1 Introduction

A *compiler* translates source code to assembler or object code for a target machine. A *retargetable* compiler has multiple targets. Machine-specific compiler parts are isolated in modules that are easily replaced to target different machines.

This book describes `lcc`, a retargetable compiler for ANSI C; it focuses on the implementation. Most compiler texts survey compiling algorithms, which leaves room for only a toy compiler. This book leaves the survey to others. It tours most of a practical compiler for full ANSI C, including code generators for three target machines. It gives only enough compiling theory to explain the methods that it uses.

1.1 Literate Programs

This book not only describes the implementation of `lcc`, it *is* the implementation. The `noweb` system for "literate programming" generates both the book and the code for `lcc` from a single source. This source consists of interleaved prose and labelled code *fragments*. The fragments are written in the order that best suits describing the program, namely the order you see in this book, not the order dictated by the C programming language. The program `noweave` accepts the source and produces the book's typescript, which includes most of the code and all of the text. The program `notangle` extracts all of the code, in the proper order for compilation.

Fragments contain source code and references to other fragments. Fragment definitions are preceded by their labels in angle brackets. For example, the code

```
⟨a fragment label 1⟩≡                                          2▼
  sum = 0;
  for (i = 0; i < 10; i++) ⟨increment sum 1⟩
```

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

sums the elements of `x`. Fragment uses are typeset as illustrated by the use of ⟨*increment* `sum`⟩ in the example above. Several fragments may have the same name; `notangle` concatenates their definitions to produce a single fragment. noweave identifies this concatenation by using $+\equiv$ instead of $\equiv$ in continued definitions:

⟨*a fragment label* 1⟩+≡ $\overset{\blacktriangle}{1}$
```
printf("%d\n", sum);
```

Fragment definitions are like macro definitions; notangle extracts a program by expanding one fragment. If its definition refers to other fragments, they are themselves expanded, and so on.

Fragment definitions include aids to help readers navigate among them. Each fragment name ends with the number of the page on which the fragment's definition begins. If there's no number, the fragment isn't defined in this book, but its code does appear on the companion diskette. Each continued definition also shows the previous definition, and the next continued definition, if there is one. $\overset{\blacktriangle}{14}$ is an example of a previous definition that appears on page 14, and $\underset{\blacktriangledown}{31}$ says the definition is continued on page 31. These annotations form a doubly linked list of definitions; the up arrow points to the previous definition in the list and down arrow points to the next one. The previous link on the first definition in a list is omitted, and the next link on the last definition is omitted. These lists are complete: If some of a fragment's definitions appear on the same page with each other, the links refer to the page on which they appear.

Most fragments also show a list of pages on which the fragment is used, as illustrated by the number 1 to the right of the definition for ⟨*increment* sum⟩, above. These unadorned use lists are omitted for root fragments, which define modules, and for some fragments that occur too frequently, as detailed below.

notangle also implements one extension to C. A long string literal can be split across several lines by ending the lines to be continued with underscores. notangle removes leading white space from continuation lines and concatenates them to form a single string. The first argument to error on page 119 is an example of this extension.

1.2 How to Read This Book

Read this book front-to-back. A few variants are possible.

- Chapter 5 describes the interface between the front end and back ends of the compiler. This chapter has been made as self-contained as possible.
- Chapters 13–18 describe the back ends of the compiler. Once you know the interface, you can read these chapters with few excursions back into their predecessors. Indeed, people have *replaced* the front end and the back ends without reading, much less understanding, the other half.

- Chapters 16–18 describe the modules that capture all information about the three targets — the MIPS, SPARC, and Intel 386 and successor architectures. Each of these chapters is independent, so you may read any subset of them. If you read more than one, you may notice some repetition, but it shouldn't be too irritating because most code common to all three targets has been factored out into Chapters 13–15.

Some parts of the book describe `lcc` from the bottom up. For example, the chapters on managing storage, strings, and symbol tables describe functions that are at or near the ends of call chains. Little context is needed to understand them.

Other parts of the book give a top-down presentation. For example, the chapters on parsing expressions, statements, and declarations begin with the top-level constructs. Top-down material presents some functions or fragments well after the code that uses them, but material near the first use tells enough about the function or fragment to understand what's going on in the interim.

Some parts of the book alternate between top-down and bottom-up presentations. A less variable explanation order would be nice, but it's unattainable. Like most compilers, `lcc` includes mutually recursive functions, so it's impossible to describe all callees before all callers or all callers before all callees.

Some fragments are easier to explain before you see the code. Others are easier to explain afterward. If you need help with a fragment, don't struggle before scanning the text just before *and* after the fragment.

Most of the code for `lcc` appears in the text, but a few fragments are used but not shown. Some of these fragments hold code that is omitted to save space. Others implement language extensions, optional debugging aids, or repetitious constructs. For example, once you've seen the code that handles C's for statement, the code that handles the do-while statement adds little. The only wholesale omission is the explanation of how `lcc` processes C's initializers, which we skipped because it is long, not very interesting, and not needed to understand anything else. Fragments that are used but not defined are easy to identify: no page number follows the fragment name.

Also omitted are assertions. `lcc` includes hundreds of assertions. Most assert something that the code assumes about the value of a parameter or data structure. One is `assert(0)`, which guarantees a diagnostic and thus identifies states that are not supposed to occur. For example, if a switch is supposed to have a bona fide case for all values of the switch expression, then the default case might include `assert(0)`.

The companion diskette is complete. Even the assertions and fragments that are omitted from the text appear on the diskette. Many of them are easily understood once the documented code nearby is understood.

A “mini-index” appears in the middle of the outside margin of many pages. It lists each program identifier that appears on the page and the page number on which the identifier is defined in code or explained in text. These indices not only help locate definitions, but highlight circularities: Identifiers that are used before they are defined appear in the mini-indices with page numbers that follow the page on which they are used. Such circularities can be confusing, but they are inevitable in any description of a large program. A few identifiers are listed with more than one definition; these name important identifiers that are used for more than one purpose or that are defined by both code and prose.

1.3 Overview

lcc transforms a source program to an assembler language program. Following a sample program through the intermediate steps in this transformation illustrates lcc’s major components and data structures. Each step transforms the program into a different representation: preprocessed source, tokens, trees, directed acyclic graphs, and lists of these graphs are examples. The initial source code is:

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

round has no prototype, so the argument is passed as a double and round reduces it to a float upon entry. Then round adds 0.5, truncates the result to an integer, and returns it.

The first phase is the C preprocessor, which expands macros, includes header files, and selects conditionally compiled code. lcc now runs under DOS and UNIX systems, but it originated on UNIX systems. Like many UNIX compilers, lcc uses a separate preprocessor, which runs as a separate process and is not part of this book. We often use the preprocessor that comes with the GNU C compiler.

A typical preprocessor reads the sample code and emits:

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

The sample uses no preprocessor features, so the preprocessor has nothing to do but strip the comment and insert a # directive to tell the compiler the file name and line number of the source code for use when issuing diagnostics. These sample coordinates are straightforward, but a program with numerous #include directives brackets each included

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

FIGURE 1.1 Token stream for the sample.

file with a pair of # directives, and every other one names a line other than 1.

The compiler proper picks up where the preprocessor leaves off. It starts with the *lexical analyzer* or *scanner*, which breaks the input into the *tokens* shown in Figure 1.1. The left column is the *token code*, which is a small integer, and the right column is the associated value, if there is one. For example, the value associated with the keyword `int` is the value of `inttype`, which represents the type integer. The token codes for single-character tokens are the ASCII codes for the characters themselves, and `EOI` marks the end of the input. The lexical analyzer posts the source *coordinate* for each token, and it processes the # directive; the rest of the compiler never sees such directives. `lcc`'s lexical analyzer is described in Chapter 6.

The next compiler phase *parses* the token stream according to the syntax rules of the C language. It also analyzes the program for *semantic* correctness. For example, it checks that the types of the operands in operations, such as addition, are legal, and it checks for implicit conversions. For example, in the sample's addition, `f` is a float and `0.5` is a double, which is a legal combination, and the sum is converted from double to int implicitly because `round`'s return type is int.

The outcome of this phase for the sample are the two decorated *abstract syntax trees* shown in Figure 1.2. Each node represents one basic operation. The first tree reduces the incoming double to a float. It assigns a float (ASGN+F) to the cell with the address `&f` (the left ADDRF+P). It computes the value to assign by converting to float (CVD+F) the double fetched (INDIR+D) from address `&f` (the right ADDRF+P).

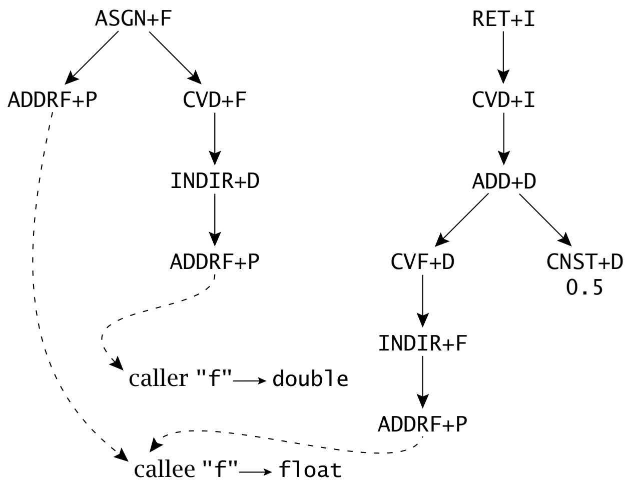

FIGURE 1.2 Abstract syntax trees for the sample.

The second tree implements the sample's lone explicit statement, and returns an int (RET+I). The value is computed by fetching the float (INDIR+F) from the cell with the address &f (ADDRF+P), converting it to double, adding (ADD+D) the double constant 0.5 (CNST+D), and truncating the result to int (CVD+I).

These trees make explicit many facts that are implicit in the source code. For example, the conversions above are all implicit in the source code, but explicit in the ANSI standard and thus in the trees. Also, the trees type all operators explicitly; for example, the addition in the source code has no explicit type, but its counterpart in the tree does. This *semantic analysis* is done as lcc's parser recognizes the input, and is covered in Chapters 7–11.

From the trees shown in Figure 1.2, lcc produces the directed acyclic graphs — *dags* — shown in Figure 1.3. The dags labelled 1 and 2 come from the trees shown in Figure 1.2. The operators are written without the plus signs to identify the structures as dags instead of trees. The transition from trees to dags makes explicit additional implicit facts. For example, the constant 0.5, which appeared in a CNST+D node in the tree, appears as the value of a static variable named 2 in the dag, and the CNST+D operator has been replaced by operators that develop the address of the variable (ADDRGP) and fetch its value (INDIRD).

The third dag, shown in Figure 1.3, defines the label named 1 that appears at the end of round. Return statements are compiled into jumps to this label, and trivial ones are elided.

As detailed in Chapter 12, the transition from trees to dags also eliminates repeated instances of the same expression, which are called *common subexpressions*. Optionally, each multiply referenced dag node can

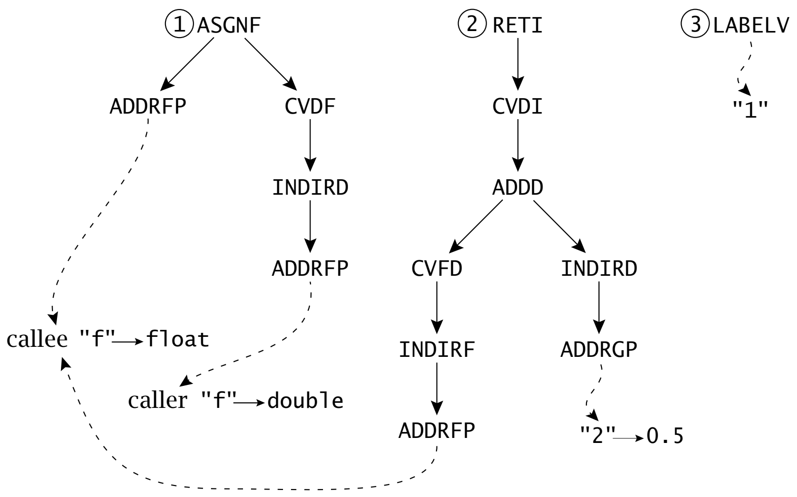

FIGURE 1.3 Dags for the sample.

be eliminated by assigning its value to a temporary and using the temporary in several places. The code generators in this book use this option.

These dags appear in the order that they must execute on the *code list* shown in Figure 1.4. Each entry in this list following the `Start` entry represents one component of the code for `round`. The `Defpoint` entries identify source locations, and the `Blockbeg` and `Blockend` entries identify the boundaries of `round`'s one compound statement. The `Gen` entries carry the dags labelled 1 and 2 in Figure 1.3, and the `Label` entry carries the dag labelled 3. The code list is described in Chapters 10 and 12.

217	Blockbeg
217	Blockend
217	Defpoint
217	Gen
217	Label
217	Start

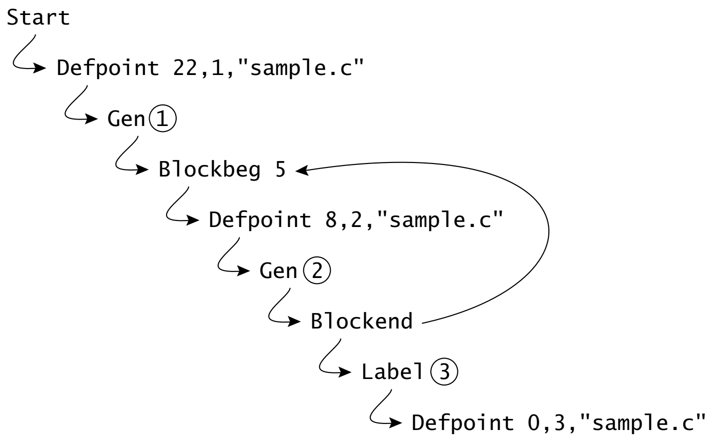

FIGURE 1.4 Code list for the sample.

At this point, the structures that represent the program pass from lcc's machine-independent front end into its back end, which translates these structures into assembler code for the target machine. One can hand-code a back end to emit code for a specific machine; such code generators are often largely machine-specific and must be replaced entirely for a new target.

The code generators in this book are driven by tables and a tree grammar that maps dags to instructions as described in Chapters 13–18. This organization makes the back ends partly independent of the target machine; that is, only part of the back end must be replaced for a new target. The other part could be moved into the front end — which serves all code generators for all target machines — but this step would complicate using lcc with a different kind of code generator, so it has not been taken.

The code generator operates by annotating the dags. It first identifies an assembler-code template — an instruction or operand — that implements each node. Figure 1.5 shows the sample's dags annotated with assembler code for the 386 or compatibles, henceforth termed X86. %*n* denotes the assembler code for child *n* where the leftmost child is numbered 0, and %*letter* denotes one of the symbol-table entries at which the node points. In this figure, the solid lines link instructions, and the dashed lines link parts of instructions, such as addressing modes, to the instructions in which they are used. For example, in the first dag, the ASGNF and INDIRD nodes hold instructions, and the two ADDRGP nodes hold their operands. Also, the CVDF node that was in the right operand of the ASGNF in Figure 1.3 is gone — it's been swallowed by the instruction selection because the instruction associated with the ASGNF does both the conversion and the assignment. Chapter 14 describes the mechanics of instruction selection and lburg, a program that generates selection code from compact specifications.

For those who don't know X86 assembler code, fld loads a floating-point value onto a stack; fstp pops one off and stores it; fistp does likewise but truncates the value and stores the resulting integer instead; fadd pops two values off and pushes their sum; and pop pops an integral value off the stack into a register. Chapter 18 elaborates.

The assembler code is easier to read after the compiler takes its next step, which chains together the nodes that correspond to instructions in the order in which they're to be emitted, and allocates a register for each node that needs one. Figure 1.6 shows the linearized instructions and registers allocated for our sample program. The figure is a bit of a fiction — the operands aren't actually substituted into the instruction templates until later — but the white lie helps here.

Like many compilers that originated on UNIX systems, lcc emits assembler code and is used with a separate assembler and linker. This book's back ends work with the vendors' assemblers on MIPS and SPARC

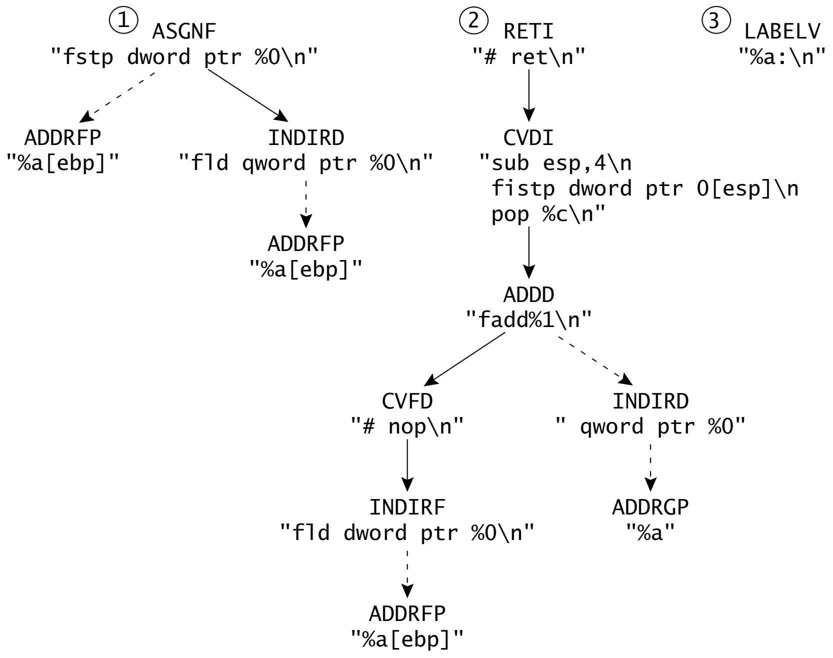

FIGURE 1.5 After selecting instructions.

systems, and with Microsoft's MASM 6.11 and Borland's Turbo Assembler 4.0 under DOS. `lcc` generates the assembler language shown in Figure 1.7 for our sample program. The lines in this code delimit its major parts. The first part is the boilerplate of assembler directives emitted for every program. The second part is the *entry sequence* for `round`. The four `push` instructions save the values of some registers, and the `mov` instruction establishes the *frame pointer* for this invocation of `round`.

The third part is the code emitted from the annotated dags shown in Figure 1.5 with the symbol-table data filled in. The fourth part is `round`'s

Register	*Assembler Template*
	`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`

FIGURE 1.6 After allocating registers.

```
.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
```

FIGURE 1.7 Generated assembler language for the sample.

exit sequence, which restores the registers saved in the entry sequence and returns to the caller. `L1` labels the exit sequence. The last part holds initialized data and concluding boilerplate. For `round`, these data consist only of the constant 0.5; `L2` is the address of a variable initialized to $\texttt{000000003fe00000}_{16}$, which is the IEEE floating-point representation for the 64-bit, double-precision constant 0.5.

1.4 Design

There was no separate design phase for lcc. It began as a compiler for a subset of C, so its initial design goals were modest and focussed on its use in teaching about compiler implementation in general and about code generation in particular. Even as lcc evolved into a compiler for ANSI C that suits production use, the design goals changed little.

Computing costs less and less, but programmers cost more and more. When obliged to choose between two designs, we usually chose the one that appeared to save our time and yours, as long as the quality of the generated code remained satisfactory. This priority made lcc simple, fast, and less ambitious at optimizing than some competing compilers. lcc was to have multiple targets, and it was *overall* simplicity that counted. That is, we wrote extra code in lcc's one machine-independent part to save code in its multiple target-specific parts. Most of the design and implementation effort devoted to lcc has been directed at making it easy to port lcc to new targets.

lcc had to be simple because it was being written by only two programmers with many other demands on their time. Simplicity saved implementation time and saves more when it comes time to change the compiler. Also, we wanted to write this book, and you'll see that it was hard to make even a simple compiler fit.

lcc is smaller and faster than most other ANSI C compilers. Compilation speed is sometimes neglected in compiler design, but it is widely appreciated; users often cite compilation speed as one of the reasons they use lcc. Fast compilation was not a design goal *per se*; it's a consequence of striving for simplicity and of paying attention to those relatively few compiler components where speed really matters. lcc's lexical analysis (Chapter 6) and instruction selection (Chapter 14) are particularly fast, and contribute most to its speed.

lcc generates reasonably efficient object code. It's designed specifically to generate good local code; global optimizations, like those done by optimizing compilers, were not part of lcc's design. Most modern compilers, particularly those written by a CPU vendor to support its machines, must implement ambitious optimizers so that benchmarks put their machines in the best light. Such compilers are complex and typically supported by groups of tens of programmers. Highly optimizing C compilers generate more efficient code than lcc does when their optimization options are enabled, but the hundreds of programmers who use lcc daily as their primary C compiler find that its generated code is fast enough for most applications, and they save another scarce resource — their own time — because lcc runs faster. And lcc is easier to understand when systems programmers find they must change it.

Compilers don't live in a vacuum. They must cooperate with preprocessors, linkers, loaders, debuggers, assemblers, and operating systems, all of which may depend on the target. Handling all of the target-dependent variants of each of these components is impractical. `lcc`'s design minimizes the adverse impact of these components as much as possible. For example, its target-dependent code generators emit assembler language and rely on the target's assembler to produce object code. It also relies on the availability of a separate preprocessor. These design decisions are not without some risk; for example, in vendor-supplied assemblers, we have tripped across several bugs over which we have no control and thus must live with.

A more important example is generating code with calling sequences that are compatible with the target's conventions. It must be possible for `lcc` to do this so it can use existing libraries. A standard ANSI C library is a significant undertaking on its own, but even if `lcc` came with its own library, it would still need to be able to call routines in target-specific libraries, such as those that supply system calls. The same constraint applies to proprietary third-party libraries, which are increasingly important and are usually available only in object-code form.

Generating compatible code has significant design consequences on both `lcc`'s target-independent front end and its target-dependent back ends. A good part of the apparent complexity in the interface between the front and back ends, detailed in Chapter 5, is due directly to the tension between this design constraint and those that strive for simplicity and retargetability. The mechanisms in the interface that deal with passing and returning structures are an example.

`lcc`'s front end is roughly 9,000 lines of code. Its target-dependent code generators are each about 700 lines, and there are about 1,000 lines of target-independent back-end code that are shared between the code generators.

With a few exceptions, `lcc`'s front end uses well established compiler techniques. As surveyed in the previous section, the front end performs lexical, syntactic, and semantic analysis. It also eliminates local common subexpressions (Chapter 12), folds constant expressions, and makes many simple, machine-independent transformations that improve the quality of local code (Chapter 9); many of these improvements are simple tree transformations that lead to better addressing code. It also lays down efficient code for loops and switch statements (Chapter 10).

`lcc`'s lexical analyzer and its recursive-descent parser are both written by hand. Using compiler-construction tools, such as parser generators, is perhaps the more modern approach for implementing these components, but using them would make `lcc` dependent on specific tools. Such dependencies are less a problem now than when `lcc` was first available, but there's little incentive to change working code. Theoretically, using these kinds of tools simplifies both future changes and fixing errors, but accommodating change is less important for a standardized language like ANSI C, and there have been few lexical or syntactic errors. Indeed, probably less than 15 percent of `lcc`'s code concerns parsing, and the error rate in that code is negligible. Despite its theoretical prominence, parsing is a relatively minor component in `lcc` and other compilers; semantic analysis and code generation are the major components and account for most of the code — and have most of the bugs.

One of the reasons that `lcc`'s back ends are its most interesting components is because they show the results of the design choices we made to enhance retargetability. For retargeting, future changes — each new target — *are* important, and the retargeting process must make it reasonably easy to cope with code-generation errors, which are certain to occur. There are many small design decisions made throughout `lcc` that affect retargetability, but two dominate.

First, the back ends use a code-generator generator, `lburg`, that produces code generators from compact specifications. These specifications describe how dags are mapped into instructions or parts thereof (Chapter 14). This approach simplifies writing a code generator, generates optimal local code, and helps avoid errors because `lburg` does most of the tedious work. One of the `lburg` specifications in this book can often be used as a starting point for a new target, so retargeters don't have to start from scratch. To avoid depending on foreign tools, the companion diskette includes `lburg`, which is written in ANSI C.

Second, whenever practical, the front end implements as much of an apparently target-dependent function as possible. For example, the front end implements switch statements completely, and it implements access to bit fields by synthesizing appropriate combinations of shifting and masking. Doing so precludes the use of instructions designed specifically for bit-field access and switch statements on those increasingly few targets that have them; simplifying retargeting was deemed more important. The front end can also completely implement passing or returning structures, and it does so using techniques that are often used in target-dependent calling conventions. These capabilities are under the control of interface options, so, on some targets, the back end can *ignore* these aspects of code generation by setting the appropriate option.

While `lcc`'s overall design goals changed little as the compiler evolved, the ways in which these goals were realized changed often. Most of these changes swept more functionality into the front end. The switch statement is an example. In earlier versions of `lcc`, the code-generation interface included functions that the back end provided specifically to emit the selection code for a switch statement. As new targets were added, it became apparent that the new versions of these functions were nearly identical to the corresponding functions in existing targets. This experience revealed the relatively simple design changes that permitted *all* of this code to be moved into the front end. Doing so required changing all of the existing back ends, but these changes *removed* code, and the design changes simplify the back ends on future targets.

The most significant and most recent design change involves the way lcc is packaged. Previously, lcc was configured with one back end; that is, the back end for target X was combined with the front end to form an instance of lcc that ran on X and generated code for X. Most of lcc's back ends generate code for more than one operating system. Its MIPS back end, for example, generates code for MIPS computers that run DEC's Ultrix or SGI's IRIX, so two instances of lcc were configured. N targets and M operating systems required $N \times M$ instances of lcc in order to test them completely, and each one was configured from a slightly different set of source modules depending on the target and the operating system. For even small values of N and M, building $N \times M$ compilers quickly becomes tedious and prone to error.

In developing the current version of lcc for this book, we changed the code-generation interface, described in Chapter 5, so that it's possible to combine *all* of the back ends into a single program. Any instance of lcc is a *cross-compiler*. That is, it can generate code for any of its targets regardless of the operating system on which it runs. A command-line option selects the desired target. This design packages all target-specific data in a structure, and the option selects the appropriate structure, which the front end then uses to communicate with the back end. This change again required modifying all of the existing back ends, but the changes added little new code. The benefits were worth the effort: Only M instances of lcc are now needed, and they're all built from *one* set of source modules. Bugs tend to be easier to decrypt because they can usually be reproduced in all instances of lcc by specifying the appropriate target, and it's possible to include targets whose sole purpose is to help diagnose bugs. It's still possible to build a one-target instance of lcc, when it's important to save space.

lcc's source code documents the results of the hundreds of subordinate design choices that must be made when implementing software of any significance. The source code for lcc and for this book is in noweb files that alternate text and code just as this book does. The code is extracted to form lcc's modules, which appear on the companion diskette. Table 1.1 shows the correspondence between chapters and modules, and groups the modules according to their primary functions. Some correspondences are one-to-one, some chapters generate several small modules, and one large module is split across three chapters.

The modules without chapter numbers are omitted from this book, but they appear on the companion diskette. list.c implements the list-manipulation functions described in Exercise 2.15, output.c holds the output functions, and init.c parses and processes C initializers. event.c implements the event hooks described in Section 8.5, trace.c emits code to trace calls and returns, and prof.c and profio.c emit profiling code.

Function	*Chapter*	*Header*	*Modules*
common definitions	1	`c.h`	
infrastructure and data structures	2		`alloc.c string.c`
	3		`sym.c`
	4		`types.c`
			`list.c`
code-generation interface	5	`ops.h`	`bind.c`
			`null.c symbolic.c`
I/O and lexical analysis	6	`token.h`	`input.c lex.c`
			`output.c`
parsing and semantic analysis	7		`error.c`
	8		`expr.c tree.c`
	9		`enode.c expr.c simp.c`
	10		`stmt.c`
	11		`decl.c main.c`
			`init.c`
intermediate-code generation	12		`dag.c`
debugging and profiling			`event.c trace.c`
			`prof.c profio.c`
target-independent instruction selection and register management	13	`config.h`	
	13, 14, 15		`gen.c`
code generators	16		`mips.md`
	17		`sparc.md`
	18		`x86.md`

TABLE 1.1 Chapters and modules.

By convention, each chapter specifies the implementation of its module by a fragment of the form

```
⟨M 15⟩≡
  #include "c.h"
  ⟨M macros⟩
  ⟨M types⟩
  ⟨M prototypes⟩
  ⟨M data⟩
  ⟨M functions⟩
```

where *M* is the module name, like `alloc.c`. ⟨*M macros*⟩, ⟨*M types*⟩, and ⟨*M prototypes*⟩ define macros and types and declare function prototypes that are used only within the module. ⟨*M data*⟩ and ⟨*M functions*⟩ include definitions (not declarations) for both external and static data and functions. Empty fragments are elided. A module is extracted by giving `notangle` a module name, such as `alloc.c`, and it extracts the fragment shown above and all the fragments it uses, which yields the code for the module.

Page numbers are *not* included in the fragments above, and they do not appear in the index; they're used in too many places, and the long lists of page numbers would be useless. Pointers to previous and subsequent definitions are given, however.

1.5 Common Declarations

Each module also specifies what identifiers it *exports* for use in other modules. Declarations for exported identifiers are given in fragments named ⟨*M typedefs*⟩, ⟨*M exported macros*⟩, ⟨*M exported types*⟩, ⟨*M exported data*⟩, and ⟨*M exported functions*⟩, where *M* names a module. The header file `c.h` collects these fragments from *all* modules by defining fragments without the *M*s whose definitions list the similarly named fragments from each module. All modules include `c.h`. These fragments are neither page-numbered nor indexed, just like those in the last section, and for the same reason.

```
⟨c.h 16⟩≡
   ⟨exported macros⟩
   ⟨typedefs⟩
   #include "config.h"
   ⟨interface 78⟩
   ⟨exported types⟩
   ⟨exported data⟩
   ⟨exported functions⟩
```

The include file `config.h` defines back-end-specific types that are referenced in ⟨*interface*⟩, as detailed in Chapter 5. `c.h` defines `lcc`'s global structures and some of its global manifest constants.

`lcc` can be compiled with pre-ANSI compilers. There are just enough of these left that it seems prudent to maintain compatibility with them. ANSI added prototypes, which are so helpful in detecting errors that we want to use them whenever we can. The following fragments from `output.c` show how `lcc` does so.

```
⟨output.c exported functions⟩≡                                    18
   extern void outs ARGS((char *));

⟨output.c functions⟩≡                                             18
   void outs(s) char *s; {
      char *p;
      for (p = bp; (*p = *s++) != 0; p++)
         ;
      bp = p;
      if (bp > io[fd]->limit)
         outflush();
   }
```

Function *definitions* omit prototypes, so old compilers compile them directly. Function *declarations* precede the definitions and give the entire list of ANSI parameter types as one argument to the macro ARGS. ANSI compilers must predefine __STDC__, so ARGS yields the types if __STDC__ is defined and discards them otherwise.

⟨*c.h exported macros*⟩≡ 17

```
   #ifdef __STDC__
   #define ARGS(list) list
   #else
   #define ARGS(list) ()
   #endif
```

A pre-ANSI compiler sees the declaration for `outs` as

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

but `lcc` and other ANSI C compilers see

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

103	limit
321	list
98	outflush
16	outs

Since the declaration for `outs` appears before its definition, ANSI compilers must treat the definition as if it included the prototype, too, and thus will check the legality of the parameters in all calls to `outs`.

ANSI also changed variadic functions. The macro `va_start` now expects the last declared parameter as an argument, and `varargs.h` became `stdarg.h`:

⟨*c.h exported macros*⟩+≡ 17 18

```
   #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
```

Definitions of variadic functions also differ. The ANSI C definition

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

replaces the pre-ANSI C definition

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

so lcc's macro VARARGS uses the ANSI parameter list or the pre-ANSI parameter list and separate declarations depending on the setting of __STDC__:

⟨*c.h exported macros*⟩+≡ 17 18

```
#ifdef __STDC__
#define VARARGS(newlist,oldlist,olddcls) newlist
#else
#define VARARGS(newlist,oldlist,olddcls) oldlist olddcls
#endif
```

The definition of print from output.c shows the use of ARGS, va_init, and VARARGS.

⟨*output.c exported functions*⟩+≡ 16 97

```
extern void print ARGS((char *, ...));
```

⟨*output.c functions*⟩+≡ 16

```
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);
}
```

```
   ARGS 17
va_init 17
```

This definition is verbose because it gives the same information in two slightly different formats, but lcc uses VARARGS so seldom that it's not worth fixing.

c.h also includes a few general-purpose macros that fit nowhere else.

⟨*c.h exported macros*⟩+≡ 18 19

```
#define NULL ((void*)0)
```

NULL is a machine-independent expression for a null pointer; in environments where integers and pointers aren't the same size, f(NULL) passes a correct pointer where f(0) can pass more bytes or fewer in the absence of a prototype for f. lcc's generated code assumes that pointers fit in unsigned integers. lcc can, however, be *compiled by* other compilers for which this assumption is false, that is, for which pointers are larger than integers. Using NULL in calls avoids these kinds of errors in environments where pointers are wider than unsigned integers, and thus permits lcc to be compiled and used as a cross-compiler in such environments.

⟨*c.h exported macros*⟩+≡ 18 97

```
  #define NELEMS(a) ((int)(sizeof (a)/sizeof ((a)[0])))
  #define roundup(x,n) (((x)+((n)-1))&(~((n)-1)))
```

`NELEMS(a)` gives the number of elements in array `a`, and `roundup(x,n)` returns `x` rounded up to the next multiple of `n`, which must be a power of two.

1.6 Syntax Specifications

Grammars are used throughout this book to specify syntax. Examples include C's lexical structure and its syntax and the specifications read by `lburg`, `lcc`'s code-generator generator.

A *grammar* defines a language, which is a set of sentences composed of symbols from an alphabet. These symbols are called *terminal* symbols or tokens. Grammar rules, or *productions*, define the structure, or *syntax*, of the sentences in the language. Productions specify the ways in which sentences can be produced from *nonterminal* symbols by repeatedly replacing a nonterminal by one of its rules.

A production specifies a sequence of grammar symbols that can replace a nonterminal, and a production is defined by listing the nonterminal, a colon, and nonterminal's replacement. A list of replacements for a nonterminal is given by displaying the alternatives on separate lines or by separating them by vertical bars (|). Optional phrases are enclosed in brackets ([...]), braces ({...}) enclose phrases that can be repeated zero or more times, and parentheses are used for grouping. Nonterminals appear in *slanted* type and terminals appear in a fixed-width `typewriter` type. The notation "one of ..." is also used to specify a list of alternatives, all of which are terminals. When vertical bars, parentheses, brackets, or braces appear as terminals, they're enclosed in single quotes to avoid confusing their use as terminals with their use in defining productions.

For example, the productions

expr:
  term { (`+` | `-`) *term* }

term:
  factor { (`*` | `/`) *factor* }

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

define a language of simple expressions. The nonterminals are *expr*, *term*, and *factor*, and the terminals are `ID + - * / ( )`. The first production says that an *expr* is a *term* followed by zero or more occurrences of `+` *term* or `-` *term*, and the second production is a similar specification for the multiplicative operators. The last two productions specify that a *factor* is an `ID` or a parenthesized *expr*. These last two productions could also be written more compactly as

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

Giving some alternatives on separate lines often makes grammars easier to read.

Simple function calls could be added to this grammar by adding the production

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

which says that a *factor* can also be an `ID` followed by a parenthesized list of one or more *expr*s separated by commas. All three productions for *factor* could be written as

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

which says that a *factor* is an `ID` optionally followed by a parenthesized list of comma-separated *expr*s, or just a parenthesized *expr*.

This notation for syntax specifications is known as extended Backus-Naur form, or EBNF. Section 7.1 gives the formalities of using EBNF grammars to derive the sentences in a language.

1.7 Errors

`lcc` is a large, complex program. We find and repair errors routinely. It's likely that errors were present when we started writing this book and that the act of writing added more. If you think that you've found an error, here's what to do.

1. If you found the error by inspecting code in this book, you might not have a source file that displays the error, so start by creating one. Most errors, however, are exposed when programmers try to compile a program they think is valid, so you probably have a demonstration program already.

2. Preprocess the source file and capture the preprocessor output. Discard the original code.

3. Prune your source code until it can be pruned no more without sending the error into hiding. We prune most error demonstrations to fewer than five lines. We need you to do this pruning because there are a lot of you and only two of us.

4. Confirm that the source file displays the error with the *distributed* version of `lcc`. If you've changed `lcc` and the error appears only in your version, then you'll have to chase the error yourself, even if it turns out to be our fault, because we can't work on your code.

5. Annotate your code with comments that explain why you think that `lcc` is wrong. If `lcc` dies with an assertion failure, please tell us where it died. If `lcc` crashes, please report the last part of the call chain if you can. If `lcc` is rejecting a program you think is valid, please tell us why you think it's valid, and include supporting page numbers in the ANSI Standard, Appendix A in *The C Programming Language* (Kernighan and Ritchie 1988), or the appropriate section in *C: A Reference Manual* (Harbison and Steele 1991). If `lcc` silently generates incorrect code for some construct, please include the corrupt assembler code in the comments and flag the bad instructions if you can.

6. Confirm that your error hasn't been fixed already. The latest version of `lcc` is always available for anonymous `ftp` in `pub/lcc` from `ftp.cs.princeton.edu`. A `LOG` file there reports what errors were fixed and when they were fixed. If you report an error that's been fixed, you might get a canned reply.

7. Send your program in an electronic mail message addressed to `lcc-bugs@cs.princeton.edu`. Please send only valid C programs; put all remarks in C comments so that we can process reports semiautomatically.

Further Reading

Most compiler texts survey the breadth of compiling algorithms and do not describe a production compiler, i.e., one that's used daily to compile production programs. This book makes the other trade-off, sacrificing the broad survey and showing a production compiler in-depth. These "breadth" and "depth" books complement one another. For example, when you read about `lcc`'s lexical analyzer, consider scanning the material in Aho, Sethi, and Ullman (1986); Fischer and LeBlanc (1991); or Waite and Goos (1984) to learn more about alternatives or the underlying theory. Other depth books include Holub (1990) and Waite and Carter (1993).

Fraser and Hanson (1991b) describe a previous version of `lcc`, and include measurements of its compilation speed and the speed of its generated code. This paper also describes some of `lcc`'s design alternatives and its tracing and profiling facilities.

This chapter tells you everything you need to know about `noweb` to use this book, but if you want to know more about the design rationale or implementation see Ramsey (1994). `noweb` is a descendant of WEB (Knuth 1984). Knuth (1992) collects several of his papers about literate programming.

The ANSI Standard (American National Standards Institute, Inc. 1990) is the definitive specification for the syntax and semantics of the C programming language. Unlike some other C compilers, `lcc` compiles only ANSI C; it does not support older features that were dropped by the ANSI committee. After the standard, Kernighan and Ritchie (1988) is the quintessential reference for C. It appeared just before the standard was finalized, and thus is slightly out of date. Harbison and Steele (1991) was published after the standard and gives the syntax for C exactly as it appears in the standard. Wirth (1977) describes EBNF.