设一段程序中含有两条指令 A 和 B，且指令 A 先于指令 B，则 A 和 B 之间一般存在两种数据依赖：

- 正依赖：A 的运算结果是 B 的操作数，B 必须等待 A 运行结束以后才可执行。
- 反依赖：B 的运算结果会改写 A 的操作数，B 必须等待 A 运行结束以后才可执行。

表 13.2: 一段汇编代码和开销分析示例。

编号	指令	开始时间	结束时间
I1	`ldr x9, [sp, #-12]`	1	4
I2	`ldr x10, [sp, #-16]`	2	5
I3	`add x9, x9, x10`	6	6
I4	`ldr x10, [sp, #-20]`	7	10
I5	`ldr x11, [sp, #-24]`	8	11
I6	`sdiv x11, x10, x11`	12	18
I7	`str x11, [sp, #-24]`	19	19
I8	`ldr x10, [sp, #-28]`	20	23
I9	`mul x10, x9, x10`	24	26
I10	`str x10, [sp, #-28]`	27	27

下面我们以代码 13.2 为例分析其中的指令依赖关系和时延。由于指令 `I3` 的操作数 `x9` 和 `x10` 分别是指令 `I1` 和 `I2` 的执行结果，因此指令 `I3` 依赖 `I1` 和 `I2`。指令 `I4` 会更新 `I3` 操作数 `x10`，因此 `I4` 反依赖 `I3`。基于该方法对所有指令间的依赖关系进行分析，我们可得到图 13.1 。

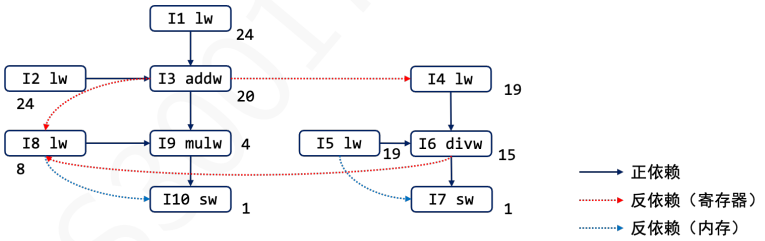


图 13.1: 代码 13.2 中指令间的依赖关系和时延

在图 13.1 中，指令 `I7` 和 `I10` 是最后执行的两条指令且不存在先后顺序要求，其自身时延均为 1，因此我们将 `I7` 和 `I10` 的程序时延标记为 1，其含义为本条指令开始执行后，程序的最早结束时间为 1。由于 `I9` 是 `mul` 指令，其自身时延为 3，因此将 `I9` 的程序时延标记为 4。同理，`I8` 的程序时延为 8；`I6` 同时依赖 `I7` 和 `I8`，其程序时延为 15，即指令自身时延 7 加上 `I7` 和 `I8` 中时延高者。基于程序时延分析，可以得到整段程序的执行瓶颈在于程序时延高的指令。如果优先执行这些程序时延高的指令，则有望提升程序的执行效率。我们对这段程序中的指令按照程序时延进行排序：`I1=I2>I3>I4=I5>I6>I8>I9>I7=I10`。

根据不同指令对程序时延的影响对指令进行重排得到表 13.3 。重排后的程序运行时延可以提升至 26。

13.1.3 消除反依赖

进一步对上述程序的性能瓶颈进行分析，可以发现 `I6` 与 `I8` 之间的的反依赖关系导致 `I8` 需等待 `I6` 执行结束后才可以执行，浪费了不少时间。同理，`I4` 和 `I3` 也存在类似问题。如果对 `I4` 中的寄存器 `x10` 进行替换，则可以将 `I4` 提前至 `I3` 之前执行。

表 13.3: 程序 13.2 指令重排后的结果。

编号	指令	开始时间	结束时间
I1	`ldr x9, [sp, #-12]`	1	4
I2	`ldr x10, [sp, #-16]`	2	5
I3	`add x9, x9, x10`	6	6
I4	`ldr x10, [sp, #-20]`	7	10
I5	`ldr x11, [sp, #-24]`	8	11
I6	`sdiv x11, x10, x11`	12	18
I8	`ldr x10, [sp, #-28]`	19	22
I9	`mul x10, x9, x10`	23	25
I7	`str x11, [sp, #-24]`	24	24
I10	`str x10, [sp, #-28]`	26	26

表 13.4: 程序 13.3 寄存器重命名后的结果。

编号	指令	开始时间	结束时间
I1	`ldr x9, [sp, #-12]`	1	4
I2	`ldr x10, [sp, #-16]`	2	5
I3	`add x9, x9, x10`	6	6
I4	`ldr x12, [sp, #-20]`	7	10
I5	`ldr x11, [sp, #-24]`	8	11
I6	`sdiv x11, x12, x11`	12	18
I8	`ldr x13, [sp, #-28]`	13	16
I9	`mul x13, x9, x13`	17	19
I7	`str x11, [sp, #-24]`	20	20
I10	`str x13, [sp, #-28]`	21	21

我们对表格 13.4 中的指令再次进行依赖性和程序时延分析，得到图 13.4 所示的结果。其指令对程序时延的影响排序为：`I4=I5>I1=I2>I6=I8>I3>I9>I7=I10`。

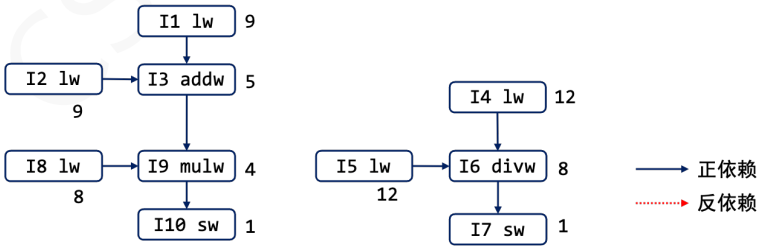


图 13.2: 代码 13.4 中指令间的依赖关系和时延

接下来，我们根据程序时延对指令再次进行重排，得到表 13.5 。由于不同指令的程序时延存在相同的情况，需要进一步考虑其先后顺序对程序时延的影响。在这段代码中，改变该表 `I6` 和 `I8` 的顺序或 `I7` 和 `I10` 的顺序会增大程序时延。

表 13.5: 程序 13.4 指令重排后的结果。

编号	指令	开始时间	结束时间
I4	`ldr x12, [sp, #-20]`	1	4
I5	`ldr x11, [sp, #-24]`	2	5
I1	`ldr x9, [sp, #-12]`	3	6
I2	`ldr x10, [sp, #-16]`	4	7
I8	`ldr x13, [sp, #-28]`	5	8
I6	`sdiv x11, x12, x11`	6	12
I3	`add x9, x9, x10`	8	8
I9	`mul x13, x9, x13`	9	11
I10	`str x13, [sp, #-28]`	12	12
I7	`str x11, [sp, #-24]`	13	13

13.1.4 应用考量

我们前面讨论的指令重排问题存在一定的局限性。首先，讨论中仅考虑了寄存器之间的数据依赖关系，但在实际应用中，还需要考虑内存之间的数据依赖关系，特别是针对同一内存单元的 `store-load` 依赖和 `load-store` 反依赖。因此，指令重排应在满足这些依赖关系的前提下进行，违背这些依赖关系会导致优化出错。其次，我们仅分析了单个代码块内部的局部数据依赖关系，但在实际优化中，应考虑整个控制流中的全局数据依赖关系和重排策略，这样能获得更好的优化效果。

此外，以 Tomasulo 算法 [2] 为代表的 CPU 乱序执行技术是一种在运行时实现指令调度和寄存器重命名的机制，被广泛应用。有兴趣的同学可以深入思考对比两种方法的优缺点。

13.2 窥孔优化

由于指令选择和寄存器的主要目标是实现与 IR 等价的翻译，不可避免会引入一些明显的冗余指令。窥孔优化的目标是识别这些明显的冗余模式并进行等价改写，从而对汇编代码进行优化。具体的冗余模式与编译器前期的实现有关，下面列举几种常见的冗余和窥孔优化模式。

13.2.1 冗余 mov

窥孔优化一般根据指令数窗口大小涉及不同的冗余模式。单条指令冗余模式主要是同一寄存器之间的 `mov`，可以直接删除。

```
mov x2, x2
```

两条 `mov` 指令的冗余模式，或 `mov-ldr` 的冗余模式，第一条 `mov` 指令可以删除。

```
mov     x10, #0
mov     x10, x9
```

```
ldr     x10, [sp]
mov     x10, x9
```

三条 `mov` 指令的冗余模式，第一条和第二条指令可以合并。类似的模式还有 `ldr-mov-mov`、`ldr-mov-ldr` 的情况。

```
mov     x10, #0
```

```
mov     x11, x10
mov     x10, x9
```

13.2.2 冗余 store-load

窥孔窗口大小为 2 时，如果遇到 str-ldr 指令组合，且内存地址相同的话，则第二条 ldr 可以替换为寄存器之间的 mov 操作；如果寄存器相同，则可以删除 ldr 指令。

```
str     x9, [sp]
ldr     x10, [sp]
```

13.3 利用 CPU 特性优化

可以利用 CPU 提供的一些特殊指令对程序进行优化，这方面典型的指令包括数据预取指令和 SIMD 指令。

数据预取指令指的是提前将数据从内存家在到 cache 中，从而避免仿存操作带来的性能损失。在 aarch64 中，数据预取指令是 PRFM，预取的单位一般是 cache line。可以通过不同的指令参数设置预取到 L1 或 L2 cache 中，以及只读或写操作。数据预取优化一般由程序员手动设置，目前尚无有效的自动数据预取分析算法。

```
; PRFM <type>, [<base>, <offset>]
PRFM PLDL1KEEP, [sp,  256] # 读操作，预取到L1 Cache
; PLDL2KEEP：读操作，预取到L2 Cache
; STL2KEEP：写操作，预取到L2 Cache
; 更多指令
ldr w1, [x0, 256]
```

SIMD 指令的全称是 Single Instruction Multiple Data，即通过一条指令实现多组数据的运算操作。aarch64 中提供的 SIMD 指令集扩展称为 neon [3]，其中含有 32 个 128bit 寄存器 v0-v31，可一次完成 4 对 32 位整数的四则运算。如下列汇编代码实现了两个向量 x 和 y 的加法运算。SIMD 的优化一般依靠程序员手动实现，如通过 C 语言 arm_neon 库提供的函数。目前尚无有效的自动 SIMD 优化分析算法。

```
ldr q0, [x8, _x@PAGEOFF]
ldr q1, [x8, _y@PAGEOFF]
add.4s v0, v1, v0
```

除了上述 CPU 核内特性以外，还可以考虑利用多核或多 CPU 进行并行优化。

练习

1) 下列 ARM 代码是编译时未进行任何优化得到的，分析其中的冗余和产生原因，手动对这段代码进行优化。

```
LBB0_0:
  str x0, [sp, #24]
  str x1, [sp, #16]
  str 0, [sp, #8]
  b LBB0_1
LBB0_1:                                 ;
  ldr x8, [sp, #8]
```

```
  ldr x9, [sp, #16]
  cmp x8, x9
b.ge  LBB0_3  b LBB0_2
LBB0_2:
  ldr x11, [sp, #8]
  ldr x9, [sp, #24]
  ldr x10, [sp, #8]
  add x8, x10, #1
  str x8, [sp, #8]
  ldr x8, [x9, x10, lsl #2]
  add x8, x8, x11
  str w8, [x9, x10, lsl #2]
  ldr x11, [sp, #8]
  ldr x9, [sp, #24]
  ldr x10, [sp, #8]
  add x8, x10, #1
  str x8, [sp, #8]
  ldr x8, [x9, x10, lsl #2]
  add x8, x8, x11
  str w8, [x9, x10, lsl #2]
  b LBB0_1
LBB0_3:
  add sp, sp, #32
  ret
```

2) 设计程序样例，使得指令调度算法可以在支持乱序执行的 CPU 上可以产生明显优化效果，并通过实验验证。

3) 设计程序样例，使得数据预取可以产生明显优化效果，并通过实验证明。

参考文献

[1] Arm Cortex-A72 Software Optimization Guide, https://developer.arm.com/documentation/uan0016/latest/.

[2] Robert M. Tomasulo, “An efficient algorithm for exploiting multiple arithmetic units.” IBM Journal of Research and Development, 1967.

[3] Neon Programmer Guide for Armv8-A Coding for Neon 4.0, https://developer.arm.com/documentation/102159/0400, 2023

后记

本教材作为编译入门教程，选用了简单易实现的 TeaPL 语言。该语言不涉及指针、字符串等高级功能，也未探讨自举实现。在 ACM/IEEE-CS 共同发布的《Computer Science Curricula 2023》[1]中，编译器并非独立的课程模块，而是作为编程语言模块的一部分。编译器的目标是实现编程语言的核心功能，二者密切相关。编程语言的研究内容十分广泛，许多与现代编程语言相关的重要特性在本教材中未涉及或未深入讨论，例如类型系统中的泛型和 Trait 支持、堆内存管理相关的智能指针与垃圾回收、并发功能、函数式编程和宏和元编程等高级功能，以及面向 GPU 的 AI 编译器等内容。

待续...

[1]Computer Science Curricula: `https://csed.acm.org/wp-content/uploads/2023/03/Version-Beta-v2.pdf`