HZ BOOKS
华章教育

计 算 机 科 学 丛 书

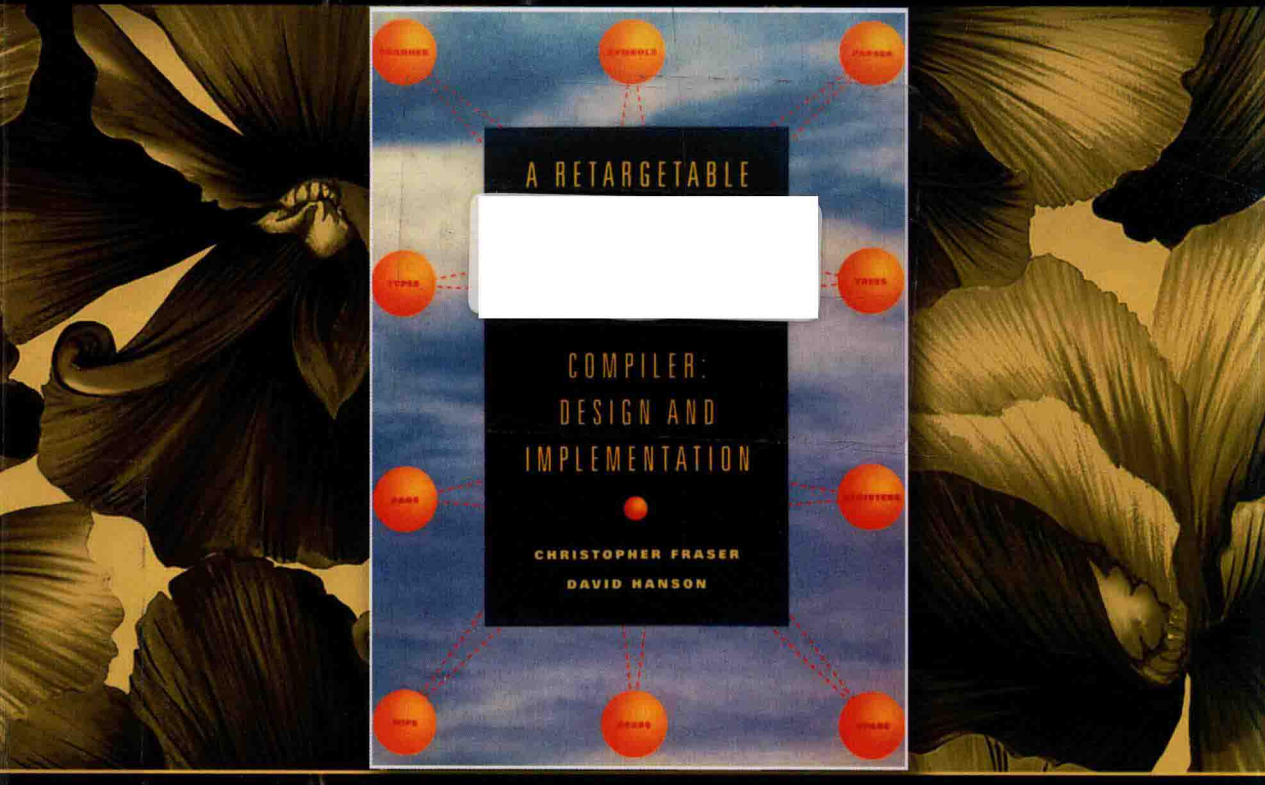
Pearson

可变目标C编译器

设计与实现

[美] 克里斯多夫 W. 弗雷泽（Christopher W. Fraser）
戴维 R. 汉森（David R. Hanson） 著
王挺 黄春 等译

A Retargetable C Compiler
Design and Implementation



机械工业出版社
China Machine Press

计 算 机 科 学 丛 书

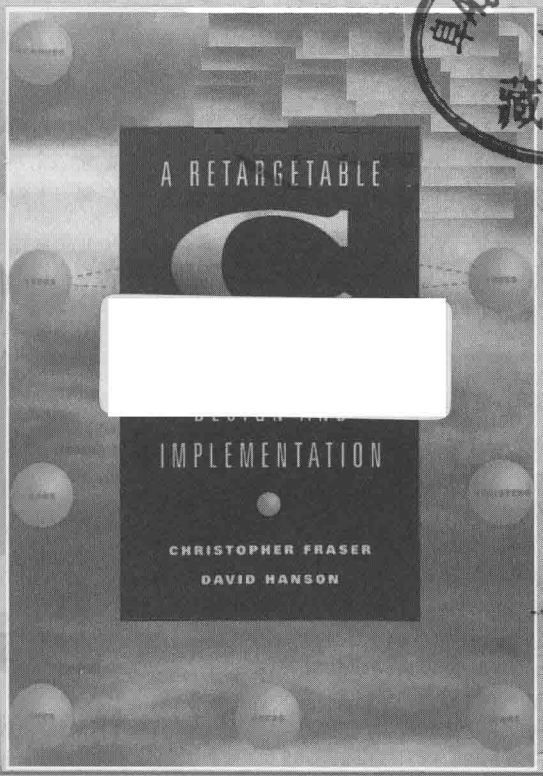
可变目标C编译器

设计与实现

[美] 克里斯多夫 W. 弗雷泽（Christopher W. Fraser）
戴维 R. 汉森（David R. Hanson） 著
王挺 黄春 等译

A Retargetable C Compiler

Design and Implementation



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

可变目标 C 编译器：设计与实现 /（美）克里斯多夫 W. 弗雷泽（Christopher W. Fraser），（美）戴维 R. 汉森（David R. Hanson）著；王挺等译 . —北京：机械工业出版社，2016.11
（计算机科学丛书）
书名原文：A Retargetable C Compiler: Design and Implementation

ISBN 978-7-111-55258-1

I. 可… II. ①克… ②戴… ③王… III. C 语言 - 编译器 - 程序设计 IV. TP312.8

中国版本图书馆 CIP 数据核字（2016）第 260794 号

本书版权登记号：图字：01-2011-2873

本书系统地介绍了可变目标 ANSI C 编译器 lcc 的设计方法和实现技术。lcc 是一个实用的编译器，能够为不同的目标机器（如 MIPS R3000、SPARC、Intel 386 及其后续产品）生成代码。本书结合 lcc 的具体实现，详细讲述了存储管理、符号表、词法分析、语法分析、中间代码生成、优化、目标代码产生等编译程序的各个部分。

本书特色鲜明，实用性强，适合作为高等院校计算机专业编译原理课程的教材或参考书，对从事编译相关工作的技术人员也有很好的参考价值。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码 100037）
责任编辑：迟振春　　责任校对：殷　虹
印　　刷：北京诚信伟业印刷有限公司　　版　　次：2016 年 11 月第 1 版第 1 次印刷
开　　本：185mm × 260mm　1/16　　印　　张：27.25
书　　号：ISBN 978-7-111-55258-1　　定　　价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自 1998 年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S. Tanenbaum，Bjarne Stroustrup，Brian W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街 1 号
邮政编码：100037

华章科技图书出版中心

译 者 序

A Retargetable C Compiler: Design and Implementation

编译器构造原理和技术可以说是计算机科学中理论与实践相结合的最好典范。到目前为止，大多数教材都是介绍编译器构造原理的，很少有详细介绍实用编译器构造的专门书籍。在编译原理课程的教学过程中，如何设计和组织实验一直是一个难题。这主要是因为，任何实用的编译器往往都是庞大的程序，而小的实验编译器又难以反映编译程序构造的许多重要技术。本书可以说是弥补了传统编译教材在实践方面的缺陷，如果希望向学生展示实用编译器是如何设计的，本书应该是最佳选择。

lcc 编译器是一个具有产品级质量的用于研究的 C 编译器，在 UNIX 界广为流行。本书深入到 lcc 编译器的内部，在代码级对该系统的设计和实现进行了详细的介绍。全书共分 19 章，详细讲述了存储管理、符号表、词法分析、语法分析、中间代码生成、优化、目标代码产生等编译程序的各个部分。本书特别针对 3 种目标机器（MIPS、SPARC 和 Intel 386）介绍了代码生成器的设计。通过学习上述内容，读者可以深入了解产品级编译程序设计中的许多关键技术，对于如何设计和实现一个实用的编译器有具体、真切的认识，这是其他教材无法达到的效果。

本书的两位作者都具有深厚的教学和研究背景。Christopher W. Fraser 从 1975 年就开始研究编译技术，尤其对于从紧缩规范自动产生代码生成器这一技术有深入的研究，在该领域发表了多篇论文。他提出了可变目标的窥孔优化方法，该方法被广为流行的 C 编译器——GCC 所采纳。从 1977 年到 1986 年，Fraser 在亚利桑那大学从事计算机科学（包括编译技术）的教学工作。1986 年以后，他在 AT&T 贝尔实验室主持计算技术的研究工作。David R. Hanson 是普林斯顿大学计算机科学教授，具有 20 多年的研究程序语言的经验，主持了与贝尔实验室的合作研究，是 lcc 的开发者之一。

本书是作者的教学、科研和开发思想以及经验的总结，读者可以从字里行间体会到两位作者在编译器的研究和设计方面的造诣。

国防科学技术大学计算机学院从事编译原理课程教学和科研工作的几位教师共同完成了本书的翻译工作。全书由王挺、黄春、刘金红、张晓燕和陈耀东负责翻译，由王挺和黄春通篇整理。由于时间和水平有限，翻译错误在所难免，恳请读者指正。

编译器是程序员使用的关键工具，程序员每天都在使用编译器，并且非常依赖于其正确性和可靠性。编译器必须接受程序语言的所有标准定义，以便源代码可以实现跨平台的可移植性。编译器必须生成高效的目标代码，但更重要的是，编译器必须生成正确的目标代码，只有可靠的编译器才能生成可靠的应用程序。

编译器本身是一个大而复杂的应用程序，值得我们深入分析研究。本书介绍了 ANSI C 语言编译器 lcc 的大部分实现，对编译器的介绍方式与 B. W. Kernighan 和 P. J. Plauger 合著的《Software Tools》（Addison-Wesley，1976）一书对文本处理（例如文本编辑和宏处理）的介绍类似。研究实用的工具软件，是学习软件设计和实现技术的最好方法。本书在代码级详细介绍了一个实用的编译器，该编译器的完整源代码可在 ftp.cs.princeton.edu（128.112.152.13）服务器的 pub/lcc 目录下，通过匿名 ftp 服务得到。

lcc 不是一个研究系统，而是一个实用的编译器产品。从 1988 年开始，lcc 就用于编译实际程序，现在每天都有数百名 C 程序员在使用它。由于本书详细分析了 lcc 编译器的设计与实现，因此用于介绍相关支撑材料的篇幅较少，仅展示了涉及的理论知识，而更为系统的编译技术的介绍可以参见其他教材。本书有意省略一些涉及琐碎和重复实现的语言特征，而将这部分内容作为练习。

显然，本书将使读者对编译器的构造有更多的了解。然而只有少数程序员需要了解编译器的设计与实现，大多数程序员从事的是应用程序或其他系统程序的开发。但是，基于以下 4 个原因，大多数 C 程序员都可以从本书中受益。

第一，一般来说，如果程序员能够理解 C 编译器的工作原理，通常可以成为较好的程序员，特别是较好的 C 程序员。编译器设计者必须全面准确地理解 C 语言的每一个特性，程序员通过学习这些特性的实现，能够更好地掌握语言本身及其在现代计算机上的高效实现。

第二，大多数程序设计教材都是通过一些精简的示例来说明编程技巧的，但大多数程序员都是在从事大型程序的开发，在开发过程中需要不断修改程序，很少有带详细说明的示例可以作为大型程序设计的参考。lcc 不是完美的，但是本书详细说明了该程序的优缺点，可以作为大型程序开发的参考。

第三，编译器是计算机科学中理论与实践相结合的最好典范。lcc 展示了理论与实践的相互作用及其精美的结果，展示了实践需求牵引理论的发展，这些都可以清楚地从代码中找到。通过一个真实的程序来研究这些相互作用，可以帮助程序员理解何时、何地以及如何运用不同的技术。此外，lcc 也阐明了众多的 C 编程技术。

第四，这本书本身是一个文本程序（literate program），如同 D. E. Knuth 所著的《TeX: The Program》（Addison-Wesley，1986）一样，本书包括 lcc 的源代码及说明。为了方便读者理解，本书并未按源程序的顺序对程序代码进行讲解，而是有意进行了调整。

无论是对于在校学生还是专业技术人员，本书都非常适合自学使用。本书为 lcc 提供了说明完整的源代码，希望进行编译技术实践的人员，以及在需要使用或实现基于语言的工具和技术的应用领域（如用户接口）中工作的专业人员，将会对本书感兴趣。lcc 的相关信息可通过以

下地址获得：www.cs.princeton.edu/software/lcc。

本书全面而真实地展示了一个大型软件系统，可作为软件工程课程的分析实例。

对于编译课程来说，本书弥补了传统编译教材的不足。本书介绍了 C 编译器的一种实现方法，而传统教材主要介绍编译过程中遇到的各种问题的解决算法，因此传统教材受篇幅限制只能介绍一些实验性的编译器，代码生成也通常面向较高的级别，以避免与具体的机器相关。

因此，许多教师要求学生完成接近实际的编译器项目，使学生获得实践经验。通常，教师必须从头开始编写编译程序，而学生复制其中的大部分，修改后利用其余的部分。然而，由于编译器只是实验性的，文档往往显得不够充分，这种情形使教学双方都不满意。本书通过对一个实际编译器的大部分程序进行文档说明，并提供源代码，为教师提供了一种新的选择。

本书介绍了完整的代码生成器，代码生成面向 MIPS R3000、SPARC 和 Intel 386 及其后续体系结构等不同的平台。本书利用了最新的研究成果，根据目标机器的紧缩规范（compact specification）生成代码生成器。这些方法使得我们能够针对多种机器展示完整的代码生成器，这是其他书籍无法做到的。通过介绍多个代码生成器，既避免了本书依赖于单一的机器，又有助于学生了解如何设计可变目标的软件。

教师布置的作业可以是增加编译器接受的语言特征、优化、改变目标机器等。本书如果与传统教材配合使用，也可以要求学生使用不同的算法代替现有的模块作为实践作业。如果以实现一个实验编译器作为实践作业，则可能在低级基础结构和重复的语言特征上花费大量的时间。采取上述方法，就能够更接近实际的编译器工程实践。本书的许多练习都涉及编译器工程问题。

除传统的编译目的外，lcc 也有其他用途。例如，它可以用于构建一个 C 程序浏览器，或者根据声明来生成远程过程调用的桩函数（stub），也能用于语言扩充、新的计算机体系结构和代码生成技术的实验。

本书假设读者熟悉某种计算机的 C 和汇编语言，了解编译器的概念，理解编译器的工作原理，同时要求读者的数据结构和算法知识达到一般本科水平，例如，R. Sedgewick 所著的《Algorithms in C》(Addison-Wesley，1990）一书中的内容对于理解 lcc 就足够了。

致谢

感谢众多 lcc 用户，他们来自于 AT&T 贝尔实验室、普林斯顿大学和其他地方，他们忍受了许多程序中的错误，并提供了有价值的反馈。感谢 Hans Boehm、Mary Fernandez、Michael Golan、Paul Haahr、Brian Kernighan、Doug McIlroy、Rob Pike、Dennis Ritchie 和 Ravi Sethi。Ronald Guilmette、David Kristol、David Prosser 和 Dennis Ritchie 在 ANSI 标准的许多细节及其解释方面提供了非常有价值的信息。David Gay 帮助我们改造了 PFORT 数值计算软件库，以作为 lcc 代码生成的测试用例，非常有价值。

Jack Davidson、Todd Proebsting、Norman Ramsey、William Waite 和 David Wall 仔细阅读了我们的代码和文字，大大提高了二者的质量。还要感谢 Steve Beck，他安装并改进了书中用到的字体；感谢 Maylee Noah，他使用 Adobe Illustrator 制作完成了本书的图片。

Christopher W. Fraser
David R. Hanson

引　论

编译器可以将源代码翻译成目标机器上的汇编代码或目标代码。可变目标编译器能够针对不同的目标机器生成代码。编译器中与机器有关的部分被独立成模块，针对不同的目标机器，这些模块可以方便地进行替换。

本书描述了一个可变目标的 ANSI C 编译器 lcc，重点介绍其实现。大多数编译器教材注重于编译算法，只附带介绍了实验性的编译器。而本书与其他教材不同，描述了一个完整的 ANSI C 实用编译器，包括针对 3 种目标机器的代码生成器。本书仅介绍了 lcc 用到的编译理论。

1.1　文本程序

本书不仅描述了 lcc 的实现全貌，其本身就是系统的实现。针对文字编程的 noweb 系统根据同一个文本源程序生成了文本和代码。该源程序中包括了交替出现的说明和带标记的代码片段。代码片段按照有利于描述程序的顺序出现，也就是说，本书说明代码的顺序并不与原来 C 语言程序中的一样。程序 noweave 以这种源程序为输入，生成了本书英文版原稿，包括大部分代码和所有文本。程序 notangle 按照书中要求的顺序抽取了所有的代码。

代码片段包括源代码和对其他代码片段的引用。代码片段的定义是以尖括号括起来的标记开始的，例如下列代码：

```
⟨a fragment label 1⟩≡                                    1
  sum = 0;
  for (i = 0; i < 10; i++) ⟨increment sum 1⟩

⟨increment sum 1⟩≡                                       1
  sum += x[i];
```

这段代码的功能是计算数组 x 的所有元素之和，其中 <increment sum 1> 显示了代码片段是如何被引用的。多个代码片段可以有相同的名字，程序 notangle 可以把这些名字相同的定义连接成一段代码。对于这些连续的程序段定义，程序 noweave 使用 + ≡而不是≡来表示：

```
⟨a fragment label 1⟩+≡                                   1
  printf("%d\n", sum);
```

程序段定义类似宏定义，程序 notangle 抽取整个程序的过程是从一个程序段开始的。如果其定义引用了其他程序段，则把引用的程序段扩展进来，如此重复进行。

程序段定义有助于读者通读这些程序。每个程序段的名字的末尾标有一个数字，该数字是这个程序段开始定义的页码。如果没有数字，则表示该程序段未在本书中定义。每个后续的定义都指明了前一个定义，如果有后续定义，还将指明下一个定义。比如 14 指明当前定义的前一个定义在第 14 页，31 指明下一个定义在第 31 页上。这种标志实际上把一个程序段的所有定义连成了一个双向链表，向前指针指向前一个定义，向后指针指向下一个定义。链表中第一个定义的向前指针和最后一个定义的向后指针被省略。这些链表是完全的：如果一个程序段的部分定义在某页中出现，则可以通过这些页码引用找到相关的定义部分。

大多数程序段也指明了该程序段在哪些页中被使用，如上例中 <increment sum> 定义后面的数字 1，表明该程序段在第 1 页使用。下面可以看到，根程序段（定义模块的程序段）以及经常使用的程序段，则没有加这些标志。

程序 notangle 还实现了 C 语言的一个扩展。较长的字符串文本可以分成若干行，每行的末尾用下划线结尾。notangle 可以剔除后续行的前导空格，把各行连成一个字符串。第 90 页中 error 的第一个参数就是这种扩展功能的例子。

1.2 如何使用本书

一般来说，应该从前往后阅读本书，但也有几种变通方法：

- 第 5 章介绍了编译器前端和后端的接口，这一章的内容尽可能做到独立。
- 第 13 ~ 18 章介绍了编译器的后端。只要读者了解了编译器前端和后端的接口，仅需简单了解前面的章节，就可以直接阅读这些内容。事实上，许多读者甚至能够替换编译器前端或后端，而不需要对另一部分内容做更多的了解。
- 第 16 ~ 18 章介绍了与 3 种目标机器 MIPS、SPARC、Intel 386 及其后续结构相关的模块。这 3 章相互独立，读者可以阅读其中的任意几章。如果读者阅读了多章的内容，就会发现在内容上有一些重复，但是由于大多数通用的代码已经分解出来并放在第 13 ~ 15 章中介绍，少许重复还是可以容忍的。

本书的部分内容采取自底向上的方式描述 lcc。例如，存储管理、字符串和符号表等章节介绍了一些最底层或接近最底层的函数，理解这些函数不需要太多的相关知识。

本书的另外一些内容则采取自顶向下的方式进行描述。例如，表达式分析、语句和声明等章节，从最顶层开始介绍，采取自顶向下的方式，先给出一些函数和程序段的使用及其功能简介，然后再介绍这些函数和程序段的细节。

本书还有一些地方采取了自顶向下和自底向上相结合的方式进行介绍。也许采取一致的方式进行介绍会更好，但是通常难以做到这一点。与大多数编译器一样，lcc 包括相互之间递归调用的函数。所以，试图完全先介绍调用程序再介绍被调用程序，或者先介绍被调用程序再介绍调用程序，都是不可能的。

有些程序段在读代码之前解释起来比较容易，而有些则在阅读代码后解释更容易一些。如果理解一个代码段有困难，不妨先看看该代码段前后的文字，可能会事半功倍。

lcc 的大多数代码都在教材中出现了，但有些程序段只是加以使用而并未给出定义。在这些程序段中，有些由于篇幅限制被省略了，有些则是因为它们实现的是语言扩展功能、可选的调试帮助或重复的成分。例如，只要了解了处理 C 语言的 for 语句的代码，处理 do-while 语句的代码就不必介绍得太多。唯一全部省略的是解释 lcc 对 C 语言的初始化机制的处理，这是因为它既冗长乏味，又无助于其他知识的理解，所以就省略了这部分内容。只使用而未给出定义的程序段很容易识别：这些程序段名字的后面没有页码。

另外我们还省略了断言。lcc 包含了几百条断言，大多数是关于参数或数据结构假设的断言。其中一个是 assert(0)，它表示程序不应执行到该状态。例如，如果分支语句需要确保测试表达式的所有值都有相应的真正的处理分支，那么，就可以在默认分支中加入 assert(0)。

1.3 概述

lcc 能够把源程序翻译成汇编程序代码。下面的例子说明了这一转换的各个阶段，介绍了 lcc 的主要组成和数据结构。每一阶段都把程序变换成另一种表示方式，例如：预处理后的源程序、单词、树、无环有向图及这些图的序列。例如最开始的源代码是：

```
int round(f) float f; {
        return f + 0.5;  /* truncates */
}
```

round 没有函数原型，所以参数 f 作为 double 类型的数据传送，round 在入口处将其转换成 float。然后 round 将 f 加上 0.5，再把结果截尾转换成整数并返回。

第一个阶段是由 C 的预处理程序进行宏扩展、引入头文件、选择条件编译代码等工作。虽然起源于 UNIX 系统，但目前 lcc 可以在 DOS 和 UNIX 系统下运行。与许多 UNIX 编译器一样，lcc 使用独立的预处理器，并且预处理器作为一个独立的进程执行，但这部分内容不在本书范围之内。我们经常使用 GNU C 编译器的预处理程序。

典型的预处理程序读取上面的源程序并产生如下代码：

```
# 1 "sample.c"
int round(f) float f; {
        return f + 0.5;
}
```

由于本例中没有使用预处理特性，所以预处理程序没有其他效果，只是去除了注解，并增加了一个 # 命令，把文件名和源代码的行号告诉编译器，以便诊断错误时使用。这段例子代码中的起始行号显然为 1，但是，如果源程序中包含多个 #include 指令，预处理后每个被包括进来的文件都用一对 # 指令括起来，指令中都包含各自的行号。

预处理程序工作完成后，编译器马上开始工作，首先由词法分析器（lexical analyzer）或扫描器（scanner）将输入的源程序分解成单词（token），见图 1-1。图中左边一栏是单词编码（token code），该编码是一个小的整数，右边一栏是附加值，附加值也可以为空。例如，关键字 int 的附加值是 inttype 的值，代表整数类型。单个字符构成的单词的编码就是该字符的 ASCII 码值，EOI 表示输入结束。词法分析器输出单词和单词的定义点位置，并处理 # 指令，编译器的其他部分无须再处理这些指令。lcc 的词法分析器将在第 6 章中介绍。

编译器的下一个阶段是根据 C 语言的语法规则对单词串进行分析（parse），同时也分析程序的语义（semantic）正确性。例如，检查加法运算中操作数的类型是否合法，以便进行隐式的类型转换。在上面例子的加法运算中，f 是 float 类型，0.5 是 double 类型，这是合法的组合，所得结果为 double 类型，由于返回类型是 int，求出的和被隐式转换成整数。

示例源程序经过分析阶段后，形成两棵抽象语法树（abstract syntax tree），见图 1-2。树中的每个节点代表一个基本运算。第一棵树将传入的 double 类型的参数转换成 float 类型。节点（INDIR+D）从调用程序中地址为 &f 的单元（ADDRF+P）取出 double 类型的值，节点（CVD+F）将该值转换成 float 值。节点（ASGN+F）把 float 值存入被调用程序的地址为 &f 的单元（ADDRF+P）。

第二棵树实现了例子中唯一的一条语句，并返回一个整数（RET+I）。节点（INDIR+F）从被调用程序中地址为 &f 的单元（ADDRF+P）取出 float 值，节点（CVF+D）将该值转换成 double 值，节点（ADD+D）把该值加上 double 常量 0.5（CNST+D），并将结果截尾成整数（CVD+I）。

```
INT     inttype
ID      "round"
'('
ID      "f"
')'
FLOAT   floattype
ID      "f"
';'
'{'
RETURN
ID      "f"
'+'
FCON    0.5
';'
'}'
EOI
```

图 1-1 示例对应的单词串

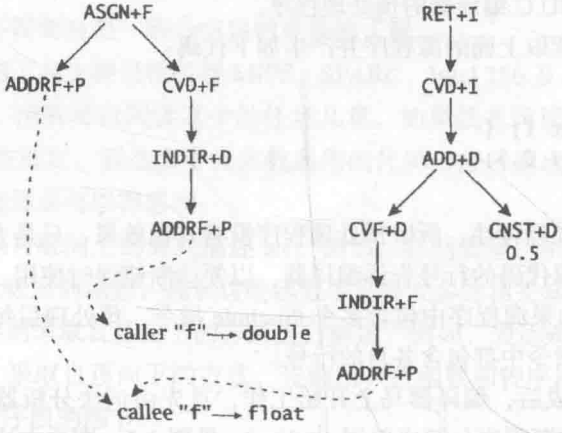


图 1-2 示例对应的抽象语法树

这些树使得隐含在源代码中的一些事实更加清晰。例如，上面的类型转换在源代码中并没有显式给出，但是在 ANSI 标准中是明确规定了的，所以在树中出现了类型转换指令。此外，树中明确给出了所有操作的类型，例如，源代码中的加法操作并没有显式给出类型，但是在树中与其对应的节点具有明显的类型信息。这些语义分析是 lcc 的分析器通过识别输入完成的，我们将在第 7 章到第 11 章进行介绍。

根据图 1-2 中的树，lcc 生成图 1-3 所示的无环有向图（directed acyclic graph，简称 dag），标号为 1 和 2 的 dag 是从图 1-2 中的树转换而来的。操作符标记中不再有 +。把树变换成 dag，使得一些隐含的事实明显化。例如，树中 CNST+D 节点的常量 0.5，在 dag 中成为名字为“2”的静态变量的值，CNST+D 操作符被替换成取地址操作符 ADDRGP 和取值操作符 INDIRD。

图 1-3 中的第三个 dag，定义了名字为“1”的标号，该标号出现在 round 的末尾，返回指令被翻译成跳转到该标号的指令，其他琐碎的细节不再详述。

在第 12 章中可以看到，从树变换成 dag 时，还能够删除相同的表达式（又称公共子表达式）。例如，若重复引用某 dag 代表的运算结果，则只要把该运算结果放入临时变量中，引用时直接使用这个临时变量即可实现删除公共子表达式。本书介绍的代码生成器就采取了这种措施。

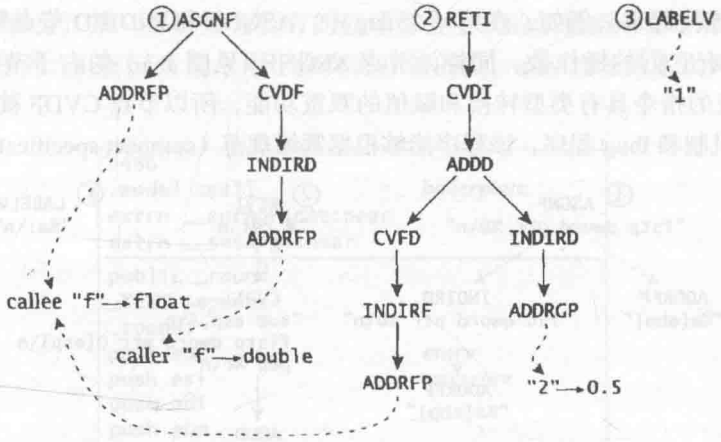


图 1-3 示例对应的 dag

这些 dag 的顺序与图 1-4 所示的代码表的执行顺序相同。列表的第一个入口是 Start，随后的每个入口代表 round 代码的一部分。入口 Defpoint 表示源代码的位置，入口 Blockbeg 和 Blockend 表示 round 代码中复合语句的边界。两个入口 Gen 分别表示执行图 1-3 的 dag 1 和 dag 2。入口 Label 表示执行 dag 3。代码表将在第 10 章和第 12 章中做详细介绍。

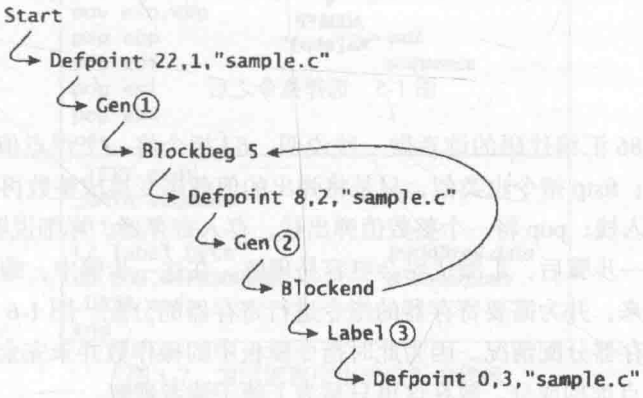


图 1-4 示例对应的代码表

此时，lcc 的与目标机器无关的前端将程序的表示结构传递给后端，由后端把这些结构翻译成目标机器上的汇编代码。我们可以针对特定机器手工编写一个后端程序实现代码的生成，这种代码生成器通常与目标机器紧密相关，如果目标机器发生变化，则需要全部进行更改替换。

本书介绍的代码生成器由表和树文法驱动，在第 13 章到第 18 章中我们将看到树文法能够把 dag 转换成指令序列。这种组织方式使得编译器的后端相对于目标机器具有部分的独立性，使得面对新的目标机器时，我们只需要替换后端的一部分。后端的其他部分也可以移入前端，以期支持针对不同目标机器的代码生成器，但这种方法会使得 lcc 不能方便地使用其他类型的代码生成器，所以我们没有采取这种方法。

代码生成器以对 dag 加注释的方式进行工作。首先为每个节点选择一个汇编代码模板——一条指令或一个操作数。从图 1-5 中可以看到，示例 dag 用 386 及其兼容、后续 X86 平台的汇编代码进行注释。“%n”表示第 n 个子节点的汇编代码，最左子节点的编号为 0；“% 字母”表示该节点所指向符号表的入口。图中，实线连接了指令，而虚线连接部分指令，如将地址模式

和使用它的指令连接起来。例如，在第一个 dag 中，ASGNF 和 INDIRD 节点标有指令，而两个 ADDRGP 节点标有它们的操作数。同样，节点 ASGNF（见图 1-3）的右子节点 CVDF 消失了，因为 ASGNF 对应的指令具有类型转换和赋值的双重功能，所以节点 CVDF 被删除了。第 14 章介绍了指令选择机制和 lbug 程序，该程序能够根据紧缩规范（compact specification）生成代码。

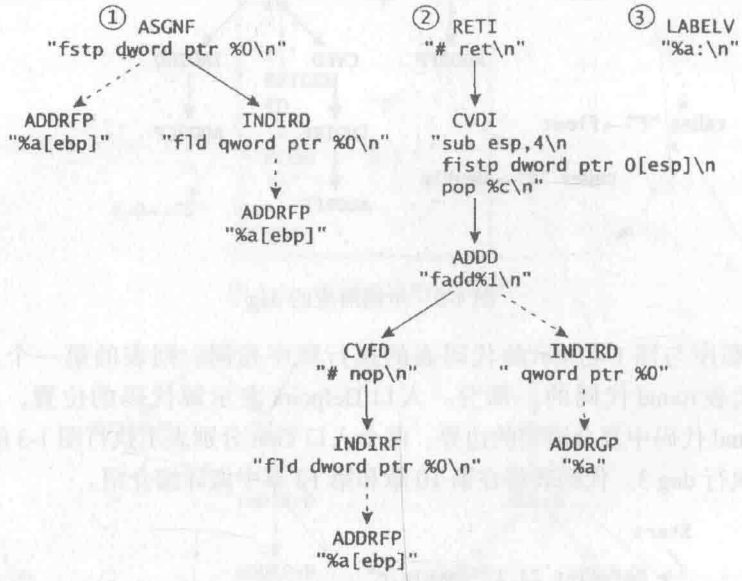


图 1-5 选择指令之后

下面为不了解 X86 汇编代码的读者做一些说明。fld 指令将一个浮点值装入栈；fstp 指令将栈顶的值弹出并存储；fistp 指令也类似，只是将弹出的值截尾变换成整数再存储；fadd 弹出两个值，再将它们的和压入栈；pop 将一个整数值弹出栈，存入寄存器。详细说明参见第 18 章。

在编译器完成下一步骤后，汇编代码会更容易理解，在这一步骤中，编译器将按照指令产生的顺序将节点连接起来，并为需要寄存器的指令进行寄存器的分配。图 1-6 说明了示例程序线性化后的指令序列和寄存器分配情况。因为此时指令模板中的操作数并未完全被替换（这些工作随后进行），图 1-6 有一点虚构成分，放在这里只是为了便于读者理解。

寄存器	汇编模板
eax	fld qword ptr %a[ebp]\n
	fstp dword ptr %a[ebp]\n
	fld dword ptr %a[ebp]\n
	#nop\n
	fadd qword ptr %a\n
	sub esp, 4\nfistp dword ptr 0[esp]\npop %c\n
	# ret\n
	%a:\n

图 1-6 分配寄存器之后

与许多源于 UNIX 系统的编译器一样，lcc 产生汇编代码，还需要和另外的汇编器和连接程序配合使用。与本书的后端配合工作的是：MIPS 和 SPARC 上的汇编器、DOS 下的微软 MASM 6.11 和 Borland 的 Turbo 汇编器 4.0。lcc 会为示例程序生成图 1-7 所示的汇编代码。图中用横线

将代码划分成若干部分，第一部分是为所有的程序都会生成的汇编指示命令样板。第二部分是round的入口指令序列，4条push指令用于保存寄存器的值，mov指令为本次调用round建立帧指针。

```
.486
.model small                    boilerplate
extrn __turboFloat:near
extrn __setargv:near
-------------------------------------------
public _round
_TEXT segment
_round:
push ebx                        entry
push esi                        sequence
push edi
push ebp
mov ebp,esp
-------------------------------------------
fld qword ptr 20[ebp]
fstp dword ptr 20[ebp]
fld dword ptr 20[ebp]           body of
fadd qword ptr L2               round
sub esp,4
fistp dword ptr 0[esp]
pop eax
-------------------------------------------
L1:
mov esp,ebp
pop ebp                         exit
pop edi                         sequence
pop esi
pop ebx
ret
-------------------------------------------
_TEXT ends
_DATA segment
align 4
L2 label byte                   initialized data
dd 00H,03fe00000H               & boilerplate
_DATA ends
end
```

图 1-7 为示例程序生成的汇编代码

第三部分代码是由图1-5中带注释的dag填充了符号表数据后产生的。第四部分是round的出口指令序列，恢复入口指令序列保存寄存器的值，并返回到调用程序。L1是出口指令序列的标号。最后一部分包括初始数据和结束样板（concluding boilerplate）。对于round来说，这些数据包括常量0.5，L2是变量的地址，该变量初始化成$000000003fe00000_{16}$，这是双精度常量0.5的IEEE 64位浮点数表示形式。

1.4 设计

对于lcc来说，并没有一个独立的设计阶段。lcc刚开始只是针对C语言的一个子集的编译器，所以其最初的设计目标非常有限，仅定位于针对一般的编译器实现特别是代码生成的教学的需要。即使后来lcc演化成了适于实用的ANSI C编译器，这一设计目标仍然没有大的改变。

计算的代价越来越低，但是程序员的代价越来越高。如果我们被迫对设计方案做出选择，则通常会选择那种在保证产生令人满意的代码的基础上，能节省开发时间的设计方案。这种选择保证了lcc简单、快速，但可能在优化上比其他编译器略为逊色。lcc面向多目标机器，应尽可能保