编译器与语言设计导论

第二版

圣母大学道格拉斯·塞恩
教授

编译器与语言设计导论

ISBN: 979-8-655-18026-0 第二版。

您可以在 http://compilerbook.org 找到最新版本的PDF版，并购买价格实惠的精装书。

Revision Date: August 24, 2023

For Lisa, William, Zachary, Emily, and Alia.

贡献

我感激以下人士对本书的贡献：

安德鲁·利特肯起草了关于ARM汇编的章节；凯文·拉蒂默绘制了从正则表达式到NFA的转换图和LR示例图；本杰明·冈宁修正了LL(1)解析表构造中的一个错误；蒂姆·沙弗完成了详细的LR(1)示例。

以下人员更正了拼写错误：

萨基布·哈克（27），约翰·韦斯托夫（26），艾米丽·斯特劳特（26），冈萨洛·马丁内斯（25），丹尼尔·凯里根（24），布赖恩·杜塞尔（23），莉维亚·约翰（22），瑞安·麦基（20），TJ·达索（18），内迪姆·米尼诺维奇（15），诺亚·吉田（14），约瑟夫·金林格（12），诺兰·麦克谢（11），郑寿·李（11），凯尔·温加特纳（10），安东尼·施密特（10），安德鲁·利特肯（9），托马斯·凯恩（9），塞缪尔·巴塔利奥（9），斯特凡·马索（8），路易斯·普里布（7），威廉·迪德里希（7），乔纳森·徐（6），加文·英格利斯（6），凯瑟琳·卡佩拉（6），爱德华·阿特金森（6），克里斯汀·弗雷迪（5），坦纳·朱德曼（5），邓善丰（4），约翰·约翰逊（4），卢克·西埃拉（4），弗朗西斯·希克尔（4），艾蒙·马尔米恩（3），莫莉·扎克林（3），大卫·蒋（3），雅各布·马祖尔（3），斯宾塞·金（2），姚贤屈（2），玛丽亚·阿兰古伦（2），帕特里克·拉赫（2），康纳·希金斯（2），唐戈·顾（2），安德鲁·西尔梅基斯（2），霍尔斯特·冯·布兰德（2），约翰·福克斯（2），杰米·张（2），约翰·沙利文（2），本杰明·甘宁（1），查尔斯·奥斯本（1），威廉·泰森（1），杰西卡·乔菲（1），本·托瓦尔（1），瑞安·米哈莱克（1），帕特里克·弗林（1），克林特·杰弗里（1），拉尔夫·西姆森（1），约翰·奎因（1），保罗·布伦茨（1），卢克·沃尔（1），布鲁斯·马德尔（1），戴恩·威廉姆斯（1），托马斯·费舍尔（1），阿兰·约翰逊（1），雅各布·哈里斯（1），杰夫·克林顿（1），雷托·哈布鲁泽尔（1），克里斯·费特基维茨（1），米格尔·帕赫（1）。

请通过电子邮件将任何意见或更正发送给Douglas Thain教授（dthain@nd.edu）。

内容

插图目录

第1章 – 引言

1.1 什么是编译器?

编译器将一种源语言中的程序翻译成一种目标语言中的程序。最广为人知的编译器形式是将像 C 这样的高级语言翻译成机器的本地汇编语言，以便能够执行。当然，还有用于其他语言的编译器，如 C++、Java、C# 和 Rust 等等。

传统编译器中使用的相同技术，也被用于任何处理语言的程序中。例如，像 TEX 这样的排版程序会将稿件转换为 PostScript 文档。像 Dot 这样的图布局程序读取节点和边的列表，并将它们在屏幕上进行布局。Web 浏览器会把 HTML 文档转换成交互式的图形显示。要编写这类程序，你需要理解并使用与传统编译器相同的技术。

编译器的存在不仅是为了*translate*程序，也是为了*improve*它们。编译器通过在编译时发现程序中的错误来协助程序员，这样用户就不必在运行时遇到这些错误。通常，更严格的语言会产生更多的编译时错误。这使程序员的工作更困难，但也更有可能保证程序是正确的。例如，Ada 语言在程序员中以难以在没有编译时错误的情况下编写而闻名，但一旦能够正常工作，就被信任用于运行诸如波音 777 飞机等安全关键系统。

编译器不同于解释器，解释器读取程序并直接执行，而不生成翻译结果。这有时也被称为虚拟机。像 Python 和 Ruby 这样的语言通常由解释器直接读取源代码来执行。

编译器和解释器密切相关，有时可以相互替代。例如，Java 编译器将 Java 源代码翻译成 Java 字节码，这是一种抽象形式的汇编语言。一些 Java 虚拟机的实现以解释器的方式工作，一次执行一条指令。另一些则通过将字节码翻译成本地机器代码，然后直接运行这些机器代码。这被称为即时编译（Just-In-Time compiling，简称 JIT）。

1.2 为什么要学习编译器?

You will be a better programmer. 一个优秀的工匠必须了解自己的工具，程序员也是如此。通过更深入地理解编译器如何将你的程序翻译成机器语言，你将更擅长编写高效的代码，并在出现问题时进行调试。

You can create tools for debugging and translating. 如果你能为某种语言编写解析器，那么你就能编写各种各样的辅助工具，帮助你（以及他人）调试你自己的程序。像 Eclipse 这样的集成开发环境内置了对 Java 等语言的解析器，因此它可以进行语法高亮、在不编译的情况下发现错误，并在你编写代码时将代码与文档关联起来。

You can create new languages. 一个令人惊讶的事实是，通过将问题以紧凑的方式表达在定制语言中，许多问题变得更容易解决。（这些有时被称为领域特定语言，或简称为小语言。）通过学习编译器技术，您将能够实现小语言并避免一些语言设计中的陷阱。

You can contribute to existing compilers. 尽管你不太可能编写下一个伟大的C编译器（因为我们已经有了几个），但语言和编译器的发展并没有停滞不前。标准的开发带来了新的语言特性；优化研究创造了改进程序的新方法；新的微处理器被创造出来；新的操作系统被开发；等等。所有这些发展都需要对现有编译器的持续改进。

You will have fun while solving challenging problems. 这还不够吗?

1.3 学习编译器的最佳方法是什么?

学习编译器的最佳方式是从头到尾*write your own compiler*。虽然一开始听起来可能令人畏惧，但你会发现这个复杂的任务可以分解成几个中等复杂度的阶段。典型的本科计算机科学学生可以在一个学期内为一个简单语言编写一个完整的编译器，这个过程分为四到五个独立的阶段。

1.4 我应该使用哪种语言?

毫无疑问，您应该使用C编程语言和X86汇编语言，当然!

好的，也许答案并没有那么简单。编程语言的数量不断增加，每种语言都有不同的优缺点。Java 简单、一致且可移植，尽管性能不高。Python 易于学习，并且有很好的库支持，但类型较弱。Rust 提供了卓越的静态类型安全，但目前还不是（完全）

被广泛使用。几乎可以用任何语言编写一个编译器，而你可以把这本书作为指南来完成这一点。

然而，我们确实认为你应该学习 C，用 C 编写一个编译器，并用它来编译一种类 C 的语言，该语言为广泛使用的处理器（如 x86 或 ARM）生成汇编代码。为什么？因为学习那些被广泛使用的技术的来龙去脉对你很重要，而不仅仅是那些在抽象层面上很美的东西。

C 是用于低级编程（编译器、库和内核）最广泛使用的可移植语言，而且它也足够小，以至于可以在一个学期内学习如何编译 C 的各个方面。诚然，C 在类型安全和指针使用方面存在一些挑战，但对于编译器这样规模的项目而言，这些都是可控的。还有其他具有不同优点的语言，但没有一种像 C 这样既简单又被如此广泛地使用。一旦你编写了一个 C 编译器，你就可以自由地设计你自己的（更好的）语言。

同样，X86 已经成为桌面、服务器和笔记本电脑中最广泛部署的计算机架构，已有几十年历史。虽然它比 MIPS、SPARC 或 ARM 等其他架构复杂得多，但人们可以快速学习构建编译器所需的基本指令子集。当然，ARM 正迅速在移动、嵌入式和低功耗领域赶超，因此我们也包括了关于 ARM 的一节。

话虽如此，本书中提出的原则具有广泛的适用性。如果你将其作为课程的一部分使用，你的授课教师很可能会选择不同的编译语言和不同的目标汇编语言，这也完全没问题。

1.5 本书与其他书籍有何不同?

大多数编译器书籍在扫描器、解析器、类型系统和寄存器分配的抽象理论上着墨甚多，而对语言设计如何影响编译器和运行时的讨论相对较少。它们中的大多数是为研究生层面的优化技术综述课程而设计的。

本书采取更为广泛的视角，减少了对优化的强调，并引入了更多关于编译器工程过程、语言设计中的权衡，以及对解释与翻译的考量等内容。

你还会注意到，这本书并没有包含大量繁琐的纸笔作业来测试你对编译器算法的理解。（好吧，在第 3 章和第 4 章中确实有少数这样的作业。）如果你想测试自己的掌握程度，那就去编写一些能够正常运行的代码。为此，每一章末尾的练习都会要求你将本章中的思想付诸实践，要么去探索一些现有的编译器，要么编写你自己编译器的部分组件。如果你按顺序完成所有练习，最终你将得到一个可以工作的编译器，其整体情况在最后的附录中进行了总结。

1.6 我还应该读哪些书?

对于编译器的一般参考，我推荐以下书籍:

- Charles N. Fischer，Ron K. Cytron，Richard J. LeBlanc Jr，《编译器构造》，皮尔逊，2009年。
  This is an excellent undergraduate textbook which focuses on object-oriented software engineering techniques for constructing a compiler, with a focus on generating output for the Java Virtual Machine.

- Christopher Fraser 和 David Hanson，“可重定向的 C 编译器：设计与实现”，Benjamin/Cummings, 1995.
  Also known as the "LCC book", this book focuses entirely on explaining the C implementation of a C compiler by taking the unusual approach of embedding the literal code into the textbook, so that code and explanation are intertwined.

- 阿尔弗雷德·V·阿霍、莫妮卡·S·拉姆、拉维·塞西和杰弗里·D·乌尔曼，《编译器：原理、技术与工具》，Addison Wesley，2006年。
  Affectionately known as the "dragon book", this is a comprehensive treatment of the theory of compilers from scanning through type theory and optimization at an advanced graduate level.

好吧，你还在等什么？让我们开始工作吧。

第2章——快速浏览

2.1 编译器工具链

编译器是用于从源代码创建可执行文件的一套工具链中的一个组件。通常，当你调用一个命令来编译程序时，后台会依次调用一整套程序。图 2.1 展示了在 Unix 系统中将 C 源代码编译为汇编代码时通常使用的程序。

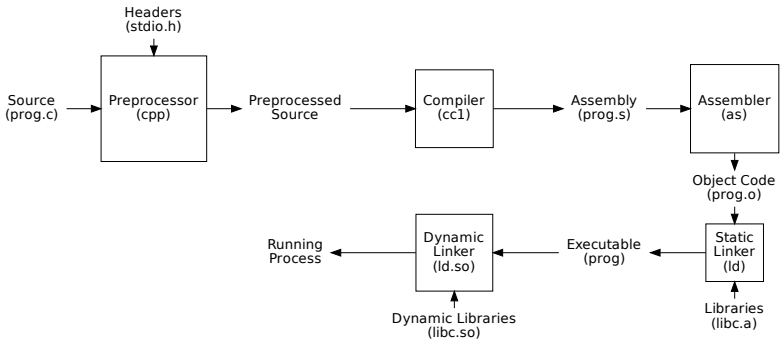


图 2.1：典型的编译器工具链

- 预处理器为真正的编译器准备源代码。在 C 和 C++ 语言中，这意味着处理所有以 # 符号开头的指令。例如，#include 指令会使预处理器打开指定的文件并将其内容插入到源代码中。#define 指令会使预处理器在遇到宏名的任何地方用相应的值进行替换。（并非所有语言都依赖预处理器。）

- 编译器本身接收预处理器的干净输出。它对源代码进行扫描和解析，执行类型检查并

其他语义例程，对代码进行优化，然后生成汇编语言作为输出。工具链的这一部分是本书的主要关注点。

- 汇编器读取汇编代码并生成目标代码。目标代码“几乎是可执行的”，因为它包含以 CPU 所需形式表示的原始机器语言指令。然而，目标代码并不知道其将被加载到的最终内存地址，因此其中包含必须由链接器填充的空缺。

- 链接器接收一个或多个目标文件和库文件，并将它们组合成一个完整的可执行程序。它选择每一段代码和数据将被加载到的最终内存位置，然后通过写入缺失的地址信息将它们“链接”在一起。例如，一个调用 printf 函数的目标文件在最初并不知道该函数的地址。在必须使用该地址的地方会留下一个空的（零）地址。一旦链接器选择了 printf 的内存位置，就必须回过头来，在每一个调用 printf 的地方写入该地址。

在类Unix操作系统中，预处理器、编译器、汇编器和链接器分别历史上被称为cpp、cc1、as和ld。用户可见的程序cc只是依次调用工具链的每个元素，以生成最终的可执行文件。

2.2 编译器内部的阶段

在本书中，我们将主要关注编译器本身，它是工具链中最有趣的组成部分。编译器本身可以划分为若干个阶段：

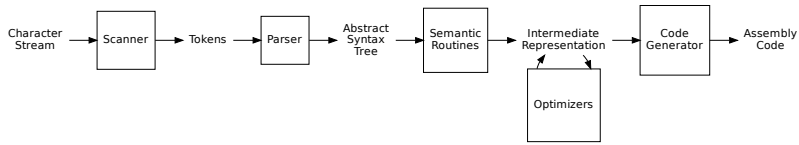


图 2.2：Unix 编译器的各个阶段

- 扫描器接收程序的纯文本，并将单个字符组合在一起形成完整的记号。这很像在自然语言中将字符组合成单词。

- 解析器会消耗标记（token），并将它们组合成完整的语句和表达式，类似于在自然语言中将单词组合成句子。解析器由语法规则引导，这些规则规定了某种语言中形式化的组合方式。解析器的输出是一棵抽象语法树（AST），用于捕捉程序的语法结构。AST 还会记住源文件中每个构造出现的位置，因此在需要时能够生成有针对性的错误信息。

- 语义例程遍历 AST，并根据语言规则以及程序元素之间的关系推导出关于程序的附加含义（语义）。例如，我们可能通过从先前的声明中观察 x 的类型，进而应用“int 与 float 值之间的加法会产生 float”的语言规则，来确定 x + 10 是一个浮点表达式。在语义例程之后，AST 通常会被转换为一种中间表示（IR），这是一种适合进行详细分析的简化汇编代码形式。IR 有多种形式，我们将在第 8 章中讨论。

- 可以将一个或多个优化器应用于中间表示，以使程序更小、更快或更高效。通常，每个优化器读取 IR 格式的程序，然后输出相同的 IR 格式，因此每个优化器都可以独立地、以任意顺序应用。

- 最后，代码生成器接收优化后的 IR，并将其转换为具体的汇编语言程序。通常，代码生成器必须进行寄存器分配，以有效管理数量有限的硬件寄存器，并进行指令选择与排序，以将汇编指令安排成最高效的形式。

2.3 示例编译

假设我们希望将这段代码片段编译成汇编：

```
height = (width+56) * factor(foo);
```

编译器的第一阶段（扫描器）将逐字符读取源代码文本，识别符号之间的边界，并生成一系列记号（token）。每个记号都是一个小型的数据结构，用于描述每个符号的性质和内容：

id:height	**=**	**(**	**id:width**	**+**	**int:56**	**)**	*****	**id:factor**	**(**	**id:foo**	**)**	**;**

在这个阶段，每个标记的用途尚不清楚。例如，factor 和 foo 仅仅被认为是标识符，尽管其中一个是

一个是函数的名称，另一个是变量的名称。同样地，我们尚不知道 width 的类型，因此 + 可能表示整数加法、浮点加法、字符串连接，或者完全是别的东西。

下一步是确定这一记号序列是否构成一个有效的程序。解析器通过寻找与语言语法相匹配的模式来完成这一点。假设我们的编译器理解一种具有如下语法的语言：

Grammar G_1

1. expr → expr + expr
2. expr → expr * expr
3. expr → expr = expr
4. expr → id (expr)
5. expr → (expr)
6. expr → id
7. expr → int

语法中的每一行称为一条规则，用于说明语言的各个部分是如何构造的。规则1–3表明，表达式可以通过用运算符连接两个表达式来形成。规则4描述了函数调用。规则5描述了括号的使用。最后，规则6和7表明标识符和整数是原子表达式。[1]

解析器寻找可以由语法规则左侧替换的标记序列。每次应用规则时，解析器会在树中创建一个节点，并将子表达式连接到抽象语法树（AST）。AST 显示了各个符号之间的结构关系：宽度和 56 进行加法运算，而函数调用应用于因子和 foo。

有了这一数据结构，我们现在已经准备好分析程序的含义。语义例程遍历抽象语法树（AST），通过将程序的各个部分彼此关联，并与编程语言的定义相联系，来推导出额外的语义。该过程中的一个重要组成部分是类型检查，在其中确定每个表达式的类型，并检查其是否与程序的其余部分保持一致。为简化起见，这里我们将假设所有变量都是普通整数。

为了生成线性中间代码，我们对 AST 进行后序遍历，并为树中的每个节点生成一条 IR 指令。典型的 IR 看起来像一种抽象的汇编语言，包含加载/存储指令、算术运算，以及无限数量的寄存器。例如，下面是我们示例程序的一种可能的 IR 表示：

[1] The careful reader will note that this example grammar has ambiguities. We will discuss that in some detail in Chapter 4.

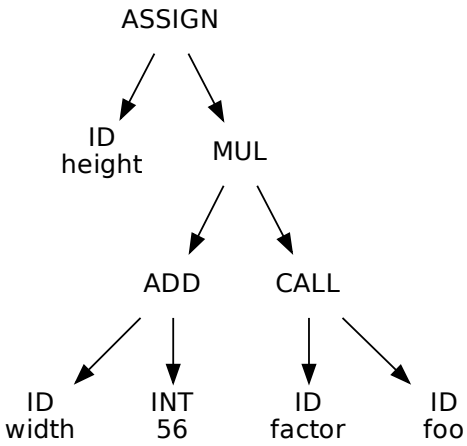


图 2.3：示例 AST

```
LOAD $56    -> r1
LOAD width  -> r2
IADD r1, r2 -> r3
ARG  foo
CALL factor -> r4
IMUL r3, r4 -> r5
STOR r5     -> height
```

图 2.4：示例中间表示

中间表示是大多数优化发生的地方。死代码被移除，常见操作被合并，代码通常会被简化，以消耗更少的资源并更快地运行。

最后，中间代码必须转换为所需的汇编代码。图 2.5 显示了 X86 汇编代码，这是上述 IR 的一种可能翻译。请注意，汇编指令不一定与 IR 指令一一对应。

一个设计良好的编译器具有高度模块化，以便可以根据需要共享和组合常见的代码元素。为了支持多种语言，编译器可以提供不同的扫描器和解析器，每个扫描器和解析器都生成相同的中间表示。不同的优化技术可以作为独立模块实现（每个模块读取和写入相同的IR），这样它们可以独立启用和禁用。

```
MOVQ    width, %rax     # load width into rax
ADDQ    $56, %rax       # add 56 to rax
MOVQ    %rax, -8(%rbp)  # save sum in temporary
MOVQ    foo, %edi       # load foo into arg 0 register
CALL    factor          # invoke factor, result in rax
MOVQ    -8(%rbp), %rbx  # load sum into rbx
IMULQ   %rbx            # multiply rbx by rax
MOVQ    %rax, height    # store result into height
```

图 2.5：示例汇编代码

dently。可重定向编译器包含多个代码生成器，因此同一 IR 可以针对多种微处理器生成。

2.4 练习

1. 确定如何在你的本地计算环境中手动调用预处理器、编译器、汇编器和链接器。编译一个计算简单表达式的小型完整程序，并在每个阶段检查输出。你是否能够在每种形式下跟踪程序的流程?

2. 确定如何更改本地编译器的优化级别。找一个非平凡的源程序，并在多个优化级别下对其进行编译。随着优化级别的变化，编译时间、程序大小以及运行时间如何变化?

3. 在互联网上搜索你熟悉的三种语言的形式语法，例如 C++、Ruby 和 Rust。并将它们并排比较。哪种语言天生更复杂？它们是否共享任何共同的结构?

第3章 – 扫描

3.1 词元的种类

扫描是从程序的原始文本源代码中识别符号的过程。乍一看，扫描可能看起来微不足道——毕竟，在自然语言中识别单词就像是在字母之间寻找空格一样简单。然而，在源代码中识别符号要求语言设计者澄清许多细节，以便明确什么是允许的，什么是不允许的。

大多数语言将在以下类别中有标记:

- 关键词是语言结构本身中的词语，比如 while、class 或 true。关键词必须小心选择，以反映语言的自然结构，同时不干扰变量和其他标识符的可能名称。

- 标识符是程序员选择的变量、函数、类和其他代码元素的名称。通常，标识符是字母和可能的数字的任意序列。某些语言要求标识符使用符号（如 Perl 中的美元符号）来明确区分标识符和关键字。

- 数字可以格式化为整数、浮点数、分数，或者使用其他进制，如二进制、八进制或十六进制。每种格式应当清楚区分，以便程序员不会将它们混淆。

- 字符串是必须与关键字或标识符清晰区分的文字字符序列。字符串通常用单引号或双引号括起来，但也必须具备包含引号、换行符和不可打印字符的功能。

- 注释和空白用于格式化程序，使其在视觉上清晰，并且在某些情况下（如 Python）对程序的结构具有重要意义。

在设计一种新语言，或为一种现有语言设计编译器时，首要工作是精确定义在每一类记号中允许使用哪些字符。最初，这可以通过非正式的方式来完成，即通过陈述，

token_t 扫描标记(FILE *fp) { int c = fgetc(fp); 如果(c==’*’) { 返回 TOKEN_MULTIPLY; } 否则如果(c==’!’) { char d = fgetc(fp); 如果(d==’=’) { 返回 TOKEN_NOT_EQUAL; } 否则 { ungetc(d,fp); 返回 TOKEN_NOT; } } 否则如果(isalpha(c)) { 执行 {char d = fgetc(fp); } 当(isalnum(d)); ungetc(d,fp); 返回 TOKEN_IDENTIFIER; } 否则如果 (. . .) { . . . } }

图3.1：一个简单的手工制作扫描器

例如，*“An identifier consists of a letter followed by any number of letters and numerals.”*，然后为这种记号分配一个符号常量（TOKEN IDENTIFIER）。正如我们将看到的，非正式的方法往往是含糊不清的，因此需要一种更为严谨的方法。

3.2 手工制作的扫描器

图3.1展示了如何使用简单的编码技术手工编写一个扫描器。为保持简单，我们只考虑少数几种记号：* 表示乘法，! 表示逻辑非，!= 表示不等于，以及由字母和数字组成的序列作为标识符。

基本方法是从输入流（fgetc(fp)）中一次读取一个字符，然后对其进行分类。一些单字符记号很容易：如果扫描器读到一个 * 字符，它会立即返回 TOKEN MULTIPLY，而加法、减法等等也是如此。

然而，有些字符属于多个标记。如果扫描器遇到 !，它可能单独表示一个逻辑非运算，或者它可能是表示不等于的 != 序列中的第一个字符。

在读取到 ! 时，扫描器必须立即读取下一个字符。如果下一个字符是 =，那么它已匹配序列 !=，并返回 TOKEN NOT EQUAL。

但是，如果紧随 ! 的字符是其他字符，那么不匹配的字符需要使用 ungetc 被 *put back* 回输入流，因为它不是当前记号的一部分。扫描器返回 TOKEN NOT，并将在下一次调用 scan token 时消耗被放回的字符。

以类似的方式，一旦通过 isalpha(c) 识别出一个字母，扫描器就会持续读取字母或数字，直到遇到不匹配的字符。该不匹配的字符会被放回，扫描器返回 TOKEN IDENTIFIER。

（我们会在编译器的每一个阶段看到这种模式：一个意外的项不符合当前的目标，因此必须被放回以备后用。这在更一般的意义上被称为回溯。）

如你所见，手工编写的扫描器相当冗长。随着添加的词法记号类型越来越多，代码可能会变得相当复杂，尤其是在不同记号共享相同字符序列的情况下。开发者也可能难以确信扫描器代码与每个记号的预期定义完全一致，这在处理复杂输入时可能导致意外行为。尽管如此，对于一个词法记号数量有限的小型语言来说，手工编写的扫描器仍然可能是一个合适的解决方案。

对于具有大量记号的复杂语言，我们需要一种更加形式化的方法来定义和扫描记号。形式化的方法能够让我们更有信心地确保记号定义之间不会发生冲突，并且扫描器的实现是正确的。此外，形式化的方法还能使扫描器更加紧凑且具有高性能——令人惊讶的是，扫描器本身可能成为编译器中的性能瓶颈，因为每一个字符都必须被单独处理。

正则表达式和有限自动机等形式化工具使我们能够非常精确地说明某一给定标记类型中可能出现的内容。随后，自动化工具可以处理这些定义，发现错误或歧义，并生成紧凑的高性能代码。

3.3 正则表达式

正则表达式（RE）是一种用于表达模式的语言。它们最早由 Stephen Kleene 于20世纪50年代提出[4]，作为他在自动机理论和可计算性方面奠基性工作的组成部分。如今，RE 以略有不同的形式出现在编程语言（Perl）、标准库（PCRE）、文本编辑器（vi）、命令行工具（grep）以及许多其他地方。我们可以将正则表达式作为一种紧凑而形式化的方式来指定编译器词法分析器所接受的记号，并将这些表达式自动翻译成可工作的代码。尽管容易解释，RE 的使用仍然有些棘手，需要一定的练习才能达到期望的效果。

让我们精确定义正则表达式:

正则表达式 s 是一个字符串，用于表示 $L(s)$，即从字母表 Σ 中取出的字符串集合。$L(s)$ 被称为“s 的语言”。

$L(s)$ 以如下基例进行归纳定义 s:

- 如果 a ∈ Σ，那么 a 是一个正则表达式，且 $L(a) = \{a\}$。
- ϵ 是一个正则表达式，并且 $L(\epsilon)$ 仅包含空字符串。

然后，对于任意正则表达式 s 和 t:

1. $s|t$ 是一个 RE，使得 $L(s|t) = L(s) \cup L(t)$。 2. st 是一个 RE，使得 $L(st)$ 包含所有由一个属于 $L(s)$ 的字符串后接一个属于 $L(t)$ 的字符串的连接所形成的字符串。 3. s^* 是一个 RE，使得 $L(s^*) = L(s)$ 连接零次或多次。

规则 #3 被称为 Kleene 闭包，并具有最高优先级。规则 #2 被称为连接。规则 #1 具有最低优先级，称为交替。可以通过添加括号来按通常方式调整操作顺序。

下面是一些仅使用基本规则的示例。（注意，有限的 RE 可以表示一个无限集合。）

正则表达式 s 的语言 L(s) hello { hello } d(o | i)g { dog,dig } moo* { mo,moo,mooo,... } (moo) * { ϵ,moo,moomoo,moomoomoo,... } a(b | a)*a { aa,aaa,aba,aaaa,aaba,abaa,... }

到目前为止所描述的语法已经完全足以编写任何正则表达式。但是，在基本语法之上再提供一些辅助操作也会很方便:

s? 表示 s 是可选的。s? 可以写作 (s | ϵ)。s+ 表示 s 重复一次或多次。s+ 可以写作 ss*。[a-z] 表示该范围内的任意字符。[a-z] 可以写作 (a | b | ... | z)。[x̂] 表示除某个字符之外的任意字符。[x̂] 可以写作 Σ - x

正则表达式也遵循几个代数性质，这使得可以根据需要重新排列它们以提高效率或清晰度：

Associativity:	`a|(b|c) = (a|b)|c`
Commutativity:	`a|b = b|a`
Distribution:	`a(b|c) = ab|ac`
Idempotency:	`a** = a*`

使用正则表达式，我们可以精确地说明在给定的词法单元中允许什么。假设我们有一种假想的编程语言，具有如下的非正式定义和正则表达式。对于每种词法单元类型，我们展示与该正则表达式匹配（以及不匹配）的字符串示例。

Informal definition:	*An identifier is a sequence of capital letters and numbers, but a number must not come first.*
Regular expression:	`[A-Z]+([A-Z]|[0-9])*`
Matches strings:	`PRINT` `MODE5`
Does not match:	`hello` `4YOU`

Informal definition:	*A number is a sequence of digits with an optional decimal point. For clarity, the decimal point must have digits on both left and right sides.*
Regular expression:	`[0-9]+(.[0-9]+)?`
Matches strings:	`123` `3.14`
Does not match:	`.15` `30.`

Informal definition:	*A comment is any text (except a right angle bracket) surrounded by angle brackets.*
Regular expression:	`<[^>]*>`
Matches strings:	`<tricky part>` `<<<<look left>`
Does not match:	`<this is an <illegal> comment>`

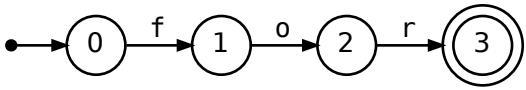
3.4 有限自动机

有限自动机（FA）是一种抽象机器，可以用来表示某些形式的计算。从图形上看，FA由多个状态（用编号的圆圈表示）和这些状态之间的多条边（用带标签的箭头表示）组成。每条边都带有一个或多个从字母表Σ中抽取的符号标签。

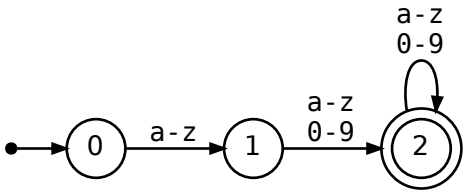
机器从起始状态 S_0 开始。对于FA呈现的每个输入符号，它会转移到由具有相同标签的边指示的状态。

作为输入符号。有限自动机（FA）的某些状态被称为接受状态，并通过双圈表示。如果有限自动机在所有输入都被处理后处于接受状态，则我们说该有限自动机接受输入。如果有限自动机以非接受状态结束，或者当前输入符号没有对应的边，则我们说该有限自动机拒绝输入字符串。

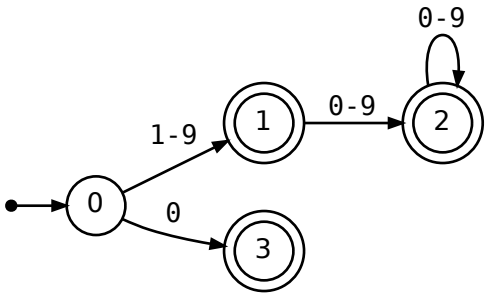
每个RE都可以写成FA，反之亦然。对于一个简单的正则表达式，可以手动构造一个FA。例如，这是关键字“for”的FA：



这里是一个用于标识符的有限自动机，形式为[a-z][a-z0-9]+



这里是一个用于形如 ([1-9][0-9]*) | 0 的数字的 FA



3.4.1 Deterministic Finite Automata

这三个示例中的每一个都是确定性有限自动机（DFA）。DFA 是有限自动机（FA）的一种特殊情况，其中每个状态在给定符号下至多只有一条出边。换言之，DFA 没有歧义：对于每一种状态与输入符号的组合，下一步的动作都恰好只有一个选择。

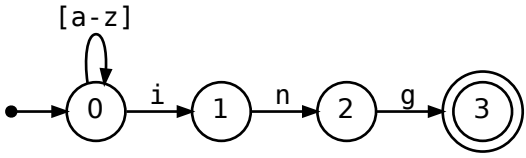
由于这一性质，DFA 在软件或硬件中都非常容易实现。只需要一个整数（c）来跟踪当前状态。

状态之间的转移由一个矩阵（$M[s,i]$）表示，该矩阵在给定当前状态和输入符号的情况下对下一个状态进行编码。（如果该转移不被允许，我们用 E 标记以表示错误。）对于每个符号，我们计算 $c = M[s,i]$，直到所有输入被消耗完，或到达错误状态。

3.4.2 Nondeterministic Finite Automata

DFA 的另一种选择是非确定性有限自动机（NFA）。NFA 是一种完全合法的有限自动机，但它具有一定的歧义性，使得使用起来稍微更困难。

考虑正则表达式[a-z]*ing，它表示所有以后缀 ing 结尾的小写单词。它可以用以下自动机来表示：

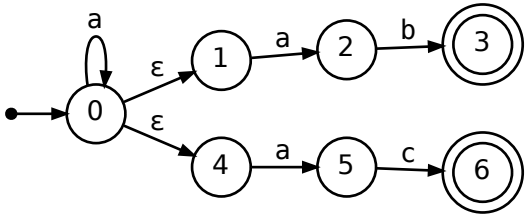


现在考虑这个自动机将如何处理单词 sing。它可以以两种不同的方式进行。一种方式是在读入 s 时转移到状态 0，在 i 时到状态 1，在 n 时到状态 2，在 g 时到状态 3。但另一种同样有效的方式是始终停留在状态 0，将每个字母都匹配到 [a-z] 的转移。两种方式都遵守转移规则，但一种导致接受，而另一种导致拒绝。

这里的问题在于，状态 0 在符号 i 上允许两种不同的转移。一种是匹配 [a-z] 并停留在状态 0，另一种是匹配 i 并转移到状态 1。

此外，并不存在一个简单的规则可以让我们选择一条路径而不是另一条。如果输入是 sing，正确的做法是在读到 i 时立即从状态零转移到状态一。但如果输入是 singing，那么我们应该在第一个 ing 时停留在状态零，在第二个 ing 时再转移到状态一。

一个 NFA 还可以包含一种 ϵ (epsilon) 转移，它表示空字符串。这种转移可以在完全不消耗任何输入符号的情况下被采用。例如，我们可以用这个 NFA 来表示正则表达式 a*(ab | ac)：



这个特定的NFA呈现出多种模糊的选择。从状态零，它可以消耗一个a并保持在状态零。或者，它可以通过ϵ到达状态一或状态四，然后无论哪种情况都消耗一个a。

对这种歧义通常有两种常见的解读方式：

- 水晶球解释表明，NFA 以某种方式“知道”最佳选择是什么，这种方式是NFA自身之外的某种手段。在上面的例子中，NFA 会在消耗第一个字符之前，选择是进入状态零、状态一还是状态四，并且总是做出正确的选择。不用说，这在实际实现中是不可能的。

- 多世界解释认为，NFA 存在于所有允许的状态 *simultaneously* 中。当输入完成时，如果这些状态中有任何一个是接受状态，则 NFA 已接受该输入。这个解释对于构建一个有效的 NFA 或将其转换为 DFA 更为有用。

让我们在上面的例子中使用多世界解释。假设输入字符串是 aaac。最初，NFA 处于状态 0。在不消耗任何输入的情况下，它可以通过一次 ε-迁移到状态 1 或状态 4。因此，我们可以认为它的初始状态是所有这些状态同时存在。继续下去，NFA 将遍历这些状态，直到接受完整的字符串 aaac：

States	**Action**
0, 1, 4	consume `a`
0, 1, 2, 4, 5	consume `a`
0, 1, 2, 4, 5	consume `a`
0, 1, 2, 4, 5	consume `c`
6	accept

原则上，可以通过简单地跟踪所有可能的状态来在软件或硬件中实现NFA。但这效率低下。在最坏的情况下，我们需要在每个输入转换上评估所有字符的所有状态。一种更好的方法是将NFA转换为等效的DFA，正如下文所示。

3.5 转换算法

正则表达式和有限自动机在表达能力上是完全等价的。对于每一个 RE，都存在一个 FA，反之亦然。然而，在这三者中，DFA 无疑是最容易在软件中实现的。在本节中，我们将展示如何将 RE 转换为 NFA，再将 NFA 转换为 DFA，最后对 DFA 的规模进行优化。

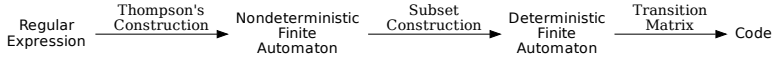


图3.2：正则表达式（RE）、非确定有限自动机（NFA）与确定有限自动机（D

3.5.1 Converting REs to NFAs

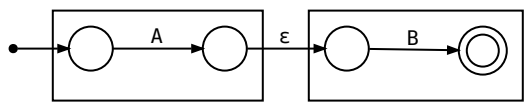
要将正则表达式转换为非确定性有限自动机，可以遵循一种算法，该算法最初由 McNaughton 和 Yamada 提出 [5]，随后由 Ken Thompson 给出 [6]。

我们遵循前面给出的正则表达式的相同归纳定义。首先，我们定义与正则表达式基例相对应的自动机：

T任何字符 a 的 NFA 是：一个 ϵ 转移的 NFA　　tion 是:

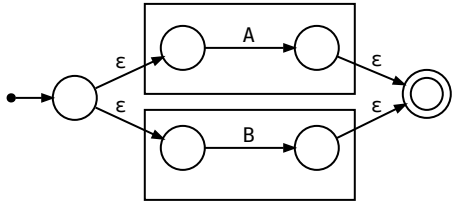
现在，假设我们已经为正则表达式 A 和 B 构造了 NFA，如下所示用矩形表示。A 和 B 都各自只有一个开始状态（在左侧）和一个接受状态（在右侧）。如果我们将 A 和 B 的连接写作 AB，那么相应的 NFA 只是通过一条 ϵ 转换将 A 和 B 连接起来。A 的开始状态成为组合的开始状态，而 B 的接受状态成为组合的接受状态：

AB 串联的 NFA 为：



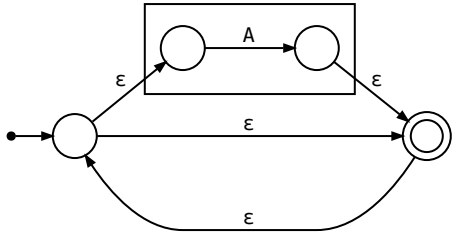
同样地，写作 A | B 的 A 与 B 的交替可以表示为由共同的起始节点和接受节点连接的两个自动机，所有连接均通过 ϵ 转移：

交替 A | B 的 NFA 为：

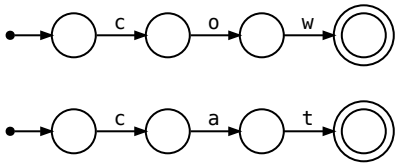


最后，Kleene 闭包 A* 的构造方法是：取 A 的自动机，添加起始节点和接受节点，然后添加 ϵ 转移以允许零次或多次重复：

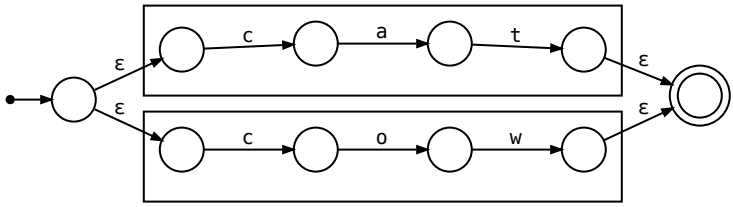
Kleene 闭包 A* 的 NFA 如下：



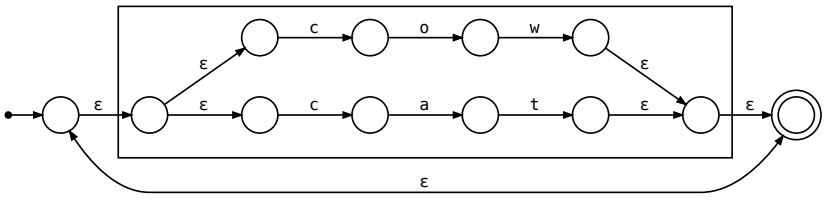
示例。让我们考虑一个示例正则表达式 a(cat | cow)* 的处理过程。首先，我们从最内层的表达式 cat 开始，将其组装为三个转移，从而得到一个接受状态。然后，对 cow 做同样的事情，得到这两个有限自动机：



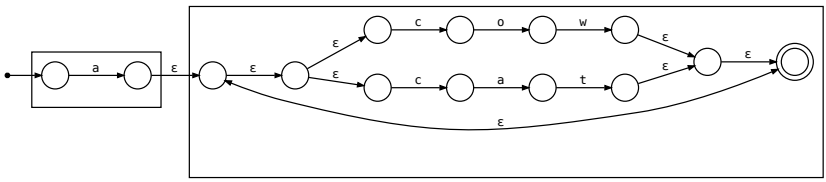
两种表达式 cat | cow 的交替是通过添加一个新的起始节点和接受节点，并使用 ε 转换来实现的。（这些方框不是图的一部分，只是用于突出显示之前的图组件被继续沿用。）



然后，通过在先前的 FA 周围添加另一个起始状态和接受状态，并在其间加入 ε 转移，来实现 Kleene 闭包 (cat | cow)*：



最后，通过在开头为 a 添加一个状态，实现了 a(cat | cow)* 的连接：



你可以很容易地看到，由构造算法生成的 NFA 虽然是正确的，但相当复杂，并且包含大量的 ε 转换。用于表示一门完整语言的词法单元的 NFA 最终可能会拥有成千上万个状态，这在实现上将非常不切实际。相反，我们可以将这个 NFA 转换为一个等价的 DFA。

3.5.2 Converting NFAs to DFAs

我们可以使用子集构造法将任何 NFA 转换为等价的 DFA。其基本思想是构造一个 DFA，使得 DFA 中的每个状态根据“多世界”解释对应于 NFA 中的多个状态。

假设我们从一个由状态 N 和起始状态 N_0 组成的 NFA 开始。我们希望构造一个等价的 DFA，其状态为 D，起始状态为 D_0。每个 D 状态将对应多个 N 状态。首先，我们定义一个称为 ε-闭包 的辅助函数：

ε-闭包。
$\epsilon-$closure(n) 是从 NFA 状态 n 通过零个或多个 ϵ 转移可到达的 NFA 状态集合。

现在我们定义子集构造算法。首先，我们创建一个起始状态 D_0，对应于 $\epsilon-$closure(N_0)。然后，对于来自 D_0 中各个状态的每一个外出字符 c，我们创建一个新状态，其中包含通过 c 可到达状态的 ε-闭包。更准确地说：

子集构造算法。
给定一个具有状态 N 且起始状态为 N_0 的 NFA，构造一个等价的 DFA，其状态为 D，起始状态为 D_0。

令 $D_0 = \epsilon-$closure(N_0)。将 D_0 添加到一个列表中。当列表中仍有项目时：令 d 为从列表中移除的下一个 DFA 状态。对于 Σ 中的每个字符 c：令 T 包含所有满足以下条件的 NFA 状态 N_k：$N_j \in d$ 和 $N_j \xrightarrow{c} N_k$。创建新的 DFA 状态 $D_i = \epsilon-$closure(T)。如果 D_i 尚未在列表中，则将其添加到末尾。

图3.3：子集构造算法

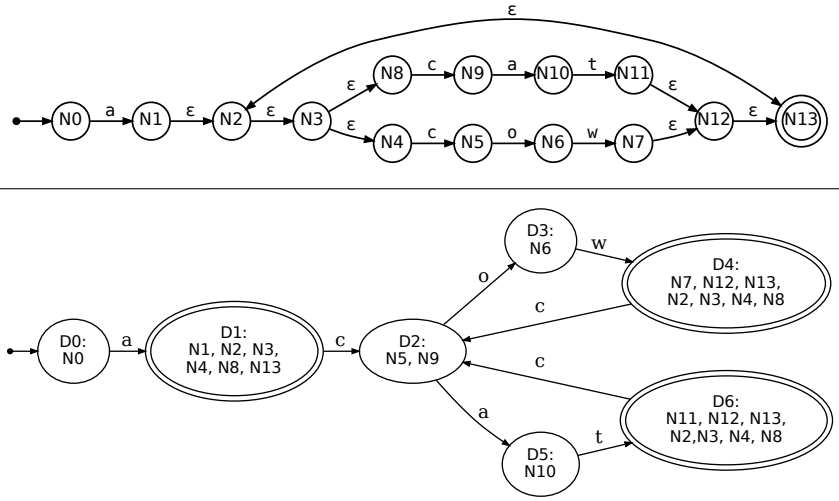


图 3.4：通过子集构造将 NFA 转换为 DFA

示例。让我们在图3.4中的 NFA 上推导该算法。这是与正则表达式 a(cat | cow)* 对应的同一个 NFA，并且为清晰起见对每个状态都进行了编号。

1. 计算 D_0，它是 $\epsilon-$closure(N_0). N_0 没有 ϵ 转移，因此 $D_0 = \{N_0\}$。将 D_0 加入工作列表。 2. 从工作列表中移除 D_0。字符 a 是从 N_0 到 N_1 的一条出边转移。$\epsilon-$closure(N_1) $= \{N_1, N_2, N_3, N_4, N_8, N_{13}\}$，因此将所有这些加入到新状态 D_1，并将 D_1 加入工作列表。 3. 从工作列表中移除 D_1。我们可以看到 $N_4 \xrightarrow{c} N_5$ 和 $N_8 \xrightarrow{c} N_9$，因此我们创建一个新状态 $D_2 = \{N_5, N_9\}$ 并将其加入工作列表。 4. 从工作列表中移除 D_2。由于 $N_5 \xrightarrow{o} N_6$ 和 $N_9 \xrightarrow{a} N_{10}$，a 和 o 都是可能的转移。因此，为到 N_6 的 o 转移创建一个新状态 D_3，并为到 N_{10} 的 a 转移创建一个新状态 D_5。将 D_3 和 D_5 都加入工作列表。

5. 从工作列表中移除 D_3。唯一可能的转移是 $N_6 \xrightarrow{w} N_7$，因此创建一个新的状态 D_4，其中包含 $\epsilon-$closure(N_7)，并将其添加到工作列表中。

6. 从工作列表中移除 D_5。唯一可能的转换是 $N_{10} \xrightarrow{t} N_{11}$，因此创建一个新的状态 D_6，包含 $\epsilon-$closure(N_{11}) 并将其添加到工作列表中。

7. 从工作列表中移除 D_4，并观察到唯一的外部转换 c 导致状态 N_5 和 N_9，它们已经作为状态 D_2 存在，因此只需添加一个转换 $D_4 \xrightarrow{c} D_2$。

8. 从工作列表中移除 D_6，并以类似的方式添加 $D_6 \xrightarrow{c} D_2$。

9. 工作列表为空，因此我们完成了。

3.5.3 Minimizing DFAs

子集构造算法一定会生成一个有效的DFA，但该DFA可能会非常大（特别是如果我们从一个复杂的NFA开始，该NFA是从RE生成的）。一个大的DFA将具有一个大的转换矩阵，这将消耗大量内存。如果它不能适应L1缓存，扫描器可能会非常慢。为了解决这个问题，我们可以应用Hopcroft算法将DFA压缩成一个更小的（但等效的）DFA。

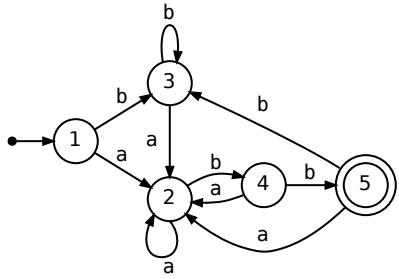
算法的一般方法是乐观地将所有可能等效的状态 S 归为超状态 T。最初，我们将所有非接受状态 S 放入超状态 T_0，将接受状态放入超状态 T_1。然后，我们检查每个状态 $s \in T_i$ 中的出边。如果某个字符 c 有边从 T_i 开始，到 *dif- ferent* 超状态结束，那么我们认为该超状态是 *inconsistent*，相对于 c。（将不允许的转换视为向 T_E 过渡，这是一个错误的超状态。）然后，超状态必须根据 c 拆分成多个一致的状态。对所有超状态和所有字符 $c \in \Sigma$ 重复此过程，直到不再需要拆分为止。

DFA Minimization Algorithm.
Given a DFA with states S, create an equivalent DFA with an equal or fewer number of states T.

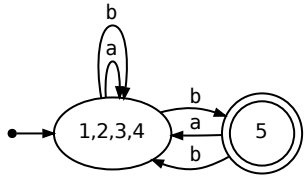
First partition S into T such that:
    $T_0 =$ non-accepting states of S.
    $T_1 =$ accepting states of S.
Repeat:
    $\forall T_i \in T$:
        $\forall c \in \Sigma$:
            if $T_i \xrightarrow{c} \{$ more than one T state $\}$,
            then split T_i into multiple T states
            such that c has the same action in each.
Until no more states are split.

图 3.5：Hopcroft 的 DFA 最小化算法

示例。假设我们有如下未优化的 DFA，并希望将其缩减为一个更小的 DFA：

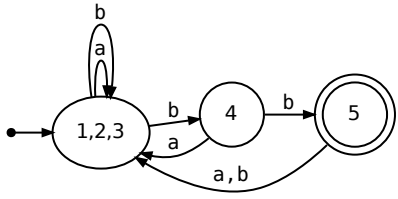


我们首先将所有非接受状态 1、2、3、4 分组为一个超级状态，并将接受状态 5 分分为另一个超级状态，如下所示：

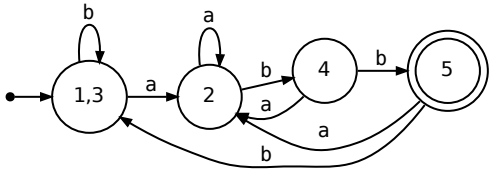


现在，我们通过回到原始 DFA，来询问该图在所有可能输入方面是否是一致的。例如，我们观察到，如果我们处于超状态 (1,2,3,4)，那么输入 a 总是转移到状态 2，这使我们保持在该超状态内。因此，该 DFA 在 a 方面是一致的。然而，从超状态 (1,2,3,4) 出发，输入 b 既可能保持在该超状态内，也可能转移到超状态 (5)。因此，该 DFA 在 b 方面是不一致的。

为了解决这一问题，我们尝试将其中一个不一致的状态（4）拆分出来，形成一个新的超状态，并将相应的转移一并带走：



再一次，我们检查每个超状态与每个输入字符的一致性。再次观察到超状态 1,2,3 对 a 一致，但对 b 不一致，因为它可以导致状态 3 或状态 4。我们尝试通过将状态 2 拆分成自己的超状态来修复此问题，从而得到这个 DFA。

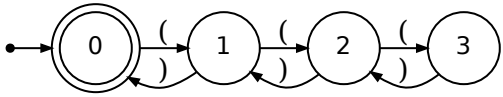


再次，我们检查每个超状态，并观察到每一种可能的输入相对于该超状态都是一致的，因此我们得到了最小的 DFA。

3.6 有限自动机的极限

正则表达式和有限自动机在识别单个词或标记中的简单模式方面既强大又有效，但它们不足以分析问题中的所有结构。例如，你能使用有限自动机来匹配任意数量的嵌套括号吗?

写出一个能够匹配最多三对嵌套括号的FA并不难，比如这样:



但关键字是 *arbitrary*！匹配任意数量的括号将需要一个无限自动机，这显然是不实际的。即使我们应用某个实际的上限（比如 100 对括号），当与语言中必须支持的其他所有元素结合时，自动机仍然会变得极其庞大。

例如，像 Python 这样的语言允许使用圆括号 () 来表示优先级，花括号 {} 来表示字典，以及方括号 [] 来表示列表。一个用于匹配最多 100 对嵌套括号（无论顺序如何）的自动机将有 1,000,000 个状态！

所以，我们将自己限制在使用正则表达式和有限自动机的范围内，目的是识别问题中的单词和符号。为了理解程序的更高层次结构，我们将改为使用第4章介绍的解析技术。

```
%{
   (C Preamble Code)
%}
   (Character Classes)
%%
   (Regular Expression Rules)
%%
   (Additional Code)
```

图 3.6：Flex 文件的结构

3.7 使用扫描器生成器

由于正则表达式精确地描述了一个记号的所有允许形式，我们可以使用程序将一组正则表达式自动转换为扫描器的代码。这样的程序称为扫描器生成器。由 AT&T 开发的 Lex 是扫描器生成器最早的实例之一。Vern Paxson 将 Lex 翻译成 C 语言，创建了 Flex；Flex 以 Berkeley 许可证发布，如今在类 Unix 操作系统中被广泛使用，用于生成以 C 或 C++ 实现的扫描器。

要使用 Flex，我们编写扫描器的规格说明，它由正则表达式、C 代码片段以及一些专用指令混合组成。Flex 程序本身会读取该规格说明并生成常规的 C 代码，然后可以按正常方式进行编译。

图 3.6 给出了 Flex 文件的整体结构。第一部分由任意的 C 代码组成，这些代码将被放置在 scanner.c 的开头，例如 include 文件、类型定义以及类似的内容。通常，这一部分用于包含一个包含各个记号的符号常量的文件。

第二部分声明字符类，它们是对常用正则表达式的符号化简写。例如，你可以声明 DIGIT [0-9]。该类随后可以作为 {DIGIT} 被引用。

第三部分是最重要的部分。它为你希望匹配的每一种记号（token）给出了一个正则表达式，后面跟着一段 C 代码片段，当该表达式被匹配时就会执行。在最简单的情况下，这段代码返回记号的类型，但它也可以用来提取记号的值、显示错误，或执行任何其他合适的操作。

第四部分是任意的 C 代码，将放在扫描器的末尾，通常用于额外的辅助函数。Flex 有一个特殊的要求：我们必须定义一个名为 yywrap() 的函数，它返回 1 以表示在文件末尾输入已经完成。如果我们希望在另一个文件中继续扫描，那么 yywrap() 将打开下一个文件并返回 0。

Flex 接受的正则表达式语言非常类似于

即上文讨论的形式正规表达式。主要区别在于，在正规表达式中具有特殊含义的字符（如圆括号、方括号和星号）必须用反斜杠进行转义，或用双引号括起来。此外，句点（.）可以用来匹配任意字符，这在捕获错误条件时非常有用。

图 3.7 展示了一个简单但完整的示例，帮助你入门。该规范只描述了少量记号：单个字符的加法（必须用反斜杠进行转义）、while 关键字、由一个或多个字母组成的标识符，以及由一个或多个数字组成的数值。正如词法分析器中通常的做法，任何其他类型的字符都会被视为错误，并为此返回一个明确的记号类型。

Flex 会生成扫描器代码，但不是一个完整的程序，因此你必须编写一个与之配套的 main 函数。图 3.8 展示了一个使用该扫描器的简单驱动程序。首先，主程序必须将它期望在生成的扫描器代码中使用的符号声明为 extern：yyin 是读取文本的文件，yylex 是实现扫描器的函数，而数组 yytext 包含所识别出的每个词法单元的实际文本。最后，我们必须在程序的各个部分之间对词法单元类型有一致的定义，因此在 token.h 中放入一个枚举，用于描述新的 token_t 类型。该文件同时被 scanner.flex 和 main.c 包含。

图 3.10 展示了所有部分是如何组合在一起的。通过调用 flex -o scanner.c scanner.flex，将 scanner.flex 转换为 scanner.c。然后，编译 main.c 和 scanner.c 以生成对象文件，最后将它们链接在一起生成完整的程序。

3.8 实际考虑

处理关键字。在许多语言中，关键字（如 while 或 if）如果不进行特殊处理，否则会与标识符的定义相匹配。对此有多种解决方案。一种方法是在 Flex 规范中为每一个关键字分别写一个正则表达式。（这些必须放在标识符定义之前，因为 Flex 会接受第一个匹配的表达式。）另一种方法是维护一个同时匹配所有标识符和关键字的单一正则表达式。与该规则关联的动作可以将词法单元文本与一个单独的关键字列表进行比较，并返回相应的类型。还有一种方法是将所有关键字和标识符都视为同一种词法单元类型，并由语法分析器来解决区分问题。（在像 PL/1 这样的语言中这是必要的，因为标识符可以与关键字同名，并通过上下文来区分。）

跟踪源代码位置。在编译器的后续阶段，解析器或类型检查器准确地知道一个标记所在的行号和列号是很有用的，通常用于打印出有帮助的错误消息。

文件内容：scanner.flex

```
%{#include "token.h" %}数字 [0-9] 字母 [a-zA-Z] %
%(" " | \t | \n) /* 跳过空白 */ \+ { return TOKEN_ADD
; } while { return TOKEN_WHILE; } {LETTER}+ { r
eturn TOKEN_IDENT; } {DIGIT}+ { return TOKEN_
NUMBER; } . { return TOKEN_ERROR; } %%int yy
wrap() { return 1; }
```

图3.7：示例 Flex 规范

文件内容：main.c 译文：

```
#include "token.h" #includ
e <stdio.h>

```cextern FILE *yyin; extern
int yylex(); extern char *yyte
xt;```

```cint main() { yyin = fopen("program.c","r"); if(!yyin) { printf("无法打
开 program.c！ \n"); return 1; }while(1) { token_t t = yylex(); if(t==TOK
EN_EOF) break; printf("令牌: %d 文本: %s\n",t,yytext); } }```
```

图 3.8：示例主程序

文件内容：token.h

```
typedef enum { TOKEN_
EOF=0, TOKEN_WHILE,
TOKEN_ADD, TOKEN_I
DENT, TOKEN_NUMBE
R, TOKEN_ERROR } tok
en_t;
```

图 3.9：示例词元枚举

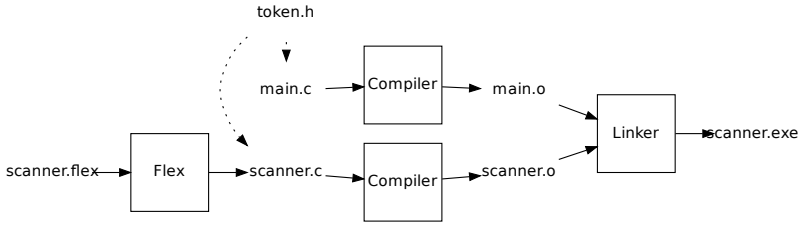


图 3.10：Flex 程序的构建过程

("在第153行发现未定义符号 spider。") 这可以通过让扫描器匹配换行符，并在每次找到时增加行号（但不返回一个标记）来轻松完成。

清理令牌。字符串、字符和类似的令牌类型在匹配后需要进行清理。例如，"hello\n" 需要去掉引号，并将反斜杠-n序列转换为字面意义上的换行符。内部来看，编译器只关心字符串的实际内容。通常，通过在 Flex 规范的尾部编写一个函数 string clean 来实现这一点。在返回所需的令牌类型之前，该函数会在匹配规则中被调用。

限制令牌。尽管正则表达式可以匹配任意长度的令牌，但并不意味着编译器必须准备接受它们。接受一个1000个字符的标识符，或者一个大于机器字长的整数，几乎没有意义。典型的做法是将最大令牌长度（在flex中为YYLMAX）设置为一个非常大的值，然后检查令牌以确定它是否超过了在匹配该令牌的操作中设定的逻辑限制。这允许在需要时发出描述违规令牌的错误消息。

错误处理。处理错误或无效输入的最简单方法就是直接打印一条消息并退出程序。然而，这对你的编译器用户并不友好——如果存在多个错误，（通常）最好一次性看到它们。一个好的做法是匹配尽可能少的无效文本（使用点规则），并返回一个明确表示错误的记号类型。随后，调用扫描器的代码就可以输出合适的消息，然后请求下一个记号。

3.9 习题

1. 为以下实体编写正则表达式。你可能需要说明在每个表达式中哪些是允许的，哪些是不允许的：

   (a) 英语中的星期名称：Monday, Tuesday, ... (b) 所有整数，其中每三位数字用逗号分隔以便清晰，例如：781,092 692,098,000 (c) 互联网电子邮件地址，例如："John Doe" <john.doe@gmail.com> (d) HTTP 统一资源定位符（URL），如 RFC-1738 所描述的那样。

2. 编写一个正则表达式，用于匹配包含任意数量的 X 以及单个成对的 < > 和 { } 的字符串；这些成对结构可以嵌套，但不能交错。例如，以下字符串是允许的：

   XXX<XX{X}XXX>X
   X{X}X<X>X{X}X<X>
   X
   但这些是不允许的：

   XXX<X<XX>>XX
   XX<XX{XX>XX}
   XX

3. 将你在前两个问题中编写的正则表达式转换为你最喜欢的、原生支持正则表达式的编程语言来进行测试。（Perl 和 Python 是两个不错的选择。）通过编写测试用例来评估程序的正确性，这些测试用例应当（以及不应当）匹配。

4. 使用 Thompson 构造法将这些正则表达式转换为 NFA：

   (a) `for | [a-z]+ | [xb]?[0-9]+`

   (b) `a ( bc*d | ed ) d*`

(c) （a*b | b*a | ba）*

5. 使用子集构造法将上一题中的 NFA 转换为 DFA。

6. 使用 Hopcroft 算法最小化上一题中的 DFA。

7. 编写一个手工实现的扫描器，用于 JavaScript 对象表示法（JSON），其规范见 http://json.org。该程序应从输入中读取 JSON，然后打印出所观察到的记号序列：LBRACKET、STRING、COLON 等……在网上找一些大型的 JSON 文档并测试你的扫描器，看看它是否正常工作。

8. 使用 Flex，为 Java 编程语言编写一个扫描器。如上所述，从输入中读取 Java 源代码并输出标记类型。通过将其应用于一个用 Java 编写的大型开源项目来进行测试。

3.10 延伸阅读

1. A.K. 德德尼，《新图灵全书：计算机科学六十六次漫游》，Holt Paperbacks，1992。*An accessible overview of many fundamental problems in computer science – including finite state machines – collected from the author's Mathematical Recreations column in Scientific American.*

2. S. Hollos 和 J.R. Hollos，《有限自动机与正则表达式：问题与解答》，Abrazol 出版社，2013年。*A collection of clever little problems and solutions relating to automata and state machines, if you are looking for more problems to work on.*

3. 马文·明斯基，《计算：有限与无限机器》，普伦蒂斯-霍尔出版社，1967年。*A classic text offering a more thorough introduction to the theory of finite automata at an undergraduate level.*

4. S. Kleene，《神经网络和有限自动机中事件的表示》，载于《自动机研究》，C. Shannon 和 J. McCarthy 编，普林斯顿大学出版社，1956 年。

5. R. McNaughton 和 H. Yamada，“用于自动机的正则表达式与状态图”，《IRE 电子计算机汇刊》，第 EC-9 卷，第 1 期，1960 年。http://dx.doi.org/10.1109/TEC.1960.5221603

6. K. Thompson，《编程技术：正则表达式搜索算法》，《ACM 通讯》（Communications of the ACM），第11卷，第6期，1968年。

http://dx.doi.org/10.1145/363347.363387

第4章 – 解析

4.1 概述

如果说扫描就像用字母构造单词，那么解析就像在自然语言中用单词构造句子。当然，并非每一串单词都能构成一个有效的句子：“horse aircraft conju- gate”是三个有效的单词，但并不是一个有意义的句子。

要解析一个计算机程序，我们必须首先描述一种语言中合法句子的形式。这种形式化的描述被称为上下文无关文法（CFG）。由于允许递归，CFG 比正则表达式更强大，能够表达更加丰富的一组结构。

虽然编写一个普通的 CFG 相对容易，但这并不意味着它易于解析。任意的 CFG 都可能包含二义性以及其他问题，使得编写自动解析器变得困难。因此，我们考虑两种被称为 LL(1) 和 LR(1) 文法的 CFG 子集。

LL(1) 文法是一类上下文无关文法（CFG），它们只需考虑当前产生式和输入流中的下一个记号即可进行解析。这一特性使得编写一种称为递归下降解析器的手工编码解析器变得容易。然而，为了确保某种语言（及其文法）是 LL(1) 文法，必须对其进行精心设计（并且有时需要重写）。并非所有语言结构都可以表示为 LL(1) 文法。

LR(1) 文法比 LL(1) 更通用、功能更强。几乎所有有用的编程语言都可以用 LR(1) 形式来描述。然而，LR(1) 文法的解析算法更为复杂，通常无法手工编写。因此，通常会使用解析器生成器，它能够接受 LR(1) 文法并自动生成解析代码。

4.2 上下文无关文法

让我们先定义 CFG 的各个组成部分。

终结符是可以出现在语言中的离散符号，也被称为前一章中的符号。终结符的例子包括关键字、运算符和标识符。我们将使用小写字母来表示终结符。在这个阶段，我们只需要考虑终结符的类型（例如整数字面量），而不需要考虑终结符的值（例如456）。

非终结符表示一种可以在语言中出现的结构，但它不是字面符号。非终结符的例子包括声明、语句和表达式。我们将使用大写字母来表示非终结符：P 表示程序，S 表示语句，E 表示表达式，等等。

一个句子是语言中终结符的有效序列，而句法形式是终结符和非终结符的有效序列。我们将使用希腊字母表示句法形式。例如，α、β 和 γ 表示（可能）混合的终结符和非终结符序列。我们将使用类似 $Y_1Y_2...Y_n$ 的序列来表示句法形式中的单个符号：Y_i 可以是终结符或非终结符。

上下文无关文法（CFG）是一组规则，正式描述了语言中允许的句子。每条规则的左侧总是一个单一的非终结符。规则的右侧是一个句法形式，描述了该非终结符的允许形式。例如，规则 A $\rightarrow$ xXy 表示非终结符 A 代表一个终结符 x，后跟一个非终结符 X 和一个终结符 y。规则的右侧可以是 ϵ，表示该规则不产生任何东西。第一条规则是特殊的：它是程序的顶级定义，其非终结符被称为开始符号。

例如，这是一个描述涉及加法、整数和标识符的表达式的简单CFG：

语法 $\mathbf{G_2}$

1. P $\rightarrow$ E 2. E $\rightarrow$ E + E 3. E $\rightarrow$ ident 4. E $\rightarrow$ int

该语法可以如下理解：(1) 一个完整的程序由一个表达式组成。(2) 一个表达式可以是任意表达式加上任意表达式。(3) 一个表达式可以是标识符。(4) 一个表达式可以是整数字面量。

为了简洁起见，我们有时通过将所有右侧合并为逻辑或符号来压缩一组具有相同左侧的规则，像这样：

$$\mathsf{E} \rightarrow \mathsf{E} + \mathsf{E} | \mathsf{ident} | \mathsf{int}$$

4.2.1 Deriving Sentences

每一种语法都描述了一个（可能是无限的）句子集合，这个集合称为该语法的语言。要证明一个给定的句子属于该语言，我们必须表明存在一系列规则应用，将开始符号与所需的句子连接起来。规则应用的序列称为一个推导，而双箭头（⇒）用于表示通过应用某条给定规则，一个句型与另一个句型相等。例如：

- 通过应用语法 G_2 的第4条规则，E ⇒ 为 int。
- 通过应用语法 G_2 的规则 3，E +、E ⇒、E + 等同。
- 通过应用文法 G_2 的所有规则来识别 P ⇒ int + ident。

推导有两种方法：自顶向下和自底向上。

在自顶向下的推导中，我们从开始符号出发，然后应用 CFG 中的规则来展开非终结符，直到得到目标句子。例如，ident + int + int 是该语言中的一个句子，下面给出证明它的一个推导过程：

Sentential Form	**Apply Rule**
P	P → E
E	E → E + E
E + E	E → ident
ident + E	E → E + E
ident + E + E	E → int
ident + int + E	E → int
ident + int + int	

在自底向上的推导中，我们从期望的句子开始，然后反向应用规则，直到到达开始符号。下面是同一句子的一个自底向上的推导：

Sentential Form	**Apply Rule**
ident + int + int	E → int
ident + int + E	E → int
ident + E + E	E → E+E
ident + E	E → ident
E + E	E → E+E
E	P → E
P	

务必注意区分 *grammar* (其本身是一个有限的规则集合) 与 *language* (其是由语法生成的字符串集合)。完全有可能两种不同的语法生成同一种语言，在这种情况下，我们称它们具有弱等价性。

4.2.2 Ambiguous Grammars

模糊语法允许同一句子有多种可能的推导方式。我们的示例语法是模糊的，因为任何涉及两个加号的句子都有两种可能的推导方式。句子 ident + int + int 可以有图 4.1 中所示的两种推导方式。

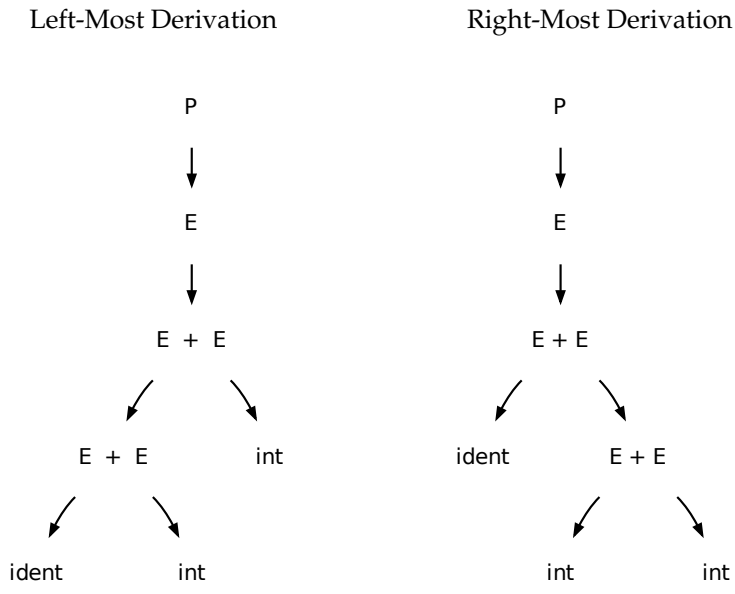


图 4.1：同一句子的两种推导

模糊语法对解析（以及语言设计）构成了一个真实的问题，因为我们不希望程序有两种可能的含义。

在这个例子中重要吗？当然重要！在像 Java 这样的语言中，+ 运算符不仅表示整数之间的加法，还表示字符串之间的连接。如果标识符是 hello，并且这两个整数的值都是 5，那么最左推导会把这三者全部连接起来得到 hello55，而最右推导会先计算 5+5=10，然后将结果连接成 hello10。

幸运的是，通常可以重写文法，使其不再具有歧义。在二元运算符的常见情况下，我们可以要求表达式的一侧是一个原子项（T），如下所示：

Grammar G_3

1. P → E
2. E → E + T
3. E → T
4. T → ident
5. T → int

通过这一改变，语法不再具有歧义，因为它只允许最左推导。但还要注意，它 *still accepts the same lan- guage as Grammar* G_2。也就是说，任何可以由语法 G_2 推导出的句子，也都可以由语法 G_3 推导出来，但每个句子只存在一种推导（以及一种含义）。（证明留作读者的练习。）

现在假设我们希望在我们的语法中添加更多的运算符。如果我们只是简单地添加更多形如 $E \rightarrow E * T$ 和 $E \rightarrow E \div T$ 的规则，我们仍然会得到一个无歧义的语法，但它不会遵循代数中的优先级规则：每个运算符都会从左到右被应用。

相反，通常的方法是构建一个具有多个层次的语法，以反映运算符的预期优先级。例如，我们可以通过将加法和乘法表示为由相乘的因子（*F*）组成的项（*T*）之和来将它们结合起来，如下所示：

Grammar G_4

1. P → E
2. E → E + T
3. E → T
4. T → T * F
5. T → F
6. F → ident
7. F → int

下面是另一个在大多数编程语言中以某种形式出现的常见示例。假设 if 语句有两种变体：if-then 在表达式为真时执行某个操作，而 if-then-else 在表达式为真和为假时分别执行不同的操作。我们可以这样来表达该语言的这一片段：

Grammar G_5

1. P → S
2. S → if E then S
3. S → if E then S else S
4. S → other

语法 G_5 是二义的，因为它允许对这个句子给出两种推导：if E then if E then other else other。你看出问题了吗？else 部分既可以属于外层 if，也可以属于内层 if。在大多数编程语言中，else 被定义为属于内层 if，但该语法并未反映这一点。

现在就做：
写出该句子的两种可能的解析树：if E then if E then othe
r else other。

4.3 LL 文法

LL(1) 文法是上下文无关文法（CFG）的一种子集，使用简单的算法即可轻松解析。若在解析过程中只需考虑一个非终结符以及输入流中的下一个记号，则该文法是 LL(1) 的。

为了确保一个文法是 LL(1)，我们必须做到以下几点：

- 消除任何歧义，如上所示。
- 消除所有左递归，如下所示。
- 消除任何公共左前缀，如下所示。

一旦我们完成了这些步骤，就可以通过为该文法生成 FIRST 和 FOLLOW 集合，并利用它们创建 LL(1) 解析表来证明它是 LL(1)。如果解析表中不包含任何冲突，那么该文法显然是 LL(1)。

4.3.1 Eliminating Left Recursion

LL(1) 文法不能包含左递归，即形如 $A \to A\alpha$ 的规则，或者更一般地说，任何规则 $A \to B\beta$，使得 $B \Rightarrow A\gamma$ 通过某个推导序列成立。例如，规则 E → E + T 是左递归的，因为 E 作为右部的第一个符号出现。

你可能会想通过简单地将规则重写为 E → T + E 来解决这个问题。虽然这样可以避免左递归，但它并不是等价的文法，因为这会导致加号运算符是右结合的。此外，它还会引入一个新的公共左前缀问题，下面将对此进行讨论。

非正式地说，我们必须重写规则，使得（原先）递归的规则以其各个备选项的前导符号开头。

形式上，如果你有如下形式的文法：

$$\text{A} \to A\alpha_1|A\alpha_2|...|\beta_1|\beta_2|...$$

替换为：

$$\text{A} \to \beta_1 A'|\beta_2 A'|...$$

$$\text{A}' \to \alpha_1 A'|\alpha_2 A'|...|\epsilon$$

将该规则应用于语法 Grammar G_3，我们可以将其重写为：

语法 $\mathbf{G_6}$

1. P → E
2. E → T E'
3. E' → + T E'
4. E' → ϵ
5. T → ident
6. T → int

虽然语法 G_6 对人来说可能稍微更难阅读，但它不再包含左递归，并且满足所有 LL(1) 的性质。在规则 2 中，当解析器考虑一个 E 时，会立即考虑 T 非终结符，然后查看输入中的 ident 或 int，以在规则 5 和 6 之间做出决定。在考虑完 T 之后，解析器继续考虑 E'，并通过在输入中查找 + 或任何其他符号来区分规则 3 和 4。

4.3.2 Eliminating Common Left Prefixes

一个更容易解决的问题是：文法中存在多条左部相同、且右部具有共同记号前缀的规则。非正式地说，我们只需寻找某个非终结符的所有公共前缀，并将它们替换为两条规则：一条包含该前缀，另一条包含各个变体。

正式地，寻找这种形式的规则：

$$A \to \alpha\beta_1|\alpha\beta_2|...$$

并替换为：

$$A \to \alpha A'$$

$$A' \to \beta_1|\beta_2|...$$

例如，这些描述标识符、数组引用和函数调用的规则都共享相同的前缀，即一个标识符：

Grammar G_7

1. P → E
2. E → id
3. E → id [E]
4. E → id (E)

如果解析器正在评估 E 并且在输入中看到一个标识符，那么这些信息不足以区分规则 2、3 和 4。然而，可以通过提取共同的前缀来挽救语法，如下所示：

Grammar G_8

1. P → E
2. E → id E'
3. E' → [E]
4. E' → (E)
5. E' → ϵ

在此公式中，解析器在评估*E*时总是消耗一个id。如果下一个符号是[，则应用规则3。如果下一个符号是(，则应用规则4；否则，应用规则5。

4.3.3 First and Follow Sets

为了构造一个完整的LL(1)文法解析器，我们必须计算两个集合，分别称为FIRST和FOLLOW。非正式地，FIRST(α)表示可能出现在α的任何推导开始处的终结符集合（包括ϵ）。FOLLOW(A)表示可能在非终结符A的任何推导后出现的终结符集合（包括\$）。根据这些集合的内容，LL(1)解析器将始终知道接下来选择哪个规则。

这是计算 FIRST 和 FOLLOW 的方法：

Computing First Sets for a Grammar G

FIRST(α) is the set of terminals that begin all strings given by α, including ϵ if $\alpha \Rightarrow \epsilon$.

For Terminals:
For each terminal $a \in \Sigma$: FIRST(a) = $\{a\}$

For Non-Terminals:
Repeat:
  For each rule $X \rightarrow Y_1Y_2...Y_k$ in a grammar G:
    Add a to FIRST(X)
      if a is in FIRST(Y_1)
      or a is in FIRST(Y_n) and $Y_1...Y_{n-1} \Rightarrow \epsilon$
    If $Y_1...Y_k \Rightarrow \epsilon$ then add ϵ to FIRST(X).
until no more changes occur.

For a Sentential Form α:
For each symbol $Y_1Y_2...Y_k$ in α:
  Add a to FIRST(α)
    if a is in FIRST(Y_1)
    or a is in FIRST(Y_n) and $Y_1...Y_{n-1} \Rightarrow \epsilon$
  If $Y_1...Y_k \Rightarrow \epsilon$ then add ϵ to FIRST(α).

Computing Follow Sets for a Grammar G

FOLLOW(A) is the set of terminals that can come after non-terminal A, including \$ if A occurs at the end of the input.

FOLLOW(S) = {\$} where S is the start symbol.

Repeat:
    If A $\rightarrow \alpha B \beta$ then:
        add FIRST(β) (excepting ϵ) to FOLLOW(B).
    If A $\rightarrow \alpha B$ or FIRST(β) contains ϵ then:
        add FOLLOW(A) to FOLLOW(B).
until no more changes occur.

下面是为文法 G_9 计算 FIRST 和 FOLLOW 的一个示例：

Grammar G_9

1. P $\rightarrow$ E
2. E $\rightarrow$ T E'
3. E' $\rightarrow$ + T E'
4. E' $\rightarrow \epsilon$
5. T $\rightarrow$ F T'
6. T' $\rightarrow$ * F T'
7. T' $\rightarrow \epsilon$
8. F $\rightarrow$ (E)
9. F $\rightarrow$ int

语法 G_9 的 First 和 Follow

	P	E	E′	T	T′	F
FIRST	(int	(int	+ ϵ	(int	* ϵ	(int
FOLLOW	\$	) \$	) \$	+) \$	+) \$	+ *) \$

一旦我们将文法整理为 LL(1)，并计算出其 FIRST 集和 FOLLOW 集，就可以开始编写解析器的代码了。这可以手工完成，也可以采用表驱动的方法。

4.3.4 Recursive Descent Parsing

LL(1) 文法非常适合编写简单的手工编码解析器。一种常见的方法是递归下降解析器，其中语法中的每个非终结符都有一个对应的简单函数。函数体遵循相应规则的右部：非终结符会导致调用另一个解析函数，而终结符则意味着处理下一个记号。

需要三个辅助函数：

- 扫描 token() 返回输入流中的下一个标记。
- putback token(t) 将一个意外的标记放回输入流，在下一次调用扫描标记时，该标记将再次被读取。
- expect token(t) 调用扫描令牌以检索下一个令牌。如果令牌匹配预期类型，则返回 true。如果不匹配，它将令牌放回输入流并返回 false。

图 4.2 展示了如何将文法 G_9 写成一个递归下降解析器。注意，该解析器对每个非终结符都有一个函数：parse P、parse E 等。每个函数在输入匹配该文法时返回 true (1)，否则返回 false (0)。

应考虑两种特殊情况。首先，如果规则 X 无法产生 ϵ，并且我们遇到一个不在 FIRST(X) 中的符号，那么我们肯定遇到了语法错误，应显示错误信息并返回失败。其次，如果规则 X *could* 产生 ϵ，并且我们遇到一个不在 FIRST(X) 中的符号，那么我们接受规则 X $\rightarrow \epsilon$，将该符号重新放回输入，并返回成功。另一个规则将期望消耗该符号。

解析器在匹配语法的某个元素后，应该实际执行什么操作的问题也存在。在我们的简单示例中，解析器在匹配时简单地返回 true，仅用于验证输入程序是否与语法匹配。如果我们希望实际评估表达式，那么每个解析 X 函数可以计算结果并返回一个双精度值。这实际上会为我们提供一个简单的解释器。另一种方法是让每个解析 X 函数返回一个表示解析树节点的数据结构。当每个节点被解析时，结果会被组装成一个抽象语法树，根节点由解析 P 返回。

```
int parse_P() { return parse_E() && expect_token(TOKEN_EOF); }


int parse_E() { return parse_T() && parse_E_prime(); }


int 解析_E_prime() { token_t t = 扫描_令牌(); 如果(t==TOKEN
_PLUS) { 返回 解析_T() && 解析_E_prime(); } 否则 { 放回_
令牌(t); 返回 1; } }




int parse_T() { return parse_F() && parse_T_prime(); }


int parse_T_prime() { token_t t = scan_token(); if(t==TOKEN_
MULTIPLY) { return parse_F() && parse_T_prime(); } else { pu
tback_token(t); return 1; } }




int parse_F() { token_t t = scan_token(); if(t==TOKEN_LPAREN) { return parse
_E() && expect_token(TOKEN_RPAREN); } else if(t==TOKEN_INT) { return
1; } else { printf("解析错误：意外的记号 %s\n", token_string(t)); return 0; } }
```

图 4.2：递归下降解析器

4.3.5 Table Driven Parsing

LL(1) 文法也可以使用通用的表驱动代码进行解析。表驱动解析器需要一个文法、一张解析表，以及一个用于表示当前非终结符集合的栈。

LL(1) 分析表用于确定在栈中的非终结符与输入流中的下一个记号的任意组合下，应当应用哪一条规则。（按定义，LL(1) 文法对于每一种组合都恰好有一条可应用的规则。）为了创建分析表，我们如下使用 FIRST 集和 FOLLOW 集：

LL(1) 分析表的构造。
给定语法 G 和字母表 Σ，创建一个解析表 $T[A, a]$，该解析表为每个非终结符 $A \in G$ 与终结符 $a \in \Sigma$ 的组合选择规则。

对于 G 中的每条规则 $A \to \alpha$：对于 FIRST(α) 中除 ϵ) 之外的每个终结符 a (：将 $A \to \alpha$ 加入 $T[A, a]$。如果 ϵ 在 FIRST(α) 中：对于 FOLLOW(A) 中包括 $) 在内的每个终结符 b (：将 A $\to \alpha$ 加入 $T[A, b]$。

例如，这里是语法 G_9 的分析表。注意，P、E、T 和 F 的表项都很直观：每个都只能以 int 或 (开始，因此这些记号会使规则向 F 下降，并在规则 8（$F \to int$）和规则 9（$F \to (E)$）之间进行选择。E' 的表项稍微复杂一些：遇到 + 记号会应用 $E' \to +TE'$，而) 或 $ 则表示 $E' \to \epsilon$。

语法 G_9 的分析表：

	int	**+**	*****	**(**	**)**	**$**
P	1			1		
E	2			2		
E′		3			4	4
T	5			5		
T′		7	6		7	7
F	9			8		

现在我们已经具备了运行解析器所需的全部要素。非正式地说，其思路是维护一个栈，用于跟踪解析器的当前状态。在每一步中，我们考虑栈顶元素以及输入中的下一个记号。如果它们匹配，则弹出栈顶，接受该记号，并继续；如果不匹配，则查阅分析表以确定要应用的下一条规则。如果我们能够一直继续，直到匹配到文件结束符号，那么解析就成功了。

LL(1) Table Parsing Algorithm.
Given a grammar G with start symbol P and parse table T,
parse a sequence of tokens and determine whether they satisfy G.

Create a stack S.
Push \$ and P onto S.
Let c be the first token on the input.

While S is not empty:
    Let X be the top element of the stack.
    If X matches c:
        Remove X from the stack.
        Advance c to the next token and repeat.
    If X is any other terminal, stop with an error.
    If $T[X, c]$ indicates rule $X \to \alpha$:
        Remove X from the stack.
        Push symbols α on to the stack and repeat.
    If $T[X, c]$ indicates an error state, stop with an error.

这是一个 将算法应用于句子 int 的示例 * int：

Stack	Input	Action
P \$	`int * int $`	apply 1: P ⇒ E
E \$	`int * int $`	apply 2: E ⇒ T E'
T E′ \$	`int * int $`	apply 5: T ⇒ F T'
F T′ E′ \$	`int * int $`	apply 9: F ⇒ int
int T′ E′ \$	`int * int $`	match `int`
T′ E′ \$	`* int $`	apply 6: T' ⇒ * F T'
* F T′ E′ \$	`* int $`	match `*`
F T′ E′ \$	`int $`	apply 9: F ⇒ int
int T′ E′ \$	`int $`	match `int`
T′ E′ \$	`$`	apply 7: T' ⇒ ϵ
E′ \$	`$`	apply 4: E' ⇒ ϵ
\$	`$`	match `$`

4.4 LR 文法

虽然LL(1)文法和自顶向下解析技术易于使用，但它们无法表示许多编程语言中存在的所有结构。对于更通用的编程语言，我们必须使用LR(1)文法和相关的自底向上解析技术。

LR(1) 是一类可以通过移进-归约技术并使用一个记号前瞻来进行解析的文法集合。LR(1) 是 LL(1) 的超集，能够容纳 LL(1) 中不允许的左递归和公共左前缀。这使我们能够以更自然的方式表达许多程序构造。（LR(1) 文法仍然必须是非二义的，并且不能存在移进-归约或归约-归约冲突，下面我们将对此进行说明。）

例如，语法 G_{10} 是一个 LR(1) 语法：

Grammar G_{10}

1. $P \rightarrow E$
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow id\ (\ E\ )$
5. $T \rightarrow id$

我们也需要了解 LR(1) 文法的 FIRST 集和 FOLLOW 集，因此现在花点时间，使用第 4.3.3 节中的相同技术，计算文法 G_{10} 的这些集合。

	P	**E**	**T**
FIRST			
FOLLOW			

4.4.1 Shift-Reduce Parsing

LR(1) 文法必须使用移进-归约分析技术进行分析。这是一种自底向上的分析策略，它从记号开始，寻找可应用的规则，将句型归约为非终结符。如果存在一系列归约最终得到开始符号，那么分析就是成功的。

移入动作从输入流中消耗一个记号，并将其压入栈中。归约动作应用语法中的一条形如 A $\rightarrow$ α 的规则，将栈上的句型 α 替换为非终结符 A。例如，下面给出了使用语法 G_{10} 对句子 id(id+id) 进行移入-归约分析的过程：

Stack	**Input**	**Action**
	id (id + id) $	shift
id	(id + id) $	shift
id (	id + id) $	shift
id (id	+ id) $	reduce T $\rightarrow$ id
id (T	+ id) $	reduce E $\rightarrow$ T
id (E	+ id) $	shift
id (E +	id) $	shift
id (E + id	) $	reduce T $\rightarrow$ id
id (E + T	) $	reduce E $\rightarrow$ E + T
id (E	) $	shift
id (E)	$	reduce T $\rightarrow$ id(E)
T	$	reduce E $\rightarrow$ T
E	$	reduce P $\rightarrow$ E
P	$	accept

这个例子虽然表明该句子存在一个推导，但并没有解释在每一步中是如何选择各个动作的。例如，在第二步中，我们本可以选择将 id 归约为 T，而不是移入一个左括号。这将是一个错误的选择，因为没有以 T(开头的规则，但如果不继续尝试向前推进，这一点并不显而易见。为了做出这些决定，我们必须更详细地分析 LR(1) 文法。

4.4.2 The LR(0) Automaton

LR(0) 自动机表示了移进-归约分析器当前正在考虑的所有可能规则。（LR (0) 自动机也被称为该文法的规范集或紧凑有限状态机。）图 4.6 展示了文法 G_{10} 的一个完整自动机。每个方框表示机器中的一个状态，并通过针对文法中的终结符和非终结符的转换相互连接。

自动机中的每个状态由多个项目组成，这些项目是带有一个点（.）的规则，用于指示解析器在该规则中的当前位置。例如，配置 E → E . + T 表示 E 当前在栈上，而 + T 是一种可能的下一个记号序列。

自动机构造如下。状态0是通过获取起始符号的生成式（P → E）并在右侧开头添加一个点来创建的。这表示我们期望看到一个完整的程序，但尚未消耗任何符号。这被称为状态的核心。

状态0的核心

P → . E

然后，我们按如下方式计算状态的闭包。对于状态中每个点号右侧紧跟非终结符 *X* 的项，我们添加文法中所有以 *X* 为左侧的规则。新添加的项在右侧的开头有一个点号。

P → . E
E → . E + T
E → . T

该过程持续进行，直到无法再添加新的项目：

状态0的闭包

P → . E
E → . E + T
E → . T
T → . id (E)
T → . id

你可以这样理解状态：它描述了解析器的初始状态，期望一个完整的程序，形式为E。然而，已知E以E或T开始，并且T必须以id开始。所有这些符号可能代表程序的开头。

从这个状态开始，点右侧的所有符号（包括终结符和非终结符）都是可能的输出转换。如果自动机采取该转换，它将进入一个新的状态，其中包含匹配的项，且点向右移动一个位置。新状态的闭包将被计算，可能会添加如上所述的新规则。

例如，从状态零开始，E、T和 id 是可能的转移，因为每个都出现在某些规则中点的右侧。以下是这些转移的每个状态：

Transition on E:

$$\boxed{\begin{array}{l} \mathsf{P} \rightarrow \mathsf{E} \,. \\ \mathsf{E} \rightarrow \mathsf{E} \,. + \mathsf{T} \end{array}}$$

Transition on T:

$$\boxed{\mathsf{E} \rightarrow \mathsf{T} \,.}$$

Transition on id:

$$\boxed{\begin{array}{l} \mathsf{T} \rightarrow \mathsf{id} \,. \,(\, \mathsf{E} \,) \\ \mathsf{T} \rightarrow \mathsf{id} \,. \end{array}}$$

图4.3给出了文法G_{10}的完整LR(0)自动机。现在花点时间仔细查看表格，确保你理解它是如何构建的。

No, really. Stop now and study the figure carefully before continuing.

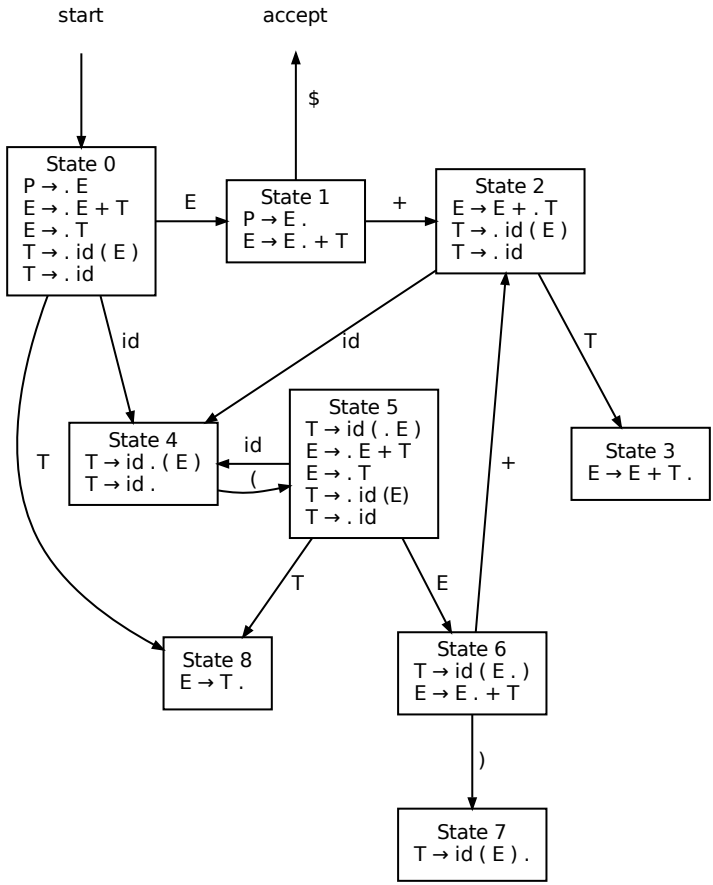


图 4.3：文法 G_{10} 的 LR(0) 自动机

LR(0) 自动机告诉我们在自底向上分析的任一步骤中可用的选择。当我们到达一个包含点位于产生式末尾的项目的状态时，这表明存在一次可能的归约。对终结符的一个转换如果将点向右移动一个位置，则表明存在一次可能的移进。虽然 LR(0) 自动机告诉我们在每一步可用的动作，但它并不总是告诉我们应该采取哪一个 *which* 动作。[1]

在 LR 文法中可能出现两种类型的冲突：

移进-归约冲突表示在移进动作和归约动作之间进行选择。例如，状态4提供了在移进一个左括号和按规则五进行归约之间的选择：

移进-归约冲突：$T \rightarrow \text{id}.\ (\ E\ )$ $T \rightarrow \text{id}\ .$

归约-归约冲突表示两条不同的规则都已被完全匹配，并且任意一条都可能适用。尽管语法 G_{10} 不包含任何归约-归约冲突，但当某种句法结构在语法中的多个层次出现时，这类冲突很常见。例如，函数调用往往既可以单独作为一条语句，也可以作为表达式中的一个元素。对于这样的语法，其自动机将包含如下所示的一个状态：

Reduce-Reduce Conflict:

$S \rightarrow \text{id}\ (\ E\ )\ .$
$E \rightarrow \text{id}\ (\ E\ )\ .$

LR(0) 自动机构成了 LR 解析的基础，它通过告诉我们在每个状态中有哪些可用的动作来实现这一点。但是，它并不告诉我们要采取 *which* 动作，或者如何解决移进-归约和归约-归约冲突。为此，我们必须考虑一些额外的信息。

[1] The 0 in LR(0) indicates that it uses zero lookahead tokens, which is a way of saying that it does not consider the next token before making a reduction. While it is possible to write out a grammar that is strictly LR(0), such a grammar has very limited utility.

4.4.3 SLR Parsing

简单 LR（SLR）分析是 LR 分析的一种基本形式，其中我们使用 FOLLOW 集来解决 LR(0) 自动机中的冲突。简而言之，只有当输入中的下一个记号属于 FOLLOW(A) 时，我们才进行归约 A $\rightarrow \alpha$。如果一种文法可以用这种技术进行分析，我们称其为 SLR 文法，它是 LR(1) 文法的一个子集。

例如，图 4.6 中状态 4 的移进-归约冲突通过查阅 FOLLOW(T) 来解决。如果下一个符号是 $+$、) 或 $，则按照规则 T $\rightarrow$ id 进行归约。如果下一个符号是 (，则移进到状态 5。如果两者都不成立，则输入无效，我们会发出语法错误。

这些决策被编码在SLR分析表中，这些表格在历史上被称为GOTO和ACTION。表格的创建过程如下：

SLR 解析表创建。

给定一个文法 G 和相应的 LR(0) 自动机，创建所有状态 s、终结符 a 和非终结符 A 在 G 中的表格 ACTION[s, a] 和 GOTO[s, A]。

对于每个状态 s：

对于每个像 A $\rightarrow \alpha$. a β 的项，ACTION[s, a] = 根据 LR(0) 自动机转移到状态 t。对于每个像 A $\rightarrow \alpha$. B β 的项，GOTO[s, B] = 根据 LR(0) 自动机转移到状态 t。对于每个像 A $\rightarrow \alpha$. 的项，对于 FOLLOW(A) 中的每个终结符 a：ACTION[s, a] = 根据规则 A $\rightarrow \alpha$ 进行归约。

所有剩余的状态都被视为错误状态。

自然地，表格中的每个状态只能由一个动作占据。如果按照程序执行后，表格中某个条目有多个状态，那么可以得出结论，语法不是 SLR。（它可能仍然是 LR(1) —— 下面会进一步说明。）

这是语法 G_{10} 的 SLR 解析表。请仔细注意状态 1 和状态 4，其中存在移进和归约的选择。在状态 1 中，$+$ 的前瞻导致移进，而前瞻为 $ 时，则会发生归约 P $\rightarrow$ E，因为 $ 是 FOLLOW(P) 的唯一成员。

State	GOTO		ACTION				
	E	T	id	(	)	+	$
0	G1	G8	S4				
1						S2	R1
2		G3	S4				
3					R2	R2	R2
4				S5	R5	R5	R5
5	G6	G8	S4				
6					S7	S2	
7					R4	R4	R4
8					R3	R3	R3

图 4.4：文法 G_{10} 的 SLR 分析表

现在我们准备按照SLR分析算法解析输入。解析过程需要在LR(0)自动机中维护一个状态栈，初始时包含起始状态S_0。然后，我们检查栈顶和前瞻符号，并根据SLR解析表采取相应的动作。在移进操作中，我们消耗该符号并将指定的状态压入栈中。在通过A $\rightarrow \beta$进行规约时，我们从栈中弹出与β中每个符号对应的状态，然后执行额外的步骤，转移到状态GOTO[t, A]。这个过程一直持续，直到我们通过规约到达起始符号成功，或者遇到错误状态失败。

SLR Parsing Algorithm.

Let S be a stack of LR(0) automaton states. Push S_0 onto S.
Let a be the first input token.

Loop:
- Let s be the top of the stack.
- If ACTION[s, a] is **accept**:
  - Parse complete.
- Else if ACTION[s, a] is **shift** t:
  - Push state t on the stack.
  - Let a be the next input token.
- Else if ACTION[s, a] is **reduce** A $\rightarrow \beta$:
  - Pop states corresponding to β from the stack.
  - Let t be the top of stack.
  - Push GOTO[t, A] onto the stack.
- Otherwise:
  - Halt with a parse error.

下面是将 SLR 解析算法应用于程序 id (id + id) 的一个示例。前三个步骤很简单：对前三个记号 id、(、id 分别执行一次移进。第四步对 T → id 进行归约。这会导致状态 4（对应于右部 id）从栈中弹出。此时状态 5 位于栈顶，并且 GOTO[5, T] = 8，因此将状态 8 入栈，得到的栈为 0 4 5 8。

Stack	**Symbols**	**Input**	**Action**
0		id (id + id) $	shift 4
0 4	id	(id + id) $	shift 5
0 4 5	id (	id + id) $	shift 4
0 4 5 4	id (id	+ id) $	reduce T → id
0 4 5 8	id (T	+ id) $	reduce E → T
0 4 5 6	id (E	+ id) $	shift 2
0 4 5 6 2	id (E +	id) $	shift 4
0 4 5 6 2 4	id (E + id	) $	reduce T → id
0 4 5 6 2 3	id (E + T	) $	reduce E → E + T
0 4 5 6	id (E	) $	shift 7
0 4 5 6 7	id (E)	$	reduce T → id(E)
0 8	T	$	reduce E → T
0 1	E	$	accept

(尽管我们展示了“Stack”和“Symbols”两列，但它们只是同一信息的两种表示。栈状态 0 4 5 8 表示 id (T 的解析状态，反之亦然。)

现在应该很清楚，SLR 解析与 LL(1) 解析具有相同的算法复杂度。两种技术都需要一个解析表和一个栈。在这两种算法的每一步中，只需要考虑当前状态以及输入中的下一个记号。区别在于，每个 LL(1) 解析状态只考虑一个非终结符，而每个 LR(1) 解析状态则考虑大量可能的配置。

SLR 解析是理解自底向上解析一般原理的一个良好起点。然而，SLR 是 LR(1) 的一个子集，并非所有 LR(1) 文法都是 SLR。例如，考虑文法 G_{11}，它允许一条语句是变量赋值，或者仅仅是一个标识符本身。注意 FOLLOW(S) = {\$} 以及 FOLLOW(V) = {=]\$}。

Grammar G_{11}

1. S → V = E
2. S → id
3. V → id
4. V → id [E]
5. E → V

我们只需要构建 LR(0) 自动机的一部分就能看到问题：

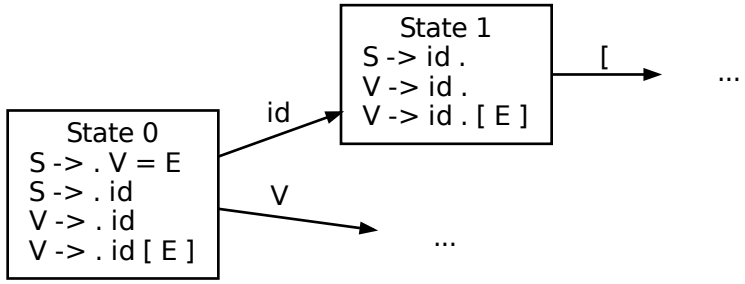


图 4.5：文法 G_{11} 的 LR(0) 自动机的一部分

在状态 1 中，我们可以按 S → id 或 V → id 进行规约。然而，FOLLOW(S) 和 FOLLOW(V) 都包含 \$，因此当下一个记号是文件结束时，我们无法决定选择哪一个。即使使用 FOLLOW 集，仍然存在规约-规约冲突。因此，文法 G_{11} 不是一个 SLR 文法。

但是，如果我们更仔细地观察该语法所允许的可能句子，两者之间的区别就会变得清晰。规则 S → id 只会在完整句子为 id \$ 的情况下应用。如果在起始的 id 之后跟随任何其他记号，则应用 V → id。因此，这个语法本身并不含糊：我们只需要一种更强大的解析算法。

4.4.4 LR(1) Parsing

LR(0) 自动机的能力有限，因为它不跟踪在某个产生式之后实际上可能出现哪些记号。SLR 解析通过使用 FOLLOW 集来决定何时进行归约，从而弥补这一弱点。如上所示，这种方法的判别能力仍不足以解析所有 LR(1) 文法。

现在我们给出 LR(1) 解析的完整或“规范”形式，它依赖于 LR(1) 自动机。LR(1) 自动机类似于 LR(0) 自动机，不同之处在于，在给定当前解析状态的情况下，每个项目都标注了一个可能跟随其后的记号集合。这个集合称为该项目的前瞻符号。前瞻符号始终是相关非终结符的 FOLLOW 集的一个子集。

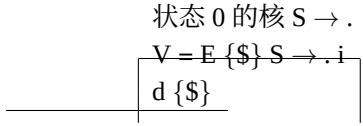
起始状态内核的向前看符号始终是 {$}。在计算一个状态的闭包时，我们考虑两种情况：

- 对于像 A $\rightarrow \alpha.B$ 这样的项，具有 $\{L\}$ 的前瞻，添加像 B $\rightarrow .\gamma$ 这样的新规则，具有 $\{L\}$ 的前瞻。
- 对于像 A $\rightarrow \alpha.B\beta$ 这样的项，具有 $\{L\}$ 的前瞻符号，按如下方式添加具有前瞻符号的新规则，如 B $\rightarrow .\gamma$：

  – 如果 β 不能产生 ϵ，则前瞻符号为 FIRST(β)。 如果 β 可以产生 ϵ，则前瞻符号为 FIRST(β) $\cup\{L\}$。

与之前一样，像 B $\rightarrow .\gamma$ 这样要加入到状态中的规则，对应于语法中所有左部为 B 的规则。

这里给出了语法 G_{11} 的一个示例。起始状态的核心由带有 $ 前瞻符号的起始符号组成：



起始状态的闭包是通过添加针对 V 且前瞻符为 $=$ 的规则来计算的，因为在规则 1 中 $=$ 跟在 V 之后：

Closure of State 0

S → . V = E	{$}
S → . id	{$}
V → . id	{=}
V → . id [E]	{=}

现在假设我们通过在终结符 id 上的一次转移来构造状态 1。每个项目的向前看符号都会被传播到新状态中：

状态1的闭合

S → id .	{$}
V → id .	{=}
V → id . [E]	{=}

现在你可以看到前瞻如何解决归约-归约冲突。当输入中的下一个符号是 $ 时，我们只能通过 S → id 进行归约。当下一个符号是 = 时，我们只能通过 V → id 进行归约。通过比 SLR 更精细地追踪前瞻，我们能够解析任意 LR(1) 语法。

图4.6给出了文法 G_{10} 的完整 LR(1) 自动机。现在花点时间沿着表格逐项查看，确保你理解它是如何构造的。

状态零的一个方面值得澄清。在构造一个状态的闭包时，我们必须考虑文法中的 *all* 条规则，包括与正在闭包的项目相对应的那条规则。项目 E → . E + T 最初以 {$} 作为向前看符号被加入。随后，在评估该项目时，我们加入所有左部为 E 的规则，并为其添加一个向前看符号 $\{+\}$。因此，我们加入 E → . E + T *again*，这一次的向前看符号为 $\{+\}$，从而得到一个向前看集合为 $\{\$, +\}$ 的单个项目

再次：现在停下来，在继续之前仔细研究该图。

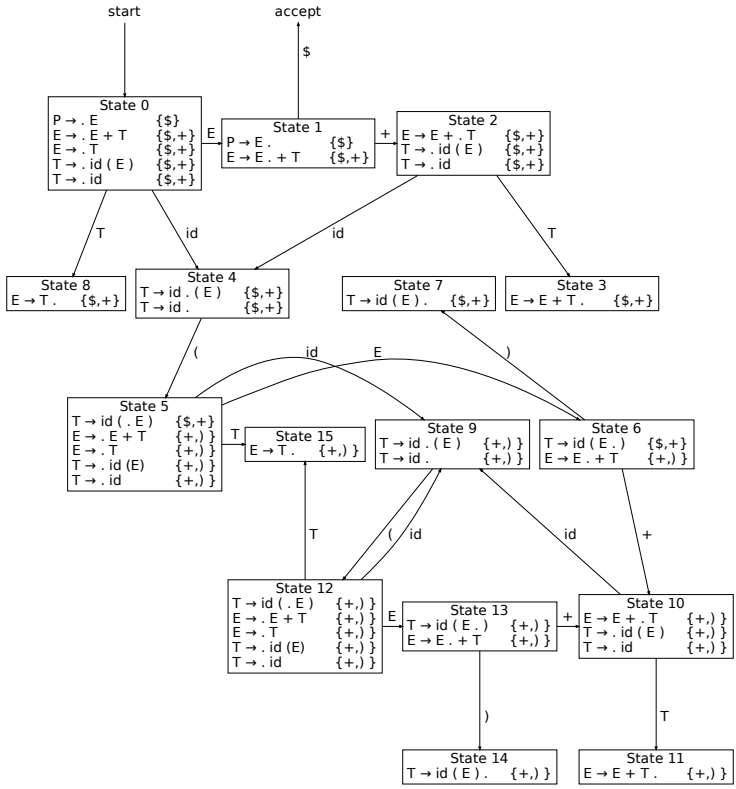


图 4.6：文法 G_{10} 的 LR(1) 自动机

4.4.5 LALR Parsing

LR(1) 解析的主要缺点是 LR(1) 自动机可能比 LR(0) 自动机要 *much* 大。任何两个具有相同项目但在 *any* 项目的前瞻集合上有所不同的状态都被视为不同的状态。结果是产生庞大的解析表，这些表占用大量内存并且使得解析算法变慢。

Lookahead LR (LALR) 解析是解决此问题的实际方法。为了构造一个 LALR 解析器，我们首先创建 LR(1) 自动机，然后合并具有相同核心的状态。一个状态的核心就是一个项目的主体，忽略前瞻。当多个 LR(1) 项目合并成一个 LALR 项目时，LALR 的前瞻是 LR(1) 项目的前瞻的并集。

例如，这两个 LR(1) 状态:

E → . E + T {$+}	E → . E + T {)+}
E → . T {$+}	E → . T {)+}

将合并为这个单一的LALR状态:

E → . E + T {$)+}
E → . T {$)+}

结果 LALR 自动机的状态数与 LR(0) 自动机相同，但为每个项目提供了更精确的前瞻信息。虽然这看起来是一个微小的区别，但经验表明，这一简单的改进在获得 SLR 解析的效率的同时，能够适应大量实际文法。

4.5 语法类别重访

现在你已经有了一些使用不同类型语法的经验，我们来回顾一下它们是如何相互关联的。

$$LL(1) \subset SLR \subset LALR \subset LR(1) \subset CFG \tag{4.1}$$

CFG：上下文无关文法是指规则形式为 A $\rightarrow \alpha$ 的任何文法。为了分析任何上下文无关文法，我们需要一个有限自动机（一个分析表）和一个堆栈来跟踪分析状态。任意上下文无关文法可能是模糊的。一个模糊的上下文无关文法将导致非确定性有限

自动机，这在实践中并不实用。相反，更可取的是重写语法，使其适合一个更受限的类别。

LR(k)：LR(k) 分析器执行一种自底向上的从左到右的输入扫描，并生成最右推导，通过检查输入中的下一个 k 个记号来决定下一步应用哪条规则。规范的 LR(1) 分析器需要一个非常大的有限自动机，因为可能的前瞻被编码进状态中。虽然在形式上只是 CFGs 的一个真子集，但几乎所有现实世界的语言构造都可以在 LR(1) 中得到充分表达。

LALR：前瞻-LR 解析器是通过先构建一个规范的 LR(1) 解析器，然后将所有具有相同核心的项目集合并而成的。这会生成一个小得多的有限自动机，同时保留了一些详细的前瞻信息。虽然在理论上其表达能力不如规范的 LR(1)，但 LALR 通常足以表达现实世界的语言。

SLR：一种 Simple-LR 解析器通过构建 LR(0) 状态机来近似 LR(1) 解析器，然后依赖 FIRST 和 FOLLOW 集来选择应用哪条规则。SLR 简单而紧凑，但存在一些容易找到的常见构造示例是它无法解析的。

LL(k)：LL(k) 解析器对输入执行自顶向下、从左到右的扫描，并给出最左推导的解析；它通过检查输入中接下来的 k 个记号来决定下一步应用哪条规则。LL(1) 解析器简单且被广泛使用，因为它们只需要一个规模为 $O(nt)$ 的表，其中 t 是记号的数量，n 是非终结符的数量。$LL(k)$ 解析器在 $k > 1$ 的情况下不太实用，因为在最坏情况下解析表的大小是 $O(nt^k)$.[2] 然而，它们通常要求对文法进行重写以更适合解析器，并且无法表达所有常见的语言结构。

4.6 乔姆斯基层级

最后，这引出了理论计算机科学中的一项基本成果，称为乔姆斯基层次结构（Chomsky hierarchy）[1]，以著名语言学家诺姆·乔姆斯基命名。该层次结构描述了四类语言（及其对应的文法），并将它们与识别这类语言所需的抽象计算机器联系起来。

正则语言是由正则表达式描述的，正如你在第 3 章中学到的那样。每一个正则表达式都对应一个有限自动机，它可以用来识别相应语言中的所有单词。正如你所知道的，有限自动机可以用一种非常简单的机制来实现：一张表和一个用于表示当前状态的整数。因此，用于正则语言的扫描器非常容易高效地实现。

上下文无关语言是由上下文无关文法描述的，其中每条规则的形式为 A $\to \gamma$，只有一个非终结符在

[2] For example, an LL(1) parser would require a row for terminals {`a`,`b`,`c`,`...`}, while an LL(2) parser would require a row for pairs {`aa`, `ab`, `ac`, `...`}.

Language Class	**Machine Required**
Regular Languages	Finite Automata
Context Free Languages	Pushdown Automata
Context Sensitive Languages	Linear Bounded Automata
Recursively Enumerable Languages	Turing Machine

图 4.7：乔姆斯基层级

左手边，以及右手边由终结符和非终结符混合组成。我们称这些为“上下文无关”，因为非终结符的含义在其出现的所有位置都是相同的。正如你在本章中学到的，CFG 需要一个下推自动机，这是通过将一个有限自动机与一个栈相结合来实现的。如果文法是有歧义的，那么该自动机将是非确定性的，因此在实践中并不实用。在实践中，我们会限制自己使用 CFG 的子集（例如 LL(1) 和 LR(1)），这些文法是无歧义的，并且会产生一个在有界时间内完成的确定性自动机。

上下文相关语言是由上下文相关文法描述的，其中每条规则都可以采用 $\alpha A\beta \rightarrow \alpha\gamma\beta$ 的形式。我们称其为“上下文相关”，因为非终结符的解释受其出现的上下文所控制。上下文相关语言需要一种非确定性的线性有界自动机，其内存消耗是有界的，但执行时间不是有界的。上下文相关语言对于计算机语言来说并不十分实用。

递归可枚举语言是限制最少的一类语言，由形如 $\alpha \rightarrow \beta$ 的规则描述，其中 α 和 β 可以是终结符和非终结符的任意组合。这类语言只能由完整的图灵机来识别，并且是在所有语言中最不实用的。

乔姆斯基层级是语言和编译器设计中一个更一般原则的具体例子：

> 能力最弱的语言提供最强的保证。

也就是说，如果我们有一个需要解决的问题，就应当使用能够解决该问题的、表达能力最弱的工具来着手解决。如果我们*can*通过使用 RE 而不是 CFG 来解决某个给定的问题，那么我们*should*使用 RE，因为它们消耗更少的状态，机制更简单，并且在通向解决方案的过程中设置的障碍更少。

同样的建议也更广泛地适用：汇编语言是我们工具箱中可用的最强大的语言，能够表达计算机所能执行的任何程序。然而，汇编语言也是最难使用的，因为它不提供高级语言中所具有的任何保证。高级语言的能力比汇编语言*less*强，正是这一点使它们更加可预测、可靠，也更易于使用。

4.7 练习

1. 写出对文法 G_5 的一种改进，使其不存在悬空 else 问题。提示：防止内部的 S 包含没有 else 的 if。

2. 为英语中一类有趣的句子子集编写一个语法，包括名词、动词、形容词、副词、连词、从属短语等。（在每个类别中只需包含少量终结符以说明概念。）该语法是 LL(1)、LR(1)，还是有歧义的？请解释原因。

3. 考虑以下文法：

语法 $\mathbf{G_{12}}$

1. P → S 2. P → S P 3. S → 如果 E 则 S 4. S → 如果 E 则 S 否则 S 5. S → 当 E 时 S 6. S → 开始 P 结束 7. S → 打印 E 8. S → E 9. E → 标识符 10. E → 整数 11. E → E + E

（a）指出文法 G_{12} 中所有不满足 LL(1) 的方面。（b）编写一个新的文法，接受相同的语言，但避免左递归和公共左前缀。（c）写出新文法的 FIRST 集和 FOLLOW 集。（d）给出新文法的 LL(1) 分析表。（e）新文法是否是 LL(1) 文法？请仔细解释你的答案。

4. 考虑以下文法：

语法 $\mathbf{G_{13}}$ 1. S → id = E 2. E → E + P 3. E → P 4. P → id 5. P → (E) 6. P → id(E)

(a) 为文法 G_{13} 画出 LR(0) 自动机。(b) 写出文法 G_{13} 的完整 SLR 分析表。(c) 该文法是 LL(1) 吗？解释原因。(d) 该文法是 SLR 吗？解释原因。

5. 考虑前文所示的语法 G_{11}。

(a) 写出语法 G_{11} 的完整 LR(1) 自动机。 (b) 将 LR(1) 自动机压缩为语法 G_{11} 的 LALR 自动机。

6. 写出描述形式正则表达式的上下文无关文法。首先写出你能想到的最简单的（可能是歧义的）文法，基于第3章中的归纳定义。然后，将文法重写为等效的LL(1)文法。

7. (a) 为 JSON 数据表示语言编写一个文法。 (b) 为你的文法写出 FIRST 集和 FOLLOW 集。 (c) 你的文法是 LL(1)、SLR 还是 LR(1)，或者都不是？如有必要，重写它，直到它属于尽可能简单的文法类别。 (d) 写出适用于你的文法的分析表。

8. 编写一个可运行的手写 JSON 表达式解析器，利用在上一章中构建的 JSON 扫描器。

9. 创建一个手工编码的扫描器和一个递归下降解析器，用于对在控制台中输入的一阶逻辑表达式进行求值。包含布尔值（T/F）以及运算符 &（与）、| (or)、!（非）、-> (implication)，以及 ()（分组）。

例如，这些表达式应该计算为真 Understood. Please provide

TT 和 T | F (F -> F)
-> T

这些表达式应该计算为假:

F! (T | F) (T -> F
) & T

10. 编写一个手动编码的解析器，读取正则表达式并输出相应的 NFA，使用 Graphviz [2] DOT 语言。

11. 编写一个解析器构建工具，读取LL(1)文法并生成用于表驱动解析器的可工作代码作为输出。

4.8 延伸阅读

1. N. Chomsky，《关于语法的某些形式性质》，《信息与控制》，第2卷，第2期，1959年。http://dx.doi.org/10.1016/S0019-9958(59)90362-6 2. J. Ellson，E. Gansner，L. Koutsofios，S. North，G. Woodhull，《Graphviz——开源图绘制工具》，国际图绘制研讨会，2001年。http://www.graphviz.org 3. J. Earley，《一种高效的上下文无关解析算法》，《ACM 通讯》，第13卷，第2期，1970年。https://doi.org/10.1145/362007.362035 4. M. Tomita（编），《广义 LR 解析》，Springer，1991年。

第5章——解析实践

在本章中，你将运用你所学的 LL(1) 和 LR(1) 文法理论，构建一个用于简单表达式语言的可工作解析器。这将为你在下一章编写一个更完整的 B-Minor 解析器奠定基础。

虽然LL(1)分析器通常是手写的，但LR(1)分析器实在太复杂，无法通过相同的方式编写。因此，我们依赖于一个解析器生成器，将文法的规范输入进去，并自动生成可工作的代码。在本章中，我们将使用Bison作为示例，Bison是一个广泛使用的C语言类似语言的解析器生成器。

使用 Bison，我们将为代数表达式定义一个 LALR 语法，然后利用它创建三种不同类型的程序。

- 验证器读取输入程序，然后仅向用户告知它是否是由该语法指定的语言中的一个有效句子。此类工具通常用于判断给定程序是否符合某一标准或另一标准。

- 解释器读取输入程序，然后实际执行程序以生成结果。一种解释方法是立即在解析每个操作时计算其结果。另一种方法是将程序解析为抽象语法树，然后执行它。

- 翻译器读取输入程序，将其解析为抽象语法树，然后遍历该抽象语法树，以生成一种不同格式但等价的程序。

5.1 野牛解析器生成器

手动实现LALR解析器并不现实，因此我们依赖工具根据语法规范自动生成表格和代码。YACC（Yet Another Compiler Compiler）曾是Unix环境中广泛使用的解析器生成器，最近被通常兼容的GNU Bison解析器取代。Bison被设计为在需要时自动调用Flex，因此将两者结合成一个完整的程序非常方便。

正如扫描器一样，我们必须创建一个要解析的语法规范，其中每个规则都伴随一个执行的动作。Bison 文件的整体结构与 Flex 文件类似：

%{ (C 前言代码) %} (声明)
%% (语法规则) %% (C 后记
代码)

第一部分包含任意的 C 代码，通常是 #include 语句和全局声明。第二部分可以包含特定于 Bison 语言的各种声明。我们将使用 %token 关键字来声明我们语言中的所有终结符。文件的主体部分包含一系列形式为的规则。

表达式 : 表达式 TOKEN_ADD 表达
式 | TOKEN_INT ;

表明非终结符 expr 可以生成该句子
expr TOKEN ADD expr，或者单一的终结符 TOKEN INT。空白并不重要，因此可以为了清晰起见重新排列规则。注意，通常的命名约定在这里被反转了：由于大写通常用于 C 常量，我们使用小写来表示非终结符。

生成的代码会创建一个名为 yyparse() 的单一函数，该函数返回一个整数：零表示解析成功，一表示解析错误，二表示内部问题，例如内存耗尽。yyparse 假定存在一个名为 yylex 的函数，用于返回整数类型的词法记号。这可以手工编写，或由 Flex 自动生成。在后一种情况下，可以通过修改文件指针 yyin 来更改输入源。

图 5.1 给出了一个用于整数上的简单代数表达式的 Bison 规范。请记住，Bison 接受 LR(1) 文法，因此在各个规则中包含左递归是可以的。

```
%{#include <stdio.h> %}


%token TOKEN_INT %tok
en TOKEN_PLUS %token
TOKEN_MINUS %token T
OKEN_MUL %token TOK
EN_DIV %token TOKEN_
LPAREN %token TOKEN_
RPAREN %token TOKEN_
SEMI %token TOKEN_ER
ROR

%%程序 : 表达式 TOKEN_SEMI;


表达式 : 表达式 TOKEN_PLUS 项 | 表
达式 TOKEN_MINUS 项 | 项 ;


项 : 项 TOKEN_MUL 因子 | 项 TOKE
N_DIV 因子 | 因子 ;


因子: TOKEN_MINUS 因子 | TOKEN_LPAREN 表
达式 TOKEN_RPAREN | TOKEN_INT ; %%


int yyerror( char *s ) { printf("解析错误: %s\n",s);
return 1; }
```

图5.1：表达式验证器的 Bison 规范 r

```
#include <stdio.h>

extern int yyparse();

int main() { if(yyparse()==0) { printf("解析成功！\n");
} else { printf("解析失败。\n"); } }
```

图 5.2：表达式验证器的主程序

图 5.3 显示了一个结合使用 Bison 和 Flex 的程序的一般构建过程。解析器规范放在 parser.bison 中。我们假设您已经编写了一个合适的扫描器并将其放在 scanner.flex 中。之前，我们手动编写了 token.h。在这里，我们将依赖 Bison 从 %token 声明自动生成 token.h，以便解析器和扫描器使用相同的信息。像这样调用 Bison：

```
bison --defines=token.h --output=parser.c parser.bison
```

--output=parser.c 选项指示 Bison 将其代码写入文件 parser.c，而不是晦涩的 yy.tab.c。随后，我们分别编译 parser.c、由 Flex 生成的 scanner.c 以及 main.c，并将它们链接在一起生成一个完整的可执行文件。

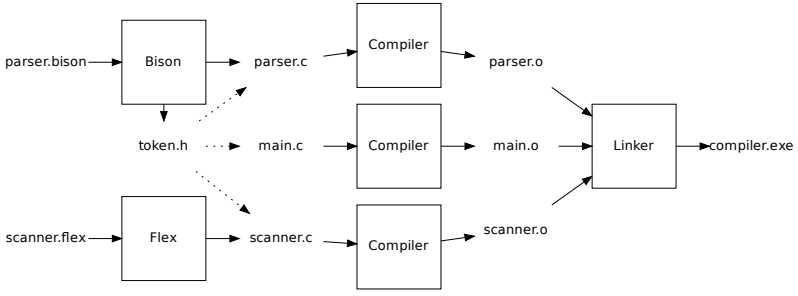


图 5.3：Bison 与 Flex 的联合构建流程

如果您给 Bison 添加 -v 选项，它将把 LALR 自动机的文本表示输出到文件 parser.output。对于每个状态，它列出项目，并使用点号表示解析器的位置。然后，它列出应用于该状态的动作。例如，假设我们修改上面的语法，使其变得模糊：

表达式：表达式 TOKEN_PLUS 表达式

Bison 会报告该语法存在一个移进-归约冲突，而 parser.output 将描述每个状态。在发生冲突时，Bison 会抑制一个或多个动作，这一点会在下面的报告中用方括号表示：

状态 9

2 expr: expr . TOKEN_PLUS expr 2 | expr TOKEN_PLUS expr .

TOKEN_PLUS 移入，并转到状态 7 TOKEN_PLUS [使用规则 2（expr）进行规约] $default 使用规则 2（expr）进行规约

请注意！如果你的语法存在移进-归约或归约-归约冲突，Bison 仍会愉快地输出可工作的代码，但会抑制其中一些发生冲突的动作。该代码在简单输入上可能看起来能够正常工作，但在完整程序中很可能产生意想不到的效果。在继续之前务必检查是否存在冲突。

5.2 表达式验证器

如所示，图 5.1 中的 Bison 规范仅会评估输入程序是否匹配所期望的语法。yyparse() 在成功时返回零，否则返回非零。这样的程序称为验证器，通常用于确定给定程序是否符合标准。

有多种用于 HTML[1]、CSS[2] 和 JSON[3] 等 Web 相关语言的在线验证器。通过将严格的语言定义与实际实现（可能包含非标准特性）相分离，程序员就更容易判断其代码是否符合标准，从而（可以推定）具有可移植性。

[1] http://validator.w3.org
[2] http://css-validator.org
[3] http://jsonlint.com

5.3 表达式解释器

为了不仅仅验证程序，我们必须利用语法本身嵌入的语义动作。在任何产生式规则的右侧，您可以在大括号内放置任意的 C 代码。该代码可以引用表示已计算的其他非终结符值的语义值。语义值由美元符号表示，指示产生式规则中非终结符的位置。两个美元符号表示当前规则的语义值。

例如，在加法规则中，适当的语义操作是将左侧值（第一个符号）加到右侧值（第三个符号）上：

```
expr : expr TOKEN_PLUS term { $$ = $1 + $3; }
```

语义值 $1 和 $3 从何而来？它们只是来自于定义这些非终结符的其他规则。最终，我们会到达一条为叶节点给出取值的规则。例如，这条规则表明，一个整数记号的语义值就是该记号文本的整数值：

```
因子 : TOKEN_INT { $$ = atoi(yytext);                    }
```

（注意：token 的值来自扫描器中的 yytext 数组，因此只有当规则右侧只有一个终结符时，才能这样做。）

在非终结符展开为单个非终结符的情况下，我们只需将一个语义值赋给另一个：

```
术语 : 因子 { $$ = $1; }
```

因为Bison是一个自底向上的解析器，它首先确定语法树中叶节点的语义值，然后将这些值传递给内部节点，依此类推，直到结果到达起始符号。

图 5.4 展示了一个实现完整解释器的 Bison 语法。主程序仅调用 yyparse()。如果成功，结果会存储在全局变量 parser result 中，供主程序提取和使用。

```
prog : expr TOKEN_SEMI       { parser_result = $1; }
     ;

expr : expr TOKEN_PLUS term        { $$ = $1 + $3; }
     | expr TOKEN_MINUS term       { $$ = $1 - $3; }
     | term                             { $$ = $1; }
     ;

term : term TOKEN_MUL factor       { $$ = $1 * $3; }
     | term TOKEN_DIV factor       { $$ = $1 / $3; }
     | factor                           { $$ = $1; }
     ;

factor
     : TOKEN_MINUS factor              { $$ = -$2; }
     | TOKEN_LPAREN expr TOKEN_RPAREN   { $$ = $2; }
     | TOKEN_INT              { $$ = atoi(yytext); }
     ;
```

图 5.4：用于解释器的 Bison 规范

5.4 表达式树

到目前为止，我们的表达式解释器在解析输入的过程中就计算结果。虽然这对简单表达式有效，但它有几个普遍的缺点：其中之一是，程序可能执行了大量计算，却在程序的后期才发现解析错误。通常更理想的是在执行*before*之前发现所有解析错误。

为了解决这个问题，我们将为解释器添加一个新阶段。我们不会直接计算值，而是构造一个称为抽象语法树的 数据结构来表示表达式。一旦创建了 AST，我们就可以遍历这棵树，根据需要检查、执行并翻译程序。

图 5.5 展示了用于表示表达式的一个简单 AST 的 C 代码。expr_t 枚举了五种表达式节点的类型。struct expr 描述了树中的一个节点，该节点由一个 kind、左右指针以及用于叶子节点的一个整数值组成。函数 expr_create 用于创建任意类型的新树节点，而 expr_create_value 则专门创建 kind 为 EXPR_VALUE 的节点。[4]

[4] Although it is verbally awkward, we are using the term “kind” rather than “type”, which will have a very specific meaning later on.

Contents of File: expr.h

```
typedef enum {
    EXPR_ADD,
    EXPR_SUBTRACT,
    EXPR_DIVIDE,
    EXPR_MULTIPLY,
    EXPR_VALUE
} expr_t;
```

```
struct expr {
        expr_t kind;
        struct expr *left;
        struct expr *right;
        int value;
};
```

文件内容：expr.c

结构体 expr * expr_create(expr_t kind, 结构体 expr *left, 结构体 ex
pr *right)

{ struct expr *e = 分配(sizeof(*e)); e->kind = kind; e->value = 0; e->left
= left; e->right = right; 返回 e; }struct expr * expr_create_value(int valu
e) { struct expr *e = expr_create(EXPR_VALUE,0,0); e->value = value;
返回 e; }

图 5.5：表达式解释器的 AST

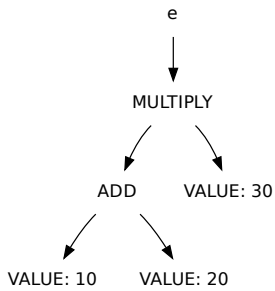
使用该表达式结构，我们可以手工创建一些简单的 AST。例如，如果我们想创建一个与表达式 (10+20)*30 对应的 AST，我们可以发出以下一系列操作：

```
struct expr *a = expr_create_value(10); struct expr *b = expr_create_va
lue(20); struct expr *c = expr_create(EXPR_ADD,a,b); struct expr *d =
expr_create_value(30); struct expr *e = expr_create(EXPR_MULTIPL
Y,c,d);
```

当然，我们也可以通过编写一个包含嵌套值的单一表达式来完成同样的事情：

```
struct expr *e = expr_create(EXPR_MULTIPLY, ex
pr_create(EXPR_ADD, expr_create_value(10), expr
_create_value(20) ),expr_create_value(30) );
```

无论哪种方式，结果都是类似这样的一个数据结构：



与其手工构建 AST 的每个节点，我们希望让 Bison 自动完成同样的工作。当表达式的每个元素被识别时，应在树中创建一个新节点，并向上传递，以便将其链接到合适的位置。通过自底向上的解析，Bison 会先创建树的叶子节点，然后再将它们链接到父节点中。

为此，我们必须为每条规则编写语义动作，以便在树中创建一个节点，或从下方节点向上传递指针。图 5.6 展示了这是如何完成的：

%{
#include "expr.h"
#define YYSTYPE struct expr *
结构体 expr * 解析结果 = 0;
It looks like you didn’t provide any source text after the “%}” marker. Could you please sh

/* 令牌定义已省略以简洁起见 请提供您需要翻译的源文本，我

程序 : 表达式 TOKEN_SEMI { 解析结果 = $1;
} ;

表达式 : 表达式 TOKEN_PLUS 项 { $$ = expr_create(EXPR_ADD,$1,$3);
} | 表达式 TOKEN_MINUS 项 { $$ = expr_create(EXPR_SUBTRACT,$1,
$3); } | 项{ $$ = $1; } ;

术语 : 术语 TOKEN_MUL 因子 { $$ = expr_create(EXPR_MULTIPLY,$1,
$3); } | 术语 TOKEN_DIV 因子 { $$ = expr_create(EXPR_DIVIDE,$1,$3);
} | 因子 { $$ = $1; } ;

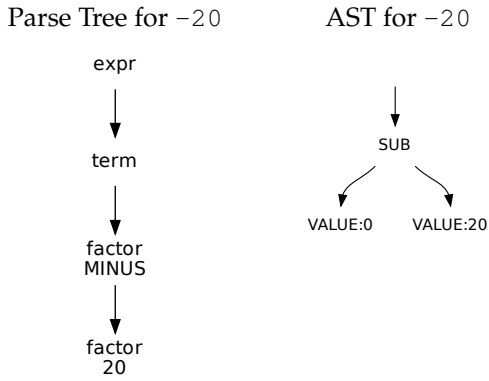
因子: TOKEN_MINUS 因子 { $$ = expr_create(EXPR_SUBTRACT, expr_creat
e_value(0),$2); } | TOKEN_LPAREN 表达式 TOKEN_RPAREN { $$ = $2; } |
TOKEN_INT { $$ = expr_create_value(atoi(yytext)); ;

图 5.6: 为表达式文法构建抽象语法树 (AST)

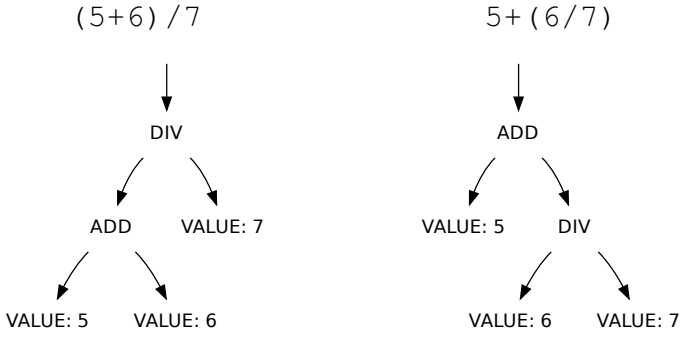
仔细查看图 5.6，并注意以下几点：

- 在前言中，我们必须通过将宏 YYSTYPE 设置为 struct expr * 来显式定义语义类型。这会使 Bison 在任何使用语义值（如 $$ 或 $1）的地方都使用 struct expr * 作为内部类型。当然，最终的解析器结果也必须具有相同的语义类型。

- AST 并不总是与解析树直接对应。举例来说，当 expr 产生一个 factor 时，我们只是将指向底层节点的指针向上传递，使用 {$$ = $1;}。另一方面，当我们在 term 中遇到一元负号时，我们返回一个子树，它实际上实现了左侧取值为零、右侧为该表达式的减法。



- 括号并不会直接在 AST 中表示。相反，它们通过对树中节点进行排序来实现所需的求值顺序。例如，考虑由以下语句生成的 AST：



```
int expr_evaluate( struct expr *e ) { if(!e) return 0; int l = expr_evaluate(e->left); int r = expr_evaluate(e->right); switch(e->kind) { case EXPR_VALUE: return e->value; case EXPR_ADD: return l+r; case EXPR_SUBTRACT: return l-r; case EXPR_MULTIPLY: return l*r; case EXPR_DIVIDE: if(r==0) { printf("错误：除以零\n"); exit(1); }return l/r; }return 0; }
```

图 5.7：表达式求值

现在我们已经构建了 AST，就可以将其作为计算以及许多其他操作的基础。

AST 可以通过调用如图 5.7 所示的 expr evaluate 进行算术求值。该函数通过在节点的左、右指针上递归调用自身来对树执行后序遍历。（注意函数开头对空指针的检查。）这些调用返回 l 和 r，其中包含左、右子树的整数结果。然后，通过根据当前节点的种类进行分支来计算该节点的结果。（还要注意我们必须显式检查除以零的情况，否则当 r 为零时，expr evaluate 将会崩溃。）

```
void expr_print( struct expr *e ) { if(!e) return; printf("("); expr_print(e->l
eft); switch(e->kind) { case EXPR_VALUE: printf("%d",e->value); break
; case EXPR_ADD: printf("+"); break; case EXPR_SUBTRACT: printf("-
"); break; case EXPR_MULTIPLY: printf("*"); break; case EXPR_DIVID
E: printf("/"); break; }expr_print(e->right); printf(")"); }
```

图 5.8：打印表达式

以类似的方式，AST 可以通过调用 expr print 转换回文本，如图 5.8 所示。该函数通过递归调用 expr print 对节点的左侧进行中序遍历，显示当前节点，然后在右侧调用 expr print。再次注意函数开始时的空值检查。

如前所述，括号不会直接反映在 AST 中。为了保守起见，这个函数会在每个值外都显示一对括号。虽然这是正确的，但会产生大量括号！一个更好的解决方案是，仅当某个子树包含优先级较低的运算符时才打印括号。

5.5 练习

1. 查阅 Bison 手册，确定如何从你的语法中自动生成 LALR 自动机的图。将 Bison 的输出与你手工绘制的版本进行比较。 2. 修改 expr evaluate()（以及任何需要的其他部分），使其能够处理浮点值而不是整数。 3. 修改 expr print()，使其只显示为保证正确性所必需的最少数量的括号。

4. 扩展解析器和解释器，允许调用多个内置数学函数，如 sin(x)、sqrt(x) 等。在编码之前，思考一下函数名称应该放在哪里。它们应该是语言中的关键字吗？还是所有函数名称应该简单地作为标识符，并在 expr evaluate() 中进行检查？5. 扩展解析器和解释器，允许变量赋值和使用，以便可以编写多个赋值语句，后面跟着一个待求值的表达式，如：g = 9.8; t = 5; g*t*t - 7*t + 10;

5.6 进一步阅读

顾名思义，YACC 并不是第一个编译器构造工具，但它至今仍被广泛使用，并催生了大量类似的工具，这些工具使用各种语言编写，面向不同类别的文法。下面只是其中的一小部分：

1. S. C. Johnson，《YACC：又一个编译器-编译器》，贝尔实验室技术期刊，1975年。 2. D. Grune 和 C.J.H. Jacobs，《一种对程序员友好的 LL(1) 解析器生成器》，《Software: Practice and Experience》，第18卷，第1期。 3. T.J. Parr 和 R.W. Quong，《ANTLR：一种带谓词的 LL(k) 解析器生成器》，《Software: Practice and Experience》，1995年。 4. S. McPeak，G.C. Necula，《Elkhound：一种快速、实用的 GLR 解析器生成器》，编译器构造国际会议，2004年。

.

第6章——抽象语法树

6.1 概述

抽象语法树（AST）是一种重要的内部数据结构，用于表示程序的主要结构。AST 是对程序进行语义分析的起点。之所以称其为“抽象”，是因为该结构省略了解析过程中的具体细节：AST 并不关心一种语言是采用前缀、后缀还是中缀表达式。（事实上，我们在这里描述的 AST 可用于表示大多数过程式语言。）

对于我们的项目编译器，我们将用五种 C 结构来定义一个 AST，分别表示声明、语句、表达式、类型和参数。虽然你在学习编程的过程中肯定遇到过这些术语，但在实际使用中它们并不总是被精确定义。本章将帮助你非常清晰地理清这些概念：

- 声明指定符号的名称、类型和值，以便在程序中使用。符号包括常量、变量和函数等项。

- 语句表示要执行的一个操作，该操作会改变程序的状态。示例包括循环、条件语句以及函数返回。

- 表达式是由值和操作组成的组合，按照特定规则进行求值，并产生一个值，例如整数、浮点数或字符串。在某些编程语言中，表达式还可能具有副作用，从而改变程序的状态。

对于 AST 中的每一种元素，我们都会给出代码示例以及其构建方式。由于这些结构中的每一个都可能指向其他各类的指针，因此在了解它们如何协同工作之前，有必要先对它们进行整体预览。

一旦你理解了 AST 的所有元素，我们将通过演示如何使用 Bison 解析器生成器自动创建整个结构来结束本章。

6.2 声明

一个完整的B小调程序是声明的序列。每个声明都说明一个变量或函数的存在。变量声明可以选择性地给出初始化值。如果没有给出初始化值，则默认值为零。函数声明可以选择性地给出函数的代码体；如果没有给出代码体，那么该声明作为其他地方声明的函数的原型。

例如，以下都是有效的声明：

b: 布尔值; s: 字符串 = "hello"; f: 函数 整数 (x: 整数) = { 返回 x*x; }

声明由一个decl结构表示，该结构给出名称、类型、值（如果是表达式）、代码（如果是函数）以及指向程序中下一个声明的指针：

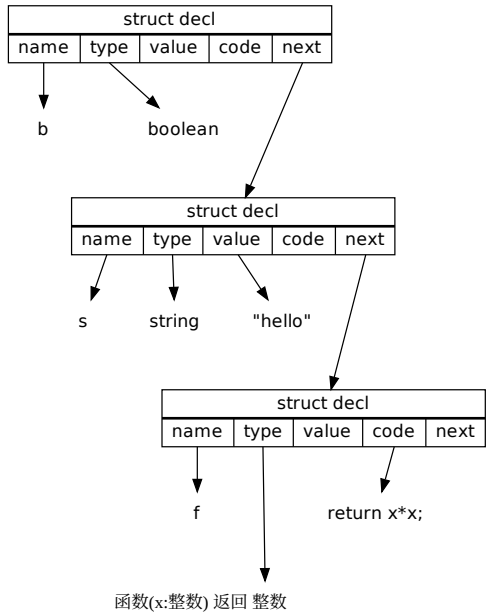
结构声明 { char *name; 结构类
型 *type; 结构表达式 *value; 结
构语句 *code; 结构声明 *next; }
;

因为我们将创建许多这样的结构体，因此你需要一个工厂函数来分配结构体并初始化它的字段，如下所示：

struct decl * decl_create(char *name, struct type *type, struct expr *
value, struct stmt *code, struct decl *next) { struct decl *d = malloc
(sizeof(*d)); d->name = name; d->type = type; d->value = value; d-
>code = code; d->next = next; return d; }

（你需要为语句、表达式等编写类似的代码，但我们不会在这里反复说明。）

前一页上的三个声明可以用图形方式表示为一个链表，如下所示：



请注意，有些字段并不指向任何内容：这些将由空指针表示，为了清晰起见我们将其省略。此外，我们的示意图并不完整，必须加以扩展：表示类型、表达式和语句的各项本身都是复杂的结构，需要加以描述。

6.3 语句

函数体由一系列语句组成。语句表示程序将按照指定的顺序采取特定的动作，例如计算一个值、执行循环，或在备选分支之间进行选择。语句也可以是局部变量的声明。下面是 stmt 的结构：

```
结构体 stmt { stmt_t 种类; 结构体 decl *声明; 结构体 expr *初始化表达式; 结构体 expr *表达式; 结构体 expr *下一个表达式; 结构体 stmt *主体; 结构体 stmt *否则主体; 结构体 stmt *下一个; };
```

```
typedef enum { STMT_DECL, STMT_EXPR, STMT_IF_ELSE, STMT_FOR, STMT_PRINT, STMT_RETURN, STMT_BLOCK } stmt_t;
```

kind 字段指示语句的类型：

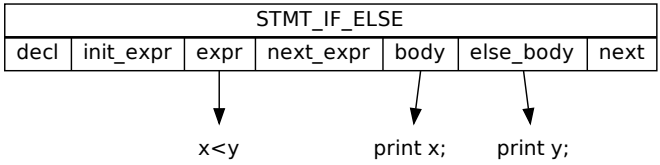
- STMT_DECL 表示一个（局部）声明，且 decl 字段将指向它。
- STMT_EXPR 表示一个表达式语句，expr 字段将指向它。
- STMT_IF_ELSE 表示一个 if-else 表达式，其中 expr 字段指向控制表达式，body 字段指向在条件为真时执行的语句，而 else body 字段指向在条件为假时执行的语句。
- STMT_FOR 表示一个 for 循环，其中 init expr、expr 和 next expr 是循环头中的三个表达式，而 body 指向循环中的语句。
- STMT_PRINT 表示一个打印语句，而 expr 指向要打印的表达式。
- STMT_RETURN 表示一个 return 语句，而 expr 指向要返回的表达式。
- STMT_BLOCK 表示花括号内的一个语句块，而 body 指向其中包含的语句。

并且，像我们对声明所做的那样，我们需要一个函数 stmt create 来创建并返回一个语句结构：

struct stmt * stmt_create(stmt_t kind,
结构体 decl *decl，结构体 expr *init_expr，结构体 expr *e
xpr，结构体 expr *next_expr，结构体 stmt *body，结构体
stmt *else_body，结构体 stmt *next);

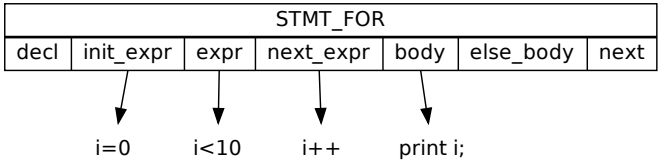
这种结构包含许多字段，但每个字段都有其用途，并会在特定类型的语句需要时使用。例如，if-else 语句只使用 expr、body 和 else body 字段，其余字段保持为 null：

如果(x<y) 输出 x; 否则 输出 y;



for 循环使用三个 expr 字段来表示循环控制的三个部分，并使用 body 字段来表示正在执行的代码：

```
for(i=0;i<10;i++) print i;
```



6.4 表达式

表达式的实现方式与第 5 章中展示的简单表达式 AST 十分相似。不同之处在于，我们需要更多的二元类型：语言中的每个运算符都需要一个类型，包括算术、逻辑、比较、赋值等运算符。我们还需要为每一种叶子节点的值提供一个类型，包括变量名、常量值等。对于 EXPR NAME，会设置 name 字段；对于整数值，则会设置整数值字段。

EXPR_INTEGER_LITERAL，等等。随着你扩展编译器，可能需要向该结构添加值和类型。

```
struct expr {
    expr_t kind;
    struct expr *left;
    struct expr *right;

    const char *name;
    int integer_value;
    const char *
       string_literal;
};
```

```
typedef enum {
    EXPR_ADD,
    EXPR_SUB,
    EXPR_MUL,
    EXPR_DIV,
    ...
    EXPR_NAME,
    EXPR_INTEGER_LITERAL,
    EXPR_STRING_LITERAL
} expr_t;
```

和之前一样，你应该为一个二元运算符创建一个工厂：

```
struct expr * expr_create( expr_t kind, struct expr *L, struct expr
*R );
```

然后为每种叶子类型各自提供一个工厂：

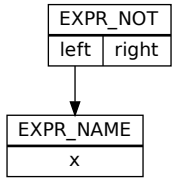
```
struct expr * expr_create_name( const char *name ); struct expr * expr_crea
te_integer_literal( int i ); struct expr * expr_create_boolean_literal( int b ); st
ruct expr * expr_create_char_literal( char c ); struct expr * expr_create_strin
g_literal

                                        ( const char *str );
```

请注意，您可以将整数、布尔值和字符文字值都存储在整数值字段中。

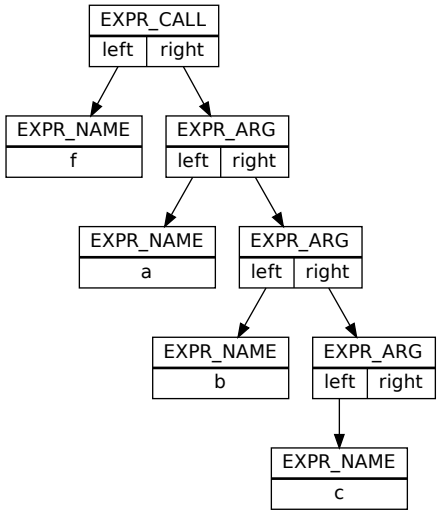
有几个案例值得特别提及。像逻辑非这样的单目运算符通常将它们的唯一参数放在左指针中：

!x



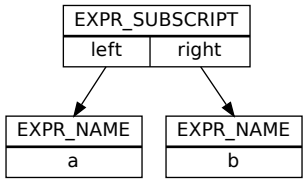
函数调用是通过创建一个 EXPR CALL 节点来构造的，其中左侧是函数名，右侧是不平衡的 EXPR ARG 节点树。虽然这种表示看起来有些别扭，但它允许我们用树来表达链表，并且在代码生成过程中会简化函数调用参数在栈上的处理。

f(a,b,c)



数组下标访问被视为一种二元运算符，因此数组名位于 EXPR SUBSCR IPT 运算符的左侧，右侧是一个整数表达式:-

a[b]



6.5 类型

类型结构对声明中提到的每个变量和函数的类型进行编码。像整数和布尔这样的原始类型，只需将 kind 字段设置为合适的值，并将其他字段置为 nu ll 即可表示。像数组和函数这样的复合类型，则通过将多个类型结构连接在一起来构建。

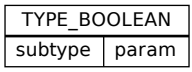
typedef 枚举 { TYPE_VOI
D, TYPE_BOOLEAN, TYP
E_CHARACTER, TYPE_I
NTEGER, TYPE_STRING,
TYPE_ARRAY, TYPE_FU
NCTION } type_t;

结构体 type { type_t kind; 结构体 type *su
btype; 结构体 param_list *params; };

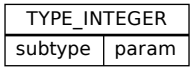
struct param_list { char *name; struct typ
e *type; struct param_list *next; };

例如，为了表示布尔或整数这样的基本类型，我们只需创建一个独立的类型结构，将 kind 设置为适当的值，其余字段为 null：

`boolean`

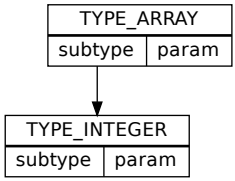


`integer`



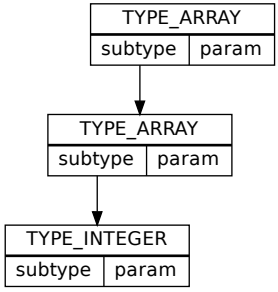
要表示诸如整数数组这样的复合类型，我们将 kind 设置为 TYPE ARRAY，并将 subtype 设置为指向 TYPE INTEGER：

数组 [] 整数



这些可以链接到任意深度，因此要表示一个整数数组的数组：

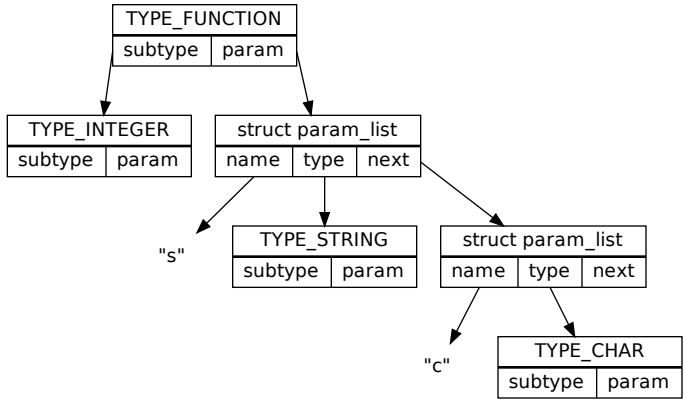
数组 [] 数组 [] 整数



为了表达函数的类型，我们使用 subtype 来表示函数的返回类型，然后连接一个参数列表节点的链表，用于描述函数中每个参数的名称和类型。

例如，这是一个接受两个参数并返回一个整数的函数的类型：

函数 整数 (s:字符串, c:字符)



请注意，这里的类型结构使我们能够表达一些复杂而强大的高阶编程概念。只需替换为复杂类型，你就可以描述一个包含十个函数的数组，每个函数都返回一个整数：

a：数组[10] 函数 整数（x：整数）；

或者一个返回函数的函数呢?

f：函数 函数 整数 (x:整数) (y:整数);

或者甚至是一个返回函数数组的函数！

g: 函数 数组 [10] 函数 整数 (x:整数) (y:整数);

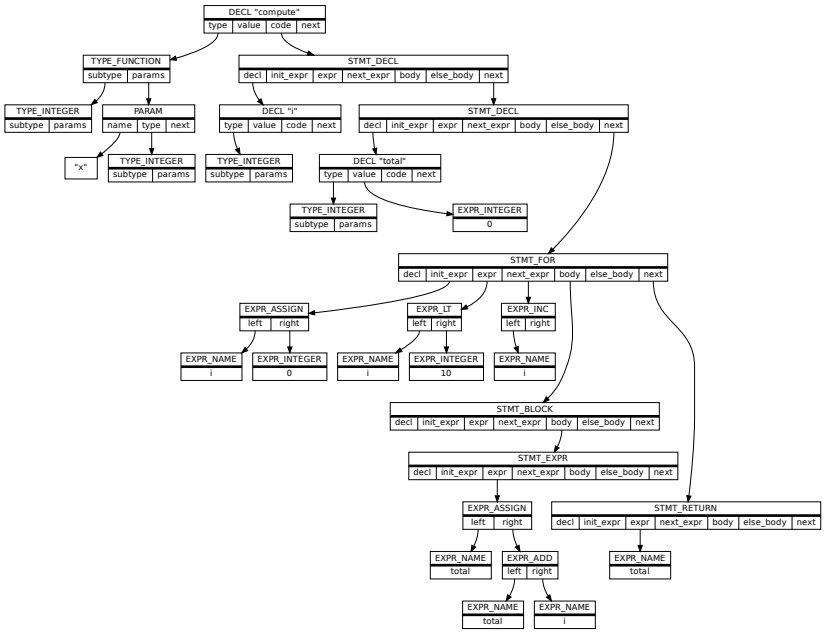
虽然 B-Minor 类型系统能够 *expressing* 这些想法，但这些组合会在之后的类型检查中被拒绝，因为它们需要一种比我们准备实现的更加动态的实现。如果你觉得这些想法很有趣，那么你应该去了解诸如 Scheme 和 Haskell 这样的函数式语言。

6.6 综合起来

既然你已经看过每个独立的组件，让我们看看一个完整的 B-Minor 函数将如何以 AST 的形式表示：

计算: 函数 整数 (x:整数) = { i: 整数; 总计: 整数 = 0; for(i=0
;i<10;i++) { 总计 = 总计 + i; }返回 总计;

}



6.7 构建 AST

利用本章到目前为止创建的函数，原则上我们可以以一种嵌套的方式手动构造 AST。例如，下面的代码表示一个名为 square 的函数，它接受一个整数 x 作为参数，并返回 x*x 的值：

源文本：d = decl_create("square", type_create(TYPE_FUN
CTION, type_create(TYPE_INTEGER,0,0), param_list_crea
te("x", type_create(TYPE_INTEGER,0,0), 0)), 0,stmt_create
(STMT_RETURN,0,0, expr_create(EXPR_MUL, expr_creat
e_name("x"), expr_create_name("x")), 0,0,0,0), 0);

显然，这根本不是编写代码的方式！相反，我们希望解析器在每次产生式被归约时调用相应的创建函数，然后将它们连接成一棵完整的树。使用像 Bison 这样的 LR 解析器生成器，这个过程是直截了当的。（这里我会给出推进的方法思路，但要完成解析器，你还需要自己弄清许多细节。）

在顶层，B-Minor 程序是一个声明序列：

```
program : decl_list
          { parser_result = $1; }
        ;
```

然后，我们为 B-Minor 程序中各种不同类型的声明编写规则：

decl : name TOKEN_COLON type TOKEN_SEMI { $$ = decl_create($1,$3,0,0,
0); } | name TOKEN_COLON type TOKEN_ASSIGN expr TOKEN_SEMI { $$
= decl_create($1,$3,$5,0,0); } | /* 以及此处还有更多情况 */ . . . ;

由于每个 decl 结构都是分别创建的，我们必须将它们连接成一个由 decl list 形成的链表。最简单的方法是将该规则设为右递归，这样左侧的 decl 表示一个声明，而右侧的 decl list 表示链表的其余部分。当 decl list 产生 ϵ 时，链表的末尾是一个空值。

```
decl_list : decl decl_list
            { $$ = $1; $1->next = $2; }
          | /* epsilon */
            { $$ = 0; }
          ;
```

对于每一种语句，我们创建一个 stmt 结构，用于从语法中提取所需的元素。

源文本：stmt : TOKEN_IF TOKEN_LPAREN expr TOKEN_RPAREN stmt { $$ = s tmt_create(STMT_IF_ELSE,0,0,$3,0,$5,0,0); } | TOKEN_LBRACE stmt_list TOKE N_RBRACE { $$ = stmt_create(STMT_BLOCK,0,0,0,0,$2,0,0); } | /* 以及更多情况在此 */ . . . ; 翻译文本：

以这种方式逐一深入 B-Minor 程序的各个语法要素：声明、语句、表达式、类型、参数，直到到达字面值和符号这些叶子元素，它们的处理方式与第 5 章相同。

还有最后一个复杂点：当每条规则被归约时，返回的值在语义上究竟是什么类型？它并不是单一类型，因为每种规则都会返回不同的数据结构：声明规则返回一个 struct decl *，而标识符规则返回一个 char *。为了解决这个问题，我们需要告知 Bison，语义值是 AST 中所有这些类型的联合。

%union {struct decl *decl; struct stmt
*stmt; . . . char *name;

};

然后指出每条规则所使用的并集的具体子字段：

%type <decl> 程序 decl_list decl . . . %type <stmt> stmt_list
stmt . . . . . %type <name> name

6.8 练习

1. 为 B-Minor 编写一个完整的 LR 文法，并使用 Bison 对其进行测试。你的第一次尝试肯定会产生许多移进-归约和归约-归约冲突，因此请运用第 4 章中关于文法的知识来重写文法并消除这些冲突。 2. 按照本章的说明编写 AST 结构及其生成函数，并像上面所示那样，使用嵌套的函数调用手动构造一些简单的 AST。 3. 添加新的函数，如 decl print()、stmt print() 等，用于将 AST 再次打印出来，以便验证程序是否正确生成。请使用缩进和一致的间距对输出进行良好格式化，使代码易于阅读。 4. 将 AST 生成函数作为动作规则加入你的 Bison 文法中，这样你就可以解析完整的程序，并再次将其打印出来。 5. 添加新的函数，如 decl translate()、stmt translate() 等，将 B-Minor 的 AST 输出为你自己选择的另一种语言，例如 Python、Java 或 Rust。 6. 添加新的函数，以图形化形式输出 AST，从而可以“看到”程序的结构。一种方法是使用 Graphviz 的 DOT 格式：让每个声明、语句等作为图中的一个节点，然后让结构之间的每个指针作为图中的一条边。

第7章——语义分析

既然我们已经完成了 AST 的构建，我们就可以开始分析语义，也就是程序的实际含义，而不仅仅是它的结构。

类型检查是语义分析的重要组成部分。总体而言，编程语言的类型系统为程序员提供了一种方式，用以做出可验证的断言，编译器可以自动对其进行检查。这使得错误能够在编译期被发现，而不是在运行期。

不同的编程语言在类型检查方面有不同的方法。有些语言（如 C）具有相当弱的类型系统，因此如果不小心就可能犯下严重的错误。其他语言（如 Ada）拥有非常强的类型系统，但这也使得编写一个能够通过编译的程序变得更加困难！

在我们执行类型检查之前，必须确定表达式中使用的每个标识符的类型。然而，变量名与其实际存储位置之间的映射并非一目了然。表达式中的变量 x 可能引用局部变量、函数参数、全局变量，或完全不同的其他实体。我们通过执行名称解析来解决这一问题，在该过程中，每个变量的定义都会被录入符号表。在语义分析阶段，只要需要评估某段代码的正确性，就会查阅该符号表。

一旦名称解析完成，我们就拥有了进行类型检查所需的全部信息。在这一阶段，我们根据标准的类型转换规则，将每个值的基本类型组合起来，从而计算复杂表达式的类型。如果某个类型以不被允许的方式使用，编译器将输出一条（理想情况下是有帮助的）错误信息，以协助程序员解决问题。

语义分析还包括对程序正确性的其他形式的检查，例如检查数组的边界、避免错误的指针遍历，以及检查控制流。根据语言的设计，其中一些问题可以在编译时检测到，而另一些可能需要等到运行时。

7.1 类型系统概述

大多数编程语言都会为每一个值（无论是字面量、常量还是变量）赋予一种类型，该类型描述了该变量中数据的解释方式。类型表明该值是整数、浮点数、布尔值、字符串、指针或其他形式。在大多数语言中，这些原子类型可以组合成更高阶的类型，如枚举、结构体和变体类型，以表达复杂的约束。

一门语言的类型系统有多种用途：

- 正确性。编译器利用程序员提供的类型信息，在程序试图执行不恰当的操作时发出警告或错误。例如，将一个整数值赋给一个指针变量几乎可以肯定是错误的，即使二者在内存中都可能被实现为一个机器字。一个良好的类型系统可以在编译时标记这些问题，从而帮助消除运行时错误。

- 性能。编译器可以利用类型信息来找到一段代码最有效的实现。例如，如果程序员告诉编译器某个给定的变量是常量，那么同一个值可以被加载到寄存器中并多次使用，而不是不断地从内存中加载。

- 表达性。如果语言允许程序员省略那些可以从类型系统推断出的事实，那么程序可以变得更加紧凑和富有表现力。比如，在 B-Minor 中，print 语句不需要被告知它正在打印的是整数、字符串还是布尔值：类型会从表达式中推断出来，值也会自动以适当的方式显示。

编程语言（及其类型系统）通常按照以下几个维度进行分类：

- 安全或不安全
- 静态或动态
- 显式或隐式

在不安全的编程语言中，可以编写出形式上有效、但其行为却极其未定义、并且违背程序基本结构的程序。例如，在 C 编程语言中，程序可以构造任意指针来修改内存中的任意字，从而改变已编译程序的数据和代码。这种能力在实现诸如操作系统或驱动程序等底层代码时可能是必要的，但在一般的应用程序代码中却是有问题的。

例如，下面的 C 代码在语法上是合法的并且可以编译，但它是不安全的，因为它会向数组 a[] 的边界之外写入数据。结果，程序几乎可能产生任何结果，包括错误的输出、静默的数据损坏，或无限循环。

```
/* 这是 C 代码 */ int i; int a[10]; for(i=0;
i<100;i++) a[i] = i;
```

在一种安全的编程语言中，不可能编写出违反该语言基本结构的程序。也就是说，无论向用安全语言编写的程序提供什么输入，它都会以良好定义的方式执行，并保持该语言的抽象。安全的编程语言通过强制数组边界、指针的使用以及类型的分配来防止未定义行为。大多数解释型语言，如 Perl、Python 和 Java，都是安全语言。

例如，在 C# 中，数组的边界会在运行时进行检查，因此越界访问数组会产生可预测的效果：抛出一个 IndexOutOfRangeException：

```
/* 这是 C# 代码 */ a = new int[10]; for(int i=0;
i<100;i++) a[i] = i;
```

在静态类型语言中，所有的类型检查都在编译时完成，远在程序运行之前。这意味着程序可以被翻译成基本的机器代码，而无需保留任何类型信息，因为所有操作都已经被检查并判定为安全。这会产生最高性能的代码，但也确实消除了某些方便的编程习惯。

静态类型通常用于区分整数和浮点运算。尽管诸如加法和乘法等操作在源语言中通常用相同的符号表示，但它们在实现上对应着本质上不同的机器代码。例如，在 x86 机器上的 C 语言中，(a+b) 对于整数会被翻译为 ADDL 指令，而对于浮点值则会翻译为 FADD 指令。为了知道应当应用哪条指令，我们必须首先确定 a 和 b 的类型，并推断 + 的预期含义。

在一种动态类型语言中，类型信息在运行时可用，并与其所描述的数据一起存储在内存中。随着程序的执行，每个操作的安全性都会通过比较各个操作数的类型来进行检查。如果观察到类型不兼容，那么程序必须因运行时类型错误而终止。这也使得

能够显式检查变量类型的代码。例如，Java 中的 instanceof 运算符允许显式地测试类型：

```
/* 这是 Java 代码 */ public void sit( Furniture f ) { if (f instanceof Chair) { System.out.println("坐直！"); } else if ( f instanceof Couch ) { System.out.println("你可以瘫坐。"); } else { System.out.println("你可以正常坐。"); } }
```

在显式类型语言中，程序员需要负责在代码中明确指明变量及其他项目的类型。这对程序员来说需要更多的工作量，但可以降低出现意外错误的可能性。例如，在像 C 这样的显式类型语言中，由于将浮点数赋值给整数会导致精度丢失，下面的代码可能会产生错误或警告：[1]

```
/* 这是 C 代码 */ int x = 32.
5;
```

显式类型也可以用来防止在具有相同底层表示但含义不同的变量之间进行赋值。例如，在 C 和 C++ 中，不同类型的指针具有相同的实现（一个指针），但将它们相互替换是没有意义的。下面的代码应当产生一个错误，或者至少一个警告：

```
/* 这是 C 代码 */ int *i; float
*f = i;
```

在隐式类型语言中，编译器会推断变量和表达式的类型（尽可能地），无需程序员明确输入。这使得程序更加简洁，但可能会导致意外的行为。例如，最近的 C++ 标准允许声明一个自动类型的变量 auto，如下所示：

```
/* 这是 C++11 代码 */ auto x =
32.5; cout << x << endl;
```

[1] Not all C compilers will generate a warning, but they should!

编译器判定 32.5 的类型是 double，因此 x 也必须具有 double 类型。类似地，输出运算符 << 被定义为在整数上具有某种行为，在字符串上具有另一种行为，等等。在这种情况下，编译器已经确定 x 的类型是 double，因此它选择对 double 进行操作的 << 变体。

7.2 设计类型系统

为了描述一门语言的类型系统，我们必须解释其原子类型、其复合类型，以及在类型之间进行赋值和转换的规则。

一种语言的原子类型是用于描述单个变量的简单类型，这些变量通常（但并非总是）在汇编语言中存储在单个寄存器里：整数、浮点数、布尔值等等。对于每一种原子类型，都有必要清晰地定义其所支持的取值范围。例如，整数可以是有符号或无符号的，可以是 8 位、16 位、32 位或 64 位；浮点数可以是 32 位、40 位或 64 位；字符可以是 ASCII 或 Unicode。

许多语言允许用户定义类型，在这种类型中，程序员定义一种新的类型，该类型使用原子类型实现，但通过限制其取值范围来赋予新的含义。例如，在 Ada 中，你可以为天和月定义新的类型：

```
-- 这是 Ada 代码 type Day is range 1..
31; type Month is range 1..12;
```

这很有用，因为处理“天”和“月”的变量与函数现在被分开保存，从而防止你不小心把一个赋给另一个，或者把值 13 赋给 Month 类型的变量。

C 也有一个类似的特性，但要弱得多：typedef 为某个类型声明了一个新名称，但无法限制其取值范围，也不能阻止你在具有相同基础类型的类型之间进行赋值：

```
/* 这是 C 代码 */ typedef i
nt Month; typedef int Day;

/* 在 C 中，将 m 赋值给 d 是允许的，因为它们都
是整数。 */

月份 m = 10; 日期
d = m;
```

枚举是另一种用户自定义类型，其中程序员指定一个变量可以包含的有限符号值集合。例如，如果你在 Rust 中处理不确定的布尔变量，你可能会声明：

```
/ * 这是 Rust 代码 */ enum Fuzzy { True, False, Uncert
ain };
```

在内部，枚举值本质上只是一个整数，但它能使源代码更具可读性，同时也允许编译器防止程序员赋予非法的值。再一次，C 语言允许你声明枚举，但并不阻止你将整数与枚举类型混合使用。

一种语言的复合类型将已有的类型组合在一起，形成更复杂的聚合体。你肯定熟悉结构类型（或记录类型），它把多个值组合成一个更大的整体。例如，你可以把纬度和经度组合在一起，将它们视为一个单一的坐标结构：

```
/* 这是 Go 代码 */ type coordinates
struct { latitude float64 longitude flo
at64 }
```

较少使用的是联合类型，其中多个符号占用同一块内存。例如，在 C 语言中，你可以声明一个数字的联合类型，其中浮点数和整数相互重叠存储：

```
/* 这是 C 代码 */ union num
ber { int i; float f; };



联合体 数字 n；n.i =
10；n.f = 3.14；
```

在这种情况下，n.i 和 n.f 占用同一块内存。如果你将 10 赋给 n.i 并再读回它，你会如预期看到 10。然而，如果你将 10 赋给 n.i 却读取 n.f，取决于这两个值在内存中的具体映射方式，你很可能会看到一个垃圾值。在实现诸如设备驱动程序之类的操作系统特性时，联合类型有时非常有用，因为硬件接口经常为多种用途重复使用同一内存位置。

一些语言提供了一种变体类型，允许程序员显式地描述一个具有多个变体的类型，每个变体都有不同的字段。这类似于联合类型（union type）的概念，但可以防止程序员进行不安全的访问。例如，Rust 允许我们创建一个表示表达式树的变体类型：

```
/* 这是 Rust 代码 */ enum Expression { ADD{ left: Expression, right: Expre
ssion }, MULTIPLY{ left: Expression, right: Expression }, INTEGER{ value
: i32 }, NAME{ name: string } }
```

这种变体类型受到严格控制，因此很难被错误使用。对于类型为 ADD 的 Expression，它具有 left 和 right 字段，可以按预期方式使用。对于类型为 NAME 的 Expression，可以使用 name 字段。除非选择了相应的类型，否则其他字段根本不可用。

最后，我们必须定义当不同类型一起使用时会发生什么。假设将一个整数 i 赋值给一个浮点数 f。当把一个整数作为参数传递给一个期望浮点数的函数时，也会出现类似的情况。在这种情况下，语言可能采取的做法有几种：

- 禁止该赋值。一种非常严格的语言（如 B-Minor）可以直接发出错误并阻止程序编译！也许进行这种赋值本身就毫无意义，而编译器是在将程序员从一次严重的错误中拯救出来。如果确实需要 *really* 赋值，则可以通过要求程序员调用一个内置的转换函数（例如 IntToFloat）来实现，该函数接受一种类型并返回另一种类型。

- 进行按位复制。如果两个变量具有相同的底层存储大小，那么不同类型的赋值可以通过仅将一个变量中的比特复制到另一个变量的位置来完成。这*usually*是个坏主意，因为无法保证一种数据类型在另一种上下文中具有任何意义。但在少数特定情况下确实会这样做，例如在 C 中给不同的指针类型赋值时。

- 转换为等效的值。对于某些类型，编译器可能具有内置转换，会将值隐式地转换为所需的类型。例如，在整数与浮点数之间，或在有符号整数与无符号整数之间进行隐式转换是很常见的。但这并不意味着该操作是安全的！隐式转换可能会丢失信息，从而导致非常棘手的错误。

- 以不同的方式解释该值。在某些情况下，可能需要将该值转换为其他不等价但仍对程序员有用的值。例如，在Perl中，当列表被复制到标量上下文时，列表的*length*被放入目标变量中，而不是列表的内容。

  @days = （"星期一", "星期二", "星期三", ... ）; @a = @days; # 将数组复制到数组 a $b = @days; # 将数组的长度放入 b

7.3 B-Minor 类型系统

B小调类型系统是安全的、静态的和显式的。因此，它相对紧凑，易于描述，易于实现，并消除了大量的编程错误。然而，它可能比某些语言更严格，因此我们必须检测大量错误。

B-Minor 具有以下原子类型:

- 整数 - 一个64位有符号整数。
- boolean - 仅限于符号 true 或 false。
- 字符 - 限制为ASCII值。
- 字符串 - ASCII 值，以空字符终止。
- void - 仅用于不返回任何值的函数。

以及以下复合类型:

- 数组 [大小] 类型
- 函数类型 (a: 类型, b: 类型, ...)

以下是必须执行的类型规则:

- 一个值只能赋给同一类型的变量。
- 函数参数只能接受相同类型的值。
- return 语句的类型必须与函数的返回类型匹配。
- 所有二元运算符的左右操作数必须具有相同的类型。
- 相等运算符 != 和 == 可应用于除 void、数组或函数之外的任何类型，并且始终返回 boolean。

- 比较运算符 < <= >= > 仅可应用于整数值，并且始终返回布尔值。
- 布尔运算符 ! && || 仅能应用于布尔值，并始终返回布尔值。
- 算术运算符 + - * / % ^ ++ -- 只能应用于整数值，并且总是返回整数。

7.4 符号表

符号表记录了我们需要了解的程序中每个已声明变量（以及其他命名项，如函数）的所有信息。表中的每个条目都是一个 struct symbol，如图 7.1 所示。

```
struct symbol {
    symbol_t kind;
    struct type *type;
    char *name;
    int which;
};
```

```
typedef enum {
    SYMBOL_LOCAL,
    SYMBOL_PARAM,
    SYMBOL_GLOBAL
} symbol_t;
```

图 7.1：符号结构

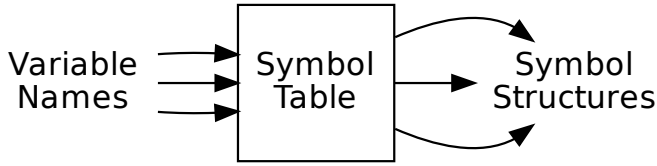
种类字段指示符号是局部变量、全局变量还是函数参数。类型字段指向一个类型结构，表示变量的类型。名称字段给出名称（显然），而“哪一个”字段给出局部变量和参数的序号位置。（稍后会详细说明。）

与我们迄今为止创建的所有其他数据结构一样，我们必须拥有一个像这样的工厂函数：

struct 符号 * symbol_create(symbol_t 种类, struct 类型 *type, char *名称) { struct 符号 *s = malloc(sizeof(*s)); s->种类 = 种类; s->类型 = 类型; s->名称 = 名称; 返回 s; }

要开始语义分析，我们必须为每个变量声明创建一个合适的符号结构，并将其输入符号表中。

从概念上讲，符号表只是一个映射，将每个变量的名称与描述它的符号结构对应起来：



然而，事情并没有 *quite* 那么简单，因为大多数编程语言允许同一个变量名被多次使用，只要每个定义位于不同的作用域中。在类 C 的语言（包括 B-Minor）中，存在全局作用域、函数参数和局部变量的作用域，以及在每个出现花括号的地方形成的嵌套作用域。

例如，下面的 B-Minor 程序将符号 x 定义了三次，每次都有不同的类型和存储类。运行时，程序应当打印 10 hello false。

```
x：整数 = 10； f：函数 空 ( x：字符串 ) = {
打印 x， "\n"; { x：布尔 = 假； 打印 x， "\n"
; } }主函数：函数 空 () = { 打印 x， "\n"; f("
你好"); }
```

为适应这些多重定义，我们将把符号表构造成一组哈希表的栈，如图 7.2 所示。每个哈希表将给定作用域中的名称映射到其对应的符号。这使得同一个符号（如 x）可以在多个作用域中存在而不会发生冲突。随着程序的执行，每当进入一个作用域时，我们就压入一个新表；每当离开一个作用域时，我们就弹出一个表。

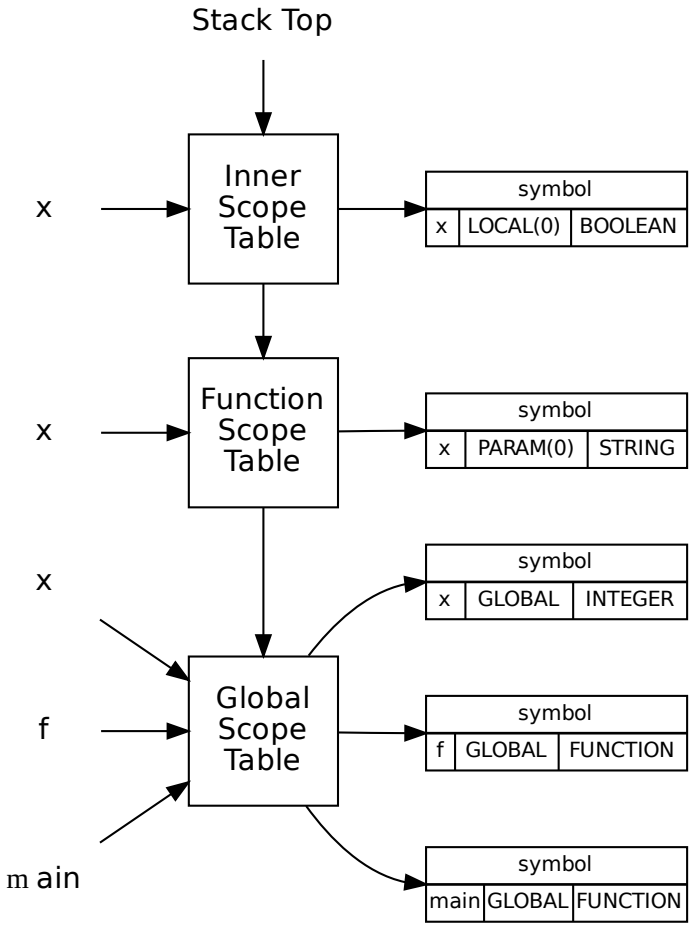


图 7.2：嵌套符号表

```
void scope_enter(); void sco
pe_exit(); int scope_level();


void scope_bind( const char *name, struct symbol *sym ); struct symbol *scope_lo
okup( const char *name ); struct symbol *scope_lookup_current( const char *name
);
```

图 7.3：符号表 API

为了操作符号表，我们在图 7.3 所示的 API 中定义了六种操作。它们具有以下含义：

- scope enter() 会在栈顶压入一个新的哈希表，表示一个新的作用域。
- scope exit() 会导致移除最顶层的哈希表。
- scope level() 返回当前栈中哈希表的数量。（这有助于判断我们是否处于全局作用域。）
- scope bind(name,sym) 会向栈顶的哈希表添加一个条目，将 name 映射到符号结构 sym。
- scope lookup(name) 从顶部到底部搜索哈希表栈，查找第一个与 name 完全匹配的条目。如果未找到匹配项，则返回 null。
- 作用域查找 current(name) 的工作方式与作用域查找相同，不同之处在于它只搜索最顶层的表。它用于确定某个符号是否已经在当前作用域中被定义。

7.5 名称解析

在符号表就绪之后，我们现在可以将每一次变量名的使用与其对应的定义进行匹配。这个过程称为名称解析。为实现名称解析，我们将为 AST 中的每一种结构编写一个 resolve 方法，包括 decl resolve()、stmt resolve() 等。

总体而言，这些方法必须遍历整个 AST，查找变量声明和使用的位置。凡是声明变量之处，都必须将其录入符号表，并将符号结构链接到 AST 中。凡是使用变量之处，都必须在符号表中进行查找，并将符号结构链接到 AST 中。当然，如果在同一作用域内对某个符号进行了重复声明，或者在未声明的情况下使用了符号，则必须发出相应的错误信息。

我们将从声明开始，如图 7.4 所示。每个 decl 表示某种形式的变量声明，因此 decl resolve 会创建一个新的符号，然后将其绑定到当前作用域中该声明的名称。如果该声明表示一个表达式（d->value 不为 null），则应解析该表达式。如果该声明表示一个函数（d->code 不为 null），那么我们必须创建一个新的作用域，并解析其参数和代码。

图 7.4 给出了一些用于解析声明的示例代码。与本书一贯做法一样，请将这段起始代码作为理解基本思路的参考。你需要进行一些修改，以适配该语言的所有特性、干净地处理错误，等等。

以类似的方式，我们必须为 AST 中的每一种结构编写 resolve 方法。stmt resolve()（未展示）只需对其每个子组件调用相应的 resolve。在 STMT BLOCK 的情况下，它还必须进入并离开一个新的作用域。param list resolve()（同样未展示）必须为函数的每个参数进入一个新的变量声明，使这些定义对函数的代码可用。

要在整个 AST 上执行名称解析，你只需在 AST 的根节点上调用一次 decl resolve()。该函数将通过调用必要的子函数遍历整棵树。

```
void decl_resolve( 结构 decl *d ) { 如果(!d) 返回; symbol_t kind = scope_le
vel() > 1 ? SYMBOL_LOCAL : SYMBOL_GLOBAL; d->symbol = symbol
_create(kind,d->type,d->name); expr_resolve(d->value); scope_bind(d->nam
e,d->symbol); 如果(d->code) { scope_enter(); param_list_resolve(d->type->
params); stmt_resolve(d->code); scope_exit(); }decl_resolve(d->next); }
```

图 7.4：声明的名称解析

```
void 表达式_解析( 结构体 表达式 *e ) { 如果(!e) 返回; 如
果( e->种类==表达式_名称 ) { e->符号 = 作用域_查找(e->
名称); } 否则 { 表达式_解析( e->左 ); 表达式_解析( e->右 )
; } }
```

图 7.5：表达式的名称解析

7.6 实现类型检查

在检查表达式之前，我们需要一些用于检查和操作类型结构的辅助函数。下面是用于检查相等性、复制以及删除类型的伪代码：

布尔类型 type_equals(结构体 type *a, 结构体 type *b) { 如果(a->kind == b->kind) { 如果(a 和 b 是原子类型){ 返回 true } 否则如果(都是数组) { 如果子类型递归相等则返回 true } 否则如果(都是函数) { 如果子类型和参数递归相等则返回 true } } 否则 { 返回 false } }

struct type * type_copy(struct type *t) { 返回 t 的一个重复副本，确保递归地复制子类型和参数。 }void type_delete(struct type *t) { 递归地释放 t 的所有元素。 }

接下来，我们构造一个函数 expr typecheck，用于计算表达式的正确类型并返回它。为了简化代码，我们断言，当 expr typecheck 被调用于一个非空的表达式时，它将始终返回一个新分配的类型结构。如果表达式包含无效的类型组合，则 expr typecheck 会打印出错误，但仍返回一个有效的类型，以便编译器能够继续执行并尽可能多地找到错误。

一般方法是执行表达式树的递归后序遍历。在树的叶子节点处，节点的类型直接对应于表达式节点的类型：整数字面量具有整数类型，字符串字面量具有字符串类型，依此类推。如果遇到一个变量名，则可以通过跟踪符号指针来确定其类型。

到符号结构，其中包含类型。此类型被复制并返回到父节点。

对于表达式树的内部节点，我们必须比较左右子树的类型，并确定它们是否与第7.3节中指示的规则兼容。如果不兼容，我们将发出错误消息并增加全局错误计数器。无论如何，我们返回操作符的适当类型。左右分支的类型不再需要，可以在返回之前删除。

这是基本的代码结构：

```cstruct type * expr_typecheck(struct expr *e) { if(!e) return 0; struc t type *lt = expr_typecheck(e->left); struct type *rt = expr_typecheck(e-> right); struct type *result; switch(e->kind) { case EXPR_INTEGE R_LITERAL: result = type_create(TYPE_INTEGER,0,0); br eak; case EXPR_STRING_LITERAL: result = type_create(TY PE_STRING,0,0); break; /* 更多情况在这里 */ } type_de lete(lt); type_delete(rt); return result; }```
```

让我们详细考虑几种运算符的情况。算术运算符只能应用于整数，并且始终返回整数类型：

案例 EXPR_ADD: 如果(lt->kind!=TYPE_INTEGER || rt->kind!=TYPE_INTEGER) { /* 显示错误 */ }结果 = type_create(TYPE_INTEGER,0,0); break;

等式运算符可以应用于大多数类型，只要两边的类型相等。这些运算符总是返回布尔值。

案例 EXPR_EQ: 案例 EXPR_NE: 如果(!type_equals(lt, rt)) { /* 显示错误 */ }如果(lt->kind==TYPE_VOID || lt->kind==TYPE_ARRAY || lt->kind==TYPE_FUNCTION) { /* 显示错误 */ }结果 = type_create(TYPE_BOOLEAN, 0, 0); 终止;

像 a[i] 这样的数组解引用要求 a 是一个数组，i 是一个整数，并返回数组的子类型：

case EXPR_DEREF: if(lt->kind==TYPE_ARRAY) { if(rt->kind!=TYPE_INTEGER) { /* 错误：索引不是整数 */ }result = type_copy(lt->subtype); } else { /* 错误：不是数组 */ /* 但我们需要返回一个有效的类型 */ result = type_copy(lt); }break;

类型检查中的大部分繁重工作是在 expr typecheck 中完成的，但我们仍然需要为声明、语句以及 AST 的其他元素实现类型检查。decl typecheck, stmt typecheck

而其他的类型检查方法只是遍历 AST，计算表达式的类型，然后根据需要将它们与声明和其他约束进行检查。

例如，decl typecheck 只是确认变量声明与其初始化器相匹配，并在其他情况下对函数声明的主体进行类型检查：

```
void decl_typecheck( struct decl *d ) { if(d->value) { struct type *t
; t = expr_typecheck(d->value); if(!type_equals(t,d->symbol->type
)) { /* 显示一个错误 */ } }if(d->code) { stmt_typecheck(d->code)
; } }
```

语句必须通过对其各个组成部分进行求值来进行类型检查，然后在需要的地方验证类型是否匹配。类型一旦被检查，就不再需要，可以被删除。例如，if-then 语句要求控制表达式具有布尔类型：

```
void stmt_typecheck( struct stmt *s ) { struct type *t; switch(
s->kind) { case STMT_EXPR: t = expr_typecheck(s->expr);
type_delete(t); break; case STMT_IF_THEN: t = expr_typec
heck(s->expr); if(t->kind!=TYPE_BOOLEAN) { /* 显示一
个错误 */ }type_delete(t); stmt_typecheck(s->body); stmt_ty
pecheck(s->else_body); break; /* 此处还有更多情况 */ } }
```

7.7 错误信息

编译器通常以显示糟糕的错误信息而臭名昭著。幸运的是，我们已经开发了足够的代码结构，可以直接显示一条信息充分的错误消息，准确说明发现了哪些类型，以及问题究竟出在哪里。

对于例如，这段 B-Minor 代码有一团类型问题 引理:

```
s: 字符串 = "hello"; b: 布尔 = fal
se; i: 整数 = s + （b<5）;
```

大多数编译器都会输出这样一条毫无帮助的消息：

```
错误：表达式中的类型兼容性
```

但是，你的项目编译器可以非常容易地提供像这样的更为详细的错误信息：

```
错误：无法将布尔值 (b) 与整数 (5) 进行比较 错误：无法将布尔值 (b<5) 与字符串 (s) 相加
```

当发现问题时，只需要在打印出所涉及的每个表达式和类型时多加一些注意即可：

```
printf("错误：无法将一个 "); type_print(lt);
printf(" ("); expr_print(e->left); printf(") 添加
到一个 "); type_print(rt); printf(" ("); expr_pri
nt(e->right); printf(")\n");
```

7.8 练习

1. 在 symbol.c 和 scope.c 中实现符号和作用域函数，以现有的哈希表实现作为起点。

2. 通过编写 stmt resolve() 和 param list resolve() 以及任何其他所需的支持代码，完成名称解析代码

3. 修改 decl resolve() 和 expr resolve()，在同名被重复声明，或在未声明的情况下使用变量时显示错误。4. 完成 expr typecheck 的实现，使其能够检查并返回所有类型表达式的类型。5. 完成 stmt typecheck 的实现，针对每一种语句强制执行相应的约束。6. 编写一个函数 myprintf，用于显示类似 printf 风格的格式字符串，但支持诸如 %T 表示类型、%E 表示表达式等符号。这将使输出错误消息更加容易，例如这样：

```
myprintf( "错误：无法将一个 %T (%E) 加到一个 %T (%E)\n", lt,e->left,rt,e->right );
```

查阅标准的 C 语言手册，了解 stdarg.h 头文件中用于创建可变参数函数的各个函数。

7.9 延伸阅读

1. H. Abelson、G. Sussman 和 J. Sussman，《计算机程序的构造和解释》，麻省理工学院出版社，1996 年。

2. B. Pierce，《类型与程序设计语言》，MIT出版社，2002年。

3. D. Friedman 和 D. Christiansen，《The Little Typer》，MIT 出版社，2018 年。

第8章——中间表示

8.1 引言

大多数生产级编译器都会使用一种中间表示（IR），它位于源语言的抽象结构与目标汇编语言的具体结构之间。

IR 被设计为具有简单而规整的结构，以便于优化、分析以及高效的代码生成。模块化编译器通常会将每一种优化或分析工具实现为独立的模块，这些模块都使用并生成同一种 IR，从而可以轻松地以不同的顺序选择和组合各种优化。

IR 通常具有一种已定义的外部格式，可以以文本形式写入文件，从而在彼此无关的工具之间进行交换。尽管对有决心的程序员来说它可能是可见的，但通常并不打算让人轻松阅读。加载到内存中时，IR 被表示为一种数据结构，以便于遍历其结构的算法。

可以使用的 IR 有许多不同类型；有些与我们目前为止使用的 AST 非常接近，而另一些则与目标汇编语言只有很短的距离。有些编译器甚至使用多个 IR，并按抽象层次递减进行组织。在本章中，我们将考察不同的 IR 方法，并讨论它们的优点和缺点。

8.2 抽象语法树

首先，我们将指出，如果目标只是发射汇编语言而不进行大量优化或其他变换，那么 AST 本身就可以作为一种可用的 IR。完成类型检查之后，可以直接在 AST 上应用一些简单的优化，例如强度削弱和常量折叠。然后，为了生成汇编语言，你只需对 AST 进行一次后序遍历，并为每个节点发射几条对应的汇编指令。[1]

[1]This is the approach we use in a one-semester course to implement a project compiler, since there is a limited amount of time to get to the final goal of generating assembly language.

```
typedef enum { DAG_ASSIGN,
DAG_DEREF, DAG_IADD, D
AG_IMUL, ... DAG_NAME, D
AG_FLOAT_VALUE, DAG_IN
TEGER_VALUE } dag_kind_t;

struct dag_node { dag_kind_t kind; struct d
ag_node *left; struct dag_node *right; unio
n {const char *name; double float_value; i
nt integer_value; } u; };
```

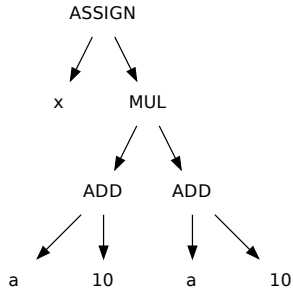
图 8.1：示例 DAG 数据结构定义

然而，在一个生产编译器中，AST 并不是一个很好的 IR 选择，主要因为其结构是 *too* 丰富的。每个节点都有大量不同的选项和子结构：例如，一个加法节点可以表示整数加法、浮点加法、布尔或运算或字符串连接，具体取决于涉及的值的类型。这使得执行稳健的转换以及生成外部表示变得困难。需要一种更低级别的表示。

8.3 有向无环图

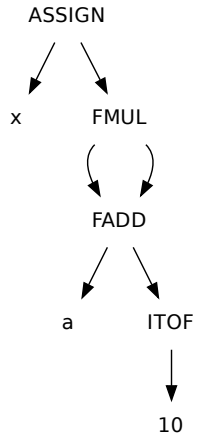
有向无环图（DAG）是在 AST 的基础上进一步简化的一步。DAG 与 AST 相似，不同之处在于它可以具有任意的图结构，并且各个节点被大幅简化，使得除了每个节点的类型和值之外，几乎没有或完全没有辅助信息。这要求我们拥有更多数量的节点类型，并且每一种类型都明确其用途。例如，图 8.1 展示了一个与我们的项目编译器兼容的 DAG 数据结构定义。

现在假设我们编译一个简单的表达式，比如 x=（a+10）*(a+10)。该表达式的 AST 表示将直接捕捉其语法结构：

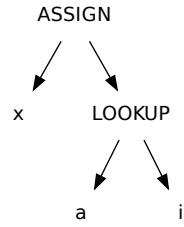


在执行类型检查后，我们可能会得知 a 是一个浮点值，因此 10 必须在执行浮点运算之前转换为浮点数。此外，计算 a+10 只需执行一次，结果值可以使用两次。

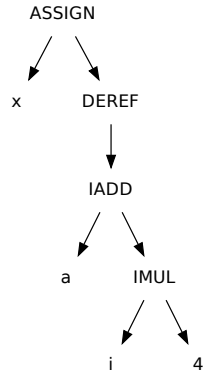
所有这些可以通过以下DAG表示，该DAG引入了一种新的节点类型ITOF，用于执行整数到浮点的转换，以及节点FADD和FMUL，用于执行浮点算术运算：



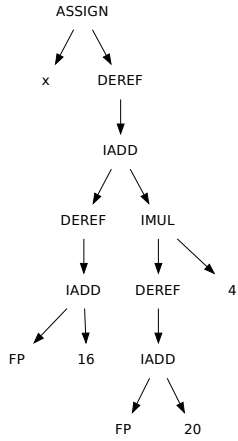
DAG 也常用于更详细地表示与指针和数组相关的地址计算，以便在可能的情况下进行共享和优化。例如，数组查找 x=a[i] 在 AST 中的表示将非常简单：



然而，数组查找实际上是通过将数组 a 的起始地址与元素索引 i 乘以数组中对象大小（该大小通过查阅符号表确定）相加来完成的。这可以用如下所示的 DAG 来表示：



作为代码生成的最后一步，DAG 可能会扩展以包括局部变量的地址计算。例如，如果 a 和 i 分别存储在栈上，距离帧指针 FP 分别为十六字节和二十字节，则 DAG 可以像这样扩展：



值编号方法可用于从 AST 构建 DAG。其思想是构建一个数组，其中每个条目由一个 DAG 节点类型以及其子节点的数组索引组成。每当我们希望向 DAG 添加一个新节点时，就在数组中搜索是否存在匹配的节点，并复用它以避免重复。DAG 通过对 AST 进行后序遍历，并将每个元素加入数组来构建。

上述的 DAG 可以用这个值编号数组来表示：

#	Type	Left	Right	Value
0	NAME			x
1	NAME			a
2	INT			10
3	ITOF	2		
4	FADD	1	3	
5	FMUL	4	4	
6	ASSIGN	0	5	

显然，每次新增一个节点时都在表中搜索等价节点会具有多项式复杂度。然而，只要每个单独的表达式都有其自己的 DAG，其绝对规模仍然保持相对较小。

通过将 DAG 表示设计为把所有必要的信息编码到节点类型中，就可以很容易地编写一种可移植的外部表示。例如，我们可以将每个节点表示为一个符号，后面用括号列出其子节点：

```
ASSIGN(x,DEREF(IADD(DEREF(IADD(FP,16)), IMUL(DEREF(IAD
D(FP,20)),4))))
```

显然，这类代码对人类来说并不容易手工阅读和编写，但它非常容易打印、也非常容易解析，从而便于在编译器各个阶段之间传递，用于分析和优化。

那么，你可以对 DAG 做哪些优化呢？一种简单的优化是常量折叠。这是将仅由常量组成的表达式归约为单一值的过程。[2] 这一能力很实用，因为程序员可能出于可读性或可维护性的考虑，希望将某些表达式保留为显式形式，同时仍然在可执行代码中获得单个常量所带来的性能优势。

DAG Constant Folding Algorithm
Examine a DAG recursively and collapse all operators on two constants into a single constant.

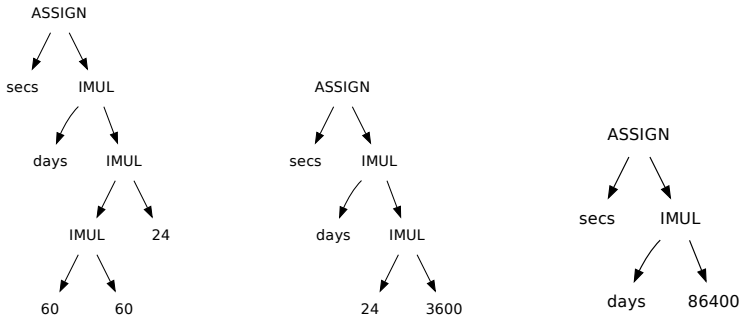
```
ConstantFold( DagNode n ):

If n is a leaf:
    return;
Else:
    n.left = ConstantFold(n.left);
    n.right = ConstantFold(n.right);

    If n.left and n.right are constants:
        n.value = n.operator(n.left,n.right);
        n.kind = constant;
        delete n.left and n.right;
```

[2] Constant folding is a narrow example of the more general technique of **partial execution** in which some parts of the program are executed at compile time, while the rest is left for runtime.

图 8.2：常量折叠示例



假设你有一个表达式，用于计算给定天数中包含的秒数。程序员将其表示为 secs=days*24*60*60，以清楚地表明一天有 24 小时，一小时有 60 分钟，一分钟有 60 秒。图 8.2 展示了 ConstantFold 如何约简该 DAG。该算法沿着树向下遍历，将 IMUL(60,60) 合并为 3600，然后再将 IMUL(3600,24) 合并为 86400。

8.4 控制流图

需要注意的是，DAG 本身适合用于编码表达式，但对于控制流或其他有序的程序结构并不那么有效。公共子表达式是在这样的假设下被合并的：它们可以以任意顺序求值（只要符合运算符优先级），并且 DAG 中已有的值不会发生变化。当我们考虑会修改值的多条语句，或会重复或跳过语句的控制流结构时，这一假设就不再成立。

为此，我们可以使用控制流图来表示程序的高层结构。控制流图是一个有向图（可能包含环），其中图中的每个节点由一个顺序语句的基本块组成。图中的边表示基本块之间可能的控制流。条件构造（如 if 或 switch）会在图中产生分支，而循环构造（如 for 或 while）会产生回边。

例如，这段代码：

```
for(i=0;i<10;i++) { if(i%2==0)
{ print "偶数"; } else { print "
奇数"; }print "\n"; }返回;
```

将会得到如下控制流图：

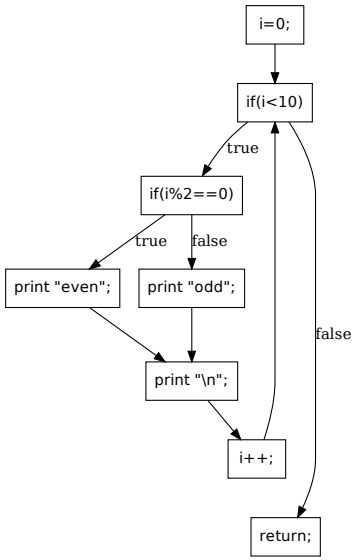


图 8.3：示例控制流图

请注意，控制流图的结构与 AST 不同。用于 for 循环的 AST 会将每个控制表达式作为循环节点的直接子节点，而控制流图则按它们在实际执行中的顺序放置每一个。同样，if 语句从条件的每个分支到后续节点都有边，因此可以轻松地从一个组成部分追踪到下一个的执行流程。

8.5 静态单赋值形式

静态单赋值（SSA）[1] 形式是一种常用于复杂优化的表示。SSA 利用控制流中的信息，并为每个基本块引入一个新的约束：*variables cannot change their values*。相反地，每当一个变量被赋予新值时，都会给它一个新的版本号。

例如，假设我们有这样一段代码：

int x = 1; int a = x; int
b = a + 10; x = 20 * b;
x = x + 30;

我们可以像这样将其重写为 SSA 形式：

int x_1 = 1; int a_1 = x_1; i
nt b_1 = a_1 + 10; x_2 = 20
* b_1; x_3 = x_2 + 30;

当变量在条件的两个分支中赋予不同的值时，出现了一种特殊情况。在条件判断之后，变量可能具有任一值，但我们不知道是哪一个。为了表示这一点，我们引入了一个新函数 $\phi(x, y)$，它表示在运行时可能选择 x 或 y 中的任意一个值。ϕ 函数不一定会转化为汇编输出中的指令，但它的作用是将新值与其可能的旧值连接起来，反映控制流图。

例如，这个代码片段：

如果(y<10) {
x=a; } 否则 {
x=b; }

Got it! Please provide the source text that you'd like me to translate into Chinese.

如果(y_1<10) { x_2=a; }
否则 { x_3=b; }x_4 = phi(x
_2,x_3);

8.6 线性红外

线性 IR 是一组有序的指令序列，接近汇编语言的最终目标。它失去了 DAG（它不承诺特定顺序）的某些灵活性，但可以在一个数据结构中捕捉表达式、语句和控制流。这使得跨多个表达式的一些优化技术成为可能。

没有通用的线性中间表示（IR）标准。线性中间表示通常像一个理想化的汇编语言，具有大量（或无限）寄存器以及常见的算术和控制流操作。在这里，我们假设一个中间表示，其中LOAD和STOR用于在内存和寄存器之间移动值，三地址算术操作读取两个寄存器并将结果写入第三个寄存器，按从左到右的顺序。我们的示例表达式如下所示：

1. 加载 a -> %r1 2. 加载 $10 ->
%r2 3. ITOF %r2 -> %r3 4. FA
DD %r1, %r3 -> %r4 5. FMUL
%r4, %r4 -> %r5 6. 存储 %r5 ->
x

这个IR易于高效存储，因为每条指令可以是一个固定大小的4元组，表示操作和最多三个参数的值。外部表示也很简单。

如示例所示，最方便的做法是假设存在无限数量的虚拟寄存器，使得每个新计算的值都写入一个新的寄存器。在这种形式下，我们可以通过观察寄存器写入的第一个点和寄存器使用的最后一个点，轻松识别一个值的生命周期。在这两点之间，寄存器一个的值必须保持。例如，%r1 的生命周期是从指令 1 到指令 4。

在任何给定的指令中，我们还可以观察到当前活跃的虚拟寄存器集合：

1. 加载 a -> %r1 活跃: %r1 2. 加载 $10 -> %r2 活跃: %r1 %r2 3
. ITOF %r2 -> %r3 活跃: %r1 %r2 %r3 4. FADD %r1, %r3 -> %r
4 活跃: %r1 %r3 %r4 5. FMUL %r4, %r4 -> %r5 活跃: %r4 %r5
6. 存储 %r5 -> x 活跃: %r5

这个观察使得执行与指令排序相关的操作变得简单。只要指令读取的值没有被移动到它们的定义之上，任何指令都可以移到前面的位置（在同一个基本块内）。同样，任何指令也可以移到后面的位置。

只要它写入的值没有被移到它们的使用位置下面。移动指令可以减少代码生成中所需的物理寄存器数量，同时也能减少流水线架构中的执行时间。

8.7 栈式机器 IR

一种更加紧凑的中间表示是栈机IR。这种表示旨在执行在一个虚拟栈机上，栈机没有传统的寄存器，而只有一个栈来存储中间寄存器。栈机IR通常具有一个PUSH指令，用于将变量或字面值推入栈中，还有一个POP指令，用于移除栈中的项并将其存储到内存中。二进制算术运算符（如FADD或FMUL）隐式地从栈中弹出两个值，并将结果推入栈中，而一元运算符（ITOF）则弹出一个值并推入一个值。还需要一些实用指令来操作栈，例如COPY指令，用于将一个副本值推入栈中。

为了从 DAG 生成栈式机器的 IR，我们只需对 AST 进行后序遍历：对每个叶子值发出一条 PUSH 指令，对每个内部节点发出一条算术指令，并发出一条 POP 指令将值赋给变量。

我们的示例表达式在栈机器 IR 中看起来会是这样的：

压栈 a 压
栈 10 整
数转浮点
浮点加法
复制 浮点
乘法 出栈
x

如果我们假设 a 的值为 5.0，那么直接执行 IR 将会得到如下结果：

IR Op:	`PUSH a`	`PUSH 10`	`ITOF`	`FADD`	`COPY`	`FMUL`	`POP x`
Stack	5.0	10	10.0	15.0	15.0	225.0	-
State:	-	5.0	5.0	-	15.0	-	-

栈式机器 IR 具有许多优势。由于无需记录寄存器的细节，它比三元组或四元组的线性表示要紧凑得多。此外，在一个简单的解释器中实现这种语言也很容易。

然而，基于栈的 IR 略微更难翻译成传统的基于寄存器的汇编语言，正是因为显式的寄存器名称已经丢失。对这种形式进行进一步的变换或优化，要求我们将栈式基本 IR 中隐含的信息依赖关系重新转换回一种更显式的形式，例如 DAG，或带有显式寄存器名称的线性 IR。

8.8 示例

几乎每一种编译器或语言都有其自身的中间表示，并带有一些独特的局部特性。为了让你了解可能性，本节比较了2017年编译器中使用的三种不同的 IR。对于每一种，我们将展示编译这个简单算术表达式的输出：

浮点 f(整数 a, 整数 b, 浮点 x) { 浮点 y = a*x*x
+ b*x + 100; 返回 y; }

8.8.1 GIMPLE - GNU Simple Representation

GNU 简单表示（GIMPLE）是 GNU C 编译器在最早阶段使用的一种内部 IR。GIMPLE 可以被看作是 C 语言的一种大幅简化形式，其中所有表达式都被拆分为在静态单赋值（SSA）形式下对值进行的单个运算符。允许基本的条件判断，循环则通过 goto 来实现。

我们的简单函数生成了以下 GIMPLE。请注意，每个 SSA 值都被声明为局部变量（具有长名称），并且每个运算符的类型仍然是从局部类型声明中推断出来的。

f (int a, int b, float x) { float D.1597
D.1597; float D.1598D.1598; float D
.1599D.1599; float D.1600D.1600; fl
oat D.1601D.1601; float D.1602D.16
02; float D.1603D.1603; float y;

D.1597D.1597 = （浮点数）a; D.1598D.1598 = D.1597D.1597 * x; D.1599D.1599 = D.1598D.1598 * x; D.1600D.1600 = （浮点数）b; D.1601D.1601 = D.1600D.1600 * x; D.1602D.1602 = D.1599D.1599 + D.1601D.1601; y = D.1602D.1602 + 1.0e+2; D.1603D.1603 = y; 返回 D.1603D.1603;

}

8.8.2 LLVM - Low-Level Virtual Machine

低级虚拟机（LLVM）[3] 项目是一种语言以及一套用于构建优化型编译器和解释器的相应工具。多种编译器前端支持生成 LLVM 中间代码，该中间代码可以由多种独立工具进行优化，然后再被转换为本地机器代码，或转换为诸如 Oracle 的 JVM 或 Microsoft 的 CLR 等虚拟机的字节码。

我们的简单函数生成了如下的 LLVM。请注意，最开始的几个 alloca 指令为局部变量分配空间，随后是 store 指令，将参数移动到局部变量中。然后，表达式的每一步都以 SSA 形式进行计算，并将结果存储到局部变量 y 中。代码在每一步都明确标注了类型（32 位整数或浮点数）以及每个值的对齐方式。

定义 float @f(i32 %a, i32 %b, float %x) #0 { %1 = 分配 i32，对齐 4 %2 = 分配 i32，对齐 4 %3 = 分配 float，对齐 4 %y = 分配 floa t，对齐 4 存储 i32 %a，i32* %1，对齐 4 存储 i32 %b，i32* %2 ，对齐 4 存储 float %x，float* %3，对齐 4 %4 = 加载 i32* %1， 对齐 4 %5 = 将 i32 %4 有符号转换为 float %6 = 加载 float* %3 ，对齐 4 %7 = 浮点乘法 float %5，%6 %8 = 加载 float* %3，对 齐 4 %9 = 浮点乘法 float %7，%8 %10 = 加载 i32* %2，对齐 4 %11 = 将 i32 %10 有符号转换为 float %12 = 加载 float* %3，对 齐 4 %13 = 浮点乘法 float %11，%12 %14 = 浮点加法 float %9 ，%13 %15 = 浮点加法 float %14，1.000000e+02 存储 float %15 ，float* %y，对齐 4 %16 = 加载 float* %y，对齐 4 返回 float %1 6 }

[3] http://llvm.org

8.8.3 JVM - Java Virtual Machine

Java 虚拟机（JVM）是对一种基于栈的机器的抽象定义。用 Java 编写的高级代码会被编译为 .class 文件，其中包含 JVM 字节码的二进制表示。JVM 的最早实现是解释器，它们以直观的方式读取并执行 JVM 字节码。后来的实现对字节码进行即时（JIT）编译，将其转换为本地汇编语言，从而可以直接执行。

我们的简单函数生成了如下 JVM 字节码。注意，每条 iload 指令都引用一个局部变量，其中参数被视为最前面的几个局部变量。因此，iload 1 会将第一个局部变量（int a）压入栈中，而 fload 3 会将第三个局部变量（float x）压入栈中。固定常量存储在类文件中的一个数组里，并按位置进行引用，因此 ldc #2 会将位置为 2 的常量（100）压入栈中。

源文本： 0: iload
1 1: i2f 2: fload 3 4:
fmul 5: fload 3 7: fm
ul 8: iload 2 9: i2f 10
: fload 3 12: fmul 13
: fadd 14: ldc #2 16:
fadd 17: fstore 4 19:
fload 4 21: freturn 翻
译文本：

8.9 习题

1. 在你的编译器中添加一个步骤，通过执行后序遍历将 AST 转换为 DAG，并为每个 AST 节点创建一个或多个对应的 DAG 节点。

2. 编写代码，将一个 DAG 导出为本章所示的简单外部表示形式。适当地扩展该 DAG，以表示控制流结构和函数定义。

3. 编写一个扫描器和解析器来读取 DAG 的外部表示，并将其重建为数据结构。仔细思考该中间表示的文法类别，并选择最简单且可行的实现方式。

4. 在步骤 2 和 3 的基础上，编写一个独立的优化工具，读取 DAG 格式，执行诸如常量折叠之类的简单优化，并以相同的格式将 DAG 写回。

8.10 延伸阅读

1. R. Cytron、J. Ferrante、B. Rosen、M. Wegman 和 F. Kenneth Zadeck。“高效计算静态单赋值形式及控制依赖图。”《ACM 编程语言与系统汇刊（TOPLAS）》第 13 卷，第 4 期，1991 年。https://doi.org/10.1145/115372.115320

2. J. Merrill，《Generic 和 GIMPLE：用于整个函数的新树表示》。GCC 开发者峰会，2003年。

3. C. Lattner 和 V. Adve，“LLVM：用于终身程序分析与变换的编译框架”，IEEE 代码生成与优化国际研讨会，2004 年。

https://dl.acm.org/citation.cfm?id=977673

第9章——存储器组织

9.1 引言

在深入讨论中间代码向汇编语言的翻译之前，我们必须先讨论正在运行的程序其内部内存是如何布局的。尽管一个进程可以按照自己喜欢的任何方式使用内存，但已经形成了一种约定，将程序的各个区域划分为逻辑段，每个段都具有不同的内部管理策略。

9.2 逻辑分段

传统程序将内存视为一系列线性的字（word），每个字都有一个从零开始的数值地址，并一直递增到某个很大的数（例如，在 32 位处理器上为 4GB）。

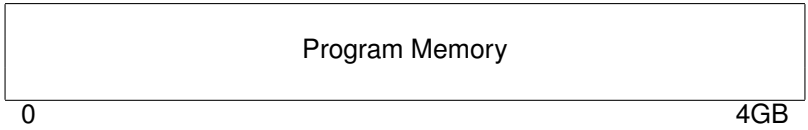


图 9.1：平坦内存模型

原则上，CPU 可以以其认为合适的任何方式使用内存。代码和数据可以按照任何方便的顺序分散并交错地分布在内存中。从技术上讲，CPU 甚至可以在运行时修改包含其代码的内存。不言而喻，以这种方式进行编程将会复杂、令人困惑且难以调试。

相反，程序内存通常通过将其划分为逻辑段来进行布局。每个段都是一个连续的地址范围，专门用于程序中的特定用途。这些段通常按以下顺序进行布局：

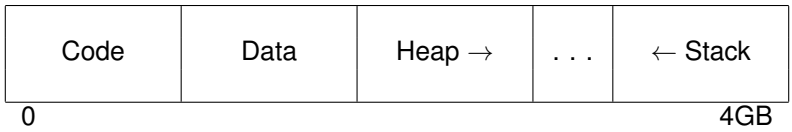


图 9.2：逻辑段

- 代码段（也称为文本段）包含程序的机器码，对应于 C 程序中各个函数的函数体。

- 数据段包含程序的全局数据，对应于C程序中的全局变量。数据段还可以进一步细分为读写数据（变量）和只读数据（常量）。

- 堆段包含堆，堆是在运行时由 C 程序中的 malloc 和 free，或其他语言中的 new 和 delete 动态管理的内存区域。堆的顶部在历史上被称为 break（程序断点）。

- 堆栈段包含堆栈，记录程序的当前执行状态以及当前使用的局部变量。

通常，堆从较低地址向较高地址“向上”增长，而栈则从较高地址向较低地址“向下”增长。在这两个段之间是一个无效的内存区域，直到被其中一个段占用为止。

在简单的计算机上，比如嵌入式机器或微控制器，逻辑段不过是一种组织约定：没有什么能阻止程序不当使用内存。如果堆积增长过大，它可能会与栈段发生冲突（反之亦然），程序会崩溃（如果运气好的话），或者遭遇静默的数据损坏（如果运气不好）。

在使用多程序设计和内存保护的操作系统的计算机上，情况更好。操作系统中运行的每个进程都有自己的私有内存空间，给人一种从地址零开始并扩展到高地址的假象。因此，每个进程可以任意访问自己的内存，但无法访问或修改其他进程的内存。在自己的内存空间内，每个进程都会布局自己的代码、数据、堆和栈段。

在某些操作系统中，当程序初次加载到内存时，会根据其用途为每个内存范围设置权限：每个段对应的内存可以设置为适当的权限。

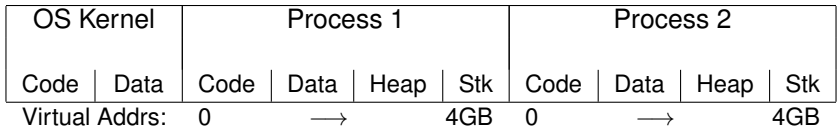


图 9.3：多程序内存布局

最近：数据、堆和栈为读写；常量为只读；代码为读执行；未使用区域为无。

逻辑段上的权限也保护进程免受某些方式的自我损害。例如，代码在运行时无法被修改，因为它被标记为只读/可执行，而堆上的项不能被执行，因为它们被标记为读/写。（为了明确，这仅仅防止意外，而不是恶意行为；程序始终可以请求操作系统更改其某一页面的权限。举个例子，可以查阅Unix中的mprotect系统调用。）

如果一个进程试图以禁止的方式访问内存或试图访问未使用的区域，就会发生页面错误。这会迫使控制权转交给操作系统，操作系统会检查进程和出错的地址。如果该访问表明程序的逻辑分段存在违规，则进程会被强制终止，并显示错误信息“分段错误”。[1]

最初，进程为堆段分配了一小部分内存，它内部管理这些内存以实现 malloc 和 free。如果该区域耗尽且程序仍然需要更多内存，它必须显式地向操作系统请求。在传统的 Unix 系统中，这是通过 brk 系统调用完成的，该调用请求将堆段扩展到一个新地址。如果操作系统同意，它将会在无效区域的开始处分配新的页面，从而有效地扩展堆段。如果操作系统不同意，brk 将返回错误代码，导致 malloc 返回错误（表示为空指针），程序必须处理这个错误。

堆栈有一个类似的问题，即它必须能够向下增长。程序很难确定何时需要更多的堆栈空间，因为每当调用一个新函数或分配新的局部变量时，都会发生这种情况。现代操作系统通过在无效区域的顶部，紧邻当前堆栈的位置，维护一个保护页。当进程试图将堆栈扩展到无效区域时，就会发生页面错误，控制权会转移到操作系统。如果操作系统看到发生错误的地址位于保护页中，它可以简单地为堆栈分配更多页面，适当地设置页面权限，并将保护页移到新的无效区域顶部。

当然，堆和栈的增长是有上限的；

[1] And now you understand this mysterious Unix term!

每个操作系统都实现了控制任何进程或用户可以消耗多少内存的策略。如果违反了这些策略，操作系统可能会拒绝扩展进程的内存。

将程序分成段的想法是如此强大和有用，以至于在许多年代里，硬件中常常实现这一概念。（如果你上过计算机架构和操作系统的课程，你可能已经详细学习过这一内容。）基本的想法是，CPU 维护一个段表，记录每个段的起始地址和长度，以及与每个段相关的权限。操作系统通常会设置硬件段，以对应刚才描述的逻辑组织。

尽管硬件分段在 20 世纪 80 年代的操作系统中被广泛使用，但它在很大程度上已被分页机制所取代，后者被认为更简单且更灵活。处理器厂商也作出了回应，在新的设计中移除了对硬件分段的支持。例如，从 8086 到 Pentium 的每一代 Intel x86 架构都在 32 位保护模式下支持分段。最新的 64 位架构仅提供分页机制，而不再提供分段。逻辑分段仍然是一种在内存中组织程序的有用方式。

让我们继续更详细地查看每一个逻辑部分。

9.3 堆管理

堆包含在运行时动态管理的内存。操作系统不控制堆的内部组织，除非对其总大小加以限制。相反，堆的内部结构由标准库或其他自动链接到程序中的运行时支持软件来管理。在 C 程序中，函数 malloc 和 free 分别在堆上分配和释放内存。在 C++ 中，new 和 delete 具有相同的效果。其他语言在创建和删除对象和数组时会隐式地操作堆。

最简单的 malloc 和 free 实现方式是将整个堆视为一个大的内存区域链表。链表中的每个条目记录了区域的状态（空闲或正在使用）、区域的大小，并且具有指向前一个和下一个区域的指针。以下是在 C 语言中可能的实现方式：

```
结构 chunk { 枚举 { 空闲, 已用 } 状态; int
大小; 结构 chunk *next; 结构 chunk *prev;
char 数据[0];


};
```

(请注意，我们将 data 声明为一个长度为零的数组。这是一个小技巧，只要底层内存实际存在，它就允许我们将 data 当作一个可变长度数组来处理。)

在该方案下，堆的初始状态仅仅是链表中的一个条目：

FREE	1000	data
prev	next	

假设用户调用 malloc(100) 来分配 100 字节的内存。malloc 会发现（唯一的）内存块是空闲的，但其大小远大于所请求的大小。因此，它会将其拆分为一个 100 字节的小块和一个包含剩余部分的较大块。这是通过在 100 字节之后的数据区域中简单地写入一个新的块头来完成的。然后，将它们连接成一个链表：

USED	100	data	FREE	900	data
prev	next		prev	next	

一旦列表被修改，malloc 就会返回块中数据元素的地址，使用户可以直接访问它。它不会返回链表节点本身，因为用户不需要了解实现细节。如果没有足够大的块来满足当前请求，那么进程必须通过调用 brk 向操作系统请求扩展堆。

当用户对一块内存调用 free 时，该块在链表中的状态被标记为 FREE，并且如果相邻节点也处于空闲状态，则会与它们合并。

（顺便一提，现在你可以看出，程序在给定的内存块之外不小心修改内存是多么危险。不仅可能影响其他内存块，还可能破坏链表本身，从而在下一次 malloc 或 free 时导致不可预测的行为！）

如果程序总是以与分配相反的顺序释放内存，那么堆就会被整齐地划分为已分配内存和空闲内存。然而，实际情况并非如此：内存可以以任意顺序被分配和释放。随着时间推移，堆可能退化为由大小各异的已分配和已释放内存块混杂组成的状态。这被称为内存碎片化。

过度碎片化会导致浪费：如果存在许多小的内存块，但没有任何一个足够大以满足当前的 malloc 请求，那么进程别无选择，只能扩展堆，从而使这些小块处于未使用状态。这会增加操作系统中总虚拟内存消耗的压力。

在像 C 这样的语言中，内存块在使用期间不能被移动，因此一旦发生碎片化就无法在事后修复。如何-

然而，内存分配器通过谨慎选择新分配的位置，在一定程度上能够避免碎片化。一些简单的策略很容易想到，并且已经得到了广泛研究：

- 最佳适配。在每次分配时，搜索整个链表并找到比请求大的*smallest*空闲块。这通常会留下较大的空闲空间，但会生成太小而无法使用的零碎空闲块。

- 最坏适配。每次分配时，搜索整个链表，找到大于请求的 *largest* 空闲块。颇为反直觉的是，这种方法往往通过避免产生微小且不可用的碎片来减少碎片化。

- 首次适配。每次分配时，从链表的起始处开始搜索，找到满足请求的 *first* 片段（无论大小）。与最佳适配或最坏适配相比，它所需的工作量更少，但随着链表规模的增大，其工作量也会不断增加。

- 下一次适配（Next Fit）。在每次分配时，从上一次检查的位置开始搜索链表，并找到满足请求的 *next* 碎片（大或小）。这种方法在每次分配时将所需的工作量降到最低，同时将分配分散到整个堆中。

对于通用型分配器，在无法对应用程序行为作出假设的情况下，传统观点认为 Next Fit 能够带来良好的性能，并且碎片化水平是可以接受的。

9.4 栈管理

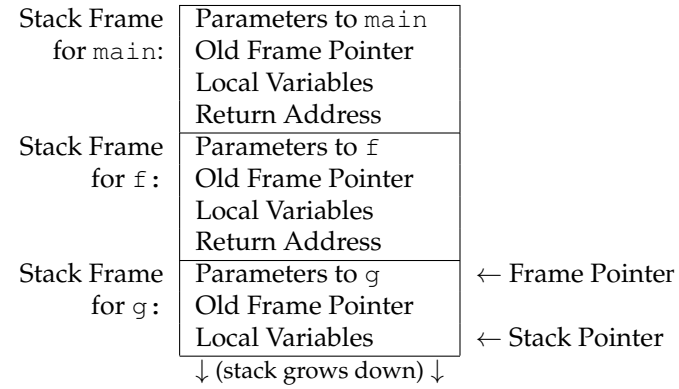
栈用于记录正在运行的程序的当前状态。大多数 CPU 都有一个专用寄存器——栈指针——用于存储下一个元素将被压入或弹出的地址。由于栈从内存的顶部向下增长，存在一个令人困惑的约定：向栈中压入一个元素会使栈指针移动到编号更低的地址，而从栈中弹出一个元素会使栈指针移动到更高的地址。栈的“顶部”实际上位于最低地址处！

每次函数调用都会在栈中占用一段内存，称为栈帧。栈帧包含该函数使用的参数和局部变量。当函数被调用时，会压入一个新的栈帧；当函数返回时，该栈帧被弹出，执行继续在调用者的栈帧中进行。

另一种称为帧指针（或有时称为基指针）的专用寄存器指示当前帧的起始位置。代码

在函数内部，依赖帧指针来确定当前参数和局部变量的位置。

例如，假设主函数调用函数 f，然后 f 再调用 g。如果我们在执行 g 的过程中将程序暂停，栈将如下所示：



栈帧中各个元素的顺序和细节在不同的 CPU 架构和操作系统之间会有所不同。只要调用方和被调用方对栈帧中包含的内容达成一致，那么任何函数都可以调用另一个函数，即使它们是用不同的语言编写的，或由不同的编译器构建的。

关于活动记录内容的约定称为调用约定。这通常会被写成一份详细的技术文档，供编译器、操作系统和库的设计者使用，以确保代码实现互操作性。

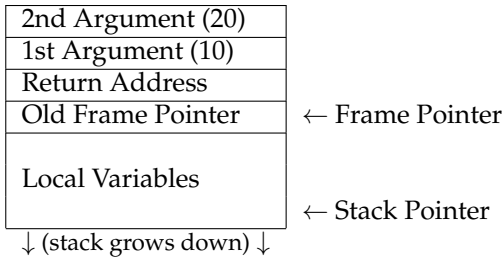
调用约定大致分为两大类，其间存在许多变体的可能性。一种是将函数调用的参数放在栈上，另一种是将它们放在寄存器中。

9.4.1 Stack Calling Convention

调用函数的传统方法是将传递给该函数的参数（按相反顺序）压入栈中，然后跳转到函数的地址，同时在栈中留下一个返回地址。大多数 CPU 都为此提供了专用的 CALL 指令。例如，用于调用 f(10,20) 的汇编代码可以简单到如下所示：

```
压栈 $20
压栈 $10
调用 f
```

当 f 开始执行时，它会保存当前生效的旧帧指针，并为自身的局部变量腾出空间。因此，f(10,20) 的栈帧如下所示：



为了访问其参数或局部变量，f 必须从相对于帧指针的内存中加载它们。正如你所看到的，函数参数位于帧指针*above*的固定位置，而局部变量位于帧指针*below*。[2]

9.4.2 Register Calling Convention

调用函数的另一种方法是先将参数放入寄存器中，然后再调用该函数。例如，假设我们的调用约定规定使用寄存器 %R10、%R11 等来传递参数。在这种调用约定下，用于调用 f(10,20) 的汇编代码可能如下所示：

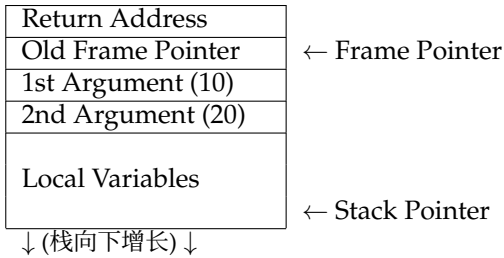
```
MOVE $10 -> %R10
MOVE $20 -> %R11 C
ALL f
```

当 f 开始执行时，它仍然必须保存旧的帧指针并为局部变量腾出空间。它不必从栈中加载参数；它只需期望这些值已经在 %R10 和 %R11 中，并且可以立即对它们进行计算。通过避免内存访问，这可能带来显著的速度优势。

但是，如果 f 是一个需要调用其他函数的复杂函数呢？它仍然需要保存参数寄存器的当前值，以便释放它们供自身使用。

为允许这种可能性，f 的栈帧必须为参数预留空间，以防它们需要被保存。调用约定必须定义参数的位置，并且它们通常*below*返回地址和旧帧指针存储，如下所示：

[2]The arguments are pushed in reverse order in order to allow the possibility of a variable number of arguments. Argument 1 is always two words above the frame pointer, argument 2 is always three words above, and so on.



如果函数的参数数量超过为参数预留的寄存器数量，会发生什么？在这种情况下，额外的参数会被压入栈中，就像在栈调用约定中一样。

从整体来看，在栈调用约定和寄存器调用约定之间的选择并没有太大影响，唯一重要的是各方必须就约定的细节达成一致。寄存器调用约定有一个小优势：叶子函数（不调用其他函数的函数）可以在不访问内存的情况下计算一个简单的结果。通常，寄存器调用约定用于那些拥有大量寄存器、否则可能会闲置的体系结构。

可以在单个程序中混合使用不同的约定，只要调用者和被调用者都知道这种区别。例如，Microsoft X86 编译器允许在函数原型中使用关键字来选择约定：cdecl 选择堆栈调用约定，而 fastcall 使用寄存器传递前两个参数。

9.5 定位数据

对于程序中的每种数据，必须有一种明确的方法来定位该数据在内存中的位置。编译器必须使用关于符号的基本信息生成地址计算。计算方式随数据的存储类别而变化：

- 全局数据的地址计算最为简单。事实上，编译器通常不会计算全局地址，而是将每个全局符号的 *name* 传递给汇编器，由汇编器来选择地址计算方式。在最简单的情况下，汇编器会生成一个绝对地址，给出数据在程序内存中的精确位置。

  然而，简单的方法不一定高效，因为绝对地址是一个完整的字（例如，64位），与内存中的指令大小相同。这意味着汇编程序必须使用多个指令（RISC）或多字指令（CISC）将地址加载到寄存器中。假设大多数程序不使用整个地址空间，通常不需要使用整个字。

  另一种选择是使用一种基址相对地址，它由寄存器给出的基地址加上由汇编器给定的固定偏移量组成。

例如，全球数据地址可以通过一个寄存器给出，该寄存器指示数据段的开始位置，再加上一个固定的偏移量；而函数地址则可以通过一个寄存器给出，该寄存器指示代码段的开始位置，再加上一个固定的偏移量。此方法可用于动态加载的库，当库的位置事先未固定，但库内函数的位置是已知时。

另一种方法是使用相对于PC的地址，其中计算引用指令与目标数据之间的精确字节距离，然后将其编码到指令中。只要相对距离足够小（例如16位），可以容纳在指令的地址字段中，这种方法就可以工作。此任务由汇编器执行，通常对程序员是不可见的。

- 本地数据的工作方式不同。由于局部变量存储在栈上，给定的局部变量每次使用时不一定占用相同的绝对地址。如果一个函数递归调用，可能会同时存在多个给定局部变量的实例！因此，局部变量总是相对于当前帧指针作为偏移量来指定。（偏移量可以是正数或负数，具体取决于函数调用约定。）函数参数只是局部变量的一个特例：参数在栈上的位置由其在参数中的顺序位置精确给出。

- 堆数据只能通过存储为全局或局部变量的指针来访问。要访问堆上的数据，编译器必须为指针本身生成一个地址计算，然后解引用指针以访问堆上的项。

迄今为止，我们只考虑了可以轻松存储在一个内存字中的原子数据类型：布尔值、整数、浮点数等。然而，任何更复杂的数据类型都可以放置在三种存储类别中的任意一种，并且需要一些额外的处理。

数组可以存储在全局、局部或堆内存中，数组的起始位置可以通过上述方法之一找到。数组中的元素通过将索引乘以数组中项的大小，然后将结果加到数组本身的地址来找到：

地址(a[i]) = 地址(a) + sizeof(类型) * i

更有趣的问题是如何处理数组本身的长度。在像C这样的不安全语言中，简单的方法是干脆什么都不做：如果程序恰好越过数组的末尾，编译器会高兴地计算一个数组边界之外的地址，结果是混乱。在一些对性能要求极高的应用中，这种方法的简便性胜过了任何安全性的提升。

一种更安全的方法是将数组的长度存储在数组本身的基地址处。然后，编译器可能会生成代码，在生成地址之前检查实际索引是否在数组边界内。这可以防止程序员在运行时发生任何意外。然而，缺点是性能。每次程序员提到 a[i] 时，生成的代码必须包含以下内容：

1. 计算数组 a 的地址。 2. 将 a 的长度加载到寄存器中。 3. 将数组索引 i 与寄存器进行比较。 4. 如果 i 超出数组边界，抛出异常。 5. 否则，计算 a[i] 的地址并继续。

这种模式非常常见，以至于一些计算机架构提供了专门的数组边界检查支持。Intel X86 架构（我们将在下一章详细讨论）提供了一个独特的 BOUND 指令，其唯一目的是将一个值与两个数组边界值进行比较，然后在该值超出范围时触发一个独特的“数组边界异常”。

结构具有类似的考虑因素。在内存中，结构非常类似于数组，除了它可以包含不规则大小的项。为了访问结构中的项，编译器必须生成结构起始位置的地址计算，然后加上与结构内项名称（称为结构标签）相对应的偏移量。当然，不需要执行边界检查，因为偏移量在编译时是固定的。

对于复杂的嵌套数据结构，定位单个元素所需的地址计算可能会变得相当复杂。例如，考虑下面这段用于表示一副扑克牌的代码：

```
结构体卡片 { 整数花色; 整数点数; }
; 结构体牌组 { 整数是否已洗牌; 结
构体卡片 卡片[52]; }; 结构体牌组 d;
```

```
d.cards[10].rank = 10;
```

要计算 d.cards[10].rank，编译器必须首先生成 d 的地址计算，具体取决于它是局部变量还是全局变量。然后，加上 cards 的偏移量，再加上第十个元素的偏移量，最后是 rank 在 card 中的偏移量。完整的地址计算为：

地址(d.card[10].rank) = 地址(d) + 偏移(cards)
+ sizeof(struct card)*10 + 偏移(rank)

9.6 程序加载

在程序开始在内存中执行之前，它首先以磁盘上的一个文件形式存在，因此必须有一种约定将其加载到内存中。用于在磁盘上组织程序的可执行文件格式有多种，从非常简单到非常复杂不等。下面给出几个示例以帮助你理解。

最简单的计算机系统只是将可执行文件作为一个二进制块存储在磁盘上。程序代码、数据以及堆和栈的初始状态被不加区分地直接转储到一个文件中。要运行该程序，操作系统只需将文件内容加载到内存中，然后跳转到程序的第一个位置开始执行。

这种方法几乎简单到不能再简单了。它确实可行，但也有若干局限性。其中之一是这种格式会在未初始化的数据上浪费空间。例如，如果程序声明了一个大型全局数组，而每个元素的值都是零，那么该数组中的每一个零都会被存储到文件中。另一个局限是，操作系统无法洞察程序打算如何使用内存，因此无法像前面讨论的那样为每个逻辑段设置权限。还有一点是，这个二进制块本身不包含任何标识信息，用以表明它是一个可执行文件。

然而，在程序规模很小且以简洁性为首要目标的场合，二进制块（binary blob）的方法仍然偶尔被使用。例如，PC 操作系统的最初引导阶段会从启动硬盘读取一个包含二进制块的单个扇区，该二进制块随后执行第二阶段的引导。嵌入式系统通常只有以几千字节计的非常小的程序，并依赖于二进制块。

在经典 Unix 系统中多年来使用的一种改进方法是 a.out 可执行文件格式。[3] 该格式有许多细微的变体，但它们都共享相同的基本结构。可执行文件由一个简短的头部结构组成，随后是文本段、已初始化的数据以及符号表：

[3] The first Unix assembler sent its output to a file named `a.out` by default. In the absence of any other name for the format, the name stuck.

Header	Text	Data	Symbols

头部结构本身只是几个字节，允许操作系统解释文件的其余部分：

Magic Number
Text Section Size
Data Section Size
BSS Size
Symbol Table Size
Entry Point

魔数是一个唯一的整数，明确地将文件定义为可执行文件：如果文件不是以此魔数开头，操作系统甚至不会尝试执行它。不同的魔数定义适用于可执行文件、未链接的目标文件和共享库。文本大小字段表示紧随头部后的文本部分的字节数。数据大小字段表示文件中出现的*initialized*数据量，而BSS大小字段表示*uninitialized data*的量。[4]

未初始化的数据不需要存储在文件中。相反，它仅在程序加载时作为数据段的一部分分配到内存中。可执行文件中的符号表列出了程序中使用的每个变量和函数的名称，以及它们在代码段和数据段中的位置；这使得调试器能够解释地址的含义。最后，入口点提供了程序起始点的地址（通常是 main），该地址位于文本段中。这允许起始点是程序中的其他地址，而不是第一个地址。

a.out 格式比二进制块有了很大改进，至今在许多操作系统中仍然得到支持和使用。然而，它的功能还不足以支持现代语言所需的许多特性，特别是动态加载的库。

可扩展链接格式（ELF）今天在许多操作系统中广泛使用，用于表示可执行文件、目标文件和共享库。像a.out一样，ELF文件有多个部分表示代码、数据和bss，但它可以有任意数量的附加部分，用于调试数据、初始化和终止代码、关于工具使用的元数据等。文件中的*sections*数量超过了内存中的*seg- ments*，因此ELF文件中的节表指示如何将多个节映射到一个单独的段中。

[4] BSS stands for “Block Started by Symbol” and first appeared in an assembler for IBM 704 in the 1950s.

File Header
Program Header
Code Section
Data Section
Read-Only Section
. . .
Section Header

9.7 延伸阅读

1. 雷姆齐·阿尔帕奇-杜索 和 安德里亚·阿尔帕奇-杜索，《操作系统导论：三大简易篇》，Arpaci-Dusseau Books，2015 年。

   http://www.ostep.org

   Operating systems is usually the course in which memory management is covered in great detail. If you need a refresher on memory allocators (or anything else in operating systems), check out this online textbook.

2. John R. Levine，《链接器和加载器》，摩根·考夫曼出版社，1999年。
This book provides a detailed look at linkers and loaders, which is an often-overlooked topic that falls in cracks between compilers and operating systems. A solid understanding of linking is necessarily to create and use libraries effectively.

3. Paul R. Wilson，《单处理器垃圾回收技术》，计算机科学讲义（Lecture Notes in Computer Science），第637卷，1992年。

   源文本：https://link.springer.com/chapter/10.1007/BFb0017182

   This widely-read article gives an accessible overview of the key techniques of garbage collection, which is an essential component of the runtime of modern dynamic languages.

第10章——汇编语言

10.1 引言

为了构建一个编译器，你必须至少具备对一种汇编语言的实际工作知识。此外，了解两种或以上的汇编语言变体有助于充分理解不同体系结构之间的差异。这些差异中，有些（如寄存器结构）非常根本，而另一些则只是表面的。

我们观察到，许多学生似乎认为汇编语言相当晦涩而复杂。确实，CPU 的完整手册异常厚重，可能记录了数百条指令以及晦涩的寻址模式。然而，根据我们的经验，实际上只需要学习某种汇编语言的一小部分（也许 30 条指令）就足以编写一个可用的编译器。许多额外的指令和特性是为了处理操作系统、浮点数学以及多媒体计算中的特殊情况而存在的。使用这个基本子集，你几乎可以完成所需的一切。

我们将考察当今广泛使用的两种不同 CPU 架构：X86 和 ARM。Intel X86 是一种 CISC 架构，自 20 世纪 70 年代以来从 8 位发展到 64 位，如今已成为个人计算机、笔记本电脑以及高性能服务器中的主导芯片。ARM 处理器是一种 RISC 架构，最初作为面向个人计算机市场的 32 位芯片诞生，如今已成为低功耗和嵌入式设备（如手机和平板电脑）的主导芯片。

本章将使你对每种体系结构的基础有一个实用的了解，但你仍需要一本可靠的参考资料来查阅更多细节。我们建议你查阅《Intel 软件开发者手册》[1] 和《ARM 架构参考手册》[3] 以获取完整细节。（请注意，每一节都旨在相互对应且自成体系，因此针对 X86 和 ARM 的一些说明性内容会有所重复。）

10.2 开源汇编器工具

同一种 CPU 的汇编语言可能存在多种方言，这取决于使用的是芯片厂商提供的汇编器，还是其他开源工具。为保持一致性，我们将给出使用 GNU 编译器和汇编器所支持的汇编方言的示例，它们分别称为 gcc 和 as（有时也称为 gas）。

一个很好的入门方法是查看编译器为 C 程序生成的汇编输出。为此，使用 -S 标志运行 gcc，编译器将生成汇编输出而不是二进制程序。在类 Unix 系统中，汇编代码存储在以 .s 结尾的文件中，这表示“源”文件。

如果你对这个 C 程序运行 gcc -S hello.c -o hello.s：

```
#include <stdio.h>

int main( int argc, char *argv[] ) { printf("你好 %s
\n","世界"); return 0; }
```

然后你应该在 hello.s 中看到类似这样的输出

```
.file "test.c" .data .LC0: .string "你好 %s\
n" .LC1: .string "世界" .text .global main

main: PUSHQ %rbp MOVQ %rsp, %rbp SUB
Q $16, %rsp MOVQ %rdi, -8(%rbp) MOVQ
%rsi, -16(%rbp) MOVQ $.LC0, %rax MOVQ
$.LC1, %rsi MOVQ %rax, %rdi MOVQ $0,
%rax CALL printf MOVQ $0, %rax LEAVE
RET
```

(有许多有效的方法可以编译hello.c，因此您的编译器输出可能会有所不同。)

无论 CPU 架构如何，汇编代码都有三种不同类型的元素：

指令以点号开头，用于指示对汇编器、链接器或调试器有用的结构性信息，但其本身并不是汇编指令。例如，.file 只是记录原始源文件的名称以辅助调试器。.data 表示程序数据段的开始，而 .text 表示程序代码段的开始。.string 表示数据段中的字符串常量，而 .global main 表示标签 main 是一个全局符号，可被其他代码模块访问。

标签以冒号结尾，通过其位置指示名称与位置之间的关联。例如，标签 .LC0: 表示紧接着的字符串应被称为 .LC0。标签 main: 表示指令 PUSHQ %rbp 是主函数的第一条指令。根据约定，以点号开头的标签是由编译器生成的临时局部标签，而其他符号是用户可见的函数和全局变量。标签不会成为生成的机器代码 *per se* 的一部分，但它们会出现在生成的目标代码中，用于链接，并且在最终的可执行文件中，用于调试。

指令是实际的汇编代码，如（PUSHQ %rbp），通常会缩进以在视觉上将其与指令和标签区分开来。GNU 汇编中的指令不区分大小写，但为了保持一致性，我们通常会将其大写。

要将这个 hello.s 转换成可运行的程序，只需运行 gcc，它会识别这是一个汇编程序，将其汇编并与标准库链接：

% gcc hello.s -o hello % ./hello
你好，世界

将汇编代码编译成目标代码，然后使用 nm 工具显示代码中存在的符号（“名称”）也是很有趣的：

% gcc hello.s -c -o hello.o % nm hello.o
0000000000000000 T main U printf

这显示了链接器可用的信息。main 位于对象的文本（T）段，位置为零，而 printf 是未定义的。

(U)，因为它必须从标准库中获取。但像.LC0这样的标签没有出现，因为它们没有声明为.global。

在学习汇编语言时，利用现有的编译器：编写一些简单的函数，查看 gcc 生成的代码。这可以为你提供一些起点，帮助你识别可以使用的新指令和技术。

10.3 X86汇编语言

X86 是一个通用术语，指的是一系列源自（或与之兼容的）英特尔 8088 处理器的微处理器，该处理器用于原始的 IBM PC，包括 8086、80286、’386、’486 及其他许多型号。每一代 CPU 都新增了指令和寻址模式，从 8 位到 16 位再到 32 位，同时保持与旧代码的向后兼容性。多家竞争者（如 AMD）生产了兼容的芯片，实施相同的指令集。

然而，英特尔在64位时代打破了传统，推出了一个新品牌（Itanium）和架构（IA64），该架构与旧代码具有*not*向后兼容性。相反，它实施了一个新概念——非常长指令字（VLIW），在该概念中，多个并发操作被编码到一个单一的指令字中。由于指令级并行性，这具有显著加速的潜力，但也意味着与过去的断裂。

AMD坚持使用旧的方式，生产了一个64位架构（AMD64），该架构与*was*向后兼容Intel和AMD的芯片。虽然两种方法的技术优点存在争议，但AMD的方法在市场上获胜，Intel随后也推出了自己的64位架构（Intel64），该架构与AMD64以及其自身的上一代芯片兼容。X86-64是涵盖AMD64和Intel64架构的通用名称。

X86-64 是 CISC（复杂指令集计算）的一個典型例子。它有大量的指令，並且有許多不同的子模式，其中一些是為非常狹窄的任務設計的。然而，一小部分指令就能讓我們完成很多工作。

10.3.1 Registers and Data Types

X86-64 有十六个（几乎）通用的 64 位整数寄存器：

%rax	%rbx	%rcx	%rdx	%rsi	%rdi	%rbp	%rsp
%r8	%r9	%r10	%r11	%r12	%r13	%r14	%r15

这些寄存器是 *almost* 通用的，因为早期版本的处理器计划将每个寄存器用于特定目的，并且并非所有指令都可以应用于每个寄存器。

A Note on AT&T Syntax versus Intel Syntax

Note that the GNU tools use the traditional AT&T syntax, which is used across many processors on Unix-like operating systems, as opposed to the Intel syntax typically used on DOS and Windows systems. The following instruction is given in AT&T syntax:

```
MOVQ %RSP, %RBP
```

`MOVQ` is the name of the instruction, and the percent signs indicate that `RSP` and `RBP` are registers. In the AT&T syntax, the source is always given first, and the destination is always given second.

In other places (such as the Intel manual), you will see the Intel syntax, which (among other things) dispenses with the percent signs and *reverses* the order of the arguments. For example, this is the same instruction in the Intel syntax:

```
MOVQ RBP, RSP
```

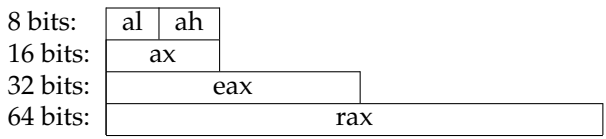
When reading manuals and web pages, be careful to determine whether you are looking at AT&T or Intel syntax: look for the percent signs!

较低的八个寄存器表明了它们最初的用途：例如，%rax 是累加器。

随着设计的发展，新增了指令和寻址方式，使各个寄存器几乎等同。少数剩余的指令，尤其是与字符串处理相关的指令，仍然需要使用 %rsi 和 %rdi。此外，有两个寄存器被保留用作栈指针（%rsp）和基址指针（%rbp）。最后八个寄存器以编号区分，没有任何特定限制。

多年来，该体系结构从 8 位扩展到 64 位，因此每个寄存器都有一定的内部结构。%rax 寄存器的最低 8 位是一个 8 位寄存器 %al，接下来的 8 位称为 %ah。低 16 位统称为 %ax，低 32 位称为 %eax，而整个 64 位称为 %rax。

图 10.1：X86 寄存器结构



编号寄存器 %r8-%r15 具有相同的结构，但命名方案略有不同：

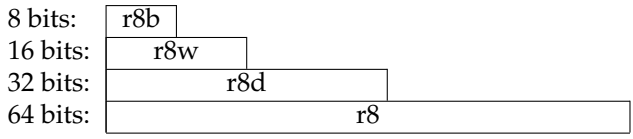


图 10.2：x86 寄存器结构

为了简化问题，我们将重点关注64位寄存器。然而，大多数生产环境中的编译器使用多种模式：一个字节可以表示一个布尔值；一个长字通常足以进行整数运算，因为大多数程序不需要超过2^{32}的整数值；而一个四字则需要表示内存地址，从而支持最多16EB（艾字节）的虚拟内存。

10.3.2 Addressing Modes

MOV 指令在寄存器之间以及内存之间以多种不同模式传输数据。一个单字母后缀决定了要移动的数据的大小：

Suffix	Name	Size	
B	BYTE	1 byte	(8 bits)
W	WORD	2 bytes	(16 bits)
L	LONG	4 bytes	(32 bits)
Q	QUADWORD	8 bytes	(64 bits)

MOVB 移动一个字节，MOVW 移动一个字，MOVL 移动一个长整型，MOVQ 移动一个四字。[1] 通常，您要移动到和从中移动的位置的大小必须与后缀匹配。在某些情况下，您可以省略后缀，汇编器会推断出正确的大小。然而，这可能会产生意想不到的后果，因此我们将养成使用后缀的习惯。

MOV 的参数可以采用多种寻址模式之一。

- 全局值仅通过一个简单的名称（如 x 或 printf）来引用，汇编器将其转换为绝对地址或地址计算。

- 立即数是由美元符号表示的常数值，例如$56，且其范围有限，具体取决于所使用的指令。

- 寄存器值是寄存器的名称，例如 %rbx。

1注意：这些术语不可移植。在不同的机器上，*word* 的大小不同。

- 间接值是指通过寄存器中包含的地址所指向的值。例如，(%rsp) 指的是 %rsp 所指向的值。

- 基址相对值是通过在寄存器名称上加一个常量来给出的。例如，-16(%rcx) 指的是位于 %rcx 所指示地址之下 16 个字节处的内存位置中的值。这种寻址方式对于操作栈、本地变量以及函数参数非常重要，因为对象的起始位置由一个寄存器给出。

- 复杂地址的形式为 $D(R_A, R_B, C)$，它指的是地址 $R_A + R_B * C + D$ 处的值。R_A 和 R_B 都是通用寄存器，而 C 可以取 1、2、4 或 8，D 可以是任意整数位移。该模式用于在数组中选择一个元素，其中 R_A 给出数组的基址，R_B 给出数组中的索引，C 给出数组中元素的大小，D 是相对于该元素的偏移量。

下面是一个使用每种寻址模式将一个 64 位值加载到 %rax 中的示例：

Mode	Example
Global Symbol	`MOVQ x, %rax`
Immediate	`MOVQ $56, %rax`
Register	`MOVQ %rbx, %rax`
Indirect	`MOVQ (%rsp), %rax`
Base-Relative	`MOVQ -8(%rbp), %rax`
Complex	`MOVQ -16(%rbx,%rcx,8), %rax`

在大多数情况下，用于将数据存入寄存器和内存位置的寻址模式是相同的。但也存在一些例外。例如，不能在 MOV 的两个操作数中同时使用基址相对寻址：MOVQ -8(%rbx), -8(%rbx)。要准确了解某条指令支持哪些寻址模式组合，必须查阅该指令对应的手册页。

在某些情况下，你可能希望加载变量的 *address*，而不是它的值。这在处理字符串或数组时非常方便。为此，可以使用 LEA（load effective address，加载有效地址）指令，它能够执行与 MOV 相同的地址计算：

Mode	Example
Global Symbol	`LEAQ x, %rax`
Base-Relative	`LEAQ -8(%rbp), %rax`
Complex	`LEAQ -16(%rbx,%rcx,8), %rax`

10.3.3 Basic Arithmetic

你将需要四个基本的算术指令用于你的编译器：整数加法、减法、乘法和除法。

ADD 和 SUB 有两个操作数：一个源操作数和一个破坏性目标操作数。例如，这条指令：

```
ADDQ %rbx, %rax
```

将 %rbx 加到 %rax 上，并将结果放入 %rax，覆盖之前可能存在的内容。这需要稍加小心，以免不小心破坏你之后可能还想使用的值。例如，你可以将 c = a+b+b; 翻译成这样：

```
MOVQ a, %rax MOV
Q b, %rbx ADDQ %r
bx, %rax ADDQ %rb
x, %rax MOVQ %rax,
c
```

IMUL 指令有些不寻常，因为两个 64 位整数相乘通常会得到一个 128 位整数。IMUL 采用其参数，将其与 %rax 寄存器的内容相乘，然后将结果的低 64 位放入 %rax 寄存器，将高 64 位放入 %rdx 寄存器。（这是隐式的：指令中并未提及 %rdx。）

例如，假设您希望翻译 c = b*(b+a);，其中 a、b 和 c 是全局整数。以下是一个可能的翻译：

```
MOVQ a, %rax  MOV
Q b, %rbx  ADDQ %rb
x, %rax  IMULQ %rbx
MOVQ %rax, c
```

IDIV 指令做的事情与此相同，只是方向相反：它以一个 128 位的整数值开始，其中低 64 位存储在 %rax 中，高 64 位存储在 %rdx 中，然后用指令中给定的值进行除法运算。商存储在 %rax 中，余数存储在 %rdx 中。（如果你想实现取模指令而不是除法，只需使用 %rdx 的值。）

要设置除法，必须确保两个寄存器都具有必要的符号扩展值。如果被除数适合于低64位，但为负数，则高64位必须全部为1，以完成二进制补码表示。CQO指令的作用是将%rax符号扩展到%rdx，以进行除法运算。

例如，要将a除以五：

```
MOVQ a,  %rax     # set the low 64 bits of the dividend
CQO               # sign-extend %rax into %rdx
IDIVQ $5          # divide %rdx:%rax by 5,
                  #          leaving result in %rax
```

指令 INC 和 DEC 会以破坏性的方式对寄存器进行递增和递减。例如，语句 a = ++b 可以被翻译为：

MOVQ b, %rax I
NCQ %rax MOV
Q %rax,b MOVQ
%rax, a

指令 AND、OR 和 XOR 对两个值执行破坏性的 *bitwise* 布尔运算。按位（bitwise）意味着该运算会应用于操作数中的每一个独立位，并将结果存储起来。
因此，AND $0101B $0110B 将产生结果 $0100B。以类似的方式，NOT 指令会反转操作数中的每一位。例如，按位的 C 表达式 c = (a & b); 可以这样翻译：

MOVQ a, %rax MOV
Q b, %rbx NOTQ %rb
x ANDQ %rax, %rbx
MOVQ %rbx, c

请注意这里：这些指令 *do not* 按照你可能已经熟悉的 C 语言表示方式来实现逻辑布尔运算。例如，如果你将“false”定义为整数零，将“true”定义为任何非零值。在这种情况下，$0001 为 true，但对 $0001B 取 NOT 得到的是 $1110B，而这同样也是 true！要正确地实现这一点，你需要使用下面所描述的带条件的 CMP 指令。[2]

与 MOV 指令类似，各种算术指令也可以作用于多种寻址模式。然而，对于你的编译器项目而言，你很可能会发现，使用 MOV 在寄存器与寄存器之间加载和存储值最为方便，然后仅使用寄存器来执行算术运算。

[2] Alternatively, you could could use the bitwise operators as logical operators if you give `true` the integer value -1 (all ones) and `false` the integer value zero.

10.3.4 Comparisons and Jumps

使用JMP指令，我们可以创建一个简单的无限循环，使用%rax寄存器从零开始计数：

```
        MOVQ $0, %rax  l
oop:  INCQ %rax  JMP loop
```

为了定义更有用的结构，如终止循环和 if-then 语句，我们必须拥有评估值和改变程序流程的机制。在大多数汇编语言中，这些通过两种不同类型的指令来处理：比较和跳转。

所有比较通过CMP指令完成。CMP比较两个不同的寄存器，然后设置内部EFLAGS寄存器中的一些位，记录值是否相同、更大或更小。你不需要直接查看EFLAGS寄存器。相反，一些条件跳转指令会检查EFLAGS寄存器并进行适当的跳转：

Instruction	**Meaning**
JE	Jump if Equal
JNE	Jump if Not Equal
JL	Jump if Less
JLE	Jump if Less or Equal
JG	Jump if Greater
JGE	Jump if Greater or Equal

例如，这里有一个循环，将 %rax 从零计数到五：

```
        MOVQ $0, %rax  l
oop: INCQ %rax  CMPQ $5,
%rax  JLE loop
```

这是一个条件赋值：如果全局变量 x 大于零，则全局变量 y 赋值为十，否则为二十：

```
        MOVQ x, %rax  CM
PQ $0, %rax  JLE .L1  .L0: MO
VQ $10, $rbx  JMP .L2  .L1: M
OVQ $20, $rbx  .L2: MOVQ %r
bx, y
```

注意，跳转需要编译器定义目标标签。这些标签在一个汇编文件内必须是唯一且私有的，但除非给出 .global 指令，否则在文件外不可见。像 .L0 、.L1 等这样的标签可以由编译器按需生成。

10.3.5 The Stack

栈是一种辅助数据结构，主要用于记录程序的函数调用历史，以及那些无法放入寄存器的局部变量。按照约定，栈从高地址向低地址增长 *downward* 。%rsp 寄存器被称为栈指针，用于跟踪栈中最底部的项。

要将 %rax 压入栈中，我们必须从 %rsp 中减去 8（%rax 的字节大小） ，然后写入 %rsp 指向的位置：

SUBQ $8, %rsp MOVQ
%rax, (%rsp)

从栈中弹出一个值则涉及相反的过程：

将 (%rsp) 的值移动到
%rax，将 $8 加到 %rsp

要从栈中丢弃最近的值，只需将栈指针移动相应数量的字节：

ADDQ $8, %rsp

当然，推送到和从由 %rsp 引用的栈弹出的操作是如此常见，以至于这两个操作都有自己的指令，行为与上述完全相同：

将 %rax 压栈
将 %rax 出栈

请注意，在 64 位代码中，PUSH 和 POP 仅限于处理 64 位值，因此如果需要将较小的数据项压入或从栈中取出，则必须使用手动的 MOV 和 AD D。

10.3.6 Calling a Function

在这里描述的64位架构之前，使用了简单的堆栈调用约定：参数按逆序推入堆栈，然后通过CALL调用函数。被调用的函数在堆栈上查找参数，执行工作，并将结果返回到%eax。调用者随后从堆栈中移除参数。

然而，64 位代码使用寄存器调用约定，以便利用 X86-64 架构中数量更多的可用寄存器。[3] 这种约定被称为 System V ABI [2]，并在一份冗长的技术文档中有详细说明。完整的约定相当复杂，但本摘要涵盖了基本情况：

图 10.3：System V ABI 调用约定概要

- 前六个整数参数（包括指针和其他可以存储为整数的类型）按顺序放入寄存器 %rdi、%rsi、%rdx、%rcx、%r8 和 %r9。
- 前八个浮点参数按顺序放置在寄存器 %xmm0-%xmm7 中。
- 超出这些寄存器数量的参数会被压入栈中。
- 如果函数接受可变数量的参数（如 printf），那么必须将 %rax 寄存器设置为浮点参数的数量。
- 函数的返回值被放置在%rax中。

此外，我们还需要了解其余寄存器是如何处理的。有些是调用者保存的，这意味着调用函数必须在调用另一个函数之前保存这些值。其他的是被调用者保存的，这意味着一个函数在被调用时，必须保存这些寄存器的值，并在返回时恢复它们。参数和结果寄存器无需保存。图 10.4 显示了这些要求。

要调用一个函数，我们必须首先计算参数并将它们放入所需的寄存器中。然后，我们必须将两个调用者保存寄存器（%r10 和 %r11）压入栈中，以保存它们的值。接着发出 CALL 指令，该指令会将当前指令指针压入栈中，然后跳转到函数的代码位置。从函数返回后，我们将这两个调用者保存寄存器从栈中弹出，并在 %rax 寄存器中查找函数的返回值。

[3] Note that there is nothing *stopping* you from writing a compiler that uses a stack calling convention. But if you want to invoke functions compiled by others (like the standard library) then you need to stick to the convention already in use.

这是一个示例。下面的 C 程序：

int x=0; int y
=10;

```cint main() { x = printf("值: %d\n",y); }```

Sure! Please provide the source text you'd like me to translate into Chinese.

.data x: .quad 0 y: .quad 10 str: .string "
值: %d\n"

请提供您需要翻译的源文本。
.global main
主函数: MOVQ $str, %rdi # 第一个参数放入 %rdi: 字符串 MOVQ y, %rsi # 第二个参数放入 %rsi: y MOVQ $0, %rax # 没有浮点参数 PUSHQ %r10 # 保存调用者保存的寄存器 PUSHQ %r11 CALL printf # 调用 printf POPQ %r11 # 恢复调用者保存的寄存器 POPQ %r10 MOVQ %rax, x # 将结果保存到 x RET # 从主函数返回
```

图 10.4：系统 V ABI 寄存器分配

Register	Purpose	Who Saves?
%rax	result	not saved
%rbx	scratch	callee saves
%rcx	argument 4	not saved
%rdx	argument 3	not saved
%rsi	argument 2	not saved
%rdi	argument 1	not saved
%rbp	base pointer	callee saves
%rsp	stack pointer	callee saves
%r8	argument 5	not saved
%r9	argument 6	not saved
%r10	scratch	CALLER saves
%r11	scratch	CALLER saves
%r12	scratch	callee saves
%r13	scratch	callee saves
%r14	scratch	callee saves
%r15	scratch	callee saves

10.3.7 Defining a Leaf Function

因为函数参数是通过寄存器传递的，因此很容易编写一个叶子函数，该函数计算一个值而不调用任何其他函数。例如，以下函数的代码：

平方：函数 整数 (x: 整数) = { 返回 x*x; } 可以简单地这样写：

.global square square: MOVQ %rdi, %rax # 将第一个参数复制到 %rax IMULQ %rax # 将其与自身相乘 # 结果已经在 %rax 中 RET # 返回调用者

不幸的是，这对于想要调用其他函数的函数不起作用，因为我们没有正确设置堆栈。一般情况需要更复杂的方法。

10.3.8 Defining a Complex Function

一个复杂的函数必须能够调用其他函数并计算任意复杂度的表达式，然后以原始状态完整返回给调用者。考虑以下一个接受三个参数并使用两个局部变量的函数示例：

```
.global func
func:
    pushq %rbp          # save the base pointer
    movq  %rsp, %rbp    # set new base pointer

    pushq %rdi          # save first argument on the stack
    pushq %rsi          # save second argument on the stack
    pushq %rdx          # save third argument on the stack

    subq  $16, %rsp     # allocate two more local variables

    pushq %rbx          # save callee-saved registers
    pushq %r12
    pushq %r13
    pushq %r14
    pushq %r15

    ### body of function goes here ###

    popq %r15            # restore callee-saved registers
    popq %r14
    popq %r13
    popq %r12
    popq %rbx

    movq   %rbp, %rsp    # reset stack pointer
    popq   %rbp          # recover previous base pointer
    ret                  # return to the caller
```

这里有很多内容需要跟踪：传递给函数的参数、返回所需的信息以及用于局部计算的空间。为此，我们使用基址寄存器指针 %rbp。栈指针 %rsp 指向栈的末尾，新数据将在那里压入，而基址指针 %rbp 指向该函数使用的值的起始位置。%rbp 和 %rsp 之间的空间被称为此函数调用的栈帧。

还有一个复杂之处：每个函数都需要使用若干寄存器来执行计算。然而，当一个函数在另一个函数执行过程中被调用时会发生什么？我们不希望调用者当前正在使用的任何寄存器被被调用函数覆盖。为防止这种情况，每个函数必须在开始时将其使用的所有寄存器压入栈中，并在返回前将它们从栈中弹出并恢复。根据图 10.4，每个函数在完成时必须保持 %rsp、%rbp、%rbx 和 %r12-%r15 的值不变。

以下是上面定义的 func 的栈布局：

Contents	**Address**	
old %rip register	8(%rbp)	
old %rbp register	(%rbp)	← %rbp points here
argument 0	-8(%rbp)	
argument 1	-16(%rbp)	
argument 2	-24(%rbp)	
local variable 0	-32(%rbp)	
local variable 1	-40(%rbp)	
saved register %rbx	-48(%rbp)	
saved register %r12	-56(%rbp)	
saved register %r13	-64(%rbp)	
saved register %r14	-72(%rbp)	
saved register %r15	-80(%rbp)	← %rsp points here

图 10.5：X86-64 栈布局示例

请注意，基址指针（%rbp）标识了栈帧的起始位置。因此，在函数体内部，我们可以使用相对于基址指针的基址相对寻址来同时引用参数和局部变量。函数的参数位于基址指针之后，因此参数零位于 -8(%rbp)，参数一位于 -16(%rbp)，依此类推。再往后是函数的局部变量，位于 -32(%rbp)，然后是在 -48(%rbp) 处保存的寄存器。栈指针指向栈中的最后一个元素。如果我们将栈用于其他目的，数据将被压入到更负的地址。（请注意，这里我们假设所有参数和变量都是 8 字节大小：不同的数据类型会导致不同的偏移量。）

下面是一个将所有内容整合在一起的完整示例。假设你有一个如下所定义的 B-minor 函数：

```
计算: 函数 整数 ( a: 整数, b: 整数, c: 整数 ) = { x:整数 = a+b+c; y:整
数 = x*5; 返回 y; }
```

该函数的完整翻译在下一页。给出的代码是正确的，但相当保守。事实证明，这个特定的函数并不需要使用寄存器 %rbx 和 %r15，因此也就没有必要保存并恢复它们。类似地，我们本可以将参数一直保存在寄存器中，而不必将它们保存到栈上。结果也可以直接计算到 %rax 中，而不是保存到一个局部变量里。这些优化在手写代码时很容易做到，但在编写编译器时就没那么容易了。

在你第一次尝试构建编译器时，如果每条语句都是独立翻译的，那么你生成的代码（很可能）不会非常高效。函数的前导部分必须保存所有寄存器，因为它并不知道 *a priori* 之后会使用哪些寄存器。同样地，一个计算值的语句必须将其保存回本地变量，因为它事先不知道该本地变量是否会被用作返回值。我们将在第 12 章关于优化的内容中进一步探讨这些问题。

图 10.6：完整的 X86 示例.global compute compute: ### ################## 函数前言部分设置栈pushq %rbp # 保存基址指针 movq %rsp, %rbp # 将新的基址指针设置为 rsp pushq %rdi # 将第一个参数 (a) 保存到栈上 pushq %rsi # 将第二个参数 (b) 保存到栈上 pushq %rdx # 将第三个参数 (c) 保存到栈上 subq $16, %rsp # 为两个局部变量分配空间 pushq %rbx # 保存被调用者保存的寄存器 pushq %r12 pushq %r13 pushq %r14 pushq %r15 ######################## 函数主体部分从这里开始movq -8(%rbp), %rbx # 将每个参数加载到寄存器中 movq -16(%rbp), %rcx movq -24(%rbp), %rdx addq %rdx, %rcx # 将参数相加 addq %rcx, %rbx movq %rbx, -32(%rbp) # 将结果存储到局部变量 0 (x) movq -32(%rbp), %rbx # 将局部变量 0 (x) 加载到寄存器中。 movq $5, %rcx # 将 5 加载到寄存器中 movq %rbx, %rax # 将参数移动到 rax 中 imulq %rcx # 将它们相乘 movq %rax, -40(%rbp) # 将结果存储到局部变量 1 (y) movq -40(%rbp), %rax # 将局部变量 1 (y) 移动到结果中 ################### 函数尾声部分恢复栈popq %r15 # 恢复被调用者保存的寄存器 popq %r14 popq %r13 popq %r12 popq %rbx movq %rbp, %rsp # 将栈重置为基址指针。 popq %rbp # 恢复旧的基址指针 ret # 返回给调用者

10.4 ARM 汇编

ARM处理器架构的历史几乎与X86架构一样悠久。它起源于1987年在Acorn Archimedes个人计算机中使用的32位Acorn RISC机器，并且自那时以来已发展成一种广泛使用的低功耗CPU，广泛应用于许多嵌入式和移动系统，现在被称为高级RISC机器（ARM）。这一不断发展的架构已经被多个芯片厂商基于共同的架构定义实现。该架构的最新版本是ARMv7-A（32位）和ARMv8-A（64位）。本章将重点讨论32位架构，并在最后简要提及64位架构的不同之处。

ARM 是一种精简指令集计算机（RISC）的例子，而不是复杂指令集计算机（CISC）。与 X86 相比，ARM 依赖于一组较小的指令，这些指令更容易进行流水线处理或并行执行，从而减少了芯片的复杂性和能量消耗。由于一些例外，ARM 有时被认为是“部分” RISC。例如，某些 ARM 指令的执行时间差异使得流水线不完美，包含用于预处理的桶式移位器引入了更复杂的指令，而条件执行减少了执行的潜在指令，并导致更少的分支指令，从而减少了处理器的能量消耗。我们将主要关注指令集的元素，这些元素使我们能够在编译器中实现更多功能，将编程语言的更复杂方面和优化留到以后再讨论。

10.4.1 Registers and Data Types

ARM-32 具有一组 16 个通用寄存器，r0–r15，其使用遵循以下约定：

Name	**Alias**	**Purpose**
r0		General Purpose Register
r1		General Purpose Register
...		. . .
r10		General Purpose Register
r11	fp	Frame Pointer
r12	ip	Intra-Procedure-Call Scratch Register
r13	sp	Stack Pointer
r14	lr	Link Register (Return Address)
r15	pc	Program Counter

除了通用寄存器之外，还有两个不能直接访问的寄存器：当前程序状态寄存器（CPSR）和保存的程序状态寄存器（SPSR）。它们保存比较操作的结果以及关于进程的特权数据。

状态。用户级程序无法直接访问这些，但它们可以作为某些操作的副作用被设置。

ARM使用以下后缀来表示数据大小。请注意，这些后缀在X86汇编中的含义不同！如果没有给出后缀，汇编器将假定为无符号字操作数。签名类型用于在将小数据类型加载到较大的寄存器时提供适当的符号扩展。对于小于字的数据类型，没有寄存器命名结构。

Data Type	Suffix	Size
Byte	B	8 bits
Halfword	H	16 bits
Word	W	32 bits
Double Word	-	64 bits
Signed Byte	SB	8 bits
Signed Halfword	SH	16 bits
Signed Word	SW	32 bits
Double Word	-	64 bits

10.4.2 Addressing Modes

ARM 使用两类不同的指令在寄存器之间以及寄存器与内存之间移动数据。MOV 在寄存器之间复制数据和常量，而 LDR（加载）和 STR（存储）指令用于在寄存器与内存之间移动数据。

MOV 指令用于将一个已知的立即数移动到指定寄存器中，或将另一个寄存器的值移动到第一个寄存器中。在 ARM 中，立即数用 # 表示。然而，这些立即数必须是 16 位或更小的值；如果更大，则必须改用 LDR 指令。大多数 ARM 指令在左侧指明目标寄存器，在右侧指明源寄存器。（STR 是一个例外。）因此，在立即数与寄存器之间移动数据时，将具有如下形式：

Mode	Example
Immediate	`MOV r0, #3`
Register	`MOV r1, r0`

每种数据类型的助记字母都可以附加到 MOV 指令上，使我们能够确定正在传送的是哪种数据，以及该数据是如何被传送的。否则，汇编器会假定为一个完整的字。

要在内存中读写数据，请使用加载（LDR）和存储（STR）指令。这两条指令都将源或目标寄存器作为第一个参数，将源或目标的内存地址作为第二个参数。在最简单的情况下，地址由一个寄存器给出，并用方括号表示：

图 10.7：ARM 寻址模式

Address Mode	**Example**
Literal	`LDR Rd, =0xABCD1234`
Absolute Address	`LDR Rd, =label`
Register Indirect	`LDR Rd, [Ra]`
Pre-indexing - Immediate	`LDR Rd, [Ra, #4]`
Pre-indexing - Register	`LDR Rd, [Ra, Ro]`
Pre-indexing - Immediate & Writeback	`LDR Rd, [Ra, #4]!`
Pre-indexing - Register & Writeback	`LDR Rd, [Ra, Ro]!`
Post-indexing - Immediate	`LDR Rd, [Ra], #4`
Post-indexing - Register	`LDR Rd, [Ra], Ro`

LDR Rd, [Ra] ST
R Rs, [Ra]

在这种情况下，Rd 表示目标寄存器，Rs 表示源寄存器，Ra 表示包含地址的寄存器。（注意，内存地址必须与数据类型对齐：字节可以从任何地址加载值，半字从每个偶数地址加载，以此类推。）

LDR 和 STR 都支持多种寻址模式，如图 10.7 所示。首先，LDR 可用于将一个完整的 32 位字面值（或标签地址）加载到寄存器中。（完整说明见下一节。）与 X86 不同，这里不存在一条能够从给定内存地址直接加载值的单条指令。要实现这一点，必须先将地址加载到寄存器中，然后再执行一次寄存器间接加载：

LDR r1, =x LDR
r2, [r1]

有多种更复杂的寻址方式可供选择，这些方式有助于在高级程序中实现指针、数组和结构体。预索引模式将一个常数（或寄存器）加到基寄存器上，然后从计算得到的地址加载数据：

LDR r1, [r2, #4] ; 从地址 r2 + 加载 4 LDR r1, [r2, r3] ; 从地址
r2 + r3 加载

也可以通过附加叹号 (!) 字符来写回基址寄存器。这表示在加载地址后，计算得到的地址应保存到基址寄存器中：

LDR r1, [r2, #4]! ; 从 r2 + 4 加载，然后 r2 += 4 LDR r1, [r2, r3]! ; 从 r2
+ r3 加载，然后 r2 += r3

后索引模式执行的是相同的操作，但顺序相反。首先，从基址寄存器进行加载，然后基址寄存器被递增：

```
LDR r1, [r2], #4 ; 从 r2 加载，然后 r2 += 4 LDR r1, [r2], r3 ; 从 r
2 加载，然后 r2 += r3
```

这些复杂的前索引和后索引模式使得能够用单条指令实现诸如 b = a++ 这样的惯用 C 操作。相应的模式也同样适用于 STR 指令。

绝对地址（以及其他大型字面量）在 ARM 中要复杂一些。由于每条指令都必须装入一个 32 位字中，无法在包含操作码的同时把一个 32 位地址装入一条指令。相反，较大的字面量必须存放在字面量池中，字面量池是程序代码段内的一小块数据区域。可以使用 PC 相对的加载指令从池中加载字面量，该指令可以从加载指令起引用 ±4096 字节。这会导致多个小的字面量池散布在整个程序中，从而使每一个池都靠近使用它的加载指令。

ARM 汇编器通常会向用户隐藏这种复杂性。当一个标签或较大的字面量以前缀等号标记时，这表示汇编器应将所标记的值放入字面量池中，并相应地生成一条 PC 相对寻址的指令。

例如，下面的指令将 x 的地址加载到 r1 中，然后将 x 的值加载到 r2 中：

```
LDR r1, =x LDR
r2, [r1]
```

将被展开为一次从相邻的字面量池中加载 x 的地址的操作，随后加载 x 的值：

```
LDR r1, .L1  LD
R r2, [r1]  B .end
.L1:.word x  .end:
```

10.4.3 Basic Arithmetic

ARM 提供三地址算术指令用于寄存器。ADD 和 SUB 指令将结果寄存器指定为第一个参数，并对第二个和第三个参数进行运算。第三个操作数可以是一个 8 位常数，或者是应用了可选移位的寄存器。变体

启用进位输入将把CPSR的C位加到结果中。所有四种操作都可以加上可选的S后缀，这将在完成时设置条件标志（包括进位）。

Instruction	Example
Add	`ADD Rd, Rm, Rn`
Add with carry-in	`ADC Rd, Rm, Rn`
Subtract	`SUB Rd, Rm, Rn`
Subtract with carry-in	`SBC Rd, Rm, Rn`

乘法的工作方式基本相同，不同之处在于，乘法两个 32 位数可能会得到一个 64 位的结果。普通的 MUL 会丢弃结果的高位，而 UMULL 会将 64 位结果存放在两个 32 位寄存器中。带符号的变体 SMULL 会根据需要对高位寄存器进行符号扩展。

Instruction	Example
Multiplication	`MUL Rd, Rm, Rn`
Unsigned Long Multiplication	`UMULL RdHi, RdLo, Rm, Rn`
Signed Long Multiplication	`SMULL RdHi, RdLo, Rm, Rn`

ARM 中没有除法指令，因为除法无法在单个流水线周期内完成。相反，当需要除法时，它通过调用标准库中的外部函数来实现。这部分留给读者作为练习。

逻辑指令在结构上与算术指令非常相似。我们有按位与、按位或、按位异或和按位清除，这相当于将第一个值与第二个值的反转进行按位与。移位非 MVN 指令在将数据从一个寄存器移动到另一个寄存器的同时执行按位非操作。

Instruction	Example
Bitwise-And	`AND Rd, Rm, Rn`
Bitwise-Or	`ORR Rd, Rm, Rn`
Bitwise-Xor	`EOR Rd, Rm, Rn`
Bitwise-Bit-Clear	`BIC Rd, RM, Rn`
Move-Not	`MVN Rd, Rn`

10.4.4 Comparisons and Branches

CMP指令比较两个值，并设置CPSR中的N（负数）和Z（零）标志，以供后续指令读取。在比较寄存器和立即数的情况下，立即数必须是第二操作数：

```
CMP Rd, Rn
CMP Rd, #imm
```

图 10.8：ARM 分支指令

Opcode	Meaning		
B	Branch Always	BL	Branch and Link
BX	Branch and Exchange	BLX	Branch-Link-Exchange
BEQ	Equal	BVS	Overflow Set
BNE	Not Equal	BVC	Overflow Clear
BGT	Greater Than	BHI	Higher (unsigned >)
BGE	Greater Than or Equal	BHS	Higher or Same (uns. >=)
BLT	Less Than	BLO	Lower (unsigned <)
BLE	Less Than or Equal	BLS	Lower or Same (uns. <=)
BMI	Negative	BPL	Positive or Zero

此外，可以在算术指令后附加“S”以类似的方式更新CPSR。例如，SUBS将减去两个值，存储结果，并更新CPSR。

各种分支指令会查阅先前设置的CPSR值，然后在适当的标志位被设置时跳转到给定标签。无条件分支通过简单的B指令来指定。

例如，要从零数到五：

```
MOV r0, #0 loop:
ADD r0, r0, 1 CMP r0, #5 BLT
loop
```

并且有条件地为全局变量 y 赋值：如果 x 大于 0，则为 10，否则为 20。

```
LDR r0, =x LDR
r0, [r0] CMP r0, #0 BGT .L1
.L0: MOV r0, #20 B .L2 .L1:
MOV r0, #10 .L2: LDR r1, =
y STR r0, [r1]
```

分支并链接（BL）指令用于实现函数调用。BL 将链接寄存器设置为下一条指令的地址，然后跳转到给定的标签。随后，链接寄存器被用作返回

函数结束时的地址。BX 指令跳转到寄存器中给定的地址，最常用于通过跳转到链接寄存器从函数调用中返回。BLX 执行一次分支并链接到寄存器给定的地址，可用于调用函数指针、虚方法或任何其他间接跳转。

ARM 指令集的一个特殊特性是条件执行。每条指令字中都有一个 4 位字段，用于指示 16 种可能的条件之一；如果条件不成立，该指令将被忽略。上面所示的各种条件分支类型，本质上只是对普通分支（B）指令应用了不同的条件。几乎所有指令都可以加上相同的两个字母后缀。

例如，假设我们有如下代码片段，它会根据哪个更小来递增 a 或 b：

如果(a<b) { a++; } 否则 { b++; }

与其使用分支和标签将其实现为控制流，我们可以直接让这两个加法各自依赖于先前比较的结果而有条件地执行。满足条件的那个将被执行，另一个则被跳过。假设 a 和 b 分别存放在 r0 和 r1 中：

CMP r0, r1 ADDLT r0,
r0, #1 ADDGE r1, r1, #
1

10.4.5 The Stack

栈是一种辅助数据结构，主要用于记录程序的函数调用历史，以及那些无法放入寄存器的局部变量。按照约定，栈从高地址向低地址 *downward* 增长。sp 寄存器称为栈指针，用于跟踪栈中最底部的项。

要将 r0 寄存器压入栈中，我们必须从 sp 中减去该寄存器的大小，然后将 r0 存储到 sp 指向的位置：

SUB sp, sp, #4 STR
r0, [sp]

另外，也可以通过一条指令，利用预索引和回写来完成：

将 r0 存储到 [sp, #-4]!

PUSH 伪指令完成相同的操作，但还可以将任意数量的寄存器（以位掩码编码）压入栈中。使用花括号来表示要压入的寄存器列表：

图10.9：ARM 调用约定摘要

- 前四个参数被放置在寄存器 r0、r1、r2 和 r3 中。
- 额外的参数会以相反的顺序压入栈中。
- 如果需要，调用者必须保存 r0-r3 和 r12。
- 调用者必须始终保存 r14，即链接寄存器。
- 如有需要，被调用者必须保存 r4–r11 寄存器。
- 结果存放在 r0 中。

压栈 {r0,r1,r2}

从栈中弹出一个值则是相反的操作：

LDR r0, [sp] ADD s
p, sp, #4

再次，这可以通过一条指令完成：

LDR r0, [sp], #4

并且，要一次性弹出一组寄存器：

弹出 {r0,r1,r2}

与 X86 不同，只要遵守数据对齐要求，从一个字节到一个双字的任何数据项都可以被压入栈中。

10.4.6 Calling a Function

ARM 使用一种由 ARM-Thumb 过程调用标准（ATPCS）[4] 描述的寄存器调用约定，其概要如图 10.9 所示。

要调用一个函数，将所需的参数放入寄存器 r0-r3 中，保存（当前）链接寄存器的值，然后使用 BL 指令跳转到该函数。返回时，恢复链接寄存器的先前值，并检查寄存器 r0 中的结果。

例如，下面的 C 函数：

int x=0; int y=10; int main() { x = printf("值: %
d\n",y); }

图 10.10：ARM 寄存器分配

Register	Purpose	Who Saves?
r0	argument 0 / result	not saved
r1	argument 1	CALLER saves
r2	argument 2	CALLER saves
r3	argument 3	CALLER saves
r4	scratch	callee saves
...	...	...
r10	scratch	callee saves
r11	frame pointer	callee saves
r12	intraprocedure	CALLER saves
r13	stack pointer	callee saves
r14	link register	CALLER saves
r15	program counter	saved in link register

在 ARM 中将变为如下：

.data x: .word 0 y: .word 10 S0: .ascii "值: %d\0
12\000"

.text main: LDR r0, =S0 @ 加载 S0 的地址 LDR r1, =y @
加载 y 的地址 LDR r1, [r1] @ 加载 y 的值 PUSH {ip,lr} @
保存寄存器 BL printf @ 调用 printf POP {ip,lr} @ 恢复寄
存器 LDR r1, =x @ 加载 x 的地址 STR r0, [r1] @ 将返回
值存储到 x .end

10.4.7 Defining a Leaf Function

因为函数参数是通过寄存器传递的，因此很容易编写一个叶子函数，该函数计算一个值而不调用任何其他函数。例如，以下函数的代码：

平方：函数 整数 (x：整数) = { 返回 x*x;

}

可能就这么简单：

.global 正方形
平方：
MUL r0, r0, r0 @ 将参数与自身相乘 BX lr @ 返回给调用者

不幸的是，这对于想要调用其他函数的函数不起作用，因为我们没有正确设置堆栈。一般情况需要更复杂的方法。

10.4.8 Defining a Complex Function

一个复杂的函数必须能够调用其他函数并计算任意复杂度的表达式，然后以原始状态完整返回给调用者。考虑以下一个接受三个参数并使用两个局部变量的函数示例：

函数：
PUSH {fp} @ 保存帧指针MOV fp, sp @ 设置新的帧指针PUSH {r0,r1,r2} @ 将参数保存在栈上SUB sp, sp, #8 @ 分配另外两个局部变量PUSH {r4-r10} @ 保存被调用者保存的寄存器

@@@ 函数体在此处 @@@

POP {r4-r10} @ 恢复被调用者保存的寄存器MOV sp, fp @ 重置栈指针POP {fp} @ 恢复先前的帧指针BX lr @ 返回到调用者

通过这种方法，我们确保能够将寄存器中的所有值保存到栈中，并确保不会丢失任何数据。完成之后，栈的结构看起来与 X86 的栈非常相似，只是在栈上额外存放了一些被调用者保存的寄存器。

下面是一个将所有内容整合在一起的完整示例。假设你有一个如下所定义的 B-minor 函数：

计算：函数 整数 (a：整数, b：整数, c：整数) = { x：整数 = a+b+c ； y：整数 = x*5； 返回 y； }

图 10.11：示例 ARM 栈帧

Contents	Address	
Saved r12	[fp, #8]	
Old LR	[fp, #4]	
Old Frame Pointer	[fp]	← fp points here
Argument 2	[fp, #-4]	
Argument 1	[fp, #-8]	
Argument 0	[fp, #-12]	
Local Variable 1	[fp, #-16]	
Local Variable 0	[fp, #-20]	
Saved r10	[fp, #-24]	
Saved r9	[fp, #-28]	
Saved r8	[fp, #-32]	
Saved r7	[fp, #-36]	
Saved r6	[fp, #-40]	
Saved r5	[fp, #-44]	
Saved r4	[fp, #-48]	← sp points here

完整的函数翻译在下一页。请注意，这只是构建有效堆栈帧的一种方法，适用于函数定义。其他方法也是有效的，只要函数一致地使用堆栈帧。例如，被调用者可以首先将所有参数和临时寄存器压入堆栈，然后在其下方为局部变量分配空间。（当然，在函数退出时必须使用相反的过程。）

另一种常见的方法是被调用方在将参数和局部变量压入栈之前，将{fp,ip,lr,pc}压入栈中。虽然实现该功能并非严格要求，但它通过堆栈回溯的形式提供了额外的调试信息，使调试器能够向后查看调用栈并轻松重建程序的当前执行状态。

给出的代码是正确的，但相对保守。事实证明，这个特定的函数并不需要使用寄存器 r4 和 r5，因此没有必要保存和恢复它们。同样，我们本可以将参数保留在寄存器中，而不是将其保存到栈中。结果本可以直接计算到 r0 中，而不是先保存到一个局部变量。这些优化在手写代码时很容易做到，但在编写编译器时就不那么容易了。

对于你第一次尝试构建编译器，所创建的代码（可能）不会非常高效，如果每个语句都是独立翻译的。函数的前导部分必须保存所有寄存器，因为它不知道 *a priori* 哪些寄存器将在后续使用。同样，一个计算值的语句必须将其保存回局部变量，因为

图 10.12：完整的 ARM 示例

.global compute compute: @@@@@@@@@@@@@@@@@@@@ 函数的前导部分设置栈 PUSH {fp} @ 保存帧指针 MOV fp, sp @ 设置新的帧指针 PUSH {r0,r1,r2} @ 保存参数到栈 SUB sp, sp, #8 @ 分配两个本地变量 PUSH {r4-r10} @ 保存被调用者保存的寄存器

@@@@@@@@@@@@@@@@@@@@@@@@@@@@ LDR r0, [fp,#-12] @ 将参数 0 (a) 加载到 r0 LDR r1, [fp,#-8] @ 将参数 1 (b) 加载到 r1 LDR r2, [fp,#-4] @ 将参数 2 (c) 加载到 r2 ADD r1, r1, r2 @ 将参数相加 ADD r0, r0, r1 STR r0, [fp,#-20] @ 将结果存储到局部变量 0 (x) LDR r0, [fp,#-20] @ 将局部变量 0 (x) 加载到寄存器 MOV r1, #5 @ 将 5 移动到寄存器 MUL r2, r0, r1 @ 将两者相乘并存入 r2 STR r2, [fp,#-16] @ 将结果存储到局部变量 1 (y) LDR r0, [fp,#-16] @ 将局部变量 1 (y) 加载到结果

@@@@@@@@@@@@@@@@@@@@@@ 结束语功能恢复栈 POP {r4-r10} @ 恢复被调用者保存的寄存器 MOV sp, fp @ 重置栈指针 POP {fp} @ 恢复先前的帧指针 BX lr @ 返回给调用者

它事先并不知道该局部变量是否会被用作返回值。我们将在第12章关于优化中进一步探讨这些问题。

10.4.9 64-bit Differences

64 位 ARMv8-A 架构提供两种执行模式：A32 模式支持上述描述的 32 位指令集，而 A64 模式支持一种新的 64 位执行模型。这使得在具备相应操作系统支持的情况下，64 位 CPU 能够同时执行 32 位和 64 位程序的混合。尽管与 A32 在二进制层面不兼容，A64 模型仍遵循了大部分相同的架构原则，只是在若干关键方面有所变化：

字长。A64 指令仍为固定的 32 位，但寄存器和地址计算为 64 位。

寄存器。A64 有 32 个 64 位寄存器，命名为 x0–x31。x0 是一个专用的零寄存器：读取时始终返回零值，写入时不起任何作用。x1–x15 是通用寄存器，x16 和 x17 用于进程间通信，x29 是帧指针，x30 是链接寄存器，x31 是栈指针。（程序计数器不能从用户代码直接访问。）不使用数据类型后缀时，可以通过将寄存器命名为 w# 来表示一个 32 位值。

指令。A64 指令在很大程度上与 A32 相同，使用相同的助记符，但有一些差异。条件谓词不再是每条指令的一部分。相反，所有条件码都必须执行一次显式的 CMP，然后再进行条件分支。LDM/STM 指令以及伪指令 PUSH/POP 不再可用，必须用一系列显式的加载和存储来替代。（可以通过使用 LDP/STP 来提高效率，它们可以成对加载和存储寄存器。）

调用约定。调用函数时，前八个参数放置在寄存器 x0–x7 中，其余参数压入栈中。调用者必须保存寄存器 x9–x15 和 x30，而被调用者必须保存 x19–x29。（标量）返回值放在 x0 中，而扩展返回值由 x8 指向。

10.5 延伸阅读

本章为你简要介绍了 X86 和 ARM 架构的核心特性，足以让你以最直接的方式编写简单程序。然而，你当然还需要查阅各条指令的具体细节，才能更好地理解它们的选项和限制。现在，你已经准备好阅读详细的参考手册，并在构建编译器的过程中随手查阅它们:

1. Intel64 和 IA-32 架构软件开发人员手册。英特尔公司，2017。http://www.intel.com/content/www/us/en/processors/ architectures-software-developer-manuals.html

2. System V 应用二进制接口，Jan Hubicka、Andreas Jaeger、Michael Matz 和 Mark Mitchell（编辑），2013 年。https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

3. ARM 架构参考手册 ARMv8。ARM Limited，2017 年。https://static.docs.arm.com/ddi0487/bb/DDI0487B_b_ armv8_arm.pdf。

4. ARM-THUMB 过程调用标准。ARM Limited，2000年。 https://developer.arm.com/documentation/ihi0042/f/

第11章——代码生成

11.1 引言

恭喜，你已经进入编译器的最后阶段！在完成对源代码的扫描和解析、构建 AST、进行类型检查以及生成中间表示之后，我们现在已经准备好生成一些代码了。

首先，我们将采用一种 na¨ve 的代码生成方法，在这种方法中我们将程序的每个元素孤立地考虑。每个表达式和语句都将作为独立单元生成，不参考其相邻部分。这种方法简单直接，但较为保守，会导致生成大量非最优代码。不过它是可行的，并为你思考更复杂的技术提供一个起点。

这些示例将聚焦于 X86-64 汇编代码，但根据需要将它们改写为 ARM 或其他汇编语言并不困难。与前几个阶段一样，我们将为程序的每个元素定义一个方法。`decl codegen` 将为声明生成代码，调用 `stmt codegen` 为语句生成代码，`expr codegen` 为表达式生成代码，等等。这些关系如图 11.1 所示。

一旦你掌握了这种代码生成的基本方法，你就可以准备好进入*following*章，在该章中我们将探讨用于生成更高度优化代码的更复杂方法。

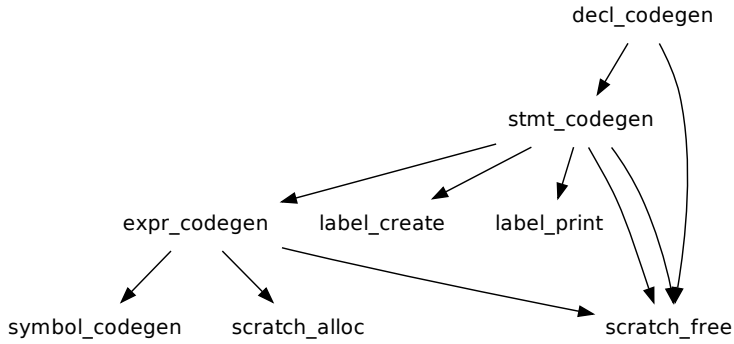
11.2 支持函数

在生成一些代码之前，我们需要设置一些辅助函数来跟踪若干细节。为了生成表达式，你将需要一些临时寄存器，用于在运算符之间保存中间值。按照以下接口编写三个函数：

```
int scratch_alloc(); void scratch_free( int r ); const c
har * scratch_name( int r );
```

回顾第10章，你可以看到我们为每个寄存器都预留了用途：有些用于函数参数，有些用于栈帧

图 11.1：代码生成函数



管理，并且有一些可用于临时值。将这些临时寄存器整理成如下这样的表格：

r	0	1	2	3	4	5	6
name	%rbx	%r10	%r11	%r12	%r13	%r14	%r15
inuse	X		X				

然后，编写 scratch alloc，在表中查找一个未使用的寄存器，将其标记为正在使用，并返回寄存器编号 r。scratch free 应将指定的寄存器标记为可用。scratch name 应在给定编号 r 的情况下返回寄存器的名称。耗尽 scratch 寄存器是可能的，但正如我们下面将看到的那样，这种情况不太可能发生。现在，如果 scratch alloc 无法找到空闲寄存器，只需发出一条错误消息并停止运行。

接下来，我们需要生成大量唯一的匿名标签，用于指示跳转和条件分支的目标。编写两个函数来生成并显示标签：

```
int label_create(); const char * label_name( int label );
```

label create 只是将一个全局计数器递增并返回当前值。label name 以字符串形式返回该标签，- 因此标签 15 表示为 ".L15"。

最后，我们需要一种方法将程序中的符号映射到表示这些符号的汇编语言代码。为此，编写一个函数来生成符号的地址：

```
const char * symbol_codegen( struct symbol *s );
```

该函数返回一个字符串，它是指令的一个片段，用于表示给定符号所需的地址计算。编写符号的代码生成时，应首先检查符号的作用域。全局变量很简单：其在汇编语言中的名称与源语言中的名称相同。如果你有一个表示全局变量 count: integer 的符号结构，那么符号代码生成应当直接返回 count。

表示局部变量和函数参数的符号应返回一个地址计算，用于计算该局部变量或参数在栈中的位置。这一工作的基础在类型检查阶段已奠定，在该阶段中，你为每个参数和每个局部变量分配了唯一的序列号。

例如，假设你有如下的函数定义 Understood. Please provide th

```
f: 函数 空 ( x: 整数, y: 整数 ) =
{ z: 整数 = 10; 返回 x + y + z;
}
```

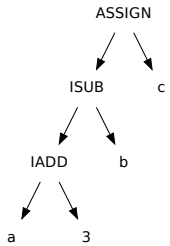
在这种情况下，x 的参数位置为 0，y 的位置为 1，而 z 是局部变量 0。现在回看图 10.5，它展示了 X86-64 处理器上的栈布局。参数位置 0 位于地址 -8(%rbp)，参数 1 位于 -16(%rbp)，随后是局部变量 0，位于 -24(%rbp)。

鉴于此，你现在可以扩展符号代码生成，使其在仅知道其在栈帧中的位置的情况下，返回一个描述局部变量和参数精确栈地址的字符串。

11.3 生成表达式

生成表达式的汇编代码的基本方法是对 AST 或 DAG 进行后序遍历，并为每个节点发出一条或多条指令。其主要思想是跟踪每个中间值存放在哪些寄存器中。为此，在 AST 或 DAG 节点结构中添加一个 reg 字段，用于保存由 scratch alloc 返回的寄存器编号。当访问每个节点时，发出一条指令，并将包含该值的寄存器编号放入 reg 字段。当该节点不再需要时，调用 scratch free 释放它。

假设我们要为以下 DAG 生成 X86 代码，其中 a、b 和 c 是全局整数：




```
1. MOVQ  a, R0
2. MOVQ $3, R1
3. ADDQ R0, R1
4. MOVQ  b, R0
5. SUBQ R0, R1
6. MOVQ R1, c
```

图 11.2：从 DAG 生成 X86 代码的示例

A po st-order 遍历将按以下顺序访问这些节点 订单：

1. 访问 a 节点。调用 scratch alloc 分配一个新的寄存器（0），并将其保存在 node->reg 中。然后发出指令 MOVQ a, R0，将该值加载到寄存器 0 中。[1] 2. 访问 3 节点。调用 scratch alloc 分配一个新的寄存器（1），并发出指令 MOVQ $3, R1。3. 访问 IADD 节点。通过检查该节点的两个子节点，我们可以看到它们的值分别存储在寄存器 R0 和 R1 中，因此我们发出一条将它们相加的指令：ADDQ R0, R1。这是一条破坏性的双地址指令，结果保留在 R1 中。R0 不再需要，因此我们调用 scratch free(0)。

4. 访问 b 节点。调用 scratch alloc 分配一个新的寄存器（0），并生成 MOVQ b, R0。

5. 访问 ISUB 节点。发出 SUBQ R0, R1，将结果保留在 R1 中，并释放寄存器 R0。

6. 访问 c 节点，但不要发出任何内容，因为它是赋值的目标。

7. 访问 ASSIGN 节点并生成 MOVQ R1, c。

[1] Note that the actual name of register R0 is `scratch_name(0)`, which is `%rbx`. To keep the example clear, we will call them R0, R1, etc. for now.

这里是使用由临时名称提供的实际寄存器名称的同一段代码：

MOVQ a, %rbx MOV
Q $3, %r10 ADDQ %
r10, %rbx MOVQ b,
%rbx SUBQ %rbx, %
r10 MOVQ %r10, c

以下是如何在代码中实现它。编写一个名为 expr codegen 的函数，它首先递归地为其左、右子节点调用 expr codegen。这将使每个子节点生成代码，使结果保留在 reg 字段中标注的寄存器编号里。随后，当前节点使用这些寄存器生成代码，并释放不再需要的寄存器。图11.3 给出了这种方法的一个骨架。

基本流程还需要进行一些额外的改进。

首先，并非所有符号都是简单的全局变量。当符号构成指令的一部分时，使用符号 codegen 来返回给出该符号具体地址的字符串。例如，如果 a 是该函数的第一个参数，那么序列中的第一条指令将会像下面这样：

MOVQ -8(%rbp), %rbx

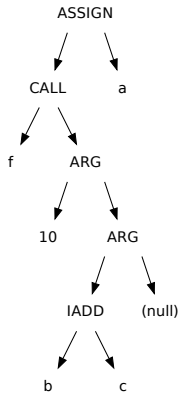
其次，DAG 中的一些节点可能需要多条指令，以处理指令集的特殊性。你可能还记得，X86 的 IMUL 只接受一个参数，因为第一个参数始终是 %rax，结果也始终放在 %rax 中，而溢出部分放在 %rdx 中。为了执行乘法，我们必须将一个子节点寄存器移动到 %rax，用另一个子节点寄存器进行乘法运算，然后再将结果从 %rax 移动到目标的临时寄存器中。例如，表达式 (x*10) 将被翻译成如下形式：

MOV $10, %rbx MO
V x, %r10 MOV %rb
x, %rax IMUL %r10
MOV %rax, %r11

当然，这也意味着在乘法执行期间，%rax 和 %rdx 不能用于其他用途。鉴于我们有大量可用的临时寄存器，我们将在基本代码生成器中不将 %rdx 用于任何其他用途。

```
void expr_codegen( struct expr *e ) { if(!e) return; switch(e->kind) { // 叶子节点：分配寄存器并加载值。 case EXPR_NAME: e->reg = scratch_alloc(); printf("MOVQ %s, %s\n", symbol_codegen(e->symbol), scratch_name(e->reg)); break; // 内部节点：先生成子节点，然后将它们相加。 case EXPR_ADD: expr_codegen(e->left); expr_codegen(e->right); printf("ADDQ %s, %s\n", scratch_name(e->left->reg), scratch_name(e->right->reg)); e->reg = e->right->reg; scratch_free(e->left->reg); break; case EXPR_ASSIGN: expr_codegen(e->left); printf("MOVQ %s, %s\n", scratch_name(e->left->reg), symbol_codegen(e->right->symbol)); e->reg = e->left->reg; break; . . . } }
```

图 11.3：表达式生成骨架




```
MOVQ  $10, %rbx
MOVQ  b, %r10
MOVQ  c, %r11
ADDQ  %r10, %r11
MOVQ  %r11, %rsi
MOVQ  %rbx, %rdi
PUSHQ %r10
PUSHQ %r11
CALL  f
POPQ  %r11
POPQ  %r10
MOVQ  %rax, %rbx
MOVQ  %rbx, a
```

图 11.4：为函数调用生成代码

第三，我们如何调用一个函数？回想一下，一个函数由一个单一的 CALL 节点表示，其各个参数位于由 ARG 节点组成的一棵不平衡树中。图 11.4 给出了一个表示表达式 a=f(10,b+c) 的 DAG 示例。

代码生成器必须对每个 ARG 节点进行求值，计算其每个左子节点的值。如果机器采用栈调用约定，那么每个 ARG 节点都对应一次对栈的 PUSH 操作。如果机器采用寄存器调用约定，则先生成所有参数，然后在所有参数生成完成后，将每个参数复制到相应的参数寄存器中。接着，在保存所有调用者保存寄存器之后，针对函数名发出 CALL 指令。当函数调用返回时，将返回寄存器（%rax）中的值移动到一个新分配的临时寄存器中，并恢复调用者保存寄存器。

最后，请仔细注意表达式的副作用。每个表达式都有一个值，该值会被计算出来并保存在一个临时寄存器中，供其父节点使用。有些表达式还具有副作用，即除了返回值之外还会执行的动作。对于某些运算符，很容易忽略其中之一！

例如，表达式 (x=10) 产生一个值为 10 的 *value*，这意味着你可以在任何需要一个值的地方使用该表达式。这正是你能够写出 y=x=10 或 f(x=10) 的原因。该表达式还具有将值 10 存储到变量 x 中的 *side effect*。当你为 x=10 赋值生成代码时，一定要执行将 10 存入 x 的副作用（这一点显而易见），同时还要在表示该表达式值的寄存器中保留值 10。

11.4 生成语句

既然我们已经将表达式生成封装到一个单一的 expr codegen 函数中，就可以开始构建依赖于表达式的更大型代码结构了。stmt codegen 将为所有控制流语句生成代码。首先，按如下方式为 stmt codegen 编写一个骨架：

```
源文本：void stmt_codegen( struct stmt *s ) { if(!s)
return; switch(s->kind) { case STMT_EXPR: ... bre
ak; case STMT_DECL: ... break; ... }stmt_codegen(
s->next); }
```

图 11.5：语句生成器骨架

现在依次考虑每一种语句，从简单的情况开始。如果该语句描述的是一个局部变量的声明 STMT DECL，那么只需通过调用 decl codegen 来委托处理即可：

```
case STMT_DECL: decl_codegen(s->
decl); break;
```

由一个表达式（STMT EXPR）构成的语句只需要对该表达式调用 expr 的代码生成，然后释放包含该表达式顶层值的临时寄存器。（事实上，每次调用 expr 的代码生成时，都应释放一个临时寄存器。）

```
情况 STMT_EXPR：expr_codegen(s->expr);
scratch_free(s->expr->reg); 跳出;
```

返回语句必须对一个表达式求值，将其放入用于返回值的指定寄存器 %rax，然后跳转到函数尾声（epilogue），该尾声会展开栈并返回到调用点。（有关序言（prologue）的更多细节请参见下文。）

案例 STMT_RETURN:
expr_codegen(s->expr); printf("MOV %s, %%rax\n",scratch_name(s->expr->reg)); printf("JMP .%s_epilogue\n",function_name); scratch_free(s->expr->reg); break;

（细心的读者会注意到，这段代码需要知道包含该语句的函数名。你将不得不想办法把这些信息向下传递。）

控制流语句更有趣。先考虑我们希望输出的汇编语言长什么样是很有用的，然后再反向推导以获得生成它的代码。

这是一个条件语句的模板：

if (*expr* **)** {
true-statements
} **else** {
false-statements
}

要用汇编来表达这一点，我们必须先对控制表达式求值，使其结果存放在一个已知的寄存器中。然后使用一条 CMP 表达式来测试该值是否等于零（即为假）。如果表达式为假，则必须使用一条 JE（相等则跳转）指令跳转到语句的假分支。否则，程序将继续执行到语句的真分支。在真分支语句结束时，我们必须使用一条 JMP 指令跳过 else 体，跳转到整个语句的末尾。

expr
CMP $0,
register JE
false-label JMP
true-statements *done-label:*
false-label:
false-statements
done-label Understood. Please provide the source text you would like me to translate.

一旦你拥有了所需代码的骨架，编写代码生成器就容易了。首先，生成两个新的标签，然后对每个表达式调用 expr 代码生成， 对每个语句调用 stmt 代码生成，并根据需要替换几个额外的指令，以使整体结构完整。

案例 STMT_IF: 整数 else_label = label_create(); 整数 done_label = label_create(); expr_codegen(s->expr); printf("CMP $0, %s\n",scratch_name(s->expr->reg)); scratch_free(s->expr->reg); printf("JE %s\n",label_name(else_label)); stmt_codegen(s->body); printf("JMP %s\n",label_name(done_label)); printf("%s:\n",label_name(else_label)); stmt_codegen(s->else_body); printf("%s:\n",label_name(done_label)); break;

可以使用类似的方法来生成循环。以下是 for 循环的源模板：

for (*init-expr* **;** *expr* **;** *next-expr* **)** {
    body-statements
}

这是相应的汇编模板。首先，评估初始化表达式。然后，在每次循环迭代时，评估控制表达式。如果为假，则跳转到循环的结束。如果不是，执行循环体，然后评估下一个表达式。

    init-expr
*top-label***:**
    expr
    `CMP $0,` *register*
    `JE` *done-label*
    body-statements
    next-expression
    `JMP` *top-label*
done-label Understood. Please provide the source text you would like me to translate.

编写代码生成器作为留给读者的练习。请记住，for 循环中的三个表达式中的每一个都可以省略。如果

如果省略 *init-expr* 或 *next-expr*，它们不起任何作用。如果省略 *expr*，则假定为 true。[2]

许多语言都有循环控制结构，如 continue; 和 break;。在这些情况下，编译器必须跟踪与当前正在生成的循环相关联的标签，并分别将它们转换为跳转到顶部标签或完成标签的 JMP。

B-Minor 中的 print 语句是命令式语句的一种特殊情况，其行为会根据要打印的表达式类型而有所不同。例如，下面的 print 语句在打印整数、布尔值和字符串时必须生成略有不同的代码：

```
i：整数 = 10； b：布尔
= 真； s：字符串 = "\n"
； 打印 i, b, s；
```

显然，并不存在与整数显示直接对应的简单汇编代码。在这种情况下，一种常见的方法是将任务归约为我们已经熟悉的抽象。整数、布尔值、字符串等的打印可以委托给明确执行这些操作的函数调用。因此，为 print i 、b、s 生成的代码等价于如下内容：

```
print_integer(i); print_bo
olean(b); print_string(s);
```

因此，为了生成一条 print 语句，我们只需为每个要打印的表达式生成代码，使用 expr typecheck 确定表达式的类型，然后生成相应的函数调用。

当然，这些函数中的每一个都必须先编写出来，然后链接到每一个 B-Minor 程序实例中，以便在需要时可用。这些函数，以及任何其他必要的函数，统称为 B-Minor 程序的运行时库。一般来说，编程语言的抽象层次越高，所需要的运行时支持就越多。

[2] Yes, each of the three components of a for-loop are expressions. It is customary that the first has a side effect of initialization (`i=0`), the second is a comparison (`i<10`), and the third has a side effect to generate the next value (`i++`), but they are all just plain expressions.

11.5 条件表达式

既然你已经知道如何生成控制流语句，我们必须回到表达式生成的一个方面。条件表达式（小于、大于、等于等）比较两个值并返回一个布尔值。它们最常出现在控制流表达式中，但也可以作为简单的值使用，如下所示：

b: 布尔值 = x < y;

问题在于没有单一指令可以简单地执行比较并将布尔结果放入寄存器中。相反，您必须走弯路，创建一个控制流结构，比较两个表达式，然后构建所需的结果。

例如，如果你有像这样的条件表达式：

left-expr	<	*right-expr*

然后根据此模板生成代码：

left-expr
right-expr

CMP *left-register*, *right-register* JLT *true-label* MOV *false, result-register* JMP *done-label* MOV *true, result-register* *true-label:* *done-label:*

当然，对于不同的条件运算符，应在合适的位置使用不同的跳转指令。稍作变化，你也可以用同样的方法来实现许多语言中常见的三元条件运算符 (x?a:b)。

这种方法的一个有趣结果是，如果你以显而易见的方式为类似 if(x>y){...} 的 if 语句生成代码，那么你最终会在汇编代码中得到 *two* 个条件结构。第一个条件计算 x>y 的结果，并将其放入一个寄存器中。第二个条件将该结果与零进行比较，然后跳转到语句的真分支或假分支。通过稍加细致的编码，你可以检查这种常见情况，并生成一个单一的条件语句，对表达式进行求值，并使用一次条件跳转来实现该语句。

11.6 生成声明

最后，输出整个程序只是遍历每个代码或数据的声明并生成其基本结构的问题。声明可分为三种情况：全局变量声明、局部变量声明以及全局函数声明。（B-Minor 不允许局部函数声明。）

全局数据声明本质上只是发出一个标签，并配合一个合适的指令来保留所需的空间，以及在需要时提供一个初始化器。例如，以下是在全局作用域中的这些 B-Minor 声明：

i: 整数 = 10; s: 字符串 = "hello"; b: 数组 [4] 布尔 = {true, false, true, fals
e};

应产生以下输出指令： 翻译文本：

.data
i: .quad 10 s: .string "你好" b:
.quad 1, 0, 1, 0

请注意，全局变量声明只能由常量值（而非一般表达式）进行初始化，正是因为程序的数据段只能包含常量（而不能包含代码）。如果程序员不小心将代码放入初始化器中，那么类型检查器应当在代码生成开始之前就发现这一点并报错。

生成局部变量声明要容易得多。（这仅在函数声明内部由语句代码生成调用声明代码生成时才会发生。）在这里，可以假定函数序言已经为局部变量建立了空间，因此不需要进行任何栈操作。然而，如果变量声明包含初始化表达式（x:integer=y*10;），那么你必须为该表达式生成代码，将其存入局部变量，并释放寄存器。

函数声明是最后一块拼图。要生成一个函数，你必须先发出一个带有函数名的标签，然后是函数的序言。序言必须考虑参数和局部变量的数量，并在栈上分配相应大小的空间。接下来是函数体，随后是函数的尾声。尾声应当具有一个唯一的标签，以便返回语句可以方便地跳转到那里。

11.7 练习

1. 编写一个合法的表达式，使其在使用本章所描述的技术时耗尽六个可用的临时寄存器。一般来说，为任意表达式树生成代码需要多少个寄存器?

2. 在使用寄存器调用约定时，为什么有必要在将这些值移动到参数寄存器之前，先为函数参数 *all* 生成值?

3. 全局变量声明可以具有非常量的初始化表达式吗？解释原因。

4. 假设 B-Minor 包含一个 switch 语句。勾勒出两种不同的用于实现 switch 的汇编语言模板。

5. 按照本章所概述的内容，为 X86-64 架构编写完整的代码生成器。

6. 编写若干测试程序来测试 B-Minor 的各个方面，然后使用你的编译器对它们进行构建、测试并运行。

7. 将你的编译器在测试程序上的汇编输出与像 gcc 这样的生产级编译器在用 C 编写的等价程序上的输出进行比较。你看到了哪些差异?

8. 为你的编译器添加一个额外的代码生成器，使其能够生成不同的汇编语言（如 ARM）或中间表示（如 LLVM）。描述在此过程中所需的任何方法上的改变。

第12章——优化

12.1 概述

使用上一章所示的基本代码生成策略，你可以构建一个能够生成完全可用、可运行代码的编译器。然而，如果你检查编译器的输出，就会发现其中存在许多明显的低效之处。这源于这样一个事实：基本的代码生成策略是将每个程序元素孤立地考虑的，因此在把它们连接在一起时，必须采用最保守的策略。

在高级语言发展的早期，在优化策略尚未普及之前，编译器生成的代码被普遍认为不如人类手写的代码。如今，现代编译器拥有大量优化技术，并且对底层体系结构具有非常深入的了解，因此编译后的代码通常（但并非总是）优于人类编写的代码。

优化可以在编译器的多个阶段应用。通常，最好在可能的最高抽象层次上解决问题。例如，一旦我们生成了具体的汇编代码，我们能做的最多就是消除一些冗余指令。但是，如果我们使用线性中间表示（IR），可以通过智能寄存器分配加速一段较长的代码序列。如果我们在有向无环图（DAG）或抽象语法树（AST）层次工作，我们可以消除整块未使用的代码。

优化可以在程序的不同范围内发生。局部优化是指仅限于单个基本块的变化，基本块是没有任何控制流的直线代码序列。全局优化是指应用于整个函数（或过程、方法等）体的变化，函数体由一个控制流图组成，其中每个节点是一个基本块。跨过程优化则更大，考虑了不同函数之间的关系。通常，更大范围的优化更具挑战性，但也更有潜力提高程序的性能。

本章将带您了解一些常见的代码优化技术，您可以在项目编译器中实现它们，或者通过手动实现它们进行探索。但这只是一个简介：代码优化是一个非常庞大的主题，可能占据整本第二本教科书，而且至今仍是一个活跃的研究领域。如果本章内容引起了您的兴趣

如果是你，那么可以查看本章末尾引用的一些更高级的书籍和文章。

12.2 从整体视角看优化

作为一名程序员——编译器的使用者——保持对编译器优化的整体认识非常重要。大多数生产级编译器在使用默认选项运行时并不会执行任何重大的优化。这有几个原因。其一是编译时间：在最初开发和调试程序时，你希望能够快速地反复进行编辑、编译和测试。几乎所有优化都会需要额外的编译时间，而这在程序开发的初始阶段并没有帮助。另一个原因是，并非所有优化都会自动提升程序性能：它们可能会导致程序使用更多内存，甚至运行得更慢！还有一个原因是，优化可能会干扰调试工具，使得难以将程序的执行过程与源代码对应起来。

因此，如果你的程序运行速度不如你所期望，最好停下来，从第一性原理重新思考你的程序。需要牢记的两点建议：

- 审视你程序的整体算法复杂度：对于足够大的 n，二分查找（$O(\log n)$）总是会优于线性查找（$O(n)$）。改进程序的高层方法很可能比低层优化带来大得多的收益。

- 衡量你程序的性能。使用诸如 gprof [4] 这样的标准性能分析工具，精确测量你的程序大部分时间究竟花在了哪里，然后将精力集中在改进那一段代码上，可以通过重写它，或者启用相应的编译器优化来实现。

一旦你的程序从第一性原理出发被良好地编写完成，就该考虑启用特定的编译器优化了。大多数优化旨在针对代码中的某种行为模式，因此将代码写成这些易于识别的模式可能会对你有所帮助。事实上，下面讨论的大多数模式都可以在没有编译器帮助的情况下手工完成，这使你能够对不同的代码模式进行正面对比。

当然，本章中展示的代码片段都相当小，因此只有在程序中被执行大量次数时才具有意义。这通常发生在一个或多个嵌套循环之中，它们构成了程序的主要活动，通常被称为一次计算的内核。为了衡量例如将两个相乘的代价

```
#include <time.h>

struct timeval start,stop,elapsed;
gettimeofday(&start,0);

for(i=0;i<1000000;i++) {
    x = x * y;
}

gettimeofday(&stop,0);
timersub(&stop,&start,&elapsed);

printf("elapsed: %d.%06d sec",
       elapsed.tv_sec,elapsed.tv_usec);
```

图 12.1：计时快速操作

将值合并，在计时器间隔内执行一百万次，如图12.1所示。

注意：计时器不仅会统计循环中的操作，还会统计实现循环本身的代码，因此你需要通过减去空循环的运行时间来对结果进行归一化。

12.3 高级优化

12.3.1 Constant Folding

良好的编程实践通常鼓励在代码中广泛使用命名常量，以便明确值的目的和含义。例如，代替写出晦涩的数字86400，可以写出以下表达式来得到相同的数字：

const int 每分钟秒数=60; const int 每小时分钟数=60; const int 每天小时数=24;

每天秒数 = 每分钟秒数 * 每小时分钟数 * 每天小时数;

最终结果是相同的（86400），但代码更清晰地说明了该数字的目的和来源。然而，如果按字面翻译，程序将包含三个多余的常量、多个内存查找和两个乘法运算，以获得相同的结果。如果在复杂程序的内层循环中进行，这可能是一个显著的浪费。理想情况下，它

结构体 expr * expr_fold(结构体 expr *e) { expr_fold(e->left) expr_fold(e->right) 如果(e->left 和 e->right 都是常量) { f = expr_create(EXPR_CONSTANT); f->value = e->operator 应用于 e->left->value 和 e->right->value expr_delete(e->left); expr_delete(e->right); 返回 f; } 否则 { 返回 e; } }

图 12.2：常量折叠伪代码

应当使程序员能够进行详细的表达，而不会导致程序变得低效。

常量折叠是一种通过将多个常量合并为一个常量来转换表达式（或表达式的一部分）的技术。在语法树中，具有两个常量子节点的运算符节点可以被转换为一个单一节点，其运算结果被预先计算。该过程可以逐级向上级联，从而将复杂表达式简化为一个常量。实际上，它将程序的一部分工作从运行时移到了编译时。

这可以通过一个递归函数来实现，该函数对表达式树执行后序遍历。图 12.2 给出了在 AST 上进行常量折叠的伪代码。

必须小心，预先计算得到的结果 *precisely* 要与在运行时执行时得到的结果一致。这需要使用相同精度的变量，并处理诸如下溢、上溢以及除以零等边界情况。在这些情况下，通常应当强制产生编译期错误，而不是计算出一个意外的结果。

尽管常量折叠的效果看起来微不足道，但它往往是开启一系列后续优化的第一步。

12.3.2 Strength Reduction

强度削弱是一种将代价高昂操作的特殊情况转换为代价较低操作的技术。例如，用于浮点值指数运算的源代码表达式 x^y，通常被实现为对函数 pow(x,y) 的调用，而该函数可能通过泰勒级数展开来实现。然而，在 x2 这一特定情况下，我们可以用表达式 x*x 来替代，它实现了相同的功能。这避免了函数调用的额外开销以及多次循环迭代。类似地，乘以或除以任意 2 的幂，分别可以用按位左移或右移来替换。例如，x*8 可以替换为 x<<3。

一些编译器还包含用于标准库中操作强度减少的规则。例如，gcc的最近版本将把对printf(s)的调用替换为等效的puts(s)调用，其中s是一个常量字符串。在这种情况下，强度减少来自于减少必须链接到程序中的代码量：puts非常简单，而printf具有大量的功能和进一步的代码依赖关系。[1]

12.3.3 Loop Unrolling

考虑一种常见的结构：使用循环多次计算一个简单表达式的不同变体：

```
for(i=0;i<400;i++) {
   a[i] = i*2 + 10;
}
```

在任何汇编语言中，这只需要在循环体中使用几条指令来计算每次的 a[i] 值。但是，用于控制循环的指令将占据执行时间的很大一部分：每次循环时，我们必须检查是否 i<400，并跳回到循环的顶部。

循环展开是将一个循环转换为另一个循环的技术，后者具有更少的迭代次数，但每次迭代执行更多的工作。循环内的重复次数称为展开因子。上面的示例可以安全地转换为如下形式：

```
for(i=0;i<400;i+=4) {
   a[i] = i*2 + 10;
   a[i+1] = (i+1)*2 + 10;
   a[i+2] = (i+2)*2 + 10;
```

[1]While there is a logic to this sort of optimization, it does seem like an unseemly level of familiarity between the compiler and the standard library, which may have different developers and evolve independently.

```
a[i+3] = (i+3) *2 + 10;
```

或者这个：

```
for(i=0;i<400;i++) { a[i] = i*
2 + 10; i++; a[i] = i*2 + 10; i
++; a[i] = i*2 + 10; i++; a[i]
= i*2 + 10; }
```

增加每次循环迭代中的工作量可以节省一些对 i<400 的不必要求值，同时还能从指令流中消除分支，从而避免流水线停顿以及微处理器内部的其他复杂性。

但是循环应该展开多少呢？展开后的循环在每次迭代中可以包含 4、8、16 甚至更多个项。在极端情况下，编译器甚至可以完全消除循环，并将其替换为一个有限的语句序列，其中每条语句都具有一个常量值：

```
a[0] = 0 + 10; a[1] =
2 + 10; a[2] = 4 + 10
; . . .
```

随着展开因子的增加，循环结构中的不必要工作被消除。然而，增加的代码大小也有其代价：处理器必须不断从内存加载新的指令，而不是一遍又一遍地读取相同的指令。如果展开的循环导致工作集大于指令缓存，那么性能可能会比原始代码更差 *worse*。

因此，关于何时使用循环展开并没有固定的规则。编译器通常有全局选项可以展开循环，这些选项可以通过放置在特定循环前的 #pragma 进行修改。可能需要手动实验来获得针对特定程序的良好结果。[2]

12.3.4 Code Hoisting

有时，循环内部的代码片段在循环的每一次迭代中都是恒定的。在这种情况下，没有必要在每一次迭代中重新计算，

[2]The GCC manual has this to say about the `-funroll-all-loops` option: “This usually makes programs run more slowly.”

因此，代码可以被移动到循环之前的块中，这被称为代码提升。例如，示例中的数组索引在整个循环中是常量，可以在循环体之前计算一次：

```
for(i=0;i<400;i++) {
   a[x*y] += i;
}
```

```
t = x*y;
for(i=0;i<400;i++) {
   a[t] += i;
}
```

与循环展开不同，代码提升是一种相对温和的优化。该优化的唯一成本是计算结果必须在整个循环过程中占用一个临时位置，这会稍微增加寄存器压力或局部存储消耗。通过消除不必要的计算，这一点得到了补偿。

12.3.5 Function Inlining

函数内联是将函数调用替换为该函数调用效果的过程，直接嵌入代码中。这对于简短的函数特别有用，这些函数旨在提高代码的清晰度或模块化，但不会执行大量计算。例如，假设简单函数 quadratic 在循环中多次被调用，像这样：

```
int quadratic( int a, int b, int x ) { return a*x*x + b*x + 3
0; }for(i=0;i<1000;i++) { y = quadratic(10,20,i*2); }
```

设置参数和调用函数的开销可能超过在函数内部执行少量加法和乘法的成本。通过将函数代码内联到循环中，我们可以提高程序的整体性能。

函数内联最容易在高级表示形式上执行，例如AST或DAG。首先，必须复制函数体，然后必须将调用的参数代入。请注意，在这个评估级别，参数不一定是常量，而可能是包含未绑定值的复杂表达式。

例如，上述二次方程的调用可以通过表达式 (a*x*x+b*x+30) 进行替代，在绑定 a=10, b=20 和 x=i*2 的情况下。一旦进行这种替代，未绑定的变量如 i 就相对于调用二次方程的范围，而不是定义它的范围。生成的代码如下所示：

```
for(i=0;i<1000;i++) { y = 10*(i*2)*(i*2) + 20*(i*2) + 30
; }
```

这个例子凸显了函数内联的一个隐藏潜在成本：一个原本只计算一次、然后作为参数传递给函数的表达式（例如 i*2），现在会被多次计算，这可能会增加该表达式的成本。另一方面，这种展开可能会被代数优化所抵消，因为这些优化现在有机会简化函数与其具体参数的组合。例如，对上述示例应用常量折叠会得到如下结果：

```
for(i=0;i<1000;i++) { y = 40*i*i + 40*i
+ 30; }
```

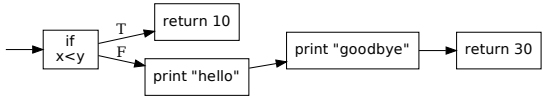
一般来说，函数内联最适用于那些被频繁调用、且相对于调用开销本身只做很少工作的简单叶子函数。然而，自动做出这一判断并做到恰到好处是具有挑战性的，因为收益相对清晰，而以代码体积增大和重复求值为代价的成本却不那么容易量化。因此，许多语言提供了一个关键字（例如 C 和 C++ 中的 inline），允许程序员手动做出这一判断。

12.3.6 Dead Code Detection and Elimination

在已编译的程序中，包含一些在任何可能的输入下都完全不可达、且永远不会被执行的代码并不罕见。这可能只是程序员的一个简单错误，例如不小心在最终语句之前就从函数中返回。或者，也可能是按顺序应用了多种优化，最终导致出现一个永远不会被执行的分支。无论哪种情况，编译器都可以通过将其标记出来提醒程序员，或直接将其移除来提供帮助。

死代码检测通常在完成常量折叠和其他表达式优化之后，在控制流图上进行。例如，考虑下面的代码片段及其控制流图：

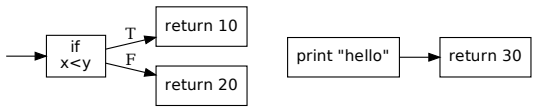
```
if( x<y ) {
   return 10;
} else {
   print "hello";
}
print "goodbye";
return 30;
```



返回语句会立即终止函数的执行，并且（从控制流图的角度来看）是执行路径的结束。在这里，if 语句的真分支会立即返回，而假分支会继续执行到下一个语句。对于 *some* 的 x 和 y 值，所有语句都可以被执行到。

然而，如果我们做一个小的改变，像这样：

```
if( x<y ) {
   return 10;
} else {
   return 20;
}
print "goodbye";
return 30;
```



然后，if 语句的两个分支都以 return 结束，并且无法到达最终的 print 和 return 语句。这（很可能）是程序员的错误，应该标记出来。

一旦控制流图创建完成，确定可达性就很简单：从函数的入口点开始遍历CFG，访问时标记每个节点。一旦遍历完成，任何未标记的节点就被认为是不可达的。编译器可以生成适当的错误消息，或者简单地不为不可达部分生成代码。[3]

可达性分析与其他形式的静态分析结合时变得尤为强大。在上面的例子中，假设变量 x 和 y 分别定义为常量 100 和 200。常量折叠可以将 x<y 简化为 true，从而导致 if 语句的 false 分支永远不会被执行，因此是不可达的。

现在，别被这种思路带偏了。如果你在计算理论课程上认真听过课，这听起来可能可疑地像停机问题：我们能否在不实际执行的情况下，判断一个用图灵完备语言编写的 *arbitrary* 程序是否会运行至完成？答案当然是 *no, not in the gen- eral case*。可达性分析只是确定 *in some limited cases* 即

[3] A slight variation on this technique can be used to evaluate whether every code path through a function results in a suitable `return` statement. This is left as an exercise to the reader.

程序的某个分支无论程序输入如何都是不可能被执行的。它并*not*表明程序中“可达”的分支*will*会在某些输入下被执行，或在任何输入下被执行。

12.4 底层优化

到目前为止讨论的所有优化都可以应用于程序的高层结构，而无需考虑具体的目标机器。低层优化则更多地关注以最佳方式翻译程序结构，从而最大限度地利用底层机器的特性。

12.4.1 Peephole Optimizations

窥孔优化是指任何只非常狭窄地检查一小段代码——也许仅两三条指令——并在该段内进行安全、集中的修改的优化。这类优化作为编译的最后阶段非常容易实现，但对整体效果的提升有限。

冗余加载消除是一种常见的窥孔优化。一系列既修改又使用同一变量的表达式，很容易产生两条相邻的指令：先将寄存器保存到内存中，然后又立即把同一个值加载回来：

Before:	**After:**
`MOVQ %R8, x`	`MOVQ %R8, x`
`MOVQ x, %R8`	

一个轻微的变体是，将对不同寄存器的加载转换为寄存器之间的直接移动，从而节省一次不必要的加载和流水线停顿：

Before:	**After:**
`MOVQ %R8, x`	`MOVQ %R8, x`
`MOVQ x, %R9`	`MOVQ %R8, %R9`

12.4.2 Instruction Selection

在第11章中，我们介绍了一种简单的代码生成方法，其中将 AST（或 DAG）的每个节点替换为至少一条指令（在某些情况下为多条指令）。在功能丰富的 CISC 指令集中，单条指令可以很容易地组合多个操作，例如对指针进行解引用、访问内存以及执行算术运算。

为了利用这些强大的指令，我们可以使用通过树覆盖进行指令选择的技术。[5] 其思想是首先表示每个

将体系结构中的可能指令表示为模板树，其中叶节点可以是寄存器、常量或可替换到指令中的内存地址。

例如，X86 的 ADDQ 指令有一种变体可以将两个寄存器相加。这可以用于在 DAG 中实现一个 IADD 节点，前提是 IADD 的叶子节点存储在寄存器中。加法完成后，ADDQ 会将结果放入与第二个参数相同的寄存器中。所有这些都表示为一个树片段，用于匹配 DAG 的一部分，以及在选择了特定寄存器编号之后要发射的一条指令：

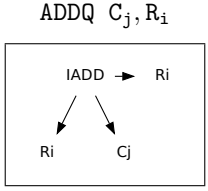


图12.3 给出了更多可以表示为树模板的 X86 指令示例。顶部的简单指令只是将 DAG 中的一个实体替换为另一个：MOV $Cj, Ri 将常量转换为寄存器，而 MOV Mx, Ri 将内存位置转换为寄存器。更复杂的指令具有更多结构：复杂的加载指令 MOV Cj(Rl,8), Ri 可用于表示加法、乘法和解引用的组合。

当然，图 12.3 并不是完整的 X86 指令集。即便要描述一个相当重要的子集，也需要数百个条目，并且每条指令还需要多个条目来刻画该指令的多种变体。（例如，对两个寄存器之间的 ADDQ 需要一个模板，而对寄存器—内存组合则需要另一个模板。）不过，这是一个可行的任务，而且或许比手工编写一个完整的代码生成器更容易完成。

在完整的模板库编写完成后，代码生成器的任务是检查语法树，查找与某个指令模板相匹配的子树。一旦找到，就发出相应的指令（并对寄存器编号等进行适当的替换），并用模板右侧替换树中匹配的部分。

例如，假设我们希望为语句 a[i] = b + 1; 生成 X86 代码。再假设 b 是一个全局变量，而 a 和 i 分别是位于基址指针之上 40 和 32 位置的局部变量。图 12.4 展示了树重写的步骤。在每个 DAG 中，方框表示与图 12.3 中某条规则匹配的子树。

步骤 1：应首先执行左侧的 IADD 以计算表达式的值。查看我们的模板表，没有任何 IADD 可以直接将一个内存位置与一个常量相加。因此，我们改为选择规则（2），该规则生成指令 MOVQ b, %R0，并将左侧的

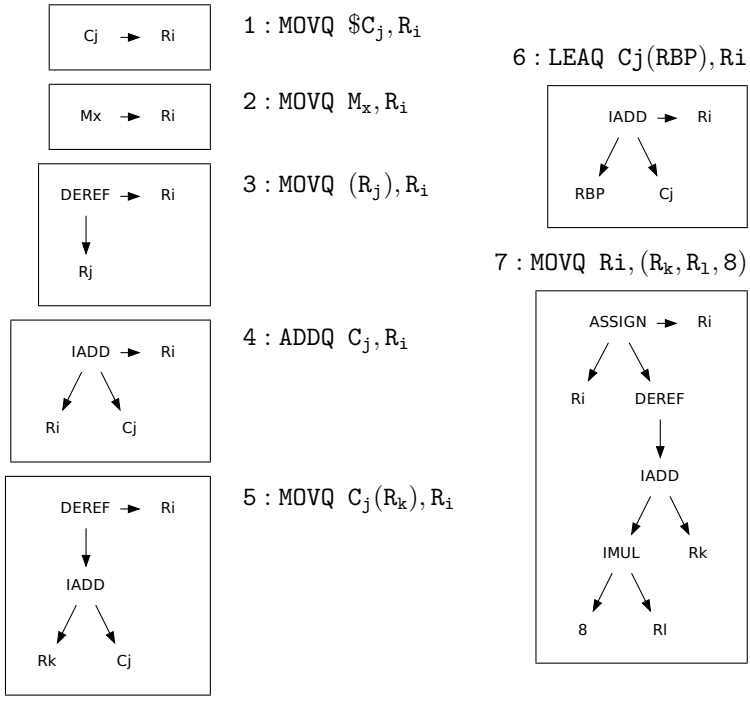


图12.3：X86 指令模板示例

将模板的一侧（一个内存位置）移入右侧（寄存器 %R0）。

步骤 2：现在我们可以看到一个寄存器与常量的 IADD，这与规则（4）的模板相匹配。我们生成指令 ADDQ $1, %R0，并用寄存器 %R0 替换 IADD 子树。

步骤 3：现在让我们看看树的另一侧。我们可以使用规则（5）来匹配整个子树，该子树通过发出指令 MOVQ 32(%RBP), %R1 从 %RBP+32 加载变量 i，并用寄存器 %R1 替换该子树。

步骤 4：以类似的方式，我们可以使用规则（6）通过发出 LEAQ 40(%RBP), %R2 来计算 a 的地址。请注意，这在本质上是一个三地址加法，专门用于寄存器与基址指针。与规则 4 不同，它不会修改源寄存器。

步骤 5：最后，模板规则（7）匹配了剩余的大部分内容。我们可以生成 MOVQ %R0,(%R2,%R1,8)，它将 R1 中的值存储到 a 的计算数组地址中。模板的左侧被右侧替换，只留下寄存器 %R0。随着语法树被完全规约为单个寄存器，代码生成完成，寄存器 %R0 可以被释放。

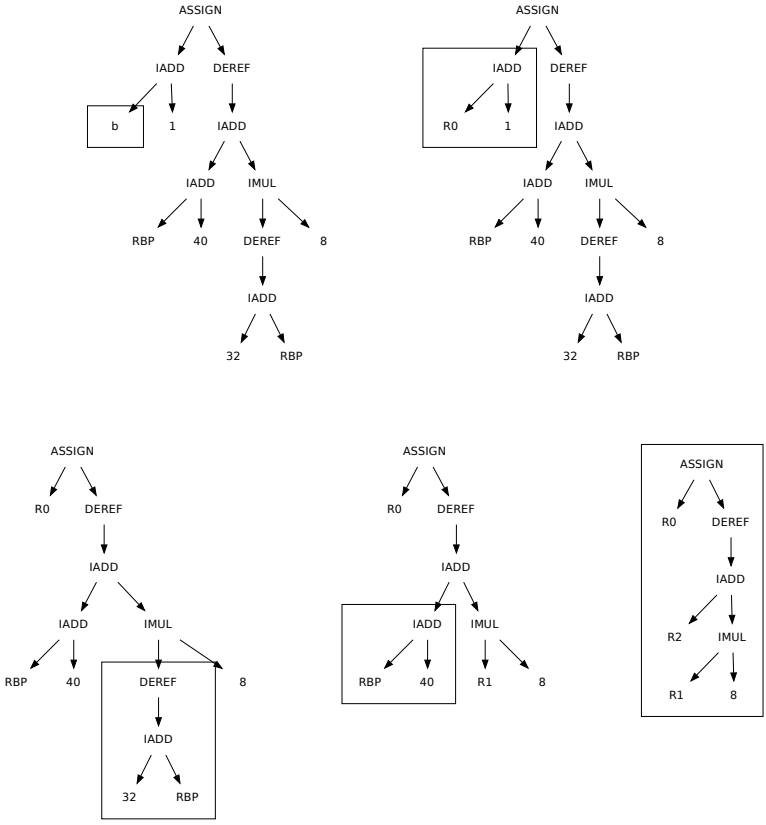


图 12.4：树重写示例

在简单的代码生成方案下，这个16节点的DAG将产生（至少）16条指令作为输出。通过使用树覆盖，我们将输出减少到这五条指令：

MOVQ b,%R0 ADDQ $1,%
R0 MOVQ 32(%RBP),%R1
LEAQ 40(%RBP),%R2 MO
VQ %R0,(%R2,%R1,8)

12.5 寄存器分配

在现代CPU中，芯片内的计算速度远远超过内存延迟：在这个时间内可以完成数千次算术运算。

执行单次加载或存储所需的时间。因此，任何能够消除内存中的加载或存储的优化都可能对程序的性能产生显著影响。完全消除不必要的代码和变量是实现这一目标的第一步。

下一步是将特定的局部变量分配到寄存器中，这样它们就无需从内存加载或写回内存。当然，寄存器的数量是有限的，并非每个变量都能占用一个。寄存器分配的过程就是识别那些最适合放在寄存器而不是内存中的变量。

将一个变量转换为寄存器的机制是很直接的。在每一种需要从内存位置加载或将值存储到内存位置的情况下，编译器只需用所分配的寄存器替代该值的位置，使其直接作为指令的源或目标使用。更复杂的问题在于：是否有必要将某个变量进行寄存器化，哪些变量最适合进行寄存器化，以及哪些变量可以同时共存于寄存器中。让我们依次来看这些问题。

12.5.1 Safety of Register Allocation

如果被消除的内存访问在考虑中的代码之外有一些重要的副作用或可见性，那么将变量注册化是不安全的。应该避免注册化的变量示例包括：

- 多个函数或模块之间共享的全局变量。
- 用作并发线程之间通信的变量。
- 由中断处理程序异步访问的变量。
- 用作内存映射 I/O 区域的变量。

请注意，其中一些情况比其他情况更难检测！全局共享变量已经为编译器所知，但另外三种情况（通常）并未在语言本身中体现出来。在 C 语言中，可以使用 volatile 关键字标记一个变量，以表明它可能会通过编译器未知的某种方式被修改，因此不应进行过于激进的优化。操作系统或并行程序中的低层代码通常在编译时会关闭这类优化，以避免这些问题。

12.5.2 Priority of Register Allocation

对于一个小函数，可能可以将*all*变量寄存器化，从而完全不使用内存。但即便是中等复杂的函数（或可用寄存器很少的 CPU），编译器也很可能必须选择有限数量的变量进行寄存器化。

在自动寄存器分配开发之前，程序员负责手动识别这些变量。例如，在早期的C编译器中，可以在变量声明中添加register关键字，强制将其存储在寄存器中。这通常是为最内层循环的索引变量所做的。当然，程序员可能不会选择最佳的变量，或者可能选择了过多的寄存器变量，从而留下的临时变量空间过少。如今，C编译器基本上忽略了register关键字，因为它们能够做出更为明智的决策。

我们应该使用什么策略来自动选择变量进行寄存器化？自然地，选择那些在程序运行时经历最多加载和存储操作的变量。可以通过分析程序的执行过程，统计每个变量的内存访问次数，然后返回选择前n个变量。当然，这对于优化一个程序来说是一个非常缓慢且昂贵的过程，但对于一个对性能要求极高的程序，可能会考虑这样做。

一个更合理的方法是通过静态分析和一些简单的启发式方法对变量进行评分。在一段线性代码序列中，每个变量可以通过它执行的加载和存储次数直接评分：得分最高的变量是最佳候选。然而，出现在循环内的变量访问很可能具有更高的访问次数。我们无法确切知道它有多大，但我们可以假设一个循环（以及每个嵌套循环）会将变量的重要性乘以一个大常数。多重嵌套的循环以相同的方式增加重要性。

12.5.3 Conflicts Between Variables

并非每个变量都需要一个独立的寄存器。如果两个变量的使用不会发生冲突，它们可以被分配到同一个寄存器。为此，我们必须首先计算每个变量的活跃区间，然后构建冲突图。在一种线性 IR 的基本块内，一个变量从其第一次定义起一直处于活跃状态，直到其最后一次使用为止。（如果相同的代码以 SSA 形式表示，那么变量的每个版本都可以独立对待，并拥有各自的活跃区间。）

现在，具有重叠范围的每个变量不能共享相同的寄存器，因为它们必须独立存在。相反，两个没有重叠活跃范围的变量可以分配相同的寄存器。我们可以构建一个冲突图，其中图中的每个节点代表一个变量，然后在活跃范围重叠的节点之间添加边。图 12.5 给出了一个冲突图的示例。

寄存器分配现在成为图着色问题的一个实例。[7] 目标是为图中的每个节点分配一种不同的颜色（寄存器），使得任何两个相邻节点都不具有相同的颜色。平面图（如二维政治地图）总是可以用

```
   Code            Live Variables
1. x = 10;         x
2. y = 10;         x, y
3. a = x * y;      x, y, a
4. b = a * a;      x, a, b
5. a = b + 30;     x, a, b
6. x = x + 1;      x,
7. z = x * x;      x, z
8. return z;       z
```

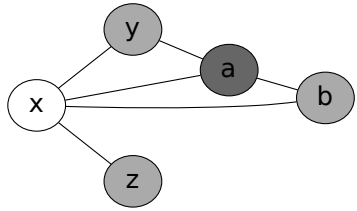


图12.5：活跃范围与寄存器冲突图

四种颜色。[4] 然而，寄存器冲突图不一定是平面的，因此可能需要大量的颜色。寻找所需最小颜色数的普遍问题是NP完全的，但实际上有许多简单的启发式方法在实践中是有效的。

一种常见的方法是按边的数量（冲突数）对图的节点进行排序，然后优先为冲突最多的节点分配寄存器。接着沿着列表向下，为每个节点分配一个未被相邻节点占用的寄存器。如果在某个时刻可用寄存器被耗尽，则将该节点标记为非寄存器化变量，并继续执行，因为仍然可能为冲突较少的节点分配寄存器。

12.5.4 Global Register Allocation

上述过程描述了对单个基本块进行活跃变量分析和寄存器分配。然而，如果每个基本块都被独立分配，那么将很难将基本块组合起来，因为变量会被分配到不同的寄存器，或者根本没有被分配。这样就必须在每个基本块之间引入代码，用于在寄存器之间或在寄存器与内存之间移动变量，这反而可能抵消寄存器分配本身所带来的好处。

为了在整个函数体中执行全局寄存器分配，我们必须以一种保持赋值一致性的方式来进行，无论

[4]This mathematical problem has a particularly colorful (ahem) history. In 1852, Francis Guthrie conjectured that only four colors were necessary to color a planar graph, while attempting to color a map of Europe. He brought this problem to Augustus De Morgan, who popularized it, leading to several other mathematicians who published several (incorrect) proofs in the late 1800s. In 1891, Percy John Heawood proved that no more than *five* colors were sufficient, and that's where things stood for the next 85 years. In 1976, Kenneth Appel and Wolfgang Haken produced a computer-assisted proof of the four-color theorem, but the proof contained over 400 pages of case-by-case analysis which had to be painstakingly verified by hand. This caused consternation in the mathematical community, partly due to the practical difficulty of verifying such a result, but also because this proof was unlike any that had come before. Does it really count as a "proof" if it cannot be easily contemplated and verified by a human? [2]

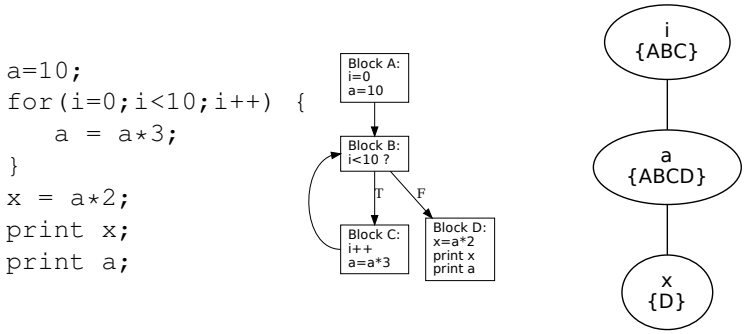


图 12.6：全局寄存器分配示例

To perform register allocation on code (left) consisting of more than a basic block, build the control flow graph (middle) and then determine the blocks in which a given variable is live. Construct a conflict graph (right) such that variables sharing a live block are in conflict, and then color the graph.

控制流的走向。为此，我们首先为该函数构建控制流图。对于图中的每个变量定义，沿着可能的前向路径追踪该变量的使用，并考虑由循环和分支所带来的多条路径。如果存在一条从定义到使用的路径，那么该路径上的所有基本块都属于该变量的活跃基本块集合。最后，可以基于这些活跃块集合构建冲突图：每个节点表示一个变量及其活跃基本块集合；每条边表示两个变量之间的冲突，即它们的活跃集合存在交集。（如前所述，也可以在 SSA 形式下执行相同的分析，以实现更细粒度的寄存器分配。）图 12.6 给出了对一个简单代码片段进行该分析的示例。

12.6 优化陷阱

既然你已经了解了一些常见的优化技术，就应该注意一些所有技术都共有的陷阱。

注意优化的正确性。对所有可能的输入，一段代码在优化前后必须产生相同的结果。我们必须特别关注代码的边界条件，即输入特别大、特别小或触及机器的基本限制时的情况。在将一般的数学或逻辑结论应用到具体代码时，这些问题需要格外谨慎地对待。

例如，人们往往会想要应用常见的代数变换

到算术表达式。我们可以将 a/x + b/x 转换为 (a+b)/x，因为在实数的抽象世界中，这些表达式是相等的。

不幸的是，这两个表达式在有限精度数学的具体世界中是 *not* 相同的。假设 a、b 和 x 是 32 位有符号整数，范围为 $[-2^{31}, +2^{31})$。如果 a 和 b 的值都是 2,000,000,000，那么 a/5 等于 400,000,000，而 a/5+b/5 等于 800,000,000。然而，a+b 会溢出 32 位寄存器，并环绕到一个负值，因此 (a+b)/5 等于 -58993459！一个优化编译器必须非常小心，确保任何代码转换在程序中都能产生相同的结果。

小心不要改变外部副作用。许多真实程序的方面依赖于副作用，而不是计算结果。这在嵌入式系统和硬件驱动程序中最为明显，其中操作系统或应用程序通过内存映射寄存器与外部设备通信。但在传统的用户模式程序中，这种情况也是存在的，这些程序可能通过系统调用执行I/O或其他形式的通信：例如，write()系统调用绝不应该被优化所消除。不幸的是，明确识别每个具有副作用的外部函数是不可行的。一个优化编译器必须保守地假设任何外部函数 *might* 都具有副作用，并且不对其进行修改。

小心优化如何改变调试。一个使用激进优化编译的程序，在调试器下运行时可能会表现出令人惊讶的行为。语句的执行顺序可能与程序中声明的完全不同，给人一种程序流程前后跳动的印象，且没有解释。程序的某些部分可能会被完全跳过，如果它们被判定为不可达，或者已经被简化掉。源代码中提到的变量可能根本不存在于可执行文件中。设在函数调用上的断点可能永远无法触发，如果该函数已经被内联。简而言之，通过调试器可观察到的许多内容并不是真正的程序结果，而是隐藏的内部状态，并且不保证出现在最终的可执行程序中。如果你预期你的程序会有 bug——而它一定会——最好在启用优化之前修复它们。

12.7 优化相互作用

多种优化可能以不可预测的方式相互作用。有时，这些相互作用会以积极的方式级联：常量折叠可以支持可达性分析，从而实现死代码消除。另一方面，优化也可能以消极的方式相互作用：函数内联可能导致表达式更加复杂，从而使寄存器分配效率降低。更糟糕的是，一组优化在某个程序上可能非常有效，但在另一个程序上却可能适得其反。

现代优化编译器可以轻松实现五十种不同的优化

不同复杂度的优化技术。如果只是简单地开关每个优化，那么只有 2^{50} 种组合。但是，如果优化可以按照任意顺序应用，那么就有 fact(50) 种排列！用户如何决定启用哪些优化?

大多数生产编译器定义了几个离散的优化级别。例如，gcc 将 -O0 定义为最快的编译速度，没有任何优化，-O1 启用大约三十个优化，编译时间开销适中，且运行时影响较小（例如死代码消除），-O2 启用另外三十个优化，编译时间开销更大（例如代码提升），-O3 启用激进的优化，这些优化可能有效，也可能无效（例如循环展开）。除此之外，单个优化可以手动启用或禁用。

但通过更精细的控制能做到更好吗？一些研究人员探索了在给定一个运行相对较快的基准程序的情况下，找到最佳优化组合的方法。例如，CHiL L [8] 框架将基准程序的并行执行与高级启发式搜索算法结合，以修剪整个搜索空间。另一种方法是使用遗传算法 [9]，在其中一个代表性配置集被迭代地评估、变异、重新组合，直到出现一个强配置。

12.8 练习

1. 为了对你的机器性能有一个良好的了解，遵循 Jon Bentley [1] 的建议，编写一些简单的基准测试来衡量这些基本操作：

   (a) 整数算术运算。 (b) 浮点算术运算。 (c) 数组元素访问。 (d) 一个简单的函数调用。 (e) 一次内存分配。 (f) 类似 open() 的系统调用。

2. 获取一套标准的基准测试代码（如 SPEC），并评估在你常用的编译器上各种可用优化选项的效果。

3. 在你的项目编译器的 AST 中实现常量折叠优化。4. 在 B-Minor 语言中识别三种强度削弱的机会，并在你的项目编译器中实现它们。5. 在你的项目编译器中实现可达性分析，可以使用 AST 或 CFG。利用该分析确保函数中的所有控制流路径都以合适的 return 语句结束。6. 编写代码以计算并显示你的项目编译器中使用的所有变量的活跃区间。7. 基于前一练习中计算的活跃区间，在基本块上实现线性扫描寄存器分配 [6]。8. 基于前一练习中计算的活跃区间，在基本块上实现图着色寄存器分配 [7]。

12.9 延伸阅读

1. J. Bentley, 《编程珠玑》, Addison-Wesley, 1999年。
A timeless book that offers the programmer a variety of strategies for evaluating the performance of a program and improving its algorithms and data structures.

2. R. Wilson, 《四色足够：地图问题是如何被解决的》, 普林斯顿大学出版社, 2013年。
A history of the long winding road from the four-color conjecture in the nineteenth century to its proof by computer in the twentieth century.

3. A. Aho, M. S. Lam, R. Sethi, J. D. Ullman, 《编译原理：原理、技术与工具》, 第2版, 培生, 2013年。
Affectionately known as the "Dragon Book", this is an advanced and comprehensive book on optimizing compilers, and you are now ready to tackle it.

4. S. L. Graham、P. B. Kessler 和 M. K. McKusick。《Gprof：一种调用图执行性能分析器。》ACM SIGPLAN Notices，第17卷，第6期，1982年。https://doi.org/10.1145/872726.806987

5. A. Aho、M. Ganapathi 和 S. Tjiang, 《使用树匹配和动态规划的代码生成》, ACM《编程语言与系统汇刊》, 第 11 卷, 第 4 期, 1989 年。https://doi.org/10.1145/69558.75700

6. M. Poletto 和 V. Sarkar, 《线性扫描寄存器分配》, ACM《编程语言与系统汇刊》, 第 21 卷, 第 5 期, 1999 年。https://doi.org/10.1145/330249.330250 7. G. J. Chaitin, 《通过图着色的寄存器分配与溢出》, ACM SIGPLAN 通讯, 第 17 卷, 第 6 期, 1982 年 6 月。https://doi.org/10.1145/800230.806984

8. A. Tiwari, C. Chen, J. Chame, M. Hall, J. Hollingsworth, "一种用于编译器优化的可扩展自动调优框架", IEEE 并行与分布式处理国际研讨会, 2009 年。https://doi.org/10.1109/IPDPS.2009.5161054

9. M. Stephenson, S. Amarasinghe, M. Martin, U. O'Reilly, 《元优化：利用机器学习改进编译器启发式方法》, ACM SIGPLAN 程序设计语言设计与实现会议, 2003。 https://doi.org/10.1145/781131.781141

附录 A – 课程项目示例

本附录描述了一个贯穿整个学期的课程项目，这是本书建议配套的内容。你的任课教师可以决定按原样使用，或根据你们课程的时间和地点进行适当修改。该项目的总体目标是构建一个完整的编译器，它以高级语言作为输入，并生成可正常工作的汇编代码作为输出。该项目可以自然地划分为若干阶段，每隔几周完成一个阶段，从而在一个学期内安排 4–6 次作业。

推荐的项目是以 B-Minor 语言作为源语言，并以 x86 或 ARM 汇编作为输出，因为这两者都在本书中有所描述。不过，你也可以使用不同的源语言（如 C、Pascal 或 Rust）或不同的汇编语言或中间表示（如 MIPS、JVM 或 LLVM）来实现类似的目标。

当然，各个阶段是累积的：除非扫描器正确工作，否则解析器无法正确工作，因此在进入下一阶段之前，把每一阶段都做对非常重要。一个关键的开发技术是在每个阶段创建大量（30 个或更多）测试用例，并提供脚本或其他自动化方式来自动运行它们。这将让你对编译器在 B-Minor 的各个不同方面都能正常工作充满信心，并且对某个问题的修复不会破坏其他内容。

A.1 扫描器分配

为 B-Minor 构建一个扫描器，它读取一个源文件，并逐个产生每个记号的清单，标注记号的种类（标识符、整数、字符串等）以及其在源代码中的位置。如果发现无效输入，生成一条消息，从错误中恢复并继续执行。创建一组完整的测试，用于覆盖注释、字符串、转义字符等所有棘手的边界情况。

A.2 解析器作业

在扫描器的基础上，使用 Bison（或其他合适的工具）为 B-Minor 构建一个解析器，该解析器读取一个源文件，并判断是否

该语法是有效的，并表示成功或失败。使用 Bison 的诊断功能来评估给定语法中的歧义，并着手解决诸如悬空 else 问题之类的问题。创建一套完整的测试，用于覆盖并检验所有棘手的边界情况。

A.3 格式化打印器作业

接下来，使用解析器为源程序构建完整的 AST。为了验证 AST 的正确性，将其重新打印为一个等价的源程序，但要将所有空白符排布得整齐美观，以便阅读。这将引发与授课教师关于什么构成“等价”程序的一些有趣讨论。一个必要（但并不充分）的要求是，输出的程序应当能够被同一工具再次解析。这需要在注释、字符串以及整体格式等方面注意一些细节。同样地，创建一组测试用例。

A.4 类型检查器作业

接下来，添加遍历 AST 并执行语义分析的方法，以确定程序的正确性。必须将符号引用解析为其定义，推断表达式的类型，并检查在上下文中值的兼容性。你可能已经习惯于遇到编译器给出的难以理解的错误信息：这是你改进这种情况的机会。同样，创建一组测试用例。

A.5 可选：中间表示

可选地，可以通过添加一个将 AST 转换为中间表示的过程来扩展该项目。这可以是自定义的三元或四元代码、内部 DAG，或者是成熟的 IR，例如 JVM 或 LLVM。使用后者的优势在于，输出可以轻松地输入到现有工具中并实际执行，这应该会给你带来一些成就感。同样，创建一组测试用例。

A.6 代码生成器作业

最令人兴奋的一步是终于生成可运行的汇编代码。最直接的代码生成最容易直接在 AST 本身上完成，或者在可选的 IR 作业中从 AST 派生出的 DAG 上完成，遵循第 11 章中的流程。第一次尝试时，最好不要过多关注代码的效率，而是让每个代码块以保守的方式各自独立存在。最好从一些极其简单的程序开始（例如 return 2+2;），然后逐步一点一点地增加复杂性。在这里，你在构造测试用例方面的练习将真正获得回报，因为

您将能够快速验证多少个测试程序受单个编译器更改的影响。

A.7 可选：扩展语言

在最后一步，鼓励你发挥自己的想法，通过新的数据类型或控制结构扩展B-Minor，创建一个新的后端以支持不同的CPU架构，或实现第12章中描述的一个或多个优化。

附录 B——B 小调语言

B.1 概述

B-Minor语言是一种适合用于本科编译器课程的“小”语言。B-Minor包括表达式、基本控制流、递归函数和严格的类型检查。它与普通C语言的目标代码兼容，因此可以利用标准C库，在其定义的类型范围内。

B-小调与C非常相似，因此应该感觉很熟悉，但也有足够的差异，可以讨论不同语言选择如何影响实现。例如，B-小调的类型语法比C更接近Pascal或SQL的语法。学生们刚开始可能会觉得不习惯，但当构建解析器并讨论与符号无关的类型时，它的价值会变得更加明显。打印语句提供了一个进行简单类型推理并与运行时支持交互的机会。一些不寻常的运算符无法通过单条汇编指令实现，展示了复杂语言内建功能的实现方式。严格的类型系统让学生们有机会思考严格类型代数并生成详细的错误消息。

一个合适的语言定义会相当正式，包括每种标记类型的正则表达式、一个上下文无关文法、类型代数等。然而，如果我们提供所有这些细节，就会剥夺你（学生）亲自处理这些细节的宝贵经验。相反，我们将通过示例来描述语言，留给你仔细阅读，然后提取出你代码所需的正式规范。你一定会发现一些不清晰或不完全指定的细节和边界情况。把它作为一个机会，在课堂上或办公时间提问，并朝着更精确的规范努力。

B.2 词元

在 B-Minor 中，空白可以是以下字符的任意组合：制表符、空格、换行符和回车符。空白在 B-Minor 中的位置并不重要。C 风格和 C++ 风格的注释在 B-Minor 中都是有效的：

```
/* 一个 C 风格的注释 */ a=5; // 一个 C
++ 风格的注释
```

标识符（即变量名和函数名）可以包含字母、数字和下划线。标识符必须以字母或下划线开头。以下是有效标识符的示例：

```
i x mystr fog123 BigLongName55
```

以下字符串是 B-Minor 关键字，不能用作标识符：

```
数组 布尔 字符 否则 假 为 函数 如果 整数 打印 返回 字符串 真
空 当
```

B.3 类型

B-Minor 有四种原子类型：整数、布尔值、字符和字符串。变量的声明方式是：名称后跟一个冒号，然后是类型以及一个可选的初始化器。例如：

```
x: 整数; y: 整数 = 123; b: 布尔值 = 假; c:
字符 = ’ q’ ; s: 字符串 = "hello world\n"
;
```

整数始终是一个有符号的 64 位值。布尔值可以取字面值 true 或 false。字符是一个单一的 8 位 ASCII 字符。字符串是一个双引号常量字符串，且以 null 结尾，不能修改。（注意，与 C 不同，字符串不是字符数组，它是一个完全独立的类型。）

char 和 string 都可以包含以下反斜杠转义码。n 表示换行符（ASCII 值 10），0 表示空字符（ASCII 值 0），而反斜杠后跟任何其他字符则表示紧随其后的那个字符本身。字符串和标识符的长度都可以达到最多 256 个字符。

B小调还允许固定大小的数组。它们可以声明时不指定值，这将导致它们包含所有零：

```
a: 数组 [5] 整数;
```

或者，整个数组可以赋予特定的值：

```
a: 数组 [5] 整数 = {1,2,3,4,5};
```

B.4 表达式

B-Minor 拥有许多在 C 中常见的算术运算符，具有相同的含义和优先级：

`[] f()`	array subscript, function call
`++ --`	postfix increment, decrement
`- !`	unary negation, logical not
`^`	exponentiation
`* / %`	multiplication, division, modulus
`+ -`	addition, subtraction
`< <= > >= == !=`	comparison
`&& ||`	logical and, logical or
`=`	assignment

B-Minor 是 *strictly typed*。这意味着，只有当类型匹配 *exactly* 时，你才能给变量（或函数参数）赋值。你不能进行 C 中常见的许多宽松的类型转换。例如，算术运算符只能作用于整数。比较可以对任何类型的参数执行，但前提是它们的类型必须匹配。逻辑运算只能对布尔值执行。

以下是一些（但并非全部）类型错误的示例 s：

x：整数 = 65； y：字符 = ’A’； if(x>y) ... // 错误：x 和 y 是不同的类型！

f: 整数 = 0; if(f) ... // 错误：f 不是布尔值！

writechar: 函数 void (char c);
a: 整数 = 65;
writechar(a); // 错误：a 不是一个 char !

b: 数组 [2] 布尔 = {true,false};
x: 整数 = 0;
x = b[0]; // 错误：x 不是布尔值！

以下是一些（但并非全部）正确类型赋值的示例：

b：布尔值；x：整数
= 3;
y: 整数 = 5;
b = x<y; // ok：表达式 x<y 是布尔值

f: 整数 = 0;

```
if(f==0) ...     // ok: f==0 is a boolean expression
```

c: 字符 = ’a’;
如果(c==’a’) ... // 正确：c 和 ’a’ 都是 c hars

B.5 声明和语句

在 B-Minor 中，你可以声明带有可选常量初始化器的全局变量、函数原型以及函数定义。在函数内部，你可以声明局部变量（包括数组），并可带有可选的初始化表达式。作用域规则与 C 完全相同。函数定义不能嵌套。

在函数内部，基本语句可以是算术表达式、return 语句、print 语句、if 和 if-else 语句、for 循环，或位于内部 {} 组中的代码。B-Minor 没有 switch 语句、while 循环或 do-while 循环，因为这些都可以很容易地表示为 for 和 if 的特殊情况。

print 语句有点不寻常，因为它是一个语句而不是函数调用。print 接受一个由逗号分隔的表达式列表，并将每个表达式输出到控制台，如下所示：

print "温度是：", temp, " 度\n";

请注意，print 语句后面的列表中的每个元素都是 *any* 类型的表达式。打印机制会自动推断其类型，并输出恰当的表示形式。

B.6 函数

函数的声明方式与变量相同，只不过需要指定函数的类型，其后依次是返回类型、参数和代码：

平方: 函数 整数 (x: 整数) = { 返回 x2; }

函数的返回类型必须是四种原子类型之一，或者使用 void 表示无返回类型。函数参数可以是任意类型。integer、boolean 和 char 类型的参数按值传递，而 string 和 array 类型的参数按引用传递。与 C 语言一样，按引用传递的数组大小是不确定的，因此其长度通常作为一个额外的参数传递：

打印数组：函数 无返回值 (a: 数组 [] 整数, size: 整数) = { i:
整数; for(i=0;i<size;i++) { 打印 a[i], "\n"; } }

一个函数原型声明了函数的存在性和类型，但不包含任何代码。如果用户希望调用由另一个库链接的外部函数，则必须这样做。例如，要调用 C 函数 puts：

puts: 函数 无返回 (s: 字符串);

main: function integer () = { puts("你好,
世界"); }

一个完整的程序必须有一个返回整数的 main 函数。main 的参数可以为空，或者像 C 语言一样使用 argc 和 argv。（argc 和 argv 的声明留给读者作为练习。）

B.7 可选元素

从头到尾实现上述语言的完整实现应该足以让本科生课程忙碌一个学期。然而，如果你需要一些额外的挑战，可以考虑以下想法：

- 添加一种新的原生类型 complex，用于实现复数。为了使其有用，你需要添加一些额外的函数或运算符，用来构造复数值、执行算术运算，并提取实部和虚部。
- 添加一个新的自动类型 var，允许声明一个没有具体类型的变量。编译器应根据对该变量的赋值自动推断类型。仔细考虑如果函数定义中有一个类型为 var 的参数时应该发生什么。
- 通过使数组访问在运行时根据已知的数组大小自动进行检查来提高数组的安全性。这需要将数组的长度作为运行时属性存储在内存中，与数组数据一起存放，并对每一次数组访问进行检查，对照

边界，并在发生违规时采取适当的行动。比较经过检查的数组与未经过检查的数组的性能。（X86 BOUND 指令可能会有所帮助。）

- 添加一种新的可变字符串类型 mutstring，它具有固定大小，但可以原地修改，并且可根据需要与普通字符串相互转换。

- 添加一种替代性的控制流结构（如 switch），它对单一的控制表达式进行求值，然后分支到具有匹配值的各个选项。作为额外挑战，允许 switch 选择值范围，而不仅仅是常量值。

- 实现允许将多个数据项组合在一起的结构类型。在汇编层面，这与实现数组并没有太大不同，因为每个元素只是位于相对于基对象的已知偏移处。然而，解析和类型检查会变得更加复杂，因为必须跟踪与结构类型关联的各个元素。

附录 C – 编码规范

C自1980年代以来一直是实现低级系统（如编译器、操作系统和驱动程序）的首选语言。然而，公正地说，与其他语言相比，C并没有强制执行广泛的良好编程实践。要编写稳健的C代码，你需要具备高度的自律性。[1]

对于许多学生来说，大学里的编译器课程是第一个要求你创建一个较大软件项目的地方，在多个开发周期中不断优化，直到最终项目完成。这是一个很好的机会，让你养成一些能提高生产力的好习惯。

为此，以下是我要求我的学生在编写 C 代码时遵守的编码规范。每条建议都需要在开始时多花一些功夫，但从长远来看会为你节省很多麻烦。

使用版本控制系统。现在有多种很好的开源系统可以用来跟踪你的源代码。今天，Git、Mercurial 和 Subversion 非常流行，我相信明年会出现新的系统。选择一个，学习基本功能，并通过进行小的提交逐步改进你的代码。[2]

从工作到工作。永远不要让你的代码处于破损状态。首先检查一个最简单的、能够编译并且有效的程序草图，即使它只打印“hello world”。然后，以一种小的方式修改程序，确保它能够编译并运行，然后再次提交。[3]

消除死代码。学生们经常养成在尝试更改并测试代码时，将一部分代码注释掉的习惯。虽然这样做

[1] Why not use C++ to address some of these disciplines? Although C++ is a common part of many computer science curricula, I generally discourage the use of C++ by students. Although it has many features that are attractive at first glance, they are not powerful enough to allow you to dispense with the basic C mechanisms. (For example, even if you use the C++ string class, you still need to understand basic character arrays and pointers.) Further, the language is so complex that very few people really understand the complete set of features and how they interact. If you stick with C, what you see is what you get.

[2] Some people like to spend endless hours arguing about the proper way to use arcane features of these tools. Don’t be one of those people: learn the basic operations and spend your mental energy on your code instead.

[3] This advice is often attributed as one of Jim Gray’s “Laws of Data Engineering” in slide presentations, but I haven’t been able to find an authoratative reference.

是一种合理的策略，用于快速测试，不要让这些死代码在程序中堆积，否则你的源代码将很快变得难以理解。删除未使用的代码、数据、注释、文件以及任何对程序不必要的内容，这样你可以清楚地看到现在程序的功能。如果需要，可以信任你的版本控制系统，允许你回到之前的工作版本。

使用工具来处理缩进。不要把时间浪费在争论缩进风格上；找一个能自动为你完成缩进的工具，然后就别再操心了。你的编辑器很可能有自动缩进的模式。如果没有，就使用标准的 Unix 工具 indent。

保持命名一致。在本书中，你会看到每个函数都由一个名词和一个动词组成：expr typecheck、decl codegen 等。它们的用法始终一致：expr 始终用于表达式，codegen 始终用于代码生成。所有处理表达式的函数都位于 expr 模块中。在激烈的开发过程中，走捷径或做缩写可能很诱人，但这最终会反噬你。第一次就把它做好。

只把接口放在头文件中。在 C 语言中，头文件（如 expr.h）用于描述调用函数所需的要素：函数原型以及调用这些函数所必需的类型和常量。如果一个函数只在一个模块内部使用，它 *not* 应该 在头文件中被提及，因为模块之外没有人需要这些信息。

只将实现放在源文件中。在 C 语言中，源文件（如 expr.c）用于提供函数的定义。在源文件中，你应该包含相应的头文件（expr.h），以便编译器能够检查你的函数定义是否与原型匹配。任何对模块而言是私有的函数或变量都应声明为 static。

要懒惰并递归。许多语言的数据结构是分层嵌套的。在设计算法时，要注意这些嵌套的数据结构，并将责任交给其他函数，即使你还没有写出它们。这种技术通常会产生简单、紧凑且可读的代码。例如，要打印一个变量声明，可以将其拆分为先打印名称，然后打印类型，再打印值，中间加上一些标点符号：

```
printf("%s:\n",d->name); type_print
(d->type); printf(" = "); expr_print(
d->value); printf(" ;\n");
```

然后继续编写 type print 和 expr_print，如果你还没有完成的话。

使用 Makefile 自动构建一切。如果你还没有学过，学习如何编写 Makefile。Make 的基本语法非常简单。

以下规则说明 expr.o 依赖于 expr.c 和 expr.h，并且可以通过运行 gcc 命令来构建：

```
expr.o: expr.c expr.h gcc expr.c -c -o expr.o -Wal
l
```

Make 有许多变体，包括通配符、模式替换，以及各种可能让非专家感到困惑的其他内容。只需从编写含义清晰的朴素规则开始。

空指针是你的朋友。在设计数据结构时，使用空指针来表示不存在任何内容。当然，你不能解引用一个空指针，因此在使用它之前必须进行检查。这可能会导致代码到处充斥着空值检查，就像这样：

```
void expr_codegen( struct expr *e, FILE *output ) { if(e->left) expr_cod
egen(e->left,output); if(e->right) expr_codegen(e->right,output); . . . }
```

你可以通过把检查放在函数的开头，并采用递归风格进行编程，来消除其中的许多问题：

```
void expr_codegen( struct expr *e, FILE *output ) { if(!e) return; expr_c
odegen(e->left,output); expr_codegen(e->right,output); . . . }
```

自动化回归测试。编译器必须处理大量细节，在尝试修复旧错误时，你很容易不小心引入新的错误。为此，可以创建一个简单的测试套件，由一组示例程序组成，其中一些是正确的，一些是错误的。编写一个小脚本，对每个示例程序调用你的编译器，并确保它在正确的测试上成功，在错误的测试上失败。将其作为 Makefile 的一部分，这样每次你修改代码时都会运行测试，你就能知道一切是否仍然正常工作。

索引