编译授课笔记

徐辉

复旦大学　计算机科学技术学院

2025 年 2 月 1 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 TeaPL 编程语言——一门专为教学设计的语言，语法简洁且功能精炼。编译器的中间代码采用 LLVM IR，以实现与 LLVM 工具链的兼容性，而后端则主要面向开源的 RISC-V 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 TeaPL 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 TeaPL 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：

- 了解学习编译的意义
- 了解编译流程
- ⋆ 掌握运算符优先级解析算法

1.1 为什么学习编译原理？

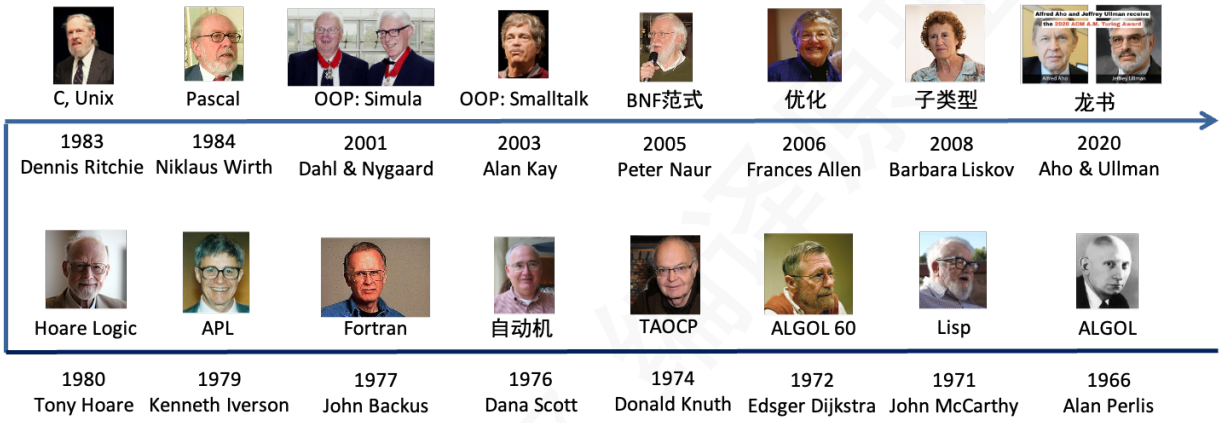


图 1.1: 主要成就与编程语言有关的图灵奖得主

- **有用**：当现有工具无法满足新的需求时，我们就需要开发新的“轮子”。例如，图灵奖得主 Leslie Lamport 因为需要一款便捷的排版工具而开发了 LaTeX；Mozilla 公司程序员 Graydon Hoare 为了构建一个安全高效的浏览器引擎，设计了 Rust 语言。近年来，随着深度学习和大模型的快速发展，许多适应这一趋势的编程语言和编译技术应运而生，其中包括 LLVM 的作者 Chris Lattner 开发的 Mojo，以及 OpenAI 的 Triton 等。另外，许多芯片厂商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优化。
- **经典**：许多历届图灵奖得主的成就与编程语言或编译密切相关，如图 1.1 所示。其中，多位获奖者因在早期参与 ALGOL 语言及其编译器的设计，贡献了重要的理论、算法或技术，包括 BNF 范式的作者 John Backus 和 Peter Naur、人工智能之父和 Lisp 语言的主要创造者 John McCarthy，以及《The Art of Computer Programming》一书的作者 Donald Knuth 等。对这一领域感兴趣的同学可以阅读《图灵和 ACM 图灵奖》 [2] 或访问 ACM 网站[1]进一步查阅相关资料。

[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

计算器能够识别算式，因此可以将其视为一种功能简单的编译器。本节将通过实现一个计算器为例，分析编译器的实现思路。

1.2.1 功能需求

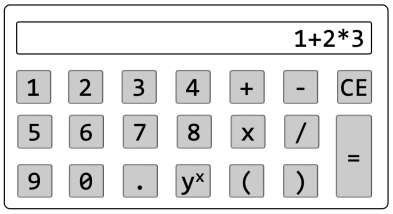


图 1.2: 目标计算器样例

我们假设目标计算器如图 1.2 所示，其主要功能参数如下：

- **操作数**：支持整数和小数。
- **运算符**：支持加、减、乘、除四则运算以及指数运算。
- **括号**：支持小括号。

1.2.2 实现思路

要实现上述计算器，通常需要经过以下基本步骤：

1) **词法分析**：扫描算式中的操作数、运算符和括号，生成标签流。

2) **句法解析**：根据运算符的优先级、结合律等规则组织标签，构建语法解析树。

3) **解释执行**：根据语法解析树计算运算结果。

词法分析：识别操作数和运算符

以算式 `123+456` 为例，词法分析过程应按顺序识别出操作数 `123`、运算符 `+` 及操作数 `456`，并将其转换为标签流：`<NUM(123)><ADD><NUM(456)>`。算法 1 描述了标签识别的思路。其关键点在于使用一个缓冲区`num`来记录当前已读取的操作数位。在此步骤中，我们暂不考虑算式的合法性问题（例如 `123++456`），也不区分 “`-`” 是负号还是减号。

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
```

句法分析：操作符优先级解析算法

算式解析问题非常经典。由于我们通常采用的算式表示形式是中缀（infix）模式，因此解析时必须遵循操作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，如`2^3^2` = `2^(3^2)`，而非 `(2^3)^2`。

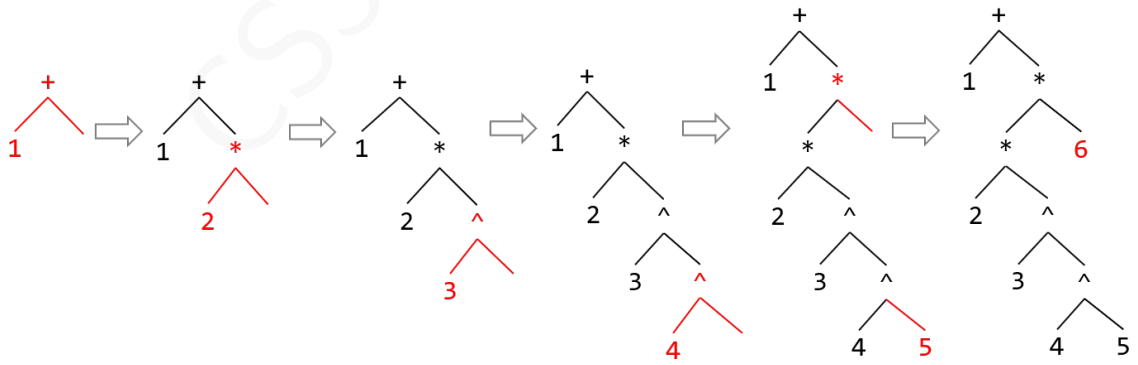


图 1.3: 算式 `1+2*3^4^5*6` 的解析过程

图 1.3 展示了算式 `1+2*3^4^5*6` 的解析过程。通过这个例子可以帮助我们梳理无括号算式的解析思路。算式解析最终得到的解析树为满二叉树，所有的叶子节点表示操作数，非叶子节点则为运算符。每个运算符在其父节点之前进行运算，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解析，并利用栈来记录已经读取的运算符。解析过程可分为以下三种情况：

- 如果当前遇到的运算符为左结合，且其优先级高于栈顶运算符的优先级，则应将该运算符作为栈顶运算符的右孩子节点，此时栈顶运算符的左孩子节点已存在。
- 如果当前遇到的运算符为左结合，且其优先级不高于栈顶运算符的优先级，则应将其作为栈顶运算符的父节点或祖先节点。具体做法是，从栈中弹出已读取的运算符，直到遇到一个优先级低于当前运算符的运算符为止。
- 如果当前遇到的运算符为右结合，则应将其作为栈顶运算符的右孩子节点。

图 1.4: 算式 `1+2*3^4^5*6` 的运算符优先级标注

Pratt 解析 [1] 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一个优先级数字，以便既能体现运算符的优先级，又能反映其结合性。对于左结合的运算符，其左侧优先级低于右侧；而对于右结合的运算符，左侧优先级则高于右侧。以图 1.4 中的优先级标注为例，运算符 "`+`" 和 "`-`" 的左右两侧优先级分别为 1 和 2，运算符 "`*`" 和 "`/`" 的左右两侧优先级分别为 3 和 4，运算符 "`^`" 的左右两侧优先级分别为 6 和 5。

算法 2 运算符优先级解析算法

```
1: input: token stream, precedence (init with 0)
2: output: binary parse tree
3: p[ADD] = (1,2),  p[SUB] = (1,2),  p[MUL] = (3,4),  p[DIV] = (3,4),  p[POW] = (6,5);
4: procedure PrattParse(cur,preced)
5:     let l = cur.next(); // next() moves cur to the next position and return the value of that position.
6:     if l.type() ≠ TOK::NUM then
7:         return ERROR;
8:     end if
9:     while ture do // corresponds to pop operators from the operator stack
10:        let op = cur.peek(); // peek() returns the value of the next position.
11:        match op.type() :
12:            case TOK::NUM ⇒ exit ERROR;
13:            case TOK::EOF ⇒ return l;
14:        end match
15:        (lp, rp) = p[op];
16:        if lp < preced then:
17:            return l;
18:        end if
19:        cur.next();
20:        let r = PrattParse(cur, rp);
21:        let l = CreateBinTree(op, l, r);
22:    end while
23:    return l;
24: end procedure
```

算法 2 展示了 Pratt 算法的伪代码实现。假设初始位置的优先级为 0，调用 PrattParse 函数即可得到图 1.3 中的语法解析树。具体过程如表 1.1 所示。

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：逆波兰表达式

基于语法解析树，我们可以通过后序遍历来完成算式的计算。对于计算器程序来说，另一种常见方法是将算式转换为逆波兰表达式（Reverse Polish Notation），这实际上是对语法解析树进行后序遍历得到的符号序列。例如，算式`1+2*3^4^5*6`的逆波兰表达式是：`1 2 3 4 5 ^ ^ * 6 * +`。逆波兰表达式的计算非常直观：按照顺序读取符号，如果遇到操作数则将其压入栈中；如果遇到运算符，则弹出栈顶的两个操作数，进行计算后将结果重新压入栈中。待字符串读取完毕时，栈顶的元素即为最终结果。

1.3 编译流程概览

由于编程语言比算式复杂，真实的编译器比计算器复杂得多。图 1.5 展示了编译的主要流程和技术分支。由于算式的复杂度较低，可以直接进行解释执行；而一般的通用编程语言是图灵完备的，因此用其编写的程序需要通过通用图灵机来运行，在实际应用中通常体现为虚拟机和实机两种方式。本学期后续的课程将对上述过程进行详细讲解。

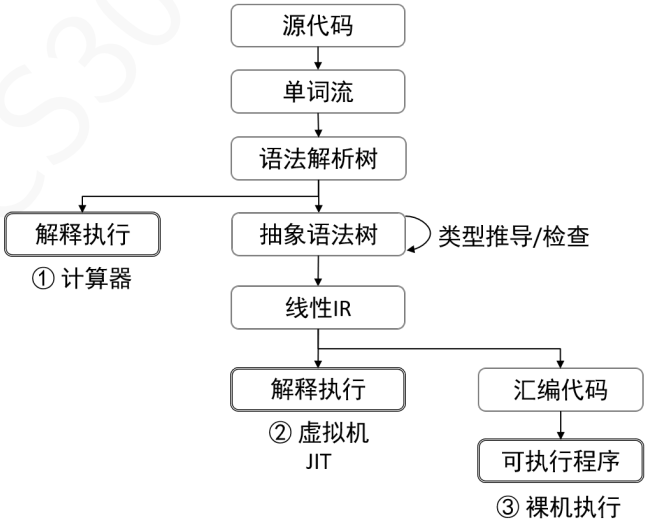


图 1.5: 编译流程

练习

1) 如下图所示，一些计算器输入算式：`1+60%+60%`，计算结果为 `2.56`，分析原因。

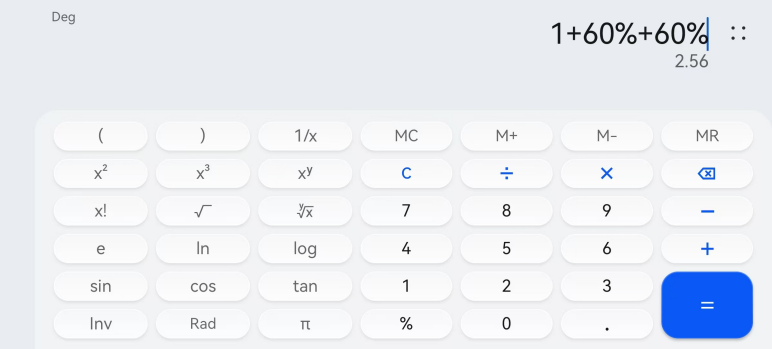


图 1.6: 某计算器截图

2) 实现 Pratt 算法并通过用例验证其正确性。

   (a) 不考虑括号的情况。

   (b) 支持小括号。

3) 你日常学习和工作中用到的哪些技术或工具与编译有关？举例说明。

参考文献

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

[2] 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 词法分析

本章学习目标:

- ⋆⋆⋆ 掌握正则表达式的表示方法与应用
- ⋆ 掌握 Thompson 构造法及其实现
- ⋆ 掌握子集构造法并理解其在自动机构建中的应用

2.1 词法声明：正则表达式

正则表达式（Regular Expression，简称 Regex）定义了字母表 Σ 上的字符串集合，通常由基本字符元素和一系列构造规则组成。

2.1.1 字符表示

表 2.1 列出了单个字符元素的主要表示形式及其对应含义。

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

单个字符的组合方式包括选择、连接和闭包三种基本形式。为了保持一般性，我们通常采用表 2.2 中的构造方式递归地定义正则表达式。

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	$S^{\{min,max\}}$	$x \in \{s_1..s_n \mid \forall 1 \le i \le n, s_i \in S, min \le n \le max\}$

在解析采用上述构造方式定义的正则表达式时，需要遵循一定的优先级顺序，即闭包 > 连接 > 选择。以正则表达式 “`a|bc*`” 为例，其对应的字符串集合（又称正则集）为 `{a,bc,bcc,...}`；而字符串

“abc” 不在该集合中。此外，在定义正则表达式时，可以使用括号，例如 “(a|b)c*”，其对应的正则集为 {ac,bc,acc,bcc,...}。

下面我们将使用正则表达式来设计计算器的词法规则。首先，计算器的输入符号来自一个有限的符号集：

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

表 2.3 定义了计算器的词法规则。

表 2.3: 计算器词法定义

标签类型	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	)

值得注意的是，一组正则表达式定义可能会出现二义性问题，即不同标签类型的规则存在交集，或者一个字符串同时符合两种标签类型的规则。此外，在使用正则表达式扫描字符串时，如果已匹配到特定标签，并且仍可继续向前移动，通常应采用最长匹配原则。例如，字符串 “123” 应被识别为一个标签 “<UNUM(123)>”，而不是三个标签 “<UNUM(1)><UNUM(2)><UNUM(3)>”。

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标签规则，如何自动生成对应的标签识别程序。从理论上讲，所有正则表达式都可以用确定性有限自动机（DFA：Deterministic Finite Automaton）表示；而 DFA 则可以进一步转化为等价的程序。

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。如果对于任意状态 $s_i \in S$ 和输入字符，至多存在一个转移目标状态，则该有穷自动机为确定性有穷自动机；否则，为非确定性有穷自动机（NFA: Non-Deterministic Finite Automaton）。

将一组正则表达式转化为 DFA 的过程大致可分为以下三步：

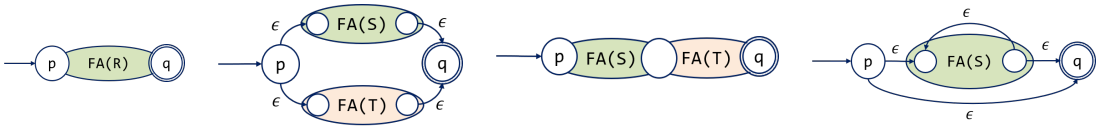
1) 为 Regex 构造 NFA

2) 将 NFA 转化为 DFA

3) 优化 DFA

2.2.1 为 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]。其基本思想是为正则表达式中的三种基本构造方式分别设计相应的 NFA 构造方法。通过从初始 NFA 开始，按照运算次序（逆序）递归展开正则表达式，并根据当前正则表达式的构造方式选择相应的 NFA 构造方法。



(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 开始，该 NFA 的边表示目标正则表达式 R。如果 R 的最后一层构造方式为选择 $S|T$，则按照图 2.1b 的方式展开；如果是连接 ST，则根据图 2.1c 展开；如果是闭包 S^*，则采用图 2.1d 的方式展开。图 2.2 以表 2.3 中 “<UNUM>” 标签对应的正则表达式为例，详细阐述了该递归构造过程。在此之前，我们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)`。

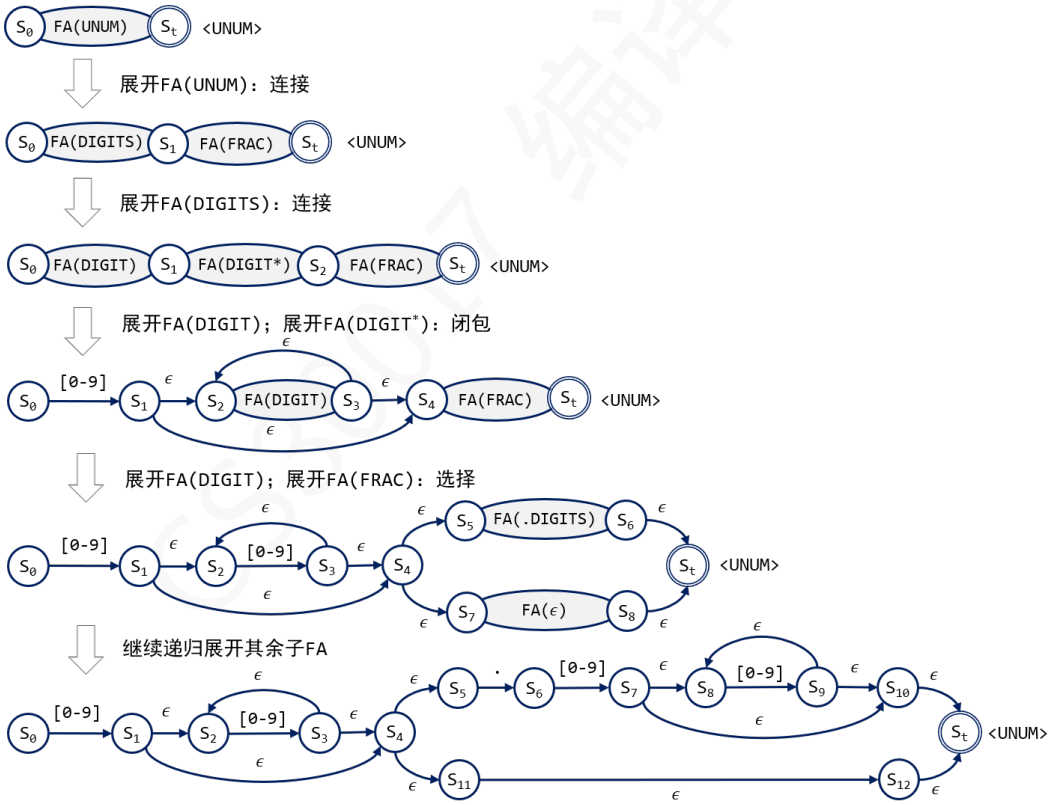
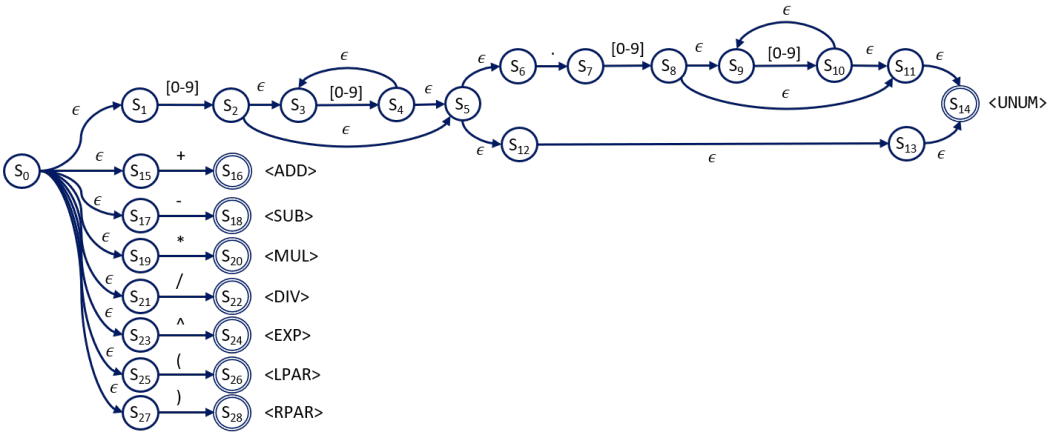
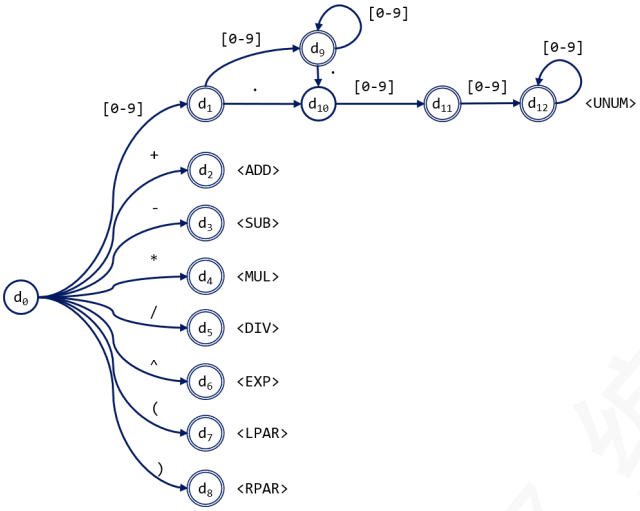


图 2.2: 应用 Thompson 构造法将标签 <UNUM> 对应的正则表达式转化为 NFA

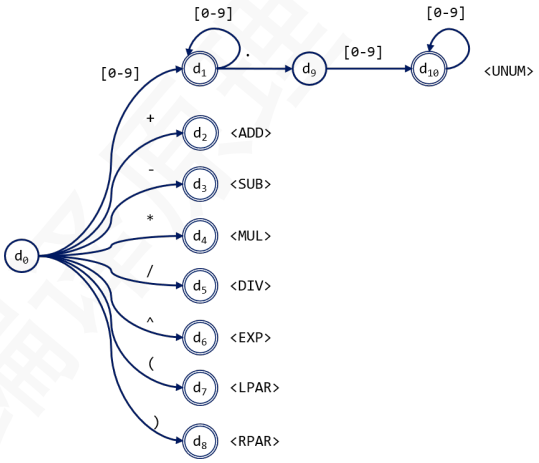
有了每种标签类型对应的 NFA，我们可以通过 ϵ 转移将它们合并为一个大的 NFA。图 2.3a 展示了表 2.3 中所有标签类型对应的 NFA。



(a) 合并所有标签后的 NFA



(b) NFA 转化后的 DFA



(c) 优化后的 DFA

图 2.3: NFA 转化 DFA

2.2.2 将 NFA 转化为 DFA

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

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

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

对于 NFA 上的状态集合 S，其 ϵ 闭包指 S 中所有状态的 ϵ 闭包的并集：

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

对于 NFA 上的状态集合 S，其 α 转移指 S 读取字符 α 后，所有状态的 ϵ 闭包的并集：

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

基于上述工具，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：

- D 表示 DFA 的状态集合，其中每个状态 d_i 是 NFA 中若干状态的集合；
- d_0 为 DFA 的初始状态，且 d_0 是 n_0 的 ϵ 闭包，即 $d_0 = Cl^{\epsilon}(n_0)$；
- D_f 为 DFA 的结束状态集合，且对于每个结束状态 $d_f \in D_f$，都有 $d_f \cap N_f \neq \emptyset$；
- Θ 为边的集合，对应 NFA 上状态集合 S 的 α 转移 $\Theta = \{(S, a, \Delta(S, \alpha)), \alpha \in \Sigma\}$。

算法 3 描述了如何将 NFA 转化为 DFA 的具体步骤。首先，将初始状态 *worklist* 设置为 d_0，然后分析 d_0 对于每个字符的 α 转移，得到一组新的状态集合并将其加入到 *worklist* 中。接着，针对 *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 转化为 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 转化为等价的 DFA。表 2.4 展示了具体的计算过程，最终得到的 DFA 如图 2.3b 所示。从图中可以看出，该 DFA 包含一些冗余的 ϵ 转移，前后状态之间可以合并，因此可以进一步对其进行优化。

表 2.4: 应用子集构造法将图 2.3a 中的 NFA 转化为 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 DFA 状态节点。对于两个同类型的状态节点 d_i 和 d_j（例如，都是初始状态、中间状态，或标签类型相同的结束状态），如果满足以下条件，则可以合并：

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

接下来，我们介绍一种基于上述思路的算法实现，但采用了基于分割的实现方式，即 Hopcroft 分割算法 [4]。如算法 4 所示，该方法将 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 $D \backslash D_f$，然后依次检查每个子集是否需要继续分割；重复这一过程，直到无法再分割为止，从而得到最优的 DFA。图 2.3c 展示了对图 2.3b 优化后的最终 DFA。除了上述方法，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。

算法 4 Hopcroft 分割算法

```
1: procedure OptDFA({D, d0, Df, Σ, Θ})
2:     let R = {Df, D\Df}
3:     let S = {}
4:     while S ≠ R do
5:         S = R
6:         R = {}
7:         for each si ∈ S do:
8:             R = R∪ Split(si)
9:         end for
10:    end while
11: end procedure
12: procedure Split(S)
13:    for each α ∈ Σ do:
14:        if α splits S into {s1, s2} then:
15:            Return {s1, s2}
16:        end if
17:    end for
18: end procedure
```

本章学习的正则表达式解析技术已经发展得相当成熟，并广泛应用于实际工程中。许多强大且易用的工具，如 Flex[1]，可以直接用于处理各种词法分析任务，大幅提升相关软件原型的实现效率。

练习

1) 假设某编程语言需要在词法分析环节支持日期识别，日期的格式是 YYYY-MM-DD，如 1980-01-12 或 0001-01-02 吗。回答以下两个问题：

   (a) 写出正则表达式，合法的格式要求 MM 必须是 01-12，DD 是 01-31。

   (b) 改进正则表达式，要求 DD 满足如下条件：如果月份是 01, 03, 05, 07, 08, 10, 12，日期范围为 01-31；如果月份是 04, 06, 09, 11，日期范围为 01-30；如果月份是 02，日期范围为 01-28。

2) 选择一门你熟悉的语言，例如 Markdown、Latex、HTML、python 或 Golang，并按照以下步骤进行词法分析：

   (a) 找出该语言中的词法标签；

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

(b) 使用正则表达式设计词法规则；

(c) 将正则表达式转化为 NFA；

(d) 将 NFA 转化为 DFA 并进行优化。

3) RESTful API 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
```

变量在 API 被调用时会替换为相应的参数值，如下列三条 API 调用日志所示，分别对应 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？

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.

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
- 了解 Chomsky 文法的分类

3.1 上下文无关文法

在上一节课中，我们已经使用正则表达式定义了计算器中的标签类型，但正则表达式无法进一步处理计算器表达式，特别是无法解决括号匹配问题。本节将学习上下文无关文法，它是一种比正则文法更强大的工具，能够表达更复杂的结构。

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

规则 3.1 尝试通过 CFG 定义合法的计算器表达式。从非终结符 `E` 开始，应用不同的语法规则逐步展开，最终可以得到所有合法的计算器表达式。

```
[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>
```

注意，CFG 要求每条文法规则的左侧只能有一个非终结符，不能包含其他限制条件。例如，$aX \mapsto ab$ 和 $bX \mapsto bc$ 的左侧对 X 的展开存在条件限制，因此不符合 CFG 的要求。

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，可能会出现多种解析方式，从而引发歧义。以算式 `1+2*3` 为例，图 3.1 展示了两种解析方式。这两棵语法解析树所对应的计算结果不同，只有解析树 1 是正确的。

语法规则 3.1 存在二义性的主要原因有两点：首先，未考虑运算符的优先级；其次，未考虑运算符的结合性，导致解析像 “`2^3^4`” 这种连续指数运算时可能出错。为解决优先级引起的二义性，应在语法设

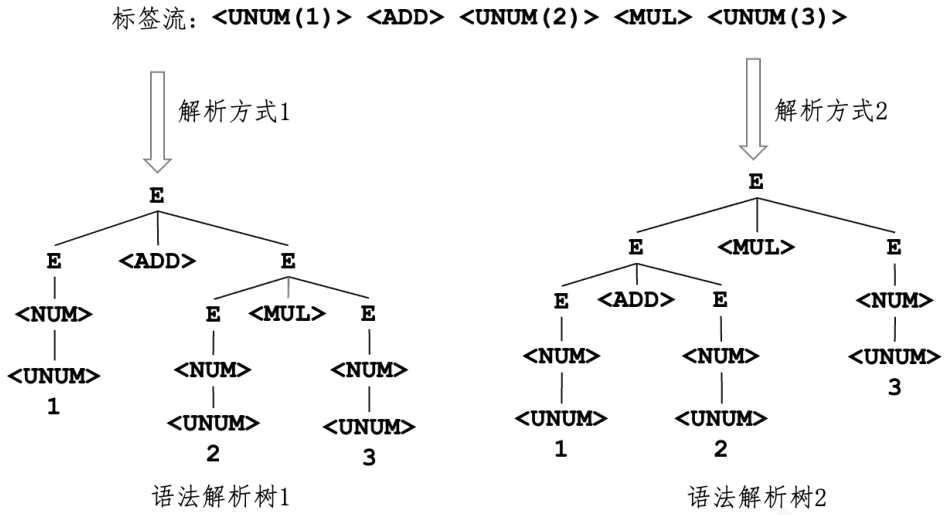


图 3.1: 解析算式 1+2*3

计中引入优先级关系，使低优先级运算的结果只能作为高优先级运算的操作数。为消除结合性引起的二义性，应在语法中强制左结合运算递归展开其左侧操作数，右结合运算递归展开其右侧操作数。

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

基于上述思路，对规则 3.1 进行改写，得到无二义性的规则 3.2 。改写过程主要包括以下几点：

- **区分运算符优先级**：使用 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。
- **区分操作数和结果优先级**：例如，使用 E 表示加减运算的结果，它的操作数可以是优先级更高的乘除运算结果 E1，也可以是同等优先级的加减运算结果 E。为了保证规则的等价性，E 也可以直接对应乘除运算结果 E1。
- **使语法规则满足结合性**：如运算符 OP1 为左结合，则规则应为左递归形式（E ↦ E OP1 E1）；如运

算符 OP3 为右结合，则规则应为右递归形式（E2 $\mapsto$ E3 OP3 E2）。

3.3 扩展 BNF 范式

由于上下文无关文法（CFG）较为复杂，我们通常使用扩展 BNF 范式（EBNF: Extended Backus-Naur Form）[2] 来描述具体的语言规则。EBNF 通过引入闭包和选择等构造方式简化了规则的表达。为了与上一节学习的正则表达式符号兼容，并提高书写的便捷性，我们采用表 3.1 中的符号来构造 EBNF。这种表示方法借鉴了 PEG 文法 [3] 中的构造符号设计。

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	$\alpha?$	匹配任意非终结符或字符串 α，或为 ϵ
闭包	`*`	4	$\alpha*$	匹配连续若干个（≥ 0）α
正闭包	$^+$	4	$\alpha+$	匹配连续若干个（≥ 1）α
排除	`-`	3	$\alpha - \beta$	匹配 α，但不能是 β；α 和 β 都是正则文法
连接		2	$\alpha\beta$	连续匹配 α 和 β
选择	`|`	1	$\alpha\|\beta$	匹配 α 或 β

语法规则 3.3 通过 EBNF 对规则 3.2 进行了改写，新规则更加简洁且具有更好的可读性。因此，在定义编程语言的语法规则时，我们通常采用这种表示形式。

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

上下文无关文法技术已经发展非常成熟，有许多非常好用的工具可以直接使用，如 Bison[1]和 ANTLR[2]。在这些工具中，用户可以通过单独声明运算符的优先级来筛选正确的解析方式，而不需要在规则中显式区分优先级，从而降低规则描述的复杂性。

3.4 TeaPL 语法规则

TeaPL 语言（Teaching Programming Language）是为编译课程设计的一门语言。该语言在使用上与 C 语言相似，但在语法设计上融合了 Rust 等新兴语言的特点，旨在简化语法解析和缺省类型的实现。图 3.2 展示了一段使用 TeaPL 编写的阶乘函数代码样例。接下来，我们将使用 EBNF 对其语法标准进行定义。

3.4.1 运算符和优先级

TeaPL 中使用的运算符及其特性如表 3.2 所示，且这些运算符的优先级和结合性设置与 C 语言标准保持一致[3]。

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

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

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

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

fn main() -> int {
    let r = factorial(n);
    ret r;
}
```

图 3.2: TeaPL 代码样例

表 3.2: TeaPL 中的运算符和优先级

优先级（C）	运算符	描述	结合性（C）	TeaPL 使用限制
8	-, !	单目运算符：负号、逻辑非	右	-后只允许跟数字，! 后只允许跟括号
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：比大小	左	比较对象不支持其它比较或逻辑运算
4	==, !=	比较运算符：等价性	左	比较对象不支持其它比较或逻辑运算
3	&&	逻辑运算符：与	左	
2	||	逻辑运算符：或	左	
1	=	赋值	右	不支持连续赋值

3.4.2 代码基本组成

$$\text{program} \mapsto (\text{varDeclStmt} \mid \text{fnDeclStmt} \mid \text{fnDef} \mid \text{structDef} \mid \text{comment} \mid ';')* \tag{3.4}$$

3.4.3 变量声明

$$\begin{aligned} \text{varDeclStmt} &\mapsto \text{'let'}\ (\text{varDecl} \mid \text{varDef})\ \text{';'} \\ \text{varDecl} &\mapsto \text{id}\ (\text{':'}\ \text{type})? \mid \text{id}\ \text{'['}\ \text{num}\ \text{']'}\ (\text{':'}\ \text{type})? \\ \text{varDef} &\mapsto \text{id}\ (\text{':'}\ \text{type})?\ \text{'='}\ \text{rightVal} \\ &\mid \text{id}\ \text{'['}\ \text{num}\ \text{']'}\ (\text{':'}\ \text{type})?\ \text{'='}\ \text{'\{'}\ \text{num}\ (,\ \text{num})*\ \text{'\}'} \end{aligned} \tag{3.5}$$

3.4.4 类型

$$\begin{aligned} \text{type} &\mapsto \text{primitiveType} \mid \text{structType} \\ \text{primitiveType} &\mapsto \text{int} \\ \text{structType} &\mapsto \text{id} \\ \text{structDef} &\mapsto \text{'struct'}\ \text{id}\ \text{'\{'}\ \text{fieldDecl}\ (\text{','}\ \text{fieldDecl})*\ \text{'\}'} \\ \text{fieldDecl} &\mapsto \text{id}\ \text{':'}\ \text{type} \mid \text{id}\ \text{'['}\ \text{num}\ \text{']'}\ \text{':'}\ \text{type} \end{aligned} \tag{3.6}$$

3.4.5 右值

```
 rightVal ↦ arithExpr
arithExpr ↦ (arithExpr ('+' | '-'))? factor
   factor ↦ (factor ('*' | '/'))? exprUnit
 exprUnit ↦ num | id | fnCall | '(' rightVal ')' | id '.' id | id '[' (id | num) ']'
```
(3.7)

3.4.6 函数声明

```
fnDeclStmt ↦ 'fn' fnSign ';'
    fnSign ↦ id '(' params? ')' ('->' type)?
    params ↦ param (',' param)*
     param ↦ id ':' type | id '[' num ']' ':' type
```
(3.8)

3.4.7 函数定义

```
       fnDef ↦ fn fnSign codeBlock
   codeBlock ↦ '{' stmt* '}'
        stmt ↦ varDeclStmt | assignStmt | callStmt | retStmt | ifStmt
             | whileStmt | breakStmt | continueStmt | ';'
  assignStmt ↦ leftVal '=' rightVal ';'
     leftVal ↦ id | id '[' (num | id) ']' | id '.' id
    callStmt ↦ fnCall ';'
     fnCall ↦ id '(' (rightVal (, rightVal)*)? ')'
    retStmt ↦ 'ret' rightVal? ';'
     ifStmt ↦ 'if' '(' boolExpr ')' codeBlock ('else' codeBlock)?
   whileStmt ↦ 'while' '(' boolExpr ')' codeBlock
   breakStmt ↦ 'break' ';'
continueStmt ↦ 'continue' ';'
```
(3.9)

3.4.8 布尔表达式

```
boolExpr ↦ (boolExpr '||')? andExpr
 andExpr ↦ (andExpr '&&')? boolUnit
boolUnit ↦ cmpExpr | '!(' cmpExpr ')' | '!'? '(' boolExpr ')'
 cmpExpr ↦ exprUnit ('==' | '!=' | '>' | '>=' | '<' | '<=') exprUnit
```
(3.10)

3.4.9 标识符和数字

最后，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

我们同样用正则表达式定义代码注释。

$$\texttt{comment} \mapsto \texttt{'//' (.* - '\textbackslash n') '\textbackslash n' | '/*' (.* - '*/') '*/'} \tag{3.12}$$

3.5 文法能力分类

根据表示能力的不同，Chomsky 将文法分为四个等级 [1]。如表 3.3 所示，正则文法的表示能力最弱，无法表示 $a^n b^n$（$n \in N$）这类要求字符出现次数相同的语言。所有正则文法都可以通过 CFG 表示，即产生式右侧不含非终结符。由于 CFG 不考虑上下文，它在编程语言语法规则设计中无法满足变量定义和使用时的类型一致性要求。因此，类型检查规则的设计需要使用上下文敏感文法。常见的编程语言如 C、Python 等通常采用 0 型文法。

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	$S \to a \vert b$	$a^n, n \in N$

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

练习

1) 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 CFG 规则。

   (a) 使用 BNF 表示。

   (b) 使用 EBNF 表示。

   。

3) Scheme 是一种函数式编程语言，属于 Lisp 语言家族的一个分支，其中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。不用全面，做到可以解析 factorial 函数并逻辑自洽即可。

表 3.4: Scheme 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

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.

4 自顶向下解析

本章学习目标：

- 了解 Earley 解析算法
- ⋆⋆ 掌握 LL(1) 文法的基本概念
- ⋆⋆ 掌握 LL(1) 解析算法的原理与应用

4.1 自顶向下解析思路

给定一套语法规则 G 和句子 s，找到由 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 的起始符号开始，逐步展开每个非终结符，直到最终生成的终结符序列与目标句子完全匹配。具体来说，本章采用最左推导策略，即在每一步中选择当前状态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解析方式；若 $s \notin L(G)$，则不存在任何有效的解析过程。

该语法解析问题的难点在于，每一步推导中，非终结符可能有多条展开规则，如何选择唯一合适的规则成为关键。一个基本思路是根据当前非终结符和目标终结符来选择合适的产生式。接下来，将分别介绍两种解析方法：Earley 算法和 LL(1) 文法。

4.2 Earley 解析算法

Earley 解析算法 [1] 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 5 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 5 Earley 解析算法

input: G: context-free grammar with a start symbol P; ts: token stream;
output: a parse tree;

```
1: procedure EarleyParse(ts, G)
2:     for each P → γ ∈ G do // 初始化，选取 G 中每一条以 P 开头的规则
3:         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]
4:     end for
5:     for each i in 0..ts.len() do // 遍历每一个终结符
6:         for each item in S[i] do // 遍历 S[i] 中的每一条规则状态
7:             match NextSymbol(item) :
8:                 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
9:                 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
10:                case NON-TERMINAL ⇒
11:                    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
12:            end match
13:        end for
14:    end for
15: end procedure
16: 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
17:    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
18:        S[i].add((B → αA∘δ, k)) // 移进完成的非终结符 A
19:        if δ == ϵ then // 规则 B → αA∘ 已经扫描完成
20:            Complete((B → αA∘, k), i) // 继续对 S[k] 中相关的规则执行完成操作
21:        end if
22:    end for
23: end procedure
24: procedure Predict((A → α∘Bβ, j), i) // 预测操作
25:    for each B → γ in G do
26:        S[i].add((B → γ, i))
27:    end for
28: end procedure
29: procedure Scan((A → α∘aβ, j), i) // 扫描操作
30:    if a == ts[i] then
31:        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
32:    end if
33: end procedure
```

下面以标签序列 `<UNUM(1)> '+' <UNUM(2)> '*' <UNUM(3)>` 为例[1]，展示 Earley 算法的解析步骤，具体内容参见表 4.1-4.6 。

表 4.1: 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 5	`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	-	-

[1]符号说明：`'+'` 即标签 `<ADD>`，`'*'` 即标签 `<MUL>`。

表 4.2: 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	-	-

表 4.3: 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 4.4: 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，如果下一个标签是'+' 或'-' 时有用；为节约空间，此处未列出相关规范项。

表 4.5: 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 4.6: 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]

4.3 LL(1) 文法和解析

4.3.1 LL(1) 文法

为了降低解析算法的复杂度，我们可以强制要求 CFG 文法具备某些特性。LL(1) 文法（Left-to-right, Leftmost, 前瞻 1 个字符）有两个基本要求：一是不能含有左递归，二是无回溯。接下来，我们将讨论这两个特性。

左递归与消除

对于一条语法规则，如果其右侧推导出的第一个符号与左侧非终结符相同，则存在左递归问题，例如：`E` $\mapsto$ `E OP1 E1`。左递归可能导致解析过程中的无限递归，从而使得解析无法终止。通常，可以通过修改语法规则来消除左递归，常见的方式如下。

$$
\begin{aligned}
&\texttt{X} \mapsto \texttt{X 'a'} \mid \texttt{X 'b'} \mid \texttt{'c'} \mid \texttt{'d'} \\
&\qquad\quad \Downarrow \\
&\texttt{X} \mapsto \texttt{'c' Y} \mid \texttt{'d' Y} \\
&\texttt{Y} \mapsto \texttt{'a' Y} \mid \texttt{'b' Y} \mid \epsilon
\end{aligned}
\tag{4.1}
$$

上一章定义的计算器语法规则中，第 [1] 和 [3] 条存在左递归问题。通过应用上述方法，可以消除其中的左递归，修改后的语法规则见式 4.2 。

$$
\begin{aligned}
&\texttt{[1] E} \mapsto \texttt{E1 E'} \\
&\texttt{[2] E'} \mapsto \texttt{OP1 E1 E'} \\
&\texttt{[3] E'} \mapsto \epsilon \\
&\texttt{[4] E1} \mapsto \texttt{E2 E1'} \\
&\texttt{[5] E1'} \mapsto \texttt{OP2 E2 E1'} \\
&\texttt{[6] E1'} \mapsto \epsilon \\
&\texttt{[7] E2} \mapsto \texttt{E3 OP3 E2} \\
&\texttt{[8] E2} \mapsto \texttt{E3} \\
&\texttt{[9] E3} \mapsto \texttt{NUM} \\
&\texttt{[10] E3} \mapsto \texttt{'(' E ')'} \\
&\texttt{[11] NUM} \mapsto \texttt{<UNUM>} \\
&\texttt{[12] NUM} \mapsto \texttt{'-' <UNUM>} \\
&\texttt{[13] OP1} \mapsto \texttt{'+'} \\
&\texttt{[14] OP1} \mapsto \texttt{'-'} \\
&\texttt{[15] OP2} \mapsto \texttt{'*'} \\
&\texttt{[16] OP2} \mapsto \texttt{'/'} \\
&\texttt{[17] OP3} \mapsto \texttt{'\^{}'}
\end{aligned}
\tag{4.2}
$$

无回溯语法

对于每个非终结符的任意两条规则，如果它们产生的首个终结符不同，则通过前瞻一个终结符可以选择正确的规则。当规则的首个字符是非终结符时，应递归展开该非终结符，直到遇到终结符。以规则 4.3 为例，如果当前规范项待展开的非终结符是 `X`，则根据下一个终结符是`'a'`、`'b'` 或`'c'`，总能选择出唯一的正确规则。

```
[i] X ↦ 'a'...
[j] X ↦ 'b'...
[k] X ↦ Y...
[p] Y ↦ 'c'...
...
```
(4.3)

当语法规则存在回溯问题时，可以通过以下方式提取左公因子来消除回溯。

```
X ↦ 'a' A | 'a' B | 'b'
        ⇓
X ↦ 'a' Y | 'b'
Y ↦ A | B
```
(4.4)

语法规则 4.2 中的第 [7] 和 [8] 条规则右侧的首个符号都是 `E3`，因此存在回溯问题。通过应用上述方法改写这两条规则，可以消除回溯问题。修改后的结果见语法规则 4.5 。

```
[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 ↦ '^'
```
(4.5)

4.3.2 LL(1) 文法解析

接下来，我们利用 LL(1) 文法的无回溯特性，构建一张解析表，记录每个非终结符可产生的首个终结符及其相应的规则。我们定义 $First(X \xrightarrow{[i]} \beta_1\beta_2...\beta_n)$ 表示应用 X 的第 i 条规则所能产生的首字符集合。如果 $\epsilon \notin \beta_1$，则有 $First(X \xrightarrow{[i]} \beta_1\beta_2...\beta_n) = First(\beta_1)$；如果 $\epsilon \in \beta_1 \& ... \& \epsilon \in \beta_i$，则有 $First(X \xrightarrow{[i]} \beta_1\beta_2...\beta_n) = First(\beta_1) \cup ... \cup First(\beta_{i+1})$。表 4.7 展示了语法 4.5 中每条规则对应的 $First$ 集合。

表 4.7: 语法 4.5 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。

由于句子标签序列中不包含 ϵ，$\epsilon \in First(X \to \beta)$ 对规则选择的帮助有限。因此需要对表 4.7 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）。

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

基于上述定义，我们可以准确定义无回溯语法的必要性质：

$$\forall 1 \le i, j \le n, First^+(X \to \beta_i) \cap \text{ First }^+(X \to \beta_j) = \emptyset$$

根据 $First^+$ 集合的计算方法，我们更新表 4.7 并消除其中的 ϵ 列，最终得到 LL(1) 解析表。结果如表 4.8 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 4.8: 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]		

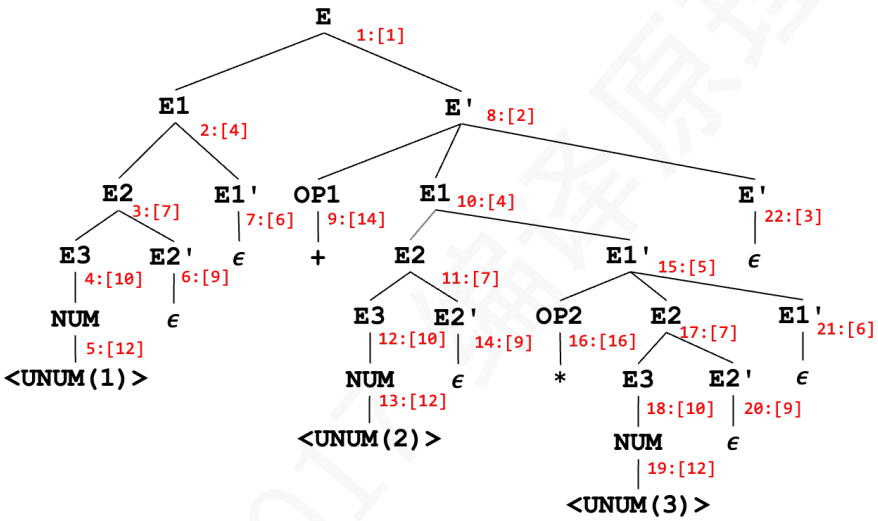


图 4.1: 应用表 4.8 解析 <UNUM(1)> '+' <UNUM(2)> '*' <UNUM(3)

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 4.8 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。

练习

1) 已知下列正则表达式 CFG 语法规则，应用 Earley 算法解析正则表达式 `ab*|c`。

```
[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)

2) 上述语法规则是否是 LL(1) 文法？如果不是将其改为 LL(1) 并构造解析表。

3) 分析比较 Earley 算法和 LL(1) 算法的复杂度。

参考文献

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

5 自底向上解析

本章学习目标：

- ⋆ 理解自底向上的解析思路
- ⋆ 掌握 SLR 文法及其解析方法
- 了解 LR(1) 和 GLR 解析方法

5.1 自底向上解析思想

自底向上解析是从句子开始，逐步将其规约为语法规则初始符号的过程。本章主要讲解 LR（Left-to-right, Right-most）自底向上解析方法，该方法包括两种基本操作：

- 移进：将句子中的下一个标签读入解析栈。
- 规约：根据语法规则 $X \mapsto \beta$，将当前解析栈顶的 β 规约为 X。

该问题的难点在于解析过程中某些步骤可能有多种操作选择，必须选取合适的操作才能正确解析。本章将以 SLR（Simple LR）文法为主，详细讲解 LR 解析方法，并在此基础上探讨更多扩展方法。

5.2 SLR 文法和解析

SLR 文法是一种特殊的 CFG 文法，其要求构建的 SLR 解析表不存在冲突。对于一套 SLR 文法，其解析表构建包括两个步骤：1）构造 LR(0) 有穷自动机；2）创建 SLR 解析表。下面以计算器文法为例，讲解 SLR 解析表的构造与应用方法。

5.2.1 构造 LR(0) 有穷自动机

由于计算器语法规则中初始符号对应的规则不唯一，为便于后续分析，我们在原文法基础上增加一条目标语法 S ↦ 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)

从规范项 S ↦ ∘ E 开始，我们对其产生的符号进行预测，得到一个初始的规范项集合（或规范族），如算式 5.2 所示，即 LR(0) 自动机的初始状态 S0。算法6 展示了具体的规范族计算过程。

算法 6 规范族生成算法

```
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
```

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

LR(0) 有穷自动机中的状态是规范族，边表示上下文无关文法中的符号。该有穷自动机描述了规范族在移进一个符号后的状态转移关系。构造过程从 S0 开始，分析可移进的符号及产生的新规范族；迭代该过程，直到没有新的规范族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) 自动机。

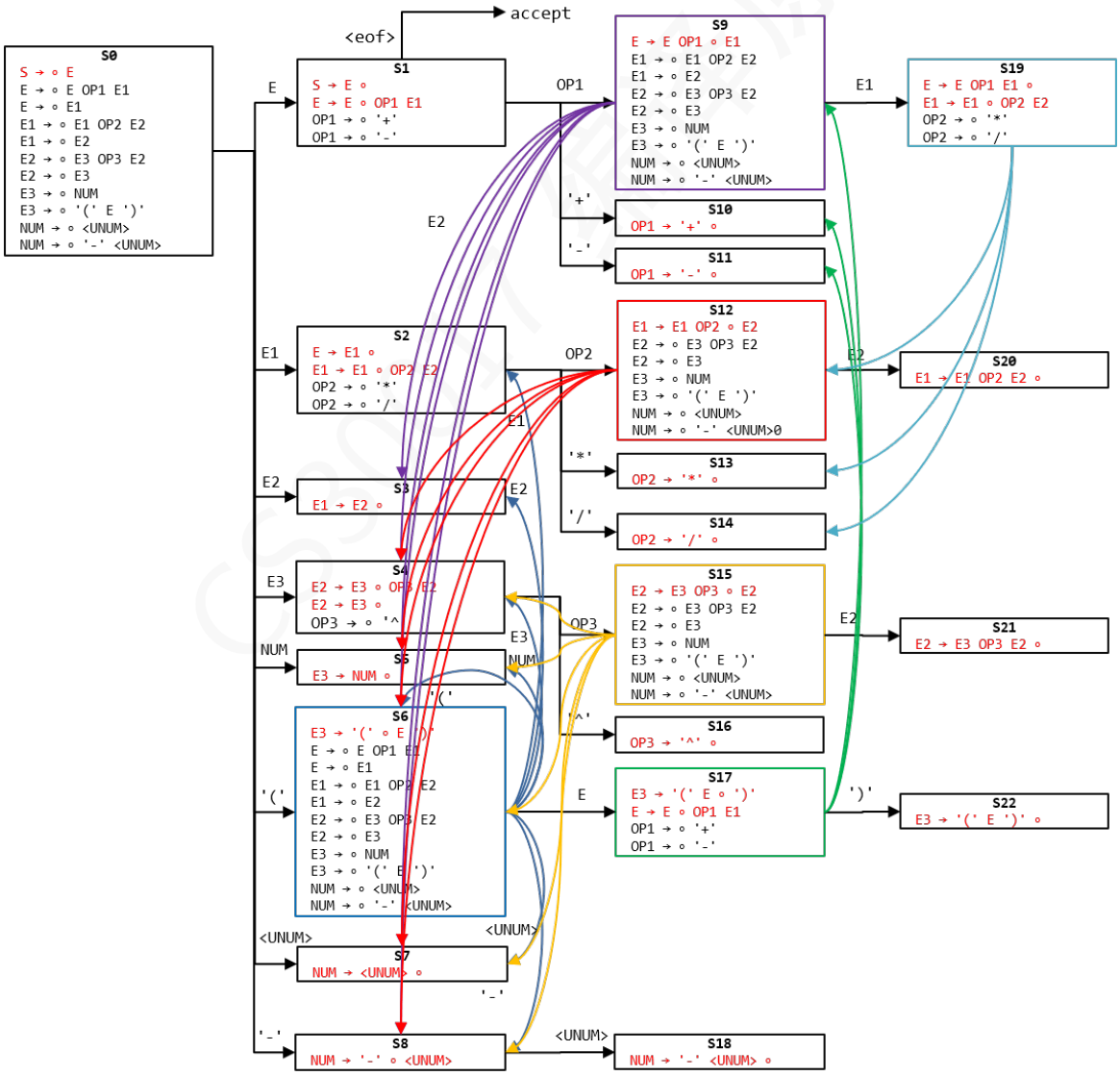


图 5.1: 语法规则 5.1 对应的 LR(0) 自动机

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表示，即可得到一张初步的 SLR 解析表。如表 5.1 所示，每一行表示一个 LR(0) 有穷自动机状态，每一列表示一个语法规则符号，每个单元格表示有穷自动机读入特定符号后的目标状态。此外，还有一些单元格表示可应用的规约规则，例如 R[2] 表示可将规范族中的规范项 `E` $\mapsto$ `E1` $\circ$ 根据语法规则 [2] 规约为 `E`。需要注意的是，该规约并非在所有情况下都可行，应满足的前提条件是下一个标签属于 $Follow(E)$。因此，每个状态可应用的规约操作仅出现在 SLR 解析表特定的列中。通常，根据语法规则符号是否为终结符，SLR 表被分为左右两部分：Goto（非终结符）和 Action（终结符），其中仅 Action 部分含规约操作。

表 5.1: 语法规则 5.1 对应的 SLR 解析表

规范族	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。解析过程需要使用两个栈分别记录状态和符号，每次根据栈顶状态以及下一个待读入标签选择具体的操作。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	待读入标签	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 LR 解析方法

SLR 文法的能力比较有限，如果 SLR 解析表的单元格如果存在多个操作选项，则不适合采用 SLR 解析方法。LR(1) 是一种典型的能力更强的自底向下解析方法，相比 SLR 其增强的方法是在构造 LR(1) 有穷自动机时即考虑每条规范项的 Follow 集合。如果两个规范族相同，但其中某条规范项的 Follow 信息不同，则创建一个新的 LR(1) 有穷自动机状态，从而避免潜在的操作选项冲突。为减小 LR(1) 状态增加带来的副作用，LALR 将规范族相同，但 Follow 信息不同的状态合并。因此，LALR 相比 SLR 在 Follow 信息的使用上更为精准。

虽然 LR(1) 的能力很强，但依然无法应对所有的 CFG 文法。如果前瞻 k 个字符则可以避免更多的操作冲突，即 LR(k) 文法 [1]。通用的自底向上 CFG 解析方法包括 GLR（Generalized LR）和 CYK。GLR [3] 是 LR 解析方法的正交扩展，即在出现冲突时广序遍历所可能有的解析方案，可与 LALR、LR(1) 等方法搭配。CYK（Cocke-Younger-Kasami [2] 则是有别于 LR 的一种采用动态规划思想的解析方法。

练习

1)（多选题）下列文法属于:

(a) LL(1)

(b) SLR

$$\begin{aligned} \texttt{S} &\mapsto \texttt{S A} \\ \texttt{S} &\mapsto \texttt{A} \\ \texttt{A} &\mapsto \texttt{a} \end{aligned} \tag{5.3}$$

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。

$$
\begin{aligned}
\texttt{Regex} &\mapsto \texttt{Regex '|' Concat} \\
\texttt{Regex} &\mapsto \texttt{Concat} \\
\texttt{Concat} &\mapsto \texttt{Concat Closure} \\
\texttt{Concat} &\mapsto \texttt{Closure} \\
\texttt{Closure} &\mapsto \texttt{Closure '*'} \\
\texttt{Closure} &\mapsto \texttt{Item} \\
\texttt{Item} &\mapsto \texttt{'(' Regex ')'} \\
\texttt{Item} &\mapsto \texttt{<Char>}
\end{aligned}
\tag{5.4}
$$

3) 满足 LL(1) 文法的语法规则一定符合 SLR 文法吗?

参考文献

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

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

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

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 TeaPL 的类型系统

TeaPL 采用静态类型系统，即所有标识符的类型必须在编译时确定。类型系统由类型和规则组成。在 TeaPL 中，基础类型包括标量类型 int 和 bool，复合类型如数组，以及函数类型。其中，bool 类型不能直接由用户使用，而是作为中间结果存在。此外，用户还可以使用 struct 自定义数据类型，在定义数据类型时其每个 field 的类型不能缺省。

TeaPL 的类型规则主要包括以下几条：

- 所有函数在声明时必须明确指定参数和返回值类型，不能省略；而变量声明则不做此要求。
- 相同标识符的作用域不能重叠或包含，例如同一函数内的局部变量重名（如代码 6.1 中的变量 x），或者局部变量与全局变量重名的情况。
- 对于全局标识符在文件中的声明和引用出现顺序不做要求。

```
fn foo(n: int) -> int {
    let x = n;
    if (n>0) {
        let x = n-1;
    }
    ret x;
}
```

代码 6.1: 类型错误举例：x 被重复声明

6.2 类型推导问题

在 TeaPL 中，标识符的类型在声明时可以缺省，但需要确定其具体类型才可以进行后续的编译，该问题称为类型推导。类型推导依赖于变量使用的上下文，且不一定能得出解。如果有解，则说明代码可类型（typable）；否则，编译器会提示类型错误或进行隐式类型转换。因此，类型推导也可以达到类型检查的效果；类型检查是类型已知情况下的类型推导特例。

这种类型推导一般是基于抽象语法树进行的。抽象语法树（Abstract Syntax Tree，缩写为 AST）相较于语法解析树 (Parse Tree 或 Concrete Syntax Tree) 是一种更精简的树形中间代码。AST 一般去除了语法解析树中的括号等冗余节点，并且对单一展开形式（只有一个孩子节点）的情况进行了塌陷处理，如将 A->B->C->D 缩短为 A->D。AST 在整个编译过程中可能会被编译器多次编辑，记录更新代码编译过程的中间结果。

类型推导通常基于抽象语法树（Abstract Syntax Tree，简称 AST）进行。相较于语法解析树（Parse Tree 或 Concrete Syntax Tree），AST 是一种更加简洁的中间代码表示。AST 去除了语法解析树中的冗余节点（如括号），并对单一展开形式（只有一个子节点）进行了塌陷处理。例如，将 A->B->C->D 简化为 A->D。在编译过程中，AST 可能会被编译器多次编辑，记录更新代码编译过程的中间结果。

类型推导一般包括两个步骤：1）标识符索引化；2）根据类型规则从 AST 中提取类型约束并求解。

6.3 标识符索引化

标识符索引化的目的是对标识符去重，解决代码中存在的同名标识符指代不同对象的问题。此步骤的输出包括一个缺少类型信息的符号表和索引化后的 AST。类型推导本质上是为去重后的标识符分配类型。

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 中的变量和函数定义节点，可以生成符号表。创建符号表时，无需考虑变量和函数的使用节点；如果某些变量的类型缺省，可在后续类型推导时填充其类型。

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 6.1 ）和两个函数的局部变量符号表（见表 6.2 和 6.3 ）。

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

fn main() { // scope main
    let r = fib(10) + g; // { scope 1
    // }
}
```

代码 6.2: TeaPL 代码

表 6.1: 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	0xd9c2	int
fib	global	0xd470	(int) → int
main	global	0xd318	(void) → void

表 6.2: 代码 6.2 中函数 main 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	0x82d0	未知

表 6.3: 代码 6.2 中函数 fib 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	0xd398	int
a	fib:scope1	0xd5b0	未知
b	fib:scope2	0xd2c2	未知
r	fib:scope3	0x1234	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，这一步骤通常与符号表的创建一起进行：在遇到标识符声明时创建新的索引，而在遇到标识符引用时关联已有的索引。

假设全局标识符都已经具备索引，下面以函数内部的标识符索引问题为例阐述一种标识符索引化算法。图 6.1 对该问题进行了抽象表示，其中红色节点表示声明一个局部变量，蓝色节点表示引用一个标识符；另外还包括声明 + 引用的情况，即使用其它标识符对新声明的变量进行初始化。算法 7 描述了标识符的索引化的过程。其主要思路是为每个函数维护一个标识符字典 `dict`，记录当前节点可用的标识符。在遍历 AST 时为每一个中间节点都维护一个字典 `subdict`，记录当前子树中声明的标识符，跳出该节点作用域时应将其子树中声明的标识符从 `dict` 移出。

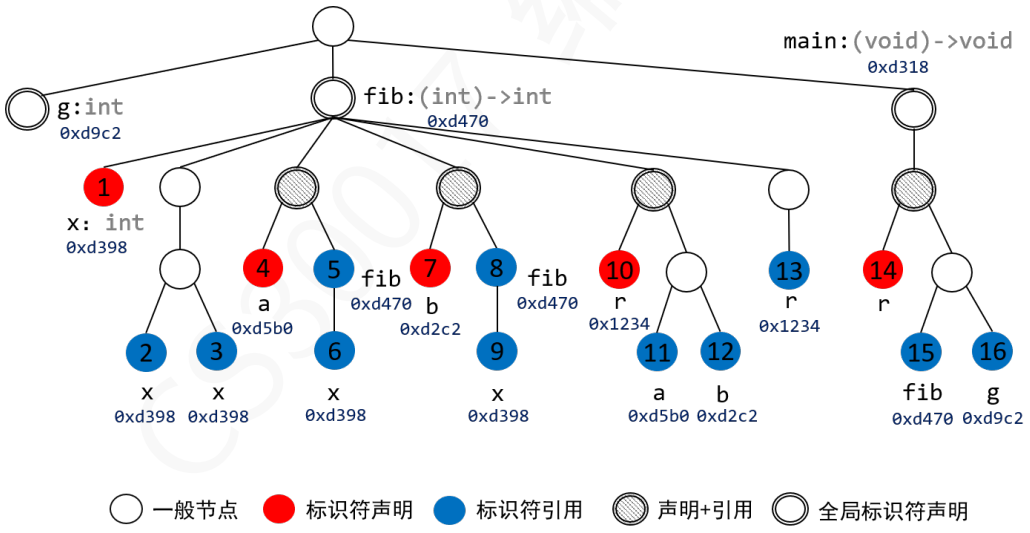


图 6.1: 变量索引问题举例

算法 7 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

类型推导是指为缺省类型的标识符分配具体类型。常用的类型推导方法是基于约束求解的 Hindley-Milner 方法 [1]。该方法首先为不同标识符定义不同的约束提取规则，然后通过提取代码中的类型约束并求解来确定标识符的类型；如果无法求解，则说明存在类型错误，此时需进行隐式类型转换或直接报错。表 6.4 定义了 TeaPL 语言的主要类型约束规则。

表 6.4: TeaPL 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`	$[\![I]\!] = \text{int}$	数字类型为 int
`X[I]: Ty`	$[\![X]\!] = \&\text{Ty}$, $[\![I]\!] = \text{int}$	声明 X 数组的类型为 &Ty
`{I1,...,In}`	$[\![I_1, ..., I_n]\!] = \&\text{int}$	数组类型为 &int
`{I; N}`	$[\![I; N]\!] = \&\text{int}$	数组类型为 &int
`X = Y`	$[\![X]\!] = [\![Y]\!]$	等号左右节点类型相同
`X = Y[Z]`	$[\![Z]\!] = \text{int}$, $[\![X]\!]=[\![*Y]\!]$, $[\![Y]\!]=\&[\![*Y]\!]$	数组解引用作为右值
`X[Z] = Y`	$[\![Z]\!] = \text{int}$, $[\![X]\!]= \&[\![Y]\!]$	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 {`
`  ret Y;`
`}`
	$[\![F]\!] = (\text{Ty1}) \to \text{Ty2}$, $[\![Y]\!] = \text{Ty2}$	函数声明/定义涉及的类型约束

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

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

`X.A = Y`	$[\![X.0]\!] = [\![Y]\!]$	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 6.2 的 AST 中，可以得到类型约束。以函数 fib 为例，其类型约束模型如下：

$$
\begin{aligned}
&[\![0xd9c2]\!] = int,\ [\![0xd9c2]\!] = [\![10]\!],\ [\![10]\!] = int \\
&[\![0xd470]\!] = (int) -> int,\ [\![0xd398]\!] = int,\ [\![0x1234]\!] = int \\
&[\![0xd398]\!] = [\![1]\!],\ [\![1]\!] = int,\ [\![0xd398 <= 1]\!] = bool \\
&[\![0xd5b0]\!] = [\![0xd470(0xd398 - 1)]\!],\ [\![0xd470]\!] = ([\![0xd398 - 1]\!]) \to [\![0xd470(0xd398 - 1)]\!] \\
&[\![0xd398]\!] = [\![1]\!] = [\![0xd398 - 1]\!] \\
&[\![0xd2c2]\!] = [\![0xd470(0xd398 - 2)]\!],\ [\![0xd470]\!] = ([\![0xd398 - 2]\!]) \to [\![0xd470(0xd398 - 2)]\!] \\
&[\![0xd398]\!] = [\![2]\!] = [\![0xd398 - 2]\!] \\
&[\![0x1234]\!] = [\![0xd5b0]\!] = [\![0xd2c2]\!] = [\![0xd5b0 + 0xd2c2]\!] \\
&[\![0xd318]\!] = (void) -> void \\
&[\![0x1234]\!] = [\![0xd470(10) + 0xd9c2]\!] = [\![0xd470(10)]\!] = [\![0xd9c2]\!]
\end{aligned}
\tag{6.1}
$$

由于上述类型约束关系都是等价关系，因此可采用并查集方法求解得到 $[\![0xd5b0]\!]$ = int，$[\![0xd2c2]\!]$ = int, $[\![0x1234]\!]$ = int。如果类型系统中包括子类型或范型，则类型约束关系为包含关系。

根据 TeaPL 语法规则设计，代码中有可能出现递归类型定义，这种情况通常会导致约束求解失败，应当抛出类型推导错误。此外，TeaPL 语法允许程序员编写部分路径缺少 `ret` 语句的代码。本文介绍的类型推导方法未考虑此类情况，因此可能会导致类型推导通过但实际存在类型错误的问题。为了解决这一问题，可以采用独立的方法进行检查。例如：如果函数最外层代码块包含 `ret` 语句，则视为通过；若最外层代码块缺少 `ret` 语句，则检查其内部并列的代码块是否都包含 `ret` 语句；对于仍然缺少 `ret` 的内层代码块，递归应用上述规则进行检查，确保所有可能的执行路径均有 `ret` 语句。

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 TeaPL 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: int) -> int {
    let r = 1;
    while (n>0) {
        r = r * n;
        n = n-1;
    }
    ret r;
}
```

代码 6.3: TeaPL 代码

3) 思考：如果对 TeaPL 的类型系统规则进行修改，分别支持以下功能，应如何修改类型推导方法?

   (a) 允许在不同作用域中存在相同的标识符，并且在引用时以作用域最小的标识符为准。

   (b) 允许同名函数，但函数签名不能相同。

参考文献

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

7 线性 IR

本章学习目标：

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

7.1 线性 IR

本章介绍一套线性中间表示（IR）的定义及其使用方法，该 IR 是 LLVM IR [1] 的一个子集。选择 LLVM IR 作为中间表示的优势在于，可以直接利用现有的 LLVM 工具链，如通过现有 C 语言编译器 clang 来了解不同功能对应的 LLVM IR，或使用 lli 解释执行中间代码。

```
; 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: 输出和执行 llvm ir 命令

代码 7.2 展示了一段 LLVM IR 示例，包括一个简单的全局变量声明 `@g` 和两个函数定义`%foo` 和`%main`。

```
@g = global i32 10 ; 声明全局变量g，类型为int32，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 ; 申请i32的栈空间，返回指针%x
    store i32 %0, ptr %x ; 将%0的值存入%x的内存单元
    %g0 = load i32, ptr @g ; 加载全局变量@g的值，命名为%g0
    ret i32 %g0 ; 返回%g0
}
define i32 @main() {
    %r0 = call i32 @foo(i32 1)
    ret i32 %r0;
}
```

代码 7.2: LLVM IR 代码示例

下面，我们对 TeaPL 用到的 IR 指令和相关知识进行详细介绍。

7.1.1 类型、变量和常量

在 TeaPL 中使用的 LLVM IR 类型包括以下几种：

- **标量类型**：包括不同长度的有符号整数，包括`i32`、`i8`、`i1`。
- **指针类型**：`ptr`或以`*`结尾的类型，如`i32*`、`i8*`。LLVM 17 版之后的指针无需再考虑具体的数据类型，可统一使用`ptr`表示。
- **数组类型**：若干个同一类型的对象，例如`[i32 * 2]`表示长度为 2 的`i32`数组。

- **自定义类型**：用户可使用type关键字定义类型，如%mytype = type {i32, i32}。

变量名、函数名和代码块标识在 LLVM IR 中都属于标识符。LLVM IR 中的标识符有两种基本类型：

- **局部标识符**：以%开头，后跟字母数字组合，例如%r1或%1都是合法的局部标识符。局部标识符仅在当前函数内有效。需要注意的是，lli 要求如果局部标识符（变量和代码块）采用纯数字编号命名，则必须从%0开始，并依次递增使用数字编号，否则无法成功执行。
- **全局标识符**：以@开头，后跟字母数字组合，如@g1或@1。全局变量在整个程序范围内有效。

LLVM IR 要求每个变量只能被定义一次，即只能通过 "=" 进行一次赋值或初始化。因此，LLVM IR 被称为静态单赋值形式（SSA）。采用这种形式是为了简化后续编译器优化算法的设计。我们将在下一章详细讲解静态单赋值形式的相关内容。

7.1.2 内存分配和数据存取

我们主要讨论函数栈帧上的内存分配。如代码 7.3 所示，局部变量的内存空间通过alloca指令进行分配，该指令返回指向内存单元的指针。内存分配的大小以字节为单位，因此，alloca指令的内存大小不能是i1类型。数据存取操作则通过store和load指令分别完成，前者用于存储数据，后者用于加载数据。store和load指令的一个参数需要明确指针类型信息。在 LLVM 14 之前的版本中，必须使用如 i32 * 这样的精确指针类型；而在 LLVM 17 之后，ptr 可以用来表示所有类型的指针。

```
; alloca指令形式: <ptr> = alloca <value type>
%x = alloca i32 ;返回指针类型: i32*
; store指令形式: store <value type> <value>, <ptr type> <ptr>
store i32 1, ptr %x ; 将整数1存入%x指向的内存
; load指令形式: value = load <value type>, <ptr type> <ptr>
%t1 = load i32, ptr %x ; 将%x指向内存的内容加载到%t1
```

代码 7.3: LLVM IR 代码示例：内存分配和数据存取

注意，TeaPL 原代码中的每一个变量在 IR 中都对应一块内存单元，数据可以通过store和load指令进行存取；而在 LLVM IR 中，会引入大量临时变量，这些临时变量可以视为虚拟寄存器。

LLVM IR 提供了数据类型转换的指令，例如通过zext..to将小数据类型扩展为大数据类型（高位补零），或者通过trunc..to将大数据类型转换为小数据类型（仅保留低位数据）。

```
; <dst> = zext <src type> <src> to <dst type>
%t2 = zext i1 %t1 to i32 ; 将i1类型的%t1转换为i32类型的%t2
; <dst> = trunc <src type> <src> to <dst type>
%t3 = trunc i32 %t2 to i8 ; 将i32类型的%t2转换为i8类型的%t3
```

代码 7.4: LLVM IR 代码示例：类型转换

数组和结构体元素的存取涉及寻址问题，必须先使用getelementptr指令获取目标元素的地址，才能进行数据存取。代码 7.5 和 7.6 分别展示了数组元素和结构体域的数据存取示例。

```
; <res> = getelementptr <pointee type>, <ptr type> <ptr>, <offset type> <offset>
%t1 = getelementptr i32, ptr %p, i32 1 ; 如果%p是指向i32的裸指针。
%t2 = load i32, ptr %t1;
%a = alloca [10 x i32] ;返回数组指针: [10 x i32]*
%t3 = getelementptr [10 x i32], ptr %a, i32 0, i32 1
;第一个索引0对应当前数组的基地址，第二个索引1表示数组的第2个元素的偏移量。
store i32 99, ptr %t3
```

代码 7.5: LLVM IR 代码示例：数组元素存取

```
%mystruct = type { i32, i32 }
%st = alloca %mystruct; <src> = getelementptr <pointee type>, <ptr type> <ptr>, <l1
  offset type> <l1 offset>, <l2 offset type> <l2 offset>
%t1 = getelementptr %mystruct, ptr %st, i32 0, i32 0
store i32 1, ptr %t1
```

代码 7.6: LLVM IR 代码示例：结构体域数据存取

7.1.3 算数运算

TeaPL 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

7.1.4 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
%t3 = icmp sgt i32 %t1, %t2 ; 大于
%t3 = icmp sge i32 %t1, %t2 ; 大于等于
%t3 = icmp slt i32 %t1, %t2; 小于
%t3 = icmp sle i32 %t1, %t2; 小于等于
```

代码 7.8: LLVM IR 代码示例：比较运算

7.1.5 控制流

控制流指的是程序执行过程中，代码块之间的跳转关系。IR 中的跳转指令是 br。代码块的定义以标识符和冒号开始，例如 “bb1:”；跳转到特定代码块时，需要在标识符前加上 “%”，如 “br %bb1”。br 指令不仅可以直接跳转到目标代码块，还可以加入条件判断，实现条件跳转。

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

```
    ...
```

代码 7.9: LLVM IR 代码示例：控制流

还有一条与控制流相关的条件赋值指令phi，下一章讲静态单赋值形式时会详细讲解，此处暂不展开。

```
; 程序运行时如果前一个代码块是<label 1>，则<result>的值是<value 1>;
; 如果前一个代码块是<label 2>，则<res>的值是<value 2>
; <res> = phi <type> [<value 1>, <label 1>], [<value 2>, <label 2>], ...
%t3 = phi i32 [%t1, %bb1], [%t2, %bb2]
```

代码 7.10: LLVM IR 代码示例：phi 指令

7.1.6 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:
    %t1 = xor i1 %a, true
    br i1 %t1, 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

7.1.7 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() {
    ; <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 翻译成 IR 代码的主要思路如下：

1) 遍历顶层 AST，生成函数和全局变量的 IR 表示；

2) 递归下降遍历每个函数的 AST，创建代码块编号和跳转关系；

3) 遍历每个代码块，逐条翻译代码块中的指令。

该翻译过程有两个主要难点，一是创建代码块及其跳转关系，二是关联指令的参数定义和使用（def-use）关系。

7.2.1 创建代码块及其跳转关系

LLVM 要求每个代码块必须以终结指令（如`br`或`ret`）结束。在递归下降遍历 AST 时，遇到以下几种情况时需要创建新的代码块：

- **函数定义**：创建代码块%bb0，添加返回指令 `ret <type> %tobeDetemined`。
- **if-else 节点**：创建三个代码块`%bb-true`、`%bb-false` 和后继代码块`%bb-after`。在当前代码块中添加条件跳转指令：`br i1 %tobeDetemined, label %bb-true, label %bb-false`，并将当前代码块中原有的终结指令移动到`%bb-after` 中，作为其终结指令。在%bb-true 和%bb-false 中添加跳转指令，直接跳转到`%bb-after`。
- **while 节点**：创建三个代码块、`%bb-cond`、`%bb-body` 和后继代码块`%bb-after`。在当前代码块中添加跳转指令，跳转到`%bb-cond`，并将当前代码块中原有的终结指令移动到%bb-after 中。在`%bb-cond` 添加条件跳转指令：`br i1 %tobeDetemined, label %bb-body, label %bb-after`；在`%bb-body` 中添加跳转指令，回到`%bb-cond`。

这种设计能够有效处理 TeaPL 中的各种控制流，包括 `while` 和 `if-else` 嵌套的情况。在实际实现中，代码块编号可采用`%bb` 后跟数字计数的形式；不建议使用纯数字编号，以避免编号不连贯导致的问题，这会使得 lli 无法正确执行。

7.2.2 指令参数的定义和使用

在翻译每条 IR 指令时，需要确定其参数。理想情况下，应尽可能复用已保存在寄存器中的结果，而非重新从局部变量中 load 到寄存器。但由于参数很可能定义于其它代码块，并且可能有多种定义，直接在翻译 IR 时解决这一问题较为复杂。因此，在翻译过程中，我们暂时不考虑性能优化，而是将参数的定义和使用关系限制在当前代码块内。具体来说，局部变量在使用前需要先 load，并在更新后立即 store，避免直接使用其它代码块中通过 load 或计算得到的数值。代码 7.15 和7.16 展示了阶乘函数的 TeaPL 源代码及其对应的 IR 代码。

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

代码 7.15: TeaPL 代码

```
define i32 @foo(i32 %0) {
bb0:
    %n = alloca i32 ; 参数内存单元
    %r = alloca i32
    store i32 %0, i32* %n ; 保存参数值
    store i32 1, i32* %r
    br label %bb1
bb1:
    %t1 = load i32, i32* %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, i32* %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, i32* %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, i32* %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, i32* %n
    %t7 = sub i32 %t6, 1
    store i32 %t7, i32* %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, i32* %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 等语法糖，已非常接近汇编代码形式，可以从主函数入口开始依次解释执行每条指令。解释执行的关键在于如何保存前序指令的运行结果，以确保后续指令能够获取正

确的数据。因此，解释执行通常需要配合虚拟机使用，虚拟机模拟函数栈帧和寄存器的操作。由于解释执行和虚拟机不属于本课程的重点内容，我们将不做深入探讨。

练习

1) 使用控制流指令，通过短路方法改写以下代码中的 and 指令，并利用 lli 工具进行测试。

```
define i32 @foo(i32 %0, i32 %1) {
     %t0 = alloca i32
     %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 TeaPL 代码翻译为线性 IR，并使用 lli 工具进行测试。

```
let a[10]:int = {1,2,3,4,5,6,7,8,9,10};
fn binsearch(x:int) -> int {
    let high:int = 9;
    let low:int = 0;
    let mid:int = (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]) {
        ret mid;
    }
    else {
        ret -1;
    }
}
fn main() -> int {
```

```
    let r = binsearch(2);
    ret r;
}
```

代码 7.18: TeaPL 代码片段

参考文献

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

8 静态单赋值

本章学习目标：

- ⋆⋆ 了解静态单赋值形式
- ⋆⋆⋆ 掌握基于循环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] 是一类特殊的线性 IR，其提出目的是为了简明表示变量的定义和使用（def-use）关系，从而便于后续的代码优化。SSA 通常有如下要求：

- 标识符定义：每个标识符只能被定义或赋值（def）一次，如需要修改其值，则只能使用其它的标识符。
- Phi 指令：如果由于控制流原因导致标识符在某处被使用时（use）对应多种不同来源的 def，应使用 phi 指令表示。
- 优化：使用最少数目的 phi 指令，简化数据流关系。

我们上一章使用的 LLVM IR 已经满足标识符只定义一次的要求，但并未使用 phi 指令。当存在不同控制流对应不同的变量值的时候，我们是使用 store-load 解决的，而非 phi 指令。接下来我们讨论如何将上一章使用的 LLVM IR 中的 load-store 替换为 phi，并最终转化为最优的 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

AST 翻译 IR 时为了降低 def-use 的复杂性，我们要求使用变量前必须先 load，更新变量值后必须立即 store，这样会引入大量冗余的 load 和 store 指令。本节我们采用基于循环迭代（Chaotic Iteration）的数据流分析方法消除 IR 中冗余的 load 和 store。

```
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

消除冗余 load

图 8.1 展示了一段 IR，其 bb2 代码块中将 n 的值 load 到 t4 和 t6 的操作是冗余的，可以直接使用 bb1 代码块中定义的 t1。其规律是针对同一个变量的两次 load 之间没有 store，说明该变量的值没有更新，因此后一次 load 是冗余的，可以使用前一次 load 的虚拟寄存器代替；反之如果两次 load 之间有 store，则说明之前的 load 失效。

基于上述分析，我们总结出与该分析相关的指令及其影响，即表 8.1 定义的 transfer 函数。我们可以将上述 transfer 函数应用于代码块中的指令序列，但如果涉及到控制流和循环，还需要更多的设计。循环迭代算法是一种应对控制流的常用分析框架，根据具体的分析任务需要设计不同的操作。如算法 8 所示，其对每条指令 `i` 进行分析，得到 `OUT[i]`。如果该指令有若干个前驱节点，则取并集处理。如果有循环则迭代该分析过程直到每个程序节点的分析结果不再变化为止。将该算法应用于图 8.1 便可得到所有变量在每个程序节点对应的可用虚拟寄存器。

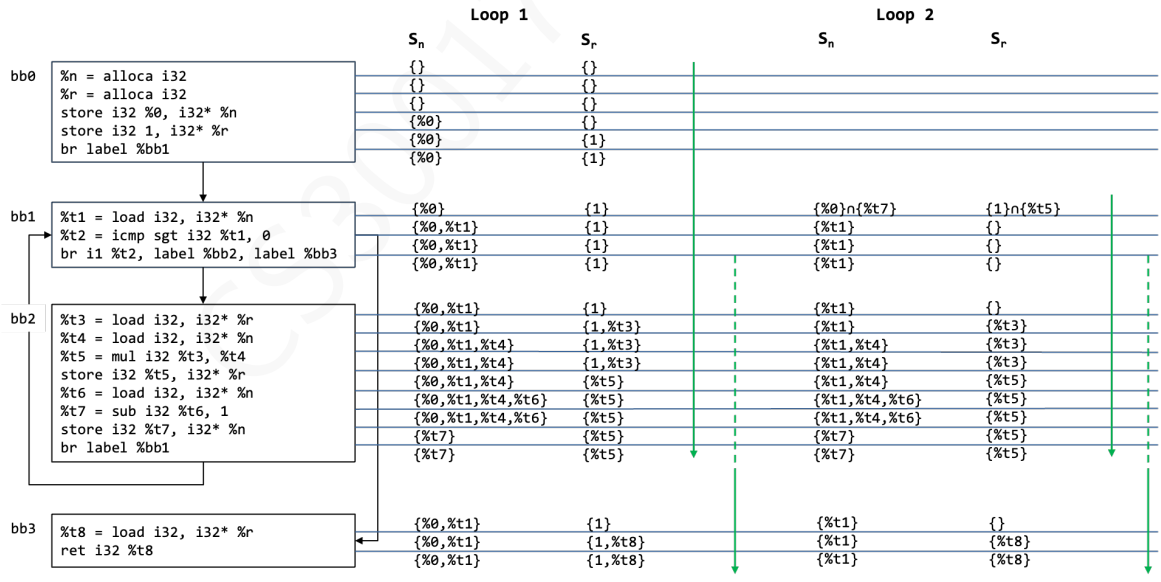


图 8.1: 冗余 load 指令分析

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`	$S_x = \{t\}$

算法 8 循环迭代算法：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 $p \in$ Predecessor(i) **do**
      $IN[i] \leftarrow IN[i] \cap OUT[p]$;
    end for
    $OUT[i] \leftarrow$ Transfer(i);
  end for
until $IN[i]$ and $OUT[i]$ stop changing for all i

消除冗余 store

如果一个变量的两条 store 语句之间没有 load 操作，则前一条 store 是冗余操作，可以直接删除。在该分析中，我们只需维护一个已经 store 的变量的集合即可，无需像冗余 load 分析一样针对每个变量设置一个集合。表 8.2 总结了不同指令对应的 transfer 函数。根据算法 9 对 IR 控制流图进行逆向遍历可识别出所有符合条件的冗余 store 操作。图 8.1 不涉及冗余 sotre。

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 9 循环迭代算法：可用 store 分析

Require: IR and variables of a target function

for each $i \in irs$ **do**
  $IN[i] \leftarrow \emptyset$;
  $OUT[i] \leftarrow \emptyset$;
end for
repeat
  for each $i \in irs$ **do**
    for each $s \in$Successor(n) **do**
      $OUT[i] \leftarrow OUT[i] \cap IN[s]$;
    end for
    $IN[i] \leftarrow$ Transfer(i);
  end for
until $IN[i]$ and $OUT[i]$ stop changing for all i

8.2.2 转换为静态单赋值形式

这一步的目的是消除 IR 中所有针对局部变量的 store 和 load 指令，即不使用栈帧内存。其关键问题是有些 load 可能对应多个控制流带来的不同定义，需要引入 phi 指令来表示。以图 8.2 为例，bb1 中的

load(t1) 可能对应 bb0 中的%0（路径：bb0->bb1）或 bb2 中的 t7（路径：bb2->bb1）。下面介绍 LLVM IR 到 SSA 的翻译方法，分为两步：1）数值流分析；2）使用 phi 指令替换 store-load。

数值流分析

对于数值流分析，我们可以继续采用循环迭代方法分析 store 对 def-use 关系的影响，即正向遍历控制流图，遇到 store 指令则应用表 8.3 中定义的 transfer 函数，遇到合并节点则取并集。

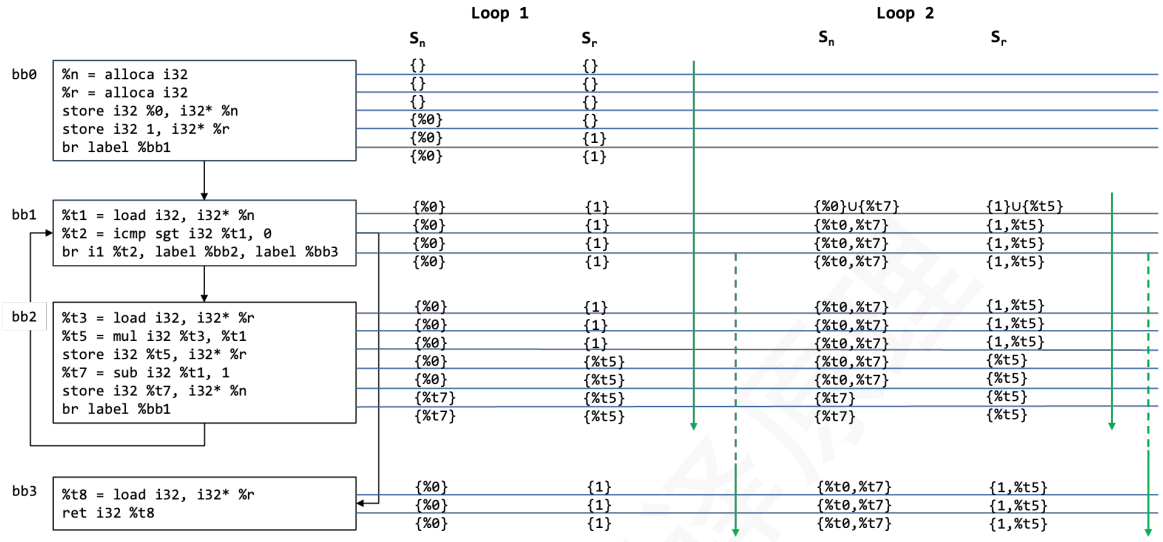


图 8.2: 数值流分析

表 8.3: Transfer 函数定义：def-use 分析

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

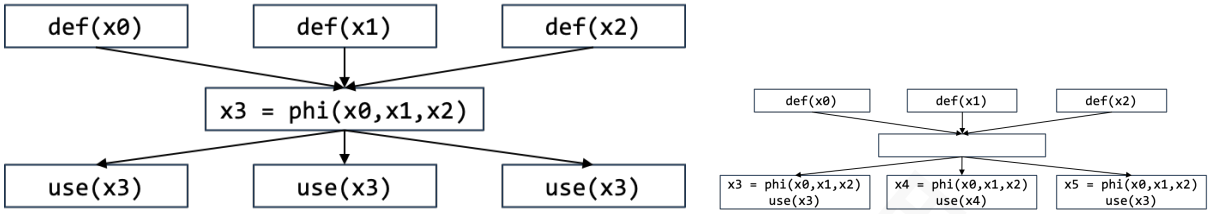
```
define i32 @fac(i32 %0) {
bb0:
    br label %bb1
bb1:
    %n0 = phi i32 [%0 %bb0], [%t7:%bb2];
    %r0 = phi i32 [1 %bb0], [%t5:%bb2];
    %t2 = icmp sgt i32 %n0, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t5 = mul i32 %r0, %n0
    store i32 %t5, i32* %r
    %t7 = sub i32 %n0, 1
    store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

代码 8.2: IR 代码

使用 phi 指令替换 store-load

确定了每个程序节点可能的变量数值定义，只需在存在多个来源的数值定义处使用 phi 指令即可。对于图 8.2 来说，必须在 bb1 中插入 phi(n)，而 phi(r) 在 bb1 或 bb2 中插入均可。

值得注意的是，纯寄存器表示形式的 IR 将变量的 def-use 关系显式表示出来，但未必可以有效优化 def-use 关系的复杂度。为了达到最优的 phi 指令使用方法，应当尽量在最靠近起始代码块的地方插入 phi 指令。以图 8.3a 为例，使用时寄存器表示后，其 def-use 关系数量是 3*3，并且随控制流深度增加呈指数增加。如果将 phi 指令前移（图 8.3b ），def-use 关系变为 3+3，避免了指数爆炸问题。



(a) 非最优 phi 指令放置　　(b) 最优 phi 指令放置

图 8.3: Phi 指令放置位置与 def-use 关系的优化举例

8.3 基于支配边界的 SSA 构造方法

对于如何确定 phi 指令的放置位置，实际中更常用的是基于支配边界的构造方法，即先确定 phi 指令的放置位置，再通过数据流分析更新 IR 中的虚拟寄存器使用关系。

8.3.1 支配边界

定义 4 (支配)**.** 给定有向图 $G(V,E)$ 与起点 v_0，如果从 v_0 到某个点 v_j 均需要经过点 v_i ，则称 v_i 支配 v_j 或 $v_i \in Dom(v_j)$ 如果 $v_i \neq v_j$，则称 v_i 严格支配 v_j 或 $v_i \in IDom(v_j)$。

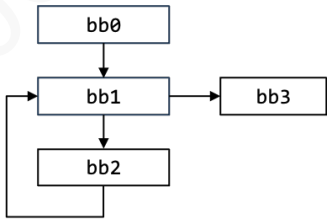


图 8.4: 控制流图举例

支配节点的计算可以采用基于循环迭代的分析方法实现，即正向遍历控制流，并维护从起始节点到当前节点所有经过的代码块；如果遇到分支节点取交集即可。以图 8.4 为例，每个节点的支配节点分析结果如下：

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

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

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

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

定义 5 (支配边界)**.** v_i 的支配边界是所有满足条件的 v_j 的集合：

- v_i 支配 v_j 的一个前序节点
- v_i 并不严格支配 v_j

有了控制流图每个节点的 $v_j \in V$ 的前驱节点集合 P_j 和支配节点集合 D_j，则节点的支配关系可以直接基于集合分析得到，即 $\forall v_p \in P_j, \forall v_i \in D_p \backslash ID_j, v_j \in DF(v_i)$。图 8.4 中每个节点的支配边界分析结果如下：

$$DF(bb_0) = \emptyset$$

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

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

$$DF(bb_3) = \emptyset$$

如果在某节点对变量 x 进行了赋值，则应在其支配边界放置 phi(x)。以图 8.4 为例，由于 bb2 的支配边界是 bb1，并且 bb2 对 r 和 n 进行了赋值，因此应在 bb1 插入 phi(r) 和 phi(n)。

8.3.2 更新 def-use 关系

确定了 phi 指令后，可以采用第 8.2.2 节介绍的数值流分析方法确定 phi 的具体参数。然后更新虚拟寄存器的定义和引用关系即可。

练习

1) 分析图8.5 中每个节点的支配边界。

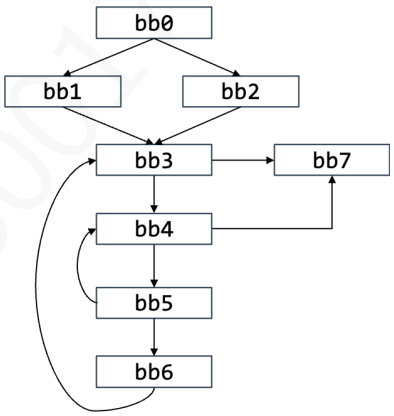


图 8.5: 控制流图

2) 代码 8.6 是 Eratosthenes 质数筛选算法的 IR，通过以下步骤将其转化为 SSA 形式：

(a) 分析每个代码块的支配边界；

(b) 插入 phi 节点；

(c) 更新虚拟寄存器引用关系。

参考文献

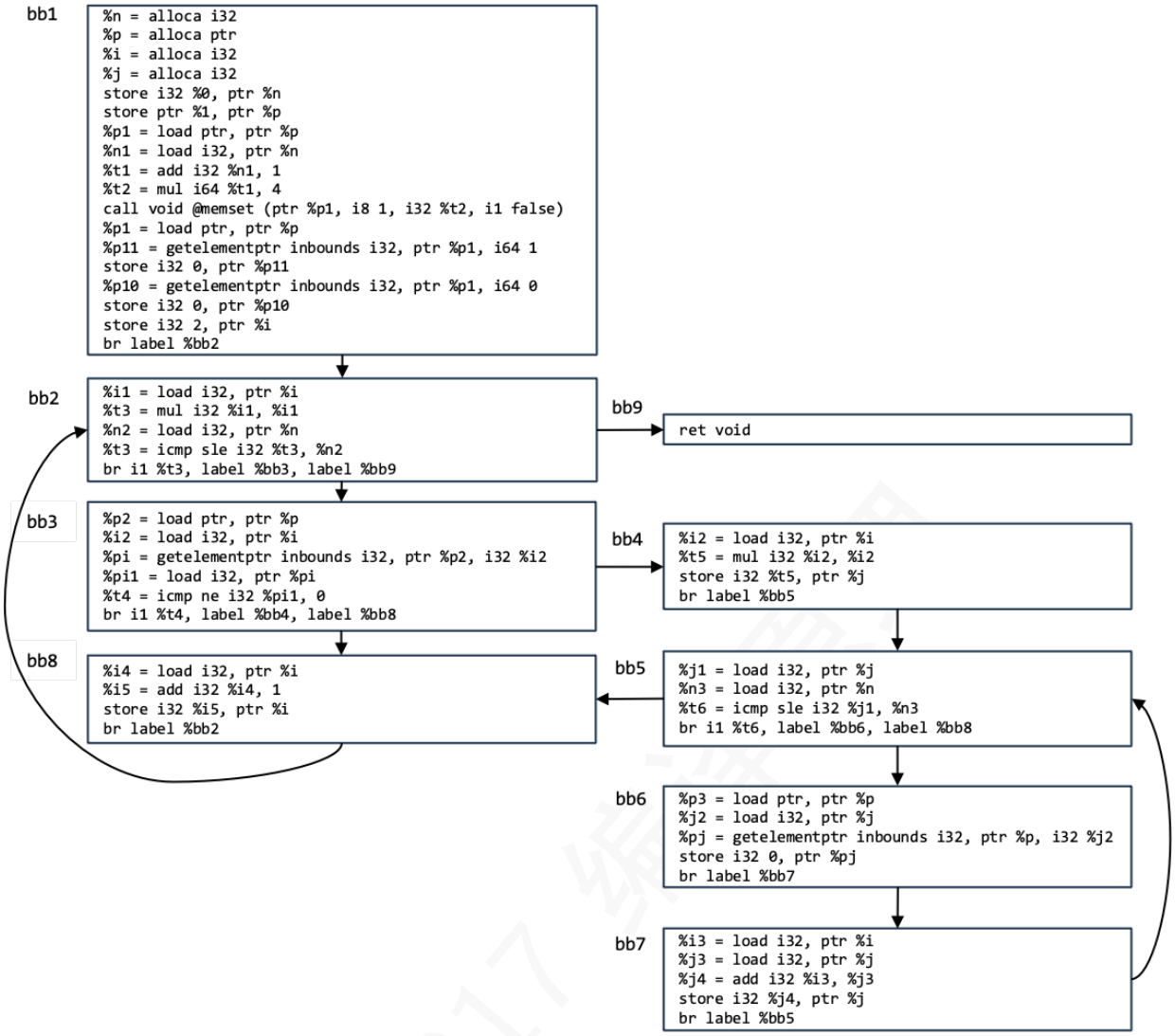


图 8.6: 质数筛选算法对应的 IR

[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.

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。

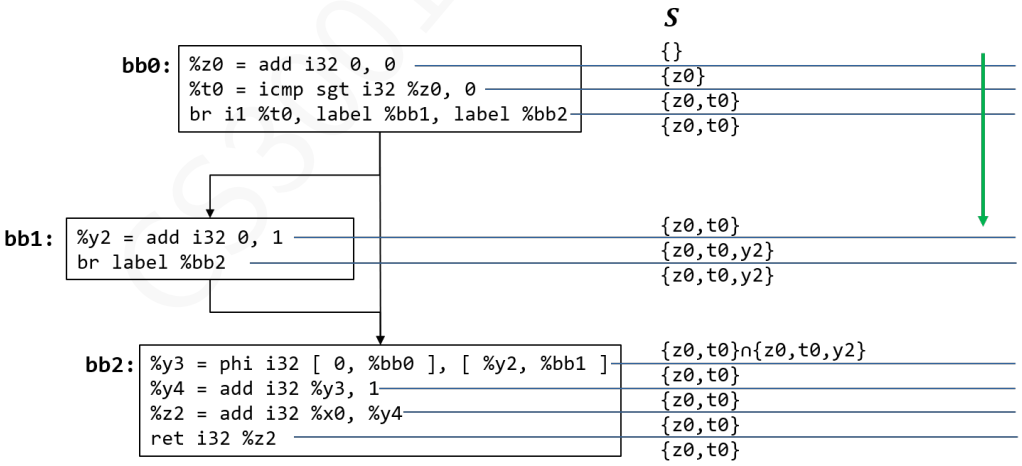


图 9.1: 面向 LLVM IR 的常量分析

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。

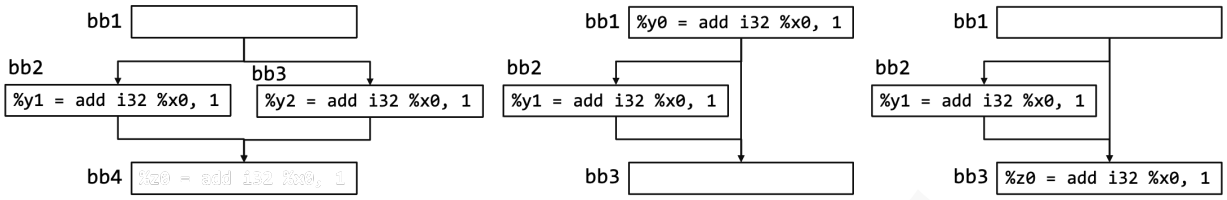
9.3.3 全局值编号

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	$S_r = S \cup r,\ S = \text{Find}(\text{icmp}, \text{sgt}, S_{op1}, S_{op2}))$
zext/trunc	%r = zext i8 op1 to i32	$S = S \cup r,\ S_r = \text{Find}(\text{zext}, \text{i32}, S_{op1})$

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

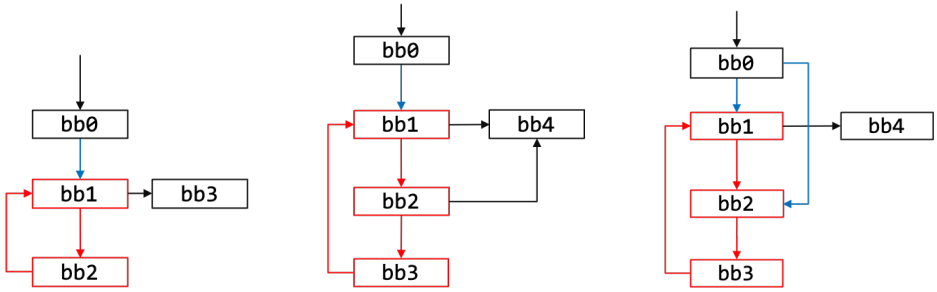
循环优化是对于提升代码运行效率效果最为显著的优化手段之一，其核心思想是将循环内重复执行的代码提前至循环外的支配节点，避免代码被重复执行。下面先介绍面向代码控制流图中循环路经的检测方法，再介绍具体的循环优化技巧。

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。

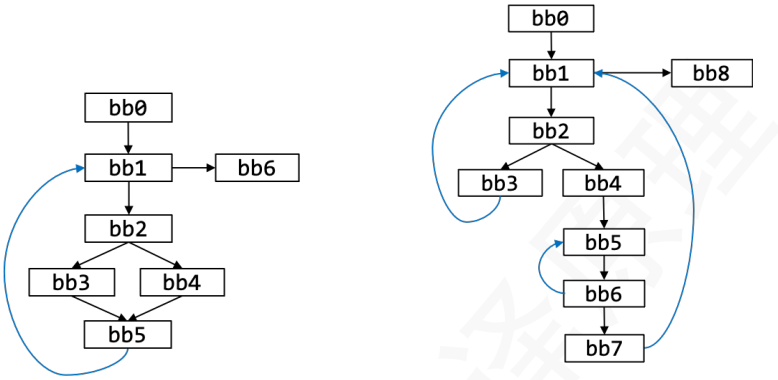
图 9.3 展示了几个控制流图的例子。其中，图 9.3a 是 while 循环的控制流图，很明显 bb1->bb2->bb1 是一个自然循环；图 9.3b 中的 bb1->bb2->bb3->bb1 也是自然循环，通过在循环体内部加入一个条件跳转分支和 break 语句可以达到此效果；图 9.3c 中的循环存在两个入口，因此是非自然循环。

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

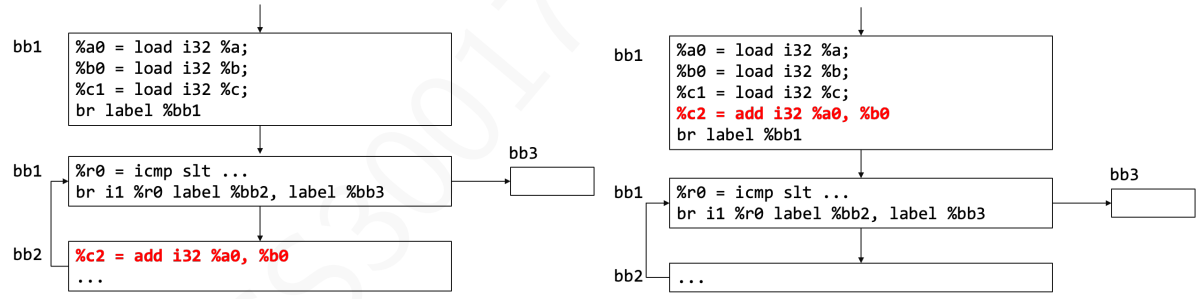
有两个公共节点 `bb1` 和 `bb2`，以及一条公共边 `bb1->bb2`，这两个循环为相切的关系。存在嵌套或相切关系的两个循环不需要经过其它节点可以组成一个大的强联通分量。自然循环之间不会出现相交，即存在多个入口节点的情况。

基于上述分析，我们可以设计出算法 10 ，用于检测 IR 中的自然循环。

算法 10 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

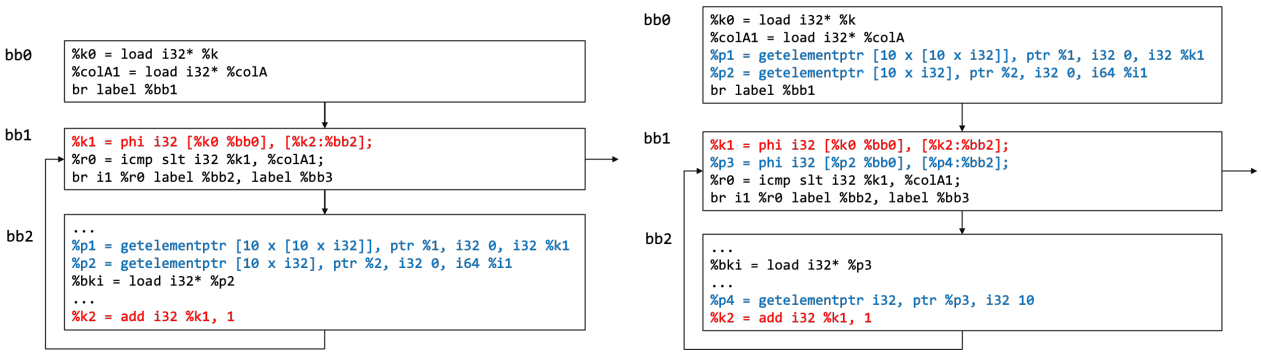


(a) 原始 IR　　(b) 优化后的 IR

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。

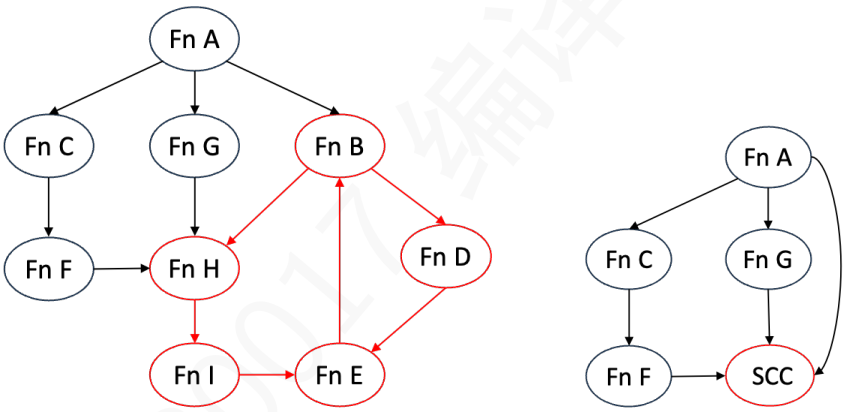
结合上述因素，一个函数越小，则将其内联带来副作用的概率越小。其它情况下，内联是否会带来正优化收益取决于很多因素，难以给出确切的建议 [1]。编译器一般不会主动对循环和递归调用（尾递归除外）进行内联。

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 11 ）。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 11 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 12 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 12 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：

- ⋆⋆ 掌握基础的 ARMv8-A 指令和函数调用规约
- ⋆⋆⋆ 掌握指令选择问题和建模方法

11.1 ARMv8-A 指令集

本章学习如何将 IR 代码翻译为 AArch64 指令集的汇编代码，目标指令集版本是 ARM-v8A。我们先介绍单条 IR 指令对应的 AArch64 指令，然后介绍针对整段 IR 代码翻译的指令选择问题和解法。我们暂不考虑具体可用的寄存器编号和数目，均使用 `w0-wn`（32 位寄存器）或 `x0-xn`（64 位寄存器）表示。

ARMv8-A 是精简指令集，其主要特点是访存与运算由不同的指令分别完成。代码 11.2 展示了一段简单的 hello world 汇编代码。这段代码先通过 `str` 指令将返回地址寄存器 `x30` 保存到栈上，然后通过 `adrp` 或取字符串`"Hello World!"` 字符串所在的内存页地址，加上偏移量后获得字符串的地址，最后通过 `bl` 调用 `puts` 函数，恢复寄存器 `x30` 后返回。

```
    .text
    .global main
main:
    str     x30, [sp, -16]!
    adrp    x0, .LC0
    add     x0, x0, :lo12:.LC0
    bl      puts
    ldr     x30, [sp], 16
    ret
.LC0:
    .string "Hello World!"
```

代码 11.1: ARM-v8A 指令：Hello World 程序

11.1.1 寻址模式

在 ARMv8-A 架构中，根据程序的变量作用域和存储位置，可以将寻址需求划分为以下两大类：全局变量的寻址和局部变量的寻址。全局变量通常位于静态存储区（如数据段或只读段），它们的地址在程序运行时通常是固定的。一般需要先通过 `adrp` 获取其内存页的地址（4KB 对齐），然后加上其低 12 位地址。

局部变量通常分配在栈上，包括以下寻址方式：

- **立即偏移寻址模式**：使用基地址寄存器和一个立即数偏移值计算目标地址，如 `ldr x2, [x1]` 表示加载内存地址 `x1` 中的数据到 `x2` 中，`ldr x2, [x1, #10]` 表示加载地址 `x1+10` 中的数据到 x2 中。
- **寄存器偏移寻址模式**：偏移量由另一个寄存器的值提供，如 `ldr x2, [x1, x0]` 表示加载地址 `x1+x0` 中的数据到 x2 中。另外，也可以结合移位操作实现更复杂的偏移，如 `ldr x2, [x1, x0, lsl #3]` 表示加载地址 `x1+x0*8` 的数值到 `x2` 中。

- **预索引寻址模式**：访存前先调整基地址，如 `ldr x2, [x1, #10]!` 表示先更新 `x1` 的值为 `x1+10`，再将加载地址 `x1` 中的数据到 `x2` 中。
- **后索引寻址模式**：访存前再调整基地址，如 `ldr x2, [x1], #10` 先加载地址 `x1` 中的数据到 `x2` 中，再将 `x1` 更新为 `x1+10`。
- **PC 相对寻址模式**：地址是当前程序计数器的值加上一个偏移量，如 `ldr x2, label`。
- **栈寻址模式**：通过栈指针访问数据：如 `ldr x2, [sp, #8]` 加载地址 `sp+8` 中的数据到 `x1`。

11.1.2 立即数支持

由于 ARMv8-A 指令是 32 位定长编码的，对于参数是立即数的情况支持比较有限。大多数算术数据处理指令支持 12 位无符号整数（范围为 0-4095）。如果立即数过大，可通过移位机制扩展表示。例如，加载立即数 65539 可以通过以下两条指令实现。其中 `movk` 指令表示保持寄存器低位不变。另外，如果一个整数的低 12 为 0，且大小不超过 2^{24}，一般也可以直接使用。

```
mov x8, 3 ; 将\texttt{x8}设置为3
movk x8, 1, lsl 16 ; 将\texttt{x8}寄存器16-31位设置为1
```

代码 11.2: ARM-v8A 指令：立即数示例

逻辑运算的立即数支持与算数运算指令不同，它采用了一种特殊的编码机制，允许表示复杂的位模式，如连续的 1 位掩码（`0xFF`）和周期性掩码（如 `0xAAAAAAAA`）。

11.1.3 主要指令

下面以翻译单条 LLVM IR 指令为目标讲解主要的 ARM-v8A 指令，见表 11.1 。实际 ARMv8-A 手册中的指令有几百条。如需了解更多的指令和机制，可参考 ARM 公司提供的官方手册 [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`	为局部变量分配栈内存；sp 指针要求 16 字节对齐
`store i32 %0, i32* %a`	`str w0, [sp, 12]`	将寄存器值保存到内存地址 sp+12
`store i32 1, i32* %a`	`mov w0, 1` `str w0, [sp, 12]`	将整数保存到栈空间；str 操作数不能为立即数
`%a0 = load i32, i32* %a`	`ldr w0, [sp, 12]`	将局部变量值由内存地址 sp+12 加载到寄存器
`%g0 = load i32, i32* @g`	`adrp x8, g` `ldr w0, [x8, :lo12:g]`	将全局变量值加载到寄存器
`%r = add i32 %a, %b`	`add w0, w1, w2`	两个寄存器的值相加，结果保存到 w1
`%r = add i32 %a, 4095`	`add w0, w1, 4095`	整数范围：$x \in [0, 2^{12})$，以及 $x * 2^{12}$
`%r = sub i32 %a, %b`	`sub w0, w1, w2`	两个寄存器的值相减；亦支持减立即数，方法同加法
`%r = mul i32 %a, %b`	`mul w0, w1, w2`	不支持立即数
`%r = sdiv i32 %a, %b`	`sdiv w0, w1, w2`	不支持立即数
`%r = icmp sgt i32 %a, %b` `br i1 %r, label %bb1, label %bb2`	`cmp w0, w1` `b.le .LBB2`	比较：支持一个立即数，结果存到 CPSR 寄存器；然后条件跳转
`%r = xor i32 %a, %b`	`eor w0, w1, w1`	异或运算，支持一个立即数
`%r = and i32 %a, %b`	`and w0, w1, w1`	与运算，支持一个立即数
`%r = or i32 %a, %b`	`orr w0, w1, w1`	或运算，支持一个立即数
`call void @foo()`	`bl foo`	函数调用
`ret i32 %r`	`ldr x30, [sp], 16` `ret`	将返回地址存入 x30 寄存器，还原栈顶指针，返回

11.2 消除 phi 指令

LLVM IR 中由于转换 SSA 会引入 phi 指令，但不存在直接与 phi 匹配的 ARM 指令。以代码 11.3 为例，我们可以通过两种方式消除 phi 指令：

```
bb1:
    %r1 = icmp eq i32 %a1, 0
    ; 方式一：store i32 %a1, i32* %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, i32* %a
    ; 方式二：%a3 = %a2
    br label %bb2
bb3:
    %a3 = phi i1 [%a1, %bb1], [%a2, %bb2]
    ; 方式一：%a3 = load i32, i32* %a
    %r1 = add i32 %a3, %b1
```

代码 11.3: LLVM IR 代码：消除 phi 指令的例子

- **使用 store-load 替换 phi**：在 phi 指令的前驱代码块跳转指令前增加 store 指令，并将 phi 指令替换为 load 指令。以代码 11.3 为例，我们分别在代码块%bb1 和%bb2 中增加对变量 a 的 store 语句，并将 phi 指令替换为对变量 a 的 load 语句。这种方法的缺点是会引入非必要的仿存操作。
- **使用伪赋值指令替换 phi**：在 phi 指令的前驱代码块直接对 phi 指令的目标寄存器进行进行赋值。以代码 11.3 为例，在代码块%bb1 和%bb2 中增加对虚拟寄存器%a3 赋值的伪指令。由于 LLVM IR 并不支持这种直接赋值的指令形式，因此这种修改后的代码无法使用标准的 lli 执行。这种方法的优点是可以避免不必要的仿存操作，最大化寄存器使用。

11.3 指令选择问题

通过前面的介绍，我们可以轻松地为单条 IR 指令找到对应的汇编指令翻译方式。然而，仅考虑单条 IR 指令直接翻译为汇编代码的方式通常并非最优，因为许多汇编指令能够覆盖多条 IR 指令的功能。例如，一些复合算术运算指令（如乘法累加或减法）可以同时对应 IR 中的两条指令：乘法和加法（或减法）。本节将重点讨论指令选择问题，特别是如何为这些情况生成更加高效的汇编代码。

```
madd x0, x1, x2, x3 ; x0 = x1 * x2 + x3; mul指令本质上是该指令在x3=0时的特例
msub x0, x1, x2, x3 ; x0 = x1 * x2 - x3;
```

代码 11.4: ARM-v8A 指令：复合算数运算

下面我们将重点讨论单个代码块的指令选择问题。由于一个函数可以划分为多个代码块，因此可以独立处理各代码块的指令选择问题，从而简化指令翻译问题的复杂度。

11.3.1 指令选择图

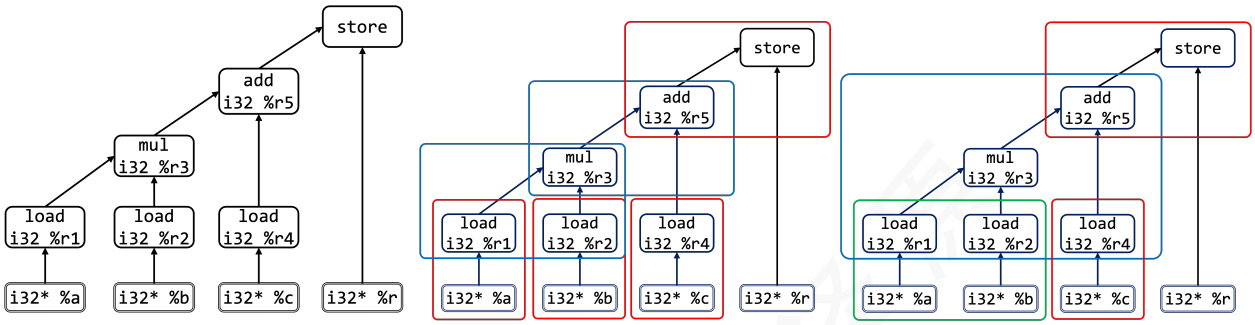
我们将该单个代码块内的 IR 表示为指令选择图，从而对该问题进行建模。

定义 6 (指令选择图)**.** 指令选择图是一个有向无环图，包括两种类型的节点：指令节点和数据存储节点；其中的边表示指令运行所需的参数。

以代码 11.5 为例，其指令选择图可表示为图 11.1a ，指令执行顺序只需满足拓扑排序即可。

```
%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 11.5: 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

```
%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, i32* %aj
store i32 %aj1, i32* %ai
%t1 = load i32, i32* %t
store i32 %t1, i32* %aj
%ai1 = load i32, i32* %ai
store i32 %ai1, i32* %t
```

代码 11.6: LLVM IR 代码：内存同步问题

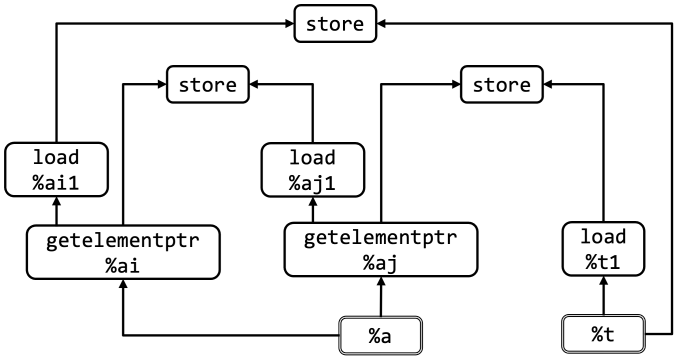
值得注意的是，当代码块中出现 store 等涉及内存同步的指令时，需要对代码块进行分割，为指令前后的 IR 分别绘制指令选择子图，从而保证汇编代码和 IR 语义的一致性。以代码11.6 为例，图 11.2a 为直接绘制的指令选择图，其中存在多种满足拓扑排序的 `store-load` 和 `store-store` 顺序，且语义不相同。以 `store` 指令为边界进行分割后可以构造三个存在顺序关系的指令选择子图：图11.2b 、11.2c 和11.2d 。

11.3.2 铺树问题

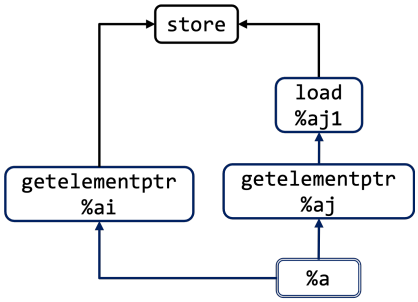
通过指令选择图，我们可以将指令选择问题转化成铺树（图）问题，即如何翻译汇编指令，使其可以覆盖指令选择图上的所有节点，同时使得目标汇编代码指令数最少、运行时间最优。以图 11.1a 为例，该图至少包含图 11.1b 和图 11.1c 中的两种铺树方案。其对应的汇编代码分别为代码 11.7 和代码 11.8 。我们很容易看出代码 11.8 对应的指令数更少。如果 `ldr` 和 `ldp` 指令的性能开销以及 `mul` 和 `madd` 指令的性能开销都相同，则明显图 11.1c 对应的铺树方案更优。

铺树问题是一个 NP-hard 问题，可以通过贪心法或动态规划求解。一种常用的贪心法称为 Maximal Munch，即每步选择覆盖节点最多的方案进行铺树。

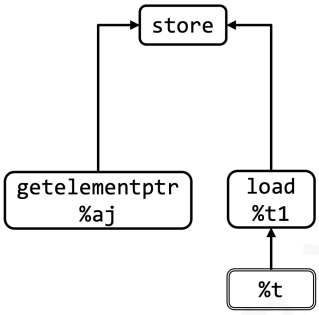
```
ldr w1, [sp, .a]
ldr w2, [sp, .b]
```



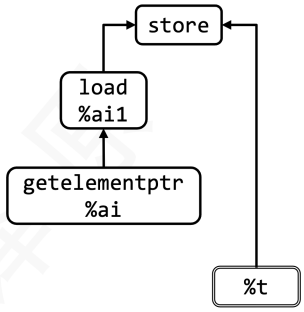
(a) 直接为代码 11.6 绘制指令选择图



(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

```
ldr w3, [sp, .c]
mul w3, w1, w2
add w5, w3, w4str w5, [sp, .r]
```

代码 11.7: 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 11.8: 图 11.1c 对应的汇编代码

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 %bb4
bb5:
    %x6 = phi i32 [ %x3, %bb3 ], [ %x5, %bb4 ]
    br label %bb1
bb6:
    ret i32 %x1
}
```

代码 11.9: LLVM IR 代码

参考文献

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

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

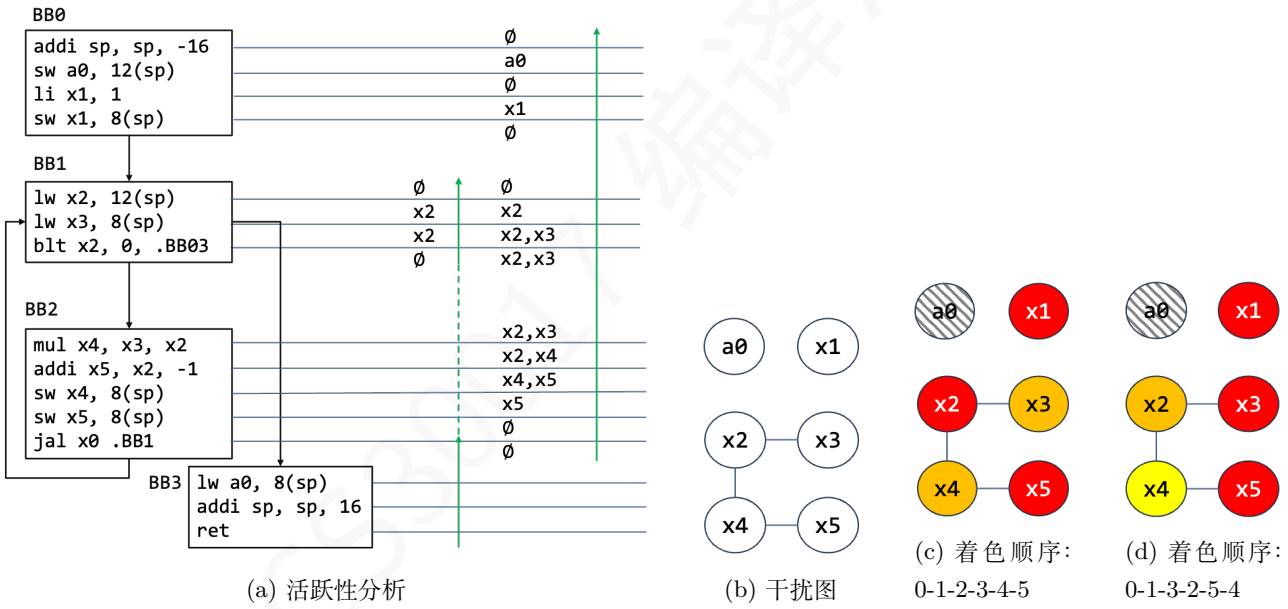
指令翻译时无需考虑虚拟寄存器数量限制，然而实际的物理寄存器数量是有限的。例如，AArch64 架构中一般使用 `x9-x15` 寄存器或 `x19-x28` 寄存器；X86 架构可用的寄存器则更少。因此，如何将虚拟寄存器翻译为物理寄存器非常关键。

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 13 。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 13 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 14 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。

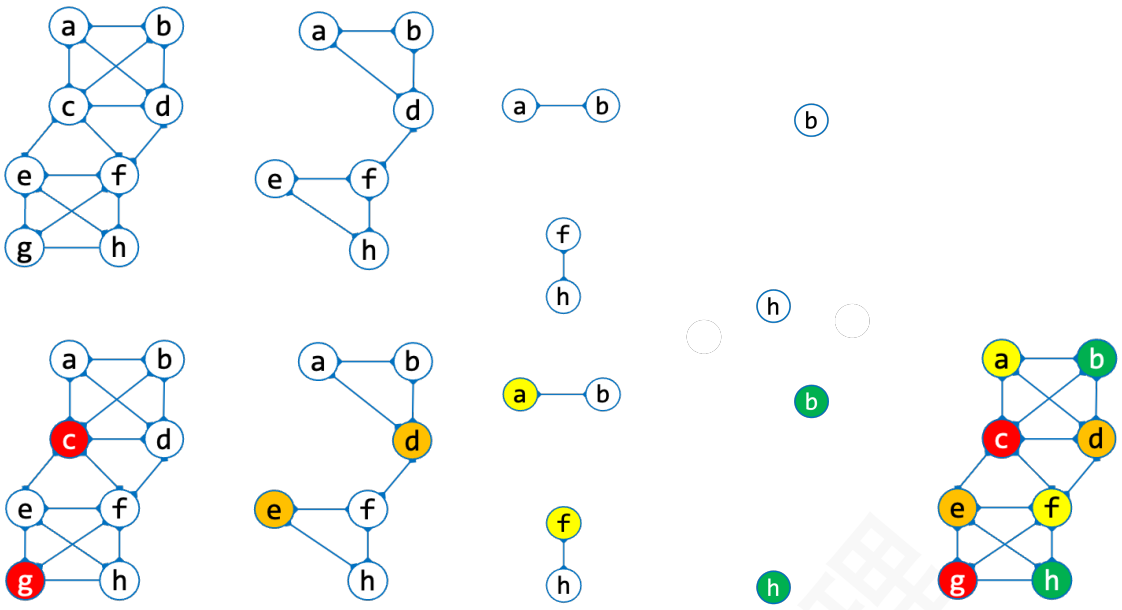
算法 14 Recursive largest first 算法

```
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 15 。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 15 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 。

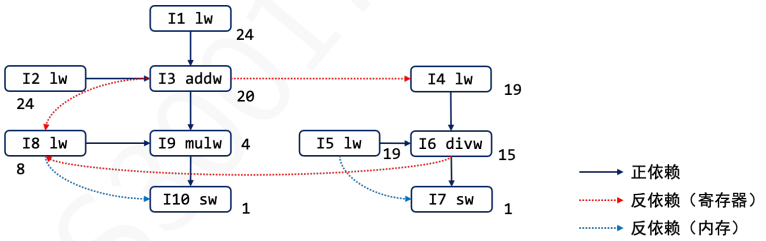


图 13.1: 代码 13.2 中指令间的依赖关系和时延

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`。

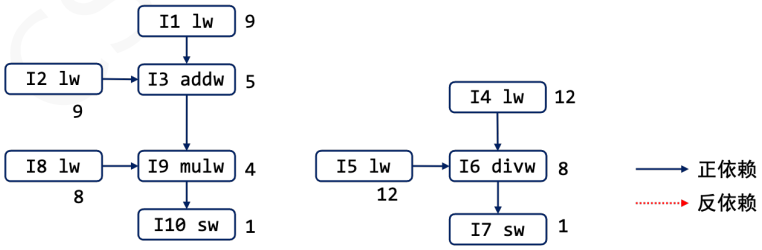


图 13.2: 代码 13.4 中指令间的依赖关系和时延

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`