第 1 章　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，但也有几种可能的变通方法：

- 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 章 ~ 第 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，仅须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 章 ~ 第 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 章 ~ 第 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，管理存储、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，试图完全地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。惟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、无环路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惟一的一条语句，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

这些树使得隐含在源代码中的一些事实更加清晰化。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。

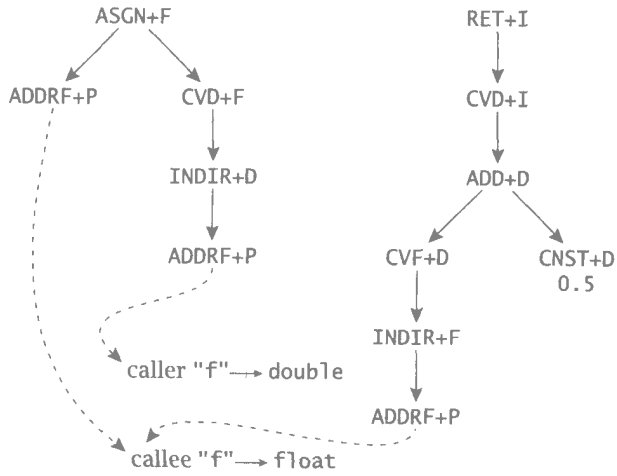

图 1.2　示例对应的抽象语法树

根据图 1.2 中的树，lcc 生成图 1.3 所示的无环路有向图（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 中成为了名字为“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。

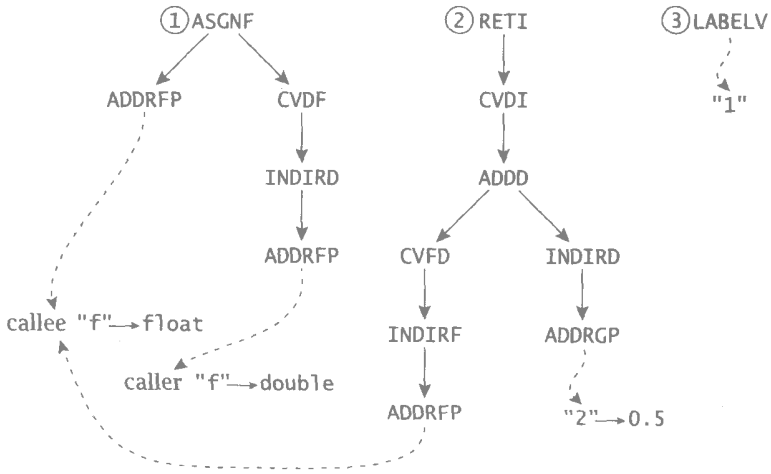

图 1.3　示例对应的 dag

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，人口 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。人口 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。

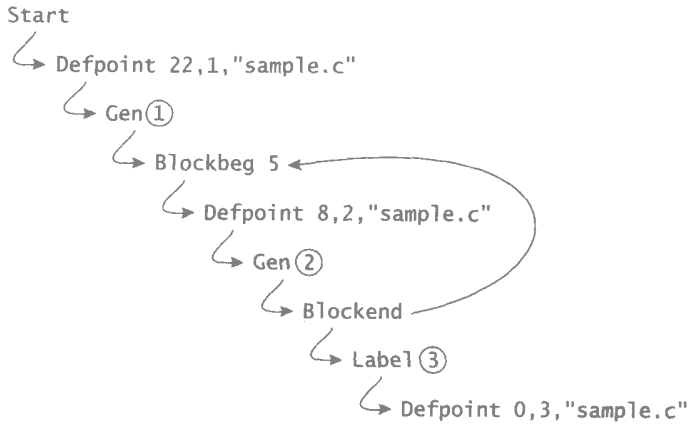

图 1.4　示例对应的代码表

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。

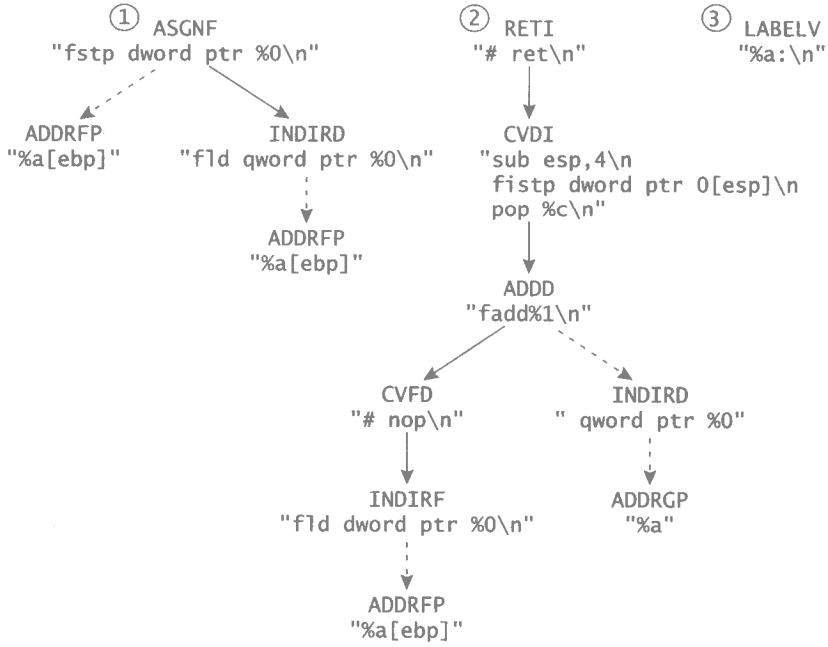

图 1.5　选择指令之后

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
	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
eax	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个部分，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的寄存器的值，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，应尽可能保持系统的简单性。我们将与目标机器无关的代码独立出来，使得与目标机器相关的代码尽可能简单。我们在设计和实现上花费了相当大的精力，使得 lcc更易于移植到新的目标机器上。

由于只有两个开发者，时间有限，lcc必须简单，以节约开发时间并且易于后续完善修改。同时，通过本书，读者可以看到即使编写一个简单的编译器也是很困难的。

lcc比大多数其他 ANSI C编译器更小、更快。尽管有时在编译器设计时会忽视编译的速度，但快速仍是广为欣赏的特点，用户经常把速度作为他们使用 lcc的原因之一。快速编译本身并不是设计的目标，它是追求简单、特别是注意编译器中严重影响速度的组成部分的实现的自然结果。 lcc的词法分析（参见第 6章）和指令选择（参见第 14章）都特别快，对整个系统的速度贡献最大。

lcc能够生成相当高效的目标代码，它以产生好的本地代码为设计目标，而其他优化编译器所具有的全局优化，不在 lcc的设计目标之中。大多数现代的编译器，特别是 CPU供应商开发的编译器，必须采取尽可能的优化措施，以期在性能测试中突出他们的机器。这些编译器很复杂，一般需要数十人组成的小组进行开发。尽管这些高度优化的 C编译器在启用优化选项时，产生的代码比lcc产生的代码更高效，但数百名使用 lcc作为日常主要 C编译器的开发者发现，对于大多数应用来说，lcc产生的代码已经足够快了，同时，由于lcc 的本身速度非常快，也帮助他们节约了许多时间。另外，系统开发者如果需要对 lcc进行修改，则会发现 lcc很容易理解。

编译器并非存在于真空中，它必须与预处理程序、连接程序、装配程序、调试程序、汇编程序和操作系统配合工作，而所有这些软件又依赖于目标机器。要想处理好所有这些软件中与目标相关的部分是不现实的。lcc 的设计把这些不利的影响尽可能降到最低点。比如，依赖于目标的代码生成器产生汇编语言代码，并需要汇编程序帮它们转换目标代码； lcc还依赖于一个独立的预处理程序。这些设计方法并非没有风险，比如，供应商提供的汇编程序存在着一些我们无法控制和回避的错误，给我们带来了很大的困难。

一个更为重要的例子是，必须根据目标机器的约定产生调用代码序列。这对于 lcc是必须实现的，只有这样才能利用现有的库。当然，提供标准的 ANSI C 库可以简化这方面的任务，但即使 lcc提供自己的库，它也仍然需要调用与目标相关的例程，比如提供系统调用的例程。调用第三方的库也有这些限制，这种方式日益重要，而且提供的库通常都是目标代码形式。

产生兼容的代码对于 lcc 中与目标无关的前端以及与目标相关的后端的设计都有明显的影响。在第 5章中可以看到，前端和后端的接口中有很大一部分复杂性是直接源于这些设计上的限制和追求简单性、可变目标性的原则之间的冲突。接口中处理传递和返回结构的机制就是一个例子。

lcc的前端大约有 9000行代码，每种与目标相关的代码生成器有 700行代码，还有约 1000行与目标无关的后端代码被所有的代码生成器共享。

除了个别例外，lcc的前端一般都使用成熟的编译技术。如上一节中介绍的，前端进行词法、句法和语义分析，它也删除公共子表达式（参见第 12章）、进行常量折叠、实现许多简单的与机器无关的转换，以改进本地代码的质量（参见第 9章），许多改进都是简单地对树进行转换，以期生成更好的代码。前端还对循环、 switch语句进行优化，以产生高效代码（参见第 10 章）。

lcc的词法分析器和递归下降分析器都是通过手工编写的。使用编译程序构造工具，如利用分析器生成器，可能是一条实现这些模块的更为现代的途径，但是如果使用这些工具，将使得 lcc 依赖于一些特定的工具。相对于 lcc刚面世时，现在看来这种依赖性已经不算什么问题了，但是，仍

然没有发现有什么必要改变这种工作方式。理论上，使用这些工具会使得日后的改进和纠错变得简单，但是，对于 ANSI C 这种标准语言来说，适应变化的需求并不突出，系统的词法和语法错误也非常少。事实上，lcc 的代码中大约不到 15%是关于分析的，并且这部分程序的错误率也可以忽略不计。尽管分析在理论上的地位非常突出，但是在 lcc 和其他编译器中只是相对较小的模块，语义分析和代码生成才是主要的模块，占了代码的大部分，而大部分错误也出现于此。

后端是 lcc 最令人感兴趣的模块，原因之一就是它体现了我们旨在提高可变目标能力的设计选择。就可变目标来说，适应将来的变化（每种新的目标机器）的能力是非常重要的。由于代码生成中的错误几乎肯定会发生，改变目标的过程对于改正代码的出错必须是相对简单的。整个 lcc 中有许多针对可变目标性的设计上的考虑，其中最为重要的有两个：

第一，后端使用了代码生成器产生器 lburg，该产生器可以根据紧缩规范产生代码生成程序。这些规范描述了 dag 如何映射到指令或指令的一部分（参见第 14 章）。由于 lburg 完成了大多数枯燥的工作，这种方法简化了编写代码生成器、生成优化的本地代码等工作，也有助于避免错误。本书提供了一个 lburg 的规范，针对新目标的规范可以在此基础上设计，因此，改变目标的过程无须从最基本的工作做起。

第二，只要可能，一些明显依赖于目标机器的函数尽可能移到前端实现。例如，前端完全实现了 switch 语句，通过综合运用移位（ shifting ）和掩码(masking ）的组合实现了对位域的访问。这样做，避免了使用一些针对位域访问和 switch 语句的特殊指令，而支持这些指令的目标机器越来越少；简化改变目标的工作更为重要。前端还能完全实现传递和返回结构，为此采用了一些在依赖于目标机器的调用约定中经常使用的技术。这些功能都可以通过接口选项进行控制，因此，对于一些目标机器，后端可以通过设置这些选项，忽略代码生成在这些方面的考虑。

虽然 lcc 的总体设计目标并没有随着其发展而有大的变化，但是实现这些目标的方法却经常改变。大多数改变是把更多的功能移入前端。 switch 语句就是一个例子。在 lcc 的早期版本中，代码生成器的接口包括由后端特殊提供的一些函数，这些函数为 switch 语句产生选择代码。随着新目标的增加，我们发现，这些函数的新版本几乎与原来已有目标中的版本完全相同。这种经验显示，只要对设计做比较简单的改变就可以把所有这些代码移入前端。这样做需要改变所有已有的后端，但是这些改变会移走部分代码，使得将来的新目标的后端变得更简单。

最近所做的最重要的设计变更包括 lcc 的打包方式。先前 lcc 只有一个后端，即针对目标机器 X 的后端和前端结合在一起形成了一个 lcc 实例，该实例在 X 上运行并生成 X 上的代码。大多数 lcc 的后端能够为多个操作系统产生代码。例如，MIPS 后端可以为 MIPS 芯片的计算机产生代码，操作系统包括 DEC 的 Ultrix 和 SGI 的 IRIX，可以构成两个 lcc 的实例。N 个目标机器和 M 个操作系统需要 $N \times M$ 个 lcc 实例才能完全满足，每个实例都是根据目标机器和操作系统对原来的模块做少许不同的修改而得到的。即使是较小的 N 和 M，构建 $N \times M$ 个编译器也是非常枯燥的，而且容易出错。

在为本书开发的 lcc 版本中，我们改变了代码生成器的接口（将在第 5 章中介绍），使得把所有后端结合到一个程序中成为可能。任意一个 lcc 的实例都是一个交叉编译器(cross-compiler)，也就是说，它能为任何目标机器生成代码，而不论该机器上运行的操作系统是什么。通过命令行选项可以选择需要的目标。这种设计方法把所有与目标相关的数据封装在一个结构中，根据选项来选择相应的结构，前端通过这些结构和后端进行通信。这种变化需要修改所有业已存在的后端，但是这种修改只新增了少量的代码。这些努力是非常值得的：现在只需要 M 个 lcc 实例，并且它们都是根据同一组模块构造而成的。同时错误也更容易被发现，通常错误在所有面向不同目标的 lcc 实例中都会出现，可能包括那些专门用于帮助诊断错误的目标。如果节约空间变得非常重要，我们也可以构建一个单目标的 lcc 实例。

从 lcc 的源代码中可以看到数百种设计上的选择，这些选择在实现其他各式各样的软件时也必须加以考虑。lcc 和本书的源代码存放在 noweb 文件中，文本与代码相间，就像本书一样。代码是从 lcc的模块中抽取出来的。表 1.1 说明了本书各章与程序模块的对应关系，并按照其主要功能将各模块进行了分组。有一些对应是一对一的，有一些章对应多个模块，也有较大的模块分在了 3 章中进行介绍。

表 1.1 章和模块的关系

功能	章	头文件	模块
公用定义	1	c.h	
基本部分和数据结构	2		alloc.c string.c
	3		sym.c
	4		types.c
			list.c
代码生成接口	5	ops.h	bind.c
			null.c symbolic.c
I/O 和词法分析	6	token.h	input.c lex.c
			output.c
语法分析和语义分析	7		error.c
	8		expr.c tree.c
	9		enode.c expr.c simp.c
	10		stmt.c
	11		dec1.c main.c
			init.c
中间代码生成	12		dag.c
调试与执行剖面			event.c trace.c
			prof.c profio.c
与目标无关的指令	13	config.h	
选择和寄存器分配	13、14、15		gen.c
代码生成	16		mips.md
	17		sparc.md
	18		x86.md

没有对应章节编号的模块在本书中未做介绍。 list.c 实现了练习 2.15 描述的列表处理函数，output.c 处理输出函数，init.c 分析和处理 C 的初始化机制，event.c 实现了 8.5 节中描述的事件钩子，trace.c 产生跟踪函数调用和返回的代码，prof.c 和 profio.c 产生执行剖面(profiling)代码。

为方便起见，每章都对其模块的实现进行了说明，程序段的形式如下：

```
⟨M 10⟩≡
  #include "c.h"
  ⟨M macros⟩
  ⟨M types⟩
  ⟨M prototypes⟩
  ⟨M data⟩
  ⟨M functions⟩
```

其中 *M* 是模块名，如 alloc.c。<*M macros*>、<*M types*>和<*M prototypes*>定义了本模块使用的宏、类型和函数原型声明。<*M data*>和<*M functions*>包括了外部的和静态的数据及函数的定义(不是声明)。

空的段可以省略。给定了模块名，如 alloc.c，notangle 就能抽取该模块，生成上面形式的程序段及其使用的程序段，所有这些程序段组成了该模块的代码。

上面的程序段中没有页码，这是因为这些程序段使用非常频繁，没必要列出一长串的页码，但仍然定义了程序段向前和向后的指针。

1.5 公共声明

每个模块都说明了输出哪些标识符供其他模块使用，输出标识符的声明通过下列程序段给出：<*M typedefs*>、<*M exported macros*>、<*M exported types*>、<*M exported data*>和<*M exported functions*>，其中 *M* 是模块名。头文件 c.h 把所有模块中的所有这些程序段进行集中声明，只是在程序段定义中不包括 *M*，其他定义与每个模块类似。因此所有的模块只要包括 c.h，就可以使用其他模块说明的标识符。与上节中的程序段相似，出于相同的原因，这些程序段也没有给出页码。

```
⟨c.h 11⟩≡
  ⟨exported macros⟩
  ⟨typedefs⟩
  #include "config.h"
  ⟨interface 59⟩
  ⟨exported types⟩
  ⟨exported data⟩
  ⟨exported functions⟩
```

头文件 config.h 定义了 <*interface*> 中使用的、与后端相关的类型，参见第 5 章。c.h 定义了 lcc 的全局结构和部分全局常量。

lcc 可以用 ANSI 前的编译器（pre-ANSI compiler）进行编译。目前，还有许多这样的 ANSI 前编译器，因此需要小心地维护与它们的兼容性。ANSI 标准增加了原型，这对于查错非常有帮助，我们可以随时利用这些原型。下面的程序段取自 output.c，它说明了 lcc 是如何工作的：

```
⟨output.c exported functions⟩≡                                    12
  extern void outs ARGS((char *));

⟨output.c functions⟩≡                                             12
  void outs(s) char *s; {
     char *p;
     for (p = bp; (*p = *s++) != 0; p++)
        ;
     bp = p;
     if (bp > io[fd]->limit)
        outflush();
  }
```

函数定义忽略了原型，因此老的编译器进行直接编译。函数声明出现在函数定义之前，把完整的 ANSI 参数类型列表作为参数交给宏 ARGS，ANSI 编译器必须预先定义 __STDC__，如果定义了 __STDC__，ARGS 则产生该类型列表，否则，忽略该声明。

```
⟨c.h exported macros⟩≡                                            12
  #ifdef __STDC__
  #define ARGS(list) list
  #else
  #define ARGS(list) ()
  #endif
```

对于 outs 来说，ANSI 前编译器认为其函数声明是：

```
extern void outs ();
```

而 lcc 和其他 ANSI C 编译器认为其函数声明是：

```
extern void outs (char * );
```

由于 outs 的声明出现在定义之前，ANSI 编译器处理定义时必须像定义中已经包括了原型一样，为所有对 outs 的调用进行参数合法性检查。

ANSI 也会改变可变参数(variadic)函数。宏 va_start 把最后声明的参数作为自己的参数，varargs.h 变成 stdarg.h:

```
⟨c.h exported macros⟩+≡                                    11 12
  #ifdef __STDC__
  #include <stdarg.h>
  #define va_init(a,b) va_start(a,b)
  #else
  #include <varargs.h>
  #define va_init(a,b) va_start(a)
  #endif
```

ANSI C 中可变参数函数的定义也有所不同。ANSI C 用定义：

```
void print(char *fmt, ...) { ... }
```

代替 ANSI C 前编译器的定义：

```
void print(fmt, va_alist) char *fmt; va_dcl; { ... }
```

lcc 的宏 VARARGS 根据 __STDC__ 的设置，获得 ANSI 参数列表或 ANSI 前参数列表：

```
⟨c.h exported macros⟩+≡                                    12 13
  #ifdef __STDC__
  #define VARARGS(newlist,oldlist,olddcls) newlist
  #else
  #define VARARGS(newlist,oldlist,olddcls) oldlist olddcls
  #endif
```

output.c 中 print 的定义说明了如何使用 ARGS、va init 和 VARARGS。

```
⟨output.c exported functions⟩+≡                            11 73
  extern void print ARGS((char *, ...));

⟨output.c functions⟩+≡                                     11
  void print VARARGS((char *fmt, ...),
  (fmt, va_alist),char *fmt; va_dcl) {
     va_list ap;

     va_init(ap, fmt);
     vprint(fmt, ap);
     va_end(ap);
  }
```

这个定义略显罗嗦，因为同样的信息用两种不同的形式给出，但是， lcc很少使用 VARARGS，所以也没有必要做改进。

c.h也包含了一些一般用途的宏，这些宏在其他地方使用。

```
⟨c.h exported macros⟩+≡                                    12 13
  #define NULL ((void*)0)
```

NULL代表空指针，是不依赖于机器的表达式，在整数和指针的存储空间大小不相同的情况下， f(NULL)传递一个正确的指针，如果 f没有原型，f(0)传递的字节数就可能不对。lcc产生的代码假设指针作为无符号整数存放。然而，即使某些编译器不支持这一假设，即指针所需的存储空间大于整数的空间，lcc仍然能够被这些编译器编译。lcc在调用时使用 NULL，这样即使在指针占的空间比无符号整数大的环境下也不会出错。所以在这些环境中 lcc仍然可以被编译，也可以作为交叉编译器来使用。

```
⟨c.h exported macros⟩+≡                                    13 73
  #define NELEMS(a) ((int)(sizeof (a)/sizeof ((a)[0])))
  #define roundup(x,n) (((x)+((n)-1))&(~((n)-1)))
```

NELEMS(a)给出了数组 a的元素的数目，roundup(x,n)返回离 x最近的 n的倍数， n是 2的幂。

1.6 语法规范

本书使用文法(grammar）来描述语法(syntax)，例如 C的词法结构、语法和 lcc的代码生成器产生器 lburg使用的规范。

文法定义了一个语言，语言由一组句子组成，句子由字母表中的符号组成，这些符号称为终结符（ terminal symbol ）或单词(token)。文法规则，也称产生式，定义了语言中句子的结构（即语法）。产生式规定了如何从非终结符（nonterminal symbol ）通过反复利用规则对非终结符进行替换而生成句子。

产生式说明了可以替换一个非终结符的文法符号序列，产生式的定义由非终结符、冒号、非终结符的替换串组成。如果一个非终结符有多个替换串，则这些替换串分行显示，或者用竖杠“ | ”分隔。可以出现或不出现的串用方括号（ [ ] ）括起来，可重复 0次或若干次的串用花括号（ {...} ）括起来，圆括号用来分组(group)。非终结符用斜体显示，终结符用固定宽度的打印机（ typewriter ）字体显示。记号“…之一”用于描述由终结符组成的候选序列。如果竖杠、圆括号、方括号和花括号需要作为终结符出现，则必须用单引号括起来以区别于作为产生式定义符号的出现。

例如，产生式：

```
expr:
    term { ( + | - ) term }
term:
    factor { ( * | / ) factor }
factor:
    ID
    '(' expr ')'
```

定义了某语言的简单表达式。非终结符有 *expr*、*term* 和 *factor*，终结符有 ID、+、-、*、/、(、)。第一个产生式说明 *expr* 由 *term* 和后继 0 个或若干个＋ *term*或 – *term*组成，第二个产生式类似于乘除

法运算说明，最后两个产生式说明了 *factor* 由一个 ID 或用括号括起来的 *expr* 组成。最后这两个产生式可以合并成：

```
factor: ID | '(' expr ')'
```

但是分行显示可以增加可读性。

我们还可以在文法中增加简单的函数调用，增加如下产生式：

```
factor: ID '(' expr { , expr } ')'
```

该产生式说明，*factor* 可以由 ID 开头，后面跟着用圆括号括起来的一个或多个表达式组成的列表，各表达式之间用逗号分隔。所有关于 *factor* 的三个产生式可以写成：

```
factor: ID [ '(' expr { , expr } ')' ] | '(' expr ')'
```

该产生式说明，*factor* 可以是 ID，或者以 ID 开头，后面跟着用圆括号括起来的以逗号分隔的表达式列表，还可以是用括号括起来的 *expr*。

这种描述语法规范的记号也叫扩充的巴克斯范式（Backus Naur form），简称 EBNF。7.1 节中说明了如何使用 EBNF 推导出语言中的句子。

1.7 错误

lcc是一个大型复杂的程序。我们常会发现一些错误并加以修正。当我们开始编写本书的时候，一些错误已经暴露出来，随着写作的进行，发现了更多的错误。如果你认为你发现了错误，则可以采取如下措施：

1. 如果你是通过研究本书的代码发现错误的，你可能还没有一个会导致该错误的源程序，那么你首先应创建一个源程序。当然大多数情况下，程序员是在试图编译一个他们自己认为是正确的程序时发现错误的，此时，你可能已经有了一个能导致该错误的源程序。
2. 对源文件进行预处理，保存预处理的结果，抛弃原来的代码。
3. 在保证错误不被隐藏的前提下，尽可能化简你的程序。大多数错误只需要不多于 5 行的代码就可以显示出来。我们需要你化简程序，因为我们只有两个人，而读者却非常多。
4. 确信是在用发布的 lcc 版本处理源文件时出现错误的。如果你修改了 lcc，而该错误仅出现在你的版本中，那么你必须自己追踪错误，即使最后发现错误是由我们造成的，因为我们不能在你的代码上工作。
5. 对你的代码进行注解，说明为什么你认为 lcc 是错误的。如果 lcc 由于断言失败而终止，请告诉我们终止的地方。如果 lcc 崩溃，请尽可能报告最后的调用链。如果 lcc 拒绝一个你认为正确的程序，请告诉我们为什么你认为程序是正确的，并在“The C Programming Language”（Kernighan and Ritchie 1988）一书附录 A 的 ANSI 标准中找到支持你的部分，或者在“C:A Reference Manual”（Harbison and Steele 1991）中找到相应的章节，把页码告诉我们。如果 lcc 生成了不正确的代码，请在注解中附带上错误的汇编代码，并尽可能标出错误的指令。
6. 确信你的错误还没有被改正。最新的 lcc 版本可以在 ftp.cs.princeton.edu 服务器的 pub/lcc 目录下，通过匿名 ftp 服务得到。LOG 文件记录了已经改正的错误及改正的时间。如果你报告的错误已被改正，你就可以找到解决方法。
7. 把你的程序以电子邮件的形式发往 lcc-bugs@cs.princeton.edu。请只发送正确的 C 程序，在程序注解中写上所有的评述，这样我们就能对报告进行半自动处理。

深入阅读

大多数编译器教材注重宽度，介绍了各种编译算法，而没有介绍实用的编译器，即日常用来编译实际程序的编译器。本书做了另一种折中，牺牲了教材的宽度，而深入地介绍了一个实用的编译器。对“宽度”和“深度”侧重不同的教材可以相互补充。例如，当你读到 lcc 的词法分析器的时候，可以考虑阅读 Aho,Sethi and Ullman（1986）、Fischer and LeBlanc（1991）、Waite and Goos（1984）所著的教材，从中学习相关的基础理论和其他方法。其他侧重“深度”的书还有 Holub（1990）及 Waite and Carter（1993）所著的教材。

Fraser and Hanson（1991b）介绍了 lcc 的一个早期版本，包括 lcc 编译速度和生成的代码运行速度的测试结果。这篇论文还介绍了 lcc 设计上的多种选择及其跟踪（tracing）和产生执行剖面（profiling）的功能。

本章向读者介绍了使用本书所需要的 noweb 的知识，如果你需要深入了解 noweb 的基本设计原理和实现技术，可阅读 Ramsey（1994）。noweb 是 WEB（Knuth 1984）的后续产品。Knuth（1992）收集了他的多篇关于文本编程的论文。

ANSI 标准（American National Standards Institute，Inc. 1990）是 C 程序设计语言的语法和语义的权威规范。与其他 C 编译器不同的是，lcc 只编译 ANSI C，它不支持被 ANSI 委员会抛弃的旧的特征。除了标准，Kernighan and Ritchie（1988）可称为 C 的精粹参考。该书面世不久，C 标准就制定完成，因此稍显过时。Harbison and Steele（1991）是在标准制定后出版的，完全按照标准介绍了 C 的语法。Wirth（1977）介绍了 EBNF。

第 2 章　存储管理

复杂的程序大都需要动态分配内存，lcc也不例外。C语言中 malloc 函数分配内存，free 函数释放内存。lcc 也可以使用 malloc 和 free，但是我们优先考虑另一种方法，这种方法更有效、更易于编程实现、更适合在编译器中使用，同时，这种方法也更容易理解。

程序中如果调用了 malloc ,就必须通过 free 释放分配的内存,这种释放工作的代价是明显的。更重要的是，程序员很容易忘记释放这些内存，而更糟的是，可能会释放一些正在被程序引用的内存。

在有些应用程序中，大多数释放工作是在同一时间进行的。以窗口系统为例，滚动条、按钮等窗口元素在窗口创建时建立并分配相应的内存空间，而在窗口撤销时释放占用的内存。编译器也是如此,如 lcc 它在处理到函数中的声明、语句和表达式时进行内存空间的分配，在语句和函数结束时释放这些内存。

大多数 malloc 的实现都采取基于对象大小的内存管理算法。而基于对象生存期（ lifetime ）的算法如果能做到内存及时释放，则更加高效一些。事实上，栈式分配方法可能最为高效，但该方法要求对象的生存期是嵌套的，而这种条件在编译器和许多其他应用程序中并不都能满足。

本章介绍了 lcc基于对象的生存期的存储管理模式。在该模式中，内存分配比 malloc 更高效，释放的代价也非常小。该模式的真正优点是，它使得编程更简单。分配的代价小，使得编程人员可以采取更加面向应用的算法，而无须因为顾及空间效率而采用复杂的算法。同时，在该模式下，内存分配也无须再释放，避免了忘记释放内存的问题。

2.1　内存管理接口

内存是在名为分配区（ arena ）的区域内进行分配的，整个分配区一起释放。生存期相同的对象在同一个分配区中分配。每个分配区都有一个非负的整数作为它的标识符，分配区在分配和释放时通过这个标识符进行标识：

```
⟨alloc.c exported functions⟩≡                                    17
  extern void  *allocate ARGS((unsigned long n, unsigned a));
  extern void deallocate ARGS((unsigned a));
```

许多分配具有下列形式：

```
struct T *p;
p = allocate(sizeof *p, a);
```

其中 T是 C语言的结构，a是一个分配区，p是一个指针，不论 p是什么类型的指针，使用 sizeof *p 都可以正确地分配空间。如果采取另一种需要依赖于指针指向的类型的分配方式，那么当代码发生变化时就容易出错。例如：

```
p = allocate(sizeof (struct T), a);
```

只有当 p确实是一个指向 struct T的指针时，才是正确的；如果 p 改成指向其他结构的指针，那么上面语句分配空间还是一个 struct T ,因此分配就不正确了。第一种方法在效率上并无多大损失,而第二种方法可能造成的错误却是灾难性的。

这种分配方式经常使用，因此我们将其定义成宏：

```
⟨alloc.c exported macros⟩≡
  #define NEW(p,a) ((p) = allocate(sizeof *(p), (a)))
  #define NEW0(p,a) memset(NEW((p),(a)), 0, sizeof *(p))
```

宏NEW返回一个指向尚未初始化的空间的指针，一般来说，大多数使用者会马上初始化这块空间。宏 NEW0 在分配空间后，通过用 C 语言的 memset 将其初始化为 0。memset 将其第一个参数作为函数结果返回。注意，NEW 和 NEW0 对 p 都只计算一次，所以，调用这两个宏时，即使传递的参数表达式有副作用也无妨，如 NEW(a[i++])。

此外，sizeof 的结果是 size_t 类型，该类型必须是无符号整数的，要求能够表示可能声明的最大的对象的大小。实际上，size_t 一般是 unsigned int 和 unsigned long 类型。allocate 的声明使用的是 unsigned long 所以总能表示 sizeof 的结果。

数组是另外一种常见的分配方式，下面的newarray 在给定的分配区中为数组分配足够的空间，假设数组有 m 个元素，每个元素占用 n 个字节：

```
⟨alloc.c exported functions⟩+≡                                    16
  extern void *newarray
     ARGS((unsigned long m, unsigned long n, unsigned a));
```

2.2 分配区的表示

内存管理模块的实现如下：

```
⟨alloc.c 17⟩≡
  #include "c.h"
  ⟨alloc.c types⟩
  #ifdef PURIFY
  ⟨debugging implementation⟩
  #else
  ⟨alloc.c data⟩
  ⟨alloc.c functions⟩
  #endif
```

如果 PURIFY 已定义，则采用 malloc 和 free 方式进行内存分配，这种方式适合查错，详情参见练习 2.1。

如前所述，每个分配区都是由一组很大的内存块构成的链表。每个内存块的块头的数据结构定义如下：

```
⟨alloc.c types⟩≡                                                  19
  struct block {
     struct block *next;
     char *limit;
     char *avail;
  };
```

紧接在这些分配区结构数据之后，直到 limit所指的地址，都是可分配的空间。avail指向块中可分配区域的首地址；地址小于 avail 的空间都是已经分配的区域，从 avail开始，直到 limit 所指的地址为止，都是继续可分配的空间。next 指向链表中的下一块。程序实现中有一个分配区指针，指向链表中第一个还有可分配空间的内存块。下面可以看到，分配时，内存块动态地加入到链表中。